J Comput Virol (2008) 4:161–178
DOI 10.1007/s11416-007-0076-7

ORIGINAL PAPER

On JavaScript Malware and related threats

Web page based attacks revisited

Martin Johns

Received: 31 August 2007 / Revised: 7 November 2007 / Accepted: 21 November 2007 / Published online: 18 December 2007

Abstract The term *JavaScript Malware* describes attacks that abuse the web browser's capabilities to execute malicious script-code within the victim's local execution context. Unlike related attacks, JavaScript Malware does not rely on security vulnerabilities in the web browser's code but instead solely utilizes legal means in respect to the applying specification documents. Such attacks can either invade the user's privacy, explore and exploit the LAN, or use the victimized browser as an attack proxy. This paper documents the state of the art concerning this class of attacks, sums up relevant protection approaches, and provides directions for future research.

1 Introduction

> *We're entering a time when XSS has become the new Buffer Overflow*
> *and JavaScript Malware is the new shellcode.*
> Jeremiah Grossman [22]

Web browsers are installed on virtually every contemporary desktop computer and the evolution of active technologies such as JavaScript, Java or Flash has slowly but steadily transformed the web browser into a rich application platform. Furthermore, due to the commonness of Cross Site Scripting (XSS) [13] vulnerabilities the number of XSS worms [63] is increasing continuously. Therefore, large scale execution of malicious JavaScripts is a reality nowadays. Additionally, if no XSS flaw is at hand, a simple well written email usually suffices to lure a potential victim into visiting an innocent looking web page that contains a malicious payload. For all these reasons, the browser was recently (re)discovered as a convenient tool to smuggle malicious code on the victim's computer and behind the boundaries of the company's firewall. While earlier related attacks required the existence of a security vulnerability in the browser's source code or libraries, the attacks which are covered in this paper simply employ the legal means that are provided by today's browser technology.

Within this context, the term "JavaScript Malware" was coined by Grossman [22] in 2006 to describe this class of script code that stealthy uses the web browser as vehicle for attacks on the victim's local context. In this paper we discuss capabilities of such scripts in respect to invading the user's privacy (Sect. 3), exploiting the local LAN (Sects. 4 and 5), and abusing the victimized browser as an attack proxy (Sect. 6).

2 Circumventing the same origin policy

2.1 The JavaScript security model

For security reasons, certain functions of client-side browser technologies are subject to major restrictions. In this section, we describe these restrictions in respect to JavaScript, but similar concepts apply to other active client-side technologies such as Flash or Java applets.

The *Same Origin Policy (SOP)* is the fundamental security policy that applies to active client-side content that is embedded in web pages. It was introduced by Netscape Navigator 2.0 [62] and derives access right from the URLs

This work was supported by the German Ministry of Economics (BMWi) as part of the project "secologic", http://www.secologic.org.

M. Johns (✉)
Security in Distributed Systems (SVS),
Department of Informatics, University of Hamburg,
Vogt-Koelln-Str. 30, 22527 Hamburg, Germany
e-mail: johns@informatik.uni-hamburg.de

Table 1 The SOP in respect to the URL http://store.company.com/dir/page.html [62]

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

of the respective elements. More specifically, a given JavaScript is only allowed read and/or write access to properties of elements, windows, or documents that share the same origin with the script. In this context the origin of an element is defined by the *protocol*, the *domain* and the *port* that were used to access this element. See Table 1 for examples.

Effectively the SOP restricts a JavaScript that was received over the Internet to a sandbox defined by its serving web server. More precisely such a script is subject to the following restrictions:

1. No direct access to the local file system. From within a JavaScript/HTML context local files can only be referenced through the `file://` meta-protocol. An attempt by a script delivered through HTTP to directly access the target of such a reference would be a violation of the "protocol"-rule of the SOP.
2. No direct access to other hosts but the one that served the web page in which the script was included, due to the "domain"-rule of the SOP.
3. No direct access to other applications on the same host that are not hosted by the same web server, due to the "port"-rule of the SOP.

While port and protocol are fixed characteristics, JavaScript can influence the host property to mitigate this policy. This is possible because a webpage's host value is reflected in its DOM (Document Object Model [30]) tree as the `domain` attribute of the `document` object. JavaScript is allowed to set this property to a valid domain suffix of the original host. For example, a JavaScript could change `document.domain` from www.example.org to the suffix `example.org`. JavaScript is not allowed to change it into containing only the top level domain (i.e., `.org`) or some arbitrary domain value.

As stated above, the SOP defines the access rights of a given script. A JavaScript is only allowed access to elements that are part of a document which shares the same origin as the document the respective JavaScript is included in. In this respect, the SOP applies on a *document level*. Thus, if a JavaScript and a document share a common origin, the SOP allows the script to access all elements that are embedded in the document. Such elements could be, e.g., images, stylesheets, or other scripts. These granted access rights hold even if the elements themselves where obtained from a different origin.

Example The script http://exa.org/s.js is included in the document http://exa.org/i.html. Furthermore `i.html` contains various images from http://picspicspics.com. As the script's and the document's origin match, the script has access to the properties of the images, even though their origin differs from the script's.

2.2 JavaScript networking capabilities

JavaScript is limited to HTTP communication only. The JavaScript-interpreter possesses neither the means to create low-level TCP/UDP sockets nor other capabilities to initiate communication using other protocols. More precisely, there are two distinct ways for a JavaScript to create network connections: Direct and indirect communication:

Direct communication: With the term *direct communication* we denote the capabilities of a JavaScript to initiate a direct read/write HTTP connection to a remote host. For this purpose modern JavaScript implementation provide the XMLHTTPRequest-API [32] which was originally developed by Microsoft as part of Outlook Web Access 2000. XMLHTTPRequest allows the creation of HTTP GET and POST requests. The target URL of the request is subject to the same-origin policy, i.e., only URLs that satisfy the SOP in respect to the web page that contains the initiating script are permitted. This effectively limits a script to direct communication with the web application's origin host.

Alternatively direct communication can be accomplish by combining iframes with dynamically submitted HTML forms [52]: The JavaScript creates an HTML form inside an iframe and submits it, thus creating a GET or POST HTTP request. The server includes the requested data inside the HTTP request's response which replaces the iframe's content. As long as the SOP is satisfied in respect to the containing web page and the iframe's URL, the JavaScript can access the iframe's DOM tree to retrieve the response's data. Again, for a read/write communication the URL of the target host is restricted by the SOP.

Indirect communication: Furthermore, a JavaScript is able to initiate network communication *indirectly* via DOM tree manipulation. Some HTML elements employ URLs to

Table 2 Cross domain network capabilities

Type	Events[a]	Readable info[a]	Access rights[a]
IFrame	onload	–	–
Image	onload, onerror	width, height	–
Script	onload	–	known elements can be called and read

[a] Depending on the browser the actual capabilities may differ

reference remote data which is meant to be included in the web page, such as images, or scripts. If a web browser encounters such an element during the rendering process, it initiates a network connection in order to retrieve the referenced data. In this case, the URL of the remote entity is not restricted and can therefore point to cross-domain or cross-protocol targets. JavaScript is able to add elements to the DOM tree [30] of its containing page dynamically. By including elements that reference remote data, the JavaScript indirectly creates a network connection to the host that serves this data. Outgoing data can be included in the request by adding GET parameters to the elements URL.

In most cases indirect communication can only be used to send but not to receive data. An exception to this rule (besides the side-effect based channels that will be discussed in Sect. 2.3) can be created with the `script`-elements. By providing a remote-script with a local callback-function, the remote script can communicate data back to the calling script (see [57] for details). For instance, so called "web APIs" that export certain functionality of a web application and "web mashups", that employ such APIs to include cross-domain content dynamically into web pages, are often created this way.

2.3 A loophole in the SOP

As explained above, the direct cross-domain, cross-application and cross-protocol networking capabilities of JavaScript are restricted by the SOP. However, JavaScript is permitted to dynamically include elements from arbitrary locations into the DOM tree of its container document. This exception in the networking policy and the fact that the SOP applies on a document level creates a loophole in SOP: While a script has no direct access to remote targets that do not satisfy the SOP, the script can deduct various information from the process of including the remote element into the page's DOM tree:

- The script can receive and intercept events that might be triggered by the inclusion process.
- After the inclusion process has ended, the remote element is part of the same document as the script. Due to the document-level nature of the SOP, the script now has access to properties of the element that are readable through JavaScript calls.

Depending on the type of the element that was included in the document and the web browser, the JavaScript's specific capabilities to gain information by the inclusion differs (see Table 2). In the next sections we explain how this loophole can be exploited for malicious purposes.

The basic reconnaissance attack (BRA)

Based on the above identified loophole in the SOP, several reconnaissance techniques are possible. All these techniques share the same basic method which allows a binary decision towards the existence of a specified object. From here on we denote the underlying method as *basic reconnaissance attack (BRA)*. While not being the sole technique that is employed, variations of the BRA are used in several different contexts through the remainder of this paper.

The BRA utilizes JavaScript's event-handler framework which provides hooks to intercept various events that occur during the rendering of a web page. More specifically the events `onload` and `onerror` in combination with `timeout`-events are employed. Using these three indicators a JavaScript can conclude the outcome of an indirect communication (see Sect. 2.2) attempt. The BRA consist of the following steps (see Listing 1 for an example):

1. The script constructs a URL pointing to the remote entity of which the existence should be examined.
2. Then the script includes a suiting, network-aware element in the webpage that employs the constructed URL to reference the remote entity. Such elements can be, e.g., images, iframes or remote scripts.

```
<script>
function loaded(){
  // resource exists
}

function timed_out(){
  // resource does not exist
}

function err(){
  // requesting the resource created an
     error
}

i = new Image();
i.onload = loaded;
i.onerror = err;
window.setTimeout(timed_out,1000);

i.src = "http://target.tld/path";
</script>
```

Listing 1 The basic reconnaissance attack

3. Additionally, the script might initiate a `timeout`-event to receive information about the time needed for the inclusion process.
4. Using JavaScript's event-framework the script collects evidence in respect to the remote entity:
   - An `onload`-event signals the successful inclusion of the element and thus verifies its existence.
   - The indication given by an `onerror`-event depends on the specific context. This events is triggered either if the received data does not match the requirements of the respective element or alternatively if the network connection was terminated. Additionally, depending on the employed HTML element the JavaScript error console can be employed to gain further evidence.
   - The occurrence of the `timeout`-event prior to any other event related to the inclusion process indicates a pending network connection, hinting that the target host does not exist.

2.4 Cross site request forgery

Cross Site Request Forgery (CSRF/XSRF) a.k.a. *Session Riding* a.k.a *Sea Surf* is a client-side attack on web applications that uses *indirect communication* (see above) to exploit implicit authentication mechanisms (see Sect. 4.1). More precisely, CSRF uses the cross-domain, cross-application and cross-protocol capabilities provided by indirect communication initiate execute hidden, state-changing actions on the attacked web application.

The actual attack is executed by causing the victim's web browser to create HTTP requests to restricted resources. This can be achieved, e.g., by including hidden images in harmless appearing webpages. The image itself references a state changing URL of a remote web application, thus creating an HTTP request (see Fig. 1). As the browser provides this requests automatically with authentication information, the target of the request is accessed with the privileges of the person that is currently using the attacked browser. See [5,41,64] for further details.

3 Accessing local information: fingerprinting and privacy attacks

In this section we discuss various techniques that aim at collecting evidence concerning the malicious script's local execution context. More precisely, the here discussed attacks aim to gather information on the executing web browser, the local machine, and the browser's user. Such information include previously visited sites, indicators whether the user is currently logged into given web applications, characteristics of the local machine, and installed web browser add-ons.

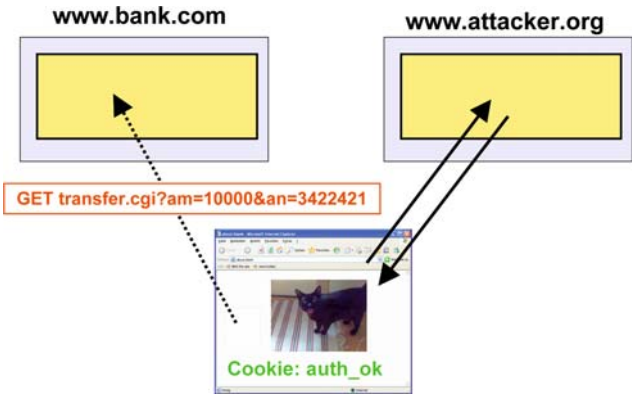

Fig. 1 A CSRF attack on an online banking site

While in some cases such attacks are targeted directly at the user's privacy, the adversary might also use his findings to identify existing vulnerabilities in order to decide on the next step in his attack.

3.1 Subverting the user's privacy with cascading style sheets

3.1.1 Browser history disclosure using unique background-picture URLs

In 2002, Clover [10] showed how CSS can be employed to examine whether a given URL is contained in the browser's history of visited pages. The underlying mechanism, that is the basis of the attack, dates back to the early days of web browsers: The colour used by the browser's rendering engine to display a hyperlink varies, depending on if the user has visited the referenced site before. This behaviour was adopted by CSS, resulting in the hyperlink pseudo-class `a:visited`. Using this pseudo-class, visited links can be outfitted with the complete set of CSS properties, including the capability to include remotely hosted images to be used as background pattern. Thus, sophisticated construction of lists of hyperlinks and an according style sheet can be employed to initiate indirect communication of private data:

The attacker compiles a list of URLs that should be matched against the user's history. For each of these URLs he creates a hyperlink labelled with a unique id-attribute. Furthermore, he fabricates a style-sheet that contains unique `visited`-selectors which are linked to the respective id-attributes. By referencing an attacker-controlled cgi-script in the CSS-selectors the adversary is able to deduct which of the sites are contained in the browser's history (see Listing 2).

```
1 <style>
2   #ebay:visited { background: url(http://
3   evil.com/visited.cgi?site=ebay); }
4 </style>
5
6 <a id="ebay" href="http://www.ebay.com"></a>
```

Listing 2 CSS visited page disclosure [10]

3.1.2 Dynamic browser history disclosure with JavaScript

Furthermore, as discussed by Clover [10] and refined by Grossman [21], it is also possible to accomplish the same results completely on the client-side without requiring a remote counterpart: JavaScript is able to read the applied CSS-style (using the `getComputedStyle`-method on Mozilla based browsers or IE's `currentStyle`-property): The attacker creates a CSS style-sheet that defines two distinct styles for visited and unvisited hyperlinks. For each tested hyperlink, a JavaScript reads the style information that the browser applied to the element. Depending on which of the two predefined styles is returned by the query, the script can decide whether the user has visited the site in the recent past (see Listing 3).

Usage of JavaScript to execute this attack provides several advantages for the adversary: The malicious JavaScript can construct the list of hyperlinks dynamically. This allows more targeted, incremental attacks. Furthermore, only a rather simple style-sheet defining two separate styles is required. This eliminates the need for a list of unique selectors, thus reducing the attack-code significantly.

```
<style>
  a:visited { color: rgb(0,0,255) }
</style>

<a id="ebay" href="http://www.ebay.com"></a>

<script>
var link = document.getElementById('ebay');
var color = document.defaultView.getComputed
Style(link,null).getPropertyValue("color");
if (color == "rgb(0, 0, 255)") {
    // found
}
</script>
```

Listing 3 CSS visited page disclosure using JavaScript [21]

Utilizing the flexibility provided by JavaScript based dynamic URL-construction, it has been proposed to extend this technique to establish a list of terms that the user has looked up using Internet search engines. This undertaking requires examining large numbers of URLs, as most search engines employ URL-schemes that contain numerous, context dependent, and frequently changing parameters resulting in rather large sets of possible URLs for non-trivial search terms. SPI Labs [50] has shown that it is possible to stealthy check for more than 40,000 unique URLs in five seconds using recent hardware, thus rendering this privacy attack feasible.

3.2 Privacy attacks through timing attacks

Besides CSS based attacks on the user's privacy, a whole class of similar targeted timing attacks has been discussed.

3.2.1 Browser cache disclosure

Felten et al. [16] documented in 2000 how attackers can initiate timing attacks to determine if a given web browser has visited a certain web page in the recent past. For this purpose the adversary employs cacheable web-objects such as static images and measures the time it takes to retrieve a given object.

For example the attacker wants to know whether the victim has been to http://company.com recently. For this purpose, the malicious script stealthily embeds the logo of the targeted site (http://company.com/logo.jpg) into its DOM tree, causing the web browser to request the image. Before its actual inclusion in the web page, the `image`-element is outfitted with an `onload`-eventhandler to measure the time that the inclusion process takes. If the victim has been to the respective site in the last days, the image is still in the browser's cache, resulting in a very short loading time. If not, the image has to be retrieved over the Internet, causing a significant longer loading process.

By measuring and matching the loading time against a certain threshold the attacker can conclude if the image is in the browser's cache and thus, whether the victim has been to the targeted web site.

3.2.2 Assessing the contents of non-cacheable pages

In 2007, Bortz et al. [4] extended Felten's technique towards non-cacheable web-objects. Their technique is based on the observation that the time required to finish an HTTP request depends on two factors: the time it takes to deliver the content over the Internet and the time the web server consumes to create the content. While the first factor is mostly constant for a given network location, the second factor depends on the actual query. A request for a static element (like a predefined image) can be computed by the web server in a very short time. However, if business logic and database queries are involved, the time to create the response's HTML document differs heavily in dependence to the page's final content.

Using this observation, Bortz et al. introduced *cross-site timing attacks* which employ a technique related to the BRA (see Sect. 2.3) to time the loading process of web pages. Their technique is based on dynamically creating a hidden `img`-tag that points to the targeted web page. This image-element is outfitted with an `onerror`-eventhandler which is triggered as soon as the first data-chunk is received by the browser, as instead of image-data the response consists of HTML code. By measuring the time-difference between the actual image-creation process and the occurrence of the `onerror`-event, the loading-time of the web page can be measured.

Furthermore, the described technique requires two timing sources in order to factor out the timing overhead that is introduced by the network. For this reason, first a request to a static element of the targeted web site is created. Such an element could be for example an image or a 404 error page. As for such elements almost no processing time is required by the respective web server, the time to load the element

is roughly equal to the network-induced overhead. Therefore, the result of this step can be used for future reference. The second request is aimed at the actual target of the privacy attack. By calculating the difference between the prior obtained reference value and the time it took to complete the loading process, an estimate of the server-side computing time can be obtained. Using this technique, Bortz et al. showed the possibility to determine, e.g., if a given browser is logged into a given web application, or how many items are currently contained in the user's shopping cart of a given online-retail store.

3.2.3 Precision of timing attacks

The precision of browser-based timing attacks is limited by the precision of the employed timing-functions. JavaScript provides the `Date()`-object. This object's smallest unit of time is a millisecond, thus setting an according lower bound for the precision of attacks that are written completely in JavaScript. However, Java's timing-function `nanoTime()` provides a timer that returns time to the nearest nanosecond. Meer and Slaviero discussed in [55] how this timer can be used within JavaScript timing attacks, utilizing either a special purpose JavaApplet or the LiveConnect-feature [7].

3.3 CSRF-based privacy attacks

3.3.1 Establishing if the user is currently logged into a given web application

Using a variation of the BRA (see Sect. 2.3), Grossman [20] discussed a method to disclose whether the web browser possesses valid authentication credentials (for instance session cookies or http-authentication information) for a given web application: The basis of the attack is to test a browser's capability to load a web page that is only accessible to authenticated users.

The attack utilizes advanced error-handling which is provided by modern JavaScript interpreters. In the case of an error within a JavaScript, the `window.onerror`-event provides limited access to the JavaScript error-console by communicating a short error-message, the URL of the triggering script, and a numeric error code. This additional information can be employed for fingerprinting purposes. By attempting to load an HTML page into a `script`-tag by using the page's URL in the tag's `src`-attribute, inevitably a JavaScript parsing error is triggered, as the included data is HTML and cannot be parsed by the JavaScript interpreter. Furthermore, depending on the authenticated state of the requesting entity, a web application responds to a request for a restricted resource with different HTML content (either with the requested page or alternatively with an error page/a login form). In many cases different HTML code leads to distinct parsing errors. Therefore, a malicious script can, by intercepting the error-code and messages, differentiate between parse errors triggered by either response and thus determining whether the browser's user is currently logged into the application (see Listing 4).

```
<script>
function err(msg, url, code) {
    if ((msg == "missing } in XML expression")
     && (code == 1)) {
        // logged in
    } else if ((msg == "syntax error" ) &&
    (code == 3)) {
        // not logged in
    }
}
window.onerror = err;
</script>

<script src="http://webapp.org"></script>
```

Listing 4 JavaScript login checker [23]

In addition, Robert Hansen discussed using restricted images for the same purpose [27]. Some web applications grant or deny access to resources like images depending on the user's authenticated state. By employing the BRA to verify whether the images are accessible or not, the malicious script can conclude if the user is currently logged into the application.

3.3.2 Browser and local machine fingerprinting

Various local resources provided either by the local machine or the web browser are accessible by specialized URL-schemes, such as `file://`, `chrome://`, or `res://`. In late 2006 and 2007 several techniques have been discussed that employ these URL-schemes to gather information concerning the malicious JavaScript's local execution context. The majority of these attacks is based on the BRA. By dynamically including a cross-protocol resource in a webpage and intercepting `onload`- and `onerror`-events, certain characteristics of the local context might be disclosed. More specifically it has been shown how to:

- compile a list of installed Firefox extensions using `chrome`-URLs [26],
- establish which software is installed on the local machine using `res`-URLs [54],
- execute a dictionary attacks to disclose Windows user accounts [71] or drive names [48],
- read Firefox settings [72],
- and check the existence of local files using the `file`-handler [66].

As special-purpose URL-schemes are often treated differently by the various web browser, most of these techniques only work if a certain execution context (defined by browser, browser version, and operating system) is given, thus limiting the respective attack vector to a subset of browsers or operating systems. Furthermore, many of these issues can be

regarded rather as implementation faults than fundamental flaws rooted in the web-paradigm and thus should be resolved in the near future. For these reasons, a further discussion of these techniques is omitted in this paper.

4 Circumventing the firewall: Intranet reconnaissance and exploitation

4.1 Using the firewall as a means for implicit authentication

With the term *implicit authentication* we denote authentication mechanisms, that do not require further interaction after the initial authentication step. For example the way HTTP authentication is implemented in modern browsers requires the user to enter his credential for a certain web application only once per session. Every further request to the application's restricted resources is outfitted with the user's credentials automatically.

Furthermore, with the term *transparent implicit authentication* we denote authentication mechanisms that also execute the initial authentication step in a way that is transparent to the entity that is being authenticated. For example NTLM authentication [17] is such an authentication mechanism for web applications. Web browsers that support the NTLM scheme obtain authentication credentials from their underlying operating system. These credentials are derived from the user's operating system login information. In most cases the user does not notice such an automatic authentication process at all.

Especially in the intranet context, transparent implicit authentication is used frequently. This way the company makes sure that only authorized users access restricted resources without requiring the employees to remember additional passwords or execute numerous, time-consuming authentication processes on a daily basis.

A company's firewall is often used as a means of transparent implicit authentication: The intranet server are positioned behind the company's firewall and only the company's staff has access to computers inside the intranet. As the firewall blocks all outside traffic to the server, it is believed that only members of the staff can access these servers. For this reason intranet server and especially intranet web server are often not protected by specific access control mechanisms. For the same reason intranet applications often remain unpatched even though well known security problems may exist and home-grown applications are often not audited for security problems thoroughly.

4.2 Using a webpage to execute code within the firewall perimeter

As motivated in Sect. 1 many companies allow their employees to access the WWW from within the company's network.

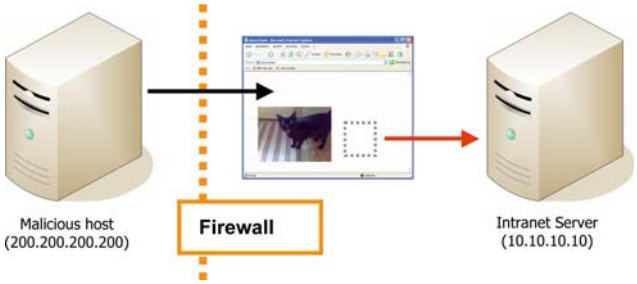

Fig. 2 Using a webpage to access restricted web servers

Therefore, by constructing a malicious webpage and succeeding to lure an unsuspecting employee of the target company into visiting this page, an attacker can create malicious script code that is executed in the employee's browser. As discussed in Sect. 2 current browser scripting technologies allow certain cross-protocol, cross-domain, and cross-host operations that can be used for reconnaissance attacks.

The employee's browser is executed on a computer within the company's intranet and the employee is in general outfitted with valid credentials for possibly existing authentication mechanisms (see Sect. 4.1). Consequently, any script that runs inside his browser is able to access restricted intranet resources with the same permissions as the employee would (see Fig. 2).

4.3 Intranet reconnaissance attacks

This section documents various techniques that aim to collect information concerning a given intranet (Fig 3).

4.3.1 Portscanning the intranet

It was shown by various parties [24,49] how malicious web pages can use indirect communication to port-scan the local intranet. While the discussed techniques slightly differ, they all are variants of the BRA (see Sect. 2.3):

1. The script constructs a local HTTP URL that contains the IP-address and the port that shall be scanned.
2. Then the script includes an element in the webpage that is addressed by this URL. Such elements can be, e.g., images, iframes or remote scripts.
3. Using JavaScript's time-out functions and eventhandlers as discussed in Sect. 2.3 the script can decide whether the host exists and the given port is open: If a time-out occurs, the port is probably closed. If an `onload`- or `onerror`-event happens, the host answered with some data, indicating that the host is up and is listening on the targeted port.

Limitation: Some browsers like Firefox enforce a blacklist of forbidden ports [60] that are not allowed in URLs. In this case JavaScript's port scanning abilities are limited to

ports that are not on this list. Other browsers like Safari allow access to all ports.

4.3.2 Selecting potential attack targets

To launch such an discovery attack, the malicious script needs to have knowledge about possible reconnaissance targets. Such knowledge can either be the intranet's IP-range or the internal DNS names of local hosts.

IP discovery with Java: In case the local IP-range is unknown to the attacker, he can use the browser's Java plug-in to obtain the local IP-address of the computer that currently executes the web browser which is vehicle of the attack.

Unlike JavaScript, the Java plug-in provides low-level TCP and UDP sockets. The target address of network connections opened by these sockets is restricted by Java's version of the same-origin policy which only allows connections to the IP address from which the Java applet was originally received. After being instantiated, a Java `Socket`-object contains full information about the connection including the IP addresses of the to connected hosts. Thus, by creating a socket and using the socket-object's API, the browser's local IP address can be read by a Java-applet [47] and subsequently exported to the JavaScript scope.

Additionally, Mozilla based and Opera web browser allow JavaScript to directly instantiate Java objects using the Live Connect-feature [7]. This removes the attacker's necessity to provide a separately hosted Java applet, thus allowing more self-contained attacks:

```
function natIP() {
  var w = window.location;
  var host = w.host;
  var port = w.port || 80;
  var Socket = (new java.net.Socket(host,
  port)).getLocalAddress().getHostAddress();
  return Socket;
}
```

Listing 5 Obtaining the local IP address

DNS bruteforcing: In the case that the execution of Java content is disabled in the attacked web browser, the adversary can resort to bruteforcing internal DNS names. Robert Hansen documented in [28] that many companies employ the same DNS server to resolve both their external and their internal hostnames (see Listing 6). The attacker can either try to obtain the full list of DNS entries using a zone transfer or by bruteforcing well known names for intranet servers (e.g., intranet.company.com; [29] list more than 1,500 of possible hostnames). If such external lookups for internal servers is possible, the attacker can gain knowledge on the IP-range used in the intranet this way.

```
...
10.0.1.10    intranet.godaddy.com
10.210.136.22    internal.iask.com
10.25.0.31   intranet.joyo.com
10.30.100.238    intranet.shopping.com
10.50.11.131     intranet.monster.com...
...
```

Listing 6 Misconfigured DNS servers leaking internal IP addresses [28]

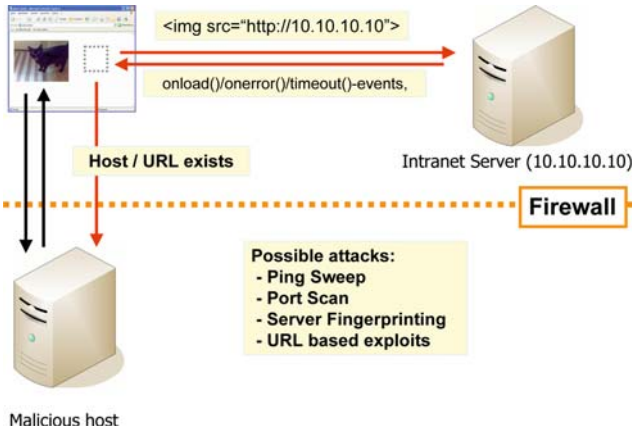

Fig. 3 Web page based reconnaissance of the intranet

If the respective company does not leak the internal IP-range through misconfigured DNS servers, the attacker can use the list of known DNS names to test if the targeted intranet contains hosts that are assigned to one of these names. To do so the malicious JavaScript has two options: It can either employ a series of BRAs using the guessed internal domain names as part of the URL (i.e., testing if a host is assigned to http://intranet exists). Alternatively the script can check if any of the domain names are contained in the browser's history using any of the techniques discussed in Sects. 3.1, 3.1.2, or 3.2. Both methods do not suffice to directly leak the actual IP-range to the attacker. However, using these methods the attacker might identify possible targets for further attacks in the intranet. Furthermore, in certain cases the existence of certain internal domain names provides strong indications for the employed IP-range. For instance, the home router "Fritz Box" [18] introduces the domain name `fritz.box` to the intranet, pointing to the router's admin interface. By using the BRA to test for http://fritz.box the attacker is able to learn that a Fritz Box is used in the examined intranet and based on the knowledge can guess the used IP-range to be equal or close to the default range used by this specific router (e.g., 192.168.172.0/24).

4.3.3 Fingerprinting of intranet hosts

After determining available hosts and their open ports, a malicious script can try to use fingerprinting techniques to get more information about the offered services. Again the script has to work around the limitations that are posed by the SOP. For this reason the fingerprinting method resembles closely the port-scanning method that was described above [24,49].

The basic idea of this technique is to request URLs that are characteristic for a specific device, server, or application. If such a URL exists, i.e., the request for this URL succeeds, the script has a strong indication about the technology that

is hosted on the fingerprinted host. For example, the default installation of the Apache web server creates a directory called "icons" in the document root of the web server. This directory contains image files that are used by the server's directory listing functionality. If a script is able to successfully access such an image for a given IP-address, it can conclude that the scanned host runs an Apache web server. The same method can be used to identify web applications, web interfaces of network devices or installed scripting languages (e.g., by accessing PHP easteregg s).

4.3.4 Avoiding and brute-forcing HTTP authentication

If during the reconnaissance step, the malicious JavaScript encounters a resource that is protected by HTTP authentication to which the browser not currently possesses valid credentials, the web browser displays an authentication dialogue. This in fact discloses the ongoing attack and alarms the victimized browser's user that something unusual is going on.

Esser [15] documented a method that can be employed by the attacker to suppress such authentication pop-ups. Esser's technique is based on the creation of malformed HTTP URLs. If such an URL is sent to the web server, the server identifies the error in the URL before determining the addressed resource. Therefore, the server does not map the URL to a hosted resource and therefore does not recognize that a HTTP authentication dialogue should have been trigger: Instead of replying with a "401 Authorization Required" status code, the server responds with "400 Bad Request". Esser documented two ways to create such URLs that work with most web servers: Either incomplete URL entities (e.g., http://10.10.10.10/%) or overly long URLs (e.g., http://10.10.10.1/AAA...AAA) that exceed the server's restrictions on URL size. Using Esser's method, the adversary is still able to determine whether the host in question exists. However, determining if the examined IP hosts a web server and using the fingerprinting technique detailed in Sect. 4.3.3 do not work in combination with this evading method, as these techniques require valid HTTP responses.

Additionally, Esser [14] demonstrated how Firefox's `link`-tag can be abused to execute brute-force attacks on URLs that are protected by HTTP authentication. As the content that is requested through `link`-tags is regarded as optional by the browser, the tag does not initiate authentication pop-ups if it encounters a 401 response. Furthermore, the rendering process of the page halts until the request that was initiated by the `link`-tag has terminated. Thus, by using the http://username:password@domain.tld-scheme the attacker can iterate through username/password-combinations and measure his success using the timing attacks that were discussed in Sect. 3.2.

4.3.5 Technical Limitations of reconnaissance attacks

As employing the BRA for reconnaissance purposes depends on the usage of `timeout`-events, any attack in this class is subject to throughput-limitations induced by the timing-induced overhead. While detecting existing resources takes only a very short amount of time, probing a non-existing resource requires at minimum the full timeout-period. As the precision of the attack is related to the length of the chosen timeout-period, the attacker has to decide between speed and accuracy.

A series of reconnaissance probes can be accelerated by parallelizing the requests, e.g., by creating several hidden image-elements at the same time. However, the operating systems and web browsers enforce various limitations in respect to parallel connections. For instance, Windows XP/SP2 does not allow more than 10 outstanding connections at the same time and the Firefox web browser only permits a maximum of 24 simultaneous established connections.

Lam et al. [51] documented that in their experiments they were able to achieve maximum scanning rates between approximately 60 scans/min (Windows XP SP2) and 600 scans/min (Linux). In addition, Grossman et al. [23] discussed that JavaScripts `stop`-function can be used to terminate outstanding connections in order to speed up the scanning process.

4.4 CSRF attacks on intranet servers

After discovering and fingerprinting potential victims in the intranet, the actual attack can take place. A malicious JavaScript has for example the following options:

Exploiting unpatched vulnerabilities: Intranet hosts are frequently not as rigorously patched as their publicly accessible counterparts as they are believed to be protected by the firewall. Thus, there is a certain probability that comparatively old exploits may still succeed if used against an intranet host. A prerequisite for this attack is that these exploits can be executed by the means of a web browser [24].

Opening home networks: The following attack scenario mostly applies to home users. Numerous end-user devices such as wifi routers, firewall appliances or DSL modems employ web interfaces for configuration purposes. Not all of these web interfaces require authentication per default and even if they do, the standard passwords frequently remain unchanged as the device is only accessible from within the "trusted" home network. If a malicious script was able to successfully fingerprint such a device, there is a certain probability that it also might be able to send state changing requests to the device. In this case the script could, e.g., turn of the firewall that is provided by the device or configure the forwarding of certain ports to a host in the network, e.g., with the result that the old unmaintained Windows 98 box in the

cellar is suddenly reachable from the Internet. Thus, using this method the attacker can create conditions for further attacks that are not limited to the web browser any longer. Furthermore, Stamm et al. documented in [68] that by changing the device's DNS settings this way, the attacker can initiate a persistent "drive-by pharming" attack.

4.5 Cross protocol communication

Based on [70], Wade Alcorn [1,2] showed how multi-part HTML forms can be employed to send (semi-)valid messages to ASCII-based protocols. Prerequisite for such an attempt is that the targeted protocol implementation is sufficient error tolerant, as every message that is produced this way still contains HTTP-meta information like request-headers. Alcorn exemplified the usage of an HTML-form to send IMAP3-messages to a mail-server which are interpreted by the server in turn. Depending on the targeted server, this method might open further fingerprinting and exploitation capabilities.

4.6 Attacks that do not rely on JavaScript

Intranet exploration attacks like portscanning do not necessary have to rely on JavaScript. It has been shown recently [3] that attacks similar to the vectors show in Sect. 4.3.1 can be staged without requiring active client-side technologies. Instead timing analysis is employed.

Currently these attacks rely on a certain, not-standardized behaviour of the Firefox web browser: In general whenever a browser's rendering engine encounters an HTML element that includes remote content into the page, such as `image`, `script` or `iframe`-tags, the browser sends an asynchronous HTTP request to retrieve the remote resource and resumes rendering the web page. However, the `link`-tag does not adhere to this behaviour. Instead the rendering engine stops the rendering process until the HTTP request-response pair, that was initiated because of the tag, has terminated. Thus, by creating a webpage that contains a `link`-element, that references a local URL, and an `image`-element, that is requested from the attacker's host, the attacker can use timing analysis to conclude if in fact an actual host can be reached under a given local URL. Employing this technique, an attacker can reliably create a mapping of the local lan. However, the timing differences between the response time of a RST-package, that was generated because of a closed port, and an actual HTTP-response are hard to measure from the attacker's position. For this reason fingerprinting attacks are not yet feasible. As research in the area of these attack techniques is comparatively young and web browsers are still evolving, it is probable that there exist more attack vectors which do not rely on active technologies.

5 DNS rebinding attacks

DNS rebinding is a powerful technique to undermine the SOP. It was originally discussed 1996 by [65] in respect to Java applets. In 2002 [56] showed that JavaScript's SOP is affected by the same issue. The attack is also known as "anti DNS pinning" and "Quick Swap DNS".

5.1 Leaking intranet content

As discussed above, the SOP should prevent read access to content hosted on cross-domain web servers. In 1996 [65] showed how short lived DNS entries can be used to weaken this policy.

Example: Attacking an intranet host located at 10.10.10.10 would roughly work like this:

1. The victim downloads a malicious script from www.attacker.org.
2. After the script has been downloaded, the attacker modifies the DNS answer for www.attacker.org to 10.10.10.10.
3. The malicious script requests a web page from www.attacker.org (e.g., via loading it into an iframe).
4. The web browser again does a DNS lookup request for www.attacker.org, now resolving to the intranet host at 10.10.10.10.
5. The web browser assumes that the domain values of the malicious script and the intranet server match. Thus, the SOP is satisfied and browser grants the script unlimited access to the intranet server.

Using this attack, the script can access the server's content. With this ability the script can execute refined fingerprinting, leak the content to the outside or locally analyse the content in order to find further security problems.

To counter this attack most modern browsers employ "DNS pinning": The mapping between a URL and an IP-address is kept by the web browser for the entire lifetime of the browser process even if the DNS answer has already expired.

5.2 Breaking the browser's DNS pinning

While in general DNS pinning is an effective countermeasure against such an attack, unfortunately there are scenarios that still allow the attack to work: Josh Soref has shown in [67] how in a multi session attack a script that was retrieved from the browser's cache still can execute this attack. Furthermore, we have shown [39] that current browsers are vulnerable to breaking DNS pinning by selectively refusing connections: After the DNS rebinding took place, the attacker's web server refuses connection attempts from the victim. These failed

connections cause the web browser to renew its DNS information for the respective domain, thus receiving the new mapping.

Following our disclosure, Kanatoko [44] found out that many browsers also drop their DNS pinning when the browser tries to access a closed port on the attacker's host. Thus, dynamically requesting an image from http://attacker.org:81 is sufficient to propagate the new DNS information to the browser. See [34] for a full list of the different browser's pinning implementation which was compiled late 2007.

Additionally, Stuttard [69] pointed out that the web browser's DNS pinning does not apply when a web proxy is being used, because in this situation DNS resolution is performed by the proxy, not the browser. As he documented, until now non of the widely deployed web proxies take DNS rebinding attacks into account, thus rendering any browser-based countermeasures useless.

5.3 DNS rebinding and Java

As detailed above, DNS rebinding attacks were originally invented to undermine the SOP of Java applets [65]. For this reason the JVM introduced strict DNS pinning in 1996 [59]. The JVM maintains its own DNS pinning table that is populated separately from the browser's internal pinning table. This table is kept for the complete livetime of the JVM-process. However, several approaches to undermine the JVM's DNS pinning mechanism have been disclosed recently.

5.3.1 LiveConnect

As explained in Sect. 4.3.2, some web browsers allow JavaScript to directly instantiate Java objects using the LiveConnect-feature [7]. If a networking object like a socket is created this way, the JVM immediately initiates a DNS query to pin the object to the documents origin. We have shown [40] that if at this point the DNS setting for the respective domain already has been rebound to the internal address, the JVM pins the object to the internal IP address. This way the attacker is able to employ Java objects that are available through Java's standard library in his attack.

5.3.2 Java applets and proxies

Byrne [6] documented that in cases where the browser and the JVM are configured to use an outbound HTTP proxy, the JVM can be tricked into pinning the internal IP address:

1. The victim's web browser tells the proxy to load a web page using the URL http://attacker.org/index.html.
2. The proxy queries the adversary's DNS server for the IP address of `attacker.org`.
3. The DNS server replies with the server's address (e.g., 200.200.200.200).
4. The proxy retrieved the web page and passes it on to the browser.
5. The web page contains an embedded Java applet which is hosted at http://attacker.org/bad.class.
6. The browser's rendering engine encounters the `applet`-element and instantiates the JVM with the applet's URL.
7. The JVM queries the attacker's DNS server for `attacker.org`'s IP address, bypassing the proxy.
8. The DNS servers replies to this second request with the internal IP address (e.g., 10.10.10.10), which is pinned by the JVM.
9. Then, the JVM initiates the retrieval of the applet's code. However, as the JVM is configured to use the HTTP proxy, the applet itself is requested by the proxy which still uses the server's original IP address.

The result of this series of events is that the applet which was retrieved by the proxy from 200.200.200.200 is pinned by the JVM to 10.10.10.10, thus enabling the attacker to use it as part of his rebinding attack.

5.3.3 Cached Java applets

To minimize loading times, the JVM maintains a cache of previously loaded applets. In this cache the applets are stored accordingly to the respective HTTP `expires`-header. In addition to the applet itself, the cache also contains the applet's original URL. Only if an applet is requested from the exact same URL, the cached version of the applet will be used.

Rios and McFeters [61] have documented how this behaviour can be exploited in a multi session attack: For this method to work, the victim has to be tricked twice into executing the malicious code. The first time the code solely loads the applet from `attacker.org` using the server's real IP address. This process only serves the purpose to store the applet in the JVM's cache. After the applet has been delivered, the attacker changes the DNS mapping of `attacker.org` to point to the targeted internal address. If, in a new browser session, the victim visits again a page that is controlled by the attacker, the malicious code dynamically includes the applet in the page using the exact same URL as in the attack's first step. As the URL matched the cache entry, the applet is retrieved from the cache. However, as the cache only contains the URL but not the actual IP address, the JVM queries the attacker's DNS server for the IP of `attacker.org` and consequently pins the applet to the internal address.

In addition Rios discussed that the same behaviour might be triggered either by loading new JVM instances through URL handlers (like `picassa://`) or by enforcing the

execution of the applet using a different Java version (a capability that Sun intends to disable in the future) [61].

5.3.4 Attack capabilities provided by Java

Java provides full TCP and UDP socket connections to the targeted host. This can be used for, e.g., port-scanning, fingerprinting of non-HTTP services, or communication using arbitrary protocols. In cases where the network services of the targeted intranet host are not fully patched, the adversary could use arbitrary, well known exploits to, e.g., trigger buffer overflow vulnerabilities.

Furthermore, as Rios and McFeters [61] pointed out the availability of mature Java libraries for virtually every purpose enables the adversary to easily and quickly create attack code that targets non-trivial or obscure network services.

5.4 DNS rebinding and Flash

Kanatoko [45] demonstrated, that Flash player plug-in is also susceptible to DNS rebinding attacks. The Flash player does neither inherit the browser's pinning table nor implements DNS pinning itself. Like Java applets, the Flash player's scripting language ActionScript supports low level socket communication, thus extending the adversary's capabilities towards binary protocols. Using this capability non-HTTP attacks like full portscans on the intranet or accessing binary protocols are possible. Using the capabilities, Duong [12] demonstrated how to uses a malicious Flash file to run a TCP socket relay in the victim's browser. Using a similar approach, Kaminsky [43] was able to tunnel arbitrary TCP protocols into the internal network. He exemplified this capability by running Nessus-scans on intranet hosts.

In order to evaluate the attack surface, Jackson et al. ran a Flash 9 advertizement on a minor advertizing network. Within three days they temporary compromised more than 25,000 internal networks while paying less than 50 USD [34].

6 Using the web browser as an attack proxy

In addition to attacks that either target the victim's privacy (Sect. 3) or the browser's LAN (Sect. 4), it has also been discussed how active, client-side content could be employed to attack third parties. In such cases the victims web browser is misused by the adversary as an attack proxy. This way the attack's real origin can be obfuscated as the actual network connections come from the hijacked browser's IP address.

This topic has received comparatively little attention in the past. Therefore, we discus only three selected techniques in this section. However, we anticipate this class of attacks to gain momentum in the future, especially since Jackson et al. have demonstrated in [34] that distributing a malicious Flash advertizement can provide the adversary with capabilities comparable to a mid-sized botnet.

6.1 Scanning Internet web applications for vulnerabilities

With the birth of the so called "Web 2.0" phenomena, various web applications started to offer services that can be included seamlessly into other web pages. In many cases, these services are realized through cross-domain `script`-tags that in turn invoke pre-defined callback functions [57]. Other services transfer various cross-domain content into one single domain, thus allowing cross-domain interaction. Hoffman [31] demonstrated how victimized web browsers can abuse such services to scan third party web applications for XSS vulnerabilities. Instead of relying on DNS rebinding in order to establish cross-domain read/write HTTP connections, they employ Web 2.0 services which provide access to arbitrary cross-domain web content.

In his example, Hoffman used "Google Translate" [19], a service that offers to translate complete web pages from one natural language into another one while preserving the page's HTML markup and active client-side content. The service is invoked by providing an URL to the web page that is supposed to be translated and then provides the translation result hosted on the service's domain. The adversary first requests the translation of a page that contains his malicious JavaScript. This way the script runs on a webpage that belongs to the translation service's domain. Then the script itself requests the translation of a page belonging to the target site. As the result of this translation again is hosted on the service's domain, the malicious script gains complete read access to the foreign page. This way, the adversary's script can spider and analyse complete web applications. The translation step does not interfere with the scripts purpose, as the adversary is mainly interested in the web site's structure which is defined by its HTML markup. This way the script is able to identify potential insertion points for XSS attacks and evaluate if these points indeed represent vulnerabilities.

As the adversary only uses a hijacked browser and a public available Internet service in his scan, the real source of the vulnerability scan is hidden effectively.

6.2 Assisting worm propagation

Lam et al. [51] discussed how web browsers can be abused to aid the propagation of worms that spread through vulnerabilities in web applications. Such vulnerabilities are usually exploited by requesting specially crafted URLs. Therefore, the actual attack can be executed by any web browser via CSRF.

In Lam et al.'s worm propagation model, a worm-infected web server adds the malicious client-side code to all served web pages. This code then uses the web browsers that have received the code to spread the worm further: First, the victimized web browsers try to find vulnerable web servers using the BRA. If a potential attack target has been located, the client-side script executes the attack by sending the exploit using JavaScript's indirect communication capabilities. Finally, the freshly infected web server also adds the malicious payload to its own web pages, thus helping to spread the worm further.

Additionally, Lam et al. [51] shows how the resulting network of infiltrated web browsers can be abused for distributed denial of service attacks.

6.3 Committing click-fraud through DNS rebinding

Payment models of online advertizement networks are based on the amount of visitors that reach the advertized site through the respective ad. To counter naive fraud attempts, online ads employ a random nonce as part of interaction between the promoted site and the page that carries the advertizement. As the content of this nonce is protected by the SOP, an adversary cannot create accountable clicks using CSRF.

However, as shown above, Java or Flash based DNS rebinding attacks allow socket connections to arbitrary network locations. This enables browser-based cross domain read/-write HTTP connections to both sites—the site that carry ads and the promoted sites themselves. This capability empowers the attacker to initiate sophisticated click-fraud attacks that go as far as simulating realistic click-through site access on the advertized site. Jackson et al. [34] exemplified how this way large scale rebinding attacks can be used to commit lucrative and hard to detect click-fraud.

7 Protecting against the specified threats

7.1 General observations

7.1.1 The fundamental dilemma

As already motivated in Sect. 1, all actions that are executed by the discussed attacks are indeed to be considered "legal" in respect to the applying specifications, RFCs and Internet drafts. The ability to include, load, and link to cross-domain resources is a fundamental characteristic of the hypertext-Internet. Thus, prohibiting cross-context requests is not an feasible option. However, solving the problems by introducing new restrictions on active client-side code might also prove to be insufficient due to attacks that do not rely on client-side code (see Sects. 3.1, 3.2, or 4.6) and the amount of possible side-channels.

7.1.2 Cross site request forgery

From an abstract point of view, all threats that were specified in this paper are CSRF attacks, as they all rely on the adversary's capability to create hidden cross-domain, cross-protocol, or cross-host requests.

As explained in Sect. 2.4, in general CSRF attacks target implicit authentication mechanisms, e.g., by creating hidden HTTP requests that contain valid session cookies. The attacks covered in this paper target authentication mechanisms which are based on (physical) location: The user's web browser provides access to the browser's local attributes (like its cache) and to the browser's local LAN. In both cases the location is used as an mean of implicit authentication: Only content running inside the browser has access to the internal attributes and only computers behind the firewall are allowed to access the LAN, thus as discussed in Sect. 4.1, rendering the firewall to a means of transparent implicit authentication.

Therefore, one can argue that the discussed issues are in fact a subclass of a larger set of vulnerabilities. For this reason, a potential direction for developing countermeasures is to consider protection approaches that address CSRF in general.

7.1.3 Server-side detection of local reconnaissance attacks

The main problem in the context of the specified issues is that the attacked intranet servers have very limited means to protect themselves against such attacks. All they receive are HTTP requests from legitimate users, sometimes even in a valid authentication context. Therefore, at the server-side it is not always possible to distinguish between requests that were intended by the user and requests that were generated by a malicious JavaScript. In some cases evidence such as external referrers or mismatching host headers are available but this is not always the case. Furthermore, some of the described attacks will still work even when the server would be able to identify fraudulent requests.

For theses reasons, a reliable protection mechanism has also to involve the client-side. Only at the client-side all required context information concerning the single requests is available. Furthermore, to stop certain attacks, like the exploitation of unpatched vulnerabilities, it has to be prevented that the malicious request even reaches the targeted host.

7.2 Shortcomings of the same origin policy

The current state of JavaScript's SOP leaves a lot to be desired: On the one hand, the SOP is often regarded to be too strict for modern application's interoperability requirements. Creating web mashups or implementing web APIs requires the programmers to work around the SOP [52] which often

leads to insecure programming practises [8]. Flash's security policy [53] which allows partial cross-domain access demonstrates how cross-domain interaction can be allowed without reducing the overall security of the application: All permitted cross-domain interactions are configured in a `crossdomain.xml` policy file which can be obtained from the application's server.

On the other hand, the SOP is also not fine-grained enough. Based on the policy, there are only two possible decisions: Either "no access at all" or "unlimited access". A JavaScript is almighty in respect to documents that share the same origin. For this reason every single XSS issue compromises the complete vulnerable application. Measures to effectively limit the capabilities of malicious scripts are cumbersome and complicated [38]. Furthermore, as discussed in Sect. 2.3, the document-level nature of the SOP is responsible for the BRA to function.

More generally, it appears as if the design decision to derive security properties exclusively from the site's domain name is questionable. The DNS mapping of the domain name to the web server's IP address is not within the power of the web server. Consequently, the application's security depends on an outside entity which cannot be controlled by the application itself. This fact leads to the discussed DNS rebinding vulnerabilities. In turn DNS pinning, which was introduced to counter rebinding attacks, introduces problems with dynamic DNS services and DNS based redundancy solutions. Furthermore, DNS pinning is unable to protect against multi-session attacks as they have been described by Soref [67] and Rios and McFeters [61].

7.3 General advice

Before discussing potential countermeasures in Sect. 7.4 we give a brief list of general protection advices that can be applied without introducing specific protection technologies:

Server-side: As we have shown, victimized web browsers provide the adversary with powerful capabilities in respect to targets in the intranet which are comparable to situations in which the attacker can connect directly to the potential targets. Therefore, the security of intranet servers should be treated carefully:

- **Do not use the firewall for authentication:** All sensitive HTTP services in the intranet should employ explicit authentication mechanisms on their own.
- **Harden intranet services and hosts:** Treat every potential attack target in the intranet as if it was exposed to the public Internet. Apply all available security patches and updates. Do not run applications that are not longer maintained. Evaluate the necessity of every service to reduce the potential attack surface.
- **Change all default passwords on appliances and applications:** As shown above, even web interfaces that are only accessible from within the local LAN are not save against fraudulent requests. For this reason only non-default passwords are able to provide sufficient protection.
- **Eliminate potential information leaks:** Check all network services and especially DNS servers if these leak information about the internal IP-range to the outside.

Client-side: Besides the countermeasures that are discussed in Sect. 7.4, client-side measures comes down to reducing the amount of active client-side technologies the adversary might employ in his attack:

- **JavaScript:** Client-side scripting should only be enabled for trusted pages which really require JavaScript to function. This does not provide protection for the case that one of these pages were victim of an XSS attack, but it reduces the attack surface significantly.
- **Java:** Without Java, a potential attacker is unable to obtain the browser's local IP address. This raises the bar for a successful reconnaissance attack significantly. Furthermore, Java is a technology that provides TCP and UDP sockets within a DNS rebinding attack.
- **Flash:** As shown in Sect. 5.4 Flash also provides the attack with socket functionality in DNS rebinding attacks. Furthermore, Jackson et al. [34] demonstrated that it is easily possible to place malicious Flash applets onto web advertizement networks.

In addition, all other, more obscure client-side scripting languages and active technologies (such as VBScript or ActiveX) should be disabled as well.

7.4 Proposed countermeasures

Several countermeasures have been proposed to address selected attack-types. To a certain, individual degree these proposed methods are able to protect against their targeted threat. Please refer to the cited publications for a detailed account on the respective limitations. However, even a combination of all techniques still does not provide complete protection. In this section we list all relevant proposals to document the state of the art in protection and to collect hints in which direction future work should be aimed.

7.4.1 General countermeasures

As detailed above, turning of active client-side technologies deprives the attacker of many of his capabilities. Unfortunately, only few modern websites function without JavaScript.

For this reason, it is rarely feasible to disable JavaScript completely. The NoScript Firefox extension [33] permits a per site configuration whether JavaScript and other client-side code should be enabled. NoScript's default policy is to disable all active content on unknown domains. However, the extension does not defend against attacks that are executed through XSS attacks on trusted sites. If a site, for which script execution is allowed, has a XSS vulnerability, the adversary can use this site to execute his malicious code in the browser. Therefore, due to the high number of existing XSS flaws [9], even with NoScript the attack surface remains rather large.

A more general protection approach is described by Hallaraker and Vigna [25]. Their paper shows how to modify the JavaScript-engine of a web browser to allow behaviour based analysis of JavaScript execution. Using this newly introduced capability, they apply intrusion detection mechanisms to, e.g., prevent denial of service or XSS attacks. While the paper does not address the threats that are subject of our work, it may be possible to extend their work towards detecting and preventing JavaScript Malware. To verify this assumption further research work is necessary.

7.4.2 Countering privacy attacks

Jackson et al. [35] propose measures to defend against CSS based cache-disclosure attacks (see Sect. 3.1). Their technique extends the Same Origin Policy to also apply to cache and history information. This has the effect, that a JavaScript can only obtain cache and history information about elements that have the same origin as the script itself. The authors implemented their concepts as "SafeHistory" [36], an extension to the Firefox browser.

Jakobsson and Stamm [37] discuss a server-side technique to protect against CSS-based browser history disclosure. Their method prevents the attacker from guessing possible URLs that may be contained in the user's history by adding semi-random parts to the URLs, thus effectively creating URL pseudonyms. For this purpose, they introduced a server-side web-proxy to apply the randomized components to the application's URL transparently.

To defend against *cross-site timing attacks* (see Sect. 3.2), Bortz et al. [4] proposed a server-side module to ensure that the web server always takes a constant amount of time to process a request. For this reason they implemented "mod_timepad", an Apache module, which guaranties that every HTTP chunk is sent a time since the request was received which is a multiple of n milliseconds, with n being a user-adjustable parameter.

7.4.3 Countering intranet and DNS rebinding attacks

To protect the intranet against script-based reconnaissance and exploitation (see Sect. 4), we proposed in [42] a segmentation of network addresses. Every possible IP address is classified to be either *local* (i.e., part of the intranet) or *remote*. When the rendering of a web page or the execution of client-side script causes additional HTTP requests, these requests are intercepted. If the origin of the request is a page which was retrieved from a remote address and the respective target lies within the local LAN, the request is prohibited. Additionally, our approach provides partial protection against DNS rebinding attacks by monitoring the *local*/*remote* status of domain-name resolution. If the status of a domain switches from *remote* to *local*, this is a clear indicator of a rebinding attack that targets an intranet resource. If such a rebinding attempt was identified, all further HTTP request or other cross-document interaction originating from the offending domain are denied. We implemented and evaluated our solution in the form of a Firefox extension [73].

To defend against DNS rebinding attacks (see Sect. 5), Karlof et al. propose in [46] a same-origin policy that is based on public-key cryptography. Instead of identifying the origin of a given web-object by the URL that was used to request it, the element's "origin" is defined by the public key that is associated with the element (e.g., the key that was employed for an SSL-connection which delivered the element). Consequently, in the proposed model, the browser would allow a web object to access another web object only if their public keys match.

Furthermore, Jackson et al. [34] propose several countermeasures against DNS rebinding, ranging from smarter DNS pinning strategies, over utilizing reverse DNS lookups, up to implementing a firewall solution that prohibits the resolution of external domain names to internal IP addresses. See [34] for details.

8 Conclusion

8.1 Summary

In this paper we discussed and exemplified the capabilities of a malicious JavaScript in respect to the victimized browser's local context. More precisely, we have shown that such a script can execute various attacks to undermine the user's privacy (see Sect. 3), explore the browser's LAN (see Sect. 4 and 5), and stealthily attack third party applications using the victim's IP address (see Sect. 6).

Unfortunately, there is no straight forward solution to resolve the presented issues. As discussed in Sect. 7.1.1, the atomic actions taken by the offending script are permitted by the fundamental specification documents that define the WWW. Most technical foundations that are employed by the JavaScript-based attacks have already existed for several years. However, many of the attack vectors have just recently been disclosed in late 2006 and the first half of 2007. For

this reason, it stands to argue that there still exist further undisclosed attack methods.

In addition one can observe a rather uncontrolled growth of third party browser add-ons such as Flash [11] or Silverlight [58] that either support interfaces to JavaScript or provide active client-side scripting capabilities on their own. The power of these technologies often equals or exceeds JavaScript's capabilities. Therefore, concentrating on JavaScript/HTML/HTTP will not suffice to counter the exemplified threats. Furthermore, the ongoing evolution of these components adds to the probability for new attack vectors.

8.2 Outlook and future work

During our work on [42] and this paper, we identified several possible directions for future research:

- **Unified client-side security policy and reference monitor:** Future browsers should investigate and implement a central reference monitor which has to be used by all active client-side technologies. Such a monitor should assess and approve every security-relevant action (like network connections) and, thus, enforce a unified security policy. Especially in the case of DNS rebinding attacks, the lack of such a common means for browser based security mechanisms increases the problem's severity. If all client-side technologies (JavaScript, Java, Flash) would utilize the same DNS pinning table, many of the exemplified issues would not exist.
- **Combined client/server approach:** As reasoned in Sect. 7.1.3 most attacks are not deterministically detectable solely on the server-side. Additionally, approaching the problem solely on the client-side is also infeasible (see Sect. 7.1.1). Future research should investigate strategies towards the security properties of cross-context interactions that involve both the server and the browser.
- **Overhauling the same origin policy:** As reasoned in Sect. 7.2, the SOP has several severe shortcomings. In addition to the above proposed combined client/server approach towards the regulations of cross-context interactions, the SOP should be modified to match the new concepts.

Trying to solve the problems in a way that is backwards compatible with the current technologies will add further obstacles.

References

1. Alcorn, W.: Inter-protocol communication. Whitepaper, http://www.ngssoftware.com/research/papers/InterProtocolCommunication.pdf (11/13/06), August 2006
2. Alcorn, W.: Inter-protocol exploitation. Whitepaper, NGSSoftware Insight Security Research (NISR), http://www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf, March 2007
3. Alshanetsky, I.: Network scanning with http without javascript. [online], http://ilia.ws/archives/145-Network-Scanning-with-HTTP-without-JavaScript.html (09/11/07), November 2006
4. Bortz, A., Boneh, D., Nandy, P.: Exposing private information by timing web applications. In: WWW 2007, 2007
5. Burns, J.: Cross site reference forgery—an introduction to a common web application weakness. Whitepaper, http://www.isecpartners.com/documents/XSRF_Paper.pdf, 2005
6. Byrne, D.: Anti-dns pinning and java applets. Posting to the Bugtraq mailing list, http://seclists.org/fulldisclosure/2007/Jul/0159.html, July 2007
7. Mozilla Developer Center.: Liveconnect. [online], http://developer.mozilla.org/en/docs/LiveConnect (08/08/07), 2007
8. Chess, B., O'Neil, Y.T., West, J.: Javascript hijacking. [whitepaper], Fortify Software, http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf, March 2007
9. Christey, S., Martin, R.A.: Vulnerability type distributions in cve, version 1.1. [online], http://cwe.mitre.org/documents/vuln-trends/index.html (09/11/07), May 2007
10. Clover, A.: Css visited pages disclosure. Posting to the Bugtraq mailing list, http://seclists.org/bugtraq/2002/Feb/0271.html, February 2002
11. Adobe Coperation. Adobe flash. [online] http://www.adobe.com/products/flash/flashpro/
12. Duong, T.N.: Zombilizing the browser via flash player 9. talk at the VNSecurity 2007 conference, http://vnhacker.blogspot.com/2007/08/zombilizing-web-browsers-via-flash.html, August 2007
13. Endler, D.: The evolution of cross-site scripting attacks. Whitepaper, iDefense Inc., http://www.cgisecurity.com/lib/XSS.pdf, May 2002
14. Esser, S.: Bruteforcing http auth in firefox with javascript. [online], http://blog.php-security.org/archives/56-Bruteforcing-HTTP-Auth-in-Firefox-with-JavaScript.html (08/31/07), December 2006
15. Esser, S.: Javascript/html portscanning and http auth. [online], http://blog.php-security.org/archives/54-JavaScriptHTML-Portscanning-and-HTTP-Auth.html (08/27/07), November 2006
16. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS '02), 2000
17. Glass, E.: The ntlm authentication protocol. [online], http://davenport.sourceforge.net/ntlm.html (03/13/06), 2003
18. AVM Gmbh. Fritz! box. [online], product website, http://www.avm.de/en/Produkte/FRITZBox/index.html (09/06/07)
19. Google. Google translate. [online service], http://www.google.com/translate_t (09/11/07)
20. Grossman, J.: I know if you're logged-in, anywhere. [online], http://jeremiahgrossman.blogspot.com/2006/12/i-know-if-youre-logged-in-anywhere.html (08/08/07), December 2006
21. Grossman, J.: I know where you've been. [online], http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html, August 2006
22. Grossman, J.: Javascript malware, port scanning, and beyond. Posting to the websecurity mailing list, http://www.webappsec.org/lists/websecurity/archive/2006-07/msg00097.html, July 2006
23. Grossman, J., Hansen, R., Petkov, P., Rager, A.: Cross Site Scripting Attacks: Xss Exploits and Defense. Syngress, 2007
24. Grossman, J., Niedzialkowski, T.C.: Hacking intranet websites from the outside. Talk at Black Hat USA 2006, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf, August 2006

25. Hallaraker, O., Vigna, G.: Detecting malicious javascript code in mozilla. In: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 85–94, June 2005
26. Hansen, R.: Detecting firefox extentions. [online], http://ha.ckers.org/blog/20060823/detecting-firefox-extentions/ (08/08/07), August 2006
27. Hansen, R.: Detecting states of authentication with protected images. [online], http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/ (08/31/07), November 2006
28. Hansen, R.: Hacking intranets via brute force. [online], http://ha.ckers.org/blog/20061228/hacking-intranets-via-brute-force/, December 2006
29. Hansen, R.: List of common internal domain names. [online], http://ha.ckers.org/fierce/hosts.txt (09/06/07), March 2007
30. Hegaret, P.L., Whitmer, R., Wood, L.: Document object model (dom). W3C recommendation, http://www.w3.org/DOM/, January 2005
31. Hoffman, B.: Javascript malware for a gray goo tomorrow! Talk at the Shmoocon'07, http://www.spidynamics.com/spilabs/education/presentations/Javascript_malware.pdf, March 2007
32. Apple Inc.: Dynamic html and xml: The xmlhttprequest object. [online], http://developer.apple.com/internet/webcontent/xmlhttpreq.html (08/08/07), June 2005
33. InformAction.: Noscript firefox extension. Software, http://www.noscript.net/whats, 2006
34. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D.: Protecting browsers from dns rebinding attack. In: Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07), October 2007
35. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Protecting browser state from web privacy attacks. In: Proceedings of the 15th ACM World Wide Web Conference (WWW 2006), 2006
36. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Safehistory. software, http://www.safehistory.com/, 2006
37. Jakobsson, M., Stamm, S.: Invasive browser sniffing and countermeasures. In: Proceedings of the 15th Annual World Wide Web Conference (WWW2006), 2006
38. Johns, M.: Sessionsafe: implementing xss immune session handling. In: European Symposium on Research in Computer Security (ESORICS 2006), September 2006
39. Johns, M.: (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, August 2006
40. Johns, M., Kanatoko.: Using java in anti dns-pinning attacks (firefox and opera). [online], http://shampoo.antville.org/stories/1566124/ (08/27/07), February 2007
41. Johns, M., Winter, J.: Requestrodeo: client side protection against session riding. In: Frank Piessens, editor, OWASP Conference 2006, Report CW448, pp. 5–17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 2006
42. Johns, M., Winter, J.: Protecting the intranet against "javascript malware" and related attacks. In: Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2007), July 2007
43. Kaminsky, D.: Black ops 2007: Design reviewing the web. talk at the Black Hat 2007 conference, http://www.doxpara.com/?q=node/1149, August 2007
44. Kanatoko.: Stealing information using anti-dns pinning: Online demonstration. [online], http://www.jumperz.net/index.php?i=2&a=1&b=7 (30/01/07), 2006
45. 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
46. Karlof, C., Shankar, U., Tygar, J.D., Wagner, D.: 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
47. Kindermann, L.: My address java applet. [online], http://reglos.de/myaddress/MyAddress.html (11/08/06), 2003
48. Kishor.: Ie—guessing the names of the fixed drives on your computer. [online], http://wasjournal.blogspot.com/2007/07/ie-guessing-names-of-fixed-drives-on.html (08/31/07), July 2007
49. SPI Labs.: Detecting, analyzing, and exploiting intranet applications using javascript. Whitepaper, http://www.spidynamics.com/assets/documents/JSportscan.pdf, July 2006
50. SPI Labs.: Stealing search engine queries with javascript. Whitepaper, http://www.spidynamics.com/assets/documents/JS_SearchQueryTheft.pdf, 2006
51. Lam, V.T., Antonatos, S., Akritidis, P., Anagnostakis, K.G.: Puppetnets: misusing web browsers as a distributed attack infrastructure. In: Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS '06), pp. 221–234, 2006
52. Lamarre, J.: Ajax without xmlhttprequest, frame, iframe, java or flash. [online], http://zingzoom.com/ajax/ajax_with_image.php (02/02/2006), September 2005
53. Ludwig, A.: Macromedia flash player 8 security. Whitepaper, Macromedia, http://www.adobe.com/devnet/flashplayer/articles/flash_player_8_security.pdf, September 2005
54. McFeters, N., Rios, B.: Uri use and abuse. Whitepaper, http://www.xs-sniper.com/nmcfeters/URI_Use_and_Abuse.pdf, July 2007
55. Meer, H., Slaviero, M.: It's all about the timing... Whitepaper, http://www.sensepost.com/research/squeeza/dc-15-meer_and_slaviero-WP.pdf, August 2007
56. Megacz, A.: Firewall circumvention possible with all browsers. Posting to the Bugtraq mailing list, http://seclists.org/bugtraq/2002/Jul/0362.html, July 2002
57. Meschkat, S.: Json rpc—cross site scripting and client side web services. Talk at the 23C3 Congress, http://events.ccc.de/congress/2006/Fahrplan/attachments/1198-jsonrpcmesch.pdf, December 2006
58. Microsoft.: Microsoft silverlight. [online], http://www.microsoft.com/silverlight/ (09/14/07), 2007
59. Mueller, M.: Sun's response to the dns spoofing attack. [online], http://www.cs.princeton.edu/sip/news/sun-02-22-96.html (09/09/07), February 1996
60. Project, M.: Mozilla port blocking. [online], http://www.mozilla.org/projects/netlib/PortBanning.html (11/13/06), 2001
61. Rios, B.K., McFeters, N.: Slipping past the firewall. Talk at the HITBSecConf2007 conference, http://conference.hitb.org/hitbsecconf2007kl/agenda.htm, September 2007
62. Ruderman, J.: The same origin policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001
63. Samy.: Technical explanation of the myspace worm. [online], http://namb.la/popular/tech.html (01/10/06), October 2005
64. Schreiber, T.: Session riding—a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH, http://www.securenet.de/papers/Session_Riding.pdf, December 2004
65. Princeton University Secure Internet Programming Group. Dns attack scenario. [online], http://www.cs.princeton.edu/sip/news/dns-scenario.html, February 1996
66. Sethumadhavan, R.: Microsoft Internet explorer local file accesses vulnerability. Posting to the full disclosure mailing list, http://seclists.org/fulldisclosure/2007/Feb/0434.html, February 2007

67. Soref, J.: Dns: spoofing and pinning. [online], http://viper.haque.net/~timeless/blog/11/ (14/11/06), September 2003
68. Stamm, S., Ramzan, Z., Jakobsson, M.: Drive-by pharming. Technical Report 641, Indiana University Computer Science, December 2006
69. Stuttard, D.: Dns pinning and web proxies. NISR whitepaper, http://www.ngssoftware.com/research/papers/DnsPinningAndWebProxies.pdf, 2007
70. Topf, J.: The html form protocol attack. Whitepaper, http://www.remote.org/jochen/sec/hfpa/hfpa.pdf, August 2001
71. Vzloman, S., Hansen, R.: Enumerate windows users in js. [online], http://ha.ckers.org/blog/20070518/enumerate-windows-users-in-js/ (08/08/07), May 2007
72. Vzloman, S., Hansen, R.: Read firefox settings (poc). [online], http://ha.ckers.org/blog/20070516/read-firefox-settings-poc/ (08/08/07), May 2007
73. Winter, J., Johns, M.: Localrodeo: Client side protection against javascript malware. [online], http://databasement.net/labs/localrodeo (01/02/07), January 2007