IOS 反调机制梳理

1 常用反调机制梳理

1.1 ptrace(Android&IOS)

1.1.1 说明

为了方便应用软件的开发和调试，从Unix的早期版本开始就提供了一种对运行中的进程进行跟踪和控制的手段，那就是系统调用`ptrace()`。

通过`ptrace`可以对另一个进程实现调试跟踪，同时`ptrace`还提供了一个非常有用的参数那就是`PT_DENY_ATTACH`，这个参数用来告诉系统，阻止调试器依附。

所以最常用的反调试方案就是通过调用`ptrace`来实现反调试。

1.1.2 代码示例

```
#ifndef PT_DENY_ATTACH
    #define PT_DENY_ATTACH 31
#endif

typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);

ptrace(PT_DENY_ATTACH, 0, 0, 0);

void *handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);
ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(handle, "ptrace");
ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
```

1.2 sysctl

1.2.1 说明

当一个进程被调试的时候，该进程会有一个标记来标记自己正在被调试，所以可以通过`sysctl`去查看当前进程的信息，看有没有这个标记位即可检查当前调试状态。

1.2.2 代码示例

```
BOOL isDebuggerPresent(){
    int name[4];                //指定查询信息的数组

    struct kinfo_proc info;     //查询的返回结果
    size_t info_size = sizeof(info);

    info.kp_proc.p_flag = 0;

    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_PID;
    name[3] = getpid();

    if(sysctl(name, 4, &info, &info_size, NULL, 0) == -1){
        NSLog(@"sysctl error ...");
        return NO;
    }

    return ((info.kp_proc.p_flag & P_TRACED) != 0);
}
```

1.3 syscall

1.3.1 说明

为从实现从用户态切换到内核态，系统提供了一个系统调用函数`syscall`，上面讲到的`ptrace`也是通过系统调用去实现的。

1.3.2 代码示例

```
syscall(26,31,0,0,0);
```

1.4 指令集(SVC-syscall)

1.4.1 说明

`syscall`是通过软中断来实现从用户态到内核态，也可以通过汇编`svc`调用来实现。

1.4.2 代码示例

```
#ifdef __arm__
        asm volatile(
            "mov r0,#31\n"
            "mov r1,#0\n"
            "mov r2,#0\n"
            "mov r12,#26\n"
            "svc #80\n"

        );
#endif
#ifdef __arm64__
        asm volatile(
            "mov x0,#26\n"
            "mov x1,#31\n"
```

```
            "mov x2,#0\n"
            "mov x3,#0\n"
            "mov x16,#0\n"
            "svc #128\n"
        );
#endif
```

1.5 SIGSTOP

1.5.1 说明

捕获系统中断`SIGSTOP`信号来判断

1.5.2 代码示例

```
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL, SIGSTOP, 0,
dispatch_get_main_queue());
dispatch_source_set_event_handler(source, ^{
    NSLog(@"SIGSTOP!!!");
    exit(0);
});
dispatch_resume(source);
```

备注1：上述代码适用于Objective-C 和 Swift代码，事件处理单元主要利用block实现

备注2：代码exception 常用 exit 函数直接退出，正常App设计逻辑中不会出现exit强制退出，因此遇此函数大概率存在反调

1.6 task_get_exception_ports

1.6.1 说明

捕获异常 task_port ===>

1.6.2 代码示例

```
struct macosx_exception_info{
    exception_mask_t masks[EXC_TYPES_COUNT];
    mach_port_t ports[EXC_TYPES_COUNT];
    exception_behavior_t behaviors[EXC_TYPES_COUNT];
    thread_state_flavor_t flavors[EXC_TYPES_COUNT];
    mach_msg_type_number_t cout;
};

struct macosx_exception_info *info = malloc(sizeof(struct macosx_exception_info));
task_get_exception_ports(mach_task_self(),
                                          EXC_MASK_ALL,
                                          info->masks,
                                          &info->cout,
                                          info->ports,
                                          info->behaviors,
                                          info->flavors);

for(uint32_t i = 0; i < info->cout; i ++){
    if(info->ports[i] != 0 || info->flavors[i] == THREAD_STATE_NONE){
        NSLog(@"debugger detected via exception ports (null port)!\n");
    }
}
```

备注：代码exception 常用 exit 函数直接退出，正常App设计逻辑中不会出现exit强制退出，因此遇此函数大概率存在反调

1.7 isatty

1.7.1 代码示例

```
if (isatty(1)) {
    NSLog(@"Being Debugged isatty");
}
```

备注：代码exception 常用 exit 函数直接退出，正常App设计逻辑中不会出现exit强制退出，因此遇此函数大概率存在反调

1.8 ioctl

1.8.1 代码示例

```
if (!ioctl(1, TIOCGWINSZ)) {
    NSLog(@"Being Debugged ioctl");
}
```

备注：代码exception 常用 exit 函数直接退出，正常App设计逻辑中不会出现exit强制退出，因此遇此函数大概率存在反调

2 反反调机制分析

2.1 说明

对于应用安全甲方一般会在这三个方面做防御.

按逻辑分类的话应该应该分为这几类, 但如果从实现原理的话, 应该分为两类, `用API实现的` 和 `不用API实现的`(这说的不用 API 实现, 不是指换成 inine 函数就行) . 首先使用 API 实现基本统统沦陷. 直接通过指令实现的机制还有一丝存活的可能. 逻辑的话应该分为, 反调试, 反注入, 越狱检测, hook 检测.

本文所有相关仅仅针对 aarch64.

假设读者对下知识有了解

1. arm64 相关知识
2. macho 文件结构以及加载相关知识
3. dyld 链接 dylib 相关函数等知识

如何 hook 不定参数函数?

技巧在于伪造原栈的副本. 具体参考下文.

通常来说必备手册

// dyld

https://opensource.apple.com/tarballs/dyld/

// xnu

https://opensource.apple.com/tarballs/xnu/

// objc

https://opensource.apple.com/tarballs/objc4/

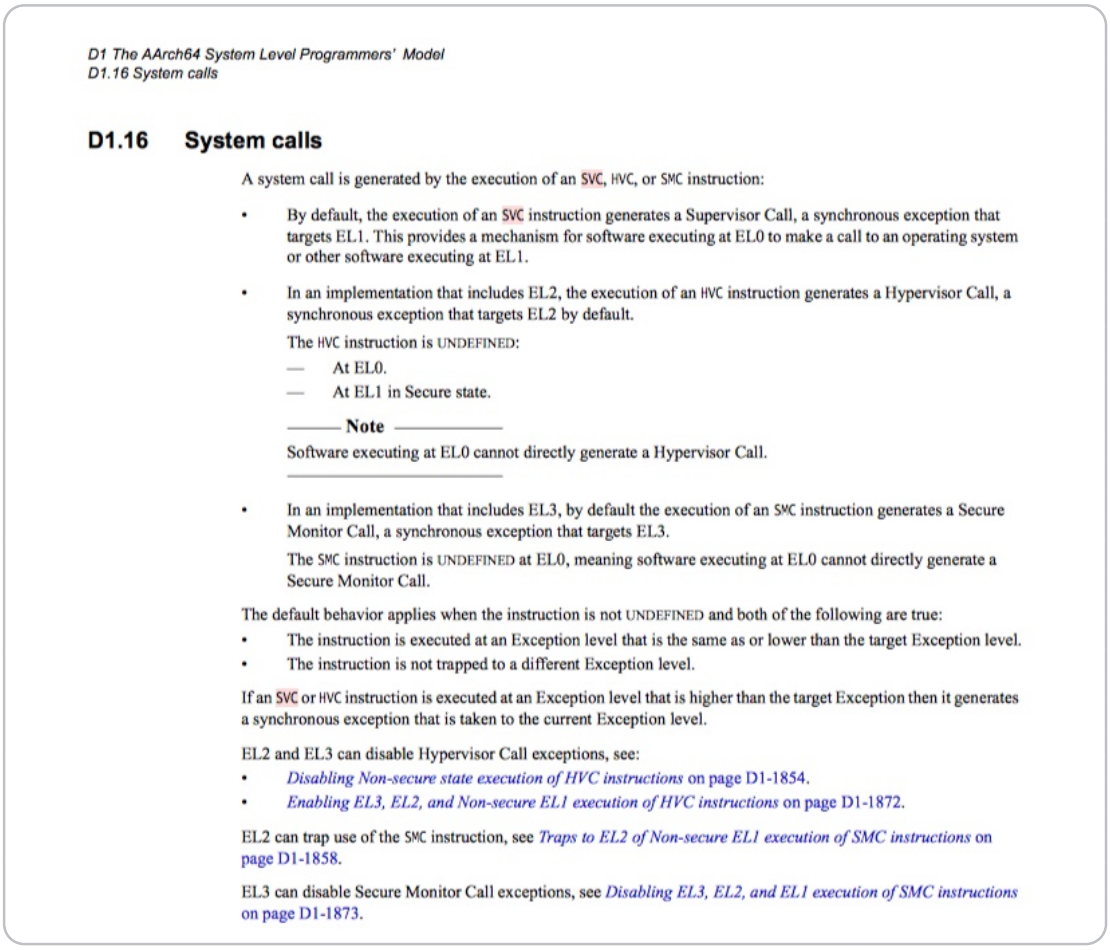
https://github.com/RetVal/objc-runtime (可编译)

// cctools

https://opensource.apple.com/tarballs/cctools (很全的头文件)

2.2 系统调用流程剖析(xnu-3789.41.3为例)

Supervisor Call causes a Supervisor Call exception. svc 切换 `Exception Levels` 从 `EL0(Unprivileged)` 到 `EL1(Privileged)`



D1 The AArch64 System Level Programmers' Model
D1.16 System calls

D1.16 System calls

A system call is generated by the execution of an SVC, HVC, or SMC instruction:

- By default, the execution of an SVC instruction generates a Supervisor Call, a synchronous exception that targets EL1. This provides a mechanism for software executing at EL0 to make a call to an operating system or other software executing at EL1.
- In an implementation that includes EL2, the execution of an HVC instruction generates a Hypervisor Call, a synchronous exception that targets EL2 by default.

  The HVC instruction is UNDEFINED:
  - At EL0.
  - At EL1 in Secure state.

  Note

  Software executing at EL0 cannot directly generate a Hypervisor Call.
- In an implementation that includes EL3, by default the execution of an SMC instruction generates a Secure Monitor Call, a synchronous exception that targets EL3.

  The SMC instruction is UNDEFINED at EL0, meaning software executing at EL0 cannot directly generate a Secure Monitor Call.

The default behavior applies when the instruction is not UNDEFINED and both of the following are true:

- The instruction is executed at an Exception level that is the same as or lower than the target Exception level.
- The instruction is not trapped to a different Exception level.

If an SVC or HVC instruction is executed at an Exception level that is higher than the target Exception then it generates a synchronous exception that is taken to the current Exception level.

EL2 and EL3 can disable Hypervisor Call exceptions, see:

- *Disabling Non-secure state execution of HVC instructions* on page D1-1854.
- *Enabling EL3, EL2, and Non-secure EL1 execution of HVC instructions* on page D1-1872.

EL2 can trap use of the SMC instruction, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1858.

EL3 can disable Secure Monitor Call exceptions, see *Disabling EL3, EL2, and EL1 execution of SMC instructions* on page D1-1873.

上面说的是指令层相关, 再说系统层相关, 使用 svc 进行系统中断调用需要明确 3 个点: 中断号, 系统调用号, 以及参数. 下面以 x86-64 举例.

中断向量表

```
// xnu-3789.41.3/osfmk/x86_64/idt_table.h
USER_TRAP_SPC(0x80,idt64_unix_scall)
USER_TRAP_SPC(0x81,idt64_mach_scall)
USER_TRAP_SPC(0x82,idt64_mdep_scall)
```

中断处理函数

```
// xnu-3789.41.3/osfmk/x86_64/idt64.s
/*
 * System call handlers.
 * These are entered via a syscall interrupt. The system call number in %rax
 * is saved to the error code slot in the stack frame. We then branch to the
 * common state saving code.
 */

#ifndef UNIX_INT
```

```
#error NO UNIX INT!!!
#endif
Entry(idt64_unix_scall)
    swapgs              /* switch to kernel gs (cpu_data) */
    pushq   %rax            /* save system call number */
    PUSH_FUNCTION(HNDL_UNIX_SCALL)
    pushq   $(UNIX_INT)
    jmp L_32bit_entry_check
```

```
// xnu-3789.41.3/bsd/dev/i386/systemcalls.c
__attribute__((noreturn))
void
unix_syscall64(x86_saved_state_t *state)
{
    thread_t    thread;
    void            *vt;
    unsigned int    code;
    struct sysent   *callp;
    int     args_in_regs;
    boolean_t   args_start_at_rdi;
    int     error;
    struct proc *p;
    struct uthread  *uthread;
    x86_saved_state64_t *regs;
    pid_t       pid;

    assert(is_saved_state64(state));
    regs = saved_state64(state);
#if DEBUG
    if (regs->rax == 0x2000800)
        thread_exception_return();
#endif
    thread = current_thread();
    uthread = get_bsdthread_info(thread);

#if PROC_REF_DEBUG
    uthread_reset_proc_refcount(uthread);
#endif

    /* Get the approriate proc; may be different from task's for vfork() */
    if (__probable(!(uthread->uu_flag & UT_VFORK)))
        p = (struct proc *)get_bsdtask_info(current_task());
    else
        p = current_proc();

    /* Verify that we are not being called from a task without a proc */
    if (__improbable(p == NULL)) {
        regs->rax = EPERM;
```

```
        regs->isf.rflags |= EFL_CF;
        task_terminate_internal(current_task());
        thread_exception_return();
        /* NOTREACHED */
    }

    code = regs->rax & SYSCALL_NUMBER_MASK;
    DEBUG_KPRINT_SYSCALL_UNIX(
        "unix_syscall64: code=%d(%s) rip=%llx\n",
        code, syscallnames[code >= nsysent ? SYS_invalid : code], regs->isf.rip);
    callp = (code >= nsysent) ? &sysent[SYS_invalid] : &sysent[code];

```

系统调用表

```
xnu-3789.41.3/bsd/kern/syscall.h
#define SYS_setuid         23
#define SYS_getuid         24
#define SYS_geteuid        25
#define SYS_ptrace         26
#define SYS_recvmsg        27
#define SYS_sendmsg        28
```

2.3 绕过机制

2.3.1 syscall 反调试绕过

因为 `syscall` 反调试有些特殊, 这里需要介绍下如何绕过 `syscall` 反调试, 使用的是 `va_list` 进行传递参数.

`http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf` 参考阅读 `va_list` 相关.

> **函数级反调绕过，直接使用hook函数处理即可(Frida,Theos,Dobby,Substrate,Substitute等)**

这里以国产Hook框架为例

使用 `replace_call` 绕过

这里的 `syscall` 使用的是 `va_list` 传递参数. 所以这里问题在于如何 hook 不定参数函数. 因为在 hook 之后不确定原函数的参数个数. 所以没有办法调用原函数.

所以这里有一个 trick, 在

`orig_syscall(number, stack[0], stack[1], stack[2], stack[3], stack[4], stack[5], stack[6], stack[7]);`

时伪造了一个栈, 这个栈的内容和原栈相同(应该是大于等于原栈的参数内容). 虽然传递了很多参数, 如果理解 `function call` 的原理的话, 即使传递了很多参数, 但是只要栈的内容不变, 准确的说的是从低地址到高地址的栈里的内容不变(这里可能多压了很多无用的内容到栈里), 函数调用就不会变.

这里不要使用 `large structure`, 编译时会使用隐含的 `memcpy` 最终传入的其实是地址. 大部分注释请参考下文.

Procedure Call Standard for the ARM 64-bit Architecture

Stage A – Initialization

This stage is performed exactly once, before processing of the arguments commences.

A.1 The Next General-purpose Register Number (NGRN) is set to zero.

A.2 The Next SIMD and Floating-point Register Number (NSRN) is set to zero.

A.3 The next stacked argument address (NSAA) is set to the current stack-pointer value (SP).

Stage B – Pre-padding and extension of arguments

For each argument in the list the first matching rule from the following list is applied. If no rule matches the argument is used unmodified.

B.1 If the argument type is a Composite Type whose size cannot be statically determined by both the caller and the callee, the argument is copied to memory and the argument is replaced by a pointer to the copy. (There are no such types in C/C++ but they exist in other languages or in language extensions).

B.2 If the argument type is an HFA or an HVA, then the argument is used unmodified.

B.3 If the argument type is a Composite Type that is larger than 16 bytes, then the argument is copied to memory allocated by the caller and the argument is replaced by a pointer to the copy.

B.4 If the argument type is a Composite Type then the size of the argument is rounded up to the nearest multiple of 8 bytes.

```
int (*orig_syscall)(int number, ...);
int fake_syscall(int number, ...) {
    int request;
    pid_t pid;
    caddr_t addr;
    int data;

    // fake stack, why use `char *` ? hah
    char *stack[8];

    va_list args;
    va_start(args, number);

    // get the origin stack args copy.(must >= origin stack args)
    memcpy(stack, args, 8 * 8);

    if (number == SYS_ptrace) {
        request = va_arg(args, int);
        pid = va_arg(args, pid_t);
        addr = va_arg(args, caddr_t);
        data = va_arg(args, int);
        va_end(args);
        if (request == PT_DENY_ATTACH) {
            NSLog(@"[AntiDebugBypass] catch 'syscall(SYS_ptrace, PT_DENY_ATTACH, 0, "
```

```
                @"0, 0)' and bypass.");
            return 0;
        }
    } else {
        va_end(args);
    }

    // must understand the principle of `function call`. `parameter pass` is
    // before `switch to target` so, pass the whole `stack`, it just actually
    // faked an original stack. Do not pass a large structure,  will be replace with
    // a `hidden memcpy`.
    int x = orig_syscall(number, stack[0], stack[1], stack[2], stack[3], stack[4],
                         stack[5], stack[6], stack[7]);
    return x;
}
```

2. 使用 `pre_call` 绕过

这种方法需要查看 `syscall` 的汇编实现, 来确定 `PT_DENY_ATTACH` 放在哪一个寄存器.

```
libsystem_kernel.dylib`__syscall:
    0x1815c0900 <+0>:  ldp    x1, x2, [sp]
    0x1815c0904 <+4>:  ldp    x3, x4, [sp, #0x10]
    0x1815c0908 <+8>:  ldp    x5, x6, [sp, #0x20]
    0x1815c090c <+12>: ldr    x7, [sp, #0x30]
    0x1815c0910 <+16>: mov    x16, #0x0
    0x1815c0914 <+20>: svc    #0x80
    0x1815c0918 <+24>: b.lo   0x1815c0930               ; <+48>
    0x1815c091c <+28>: stp    x29, x30, [sp, #-0x10]!
    0x1815c0920 <+32>: mov    x29, sp
    0x1815c0924 <+36>: bl     0x1815a6dc0               ; cerror
    0x1815c0928 <+40>: mov    sp, x29
    0x1815c092c <+44>: ldp    x29, x30, [sp], #0x10
    0x1815c0930 <+48>: ret
```

可以看到调用如果 `x0` 是 `SYS_ptrace`, 那么 `PT_DENY_ATTACH` 存放在 `[sp]`.

```
void syscall_pre_call(RegState *rs, ThreadStack *threadstack, CallStack *callstack) {
    int num_syscall;
    int request;
    zpointer sp;
    num_syscall = (int)(uint64_t)(rs->general.regs.x0);
    if (num_syscall == SYS_ptrace) {
        sp = (zpointer)(rs->sp);
        request = *(int *)sp;
        if (request == PT_DENY_ATTACH) {
            *(long *)sp = 10;
            NSLog(@"[AntiDebugBypass] catch 'syscall(SYS_ptrace, PT_DENY_ATTACH, 0, "
                  @"0, 0)' and bypass.");
```

```
        }
    }
}
__attribute__((constructor)) void patch_syscall_by_pre_call() {
    zpointer syscall_ptr = (void *)syscall;
    #if 0
    ZzBuildHook((void *)syscall_ptr, NULL, NULL, syscall_pre_call, NULL);
    ZzEnableHook((void *)syscall_ptr);
    #endif
}
// --- end ---
```

2.3.2 "SVC #0x80" 反调试绕过

这里介绍关键是介绍如何对 svc 反调试的绕过.

上面已经对 svc 进行了简单的介绍. 所以理所当然想到的是希望通过 `syscall hook`, 劫持 `system call table(sysent)` . 这里相当于实现 `syscall hook`. 但是难点之一是需要找到 `system call table(sysent)`, 这一步可以通过 joker, 对于 IOS 10.x 可以参考 `http://ioshackerwiki.com/syscalls/`, 难点之二是作为 kext 加载. 可以参考 **附录**, 对于具体的 `kernel patch` 没有做过深入研究, 应该可以参考 comex 的 datautils0

ok, 接下来使用另一种思路对绕过, 其实也就是 `code patch` + `hook address`. 对 `__TEXT` 扫描 `svc #0x80` 指令, 对于 cracker 来说, 在 `__TEXT` 段使用 `svc #0x80` 具有一定的反调试可能, 所以需要对 `svc #0x80` 进行 `hook addres`, 这里并不直接对 `svc #0x80` 进行覆盖操作.

这里以国产Hook框架为例

大致原理就是先搜索到 `svc #0x80` 指令后, 对该指令地址进行 hook, 之后使用 `pre_call` 修改寄存器的值.

```
void hook_svc_pre_call(RegState *rs, ThreadStack *threadstack, CallStack *callstack) {
    int num_syscall;
    int request;
    num_syscall = (int)(uint64_t)(rs->general.regs.x16);
    request = (int)(uint64_t)(rs->general.regs.x0);

    if (num_syscall == SYS_syscall) {
        int arg1 = (int)(uint64_t)(rs->general.regs.x1);
        if (request == SYS_ptrace && arg1 == PT_DENY_ATTACH) {
            *(unsigned long *)(&rs->general.regs.x1) = 10;
            NSLog(@"[AntiDebugBypass] catch 'SVC #0x80; syscall(ptrace)' and bypass");
        }

    } else if (num_syscall == SYS_ptrace) {
        request = (int)(uint64_t)(rs->general.regs.x0);
        if (request == PT_DENY_ATTACH) {
```

```
            *(unsigned long *)(&rs->general.regs.x0) = 10;
            NSLog(@"[AntiDebugBypass] catch 'SVC-0x80; ptrace' and bypass");
        }
    } else if(num_syscall == SYS_sysctl) {
        STACK_SET(callstack, (char *)"num_syscall", num_syscall, int);
        STACK_SET(callstack, (char *)"info_ptr", rs->general.regs.x2, zpointer);
    }
}

void hook_svc_half_call(RegState *rs, ThreadStack *threadstack, CallStack *callstack) {
    // emmm... little long...
    if(STACK_CHECK_KEY(callstack, (char *)"num_syscall")) {
        int num_syscall = STACK_GET(callstack, (char *)"num_syscall", int);
        struct kinfo_proc *info = STACK_GET(callstack, (char *)"info_ptr", struct kinfo_proc *);
        if (num_syscall == SYS_sysctl)
        {
            NSLog(@"[AntiDebugBypass] catch 'SVC-0x80; sysctl' and bypass");
            info->kp_proc.p_flag &= ~(P_TRACED);
        }
    }
}

__attribute__((constructor)) void hook_svc_x80() {
    zaddr svc_x80_addr;
    zaddr curr_addr, text_start_addr, text_end_addr;
    uint32_t svc_x80_byte = 0xd4001001;

    const struct mach_header *header = _dyld_get_image_header(0);
    struct segment_command_64 *seg_cmd_64 = zz_macho_get_segment_64_via_name((struct mach_header_64 *)header, (char *)"__TEXT");
    zsize slide = (zaddr)header - (zaddr)seg_cmd_64->vmaddr;

    struct section_64 *sect_64 = zz_macho_get_section_64_via_name((struct mach_header_64 *)header, (char *)"__text");

    text_start_addr = slide + (zaddr)sect_64->addr;
    text_end_addr = text_start_addr + sect_64->size;
    curr_addr = text_start_addr;

    while (curr_addr < text_end_addr) {
        svc_x80_addr = (zaddr)zz_vm_search_data((zpointer)curr_addr, (zpointer)text_end_addr, (zbyte *)&svc_x80_byte, 4);
        if (svc_x80_addr) {
            NSLog(@"hook svc #0x80 at %p with aslr (%p without aslr)",
                  (void *)svc_x80_addr, (void *)(svc_x80_addr - slide));
            ZzBuildHookAddress((void *)svc_x80_addr, (void *)(svc_x80_addr + 4),
                               hook_svc_pre_call, hook_svc_half_call);
            ZzEnableHook((void *)svc_x80_addr);
```

```
            curr_addr = svc_x80_addr + 4;
        } else {
            break;
        }
    }
}
```

3 参考链接

1. http://siliconblade.blogspot.jp/2013/07/offensive-volatility-messing-with-os-x.html
2. https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-bosse_eriksson-kernel_patching_on_osx.pdf
3. http://d.hatena.ne.jp/hon53/20100926/1285476759
4. https://papers.put.as/papers/ios/2011/SysScan-Singapore-Targeting_The_IOS_Kernel.pdf
5. https://www.blackhat.com/docs/us-15/materials/us-15-Diquet-TrustKit-Code-Injection-On-iOS-8-For-The-Greater-Good.pdf
6. https://github.com/LinusHenze/anyKextLoader
7. https://github.com/Jailbreaks/trident-kloader
8. https://github.com/saelo/ios-kern-utils
9. https://github.com/xerub/kexty
10. https://github.com/sbingner/substitute
11. https://github.com/frida/frida
12. https://github.com/4ch12dy/xia0LLDB