Security Guidelines for Plugin Authors

Creating plugins for DokuWiki is very easy even for novice PHP programmers. To make sure your plugin does not compromise the security of the whole wiki it is installed on, you should follow the guidelines outlined on this page.

 Improvement of this page is always welcome. It's in a very raw state and should be extended with more indepth info, links and examples.

Summary

A list of the most common security issues and how to avoid them can be found on this page. A short summary:

- Cross Site Scripting (XSS) – inserts malicious code into website to manipulate site in browser of user
- Cross Site Request Forgery (CSRF) – tricks to let you do unknowingly harmful actions on your site
- Remote Code Inclusion – includes code on server that's executed there
- Information leaks – there is too much information shown
- SQL injection – one can do unwanted requests on your data

Also there is added a note about reporting Security Issues.

Cross Site Scripting (XSS)

This is probably the most common vulnerability to be found in DokuWiki plugins.

Cross Site Scripting refers to an attack where malicious JavaScript code is introduced into a website. This can be used to redirect innocent users to malicious websites or to steal authentication cookies.

DokuWiki's plugin mechanism gives plugin developers a great deal of flexibility. In the case of syntax plugins in particular, the framework gives plugins the ability to work with raw unprocessed output. This means the wiki page data which reaches your plugin has not been processed at all. And there will be no further processing on the output after it leaves your plugin.

Escaping output

At an absolute minimum the plugin should ensure any raw data output has all HTML special characters converted to HTML entities using the htmlspecialchars() function. DokuWiki provides a convenient shortcut called hsc() for the function. URLs values should be escaped using rawurlencode().

Also any wiki data extracted and used internally (eg. user names) should be treated with suspicion.

Input checking

Check always all your input. Use whitelists, filters, conversions to the exact data type you mean e.g. from a number inputted as mixed php value to integer and more to ensure you have only data you allowed.

Please also refer to our chapter on processing request vars like _GET, _POST or _SERVER.

See also:

- Cross-site scripting
- XSS Cheat Sheet

Typical Vulnerability Examples

Below are some very common problems shown. The examples are very simple to make the general problem clear. Your plugin is probably more complicated, but you need to keep track of the same vulnerabilities.

Syntax Bodies

Many simple syntax plugins will take some user input and format it in custom HTML.

Example: Here is a abridged syntax plugin to wrap a given input in a bold tag.

```
class syntax_plugin_bold extends DokuWiki_Syntax_Plugin {
    // common plugin functions ommited

    public function connectTo($mode) {
        $this->Lexer->addSpecialPattern('!!!.*?!!!', $mode, 'plugin_bold');
    }

    public function handle($match, $state, $pos, Doku_Handler $handler){
        return [substring($match, 3, -3)];
    }

    public function render($format, Doku_Renderer $renderer, $data) {
        if($format != 'xhtml') return false;
        $renderer->doc .= '<b>' . $data[0] . '</b>';  // no escaping
    }
}
```

As you can see, the raw input data captured in the lexer pattern is just passed on to the render method, where no escaping at all is done. Malicious users could introduce what ever JavaScript and HTML code they want.

The fix is simple: proper escaping.

```
class syntax_plugin_bold extends DokuWiki_Syntax_Plugin {
    // common plugin functions ommited

    public function connectTo($mode) {
        $this->Lexer->addSpecialPattern('!!!.*?!!!', $mode, 'plugin_bold');
    }

    public function handle($match, $state, $pos, Doku_Handler $handler){
        return [substring($match, 3, -3)];
    }

    public function render($format, Doku_Renderer $renderer, $data) {
        if($format != 'xhtml') return false;
        $renderer->doc .= '<b>' . htmlspecialchars($data[0]) . '</b>';
//escaping
    }
}
```

Forms

When your plugin provides a form it is very common to validate the input and redisplay the form with the received user input when a validation error occurs.

Example: The following shows a form vulenerable to an XSS attack because it does not escape the user provided input correctly:

```
<form action="" method="post">
    <input type="text" name="q" value="<?php echo $_REQUEST['q']?>" />
    <input type="submit" />                    //no escaping
</form>
```

Providing `"><script>alert('bang')</script>` as user input would exploit the vulnerability.

To fix the form use the htmlspecialchars() or DokuWiki shortcut hsc() function:

```
<form action="" method="post">
    <input type="text" name="q" value="<?php echo hsc($_REQUEST['q'])?>" />
    <input type="submit" />                                          //escaping
</form>
```

In general it is recommended to not hand-craft forms, but use DokuWiki's form library.

Classes and other Attributes

Often plugins will accept multiple parameters and options that are used to modify the output of the plugin.

Imagine a plugin accepting the following input to display a message box:

```
<msg warning>Do not believe anything!</msg>
```

In the render method of this syntax there might be code like this:

```
$renderer->doc .= '<div class="msg_' . $class . '">'  //$class can be
everything
                        . htmlspecialchars($message)
                . '</div>';
```

As you can see the message itself is properly escaped, but the class is not. Instead of escaping it might be more sensible to use a whitelist of allowed classes instead with a default fallback:

```
$allowed = ['notice', 'info', 'warning', 'error'];   // whitelist
if(!in_array($class, $allowed){
    $class = 'notice'; // unknown input, fall back to a sane default
}
$renderer->doc .= '<div class="msg_' . $class . '">'
                        . htmlspecialchars($message)
                . '</div>';
```

input URLs

When a plugin accepts URLs as input you need to make sure, users can not pass the `javascript://` pseudo protocol.

Here is an example how a very simple check could look like, to make sure only http and https URLs are used.

```
// empty URL on protocol mismatch
if(!preg_match('/^https?:\/\//i', $url)) {
    $url = '';
}
```

Cross Site Request Forgery (CSRF)

This vulnerability often appears into plugins due to the lack of understanding of this issue, often confused with the XSS.

Cross Site Request Forgery refers to an attack where the victim's browser is tricked by a malicious site to ask for a page on a vulnerable site to do an unwanted action. The attack assumes the victim's browser has credentials to change something on the vulnerable site.

Adding security token

DokuWiki offers functions to help you deal against CSRF attacks. getSecurityToken() will create a

token that should be used to protect any authenticated action. It has to be included in links or forms triggering that action. All forms created with the form library will have security tokens added automatically, for handcrafted forms the formSecurityToken() function can be used.

It is your resposibility as the plugin author to actually check the token before executing authorized actions using the checkSecurityToken() function.

See also

- Cross Site Request Forgery
- OWASP explanation

Typical Vulnerability Example

Below is the simplest example to start. You may have a more complicated plugin of your own to secure, here is just a simple example based on form.

Imagine you want to know something which can be answered to Yes or No, you would have a form of this type:

```
<form action="" method="GET">
    <input type="radio" name="yn" value="Yes" />
    <input type="radio" name="yn" value="No" />
    <input type="submit" value="Answer" />
</form>
```

Then you process this form as follows:

```
global $INPUT;

if($INPUT->get->has('yn')){
    do_something_with_yn($INPUT->get->str('yn'));
}
```

So a user is connected to answer this question, but he doesn't know the response yet. Let's take time to think and browse the web... Now the user is visiting a malicious website, one which know, or not, that the user may be connected to your DokuWiki. In this website, the developer included this HTML image tag:

```
<img src="http://your.dokuwi.ki/formpage?yn=Yes" />
```

What will the user's browser do then?

The browser will process this image as any other and will send a request to this URL. Your plugin will then see that `$_GET['yn']` is set and will call the `do_something_with_yn()` function.

That's one of the examples of CSRF. Now, how to fix this security hole?

Prevent CSRF

Remember your form above? Let's add an input in it:

```
<form action="" method="GET">
    <input type="hidden" name="sectok" value="<?php getSecurityToken(); ?>"
/>
    <input type="radio" name="yn" value="Yes" />
    <input type="radio" name="yn" value="No" />
    <input type="submit" value="Answer" />
</form>
```

Do you see the first input? Yes? Good. Now you have to check the security token when you receive the form, before processing it:

```
global $INPUT;

if($INPUT->get->has('yn') && checkSecurityToken()) {
    do_something_with_yn($INPUT->get->str('yn'));
}
```

As the malicious website will never find the value of the «sectok» hidden input, your form is no longer vulnerable to CSRF.

Note: If the security token is not valid, the `checkSecurityToken()` function displays a message which informs the user.

Remote Code Inclusion

This attack allows an attacker to inject (PHP) code into your application. This may occur on including files, or using unsafe operations functions like eval() or system().

Always filter any input that will be used to load files or that is passed as an argument to external commands.

Information leaks

This attack may lead to the exposure of files that should usually be protected by DokuWiki's ACL or it might expose files on the server (like `/etc/passwd`).

Always filter any input that will be used to load files or that is passed as an argument to external commands.

Always use DokuWiki's ACL check functions when accessing page data.

SQL injection

This attack is rarely relevant in DokuWiki because no database is used. However if your plugin accesses a database always escape all values before using them in SQL statements.

More info:

- SQL injection

Reporting Security Issues

If you encounter an issue with a plugin please inform the author of the plugin via email, optionally putting Andi or the mailinglist on CC.

Additionally a `securityissue` field with a short description of the problem should be added to the data on the page of the plugin. This will create a red warning box and will delist the plugin from the main plugin list.

Once the issue was fixed and a new release was made, this field should be removed again.