Session-Cookies and SSL

Johannes Böck, www.hboeck.de

December 29, 2008

Study research project at the EISS (European Institute for system security),
University of Karlsruhe
http://iaks-www.ira.uka.de/eiss/

Hiermit erkläre ich, Johannes Böck, dass ich diese Studienarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Karlsruhe, December 29, 2008

Johannes Böck

Contents

1 Introduction

Web security is an important topic today. While websites were primarily invented to deliver content, today they are used for complex applications, often bound to very sensible information and actions (like bank account management).

In 2006, the NIST (National Institute for Standards and Technology in the USA) that delivers identifiers (the CVE numbers) for software vulnerabilities, provided a report with the title "Vulnerability Type Distributions in CVE" [1]. The number one vulnerability changed from Buffer Overflows to Cross Site Scripting (XSS) in 2005, number two were SQL injections. Both Cross Site Scripting and SQL injections are vulnerabilities in web applications.

In 2007 and 2008, the security of web application sessions over SSL connections received increasing attention. The interesting fact about it was that even sessions encrypted with SSL are vulnerable to session hijacking.

These vulnerabilities and attacks will be discussed in this work. In the chapter 2, the protocols needed for sessions in web applications will be discussed. Chapter 3 will describe network sniffing and simple attacks on unencrypted sessions. Chapter 4 will be the main part of this work and will describe possibilities of attacking SSL-secured sessions. Some examples from real-world applications will be given in 5. In chapter 6, I'll compare this attack to other problems based on sessions (CSRF and XSS) and will discuss the general problems of HTTP and web applications. In the appendix, some tools are listed that were used for this work.

2 Protocols

2.1 ISO / OSI layer model

The OSI layer model is a common abstract description for the interaction of network protocols. It has been in development since 1979 as part of the Open Systems Interconnection (OSI) initiative. It is an official standard by the ISO (International Organization for Standardization) and the ITU (International Telecommunication Union) since 1994 with the reference number ITU-T X.200 [2].

The OSI layer model defines seven layers of abstractions to design complex networking environments. The lowest layer is the plain transfer of bits through a physical medium (layer 1). The basic idea is that when working on a layer, one can expect that the lower layers just work as intended and the layer directly below can be used. It should also be possible to replace technology in one layer to fit new requirements without having to change technology in the other layers.

There are some relevant differences between the theory of the ISO / OSI model and what is used in the TCP/IP-based Internet (cf. table 1). The TCP protocol is on Layer 4 of the OSI model, the HTTP protocol on Layer 7. The OSI model knows a Session Layer, which is Layer 5. HTTP connections are stateless, so they know no sessions. To use sessions, one has to emulate them

Layer	OSI model	World Wide Web
1	Physical	Ethernet
2	Link	
3	Network	IP
4	Transport	TCP
5	Session	
6	Presentation	
7	Application	HTTP

Table 1: Comparison of the OSI layer model with the World Wide Web

with further technologies.

There are basically two methods, one based on HTTP authentication and the other based on POST data in combination with cookies. HTTP authentication has a number of limitations, for example it is impossible to end a session without exiting the browser, so it is is rarely used these days. The main focus of this work will be on the POST data / cookies method.

2.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) is a protocol on the transport layer (Layer 4, cf. 2.1) in the Internet [3]. Most application protocols used in the Internet are TCP-based, like POP3, SMTP, HTTP, XMPP.

To distinguish between different TCP-based protocols, every TCP service has a so-called port to identify different services running on the same host. All protocols have a default port assigned to, though it is still possible to run them on a different port. For the protocols discussed in this paper, the default ports are 80 (HTTP) and 443 (HTTPS).

2.3 Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (abbreviation HTTP) is a protocol on the Application Layer (Layer 7, cf. 2.1) and is the main protocol used in today's Internet communication.

The original HTTP protocol [4] was invented as a simple protocol for file transfer. It had no major interaction capabilities. POST-Variables were added 1996 in HTTP/1.0 [5], cookies in 1997 [6]. The first JavaScript [7] implementation was added 1995.

To understand the problems of web security today, one has to understand the history of HTTP. Its original purpose was to serve static content in the form of HTML-pages and images. The World Wide Web was originally not designed for interaction. Today, this is completely different. Pretty much every modern web page has some kind of interaction, like individual start pages for each visitor depending on his previous visits, his language or his used web software.

Even further, there is a step towards complex web applications. Web mailers are a common example. Today there are even efforts to bring classic desktop and office applications to the web.

This leads to many security issues that need nontrivial workarounds by the application designer. The most well-known problems are Cross Site Scripting (abbreviation XSS) and Cross Site Request Forgery (CSRF).

HTTP is a stateless protocol. Stateless means that every HTTP request is an independent action unrelated to any previous requests. The server does not remember anything from previous connections.

A typical HTTP interaction works like this: The client sends a one line request (usually through TCP-port 80) followed by a number of optional headers (listing 1). The request consists of the request method (GET is a simple request, POST is for sending data from the client to the server, HEAD is for debugging purposes, other methods exist for special applications), the path of the requested file (here /index.html) and the protocol version (here HTTP/1.1). It is finished by two newlines. The reply contains a line with the HTTP status code (200 for a common OK) followed by some optional header lines, also terminated by two newlines (listing 2). After that, the real content (depending on the kind of request and answer) follows.

Listing 1: Simple HTTP request

```
GET /index.html HTTP/1.1
Host: www.somehost.com
```

Listing 2: Simple HTTP reply

```
HTTP/1.1 200 OK
Server: Apache/2.2.10 (Unix) PHP/5.2.6
Content-Length: 65
Content-Language: en
Content-Type: text/html
Connection: close

<html><head><title>Test</title></head>
<body>Hello</body></html>
```

2.3.1 HTML Forms, GET and POST data

The Hypertext Markup Language (HTML) is the common format for publishing content in the World Wide Web. It has some elements for further interaction with the HTTP protocol.

HTML knows the `<form>` tag that can be used to create forms that transfer their data to an application on the server. A typical login form can be seen in listing 3.

Listing 3: HTML login form

```
<form action="send.php" method="POST">
```

```
Username: <input type="text" name="username" />
Password: <input type="password" name="password" />
</form>
```

This will present the user a simple login form with two fields for Username and Password. This data is afterwards transferred with a POST request to the file send.php on the server.

The difference between GET and POST variables is that GET-variables are transferred inside the URL (like `http://www.example.com/?variable=value`), while POST data is transferred through the HTTP request body. This sometimes leads to the false assumption that POST variables provide better security, for example against Cross Site Request Forgery. This is usually not the case, as all attack vectors relying on GET variables can somehow be transferred (mostly through JavaScript) to work with POST variables as well (cf. [8]).

2.3.2 Cookies

In 1997, RFC 2109 introduced the HTTP State Management Mechanism [6], known as cookies. Cookies give applications running on a web server the possibility to store data on the client's web browser.

A cookie consists of a simple key/value pair. The server sends Cookies inside the HTTP header to the client. If cookies are set, the client will send them back with any connection it makes to the same host. It is also possible to read and set cookies with JavaScript.

Cookies follow the "same origin policy", that means one cookie belongs to a single hostname. It is crucial for the security of cookies that they can not be accessed by any foreign application or adversary.

Cookies can also be restricted on a specific path on the web server, for example a cookie defined for the path /admin/ will be sent on a request `http://samplesite.com/admin/`, but not for `http://samplesite.com/user/`.

Cookies are sent before any other HTML-data is transferred. This is crucial for parts of the attack described later in this document.

Listing 4: HTTP reply header with cookie

```
HTTP/1.1 200 OK
Server: Apache/2.2.10 (Unix) PHP/5.2.6
Content-Length: 65
Content-Language: en
Content-Type: text/html
Set-Cookie: Cookie1=test; expires=Fri, 17-Dec-2010
   13:02:14 GMT; path=/;
Connection: close
```

2.3.3 Cookies for sessions

Most web applications use cookies to handle sessions.

After a user enters some login data into a form (cf. listing 3), the application on the server checks if that login data is valid. The application then creates a session (it needs some internal storage for this) for that user with information about the user's rights and data. This session gets an ID assigned (called the session ID), which is a random value. It is important that this value is not predictable by an adversary. There have been attacks against web applications in the past based on weak random number generators [9].

The application sends the session ID back to the user as a cookie. This may look like `SESSID=a38e3b0e51932347294581c5dbb70d80`.

With every request, the user sends the session ID back to the server. We can see that the whole security of the session relies on the fact that the session ID is only known by the server application and the user's web browser. If an adversary learns the session ID, he can add it to his browser and hijack the session of the victim.

2.4 SSL / TLS and HTTPS

Plain protocols on the application layer usually do not use any encryption. SSL is a protocol to provide encrypted communication for existing plain-text protocols. It can be added between the transport and the application layer for every TCP-based protocol. The SSL-enabled protocol is usually marked with an ending S, so POP3 becomes POP3S, SMTP becomes SMTPS and HTTP becomes HTTPS. SSL-enabled protocols have other ports defined. HTTP usually runs on port 80, HTTPS on port 443.

TLS is the successor of SSL. For this work, the difference is irrelevant, so when writing SSL, it means both SSL / TLS. The sometimes used STARTTLS-method (which is on a different layer) is never used for HTTP in real-world-scenarios.

The current version of SSL / TLS is defined in RFC 4346 [10].

If we have some application where security is crucial (bank applications, web shops), we need SSL to ensure that sniffing (cf. 3.1) of data and session IDs is not possible.

2.4.1 SSL connections on a cryptographic level

SSL connections usually start with a handshake where one or both partners authenticate with a public key certificate.

After that, a session key is generated. There are two ways for that, depending on the SSL algorithm used. In one case, the key is just generated by one party and sent encrypted to the other. Preferable are SSL-methods that do a signed Diffie-Hellman key exchange, because this provides Perfect Forward Secrecy (PFS, cf. F.1.1.2 in [10]).

2.4.2 Certificates

SSL is based on certificates defined in the X.509 standard [11].

SSL-connections can only be negotiated if at least the server part has a certificate. Common web browsers check the validity of the server certificate against a list of trusted authorities. A party running a website can buy certificates from such a trusted authority.

If the website does not have a certificate by such an authority, the browser throws a warning. In recent versions of the webbrowser Firefox, it became much more difficult for the user to connect to a website with an unknown certificate issuer (so-called self-signed certificate).

This concept is often criticised, because it can force website operators to use less security (no SSL at all) if they can't afford to buy a certificate. An alternative would be a CA that issues certificates for free and checks the identity of the certificate owner by volunteers. A project for such an authority is CAcert.org.

3 Attacks on Sessions

3.1 Sniffing

Sniffing in a security context means that an adversary reads and analyzes the network traffic of a victim. In their original form, the common Internet protocols all consist of plain text that is transferred unencrypted through the Internet. SSL is often used to prevent this.

Again this is a reminder of the history of the Internet: It was never designed to provide secret communication, SSL was added later on to secure protocols that were (and often still are) just plain text protocols.

Sniffing tools just read the network traffic. In ethernet setups with a hub, all network traffic is transferred to all computers. It is the job of the client to know if the traffic is meant for him or for some other client in the same network. So a sniffing tool just has to read all traffic.

In networks with a switch (which is the modern replacement for a hub), this has changed. Sniffing in switched networks is a bit more complex, but still possible. The decision of the switch where network traffic is routed is based on the MAC address. The MAC address can easily be faked. Sniffing attacks on switched networks are also called ARP spoofing [12].

In wireless networks, sniffing is more like in hub networks. Every member of a network can read all network traffic.

Obviously, it does not matter if the adversary is in the client's or server's network.

Another possibility for sniffing is on some of the routing machines. An adversary who has access to a router at an Internet service provider between the client and the server (either through physical access or through a hacked machine) can also read network traffic. The same is true if the adversary is within some network of one of the routing machines.

An unlikely case could also be that the adversary is in some way able to manipulate an IP routing protocol at some point to force a routing machine to forward traffic to him.

3.2 Session hijacking

As the security of HTTP-sessions completely relies on the cookie to be secret, an adversary who can read the victim's unencrypted network traffic is easily able to take over the session. A simple approach would be just reading the cookie in plain text with some network sniffing program like Wireshark (cf. A.3) and manually add the cookie to the web browser.

Some session frameworks also check that one session can only be used by one IP. This makes the attack sometimes less feasible, though an adversary who is able to read network traffic is often also able to fake the victim's IP or already has the same (for example in a NAT network where several clients share the same public IP).

3.3 Forwarding traffic to SSL

For web applications that require security it is advisable to forward all HTTP access to HTTPS on the server side. For the Apache web server, a configuration could look like listing 5. What happens is that connections on plain HTTP (port 80) all get not a normal HTTP reply with status code 200, but a reply with status code 403, which stands for "Redirect Permanently".

Listing 5: Apache configuration forwarding to HTTPS

```
<VirtualHost *:80>
 RedirectPermanent / https://servername.com/
 ServerName servername.com
</VirtualHost>
<VirtualHost *:443>
 SSLEngine on
 SSLCertificateFile /etc/apache2/ssl/server.crt
 SSLCertificateKeyFile /etc/apache2/ssl/server.key
 ServerName servername.com
 DocumentRoot "/var/www/servername.com/htdocs"
</VirtualHost>
```

For some applications, it is more feasible to only forward specific URLs. An example would be a weblog system. Normal visitors that only read do not need any secure access, but the login for authors should be encrypted. Apache can achieve this through the mod_rewrite module (cf. .htaccess example in listing 6, it forwards all access to URLs inside the /admin directory to HTTPS).

Listing 6: .htaccess to forward subdir to HTTPS

```
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/admin.*
RewriteCond %{HTTPS} off
RewriteRule (.*) https://%{HTTP_HOST}/$1 [R=302]
```

4 An attack on SSL-secured sessions

One may think at a first glance that forwarding all HTTP-traffic to HTTPS will avoid session sniffing. This is not the case.

Assuming a user has an open session on some web application. The cookie is already part of the HTTP request, so a single HTTP request to the same host will transfer the cookie unencrypted over the network, even if it is only answered by some HTTP forward.

There are various ways of social engineering that allow to get a user to open an HTTP connection. A simple approach would be giving him some link that contains an image or `<iframe>` tag that refers to some HTTP URL on the host to be attacked.

An `<iframe>` tag can be used to embed a web page as a box inside another web page. Though one can send the victim a harmless looking URL on another host, which will trigger the connection intended by the adversary.

The cookie specification in RFC 2109 knows a flag "Secure", which can be appended to any cookie. The attribute means that the cookie shall only be transferred through a secure connection. The RFC does not define what a secure connection is. On all modern browsers this is interpreted that the cookie shall only be transferred through HTTPS connections. Thus this prevents unencrypted cookie transfer. So setting this flag can stop the described attack.

4.1 Publications about SSL-Session hijacking

In September 2007, the US Cert published an advisory on a couple of popular web sites describing the vulnerability above [13]. The advisory did not get much attention.

This changed 2008 with two publications. Sandro Gauci [14] presented SurfJack, an easy to use tool to hijack sessions. Nearly at the same time, Mike Perry announced a talk about the subject on the DEFCON 16.

4.2 Disabling HTTP will not help

One solution one would think of to prevent such attacks would be just running HTTPS on one IP. Thus, if there is for example `https://securesite.com/`, a connection to `http://securesite.com/` would already stop on the TCP layer. Thus the HTTP headers (and with it the cookie) will not be sent.

This approach does not help. An adversary could then just force the victim to open an URL like `http://securesite.com:443/`. This would open an HTTP connection on the HTTPS port, which will obviously only lead to some error. Though the adversary has reached his goal, as the first part of the HTTP connection is transferred unencrypted and the cookie can be sniffed.

4.3 HTTP basic access authentication (HTTP auth)

Beside the method with cookies, HTTP also knows its own method for authentication and sessions. It is described in chapter 11 of RFC 1945 [5] and often just called HTTP auth.

Unlike cookies (which have the "same origin" policy), the HTTP auth method is bound to URLs. An authenticated session for `https://somesite.com/app/` will apply to all URLs below that (e. g. `https://somesite.com/app/test.php`), but not on e. g. `https://somesite.com/test/`. It also does not apply on a protocol change, thus a session on an HTTPS URL will never be transferred through an HTTP request.

Therefore the above attack scenario is most probably no threat to direct HTTP authentication. It may be worth deeper research if a similar attack is possible on HTTP authentication.

4.4 Attack step by step

To illustrate better how the attack works this section gives a step-by-step example for session hijacking. The example is based on the content management system Drupal, which is still vulnerable (cf. 5.2) to this kind of attack.

Assume we have user Alice (the victim) and user Eve (the adversary). Both are on the same network. To replicate the attack, this can also be done on the same client with different browsers, different users running the browser or virtualization environments - the attack works exactly the same way here.

Eve will use the sniffing tool Wireshark (cf. A.3) and the Add N Edit Cookies (AnEC) extension for Firefox (cf. A.2).

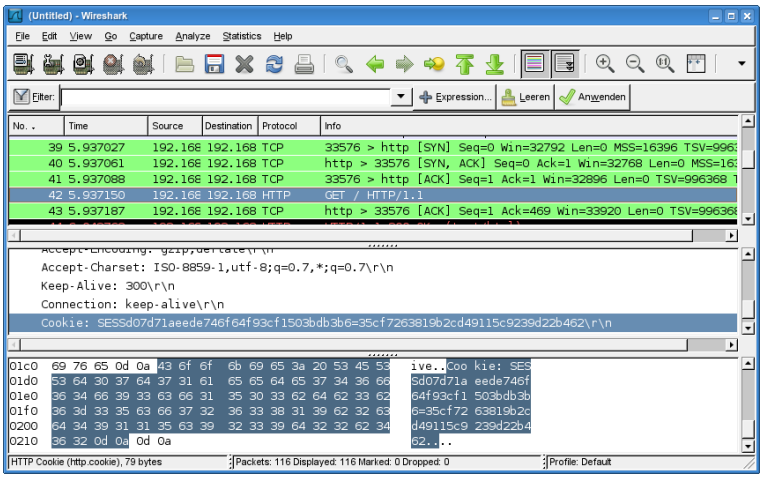
We have `samplesite.com`, which contains a plain Drupal 6.6 installation. Alice has an administration account on it.

As Alice cares about security, she always logs in via HTTPS. So she calls `https://samplesite.com/user` and enters her user name and password. She's logged in and creates some content.

While that is happening, Eve starts Wireshark and records the network traffic.

After Alice is done with her work on the web page, she wants to see how it looks. She calls the normal page with `http://samplesite.com/`, thinks it looks fine and is finished with her work on it.

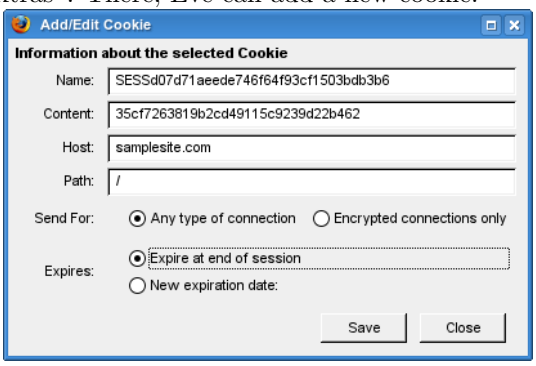
When this is happening, Eve will notice some HTTP connections to samplesite.com in her Wireshark. She now looks up the first GET request, which has the string "GET / HTTP/1.1" in the "Info"-column.



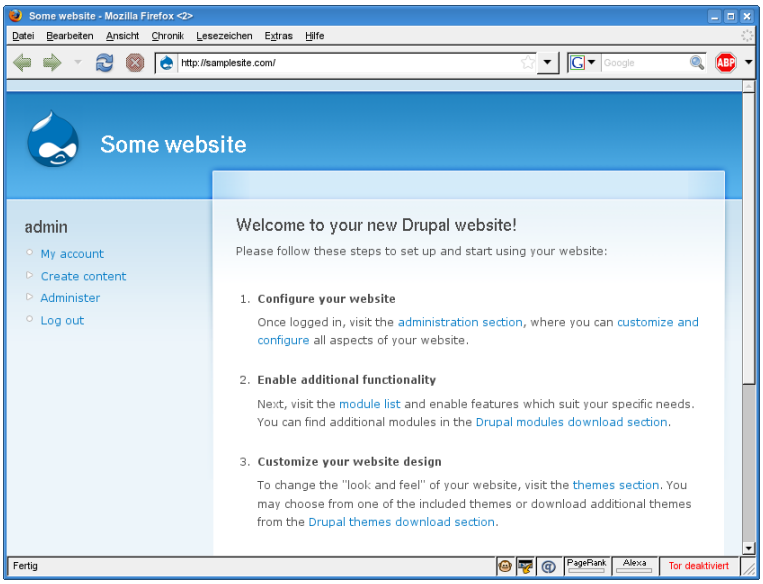
As you can see on the screenshot, the HTTP package contains a line saying:

```
Cookie: SESSd07d71aeede746f64f93cf1503bdb3b6=35
  cf7263819b2cd49115c9239d22b462
```

Now, Eve starts the Firefox webbrowser with the AnEC extension (cf. Appendix). With this extension, there is a menu point “Cookie Editor” under “Extras”. There, Eve can add a new cookie.



In the Name-field, she adds SESSd07d71aeede746f64f93cf1503bdb3b6, in the Content field 35cf7263819b2cd49115c9239d22b462. Host is samplesite.com and Path /, rest stays default. After saving that cookie, Eve calls `http://samplesite.com/`. Eve is logged in with an administrator account and can start modifying content.



4.5 Solution: Code example in PHP

PHP knows a function `session_set_cookie_params`. Starting with PHP 4.x.x, it has a method to set the secure flag on session cookies.

A simple approach would be to look for the `session_start` function and add some code before it.

Listing 7: PHP code for secure sessions

```
if ($_SERVER['HTTPS'])
 session_set_cookie_params( 0, '/', '', true, true );
else
 session_set_cookie_params( 0, '/', '', false, true );
```

At first, we check if the server variable HTTPS is set. If this is the case, we are running on an SSL secured connection. The first three parameters to `session_set_cookie_params` are just the defaults. The fourth parameter specifies if we want to have secure cookies. The fifth restricts cookies to be inaccessible by JavaScript. This is generally a good idea, as it makes XSS and CSRF attacks harder.

Obviously this only applies to applications using the session extension included in PHP. Many applications do their own session management and need to take care of that. It is advisable to only do this if there is a strong reason and if the programmers have a deep knowledge of the security aspects of sessions.

4.6 Hybrid solution

From a security point of view, it's the easiest and preferrable option to fully run sessions over HTTPS and set the secure flag for every cookie.

For large scale applications, this is a problem as SSL encryptions are performance expensive. On the client side, the performance impact is negligible. But on servers with a large amount of parallel connections, the cost for SSL encryption matters. Many large scale web pages (like amazon.com) only encrypt the login process and certain crucial actions. The normal session runs over unencrypted HTTP. This obviously has an impact on privacy.

A proper hybrid solution does the login over HTTPS and set's two cookies from which one has the secure flag set. After that, the user is forwarded to HTTP. As soon as some crucial action is done (for example buying an item), the application does a single HTTPS connection for that action and verifies the cookie with the secure flag. With such a design, an adversary could gather information about what the user is doing, but he cannot hijack the session. This reduces the costs for server performance to a minimum by giving up some privacy, but still providing secure authentication.

5 Examples

In August 2008, I had a look at some common web applications, especially the ones I was using myself. A useful tool for that was the Firefox extension CookieMonster (cf. Appendix A.1).

I found four applications vulnerable to that class of attacks. I contacted the developers in August 2008 with mixed results.

5.1 Menalto Gallery, Mantis, Squirrelmail

The developers of Menalto Gallery, Mantis (bugtracker) and Squirrelmail (webmail) all fixed the session hijacking issue after my report. I had the best experience with the developers of Menalto Gallery, they coordinated the disclosure timeline with me and released an updated version [15]. Both the developers of Mantis [16] and Squirrelmail [17] released an updated version after I released a public advisory.

5.2 Drupal

I found the same session hijacking issue also in the popular Drupal Content Management System [18]. The Drupal team decided that they do not consider this an issue in their application and have not provided a fix yet. Their reaction, which can be found at `http://drupal.org/node/315703`, is:

> "[...] we consider that this is a configuration problem. It's your responsibility to set session.cookie_secure in the SSL virtual host if you want an SSL-only website."

Their statement concludes that if someone wants protection against session sniffing, it is up to the user to configure the environment to set the secure flag.

PHP has an option to enforce secure cookies that can be set within the web server or via htaccess. This has, however, some disadvantages.

Setting this option will let PHP set the secure flag on all session cookies, which means that a user has no option to open a session without SSL. There are lot's of possible hybrid situations where one might want to have some normal web page usage without SSL (e. g. for performance reasons or because one has no payed certificate and does not want normal users to fiddle with certificate issues).

The other problem is that a user might not be able to set PHP options for a virtual host. Drupal is a common PHP CMS, which is probably mainly used in shared hosting environments. For such users it is impossible to get secure sessions on Drupal without modifying the code.

5.3 Serendipity

The Serendipity weblog system has a hybrid solution. It generates a normal PHP session for every user accessing the page and stores things like the entered name in the comment field. It also provides the session for plugins, e. g. the Karma-plugin uses the session.

The PHP session is not setting the secure flag for sessions. This is fine, as there are no sensitive things one can do with a hijacked Serendipity PHP session.

If an author / admin of the weblog logs into serendipity, it sets another cookie named `author_token`, which contains a hashed random value. For this cookie, the secure flag is set, so if an author logs in on an SSL-enabled connection, the cookie is never transferred unencrypted. If the web server is configured to forward all connections to /admin and /serendipity_admin.php to https, it is guaranteed that the session data is kept safe.

The `author_token` cookie contains a value independent from the PHP session ID. I found no flaw in this design and think it is a well-designed example for a hybrid solution.

5.4 Wordpress

Starting with version 2.6.0, Wordpress also generates two cookies for authentication. Wordpress stores a value it internally calls salt, which is randomly generated on session creation. It sets three cookies, one `wordpress_logged_in` (which is not secure) and two `wordpress_sec` (which have the secure flag set). There are two `wordpress_sec` cookies because they are restricted to specific paths on the installation, else they are identical.

The content of the cookies is generated out of a random value and hashed together with some other values (username, expiration time) using the MD5-algorithm to the content of the cookie. Considering the oneway functionality of the hash function is guaranteed (which is a proper assumption, the weaknesses known to MD5 do not endanger that), that design should be secure.

What may be confusing is that the whole security is based on a value they call salt. In common cryptographic designs, the term salt is not used that way, it is for a random but usually not a secret value.

5.5 eBay

I did some research on the german page of the popular eBay platform on 2008-11-28. eBay sets a number of cookies on login. None of them sets the secure flag, so eBay is completely open to session hijacking.

It has three cookies that seem to be important for the session, `s`, `cid` and `npii`. A couple of other cookies seem to have no meaning for the session (`ds1`, `dp1`, `ns1`, `shs`, `ebay`, `lucky9`, `nosession`), they can be deleted and the user is still logged in.

With a hijacked session (done with the manual method described above hijacking the three cookies `s`, `cid` and `npii`), I was able to do a bid on an article. I was even able to change my personal address.

For some actions ebay requires an extra login that is SSL-secured. This is done for password changing. It would be a possibility to use this as a security concept and secure all important actions that way, making it impossible to use a hijacked session to do any harm. Though, as eBay does not secure bids and address changing in any way, it is obvious that this is not secure.

It seems that eBay recently changed its way of handling cookies, on a first look I had some weeks ago, they had two secure cookies set. I have not investigated that further at that point, so I cannot say if they had a secure cookie concept.

In April 2008, eBay published a press release stating that they'd improve security by using Flash cookies if a Flash plugin is installed. Flash cookies are more or less a duplication of the cookie concept inside the proprietary Flash system. On my tests, although the Flash plugin was installed, eBay did not set any Flash cookies.

5.6 Other examples for attacks against sessions

The attack described in this paper has some similiarities to other attacks based on the fragile design of sessions in the world wide web. I'll give a short introduction to two very popular examples.

5.6.1 Cross Site Scripting (XSS)

An attack that forces a victim to run some malicious client side script (usually JavaScript) code in the context of the attacked web application is called Cross Site Scripting (abbreviation XSS, cf. [19]).

Imagine we have some web application that has some search form. The search form sends its request with a GET-variable to another script on the server. Though we have an URL like `http://www.some-site.com/search.php?searchterm=example`. If the output page prints the search term without

any preprocessing, e. g. a PHP-code like `echo "You've searched for ".\$\_GET['searchterm'];`, we have a typical Cross Site Scripting vulnerability.

What an adversary can do now is sending the victim some URL like `http://www.some-site.com/search.php?searchterm=<script>alert(1)</script>`, that will execute JavaScript code in the context of `www.some-site.com`. The adversary does not have to send the victim such an URL directly. He could also send some harmless looking link to his own site which embeds the malicious link in some way, like within an `<iframe>` tag.

The above example will just open a bogus alert window. But it could be any JavaScript code. Especially, if `www.some-site.com` is using sessions, the URL could look like this: `http://www.some-site.com/search.php?searchterm=sendcookie`. This would transfer the session cookie to `maliciousssite.com` - the adversary has the session cookie of the victim.

To prevent XSS, it is crucial that a web application never outputs any content from an unknown source without proper escaping. Escaping means that all characters with a special meaning in HTML (`<`, `>` and `&`) are transferred into their HTML-entities (<, >, &). PHP knows the function `htmlspecialchars()` for this. Even this is not 100 % secure, as charset issues can circumvent such escaping.

5.6.2 Cross Site Request Forgery (CSRF)

A Cross Site Request Forgery (abbreviation CSRF) attack is a way to trigger a certain action in a web application externally through a link. It is quite nontrivial to prevent CSRF attacks. All methods have at least some disadvantage.

Let's assume we have some web application for E-mails. It has a list of mails, each mail contains a link like `http://www.mywebmailer.com/delete.php?mailid=1` to delete the mail. What an adversary could do now is sent the victim some link that triggers this action. The adversary could, for example, place small/invisible `<iframe>` tags on his web page for all mailids from 1 to 1000. If the victim calls the adversarys web page while logged into the web mailer, all his mails get deleted.

Similar attacks can also be used for pretty much every action inside the web application of the victim. For example, in an online shop it could be used to buy some article for a certain address. If the actions use POST instead of GET variables, the attack get's more complex, as the adversary has to use JavaScript to send the data. For nontrivial actions (buying in a web shop may require several steps), it adds even more complexity to the attack. Though this is no protection, it just increases the effort needed for an attack.

To prevent CSRF, the common method is to add some token as a hidden value in all forms that trigger data-changing actions. The web application has to check the validity of the token before doing any actions. There are various ways how such a token can be generated. It could be just some random value where the application stores the kind of action and the token in some table. On complex applications and long usage sessions, such a table can become pretty large. Just saving the last token is usually not feasible. This would make parallel

usage of web applications in more than one browser window impossible.

A method that combines several advantages looks like this: On session creation, the web application generates some random value (let's call it the session token) that is valid for the whole session. One could use the session ID, but in certain scenarios, using an independent value is more secure. With every form, a hash value is generated out of the session token and an identifier for the action. The script performing the action can check if that token is valid.

An important note: If there is an XSS in a web application, the token-method to prevent CSRF is useless - the token can be transferred via JavaScript.

The attack known today as CSRF has already been described 1997 in chapter 4.3.5 of RFC 2109 [6]. Nevertheless CSRF still can be considered a vulnerability that is rarely known today. There are many web applications still vulnerable to CSRF.

6 Conclusion

6.1 Severity

For a successful attack, the adversary needs to be able to read the network traffic from the victim. There are various scenarios where this is the case (cf. 3.1).

The most likely case is that the adversary and the victim are in the same network. This can be in an Internet cafe, at a public wireless hotspot or in a company. The advantage for the adversary is that in this case he usually already has the same public IP address through NAT, so any IP-based filtering by the session management will not stop the attack.

6.2 Measurements

When opening a session, web applications should always check the connection type (HTTP or HTTPS) and add the secure flag to the session cookie if the session is HTTPS. This will in almost all cases be the expected behaviour by the user.

It could be discussed if session frameworks (like the PHP session extension) should handle this themselves.

Statements like the one from the Drupal team (cf. 5.2) ignoring the security issues around sessions, are dangerous. Web applications should be shipped in a way that will make them secure by default, even if an unskilled user uses them.

6.3 An alternative to HTTP?

What we've seen in this document:

HTTP is a stateless protocol

and

this is a problem!

A large number of security issues in the web rely on this simple fact. We have workarounds, but on a closer look, they are not more than workarounds and far away from solid solutions. As long as there are no better alternatives, one should use secure flags of cookies and set tokens for actions. Though the question arises if this is the solution on the long term.

The only way to properly fix those issues would be replacing HTTP. I do not know of any efforts to do that. I doubt such an attempt would gain much success. Looking at the history of other networking standards (just think of IPv6), the Internet is very inertial in implementing new technology, even if there are strong reasons. Replacing HTTP would change the whole basis of what is called the World Wide Web. Such a change would for example require all web browsers to be replaced.

It is probably a thing one has to accept about the Internet: It is made up by poor designs and often inadequate technologies. It is a compromise of what works and what is available at a time. From a security point of view, this often leads to unelegant workarounds to keep the whole thing together.

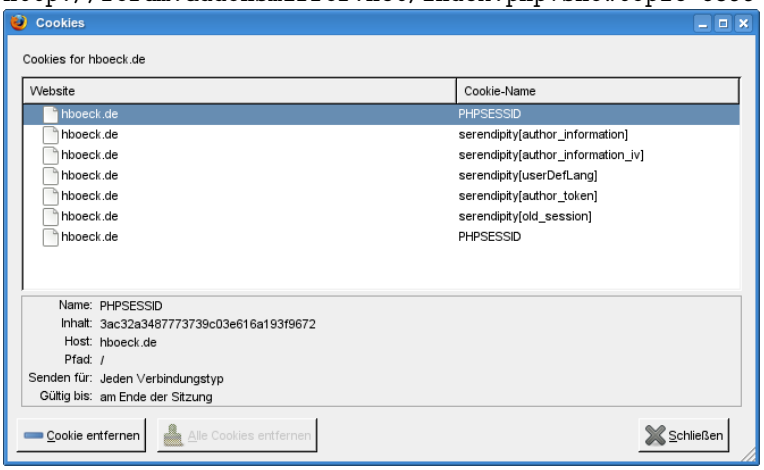
A Used tools

This appendix describes some tools mentioned in this work.

A.1 CookieMonster extension for Firefox

The Firefox web browser has a large number of extensions available. The CookieMonster extension is very useful to monitor the behaviour of web pages regarding cookies. It allows to open a simple popup listing all cookies on the current host. As it also shows if the secure flag is set, this is a handy tool to check web applications if they properly set the secure flag on HTTPS connections.

`http://forum.addonsmirror.net/index.php?showtopic=6599`



A.2 Add N Edit Cookies (AnEC) extension for Firefox

The Add N Edit Cookies (AnEC) extension for Firefox adds a cookie editor to Firefox. Usually cookies are not meant to be directly manipulated by the user, so a normal web browser interface has no cookie editing capabilities.

For a cookie hijacking attack, we need to be able to add cookies to our browser session, the cookie editor of AnEC can do this.

`http://addneditcookies.mozdev.org/`

A.3 Wireshark

Wireshark (former name Ethereal) is a popular sniffing tool. It has various capabilities to decode known protocols. Though one does not have to search for the HTTP header inside a bunch of TCP packages, Wireshark already knows TCP and HTTP and does the decoding. It also has complex filtering capabilities.

`http://www.wireshark.org/`

A.4 surfjack

With his publications about the SSL session hijacking attack, Sandro Gauci released a tool called surfjack which automates it. It listens on a network interface and automatically looks for unencrypted cookies. It then provides a proxy server which can be used by the adversary to set his cookies (avoiding the need of a cookie editing tool).

`http://code.google.com/p/surfjack/`

There is a short video tutorial that shows how surfjack works:

`http://www.vimeo.com/1507697`

During my research, I was unable to get surfjack to work. It is barely documented, it starts and it looks like it is collecting cookies, but the proxy page does not show any of them.

B Sources

This work was done with LaTeX. For RFC citations, I've used the rfc.bib from `http://www.tm.uka.de/~bless/bibrfcindex.html`.

C License

This work is licensed under the Creative Commons Attribution 3.0 License, that means you are free to copy and reuse this work as long as you mention my name as the source.

`http://creativecommons.org/licenses/by/3.0/`

References

[1] Steve Christey and Robert A. Martin. Vulnerability type distributions in CVE, 2007. Available from: http://cwe.mitre.org/documents/vuln-trends/.

[2] International Telecommunication Union. ITU-T X.200 – open systems interconnection - model and notation, 1994. Available from: http://www.itu.int/rec/T-REC-X.200-199407-I/en.

[3] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168. Available from: http://www.ietf.org/rfc/rfc793.txt.

[4] Tim Berners-Lee. Original HTTP 0.9, 1991. Available from: http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996. Available from: http://www.ietf.org/rfc/rfc1945.txt.

[6] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Proposed Standard), February 1997. Obsoleted by RFC 2965. Available from: http://www.ietf.org/rfc/rfc2109.txt.

[7] ECMA. ECMAScript language specification, 1999. Available from: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[8] Robert Auger. The cross-site request forgery (CSRF/XSRF) FAQ, 2008. Available from: http://www.cgisecurity.com/articles/csrf-faq.shtml.

[9] NIST and Stefan Esser. CVE-2008-4107 – the rand and mt_rand functions in php 5.2.6 do not produce cryptographically strong random numbers, 2008. Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4107.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Available from: http://www.ietf.org/rfc/rfc5246.txt.

[11] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), April 2002. Obsoleted by RFC 5280, updated by RFCs 4325, 4630. Available from: http://www.ietf.org/rfc/rfc3280.txt.

[12] Gibson Research Corporation. ARP cache poisoning, 2005. Available from: http://www.grc.com/nat/arp.htm.

[13] US CERT. Web sites may transmit authentication tokens unencrypted. Available from: http://www.kb.cert.org/vuls/id/466433.

[14] Sandro Gauci. Surf jack: HTTPS will not safe you, 2008. Available from: http://enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you/.

[15] NIST and Hanno Böck. CVE-2008-3662 - session hijacking in menalto gallery, sep 2008. Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3662.

[16] NIST and Hanno Böck. CVE-2008-3102 - session hijacking in mantis bugtracker, 2008. Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3102.

[17] NIST and Hanno Böck. CVE-2008-3663 - session hijacking in squirrelmail, sep 2008. Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3663.

[18] NIST and Hanno Böck. CVE-2008-3661 - sessio hijacking in drupal cms, sep 2008. Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3661.

[19] Jason Rafail and CERT Coordination Center. Cross-site scripting vulnerabilities, 2001. Available from: http://www.cert.org/archive/pdf/cross_site_scripting.pdf.