Security of Community Developed and 3rd-party Wiki Plug-ins

Andy Webber
Oracle Corporation
Thames Valley Park
Reading, RG6 1RA
United Kingdom
andy.webber@oracle.com

ABSTRACT

This paper discusses the significant security vulnerabilities that can occur in community developed wiki plug-ins and issues associated with managing the process of remediation. General guidance is given on how the vulnerabilities can be detected and rectified.

The basis for the paper is direct experience with a number of community developed plug-ins for DokuWiki, although the findings have also been transferred to other wikis such as MediaWiki[1]. The findings are also transferable to other similar web server technologies - such as blogs - that support similar plug-in frameworks[2].

Keywords

plug-in, cross site scripting, security, responsible disclosure

1. INTRODUCTION

Many wikis provide a plug-in framework that allows for the installation of extensions to enhance the functionality provided by the wiki. The frameworks are normally documented to allow 3rd parties and the user communities to develop plug-ins to provide the functionality that they desire. To provide the maximum flexibility, the framework will often use the generic underlying programming language and give the plug-in developer full control over the rendered output and the internal state of the wiki.

With this flexibility comes an inherent risk: that the plug-in author has failed to consider the undesirable effects that such a large degree of freedom may bring and has therefore not mitigated the concomitant security risks.

[1]Of the approx. one thousand MediaWiki Extensions, a targetted sample of three were examined and two were found to be vulnerable.

[2]Although not specifically examined in this paper, blogs have had similar vulnerabilities in plug-ins, such as Geeklog [7] and Serendipity [15].

A characteristic of community developed plug-ins is that the plug-in developer may be familiar with what they want to achieve, but unfamiliar with the particular programming language they have to use and with the plug-in framework. They may also be extremely optimistic about the input that their plug-in will receive (*ie* they may only consider valid inputs) and be unaware of what an attacker might try to do and how she might do it.

This paper examines some of the more common classes of vulnerabilities that may be introduced, strategies for avoiding them and also some of the procedural aspects of managing the process of notifying plug-ins users.

The remainder of this paper uses DokuWiki[3] as a worked example. This is the wiki with which the authors are most familiar and the one which they had cause to review as a result of a practical deployment. A brief examination of the plug-in frameworks for other wikis suggests that they are likely to have similar issues although the specifics may vary. It is also likely that other web based applications that support a plug-in framework to allow more flexibility in presenting user-contributed content will also have similar issues (for example, bulletin boards). However, these are not examined in this paper.

DokuWiki's intention is to provide a small core wiki that is easy to deploy and use, and to provide a flexible framework for plug-ins to extend this core functionality. Although the plug-in framework allows for a lot of flexibility, it is easy to write plug-ins (or adapt existing ones) for a wide range of additional features. DokuWiki also provides a centralised register of plug-ins from which to choose (whether to use "as is" or to adapt). A detailed tutorial on writing plug-ins, which includes a section on security, is available [4]. DokuWiki's register currently has over 340 plug-ins[4] - a testament to the ease with which they can be developed and the variety of things that users want to do with a wiki. With the wide range of plug-ins available for wikis, they are becoming a framework for a variety of *ad hoc* applications in place of more specialist web based applications. It is therefore increasingly likely that any particular instance of a wiki will include a number of plug-ins, and hence more likely to include a vulnerable plug-in.

With such a large number of plug-ins there is likely to be a wide range of programming skills represented by the developers. Major software corporations have trouble recruiting graduates, even Computer Science majors from prestigious

[3]http://wiki.splitbrain.org/wiki:dokuwiki
[4]http://wiki.splitbrain.org/wiki:plugins

universities, that have an appreciation of how to code securely [10]. Expecting high standards from voluntary contributions from a wide variety of backgrounds is unrealistic. With appropriate guidance, however, we believe that the potential risks in wiki plug-ins can be significantly reduced. This paper doesn't aim to teach "secure coding 101", but does aim to highlight the issues specific to wiki plug-ins and HTML/xHTML rendered output.

Note that the emphasis of this paper is on the community developed plug-ins rather than the core wiki software. These issues are not, however, constrained to community developed plug-ins (verses commercial plug-ins or plug-ins provided by wiki developers). Furthermore, most of the discussion is not, in fact, constrained to plug-ins and could apply equally to the core software by extrapolation.

There is no intention in this paper to compare the risk of using any one wiki over another, or of using any plug-in over another. However, this paper may help the reader to make more informed comparisons themselves when evaluating software.

2. HOW PLUG-INS WORK

DokuWiki supports various types of plug-in that allow the wiki to be extended without the need to modify the core code. These include: Syntax; Helper; Admin; Action and Render. The most common form is Syntax and this is where we have seen most security issues, so we will concentrate on this subset. It must be remembered, however, that the issues could also apply to other plug-in types as well.

In the case of syntax plug-ins, they register themselves with the wiki parser using a regular expression (regex [13]) that represents the syntax that the plug-in expects. For example, the "color" plug-in [3] allows a document to embed a change in font colour, a feature otherwise not present in DokuWiki. An example of the extended wiki markup is:

```
<color red>this text will be in red</color>
```

The plug-in registers itself with the parser using a PHP regex. This only needs to be specific enough to invoke the plug-in. The plug-in itself can later decide what action to take once it has been invoked. However, the regex should be specific enough that the plug-in is not invoked where it is not required. The regex for this example is:

```
<color.*?>(?=.*?</color>)
```

It can be seen that this regex initially allows any data to be included between `<color` and `>` and then also between that `>` and the closing `</color>`. We will see later that the plug-in must subsequently be less liberal in what it includes in the rendered output.

An example of the rendered xHTML output that the color plug-in generates given the above input is:

```
<span style='red'>this text will be in red</span>
```

It can be seen that the processing required to convert the markup to the rendered output is very simple. However, unless consideration is given to the risks, it is also very simple to create a plug-in that allows an attacker to inject HTML of her choosing.

3. CROSS SITE SCRIPTING

The most common error that we encountered was HTML injection, most commonly exploited as Cross Site Scripting (XSS) [1] or IFRAME injection [12]. This was most often found to be of the "persistent" variety.

XSS is often underestimated as a form of attack as it does not appear to be all that severe. In practice, however, the ability of an attacker to run arbitrary JavaScript in a user's browser is tantamount to being able to run arbitrary code on their workstation because JavaScript has a weak security policy [18]. This is not a condition that is likely to change any time soon, not least because of the number of websites and browsers that would need to be obsoleted. Regardless of issues in JavaScript there are still many attacks that would be possible based on browser's relatively weak security policies without using JavaScript.

Although a whole collection of attacks is possible with HTML injection, we will, for the sake of simplicity, concentrate on XSS. Fixing HTML injection will, in most cases, fix all the issues, so in the resolution section we will concentrate on fixing HTML injection.

The problem is most severe on wikis which allow untrusted users (whether anonymous, self-registered or just not altogether trustworthy) to edit wiki pages where arbitrary HTML/ JavaScript is not intended to be permitted in the pages. In these cases an untrusted user that can successfully exploit a vulnerable plug-in is able to defeat the constraints normally imposed on them by the wiki.

The issue is prevalent[5] because, in normal use, the plug-in is likely to operate correctly. A plug-in developer who is mainly interested in ensuring that his plug-in works would never need to consider using special filtering/encoding routines as they serve no purpose to most legitimate use cases. It is only when legitimate use cases happen to also use some of the special characters that the developer is forced to address the issue for the sake of correct functioning. In more than one case we have observed, a user reported an issue whilst trying to make legitimate use of a character such as `<` or `"` in their data. We were able to expand the apparently niggling functionality issue to an XSS attack. In the process of helping the developer address the XSS vulnerability the end user's bug has also been fixed.

Let's look at a concrete example (for a moment) based on the color plug-in described above. In the absence of any other processing by the plug-in, an attacker can attempt to attack two parts of this plug-in, the colour and the text. There are a few limitations imposed by the regex used: the colour can not contain `>` and the text can not contain `</color>`.

If we assume for the moment that there is no other checking, filtering or encoding then an attacker can insert:

```
<color red' onmouseover="alert('XSS attack')"
 alt='>Hurrah!</color>
```

and the rendered output is likely to look like this:

```
<span style='red' onmouseover="alert('XSS attack')"
 alt=''>Hurrah!</span>
```

As a consequence, when a user browsing the page hovers their pointing device over the (red) text "Hurrah!", the "onmouseover" event is triggered and JavaScript is run. In this case, an alert dialogue box will be displayed indicating that the attack was successful.

[5]DokuWiki, at the time of writing, has around 340 contributed plug-ins on the register. Of these, code has been reviewed for about 75 with the sample being selected on a partly alphabetic and partly risk basis; 20 were identified as vulnerable.

In this section we will discuss the insertion of JavaScript[6]. However, more generally, it is usually the case that an attacker can also insert HTML/xHTML in web based renderings. They can, therefore, insert anything - including objects. The source for these objects (and indeed the JavaScript) may be elsewhere on the Internet. Thus, combined with other platform and/or browser bugs it may be possible to completely compromise the platform of anyone that views an affected page. There is a current and increasing trend for miscreants to seek out websites that can be infected with JavaScript in order to recruit individual PCs into 'botnets' [14][7].

We identified cases that fall into the following categories:

- the issue of an attacker trying to gain some advantage via the plug-in had simply not been considered, so no attempt had been made to prevent such attacks;
- a deliberate intention to allow the page editor a lot of flexibility, on which, by design, there were few if any constraints, making an attack trivial;
- filtering and validation was in place to attempt to defeat attacks but flaws in the mechanism meant that attacks were still possible;
- in the case of MediaWiki extensions, although an API exists with automatic encoding, the extension developer had chosen not to use it.

In addition there were cases where, by chance rather than by design, a "perfect" attack was not possible (*ie* an attack where the resulting HTML/xHTML was not well-formed). However, it is not always necessary to create a perfect attack since web browsers are very tolerant of invalid markup (which is just as well given the prevalence of invalid web pages). Thus the attack only needs to be "good enough" to have the browser "fix" the markup and complete the attack. This then is the final category where an attack shouldn't be possible because it cannot generate (for whatever reason) correct HTML/xHTML, but does in practice work.

For the moment, we will consider a plug-in as a piece of code that takes some input data, processes it in some way and uses that data in constructing some HTML/xHTML markup as output[8]. The source of the data could include, for example:

- from parameters to the plug-in;
- from the wiki page from which it was invoked;
- from elsewhere in the wiki;
- from the file system or an external program;
- from an external resource such as a SOAP response

In the following discussion we will use the terms "tag", "data", "attribute", "value" and "uri" to represent either static values from the plug-in or input, however it has been filtered, split, combined or transformed. A typical plug-in will generate HTML/xHTML markup as output in some combination of any number of the following forms:

1. `<tag>data</tag>`
2. `<tag attribute="value"/>`
3. `<tag src="uri/url"/>` [9]

Depending on which form is being used determines which characters must be prevented from being emitted. A successful attack involves getting characters to be emitted by the plug-in that cause the browser to parse the page in a way that is to an attacker's advantage. For XSS this is done in a way that gets JavaScript executed.

Note that the markup generated by an attack does not have to be perfect/DTD compliant. It only needs to be good enough for the browser to interpret it as the attacker intended. We have identified plug-ins where it is not possible to get the plug-in to emit certain characters that ought to be required for an attack to work, but where an attack was still successful because the browser made its best efforts to interpret the invalid xHTML.

The following guidance applies to each of the corresponding cases above:

1. It is rare, and dangerous, for a plug-in to allow users to insert arbitrary tags. "tag" **MUST** be validated against a white-list (see 3.1.1 below) of allowed tags. Ideally, "tag" **SHOULD** be generated by the plug-in rather than passed through from the user input. This makes it easier to be confident that the "tag" emitted is compliant with the standards for the rendered output (*eg* xHTML is case sensitive).

   "data" **MUST** be HTML filtered/encoded (see 3.1.2 below) - in particular, `<` to `&lt;`, `>` to `&gt;`, `"` to `&quot;`.

2. As with "tag" above, it is rare and dangerous for a plug-in to allow users to insert arbitrary attributes. "attribute" **MUST** be validated against a white-list (see 3.1.1 below) of allowed attributes. "attribute" **SHOULD** be generated by the plug-in rather than passed through from the user input. Particular care is required to ensure that intrinsic events [16] are not allowed.

   "value" **MUST** be HTML filtered/encoded. In addition it **SHOULD** have the single-quote character `'` encoded, for example as `&#039;`. "value" **SHOULD** be parsed, where possible, to allow only a minimal valid range of values. Validation can be troublesome whilst maintaining flexibility - for example a `color` attribute can be represented as a name such as `red` or as an sRGB value such as `#FF0000`. In some cases, an acceptable substitute for encoding is a filtering mechanism that ensures that no special characters are present. For

[6]We will use the common term "JavaScript" to refer to any browser/client-side scripting similar to ECMAScript, including Microsoft's JScript [9]

[7]Although the attack described in the reference uses an SQL injection to insert an IFRAME, it would be equally feasible to use a persistent XSS attack

[8]In this paper we only consider HTML/xHTML rendering as the output. However, a wiki might support the generation of other forms of markup. In this case, the nature of the attack is essentially the same, although the characters that an attacker is interested in injecting may differ, and the approach to filtering and escaping will differ accordingly.

[9]Note that although we have used the term "src" in this example, it applies to any attribute value that takes a URI/URL.

example, "height" and "width" attributes of images are, by default, in pixels; if the user supplied data has been passed through a routine that will only allow integers and no other characters then it *may* be the case that such a routine can be considered an adequate filter [10].

3. URLs and URIs require special handling. URLs/URIs **MUST** be URL encoded and then **MUST** also be HTML encoded (since the `&` character must to be encoded) [17]. In the general case, filtering is unlikely to be sufficiently flexible so encoding is recommended.

It must be remembered that the data that needs to be filtered/encoded could come from anywere. A plug-in that uses a SOAP request in order to obtain the URL of an image to embed in a rendered page must be aware of the fact that the URL that is returned from the SOAP interface will not necessarily be well-formed and could in fact represent a source of attack. Equally, the data could originate from the wiki core; for example, a plug-in may embed metadata such as the username of the last editor of a page. Such a plug-in needs to either be certain that the username has already been filtered/encoded appropriately, or perform this duty itself.

3.1 Filtering and Encoding Strategies

The keys to resolving XSS issues are filtering, encoding and quoting, so it is appropriate to dwell on the topic and highlight some of the pitfalls for the unwary.

Where the core software provides an API to perform filtering or encoding it is strongly advised that plug-in developers use the API provided. Such APIs are most likely to be the easiest way to avoid the most common issues.

3.1.1 Filtering

Filtering is a process of removing what might be called "contamination" from the original data, data where it is undesirable to allow it to be emitted. The approach to filtering may be further subdivided:

- Black-list: remove selected parts of the data.
- White-list: only allow selected parts of the data to pass.

There may appear to be no significant difference in these approaches as they should both achieve the same end. However, it is generally far safer to use the white-list approach. By using a white-list, you can be confident that the output data *only* includes the characters that have been expressly allowed (or the data is rejected if it contains anything other than the characters that have been expressly allowed).

By using a black-list there is a danger that some value, previously not known to be dangerous, could be allowed[11]. With a black-list an attacker may be able to evade the filter by using, for example, character encoding, a change of case or a Unicode regional variation of the character. Finally, a black-list may be ineffective because the program that ultimately receives the output (*eg* a web browser) may have extended the interface specification. For example a browser may support more "intrinsic events" than are specified in the W3C specifications. Thus an attacker may be able to generate an attack that uses the extended events that gets past the plug-in's black-list.

In contrast, a white-list will only allow that which has been explicitly allowed. If the white-list does not allow character encodings, then an attack using them will not be successful. If an attack uses extended intrinsic events that are not in the white list then it too will be unsuccessful.

Whilst we recommend the use of white-lists where possible, this strategy does still require some care, in particular where regular expressions are used to represent the valid data. The greatest problem arises when a test is made to ensure that the data contains a valid pattern, say `[a-z]*`. If the test is just "does the input data contain this pattern", then the test is not sufficient. The data may well contain that pattern, but may also contain undesirable characters. The test must therefore be, "does the input **only** contain this pattern". To achieve this, it may be necessary to anchor the expression to both the beginning and the end of the input data, for example `^[a-z]*$`. An alternative is to reverse the logic of the test and to reject the input if it contains anything other than the permitted characters, for example reject if this pattern matches: `[^a-z]`.

We identified two DokuWiki plug-ins (one derived from the other) that were vulnerable because the regular expression was not anchored to the end of the input. It was therefore possible to append the attack characters after a string that passed the validation and create a successful attack. The fix in both cases was to anchor the regex to the end of the input.

The "safe subset" of characters for HTML depends on the exact context in which they are emitted. As noted in section 3 above, because the data may be emitted inbetween `<` and `>`, filtering these characters will not necessarily solve the problem as they do not need to be part of the attack.

3.1.2 Encoding and Quoting

In this section we will discuss both encoding and quoting which are closely related. Quoting is the use of special characters in order to indicate that the quoted characters are not to be treated as special in the grammar in the way that they normally would. An example is the PHP programming language where various characters are special, for example the quote character inside a string would be confused with the quote character at the end of the string. In PHP, special characters may be quoted by preceding them with the baskslash so that they are treated as any other character, for example `\"`.

Another mechanism that is used is encoding. Encoding uses a different representation for the character. In the case of HTML, for example, the angle brackets `<` and `>` must be encoded as `&lt;` and `&gt;` so as not to be confused with markup. In URLs, the space character is special in that it is forbidden. If the space character is required in a URL then it it must be encoded for example as `+` or `%20`.

Note that encoding is not required solely to thwart potential attackers but it is also required in order for a plug-in to

[10] Care is still required in using such routines; it may still be possible, in some circumstances, for an attacker to achieve her goals if the output of these routines can usefully be affected by other inputs from the environment (as opposed to via formal parameters). "Lateral SQL Injection" is an attack that leverages this principle [21].

[11] This is similar to the approach of most anti-virus products at present - they block things that they know to be bad, but are playing a never-ending game of catch-up with the latest threat or the latest variant of a known threat.

operate across a wide variety of input data. For example, a plug-in might insert an Adobe Flash object onto a page allowing the page editor to define the "alternate text" attribute This text is typically displayed if the object cannot be rendered or when the pointing device is hovered over the object. The rendered output is likely to look like this:

```
<object src="..." alt="..."/>
```

What can be done if the user supplies an alternate text that contains the quote character? If no action is taken then the data will be rendered incorrectly and an injection will be possible. If the text is filtered for the quote character then the display will not represent what the editor intended. Encoding is the most satisfactory option, in this case as `&quot;`.

3.2 Other Similar Attacks

This section briefly outlines how attacks may also be possible in other contexts, depending on what the plug-in seeks to achieve. Some of these are speculative since we have not yet seen practical examples.

As noted in the preceding section, an attack is possible where the attacker is able to introduce special characters into the output markup without the appropriate encoding or quoting. The section above concentrates on HTML/xHTML as the output markup, but it could be in any grammar.

3.2.1 HTML/XML Injection

As noted at the beginning of section 3 above, XSS is really just a specific example of a more general issue of HTML injection. A user may have a browser that does not support JavaScript or has it temporarily disabled, but using the wider form of attack they can still be vulnerable. For example an attacker may combine a plug-in vulnerability with a vulnerability in other software on the user's machine - such as a buffer overflow in a music player - and inject an object to the wiki page that exploits the music player vulnerability. This is merely mentioned for completeness since the strategies mentioned above should mitigate this threat at the same time as mitigating XSS.

Similarly, a plug-in may generate another related form of structured output such as XML (possibly as part of a SOAP request). Again, the strategies already mentioned should mitigate most attacks, but a general case of an XML document includes more special characters than are normally considered for xHTML so additional care is required.

3.2.2 HTTP Splitting/Injection

For a plug-in that manipulates the HTTP headers of a generated page, the grammar is HTTP, the special characters are different (for example, line feed and carriage return become significant as end of line markers), and an attack could be HTTP header splitting or header injection [20].

3.2.3 Shell Injection

If a plug-in will cause the execution of an external program on the host then the grammar to attack may be the command line that is used to launch the external program. If this passes through a shell interpreter such as sh/csh/ksh then the grammar is rich and complex with a vast range of opportunity for the attacker to cause additional programs to be run, to gain access to the underlying file system. This is likely to give an attacker privileges the same as those the web server runs with[12]. An attacker might also try to use a shell injection to exploit a flaw in the software being shelled out to - for example buffer overflows - in order to get her own code running on the wiki host server.

3.2.4 Log Injection

For a plug-in that creates a log, such as a web access log, the input data might include the HTTP headers of the request. However, the HTTP headers are under the control of the attacker, so the plug-in developer needs to be careful. If the plug-in logs the "Referer:" or the "User-agent:" fields from the request into the logs then the grammar that the attacker may want to injection into is the log format. For example, the NCSA log format [11] has space separated fields, and any field that may contain a space is enclosed in quotes. In addition, if a field contains the quote character then this is in turn quoted using a backslash. Since the user agent field is likely to contain spaces, the plug-in may arbitrarily enclose that field in quotes. However, unless the plug-in backslash-quotes any quote characters inside the field then an attacker may be able to inject log entries of their choosing.

In addition to the above, if the plug-in uses an SQL database to store log records then the attacker may have the potential to inject into the SQL grammar to perform a SQL injection if the plug-in developer has not used an database appropriate API.

4. FINDING VULNERABILITIES

We will briefly describe the strategy that we used when finding the vulnerabilities in plug-ins. This is, of course, not the only strategy, but it did prove effective for the DokuWiki and MediaWiki plug-ins/extensions that we reviewed.

Essentially two strategies were used in parallel, each complementing the other:

- code review;
- testing;

This could be called "grey-box" testing if it were quite so formalised. In reality we initially tested a few plug-ins that we were interested in using ourselves on a deployment. Having identified flaws, we reviewed the code to better understand why the flaws existed and then refined our test to demonstrate an exploit.

Subsequently on other plug-ins, having identified the characteristics of vulnerable and in-vulnerable plug-ins, we reviewed the code first to short-list test candidates. Ultimately, we wanted to have a working test case in order to include it in the advisory (see section 5 below). In a very few cases, we were unable to get the plug-in working immediately (due to dependencies on or conflicts with other software) but were sufficiently convinced from the code review that we issued an advisory anyway.

The strategy to the code review was:

- identify the constraints on the input pattern - is it flexible enough that an attack is likely to be possible?

[12]The exact privileges obtained in the filesystem will vary by wiki, webserver, configuration of the execution environment and operating system

- identify the encoding/filtering on the outputs - what characters are likely to be able to be emitted in an attack?
- identify the intermediate processing - is this likely to help or hinder an attack?

In many cases there was clear input filtering or output encoding, making it easy to eliminate a plug-in from the candidate list to test. Whilst such plug-ins may still have vulnerabilities, our aim was not to find *all* vulnerabilities, or to look at all plug-ins. Where the code review was inconclusive, we sometimes performed testing to determine the extent to which attacks may be possible.

The approach used to generate test patterns was based on a targetted version of fuzzing [2]. In particular, care was taken to reduce the chances that test data was rejected by input validation. Ensuring that the test patterns passed the first level of validation significantly reduced the effective test space and improved testing efficiency. For example, care was taken not to include too many test characters in each test case since any one character could cause input validation to reject the whole attack (for example on the plug-in registration regex). An automated fuzzing test may well have included a complex test case featuring several problematic characters, any one of which could have resulted in the plug-in rejecting the input as invalid but without allowing a successful attack.

Extensive use was made of the HTML Validator Firefox add-on [5] to review the generated xHTML for evidence that our attack may have been at least partially successful. As noted previously, there is no need to generate compliant xHTML in every case. In practice, the validator was most useful when a partial attack was attempted (for example inserting a single quote or a single angle bracket) as the validator would report the page as containing invalid markup, this was used as a sentinel that the plug-in may be vulnerable and that more extensive testing may generate a full attack.

5. REPORTING AND FIXING VULNERABILITIES

There is, at present, little concensus on what constitutes “responsible disclosure”. Various individuals and organisations have published guidance but none has achieved much traction. The International Organisation for Standards is the latest to join the fray [8] with a target publication date of the end of 2010. Some guidance is also available to Open Source project maintainers, for example in [19] Chapter 6.

In the absence of an established standard or DokuWiki published guidance, we took the approach of sending an advisory to the plug-in developer directly and copying the DokuWiki maintainer. We made no general public announcement of an issue being found or of it being fixed - although plug-in developers are, of course, free to do so if they wish. Essentially, as reporters of the issue, we have no desire to hold plug-in developers to ransom and did not want to draw public attention to a vulnerability until plug-in users had an opportunity to respond.

As tracking the number of issues identified became more troublesome both for us as reporters and for DokuWiki maintainers, the maintainers decided to institute a policy where plug-ins with known vulnerabilities are flagged as such in the plug-in register until they are fixed. This means that there are unlikely to be any new installations of the plug-in (without people at least being made aware that there is a risk). It also means that people that check for updates to plug-ins can be made aware that a plug-in is vulnerable. It is of course possible that an attacker then seeks to determine for themselves the nature of the vulnerability in order to exploit it.

As reporters, we are content to follow the procedure recommended by the DokuWiki maintainers. Although flagging the plug-ins as vulnerable when there is no fix may be considered tantamount to public disclosure, in our experience, plug-in developers responded with fixes either very quickly (in one case in around 25 minutes from advisory to fix uploaded) or not at all. In some cases, developers wanted to take some time (a few weeks) to understand the issue properly to ensure an adequate fix and/or to address several plug-ins at once.

When sending an advisory, we typically include the following information:

- identification of the plug-in and version,
- a summary of the issue (for example, that it is vulnerable to persistent cross site scripting, with a link to definitions),
- an exploit example (wiki markup and corresponding rendering) for illustration and testing,
- an analysis of why the exploit was possible (for example the lack of encoding before rendering),
- a summary of the significance of the issue,
- references to appropriate material on how to fix the issue,
- a suggested procedure to follow, and
- information about the extent to which we intended to publicise the vulnerability discovery (aka “disclosure”) [in our case, we had no plans to issue a disclosure for any of the vulnerabilities].

The suggested procedure to follow mentioned above ensures some consistency in the consequent actions. The procedure evolved to also include:

- a request to confirm receipt of the e-mail (to ensure that the address was still valid and that the alert was not dropped by spam filters)
- a request to confirm that the plug-in is still being maintained
- details on how the plug-in developer could request assistance with understanding and addressing the vulnerability and how to obtain a review if is felt necessary
- how to restore the plug-in's status on the plug-in register

A characteristic of community developed plug-ins is that they are often developed by individuals rather than groups. In addition, they may be developed at a single point in time to address a particular need and require no routine maintenance. The responsiveness to vulnerabilities reported therefore depends on many things, including:

- Whether the developer is still contactable at all;
- Whether the developer has the time and motivation to fix an issue - in particular when they may not understand either the attack, the severity or how to fix it properly;
- Whether anybody is prepared to update or fork the plug-in.

Our advisories and the DokuWiki reporting process evolved to try to mitigate delays that may otherwise occur. In particular, flagging plug-ins as vulnerable in the register ensures that plug-in users are not left completely in the dark with respect to un-responsive plug-in developers.

Generally, a wiki maintainer has no control over who develops plug-ins, who publishes them, how they are published, how they are maintained, and has little opportunity to withdraw plug-ins.

Maintainers of core software that supports plug-ins have taken various approaches to managing the plug-ins. For the Mozilla web browser plug-ins/add-ons, the approach is to separate out add-ons that have passed a QA review [6]; anyone is still free to develop and publish add-ons but only the QA approved ones are listed on the official Mozilla site. This, of course, requires a QA process and staffing; something not readily available on smaller volunteer developments.

DokuWiki, like several other wikis, chooses to provide an open space where plug-in developers can publish plug-ins; no warranty is implied by a plug-in being listed on this register. However, in order to help manage security, any DokuWiki plug-in that is identified as representing a security risk can be flagged as such and it will then automatically be removed from the index of plug-ins. A user can, if they know the name of the plug-in, still navigate to the page, but that page contains a prominent warning that the plug-in is known to be vulnerable and un-patched.

The DokuWiki approach does not provide for proactively alerting plug-in users to the fact that a vulnerability has been identified; it relies on users checking frequently. It has been suggested on the DokuWiki mailing list[13] that a message should be sent to the mailing list. This is however not likely to reach many wiki site owners, indeed it appears that few plug-in developers routinely read the mailing list.

When writing advisories we have sought to include a lot of detail about the issue so that plug-in developers can understand the issue being described. The hope was that by doing so, we would minimise the need for developers to ask follow-up questions. In only a few cases was there any need for subsequent communication with the developers, suggesting that this approach was successful. In only one case was a fix identified as being inadequate.

Advice to plug-in developers on security in DokuWiki has been enhanced in order to help developers not to fall into the same traps when developing new plug-ins.

6. CONCLUSIONS

Community developed and 3rd party wiki plug-ins appear to be far more prone to security vulnerabilities than the core wiki software. This is not surprising since many plug-in developers are not experienced programmers. In addition, plug-in developers have a lot to learn before getting their first plug-in to work at all, let alone worry about security issues. Guidance on making the plug-in secure therefore needs to be integrated with general guidance on writing a plug-in. It also needs to be sufficiently straight-forward that the developer is likely to incorporate appropriate measures in the plug-in as it evolves.

[13]http://www.freelists.org/list/dokuwiki

All sample code and tutorials made available to plug-in developers that may be used as a template for plug-in development must also be secure and where appropriate demonstrate good use of available APIs to help avoid vulnerabilities.

Plug-in developers can assist by writing plug-ins in such a way that they are relatively easy to audit by source code examination. This may include:

- where an API is available to correctly encode output this should be used;
- use of constructs common to other similar plug-ins;
- performing input filtering as early as is practical;
- where filtering has to be embedded with other processing, laying out the code so that it is clear what filtering has been performed in each possible branch;
- performing output encoding, quoting or filtering close to the point where the output is generated, or grouping all the encodings together, whichever is most readable;
- use common routines for performing operations like encoding and quoting;
- avoid relying on routines that perform complex processing that co-incidentally acts as a validation/filtering routine by virtue of rejecting most data that appears to be an attack.

Unfortunately, despite the best will in the world, bugs happen and some of them will be security vulnerabilities. Wiki maintainers should have a procedure in place to deal with security advisories about plug-ins as well as for the core software, including communicating with plug-in developers and plug-in users.

When reporting vulnerabilities, effort spent in making advisories clear and detailed will be rewarded by the reduction in follow-up correspondence. It is quite likely that the plug-in developer may never have seen a security advisory, may not have any idea what they should do about it, or even understand why there is an issue. If a wiki maintainer does not have a published reporting process then we would recommend contacting the wiki maintainer asking what procedure should be followed.

7. REFERENCES

[1] *http://en.wikipedia.org/wiki/Cross-site_scripting.*

[2] *http://en.wikipedia.org/wiki/Fuzzing.*

[3] Dokuwiki: Color plug-in. *http://wiki.splitbrain.org/plugin:color.*

[4] Dokuwiki tutorial: Syntax plugins explained. *http://wiki.splitbrain.org/wiki:plugins:syntax_tutorial.*

[5] Firefox add-on: Html validator. *http://users.skynet.be/mgueury/mozilla/.*

[6] Firefox add-ons policy. *https://addons.mozilla.org/en-US/firefox/pages/policy.*

[7] Geeklog furum plug-in xss. *http://archives.neohapsis.com/archives/fulldisclosure/2003-q4/0376.html*.

[8] Iso/iec np 29147 responsible vulnerability disclosure. *http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45170*.

[9] Java script. *http://en.wikipedia.org/wiki/Java_script*.

[10] Mary ann davidson blog: The supply chain problem. *http://blogs.oracle.com/maryanndavidson/2008/04/08*.

[11] Ncsa httpd: Log format. *http://hoohoo.ncsa.uiuc.edu/docs/setup/httpd/LogOptions.html*.

[12] Owasp: Cross frame scripting. *http://www.owasp.org/index.php/Cross_Frame_Scripting*.

[13] Php: Regular expression details. *http://php.net/manual/en/regexp.reference.php*.

[14] Sans diary: The 10.000 web sites infection mystery solved. *http://isc.sans.org/diary.html?storyid=4294*.

[15] Serendipity bbcode plug-in xss. *http://www.s9y.org/63.html#A9*.

[16] W3c html v4.01 specification 18 scripts. *http://www.w3.org/TR/html401/interact/scripts.html#events*.

[17] W3c html v4.01 specification b.2 special characters in uri attribute values. *http://www.w3.org/TR/html401/appendix/notes.html#h-B.2*.

[18] V. Anupam and A. Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 15–15, Berkeley, CA, USA, 1998. USENIX Association.

[19] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., Sebastopol, CA 95472, 2005.

[20] A. Klein. 'divide and conquer' http response splitting, web cache poisoning attacks, and related topics. March 2004.

[21] D. Litchfield. Lateral sql injection: A new class of vulnerability in oracle. *http://www.databasesecurity.com/dbsec/lateral-sql-injection.pdf*.