中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核分析与安全

9. 内核安全机制

授课教师：游伟 副教授

授课时间：周二14:00 – 15:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

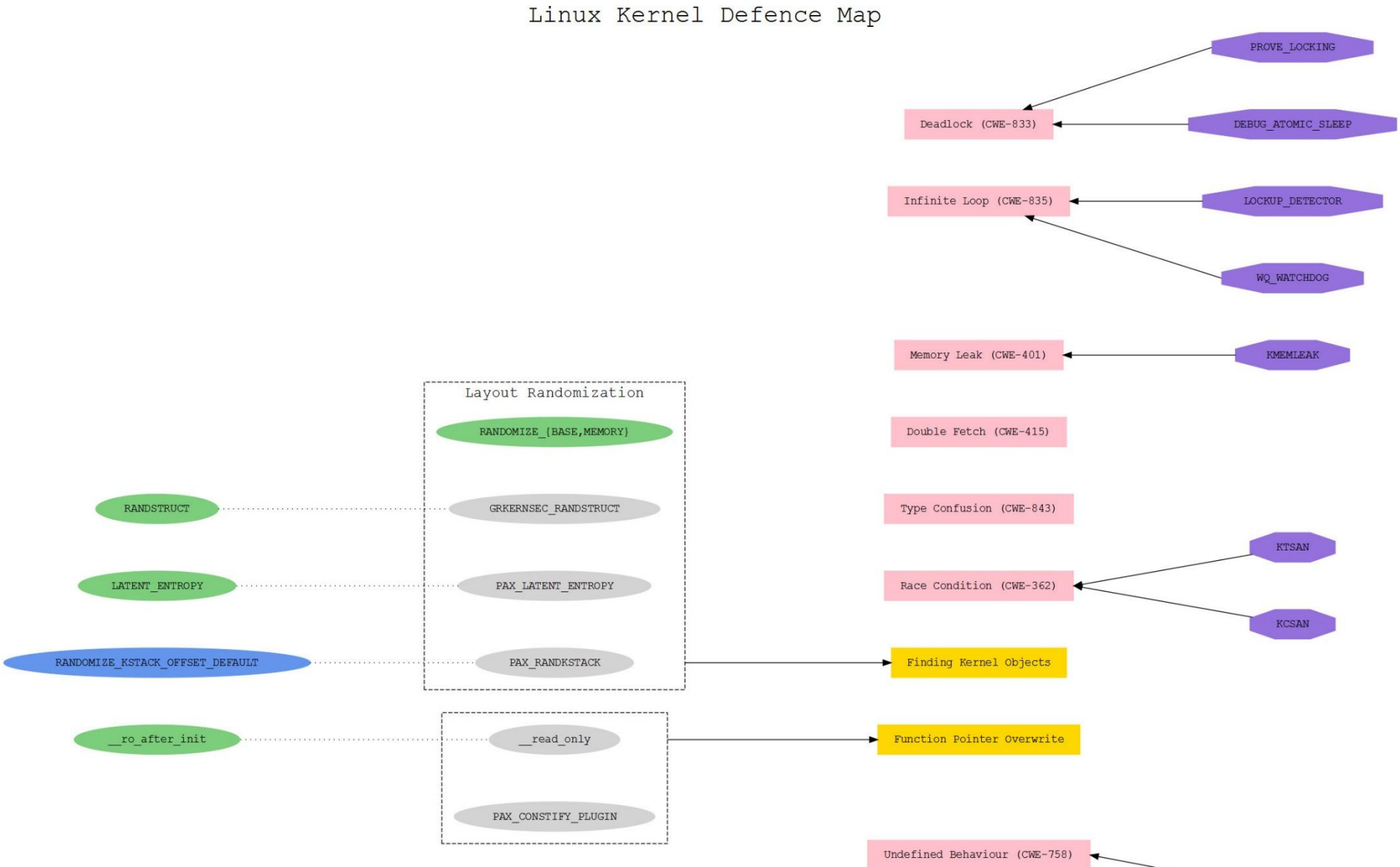
目录

1. 安全概览
2. 访问控制
3. 防御机制

9.1安全概览

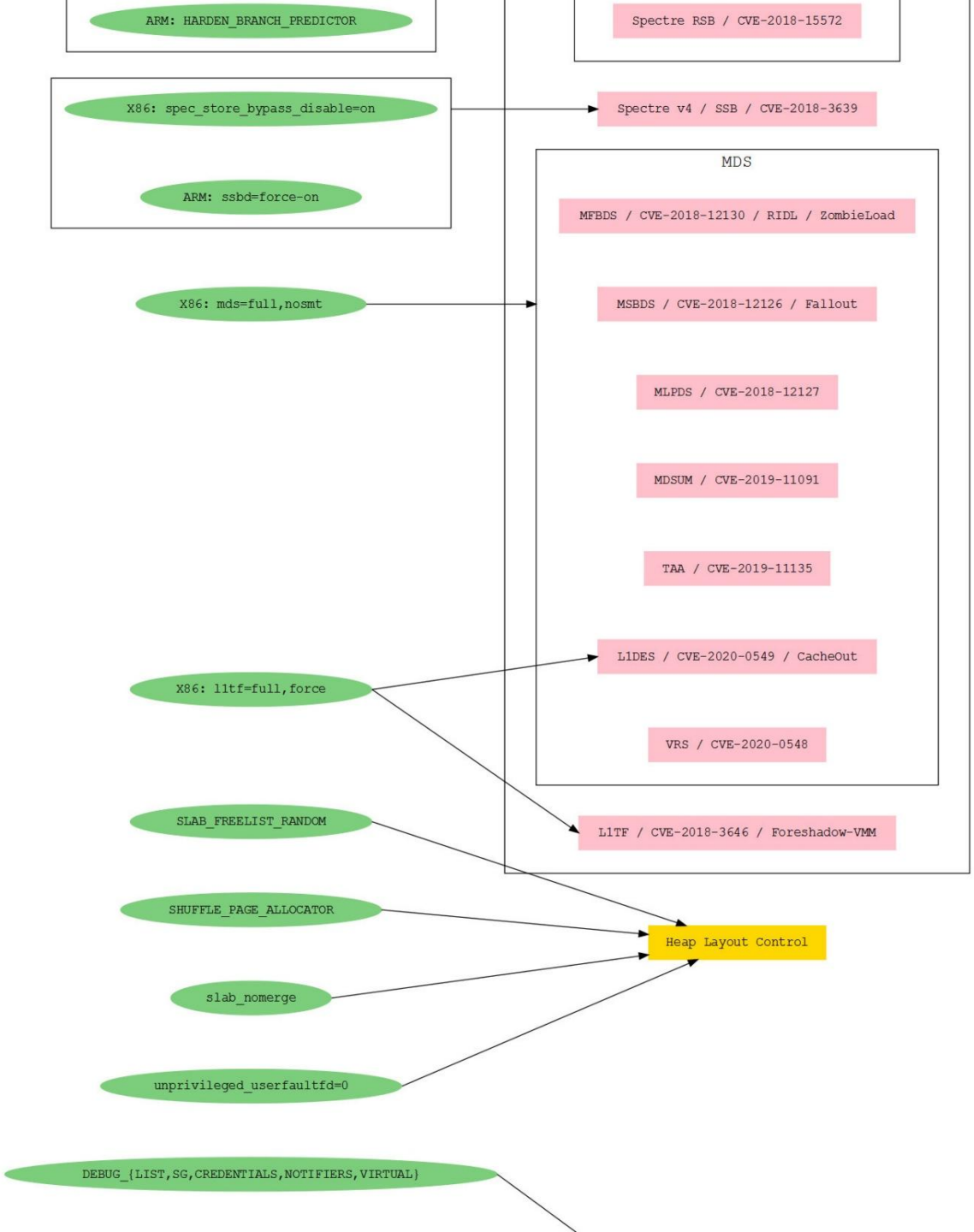
- 随机化：RANDOMIZE_BASE、RANDSTRUCT、LATENT_ENTROPY...
- 放篡改：__ro_after_init...
- 放整数溢出：REFCOUNT_FULL ...
- 保护模块：MODULE_SIG*、SECURITY_LOADPIN ...
- 读写保护：STRICT_{KERNEL,MODULE}_RWX、smep/PXN 与 smap/PAN ...
- 堆布局保护：SLAB_FREELIST_RANDOM、SLAB_FREELIST_HARDENED、SHUFFLE_PAGE_ALLOCATOR、slab_nomerge、unprivileged_userfaultfd ...
- 越界保护：STACKPROTECTOR、FORTIFY_SOURCE、slub_debug、HARDENED_USERCOPY...
- 栈保护：THREAD_INFO_IN_TASK、VMAP_STACK、SCHED_STACK_END_CHECK、PAX_MEMORY_STACKLEAK ...
- 遗留数据清理：init_on_free/init_on_alloc、PAGE_POISONING、PAX_MEMORY_SANITIZE、INIT_STACK_ALL、STRUCTLEAK_BYREF_ALL ...
- 具体利用防御： KPTI、kptr_restrict、STATIC_USERMODEHELPER ...

9.1安全概览


Linux Kernel Defence Map
PROVE_LOCKING
Deadlock (CWE-833)
DEBUG_ATOMIC_SLEEP
Infinite Loop (CWE-835)
LOCKUP_DETECTOR
WQ_WATCHDOG
Memory Leak (CWE-401)
KMEMLEAK
Layout Randomization
RANDOMIZE_{BASE,MEMORY}
Double Fetch (CWE-415)
RANDSTRUCT
GRKERNSEC_RANDSTRUCT
Type Confusion (CWE-843)
LATENT_ENTROPY
PAX_LATENT_ENTROPY
KTSAN
Race Condition (CWE-362)
KCSAN
RANDOMIZE_KSTACK_OFFSET_DEFAULT
PAX_RANDKSTACK
Finding Kernel Objects
__ro_after_init
__read_only
Function Pointer Overwrite
PAX_CONSTIFY_PLUGIN
Undefined Behaviour (CWE-758)

PAX_SIZE_OVERFLOW
UBSAN
Int Overflow/Underflow (CWE-190/191)
REFCOUNT_FULL
PAX_REFCOUNT
TIF_FSCHECK flag
Unbalanced set_fs()
bpf_jit_harden
JIT Abuse
GRKERNSEC_JIT_HARDEN
MODULE_SIG*
SECURITY_LOADPIN
LDISC_AUTOLOAD is not set
Bad Module Loading
request_module_cap()
GRKERNSEC_MODHARDEN
STRICT_{KERNEL,MODULE}_RWX
DEBUG_WX
WX Area Abuse
ARM: RODATA_FULL_DEFAULT_ENABLED
PAX_KERNEXEC
Control Flow Hijack Techniques

XPFO
ret2dir
ret2dir + ROP
SMEP/PXN
PAX_RAP
ROP
Clang CFI
ret2usr
ARM: CPU_SW_DOMAIN_PAN
ret2usr + ROP
SMAP/PAN
ARM: ARM64_SW_TTBR0_PAN
Userspace Data Access
PAX_UDEREF
NULL Pointer Dereference (CWE-476)
DEFAULT_MMAP_MIN_ADDR=65536
Transient Execution Vulnerabilities
(Covert Channels, CWE-514)
X86: pti=on (PAGE_TABLE_ISOLATION)
Spectre v3 / RDCL / CVE-2017-5754 / Meltdown
ARM: kpti=on (UNMAP_KERNEL_AT_EL0)
Spectre v3a / RSRE / CVE-2018-3640
X86: MICROCODE
Spectre v1 / BCB / CVE-2017-5753
Respectre
Spectre v1.1 / BCBS / CVE-2018-3693
Manual *_nospec() usage
Spectre v1.2 / RPB
X86: spectre_v2=on (RETPOLINE)
Spectre v2 / BTI / CVE-2017-5715


ARM: HARDEN_BRANCH_PREDICTOR
Spectre RSB / CVE-2018-15572
X86: spec_store_bypass_disable=on
Spectre v4 / SSB / CVE-2018-3639
ARM: ssbd=force-on
MDS
MFBDS / CVE-2018-12130 / RIDL / ZombieLoad
X86: mds=full,nosmt
MSBDS / CVE-2018-12126 / Fallout
MLPDS / CVE-2018-12127
MDSUM / CVE-2019-11091
TAA / CVE-2019-11135
L1DES / CVE-2020-0549 / CacheOut
X86: l1tf=full,force
VRS / CVE-2020-0548
SLAB_FREELIST_RANDOM
L1TF / CVE-2018-3646 / Foreshadow-VMM
SHUFFLE_PAGE_ALLOCATOR
Heap Layout Control
slab_nomerge
unprivileged_userfaultfd=0
DEBUG_{LIST,SG,CREDENTIALS,NOTIFIERS,VIRTUAL}

BUG_ON_DATA_CORRUPTION
PROTECTABLE_MEMORY
Metadata Corruption
STATIC_USERMODEHELPER
by default in grsecurity
LKRG
Changing Kernel Image
LOCKDOWN_LSM
ARM: STACKPROTECTOR_PER_TASK
STACKPROTECTOR
FORTIFY_SOURCE
Stack Buffer Overflow (CWE-121)
slub_debug=Z
Heap Out-of-Bounds Access (CWE-122,119)
HARDENED_USERCOPY
PAX_USERCOPY
THREAD_INFO_IN_TASK
GRKERNSEC_KSTACKOVERFLOW
Stack Depth Overflow (CWE-674,?)
KASAN
VMAP_STACK
SCHED_STACK_END_CHECK
STACKLEAK
PAX_MEMORY_STACKLEAK
Use-After-Free (CWE-416)
init_on_free
Heap free() Poisoning
slub_debug=P
init_on_alloc

PAGE_POISONING
PAX_MEMORY_SANITIZE
X86: X86_INTEL_UMIP
ARM: HARDEN_EL2_VECTORS
kptr_restrict
GRKERNSEC_HIDESYM
SECURITY_DMESG_RESTRICT
GRKERNSEC_DMESG
INIT_STACK_ALL
STRUCTLEAK_BYREF_ALL
PAX_MEMORY_STRUCTLEAK
Uninitialized Vars (CWE-457)
KMSAN
Info Exposure (CWE-200)
slub_debug=F
Double Free (CWE-415)
SLAB_FREELIST_HARDENED
Allocator Data Corruption
by default in grsecurity
Legend:
Mainline Defences
Out-of-tree Defences
Commercial Defences
HW Defences
Generic Defence Techniques
Bug Detection
Vulnerabilities
Exploitation Techniques

9.2 访问控制

- 核心需求
  - 资源隔离：确保恶意进程无法越权访问敏感数据（如/etc/shadow）
  - 权限最小化：即使进程被攻陷，其能执行的操作也被严格限制
  - 防御深度：通过多层级安全检查阻挡复杂攻击链
- 传统访问控制方式DAC
  - 自主访问控制模型（DAC，Discretionary Access Control）是根据自主访问控制策略建立的一种模型，允许合法用户以用户或用户组的身份访问策略规定的客体，同时阻止非授权用户访问客体。拥有客体权限的用户，可以将该客体的权限分配给其他用户。
  - DAC权限滥用：

```
# 用户误操作导致安全漏洞
chmod 777 /var/log/auth.log  # 攻击者可篡改登录日志
```

  - Root权限失控：

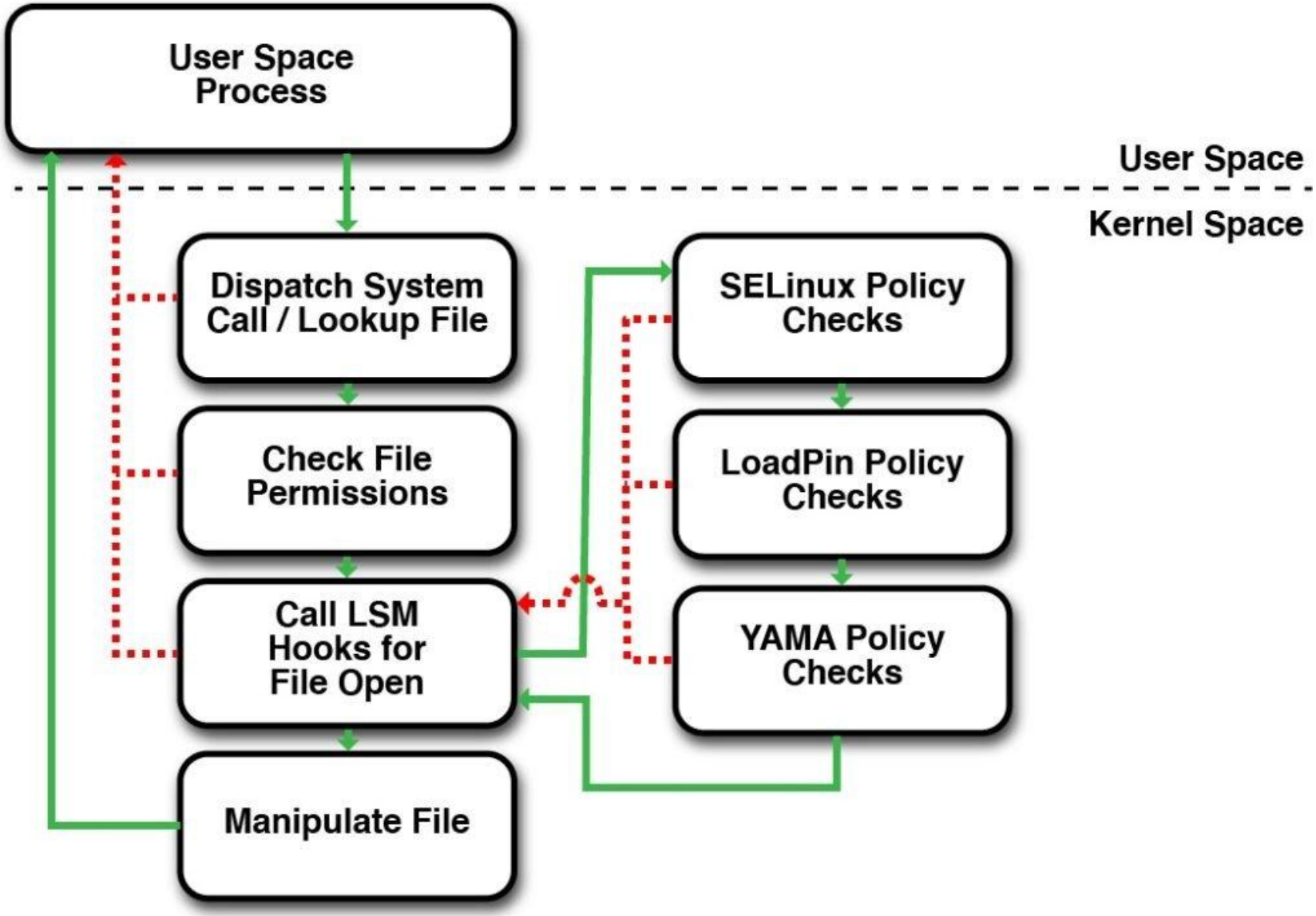
```
sudo vim /etc/passwd  # 普通用户提权后任意修改系统配置
```

9.2.1 访问控制-LSM

- Linux安全模块（Linux Security Modules, LSM）是Linux内核提供的一个动态安全框架，允许开发者通过可插拔的模块扩展内核的安全功能。它的核心目标是为操作系统提供细粒度的强制访问控制（MAC），而无需修改内核的核心代码。简单来说，LSM就像为操作系统安装了一套“安检系统”，所有关键操作（如文件访问、进程创建）都必须通过安全检查才能执行。

- 核心特性
  - 模块化设计：允许动态加载安全模块（如SELinux、AppArmor），而无需重新编译内核。
  - 灵活性：支持多种安全策略，适用于不同场景（服务器、嵌入式设备、个人电脑）。
  - 透明性：对用户和应用程序无感知，安全策略由内核强制执行。

9.2.1 访问控制-LSM


User Space Process
User Space
Kernel Space
Dispatch System Call / Lookup File
Check File Permissions
Call LSM Hooks for File Open
Manipulate File
SELinux Policy Checks
LoadPin Policy Checks
YAMA Policy Checks

9.2.1 访问控制-LSM组成

- 安全域（Security Fields）
  - 作用：为内核对象（文件、进程、网络端口等）附加安全属性标签。
  - 实现：在内核数据结构中增加void *security指针字段，存储安全模块的私有数据。

```
// 示例：进程控制块中的安全域
struct task_struct {
    // ...
    void *security;  // 指向SELinux的进程标签或AppArmor策略
};
```

- 标签示例：
  - SELinux：system_u:system_r:httpd_t:s0（表示一个Apache进程的安全上下文）。
  - AppArmor：/usr/sbin/nginx (enforce)（表示Nginx进程的强制模式策略）。

9.2.1 访问控制-LSM组成

- 钩子函数（Hook Functions）
  - 作用：在内核执行关键操作前插入检查点，触发安全模块的决策逻辑。
  - 钩子分类（覆盖200+个关键操作）：

钩子类型	示例场景	内核调用点
文件操作	打开文件、修改权限	file_open ， inode_permission
进程控制	创建新进程、发送信号	task_create ， task_kill
网络通信	绑定端口、建立连接	socket_bind ， socket_connect
设备访问	读写硬件设备	dev_access ， ioctl
能力检查	验证进程的root权限	capable

- 钩子调用流程：用户发起系统调用 → 内核执行操作 → 调用LSM钩子函数 → 安全模块决策 → 允许/拒绝

9.2.1 访问控制-LSM组成

- 安全模块管理
  - 模块注册：安全模块通过struct security_operations结构体向LSM注册回调函数。

```
// SELinux模块注册示例
static struct security_operations selinux_ops = {
    .file_open = selinux_file_open,       // 文件打开时的检查
    .task_create = selinux_task_create,   // 进程创建时的检查
    // ...注册其他200+个钩子函数
};
```

- 加载规则：
  - 单一主模块：同一时间只能激活一个主LSM模块（如SELinux或AppArmor）。
  - 辅助模块：可同时加载多个辅助模块（如Yama用于限制进程调试）。

9.2.1 访问控制-LSM启用

- 1. 安全模块的加载
  - 启动时加载：通过内核启动参数指定主模块。
  - 运行时验证：内核初始化时检查模块合法性，防止恶意模块注入。

```
# 在GRUB配置中指定使用AppArmor
GRUB_CMDLINE_LINUX="security=apparmor"
```

- 2. 钩子函数的插入
  - 代码注入点：在Linux内核源码的关键路径插入security_前缀的函数调用。
  - 示例：在打开文件前检查权限

```
// fs/open.c中文件打开流程
long do_sys_open(int dfd, const char __user *filename, int flags,
umode_t mode) {
    struct file *file;
    file = filp_open(filename, flags, mode);
    // 插入LSM钩子
    if (security_file_open(file) < 0) {
      return -EPERM;  // 安全模块拒绝访问
    }
    // ...继续执行打开操作
}
```

9.2.1 访问控制-LSM启用

- 3.安全模块的决策流程
  - 触发钩子：当内核执行到钩子点（如打开文件）时，调用注册的钩子函数。
  - 策略查询：安全模块根据自身规则库判断是否允许操作。
    - SELinux：查询策略库中的allow规则。
    - AppArmor：检查进程的配置文件路径权限。
  - 返回结果：
    - 允许：返回0，内核继续执行操作。
    - 拒绝：返回-EPERM，内核终止操作并记录日志。
- 实际案例：SELinux拦截攻击
  - 攻击场景：Apache进程（标签httpd_t）被入侵，试图读取/etc/shadow（标签shadow_t）。
  - 钩子触发：file_open钩子调用SELinux的selinux_file_open函数。
  - 策略检查：SELinux发现策略库中无httpd_t访问shadow_t的规则。
  - 结果：返回-EPERM，内核拒绝访问并生成审计日志：

9.2.2 访问控制-DAC vs. MAC

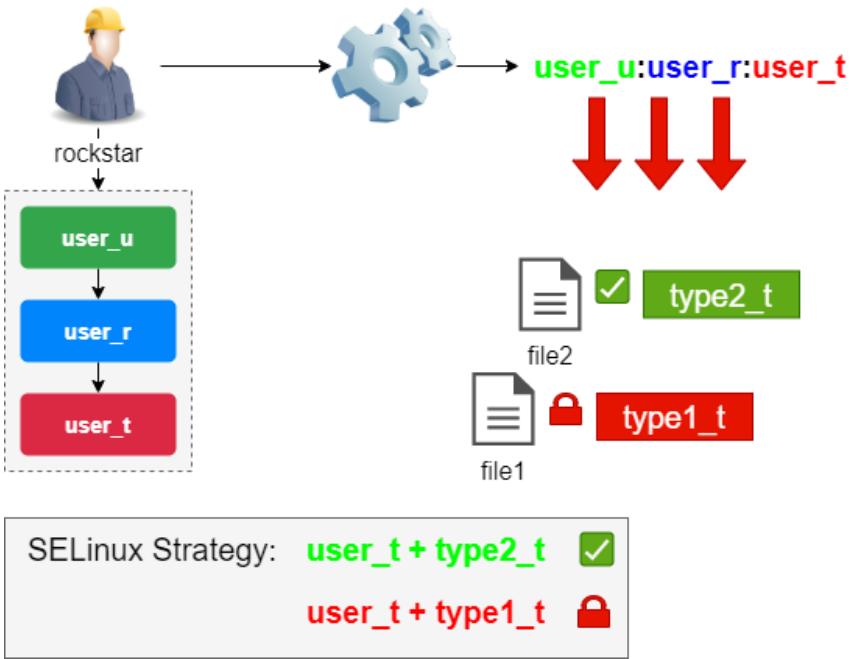
- DAC（自主访问控制，Discretionary Access Control）
  - 定义：资源的拥有者（如文件创建者）可自主决定其他用户或进程的访问权限。
  - 核心思想："我的资源我作主"，权限管理分散化。
  - 典型实现：
    - Unix/Linux文件权限（rwx）
    - Windows NTFS权限（用户可设置文件夹共享权限）
- MAC（强制访问控制，Mandatory Access Control）
  - 定义：由系统管理员集中定义安全策略，用户和进程无法绕过或修改。
  - 核心思想："安全高于一切"，权限管理集中化。
  - 典型实现：
    - SELinux（基于标签的MAC）
    - AppArmor（基于路径的MAC）
    - 军事多级安全系统（如Bell-LaPadula模型）

9.2.2 访问控制-DAC vs. MAC

对比维度	DAC	MAC
控制权	资源所有者（用户）	系统管理员
灵活性	高（用户自由授权）	低（策略集中管理）
安全性	低（依赖用户操作）	高（强制策略防御）
配置复杂度	简单（`chmod`命令）	复杂（需策略文件）
典型场景	个人电脑、开发环境	服务器、军事系统、金融核心系统
防御能力	无法阻止恶意提权	可限制root权限滥用

9.2.3 访问控制-主流安全模块-SELinux

- 背景与核心思想
  - 开发者：由美国国家安全局（NSA）主导开发，2003年并入Linux内核主线。
  - 核心目标：通过强制访问控制（MAC）实现最小权限原则，即使root用户也无法绕过安全策略。
  - 设计哲学：基于多级安全模型（MLS）和类型强制（TE），通过标签系统精细控制资源访问。



9.2.3 访问控制-主流安全模块-SELinux

- 安全上下文：为所有资源（文件、进程、端口等）分配唯一的安全标签。同LSM中所介绍（LSM的实现就是SELinux）。
- 多模式运行：
  - Enforcing模式：强制执行策略，拒绝违规操作。
  - Permissive模式：仅记录违规行为，不阻止操作（用于调试）。
  - Disabled模式：完全禁用SELinux。
- 策略管理工具
  - semanage：管理文件、端口、用户等上下文标签。
  - audit2allow：根据日志生成自定义策略。

9.2.3 访问控制-主流安全模块-SELinux

- 安装SELinux控制包：`sudo apt install policycoreutils selinux-utils selinux-basics`
- 查看SELinux安装情况：`sudo selinux-activate`

```
*** Warning -- SELinux default policy relabel is required.
*** Relabeling could take a very long time, depending on file
*** system size and speed of hard drives.
Relabeling /
29.7%_
```

- 查看文件标签

```
❯ ls -Z /etc/passwd
system_u:object_r:etc_t:s0 /etc/passwd
```

  - 用户（User）：system_u，系统级文件。
  - 角色（Role）：object_r，普通文件或目录（被动资源）。
  - 类型（Type）：etc_t，/etc 目录下的系统配置文件。
  - 级别（Level）：s0，默认级别（无特殊隔离）。
- 查看sshd是否能访问：

```
❯ sesearch -A -s sshd_t -t etc_t -c file -p read
allow nsswitch_domain etc_t:file { getattr ioctl lock open read };
allow sshd_t etc_t:file { getattr ioctl lock open read };
```

9.2.3 访问控制-主流安全模块-APPArmor

- 背景与核心思想
  - 开发者：由Immunix公司开发，现由Canonical（Ubuntu）和SUSE维护。
  - 核心目标：通过路径（而非标签）定义访问规则，简化MAC配置。
  - 设计哲学：以应用程序为中心，通过配置文件限制其行为。
- 技术特点
  - 基于路径的规则
  -

```
# /etc/apparmor.d/usr.sbin.nginx
/usr/sbin/nginx {
  # 文件访问
  /var/www/html/** r,    # 允许读取网站文件
  /etc/nginx/*.conf r,   # 允许读取配置文件
  deny /etc/shadow r,    # 禁止读取密码文件

  # 网络权限
  network inet tcp,      # 允许IPv4 TCP通信

  # 能力限制
  deny capability sys_admin,  # 禁止特权操作
}
```

  - 学习模式（Complain Mode）
    - 自动生成规则：记录应用程序的正常行为并生成白名单。

9.2.3 访问控制-主流安全模块-APPArmor

- 实际应用案例
  - 场景：防止MySQL服务被入侵后读取无关文件。

```
/usr/sbin/mysqld {
  /var/lib/mysql/** rwk,      # 仅允许访问数据库目录
  deny /etc/*.conf r,         # 禁止读取其他配置
}
```

  - 拦截效果：若攻击者尝试读取/etc/passwd，AppArmor直接拒绝并记录日志。

9.2.3 访问控制-主流安全模块-Grsecurity

- 背景与核心思想
  - 开发者：最初由PaX Team开发，后由Grsecurity团队维护。
  - 核心目标：通过内核补丁提供主动防御，而非依赖策略规则。
  - 设计哲学：以内存保护和进程限制为核心，防范未知漏洞。
- 内存安全机制
  - PAX_ASLR：地址空间随机化，增加漏洞利用难度。
  - PAX_NOEXEC：禁止执行栈和堆内存中的数据。
  - PAX_MEMORY_STACKLEAK：自动清除内核栈残留数据。
- 进程隔离
  - RBAC（基于角色的访问控制），限制进程的fork、exec等系统调用。
  - Chroot加固：防止容器逃逸的增强版chroot。
- 日志与审计：实时告警，检测到异常行为时立即生成日志。

9.2.3 访问控制-主流安全模块

维度	SELinux	AppArmor	Grsecurity
控制粒度	精细（标签系统）	中（路径规则）	粗（进程/内存保护）
配置复杂度	高（需策略语言）	低（图形化工具）	中（内核参数调整）
防御范围	权限控制	权限控制	内存安全 + 权限控制
适用场景	服务器、多级安全系统	桌面、Web服务器	嵌入式设备、高安全服务器
维护方式	策略文件更新	配置文件更新	内核补丁更新

9.3 防御机制-RANDSTRUCT

- 功能：在编译时随机化结构体字段的排列顺序，破坏攻击者对内存布局的预判，缓解缓冲区溢出攻击。
- 原理
  - 结构体内存布局：默认情况下，结构体的字段按声明顺序连续存放。
  - 随机化实现：通过GCC插件，在编译时根据随机种子（-frandom-seed）重排字段。
  - 攻击失效：攻击者无法确定目标字段的偏移量，难以构造有效载荷。
- 启用方式：编译内核时添加CONFIG_GCC_PLUGIN_RANDSTRUCT=y。

```
// 原结构体
struct vuln_struct {
    int id;
    char name[32];
    void *ptr;
};
```

```
// 编译后可能的内存布局
struct vuln_struct {
    void *ptr;
    int id;
    char name[32];
};
```

9.3 防御机制-RANDOMIZE_BASE

- 功能：通过内核地址空间布局随机化（KASLR），随机化内核关键区域的基址，增加漏洞利用难度。
- 子功能
  - RANDOMIZE_BASE：每次启动随机化内核镜像的虚拟地址。
  - RANDOMIZE_MEMORY：随机化线性映射区、vmalloc区、vmemmap区的基址。
- 启用方式：
  - 内核配置：CONFIG_RANDOMIZE_BASE=y，CONFIG_RANDOMIZE_MEMORY=y。
  - 启动参数：确保cmdline中未设置nokaslr。

```
❯ sudo cat /proc/kallsyms | grep _text | head -n 1
ffffffff8d400000 T _text
```

ffffffff81000000

9.3 防御机制-LATENT_ENTROPY

- 功能：向内核熵池注入随机性，提升加密密钥的不可预测性。
- 熵池作用：为/dev/random提供高质量随机种子。
- 实现方式：在标记为__latent_entropy的函数中混入随机值。
- 启动阶段防护：缓解早期熵不足导致的弱密钥问题。

9.3 防御机制-__ro_after_init

- 功能：将初始化后不再修改的变量标记为只读，防止运行时篡改。
- 编译属性：__ro_after_init将变量放入.data..ro_after_init段。
- 内存保护：初始化完成后，该段内存权限设为只读（RO）。
- 防护场景
  - 函数指针保护：防止攻击者通过溢出修改操作函数表（如sys_call_table）。
  - 配置启用：默认开启，无需额外配置。

9.3 防御机制-REFCOUNT_FULL

- 功能：通过引用计数溢出检测，彻底防御Use-After-Free（UAF）漏洞。
- 传统原子操作问题：atomic_t可能因溢出从INT_MAX回绕到0。
- REFCOUNT改进：若引用计数超过INT_MAX或低于0，将其饱和至INT_MIN/2。

```
void refcount_inc(refcount_t *r) {
    if (r->refs.counter >= 0)
        atomic_inc(&r->refs);
    else
        WARN_ONCE(1, "refcount overflow");
}
```

9.3 防御机制-MODULE_SIG*

- 功能：强制内核模块签名验证，阻止未授权模块加载。
- 签名生成：开发者使用私钥对模块签名。
- 内核配置：
  - CONFIG_MODULE_SIG=y：启用签名验证。
  - CONFIG_MODULE_SIG_FORCE=y：强制所有模块必须签名。
- 防护场景：阻止Rootkit通过insmod注入恶意模块。

9.3 防御机制-SECURITY_LOADPIN

- 功能：强制所有内核加载的文件（模块、固件、安全策略等）必须来自同一可信来源，防止恶意文件注入。
- 信任源验证：通过内核的dm-verity或fsverity机制验证文件完整性。
- 拦截点：在加载模块（insmod）、读取固件（request_firmware）、kexec执行镜像等关键操作时触发检查。
- 启用方式：CONFIG_SECURITY_LOADPIN=y。
- 防护场景
  - 供应链攻击：阻止攻击者替换合法模块为恶意版本。
  - 物理攻击：防止通过USB设备注入未验证的固件。

9.3 防御机制-STRICT_{KERNEL,MODULE}_RWX

- 功能：严格分离内核内存的读、写、执行权限，防御代码注入和内存篡改攻击。
- 内存段保护：
  - .text段：只读 + 可执行（防止代码篡改）。
  - .rodata段：只读（保护常量数据）。
  - .data和.bss段：不可执行（防止数据段代码执行）。
- 实现方式：利用CPU的NX（No-eXecute）位实现硬件级保护。
- 启用方式：CONFIG_STRICT_KERNEL_RWX=y（内核代码段）、CONFIG_STRICT_MODULE_RWX=y （模块代码段）
- 防护场景
  - 堆喷射攻击：阻止攻击者在堆中注入可执行代码。
  - ROP攻击：增加构造利用链的难度。

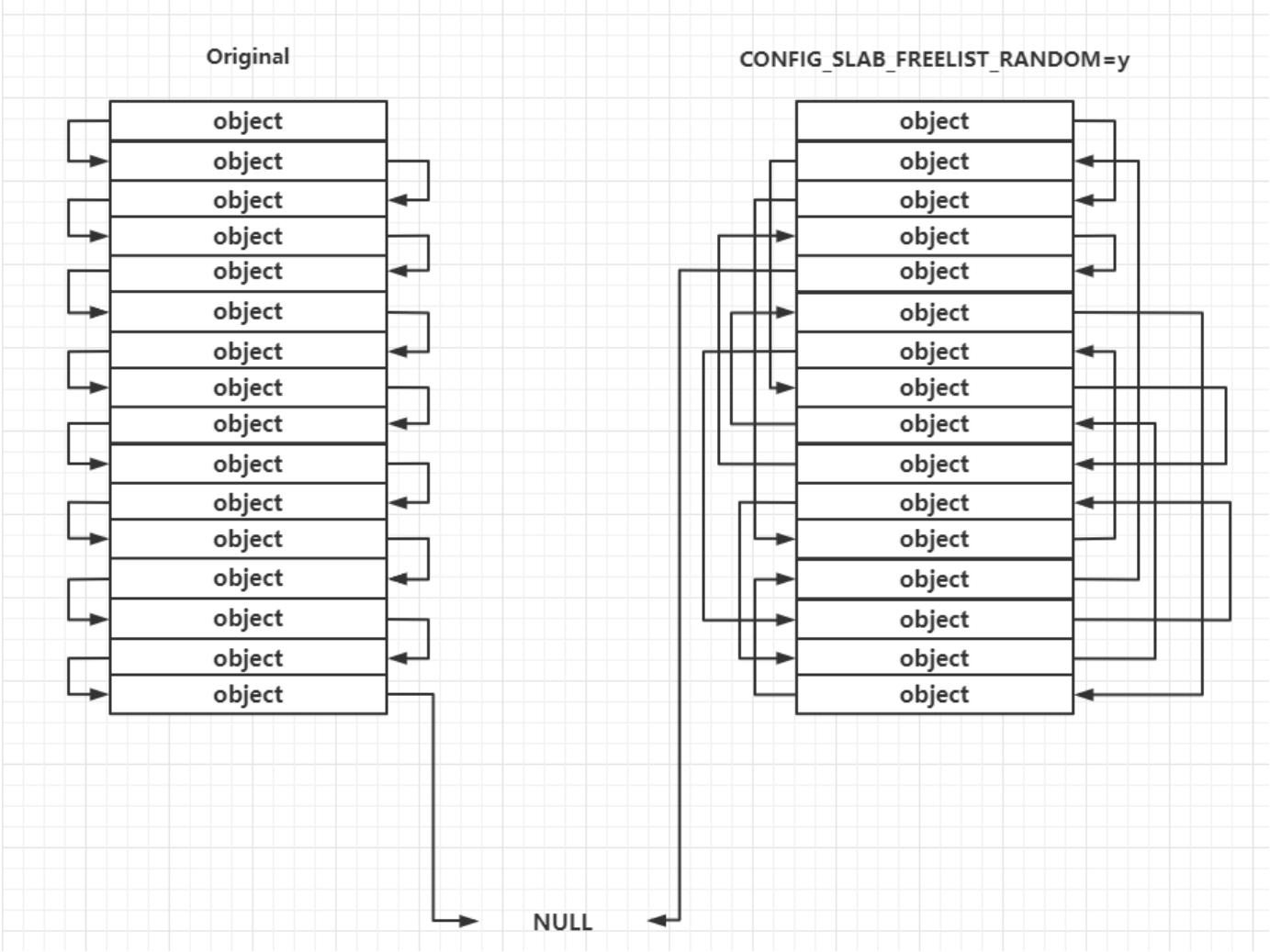
9.3 防御机制-smep/PXN 与 smap/PAN

- 功能：阻止内核态执行用户空间代码（SMEP/PXN）或访问用户空间数据（SMAP/PAN），防御提权攻击。
- SMEP（Supervisor Mode Execution Protection）：x86中禁止内核执行用户空间代码。
- PXN（Privileged Execute Never）：ARM中的等效功能。
- SMAP（Supervisor Mode Access Protection）：x86中禁止内核访问用户空间数据。
- PAN（Privileged Access Never）：ARM中的等效功能。
- 启用方式：默认启用，需要CPU支持。

9.3 防御机制-SLAB_FREELIST_RANDOM

- 功能：随机化SLAB分配器的空闲列表顺序，防御堆溢出攻击。
- 堆溢出风险：攻击者预测空闲对象布局，篡改相邻内存。
- 随机化实现：初始化时打乱空闲对象指针顺序。
- 配置：内核配置CONFIG_SLAB_FREELIST_RANDOM=y。
- 攻击难度提升：溢出后难以预测篡改目标，利用成功率显著降低。

9.3 防御机制-SLAB_FREELIST_RANDOM



9.3 防御机制-SLAB_FREELIST_HARDENED

- 功能：通过加密和随机化SLAB分配器的空闲链表指针，防御堆溢出和Use-After-Free攻击。
- 防护措施
  - 指针加密：空闲指针存储为ptr ^ random_secret，攻击者无法直接篡改。
  - 随机化插入：空闲对象插入链表的位置随机，破坏攻击者的布局预测。
- 启用方式：内核配置CONFIG_SLAB_FREELIST_HARDENED=y。
- 攻击防御：如CVE-2021-22555利用freelist篡改实现任意写。
- 防护效果：加密后的指针使攻击者无法构造有效目标地址。
- 性能影响：内存分配延迟增加约2%~5%。

9.3 防御机制-SHUFFLE_PAGE_ALLOCATOR

- 功能：随机化物理页分配顺序，增加内存布局的不可预测性。
- 内存布局泄露：攻击者通过已知布局定位敏感数据。
- 页分配随机化：打乱物理页分配顺序，破坏攻击前提。
- 配置：内核配置CONFIG_SHUFFLE_PAGE_ALLOCATOR=y。

9.3 防御机制-slab_nomerge

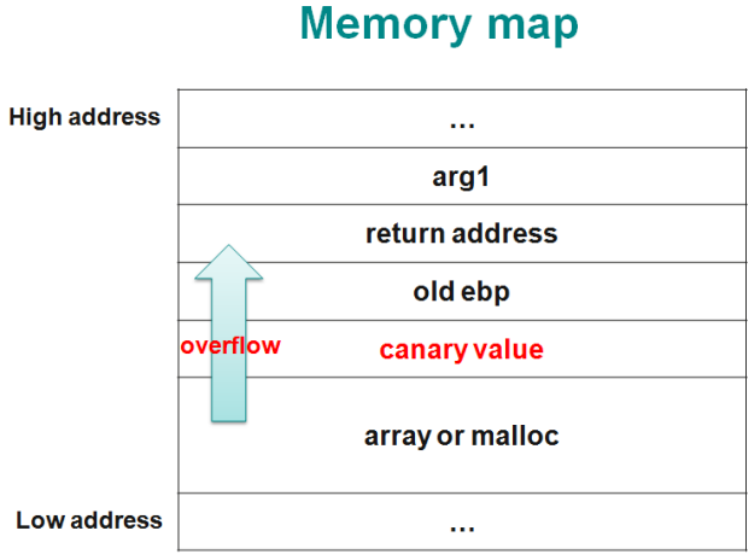
- 功能：禁止SLAB分配器合并相同大小的缓存，限制堆溢出影响范围。
- 合并风险：不同对象共享同一缓存，溢出可能篡改敏感数据。
- 禁用合并：保持各缓存独立，溢出仅影响同类对象。
- 配置：内核参数slab_nomerge。
- 优缺点
  - 优点：缩小溢出影响范围。
  - 缺点：内存碎片可能增加，需权衡安全与性能。

9.3 防御机制-unprivileged_userfaultfd

- 功能：限制非特权用户使用userfaultfd系统调用，防御利用缺页处理进行攻击。
- 背景：userfaultfd用于处理用户空间缺页异常，但可能被滥用于延迟页错误处理，构造竞争条件漏洞。
- 配置：禁用非特权访问：内核参数userfaultfd=0或sysctl vm.unprivileged_userfaultfd=0。
- 防御漏洞：利用userfaultfd延缓页错误处理，绕过权限检查。

9.3 防御机制-STACKPROTECTOR

- 功能：通过栈金丝雀（Canary）检测栈溢出攻击。
- Canary值：在栈帧中插入随机值，函数返回前验证其完整性。
- 防护效果：溢出覆盖返回地址时会破坏Canary，触发内核错误。
- 全局启用：CONFIG_STACKPROTECTOR=y。
  - ARM按任务启用：CONFIG_STACKPROTECTOR_PER_TASK=y（为每个任务生成独立Canary）。

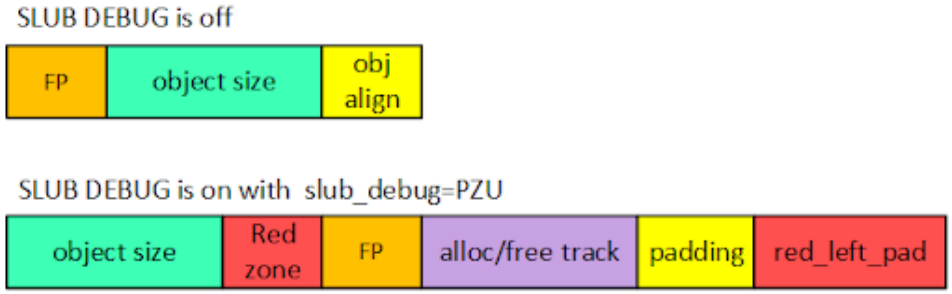


9.3 防御机制-FORTIFY_SOURCE

- 功能：在编译时对缓冲区操作函数（如memcpy、strcpy）进行边界检查，防御已知大小的缓冲区溢出攻击。
- 编译时检查：利用编译器的__builtin_object_size确定目标缓冲区大小，若检测到操作越界，触发编译警告或运行时错误。
- 局限性：仅能检查编译时已知大小的缓冲区（如静态数组），动态分配内存无法完全防护。
- 启用方式：编译时添加-D_FORTIFY_SOURCE=2。

9.3 防御机制-slub_debug

- 功能：检测SLUB内存分配器的内存损坏（如溢出、释放后使用），增强堆安全性。
- 参数选项
  - F（Free poisoning）：释放时填充0x6b（POISON_FREE）。
  - Z（Red zoning）：在对象两侧插入红色区域，检测溢出。
  - U（User tracking）：追踪分配/释放的调用栈。
- 启用方式：内核启动参数添加slub_debug=FZ。
- 性能影响：启用调试选项后，内存分配延迟增加约15%。



9.3 防御机制-HARDENED_USERCOPY

- 功能：在用户空间与内核空间之间拷贝数据时，验证地址合法性，防御越界访问。
- 检查类型
  - 堆对象：验证拷贝范围不超过分配大小。
  - 栈保护：禁止用户空间直接访问内核栈。
  - 代码段：阻止向只读段（如.text）写入。
- 启用方式：内核配置CONFIG_HARDENED_USERCOPY=y。

9.3 防御机制-THREAD_INFO_IN_TASK

- 功能：将线程信息（thread_info）从内核栈移至task_struct，减少栈溢出攻击面。
- 传统布局：thread_info位于内核栈底部，溢出可能篡改任务元数据。
- 新布局：thread_info作为task_struct的字段，与栈隔离。
- 配置：内核配置CONFIG_THREAD_INFO_IN_TASK=y。
- 性能影响：无显著开销，**默认启用于现代内核。**

9.3 防御机制-VMAP_STACK

- 功能：使用vmalloc分配内核栈，引入Guard Page检测栈溢出，减少内存碎片。
- 优点
  - Guard Page：栈溢出触碰未映射的Guard Page，触发缺页异常。
  - 物理非连续：允许栈分布在非连续物理内存，提升利用率。
- 启用方式：内核配置CONFIG_VMAP_STACK=y。
- 性能影响：栈访问延迟略增（约3%），因虚拟地址转换开销。

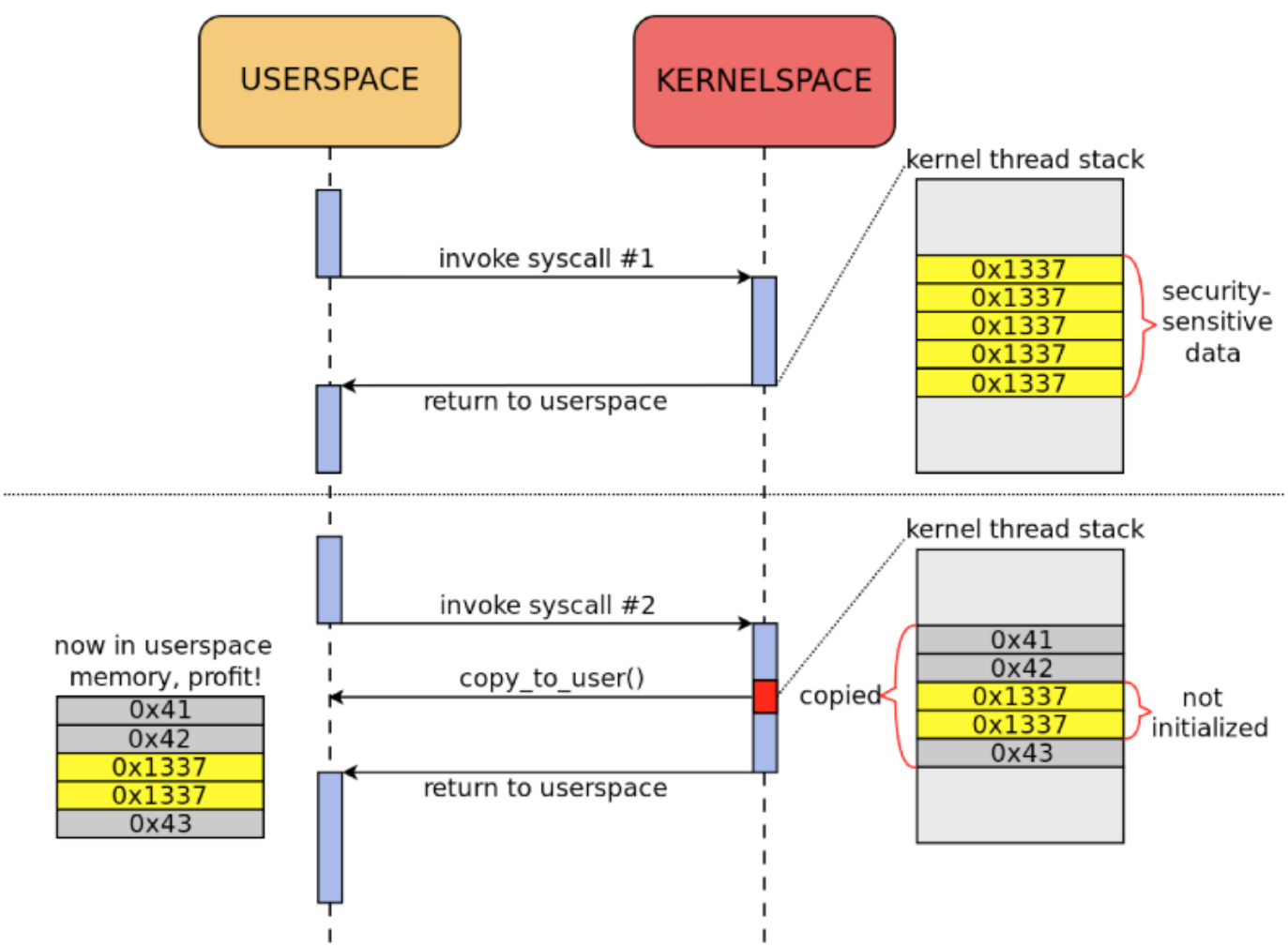
9.3 防御机制-SCHED_STACK_END_CHECK

- 功能：在调用调度器（schedule()）时检查内核栈是否溢出，防止数据损坏。
- 栈末尾标记：在栈底设置固定值（如0xDEADBEEF），调度前验证其完整性。
- 溢出响应：若标记被修改，触发内核panic。
- 配置：内核配置CONFIG_SCHED_STACK_END_CHECK=y。
- 检测漏洞：内核驱动栈溢出导致任务切换时崩溃，阻止后续不可预测行为。

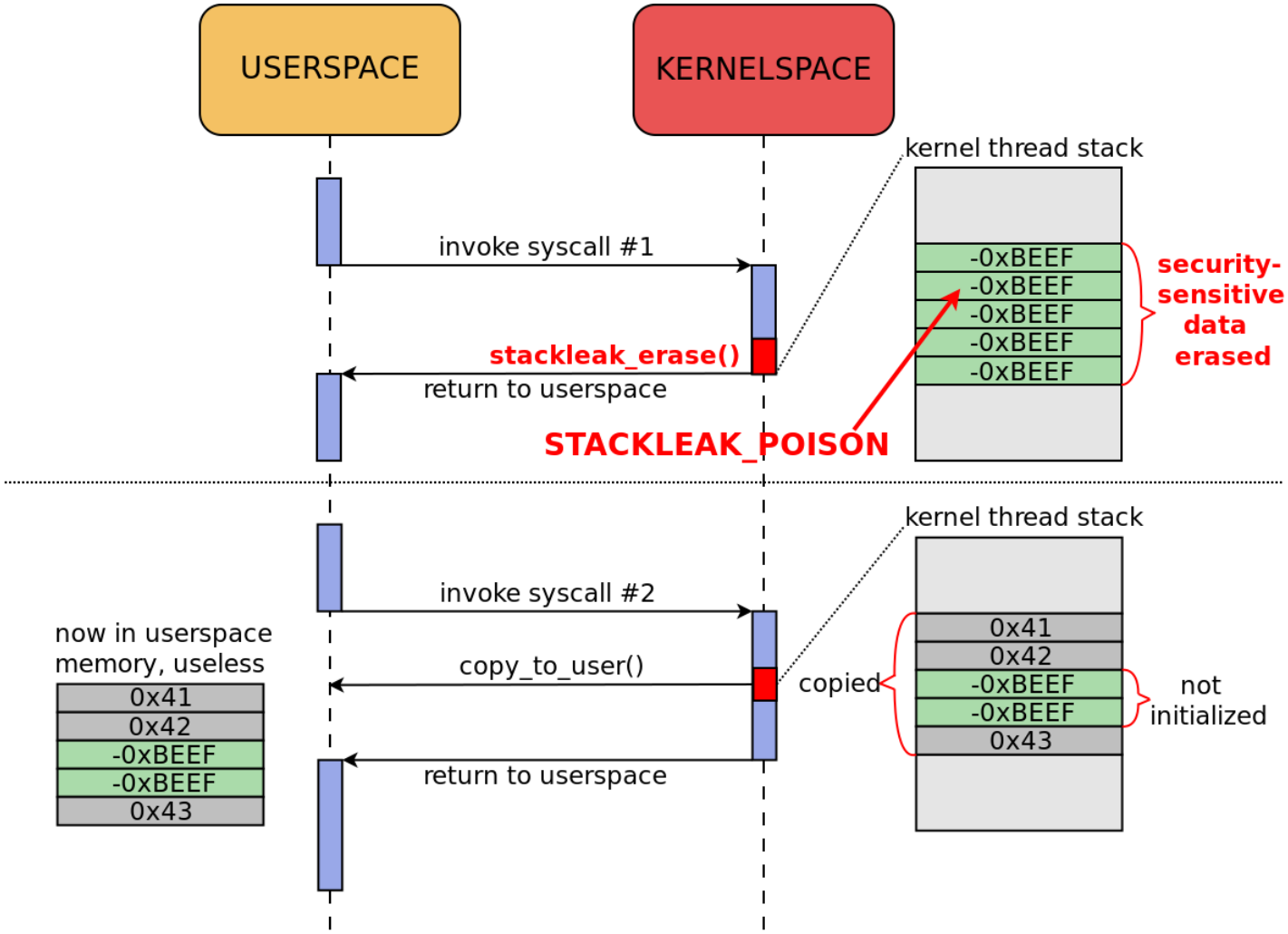
9.3 防御机制-PAX_MEMORY_STACKLEAK

- 功能：清除进程返回用户态时的内核栈残留数据，并检测栈溢出。
- 栈擦除：每次返回到用户态时，将内核栈填充为0x00。
- 溢出检查：定期扫描栈边界，检测异常模式。
- 配置：需应用Grsecurity补丁并启用CONFIG_PAX_MEMORY_STACKLEAK。
- 性能影响：上下文切换时间增加约5%。

9.3 防御机制-PAX_MEMORY_STACKLEAK



9.3 防御机制-PAX_MEMORY_STACKLEAK



9.3 防御机制-init_on_free/init_on_alloc

- 功能：在内存释放或分配时自动初始化，防止敏感数据残留。
- 配置：
  - 释放时清零：init_on_free=1。
  - 分配时清零：init_on_alloc=1。
- 防御类型：UAF漏洞重用堆喷后的页面。
- 性能影响：内存操作延迟增加约10%。

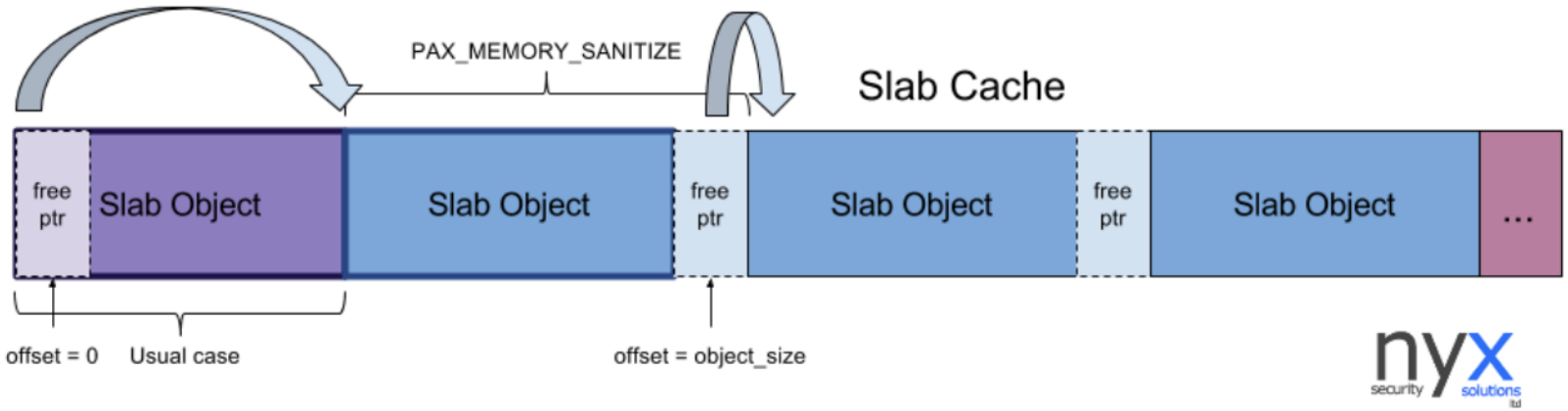
9.3 防御机制-PAGE_POISONING

- 功能：在释放内存时填充特定字节（如0xAA），分配时验证填充值，检测内存错误。
- 配置：内核参数page_poison=1
- 优点：高效检测Use-After-Free。
- 缺点：与init_on_free冲突，需二选一。

9.3 防御机制-PAX_MEMORY_SANITIZE

- 功能：擦除已释放内存的内容，防止敏感数据残留。
- 实现：在kfree时用随机数据覆盖内存。
- 配置：需Grsecurity补丁并启用

CONFIG_PAX_MEMORY_SANITIZE。



9.3 防御机制-INIT_STACK_ALL

- 功能：将未初始化的栈变量填充为0xAA，暴露未初始化使用错误。
- 配置：内核配置CONFIG_INIT_STACK_ALL=y。

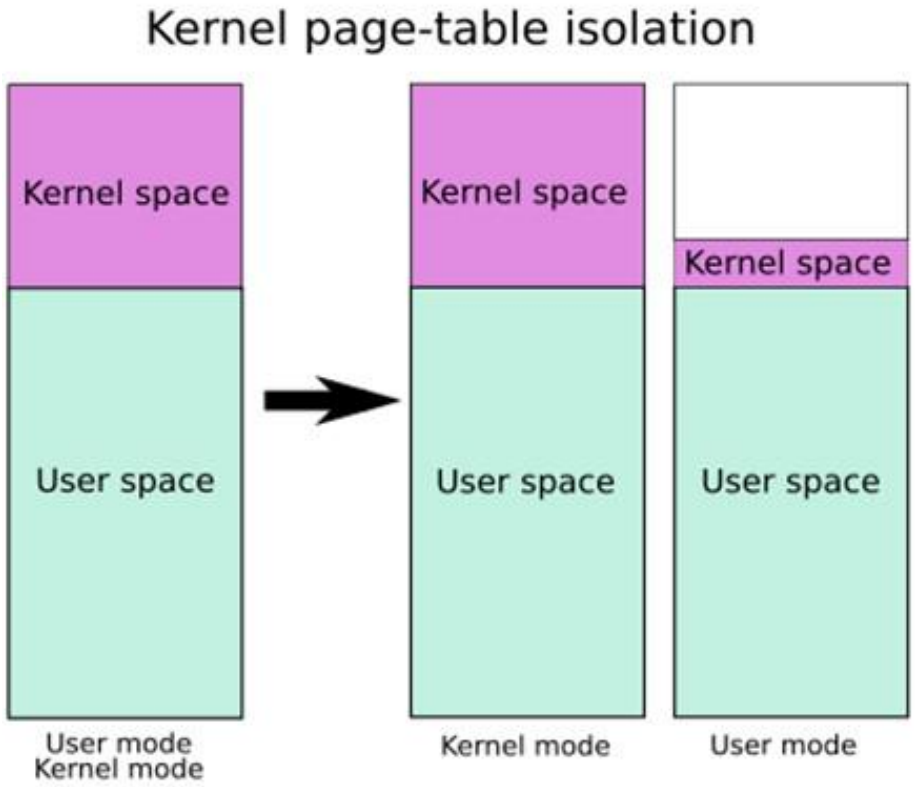
9.3 防御机制-STRUCTLEAK_BYREF_ALL

- 功能：强制初始化所有通过引用传递的栈变量，消除未初始化变量导致的漏洞（如信息泄漏或代码执行）。
- 未初始化风险：未初始化的栈变量可能残留之前的数据，攻击者可利用这些数据构造利用链。
- 编译时初始化：在内核编译阶段，对所有通过指针传递的栈变量进行强制初始化（填充0x00或特定模式）。
- 实现方式：通过GCC插件在编译时插入初始化代码。
- 启用方式：内核配置CONFIG_STRUCTLEAK_BYREF_ALL=y。

9.3 防御机制-KPTI

- 功能：将用户空间和内核空间的页表分离，防御Meltdown漏洞（CVE-2017-5754）。
  - Meltdown漏洞：利用CPU推测执行绕过权限检查，读取内核内存。
- 页表分离：用户进程的页表仅包含用户空间映射，访问内核地址触发缺页异常。
- 启用方式：内核配置CONFIG_PAGE_TABLE_ISOLATION=y。
- 性能损耗：系统调用/中断的上下文切换开销增加5%~30%（因CPU架构而异）。
- 防止数据泄露：用户进程无法通过侧信道读取内核敏感数据。

9.3 防御机制-KPTI


Kernel page-table isolation
Kernel space
User space
User mode
Kernel mode
Kernel space
User space
Kernel mode
Kernel space
User space
User mode

9.3 防御机制-kptr_restrict

- 功能：控制/proc/kallsyms中内核符号地址的可见性，增加逆向工程难度。
- 配置级别
  - 0：无限制，所有用户可见完整地址。
  - 1：仅Root用户可见地址，非特权用户看到全零。
  - 2：所有用户不可见地址（需内核配置CONFIG_KALLSYMS_ALL=y）。
- 防护场景：防止攻击者通过/proc/kallsyms获取内核函数地址，绕过KASLR。

9.3 防御机制-STATIC_USERMODEHELPER

- 功能：固定内核调用用户态帮助程序（如modprobe）的路径，防止恶意替换。
- 风险：攻击者替换/sbin/modprobe为恶意程序，提权加载模块。
- 防护：通过CONFIG_STATIC_USERMODEHELPER_PATH指定固定路径。
- 配置：
  - CONFIG_STATIC_USERMODEHELPER=y
  - CONFIG_STATIC_USERMODEHELPER_PATH="/usr/lib/systemd/system d-modules-load"

实践任务

- 添加一个系统调用，开启编译选项STACKPROTECTOR，调用该系统调用后触发栈金丝雀的报错，查看报错信息。
- 添加一个系统调用，并在启动时打开SMEP/SMAP，调用该系统调用后触发SMEP/SMAP的防御机制，查看报错信息。