Argus: All your (PHP) Injection-sinks are belong to us

So you have to listen to my presentation instead

Rasoul did all the work, but can't be here

Rasoul Jahanshahi

Manuel Egele

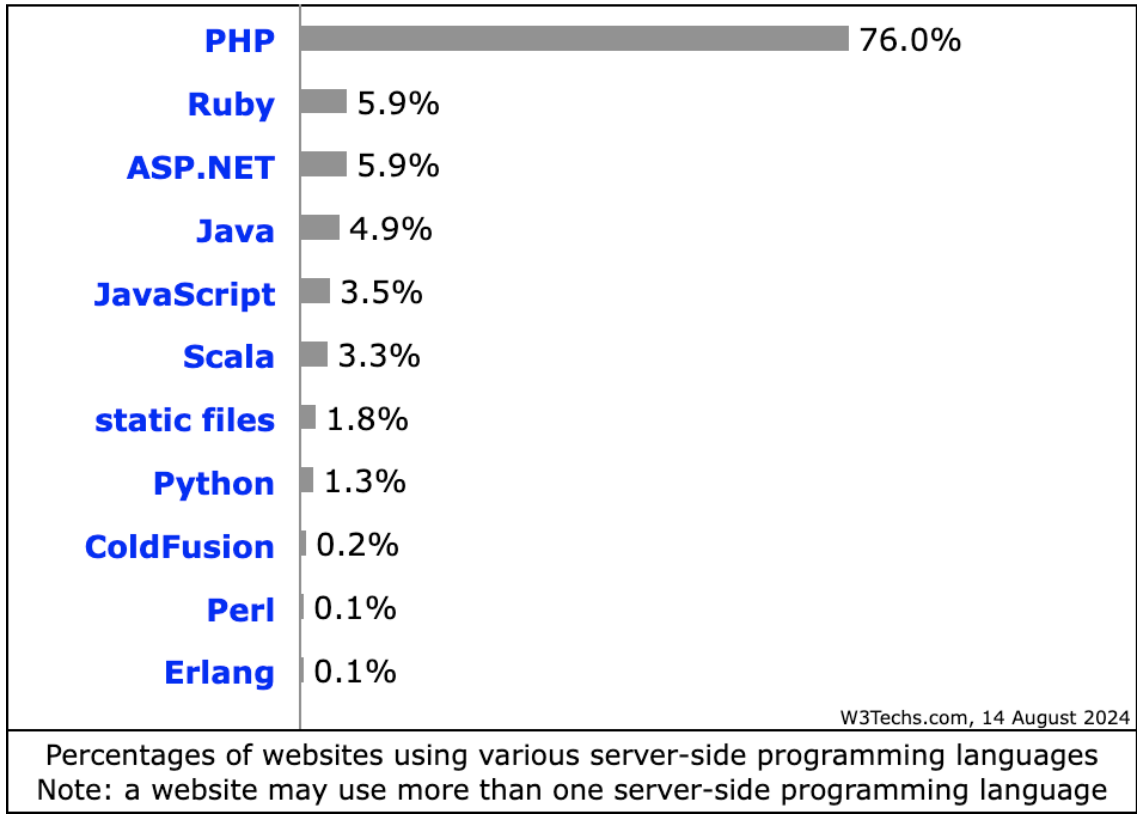
Department of Electrical & Computer Engineering

SeclaBU

BOSTON UNIVERSITY

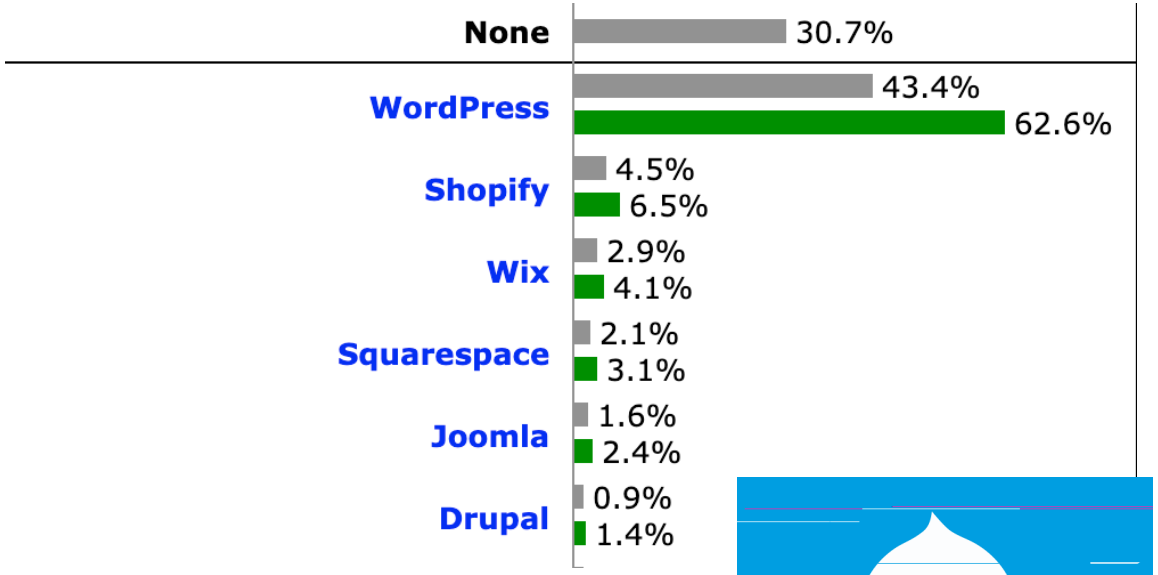
Web Apps are Prolific & Insecure

Server-Side Language



Percentages of websites using various server-side programming languages
Note: a website may use more than one server-side programming language

Market Share %/CMS %



Wordpress » Wordpress :

Versions Vulnerabilities (367)

Drupal™ WordPress Joomla!®

Find & Exploit Bugs/Vulnerabilities

1) Identify where attacker-controlled input enters the program
   (e.g., `$_GET`, `$_POST`, etc.)
2) Identify sensitive APIs leading to vulnerabilities
   (e.g., `system` (cmd inj.), `echo` (XSS), `unserialize` (POI))
3) Perform dataflow/taint analysis
   (track flow of attacker-controlled data to sensitive functions)

Exploit generation

- Hook to sensitive functions
- Track attacker-controlled data
- Identify available gadgets during their execution for exploitation

Sinks:
How do you find them?
Did you find all of them ?

Identifying Sinks for Taint Analysis

Observation

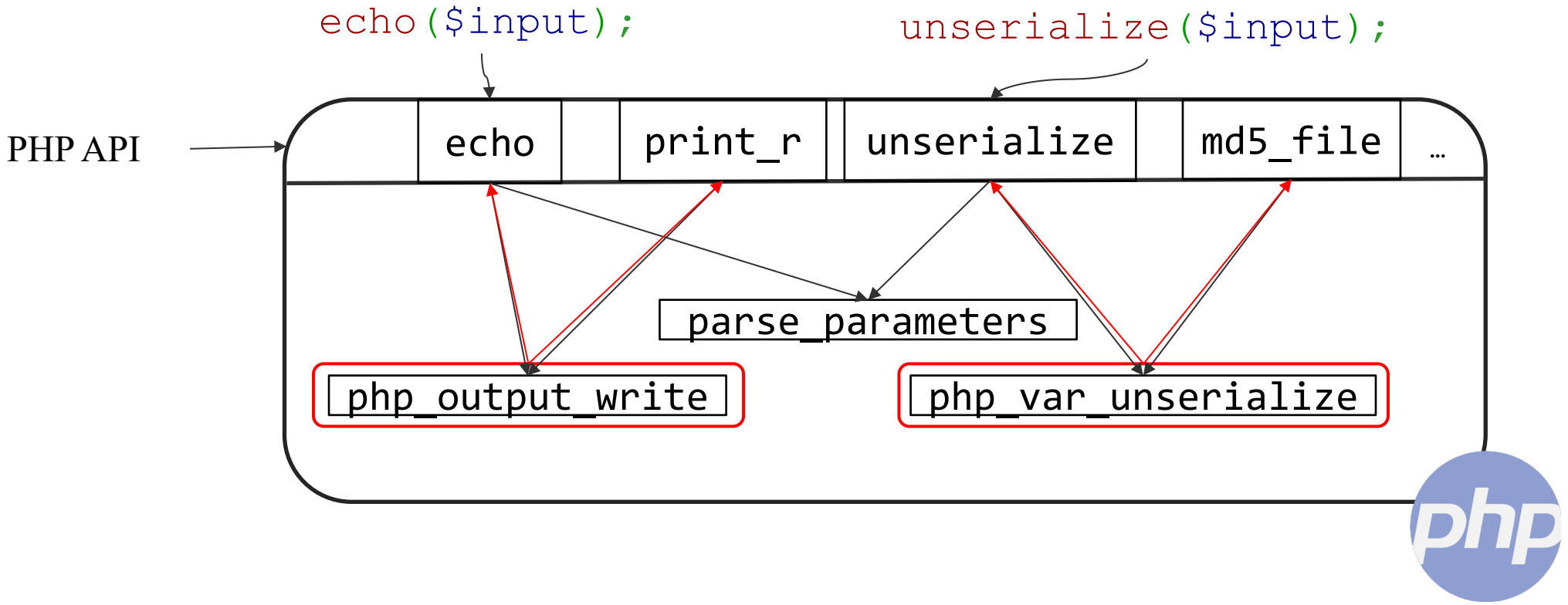
Most systems/papers rely on **manually curated** lists of Sources & Sinks
(e.g., knowledge/experience of authors, scanning docs, etc.)

Question

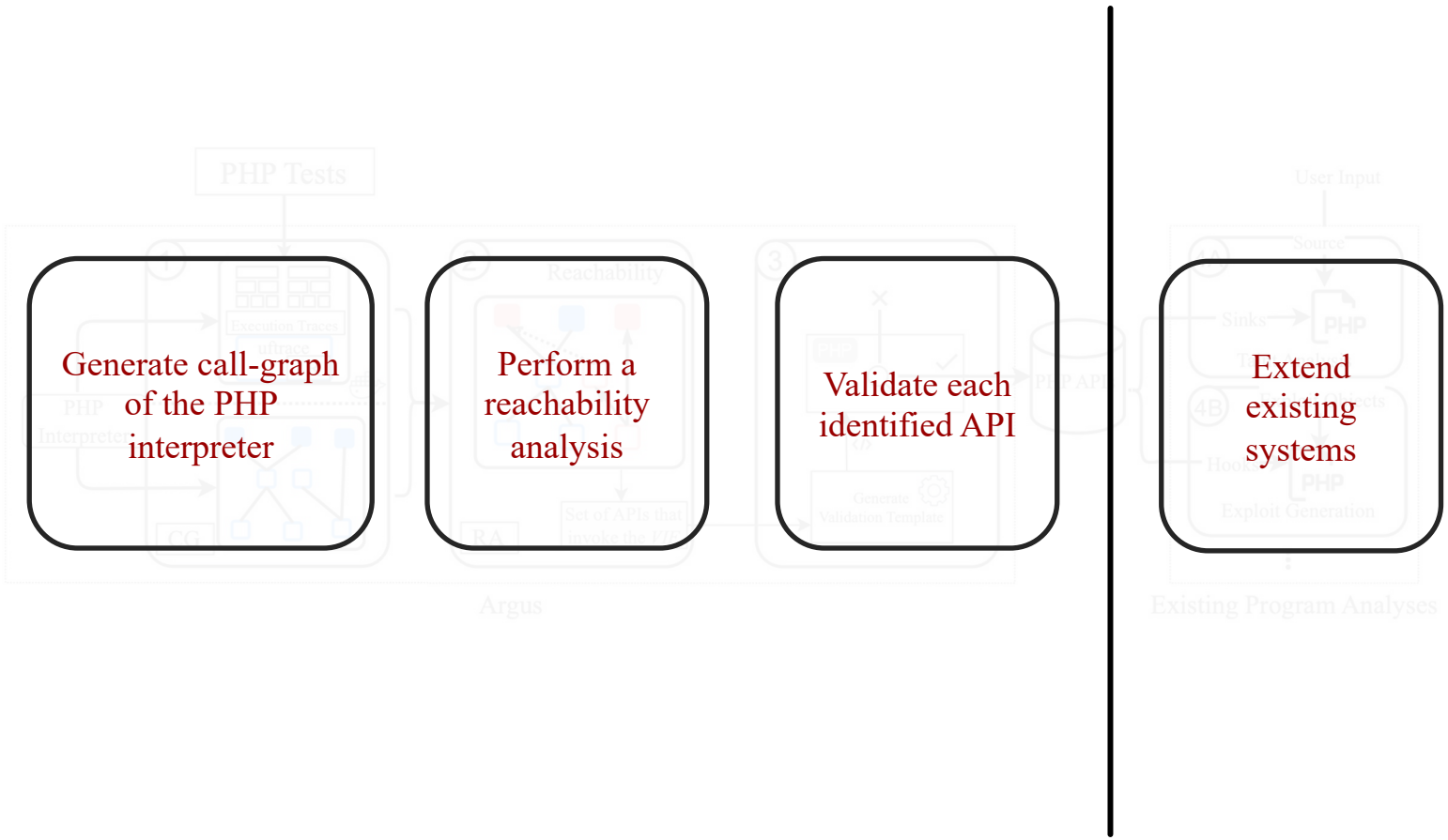
Can we do "better"?
(automated, objective, w/o expert knowledge or bias)

Argus

A principled and systematic approach to identify sensitive PHP functions leading to injection vulnerabilities



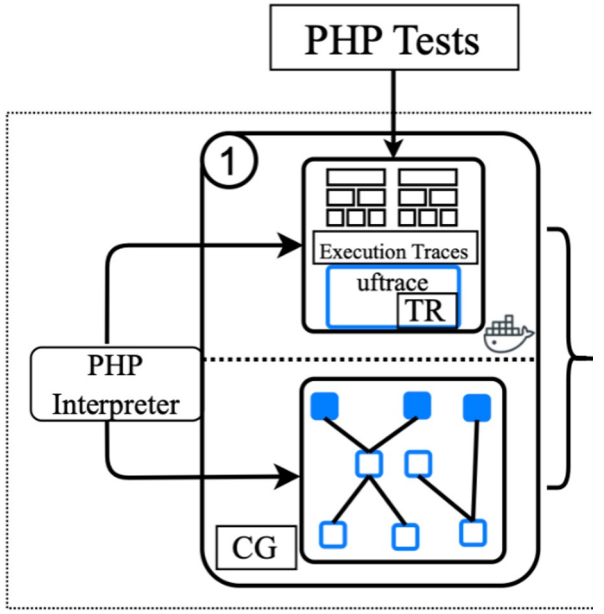
Argus: Overview (3 Step Approach)



Step 1: Generate Call Graph

Argus generates the PHP interpreter's call graph

- Build the call graph statically
  - PHP invokes different functions based on user-input
  - Determined at runtime
- Use dynamic traces to improve the call graph
  - Instrument the PHP interpreter
  - Record function traces
  - Running the unit tests
  - Add edges not already detected using static analysis




```
$file = fopen("/Rasoul/file.txt");

$file = fopen("/Rasoul/file.tar.gz");

$file = fopen("http://example.com/");

$file = fopen("ftp://user:pass@example.com/file.txt");
```

Step 2: Reachability Analysis

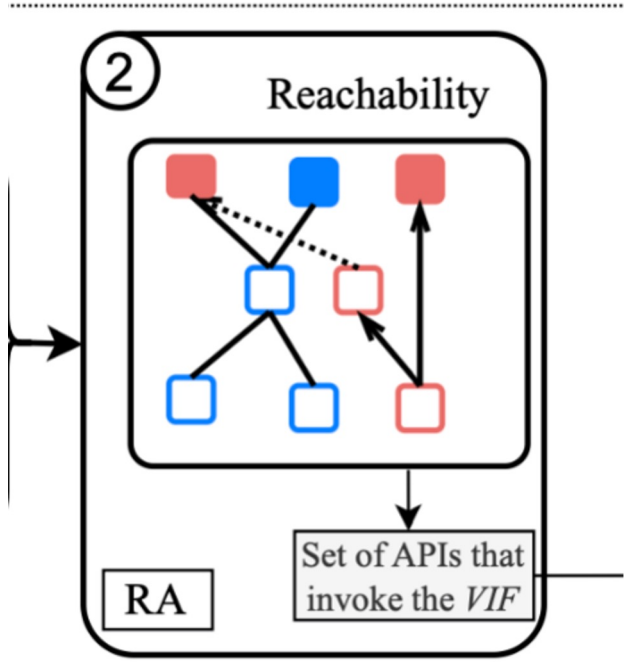
Perform a reachability analysis on the call graph

Find paths from any PHP API to:

- `php_var_unserialize` (Insecure deserialization)
- `php_output_write` (XSS)
- Invokations of the `execv` system call (Command Injection)

Invocation of the sinks

- Not necessarily a vulnerability
- E.g., due to sanitization inside the PHP interpreter



Argus

Step 3: Validation

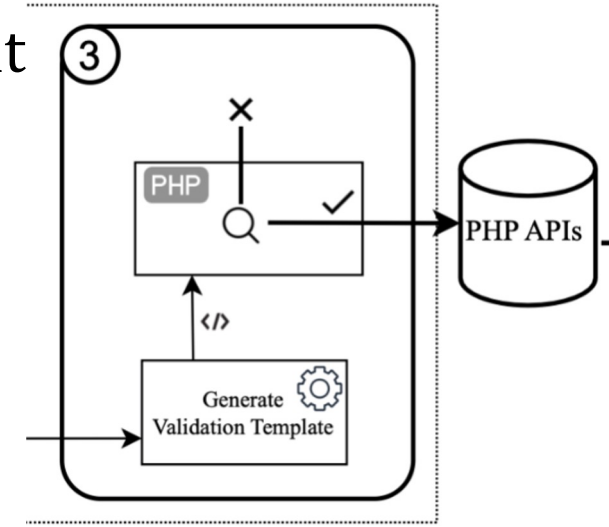
Argus validates the reachability analysis results

Insecure deserialization

- Generate PHP snippets automatically
- Execute the snippet while passing malicious serialized input
- Monitor the execution in case of deserialization

XSS and Command Injection

- Manually validated each API
- Generate a script where the API accepts user-input
- Pass malicious input to the script
- Check if malicious input triggered XSS or CI

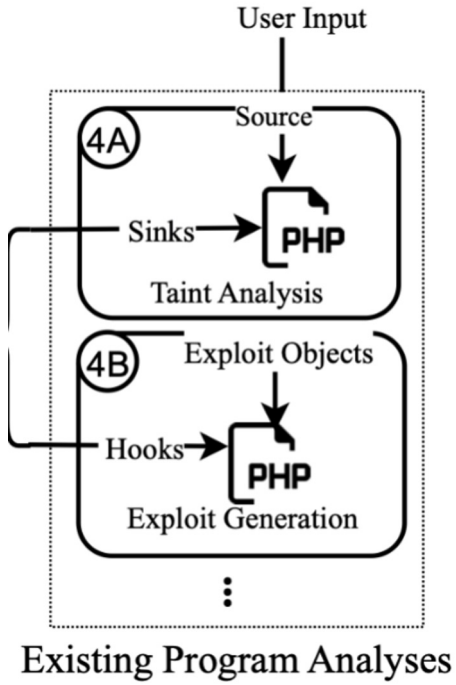


Validated APIs are the lower-bound of all vulnerable APIs

Improving Downstream Analysis

Extend existing detection/exploitation systems with Argus' result

- Static taint analysis: Psalm and RIPS
  - Extend the set of sinks
  - Detect potential XSS and insecure deserialization
- Automatic exploit generation: FUGIO
  - Extend the set of instrumented APIs
  - Monitor deserialization of more APIs



Does a more complete set of sinks actually lead to security relevant improvements?

Argus: Evaluation

- Evaluate on three most popular PHP versions
- Extend two state-of-the-art vulnerability detection/exploitation systems
- Collected 1,977 PHP applications

PHP application Repository	**# of projects**
Web applications	60
Drupal plugins	521
Typo3 plugins	400
WordPress plugins	996
Total	**1977**

Argus: Evaluation cont.

Argus detected:

- 10x more deserialization APIs than prior work
- 2x more output APIs than prior work

PHP interpreter	Deserialization API		XSS-leading API		Exec API	
	Detected	Validated	Detected	Validated	Detected	Validated
PHP 5.6	419	281 (67%)	54	22 (41%)	10	9 (90%)
PHP 7.2	425	284 (67%)	52	22 (42%)	10	9 (90%)
PHP 8.0	20	13 (65%)	46	22 (48%)	10	9 (90%)

Downstream Analysis

Detected 13 previously unknown vulnerabilities in PHP applications

- 12 insecure deserializations
- 1 XSS
- 11 CVEs assigned

Insecure Deserialization

```
1. function fts_twitter_share_url_check() {
2.     $twitter_external_url=$_REQUEST['fts_url'];
3.     // ...
4.     $tags=get_meta_tags($twitter_external_url);
5.     // ...
6. }
```

(Feed Them Social)

XSS

```
1. // wp-includes/ms-files.php
2. // ...
3. $file=rtrim( BLOGUPLOADDIR,'/').'/'.
4. str_replace('..','',$_GET['file']);
5. // ...
6. readfile($file);
```

(Wordpress)

Argus: Summary

- Analyze the PHP interpreter & identify sensitive APIs that lead to injection vulnerabilities (avoids need for expert knowledge)
- Integrates results into existing detection/exploitation systems
- Identifies previous unknown injection vulnerabilities

Takeaway: Don't rely on manually curated lists of sensitive functions (sinks). (if) you don't have to!

Code, Results, & Artifacts
https://github.com/BUseclab/argus

BOSTON UNIVERSITY