PHP: Hacking, Debugging o simplemente diversión

*— Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

El presente paper refleja diversos mecanismos implementados sobre scripts de PHP para analizar el uso de memoria de forma experimental y por lo tanto solo deben ser tomados como un curioso experimento.

Si te gusta experimentar, analizar y dejarte llevar por un espíritu de curiosidad extrema, jugar a *debuggear* y analizar internamente lo que sucede cuando ejecutas un *script* PHP, puede ser muy divertido.

Si bien PHP es un lenguaje interpretado y de alto nivel que no permite efectuar un verdadero *debug* desde el propio código (por ejemplo, no puedes saber en qué dirección de memoria se está escribiendo determinado valor), con apenas conocer el ID del proceso del *script* puedes comenzar a divertirte.

ADVERTENCIA:** el contenido de este paper solo es **de interés para el disfrute y satisfacción personal** de quien se deja invadir por la curiosidad. Fuera de ello, **carece de toda utilidad.

Introducción

Mediante el siguiente comando se puede obtener una lista de procesos relacionados con cualquier búsqueda, como por ejemplo, php:

```
ps aux | grep php
```

Sin embargo, podría tratarse de una lista extensa que se debería filtrar manualmente o mediante expresiones regulares. Por ejemplo: si estoy ejecutando un *script* llamado `file1.php` mediante el usuario *eugenia*, podría filtrar la búsqueda de forma mucho más precisa mediante lo siguiente:

```
ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"
```

Con los comandos anteriores obtendríamos solo la línea correspondiente al proceso de nuestro *script*:

```
eugenia   28608  0.6  0.3  42600  7700 pts/8    S+   13:30   0:00 php file1.php
```

Si quisiéramos «hilar más fino» aún, podríamos obtener solo el ID del proceso si partimos la cadena en palabras. Para ello, guardaremos la ejecución del comando en una variable que utilizaremos luego para extraer el ID:

```
p=(`ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"`) ; echo ${p[1]}
```

PHP Y EL USO DE MEMORIA

El ID del proceso puede ser útil para conocer el uso de memoria de un *script*.

```
eugenia@co...:$ p=(`ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"`); echo ${p[1]}
29043
eugenia@co...:$ cat /proc/29043/status
Name:      php
State:     S (sleeping)    -- estado del proceso (1)
Tgid:      29043
Pid:29043
PPid:      28263
TracerPid: 0
Uid:1000   1000   1000   1000
Gid:1000   1000   1000   1000
FDSize:    256
Groups:    4 24 27 30 46 104 109 124 1000
VmPeak:       43144 kB     -- tamaño máximo de memoria virtual
VmSize:       42600 kB     -- tamaño total de memoria virtual en uso
VmLck:            0 kB
VmPin:            0 kB
VmHWM:         7704 kB
VmRSS:         7700 kB     -- uso real en memoria física
VmData:       11816 kB     -- tamaño del segmento de datos en memoria virtual
VmStk:          136 kB     -- tamaño de la pila en memoria virtual
VmExe:         7680 kB     -- tamaño del texto en memoria virtual
VmLib:        18180 kB
VmPTE:           84 kB
VmSwap:           0 kB
Threads:   1
...
```

(1) Los posibles estados de un proceso pueden ser:

```
R = running
S = sleeping
D = sleeping (pero no puede interrumpirse)
Z = zombie
T = traced or stopped
```

Para **medir el estado del proceso**, se puede mantener «abierto» el *script* de varias formas:

1. invocando a la función `sleep(N)` para retrasar *N* segundos la ejecución
2. invocando a la función `readline()` para dejar el *script* abierto a la espera de una entrada del usuario

Alternativamente, se puede efectuar el análisis desde el propio *script* obteniendo la ID del proceso con la función `getmypid()`.
Luego, se invocará a `shell_exec()` para correr los comandos que nos permitan visualizar el consumo de memoria.

El siguiente ejemplo, compara el uso de memoria que se hace al crear un *array* extenso con diferentes métodos.

Para un resultado objetivo recomiendo ejecutar los código mediante PHP-CLI iniciando una nueva interfaz en cada ejecución.

```
$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); $i = 0; while($i <= 1024*1024) $a[] = $i++;
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   118556 kB
*/


$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); for($i=0; $i <= 1024*1024; $a[]=$i++);
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   122556 kB
*/


$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = range(0, 1024*1024);
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   122540 kB
*/
```

Para tener una idea aproximada de **cuánta memoria consume el código propiamente dicho** y cuánta es la «memoria inicial» consumida por PHP, te propongo realizar el siguiente experimento paso a paso:

1) Crea un archivo llamado `file.php` con el siguiente código:

```
<?php
$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); $i = 0; while($i <= 1024*1024) $a[] = $i++;
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
sleep(30);
?>
```

2) Corre el archivo ejecutando:

```
php -f file.php
```

3) Rápidamente, antes de transcurridos los 30 segundos del `sleep`, ejecuta el siguiente comando para conocer el uso de memoria hecho por el *script*:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; grep -i vmrss /proc/
$proid/status
```

4) Ahora, abre el archivo nuevamente y borra todo el código excepto la línea del `sleep()` de forma tal que el archivo se vea así:

```
<?php
sleep(30);
?>
```

5) Repite los pasos 2 y 3 y compara el uso de memoria.

El mismo experimento puede realizarse con diferentes algoritmos y así efectuar un análisis exhaustivo que permita extraer conclusiones confiables. Por ejemplo **¿cuánta memoria consume en realidad la creación de variables?** Puedes contrastar el último análisis contra el resultado del siguiente:

```
<?php
$a = 1234567890123456789012345678901234567890;
sleep(30);
?>
```

Y ahora, para hacerlo más interesante, agrégale un `unset()` a la variable:

```
<?php
$a = 12345678901234567890123456789012345678901234567890;
unset($a);
sleep(30);
?>
```

¡Vaya sorpresa! Sucede que **`unset()` se limita a destruir la variable a nivel de *script* pero no libera la memoria de forma inmediata.** No encontré una explicación oficial de lo que ocurre verdaderamente, pero la no oficial sería que el recolector de basura de PHP liberará esa memoria en el momento que la necesite a fin. Esto podría estar justificado por el hecho de que si la liberación fuese inmediata, debería trabajar la CPU para hacerlo.

Se puede efectuar una innumerable cantidad de pruebas incluso utilizando herramientas mucho más completas que un simple `grep`. Por ejemplo, si eres de esas personas curiosas capaces de pasar horas y hasta días *debuggeando*/analizando frente al ordenador, con `strace` podrás divertirte viendo qué es lo que va haciendo PHP mientras se ejecuta tu *script* (personalmente me encanta ver el trazo cuando llega al momento de dormir el *script* xD):

```
strace php file.php
```

También puedes utilizar:

```
strace -dC php file.php
```

(mi preferida) para ver un *debug* propio de `strace` y al final, un resumen de las diferentes llamadas al sistema que PHP va haciendo. Incluso, puedes filtrar las llamadas mediante:

```
strace -e read,write php file.php
```

`ltrace`, quien se utiliza del mismo modo que `strace`, también puede resultarte tan enloquecedor como entretenido y si bien a la mayoría resulta menos útil que `strace`, a mi me agrada mucho más:

```
ltrace php file.php
```

Otra alternativa es utilizar `gdb` aunque a pesar del enorme cariño que siento por `gdb`, para un *script* (de PHP) no suele ser demasiado útil:

```
gdb php
```

y luego:

```
(gdb) run file.php

...
^C
Program received signal SIGINT, Interrupt.
0xb7fdd424 in __kernel_vsyscall ()
(gdb) bt
#0  0xb7fdd424 in __kernel_vsyscall ()
#1  0xb78f7d06 in nanosleep () at ../sysdeps/unix/syscall-template.S:82
#2  0xb78f7aff in __sleep (seconds=0) at ../sysdeps/unix/sysv/linux/sleep.c:138
...
```

Luego, con `pmap` se puede ver el mapeo de memoria que hace el proceso de PHP:

Corriendo el *script* por un lado con `php -f file.php`, luego ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; pmap -x $proid
```

Otros recursos útiles, simpáticos y/o curiosos

top del *script*: Correr el *script* y en paralelo ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; top -p $proid
...
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8045 eugenia   20   0  153m 119m 5372 S    0  6.3   0:00.72 php
```

ps del *script*: Correr el *script* y en paralelo ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; ps -p $proid -o %cpu,%mem,cmd
%CPU %MEM CMD
14.8  6.2 php -f file.php
```

El porcentaje de CPU usado varía de acuerdo al estado del *Script*. Cuánto más tiempo transcurre y el *script* duerme con el `sleep()` menos uso de CPU habrá mientras que el uso de memoria se mantiene impoluto. Si se utilizase `readline()` en lugar de `sleep` podría ponerse en ejecución el `script` mientras se corre el comando reiteradas veces a fin de ver como el consumo de CPU va llegando a `0`.