[原创]【从源码过反调试】三、源码实现一些过检测（文件过滤，进程隐藏，notify bypass，mincore bypass）

xxxlion

极客

38分钟前

87

重发调整了格式

syscall 入口

所有的参数都是syscall传进内核区分参数，指令

参数：设置/移除pid，设置/移除 文件过滤字符串 设置/移除 nofity bypass文件路径

指令：开关文件过滤，物理地址读写内存，进程隐藏开关。nofity文件过滤

```
enum CmdDataType

{

    SET_TARGET_PID = 1 ,

    SET_SELF_PID = 2,

    REMOVE_TARGET_PID = 4,

    REMOVE_SELF_PID = 8,

    SET_FILTER_STR = 16,

    REMOVE_FILTER_STR = 32,

    SET_NOTIFY_PATH_STR = 64,

    REMVOE_NOTIFY_PATH_STR = 128

};



enum CmdSwitchType

{

    FILTER_FILE = 0,

    GLOABAL_PID,

    READ_MEMORY,

    WRITE_MEMORY,

    HIDE_PROCESS,

    REMOVE_HIDE_PROCESS,

    NOTIFY

};
```

目标管理

用户层传参数、指令，保存到内核。

开启各种功能最先做的还是选定目标了，设置target_pid开启各种过检测，设置spid保护自己的进程

文件过滤

过通用文件检测，主要用来过一些模拟器检查。可以自定义返回错误码

在getname_flags里增加判断，我测试过几个点，这个位置是文件访问的几个syscall(open, access, stat)都会走的，在这里比较合适，不过不知道有没有遗漏其他的。

修改源码fs/namei.c

判断要访问的文件是否在被保护字符串列表里。

```
struct filename *

getname_flags(const char __user *filename, int flags, int *empty)

{



        // ... 前面省略



    /* The empty path is special. */

    if (unlikely(!len)) {

        if (empty)

            *empty = 1;

        if (!(flags & LOOKUP_EMPTY)) {

            putname(result);

            return ERR_PTR(-ENOENT);

        }

    }



    result->uptr = filename;

    result->aname = NULL;

    if (is_target() && is_str_in_filter_array(result->uptr, &ecode))

    {

        // 如果在过滤列表

        printk("[AntiLog] dont access me result:%d\n", ecode);

        return ERR_PTR(ecode);

    }

    audit_getname(result);

    return result;

}
```

字符串比较

为了优化性能，这里我没有用全部字符串比较，从传参这里加了限定，比较从start，到end，取16个字节, 在内核里直接用异或计算，把O(n)的时间复杂度降低到了O1。小于16个字节的字符串使用strcmp。kernel里的strcmp是O(n)并没有做优化，glibc里的倒是有优化 但用不了。

最初第一反应是用哈希表，但内核里没有实现比较完善的哈希表，字符串求hash，碰撞函数这些都要自己写，想来过于麻烦就没用这种方式。

```
for (i = 0; i < filter_array_lens; i++) {

        if (filter_array[i].str_lens < 16) {

            // cmp 通配符查找

            memcpy(substr_str, str, filter_array[i].str_lens);

            memset(substr_str + filter_array[i].str_lens, 0,

                   16 - filter_array[i].str_lens);



            if (strcmp(substr_str, filter_array[i].str_content) == 0) {

                if (filter_array[i].is_user_custom_res) {

                    *res_errcode = filter_array[i].custom_res;

                    printk("use errcode :%d", filter_array[i].custom_res);

                } else {

                    *res_errcode = -ENOENT;

                }

                return true;

            }



        } else if (filter_array[i].str_lens >= 16 && dst_str_len >= 16) {

            // cmp by xor

            __uint128_t dst_hash = *(__uint128_t *)(&str[filter_array[i].start_pos]);

            if ((dst_hash ^ filter_array[i].str_hash) == 0) {

                if (filter_array[i].is_user_custom_res) {

                    *res_errcode = filter_array[i].custom_res;

                } else {

                    *res_errcode = -ENOENT;

                }

                return true;

            }

        } else if (filter_array[i].str_lens >= 16 && dst_str_len < 16) {

            continue;

        }

    }
```

notify访问bypass

检测不多说了，bypass思路和文件过滤相同，主要是找对位置

include/linux/fsnotify.h

```c
static inline int fsnotify_file(struct file *file, __u32 mask)

{

    const struct path *path;

    if(is_target() && is_file_fsnotify_block(file))

    {

        printk("[AntiLog] done notify me\n");

        return 0;

    }

    path = &file->f_path;



    if (file->f_mode & FMODE_NONOTIFY)

        return 0;



    return fsnotify_parent(path->dentry, mask, path, FSNOTIFY_EVENT_PATH);

}



// file -> user_path cmp syscall_user_path

int is_file_fsnotify_block(struct file *file)

{

    char *tmp;

    char *path;

    int str_pos;



    if (fsnotify_block_switch == false) {

        return -1;

    }

    if (file == NULL) {

        return -2;

    }

    tmp = (char *)__get_free_page(GFP_KERNEL);

    if (tmp == NULL) {

        return -3;

    }

    path = d_path(&file->f_path, tmp, PAGE_SIZE);

    if (IS_ERR(path)) {

        printk("[AntiLog]d_path error:%p\n", path);

        return -4;

    }



    str_pos = filepath_str_in_array_pos(path);



    free_page((unsigned long)tmp);



    return str_pos >= 0 ? str_pos : -5;
```

进程隐藏

无论是ps -A 或者是ls /proc/... 这种方式都无法枚举到自己进程

还是要找对位置，这两种方式如果用strace跟踪下，会发现核心是调用readdir，那就在内核这里加过滤即可

```
/* for the /proc/ directory itself, after non-process stuff has been done */

int proc_pid_readdir(struct file *file, struct dir_context *ctx)

{

    // 省略前面

    for (iter = next_tgid(ns, iter);

         iter.task;

         iter.tgid += 1, iter = next_tgid(ns, iter)) {



        char name[10 + 1];

        unsigned int len;



        if(is_process_hide_by_pid(iter.tgid))

        {

            printk("[AntiLog] done access my process\n");

            continue;

        }



        cond_resched();

        if (!has_pid_permissions(fs_info, iter.task, HIDEPID_INVISIBLE))

            continue;



        len = snprintf(name, sizeof(name), "%u", iter.tgid);

        ctx->pos = iter.tgid + TGID_OFFSET;

        if (!proc_fill_cache(file, ctx, name, len,

                    proc_pid_instantiate, iter.task, NULL)) {

            put_task_struct(iter.task);

            return 0;

        }

    }

    ctx->pos = PID_MAX_LIMIT + TGID_OFFSET;

    return 0;

}
```

内存读写bypass mincore

思路大家都是一样的，实现的方法是有些差异

读之前检查pagefault

读物理地址

本来也是打算用页表一级一级的算过去，将va转换到pa，但是linux5.10，安卓12.1.0，x86_64模拟器下，五级页表转换怎么都算不对，都是调用的内核提供的函数，p*d_offset(), 算出来的也不对。不知道为何算出的pmd == p4d，往下都是错的。

https://zhuanlan.zhihu.com/p/220068494

（没测试驱动内是否可以用，就像__pa宏的注释不应该在驱动中使用一样，page_to_phys宏在驱动中不一定可用）。

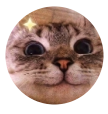
```
unsigned long get_phsy_addr_by_task(struct mm_struct* mm, unsigned long vaddr)

{

    struct page *pages;

    unsigned long paddr;

    int ret;



    ret = get_user_pages(vaddr, 1 , FOLL_FORCE, &pages, NULL);

    if(ret != 1)

    {

        printk(KERN_ERR "get user pages() failed\n");



                // 对于没有提交到物理地址的内存页，get_page这里会报错，所以在这里不读就可以过mincore了。

        if(ret == -EFAULT)

        {

            printk(KERN_INFO "page fault occured\n");

        }

        return -EFAULT;

    }



    paddr = page_to_phys(pages) | (vaddr & ~PAGE_MASK);



    printk(KERN_INFO "[AntiLog] vaddr:%p, paddr:%p\n", vaddr, paddr);

    put_page(pages);

    return paddr;

}
```

之后再把物理地址map到内核虚拟地址，读写即可

```
read = 0;

while (count > 0) {



        sz = size_inside_page(p, count);

kvaddr = ioremap(p, sz);



if(!kvaddr)

{

    printk(KERN_ERR "ioremap failed\n");

    return -ENOMEM;

}



        if (buf) {



                if (cmd_type == READ_MEMORY) {

                        memcpy_fromio(buf, kvaddr, sz);

                        printk("Read To Buffer:%s Size:%d, Src:%s", buf, sz, kvaddr);

                }

                if (cmd_type == WRITE_MEMORY) {

                        printk("Before Write To Addr:%p Content:%s", kvaddr, buf);

                        memcpy_toio(kvaddr, buf, sz);

                        printk("After Write To Addr:%p Content:%s", kvaddr, buf);

                }

        }



iounmap(kvaddr);

kvaddr = NULL;



        buf += sz;

        count -= sz;

        read += sz;

}
```

反勒索软件开发实战篇

最新回复 (0)



R0g

内容

回帖　表情　高级回复

返回