
看雪·第六届安全开发者峰会
linux 内核漏洞检测与防御
蒋浩天/张海全 腾讯安全云鼎实验室
2022 SDC

自我介绍

蒋浩天：

擅长二进制安全、虚拟化安全、游戏安全等，喜欢研究一些底层方向，将安全能力下沉，开启安全对抗的上帝模式。

张海全：

主要从事GCC和Linux内核安全功能研发，Linux内核源码贡献者。 主要方向是漏洞挖掘，CFI防御，编译器sanitizer。致力于将漏洞扼杀在编译器中。

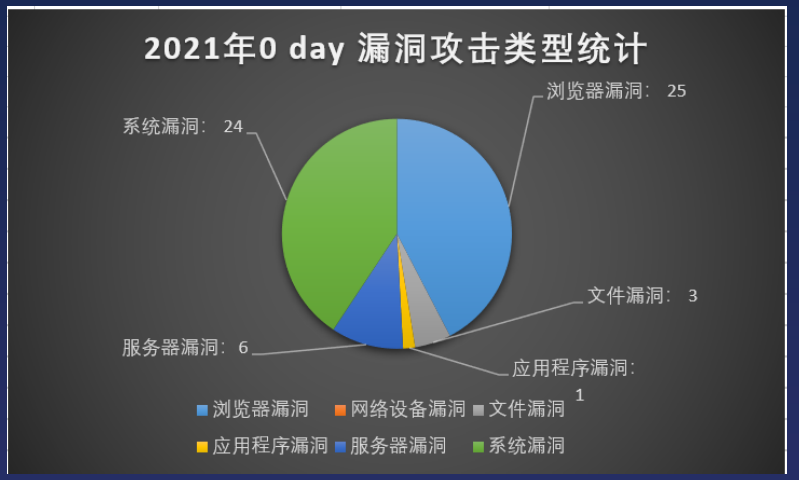
大纲

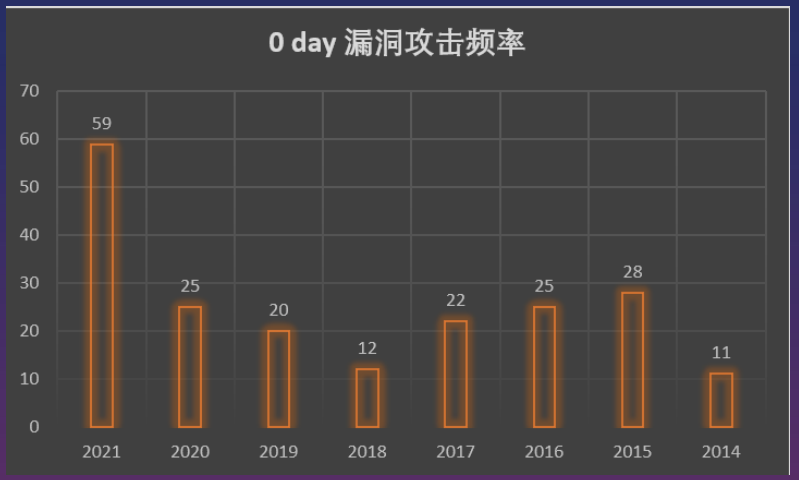
- 漏洞攻防现状
- 漏洞利用技术
- 传统漏洞防御技巧
- 基于intel pt的漏洞检测方案
- 基于编译器的漏洞防御方案
- 后续展望

漏洞攻防现状

主机漏洞防御价值

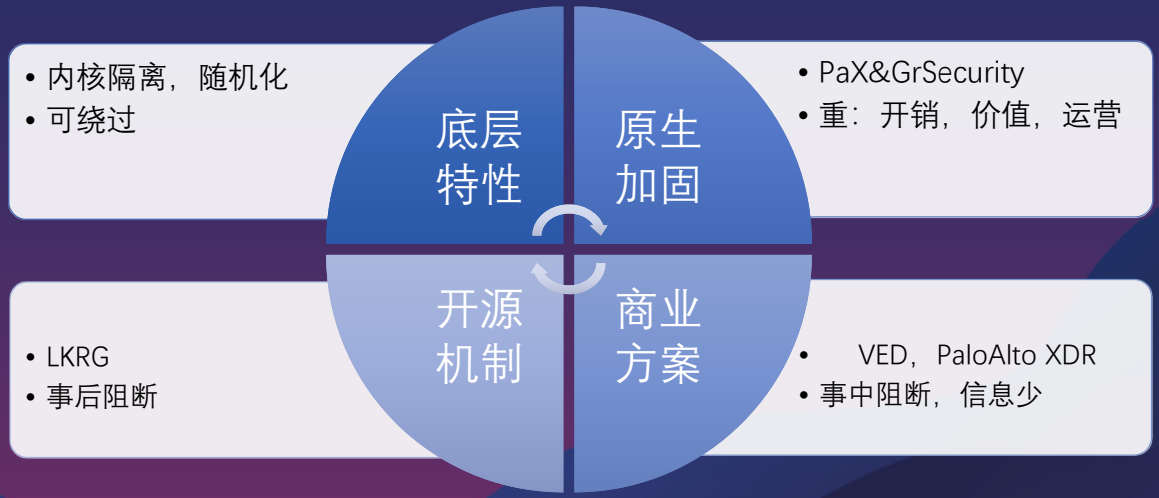
- 系统漏洞可通杀，是云主机最主要威胁。
- 挖矿/勒索等入侵的必争之地，有爆发态势。
- 内核处在纵深体系中间，漏洞多于固件/虚拟化，影响大于基础软件/用户应用，收益大。
- 建设最后一道防线，是第一优先。





内核漏洞利用：现状

- 内核漏洞攻防现状：
  - 内核版本碎片化和支持断供，关键系统难更新。
  - 利益驱动的漏洞研究暗流涌动。
- 现有防御选项。
  - 开源方案关注事后阻断，即时阻断弱，已有对抗。



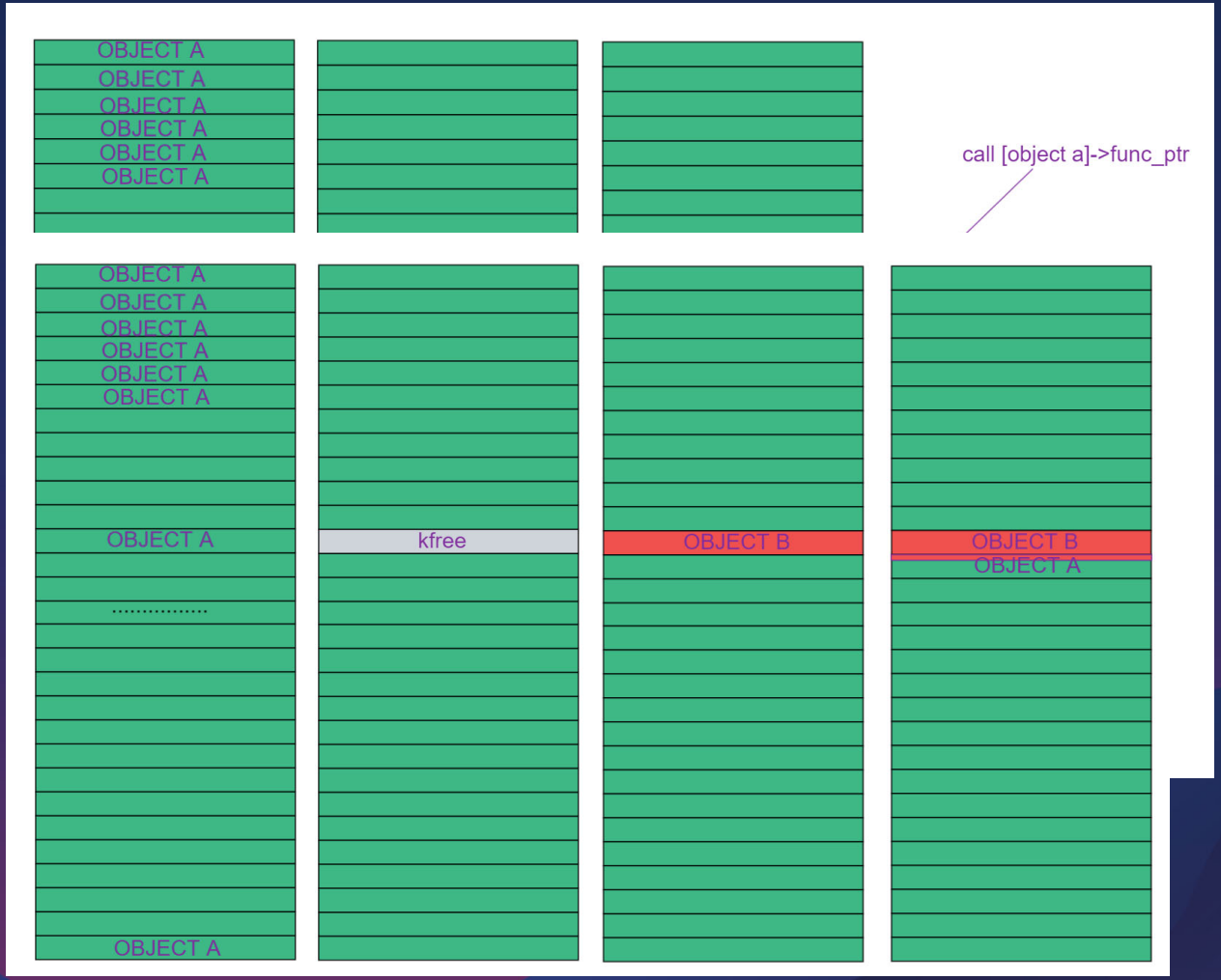
漏洞利用技术

常用的漏洞利用手法

- 返回地址劫持
- 堆喷
- she链覆盖
- 栈交换
- ROP
- Ret2user
- 等等等

堆喷

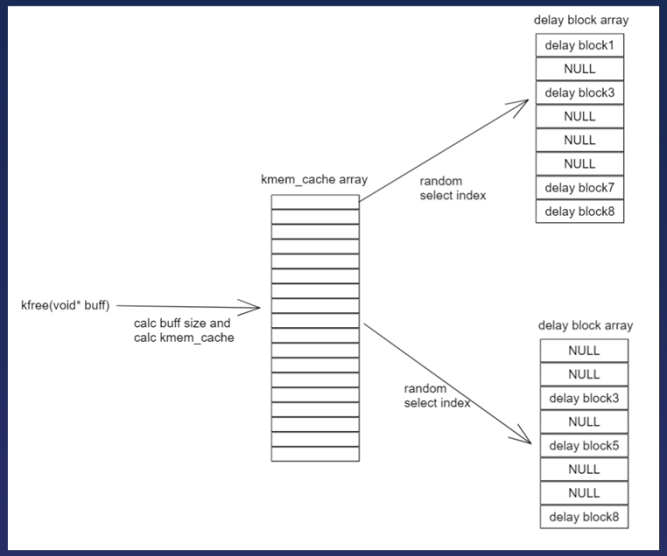
- 内核堆喷，是当前内核漏洞利用过程中非常重要的部分。80%的漏洞利用，都依赖堆喷。
- 堆喷利用手法。
- use after free/double free类型堆喷利用手法。
- 堆溢出类型堆喷利用手法。

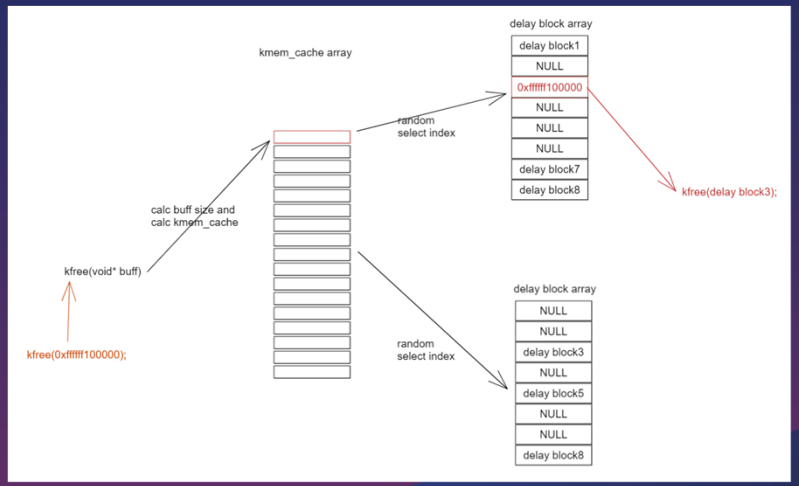


传统漏洞防御技巧

堆喷干扰

- 通过hook接管linux内核slab，slub，对申请的内存尾部追加少量随机字节，并增加随机延迟释放，可以有效的干扰堆喷，导致其失败。从而做到机制性的防御，具备更强的通用性。





堆喷干扰

- 在漏洞利用过程中，只要是利用堆喷的，都可以对其进行干扰，导致堆喷失败。

```
admin-pc@ubuntu:~/tencent$ ./exp
[.] namespace sandbox setup successfully
[.] disabling SMEP & SMAP
[.] scheduling 0xffffffff81064290(0x406e0)
[.] waiting for the timer to execute
[.] done
[.] SMEP & SMAP should be off now
[.] getting root
[.] executing 0x402003
[.] done
[.] should be root now
[.] checking if we got root
[-] something went wrong =(
[!] don't kill the exploit binary, the kernel will crash
```

```
admin-pc@ubuntu:~/tencent$ sudo insmod khm.ko
[sudo] password for admin-pc:
admin-pc@ubuntu:~/tencent$ ./exp
[.] namespace sandbox setup successfully
[.] disabling SMEP & SMAP
[.] scheduling 0xffffffff81064290(0x406e0)
setsockopt(PACKET_VERSION): Device or resource busy
```

ROP防御

- ROP在现代漏洞利用中，基本是必不可缺的一个环节。基本走到这一步，说明漏洞利用已经成功了一大半了，已经进行了控制流的劫持。漏洞通过rop来调用敏感api，来完成权限提升，容器逃逸。

- 对提权，容器逃逸相关的敏感api进行HOOK。
  - 检测栈指针的合法性。
  - 调用来源的合法性。
  - 调用指令合法性。

信息泄露缓解

- 全局变量地址泄漏。
  - 干扰堆喷。
- dmesg 异常栈指针泄漏。
  - 符号地址鉴权脱敏。
- kallsyms泄漏。
  - 符号地址鉴权脱敏。

安全状态监控

- 对系统中所有的进程进行监控，监控进程的cred和namespace，及时发现进程的异常权限提升和容器逃逸行为。
- CR4 SMEP & SMAP状态监测，及时发现漏洞利用过程中，对SMEP & SMAP的禁用。

- 传统的内核加固方式很多，不一一列举。

基于intel PT的漏洞检测方案

基于硬件特性的漏洞检测方案

- 随着时代的发展，CPU 支持的硬件特性越来越多，安全相关的大家听的最多的就是影子栈，并且在一些场景下已经得到应用。
- 基于软件的漏洞防御，CFI/CFG，在 Windows 中，已经得到广泛应用。
- CET IBT（indirect branch tracker）机制，基于硬件的简化版本CFI，但是很多CPU并不支持。
- 其实Linux kernel 目前对这些安全机制都不支持。

- 我们想对老版本系统进行改造，实现这三个特性，并且无需重新编译内核。

可行性分析

- 想要实现这些特性，我们需要具备一些前置条件：
  - Shadow stack：
    - 获取指令流，从中提取出 call 和 ret 两种指令。
    - 模拟一个栈，去模拟 call 和 ret 的入栈出栈操作，识别入栈和出栈不对等的情况。
  - IBT/CFI：
    - 捕获所有的跳转指令，获取指令跳转信息。
    - 分析跳转指令的合法性。
- 如何提取指令序列?
- 如何确定跳转指令的合法性?

技术选型

- 在x86 intel 架构上，想要获取程序的 call ， ret 和间接跳转的指令，在不修改源码的情况下有几种方式：
  - LBR (Last branch record)，将跳转分支信息保存到MSR寄存器中，记录数量有限。可以通过技巧做到实时拦截。
  - BTS (Branch Trace Messages)，可以将跳转分支记录到内存中，保存数量多。性能低，可以通过技巧做到实时拦截。
  - PT (Processor Trace)，属于BTS的进化版本，性能更好。但无法通过技巧进行实时拦截。

- 通过对三种机制的分析，最终选择采用PT技术。

技术选型

- 通过查阅资料，我们发现linux perf工具已经集成了intel PT的能力，用于做性能分析。

- 分为user模块和kernel模块：
  - Kernel模块负责完成intel PT的所有trace功能。
  - User模块负责保存PT trace数据，并解析。
  - Kernel 和user 采用ringbuff进行数据交互。

- 站在巨人的肩膀上，我们对perf进行改造。

堆喷干扰

- 支持trace 的指令类型。

- 支持用户态，内核态，和进程过滤，地址范围过滤等。非常匹配我们的需求。

Table 32-1. COFI Type for Branch Instructions

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2), RET (C3, C2 xx)
Far Transfers	INT1, INT3, INT *n*, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

7	CR3Filter	0	0: Disables CR3 filtering. 1: Enables CR3 filtering. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 0] ("CR3 Filtering Support") is 0.

2	OS	0	0: Packet generation is disabled when CPL = 0. 1: Packet generation may be enabled when CPL = 0.
3	User	0	0: Packet generation is disabled when CPL > 0. 1: Packet generation may be enabled when CPL > 0.

35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 32.2.5.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 1.

Intel PT的工作机制

- Intel PT trace buff 分为两种类型：
  - Single Range Out.
  - Table of Physical Addresses(ToPA).
- ToPA可以通过配置IA32_RTIT_OUTPUT_BASE来指定ToPA Table的第一个表，如果存在多个表，可以在表的最后一项指定下一个ToPA的地址，并将END标志位置为1。
- 每一个ToPA表，可以指定多个OutputRegion，并且每一个OutputRegion的大小可以不同，最小是4k最大是128M。

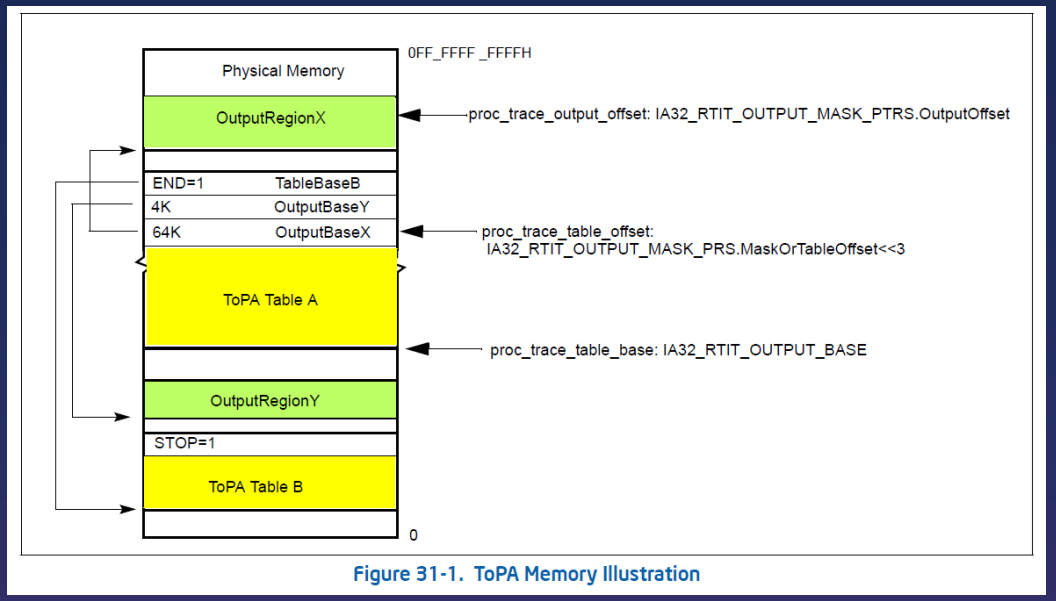


Figure 31-1. ToPA Memory Illustration

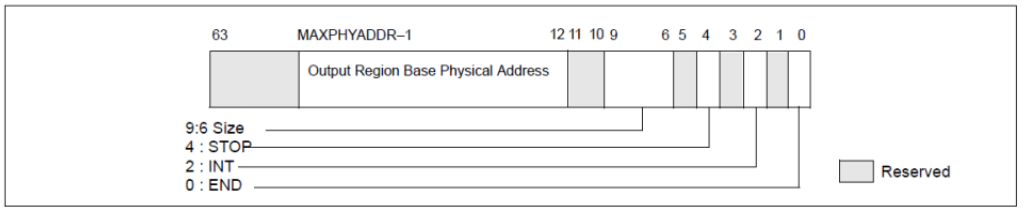


Figure 32-2. Layout of ToPA Table Entry

Intel PT的工作机制

- Intel PT 类似于生产者和消费者模型。
- 每当CPU执行指令时，如果符合filter的条件，那么将会向trace buff中写入压缩的数据。
- 当buff写满之后，将会产生PMI中断。
- 在中断中，先停止trace ，然后将buff的数据保存，并重新开启Intel PT trace。
- 但是在PMI中断中，直接将缓存的数据进行转存，会比较麻烦，所以大家的做法，普遍都是用一个用户层的程序，去读取这个trace buff数据。由用户层程序完成这个持久化操作。

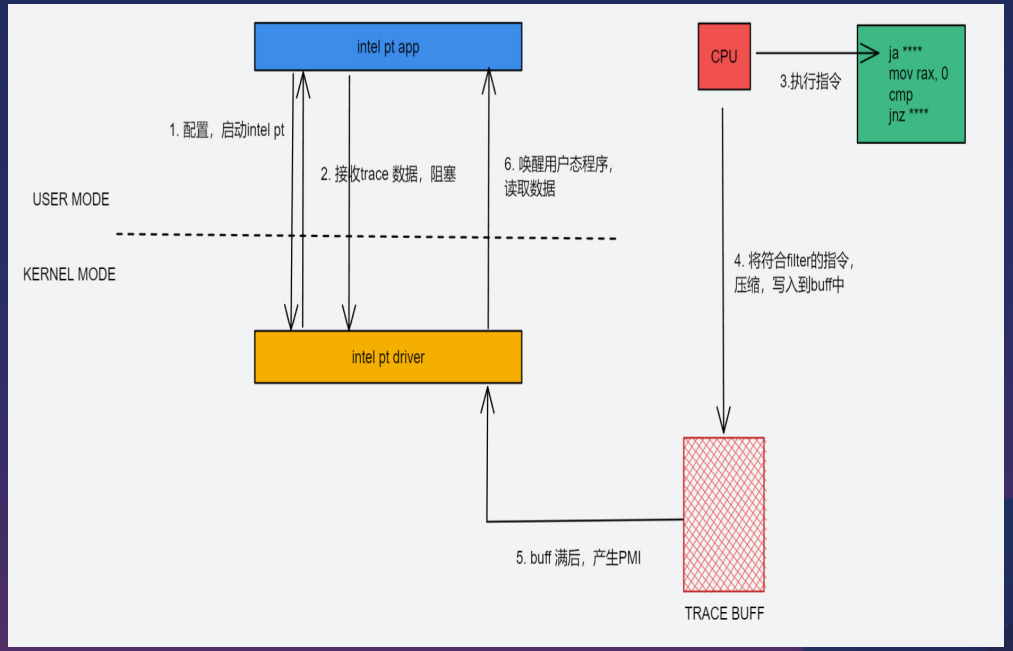
Intel PT的工作机制

- 在PMI中断中，处理Intel PT事件：
  - 关闭PT trace。
  - 更新状态。
  - 通知用户态从ringbuff中读数据。
  - 重新配置trace buff，指向下一个page entry。
  - 启动PT trace。

```
     * do anything (particularly, re-enable) for this event here.
     */
    if (!READ_ONCE(pt->handle_nmi))
        return;

    if (!event)
        return;

    pt_config_stop(event);

    buf = perf_get_aux(&pt->handle);
    if (!buf)
        return;

    pt_read_offset(buf);

    pt_handle_status(pt);

    pt_update_head(pt);

    perf_aux_output_end(&pt->handle, local_xchg(&buf->data_size, 0));

    if (!event->hw.state) {
        int ret;

        buf = perf_aux_output_begin(&pt->handle, event);
        if (!buf) {
            event->hw.state = PERF_HES_STOPPED;
            return;
        }

        pt_buffer_reset_offsets(buf, pt->handle.head);
        /* snapshot counters don't use PMI, so it's safe */
        ret = pt_buffer_reset_markers(buf, &pt->handle);
        if (ret) {
            perf_aux_output_end(&pt->handle, 0);
            return;
        }

        pt_config_buffer(buf);
        pt_config_start(event);
    }
}
```

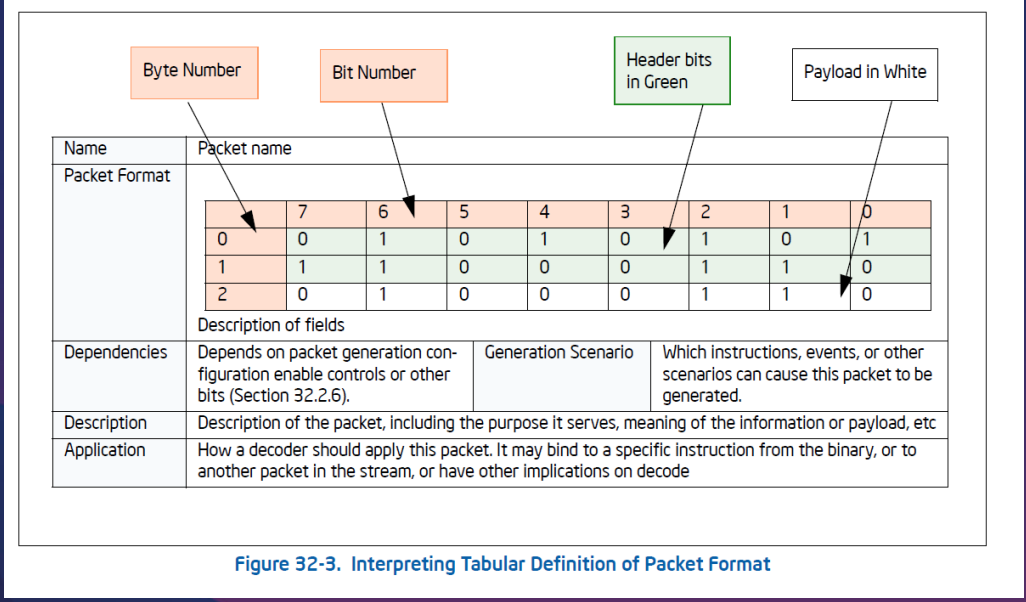
Intel PT的工作机制

- 整体流程图：
  1. 用户层发送请求给驱动，驱动层配置并启动intel PT。
  2. 用户层去拉起目标程序执行，同时请求接收trace数据，如果没有数据将阻塞。
  3. 目标程序执行，遇到符合filter的指令，将会由CPU生成压缩的数据包，保存到trace buff中。
  4. Trace buff的数据满后，会产生PMI中断。
  5. PMI handle中，先停止intel PT trace，获取trace buff的数据，配置好ring buff，唤醒用户层阻塞的线程。
  6. 分为两步：
     1. 用户层程序负责从ring buff中读取数据。
     2. 驱动重新配置trace buff entry，防止覆盖未被用户层即时读取的数据。并恢复intel PT trace。



Intel PT 数据包解析

- 我们对数据进行解析，解包，按照intel的格式进行拆解，并过滤不关注的数据。
- 数据包格式如下：我们以TIP数据包为例，10110 为数据包类型标记。



Name	Packet name							
Packet Format								
	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	0	1
1	1	1	0	0	0	1	1	0
2	0	1	0	0	0	1	1	0
	Description of fields							
Dependencies	Depends on packet generation configuration enable controls or other bits (Section 32.2.6).		Generation Scenario		Which instructions, events, or other scenarios can cause this packet to be generated.			
Description	Description of the packet, including the purpose it serves, meaning of the information or payload, etc							
Application	How a decoder should apply this packet. It may bind to a specific instruction from the binary, or to another packet in the stream, or have other implications on decode							

Figure 32-3. Interpreting Tabular Definition of Packet Format

Table 32-17. IP Packet Definition

Name	Target IP (TIP) Packet								
Packet Format									
	7	6	5	4	3	2	1	0	
0	IPBytes			0	1	1	0	1	
1	TargetIP[7:0]								
2	TargetIP[15:8]								
3	TargetIP[23:16]								
4	TargetIP[31:24]								
5	TargetIP[39:32]								
6	TargetIP[47:40]								
7	TargetIP[55:48]								
8	TargetIP[63:56]								
Dependencies	PacketEn		Generation Scenario		Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX[1].				
Description	Provides the target for some control flow transfers								
Application	Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.								
	The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.								

NOTES:
1. EENTER, EEXIT, ERESUME, AEX would be possible only for a debug enclave.

Intel PT 数据包解析

- 我们以TIP数据包为例，0~4位10110 表示TIP数据包。
- 5~7位 IPBytes含义如下：例如说 IPBytes 为000时，数据包不带任何数据。IPBytes 为010时，后续8字节的0~31位对应Payload的0~31位，32~63位对应Payload的32~63位。

Table 32-17. IP Packet Definition

Name	Target IP (TIP) Packet								
Packet Format									
	7	6	5	4	3	2	1	0	
0	IPBytes			0	1	1	0	1	
1	TargetIP[7:0]								
2	TargetIP[15:8]								
3	TargetIP[23:16]								
4	TargetIP[31:24]								
5	TargetIP[39:32]								
6	TargetIP[47:40]								
7	TargetIP[55:48]								
8	TargetIP[63:56]								
Dependencies	PacketEn	Generation Scenario	Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX[1].						
Description	Provides the target for some control flow transfers								
Application	Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.								
	The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.								

NOTES:
1. EENTER, EEXIT, ERESUME, AEX would be possible only for a debug enclave.

Table 32-18. FUP/TIP IP Reconstruction

IPBytes	Uncompressed IP Value							
	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
000b	None, IP is out of context							
001b	Last IP[63:16]						IP Payload[15:0]	
010b	Last IP[63:32]				IP Payload[31:0]			
011b	IP Payload[47] extended		IP Payload[47:0]					
100b	Last IP [63:48]		IP Payload[47:0]					
101b	Reserved							
110b	IP Payload[63:0]							
111b	Reserved							

Intel PT 数据包解析

- 并不是所有的跳转指令都记录，例如说 jmp $rip + 10，这种指令源地址和目标地址都是可以通过解析二进制进行获取，所以就不会生成数据包。
- 带条件的直接跳转指令，只会记录条或者不跳，生成TNT bit。CPU会将多个TNT bit 压缩成一个数据包。

- 注意：这里的绝大多数Payload并没有记录源IP，我们需要根据解析二进制文件，将每一个跳转信息都和二进制进行一一对应，才能够计算源IP和目标IP。

Intel PT 数据包解析

- 我们完成数据包解析后，提取出感兴趣的跳转指令序列，例如说：
  - ret， ret *
  - jmp reg， jmp [reg], jmp [reg + *], ...
  - call reg, call [reg], call [reg + *], ...

- 到此，我们就完成了trace 指令收集的任务。

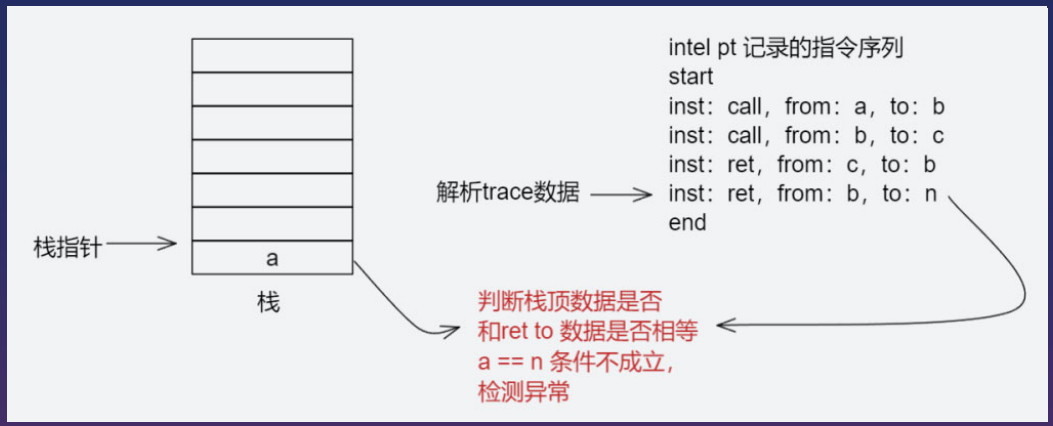
Shadow stack

- 我们目前已经拿到了所有的call 和ret 的指令序列，我们自己实现一个栈，来模拟call 的入栈操作，和ret 的出栈操作。
- 执行call指令后，会将call 指令的下一条指令的地址压入栈中，用于ret返回：
  - 返回地址 = 指令地址+ 指令长度；

- 注意：
  - 每一个线程分配一个单独的栈。
  - 中断产生，线程切换等问题需要进行逐一处理。
  - 调用栈原本就不对称，例如线程刚创建后，栈中已经有一部分数据，但是当前线程未产生call指令压栈。
  - spectre_v2 漏洞， Retpoline修复方案不兼容！！

Shadow stack

- 栈模拟，检测逻辑



IBT+CFI

- 如何确定跳转指令的合法性?
  - 直接调用。
    - 肯定合法。
  - 间接调用。
    - 可能出现问题。

- 对于间接调用，对应的多数都是函数指针，callback，虚函数等，并且是漏洞喜欢篡改的地方。

IBT+CFI

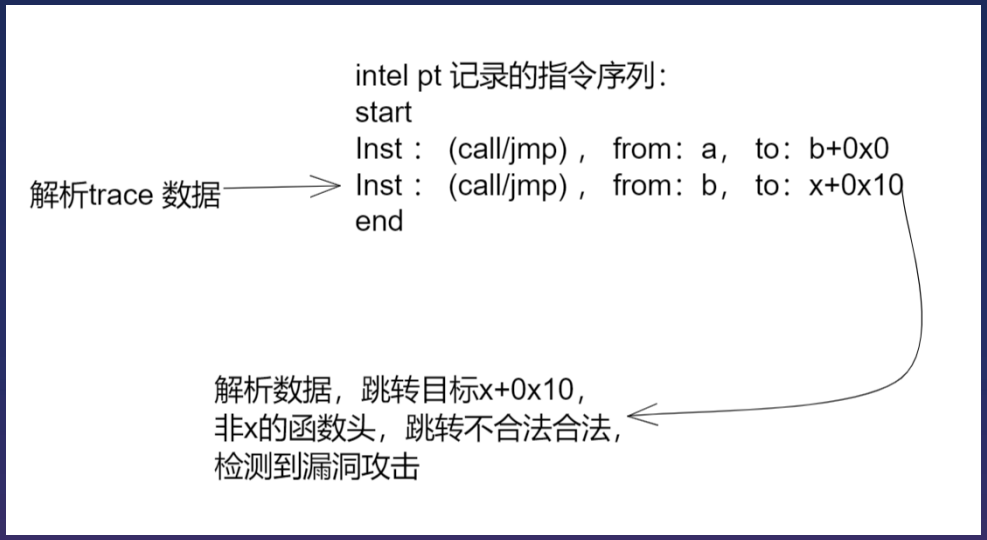
- 如何确认间接跳转的合法性?
  - 目标函数地址，是否是一个函数的起始位置，IBT机制。
  - 通过源码得知指针是否可以指向目标函数，CFI机制。
- 如何确定是否是函数起始位置?
  - 通过符号解析来实现。
- 通过源码得知函数指针是否可以指向目标函数，自动化如何实现?
  - 编译器生成call graph。

IBT+CFI

- 传统的指针分析不能满足需求。
- 如何生成call graph?
  - 获取到内核二进制文件的对应源码。
  - 通过二次开发的GCC，对源码进行编译，在编译过程中，提取基础信息，并插入到数据库。
  - 根据数据库进行关联查询。
- 后面章节详细讲解。

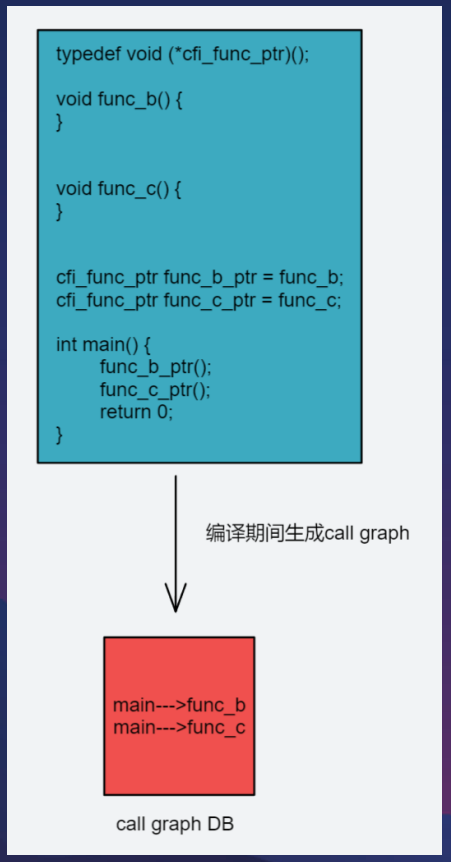
IBT+CFI

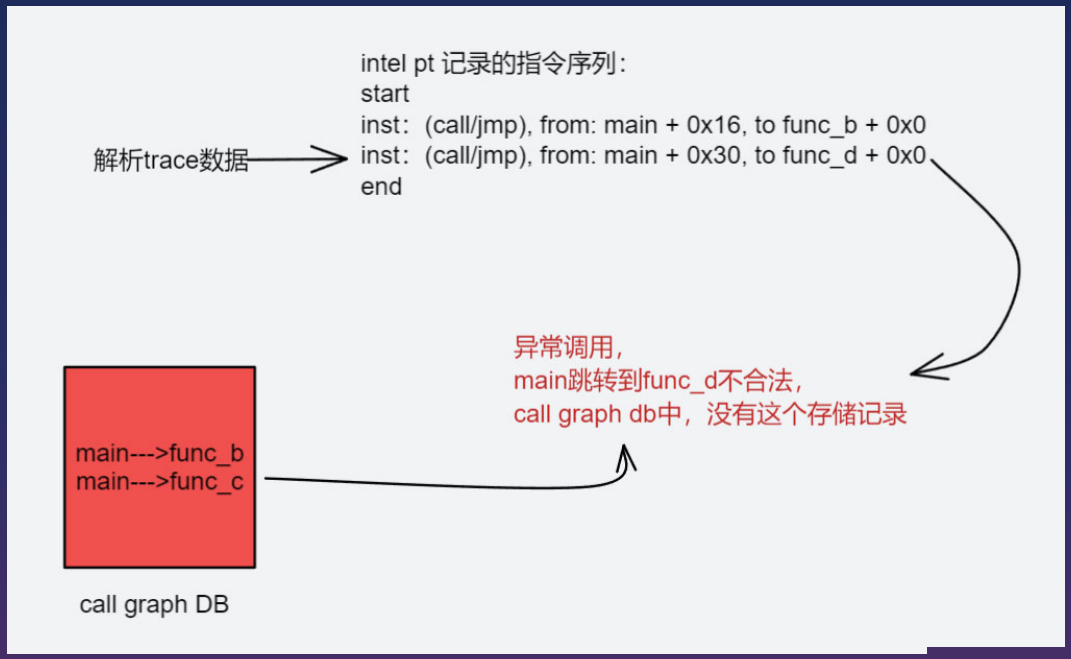
- IBT：


intel pt 记录的指令序列：
start
Inst： (call/jmp)， from：a， to：b+0x0
Inst： (call/jmp)， from：b， to：x+0x10
end
解析trace 数据
解析数据，跳转目标x+0x10，
非x的函数头，跳转不合法合法，
检测到漏洞攻击

IBT+CFI

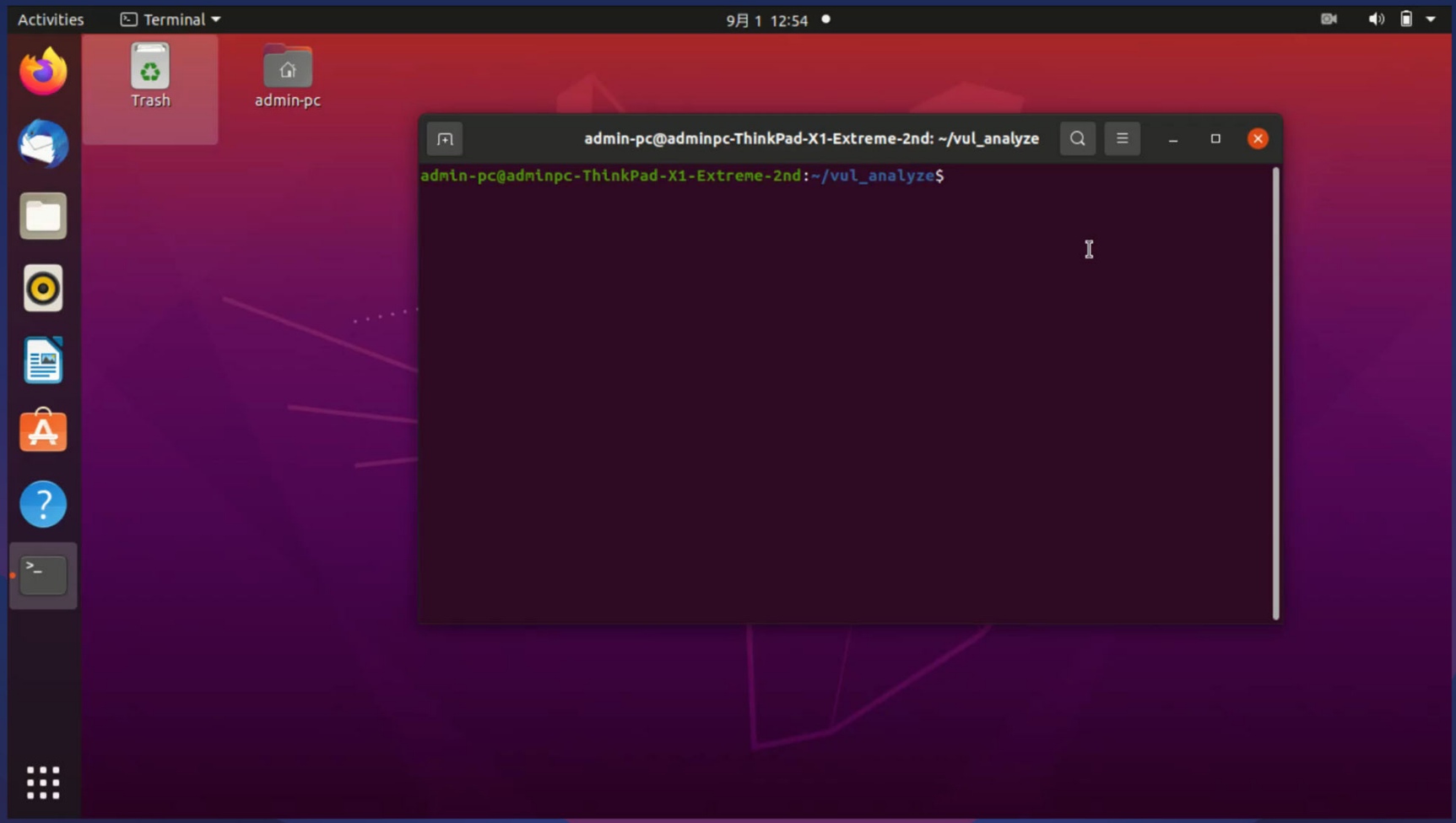
- CFI算法：





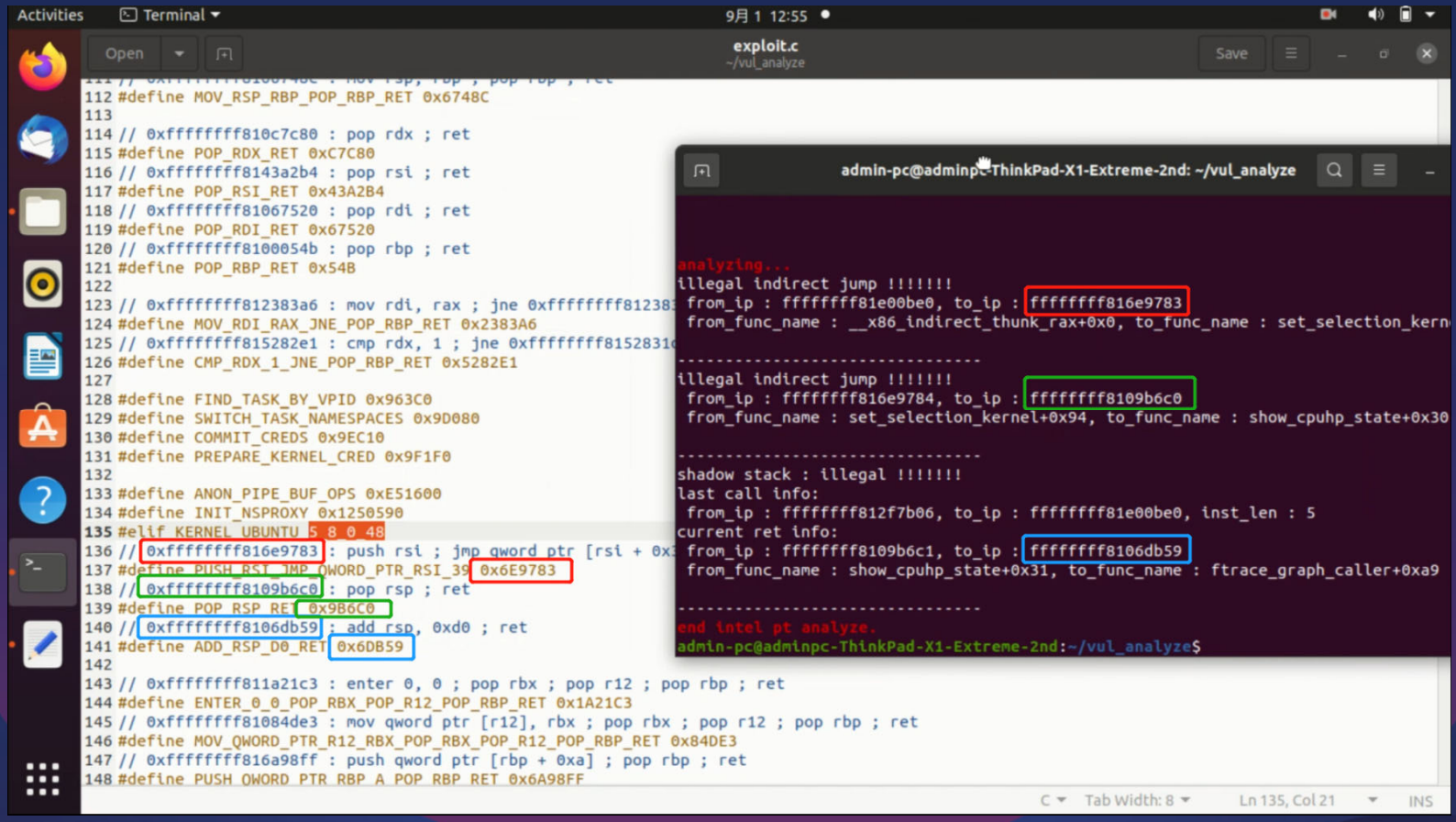
基于硬件特性的漏洞防御方案

- 演示效果如下：



基于硬件特性的漏洞防御方案

- 演示效果如下：



基于编译器的漏洞防御方案

张海全

基于编译器的漏洞防御方案

- 编译器生成call graph。
- struct_san 漏洞防御。

call graph

- 直接函数调用，我们不关注。
- 间接函数调用的方式：
  - 全局函数指针。
  - 全局函数指针链表。
  - 参数传递的函数指针。
  - 结构体内的函数指针。
  - 函数指针数组。

- 函数指针调用是程序分析的难点。
  - 源码文件太多。

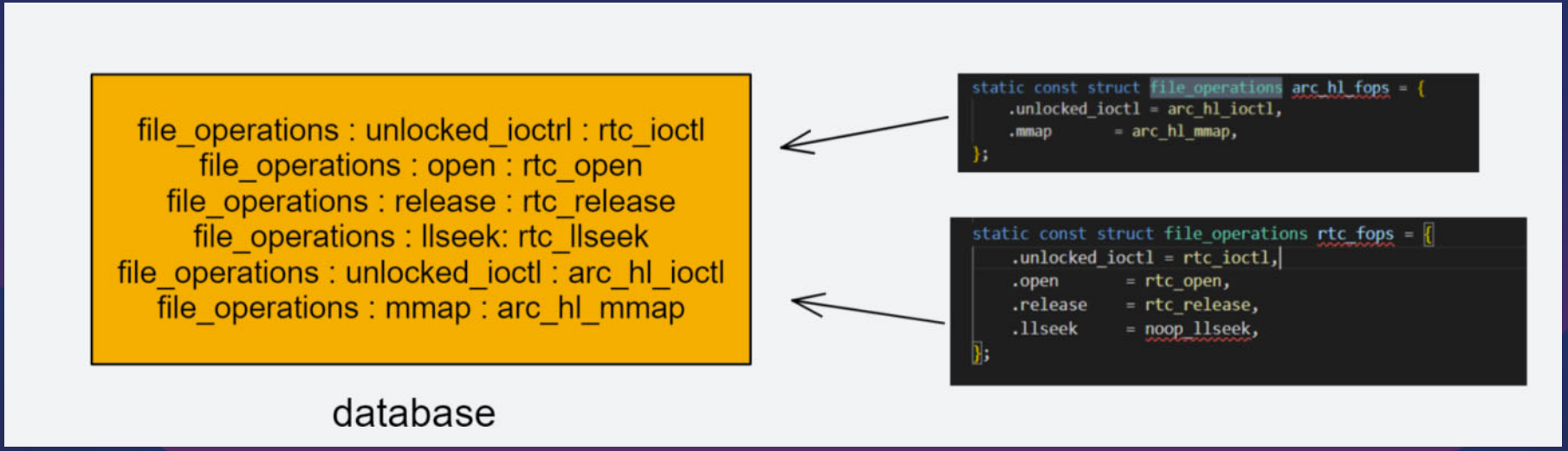
call graph

- 发现内核中的大量结构体的函数指针都是全局结构体定义，然后再使用这个结构体，定义如下：

```
static const struct file_operations rtc_fops = {
    .unlocked_ioctl = rtc_ioctl,
    .open       = rtc_open,
    .release    = rtc_release,
    .llseek     = noop_llseek,
};
```

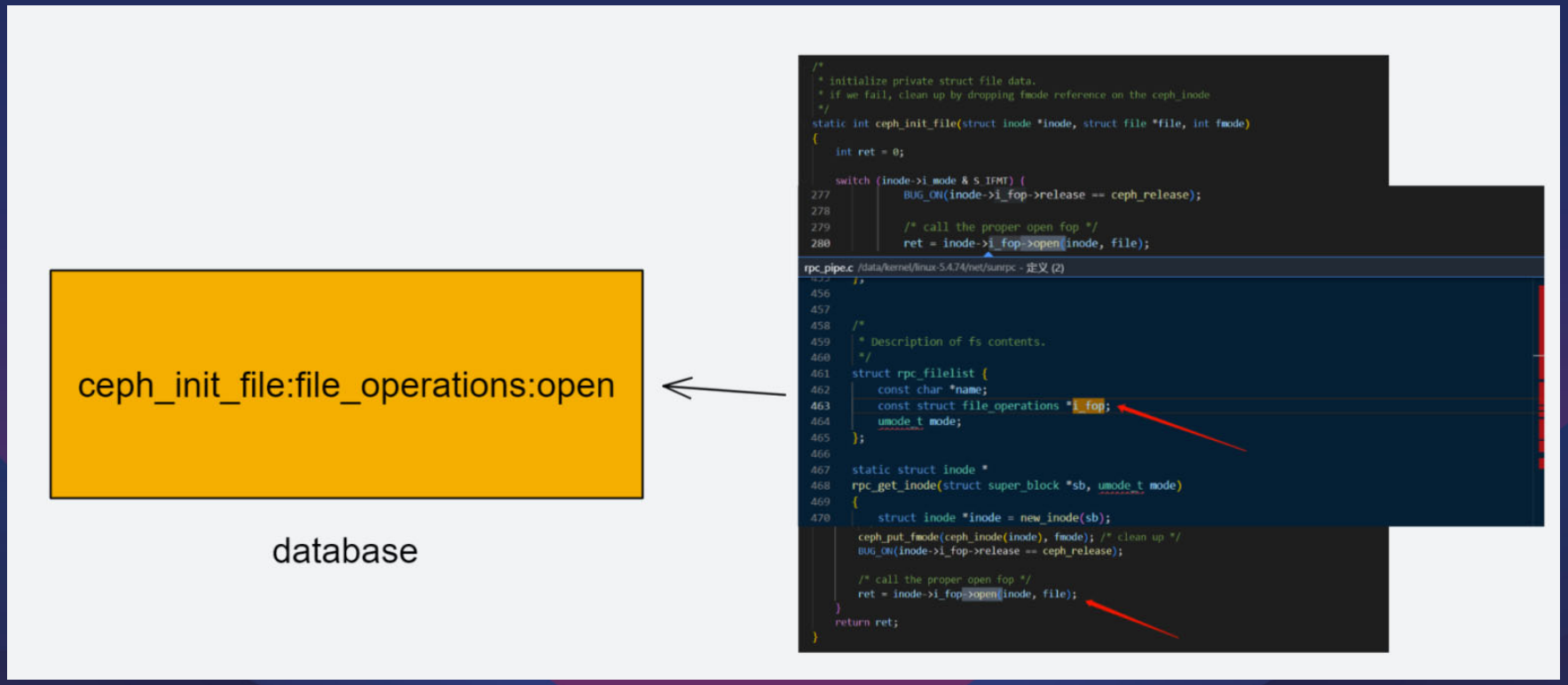
call graph

- 通过这个规律，在编译器parser阶段，编译器在parse到全局结构体时。提取信息，并插入数据库，按照如下格式：
  - 结构体的类型：字段名称：函数名字；



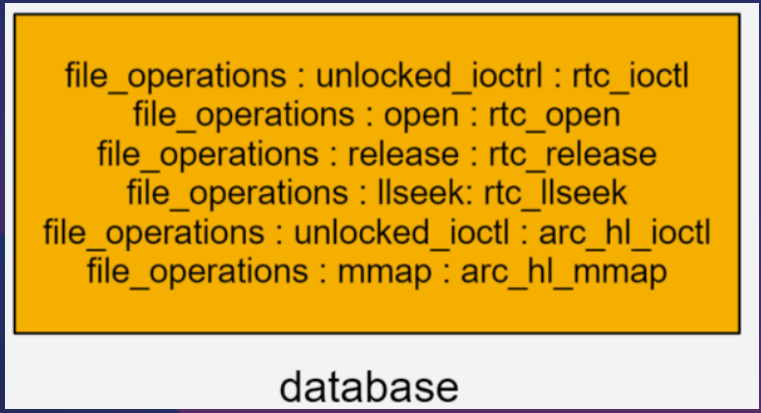
call graph

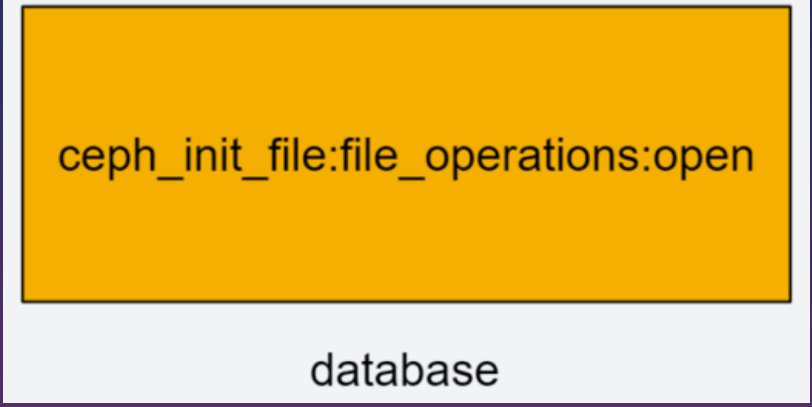
- 在编译器parser阶段，编译器在parse到函数时，如果函数内出现间接调用，通过解析gimple，最终生成如下格式数据，并插入数据库：
  - function : struct : field;



call graph

- 编译完成后进行数据整理，通过上面的两个表中的数据，我们可以得知：
  - ceph_init_file函数里存在一个file_operations->open的间接调用。
  - file_operations->open指针为rtc_open函数是合法的。
  - 从而推导：ceph_init_file调用rtc_open是合法的。





call graph

- 通过函数参数传递过来的函数指针。
  - 常数传播 + 别名分析。

- 通过向链表注册的callback函数，在加入全局符号表时，也可以进行数据原型的提取，思路如上，并结合常数传播，即可生成调用关系数据。

- 整体来说我们的思想就是提取原子信息，插入到数据库。后续进行关联查询。

- 精准度和误报率进行衡量。

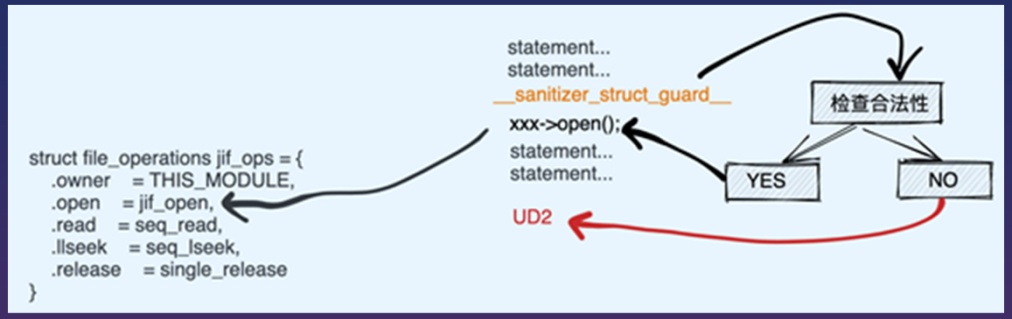
struct_san

- 为了防止堆喷，篡改结构体函数指针这种利用方式，我们提出了struct-sanitizer(struct_san)这种新的控制流完整性检测机制。以最少的插装，最小的性能损耗为宗旨，更严格的校验来抵御漏洞攻击。

	当前主流CFI	struct_san
验证方式	函数指针类型	函数指针类型+结构体实例
插装量级	全量插装	最小规模插装

struct_san

- 例如说：
  - 漏洞篡改结构体的ops指针进行流程劫持。
  - 我们只需要在所有ops函数指针执行前插桩，从而检测函数指针的合法性。



struct_san

- 如何保证只在ops函数指针调用前进行插装?
  - 新增一个GNU Attributes __attribute__ ((sanitize_struct)) 。
- 例如想要保护内核中pipe_buf_operations->release()函数。
  - 在结构体类型声明时加入此关键字。
- 编译器会在所有调用pipe_buf_operations结构体函数指针的位置插入 __sanitizer_struct_guard__()。

- struct_san会将此类型的所有结构体实例保存到.sanitize_struct段内。
  - 遵循前面描述的规律，结构体的函数指针，都是定义在一个全局结构体变量里，编译期间，就能确定地址。

struct_san

插桩前后在gcc的gimple IR中的不同表示：

```
;; Function pipe_buf_release (pipe_buf_release, funcdef_no=3005, decl_uid=35109, cgraph_uid=3098, symbol_order=3148

;;
3 basic blocks, 2 edges, last basic block 3.

;; basic block 2, loop depth 0, count 1073741824 (estimated locally), maybe hot
;;  prev block 0, next block 1, flags: (NEW, VISITED)
;;  pred:       ENTRY [always]  count:1073741824 (estimated locally) (FALLTHRU,EXECUTABLE)
;;  succ:       EXIT [always]  count:1073741824 (estimated locally) (EXECUTABLE)

__attribute__((sanitize_struct, noinline))
pipe_buf_release (struct pipe_inode_info * pipe, struct pipe_buffer * buf)
{
  const struct pipe_buf_operations * ops;
  void (*<T257e>) (struct pipe_inode_info *, struct pipe_buffer *) _1;

;;   basic block 2, loop depth 0, count 1073741824 (estimated locally), maybe hot
;;    prev block 0, next block 1, flags: (NEW, VISITED)
;;    pred:       ENTRY [always]  count:1073741824 (estimated locally) (FALLTHRU,EXECUTABLE)
  # DEBUG BEGIN_STMT
  ops_4 = buf_3(D)->ops;
  # DEBUG ops => ops_4
  # DEBUG BEGIN_STMT
  buf_3(D)->ops = 0B;
  # DEBUG BEGIN_STMT
  _1 = ops_4->release;
  _1 (pipe_6(D), buf_3(D));
  return;
;;    succ:       EXIT [always]  count:1073741824 (estimated locally) (EXECUTABLE)

}
```

```
;; Function pipe_buf_release (pipe_buf_release, funcdef_no=3005, decl_uid=35109, cgraph_uid=3098, symbol_order=3148

;;
3 basic blocks, 2 edges, last basic block 3.

;; basic block 2, loop depth 0, count 1073741824 (estimated locally), maybe hot
;;  prev block 0, next block 1, flags: (NEW, VISITED)
;;  pred:       ENTRY [always]  count:1073741824 (estimated locally) (FALLTHRU,EXECUTABLE)
;;  succ:       EXIT [always]  count:1073741824 (estimated locally) (EXECUTABLE)

__attribute__((sanitize_struct, noinline))
pipe_buf_release (struct pipe_inode_info * pipe, struct pipe_buffer * buf)
{
  const struct pipe_buf_operations * ops;
  void (*<T257e>) (struct pipe_inode_info *, struct pipe_buffer *) _1;
  void (*<T257e>) (struct pipe_inode_info *, struct pipe_buffer *) STRUCT_I_8;

;;   basic block 2, loop depth 0, count 1073741824 (estimated locally), maybe hot
;;    prev block 0, next block 1, flags: (NEW, VISITED)
;;    pred:       ENTRY [always]  count:1073741824 (estimated locally) (FALLTHRU,EXECUTABLE)
  # DEBUG BEGIN_STMT
  ops_4 = buf_3(D)->ops;
  # DEBUG ops => ops_4
  # DEBUG BEGIN_STMT
  buf_3(D)->ops = 0B;
  # DEBUG BEGIN_STMT
  _1 = ops_4->release;
  STRUCT_I_8 = __sanitizer_struct_guard__ (&ops_4->release, _1);
  STRUCT_I_8 (pipe_6(D), buf_3(D));
  return;
;;    succ:       EXIT [always]  count:1073741824 (estimated locally) (EXECUTABLE)

}
```

struct_san

- struct_san目前只在内核中完成了相关实现。其算法是在内核中开辟一个128M大小shadow memory用来保存结构体和结构指针的对应关系。
- __sanitizer_struct_guard__()在调用时会检测传入的struct和函数指针是否在shadow memory中，如果不在则抛出一个ud2异常，否则返回函数指针。实现方案如下：

```
void *__sanitizer_struct_guard__(void *p, void *fn)
{
    u64 off = (((u64)p) ^ ((u64)fn)) % STRUCT_MAX_NR;
    if (!globl_offset[off])
        asm volatile("ud2");
    return fn;
}
```

- 这个算法参考了AddressSanitizer的实现，兼顾了效果和效率。

struct_san

以漏洞CVE-2021-22555的攻击代码为例，在启用struct_san的情况下，阻断了攻击代码的执行，起到了有效的防御。

```
test1@structsan:~$ ./exploit
[+] Linux Privilege Escalation by theflow@ - 2021

[+] STAGE 0: Initialization
[*] Setting up namespace sandbox...
[*] Initializing sockets and message queues...

[+] STAGE 1: Memory corruption
[*] Spraying primary messages...
[*] Spraying secondary messages...
[*] Creating holes in primary messages...
[*] Triggering out-of-bounds write...
[   25.651883] x_tables: ip_tables: icmp.0 match: invalid size 8 (kernel) != (user) 3850
[*] Searching for corrupted primary message...
[+] fake_idx: bf9
[+] real_idx: be5

[+] STAGE 2: SMAP bypass
[*] Freeing real secondary message...
[*] Spraying fake secondary messages...
[*] Leaking adjacent secondary message...
[+] kheap_addr: ffff888111c0e000
[*] Freeing fake secondary messages...
[*] Spraying fake secondary messages...
[*] Leaking primary message...
[+] kheap_addr: ffff888112310000

[+] STAGE 3: KASLR bypass
[*] Freeing fake secondary messages...
[*] Spraying fake secondary messages...
[*] Freeing sk_buff data buffer...
[*] Spraying pipe_buffer objects...
[*] Leaking and freeing pipe_buffer object...
[+] anon_pipe_buf_ops: ffffffff82b70d60
[+] kbase_addr: ffffffff81000000

[+] STAGE 4: Kernel code execution
[*] Spraying fake pipe_buffer objects...
[*] Releasing pipe_buffer objects...
[   25.709459] invalid opcode: 0000 [#1] SMP NOPTI
[   25.709842] CPU: 0 PID: 306 Comm: exploit Tainted: G            E     5.12.0-rc4+ #34
[   25.710004] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.13.0-0-gf21b5
[   25.710004] RIP: 0010:__sanitizer_struct_guard__+0x22/0x30
[   25.710004] Code: cc cc cc cc cc cc cc cc cc cc 66 66 66 66 66 90 55 48 31 f7 48 8b 15 40 c8 e0 01 4
[   25.710004] RSP: 0018:ffffc9000826bde0 EFLAGS: 00000246
[   25.710004] RAX: ffffffff816e9783 RBX: 0000000000000000 RCX: 0000000000000000
[   25.710004] RDX: ffffc90000010000 RSI: ffffffff816e9783 RDI: 000000000035f951b
[   25.710004] RBP: ffffc9000826bde0 R08: 0000000000000000 R09: ffff88810d0131e0
[   25.710004] R10: 0000000000000008 R11: ffff88810b42bd10 R12: ffff888112310000
[   25.710004] R13: ffff88810fa1f600 R14: ffff88810d013268 R15: ffff8881095fcb40
[   25.710004] FS:  0000000000000000(0000) GS:ffff88813bc00000(0063) knlGS:000000000854b840
[   25.710004] CS:  0010 DS: 002b ES: 002b CR0: 0000000080050033
[   25.710004] CR2: 000000000080e2000 CR3: 000000010c9ce000 CR4: 00000000000006f0
[   25.710004] Call Trace:
[   25.710004]  pipe_buf_release+0x2c/0x40
[   25.710004]  free_pipe_info+0x7d/0xc0
[   25.710004]  pipe_release+0x114/0x120
[   25.710004]  __fput+0x9f/0x250
[   25.710004]  ____fput+0xe/0x10
```

struct_san

- 我们对struct_san进行了开源，期望和业界一起探讨CFI技术的改进。后续我们也会推出一些其它的漏洞缓解技术。

- 开源地址：https://github.com/YunDingLab/struct_sanitizer

后续展望

- 相关特性部分已经移植到OpenCloudOS中。
- 希望广大安全人员共建。

引用参考文献

- https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html
- https://github.com/google/security-research/tree/master/pocs/linux/cve-2021-22555
- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- https://github.com/OpenCloudOS/community

Q&A

- sundayjiang@tencent.com/huntazhang@tencent.com

Thanks for listening