USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Extract: A PHP Foot-Gun Case Study

Jannik Hartung, Simon Koch, and Martin Johns,
Technische Universität Braunschweig

https://www.usenix.org/conference/woot25/presentation/hartung

This paper is included in the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies.

August 11–12, 2025 • Seattle, WA, USA

ISBN 978-1-939133-50-2

Open access to the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies
is sponsored by USENIX.

Extract: A PHP Foot-Gun Case Study

Jannik Hartung
Technische Universität Braunschweig
jannik.hartung@tu-braunschweig.de

Simon Koch
Technische Universität Braunschweig
simon.koch@tu-braunschweig.de

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de

Abstract

The `extract` call in PHP poses a similar threat to the security of a PHP application, if used naively, as the *register_globals* configuration that has been removed from PHP in version 5.3. We provide an attack analysis of its usage, showing the impact that unsafe usage can have. To understand how the security impact of `extract` manifests, we conduct a large-scale static analysis of 28325 open-source PHP projects to detect its insecure usage. Subsequently, we investigate each detected potentially vulnerable call manually to assess its security implications for the surrounding project and discover a total of 154 injection vulnerabilities and 86 CFG high jacking threats, including 60 privilege escalations. Thus demonstrating the danger of `extract`. As our final contribution, we discuss multiple paths forward for PHP to mitigate the dangers of this call.

1 Introduction

PHP is rich in Foot-Guns, i.e., features that easily lead a programmer to make potentially fatal security mistakes. One of its well known past examples is the configuration option *register_globals* [1]. When active, this configuration would automatically insert all provided request parameters as globally accessible variables within the execution. Consequently, the developer lost control over the initialization values of any variable, with severe consequences for the integrity and security of PHP programs. This turned out to be sufficiently problematic, that the configuration option was defaulted to false with PHP 4.2 [2], deprecated in 5.3 [3], and permanently removed with 5.4 [4].

While this security Foot-Gun has been closed off for good, the next best thing in terms of Foot-Gun potential is still available and actively used in PHP projects: `extract`. `extract` takes on an associative array and injects the contained values as variables named after the corresponding key into the context of its call. Thus, as soon as an attacker can control the input to extract, i.e., if a careless developer passes a super global such as `$_GET` or `$_POST` as the parameter, they can reach the same capabilities as if *register_globals* was still a possible and activated feature.

As this feature is still available in PHP, we want to understand its usage prevalence and the resulting security implications. To this end, we developed a static analysis methodology, based on the PHP CPG [30], to detect user controlled input to `extract` and conduct a subsequent manual analysis to understand the implications of each detected call with user input. In our analysis, we observed that the threat of `extract` can be broadly grouped into two types: *injection* and *CFG Manipulation*. Injection means that the attacker can influence a variable that is later on used as input, e.g., to an SQL query or call to the command line. CFG Manipulation allows the attacker to influence the subsequent flow of the execution, which can lead to privilege escalation in the worst case.

We applied our analysis on 28325 public PHP repositories and detected 4934 calls to `extract` across 1331 of them. Our static analysis deemed 146 of those calls as problematic and thus flagged a total of 50 repositories as potentially vulnerable. With our subsequent manual inspection, we were able to verify that 117 calls across 26 repositories pose a security risk. In total, we detected 154 injection vulnerabilities covering 81 XSS, 65 SQLi, 3 Command Injection, 2 Open Redirect, and 3 SSRF. Additionally, we observed 86 CFG high jacking threats, including 60 privilege escalations.

Our results indicate that while most calls to `extract` are inherently safe, as they do not involve user input, the ones that do are vulnerable with severe security implications. Consequently, we propose that PHP changes the default behavior of `extract` to not allow overriding variables within the current scope accidentally, especially not any global or superglobals.

To summarize, our contributions are:

- an attack analysis of the `extract` PHP API
- a large scale static analysis of `extract` usage
- a manual security assessment of 146 calls to `extract`

```
extract(array &$array,
        int $flags = EXTR_OVERWRITE,
        string $prefix = ""): int
```

Figure 1: The API documentation for `extract`. It has three parameters with only the first, i.e, the associative array, being mandatory. The flags and prefix parameters are optional and set to a default value if not provided.

In the remainder of the paper, we first provide a detailed exposé on the `extract` API call and why it poses a security risk in Section 2. Afterward, we detail our static analysis methodology in Section 3 and present its results in Section 4. This is followed by a selection of case studies showcasing the (mis)use of `extract` in Section 5 and a discussion of its broader implications for the security of PHP projects in Section 6. Finally, we provide a literature overview of the related work in Section 7 and conclude with a summary of our key contributions and findings in Section 8.

2 The PHP Extract API Call

PHP powers the web, according to W3Tech about 74% of web pages are PHP-based [8]. If you are using the web, you have most likely interacted with a PHP application. It is imperative that PHP applications are secure, and understanding dangerous language features is an essential step towards this, in our case, the focus is on `extract`.

However, to understand the dangers of `extract` we first need to establish the required background framing the impact a careless call to `extract` can have. Subsequently, we do a deep dive into the call, its parameters, and its impact on the program execution. Finally, we explore the possible security implications of an insecure call.

2.1 PHP In a Nutshell

PHP is an interpreted language that is commonly used to power web applications. The applications are hosted by web servers such as Apache or nginx. When a browser directs a request to a web page hosted by the server, the server translates the request into a call to the corresponding PHP script of the web application. Commonly, this is done by taking the path of the request URL and mapping it onto the folder structure of the underlying web application, but servers do support rewriting this path. The resulting output of this execution is then returned as the response. The main takeaway is that each request against a PHP-powered web application is an individual execution of a corresponding PHP script representing the requested functionality.

Due to this scheme, each execution of a PHP script is self-contained and does not keep state from any previous execution. To keep track of a specific state requires using external storage such as the disk via the PHP Session management or access to a database. Additionally, each execution needs to retrieve the state information from that storage for the request. The required information for an execution must be provided by the request via a session cookie, headers, or the request parameter. PHP provides access to this information via the `superglobals`, associative arrays mapping the values of the query, body, header, and cookies into the global scope of the execution. Those arrays are called according to the information they map, i.e., `$_GET` for the query parameter, `$_POST` for the body values, `$_COOKIES` for the cookie values, and `$_SERVER` for the headers, path, and general information. PHP also supports array decoding in the parameters [5], e.g., the url `https://example.com?arr[a]=1&arr[b]=2` would generate a single entry `arr` for the `$_GET` superglobal containing an associative array with the keys `a` and `b`.

In summary, each execution is strictly separate from any other execution and thus no interference can happen. But this means that an execution completely relies on the provided input via the request. As a consequence, any input that stems from a request has to be distrusted implicitly to ensure a safe execution and shortcuts for developer convenience or functionality can have severe and unforeseen consequences.

Table 1: Overview of the different flags for a call to `extract` and their effect on the execution result.

Flag	Effect
EXTR_OVERWRITE	If there is a collision, overwrite the existing variable.
EXTR_SKIP	If there is a collision, don't overwrite the existing variable.
EXTR_PREFIX_SAME	If there is a collision, prefix the variable name with prefix.
EXTR_PREFIX_ALL	Prefix all variable names with prefix.
EXTR_PREFIX_INVALID	Only prefix invalid/numeric variable names with prefix.
EXTR_IF_EXISTS	Only overwrite the variable if it already exists in the current symbol table.
EXTR_PREFIX_IF_EXISTS	Only create prefixed variable names if the non-prefixed version of the same variable exists in the current symbol table.
EXTR_REFS	Extracts variables as references.

2.2 A Call to Extract

PHP's `extract` is a convenience function that allows developers to directly translate an associative array into a set of in-scope variables [6]. `extract` takes up to three parameters as input with only the first, i.e., the associative array, being re-

```
extract(array &$array,
        int $flags = EXTR_OVERWRITE,
        string $prefix = ""): int
```

Figure 2: The API call specification for extract [6].

quired. The remaining two parameters are *flags*, which guide the handling of variable extraction and creation, and a *prefix* string, which is required when setting the corresponding flags. The default value of flags is set to `EXTR_OVERWRITE` and the default prefix is the empty string. Figure 2 shows the API call signature as given by the documentation.

The PHP documentation prominently states that `extract` should not be used on untrusted data and provides eight flags to influence the extraction behavior. Table 1 provides an overview of the flags that can influence the execution of `extract` and their effect. The flags can be broadly split into two categories, with `EXTR_REFS` being an outlier. One half of the flags influence if already existing variables are overwritten by extract, namely `EXTR_OVERWRITE`, `EXTR_SKIP`, and `EXTR_IF_EXISTS`. The second half of the flags influences if a created variable is prefixed by a provided string, namely `EXTR_PREFIX_SAME`, `EXTR_PREFIX_ALL`, `EXTR_PREFIX_INVALID` and `EXTR_PREFIX_IF_EXISTS`. If `EXTR_REFS` is set, a newly created variable only references the value in the provided array. Changes to the variable are thus propagated to the caller. Multiple flags can be combined via OR'ing them.

Figure 3 provides an example of how `extract` can be used, for convenience, in live code. The code snippet represents the search functionality of a web shop and uses a user provided keyword to search its products. If such a product is found, it is displayed, otherwise the functionality displays a generic error message. For convenience, the code does not use the associative super global array `$_GET` to access 'keyword' and assign it to its required variable `$keyword` but directly passes it to `extract` which has the same overall effect but requires less code. Consequently, if provided, after the call to `extract` in line 8 the variable `$keyword` exists within the scope and has the value as provided by the query.

2.3 Security Implications of Extract

Our attacker has full control over the request and its content directed to the server, in the case of phishing, our attacker has only full control over the URL used for the request. The latter of the two capabilities becomes significant when revisiting our usage example of `extract` in Figure 3. While our code snippet provides an example of how the usage of `extract` can be convenient for the developer, it also directly provides an example of how this convenience can result in a security vulnerability.

```
GET /shop/search.php?keyword=thermomix

/shop/search.php
<?php
$keyword='unknown';
$error = "sorry, no such product";
//creates variable $keyword = 'thermomix'
extract($_GET);
$products = get_all_products_from_db();
if(in_array($keyword,$products)) {
  echo array_search($keyword,$products);
} else {
  echo $error;
}
```

Figure 3: A PHP snippet showing a search functionality using `extract` to search for a user provided product name. For convenience, the developer directly passes the `$_GET` variable to `extract` to extract the search keyword, that is directly injected into the global scope as the corresponding variable in line 8. This is an example of unsafe `extract` usage as an attacker can craft a request also setting the query parameter *error* to an arbitrary value and thus override the corresponding value within the execution, resulting in a reflected XSS, if a product is not found.

Due to their ability to craft an arbitrary GET request, an attacker can freely choose the variables that will be created or overwritten in line 8. As the call to `extract` does not provide any additional arguments besides the associative array itself, in our case `$_GET`, it takes all keys of the array and creates variables filled with the corresponding value within the array. As no flag is provided, the default value of `EXTR_OVERWRITE` is used, meaning that any already existing variable is overwritten with the content of the array entry. This allows the attacker to overwrite the variable `$error` and control its content, which is the message that is returned to the user's browser in case the product provided via the *keyword* value does not exist. Consequently, the attacker can craft a link, that results in a reflected XSS vulnerability for anybody using that link, with all its security implications.

In our example, the result of the insecure use of `extract` was a XSS vulnerability. But the potential impact an exploitable call to `extract` can have is diverse and highly dependent on the context of its call. First and foremost, it is important what variables are only read after the call to extract without writing to them first, as those form the scope an attacker can manipulate the program. Secondly, the scope of the call to `extract` is important. If `extract` is only called within a function or method call, its impact is restricted to that scope and any variable outside it is safe from manipulation. Consequently, if `extract` is called in the global scope main function of a PHP script then its impact can be global, e.g.,

```
GET /shop/search.php?INDEX_INTERVAL=0

/shop/search.php
<?php
$keyword='unknown';
$INDEX_INTERVAL = 10000;
$error = "sorry, no such product";
// sets the INDEX_INTERVAL to zero
extract($_GET);
$products = get_all_products_from_db();
if(in_array($keyword,$producs)) {
  echo array_search($keyword,$products);
} else {
  echo $error;
}
if(time()-$INDEX_INTERVAL>last_indx()) {
   run_indx();
}
```

Figure 4: An adaptation of Figure 3, showing how an unsafe call to `extract` can impact the control flow of a program with potentially costly impact for an application.

affecting globally set constants or configuration values. When grouping the potential impact `extract` can have on a local or global scope, we identified two broad impact categories:

Injection A classic injection attack is the first impact category of `extract`. As our example provided in Figure 3 demonstrates, `extract` can overwrite the value of the `$error` variable that is then later used for output, thus, resulting in a classic reflected XSS vulnerability. However, it is not required that the impacted variable be directly used for output; it can be used as input to functions such as *PDO*, a SQL interface, or *exec*, the PHP-provided interface for running system commands on the command line.

CFG Manipulation The second option for an attacker is to influence variables that are not used as input to functions but that affect the control flow. This can lead to the program taking a vastly different path through the program as intended, with corresponding costs and effects.

Figure 4 shows an adaptation of our running example to demonstrate this issue. Line 16 decides whether to (re)index the database based on the last time the index was refreshed and the current time. If the last time the index was refreshed was sufficiently long ago, a potentially costly reindexing is executed. By overwriting the variable `$INDEX_INTERVAL` an attacker can trigger the costly indexing with each request, degrading the availability of the service.

Another potential and significant impact of CFG manipulation due to insecure usage of `extract` is privilege escalation. An attacker can leverage overwriting specific variables that

```
GET /shop/search.php?IS_ADMIN=true

/shop/search.php
<?php
$keyword='unknown';
$IS_ADMIN = $_SESSION['is_admin'];
$error = "sorry, no such product";
//sets IS_ADMIN to true
extract($_GET);
$products = get_all_products_from_db();
if(in_array($keyword,$products)) {
  echo array_search($keyword,$products);
} else {
  if($IS_ADMIN) {
     create_new_product($keyword);
  } else {
     echo $error;
  }
}
```

Figure 5: An adaptation of Figure 3, showing how an unsafe call to `extract` can allow for privilege escalation, in this case allowing an attacker to change the catalog of the web shop.

affect a user's permissions in the application. Figure 5 shows how a privilege escalation can manifest in an adaptation of our running example. In this specific code snippet, if a product is not available, the administrator can create a new product in lines 14 and 15. This allows an attacker to manipulate the catalog of the attacked shop by simply overwriting the `$IS_ADMIN` variable and escalating their access privilege to admin. Given this misuse potential, we want to know how often `extract` is used in an unsafe way in real applications.

3 Analyzing Extract in PHP Source Code

We want to understand the prevalence of `extract` usage and determine how often user input reaches a call to `extract`. To be able to scale this analysis, we implemented a static analysis tool using the Code Property Graph implementation by Wessels et al. [30] and based on the concept proposed by Yamaguchi et al. [31].

3.1 A Code Property Graph

Yamaguchi et al. [31] proposed the Code Property Graph (CPG) for static code analysis as a merge of multiple graphs to analyze C code. They later followed up with a corresponding implementation for PHP [12]. However, that implementation is dated and does not follow the established CPG standard [7]. Addressing this, Wessels et al. [30] published an up-to-date version of a PHP CPG implementation based on PHP Bytecode.

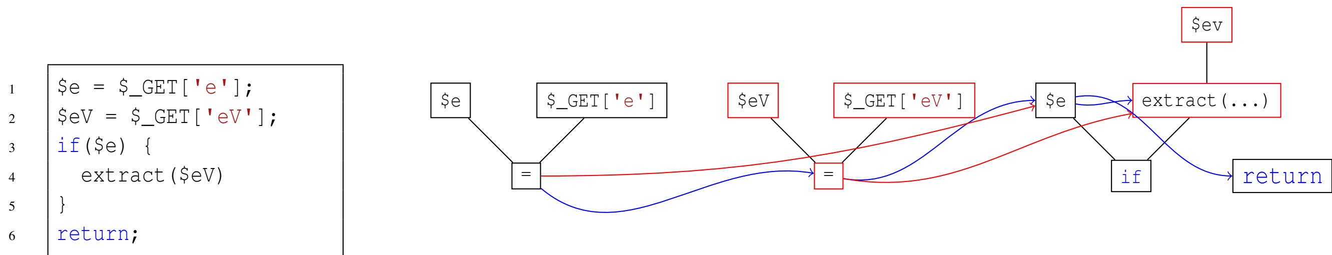

Figure 6: The creation of a CPG starting from an example code snippet on the left, resulting in the CPG on the right with AST in black, CFG in blue, and DDG in Red, and the program slice nodes starting at `extract` colored in red.

A CPG is primarily the combination of the three major graph code representations: Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Dependency Graph (DDG). The AST represents the structure of the code in the form of a tree and forms the basis of the CPG. Both the CFG and the DDG are layered into the AST by reusing its nodes and adding only additional, differently labeled edges. The CFG represents the program execution flow through the program and connects the AST nodes in the order in which they are executed during a regular program run. The DDG represents the data dependencies between nodes. For example, if node A defines a variable that is later used in node B, a DDG edge represents this dependency. If, in between A and B, there is a node A' that overwrites the variable set in A, then the edge connection is made between A' and B instead. Figure 6 visualizes this process using a small example code sample.

For the purpose of our analysis, we use the CPG published by Wessels et al. [30]. It is implemented against the CPG standard, and they also published additional tooling such as utilities to generate program slices and scale analysis. Besides it being the most current implementation, its main difference against the old CPG by Backes et al. [12] is that it is based on the PHP Bytecode instead of source code. This flattens out the AST, reducing the semantic complexity of each individual node. It also allows using the PHP interpreter-generated CFG instead of relying on a custom implementation.

3.2 From a Program Slice to Vulnerability

To analyze an `extract` call contained in a program, we have a CPG for, we used program slicing. Program slicing is a technique that uses the Data Dependency Graph to calculate all statements that may affect the input to a statement. The statement of interest is called the slicing criterion. Starting at the slicing criterion, a program slice backtracks all nodes that are connected via a data flow edge. This reduces the problem of identifying a data flow-based vulnerability to only a smaller subset of involved nodes. Figure 6 shows this subset by color coding all nodes that belong to the slice of `extract` red.

Our static analysis takes a CPG for a program, identifies all `extract` calls, and uses each call as a slicing criterion. We then analyze the resulting program slices for any usage of user controlled superglobals, i.e., (`$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, and `$_FILES`). In case a corresponding super global is part of the slice, the attacker potentially influences the input of extract leading to severe security implications (ref. Section 2.3).

However, the final exploitability of the identified `extract` calls highly depends on the context they are used in. A variable used after the call to `extract` must not be overwritten in the code. Additionally, the usage has to be in a vulnerable context, such as in creating a SQL query or as a condition for an `if`. While identifying variable use can be automated, understanding the context in which they are used is not universally trivial. Consequently, we use all potentially vulnerable program slices as a starting point for a manual investigation into their security implications.

Starting from the call to `extract`, we search the subsequent code for variables that were read but not written to. Variables (re)written after the call to `extract` before they are used cannot be controlled by the attacker and are safe from manipulation. Depending on the used flags, we looked at the previously declared variables above the `extract` call to gather the set of possible target variables. Variables containing objects are usually out of scope too, because it is only possible to write strings, which can be coerced to other primitive types using type juggling [26], and arrays. If we overwrite a variable containing an object, the script will abort on the following line when trying to access the object's methods or properties. The current scope of the `extract` call influences all other files that are imported afterwards using `require()` or `include()`. Naturally, we also looked at the global scope of the included files for possible vulnerabilities.

4 Results

To understand the prevalence and possibly unsafe usage patterns of `extract` we applied our static analysis on 30 870 open source GitHub repositories. We first looked at the general `extract` usage, followed by an analysis of popularity of the different flags. Finally, we looked at general exploitability to understand how often our two main vulnerability types are present and in what kind of vulnerabilities they manifest.

4.1 Data Set and Setup

We used the data set by Wessels et al. [30] which was compiled using PHP repositories from GitHub in 2023 by downloading all projects that had 26 stars or more. This totaled to 30 870 repositories. The timeout during CPG creation was set to 10 minutes, leading to the successful creation of 28325 CPGs we then subsequently analyzed.

Our analysis ran on an AMD EPYC 7713 with 200 GB RAM using 12 processes in parallel. Overall, we successfully analyzed 28158 CPGs. The taint-flow analysis failed for 167 repositories using `extract` due to timeouts or aborts in the CPG toolkit by Wessels et al. [30], which we analyzed for possible user input manually instead.

4.2 Extract usage

Our search for extract usage resulted in 1331 repositories containing at least one call to `extract` with a total of 4934 usages. The most prevalent usage in a single repository were 243 distinct calls.

By applying our analysis on the CPGs we identified 146 calls to extract that are using user input. This means that user influenced values, i.e., values from the superglobals (`$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, `$_FILES`), were passed into extract. As the superglobals are attacker controlled this can have security implications, i.e., an attacker can send arbitrary key value pairs that are then subsequently used. The range of variables accessible to the attacker can vary depending on the flags used for the `extract` call and thus needs to be assessed prior to any vulnerability analysis.

4.2.1 Used Flags

The fact that user input is passed to `extract` does not have to be a security problem in itself. Using one of the `EXTR_PREFIX_SAME` or `EXTR_PREFIX_ALL` flags causes all newly created variables to be separated from existing ones and `EXTR_SKIP` skips all variables that were written before. Thus, either flag prevents an attacker from overwriting an already existing variable (ref. Section 2.3). We collected the flags passed to all `extract` calls in the whole dataset. Figure 7 shows the flag usage across all calls and in calls our taint-flow analysis considered potentially controllable through user input.

Overall, 88.24% of the total `extract` calls and 63.45% of the potentially vulnerable calls do not specify a flag, causing it to default to the potentially unsafe `EXTR_OVERWRITE` flag. On the other hand, the same flag is set 2.11%/1.38% times explicitly. Developers ignore all configuration options to restrict `extract` most of the time. The secure prefix flags `EXTR_PREFIX_SAME` and `EXTR_PREFIX_ALL` are used 0.52%/1.38% and 0.72%/8.28% of the time, showing a small adoption of safer `extract` usage patterns, while the `EXTR_PREFIX_INVALID` flag is not used at all in the data set. Flags to restrict changes to old or new variables, `EXTR_IF_EXISTS` and `EXTR_SKIP`, appear more often when user input is anticipated with 0.7%/17.93% and 6.61%/7.59% of the calls. All vulnerable calls using `EXTR_IF_EXISTS` were in the same project, following a duplicated code pattern.

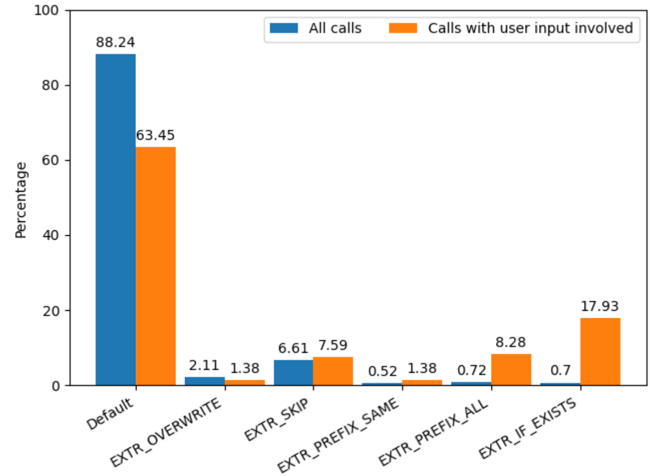

Figure 7: Usage share of each `extract` flag in calls for the whole dataset and the part which is deemed vulnerable.

4.2.2 Exploitability

Being able to overwrite arbitrary variables is a strong attack primitive, opening the application to a wide range of common web vulnerabilities. We identified two overarching vulnerability types such an attack can have (ref. Section 2.3), namely *Injection* and *CFG Manipulation*.

In Section 4.2 we reported 146 calls that are vulnerable to data flow from user controlled superglobals. Our first step was to verify that user input can flow into `extract` and affect subsequently used variables. We then manually assessed each identified `extract` call and sorted the possible effects into either type. Additionally, we further split up the injection vulnerabilities into the observed specific vulnerabilities. This resulted in observed XSS, SQLi, Command Injection, Open Redirect, and SSRF injection vulnerabilities. We split the CFG manipulation type into privilege escalation and simple CFG manipulation, i.e., the ability to abuse the `extract` call to alter the execution flow by changing variables that have no explicit relation to user input.

We identified a total of 218 different vulnerabilities across the 146 vulnerable calls. There can be multiple opportunities for exploitation from a single vulnerable call to `extract` which leads to a higher number of possible attacks than there are vulnerable `extract` calls. Table 2 shows the number of identified vulnerabilities per vulnerability type.

Injection Most vulnerabilities are injection-based, with 81 possible cross-site scripting and 65 SQL injection vulnerabilities. Since the data flows are triggered using HTTP web requests, the applications are expected to implement websites rendered using HTML and backed by a database, which can fall victim to injection vulnerabilities. XSS can be found in numerous instances because the attacker can change variables that are outputted without sanitization, since they are expected to have static values, like error messages or response codes. When a vulnerability is present in the code, it is usually also possible to get a value reflected to the user and thus gain cross-site scripting capabilities. Only 9 of the 81 identified XSS vulnerabilities were the sole type of vulnerability associated with the `extract` call, a different vulnerability accompanied all other ones. SQL Injections were the second most common vulnerability type, as again, the developer expected variables potentially affected by `extract` and used in queries to not be user controlled. The remaining three injection vulnerabilities were rarer, as the corresponding APIs are infrequently used.

CFG Manipulation The vulnerabilities in the other category all allowed the attacker to trigger control flow that should be inaccessible, protected by logic or access control. 60 of the `extract` calls permitted the attacker to work around access checks in the session with a total of 86 possible calls leading to control flow manipulation. The impact of writing to the session variable could be in a different file or code path, that can be triggered using a second request. The fact that `extract` could be used to set the access control fields is enough to count it as a privilege escalation. We did not check every usage of the session variable for additional vulnerabilities introduced by arbitrarily controlling the session because of the size of the projects. While most privilege escalation vulnerabilities lead to a CFG manipulation in the same file, some vulnerable `extract` calls can overwrite other local variables used in subsequent branches. This allowed chaining multiple actions in the same file while controlling the variables, even when the other code path did not use `extract`.

Limitations `extract` based security vulnerabilities can be subtle and easy to miss. While we did our best to spot all possible ways to influence the code, it is still a possibility that we missed some. Thus, the results presented in Table 2 represent a lower bound of the options.

Furthermore, we focused our effort on first-order vulnerabilities, i.e., a direct impact of overwritten variables. A subtle attacker can use `extract` to persistently change the application state or inject values whose influence materializes at entirely different and even for the developer unexpected points. One example would be using the influence over a SQL query to inject a XSS payload, that is later on retrieved and returned to the users of the underlying web application. We consider such second-order vulnerabilities to be out of scope.

Type	Vulnerability	#
Injection	XSS	81
	SQL Injection	65
	Command Injection	3
	Open Redirect	2
	SSRF	3
CFG Manipulation	CFG Manipulation	86
	Privilege Escalation	60

Table 2: Possible attacks found using the reachable `extract` calls.

5 Case Studies

The static analysis identified calls to `extract` involving user input, including the matching data flow from said user input to the call site. However, the raw numbers do not properly show the impact on the surrounding code. To showcase the impact and analyze the thought process (or lack thereof) by the developer, we manually evaluated the possible exploit scenarios for every `extract` call and report a selection of case studies for the different vulnerability categories to highlight the versatility of vulnerable `extract` calls.

5.1 An SQL Injection Case Study

To highlight the subtle ways an unsafe call to `extract` can compromise the developers' assumptions, we consider a snippet from the dataset vulnerable to SQL injection in Figure 8. A HTTP request to trigger the vulnerability is given at the top. Since `extract` allows the attacker to change the value of any variable in the scope, she can touch variables the developer considers constant. The value of `$table` is defined at the top of the code in line 7 from a static configuration with no user input involved. This leads the developer to falsely assume that the value is safe to use in critical contexts like SQL queries in line 21. Variables considered user-controllable are sanitized using a prepared statement, which can be interpreted in favor of the developer to consider safe coding practices on untrusted input. The table name, however, is concatenated verbatim, leading to a SQL injection.

The vulnerability in the snippet can be exploited as follows: when issuing the `"dp_act"` action (line 8) in a POST request to the application, the user can search for addresses to display or delete the selection. By setting the `$delete` POST parameter to `"Delete Address"`, the address deletion logic can be reached while controlling the value of all other variables (line 13). Ignoring the intended search logic which involves crafting a secure prepared statement in line 16, the attacker instead passes a `$table` POST parameter to allow the deletion of records from any other table. An example request is listed starting on line 1, deleting all users from the application.

```
POST /address.php

action=dp_act&delete=Delete+Address&table=users

/address.php
<?php
$table=get_settings_value("table_address");
$action=$_POST['action'];
if ($action=="dp_act")
{
  extract($_POST);
  // ...
  }else if($delete=="Delete Address"){
    $sql_query="";
    $qvalues = array();
    if ( $_POST['address_src'] != "" ) {
      $src_ip = $_POST['address_src'];
      $sql_query .= " AND ip like ?";
      $qvalues[] = $src_ip;
    }
    $sql = "DELETE FROM ".$table." WHERE (1=1)
    ↪ ".$sql_query;
    $stm = $link->prepare($sql);
    if ($stm->execute($qvalues) === false)
      die(...);
  }
}
```

Figure 8: An exploitable code snippet containing a SQL injection vulnerability despite using prepared statements.

This injection vulnerability can be attributed to the implicit definition of variables caused by `extract`. The developer was unaware that the `$table` variable can be changed through `extract` and did not use sanitization when inserting the variable into the query. The vulnerability could have been prevented using the `EXTR_SKIP` flag to skip overwriting variables that are already written to before. Since the surrounding code is accessing POST parameters by explicitly indexing into the `$_POST` variable already, having the `extract` call at all seems like it is a relic from past versions of the code that is not required nor used anymore. This highlights one of the dangers frequently mentioned when developers use `extract`: even if the code is not exploitable by the time the `extract` call is introduced, it is a foot-gun waiting to unfold when the code evolves and gets refactored or extended.

5.2 An Open Redirect Case Study

Relying on server metadata expected to be configured in the server software hosting the PHP code can lead to a different kind of injection vulnerability. Using documented values in the `$_SERVER` or `$_ENV` arrays becomes dangerous once `extract` is used on user input. In the context of open redirect

```
GET /index.php?_POST[sPhrase]=
↪ &_SERVER[REQUEST_URI]=shadyshop.com

/index.php
<?php
extract( $_GET );
define( 'CUSTOMER_PAGE ', true );
if( isset( $_POST['sPhrase '] ) ){
  header( 'Location: '.$_SERVER['REQUEST_URI
  ↪ '].'&sPhrase='.
      urlencode( $_POST['sPhrase '] ) );
  exit;
}
```

Figure 9: PHP source code of a reachable extract call in a web shop with an open redirect vulnerability.

vulnerabilities, crafting an innocent link which looks like it is hosted on the vulnerable website could forward the victim to an attacker-controlled copy of the website to phish for personal information. Instead of hard-coding the domain in a configuration file, developers opt to use the metadata of the execution environment as a base for relative paths or other settings. This is convenient for the attacker because she can replace otherwise read-only information that is expected to contain known values with arbitrary attack payloads.

One of our findings contains the code depicted in Figure 9. Here, the whole page implements a shopping website. The bare `extract($_GET)` allows an attacker to craft a convincing phishing link. For this, we know we have complete control of all variables starting from line 5, including the value of *all* superglobals. Using this information, we can craft a link that sets the POST parameter required in line 7 and overwrites the value of `$_SERVER['REQUEST_URI']` which usually holds the relative URI of the page a user requested. This way we can abuse the existing redirect to point to an attacker-controlled website. A GET request exploiting the vulnerability is included in line 1. Here the exploit makes the server on the hosting site redirect the user to `shadyshop.com`.

Even though the `extract` call is in global scope, the same exploit would also work inside a function. The difference is that the changes to `$_SERVER` would not persist outside the function scope. Still, it is possible to overwrite the global variable with a local version used in the following code inside the function. Furthermore, it grants access to other variables that would typically not be filled by the PHP runtime for the request. Even when the attacker sends a GET request with the query parameters decoded into `$_GET`, `extract` allows to store values into the `$_POST` array too. This allows for triggering code paths generally requiring a POST request through a GET request.

5.3 A Privilege Escalation Case Study

PHP access controls require users to authenticate themselves using some shared secret sent in the HTTP request. That shared secret, commonly a username and password, is checked against an authentication provider such as a user database, and access is granted or denied. Instead of repeatedly sending the same authentication information with every request, the privilege level or login status is commonly persisted in a session tied to a session identifier stored in a cookie. In PHP, the session is an opaque server-side key-value store, which gives developers access to the data saved from previous requests through the `$_SESSION` superglobal. Most importantly, the client only gets access to the session identifier but cannot change the session data manually. This guarantee is broken when `extract` is used on user-controlled data in the global scope. The user could specify to overwrite the `$_SESSION` superglobal, replacing the whole session state, and add, edit, or remove any internal session variables. One caveat is that the session has to be initialized using the `session_start()` function before `extract` is called, otherwise the changes are not persistent. One of the identified privilege escalation vulnerabilities is displayed in Figure 10 as a case study.

The code snippet is from a forum software using the session mechanism to store the administrator state of the current user. First, the session is started in line 5 after which the default values for the session variables are initialized in line 6. This only adds new keys to the array not already in the session. Later in the file `extract` is called on user-controllable data while providing default values for the expected variables in line 8, similarly to the way the session was initialized. The author expected the request to contain only some of the variables but did not consider a malicious user to send additional ones. This oversight can be abused to set the `$_SESSION` variable to include the `"admin"` key which is checked in line 7. The attacker can use the newly gained power to delete any forum topic or user. An example request triggering the vulnerability is given in line 1. The request sets the `"admin"` field in the session and deletes the user with the email address `admin@example.com`.

Overwriting an array of values is only possible due to PHP's rich support for parsing form data into structured arrays [23]. Normally, this is used for multi-select input fields or grouping form fields, but it also enables attackers to pass arrays for any given query parameter. Next to passing normal parameters `?var=1` in a GET or POST request that can be accessed using `$_GET["var"] == 1`, PHP supports parsing nested arrays. `?var[]=1&var[]=2` results in an array of `$_GET["var"] == ["1", "2"]` and `?var[key]=val` in `$_GET["var"] == ["key"=>"val"]`, including support for multi-dimensional arrays. When specifying a key name of `?_SESSION[admin]=1`, the parameter is decoded into an array of `$_GET["admin"] == ["_SESSION" => ["admin"=>"1"]]` and calling `extract` on the `$_GET` array would overwrite the `$_SESSION` variable with the specified array.

```
1  GET /forum.php?delete=user
    ↪ &email=admin@example.com&_SESSION[admin]=1
2
3  /forum.php
4  <?php
5  session_start();
6  $_SESSION += array('email' => '', 'admin' =>
    ↪ '', 'trusted' => 0, 'check' => '',
    ↪ 'posts' => 0);
7  //...
8  extract($_REQUEST + array('email' => '',
    ↪ 'topicID' => 0, 'commentID' => 0, 'title'
    ↪ => 0, 'body' => 0, 'delete' => 0));
9  //...
10 if($delete && $_SESSION['admin'])
11 {
12   if($delete == 'topic') {
13     //...
14   } elseif($delete == 'user') {
15     query('UPDATE user SET banned = 0 WHERE
        ↪ email = ?', array($email));
16   } else {
17     //...
18   }
19 }
```

Figure 10: An exploitable code snippet allowing to overwrite the `$_SESSION` for privilege escalation.

The consequences of changing the session data do not have to be reachable in the same code path that was used to trigger a call to `extract` in the global scope. The changes persist and are retrieved during subsequent requests to other endpoints in the application to escalate privileges. This allows accessing any code paths guarded by any session values - even if `extract` is not used in that part of the code! On the other hand, privilege escalation vulnerabilities are not limited to implementations using the `$_SESSION` to store the login information. Still, they can apply to any predicate variable set before calling `extract`. The concrete steps to access protected code paths are highly dependent on the analyzed application and the implementation details of the authentication mechanism.

5.4 Safe handling of user input

While the focus of this section until now was on vulnerable code patterns and the unsafe use of `extract`, we also want to include a positive example of how user input is handled properly, as this showcases that not everything is lost when `extract` is used. Safe usage either involves keeping

```php
<?php
// ...
extract($_GET, EXTR_PREFIX_ALL, 'i');
extract($_POST, EXTR_PREFIX_ALL, 'i');

$method = explode('?',basename(REQUEST_URL));
$method = $method[0];
if (strlen($method) < 1 ) {
    $method = $i_method;
}
```

Figure 11: A code snippet handling user input safely in `extract`.

the function containing the `extract` call short and writing to all variables after the call. Or use one of the `EXTR` flags to avoid overwriting unexpected variables. The code in Figure 11 is a snippet of the JSON API of a data store application as a counter example demonstrating how developers can use `extract` safely.

To put variables created by `extract` from user input into their own name space without overwriting other variables, the `EXTR_PREFIX_ALL` flag is used with a prefix of `'i'` in lines 3 and 4. This causes the names of the extracted variables to include the prefix followed by an underscore. Invoking the script using a parameter of `?method=foo` causes `extract` to define a variable `$i_method` instead of `$method`. The code expects a prefixed variable in line 9, using its value in a controlled way. It is important to specify a prefix when using one of the `EXTR_PREFIX_` flags, because an empty prefix of `''` would result in the possibility to override superglobals again. Variables would just be prefixed by an underscore, which matches the pattern of `$_SESSION` and `$_REQUEST` when passing the name without the leading underscore.

The code in the example uses it to easily merge different forms of user input without the need to write explicit logic to do so. It correctly deploys the `EXTR_PREFIX_ALL` flag and uses `extract` safely. This cannot be said for every occurrence of `extract` in the data set.

6 Discussion

When used with user input, we have shown the dangers of `extract` and the subtle bugs it can introduce into the program. This leads us to question whether developers know how to use `extract` safely and how the issues can be mitigated without increasing the complexity. In this chapter, we discuss the consequences of our results, compare the attack surface to other languages, and propose changes to `extract` to make it less likely for it to become a foot-gun.

6.1 Reasons to use extract

While looking at repositories where our static analysis did not find user input involved in the call, multiple patterns of safe usage started to emerge that align with other analysis [22]. A recurring theme is to have a single `$args` argument to a function instead of multiple ones, passing a static array to it and extracting the argument inside the function. This code pattern appears popular in WordPress plugins, a popular content management software, despite being discouraged in the official WordPress coding standards [13]. Sometimes, the code was copied from the official WordPress core and stayed the same after `extract` was purged from the upstream code [29] with the occasional addition of a comment to suppress a linter warning about `extract` instead.

Another common use case is extracting the result of a database query or function returning an array with known keys such as `parse_url` or `unpack`. The same applies to custom functions returning multiple values through an array directly, or use `extract`'s counter-part `compact` to coalesce multiple variables into an associative array automatically. Template engines use `extract` in their `render` functions on the template variables before including the template itself. It allows the engine to be generic and support any arguments made available to the template. In this case, rendering user input safely is a challenge unrelated to `extract` itself.

Most of these use cases do not require `extract` but are solely a choice of coding style. This code pattern is frequently criticized by developers dealing with code using `extract` written by other people, because it obfuscates where a variable comes from [15, 21]. Code editors cannot highlight variable definitions and the variables appear uninitialized. It raises the mental burden to understand where a value is modified and how it ended up in the current function. This use of `extract` does not make the code unsafe, but as a tool in the developer's toolbox, it could also be used on user input out of convenience. The developers must understand `extract` as a tool and know when to use it safely.

6.2 Developer Awareness

The statistics on the used `EXTR_` flags in Section 4.2.1 reveal the majority of developers using `extract` without specifying any flags. This does not necessarily have to be a problem, when the user has no control over the array keys passed to `extract`. It becomes one when developers use it as a standard language feature without considering possible side effects. Sahin et al. [20] conducted a survey on web developers to assess their knowledge of possible attack surfaces and mitigation techniques. Only 67% of the developers considered all parts of the HTTP request user-controllable. Even a developer aware of the security problems can introduce vulnerabilities due to confusion of the PHP variable scope semantics.

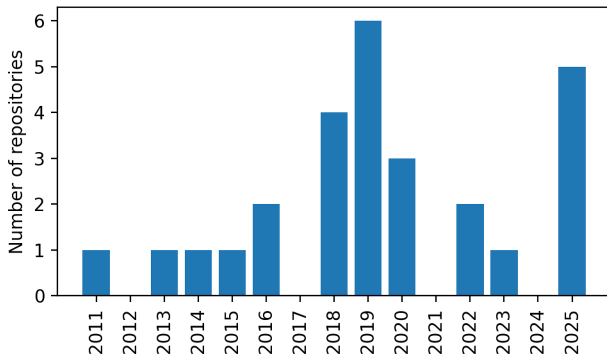

Figure 12: Number of vulnerable repositories that had their last Git commit in each year.

6.2.1 Timeline

extract is available in the standard library for a long time and appears in the data set of vulnerable PHP repositories from 15 years ago up until now. To determine if the repositories found with vulnerable extract usage are a relic from the past or still active today, we looked at the date of their most recent activity.

Figure 12 shows the number of vulnerable repositories grouped by the year of their last commit on GitHub. 18.52% of the repositories are still active in 2025 and 40.74% repositories had a commit in the past 5 years since 2020. The oldest repository has not changed since 2011 and 2 repositories, with last commits in 2016 and earlier, were already archived and are now read-only on GitHub. extract misuses persist in projects under active development up until today which shows that developers keep having trouble to use extract safely with user input or leave vulnerable calls in the code base by accident. The disclosure process is ongoing and the *Little Backup Box*[14] project fixed the vulnerability already, while another acknowledged the problem.

6.2.2 Flag Misuse

Specifying EXTR_ flags when using extract does not automatically make the code safe. Each flag's semantics could take unexpected turns due to PHP's dynamic nature, which requires thoroughly evaluating the exposed variables. One repository had a repeating pattern of using the EXTR_IF_EXISTS flag while defining only the expected input variables in advance. The beginning of the login page is shown in Figure 13.

At first glance, the developer appears to know how to use extract safely, since the call is at the top of the file with no unrelated variables declared. The problem is introduced when require is called beforehand, bringing all globally defined variables and functions in the included file into the current scope. Now the list of available existing variables

```
1 <?php
2 require ('../include/init.inc.php');
3 $user_name = $password = $remember =
  ↪ $verify_code = '';
4 extract ( $_POST, EXTR_IF_EXISTS );
```

Figure 13: An exploitable code snippet allowing to overwrite more variables than the developer intended.

is way longer than the developer anticipated and allows in this case to overwrite the $_SESSION variable for privilege escalation despite the presence of EXTR_IF_EXISTS. Even the most secure EXTR_PREFIX_ALL flag can allow attackers to control other variables when using the empty string as prefix (see Section 5.4).

6.2.3 Attacker Awareness

We did observe multiple instances in which the usage of extract was unsafe. But to become a problem, attackers need to be aware of this attack vector. When analyzing our dataset we observed that 8 of the vulnerable extract calls were from training courses or Capture the Flag (CTF) challenges, including the use of EXTR_ flags. The calls were unsafe intentionally to teach exploit techniques and showcase harmful code patterns. Thus, the security community and by extension attackers have a high chance to know this possible attack vector and have the technical skills to exploit an insecure extract call in addition to potential private research done. In contrast to the difficulty of safely using extract with user input, developers have a challenging job to defend their applications against attacks. The obvious solution would be not to use extract when user input is involved. In case of libraries, where the context in which projects use their function is unknown to the library developer, safe code patterns must be deployed when insisting on using extract.

6.2.4 Restricting the keys

To avoid problems with uninitialized variables when parts of the expected input are omitted, developers must specify default values for all input variables. Furthermore, any additional, unexpected keys in the array must be removed before the resulting array is finally passed to extract. WordPress includes the shortcode_atts function to perform those steps [27]. Implementing them securely bloats the code more than it saves time for not having to type out array accesses.

The EXTR_PREFIX_ALL flag solves the problem of potentially overwriting unwanted variables, given a unique prefix is chosen that does not appear elsewhere in the code, but still leaves the potential of uninitialized variables when one is omitted in the input. The cleanest solution would be to

have extract itself take care of the default values and excess variables.

Using default linter rules, like the one deployed by WordPress using the CodeSniffer linter [24], to discourage extract usage, will reduce extract in new code, and bring existing scenarios up for review. Including details on the attack surface and examples of safe and unsafe usage of the different flags can help motivate the need to think about extract. Since extract is such a specific feature to PHP, developers cannot use their experience from other languages to guide their assessment.

6.3 Similar features in other languages

The attack surface of extract is unique but draws parallels to similar unsafe code patterns in other programming languages. The prototype pollution vulnerability class related to overwriting and adding properties to objects in JavaScript can be compared to extract in global scope [11]. When user input is merged with an object, it could change the __proto__ property and inject variables into all instances using the same prototype. It requires a vulnerable merge function which is usually provided by libraries. Since its introduction in 2018, many JavaScript libraries were patched to disallow writing of the __proto__ property.

Another attack surface that allows the mapping of HTTP request parameters to variables is automated data binding in Java web frameworks. It allows to define a class with a list of member variables which get filled by request parameters with the same name during runtime using annotations. When implemented poorly, other sensitive member variables including the ones of parent classes can be exposed leading to compromise [32]. Similar to prototype pollution bugs, the problem was fixed by preventing binding request parameters to sensitive Java internals.

The solution cannot be applied directly to PHP because no class hierarchy is involved in a vulnerable extract call. The closest are superglobals which have the potential to influence other code paths than where extract is called.

6.4 Defenses

Considering the observed use cases and ways extract is used in open source applications, removing the function from the standard library is not a valid option, when backwards compatibility is involved. A proposal to deprecate extract in 2017 was rejected [28]. Despite its controversial history, the function was considered a valid tool for some use cases. Instead, we propose different strategies and changes to PHP that would make extract usage more secure for the average developer.

Handling of global state Our results show the prevalence of possible vulnerabilities involving the superglobals $_SESSION or $_SERVER when extract is called in the global scope. To mitigate the risks, extract could be modified to disallow writing to superglobals. The runtime fills the variables and they should not be touched by user input again. While considering a similar solution to the ones applied in other programming languages in Section 6.3, we noticed the $GLOBALS superglobal is filtered out already since 2005 [25]. We could not find additional information on why only $GLOBALS was considered harmful. The filter logic broke during code refactoring, which led to the assignment of CVE-2011-0752, showing the role as a security issue. Assigning to $_SESSION, $_SERVER, and $_ENV is equally dangerous and gives attackers access to otherwise uncontrollable data. To further limit the attacker's capabilities, extract should not allow overwriting any superglobals.

Alternatively, using extract in global scope can warn developers of the possibility of modifying the global state. The warning should be possible to suppress if the user knows about the dangers and still wants to use extract explicitly. A new EXTR_GLOBALS flag can be added to potentially signal the conscious decision to overwrite global state. It must be passed when extract is used in the global scope.

No empty prefix Allowing an empty prefix when one of the EXTR_PREFIX flags is used can leave variables starting with an underscore at risk (see Section 5.4). Thus, we consider the lack of a prefix a bug. The PHP runtime should reject the empty string as a prefix and throw an error instead. This would reveal extract calls in code where developers forgot to specify a prefix and further harden the addition of a prefix to the variable names.

Default values The problem of extracting more variables from an array than required is the root cause of the vulnerabilities described in this work. Giving developers an option to specify the variables they expect out of an extract call would mitigate the risk of overwriting additional variables. A new $defaults parameter to extract can list the array keys to overwrite and also contain the default values of the variables.

7 Related Work

Static analysis of PHP applications is a welle stablished discipline. Huang et al. [17] were the first to propose information flow analysis in PHP to detect vulnerabilities with Jovanovic et al. [18] proposing using a static taint-flow analysis. The Code Property Graph was initially proposed by Yamaguchi et al. [31] and later on adopted for PHP by Backes et al. [12]. However, their implementation prototype was not kept maintained. Moreover, later improvements led to a different standard [7] against which Wessels et al. [30] implemented their PHP Code Property Graph that works on the Bytecode

level and which was also used by Al Kassar et al. [9] to detect code patterns detrimental to static analysis. We build upon the work of Wessels et al. [30] and use their implementation for our analysis. Besides direct work with PHP CPGs Hossain Shezan et al. [16] extended CPGs across languages to search for GDPR violations and Alhuzali et al. [10] combined CPGs with dynamic analysis to create exploits.

We reused the dataset by Wessels et al. [30] who leveraged stars as a metric to select GitHub repositories. While this is common practice Koch et al. [19] studied the connection between GitHub stars and download numbers and showed that the correlation between those two metrics is middling, indicating that this selection criteria might miss less renown but used repositories.

8 Conclusion

`extract` is a subtle PHP foot-gun with high potential for unforeseen consequences if used in an unsafe manner. In the course of this paper we have conducted a deep dive into the `extract` call and discussed how the attacker can use their influence on the input value to either conduct an injection attack or affect the subsequent execution flow. Either option posing an attack vector against a web application. Following our deep dive, we have presented a static analysis methodology to detect potentially unsafe calls to `extract` due to information flow from user controlled input. We successfully applied this methodology to 28158 repositories and detected 4934 calls to `extract`. Out of those calls 146 involved user input. A manual analysis of all these calls verified that 117 of those calls were indeed vulnerable, resulting in 154 injection vulnerabilities and 86 possibilities for the attacker to impact the execution flow. We subsequently selected a set of case studies to showcase how developers miss the impact of `extract` on their code, even if they seemingly deploy it in a safe manner, showing that `extract` is indeed a foot-gun. Finally, we proposed multiple solutions on how to make `extract` inherently safer without unduly impacting the backwards compatibility.

Ethics

We have started the process of responsible disclosure. We contact each maintainer of the affected projects either via the instructions given in a *SECURITY.md* or by opening an issue asking for further instructions. In case the project is clearly abandoned, we leave a single issue warning future users about a contained and unpatched security vulnerability.

Availability

Our work resulted in two artifacts. On the one hand, we have a list of extract calls, their possible user input, and the corresponding GitHub projects. On the other hand, we have our static analysis that generated the list. The code and list can be found at https://secartifacts.github.io/woot2025 after the conference due to ongoing disclosure.

Acknowledgments

We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972. Additionally, we would like to thank our student Marius Koch for initially implementing the CPG consumer that is used as a basis for this work and our system administrator Tobias Jost for providing essential and consistent technical support.

References

[1] register_globals, . URL `https://www.mediawiki.org/wiki/Register_globals`. accessed 2025-03-12.

[2] Php 4.2.0, . URL `https://www.php.net/ChangeLog-4.php#4.2.0`. accessed 2025-03-12.

[3] Php 5.3.0, . URL `https://www.php.net/ChangeLog-5.php#5.3.0`. accessed 2025-03-12.

[4] Php 5.4, . URL `https://www.php.net/ChangeLog-5.php#5.4.0`. accessed 2025-03-12.

[5] Php: Php and html, . URL `https://www.php.net/manual/en/faq.html.php#faq.html.arrays`. accessed 2025-03-12.

[6] Php: extract - manual, . URL `https://www.php.net/manual/en/function.extract.php`. accessed 2025-03-12.

[7] Code property graph specification website, . URL `https://cpg.joern.io/`. accessed 2025-03-12.

[8] Usage statistics and market share of php for websites, march 2025, . URL `https://w3techs.com/technologies/details/pl-php`. accessed 2025-03-12.

[9] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguch. Testability tarpits: the impact of code patterns on the security testing of web applications. In *Network and Distributed System Security Symposium*, 2022.

[10] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. Navex: Precise and scalable exploit generation for dynamic web applications. In *USENIX Security Symposium*, 2018.

[11] Olivier Arteau. Prototype pollution attack in nodejs application. In *NorthSec*. Olivier Arteau, 2018.

[12] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[13] The WordPress community. Don't extract(), 2015. URL https://developer.wordpress.org/coding-standards/wordpress-coding-standards/php/#dont-extract. accessed 2025-03-11.

[14] Stefan Saam Dmitri Popov. Little Backup Box , 2025. URL https://github.com/outdoorbits/little-backup-box. accessed 2025-05-09.

[15] Robert Enyedi. PHP bad practice: the use of extract(), 2008. URL https://dzone.com/articles/php-bad-practice-use-extract. accessed 2025-03-11.

[16] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. Chkplug: Checking gdpr compliance of wordpress plugins via cross-language code property graph. In *Network and Distributed System Security Symposium*, 2023.

[17] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *The international conference on World Wide Web (WWW)*, 2004.

[18] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. In *Journal of Computer Security*, 2010.

[19] Simon Koch, David Klein, and Martin Johns. The fault in our stars: An analysis of github stars as an importance metric for web source code. In *MADWeb*, San Diego, USA, 2024.

[20] Merve Sahin, Tolga Ünlü, Cédric Hébert, Lynsay A. Shepherd, Natalie Coull, and Colin Mc Lean. Measuring developers' web security awareness from attack and defense perspectives. In *2022 IEEE Security and Privacy Workshops (SPW)*, 2022.

[21] Joseph Scott. I Don't Like PHP's extract() Function, 2009. URL https://blog.josephscott.org/2009/02/05/i-dont-like-phps-extract-function/. accessed 2025-03-11.

[22] Damien Seguy. A story of compact and extract, 2024. URL https://www.exakat.io/a-story-of-compact-and-extract-2/. accessed 2025-03-11.

[23] Damien Seguy and Philip Olson. FAQ: How do I create arrays in a HTML <form>?, 2001. URL https://www.php.net/manual/en/faq.html.php#faq.html.arrays. accessed 2025-03-11.

[24] Squizlabs. PHP_CodeSniffer linter. URL https://github.com/PHPCSStandards/PHP_CodeSniffer/. accessed 2025-03-12.

[25] PHP Development Team. Filter for $GLOBALS in extract implementation, . URL https://github.com/php/php-src/blob/a14301b2d674abd32d136214ebb361e9d73104ed/ext/standard/array.c#L2017. accessed 2025-03-12.

[26] PHP Development Team. Type Juggling, 2005. URL https://www.php.net/manual/en/language.types.type-juggling.php. accessed 2025-03-11.

[27] Wordpress Development Team. Wordpress developer documentation for shortcode_atts, . URL https://developer.wordpress.org/reference/functions/shortcode_atts/. accessed 2025-03-12.

[28] Ilija Tovilo. [RFC] Deprecate the extract function in PHP 7.3. internals@lists.php.net Mailing List, 2017. URL https://externals.io/message/100637. accessed 2025-03-11.

[29] Viper007Bond. Remove all, or at least most, uses of extract() within WordPress, 2012. URL https://core.trac.wordpress.org/ticket/22400. accessed 2025-03-11.

[30] Malte Wessels, Simon Koch, Giancarlo Pellegrino, and Martin Johns. Ssrf vs. developers: A study of ssrf-defenses in php applications. In *33rd USENIX Security Symposium (USENIX Security 2024)*. Usenix, 2024.

[31] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[32] Xiaoyong Yan, Biao He, Wenbo Shen, Yu Ouyang, Kaihang Zhou, Xingjian Zhang, Xingyu Wang, Yukai Cao, and Rui Chang. Automated data binding vulnerability detection for java web frameworks via nested property graph. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2024.