Eradicating DNS Rebinding with the Extended Same-Origin Policy

Martin Johns
SAP Research
martin.johns@sap.com

Sebastian Lekies
SAP Research
sebastian.lekies@sap.com

Ben Stock
FAU-Erlangen-Nuremberg
ben.stock@cs.fau.de

Abstract

The Web's principal security policy is the Same-Origin Policy (SOP), which enforces origin-based isolation of mutually distrusting Web applications. Since the early days, the SOP was repeatedly undermined with variants of the DNS Rebinding attack, allowing untrusted script code to gain illegitimate access to protected network resources. To counter these attacks, the browser vendors introduced countermeasures, such as DNS Pinning, to mitigate the attack. In this paper, we present a novel DNS Rebinding attack method leveraging the HTML5 Application Cache. Our attack allows reliable DNS Rebinding attacks, circumventing all currently deployed browser-based defense measures. Furthermore, we analyze the fundamental problem which allows DNS Rebinding to work in the first place: The SOP's main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the corresponding security decision. Instead, the SOP relies on information obtained from the domain name system, which is not necessarily controlled by the Web server's owners. This mismatch is exploited by DNS Rebinding. Based on this insight, we propose a light-weight extension to the SOP which takes Web server provided information into account. We successfully implemented our extended SOP for the Chromium Web browser and report on our implementation's interoperability and security properties.

1 Introduction

The Web has won. No other platform for distributed applications can rival the Web's ubiquity and flexibility. The functionality demands of the ever-expanding Web application paradigm caused the browser to evolve from a simple program to display hypertext documents into a full-fledged runtime environment for sophisticated, networked applications. This evolution is still in full effect, with HTML5 and related JavaScript APIs being the latest addition to the browser model. In the context of Web applications, fundamental security properties are governed by the Same-Origin Policy (SOP): The SOP is the Web's principal security policy. It provides origin-based isolation of Web applications.

In the recent past, low-level vulnerabilities have become considerably harder to find and exploit. Hence, the ever growing capabilities of the Web browser make it an increasingly interesting offensive tool for attackers [8]: The Web browser runs behind the firewall within the boundaries of the internal network and executes code that was retrieved from the Internet. Thus, the SOP constitutes the only barrier between attacker provided code and the crown jewels in the internal network. Unfortunately, the SOP is far from bulletproof: Soon after the introduction of the policy in 1996, clever students at Princeton university found a way to utilize attacker controlled DNS settings to subvert the policy [24]. The underlying attack is today known as "DNS Rebinding" [14]. Since then, DNS Rebinding remained a constant problem of the SOP that was (re)discovered multiple times and, subsequently, attempted to be fixed.

In this paper, we demonstrate how the HTML5 Offline Application Cache can be misused to conduct reliable DNS Rebinding attacks. Our attack works with all major browsers, circumvents all current browser-based countermeasures, and affects most browser-based scripting runtime environments (JavaScript, Flash, Silverlight). Furthermore, we revisit the underlying problem of the SOP and propose a light-weight but powerful extension to the policy, which tackles the root cause of the problem.

Contribution and paper organization: After covering the required technical background (see Sec. 2) and the history of DNS Rebinding (see Sec. 3), we make the following contributions:

- *DNS Rebinding and the AppCache (Section 4):* We present a novel attack technique, capable of circumventing any existing browser-based countermeasure

against DNS Rebinding. In our attack, we utilize the HTML5 Offline AppCache to persist a malicious script until any domain-to-IP information is lost. In theory, caching-based attack scenarios are already known. However, the unpredictable and short-lived nature of the browser's caching behavior rendered them fragile to a level of unfeasibility. In this paper, we show how the unique characteristics of the AppCache can be leveraged by the attacker to create highly reliable DNS Rebinding attacks.

- *Vulnerability demonstration (Section 5):* To validate our attack method and to demonstrate its severity, we present two practical attacks on real-world applications utilizing Web interfaces. For our experiments, we chose the light-weight proxy server Polipo, and the Unix-based printing system CUPS. The effects of our demonstration exploits range from simple information leakage to remote code execution.
- *Extended Same-Origin Policy (Section 6):* We analyze the fundamental problem that causes DNS Rebinding to work. Thereby we identify a mismatch between the semantics and the implementation of the Same-Origin Policy: The SOP's main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the corresponding security decision. In order to overcome this mismatch, we propose a light-weight extension to the Same-Origin Policy that considers server-provided origin information. Our extended SOP reliably defeats DNS Rebinding attacks while increasing interoperability with mechanisms that rely on flexible DNS setups, such as DNS-based load-balancing or Content Distribution Networks.
- *Implementation for the Chromium browser (Section 7):* To demonstrate the practical applicability of our approach, we implemented it for the open-source browser Chromium. The implementation required in total 34 lines of code and does not cause a perceivable performance overhead.

We end the paper with a review of related work (Sec. 8) and a conclusion (Sec. 9).

2 Technical Background

In this section, we briefly cover selected topics that are necessary to discuss the paper's technical content.

2.1 The Same-Origin Policy

The Same-Origin Policy (SOP) was designed to enforce origin-based isolation of mutually distrusting Web applications. Several variants of the policy exist [37]. In this section, we focus the SOP for JavaScript [30].

In general, the Same-Origin Policy [14] 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 domain, port, and protocol, which were utilized to obtain the object. Hence, a JavaScript snippet is only allowed to access a resource if its own origin exactly matches the origin of the resource. The SOP for plug-in based script content, such as Flash or Silverlight, enforces similar rules.

Developers can adjust a JavaScript snippet's origin slightly by modifying the `document.domain` DOM property: The value of this property can be set to omit the values of subdomains up to the second level domain value (e.g., relaxing *www.example.org* to *example.org*). This process is known under the term "domain relaxation".

2.2 The HTML5 AppCache

Modern Web applications have one crucial disadvantage compared to desktop applications: Such applications can only be used when a network connection is available. In order to eradicate this disadvantage the HTML5 Offline Application Cache (*AppCache*) was introduced [10]. The AppCache is a mechanism that can be utilized to store resources (such as HTML documents, images, etc) within the browser for offline usage. In order to employ the Application Cache, a Web site may provide a manifest file containing a list of resources. The manifest file's location can be specified within the manifest attribute of a document's HTML tag as shown in Listing 1.

Listing 1: HTML5 Manifest attribute

```
1 <html manifest="manifest.mf">
2 [...]
```

When a browser discovers this attribute, it fetches the file and caches the listed resources within the AppCache. Listing 2 shows an exemplary manifest file that advises the browser to cache `index.php` as well as a flash applet named `flash.swf`. As soon as a cached resource is requested again, the Application Cache returns the cached HTTP response even if an Internet connection is available. After each access to the AppCache, the browser downloads the manifest file again to check whether it has changed. The resources within the AppCache are only updated if the manifest has changed - otherwise the resources reside within the cache even if their server-side counterparts have changed.

Listing 2: Exemplary manifest file (excerpt)

```
1 CACHE MANIFEST
2
3 http://example.org/index.php
4 http://example.org/flash.swf
```

3 DNS Rebinding

DNS Rebinding is a term introduced by [14], which describes a class of Web browser-based attacks that undermine the SOP through sophisticated mapping of DNS entries to restricted network resources. In Section 3.1 we give a full account on the historical development of these attack methods. In the remainder of this section, we briefly revisit the basic attack pattern.

The decision if a given JavaScript is granted access to a certain resource (e.g., browser window, or network location) is governed by the SOP. As explained earlier, the SOP relies on the *domain* property of the respective entity's origins. However, the HTTP protocol does not require any information about the requested *domain*. The actual HTTP connections are made using the server's IP.

An attacker can exploit this fact an attacker issues a very short-lived DNS entry for an attacker controlled web page. Whenever a victim visits this particular Web site, the victim's browser fetches the DNS entry, connects to the provided IP address and downloads attacker controlled JavaScript or plug-in code. This code is only capable of creating network connection to same-domain hosts due to the SOP. In the meantime, the DNS entry expired and therefore, as soon as another request is conducted towards the same domain, a new DNS entry has to be fetched. The attacker is able to exploit this behavior by altering the domain-to-IP-mapping. By providing an IP of the victim's intranet, the browser connects to the intranet IP as soon as the JavaScript conducts a same-domain request (see Fig. 1). As the IP is not a part of the Same-Origin check, the policy is still fulfilled and, therefore, the attacker controlled script is granted access to the response of the intranet host. Thereby, potential offensive scenarios are not limited to information leakage attacks on internal servers. DNS Rebinding can, for instance, also be used to conduct click fraud, defeating IP-based authentication, or hijacking of IP addresses (refer to [14] for a comprehensive overview). Nonetheless, for readability reasons, from now on we will use the information leakage attack as the motivational example.

3.1 The History of DNS Rebinding

As we will show in this section, the history of DNS Rebinding reaches back in time to the early days in which the SOP just started to emerge. Over the years, the attack was discussed under several different names, including "anti-DNS pinning" [7] and "Quick-swap DNS" [20]. In this time, several variants of the rebinding attack have been developed, either with focus on different browser-based technologies [17, 24, 28], with new techniques to circumvent the implemented mitigation measures [3, 15, 27, 33], or with focus on novel attack targets [9, 14]. Nonetheless, the general technique remained stable: Mapping an attacker-controlled DNS entry to a restricted network resource and subsequently using active browser content to access the resource.

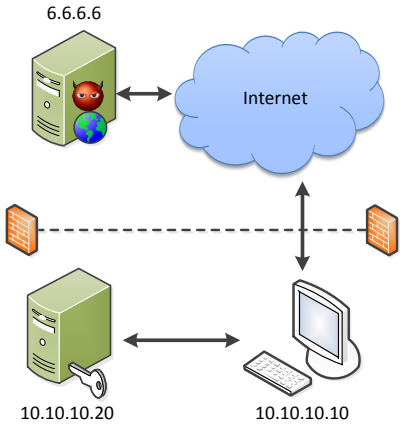

Figure 1: Intranet attack scenario

In this section, we give a brief overview on the developments of the past years. In general, the history of DNS Rebinding can be divided into three distinct time spans, each starting with the (re)discovery of the basic issue for a separate browser-based technology: 1996 (Java applets), 2002 (JavaScript), 2006 (Flash, JavaScript, Java).

3.1.1 1996 - Java Applets

Princeton's Secure Internet Programming group first mentioned the attack method in 1996 [24]. Back then, JavaScript networking capabilities were rather limited, while Java Applets already allowed comparatively sophisticated networking functionality [4, 24].

To be precise, the Princeton attack did not rely on DNS Rebinding per se. Instead, the attack utilized DNS records, which returned two IP addresses for the adversary's domain: The IP of the attacker's server, from which the applet was loaded and another IP pointing to the target of the attack. As the adversary controls the order of the values in the DNS response, the applet could be tricked to connect to the target system. To mitigate the issue, Java's vendor SUN introduced strict IP based access control [23]: After the initial loading of an applet, the only IP the applet is allowed to access is the IP address it was originally obtained from, regardless of information provided by DNS. This restriction is maintained for the entire lifespan of the applet.

3.1.2 2002 - JavaScript

The Princeton attack was extended by Adam Megacz to JavaScript in 2002 [20]. Megacz presented two variants

of the attack. For one, he utilized domain relaxation. In this case, the malicious JavaScript was hosted on a subdomain of the adversary's server, e.g., *sub.attacker.org*. The DNS entry for the father domain *attacker.org* pointed to the internal host. After being loaded in the victim's browser, the script relaxed its `document.domain` value to the father domain and, thus, was subsequently granted access to the internal server. The second attack variant, named "Quick-swap DNS" was roughly equivalent to the general attack scheme presented in Section 3.

In response to Megacz's security advisory, Netscape implemented explicit "pinning" of the domain-to-IP mapping for the lifetime of the Web page. In addition, to mitigate the domain relaxation based attack, a patch was created that required both parties in a domain relaxation scenario to assign the `document.domain` property to the same value. Versions of Internet Explorer that followed Megacz disclosure, exposed behavior similar to Netscape's browser. However, in 2007 Microsoft's Dave Ross gave to record that the observed DNS pinning was incidental and not introduced as a security measure [29].

3.1.3 2006 - The full browser experience

In 2006, Martin Johns discovered a technique to reliably cause Firefox and Internet Explorer to drop any domain-to-IP mapping, which in turn re-enabled the rebinding attack for JavaScript [15]. In the following months, several additional DNS Rebinding attack methods were disclosed: Kanatoko showed that Flash applets were also susceptible to the attack [17]. Also, Johns and Kanatoko documented a method to use the LiveConnect JavaScript-to-Java bridge to utilize Java methods in rebinding attacks [16]. Moreover, two further methods were discovered which allowed DNS Rebinding attacks on Java Applets: Rios and McFeters [27] tricked Java's applet cache by using multiple instances of the Java VM and David Byrne leveraged a mismatch in communication channels, in case the Java VM was configured to access the network with a Web proxy [3]. Finally, Dafydd Stuttard examined the effects of Web proxies on DNS pinning [33].

The susceptibility of the plug-in technologies Flash and Java enabled the usage of low-level socket communication in rebinding. This expanded the resulting attack surface towards non-HTTP network services. Furthermore, socket connections could be utilized to circumvent HTTP-based countermeasures, such as `host`-header checking [7].

Additionally, multiple public demonstrations on the capabilities of the attack vector have been given. Notable in this context are the experiments by Jackson et al. [14]: Using a specifically crafted Flash advertisement delivered by a major advertising network, the group was able to take over 27,480 unique IPs for a total amount of less then 30 US dollars. In response to the disclosed attacks, the vendors of Flash and Java introduced further restrictions on their socket-level network capabilities.

3.2 Capabilities and limitations of available countermeasures

Over the years, several practical and experimental countermeasures to protect against DNS Rebinding attacks have been introduced.

3.2.1 DNS Pinning

As previously discussed, most browser and plug-in vendors primarily reacted to DNS Rebinding by introducing DNS Pinning. When DNS Pinning is used, a Web resource's IP-to-DNS mapping is maintained for a prolonged timespan, ideally exceeding the lifetime of the resource.While being able to provide basic protection properties, DNS Pinning has security and functionality drawbacks: For one, DNS pinning is inherently incompatible with all technical measures that rely on dynamic and potentially changing DNS answers, such as load balancing, active failover, disaster recovery [1], or Content Distribution Networks. Also, DNS Pinning is unable to protect if Web proxies are part of the communication path to the server [20, 33] or in content caching scenarios (more on this in Section 4).

3.2.2 Limiting internal IP ranges

Due to the specific nature of DNS Rebinding, internal servers are the prime target of the attack. Hence, several techniques have been presented that protect internal network resources against external scripts. In general, these approaches primarily protect resources hosted on the "private" netblocks of the IPv4 space, as defined by RFC 1918 [25]. For one, such protection can be implemented on the DNS level: DNSWall [2] is a daemon that is designed to be used in conjunction with an existing recursive DNS resolver. It filters out RFC 1918 addresses in DNS responses. Also, the OpenDNS service offers a similar option [35]. Furthermore, similar protection can be achieved within the browser: Opera refuses script code which was obtained from an external source to access internal RFC 1918 IP ranges. The Firefox extensions NoScript [19] and LocalRodeo [**?**] can be configured to do the same.

The attempt to provide protection by restricting access to private IP ranges is necessarily incomplete. For one, network based access control is not limited to RFC 1918 ranges. In addition, bigger organizations, such as large companies or universities, do not always use

RFC 1918 addresses for their internal networks. Furthermore, with the growing support for IPv6 many use cases for RFC 1918 addresses cease to exist, as there is no shortage of IPv6 addresses. Finally, Craig Heffner has demonstrated [9], that even in cases where access to the private IP ranges is protected against DNS Rebinding attacks, under certain conditions the adversary can use rebinding to gain privileged access to local network resources, if these resource listens both on a private and a public IP address.

3.2.3 Application-layer protection of servers

Servers can implement active protection against the attack. A straight forward choice is requiring authentication before an internal server can be accessed. As the rebinding attack utilizes the adversary's domain, pre-existing authentication credentials, such as session cookies, cannot be abused by the attacker and, hence, the restricted data should be safe. Additionally, servers can implement `host`-header checking: The attacker's HTTP requests carry the domain name of the attacker's server in their `host`-header. Hence, the attack can be spotted and the access can be stopped, which usually is done by throwing a 400/500 server error or responding with a standard error message. However, this measure does not resolve the issue completely. The browser still allows the script to omit the request and receive the response. So even though, the server's data cannot be obtained, the attack vector may still leak valuable information to the attacker, such as validation that the server exists and material to do server-type and software fingerprinting. Also, while sounding straight-forward, `host`-header checking can be error-prone, as our experiments with CUPS has shown (see Sec. 5.2): Even though CUPS implements the check, the implementation is incomplete and grants an attacker access to a subset of the tool's data. Both techniques have in common, that they have to be introduced manually for each server on the application layer.

4 DNS rebinding using HTML5 AppCache

In the previous section we explained the basic mechanisms of DNS Rebinding. In order to counter these attacks browser vendors introduced a technique called DNS Pinning. In this section we show how this technique can be circumvented to reliably conduct DNS Rebinding attacks using the HTML5 Offline Application Cache.

4.1 Rebinding HTML/JavaScript content

pinning is to avoid the interaction of content that is served via the same origin, but received from different hosts. As soon as a DNS query is conducted, the browser pins the received domain-to-IP mapping. Subsequent requests conducted towards this origin are then exclusively sent to the host utilizing the "pinned" IP. Thus, while DNS pinning is active, content fetched from one origin always corresponds to the same host. Ideally, the pinning information should be stored as long as a resource resides within the browser. However, as mentioned already, DNS Pinning interferes with techniques such as load-balancing, active failover and disaster recovery [5]. The longer the pinning times, the bigger is the negative effect on these techniques. In the worst case, if the domain-to-IP mapping information are stored by the browser for an unlimited amount of time, these techniques would be more or less useless. Therefore, pinning durations differ substantially from browser to browser. However, all major browsers have one thing in common: As soon as the user closes the browser, the pinning information is automatically deleted. This also affects Web content which ended up in the browser's cache. Hence, a hunch about potential DNS Rebinding issues through cached content existed for some time [31].

The basic attack via cached content is similar to the general DNS Rebinding attack as described in Section 3. This time, however, we assume that DNS Pinning is in place and therefore the basic attack does not work as described. When caching comes into play, an attacker can re-enable the attack. This advanced attack, thereby, consists of two separate steps. In the first step the attacker lures the victim onto a prepared Web site and forces the browser to cache the attacker controlled contents. As DNS pinning is active this content is not yet able to launch a DNS Rebinding attack. However, browsers do not persist the domain-to-IP mapping and dispose it eventually. In the second step, at some later point in time, the attacker again lures the victim onto the Web page. This time the content is fetched from cache and therefore no DNS Queries or TCP connections are created. Only the origin information (protocol, domain, port) and the resources are retrieved from cache. When the cached resources attempt to create network connections to its own origin, no domain-to-IP mapping is available and therefore a fresh DNS Query is conducted opening up a vector for DNS rebinding.

Until today, it was difficult to launch such an attack as a browser's caching behavior is rather unpredictable and the adversary has only limited means to influence which content actually gets cached. The browser cache has a fixed size and in general handles cached content in a first-in-first-out fashion. Given the size of current Web sites, even a moderately used browser's cache fills up quickly and even recently cached content often gets discarded quickly [11]. Hence, depending on the given circumstances, the chances of keeping the attack script in the cache long enough for a successful attack tend

to be small. This changes with the introduction of the HTML5 Offline Application Cache. Compared to a traditional cache the AppCache provides an attacker with two novel capabilities that make attacks feasible:

- *Controllable caching behavior:* Using the AppCache manifest, the attacker can advise the browser to cache certain resources in a reliable way. As soon as the resources are stored within the AppCache, they reside in the browser for a potentially unlimited amount of time (until the attacker's application or the user decide to empty the cache manually).
- *JavaScript API:* The AppCache provides an API that allows JavaScript to identify whether it was loaded from cache or via the network.

Using these two ingredients, an attacker can conduct reliable DNS Rebinding attacks: In the first step the attacker lures the victim onto his Web site. The Web site uses a manifest file to cache an adversary controlled Web page within the Application Cache. After the browser deleted the DNS Pinning information, the adversary waits until the user visits the same site again. This time the Web page is loaded from the AppCache and no domain-to-IP mapping is available. Using the AppCache's JavaScript API, scripts contained in the page can verify that they indeed have been retrieved without network interaction. Hence, the cached script can now conduct same-origin requests towards the IP returned in the second DNS query (which the attacker controls completely). After the attacker's payload was loaded from cache, the AppCache revalidates the manifest file by downloading it from the attacker's domain. As this domain now points to the victim's IP address, the manifest will not be found and the cache will automatically be deleted (including the evidence for the attack). However, the attack has already taken place. In Section 4.2 we demonstrate, how an attacker is able to avoid the deletion of its content, in case he wants to conduct multiple attacks upon the same victim.

The attack demonstrated in this section only targets one specific victim. Nevertheless, the attack scheme can be extended to conduct large-scale attacks. Instead of conducting a rebinding attack directly on the main domain, the attacker could simply forward each user onto a distinct subdomain that can be rebound separately. As soon as one DNS query arrived at the attacker's DNS server for a specific subdomain, the DNS server could rebind the IP immediately. In the first step the user's browser pins the IP and therefore only sends one initial DNS request. Thus, if a second request arrives, the user's browser must have deleted the pinning information and is in need to refresh the information (opening the DNS Rebinding vector). The only challenge the attacker has to solve in step 2 is to forward the user to the same subdomain as utilized for this specific user in step 1. To identify whether the user has already conducted step 1 the attacker could simply utilize cookies that store the subdomain information on the victims computer until the next visit.

4.2 Utilizing multiple domains for reliable DNS Rebinding attacks

The previously described attack has one major weakness: As explained in Section 2.2, the AppCache revalidates the cache manifest after each access. If the manifest changed, files in the cache will be updated/deleted accordingly. Hence, in the last step of the attack, after the malicious script was fetched from the Application Cache, the browser revalidates the manifest file from the attacker's domain. Since the domain is, at this point in time, bound to the intranet host's IP, the browser requests the manifest file from the intranet host. As the file will typically not be available on the rebound server, the browser deletes the cached content. Nevertheless, the attacker is able to execute the malicious script at least once, as the cache validation takes place after the access to the cache. However, if the attack fails, e.g. because the user closed the browser before the script was executed completely, the attacker has to start the whole process of rebinding from scratch. For large-scale, automated attacks this is not a feasible solution. In order to overcome this issue, a more sophisticated attack scenario can be used. In this scenario, we are able to prevent the deletion of cached content after the rebinding step has taken place by utilizing two distinct domain names. Thereby, we are able to reliably repeat an attack multiple times without the need for rebinding a domain name over and over again. The attack thereby works as follows:

1. An attacker is in control of two domains (*attacker1.org* and *attacker2.org*) and the corresponding DNS server. In order to set up a DNS Rebinding attack, the attacker deploys an HTML document and an offline manifest to *attacker1.org*. The HTML document embeds (via frame, object or embed tags) active content (JavaScript, SVGs, Flash or Silverlight applets, etc) served by *attacker2.org*.
2. The attacker lures a user onto *attacker1.org*. Consequently, the user's browser renders the malicious HTML document and interprets the corresponding manifest file. Due to the instructions contained within the manifest, the browser caches the HTML document as well as the active elements.
3. By closing the browser, the user deletes the DNS pinning information. In the mean time, the attacker rebinds *attacker2.org* to the IP of an intranet host.

4. The attacker again lures the user onto *attacker1.org*. The Web page and the active elements are loaded directly from cache. As the page utilizes embed, frame or object tags for embedding the active elements, these elements are executed within the origin of *attacker2.org*. Due to the fact that *attacker2.org* is bound to the intranet IP, the active content is now able to communicate with intranet applications.

Analysis: In this scenario, as opposed to the first attack, the manifest file resides on a domain that is not subject to rebinding. Hence, when the cache validation takes place, the manifest is still available. Consequently, the browser does not delete the cached content. This is an important fact as it simplifies the attack a lot. If we take, e.g., a corporate wiki containing a multitude of information, the extraction and transfer of the data to the attacker would consume a large amount of time. However, the attacker can only extract the data while the user still visits the malicious Web site. If the user leaves the Web site before all parts of the data were extracted, the attacker is able to again lure the user onto the vulnerable page to continue the extraction process instead of needing to re-iterate the first rebinding step.

4.3 Caching of plug-in content

As mentioned before, the AppCache can be used to store cross-domain resources for offline usage, which is a key enabler for the attack described in the previous section. However, the browser implementations differ in the way they utilize the cache when it comes to cross-domain caching and in the way they defend against rebinding attacks. In this section we shed light on these differences and explain how an attacker can make use of them.

HTML/SVG documents Caching of HTML and SVG documents works across all browsers in the same-domain scenario. However, when it comes to cross-domain caching the behaviors of browsers differ substantially. For the second attack, a distinct document embeds an HTML or SVG file from a second domain via `frame` or `object` tags. The manifest file resides on the first domain, hence referencing the HTML/SVG file across domain boundaries. While WebKit-based browsers (e.g Safari, Chrome) and Internet Explorer do not fetch such embedded cross-domain resources from cache, Firefox and Opera expose a different behavior: Opera fetches both, content embedded via `frame` and via `object` tags, from the AppCache. Firefox, however, only fetches HTML/SVG documents from cache when they are embedded via `object` tags. Therefore, the advanced attack does not work within Safari or Chrome when utilized in combination with JavaScript. To overcome this issue an attacker can utilize plug-ins such as Flash or Silverlight.

Silverlight All popular desktop browsers except Internet Explorer 10 support the cross-domain caching of Silverlight applets within the offline application cache. This behavior can be abused to conduct DNS Rebinding attacks within these browsers. A Silverlight applet is, similar to JavaScript, able to conduct requests and read the corresponding responses. Hence, the abilities are similar to the HTML/SVG case, but the desktop browser support for the complex attack is better. Mobile browsers, however, are not able to execute Silverlight applets.

In earlier versions of Silverlight, it was possible to also create arbitrary socket connections to same-domain hosts. Fortunately, those capabilities are nowadays severely limited by the underlying security model which only allows opening of a socket connection when the receiving host explicitly grants this connection by setting up a whitelisting policy on port 943. If port 943 is closed, the Silverlight plug-in attempts to download the policy file from the Web server's root directory. Using the HTML5 Offline Cache, an attacker is able to cache such a cross-domain policy at the Web server level. This allows an attacker to open arbitrary socket connections to the rebound IP. As this behavior was already misused in earlier rebinding attacks, Microsoft limited the connection capabilities of Silverlight to a very restricted port range (4502-4534), effectively reducing the impact of such attacks.

Flash Similar to Silverlight, browsers also cache Flash applets within the AppCache. Hence, Flash can be used as an alternative to Silverlight when conducting a DNS Rebinding attack with multiple domains. Thereby, Flash also has the ability to create HTTP requests towards same-origin resources without restrictions. However, Flash has two major advantages over Silverlight:

1. *Widespread adoption:* Although its market share decreases, Flash is still present in about 95% of all browsers [26] (including some mobile browsers).
2. *Less restrictive SOP for HTTP requests:* Flash only includes the protocol and the domain into its cross-domain decision making process [34]. Hence, a Flash applet is able to send requests to any same-domain port and receive the corresponding responses. This behavior can be used to conduct DNS Rebinding attacks on non-HTTP-based intranet services.

Java Java applets do not utilize the browser's AppCache. Instead, Java uses its own caching mechanism that defends against DNS Rebinding by storing the IP address of the host that served the applet. When conducting a HTTP or Socket connection the applet is only allowed to connect back to the same IP adress.

Browser	SD	TD SVG	TD F	TD SL
IE 10	√	-	-	-
Firefox 14.0.1	√	√	√	√
Chrome 21	√	-	√	√
Safari 5.1	√	-	√	√
Opera 12	√*	√*	√*	√*

*: Opera prevents access to RFC 1918 addresses.

Table 1: Desktop browser & Attack Overview

Browser	SD	TD SVG	TD F
Mobile Safari	√	-	*n.a.*
Android Browser	√	*n.a.*	Crash
Mobile Chrome	√	Error	*n.a.*
Mobile Firefox	√	-	√

Table 2: Mobile browser & Attack Overview

Other plug-ins Beside Flash, Silverlight and Java there is a multitude of other plug-ins which can potentially be abused to conduct the presented attacks. If a plug-in applet can be cached within the browser's Offline Application Cache, it is very likely that it can also be used for the outlined rebinding attacks.

4.4 Summary

As seen in this section, there are a lot of technologies that can be abused by an attacker to gain novel capabilities in the context of a rebinding attack. In order to summarize our findings, Tables 1 and 2 outline which desktop and mobile browsers are vulnerable to the presented attacks.

As seen within the tables, the attack including a single domain (denoted as SD) works within every browser. The attacks comprising two distinct domain names (denoted as TD) affect mainly desktop browsers. The reason for this is the missing plug-in and SVG support within mobile browsers. Furthermore, the mobile versions seem to be more error-prone: The mobile version of Chrome was not able to render our SVG test case (it showed a 404 page, although the server logs indicated that the resource was properly requested), Android's standard browser even crashed every time it loaded a Flash file from cache.

5 Practical Attacks

To demonstrate the impact of the outlined vulnerabilities, we deployed a real-world setup including three distinct hosts (depicted in Figure 1). In this setup we investigated the susceptibility of two applications (Polipo and CUPS) by conducting the attack described in Sec. 4.2

5.1 Polipo

Our first attack targets a light-weight proxy server called Polipo, which can be used to connect to the TOR anonymizing network. To simplify the handling, Polipo offers a Web interface for configuration purposes. By default, this interface listens to port 8123 and does not defend against DNS Rebinding attacks. Via the Web interface, a user is able to configure the proxy settings, which are, obviously, security critical.

To evaluate Polipo's resilience against our DNS Rebinding attack we successfully conducted an attack as described in Section 4.2. Due to the fact that Polipo does not implement any countermeasures against DNS Rebinding, our malicious requests were processed as if the Web application itself created it. Via this attack, we were able to remotely change the settings of the proxy server. Beside the standard proxy functionality, Polipo also offers Web server functionality that can be abused by an attacker to download arbitrary files from the attacked host. The Web server is by default only serving the configuration interface. However, the Web server's configuration can also be changed via the configuration interface. In order to steal arbitrary files, an adversary could simply set the Web server's root directory to the server's root directory ("/" on Unix-based systems), effectively exposing all the files on the host to the outside world. For example, by requesting *http://attacker2.org:8123/etc/passwd* (were attacker2.org is already bound to the internal host) our malicious script was able to extract the information on all the registered user accounts.

5.2 CUPS

CUPS is a printing system for Unix-based operating systems. It offers a web-based administration interface running on port 631 (accessible via localhost only). Via this interface a user can administer the installed printer, monitor print jobs and configure the print server. Interestingly, the main administration panel of CUPS protects against DNS Rebinding attacks by checking the HTTP host header. Some features also require proper authorization, consequently, mitigating the risk of unauthorized access via DNS Rebinding. Nevertheless, it is still possible to extract valuable information out of the administration interface via a DNS Rebinding attack. The reason for this is an insufficient protection of log files that are accessible via the Web interface. While the main administrative functions are protected, the page and error log files can be accessed with arbitrary host headers. This allows an attacker to extract the log files containing sensitive information via DNS Rebinding attacks:

Error log: The error log contains information on failed print jobs, which can be used for reconnaissance of a corporate intranet. When a print job fails, technical

details are written into the logs, including the username of the creator, exact information on the printer addresses and the administrator of the printer. Furthermore, it contains information on the root directory of CUPS as well as the value of the current `PATH` variable of the machine CUPS is running on.

Page log: The page log gives an overview over the past print jobs sent to a printer. By extracting the page log, the adversary receives the names and dates of the documents that were printed via CUPS. On our test system, running Mac OS, we were able to extract the complete printing history of over one year. Thereby, the name of a document reveals a lot of information such as absence dates of the employee, data on intellectual property, etc.

6 Extending the Same-Origin Policy

As shown in Section 3.1, DNS Rebinding is a constant problem of the Web application paradigm (as witnessed in 1996, 2002, and 2006). Taking the attack method presented in this paper into account, this is the fourth time that wide-scale DNS Rebinding issues are discovered, even though the basic problem is known since 1996 and has received considerable attention. Hence, it is safe to conclude that DNS rebinding is a fundamental, protocol-layer flaw of the Same-Origin Policy, which is not solvable with the existing means. As discussed in Section 3.2, all currently available remedies are either incomplete (e.g., protecting specific IP ranges) and/or have to be implemented explicitly on the server-side's application layer (e.g., host header checking).

In this section, we show how the Web interaction paradigm can be extended in a non-disruptive manner to enable a robust protection. For this purpose, we first state our design goals (Sec. 6.1) and conduct a root-cause analysis of DNS rebinding (Sec. 6.2). Then, we introduce the "Extended Same-Origin Policy (eSOP)", starting with simple scenarios (Sec. 6.3) and then iteratively explaining how the policy handles non-trivial cases (Sec. 6.3.1 and Sec. 6.3.2). Finally, after stating the eSOP's decision logic (Sec. 6.3.3), we show how the policy protects against DNS Rebinding attacks (Sec. 6.3.5).

6.1 Design goals

Before going into detail concerning our solution, we briefly discuss the goals which steered its design process. As stated above, we are not aiming to create band-aid solutions or incomplete protection measures. Instead, the goal is to introduce a fundamental solution that is capable of completely solving DNS Rebinding. In this context, our design goals were as follows:

(DG1) Client-side enforcement: The Same-Origin Policy is a client-side security policy. Hence, all aspects of the policy decision and enforcement process should be conducted in the Web browser.

(DG2) Protocol layer: It should be avoided that Web applications have to explicitly implement protection or decision logic on the server-side's application layer. Instead, the designed solution should be capable of providing transparent protection by default purely on the protocol layer.

(DG3) Dedicated security functionality: The history and present of the Web is full of cases in which non-security features were (mis)used to realize security functionality. In many cases, the resulting security properties were fragile, often incomplete and not necessarily future proof. Therefore, we do not want to rely on non-security features (i.e., the host header). Instead, dedicated functionality shall be introduced where necessary.

(DG4) Non-disruptive: The solution should be backwards compatible. This means, if a given application scenario involves an entity (i.e., Web server or browser) that does not yet implement the solution, the Web application should not break and the security properties should transparently revert to the currently established state.

6.2 The three principals of Web interaction

As explained in Section 2.1, the Same-Origin Policy's duty is to isolate unrelated Web servers. To do so, the SOP enforces access control in the browser, based on the "origins" of the corresponding resources. In this context, such origins are derived from the URLs that are associated with the interacting resources - usually the URLs of the enclosing document objects. Hence, the semantics of the SOP are built around two principals: The *browser* for enforcing the policy and the *server(s)* for providing the resources which are the subjects of the policy decision.

However, the entities involved in the implementation of the SOP differ: While the *browser* remains in charge of enforcement, the underlying informations are not provided by the involved Web *server(s)*. Instead, the *network* in the form of Domain Name System and IP addresses is utilized to associate the URL-values to the server resources. Hence, the principal that is central to the SOP's purpose, the *server*, is not even involved in the actual policy decision. Even worse, security characteristics associated with the *server* are governed by *network* resources that are not necessarily controlled by the server's owner. As a consequence, a crucial mismatch exists between the semantics and the implementation of the SOP. As seen above, DNS Rebinding takes advantage of this mismatch. In a rebinding scenario, the attacker utilizes *network* resources under his control to undermine the security characteristics of the *server*.

In summary, the Web application model actually spans three principals in total: The *browser*, the *server*, and the *network*. Hence, to address the currently existing mismatch between policy semantics and implementation, it is necessary to investigate approaches that involve the *server* in the policy decision process.

6.3 eSOP: Extending the SOP with explicit server-origin

When considering the SOP from an abstract point of view, a Web "origin" defines the trust boundaries of a Web application. Everything within the application's origin is fully trusted, everything outside is completely distrusted. Additional browser capabilities, such as domain relaxation (see Sec. 2.1) and CORS [36], provide methods to selectively widen the application's trust boundaries. In the last section, we observed that the Web server itself is left out of the equation in the SOP's current implementation. This is counterintuitive, as among the involved parties, it is the Web server that should be able to set its own trust boundaries. However, the Web server can only indirectly influence the browser's enforcement decisions. Hence, to resolve this shortcoming, we propose to extend the SOP to include Web server-provided input. For this purpose, our approach expands the current, triple-based SOP with a fourth component that is provided by the server. Simplified, our proposed extended Same-Origin Policy (eSOP) works as follows: All HTTP responses of a given server carry explicit, server-provided information of the server's trust boundaries. From now on, we refer to this information as the *server-origin*. Thus, in the extended model, a Web origin consists of the quadruple {*protocol, domain, port, server-origin*}. In consequence, whenever the browser conducts an eSOP check, not only the classic protocol/domain/-port triple has to match, but also the *server-origin* values.

Example 1 (standard behavior): For simple cases, a Web origin's *domain* and *server-origin* values should not differ. Take for instance a script running under the origin {*http, example.org, 80, example.org*}. This script attempts to access a document in an iframe which also has the origin {*http, example.org, 80, example.org*}. All four elements of the respective Web origins match, thus, the eSOP is satisfied and the access is granted.

6.3.1 Multiple domains as server-origin

However, last section's simplified policy decision logic is not sufficient to cover all application scenarios, that are allowed with the current SOP. This primarily concerns Web applications which can be accessed via multiple domain names. For instance, many Web applications do not distinguish between the main domain name (e.g., example.org) and its "www" counterpart (i.e., www.example.org). Similar scenarios exist for applications accepting requests for multiple top-level domains (e.g., example.com and example.net). Hence, for resources served by such applications, it is not straight forward to decide what their corresponding *server-origin* is. As stated in design goal 6.1, our solution shall not require the implementation of application-layer decision logic on the server-side. In consequence, a solution is needed which allows server-side configuration on the protocol-layer. For this reason, the eSOP permits that the server specifies more than one domain value as its *server-origin*. This way, the *server-origin* precisely specifies a server's trust boundaries, i.e, the set of domains which it grants access in a same-origin context. Furthermore, we adjust the criteria under which two Web origin quadruples comply to the eSOP: The eSOP is satisfied if and only if the classic *protocol/domain/port* values of both quadruples match and the domain value of the acting origin (i.e., the origin of the script) is included in the *server-origin* of the resource which the script tries to access.

Example 2 (multiple server-origins): A Web application available via *example.org* and *www.example.org* specifies its *server-origin* as a tuple of both domains: ⟨*example.org, www.example.org*⟩. A script running in a document under the origin {*http, example.org, 80,* ⟨*example.org, www.example.org*⟩} tries to access a document in a iframe which also has the origin {*http, example.org, 80,* ⟨*example.org, www.example.org*⟩}. As the script's domain value (*example.org*) is included in the target document's *server-origin* list ⟨*example.org, www.example.org*⟩, the eSOP is satisfied and, thus, the access is granted.

6.3.2 Handling domain relaxation

The specific matching criterion for *server-origin* also allows simple and robust handling of domain relaxation via setting the `document.domain` property during client-side execution: As long as the newly set origin is still in the target resource's list of domains, the eSOP allows access under the relaxed domain values. This even works in situations in which the individual subdomains are handled by separate Web servers with potentially different *server-origin* configurations.

Example 3 (domain relaxation): Take a Web application on *example.org*, which has multiple subdomains, including *sub.example.org*. The application's subdomains are handled by dedicated Web servers. Furthermore, the *example.org* server hosts all resources that are shared among the subdomains. A script is executed under the extended Web origin {*http, sub.example.org, 80,* ⟨*sub.example.org*⟩}. Furthermore, the browser provides a reference to a resource from

the main application with the origin {*http, example.org, 80,* ⟨*example.org*⟩}. The script assigns the value *example.org* to the `document.domain` property, thus, effectively relaxes its domain value to the fathering domain. As a result, the script's effective origin is now {*http, example.org, 80,* ⟨*sub.example.org*⟩}. Consequently, the eSOP is now satisfied in respect to the referenced resource, as the script's domain value is included in the domain set of the resource's *server-origin*, and the access is granted.

6.3.3 The eSOP decision logic

To sum up, we now give a precise definition of the eSOP.

> The eSOP is satisfied iff:
>
> $$\{prot1, domain1, port1\} == \{prot2, domain2, port2\}$$
>
> and
>
> $$domain1 \in server\text{-}origin2$$
>
> If the *server-origin2* property is empty, the second criterion always evaluates as "true".

The last condition of the eSOP provides robustness and backwards compatibility with the old behavior. In addition, to facilitate flexible and easy configuration, we follow the example of the Content-Security Policy format [32], and allow the usage of wildcards for subdomain values within the set of domains in the *server-origin*, e.g., ⟨*.domain.com⟩.

6.3.4 Communicating the server-origin

The final missing puzzle piece is the exact method, how the server communicates the *server-origin* property of his resources to the browser. We propose to introduce a dedicated HTTP response header, `X-Server-Origin`, that carries the *server-origin* property in the form of a comma-separated list.

Choosing this approach has several advantages: Foremost, it is compatible with the caching behavior of Web browsers. Web browsers are already required to cache HTTP response headers along with the actual resources, as they otherwise would not be able to properly interpret the cached content after retrieving it from storage. Also, unlike DNS or IP-based protection schemes, properties communicated via HTTP response headers are preserved when the browser accesses the network via a Web proxy. Finally, adding features using new response headers is non-disruptive, as older browsers simply ignore unknown response headers. Furthermore, implementing server-driven security functionality via HTTP response headers is a proven technique. In the recent past, several security measures have successfully been introduced, that leverage response headers, such as Clickjacking protection via the `X-Frame-Options` header [22], protection against SSL-stripping attacks via the `Strict-Transport-Security` header [12], Content Security Policies, that are set using the `X-Content-Security-Policy` or `X-WebKit-CSP` headers [32], and cross-origin resource sharing which utilizes the `Allow-From-` header [36].

6.3.5 The eSOP and DNS Rebinding

In the previous sections, we discussed the semantics of the eSOP and the reasoning behind the corresponding design process. Now finally, we show that the eSOP is indeed capable of prevention DNS Rebinding attacks. To conduct a DNS Rebinding attack, the adversary maps the DNS setting of a *domain* to the IP address of the targeted Web server. However, the attacker controlled *domain* value is not in the Web server's trust boundary. In consequence, the value will not be included in the list of domain values in the server's *server origin* property. Therefore, the eSOP check will necessarily fail.

Example 4 (DNS Rebinding): The attacker controls the domain *attacker.org*. His goal is to access an internal wiki server under the domain *wiki.corp*, which sets a corresponding *server-origin*. In the first step of his attack, the adversary tricks the victim to access the *attacker.org*, which still is mapped to a Web server IP under his control. Hence, the script is handled by the browser under a Web origin of the form {*http, attacker.org, 80,* ⟨. . . ⟩}. Please note, that this Web origin's *server-origin* property is fully controlled by the attackers, as he creates the corresponding HTTP response. However, this does not cause any issues, as the *server-origin* of the acting script is irrelevant for the eSOP decision process. Then, the attacker conducts the DNS Rebinding step. Now, the DNS entry of *attacker.org* points to the IP address of the internal server. From this point on, the browser will interpret all resources from the server under the Web origin {*http, attacker.org, 80,* ⟨*wiki.corp*⟩}. Following the rebinding step, the attacker's script attempts to access Web resources that are provided by the internal server. However, as the attacker's script carries the *domain* property *attacker.org*, which is not included in the list of domains in the server's *server-origin*, the attack fails, even though the classic *protocol/domain/port* SOP is satisfied.

6.3.6 Invalid eSOP origins

In [13], Jackson and Barth examine a set of proposed SOP variants with finer-grained origins. Among other

techniques they discuss two approaches closely related to the eSOP: The Locked SOP and IP-based origins (for details on these techniques please refer to Sec. 8), which provide basic protection against DNS Rebinding attacks. For both techniques they uncover a loophole which re-enables DNS Rebinding attacks, even if the refined SOP variant is in place: Take a Web page on an internal host which intends to import a JavaScript file from the same host using a relative URL (see Lst. 3).

Listing 3: Direct script include using a relative URL

```
<script src="jquery.js"></script>
```

This Web page is retrieved by the browser using the adversary controlled hostname *attacker.org*, which resolves to the intranet IP *10.10.10.10*. Then, before the `script` tag is interpreted the rebinding step takes place. *Attacker.org* now points to *6.6.6.6* which is owned by the adversary. Unlike JavaScript execution, HTML-based script includes are not subject to origin restrictions. Hence, a refined SOP has no direct effect here and the script code is retrieved from the adversary's host, circumventing the protection of the refined policy. Fortunately, in the case of the eSOP such situations are reliably detectable. The following condition holds for all HTML documents with origin {*prot, domain, port, server-origin*} that were retrieved from an attacked host:

$$domain \notin server\text{-}origin$$

This necessarily results from the fact that the adversary cannot control the *server-origin* of the internal host, which only contains domain values within the server's trust boundaries (which obviously excludes the adversary's sites). In such cases, we label the page's Web origin as *invalid*. For Web documents with an *invalid origin* caching is disabled and strict DNS pinning is enforced for the whole browser session, effectively closing the loophole.

6.4 Security evaluation

As shown above the eSOP protects against DNS Rebinding attacks, without requiring additional server-side logic or specific actions on the client-side. As soon as the `X-Server-Origin` header is present, the browser is capable of transparently enforcing the policy, fulfilling design goals (DG1) and (DG2). Furthermore, due to communicating the *server-origin* in the form of an HTTP response header, the protection is robust in scenarios which caused other countermeasures to fail: HTTP response headers are cached alongside with the actual cached resources. Hence, the *server-origin* is maintained even in long-term caching scenarios, effectively closing the attack vector which is the subject of Section 4. In addition, currently problematic scenarios, in which the browser has no control over the domain-to-IP mapping, e.g., through a Web proxy, can be handled conveniently. The `X-Server-Origin` header is preserved, even if Web proxies obstruct the link between domain name and server address. Hence, the attack scenario described in [33] (see also Sec. 3.1) is not feasible anymore. Finally, the eSOP is at least as strong as the currently implemented SOP: The *protocol/domain/port*-triple is still required to match, as it is by the classic SOP. Thus, it is a necessary condition that the access to a resource is granted under the SOP for the eSOP to be satisfied. Therefore, implementing the eSOP will never lead to security degradation.

6.5 Functional evaluation

The eSOP is fully backwards compatible to the classic SOP. In cases that either the browser does not implement the extended policy or the Web server does not provide a `X-Server-Origin` header, the enforced policy transparently reverts back to the standard behavior of matching *protocol/domain/port*, fulfilling design goal (DG4).

A major concern during designing the extended policy was the aspect of maintainability: Especially in large set-ups that span multiple Web servers, ensuring that all server installations provide the exact same values for the *server-origin* property, is an unrealistic hard requirement. Fortunately, the eSOP's specific *server-origin* matching criterion (see Sec. 6.3.3) allows a robust and flexible handling of such situations. The eSOP does not require the *server-origin* values to match exactly. The only requirement is, that the acting domain is whitelisted in the receiving *server-origin*. Hence, even in situations of slightly different server configurations (much like in Example 3, Sec. 6.3.2), the functionality of the Web application remains undisturbed. Additionally, this robustness property also allows *server-origin* settings to change in long term caching scenarios. As long as the initial domain requirements of the cached resource remain fulfilled, the server's *server-origin* setting can be extended or modified without causing interoperability problems.

Last but not least, an adaption of the eSOP would obliterate the requirement of DNS Pinning for security reasons completely. Hence, for servers that provide the `X-Server-Origin` header, the DNS TTL value can be as small as desired. No security degradation will occur, when browsers respect such small TTL values. This in turn allows easy setup of highly flexible load-balancing and error-correcting network setups with multiple, redundant servers.

7 Practical Implementation

In order to validate the feasibility, security and functionality properties of the eSOP, we implemented it for the Chromium Web browser [6]. Thereby, we enhanced the so-called Security-Origin which stores the "protocol, domain and port"-triple of a Web site by adding the proposed *Server-Origin*. Data stored within this data structure is provided by the `X-Server-Origin` response header. Our implementation allows the header to have two types of values. If the server does not send the header or sends an empty header, we assume that it does not implement our approach or wants to opt-out of the protection mechanism. In these cases, we allow access regardless of the acting *domain* value for backwards compatability. Additionally, the header can be set to a list of comma-separated domains. Using the stored information we are able to successfully prevent rebinding scenarios. At this point, we need to distinguish between XmlHttpRequests (XHRs) and script access to a viewport, such as frames or popup windows.

Script access to a viewport For a viewport, we want to align our implementation to how browsers should handle cross-origin requests, thus allowing a popup or frame from any resource to be rendered but to deny script access if the origins do not match. This is also important towards keeping design goal (DG4), i.e., being downwards compatible. In the current implementation of Chromium we extended the origin check to verify the server-origin as well as the protocol, domain and port. If a Web application does not implement our suggested extended same-origin policy, the browser falls back to the normal SOP validation and renders the page properly.

XmlHttpRequests For XmlHttpRequests, we patched the functionality for same-origin requests to parse our response header field and to grant or revoke scripting access depending on the received value.

To be fully interoperable with the browser's XHR object, we had to ensure compatibility with its recently introduced cross-origin capabilities:

To allow XHRs to access cross-origin resources, the W3C specified cross-origin resource sharing (CORS) [36]. CORS allows the initiation of *simple* requests to a cross-origin resource and only checks the right to access the response after the request has been completed. In the context of CORS a requests is considered to be *simple* if it also would be possible to create an equivalent request with other means, such as *IMG*-tags or HTML forms. because simple requests cannot change the state of a web application.

For *complex* requests, CORS requires that the browser sends a preflight request to the server to retrieve the CORS-relevant headers. Only if the retrieved headers allow access to the resource, the complex request is sent to the server to ensure that state-changing operations are only performed if explicitly allowed by the application.

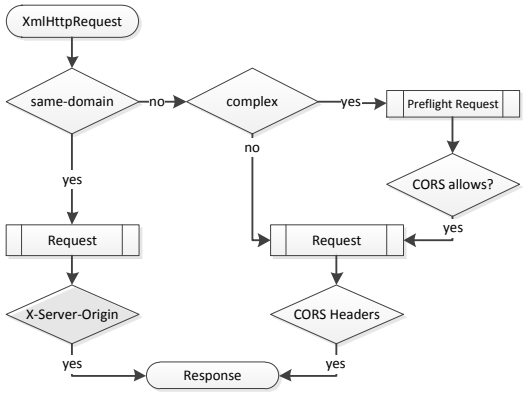

Figure 2: Implementation logic for XHR

In a sense, requests to rebound domain should also be treated as cross-origin requests. Thus, we can allow simple requests to be sent but need to verify the server-origin before allowing access to the response. For a complex request, we need to check the preflight response and only allow the actual request to be sent if the server-origin matches. To distinguish between simple and complex, we used the already existing check from the CORS implementation in Chromium. However, using the preflight functionality from CORS would break constraint DG3. If - for example - we request a same-domain resource on a server that does not implement CORS, the CORS headers would not be set and the check would fail. Therefore, we implemented a function that only check the X-Server-Origin header.

The flow chart in Fig. 2 shows the resulting implementation logic of the XHR object. Our addition to the implementation is positioned on the lower left of the chart, whereas the right part of the figure depicts the original logic as implemented by Chromium. Note that as an XHR is not rendered by the browser, we can directly block access upon receiving the response from the server.

7.1 Implementation and performance

In total, we modified 34 lines of code in Chromium. As discussed earlier, the implementation manifests itself only as parsing and extraction of the HTTP headers, the allocation of a little amount of memory to store the server-origin and a string comparison of the domain and the stored value. The parsing of HTTP headers is executed for any request, thus the performance impact is

reduced to just one more array access. Thus, in our tests we had no noticeable overhead when accessing a Web application.

8 Related Work

Related offensive and protective techniques have already been the subject of Sections 3.1 and 3.2. Hence, in this section we focus on approaches that directly relate to the eSOP, as they propose modifications to the browser-server interaction to combat DNS Rebinding:

Conceptually closest to our protection approach is the "Strong Locked SOP" by Karlof et al. [18], which also proposes to include server-provided information into the SOP decision. In the case of the "Strong Locked SOP", this information is derived from the TLS/SSL certificates of the involved Web servers in the form of the certificates' public keys. Consequently, JavaScript is only granted access to resources that share the same public key. In the special case of "pharming" attacks (which is the approach's main concern), where the attacker controls the DNS resolving process of the victim, Karlof's approach is conceptually stronger than the eSOP. Furthermore, in a scenario in which all communication is done via HTTPS and all servers are outfitted with valid SSL certificates, the Strong Locked SOP would provide reliable protection against rebinding attacks. However, expecting the Web to go completely HTTPS appears unrealistic, especially regarding intranet Web resources which only in very rare cases have valid SSL certificates. In contrast, the eSOP only requires to configure a single response header and works well in plain HTTP scenarios.

In [13] it is mentioned that early versions of the HTML5 specification included "IP-based Origins", which utilize the server's IP as a fourth factor in the origin check. Compared to the eSOP, IP-based Origins are neither able to securely handle domain relaxation nor do they provide evidence of invalid origins (see Sec. 6.3.6), thus, making them susceptible to library include attacks.

Furthermore, Jackson et al. propose "Host Name Authorization", a network based service [14], which announces the host names that are associated with a given IP address. Host Name Authorization relies on reverse DNS: Whenever the browser executes a DNS lookup, it also verifies that the requested domain is actually in the set of valid domains of the received IP address. This is done via querying the service under *auth.*`ip`*.in-addr.arpa*, with `ip` being the IP address which has been returned by the DNS server. Compared to our approach, Host Name Authorization has several drawbacks. For one, it requires considerable setup effort, as both reverse DNS as well as the actual service have to be enabled. Also, Host Name Authorization is realized within the DNS system, hence, the maintainer of the Web server also needs administrative access to the corresponding DNS server. This requirement cannot always be satisfied, e.g., in shared hosting scenarios, for local machines, or for internal services in cooperate networks. In addition, the approach requires two additional DNS round trips for each DNS resolving process, which could lead to noticeable latency under certain circumstance, e.g., cellular networks. In comparison, our approach only requires Web server-provided functionality and does not add any network overhead.

Finally, for completeness sake, the Internet draft [21] proposes the HTTP request header `X-Request Origin`. The purpose of the header is to transport the domain value or IP address of the browser-based component which was responsible for initiating the HTTP request within the browser. The draft lists DNS Rebinding attacks (in the form of "Quick-swap DNS") as one of its motivational examples. However, in the context of DNS Rebinding situations, the header's value will necessarily always equal the value of the HTTP `host` header, and hence, shares its protection properties and drawbacks.

9 Conclusion

For more than one and a half decades, DNS Rebinding continued to be a constant problem of the Web. Several attempts to mitigate the issue have been undertaken, but up to now no fundamental solution for the problem was introduced successfully. In this paper, we presented a novel attack variant, utilizing the HTML5 AppCache. We practically validated our attack and demonstrated that it affects all popular browsers and most plug-in technologies, while reliably circumventing currently existing browser-based countermeasures. Using our attack as motivation, we revisited the attack's underlying problem and identified a mismatch between the SOP's semantics and its implementation: The SOP's main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the security decision. Instead, the SOP relies on information obtained from the domain name system, which is not necessarily controlled by the Web server's owners. This mismatch is exploited by DNS Rebinding.

To overcome this problematic inconsistency, we proposed a light-weight extension to the SOP (eSOP), which takes input from the Web server into account. The eSOP robustly defeats DNS Rebinding attacks while being backward compatible with user-agents that do not yet implement the extended policy. Our solution does not require additional network traffic and fully supports previously problematic scenarios, including domain relaxation, content caching, and communication over Web proxies. Additionally, the eSOP eradicates the need for DNS Pinning. Thus, browsers implementing the pol-

icy can better inter-operate with dynamic DNS settings, such as DNS based load-balancing or Content Distribution Networks (CDNs). In summary, adopting the eSOP comes with very little costs but leads to a significant security increase and additional benefits in functionality.

Acknowledgments

This work was in parts supported by the EU Project WebSand (FP7-256964), http://www.websand.eu. The support is gratefully acknowledged.

References

[1] B. Anderson. Why Web Browser DNS Caching Can Be A Bad Thing. [online], http://dyn.com/web-browser-dns-caching-bad-thing/, last accessed 08/06/2012, 2011.

[2] A. Bortz, A. Barth, and C. Jackson. Dnswall. [software], http://code.google.com/p/google-dnswall/.

[3] D. Byrne. Anti-DNS Pinning and Java Applets. Posting to the Bugtraq mailing list, http://seclists.org/fulldisclosure/2007/Jul/0159.html, July 2007.

[4] D. Dean, E. Felten, and D. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 190–, Washington, DC, USA, 1996. IEEE Computer Society.

[5] S. Dutta. Client-side cross-domain security. Technical report, Microsoft, Dec. 2011. http://msdn.microsoft.com/en-us/library/cc709423\%28v=vs.85\%29.aspx.

[6] Google Chromium Developers. The Chromium projects. [online] http://www.chromium.org.

[7] J. Grossman, R. Hansen, P. Petkov, and A. Rager. *Cross Site Scripting Attacks: XSS Exploits and Defense*. Syngress, 2007.

[8] J. Grossman and T. Niedzialkowski. Hacking Intranet Websites from the Outside. Talk at Black Hat USA, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf, 2006.

[9] C. Heffner. How to Hack Millions of Routers. Talk at the Black Hat USA conference, 2010.

[10] I. Hickson. Html5. W3c working draft, W3C, May 2012. http://www.w3.org/TR/html5/.

[11] J. Hirth. It's Time to Rethink the Default Cache Size of Web Browsers. [online], http://kaioa.com/node/74, last access 8/5/2012, 2008.

[12] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). [IETF draft], http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec, Version 11, July 2012.

[13] C. Jackson and A. Barth. Beware of Finer-Grained Origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[14] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *In Proceedings of ACM CCS 07*, 2007.

[15] M. Johns. (somewhat) breaking the same-origin policy by undermining dns-pinning. Posting to the Bugtraq mailinglist, http://www.securityfocus.com/archive/107/443429/30/180/threaded, 2006.

[16] M. Johns and Kanatoko. Using Java in anti DNS-pinning attacks (Firefox and Opera). [online], Security Advisory, http://shampoo.antville.org/stories/1566124/, (08/27/07), Februar 2007.

[17] 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.

[18] C. Karlof, U. Shankar, J. Tygar, and D. Wagner. Dynamic pharming attacks and the locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007.

[19] G. Maone. NoScript Firefox Extension. [software], http://www.noscript.net/whats, 2012.

[20] A. Megacz. Firewall circumvention possible with all browsers. Posting to the Bugtraq mailinglist, http://seclists.org/bugtraq/2002/Jul/0362.html, July 2002.

[21] A. Megacz and D. Meketa. X-RequestOrigin. Internet Draft, http://tools.ietf.org/html/draft-megacz-x-requestorigin-00, June 2003.

[22] Microsoft. IE8 Security Part VII: ClickJacking Defenses, 2009.

[23] M. Mueller. Response to DNS spoofing attack. [Usenet posting], http://sip.cs.princeton.edu/news/sun-02-22-96.html, 1996.

[24] Princeton University. DNS Attack Scenario. [online], http://www.cs.princeton.edu/sip/news/dns-scenario.html.

[25] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, http://www.ietf.org/rfc/rfc1918.txt, February 1996.

[26] Rich internet application (ria) market share. http://www.statowl.com/custom_ria_market_penetration.php.

[27] B. K. Rios and N. McFeters. Slipping Past The Firewall. Talk at the HITBSecConf2007 conference, http://conference.hitb.org/hitbsecconf2007kl/agenda.htm, 2007.

[28] J. Roskind. Attacks Against the Netscape Browser. Talk at the RSA Conference, April 2001.

[29] D. Ross. Notes on DNS Pinning. [online], http://blogs.msdn.com/b/dross/archive/2007/07/09/notes-on-dns-pinning.aspx, last accessed 8/4/12, July 2007.

[30] J. Ruderman. The Same Origin Policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001.

[31] J. Soref. DNS: Spoofing and Pinning. [online], http://web.archive.org/web/20100211170613/http://viper.haque.net/~timeless/blog/11/, (07/07/12), 2003.

[32] B. Sterne and A. Barth. Content Security Policy. W3C Working Draft, http://www.w3.org/TR/2011/WD-CSP-20111129/, 2012.

[33] D. Stuttard. DNS Pinning and Web Proxies. NISR whitepaper, http://www.ngssoftware.com/research/papers/DnsPinningAndWebProxies.pdf, 2007.

[34] P. Uhley. Flash content and the same-origin policy. http://blogs.adobe.com/asset/2009/11/flash_content_and_the_same-ori.html, 2009.

[35] D. Ulevitch. Finally, a real solution to DNS rebinding attacks. [online], http://blog.opendns.com/2008/04/14/finally-a-real-solution-to-dns-rebinding-attacks/, last accessed 08/06/2012, April 2008.

[36] A. van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version WD-cors-20100727, http://www.w3.org/TR/cors/, July 2010.

[37] W3C. Same Origin Policy. [online], http://www.w3.org/Security/wiki/Same_Origin_Policy, (08/01/2012, 2010.