

title	published
某固件 Linux 内核逆向	false

缅甸、柬埔寨等东南亚地区国家的网络诈骗问题十分严重，相关人员罪行累累。

据悉，该地区大量使用一款国内某个小厂商 * 老版本固件的路由器，该版本于 2019 年发布了第一个 Release. 由于该厂商固件兼顾软路由、国产品牌文化认同等特性，导致该厂商产品在东南亚地区十分受欢迎。

开头

在过程中，不少师傅给我提供了思路. 虽然因为某些原因无法在本篇文章提到他名字，但是还是要特别感谢他们

声明：本篇提到的内核部分逆向基本上都是对开放源代码做逆向，patch 的解密代码部分也属于简单代码，未提及任何公司/人. 本文章所使用的 IDA 为免费试用版，代码均由自己逆向.

固件提取

首先尝试一下能不能直接解包固件，而不是从物理机器中扣 Flash 出来读取

从某处得到了该老版本的固件，观察发现可以提取出 Kernel 仅加密了文件系统，因此暂时没有从物理机中分析的必要. 首先拿到 .vmdk 后使用 qemu 挂载到 Kali 操作系统中，然后将内核和文件系统提取出来



一个便捷的挂载 vmdk 流程: qemu-nbd 虚拟挂载

```
1. 安装 qemu-utils
sudo apt update
sudo apt install qemu-utils
1. 加载 nbd 内核模块
sudo modprobe nbd max_part=16
1. 连接 vmdk 文件
sudo qemu-nbd -c /dev/nbd0 GuJian.vmdk
```

1. 查看分区情况

```
sudo fdisk -l /dev/nbd0
```

1. 挂载分区 (假设第一个分区是 /dev/nbd0p1)

```
sudo mkdir /mnt/vmdk
```

```
sudo mount /dev/nbd0p1 /mnt/vmdk
```

1. 访问文件

现在可以访问/mnt/vmdk目录下的文件:

```
ls /mnt/vmdk
```

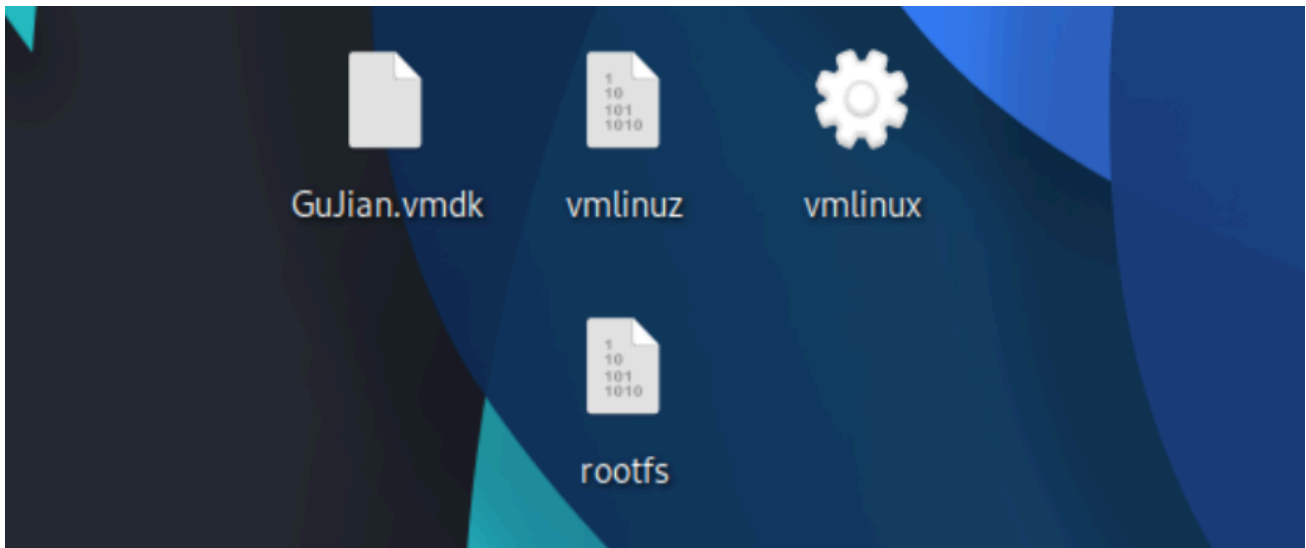
1. 卸载并断开连接

```
sudo umount /mnt/vmdk
```

```
sudo qemu-nbd -d /dev/nbd0
```

在挂载后, 将 /mnt/vmdk/boot 中的 vmlinuz(bzImage 内核)、rootfs(文件系统) 拿出来, 并使用 extract 脚本解压 vmlinuz

vmlinux 即为解压后的 vmlinuz linux 内核



该固件 Linux 系统为 X86 架构, 内核版本为 3.18.67

Linux Version 3.18.67 对应的仓库是: [Linux Version 3.18.67 - /](#)

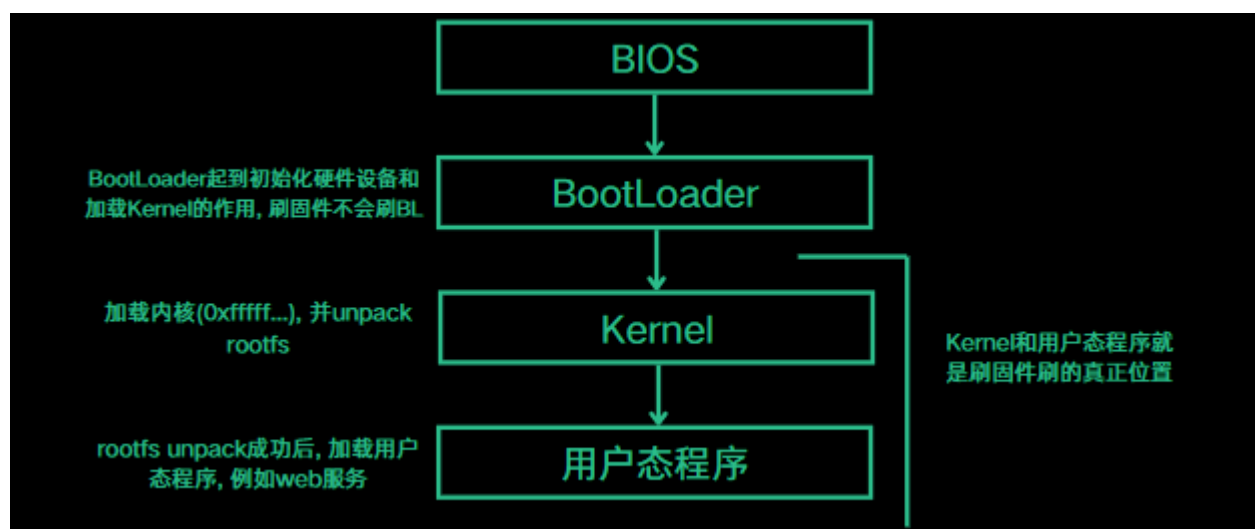


Kernel 分析与定位文件系统解密函数

在分析之前, 各位笔者需要知道固件上传与刷固件到底是怎么个刷法.

众所周知，计算机第一个加载的程序肯定是嵌入主板的 BIOS，BIOS 为计算机提供最底层的、最直接的硬件设置和控制。而后，根据情况 BIOS 会加载 bootloader，bootloader 可以看作是 BIOS 的延申，BIOS 的代码实在是太少了干不了什么事。然后，一个搭载了 Linux 操作系统的计算机 bootloader 会加载 Linux kernel，Linux kernel 会 unpack rootfs(root file system，根文件系统)然后加载用户态程序。

一个加载流程的示例图如下所示，其中可以观察到刷固件其实主要刷的是 kernel，并不会刷 bootloader。因为在刷失败后 bootloader 可以起 ip 服务作为系统救援的网络接口



cat /proc/mtd 查看分区后的结果如下所示，其中
mtd0 为 bootloader 分区
mtd2 为 kernel 分区

```
(root@kali) ~ - [~/home/kali/Desktop]
$ file vmlinuz
vmlinuz: Linux kernel x86 boot executable bzImage, version 3.18.67
```

我们的目的是提取出真实 rootfs，现在只需要直接分析厂商 patch 的内核就行，定位到文件系统解密函数

Kernel 分析

在此，笔者先简单讲一下内核加载文件系统(rootfs 全程 root file system, 根文件系统)流程，以方便笔者理解。由于本固件重点在于 populate_rootfs，因此笔者会细讲 populate_rootfs。

内核初始化时会依次调用处于 .init 段的初始化函数，其中与根文件系统相关的初始化函数为 default_rootfs 和 populate_rootfs，两个函数根据内核配置项的选择决定是否会被运行。在当前固件或者一般情况下，默认运行 populate_rootfs

populate_rootfs 在 Linux 3.18 中的主要逻辑及宏

```
static int __init populate_rootfs(void) {
    // 处理内置的 initramfs
    if (initrd_start != initrd_end) {
        unpack_to_rootfs((char *)initrd_start, initrd_end - initrd_start);
    }
    // 处理其他可能的 rootfs 初始化
    ...
}
rootfs_initcall(populate_rootfs);
```

populate_rootfs 负责解析过渡根文件系统 (early rootfs)，在 Linux 3.18 中作为宏的形式调用：上述代码中的 rootfs_initcall(populate_rootfs) 是一个 内核初始化宏，调用的 populate_rootfs 被放入 .init 段中，其作用是告诉内核在启动过程中调用 populate_rootfs() 函数。

如果该文件系统是 cpio 格式的 initramfs 或 initrd，populate_rootfs 会直接利用 unpack_to_rootfs 将其解压到根目录 /，以其内容初始化整个根文件系统。而当过渡根文件系统是 image 格式的 initrd (例如压缩过的磁盘镜像)，则需要先在内存中创建一个虚拟的 RAM 磁盘设备 (ramdisk)，再将其挂载后进行访问。

```
start_kernel()
-> rest_init()
-> kernel_init()
-> kernel_init_freeable()
-> do_basic_setup()
-> do_initcalls()
...(宏)...
-> populate_rootfs()
-> unpack_to_rootfs()
```

在 Linux 3.18 中 populate_rootfs 调用栈

让我们从 kernel_init 开始逐层分析 populate_rootfs 的调用路径。

在 kernel_init 中调用 kernel_init_freeable，旋即继续调用 do_basic_setup 执行系统的基本初始化任务，包括驱动加载、设备初始化、用户空间准备等

```
//linux/init/main.c
static int __ref kernel_init(void *unused) {
    int ret;

    /*
     * Wait until kthreadd is all set-up.
     */
```

```

        wait_for_completion(&kthreadd_done);

        kernel_init_freeable(); //调用栈
    ...
}

//linux/init/main.c
static ninline void __init kernel_init_freeable(void) {
    gfp_allowed_mask = __GFP_BITS_MASK;

    set_mems_allowed(node_states[N_MEMORY]);
    cad_pid = get_pid(task_pid(current));
    smp_prepare_cpus(setup_max_cpus);

    //...[初始化内核子队列、初始化内核管理系统]...
    //...[调用 SMP 初始化函数、初始化死锁检测子系统、调度器 SMP 相关的初始化]...
    //...[工作队列进行拓扑绑定优化]...

    padata_init();
    page_alloc_init_late();
    do_basic_setup(); //调用栈
    ...
}

//linux/init/main.c
static void __init do_basic_setup(void)
{
    //...[初始化 CPU 绑定和 NUMA 的相关设置、初始化内核设备驱动框架]...
    //...[调用内核中已注册的构造函数]...

    do_initcalls();
}

```

跳转到 do_initcalls, do_initcalls() 会按等级, 执行所有使用内核初始化宏 (如 core_initcall(), subsys_initcall() 等) 注册的初始化函数. 这些函数定义在内核和驱动模块中, 完成各种子系统或驱动的初始化

```

static void __init do_initcalls(void) {
    int level;
    size_t len = saved_command_line_len + 1;
    char *command_line;

    command_line = kzalloc(len, GFP_KERNEL);
    if (!command_line)
        panic("%s: Failed to allocate %zu bytes\n", __func__, len);

    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++) {

```

```

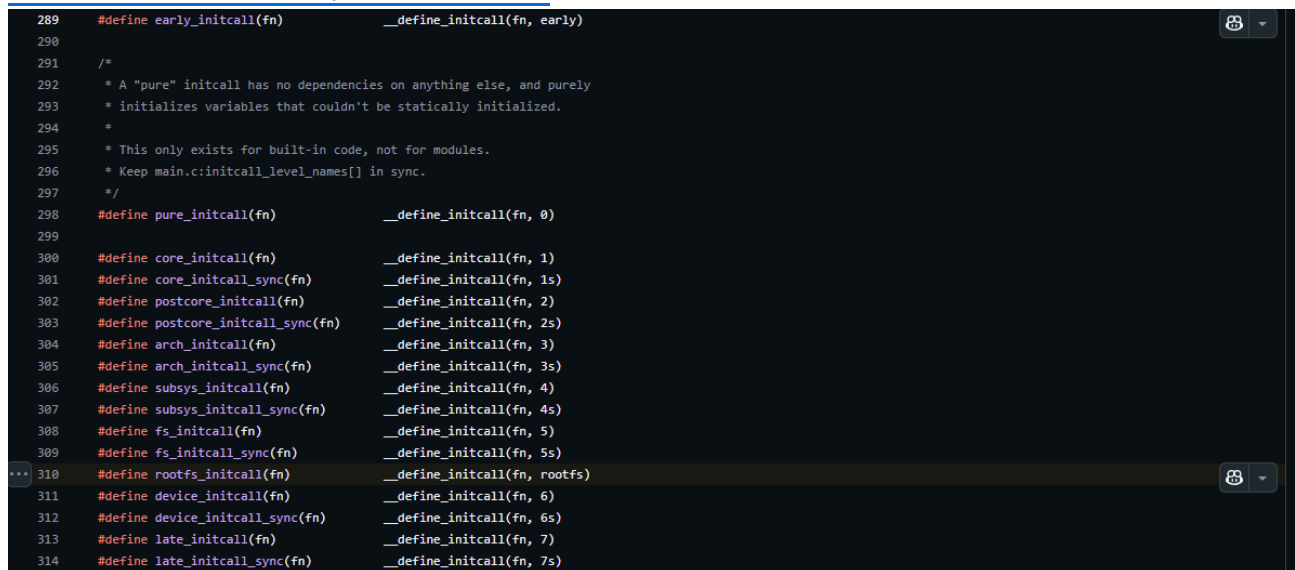
        /* Parser modifies command_line, restore it each time */
        strcpy(command_line, saved_command_line);
        do_initcall_level(level, command_line);
    }

    kfree(command_line);
}

```

各个定义的宏及其等级再次所示，你可以看到特别为 populate_rootfs 定义的 rootfs_initcall.

Linux Version 3.18.67 - /init.h Line: 310



```

289 #define early_initcall(fn)      __define_initcall(fn, early)
290
291 /*
292  * A "pure" initcall has no dependencies on anything else, and purely
293  * initializes variables that couldn't be statically initialized.
294  *
295  * This only exists for built-in code, not for modules.
296  * Keep main.c:initcall_level_names[] in sync.
297  */
298 #define pure_initcall(fn)       __define_initcall(fn, 0)
299
300 #define core_initcall(fn)       __define_initcall(fn, 1)
301 #define core_initcall_sync(fn)  __define_initcall(fn, 1s)
302 #define postcore_initcall(fn)  __define_initcall(fn, 2)
303 #define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
304 #define arch_initcall(fn)       __define_initcall(fn, 3)
305 #define arch_initcall_sync(fn)  __define_initcall(fn, 3s)
306 #define subsys_initcall(fn)     __define_initcall(fn, 4)
307 #define subsys_initcall_sync(fn) __define_initcall(fn, 4s)
308 #define fs_initcall(fn)         __define_initcall(fn, 5)
309 #define fs_initcall_sync(fn)    __define_initcall(fn, 5s)
310 #define rootfs_initcall(fn)     __define_initcall(fn, rootfs)
311 #define device_initcall(fn)     __define_initcall(fn, 6)
312 #define device_initcall_sync(fn) __define_initcall(fn, 6s)
313 #define late_initcall(fn)       __define_initcall(fn, 7)
314 #define late_initcall_sync(fn)  __define_initcall(fn, 7s)

```

这些宏都是围绕核心宏 __define_initcall(fn, id) 及其宏链来定义的：

```

...

#define __unique_initcall(fn, id, __sec, __iid) \
    __define_initcall(fn, \
        __initcall_stub(fn, __iid, id), \
        __initcall_name(initcall, __iid, id), \
        __initcall_section(__sec, __iid))

#define __define_initcall(fn, id, __sec) \
    __unique_initcall(fn, id, __sec, __initcall_id(fn))

//封装
#define __define_initcall(fn, id) __define_initcall(fn, id, .initcall##id)

```

看完了 populate_rootfs，继续看 populate_rootfs 中调用的 unpack_to_rootfs，unpack_to_rootfs 将 __initramfs_start 到 __initramfs_end 范围内的 cpio 压缩包内容，解压并提取到内存中的 rootfs（临时内存文件系统）中。其逻辑较为简单，因此笔者不再细讲。

unpack_to_rootfs 的参数和逻辑

```
char * __init unpack_to_rootfs(char *buf, unsigned long len)
{
    long written;
    decompress_fn decompress;
    const char *compress_name;

    //初始化数据各部分定义
    struct {
        char header[CPIO_HDRLEN];
        char symlink[PATH_MAX + N_ALIGN(PATH_MAX) + 1];
        char name[N_ALIGN(PATH_MAX)];
    } *bufs = kmalloc(sizeof(*bufs), GFP_KERNEL);

    if (!bufs) //...(报错)...

    ...
    state = Start;
    this_header = 0;
    message = NULL;
    while (!message && len) {
        loff_t saved_offset = this_header;
        if (*buf == '0' && !(this_header & 3)) {
            state = Start;
            written = write_buffer(buf, len);
            //...(跳过当前位, 长度减 1)...
            continue;
        }
        if (!*buf) {
            //...(跳过当前位, 长度减 1, this_header加 1)...
            continue;
        }
        this_header = 0;
        decompress = decompress_method(buf, len, &compress_name);
        if (decompress) {
            //解压流程
            int res = decompress(buf, len, NULL, flush_buffer, NULL,
&my_inptr, error);
            if (res) //...(报错)...
            } else //...(报错)...

            this_header = saved_offset + my_inptr;
            buf += my_inptr;
            len -= my_inptr;
        }

        //...(无关代码省略)...
```

```
    return message;
}
```

linux rootfs 的挂载流程实际上正如上述.

找到 populate_rootfs 的地址

笔者将先用关键字法找到 populate_rootfs 的地址, 所谓关键字法就是: 在引导程序结束后加载内核时, 观察输出, 使用报错或正常流程等手段抓 rootfs 解密关键字.

一般情况下, 只需要在系统引导加载内核前破坏掉 rootfs 中的部分值, 使其在解密后 hash 值对不上, 就可以让内核 panic, 发出 Illegal rootfs ... Invalid rootfs ... 等报错, 只要拿到报错语句就可以找到对应的字符串堆地址. 一般来说破坏 rootfs 的方法就是先将其挂载后, 再修改 rootfs 中的字节.

由于笔者调慢了速度, 可以不用破坏 rootfs 直接观测输出. 可以观察到窗口中输出了一个 .rodata 段中字符串 Trying to unpack rootfs ... , 然后到 ida 内核反编译窗口搜索字符串, 观察到该字符串确实被一个函数引用

```
[ 1.529835] UDP hash table entries: 2048 (order: 4, 65536 bytes)
[ 1.531924] UDP-Lite hash table entries: 2048 (order: 4, 65536 bytes)
[ 1.538645] NET: Registered protocol family 1
[ 1.540644] pci 0000:00:00.0: Limiting direct PCI/PCI transfers
[ 1.542322] pci 0000:00:01.0: PIIX3: Enabling Passive Release
[ 1.543924] pci 0000:00:01.0: Activating ISA DMA hang workarounds
[ 1.583116] Trying to unpack rootfs image as initramfs...
[ 61.600218] INFO: rcu_sched detected stalls on CPUs/tasks: {} (detected by 0, t=6002 jiffies, g=-228, c=-229, q=84)
[ 61.600218] INFO: Stall ended before state dump start

.rodata:FFFFFFF8194FE30 aTryingToUnpack db 1 ; DATA XREF: REVERSE_populate_rootfs+384o
.rodata:FFFFFFF8194FE31 db 36h ; 6
.rodata:FFFFFFF8194FE32 db 54h ; T
.rodata:FFFFFFF8194FE33 db 72h ; r
.rodata:FFFFFFF8194FE34 db 79h ; y
.rodata:FFFFFFF8194FE35 db 69h ; i
.rodata:FFFFFFF8194FE36 db 6Eh ; n
.rodata:FFFFFFF8194FE37 db 67h ; g
.rodata:FFFFFFF8194FE38 db 20h
.rodata:FFFFFFF8194FE39 db 74h ; t
.rodata:FFFFFFF8194FE3A db 6Fh ; o
```

在这里前缀的 0x1 0x36 为打印级别, 可忽略

该字符串的引用如下所示, 在该函数的这个地方 printk 输出. 猜测这个函数和 populate_rootfs 、 unpack_to_rootfs 脱不了干系, 这下算找到了算是和 rootfs 加解密区域相关联的函数.

```
1 __int64 REVERSE_populate_rootfs()
2 {
3     __int64 err; // rax
4     __int64 initrd_start; // rdi
5     int v2; // edx
6     int initrd_size; // rsd
7     __int64 v4; // rax
8     char *v5; // rcx
9     char *v6; // rax
10    char v7; // si
11    int v8; // ecx
12    __int64 v9; // rdx
13    __int64 v10; // rsi
14    char v11; // r8
15    __BYTE_V12; // rsi
16    __int64 i; // rsi
17    int v14; // eax
18    signed int v15; // eax
19    unsigned int v16; // ebx
20
21    err = REVERSE_unpack_to_rootfs(initramfs_start, (__int64)initramfs_size);
22    if (err)
23        panic((unsigned int)&Percents, err);
24    if (initrd_start)
25    {
26        printk((unsigned int)&aTryingToUnpack);
27        _initrd_start = initrd_start;
```

在此, 笔者已经大量还原符号, 方便笔者阅读

仔细阅读该函数反编译的源码，发现了 `unpack_to_rootfs` 函数算法的痕迹，再根据源码比较发现大多数的函数是差不多的. 可以大致推测该函数就是 `populate_rootfs` 函数. 并且与 Linux 源码做对比发现此处加了个解密的大 patch，应该就是对 `initrd` 进行解密

Linux Version 3.18.67 - /init/initramfs.c Line: 451

```
605 static int __init populate_rootfs(void)
606 {
607     /* Load the built in initramfs */
608     char *err = unpack_to_rootfs(_initramfs_start, _initramfs_size);
609     if (err)
610         panic("%s", err); /* Failed to decompress INTERNAL initramfs */
611
612     if (!initrd_start || IS_ENABLED(CONFIG_INITRAMFS_FORCE))
613         goto done;
614
615     if (IS_ENABLED(CONFIG_BLK_DEV_RAM))
616         printk(KERN_INFO "Trying to unpack rootfs image as initramfs...\n");
617     else
618         printk(KERN_INFO "Unpacking initramfs...\n");
619
620     err = unpack_to_rootfs((char *)initrd_start, initrd_end - initrd_start);
621     if (err) {
622         #ifdef CONFIG_BLK_DEV_RAM
623             populate_initrd_image(err);
624         #else
625             printk(KERN_EMERG "Initramfs unpacking failed: %s\n", err);
626         #endif
627     }
628
629     done:
630     /*
631      * If the initrd region is overlapped with crashkernel reserved region,
632      * free only memory that is not part of crashkernel region.
633      */
634     if (!do_retain_initrd && initrd_start && !kexec_free_initrd())
635         free_initrd_mem(initrd_start, initrd_end);
636     initrd_start = 0;
637     initrd_end = 0;
638
639     flush_delayed_fput();
640     return 0;
641 }
642 rootfs_initcall(populate_rootfs);
```

Linux 3.18 内核中 /init/initramfs.c `populate_rootfs` 函数源代码

```
23     panic((unsigned int)&Percents, err);
24     if (!initrd_start)
25     {
26         printk((unsigned int)&TryingToUnpack);
27         _initrd_start = initrd_start;
28         iter = 0;
29         initrd_size = initrd_end - initrd_start;
30         while (iter < initrd_size)
31         {
32             v4 = iter;
33             iter += 2;
34             first = (char *)(_initrd_start + v4);
35             second = (char *)(_initrd_start + v4 + 1);
36             temp_1 = *first;
37             *first = *second;
38             *second = temp_1;
39         }
40         v8 = initrd_size - 1;
41         iter_1 = 0LL;
42         while (initrd_size / 2 > (int)iter_1)
43         {
44             v10 = v8;
45             v11 = *((_BYTE *) (iter_1 + _initrd_start));
46             --v8;
47             v12 = *((_BYTE *) (_initrd_start + v10));
48             *((_BYTE *) (++iter_1 + _initrd_start - 1)) = *v12;
49             *v12 = v11;
50         }
51         for (iter_2 = 0LL;
52              (int)iter_2 < initrd_size;
53              *(temp_2 + _initrd_start - 1) = ((int)*(unsigned __int4 *) (iter_2 + _initrd_start - 1) >> (temp % 7 + 1)) | (*(temp_2 + _initrd_start - 1) << (7 - temp % 7)))
54         {
55             temp = (int)iter_2 * ((int)iter_2 + 1) % 32 + iter_2;
56             ++iter_2;
57         }
58         if (REVERSE_unpack_to_rootfs((char *)initrd_start, initrd_end - initrd_start))
59         {
60             sub_FFFFFFFF81024553();
61             REVERSE_unpack_to_rootfs(initramfs_start, (__int64)initramfs_size);
62             printk((unsigned int)&RootfsIsNotRamfs);
63             fd = sub_FFFFFFFF810F14A8((__int64)"/initrd.image", 0x41u, 0x100u);
64             fd = fd;
65             if (fd < 0)
66             {
67                 ...
68             }
69         }
```

逆向后的 `populate_rootfs`，红圈中的应该是解密混淆部分

函数逆向

在明确了解密算法区域后，接下来就是逆向解密算法了. 其实在这里逆不逆向无所谓，直接 `pwndbg attach` 内核调试然后将解密解混淆后内存中的文件系统拉下来就行. 笔者主要讲个思路，以防遇到不能轻易 `attach` 的情况

1. 逆向分为三个步骤，首先先初始化 `initrd_start` 和 `initrd_size`，方便后续调用. 在动态调试时这个地方 `initrd_size` 大小和 `vmdk` 中提取出来的 `rootfs` 大小一样，就可以确认这个地方是在对 `rootfs` 进行操作，而非生成密钥

```
_initrd_start = initrd_start;
iter = 0;
initrd_size = initrd_end - initrd_start;
```

2. 观察如下面第一个解密区域，首先迭代位的步数是：0、2、4、6、...，然后块中分别取了 `first = (char *)(_initrd_start + 迭代);`、`second = (char *)(_initrd_start + v4 + 1);` 并将 `first` 和 `second` 交换位置. 这不就是每两个字节调换值吗

```
30 while ( iter < initrd_size )
31 {
32     v4 = iter;
33     iter += 2;
34     first = (char *)(_initrd_start + v4);
35     second = (char *)(_initrd_start + v4 + 1);
36     temp_1 = *first;
37     *first = *second;
38     *second = temp_1;
39 }
```

其逆向的伪代码再次逆向后的正确实现方式应该是：

```
for(int i = 0; i < initrd_size; i += 2) {
    unsigned char temp = initrd[i];
    initrd[i] = initrd[i + 1];
    initrd[i + 1] = temp;
}
```

调换前

{1, 2, 3, 4, 5, 6, 7, ...}

调换后

{2, 1, 4, 3, 6, 5, 8, ...}

3. 继续观察第二个解密区域，首先该区域只进行了一半的迭代，其时间消耗砍半，这让笔者想起了字符串翻转. 其次，`iter_1` 对应前序指针，遍历方向为从前向后，`v8` 对应末尾指针，遍历方向对应从后向前. `v11` 即为前序指针对应的值，`v12` 即为末尾指针对应的值.

```
40 v8 = initrd_size - 1;
41 iter_1 = 0LL;
42 while ( initrd_size / 2 > (int)iter_1 )
43 {
44     v10 = v8;
45     v11 = *(_BYTE *) (iter_1 + _initrd_start);
46     --v8;
47     v12 = *(_BYTE *) (_initrd_start + v10);
48     *(_BYTE *) (++iter_1 + _initrd_start - 1) = *v12;
49     *v12 = v11;
50 }
```

`v11 = *(_BYTE *)(iter_1 + _initrd_start);` 首先 `v11` 保存了末尾指针的值，然后将 `v12` 赋值给末尾指针后，在把 `v11` 的值赋值给 `v12` . 至此笔者可以确定，该函数就是把前后数据大翻转，其逆向的伪代码再次逆向后的正确实现方式应该是：

```
int back = initrd_size - 1;
int i = 0;
while(initrd_size / 2 > i){
    unsigned char first = initrd[i];
    initrd[i] = initrd[back];
    initrd[back] = first;

    back --;
    i ++;
}
```

调换前

{1, 2, 3, ... 98, 99, 100}

调换后

{100, 99, 98, ... 3, 2, 1}

4. 第三个解密区域的逻辑比较复杂，大致可剥离为如下步骤

首先取个临时变量 `temp = (int)iter_2 * ((int)iter_2 + 1) % 32 + iter_2;`，然后再循环赋值. 将循环赋值的代码用伪代码表示：当前值 = (当前值 >> (temp % 7 + 1)) | (当前值 << (7 - temp % 7))

首先分析 `temp` 变量的生成过程，下面将以 `an` 代替 `temp` 变量：

$$a_n = n + n(n + 1) \bmod 32, n \in \mathbb{Z} \geq 0$$

$$\begin{aligned} & \text{\textit{非线性}} \quad \& \quad n(n+1) \bmod 32 \quad \text{\textit{模运算}} \quad \& \quad \\ & \bmod 32 \quad \text{\textit{叠加线性项}} \quad \& \quad n \end{aligned}$$

在密码学上这是一个一个非线性的扰动序列，提供随 `n` 增长而不完全线性、且带周期性 [0, 31] 整数波动的扰动值. 观察完 `an` 后，注意到， $(a_n \% 7 + 1) + (7 - a_n \% 7) == 8$ 为真，那么继续观察 当前值C 的生成过程：

$$C \leftarrow C \gg (a_n \bmod 7 + 1) \mid C \ll (7 - a_n \bmod 7)$$

以 `C` 为 166，`an` 为 4 为例，在进行计算时如下所示：

$$10100110(bin) \gg 5 = 00000101(bin)$$

$10100110(bin) \ll 3 = 00110000(bin)$

$00000101(bin) \mid 00110000(bin) = 00110101$

不难发现，将以上两者的计算结果进行二进制"或"运算就是循环右移 a_n 位. 接下来我们直接实现该算法：

```

51 for ( iter_2 = 0LL;
52       (int)iter_2 < initrd_size;
53       *(_BYTE *) (iter_2 + _initrd_start - 1) = ((int)*(unsigned __int8 *) (iter_2 + _initrd_start - 1) >> (temp % 7 + 1)) | (*( _BYTE *) (iter_2 + _initrd_start - 1) << (7 - temp % 7)) )
54 {
55     temp = (int)iter_2 * ((int)iter_2 + 1) % 32 + iter_2;
56     ++iter_2;
57 }

```

```

for (iter_2 = 0LL; iter_2 < initrd_size; ) {
    temp = iter_2 * (iter_2 + 1) % 32 + iter_2;
    ++iter_2;
    当前值 = (当前值 >> (temp % 7 + 1)) | (当前值 << (7 - temp % 7))
}

```

该算法的一个 Python 实现如下所示

```

data = 第三步输入数据
i = 0
while True:
    v14 = i * (i + 1) % 32 + i
    i += 1
    # 这个地方 0xFF 太多
    data[i - 1] = ((data[i - 1] >> (v14 % 7 + 1)) & 0xFF) | ((data[i - 1] <<
(7 - v14 % 7)) & 0xFF)
    if i >= len(data):
        break

```

整个解密函数(Python 语言版本):

```

# by leeya_bug
import os

with open('./rootfs', 'rb') as f:
    data = f.read()
data = bytearray(data)

# first
for i in range(0, len(data), 2):
    data[i], data[i + 1] = data[i + 1], data[i]
data = data[::-1] # second
# third
i = 0
while True:

```

```
v14 = i * (i + 1) % 32 + i
i += 1
data[i - 1] = ((data[i - 1] >> (v14 % 7 + 1)) & 0xFF) | ((data[i - 1] <<
(7 - v14 % 7)) & 0xFF)
if i >= len(data):
    break

data1 = bytes(data)
with open('rootfs.decoded', 'wb') as f:
    f.write(data1)
```

文件系统解密

虽然是有了 Python 脚本，但是为了确保解密过程不出错，笔者最终还是选用了通过调试内核的手段，直接提取文件系统。在调试内核前，请注意将虚拟机 watchdog 触发时长设置为足够高(最好是一两个小时)，以避免在 gdb 调试时 watchdog 抛出 CPU stuck

```
#!/bin/bash
qemu-system-x86_64 \
    -no-reboot \
    -m 4G \
    -smp 1 \
    -hda ./GuJian.vmdk \
    -kernel ./vmlinuz \
    -initrd ./rootfs \
    -append "console=ttyS0 nmi_watchdog=0 nowatchdog watchdog_thresh=1200" \ #修改
watchdog_thresh 时长为 20 分钟
    -S \
    -gdb tcp::1234 \ #监听 1234 端口
    -nographic \
```

使用 pwndbg 分析内核，在 populate_rootfs 结束解密后 call REVERSE_unpack_to_rootfs 地址为 0xFFFFFFFF81D25101 的位置打断点。

连接并 attach qemu 启动的内核调试接口，在一切正常的情况下理应会在 populate_rootfs 中的 REVERSE_unpack_to_rootfs 前卡住调试。

运行到此时，rdi 的值正好是解密后数据的位置。

```

37  *first = *second;
38  *second = temp_1;
39  }
40  v8 = initrd_size - 1;
41  iter_1 = 0LL;
42  while ( (initrd_size / 2 > (int)iter_1 )
43  {
44      v10 = v8;
45      v11 = *(_BYTE *) (iter_1 + _initrd_start);
46      --v8;
47      v12 = (_BYTE *) (_initrd_start + v10);
48      *(_BYTE *) (++iter_1 + _initrd_start - 1) = *v12;
49      *v12 = v11;
50  }
51  for ( iter_2 = 0LL;
52      (int)iter_2 < initrd_size;
53      *(_BYTE *) (iter_2 + _initrd_start - 1) = ((int) (unsigned __int8 *) (iter_2 + _initrd_start - 1) >> (temp % 7 + 1)) | (*(_BYTE *) (iter_2 + _initrd_start - 1) << (7 - temp % 7)) )
54  {
55      temp = (int)iter_2 * ((int)iter_2 + 1) % 32 + iter_2;
56      ++iter_2;
57  }
58  if ( REVERSE_unpack_to_rootfs(((_BYTE *)_initrd_start, initrd_end - initrd_start) )
59  {
60      sub_FFFFFFFF81D24553();
61      REVERSE_unpack_to_rootfs(initramfs_start, (__int4)initramfs_size);
62      printf((unsigned int)&nootfs:notramfs);
63      fd = sub_FFFFFFFF81D245A0((__int4)"/initrd.image", 0x41u, 0x100u);
64      _fd = fd;
65      if ( fd < 0 )
66      {
67          LABEL_19:
68              sub_FFFFFFFF81D23293();
69              return 0LL;
70      }
71      if ( sub_FFFFFFFF81D24CA4(fd, initrd_start, initrd_end - initrd_start) != initrd_end - initrd_start )
72          printf((unsigned int)&unk_FFFFFFFF8194FEA8);
73
74      .init.text:FFFFFFF81D25001      movzx  eax, byte ptr [rsi+rdi-1]
75      .init.text:FFFFFFF81D25006      mov    r11d, eax
76      .init.text:FFFFFFF81D25009      sub    ecx, edx
77      .init.text:FFFFFFF81D2500B      shl    r11d, cl
78      .init.text:FFFFFFF81D2500E      lea    ecx, [rdx+1]
79      .init.text:FFFFFFF81D250E1      sar    eax, cl
80      .init.text:FFFFFFF81D250E3      or     r11d, eax
81      .init.text:FFFFFFF81D250E6      mov    [rsi+rdi-1], r11b
82      .init.text:FFFFFFF81D250EB      loc_FFFFFFFF81D250EB:                                ; CODE XREF: REVERSE_populate_rootfs+8A1fj
83      .init.text:FFFFFFF81D250EB      cmp    esi, r9d
84      .init.text:FFFFFFF81D250EE      j1l    short loc_FFFFFFFF81D250BA
85      .init.text:FFFFFFF81D250F0      mov    rdi, cs:iniramfs_start
86      .init.text:FFFFFFF81D250F7      mov    rsi, cs:iniramfs_end
87      .init.text:FFFFFFF81D250FE      sub    rsi, rdi
88      .init.text:FFFFFFF81D25101      call   sub_FFFFFFFF81D24553
89      .init.text:FFFFFFF81D25106      test   rax, rax
90      .init.text:FFFFFFF81D25109      mov    rbx, rax
91      .init.text:FFFFFFF81D2510C      jnz    short loc_FFFFFFFF81D25113
92      .init.text:FFFFFFF81D2510E      jmp     loc_FFFFFFFF81D2519C
93      .init.text:FFFFFFF81D25113      ; -----
94      .init.text:FFFFFFF81D25113      loc_FFFFFFFF81D25113:                                ; CODE XREF: REVERSE_populate_rootfs+10E1fj
95      .init.text:FFFFFFF81D25113      call   sub_FFFFFFFF81D24553
96      .init.text:FFFFFFF81D25118      mov    rsi, cs:iniramfs_size
97      .init.text:FFFFFFF81D2511F      mov    rdi, offset initramfs_start
98      .init.text:FFFFFFF81D25126      call   REVERSE_unpack_to_rootfs

```

在 pwndbg 中打断点并运行到此处，观察到此时 rdi 的值为 0xfffff8800be3d9000 .

直接使用命令 dump memory dump.bin 0xfffff8800be3d9000 0xfffff8800be3d9000 + 0x1c06374
将数据 dump 到文件中

```

pwndbg> context
LEGEND: STACK | HEAP | CODE | DATA | BWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
-RAX 0x5a
-RBX 0xffffffff81a6b020 <- and byte ptr [rax - 0x7e5a], dh /* 0xffffffff81a6b020 */
-RCX 0x1
-RDX 0x0
-RDI 0xfffff8800be3d9000 <- 0x100005a587a37fd
-RSI 0x1c06374
-R8 0x7
-R9 0x1c06374
-R10 0x20
-R11 0x5a5a
-R12 0x0
-R13 0xffffffff81d247ba <- mov rsi, qword ptr [rip + 0xb7a8b] /* 0x53000b7a8b358b48 */
-R14 0x0
-R15 0x0
-RBP 0xfffff880139dff20 -> 0xfffff880139dff210 -> 0xfffff880139dff220 -> 0xfffff880139dff230 -> 0xfffff880139dff240 <- ...
-RSP 0xfffff880139c7e98 <- 0xffffffff81a6b020 <- and byte ptr [rax - 0x7e5a], dh /* 0xffffffff81a6b020 */
-RIP 0xffffffff81d247ba <- call 0xffffffff81d247ba /* 0xc08548fffffb0b4e8 */
[ DISASM / x86-64 / set emulate on ]
> 0xffffffff81d25101 call 0xffffffff81d247ba <0xffffffff81d247ba>
0xffffffff81d25106 test rax, rax
0xffffffff81d25109 mov rbx, rax
0xffffffff81d2510c jne 0xffffffff81d25113 <0xffffffff81d25113>
0xffffffff81d2510e jmp 0xffffffff81d2519c <0xffffffff81d2519c>
0xffffffff81d25113 call 0xffffffff81d24553 <0xffffffff81d24553>
0xffffffff81d25118 mov rsi, qword ptr [rip + 0xb7971]
0xffffffff81d2511f mov rdi, 0xffffffff81ddc9e8
0xffffffff81d25126 call 0xffffffff81d247ba <0xffffffff81d247ba>
0xffffffff81d2512b mov rsi, rbx
0xffffffff81d2512e mov rdi, 0xffffffff8194fe68

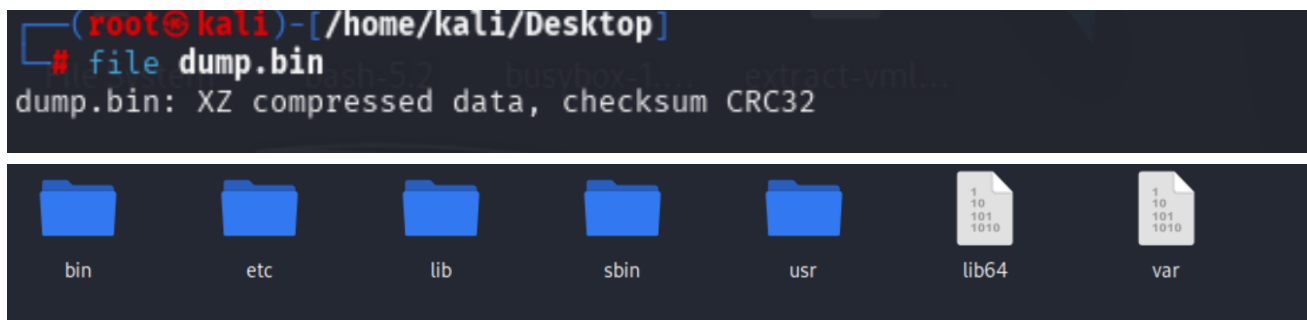
```

```

pwndbg> x/40gx 0xfffff8800be3d9000
0xfffff8800be3d9000: 0x0100005a587a37fd 0x9cf1c00436de2269 0xb2058eb000000016 0x3081b7a3ffffd6f
0xfffff8800be3d9001: 0x00005dfeef353fee 0x3ab322a24e767a60 0x5b07c763c0bba 0xbfb03aeb55cb359cf
0xfffff8800be3d9002: 0x9c3d9aec91bc66ae 0xc5a4a4f470830cf2 0x1b80f4519a439a7c 0x05b627ce1a2fcca2
0xfffff8800be3d9003: 0x0816fd4cd091f000 0x988484623c7e8dc2 0x59b383d5263a185b 0x6ca357e7d7dfceac
0xfffff8800be3d9004: 0xa20fa71af87ba08 0xa7e452ef8367c452 0x40924010c17be772 0x75b4a34765a8701d
0xfffff8800be3d9005: 0xc99357f0ee6b6d32 0x8b71d867b0c2a38c 0xae6f73014cb2777 0x20abb1ea5ccf601
0xfffff8800be3d9006: 0x1956a1a54c359850 0xa8206781fe8c3409 0x16c014558ddccbc 0x1a1ebdb47d4912fa
0xfffff8800be3d9007: 0xb6570272d27a9c2 0xc7e352f3cf012df6 0xaa09a77e6ee9401 0x43fd6208d0baafa3
0xfffff8800be3d9008: 0x8c78d2f23879aa99 0xef64d3f3cfff4d99

```

数据是 .xz 格式的压缩包，首先使用 `xz -d <压缩包>` 将其解压后，使用 binwalk 就可以将他的文件系统解出来



<script type="text/javascript" async src="<https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-autoload.js>"> </script>