Protecting the Intranet Against "JavaScript Malware" and Related Attacks*

Martin Johns and Justus Winter

Security in Distributed Systems (SVS)
University of Hamburg, Dept of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
{johns,4winter}@informatik.uni-hamburg.de

Abstract. The networking functionality of JavaScript is restricted by the Same Origin Policy (SOP). However, as the SOP applies on a document level, JavaScript still possesses certain functionality for cross domain communication. These capabilities can be employed by malicious JavaScript to gain access to intranet resources from the outside. In this paper we exemplify capabilities of such scripts. To protect intranet hosts against JavaScript based threats, we then propose three countermeasures: *Element Level SOP*, *rerouting of cross-site requests*, and *restricting the local network*. These approaches are discussed concerning their respective protection potential and disadvantages. Based on this analysis, the most promising approach, *restricting the local network*, is evaluated practically.

> *We're entering a time when XSS has become the new Buffer Overflow and JavaScript Malware is the new shellcode.*
>
> Jeremiah Grossman [6]

1 Introduction

Web browsers are installed on virtually every contemporary desktop computer and the evolution of active technologies like 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) vulnerabilities [3] the number of XSS worms [25] is increasing steadily. 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 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 work was supported by the German Ministry of Economics (BMWi) as part of the project "secologic", www.secologic.org.

B. M. Hämmerli and R. Sommer (Eds.): DIMVA 2007, LNCS 4579, pp. 40–59, 2007.

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 J. Grossman [6] in 2006 to describe this class of script code that stealthy uses the web browser as vehicle for attacks on the victim's intranet. In this paper we exemplify capabilities of such scripts and propose first defensive approaches.

1.1 Definitions

This paper focuses on web browser based attacks that target intranet resources. Therefore, we frequently have to differentiate between locations that are either within or outside the intranet. For this reason, in the remainder of this paper we will use the following naming conventions:

Local IP-addresses: The specifier *local* is used in respect to the boundaries of the intranet that a given web browser is part of. A local IP-address is therefore an address that is located inside the intranet. Such addresses are rarely accessible from the outside.

Local URL: If a URL references a resource that is hosted on a local IP-address, we refer to is as *local URL*.

The respective counterparts *external IP-address* and *external URL* are defined accordingly.

1.2 Transparent 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 [4] 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. Often such mechanism are summarized under the term "Single Sign On" (SSO).

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.

The firewall as a means of authentication. 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.

1.3 Cross Site Request Forgery

Cross Site Request Forgery (XSRF / CSRF) a.k.a. *Session Riding* is a client side attack on web applications that exploits implicit authentication mechanisms. 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 Figure 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 [26] or [2] for further details.

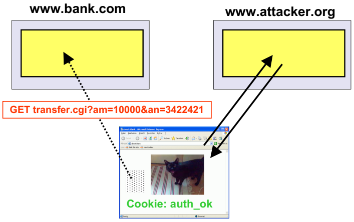


Fig. 1. A CSRF attack on an online banking site

2 Attacking the Intranet with JavaScript

2.1 Using a Webpage to Get Access to Restricted Web Resources

As described in Section 1 many companies allow their employees to access the WWW from within the company's network. 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 current browser scripting technologies possess certain network capabilities and as 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

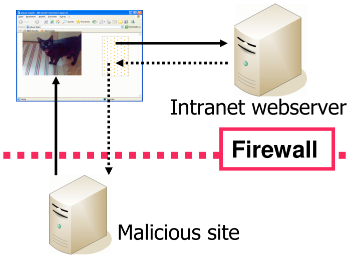


Fig. 2. Using a webpage to access restricted web servers

Section 1.2), any script that runs inside his browser is able to access restricted intranet resources with the same permissions as the employee would.

In the next Sections we examine the actual network capabilities and restrictions of existing active browser technologies and exemplify how these capabilities can be used to circumvent authentication schemes.

2.2 A Closer Look at JavaScript

For security reasons, the networking functions of client-side browser technologies are subject to major restrictions. We describe these restrictions only in respect to JavaScript, but similar concepts apply to e.g., Flash or Java applets.

Network capabilities: Foremost JavaScript is limited to HTTP communication only. Furthermore, a script is not allowed to communicate with arbitrary HTTP hosts. This is enforced by the *Same Origin Policy (SOP)*: The Same Origin Policy was introduced by Netscape Navigator 2.0 [24]. It defines and limits various rights of JavaScript. The origin of an element is defined by the protocol, the domain and the port that were used to access this element. The SOP is satisfied when the origins of two elements match. All explicit network functionality of JavaScript is restricted to communication with targets that satisfy the SOP. This effectively limits a script to direct communication with its origin host.

There is only one possibility for JavaScript to create HTTP requests to targets that do not satisfy the SOP: The script can dynamically include elements like images from foreign hosts into the document's DOM tree [9].

Access rights: Additionally, 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 has been obtained from the same origin as the JavaScript itself. 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.

A loophole in the SOP: As explained above, the cross-domain networking capabilities of JavaScript are restricted by the SOP. However, this policy allows dynamically including elements from cross domain HTTP hosts into the DOM tree by a JavaScript in 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, as this policy allows partial cross domain access. Depending on the type of the element that was included in the document, the JavaScript's capabilities to gain information by the inclusion differs. In the next sections we explain how this loophole can be exploited for malicious purposes.

2.3 Portscanning the Intranet

It was shown by various parties [19,21,7] how malicious web pages can use its capability to port-scan the local intranet. While the specific techniques vary, the general approach is always the same:

1. The script constructs a local 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 like `onload` and `onerror` 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.

To launch such an discovery attack, the malicious script needs to know the IP-range of the local intranet. In case this IP-range is unknown to the attacker, he can use a Java-Applet [17] to obtain the IP-address of the computer that currently executes the web browser which is vehicle of the attack. Using this address the attacker's script can approximate the intranet's IP-range.

Limitation: Some browsers like Firefox enforce a blacklist of forbidden ports [23] 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 Internet Explorer allow access to all ports.

2.4 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 [19,7].

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 eastereggs).

2.5 Attacking 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 [7].

Opening home networks: The following attack scenario mostly applies to home users. Numerous end-user devices like 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 off 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.

Cross protocol communication: Wade Alcorn showed in [1] 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.

2.6 Leaking Intranet Content by Breaking DNS-Pinning

The SOP should prevent cross domain access to content hosted on intranet web servers. In 1996 [27] 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, and therefore grants the script unlimited access to the intranet server.

To counter this attack 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. While in general this is an effective countermeasure against such an attack, unfortunately there are scenarios that still allow the attack to work: Josh Soref has shown in [28] how in a multi session attack a script that was retrieved from the browser’s cache still can execute this attack. Furthermore, we have recently shown [13] that current browsers are vulnerable to breaking DNS pinning by selectively refusing connections.

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 analyze the content in order to find further security problems.

Based on our findings, Kanatoko Anvil [16] demonstrated recently, that a successful anti DNS-pinning attack also effects some browser plugins, like the Flash player. As the Flash player’s scripting language ActionScript supports low level socket communication, such an attack extends the adversary’s capabilities towards binary protocols.

2.7 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 [5] that attacks similar to the vectors show in Sections 2.3 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, like `image`, `script` or `style`-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.

2.8 Analysis

In the most cases CSRF attacks (see Section 1.3) target authentication mechanisms that are executed by the web browser, e.g., by creating hidden HTTP requests that contain valid session cookies. The attacks covered in this paper are in fact CSRF attacks that target an authentication mechanism which is based on physical location: As discussed in Section 1.2, the firewall is used as a means of transparent implicit authentication which is subverted by the described 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 like 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.

Thus, a reliable protection mechanism has to be introduced at 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.

3 Defense Strategies

In this section we discuss four possible strategies to mitigate the threats described in Section 2. At first we assess to which degree already existing technology can be employed. Secondly we examine whether a refined version of the Same Origin

Policy could be applied to protect against malicious JavaScript. The third technique shows how general client-side CSRF protection mechanisms can be extended to guard intranet resources (a prior version of this approach was originally proposed by us in [14]). The final approach classifies network locations and deducts access rights on the network layer based on this classification. For every presented mechanism, we assess the anticipated protection and potential problems.

3.1 Turning Off Active Client-Side Technologies

An immediate solution to counter the described attacks is to turn of active client-side technologies in the web browser. To achieve the intended protection at least JavaScript, Flash and Java Applets should be disabled. As turning off JavaScript completely breaks the functionality of many modern websites, the usage of browser-tools that allow per-site control of JavaScript like the NoScript extension [10] is advisable.

Protection: This solution protects effectively against active content that is hosted on untrusted web sites. However, this approach does not protect against attacks, that do not rely on active technologies (see Section 3.1).

Furthermore, if an XSS weakness exists on a web page that is trusted by the user, he is still at risk. Compared to e.g. Buffer Overflows, XSS is a vulnerability class that is often regarded to be marginal. This is the case especially in respect to websites that do not provide serious services, as an XSS hole in such a site has only a limited attack surface in respect to causing "real world" damage. For this reason such web sites are frequently not audited thoroughly for XSS problems.

Any XSS hole can be employed to execute the attacks that are subject of this paper. This is the analogy between XSS and Buffer Overflows, that was mentioned in the introducing quote by J. Grossman: As a Buffer Overflow enables the attacker to run the shellcode in a trusted binary, an XSS vulnerability enables the attacker to run script code in the context of a trusted web application and therefore inside the victims browser.

Drawbacks: In addition to the limited protection, an adoption of this protection strategy will result in significant obstacles in the user's web browsing. The majority of modern websites require active client-side technologies to function properly. With the birth of the so-called "Web 2.0" phenomenon this trend even increases. The outlined solution would require a site-specific user-generated decision which client-side technologies should be permitted whenever a user visits a website for the first time. For this reason the user will be confronted with numerous and regularly occurring configuration dialogues. Furthermore, a website's requirements may change in the future. A site that does not employ JavaScript today, might include mandatory scripts in the future. In the described protection scenario such a change would only be noticeable due to the fact that the web application silently stopped working correctly. The task to determine the reason for this loss of functionality lies with the user.

3.2 Extending the SOP to Single Elements

As discussed in Section 2 a crucial part of the described attacks is the fact that the SOP applies on a document level. This allows a malicious JavaScript to explore the intranet by including elements with local URLs into documents that have an external origin. Therefore, a straight forward solution would be to close the identified loophole by extending the SOP to the granularity of single objects:

Definition 1 (Element Level SOP). *In respect to a given JavaScript an element satisfies the* Element Level SOP *if the following conditions are met:*

- *The element has been obtained from the same location as the JavaScript.*
- *The document containing the element has the same origin as the JavaScript.*

Only if these conditions are satisfied the JavaScript

- *is allowed to access the element directly and*
- *is permitted to receive events, that have been triggered by the element.*

Jackson et. al describe in [12] a similar approach. In their work they extend the SOP towards the browser's history and cache. By doing so, they are able to counter attacks that threaten the web user's privacy.

Protection: Applying the SOP on an element level would successfully counter attacks that aim to portscan the intranet or fingerprint internal HTTP-services (see Sections 2.3 and 2.4). These attacks rely on the fact that events like `onerror` that are triggered by the inclusion of local URLs can be received by attacker provided JavaScript. As the origin of this JavaScript and the included elements differs, the refined SOP would not be satisfied and therefore the malicious JavaScript would not be able to obtain any information from the inclusion attempt.

However, refined and targeted fingerprinting attacks may still be feasible. Even if elements of a different origin are not directly accessible any longer, side effects that may have been caused by these elements are. E.g., the inclusion of an image causes a certain shift in the absolute positions of adjacent elements, which in turn could be used to determine that the image was indeed loaded successfully. Furthermore, the attacks described in Section 2.5 would still be possible. Such an attack consists of creating a state-changing request to a well known URL, which would still be allowed by the refined policy. Also the content leaking attack described in Section 2.6 would not be prevented. The basis of the attack is tricking the browser to believe that the malicious script and the attacked intranet server share the same origin. Nonetheless, the feasibility of these still working attacks depends on detailed knowledge of the intranet's internal layout. As obtaining such knowledge is prevented successfully by the outlined countermeasure the protection can still be regarded as sufficient, provided the attacker has no other information leak at hand.

Drawbacks: The main disadvantage of this approach is its incompatibility to current practices of many websites. Modern websites provide so called *web APIs* that allow the inclusion of their services into other web applications. Such services are for example offered to enable the inclusion of external cartography material into webpages. Web APIs are frequently implemented using remote JavaScripts that are included in the targeted webpage by a `script`-tag. If a given browser starts to apply the SOP on an element level, such services will stop working.

A further obstacle in a potential adoption of this protection approach is the anticipated development costs, as an implementation would require profound changes in the internals of the web browser.

3.3 Rerouting Cross-Site Requests

As discussed in Section 2.8, the attacks shown in Section 2 are CSRF attacks which exploit the fact that the firewall is used as a means of transparent implicit authentication. In [14] we proposed *RequestRodeo* a client side countermeasure against CSRF attacks in general. This section presents a refined version of our original concept that is geared towards protecting companies' intranets against JavaScript Malware.

RequestRodeo's protection mechanism is based on a classification of outgoing http requests:

Definition 2 (entitled). *A given HTTP request is classified to be* entitled *if and only if:*

- *It was initiated because of the interaction with a web page and*
- *the URLs of the originating page and the requested page satisfy the SOP.*

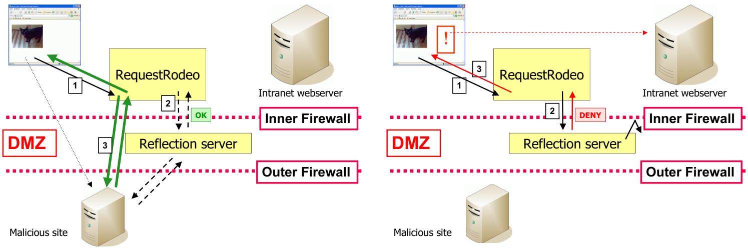
Only requests that were identified to be entitled *are permitted to carry implicit authentication information.*

According to this definition, all *unentitled* requests are "cross site requests" and therefore suspicious to be part of a CSRF attack and should be treated with caution. Cross-site request are fairly common and an integral part of the hyperlink-nature of the WWW. Therefore, a protection measure that requires the cancellation of such requests is not an option.

Instead we proposed to remove all authentication information from these requests to counter potential attacks. However, in the given case the requests do not carry any authentication information. They are implicitly authenticated as their origin is inside the boundaries that are defined by the firewall. For this reason other measures have to be taken to protect local servers. Our proposed solution introduces a *reflection service* that is positioned on the outer side of the firewall. All *unentitled* requests are routed through this server. If such a request succeeds, we can be sure that the target of the request is reachable from the outside. Such a target is therefore not specifically protected by the firewall and the request is therefore permissible.

The method that is used to do the actual classification is out of scope of this paper. In [14] we introduced a client side proxy mechanism for this purpose, though ultimately we believe such a classification should be done within the web browser.

Example: As depict in figure 3a a web browser requests a webpage from a server that is positioned outside the local intranet. In our scenario the request is



a. legal request b. prohibited request

Fig. 3. Usage of a reflection service

unentitled. It is therefore routed through the reflection service. As the reflection service can access the server unhindered, the browser is allowed to pose the request and receives the webpage's data. The delivered webpage contains a malicious script that tries to request a resource from an intranet web server (see figure 3b). As this is a cross domain request, it also is *unentitled* and therefore routed through the reflection service as well. The reflection service is not able to successfully request the resource, as the target of the request lies inside the intranet. The reflection service therefore returns a warning message which is displayed by the web browser.

Position of the service: It is generally undesirable to route internal web traffic unprotected through an outside entity. Therefore the reflection service should be positioned between the outer and an inner firewall. This way the reflection service is treated as it is not part of the intranet while still being protected by the outer firewall. Such configurations are usually used for DMZ (demilitarized zone) hosts.

Protection: The attack methods described in Section 2.3 to Section 2.5 rely on executing a JavaScript that was obtained from a domain which is under (at least partial) control of the attacker. In the course of the attack, the JavaScript creates HTTP requests that are targeted to local resources. As the domain-value for local resources differs from the domain-value of the website that contains the malicious script, all these requests are detected to be cross-site request. For this reason they are classified as *unentiteld*. Consequently, these request are routed through the reflection service and thus blocked by the firewall (see Figure 3).

Therefore, the usage of a reflection service protects effectively against malicious JavaScript that tries to either port-scan the intranet (see Section 2.3), fingerprint local servers (Section 2.4) or exploit unpatched vulnerabilities by sending state changing requests (Section 2.5).

The main problem with this approach is its incapability to protect against attacks that exploit the breaking of the web browser's DNS pinning feature (see Section 2.6). Such attacks are based on tricking the browser to access local resources using an attacker provided domain-name (e.g., `attacker.org`). Because of this attack method, all malicious requests exist within that domain and are therefore not recognised to be suspicious. Thus, these requests are not routed through the reflection service and can still execute the intended attack. As long as modern web browsers allow the breaking of DNS pinning, the protection provided by this approach is not complete. However, executing such an attack successfully requires detailed knowledge on the particular properties of the attacked intranet. As obtaining knowledge about the intranet is successfully prevented by the countermeasure, the feasibility of anti-DNS-pinning based attacks is questionable.

Drawbacks: Setting up such a protection mechanism is comparatively complex. Two dedicated components have to be introduced: The reflection service and an add-on to the web browser that is responsible for classification and routing of the HTTP requests. Furthermore, a suitable network location for the reflection service has to exist. As small-scale and home networks rarely contain a DMZ, the user either has the choice of creating one, which requires certain amounts of networking knowledge, or to position the reflection service outside the local network, which is objectionable.

The most appropriate deployment scenario for the proposed protection approach is as follows: Many companies already require their employees to use an outbound proxy for WWW-access. In such cases the classification engine, that is responsible for routing non-trusted request through the reflection service, could be included in the existing central proxy. This way all employees are transparently using the protection measure without additional configuration effort.

3.4 Restricting the Local Network

As introduced in Section 1 we refer to addresses that are located within the intranet as *local*. This notation implies a basic classification that divides network addresses into either *local* or *external* locations. If the web browser could determine to which group the origin and the target of a given request belong, it would be able to enforce a simple yet effective protection policy:

Definition 3 (restricted local network). *Hosts that are located inside a* restricted local network *are only accessible by requests that have a local origin. Therefore, inside such a network all HTTP requests with an external origin that target at a local resource are forbidden.*

With *requests with an external origin* we denote requests that were generated in the execution context of a webpage that was received from an external host. Unlike the proposed solution in Section 3.3 this classification does not take the domain-value of the request's origin or target into account. Only the actual IP-addresses are crucial for a policy-based decision.

Protection: All the attack methods specified in Section 2 depend on the capability of the malicious script to access local elements in the context of a webpage that is under the control of the attacker: The portscanning attack (Sec. 2.3) uses elements with local URLs to determine if a given host listens on the URL's port, the fingerprinting (Sec. 2.4) and local CSRF (Sec. 2.5) methods create local URLs based on prior application knowledge, breaking DNS-pinning (Sec. 2.6) tries to let the browser believe that an attacker owned domain is mapped to a local IP-address, and even the methods that do not rely on JavaScript (Sec. 3.1) require the usage local URLs to function. Therefore, the attacker's ability to successfully launch one of the specified attacks depends on his capability to create local HTTP requests from within a webpage under his control. By definition the attacker's host is located outside the intranet. Thus, the starting point of the attack is external. As the proposed countermeasure cancels all requests from an external origin to local resources, the attacker is unable to even bootstrap his attack.

Drawbacks: The configuration effort of the proposed solution grows linearly with the complexity of the intranet. Simple networks that span over a single subnet or exclusively use private IP-addresses can be entered fairly easy. However, fragmented networks, VPN setups, or mixes of public and private address ranges may require extensive configuration work.

Furthermore, another potential obstacle emerges when deploying this protection approach to mobile devices like laptops or PDAs. Depending on the current location of the device, the applicable configuration may differ. While a potential solution to this problem might be auto-configuration based on the device's current IP-address, overlapping IP-ranges of different intranets can lead to ambiguities, which then consequently may lead to holes in the protection.

3.5 Comparison of the Proposed Protection Approaches

As the individual protection features and disadvantages of the proposed approaches have already been discussed in the preceding sections, we concentrate in this section on aspects that concern either potential protection, mobility or anticipated configuration effort (see Table 1). The technique to *selectively turn of active technologies* (see Section 3.1) is left out of this discussion, due to the approach's inability to provide any protection in the case of an exploited XSS vulnerability.

Protection: The only approach that protects against all presented attack vectors is introducing a *restricted local network*, as this is the sole technique that counters effectively anti DNS-pinning attacks. However, unlike the other attack methods that rely on inherent specifics of HTTP/HTML, successfully attacking DNS-pinning has to be regarded as a flaw in the browser implementation. Therefore, we anticipate this problem to be fixed by the browser vendors eventually. If this problem is solved, the anticipated protection of the other approaches may also be regarded to be sufficient.

Configuration effort & mobility: The *element level SOP* approach has the clear advantage not to require any location-depended configuration. Therefore, the mobility of a device protected by this measure is uninhibited. But as some sites' functionality depends on external scripts, adopters of this approach instead would have to maintain a whitelist of sites, for which document level access to cross-domain content is permitted. As the technique to *reroute cross-site requests* requires a dedicated reflection service, the provided protection exists only in networks that are outfitted accordingly, thus hindering the mobility of this approach significantly. Also a *restricted local network* depends on location specific configuration, resulting in comparable restrictions. Furthermore, as discussed above, a *restricted local network* might lead to extensive configuration overhead.

Conclusion: As long as breaking DNS-pinning is still possible with current browsers, an evaluation ends in favor of the *restricted local network* approach. As soon as this browser flaw has been removed, *rerouting cross-site request* appears to be a viable alternative, especially in the context of large-sized companies with non-trivial network set-ups. Before an *element level SOP* based solution is deployed on a large scale, the approach has to be examined further for the potential existence of covert channel (see Section 3.2).

Table 1. Comparison of the proposed protection approaches

	No JavaScr.	Element SOP	Rerout. CSR	Restr. network
Prohibiting Exploring the Intranet	(+)*	(+)*	+	+
Prohibiting Fingerprinting Servers	+	+	+	+
Prohibiting IP-based CSRF	-	-	+	+
Resisting Anti-DNS Pinning	+	-	-	+
Mobile Clients	+	+	-	-
No Manual Configuration	-**	+	-	-

+: supported, -: not supported, *: Protection limited to JS based attacks, **: Per site configuration.

4 Evaluation

4.1 Implementation

Based on the discussion above, we chose to implement a software to enforce a *restricted local network*, in order to evaluate feasibility and potential practical problems of this approach [30].

We implemented the approach in form of an extension to the Firefox web browser. While being mostly used for GUI enhancements and additional functions, the Firefox extension mechanism in fact provides a powerful framework to alter almost every aspect of the web browser. In our case, the extension's functionality is based mainly on an XPCOM component which instantiates a nsIContentPolicy [22]. The nsIContentPolicy interface defines a mechanism that was originally introduced to allow the development of surf-restriction plug-ins, like parental control systems. It is therefore well suited for our purpose.

By default our extension considers the localhost (127.0.0.1), the private address-ranges (10.0.0.0/8, 192.168.0.0/16 and 172.16.0.0/12) and the link-local subnet (169.254.0.0/16) to be *local*. Additionally the extension can be configured manually to include or exclude further subnets in the local-class.

Every outgoing HTTP request is intercepted by the extension. Before passing the request to the network stack, the extension matches the IP-addresses of the request's origin and target against the specifications of the address-ranges that are included in the local-class. If a given request has an external origin and a local target it is dropped by the extension.

By creating a browser extension, we hope to encourage a wider usage of the protection approach. This way every already installed Firefox browser can be outfitted with the extension retroactively. Furthermore, in general a browser extension consists of only a small number of small or medium sized files. Thus, an external audit of the software, as it is often required by companies' security policies, is feasible.

An alternative to implementing a browser extension would have been to realize the outlined protection mechanism in the form of a client-side web proxy. A proxy has the advantage of not being restricted to one single browser brand. Furthermore, such a proxy could be installed company wide at a central location, thus minimizing configuration and maintenance effort. Unfortunately establishing the origin for a given HTTP request is a non-trivial task outside the web browser. Achieving this within a proxy requires substantial alteration of incoming HTML content (see [14]), which is an error prone exercise, due to dynamic content creation by JavaScript.

4.2 Practical Evaluation

Our testing environment consisted of a PC running Ubuntu Linux version 6.04 which was located inside a firewalled subnet employing the 192.168.1.0/24 private IP-address range. Our testing machine ran an internal Apache webserver listening on port 80 of the internal interface 127.0.0.1. Furthermore, in the same subnet an additional host existed running a default installation of the Apache webserver also listening on port 80. The web browser that was used to execute the tests was a Mozilla Firefox version 2.0.0.1. with our extension installed. The extension itself was configured using the default options.

Besides internal testing scripts, we employed public available tools for the practical evaluation of our implementation. To test the protection abilities against portscanning and fingerprinting attacks, we used the JavaScript portscanner from SPI Dynamics that is referenced in [19]. To evaluate the effectiveness against anti DNS-pinning attacks we executed the online demonstration provided by [15] which tries to execute an attack targeted at the address 127.0.0.1.

The results turned out as expected. The portscanning and fingerprinting attempts were prevented successfully, as the firewall rejected the probing requests of the reflection service. Also as expected, the anti DNS-pinning attack on the local web server was prevented successfully. Furthermore the extension was able to detect the attack, as it correctly observed the change of the adversary's domain (in this case `1170168987760.jumperz.net`) from being remote to local.

4.3 Limitations

During our tests we encountered a possible network setup that may yield problems with our approach. A company's web-services are usually served from within a DMZ using public IP-addresses. Unfortunately, the “local”/”external”-classification of hosts located in a DMZ is not a straight-forward task. As the hosts’ services are world-reachable the respective IPs should be classified as “external” to allow cross-domain interaction between these services and third party web applications. However, in many networks the firewall setup allows connections that origin from within the company’s network additional access rights to the servers positioned in the DMZ. For example internal IPs could be permitted to access the restricted FTP-port of the webserver to update the server’s content. Thus, in such setups a malicious JavaScript executed within the intranet also possesses these extended network capabilities.

5 Related Work

In this section we sum up related publications. As, to the best of our knowledge, no work has been published yet that directly deals with the threats to the intranet specified in this paper, we describe approaches that deal with related web application threats in general. We thereby focus on protection mechanisms that are positioned at the client side. If applicable we discuss if the described approaches can be extended to protect the intranet against JavaScript based attacks.

Lam et al. [20] discus the reconnaissance probing attack (see Section 2.3) as a tool to identify further victims in the context of web-server worm propagation. They propose several options for client-side defense mechanisms, like limiting the number of cross-domain requests. However, as they address the issues only in the context of large scale worm propagation and DDoS attacks, these measures do not promise to be effective against targeted intranet-attacks. The paper contains an excellent analysis of existing restrictions posed by different web browsers, like number of allowed simultaneous connections.

Vogt et al. [29] propose a combination of static analysis and dynamic data tainting to stop the effects of XSS attacks. The outlined approach does not identify or stop the actual injected script but instead aims to prohibit resulting leakage of sensitive information. To achieve this, their technique employs an enhanced JavaScript engine. The added features of this modified engine are twofold: For one, the flow of sensitive data, like cookie-values, through the script can be tracked dynamically. This way the mechanism detects and prevents the transmission of such data to the adversary. Furthermore, via static analysis, all control flow dependencies in scripts that handle sensitive information are established. This is done to identify indirect and hidden channels that could be abused for data leakage. If such channels are identified, their communication with external hosts is prevented.

In a related approach, Kirda et al. [18] describe Noxes, an application-level firewall that examines incoming HTML data in respect to potential sources for information leaks. Based on this analysis the firewall dynamically creates connection rules, to stop HTTP requests that are suspicious to transport confidential data, like cookie values. Noxes is concerned with the data content of outgoing requests and not with the target. For this reason the described algorithm is not applicable in the context of this paper. However, the protection approach, canceling suspicious HTTP requests, is closely related to our solution proposed in Section 3.4. A combination of both approaches to extend the respective range of protection is therefore possible.

Ismail et al. [11] describe a local proxy based solution towards protection against reflected XSS attacks. The proxy examines the GET and POST parameters of outgoing HTTP request for the existence of potential problematic characters like "<". If such characters are found in one of the parameters, the proxy also checks the respective HTTP response if the parameter is included verbatim and unencoded in the resulting webpage. If this is the case, the proxy concludes a potential XSS attack and encodes the offending characters itself.

A more general protection approach is described by Hallaraker and Vigna [8]. 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.

Finally, as already mentioned in Section 3.2, Jackson et al. [12] describe a solution to a related issue: Current browser technologies grant JavaScript certain capabilities to access information about the user's browsing history and cache content. These capabilities enable the adversary to create scripts that compromise the privacy of the user. In order to prevent such attacks, [12] 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. As browsing history and cache content information can provide hints about the existence and particularities of intranet servers without requiring the attacker to generate any network traffic, an adoption of the described countermeasures is advisable in addition to applying the here proposed mechanisms.

6 Conclusion and Future Work

We showed that carefully crafted script code embedded in webpages is capable to bypass the Same Origin Policy and thus can access intranet resources. For this reason simply relying on the firewall to protect intranet HTTP server against unauthorized access is not sufficient. As it is not always possible to counter such attacks at the server side, we introduced and discussed four distinct client-side countermeasures. Based on this discussion, we implemented a Firefox extension to enforce a *restricted local network*.

While our implementation reliably provides protection against the specified threats, this protection comes with a price, as additional configuration overhead and potential problems concerning mobile clients exist. Furthermore, our solution fixes a problem that occurs because of fundamental flaws in the underlying concepts - HTTP and the current JavaScript security model. Therefore future research in this area should specifically target these shortcomings to provide the basis for a future web browser generation that is not susceptible any longer to the attacks that have been discussed in this paper.

References

1. Alcorn, W.: Inter-protocol communication. Whitepaper (11/13/06) (August 2006) `http://www.ngssoftware.com/research/papers/InterProtocolCommunication.pdf`
2. Burns, J.: Cross site reference forgery - an introduction to a common web application weakness. Whitepaper (2005) `https://www.isecpartners.com/documents/XSRF_Paper.pdf`
3. Endler, D.: The evolution of cross-site scripting attacks. Whitepaper, iDefense Inc. (May 2002) `http://www.cgisecurity.com/lib/XSS.pdf`
4. Glass, E.: The ntlm authentication protocol. (03/13/06) (2003) [online] `http://davenport.sourceforge.net/ntlm.html`
5. Grossman, J.: Browser port scanning without javascript. (08/01/07) (November 2006) Website `http://jeremiahgrossman.blogspot.com/2006/11/browser-port-scanning-with out.html`
6. Grossman, J.: Javascript malware, port scanning, and beyond. Posting to the websecurity mailing list (July 2006) `http://www.webappsec.org/lists/websecurity/archive/2006-07/msg00097.html`
7. Grossman, J., Niedzialkowski, T.C: Hacking intranet websites from the outside. Talk at Black Hat USA 2006 (August 2006) `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf`
8. 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)
9. Le Hegaret, P., Whitmer, R., Wood, L.: Document object model (dom). W3C recommendation (January 2005) `http://www.w3.org/DOM/`
10. InformAction. Noscript firefox extension. Software (2006) `http://www.noscript.net/whats`
11. Ismail, O., Eto, M., Kadobayashi, Y., Yamaguchi, S.: A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In: 8th International Conference on Advanced Information Networking and Applications (AINA04), (March 2004)
12. 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)
13. Johns, M. (somewhat) breaking the same-origin policy by undermining dns-pinning. Posting to the Bug Traq Mailinglist (August 2006) `http://www.securityfocus.com/archive/107/443429/30/180/threaded`
14. Johns, M., Winter, J.: Requestrodeo: Client side protection against session riding. In: Piessens,F. (ed.) Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448, pp. 5 – 17. Departement Computerwetenschappen, Katholieke Universiteit Leuven (May 2006)
15. Kanatoko. Stealing information using anti-dns pinning (30/01/07) (2006) Online demonstration. webpage, `http://www.jumperz.net/index.php?i=2&a=1&b=7`

16. Kanatoko. Anti-dns pinning + socket in flash (19/01/07) (January 2007) Website `http://www.jumperz.net/index.php?i=2&a=3&b=3`
17. Kindermann, L.: My address java applet (11/08/06) (2003) Webpage `http://reglos.de/myaddress/MyAddress.html`
18. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: A client-side solution for mitigating cross site scripting attacks, security. In: Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006) (April 2006)
19. SPI Labs. Detecting, analyzing, and exploiting intranet applications using javascript. Whitepaper (July 2006) `http://www.spidynamics.com/assets/documents/JSportscan.pdf`
20. Lam, V.T., Antonatos, S., Akritidis, P., Anagnostakis, K.G.: Puppetnets: Misusing web browsers as a distributed attack infrastructure. In: ACM Conference on Computer and Communications Security (CCS'06), pp. 221–234 (2006)
21. Petkov, P.: Javascript port scanner (11/08/06), August (2006) Website `http://www.gnucitizen.org/projects/javascript-port-scanner/`
22. XUL Planet. nsicontentpolicy. API Reference (11/02/07) (2006) webpage `http://www. xpcomref/ifaces/nsIContentPolicy.html`
23. Mozilla Project. Mozilla port blocking (11/13/06) (2001) Webpage `http://www.mozilla.org/projects/netlib/PortBanning.html`
24. Ruderman, J.: The same origin policy (01/10/06) (August 2001) Webpage `http://www.mozilla.org/projects/security/components/same-origin.html`
25. Samy: Technical explanation of the myspace worm (01/10/06) (October 2005) website `http://namb.la/popular/tech.html`
26. Schreiber, T.: Session riding - a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH (December 2004) `http://www.securenet.de/papers/Session_Riding.pdf`
27. Princeton University Secure Internet Programming Group. Dns attack scenario (February 1996) Webpage `http://www.cs.princeton.edu/sip/news/dns-scenario.html`
28. Soref, J.: Dns: Spoofing and pinning (14/11/06) (September 2003) Webpage `http://viper.haque.net/~timeless/blog/11/`
29. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vig, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: 14th Annual Network and Distributed System Security Symposium (NDSS 2007) (2007)
30. Winter, J., Johns, M.: Localrodeo: Client side protection against javascript malware (01/02/07) (January 2007) webpage `http://databasement.net/labs/localrodeo`