

Mutillidae Security Audit

Performed By: Tyler Jackson
Date: 5/9/2015

Purpose:

This security audit is designed to provide an overview of the security vulnerabilities of Mutillidae. It will also provide recommendations on how to handle these vulnerabilities, as well as providing a breakdown of the risk of each vulnerability. While not a full security audit, this will function as a resource to significantly increase the security of the web application and in turn the company.

Pages of the Application that were reviewed:

Page	Description
login.php	This page is where a user would login.
user-info.php	This page allows a user to print someone's user information. It requires their username and password.
view-someones-blog.php	This page is used to view a user's blog entries. You can view anyone's blog entries.
capture-data.php	This page captures any packets sent over the network, and stores them to be displayed elsewhere.
add-to-your-blog.php	This page lets a user add a blog under their name if logged in or an anonymous blog if not logged in.
text-file-viewer.php	This page lets you choose between 5 different text files to view.
register.php	This lets you register as a new user. Asks for username, password, and a signature.
set-background-color.php	This lets you specify a 6 character hex-value to use as the new background color.
document-viewer.php	This page lets the user choose between 4 different documents to display to the screen.

This figure shows a list of all the pages that were reviewed, with a brief description of what the page was intended for. Later you will see that many of these pages are being used in ways that they were not intended.

Vulnerabilities:

#	Page	Vulnerability	Attack Type
1	login.php	Username/password input is not validated	Sql Injection
2	user-info.php	Username input is not validated	Sql Injection
3	view-someones-blog.php	Username input tag's value not validated	Sql Injection
4	view-someones-blog.php	Output of blog entry not encoded	XSS
5	view-someones-blog.php	Output of blog entry not encoded	Html Injection
6	capture-data.php	Url is not validated and captured data output is not encoded	XSS
7	capture-data.php	Url is not validated and captured data output is not encoded	Html Injection
8	captured-data.php	Output of captured packets is not encoded	XSS
9	captured-data.php	Output of captured packets is not encoded	Html Injection
10	add-to-your-blog.php	Blog entry input is not validated and output is not encoded	XSS
11	add-to-your-blog.php	Blog entry input is not validated and output is not encoded	Html Injection
12	register.php	Signature input is not encoded -> this gets displayed on other pages	XSS
13	register.php	Signature input is not encoded -> this gets displayed on other pages	Html Injection
14	text-file-viewer.php	Insecure Direct Object Reference	Print any file on server to screen
15	set-background-color.php	Input not validated, output not encoded	Html Injection
16	document-viewer.php	Insecure Direct Object Reference	Print any file on server to screen

The table above shows a list of vulnerabilities found across the pages we reviewed. These vulnerabilities can be exploited with sql injection, Html injection, or XSS. There are also a couple pages that have insecure direct object references. Next we will take a detailed look at exactly how these vulnerabilities could be exploited, what the consequences of each exploit would be, and ways to prevent them.

Detailed Description of Exploiting These Vulnerabilities

Vulnerability #:

- 1.) This vulnerability can be found on the login page on the website or from `login.php` in the code. The username field's input and the password field's input are not validated prior to being used in the sql script to check if the username password combination is correct in the database. This means that if someone provided sql code instead of a username then it would be injected into the current sql query. For example, if a user put the highlighted text below then it will authenticate the user as an admin.

“ OR 1=1 -- “

The images below shows this attack being used as well as the result of the attack.

The screenshot shows the login page of OWASP Mutillidae II. The top navigation bar includes: Version: 2.6.3.1, Security Level: 0 (Hosed), Hints: Disabled (0 - I try harder), and Not Logged In. The main navigation bar contains links: Home, Login/Register, Toggle Hints, Toggle Security, Reset DB, View Log, View Captured Data, Hide Popup Hints, and Enforce SSL. The login form has a title "Login" and buttons for "Back" and "Help Me!". The "Please sign-in" section contains a "Name" field with the value "' OR 1=1 --", a "Password" field, and a "Login" button. Below the form, it says "Dont have an account? Please register here".

The screenshot shows the main page of OWASP Mutillidae II. The top navigation bar includes: Version: 2.6.3.1, Security Level: 0 (Hosed), Hints: Disabled (0 - I try harder), and Logged In Admin: admin (root). The main navigation bar contains links: Home, Logout, Toggle Hints, Toggle Security, Reset DB, View Log, View Captured Data, Hide Popup Hints, and Enforce SSL. The main heading is "Mutillidae: Deliberately Vulnerable Web Pen-Testing Application". Below the heading, there are several links and icons: "Like Mutillidae? Check out how to help", "What Should I Do?", "Video Tutorials", "Help Me!", and "Listing of vulnerabilities".

This attack can be prevented by validating the input. There are a few different spots where the code needs to be changed. First is in `MySQLHandler.php` which is the main database class that all other queries use as a utility class. The function `doExecuteQuery` takes a query string as input and returns the result of the query. Here is an image of the code snippet.

```
private function doExecuteQuery($pQueryString){
    try {
        $lResult = $this->mMySQLConnection->query($pQueryString);

        echo $pQueryString;
        if (!$lResult) {
            throw (new Exception("Error executing query: ".$this->se
            ializeMySQLImprovedObjectProperties()."));
        } // end if there are no results

        return $lResult;
    } catch (Exception $e) {
        throw(new Exception($this->mCustomErrorHandler->getExcep
        tionMessage($e, "Query: " . $this->Encoder->encodeForHTML($pQueryString)));
    } // end function

} // end private function executeQuery
```

The query call needs to be replaced with the `mysqli prepare` function. This function also needs an `execute` call to execute the query, and the output will need to be handled with the `mysqli get_result()` function. This prepare statement removes the dynamic input from the query string, and allows `mysqli` to escape any dangerous characters that an attacker might have been trying to sneak in the query. Then `mysqli` will replace the `?` marks in the query string with these safe variables. Therefore, the query itself needs to be adjusted wherever `ExecuteQuery` is being called to remove the dynamic input and replace them with `?` marks. As a result of handling this at the DB level, every query that uses this function will need to make this same fix. `ExecuteQuery` should therefore also take an array of parameters to be used for the corresponding query. Although it may seem tedious to make this change here because a lot of the queries will break, you will have peace of mind that the queries that are breaking will be resistant to sql injection once they are converted.

The other file that needs to be adjusted for the login query to be resistant to sql injection is `SQLQueryHandler.php` which holds most of the queries called throughout the application. The `GetUserAccount` function needs the query string to have `?` marks instead of the username and password variables, and then these variables can get passed to our updated `ExecuteQuery` function in an array. Below is a picture of the code snippet that needs to be changed.

```

public function getUserAccount($pUsername, $pPassword){^M
    /* ^M
    * Note: While escaping works ok in some case, it is not the bes
t defense.^M
    * Using stored procedures is a much stronger defense.^M
    */^M
    ^M
    if ($this->stopSQLInjection == TRUE){^M
        $pUsername = $this->mMySQLHandler->escapeDangerousCharac
ters($pUsername);^M
        $pPassword = $this->mMySQLHandler->escapeDangerousCharac
ters($pPassword);^M
    }// end if^M
    /*
    $lQueryString = ^M
        "SELECT * FROM accounts ^M
        WHERE username='". $pUsername.^M
        "' AND password='". $pPassword.'''";           ^M
    ^M
    return $this->mMySQLHandler->executeQuery($lQueryString);^M
    */
    $lQueryString = "SELECT * FROM accounts WHERE username= ? AND pa
ssword = ?";
    return $this->mMySQLHandler->executeQueryNew($lQueryString, "ss",
array("ed", "pentest"));
} //end public function getUserAccount^M

```

2.) The user-input.php page has a similar vulnerability. This page takes a username and password as parameters to display a user's information to the page (i.e. their username, password, and signature). This input is vulnerable to the same attack as above, but instead of granting unauthorized access it will allow someone access to everyone's account information. So that this document can be viewed by multiple employees in the company we have left out the image that includes the attack and everyone's user information. Fortunately, this vulnerability will be fixed when the first one is fixed as they use the same functions.

3.) The third vulnerability is still a sql injection attack but instead of being in an input field it is in an option tag's value field. When trying to view a user's blog posts all of the usernames are stored in plaintext on the page and are used in the subsequent query. Therefore if this value is changed it creates a sql injection vulnerability.

Here is a picture of the field prior to being changed.

	Name	Date	Comn
1	admin	2009-03-01 22:31:13	Fear me, for I am
2	CHARACTER		
3	COLLATIONS		
4	COLLATION_CHARACTER_SET_APPLICABILITY		
5	COLUMNS		
6	COLUMN_PRIVILEGES		
7	ENGINES		
8	EVENTS		
9	FILES		
10	GLOBAL_STATUS		
11	GLOBAL_VARIABLES		

To fix this attack the only thing that needs to be changed is the `GetBlogRecord` function. Similarly to before, the `blogger_name` needs to be passed to our new `ExecuteQuery` function in the array of parameters, and it needs to be replaced by a `?` in the query string. Below is the print out of that function for reference.

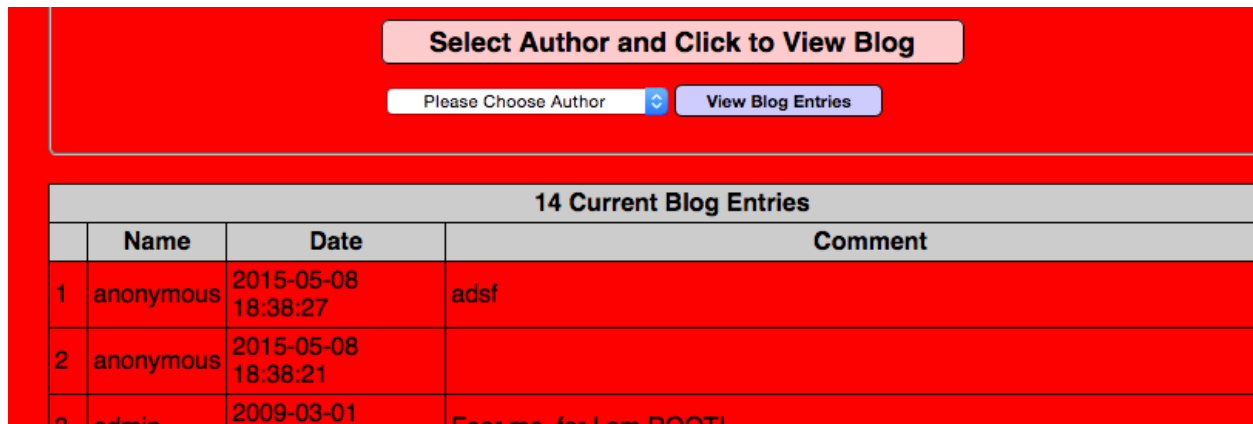
```

public function getBlogRecord($pBloggerName){^M
^M
    if ($this->stopSQLInjection == TRUE){^M
        $pBloggerName = $this->mMySQLHandler->escapeDangerousCha
racters($pBloggerName);^M
    }// end if^M
^M
    echo("in get all blog records");
    $lQueryString = "^M
        SELECT * FROM blogs_table ^M
        WHERE blogger_name like '{$pBloggerName}%' ^M
        ORDER BY date DESC ^M
        LIMIT 0 , 100";^M
        ^M
    return $this->mMySQLHandler->executeQuery($lQueryString);^M
} //end public function insertBlogRecord^M
^M

```

4 / 5). Vulnerabilities 4 and 5 can be combined as they will be fixed by the same change. Both are as a result of the blog entries not being encoded before they are output to the screen. That means if someone made a blog entry with html code inside it then that text will be interpreted as html code rather than text. Similarly, if someone made a blog post with a javascript script inside it then when that blog is output to the view-blog-entries page it will be interpreted as javascript rather than text. This can all be fixed by encoding the output based on the context. One example of a fix would be to use php's `htmlspecialchars` function that takes an input string and replaces any questionable characters to html entities. It also takes some input flags as a parameter that specify the type of encoding to be done. You could also use Owasp's HTML encoder class found in the

owasp-esapi-php/src/Encoder.php class. After this is fixed then a blog entry won't be interpreted as javascript or html. For reference, here is an example of what such an attack would look like.



14 Current Blog Entries			
	Name	Date	Comment
1	anonymous	2015-05-08 18:38:27	adsf
2	anonymous	2015-05-08 18:38:21	
3	admin	2009-03-01	Fear me, for I am BOO!

As you can see the javascript changed the background of the screen to red, and the text of the blog entry isn't there because it was interpreted as javascript instead.

6 / 7 / 8 / 9.)

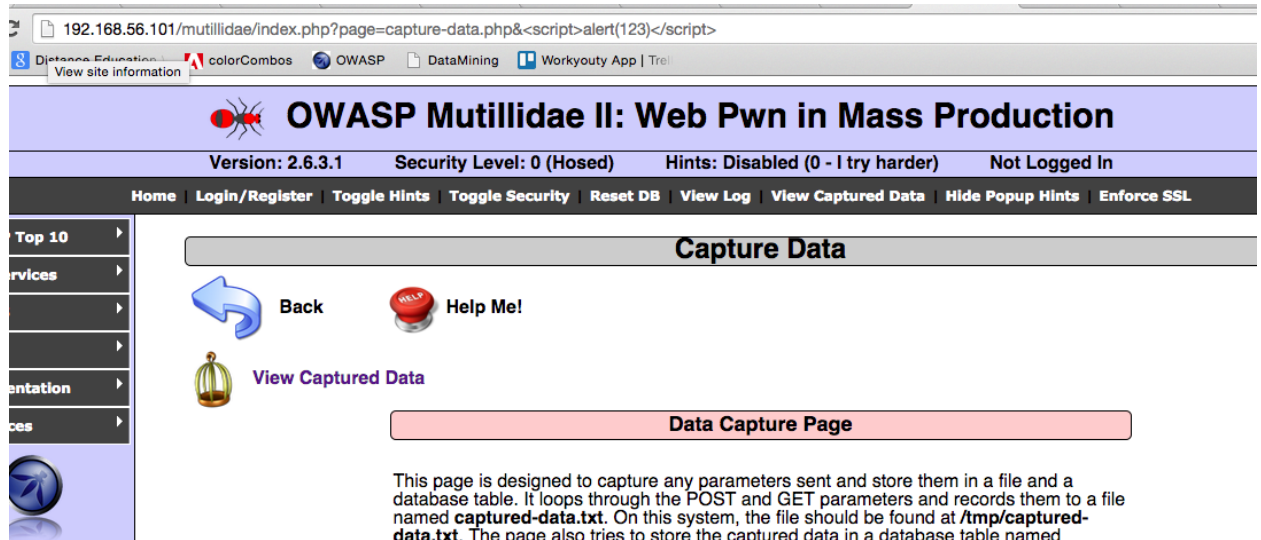
These vulnerabilities can all be fixed at the same time in one location so they will be combined here. They all result from not handling the captured data correctly which can be controlled by the user. First, the captured data that is being collected is not being validated to remove characters that could be used for javascript code or html code before it is being stored. Second, this captured data is not being html encoded when it is being displayed to the screen on the `captured-data.php` page, so when it is being injected into an html field it is being interpreted as html code or javascript code and getting executed. Sometimes, a user might want to accept a wider range of characters as input. In order to still protect against html injection and XSS we can html encode the output. This is less secure, because this data is being stored in a threatening state. Therefore you must be sure to never print it to the screen without encoding it. That being said, the spot to fix it is in `captured-data.php`. Here is a code snippet.

```
$lHostname = $row->hostname;
$clientIpAddress = $row->ip_address;
$clientPort = $row->port;
$clientUserAgentString = $row->user_agent_string;

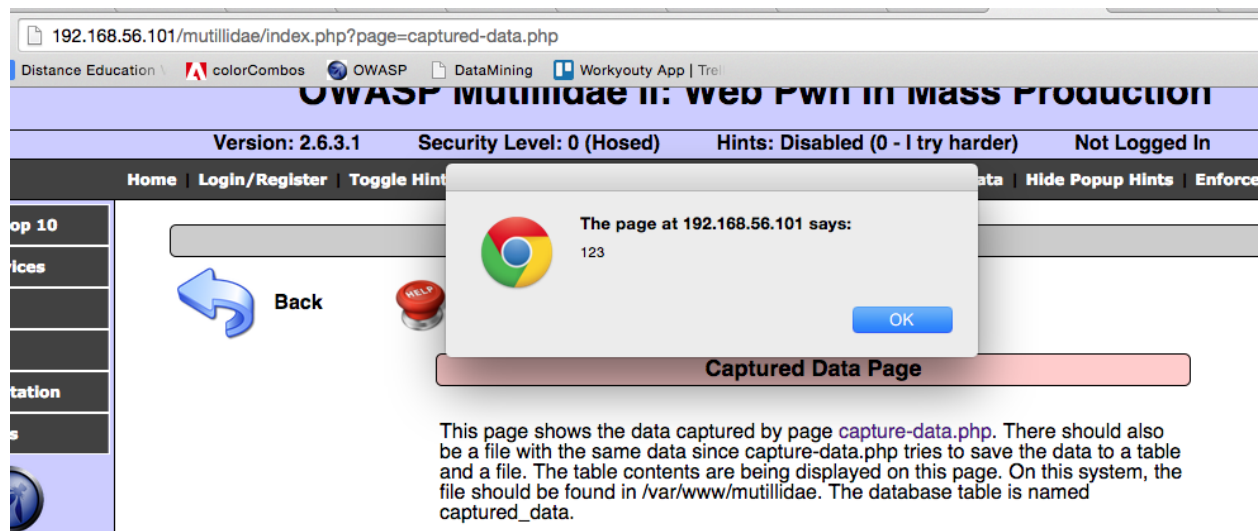
$clientReferrer = $row->referrer;

$lData = $row->data;
$captureDate = $row->capture_date;
```

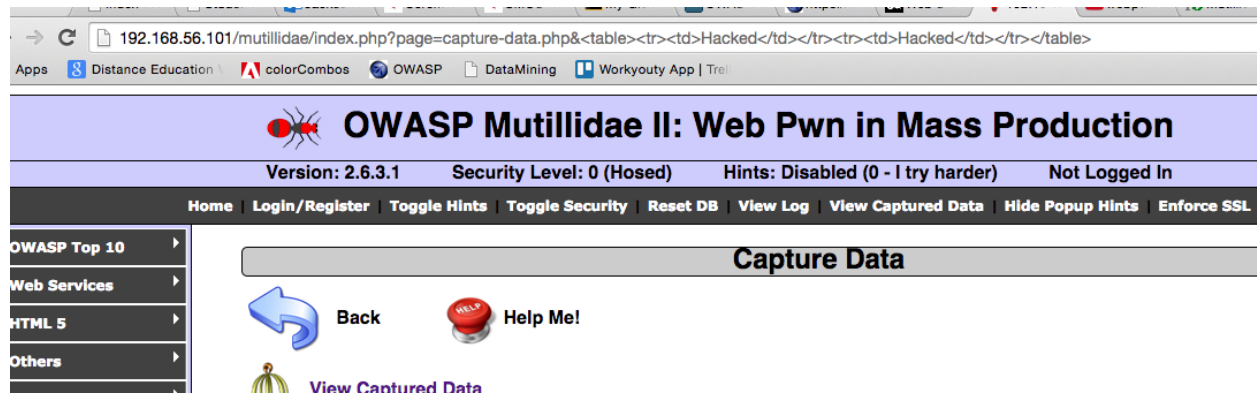
By passing these strings to Owasp's encoder class found in `owasp-esapi-php/src/Encoder.php` we can safely encode an html or javascript characters and then display these strings to the webpage. Here is an example of a XSS attack that this page is vulnerable to. Notice the javascript in the url.



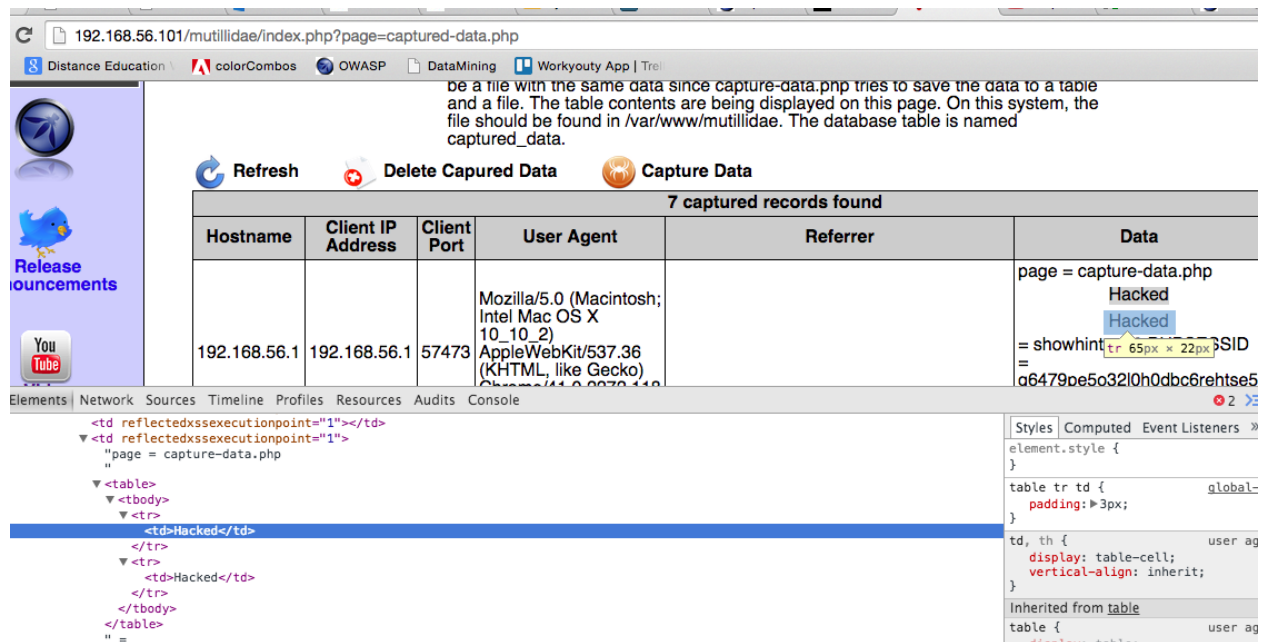
Here is an example of what this results in on the `captured-data.php` page if it is not encoded correctly. Obviously these are innocent use cases, but you can imagine what an attacker could do with this vulnerability.



Here is an example of an html injection attack performed similarly from the `capture-data.php` page. Notice again, the html code in the url.



Then here is what this captured data looks like on the `captured-data.php` page. Notice how this was output as actual html on the webpage rather than text in the data column.

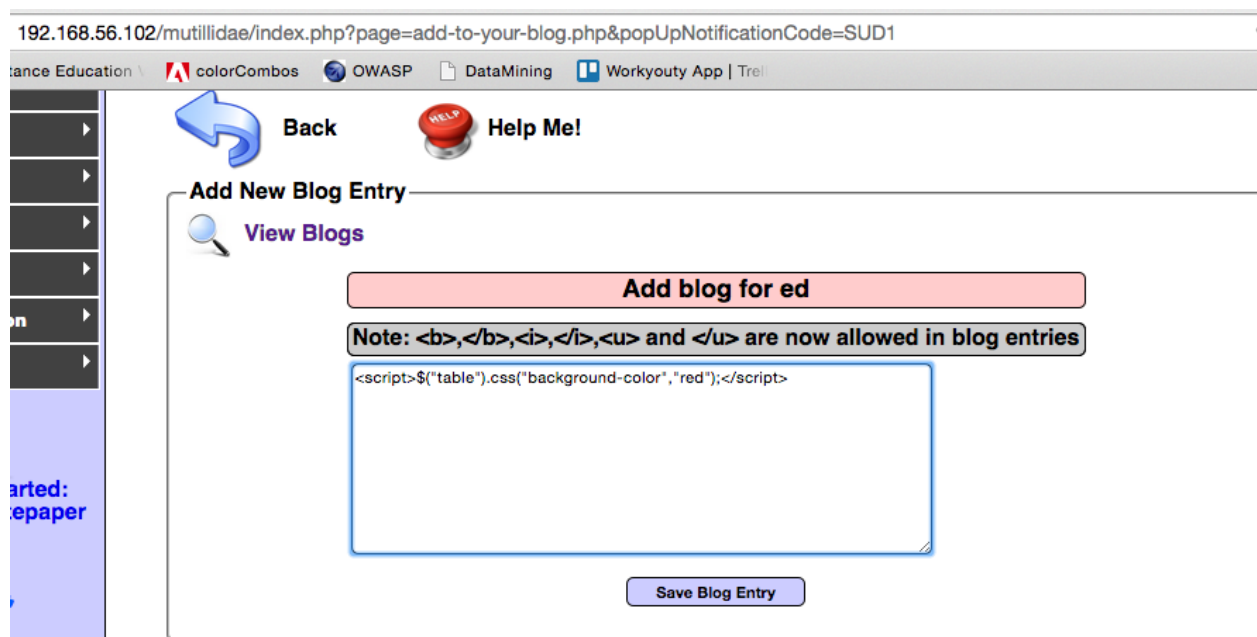


10 / 11.) These two vulnerabilities will be fixed in the same way as the previous vulnerability and therefore they will be combined. These vulnerabilities are XSS and HTML injection and result in not encoding the output from a blog post. This is the same fix as the 4 / 5 vulnerability fix where the output was not encoded when a blog entry was displayed on the `view-someones-blog.php`. When you add a new blog post the entry immediately gets displayed to the `add-to-your-blog.php` webpage. Therefore, a user's input is being

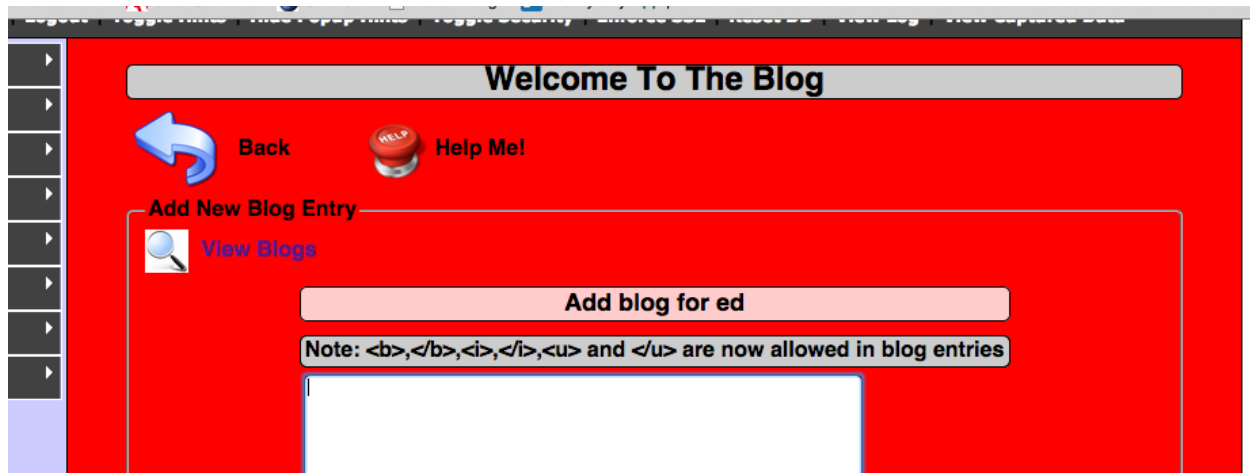
output and needs to be encoded. A blog entry is another good example of when a user might want to allow special characters to be included making it more difficult to just validate input. By encoding the output, someone can write a blog post with html tags or javascript and allow them to be displayed as text. By again using Owasp's encoder class to html encode the output these vulnerabilities will be fixed. The code that needs to be wrapped with the encode function is show below.

```
$!BloggerName = $!Record->blogger_name;  
$!Date = $!Record->date;  
$!Comment = $!Record->comment;
```

An example of what this attack could look like is in the picture below. This javascript will change the background color of any table on the page to red.



The result of the attack above is shown below. Instead of displaying the text in the comment field, it was interpreted as javascript, the screen turned red and the comment field is blank. By encoding the output, this text would show up in the comment field as it was input.



12 / 13). These vulnerabilities are XSS and HTML Injection on the `register.php` page. Again, they can be fixed with output encoding, as they are displayed occasionally throughout the site. For example on the homepage it displays the username and the signature of whoever's logged in. If javascript or html were in the signature then this would get executed when that text was meant to be displayed. Encoding the output prevents this. This might be a scenario in which using special characters as input could be forbidden and then input validation could be used to prevent any of these characters from being stored in the database. Here is an example of using javascript in the signature field when registering.

Please choose your username, password and signature

Username

Password

Confirm Password

Signature

"><script>alert('We hacked this site');</script>"

The result of this attack can be viewed from the home page whenever that user logs in.



This code needs to be adjusted wherever the signature is displayed to the page. For the home page (`index.php`) here is the code snippet that needs to be wrapped in html encoding.

```
$_SESSION['loggedin'] = 'True';^M
$_SESSION['uid'] = $row->cid;^M
$_SESSION['logged_in_user'] = $r^M
$_SESSION['logged_in_usersignatu^M
$_SESSION['is_admin'] = $row->is^M
```

14.) This vulnerability is on the `text-file-viewer.php` page and enables an attacker to print any file on the system to the screen. This can be prevented by changing the option tags' value attributes to hold aliases for the filenames, not the actual file path. This will make it harder for an attacker to guess the file paths, as well as prevent an attacker from being able to just substitute in a file name for the one provided. This alias can then be used in an `if/else` or `switch/case` statement later to choose the correct file based on the alias. If an unknown value is provided then default action can be taken. The spot the code needs to be fixed is in the `text-file-viewer.php` html code that has hardcoded the filenames. As you can see this code is already configured to be switched over to using aliases.


```

<option value="<?php if ($lUseTokenizati
on){echo 1;}else{echo 'http://www.textfiles.com/hacking/auditool.txt';}}?>">Intru
sion Detection in Computers by Victor H. Marshall (January 29, 1991)</option>
<option value="<?php if ($lUseTokenizati
on){echo 2;}else{echo 'http://www.textfiles.com/hacking/atms';}}?>">An Overview o
f ATMs and Information on the Encoding System</option>
<option value="<?php if ($lUseTokenizati
on){echo 3;}else{echo 'http://www.textfiles.com/hacking/backdoor.txt';}}?>">How t
o Hold Onto UNIX Root Once You Have It</option>
<option value="<?php if ($lUseTokenizati
on){echo 4;}else{echo 'http://www.textfiles.com/hacking/hack1.hac';}}?>">The Basi
cs of Hacking, by the Knights of Shadow (Intro)</option>
<option value="<?php if ($lUseTokenizati
on){echo 5;}else{echo 'http://www.textfiles.com/hacking/hacking101.hac';}}?>">HAC
KING 101 - By Johnny Rotten - Course #1 - Hacking, Telenet, Life</option>

```

Here is what the html code currently looks like on the webpage. As you can see the "links are displayed in plaintext."

Just choose one from the list and submit.

Text File Name 

[View File](#)

For other great old school hacking texts, check out <http://www.textfiles.com/>.

File: <http://www.textfiles.com/hacking/auditool.txt>

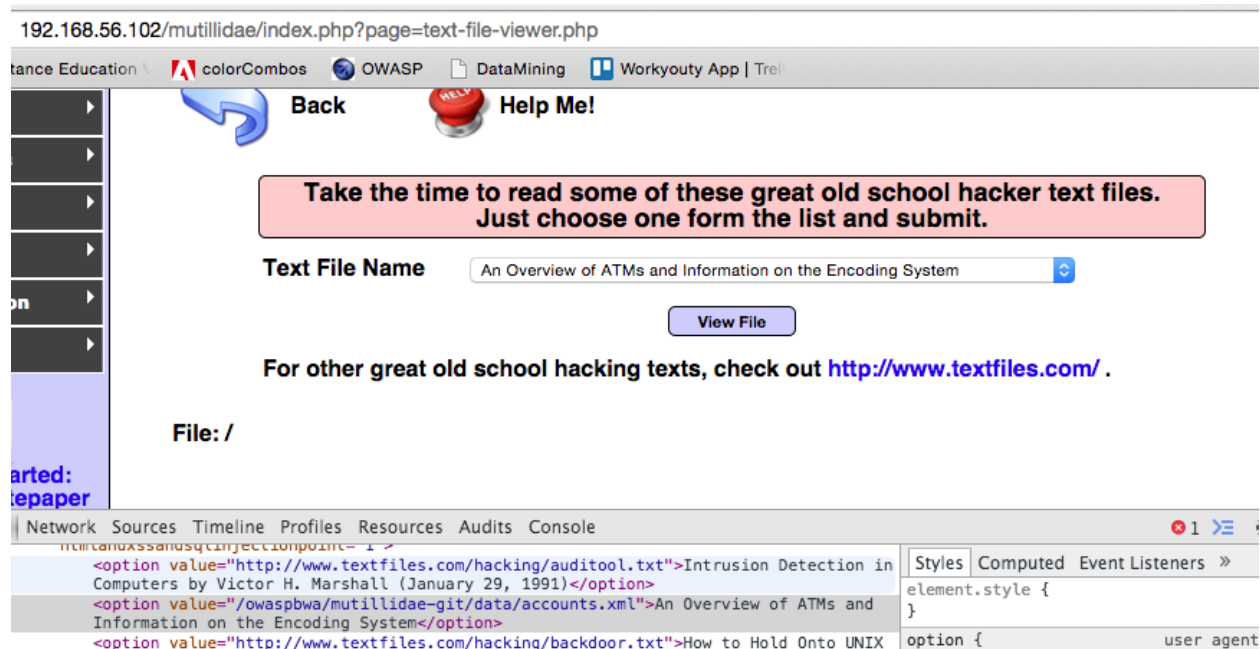
sources Timeline Profiles Resources Audits Console

xssandsqlinjectionpoint=1 title= inputs are usually a good place to start
for cross-site scripting, HTML injection and SQL injection">
:ion value="http://www.textfiles.com/hacking/auditool.txt">Intrusion Detection in
uters by Victor H. Marshall (January 29, 1991)</option>
:ion value="http://www.textfiles.com/hacking/atms">An Overview of ATMs and
ormation on the Encoding System</option>
:ion value="http://www.textfiles.com/hacking/backdoor.txt">How to Hold Onto UNIX

Styles Computed Event Listener

element.style {
}
option { use
font-weight: normal;

Below is what a potential attack would look like on this field. Notice how I am able to directly reference the file path to the xml file holding all of the account information. So that this document can be shared, we have excluded the image of the output of this attack as it displays confidential information.



15.)

Vulnerability 15 can be found on the `set-background-color.php` page and results from not validating the input and not encoding the output. This HTML Injection vulnerability is on the color input field that is expecting a 6 hex-character input to be used as a color value. This input could be validated with regex to be only values from 0-9 and A-F. An example of a regex string that would achieve this would be `^[0-9A-Fa-f]{6}$`. This would prevent any html code from being added to the input field that gets displayed in html code to the web page after submitting the form. Another fix would be to encode the output similarly to some of the above fixes by using Owasp's Encoder class. This value also gets used in a css attribute and needs to be encoded for CSS when being used there. Again, this can be done using Owasp's Encoder class. Here is where in the code this needs to be fixed for the html encoding.

```
<tr>
    <td ReflectedXSSExecutionPoint="1" class="informative-me
sage" colspan="2" style="text-align: center;">
        The current background color is <?php echo $_Bac
groundColorText; ?>
    </td>
</tr>
```

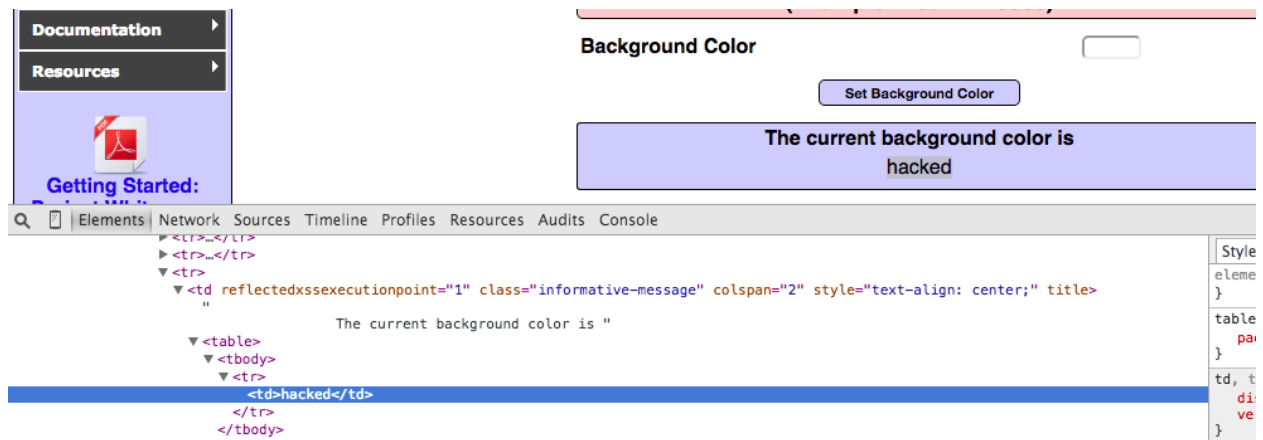
Here is what needs to be fixed for the CSS encoding.


```
<form
    action="index.php?page=set-background-color.php"
    method="post"
    enctype="application/x-www-form-urlencoded"
    style="background-color:#<?php echo $lBackgroundColor; ?>"
>
```

Here is an example of a potential attack from this vulnerability.

```
<table><tr><td>hacked</td></tr></table>
```

Here is what results from that attack. Notice how the html was interpreted as html code rather than a string.



16.) This vulnerability will be handled very similarly to the 14th vulnerability. If we provide aliases for the document names to be stored in the option tags' values then an attacker won't be able to specify a particular file to print to the screen. The image below is where in the code this needs to be fixed. Notice how not only are the filenames in the value field, but the relative path is there as well. Not only does this give an attacker the name, but it also gives he/she a reference point for where in the directory this file is located.

```

        <td style="text-align:left;">
            <input name="PathToDocument" id="id_path
_to_document" type="radio" value="documentation/change-log.html" checked="checked" />&nbsp;&nbsp;&nbsp;Change Log<br />
            <input name="PathToDocument" id="id_path
_to_document" type="radio" value="robots.txt" checked="checked" />&nbsp;&nbsp;&nbsp;Robots.txt<br />
            <input name="PathToDocument" id="id_path
_to_document" type="radio" value="documentation/mutillidae-installation-on-xampp-win7.pdf" checked="checked" />&nbsp;&nbsp;&nbsp;Installation Instructions: Windows 7 (PDF)<br />
            <input name="PathToDocument" id="id_path
_to_document" type="radio" value="documentation/how-to-access-Mutillidae-over-Virtual-Box-network.php" checked="checked" />&nbsp;&nbsp;&nbsp;How to access Mutillidae over Virtual-Box-network<br />

```

Here is an example of an attack that exploits this vulnerability. We excluded the results of this attack as it prints out all of the users' account information to the screen.

The screenshot shows a web application interface with a sidebar on the left containing 'Web Services', 'HTML 5', 'Others', 'Documentation', and 'Resources'. The main content area is titled 'Document Viewer' and contains a blue box with the text 'Please Choose Document to View'. Below this box is a list of radio buttons with the following labels: 'Change Log', 'Robots.txt', 'Installation Instructions: Windows 7 (PDF)', and 'How to access Mutillidae over Virtual-Box-network'. The 'How to access Mutillidae over Virtual-Box-network' option is selected. A 'View Document' button is located at the bottom right of the document viewer area.

The browser's developer tools are open at the bottom, showing the 'Elements' panel. The selected element is a table with the following structure:

```

<table style="margin-left:auto; margin-right:auto;">
  <tbody>
    <tr id="id-bad-path-to-document-tr" style="display: none;">
    <tr>
    <tr>
    <tr>
    <td style="text-align:left;">
      <input name="PathToDocument" id="id_path_to_document" type="radio" value="data/accounts.xml" checked="checked">
      &nbsp;&nbsp;&nbsp;Change Log
    <input name="PathToDocument" id="id_path_to_document" type="radio" value="robots.txt" checked="checked">
    &nbsp;&nbsp;&nbsp;Robots.txt
    <input name="PathToDocument" id="id_path_to_document" type="radio" value="documentation/mutillidae-installation-on-xampp-win7.pdf" checked="checked">
    &nbsp;&nbsp;&nbsp;Installation Instructions: Windows 7 (PDF)
    <input name="PathToDocument" id="id_path_to_document" type="radio" value="documentation/how-to-access-Mutillidae-over-Virtual-Box-network.php" checked="checked">
    &nbsp;&nbsp;&nbsp;How to access Mutillidae over Virtual-Box-network
  </td>
  </tr>
  </tbody>
</table>

```

The 'Styles' panel on the right shows the following styles for the selected element:

```

element.style {
}
input, textarea {
  border-radius: 5px;
}
input[type="radio" i] {
  -webkit-appearance: radio;
  box-sizing: border-box;
}
input[type="radio" i],
input[type="checkbox" i] {
  margin: 3px 0.5ex;
}

```

Risk Determination and Priority Recommendations

Here is a table that shows the likelihood of an attack being committed and the impact such an attack could have. The closer to the top left a vulnerability appears on this table, the less risk it has, and the closer to the bottom right, the more urgent a fix would be.

Uses Vulnerability Numbers		Impact		
		Low Impact	Moderate Impact	High Impact
Likelihood	Very Unlikely			
	Might Happen			
	Likely	15	3,4,5,6,7,8,9,10,11	14,16
	Definitely Will Happen			1,2,12,13

The vulnerabilities closer to the bottom right corner should be prioritized. Vulnerabilities 1,2,12, and 13 are very likely to be exploited as attackers will start with the login and registration pages. They are also going to want to exploit the page that shows a user's information because the authentication is the same as the login page. These are also High Impact because an attacker could manipulate the database which would warrant any sort of authentication within the site as worthless which would have a huge negative impact on the company. 14 and 16 were considered likely as they are somewhat harder to find. An attacker has to first find the page, and then play with it a little to discover the vulnerability. Again, it is still not difficult to find so it is listed as likely to be exploited. These exploits would have a very high impact as it would allow an attacker access to any file on the server which would be very bad for the company as any confidential file would be public access. Vulnerabilities 3-11 were deemed as likely because they are not super hard to find, but still require a basic knowledge of XSS and html injection to execute. Therefore, they are still likely to be exploited, but not guaranteed. They also have moderate impact, as mostly they deal with defacing the website which casts a very bad light on the company's reputation. However, the defaced pages are only viewed in specific scenarios (i.e. a particular user's blog entries are being viewed, or a particular user's signature is being shown), and compared to viewing the system's files they are low impact exploits. Vulnerability 15 had the lowest priority because it doesn't persist across users, meaning the color that is input is not stored anywhere to be displayed to other users, and it only allows the defacement of this one page. That being said, it is likely to happen so it should still be fixed.

Although some vulnerabilities should be prioritized over others, it is recommended that all of them be fixed. These vulnerabilities are all very simple to fix in the code, and are all at least likely to be exploited. This document can also be used as a reference later for how to fix

any similar vulnerabilities that are found, as well as how to test future code for these same exploits. This is not a comprehensive list, but it is a good starting point.