Basic memory management ¶

The engine's memory management is implemented with features important to a system like PHP. The exact functionality of the engine's memory management and the optmizations performed are out of the scope of this document. However, a good understanding of its functionality provides a basis for a good understanding of the rest of the `Hacker's Guide`, and introduce you to terminology and functionality used throughout PHP.

The most important of its features for the `Hacker`, and the first thing to mention is tracking allocations. Tracking allocations allow the memory manager to avoid leaks, a thorn in the side of most `Hackers`. When PHP is built in debug mode (`--enable-debug`), detected leaks are reported, in a perfect world they would never get to deployment.

While tracking allocations is an important, and highly useful feature, the `Hacker` should not become lazy! Always attempt to resolve leaks before deploying your code, a memory leak in a SAPI environment can become a very big problem, very quickly.

Another, perhaps more incidental but still noteworthy, feature is that the memory manager is the part that allows a hard limit on memory usage for each instance of PHP. As we all know, there is no such thing as unlimited. If some code is running out of memory, it is likely to be written wrong, either by the `Hacker`, or the programmer of PHP. Limiting the memory therefore is not a restriction on the language that is supposed to be experienced in production, it is simply a way from stopping development environments from spiraling out of control when mistakes are made, and equally, when bugs are found in production.

From the `Hacker's` perspective, the memory management API looks very much like libc's (or whoever the `Hacker` prefers !) malloc implementation.

Main memory APIs

PROTOTYPE	DESCRIPTION
`void *emalloc(size_t size)`	Allocate `size` bytes of memory.
`void *ecalloc(size_t nmemb, size_t size)`	Allocate a buffer for `nmemb` elements of `size` bytes and makes sure it is initialized with zeros.
`void *erealloc(void *ptr, size_t size)`	Resize the buffer `ptr`, which was allocated using `emalloc` to hold `size` bytes of memory.
`void efree(void *ptr)`	Free the buffer pointed by `ptr`. The buffer had to be allocated by `emalloc`.
`void *safe_emalloc(size_t nmemb, size_t size, size_t offset)`	Allocate a buffer for holding `nmemb` blocks of each `size` bytes and an additional `offset` bytes. This is similar to `emalloc(nmemb * size + offset)` but adds a special protection against overflows.
`char *estrdup(const char *s)`	Allocate a buffer that can hold the NULL-terminated string `s` and copy the `s` into that buffer.
`char *estrndup(const char *s, unsigned int length)`	Similar to `estrdup` while the length of the NULL-terminated string is already known.

> ***Note**: The engines memory management functions do not return `NULL` upon failure, if memory cannot be allocated at runtime, the engine bails and raises an error.*

Caution
Always use valgrind before deploying code and as a normal part of the `Hacker's` process. The engine can only report and detect leaks where it has allocated the memory. All of PHP is only a thin wrapper around third parties, those third parties do not use the engines memory management. Additionally, valgrind will catch errors that do not always halt or even have an apparent effect at execution time, it is just as important that there should be no errors, as it is important that avoidable leaks should be avoided.

> ***Note***: *Some leaks are unavoidable, some libraries rely on the end of a process to free some of their structures, this is normal under some circumstances and acceptable where it is out of the* `Hacker's` *control.*

While executing in a debug environment, configured with `--enable-debug`, the leak function used in the next example is actually implemented by the engine and is available to call in userland.

Example #1 Leak Detection in Action

```
ZEND_FUNCTION(leak)
{
    long leakbytes = 3;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "|l", &leakbytes) == FAILURE) {
        return;
    }

    emalloc(leakbytes);
}
```

The above example will output something similar to:

```
[Thu Oct 22 02:14:57 2009]  Script:  '-'
/home/johannes/src/PHP_5_3/Zend/zend_builtin_functions.c(1377) :  Freeing 0x088888D4 (3 bytes), script=-
=== Total 1 memory leaks detected ===
```

> ***Note***: *USE_ZEND_ALLOC=0 in the environment will stop the memory manager from functioning, all allocations fall back on the default system allocators which can be useful for debugging leaks.*