Правила безопасности для авторов плагинов

Создание плагинов для DokuWiki очень просто даже для начинающих программистов PHP. Чтобы убедиться, что ваш плагин не ставит под угрозу безопасность всей вики, на которой он установлен, вам следует следовать рекомендациям, изложенным на этой странице.

 Улучшение этой страницы всегда приветствуется. Она находится в очень сыром состоянии и должна быть расширена более подробной информацией, ссылками и примерами.

Краткое содержание

Список наиболее распространенных проблем безопасности и способы их избежания можно найти на этой странице. Краткое резюме:

- Межсайтовый скриптинг (XSS) — вставляет вредоносный код на веб-сайт для манипулирования сайтом в браузере пользователя.
- Подделка межсайтовых запросов (CSRF) — уловки, позволяющие вам совершать неосознанные вредоносные действия на вашем сайте.
- Удаленное включение кода – включает код на сервере, который там выполняется.
- Утечка информации – отображается слишком много информации
- SQL-инъекция – можно выполнить нежелательные запросы к вашим данным

Также добавлено примечание о необходимости сообщать о проблемах безопасности .

Межсайтовый скриптинг (XSS)

Это, вероятно, самая распространенная уязвимость, встречающаяся в плагинах DokuWiki.

Cross Site Scripting относится к атаке, при которой вредоносный код JavaScript внедряется на веб-сайт. Это может использоваться для перенаправления невинных пользователей на вредоносные веб-сайты или для кражи аутентификационных cookie-файлов.

Механизм плагинов DokuWiki дает разработчикам плагинов большую гибкость. В случае с плагинами синтаксиса, в частности, фреймворк дает плагинам возможность работать с сырым необработанным выводом. Это означает, что данные страницы вики, которые достигают вашего плагина, вообще не были обработаны. И не будет никакой дальнейшей обработки вывода после того, как он покинет ваш плагин.

Выход из режима экранирования

Как минимум, плагин должен гарантировать, что все выходные необработанные данные будут содержать все специальные символы HTML , преобразованные в сущности HTML с помощью

функции htmlspecialchars(). DokuWiki предоставляет удобный ярлык hsc() для этой функции. Значения URL-адресов следует экранировать с помощью rawurlencode().

Кроме того, следует с подозрением относиться к любым данным вики, извлеченным и используемым внутри компании (например, именам пользователей).

Проверка входных данных

Всегда проверяйте все ваши входные данные. Используйте белые списки, фильтры, преобразования в точный тип данных, который вы имеете в виду, например, из числа, введенного как смешанное значение php, в целое число и т. д., чтобы убедиться, что у вас есть только разрешенные вами данные.

Также ознакомьтесь с нашей главой об обработке переменных запросов, таких как _GET, _POST или _SERVER.

Смотрите также:

- Межсайтовый скриптинг
- Fix Me! Шпаргалка по XSS

Типичные примеры уязвимостей

Ниже показаны некоторые очень распространенные проблемы. Примеры очень просты, чтобы сделать общую проблему понятной. Ваш плагин, вероятно, сложнее, но вам нужно отслеживать те же уязвимости.

Синтаксис Тела

Многие простые плагины синтаксиса принимают часть введенных пользователем данных и форматируют их в виде пользовательского HTML .

Пример: Вот плагин сокращенного синтаксиса, позволяющий выделить заданный ввод жирным шрифтом.

```
class syntax_plugin_bold extends DokuWiki_Syntax_Plugin {
    // общие функции плагина опущены

    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>';  // без экранирования
    }
}
```

Как вы можете видеть, необработанные входные данные, захваченные в шаблоне лексера, просто передаются в метод рендеринга, где экранирование вообще не выполняется. Злонамеренные пользователи могут вводить любой код JavaScript и HTML , который они хотят.

Решение простое: правильный побег.

```
class syntax_plugin_bold extends DokuWiki_Syntax_Plugin {
    // общие функции плагина опущены

    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>';
//экранирование
    }
}
```

Формы

Когда ваш плагин предоставляет форму, очень часто требуется проверить вводимые данные и повторно отобразить форму с полученными данными пользователя в случае возникновения ошибки проверки.

Пример: ниже показана форма, уязвимая для атаки XSS, поскольку она не экранирует правильно введенные пользователем данные:

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

Предоставление данных `"><script>alert('bang')</script>` в качестве входных данных пользователя приведет к эксплуатации уязвимости.

Для исправления формы используйте функцию htmlspecialchars() или функцию DokuWiki shortcut hsc():

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

В целом рекомендуется не создавать формы вручную, а использовать библиотеку форм DokuWiki .

Классы и другие атрибуты

Часто плагины принимают несколько параметров и опций, которые используются для изменения выходных данных плагина.

Представьте себе плагин, принимающий следующие входные данные для отображения окна сообщения:

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

В методе рендеринга этого синтаксиса может быть такой код:

```
$renderer->doc .= '<div class="msg_' . $class . '">'  //$class может быть чем угодно
                        . htmlspecialchars($message)
                . '</div>';
```

Как вы видите, само сообщение правильно экранировано, но класс — нет. Вместо экранирования может быть разумнее использовать белый список разрешенных классов с резервным вариантом по умолчанию::

```
$allowed = ['notice', 'info', 'warning', 'error'];    // белый список
if(!in_array($class, $allowed){
    $class = 'notice'; // неизвестный ввод, вернуться к разумному значению по умолчанию
}
$renderer->doc .= '<div class="msg_' . $class . '">'
                        . htmlspecialchars($message)
                . '</div>';
```

входные URL-адреса

Когда плагин принимает URL-адреса в качестве входных данных, необходимо убедиться, что пользователи не смогут передать `javascript://` псевдо-протокол.

Вот пример того, как может выглядеть очень простая проверка, позволяющая убедиться, что используются только URL-адреса http и https.

```
// пустой URL при несоответствии протокола
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.

From:
https://wwoss.ru/ - **worldwide open-source software**

Permanent link:
https://wwoss.ru/doku.php?id=wiki:devel:security&rev=1735914485

Last update: **2025/01/03 17:28**