Magisk Sepolicy

https://topjohnwu.github.io/Magisk/details.html

> Magisk will patch the stock sepolicy to make sure root and Magisk operations can be done in a safe and secure way. The new domain magisk is effectively permissive, which is what magiskd and all root shell will run in. magisk_file is a new file type that is setup to be allowed to be accessed by every domain (unrestricted file context).

> Before Android 8.0, all allowed su client domains are allowed to directly connect to magiskd and establish connection with the daemon to get a remote root shell. Magisk also have to relax some ioctl operations so root shells can function properly.

> After Android 8.0, to reduce relaxation of rules in Android's sandbox, a new SELinux model is deployed. The magisk binary is labelled with magisk_exec file type, and processes running as allowed su client domains executing the magisk binary (this includes the su command) will transit to magisk_client by using a type_transition rule. Rules strictly restrict that only magisk domain processes are allowed to attribute files to magisk_exec. Direct connection to sockets of magiskd are not allowed; the only way to access the daemon is through a magisk_client process. These changes allow us to keep the sandbox intact, and keep Magisk specific rules separated from the rest of the policies.

> magisk定义的sepolicy规则都在magiskpolicy/rules.cpp可以找到，如果要自己编译magisk内核，也可以在这里增加或修改.

大致含义是说Magisk为了保证安全和隐秘的root，修补了系统的sepolicy文件，并且可以在源码magiskpolicy/rules.cpp里看到magisk新增的一些sepolicy规则。

同时在Magisk插件的结构规范里也可以看到Magisk支持在sepolicy.rule文件里自定义sepolicy规则。

本篇文章的目标就是弄清楚这些se规则是如何被Magisk加载并替换到系统里最终生效的。

SElinux/SEAndroid/Sepolicy

SELinux 全称 Security Enhanced Linux (安全强化 Linux)，是MAC (Mandatory Access Control，强制访问控制系统)的一个实现。其目的在于明确的指明某个进程可以访问哪些资源(文件、网络端口等)。

SEAndroid是在SELinux基础上针对Android独有的用户空间运行时所开发。

SELinux具体控制权限的策略由label标签和policy规则来实现，标签由四部分内容组成，分别是SELinux用户、SELinux 角色、类型、安全级别，每一个部分都通过一个冒号来分隔，格式为“user:role:type:rank”。

规则policy采用以下形式：allow domains types:classes permissions;，其中：

- Domain - 一个进程或一组进程的标签。也称为域类型，因为它只是指进程的类型。
- Type - 一个对象（例如，文件、套接字）或一组对象的标签。
- Class - 要访问的对象（例如，文件、套接字）的类型。
- Permission - 要执行的操作（例如，读取、写入）。

Magisk内核加载流程

熟悉Magisk安装的应该知道，Magisk刷入的核心是patch boot.img，boot.img中包含了kernel和ramdisk两部分，ramdisk中又包含了初始化系统进程的所有文件，例如kernel启动后寻找的用于启动init进程的init.rc文件（/system/core/rootdir/init.rc ）和用户空间第一个进程init。

其中init进程启动可以分为三个阶段：

- FirstStageMain
- SetupLinux
- SecondStageMain

Magisk内核刷入的过程正是修改ramdisk的过程，用magiskinit替换init进程或者init.rc来接替系统原本的init进程接管系统（具体如何patch还需看看源码）。官方文档也是如此描述：

> magiskinit will replace init as the first program to run.

并且还详细介绍了magiskinit的流程:

- Early mount required partitions. On legacy system-as-root devices, we switch root to system; - on 2SI devices, we patch fstab and execute the original init to mount partitions for us.
- Load sepolicy either from /sepolicy, precompiled sepolicy in vendor, or compile split sepolicy
- Patch sepolicy rules and dump to /sepolicy or /sbin/.se or /dev/.se
- Patch init or libselinux.so to force the system to load the patched policies
- Inject magisk services into init.rc
- Execute the original init to continue the boot process

可以看到magisk是在系统原始的init进程执行之前，先加载系统和自己的sepolicy规则然后也是通过patch原始sepolicy的方式注入magisk的sepolicy。

magisk中init/init.cpp可能就对应着系统的init.cpp，在magisk init.cpp的main函数中，会根据不同条件选择不同的init实现：

```
if (argc > 1 && argv[1] == "selinux_setup"sv) {
        setup_klog();
        init = new SecondStageInit(argv);
    } else {
        // This will also mount /sys and /proc
        load_kernel_info(&cmd);

        if (cmd.skip_initramfs)
            init = new SARInit(argv, &cmd);
        else if (cmd.force_normal_boot)
            init = new FirstStageInit(argv, &cmd);
        else if (access("/sbin/recovery", F_OK) == 0 ||
access("/system/bin/recovery", F_OK) == 0)
            init = new RecoveryInit(argv, &cmd);
        else if (check_two_stage())
            init = new FirstStageInit(argv, &cmd);
        else
            init = new RootFSInit(argv, &cmd);
    }

    // Run the main routine
    init->start();
    exit(1);
```

正常情况下走哪种实现，还需加上日志排查一番，但是，可以通过看代码的方式发现加载magisk模块的自定义的sepolicy的实现所在：

- MagiskInit::mount_rules_dir 挂载自定义sepolicy规则文件所在的路径
- MagiskInit::patch_sepolicy 加载自定义sepolicy

mount_rules_dir

```
// dev_base为/dev/block
// mnt_base为/sbin/.magisk/mirror
void MagiskInit::mount_rules_dir(const char *dev_base, const char
*mnt_base) {
    ...
    // 第一个尝试userdata路径
    strcpy(blk_info.partname, "userdata");
    strcpy(b, "/data");
    strcpy(p, "/data");
    if (setup_block(false) < 0) {
        ...
        if (setup_block(false) < 0)
            // 尝试/sbin/.magisk/mirror/magisk/cache
            goto cache;
    }

    if (!do_mount("ext4"))
        // 尝试/sbin/.magisk/mirror/magisk/cache
        goto cache;

    strcpy(p, "/data/unencrypted");
    if (xaccess(path, F_OK) == 0) {
        // 尝试/sbin/.magisk/mirror/..../unencrypted
        custom_rules_dir = path + "/magisk"s;
    } else {
        // Skip if /data/adb does not exist
        strcpy(p, SECURE_DIR);
        if (xaccess(path, F_OK) != 0)
            return;
        strcpy(p, MODULEROOT);
        if (xaccess(path, F_OK) != 0) {
            goto cache;
        }
        // Unencrypted, directly use module paths
        custom_rules_dir = string(path);
    }
    goto success;

cache:
    // Fallback to cache
    strcpy(blk_info.partname, "cache");
    strcpy(b, "/cache");
    strcpy(p, "/cache");
    if (setup_block(false) < 0) {
        // Try NVIDIA naming scheme
        strcpy(blk_info.partname, "CAC");
        if (setup_block(false) < 0)
            goto metadata;
    }
    if (!do_mount("ext4"))
        goto metadata;
    custom_rules_dir = path + "/magisk"s;
    goto success;

metadata:
    // Fallback to metadata
    strcpy(blk_info.partname, "metadata");
    strcpy(b, "/metadata");
    strcpy(p, "/metadata");
    if (setup_block(false) < 0 || !do_mount("ext4"))
        goto persist;
    custom_rules_dir = path + "/magisk"s;
    goto success;

persist:
    // Fallback to persist
    strcpy(blk_info.partname, "persist");
    strcpy(b, "/persist");
    strcpy(p, "/persist");
    if (setup_block(false) < 0 || !do_mount("ext4"))
        return;
    custom_rules_dir = path + "/magisk"s;

success:
    // Create symlinks so we don't need to go through this logic again
    strcpy(p, "/sepolicy.rules");
    xsymlink(custom_rules_dir.data(), path);
}
```

挂载部分在21.4版本中会根据不同条件选择具体sepolicy挂载的路径，20.4都是在persist中，猜测可能是为了兼容不同机型。

MagiskInit::patch_sepolicy

挂载之后，magisk便知道从哪里读取模块自定义的sepolicy文件。

```
    if (!custom_rules_dir.empty()) {
        if (auto dir = xopen_dir(custom_rules_dir.data())) {
            for (dirent *entry; (entry = xreaddir(dir.get()));) {
                auto rule = custom_rules_dir + "/" + entry->d_name + "/sepolicy.rule";
                if (xaccess(rule.data(), R_OK) == 0) {
                    LOGD("Loading custom sepolicy patch: [%s]\n", rule.data());
                    sepol->load_rule_file(rule.data());
                }
            }
        }
    }
```

在日志中，也可以根据Loading custom sepolicy patch:检查模块的sepolicy.rule是否生效。

> 查看内核日志: 开机后adb shell dmesg

custom_rules_dir中sepolicy从何而来

magisk加载模块自定义的sepolicy的流程已经找到，但是模块的sepolicy.rule如何被拷贝到这个路径，还需再分析一下。

首先看模块安装的流程，模块最后安装都是通过执行模块文件中的update-binary完成，这也是为什么编写magisk模块时这个文件要求从magisk源码中拷贝。

```
[ -f /data/adb/magisk/util_functions.sh ] || require_new_magisk
. /data/adb/magisk/util_functions.sh
[ $MAGISK_VER_CODE -lt 20400 ] && require_new_magisk

install_module
```

关键代码为最后这部分执行/data/adb/magisk/util_functions.sh的install_module方法，开头一般都是用于校验magisk内核版本。

在raw/util_functions.sh里可以看到模块文件夹被创建、文件结构中声明的customize.sh、install.sh、uninstall.sh等被执行的时机，以及我们关注的sepolicy.rule的处理流程：copy_sepolicy_rules函数。

```
copy_sepolicy_rules() {
  # Remove all existing rule folders
  rm -rf /data/unencrypted/magisk /cache/magisk /metadata/magisk /persist/magisk /mnt/vendor/persist/magisk

  # Find current active RULESDIR
  local RULESDIR
  local active_dir=$(magisk --path)/.magisk/mirror/sepolicy.rules
  if [ -e $active_dir ]; then
    RULESDIR=$(readlink -f $active_dir)
  elif [ -d /data/unencrypted ] && ! grep ' /data ' /proc/mounts | grep -qE 'dm-|f2fs'; then
    RULESDIR=/data/unencrypted/magisk
  elif grep -q ' /cache ' /proc/mounts; then
    RULESDIR=/cache/magisk
  elif grep -q ' /metadata ' /proc/mounts; then
    RULESDIR=/metadata/magisk
  elif grep -q ' /persist ' /proc/mounts; then
    RULESDIR=/persist/magisk
  elif grep -q ' /mnt/vendor/persist ' /proc/mounts; then
    RULESDIR=/mnt/vendor/persist/magisk
  else
    return
  fi

  # Copy all enabled sepolicy.rule
  for r in /data/adb/modules*/*/sepolicy.rule; do
    [ -f "$r" ] || continue
    local MODDIR=${r%/*}
    [ -f $MODDIR/disable ] && continue
    [ -f $MODDIR/remove ] && continue
    local MODNAME=${MODDIR##*/}
    mkdir -p $RULESDIR/$MODNAME
    cp -f $r $RULESDIR/$MODNAME/sepolicy.rule
  done
}
```

首先删除重置所有已存在的规则文件夹，然后按一定的优先级将所有模块下的sepolicy.rule拷贝至对应的路径，这里与刚开头挂载自定义sepolicy路径对上了。

> 另外在模块被启用时也会执行该方法：
> com/topjohnwu/magisk/core/model/module/LocalModule.kt

Edxp中system_server进程hook不生效

查看日志，可以发现在OnNativeForkSystemServerPost中会打印：

> "skip injecting into android because sepolicy was not loaded properly"

提示sepolicy未正确加载，在edxp的sepolicy.rule中声明了两条se规则：

```
allow system_server system_server process execmem
allow system_server system_server memprotect mmap_zero
```

应该是用于申请在system_server的内存空间中进行mmap等内存操作，在部分机型下可能偶尔出现该错误，此时如果自己编译Edxp和Magisk可以在Magisk的rules.cpp中加上这两行se规则，避免因为插件的sepolicy未被正确加载的问题。