2025 寒假第三次周报

按理来说上周其实应该也写一份周报，但是上周 7 天有 4 天都在过年，另外几天也没学什么东西，就没写，打算和这周的合并在一起。

1.27/1.28

把 t1d 的 rustsignout 做完之后，t1d 又发了一道很 trick 的题，考察两个知识点：

- `seccomp` 开的沙箱只和开沙箱时的进程有关，和其他（父）子进程无关；
- `/proc/$pid/mem` 可以对进程的虚拟内存空间任意地址写，哪怕这个地址在 gdb 等用户态调试器查看时是不可写的。

后者的原因是 /proc/$pid/mem 文件走的是内核的 mem_write，最后走到 get_xxx_page.

过年啦～ 过年没带笔记本回去，带了个平板坐车上随便翻了一下 glibc / procfs 的源码。

2.4

好，👴回家了，好。

先帮 t1d 测了一道题，题目考察了一些比较不常见的 syscall `__NR_copy_file_range`，这玩意比较有意思，可以从文件到文件零拷贝；还有一个 `splice`，这个可以从文件到管道 / 从管道到文件，支持偏移。

做这种题主要还是靠读文档，翻 syscall，问 ai，，，不然根本不知道这些乱七八糟的 syscall。

还考察了之前周报写过的 使用 `initstate` 令 n=8 时 `rand` 的 trick，这种东西见过一次基本上就印象深刻了。

还有和 rustsignout 比较像的 shellcode 构造，不过应该没那个难，我乱杀了，26 字节，比 t1d 短太多，花又 win！

然后研究了一下 execve 与 system 的区别，整理在下面了：

简介

关于 execve 和 system，在 Pwn 的时候经常作为题目 exp 的终端，前者经常用在 onegadget 或者 shellcode 里，后者则在 ROP 里见得更多一点。

这篇笔记主要关注 execve 和 system 在调用后的进程状态。

execve

理论

execve() 调用成功后不会返回, 其进程的正文(text), 数据(data), bss 和堆栈 (stack) 段被调入程序覆盖。调入程序继承了调用程序的 PID 和所有打开的文件描述符, 他们不会因为 exec 过程而关闭. 父进程的未决信号被清除. 所有被调用进程设置过的信号重置为缺省行为.

检验

写一个 poc：

```
// gcc execve.c -g -o execve
#include <stdio.h>
#include <unistd.h>

int main() {
    char *path = "/bin/sh";
    char *argv[] = {path, NULL};
    char *envp[] = {NULL};
```

```
    int a = 0;
    scanf("%d", &a); // 起一个打断点的作用
    if (execve(path, argv, envp) == -1) {
        perror("execve");
        return 1;
    }
    // 如果execve调用成功，下面的代码不会执行
    printf("execve 调用成功! ");
    return 0;
}
```

然后编译运行这个 poc，在输入任意数字之前，`ps -ax | grep execve` 可以看到：

```
$ ps -ax | grep execve
PID TTY      STAT   TIME COMMAND
70102 pts/6    S+     0:00 ./execve
```

然后输入任意数据，再看 70102 这个 pid：

```
$ ps -ax | grep execve
PID TTY      STAT   TIME COMMAND
70102 pts/6    S+     0:00 /bin/sh
```

显然，`execve` 执行的沙箱操作是替换当前进程，会保留 execve 之前的沙箱。

如何解读 STAT 字段呢？可以看 [[解读 ps 的 STAT 字段]]。

system

理论

The system() library function behaves as if it used fork(2) to create a child process that executed the shell command specified in command using execl(3) as follows: execl("/bin/sh", "sh", "-c", command, (char *) NULL); system() returns after the command has been completed.

检验

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a;
```

```
    scanf("%d", &a);
    int status = system("/bin/sh");
    if (status == -1) {
        perror("system");
        return EXIT_FAILURE;
    }
    printf("The shell exited with status %d\n", status);
    return EXIT_SUCCESS;
}
```

编译运行后，同样可以查看：

```
70781 pts/6    S+     0:00 ./system
```

输入任意数据后：

```
70781 pts/6    S      0:00 ./system
70920 pts/6    S+     0:00 /bin/sh
```

`ps -ef --forest` 可以看到

```
flower     57139    2577  0 2月03 pts/6    00:00:00  |   \_ /usr/bin/zsh
flower     70781   57139  0 00:00 pts/6     00:00:00  |   |   \_ ./system
flower     70920   70781  0 00:01 pts/6     00:00:00  |   |       \_ /bin/sh
```

注意到它另起了一个子进程来执行 sh，而子进程显然是不会被沙箱约束的。

2.5

继续调了一下 execveat 与 openat 的关系。

当沙箱禁用系统调用 execve，往往会想到去用 execveat，但是有的时候会发现 execveat 也无法调用，这是为什么呢？

execveat 源码：

首先在 `fs/exec.c` 里

```
static int do_execveat(int fd, struct filename *filename,
                const char __user *const __user *__argv,
                const char __user *const __user *__envp,
```

```
                int flags)
{
        struct user_arg_ptr argv = { .ptr.native = __argv };
        struct user_arg_ptr envp = { .ptr.native = __envp };

        return do_execveat_common(fd, filename, argv, envp, flags);
}
```

调用了 do_execveat_common()，跟进去看：

```
static int do_execveat_common(int fd, struct filename *filename,
                              struct user_arg_ptr argv,
                              struct user_arg_ptr envp,
                              int flags)
{
        struct linux_binprm *bprm;
        int retval;

        if (IS_ERR(filename))
                return PTR_ERR(filename);

        /*
         * We move the actual failure in case of RLIMIT_NPROC excess from
         * set*uid() to execve() because too many poorly written programs
         * don't check setuid() return code.  Here we additionally recheck
         * whether NPROC limit is still exceeded.
         */
        if ((current->flags & PF_NPROC_EXCEEDED) &&
            is_rlimit_overlimit(current_ucounts(), UCOUNT_RLIMIT_NPROC,
rlimit(RLIMIT_NPROC))) {
                retval = -EAGAIN;
                goto out_ret;
        }

        /* We're below the limit (still or again), so we don't want to make
         * further execve() calls fail. */
        current->flags &= ~PF_NPROC_EXCEEDED;

        bprm = alloc_bprm(fd, filename, flags);
        if (IS_ERR(bprm)) {
                retval = PTR_ERR(bprm);
                goto out_ret;
        }

        retval = count(argv, MAX_ARG_STRINGS);
        if (retval == 0)
                pr_warn_once("process '%s' launched '%s' with NULL argv: empty
string added\n",
                             current->comm, bprm->filename);
        if (retval < 0)
                goto out_free;
        bprm->argc = retval;

        retval = count(envp, MAX_ARG_STRINGS);
```

```
        if (retval < 0)
                goto out_free;
        bprm->envc = retval;

        retval = bprm_stack_limits(bprm);
        if (retval < 0)
                goto out_free;

        retval = copy_string_kernel(bprm->filename, bprm);
        if (retval < 0)
                goto out_free;
        bprm->exec = bprm->p;

        retval = copy_strings(bprm->envc, envp, bprm);
        if (retval < 0)
                goto out_free;

        retval = copy_strings(bprm->argc, argv, bprm);
        if (retval < 0)
                goto out_free;

        /*
         * When argv is empty, add an empty string ("") as argv[0] to
         * ensure confused userspace programs that start processing
         * from argv[1] won't end up walking envp. See also
         * bprm_stack_limits().
         */
        if (bprm->argc == 0) {
                retval = copy_string_kernel("", bprm);
                if (retval < 0)
                        goto out_free;
                bprm->argc = 1;
        }

        retval = bprm_execve(bprm);
out_free:
        free_bprm(bprm);

out_ret:
        putname(filename);
        return retval;
}
```

看 alloc_bprm

```
static struct linux_binprm *alloc_bprm(int fd, struct filename *filename, int
flags)
{
        struct linux_binprm *bprm;
        struct file *file;
        int retval = -ENOMEM;

        file = do_open_execat(fd, filename, flags);
```

```
        if (IS_ERR(file))
                return ERR_CAST(file);

        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
        if (!bprm) {
                do_close_execat(file);
                return ERR_PTR(-ENOMEM);
        }

        bprm->file = file;

        if (fd == AT_FDCWD || filename->name[0] == '/') {
                bprm->filename = filename->name;
        } else {
                if (filename->name[0] == '\0')
                        bprm->fdpath = kasprintf(GFP_KERNEL, "/dev/fd/%d", fd);
                else
                        bprm->fdpath = kasprintf(GFP_KERNEL, "/dev/fd/%d/%s",
                                                 fd, filename->name);
                if (!bprm->fdpath)
                        goto out_free;

                /*
                 * Record that a name derived from an O_CLOEXEC fd will be
                 * inaccessible after exec.  This allows the code in exec to
                 * choose to fail when the executable is not mmaped into the
                 * interpreter and an open file descriptor is not passed to
                 * the interpreter.  This makes for a better user experience
                 * than having the interpreter start and then immediately fail
                 * when it finds the executable is inaccessible.
                 */
                if (get_close_on_exec(fd))
                        bprm->interp_flags |= BINPRM_FLAGS_PATH_INACCESSIBLE;

                bprm->filename = bprm->fdpath;
        }
        bprm->interp = bprm->filename;

        retval = bprm_mm_init(bprm);
        if (!retval)
                return bprm;

out_free:
        free_bprm(bprm);
        return ERR_PTR(retval);
}
```

跟入 do_open_execat 函数

```
static struct file *do_open_execat(int fd, struct filename *name, int flags)
{
        int err;
        struct file *file __free(fput) = NULL;
        struct open_flags open_exec_flags = {
```

```
                .open_flag = O_LARGEFILE | O_RDONLY | __FMODE_EXEC,
                .acc_mode = MAY_EXEC,
                .intent = LOOKUP_OPEN,
                .lookup_flags = LOOKUP_FOLLOW,
        };

        if ((flags & ~(AT_SYMLINK_NOFOLLOW | AT_EMPTY_PATH)) != 0)
                return ERR_PTR(-EINVAL);
        if (flags & AT_SYMLINK_NOFOLLOW)
                open_exec_flags.lookup_flags &= ~LOOKUP_FOLLOW;
        if (flags & AT_EMPTY_PATH)
                open_exec_flags.lookup_flags |= LOOKUP_EMPTY;

        file = do_filp_open(fd, name, &open_exec_flags);
        if (IS_ERR(file))
                return file;

        /*
         * In the past the regular type check was here. It moved to may_open() in
         * 633fb6ac3980 ("exec: move S_ISREG() check earlier"). Since then it is
         * an invariant that all non-regular files error out before we get here.
         */
        if (WARN_ON_ONCE(!S_ISREG(file_inode(file)->i_mode)) ||
            path_noexec(&file->f_path))
                return ERR_PTR(-EACCES);

        err = deny_write_access(file);
        if (err)
                return ERR_PTR(err);

        return no_free_ptr(file);
}
```

这里会调用 `do_filp_open()`。

接下来看 `openat` 和 `openat2`：

```
SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
                umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        return do_sys_open(dfd, filename, flags, mode);
}

SYSCALL_DEFINE4(openat2, int, dfd, const char __user *, filename,
                struct open_how __user *, how, size_t, usize)
{
        int err;
        struct open_how tmp;

        BUILD_BUG_ON(sizeof(struct open_how) < OPEN_HOW_SIZE_VER0);
        BUILD_BUG_ON(sizeof(struct open_how) != OPEN_HOW_SIZE_LATEST);
```

```
        if (unlikely(usize < OPEN_HOW_SIZE_VER0))
                return -EINVAL;
        if (unlikely(usize > PAGE_SIZE))
                return -E2BIG;

        err = copy_struct_from_user(&tmp, sizeof(tmp), how, usize);
        if (err)
                return err;

        audit_openat2_how(&tmp);

        /* O_LARGEFILE is only allowed for non-O_PATH. */
        if (!(tmp.flags & O_PATH) && force_o_largefile())
                tmp.flags |= O_LARGEFILE;

        return do_sys_openat2(dfd, filename, &tmp);
}
```

看 do_sys_open()，跟进去，发现会调用 do_sys_openat2：

```
static long do_sys_openat2(int dfd, const char __user *filename,
                           struct open_how *how)
{
        struct open_flags op;
        int fd = build_open_flags(how, &op);
        struct filename *tmp;

        if (fd)
                return fd;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(how->flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}

long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
        struct open_how how = build_open_how(flags, mode);
```

```
        return do_sys_openat2(dfd, filename, &how);
}
```

注意到 `do_sys_openat2()` **也会调用** `do_filp_open()`。

综上，`execveat` 会调用 `openat` `openat2` 调用到的函数，所以下面这个 poc 运行后会 core dump:

```
#include <stdio.h>
#include <stdlib.h>
#include <seccomp.h>
#include <unistd.h>
#include <fcntl.h>

int sandbox() {
    scmp_filter_ctx ctx;
    int rc;

    ctx = seccomp_init(SCMP_ACT_ALLOW);
    if (ctx == NULL) {
        perror("seccomp_init");
        return -1;
    }
    rc = seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(openat), 0);
    // seccomp_release(ctx);
    seccomp_load(ctx);
    return 0;
}


int main() {
    sandbox();
    char buffer[100];
    char *filename = "/bin/sh";
    char *argv[] = {"/bin/sh", NULL};
    char *envp[] = {NULL};
    // syscall(59, filename, argv, envp);
    syscall(322, AT_FDCWD, filename, argv, envp, 0);
    return 0;
}
```

`execve` 同理。

省流：ban `openat` 会把 execveat 一起 ban 了。

2.6

打了一下 HGame week1，做了几道题，感觉难度尚可，主要是这几天比赛撞一起了，有点难绷。

上号看了一会，感觉还好。

2.7

继续看 HGAME，晚秋教我了一个格式化字符串 `%*p`，可以往栈上填满空格，然后用 %s 泄漏 libc。

晚秋的这个思路我没打，我用的栈迁移也能出。

栈迁移学得真不行。

ida 静态审代码的时候别手贱隐藏强制类型转换信息，md 没看见整数溢出闹麻了。

`scanf` 的格式化字符串里写 `\n` 的都是 sb，此 `\n` 非彼 `\n`，在这里面写 `\n` 不会复合预期中过滤换行符的，所以说出题人纯 sb，测题的也是 sb。

> 可能令人吃惊，\n在scanf格式串中不表示等待换行符，而是读取并放弃连续的空白字符。（事实上，scanf格式串中的任何空白字符都表示读取并放弃空白字符。而且，诸如%d这样的格式也会扔掉前边的空白，因此你通常根本不需要在scanf格式串中加入显式的空白。） ——《你必须知道的495个C语言问题》- 第12章标准输入输出库

如上。

> 晚秋：你空洞骑士五门过了吗？
>
> 👴：？

2.8

好，早上随便讲了讲 ret2text，z3 说回学校请我吃饭，有点意思。

然后打 VNCTF，签到题是比较白给的 shellcode，拿下了。

然后下午开始坐牢，先是调那个莫名其妙的高通环境，本地调不出来，t1d 说可以打远程，ok那直接打。

> 👴：感觉应该是这么打的。 写、打、寄了 👴：草 t1d：草 晚秋：出了 👴：？

原来能搭本地调试环境啊，有点意思。但是我真搞不定栈迁移，调了一会也没啥名堂，寄了。

然后上了个新的堆题，看了一通口胡了一个打法感觉拿下了，正好这时候晚秋已经拿下了：

> 晚秋：白给题 👴：确实

我是 sb，上手打了一会发现根本不是这么回事，坐了会儿牢，晚秋爷爷看不下去了，告诉我这题是 house of muney。

还真是哈，学了之后就是板子题，秒了。

2.9 2.10

ok，开始坐牢。

首先来到战场的是 N1CTF junor，上来就是两个 fmt，第一个是堆上的 fmt，其实打过一两次，但是这次打的是 ld，我之前没打过，卡了一会。

然后是另一个 fmt，这个就比较白给了，库库打 got 表，大概是这样：

- 先泄漏 got 表拿到 libc base
- 把 stack_chk_fail@got 改成 puts，这样可以绕过 canary
- 把 atoi 改成 gets，无限栈溢出。

然后题目把 onegadget ban 了，所以打 openat + read + write，先用一次 `read` 把 flag 读到 bss 附近去。

这个时候前面那个 fmt 已经有点眉目了，但是我正在打 2.23 fastbin double free + 无 edit + chunk < 0x50 了.

坐牢坐麻了，调了很久的堆风水，总算是通过伪造一个能进 ub 的 fake chunk 泄漏 libc 了，问题来了，double free 打的 fake chunk 的 size 都是 0x70+，而我最大只能申请出 0x60 的，怎么办呢？

于是我翻了很久的博客，把我能想到的终端都试了试，最后还是没打出来。

还有一个 2.35 的堆，得逆 cjson，我逆完了，但是审不出洞，打游戏去了。

然后把第一个 fmt 简单写了下，打 exit 就可以。

打比赛真坐牢，快给我打自闭了。。。哎哎。