University of Amsterdam

MSc System and Network Engineering

RP2

Dynamic access control analysis in WordPress plugins

Author:
Frank Uijtewaal

In collaboration with:
DongIT

August 29, 2016

Abstract

WordPress plugins are amongst the most vulnerable elements in a WordPress installation. One of the vulnerabilities that they may have is missing access control, which could wrongfully allow unprivileged users to execute sensitive code. To help detect such vulnerabilities we propose a model in which the plugin's code is run under various user roles while recording the execution trace. We explore two directions that one can take with the model: crawling the web interface to invoke the plugin's code within their normal context and running the plugins' code directly from within the PHP environment. The output of this model, the execution trace, contains all function calls that were made as well as their returned values. The objective of the model is to bring to light calls made by user roles that they should not be able to make.

Contents

Introduction

One of the common vulnerabilities that may exist in web applications is what is often called "missing function level access control" [1]. It implies that a function in the server code that is invoked as a result of a user's action does not properly check whether this user should be allowed access. A typical source of this vulnerability is the case where the function's public URL is not *visible* to ordinary users, yet *reachable* if people know the correct URL. If this fact is overlooked by developers, they may forget to build in the proper checks. In this report we present a method to help detect such vulnerabilities in an automated way. The research is specifically applied to WordPress plugins as WordPress is by far the most popular web framework [2] and its plugins have shown to be one of the most common attack vectors in a WordPress installation [3] [4]. For example, the Role Editor plugin[1] was found vulnerable earlier this year, not because it did not check for permissions, but because the permission that was checked for was missing an "s" in its name.

Automating the detection of missing access control is not a trivial task. How can we automatically differentiate between a "normal" database operation and one that only admins should be able to perform? Telling that a user should have elevated permissions may be the difference between a variable's value being a `1` or a `2`. Nevertheless, a simple but valuable analysis is to just look at what functions are called inside a request handler: if a particular function in the WordPress API should only be called by an admin, a flag is raised if it is accessible to any other type of user. To perform such an analysis, one needs to know what functions can be called by each of the existing users in a web application. This report proposes various methods to obtain such an overview.

On a high level, the process we propose starts by locating all of the URLs that a WordPress plugin exposes. Then, every URL is called under different user accounts, each having another access level. The execution is monitored, giving us a complete overview of what functions were called, what arguments they were called with and what values the calls returned. This overview is often called an "execution trace" We explore two approaches of implementing such an analysis. In the first approach we look into how it can be done by looking purely at the source code of the web application. The second approach addresses the same process but targets the application from its web interface. We implemented some parts of these approaches to test their validity, several other parts are suggested as future additions which we believe to yield actionable information to help detect missing access control. Finally, we built a simple tool called xtm [5] which allows one to easily filter through the execution trace.

Our main contributions are a systematic approach to obtain execution traces under various roles in WordPress plugins, both through the web interface and purely from the source code.

This report is structured as follows. Chapter 2, *Background* introduces the running example of a WordPress plugin that is referred to throughout this report and explains various concepts that the research builds upon. In Chapter 3 we outline the two approaches and specifically explain topics that both approaches share. Chapters 4, *Working from the source code* and 5, *Working from the web interface* then discuss their respective approaches to obtaining the execution trace. We conclude with Related work, Future work and Conclusions.

[1] Role Editor plugin: https://wordpress.org/plugins/user-role-editor/

1 Research questions

The main question we aim to answer is as follows:

> How can we automatically determine what functions a WordPress user role is allowed to call in arbitrary plugins?

The main question is subdivided into three sub questions:

- How can we automate locating entry points into the plugin?
- How can the entry points automatically be invoked correctly and exhaustively under different user roles?
- How can we determine what functions were called during execution?

2 Background

This chapter starts with the presentation of a simple plugin that is used to exemplify concepts later in the paper. Then it explains several topics in the field of code analysis that the reader should be familiar with and it introduces key components within WordPress.

2.1 Running example

Here we introduce the running example that will be referred to throughout this paper. While many of the concepts that play a role in it are discussed in later sections, the example is briefly explained here. The example consists of two code snippets: a simple WordPress plugin that handles a POST request and an HTML form that is used to send the request. The form is shown in Figure 1. It is used to send a POST request to WordPress' "admin-post.php" script. The input with `name="action"` on line 3 is used to tell the admin-post.php script what handler to dispatch the request to. The call to `wp_nonce_field` on line 4 inserts an input field named "_wpnonce" that holds a nonce to be verified by the request handler.

```
1 <form action="<?php echo esc_url( admin_url('admin-post.php') ); ?>" method="post">
2     <input type="text" name="val" />
3     <input type="hidden" name="action" value="simple_post" />
4     <?php wp_nonce_field();?>
5     <input type="submit", value="submit">
6 </form>
```

Figure 1: Running example of the form used to send a POST request to the plugin.

Figure 2 shows code for a simple WordPress plugin that handles the requests that are made with the form. The essential part is the `handle_simple_post` function. It is attached as a callback to the `admin_post_simple_post` hook, which effectively tells WordPress to route the submitted form to this function. The function verifies the nonce and checks whether the user has sufficient permissions. Note that the plugin is made as concise as possible and has as such stripped out important yet irrelevant checks. It also does not adhere to the Plugin best practices[1].

2.2 Types of software analysis

2.2.1 Static analysis

Static analysis means that the code is being checked without it actually running. This makes it similar to how programmers reason about their code. Static analysis tools are used to detect many classes of errors and/or vulnerabilities. Their output varies from clear descriptions on what to fix, where and how, to vague indicators describing that something may be amiss or does not adhere to some coding standard.

[1]Plugin best practices: https://developer.wordpress.org/plugins/the-basics/best-practices/

```php
/* Plugin Name: Simple Plugin */
class SimplePlugin {
    public function __construct() {
        add_action('admin_post_simple_post', array(&$this, 'handle_simple_post'));
    }
    public function handle_simple_post() {
        if (!wp_verify_nonce($_REQUEST['_wpnonce']))
            wp_die('nonce invalid');
        if (!current_user_can('edit_posts'))
            wp_die('access denied');
        else
            echo $_POST['val'];
    }
    public function activate() {}
    public function deactivate() {}
}
register_activation_hook(__FILE__, array('SimplePlugin', 'activate'));
register_deactivation_hook(__FILE__, array('SimplePlugin', 'deactivate'));
$plugin = new SimplePlugin();
```

Figure 2: Running example of a very simple plugin.

2.2.2 Dynamic analysis

Dynamic analysis implies the code under test is executed. Unit tests are a well known example of this: a piece of code is run under known circumstances, after which assertions are used to make sure the code performs as expected. Code coverage, which is further explained in Section 2.5.2, often accompanies testing to determine how much of the code was actually run. The term dynamic analysis is quite broad and it finds its uses in different areas as well. It is often a central part in automated test generation. During dynamic analysis the execution can be monitored, recording data such as variable values, control flow and code coverage.

2.2.3 Relevant analysis terms and techniques

In the field of security, static analysis can be used for "taint analysis". Taint analysis is based on the idea that user supplied input cannot be trusted and needs to be sanitised before it is used as input to sensitive sinks. An example of such a sink is the `mysqli_query` function. During the analysis, inputs are traced through the code to see whether they make their way to these sinks while still being "tainted", which could lead to a security vulnerability.

Another technique that is used frequently is "symbolic execution". The difference with normal execution is that variables do not take on actual values, but simply store, so to say, what operations their value is a result of. While symbolic execution often plays a central part in research concerning some form of dynamic analysis, it's not (really) implemented in freely available tools for PHP. Xdebug, which is a debugging extension to PHP, has an option to trace variable assignment that comes close, but that feature is (still) only supported in the human-readable output format [6]. Also, symbolic execution has been implemented as an extension to the Zend interpreter [7] [8], but the extension was never made public [9].

Analysis algorithms often make use of constraint solvers. Constraint solving, here, is the process of for example determining what value `x` should take on for a branching statement to evaluate to true or false. Consider for example the condition `x != 0 && x < 5`. A constraint solver could be used to come up with valid values for x for the condition to evaluate to true.

2.3 WordPress

2.3.1 Access control

The access control model in WordPress revolves around "capabilities". Having a capability essentially means having permission to do something. Besides capabilities, there are six default user roles. Users are assigned one or more roles, possibly custom ones. Roles group capabilities in a nicely defined package. The default user roles are "subscriber", "contributor", "author", "editor", "administrator" and "super-admin". Each of these roles has the same capabilities as all roles preceding it, plus some additional ones. The WordPress API uses the `user_can`[2], `current_user_can`[3] and similar functions to check whether the current user has a certain capability. For instance, the `handle_simple_post` method in our running example, depicted in Figure 2, checks whether the current user is allowed to edit posts.

In this example it's straightforward to establish what capability the code requires the current user to have. That's not always the case, though. Consider for example Figure 3. It shows part of the body of a URL callback from the popular Jetpack plugin [10], which essentially is a collection of many different features. The call to `$this->validate_call` is responsible for checking capabilities.

```
1 if ( is_wp_error( $error = $this->validate_call( $blog_id, $this->needed_capabilities ) ) ) {
2     return $error;
3 }
```

Figure 3: Indirection in capability assessment.

`validate_call` is a method defined within Jetpack itself. Tracing it through another function leads to the code snippet that is shown in Figure 4. The function that this snippet is part of can be passed an array of capabilities and returns true only if the user has all required capabilities.

```
1 $passed = 0;
2 foreach ( $capabilities as $cap ) {
3     if ( current_user_can( $cap ) ) {
4         $passed ++;
5     } else {
6         $failed[] = $cap;
7     }
8 }
```

Figure 4: Implementation of the actual verification. The variable "$capabilities" holds the capabilities to be checked for.

The issue that these code snippets address is exemplary for a lot of aspects in static code analysis. While simple structures can usually be successfully analysed, indirection quickly renders static code analysis ineffective. Dynamic analysis lends itself better for this type of research.

[2] user_can: https://codex.wordpress.org/Function_Reference/user_can
[3] current_user_can: https://codex.wordpress.org/Function_Reference/current_user_can

2.3.2 Hooking system

WordPress relies heavily on hooks[4], which come in two types: actions and filters. They allow plugin and theme developers to hook into the normal operation of WordPress. In other words: at various locations in its execution, the WordPress core code fires particular events, to which plugins and themes can subscribe listeners. Also, new hooks can (and for many use cases, should) be created to handle for example additional AJAX calls. Hooks form a central part in URL dispatching in WordPress.

Actions and filters are not much different from each other. They are created roughly the same way, they are attached to hooks using very similar API calls and are even stored in a single global variable, `$wp_filters`. However, actions are used to do something, like handling a request or sending an email, whereas filters take a value, modify it and then return it. The function signature for adding actions is[5]:

```
add_action($tag, $function_to_add, $priority, $accepted_args)
```

The `$function_to_add` parameter, which denotes the callback, may either be a string or an array containing both the class name and method name. In many cases the `$tag` and `$function_to_add` strings are not hardcoded but dynamically constructed at runtime. This makes static analysis impractical and therefore many of the papers discussed in Chapter 6, *Related work* for this type of web framework are not applicable. A couple of examples of how tag names and function names occur in plugins are shown below. The examples are all taken from the Jetpack plugin. Their origin is not important since they're just used to show the variety that they come in. It is interesting to see, however, that a single code base contains such diversity.

Examples of `$tag` occurrences:

- `sprintf( 'add_option_%s', self::OPTION_NAME )`
- `"comment_$new_status_$comment->comment_type"`
- `$theme_support[0]['filter']`
- `"admin_print_styles-$this->slug"`
- `'plugin_action_links_'.basename( dirname(__FILE__) ).'/'.basename(__FILE__)`

Examples of `$function_to_add` occurrences:

- `array( &$this, 'login_form_json_api_authorization' )`
- `array( Jetpack_Admin::init(), 'fix_redirect' )`
- `array( __CLASS__, __FUNCTION__ )`
- `array( $GLOBALS['publicize_ui']->publicize, 'async_publicize_post' )`
- `array( WPCom_Markdown::get_instance(), 'load' )`

Actions and filters can be invoked at any time using the `do_action`[6], `apply_filters`[7] functions or their array variants respectively.

[4] hooks: https://codex.wordpress.org/Plugin_API
[5] add_action: https://developer.wordpress.org/reference/functions/add_action/
[6] do_action: https://developer.wordpress.org/reference/functions/do_action/
[7] apply_filters: https://developer.wordpress.org/reference/functions/apply_filters/

2.3.3 Nonces

Nonces, "number used once" play an important part in securing requests to sensitive backends. In the context of web applications they're often used to protect against CSRF attacks. CSRF stands for "Cross Site Request Forgery", which is a type of web vulnerability in which an attacker can sneakily trick someone to perform an action on a target website. Usually, the victim has elevated permissions on the target website and the attacker makes use of the session that the victim has already built up to make specific calls to the backend. CSRF protection aims to mitigate this threat by sending a nonce alongside the HTML (form). A benign user would be able to send the correct nonce back along with the request, whereas an attacker does not know the correct nonce and can therefore not impersonate the user.

Nonces are part of the WordPress API[8]. Most notably one uses the `wp_create_nonce` and `wp_nonce_url` functions for creating nonces and `wp_verify_nonce` to verify the nonce once a user sent it as part of a request.

2.4 Fuzzing and automated test generation

Fuzzing, or fuzz testing, is an approach to software testing that involves running the software with a variety of different inputs. It is usually an automated or semi-automated process that intends to bring to light unforeseen, wrong behaviour.

Automated test generation, as the name implies, is used to automate the process of creating tests. It often relies on techniques like symbolic execution and constraint analysis to derive valid input to the software, as well as to try and walk all execution paths in the code. Oftentimes, fuzzing and automated test generation are used together: a testing sequence is generated first and then the test is supplied with various inputs by means of fuzzing.

There has been quite some research into automated test generation. How it's done generally boils down to two steps. In the first step, input is generated and in the second step the code under test is run with those inputs while its behaviour is observed. The observations are usually fed back into a second phase of input generation and this process is repeated until some condition is met. The goal is to iteratively come to valid set of inputs that allows us to traverse all control paths. This process used to be purely trial-end-error based, resulting in a low final code coverage. However, methods have improved. For example, Ma et al. [11] introduce a technique they call GRT: "Guided Random Testing". Their tool consists of multiple modules amongst which is one that performs static code analysis on the software under test. That module's goal is to enrich the initial test set by learning what may be valid inputs. While in part the process is still based on guesswork, sensible inputs are searched for first to increase the chance that eventually valid inputs are found and to speed up the process in general.

2.5 PHP instrumentation

Instrumentation is a broad term, used to describe the ability to monitor, debug or otherwise gain insight in what a program is doing when it runs. It provides the feedback that is needed during dynamic analysis. This section describes various types of instrumentation that are used in this paper in the context of PHP.

[8]Nonces: https://codex.wordpress.org/WordPress_Nonces

2.5.1 Execution traces

Execution traces, also known as "function traces" or simply "traces", are a way of logging a program's execution. An execution trace may contain the recording of the whole execution or only part of it, although that depends on the feature set of the tracer. PHP's most popular tracer is Xdebug[9], which is to be installed as an extension to the PHP interpreter. Xdebug can be configured to record quite some information, including wich functions were called, what arguments they were supplied with and what values the functions returned. It also records the calltree, which shows the relation between functions calls. To analyse the execution trace we created a simple program called xtm [5]. It parses Xdebug's machine readable function traces and features a pluggable filter system to search for specific calls.

2.5.2 Code coverage

Part of software testing is evaluating to what extent the test covered the software under analysis. This is called "Code coverage". For example, a coverage of 30% is usually considered to be low, as it means that 70% of our code was not touched and therefore not tested. Code coverage overviews can also be constructed with Xdebug, amongst others.

2.5.3 Monkey patching

Monkey patching is a term used to describe altering code at runtime. One of its uses is instrumentation: an existing function `f1` could be replaced with another, `f2`, which will print something to the console and thereafter call the original `f1`. Now, everytime `f1` was meant to be called, it still is eventually, but we have added a way to inspect what is going on. Another use case is to completely reimplement `f1`. PHP does not support monkey patching as well as some other languages do, but it can be done with libraries such as Patchwork[10].

2.5.4 Reflection

Reflection can be used for a program to inspect itself. Examples are finding out the name of the function that is currently being executed, or the parameter list that the function expects. It is also typical to be able to retrieve the filename and line numbers on which the function is defined, which can be of use to connect dynamic analysis to static analysis. PHP natively supports reflection[11].

[9] Xdebug: https://xdebug.org/
[10] Patchwork: http://antecedent.github.io/patchwork/
[11] Reflection in PHP: http://php.net/manual/en/book.reflection.php

3 Approach

The aim of this research is to introduce a code execution analysis framework for automated analysis of allowed functions calls per user role. This means that for each entry point into a WordPress installation that is accessible from the outside we invoke the code that lies behind it and record its execution trace. We explore two approaches that can be taken. The first approach looks solely at a web application's source code. This means that despite the fact that WordPress exposes a client side web interface, we disregard it completely and we analyse only the server side code. The entry points into the WordPress installation are extracted from the source using dynamic analysis and invoked using techniques that are prevalent in the field of automated test generation. The approach is described and evaluated in Chapter 4. *Working from the source code.* The second approach considers both the client side and server side of WordPress. It makes use of a crawling component that attempts to generate correct parameters to supply requests with. Access to the server side code is taken advantage of to better generate valid arguments for use in the requests. The second approach is discussed in Chapter 5. *Working from the web interface.*

Both approaches follow the high level model presented in Section 3.1. Two important topics that appear in each of the approaches are *state* and *path coverage*. They are explained in Sections 3.2 and 3.3, respectively.

3.1 High level model

Conceptually, both approaches follow the same process: 1) find the entry points, 2) invoke them successfully and 3) invoke them in all possible ways. This high level process is visualised in Figure 3.1. In our model, for each of the entry points we make an initial prediction of what would make valid arguments and what state the function expects to be run with. Then we try to invoke the entry point, *retrying* with modified inputs if the initial prediction turned out unsuccessful. After that step, we check whether we hit *all control paths*. If not, we modify the relevant inputs in order

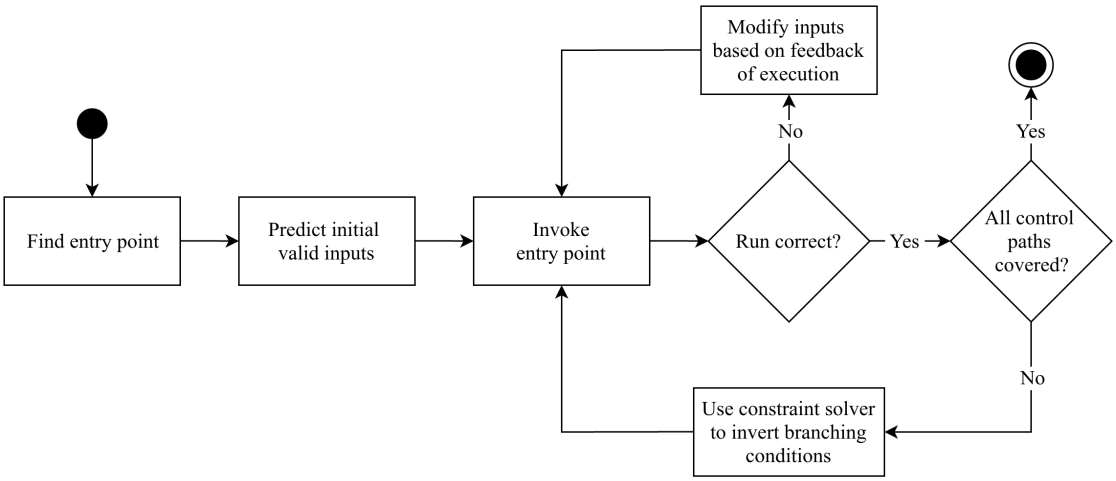

Figure 3.1: High level overview of the invocation approach.

to execute other paths, possibly revealing other function calls as well. An entry point is a point at which users can interact with the web application. The parts in the server-side code that handle the interaction are henceforth called "request handlers". Furthermore, in this model "Running correctly" is defined as "to run without PHP-related errors". A request handler returning a 403 - "Forbidden" status code is therefore still deemed a successful run.

3.2 Handling state

Arguments and state determine whether a request handler runs successfully or not. State presents a challenge, because requests can change a plugin's state, which can affect the handling of subsequent requests. This begs the question whether every attempt at invoking an entry point should be performed in a fresh state. There is no single, good answer to this question, as in some situations this is beneficial and in some cases it is not. For example, when two handlers are meant to be invoked in a set sequence, the second handler may be dependent on state set by the first. This state often plays a major role in more complex user interfaces. The answer also depends on the type of analysis that is being done. Our solution is to in principle keep state in the web interface approach and not to keep state in the source code approach. These choices are explained in detail in their respective chapters.

3.3 Covering all control paths

Invoking an entry point successfully once is not necessarily sufficient. Request handlers often contain multiple branches in their handling code and therefore multiple control paths. Each control path could be defined as a set of branch choices. A single invocation follows a single path, leaving potentially many lines of code untouched. Since we aim to find all code that could possibly be run under a particular user role, we exercise as many control paths as possible. Since branch points are often conditioned by either arguments or state, we modify these so that different branches are chosen, leading to different code covered. So, testing under a particular user role does not just involve trying to run *successfully*, but also *exhaustively*. Naturally, branching conditions existing of the `current_user_can`[1] function should left alone, because it causes the separation of control paths that we are looking for in the first place.

[1] And similar functions that check capabilities.

```
1  $allowed = current_user_can('edit_posts');
2  if (! $allowed) {
3      wp_die();
4  } else {
5      if ($x == 5) {
6          func_a($x);
7      } else {
8          func_b($x);
9      }
10 }
```

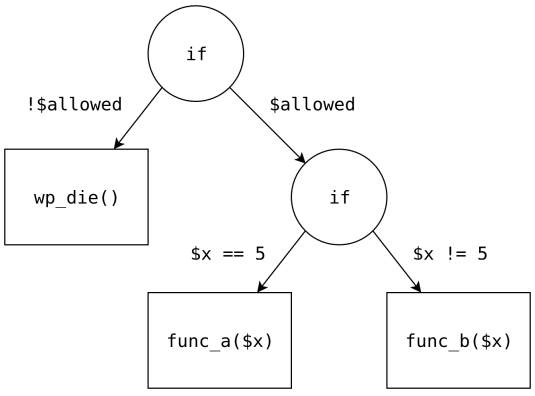

Figure 3.2: Code snippet with corresponding execution tree.

See for example Figure 3.2. It shows a small code snippet that contains some branches and a flow graph that presents those branches in a more abstract form. When run as a user with the proper permissions the execution takes the right branch at the top. Then, depending on the value of `$x`, either `func_a` or `func_b` is executed. Once a successful run is detected a constraint solver can pick up what the constraint was for the execution to go either way, and then inverse the constraint such that the other path is taken. That way, we get the complete overview of all functions that *can* get executed under a certain user role.

4 Working from the source code

This chapter approaches the problem of invoking entry point handlers solely from the source code. Section 4.1, *Locating request hooks and handlers* analyses how we obtain a complete list of request handlers defined in plugins including the hooks that they're attached to using dynamic analysis. Section 4.2, *Invocation environment* discusses how we propose to run the handlers outside of their normal enviroment. Then, Section 4.3, *Invoking the hooks* discusses mainly how valid arguments and state can be found and we conclude with summary of this approach in Section 4.4.

4.1 Locating request hooks and handlers

In this approach we specifically disregard the web interface, hence we need another way to find the locations where plugins expose entry points. We henceforth use the term "request handler". In WordPress plugins, declarations of request handlers are scattered around. Furthermore, these request handlers are nothing more than regular callbacks registered to hooks in WordPress (through the functions `add_action` and `add_filter`). Hence some extra work needs to be done to determine what callbacks serve as request handlers. Finally, plugin developers are not bound to use WordPress' hooking system to handle their requests at all. As is common in PHP, requests can be sent straight to any script that contains some handling code. As it is generally discouraged in WordPress to implement request handling this way, we assume that the more seriously developed plugins make use of hooks. Therefore, this last method is not discussed further. This section is structured as follows. Section 4.1.1 introduces various methods of locating callbacks in a WordPress installation. Then, Section 4.1.2 shows how the filter the request handlers from the rest of the callbacks.

4.1.1 Methods of locating plugins' hooks and handlers

WordPress' hook system makes it easy to retrieve the hook and function names at runtime. At every page load a large part of the WordPress environment is initialised, which includes loading all active (and valid) plugins. When a plugin is loaded, it usually immediately registers all of its callbacks. It is these hooks and callbacks that we are interested in. Often it is useful to gather more information about these callbacks. What additional information is required depends on the exact method of input and state analysis that is used, as explained in Section 4.3, *Invoking the hooks*. PHP's Reflection classes can at runtime inspect the request handlers and determine the number of parameters they accept, the file they are defined in and on what lines. To conclude, there are several methods to obtain a plugin's hooks and/or callbacks. These methods are discussed below. All of those methods involve loading the WordPress environment once while the method is implemented. This section concludes with a discussion on which method is preferred when.

Obtaining hooks from the $wp_filter variable

WordPress stores all registered callbacks in a global associative array called `$wp_filter`. Reading out this variable gives a complete overview of all callbacks that have been registered. The array's keys are the hook names and the values are made up by arrays of action and filter callbacks. The data structure makes no clear distinction between actions and filters, other than the fact that

action hooks only map to actions and filter hooks map to filters[1]. Code to read out this variable can be placed anywhere, as long as its executed *after* all actions and filters have been hooked.

We suggest creating a plugin which defines an action as shown in Figure 5. The action is hooked into the `init` hook, which is invoked after the time at which WordPress instantiates plugins. Therefore, all other plugins' hooks and callbacks are contained in the variable. Using a plugin allows us to keep the WordPress installation itself clean and easily replaceable in case different WordPress versions need to be tested with.

```
1 add_action('init', function() {
2     global $wp_filter;
3     file_put_contents('filters', print_r($wp_filter, true));
4 });
```

Figure 5: Action that writes $wp_filter's contents to file at runtime.

Obtaining hooks by instrumenting the hooking functions

A more direct method is to record the callbacks at the moment they are registered, which is inside the `add_action` and `add_filter` functions. It is here that we can still easily differentiate between actions and filters - a distinction that is somewhat lost in `$wp_filter`. Using Patchwork we can redefine the `add_action` and `add_filter` functions and include some instrumentation code. The best place to patch the functions is in the `wp-config.php` file, as it is meant to be edited and it is loaded before the inclusion of the hooking functions themselves. The latter reason is essential to Patchwork, which requires to be loaded before the code to be patched is included.

Analysing Xdebug's function trace

A third way of obtaining the hooks and attached callbacks for plugins is to simply record the entire execution of WordPress with Xdebug and then subsequently search the execution trace for instances of the `add_action` and `add_filter` functions. For instance, in order to find the `add_action` instance in our running example, we used xtm and configured it to read the function trace and to filter out everything but calls to `add_action`. It is then output as follows:

```
add_action (["$tag = 'admin_post_simple_post'", "$function_to_add =
   array (0 => class SimplePlugin   , 1 => 'handle_simple_post')",
   '$priority = ???', '$accepted_args = ???'])
```

The triple question marks are Xdebug's way of saying that those parameters were not supplied in the call. They then take on the default values `10` and `1` respectively as per the documentation[2].

Choosing the preferred method

Summing up, this section discussed three options: 1) reading out the `$wp_filters` variable, 2) instrumenting the hooking functions and 3) analysing Xdebug's function trace. All methods provide all of the hooks and callbacks that plugins register when loaded. Hence, whatever arguments remain for choosing one method over the other are implementation specific. 1 and 2 are fully contained in a single runtime, which is favourable when running the complete analysis in one

[1] add_filter's implementation: https://developer.wordpress.org/reference/functions/add_filter/
[2] add_action: https://developer.wordpress.org/reference/functions/add_action/

go is important. Since Xdebug writes its trace out to file, the file needs to be read again before the execution trace can be analysed. The methods also differ greatly in added performance overhead. We tested the response time of `/index.php` on a clean WordPress installation, with only the plugin from the running example installed. Each method was tested separately, printing only the most basic representation of the hooks and callbacks that the methods offered. Methods 1, 2 and 3 had a response time ratio of 1:20:10, respectively. Whether these results justify considering the performance overhead of the methods depends on the execution time of the rest of the analysis' implementation. To conclude, using method 2 it is not always possible to retrieve additional information about a callback. The reason for this is that the hooking functions are passed the callback's names, not the callbacks themselves. The callbacks may actually be defined later than that they are registered to the hooks, and hence they cannot be analysed with Reflection at the time of registration.

4.1.2 Finding which hooks and callbacks handle requests

Section 4.1 showed three options to obtain *all* hooks and callbacks at runtime. However, only some of them are meant to handle requests, and only some of those are actually defined by the plugin being analysed. Whether a callback is registered by a plugin can simply be determined by looking at the file that the callback is defined in. Finding out which of the callbacks are request handlers is not always that straightforward. Request handlers are registered in a couple of ways. First of all, plugins can use the admin-ajax.php script to dispatch AJAX calls to their handlers. The hooks that these handlers are registered to start with `wp_ajax`. Consequently, all hooks starting with `wp_ajax` can be considered to be used for (AJAX) request handling. The same applies to request handlers making use of WordPress' `admin_post.php` script: their hooks start with `admin_post`.

Developers also often attach their request handlers to the `init` hook[3]. WordPress invokes this hook when it is done loading. The request handler is typically implemented as shown in Figure 6. When dealing with a request, WordPress will invoke all such handlers, where each handler determines whether the request is destined for it based on the parameters in the request.

```
1 function send_email_handler() {
2     if ( isset( $_POST['unique_param'] ) && 'send_email' == $_POST['unique_param'] ) {
3         // handle request
4     }
5 }
6 add_action('init', send_email_handler};
```

Figure 6: Example of using the init hook to register request handlers.

Since `init` is a generic hook, its name does not give away what the hooked function's purpose is. Usually, this pattern queries either the `$_GET`, `$_POST` and/or `$_REQUEST` variables. Therefore the code can be (statically) searched for those occurrences to discriminate between request handlers and other callbacks. A good candidate for implementing this step is PHP-Parser[4] which is an open source PHP code parser. We say "usually", because there is no rule that prevents plugin developers from creating code that handles requests without parameters. In such cases the request itself is all that is required. Such cases are not found using this method. Finally, `init` is not the only hook that is used in this pattern. Depending on the request handlers' needs they can be attached to many different hooks. Therefore, this analysis should be performed on all callbacks

[3]init hook: https://codex.wordpress.org/Plugin_API/Action_Reference/init
[4]PHP-Parser https://github.com/nikic/PHP-Parser

that the plugin under test exposes. To conclude: there are three types of hooks that we take away from this step. These are hooks that start with `wp_admin`, hooks that start with `admin_post` and hooks that have registered to them one or more callbacks that we identified as request handlers.

4.2 Invocation environment

In Section 4.1 we discussed how we obtain hooks and attached request handlers. Under normal circumstances, these handlers are executed as the result of an HTTP request. In that case they have access to for example the database and the WordPress API. As we aim to invoke the handlers not through a web server but simply as normal code, we'll have to mimic this environment. In our approach we use the unit test classes that are built into WordPress by default[5], since they give us a complete WordPress environment out of the box. In Section 4.2.1 we discuss the difference between invoking the hooks and invoking the callbacks. Section 4.2.2 discusses how exactly we use the WordPress unit test classes.

4.2.1 Invoking hooks versus calling handlers directly

There are two ways to run request handlers: either we call the functions *directly*, or we run them *indirectly* by invoking the hooks to which they are attached. The latter is the preferred option, because of the potential dependencies that handlers of the same hook have to each other.

In Section 3.2, *Handling state* we introduced the challenges that dependencies between *entry points* bring. In this approach we choose to ignore those dependencies, since they are hard to deduce from the source code and in reality requests can be sent out of order as well. Still, there exists a potential dependency problem between multiple functions handling the same hook, which is perfectly acceptable in WordPress. Let's say a plugin uses a special action that performs access control checks. It defines a hook called `wp_ajax_post_edited` to which two handlers are attached, in order: `check_user_capabilities` and `handle_edited_post`. In this example, both handlers are run when the hook is invoked. The first simply checks whether the current user has the required capabilities and errors out if not. If control flows into the second handler that means that the current user passed the capability check. Calling the handlers separately, therefore, makes no sense. Hence we choose to invoke the hooks and not call the request handlers themselves separately.

4.2.2 Use of unit tests

The WordPress unit test framework makes it easy to invoke the hooks outside of their normal context. In its simplest form, such a unit test can be written as shown in Figure 7. It loops through the five standard user roles we want to invoke the hook as. For every individual loop it starts both a function trace and a code coverage monitor before executing `do_action('admin_post_simple_post')`. Then, the function trace and code coverage are stopped and their results written to file for further analysis. The coverage report tells us per line whether that line has been executed, giving us a preliminary overview of whether we touched all branches in the code.

The snippet in Figure 7 only shows the basic environment in which the hook is to be executed. Inducing what arguments the hook requires to be invoked with and what state needs to be set is discussed in Section 4.3, *Invoking the hooks*.

[5]WordPress automated testing: https://make.wordpress.org/core/handbook/testing/automated-testing/

```php
class TemplateTest extends WP_UnitTestCase {
    public function test_callback() {
        $role_names = array('subscriber', 'contributor',
            'author', 'editor', 'administrator');

        foreach ($role_names as $role_name) {
            $this->_setRole($role_name);
            $trace_filename = "trace-{$role_name}";

            xdebug_start_code_coverage(XDEBUG_CC_UNUSED);
            xdebug_start_trace($trace_filename);

            do_action('admin_post_simple_post');

            xdebug_stop_trace();
            $coverage = xdebug_get_code_coverage();
            xdebug_stop_code_coverage();
            file_put_contents('coverage', print_r($coverage));
        }
    }
}
```

Figure 7: Hook invocation template.

The Wordpress unit test framework provides more than just the proper environment for the plugin to run in. For example, it provides a helper to reset the global scope after each run, called `clean_up_global_scope`[6]. It also takes care of resetting the database between runs using `start_transaction`[7] and `$wpdb->query( 'ROLLBACK' )`.

4.3 Invoking the hooks

At this point we know what hooks and callbacks a plugin uses for handling requests. Now we invoke the hooks in a fully automated way. To do so we build on the topic of automated test generation, which tackles many of the challenges that we face here. This section is split in two: first we take a quick look at current research into automated test generation and how those insights can be applied to WordPress in Section 4.3.1. In Section 4.3.2 we look at how the fact that we are specifically targeting WordPress helps this process.

4.3.1 Strategy

There has been done quite some research in the field of automated test generation, mostly on statically typed programming languages [12] [13] [14]. The papers are not specific to PHP, but the general process they describe is applicable to PHP regardless. First, the code under test is statically analysed to find hints about what make good inputs, or types of input. Then a first run is performed, using these inputs. Inputs may be randomised in order to try to invoke more paths. By means of instrumentation is it possible to look into how the execution went. Based on this information new inputs are formed to better execute the code, or form new paths. Usually the analysis uses a combination of both concrete execution (running the code as usual) and symbolic

[6] clean_up_global_scope: http://develop.wp-a2z.org/oik_api/wp_unittestcaseclean_up_global_scope/
[7] start_transaction: http://develop.wp-a2z.org/oik_api/wp_unittestcasestart_transaction/

execution. The symbolic values are used to solve path constraints which are used to steer the execution into different branches.

The general process outlined above is roughly similar to what we propose in regard to invoking the hooks. Various additions are made though. To start with, statically tracing variables through a function's body often helps deduct what types of values they are expected to take on, as discussed in Section 4.3.2. To get a hold of the code to be analysed the Reflection step explained in 4.1.1 is to at least return the file name and line numbers on which the function in defined. Artzi et al. [8] statically search the source code for constants that help generate valid inputs to the request handlers. In regard to dynamic analysis, their paper made important modifications to the PHP interpreter. Most importantly, they built symbolic execution and monitoring path constraints into the interpreter. Using these features they get very clear information on the execution, which are similarly necessary in our case.

4.3.2 WordPress specific amendments

The sections below show how prior knowledge of WordPress helps make test generation more effective. Some of the techniques are directed at gaining knowledge of the parameters that are sent along with the invocation of hooks, while others aim to help set a correct state of the environment that they are run in. Such state is made up by for example the globals `$_GET`, `$_POST` and `$_REQUEST` and database contents.

Instant argument estimation with known hooks

Hooks to which request handlers are attached can be divided into two groups: custom hooks (defined by the plugin) and default hooks (defined by WordPress). Finding correct arguments to custom hooks is one of the main challenges we face. Finding correct arguments to default hooks is trivial, however, since the arguments that default hooks accept are documented. Teaching the analysis platform what arguments it should provide these hooks with is a huge improvement over having to determine them through other means.

Input estimation based on known sinks in WordPress

WordPress is built mainly for blogging and it sports a large API for building plugins that add functionality. This API is relied on extensively in plugin code. Naturally, the types of parameters that the functions and classes in the API expect are common knowledge. We can use this to our advantage when estimating valid input parameters to request handlers. We suggest building a module that traces parameters as they make their way to sinks (functions). If the sink is part of the WordPress API we can often backtrack what the initial value should have been. Symbolic execution is an excellent technique for this since at any time in the code it is capable of telling how a variable's current value came to be.

Say we are testing a request handler that takes a parameter of which the type and value is currently unknown to us. We invoke the handler through its hook regardless, supplying a bogus value for the parameter. The symbolic execution engine monitors the state of the variable as the execution of the handler progresses. Once the unknown parameter is used as an argument to a function in the WordPress API, we can determine what the parameter was expected to be. An example is `get_post_status`[8]. It accepts either a post object or an ID, which prompts us to create one and supply it as an argument for the next run.

[8] get_post_status: https://developer.wordpress.org/reference/functions/get_post_status/

We counted more than 5000 of functions in the WordPress API. We suggest analysing several plugins to find which functions and classes are used most and therefore deliver most value to the analysis platform by creating a signature for them. To our knowledge, there has been done no research into the usage of WordPress' API functions nor are there public services that have indexes on this.

Removing input dependencies

Up until now we've tasked ourselves with finding valid inputs to functions. In principle that is the driving factor behind our analysis. However, finding valid inputs is not always feasible. A very common case in which input generation will fail is the `wp_verify_nonce`[9] function. Trying the entire input domain makes no sense. A much better approach is to patch the function and have it return true or false as we see fit, regardless of the input it got. Since `wp_verify_nonce` performs a security check it is often used as a branching condition. Therefore, the value that we make the patched version return influences control flow and hence what functions are called consequently. Both paths (entered by true or false) can contain additional function calls so we invoke the request handler twice, once for each mocked return value. `wp_verify_nonce` is not the only instance to which this applies. `check_ajax_referer`[10] is very similar. Even though it wraps the former, it is best to handle it as a separate case. We assume there are a lot more of these instances, but there is no good way to automate finding them: they have to manually determined. Another way of finding them is by trial and error: during the access control analysis of plugins these functions will show up since valid inputs for them cannot be found.

4.4 Summary

In this chapter we discussed how WordPress plugins can be analysed while only considering the source code, based on the high level model depicted in Figure 3.1. The findings are summarised below.

We showed how we go about finding entry points in Section 4.1. Entry points either use hooks (the WordPress way) or they are simply implemented as requests sent straight to a PHP file that contains handling code (the generic PHP way). We looked only at the first type and started with how hooks in general are found. We proposed three methods: 1) reading out the `$wp_filters` variable, 2) instrumenting the hooking functions and 3) analysing Xdebug's function trace. No method is clearly preferable over another since priorities are very implementation specific, but method 1 seems overall the most versatile and fastest, as explained in Section 4.1.1. We subsequently filter the request handlers from the rest of the callbacks based on either the names of the hook they are registered to or their contents.

We proposed to invoke the hooks from within the WordPress unit testing framework. Plugins, and therefore their hooks and callbacks, are heavily tied into the WordPress environment and can consequently not be run without it. The unit testing framework provides the complete environment *and* sports various functions to easily manage state between runs.

Argument and state generation to invoke hooks with remain the hardest to reason about on a high level. Many papers show good results in regard to automated generation of these inputs, even in the field of PHP. We proposed various specialised techniques to improve on the general process highlighted in Section 4.3.1. First of all, many callbacks are hooked into default hooks, provided by WordPress out of the box. These hooks are known and therefore so are the

[9] wp_verify_nonce https://developer.wordpress.org/reference/functions/wp_verify_nonce/
[10] check_ajax_referer https://developer.wordpress.org/reference/functions/check_ajax_referer/

arguments that one should invoke them with. Hence, a plugin's callback attached to these can be called without additional analysis on their parameters. Furthermore, since WordPress provides a rich API we expect a considerable number of a callback's parameters to be used as an argument to these API functions. The API functions' signatures are known, so by tracing a callback's parameters into these functions we can determine of what type the parameters are and probably find valid values for them as well. Finally, we need to take care of various functions that block execution and for which we cannot properly generate valid inputs. These functions need to be manually identified and patched so that they return what we need them to in order to cover as many control paths as possible.

To conclude, our verdict of this approach is that it has a lot of potential. Referenced papers clearly show good results in automating test generation, even for PHP. It has also become very clear that effectively implementing an analysis tool takes a lot of effort and that it is not trivial to find what exact analyses deliver the most value.

5 Working from the web interface

This chapter explains the second of the two proposed approaches: working from the web interface. While the entry points are invoked from the web interface, the source code remains an important source of information in this approach. Also, code instrumentation is still required for feedback. Hence, this chapter builds on many of the techniques and principles discussed in Chapter 4, *Working from the source code* and refers to the corresponding sections when needed. This chapter is structured as follows. Section 5.1 introduces the subject of web crawling and points out its challenges. Section 5.2 starts with showing how request handlers can be found and invoked automatically, pointing at relevant papers and putting them in the context of this research. It continues with various additional analyses that can be performed knowing we're targeting WordPress. We conclude with a summary of this approach in Section 5.3.

5.1 Challenges to crawling web applications

What is usually crawled by current tools are links (formatted as anchor elements: `<a>...</a>`) and forms (`<form>...</form>`) that are part of the original HTML document sent by the server. While this covers part of the entry points, it also structurally misses potentially many more: JavaScript is frequently used to send AJAX calls to the server. To this end, JavaScript can simply be made to intercept an HTML form's "submit" event when a user clicks the submit button. This is often done to perform some client-side input checking. In fact, JavaScript can attach listeners to *any* event and then make an AJAX call when that event takes place. In the last situation, nothing about the original HTML document gives away that additional requests may be sent and therefore static crawlers fail to notice it[1]. Also, a page may retrieve additional data from the server, often immediately after the initial page has loaded [15]. This dynamically updates the page, which may bring along additional links, forms and JavaScript code. Finally, a big hurdle in crawling is the fact that a lot of content is "hidden" behind one or more user actions. This could take form in multiple AJAX calls being made from the same page, progressing its state to a final one, possibly forking along the way. Even if user actions actually take the browser to a completely new URL, it may be that only very specific consecutive requests "reveal" this URL from the user's point of view. Nevertheless, the fact that it is possible to successfully crawl dynamic content to some extent is clearly proven by Google [16] [17] [18]. Selenium[2] is a very popular framework used for web automation and can be used for similar crawling efforts. It (mainly) uses browsers such a Firefox and Chrome to load and render pages exactly as they normally are. Additionally, it allows one to programmatically interface with the browser's DOM and more.

5.2 Invoking handlers through the web interface

Here we discuss how one can successfully invoke handlers through the web interface. First we look at literature to see how we can go about the broad field of crawling, input estimation, and feedback through instrumentation. Then, we complement the findings with several techniques based on our target platform, WordPress.

[1] Some tools that perform crawling are able to search the JavaScript code for static URLs, such as The Burp Suite: https://portswigger.net/burp/

[2] Selenium: http://www.seleniumhq.org/

5.2.1 Strategy

In Section 3.1, *High level model* we introduced the three main steps involved with analysing entry points in our model: 1) find the entry points, 2) invoke them successfully and 3) invoke them in all possible ways. While logically they are separate steps, in a web based approach they are highly intertwined. For example, one can only find new entry points by successfully invoking currently known ones. Therefore, we iterate through these steps to advance our analysis through the web interface.

Artzi et al. [8] follow an approach that is very applicable to our situation, although their goal is quite different. They built a tool called Apollo which aims to find malformed HTML, crashes and errors in web applications written in PHP. We summarise our understanding of what they did and focus specifically on what is useful in our situation. Apollo extracts the elements on an HTML page that indicate entry points, such as forms and href elements. It also statically looks for some of the constructs in JavaScript code that indicate interaction with the server. It then attempts to use these extracted elements to make correct requests. The web application under test is run with a modified version of the Zend PHP interpreter, which performs quite an elaborate form of instrumentation. With the help of the modifications the request handlers are executed both concretely and symbolically. The symbolic execution is used to find alternate control paths. Request parameters are generated in two ways. Some of the input parameters are calculated based on what value they should assume in order for the execution to take a particular path. It is said that the parameters *satisfy a given path constraint*. Parameters for wich there are no such constraints are generated simply using a combination of random values and constant values mined from the source code. All state changes are monitored and used for feedback, amongst which are modifications to the database and cookies.

The approach taken by Artzi et al. provides a good foundation for our analysis. Nevertheless, we propose two additions: a module that fully handles JavaScript and several optimisations that take into account the fact that we work with WordPress. Section 5.2.2 discusses how the JavaScript handling module is retrofitted. Section 5.2.3 discusses the WordPress based additions. Finally, since we aim to find allowed code execution on a user basis the analysis is performed for every user. Naturally, once valid parameters are found for a certain request for one user role, the same parameters are probably applicable for the other user roles.

5.2.2 Handling JavaScript

The approach implemented by Artzi et al. does not dynamically analyse JavaScript: it only statically analyses JavaScript looking for places that denote interaction with the server. As explained in Section 5.1, this leaves many entry points untested. We propose to complement the crawling with a tool like Selenium so that JavaScript can be taken into account as well. Selenium renders the page, which means amongst others that it builds the DOM and executes JavaScript. The event listeners that are registered by JavaScript can be analysed to determine whether they are used to send requests to the server and invoked if so.

The module fits into the model by Artzi et al. quite well. Where normally the HTML is statically analysed, this time around the page is fully rendered and all JavaScript executed. Where previously form and href elements were taken from a page's source, now they're extracted from the DOM, which includes all elements dynamically retrieved with JavaScript. Furthermore, the callbacks to JavaScript's event listeners form a new source of entry points.

5.2.3 Optimisations based on WordPress

This section proposes a couple of optimisations to the crawling and input estimation process based on the fact that the analysis is targeted at WordPress.

Improved parameter generation

Artzi et al. generate request parameters based on constant values found in the source code, randomisation and constraint analysis. In our model, we add to this prior knowledge of WordPress. Most importantly, this involves input estimation based on known sinks and removing input dependencies. These techniques are explained Section 4.3.2. Removing input dependencies is useful in cases where valid parameters cannot be generated automatically, but it is less crucial here than it is in Chapter 4. There, the technique is required to work around for example nonces, which is not necessary in this approach since we are crawling the web interface and hence catching the proper nonce values that way.

Preconfigured runs

WordPress comes with a lot of functionality out of the box, such as creating content and managing the installation through `wp-admin`. With this functionality come a lot of default interactions between the client and server. It is possible to try to determine valid parameters for these requests for each and every plugin again, but we would be doing unnecessary work. The analysis can be made a lot more effective by creating a module that knows how to perform all the default interactions. This is useful because these interactions already cause a lot of the default hooks to invoked, which in turn can execute parts of the plugin under test.

Implementation wise, we propose teaching the crawler to start at the root of the installation (`index.php`) and crawl all URLs that WordPress exposes by default. The requests that can be sent by default are provided with, again, a default set of parameters of which we know beforehand that they are valid. At the same time we record new URLs, forms, href elements and JavaScript that has been inserted by the plugin under test so that we have as many leads as possible to continue crawling.

How much of a plugin's code is executed as result of this standardised crawling procedure obviously depends on the number of default hooks the plugin has registered callbacks to. Tests will have to show its usefulness but we believe it can be a significant improvement on starting from zero.

5.3 Summary

This chapter discussed invoking entry points into WordPress plugins from the web interface based on the model depicted in Figure 3.1. The most important findings are summarised below.

We started with an introduction on crawling web applications and why it is not trivial. Two reasons stand out: first, JavaScript makes web interfaces highly dynamic and therefore analysis more difficult. Second, complex web interfaces often expect a user to perform multiple steps in succession where each step is dependent on the previous one. This makes succesful analysis of such an interface harder because 1) state has to be correctly kept between subsequent requests and 2) if one request in the chain cannot be automatically handled all subsequent steps are out of reach as well.

To discuss how the crawling and invocation process takes form we referenced an important paper that essentially implements what we need, but does not take the dynamic nature of

JavaScript into account. We proposed to create a module based on Selenium or a similar framework that is capable of fully rendering a webpage, handing JavaScript as well. We explained how such a module can be retrofitted into the model used in the paper.

In addition to this model there are several optimisations to be made. Since in this approach we look at both the web interface and the source code, we look for "known sinks", functions in the WordPress API, and use them to deduce valid parameters to send along requests during crawling. Furthermore, server side code may need to be modified, "patched", in order to help progress the crawling in case we fail to generate correct parameters. Finally, WordPress comes with a lot of functionality out of the box. We suggest creating a tool that knows exactly how to crawl this default version of the installation in order to speed up to process. While doing so we probably invoke a considerable amount of code in the plugin under test and obtain starting points to continue crawling.

To conclude, since this approach uses both the web interface and the source code to invoke entry points, one would naturally say it should perform at least as good as the purely source-code based approach. Nevertheless, because the principle loop consists of crawling the web interface with all problems that accompany crawlers, the best conclusion we can give on this subject is that it highly depends on the quality of the implementation.

6 Related work

Gauthier et al. [19] have a very similar goal as this report has. They aim to find code blocks in the open source application Moodle[1] that are protected by capability checks. They do so using static analysis and model checking techniques, which is not an effective approach in a WordPress environment. The capabilities that they check are defined by static strings and are as such not dynamically constructed at runtime as they often are in WordPress plugins. Also, they found that the access control model exists of only a top level call to one of several API functions. Son et al. [20] take an interesting approach to finding missing security checks, as they assume no prior knowledge of the system under test. They start with several observations of constructs typically found in web application code and by looking at patterns they gradually learn where possible security errors are. As their approach relies on static analysis techniques, it's hard to say how their methodology would hold when applied to WordPress, although it would be interesting to compare. Noseevich and Petukhov aim to detect access control flaws using their tool " AcCoRuTe" [21]. In their approach, they construct the "use case graph" of the web application with some manual help. Then, they traverse the graph while applying differential analysis at the server, taking note of what changed in for example the database's contents. In [22], Near and Jackson present their tool "SPACE", which works based on the idea of matching extracted access patterns to known safe patterns. It is a static analysis approach implemented in and for Ruby programs. Whilst the goal is different, the techniques described in Artzi et al. [8] are very relevant to our research. The paper describes the algorithm and implementation of a tool called "Apollo", which is capable of automatically finding malformed HTML and PHP errors. They do so by crawling the web interface of a web application while observing the executing using a modified version of the PHP interpreter. Many of the techniques they describe have been used in this report.

[1] Moodle: https://moodle.org/

7 Future work

The approaches mentioned in this report model the most important parts of finding and invoking request handlers in plugins and various steps in the process have empirically been verified. A full-fledged software implementation will require further development; we have mentioned several features that could be incorporated in such a tool. A comparable effort is made in [8]. There and in various other papers [23] [7] we've seen that (dynamic) symbolic execution is used as one of the main techniques on which the analysis is based. Remarkably, however, we were unable to find a generic symbolic execution library or extension to the PHP interpreter, suggesting that every research group builds such a tool themselves. We believe that a freely available and well developed symbolic execution engine for PHP would greatly benefit research in this field. A good point to start such development is [9].

Another valuable future work is to take a large scale approach in estimating what are the most frequently used code constructs in WordPress plugins. For example, how often do plugins define new hooks versus simply hooking into the existing ones? How are these hooks invoked? Such knowledge can help pinpoint where to focus research on.

Finally, a lot more research can be done on the execution trace. We mentioned that one can find possible access control vulnerabilities by looking for calls to privileged functions, but there must be other analyses as well. For example, one could analyse what database calls are made. It may be very possible to generalise some of the constructs into signatures that can then be used to detect a broader range of missing access control problems.

Conclusions

In this report, we have shown how we can automatically determine what functions a WordPress user role is allowed to run in plugins. One can do so by running the plugin's code and monitoring the execution using Xdebug to create an execution trace. The execution trace contains all function calls that were made during the execution. An important aspect in generating the traces is to walk all control paths in the code so that the code coverage is as high as possible. Analysis of this execution trace is possible with for example the tool that we built, xtm.

We discussed two approaches to running a plugin's code automatically: 1) working solely from the source code and 2) applying a web crawling method. In the first approach we showed how entry points into the plugin can be found by applying dynamic analysis to find the calls to the hooking functions in WordPress. We proposed to invoke these entry points from the WordPress unit test framework in order to overcome the fact that the entry points are called out of context. Finding correct arguments and state to run the functions in is quite similar to the field of automated test generation and we applied the ideas and concepts to WordPress. Several optimisations to the test generation are possible in regard to WordPress. Most importantly, we 1) make use of the fact that many hooks are known which instantly provides us with proper arguments, 2) use the fact that plugins use many of WordPress' API functions, which help generating valid arguments and 3) remove particular API functions from the equation since generating valid arguments for them is, by design, not feasible.

The approach using the web interface relies in principle on a crawler to find and invoke entry points. Hence, the approach deals with the dynamic nature if web applications directly. The most important reason for this dynamic nature is web applications' use of JavaScript which we aim to tackle using a web automation tool such as Selenium. For the crawling methods themselves we referred to a very related paper that discussed modifying the PHP interpreter to have a very detailed look at how code runs. Using the acquired information it is then possible to try to automatically generate parameters to be sent along the requests during crawling.

While we think both approaches should yield good results it is hard to firmly say one is better than the other. Since the source code approach does not consider static HTML sources for example, it misses entry points that do not make use of WordPress' hooking functions, while they are easily detected in the crawling approach. The web approach, however, needs to deal with JavaScript, which may make it loose precision as well. In our opinion it's not possible to say at this point what approach is preferable since the performance will depend on the quality of the implementation more than on the conceptual possibilities.

Bibliography

[1] OWASP, *Top 10 2013-a7-missing function level access control*. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control (visited on 08/15/2016).

[2] W3Techs, *Usage statistics and market share of wordpress for websites*. [Online]. Available: https://w3techs.com/technologies/details/cm-wordpress/all/all (visited on 06/24/2016).

[3] Wordfence, *How attackers gain access to wordpress sites*. [Online]. Available: https://www.wordfence.com/blog/2016/03/attackers-gain-access-wordpress-sites/ (visited on 06/16/2016).

[4] WPWhiteSecurity, *Statistics highlight the biggest source of wordpress vulnerabilities*. [Online]. Available: https://www.wpwhitesecurity.com/wordpress-security/statistics-highlight-main-source-wordpress-vulnerabilities/ (visited on 06/16/2016).

[5] F. Uijtewaal, *xtm, Xdebug Trace Manipulator*, Jul. 3, 2016. [Online]. Available: https://github.com/delins/xtm.

[6] D. Rethans, *Variable tracing with xdebug*, Mar. 25, 2009. [Online]. Available: https://derickrethans.nl/variable-tracing-with-xdebug.html (visited on 08/13/2016).

[7] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 199–209, ISBN: 978-1-4244-3453-4. DOI: 10.1109/ICSE.2009.5070521.

[8] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, Jul. 2010, ISSN: 0098-5589. DOI: 10.1109/TSE.2010.31.

[9] MIT. [Online]. Available: http://groups.csail.mit.edu/pag/ardilla/zend-changes.html (visited on 08/13/2016).

[10] WordPress.com, *Jetpack*. [Online]. Available: https://wordpress.org/plugins/jetpack/ (visited on 06/08/2016).

[11] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "Grt: An automated test generator using orchestrated program analysis," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Nov. 2015, pp. 842–847. DOI: 10.1109/ASE.2015.102.

[12] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005, ISSN: 0362-1340. DOI: 10.1145/1064978.1065036.

[13] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 132–141, ISBN: 978-1-4673-3076-3.

[14] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005, ISSN: 0163-5948. DOI: 10.1145/1095430.1081750.

[15] P. A. Fedorynski, *Get, post, and safely surfacing more of the web*, Jan. 11, 2011. [Online]. Available: https://webmasters.googleblog.com/2011/11/get-post-and-safely-surfacing-more-of.html (visited on 08/11/2016).

[16] A. Audette, *We tested how googlebot crawls javascript and here's what we learned*, May 8, 2015. [Online]. Available: http://searchengineland.com/tested-googlebot-crawls-javascript-heres-learned-220157 (visited on 08/11/2016).

[17] J. Mueller, *An update (march 2016) on the current state & recommendations for javascript sites / progressive web apps in google search*, Mar. 4, 2016. [Online]. Available: https://plus.google.com/+JohnMueller/posts/LT4fU7kFB8W (visited on 08/11/2016).

[18] Google, *Use fetch as google for websites*. [Online]. Available: https://support.google.com/webmasters/answer/6066468 (visited on 08/11/2016).

[19] F. Gauthier, D. Letarte, T. Lavoie, and E. Merlo, "Extraction and comprehension of moodle's access control model: A case study," in *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on*, Jul. 2011, pp. 44–51. DOI: 10.1109/PST.2011.5971962.

[20] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: Finding missing security checks when you do not know what checks are," *SIGPLAN Not.*, vol. 46, no. 10, pp. 1069–1084, Oct. 2011, ISSN: 0362-1340. DOI: 10.1145/2076021.2048146. [Online]. Available: http://doi.acm.org/10.1145/2076021.2048146.

[21] G. Noseevich and A. Petukhov, "Detecting insufficient access control in web applications," in *SysSec Workshop (SysSec), 2011 First*, Jul. 2011, pp. 11–18. DOI: 10.1109/SysSec.2011.28.

[22] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: ACM, 2016, pp. 947–958, ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884836.

[23] G. Agosta, A. Barenghi, A. Parata, and G. Pelosi, "Automated security analysis of dynamic web applications through symbolic code execution," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, Apr. 2012, pp. 189–194. DOI: 10.1109/ITNG.2012.167.