kasan 介绍

Tong.Chen

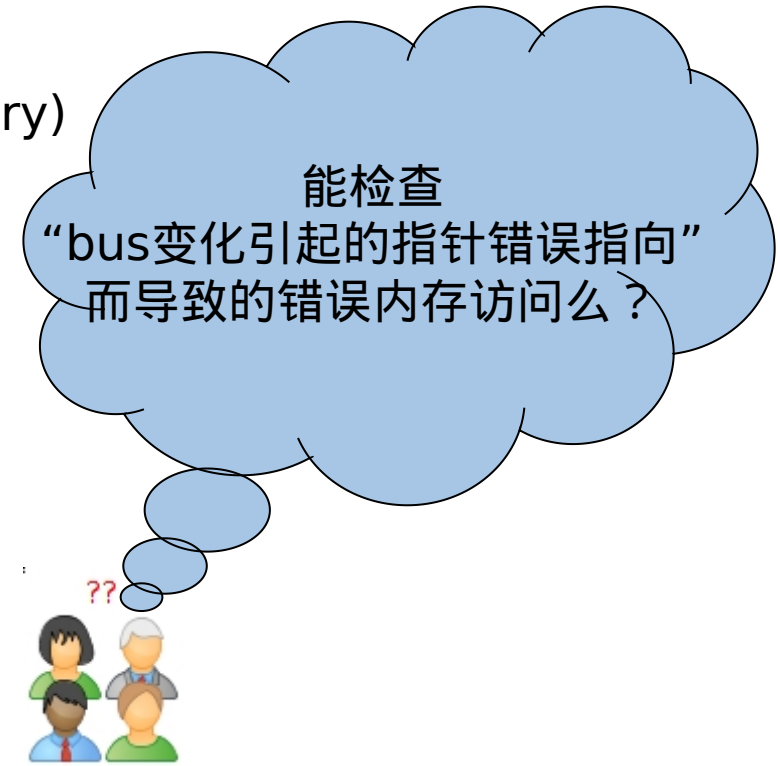
内容

1.功能
2.原理及实现
3.patch 整合
4.使用
5.参考

功能

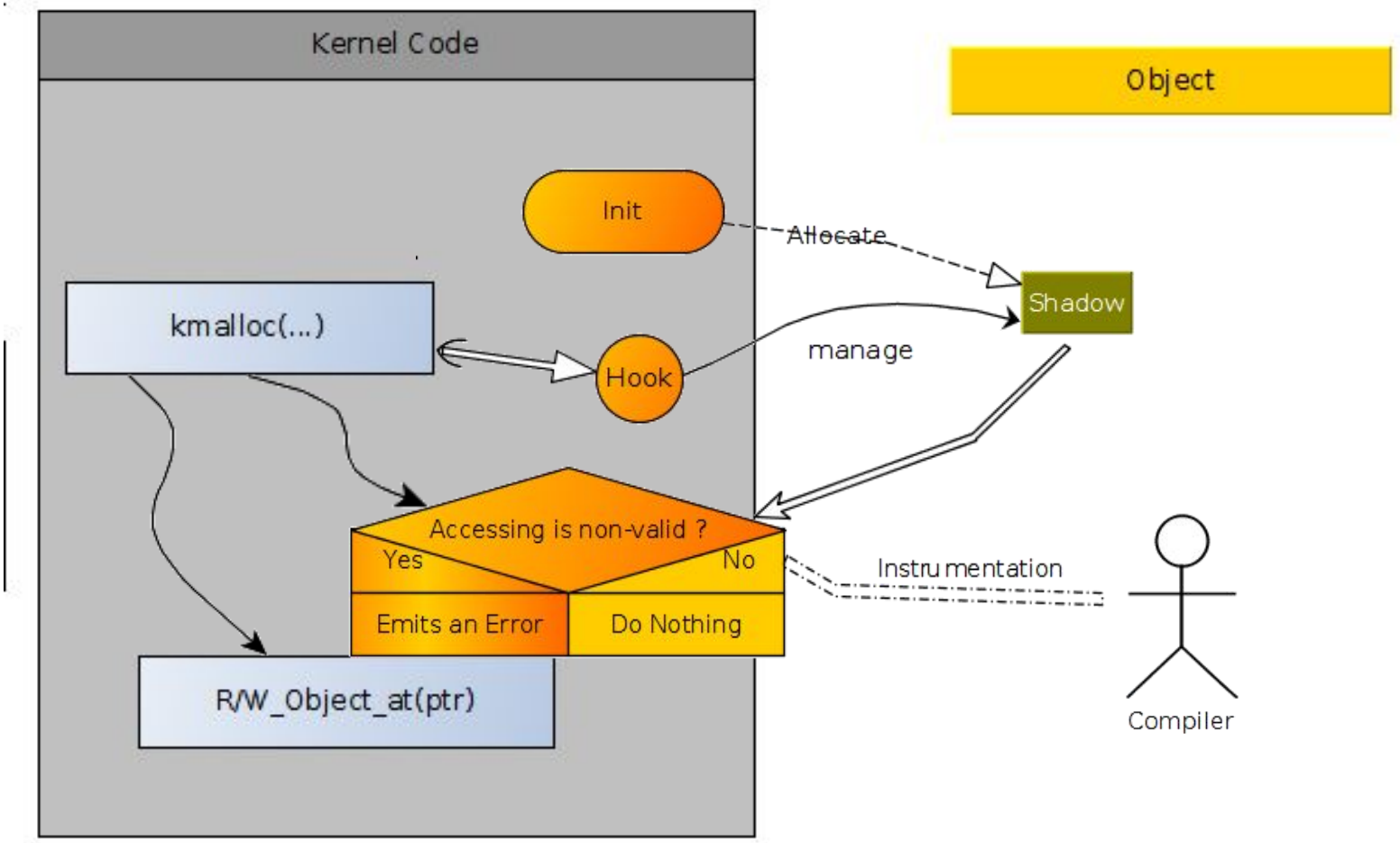
- 运行时的内存访问有效性检查
  1. Use-after-freed 错误(uaf)
     - a. slub 对象
     - b. struct page(low memory)
     - c. mempool 元素
  2. Out-of-bounds 错误(oob)
     - a. 1中所有
     - b. stack 上的局部变量
     - c. 全局数据(!)

- 当前不支持的功能
  1. vmalloc (module)
  2. 未初始化内存的访问

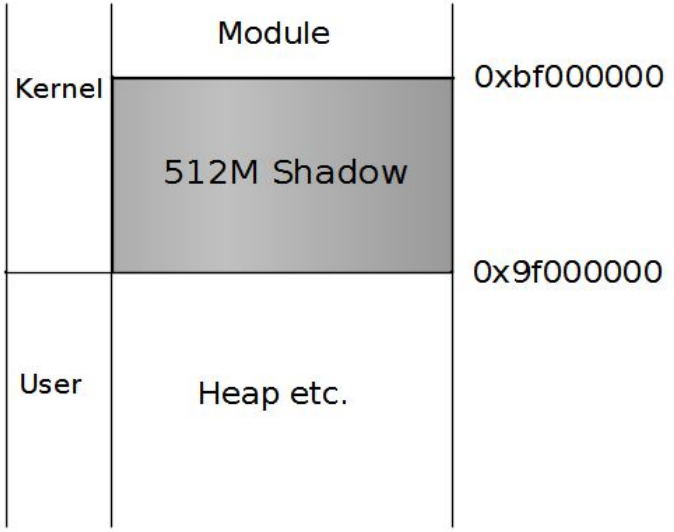


原理及实现

✦原理


Kernel Code
Object
Init
Allocate
Shadow
kmalloc(...)
Hook
manage
Accessing is non-valid ?
Yes
No
Instrumentation
Emits an Error
Do Nothing
R/W_Object_at(ptr)
Compiler

原理及实现



✦实现（ARM 32bits）

1. 架构相关部分
   a. 初始化保留 512M
   b. 分两阶段进行
      ① 首先暂时得先映射所有 shadow mem 至一个 zero page
      ② 等 low mem 地址空间准备好，再映射至实际内存

```
60 static inline const void *kasan_shadow_to_mem(const void *shadow_addr)
61 {
62         return (void *)(((unsigned long)shadow_addr - KASAN_SHADOW_OFFSET)
63                  << KASAN_SHADOW_SCALE_SHIFT);
64 }
```

   c. Shadow 字节格式
      ① 正数 n([1,7]) 表示能访问前面 n 个字节
      ② 0 表示所有 8 字节都能访问
      ③ 负数表示所有 8 字节都不能访问

原理及实现

- 实现（ARM 32bits）（续1）
  - 2. 实现的 Hooks
    - a. for slub
      - kasan_slab_alloc / kasan_slab_free
      - kasan_kmalloc_large / kasan_kfree_large
    - b. for buddy
      - kasan_alloc_pages / kasan_free_pages
    - c. for mempool
      - kasan_krealloc / kasan_kfree

```
static void kasan_poison_shadow(const void *address, size_t size, u8 value) {...}
void kasan_unpoison_shadow(const void *address, size_t size)
{
        kasan_poison_shadow(address, size, 0);

        if (size & KASAN_SHADOW_MASK) {
                u8 *shadow = (u8 *)kasan_mem_to_shadow(address + size);
                *shadow = size & KASAN_SHADOW_MASK;
        }
}
```

原理及实现

- 实现（ARM 32bits）（续2）
  - 3. Instrumentation
    - a.实现一系列函数直接给 compiler 自动做 Instr
      - ① __asan_load1 / __asan_store1
      - ② __asan_load2 / __asan_store2
      - ③ __asan_load4 / __asan_store4
      - ④ __asan_load8 / __asan_store8
      - ⑤ __asan_load16 / __asan_store16
      - ⑥ __asan_loadN / __asan_storeN

思考：
为什么这里有 N() 的版本？

```
static __always_inline void check_memory_region(unsigned long addr,
                                                size_t size, bool write)
{
        struct kasan_access_info info;
        unsigned int kf;

        //...
        if (likely(!memory_is_poisoned(addr, size)))
        {
                return;
        }

        kasan_report(addr, size, write, _RET_IP_);
}
```

原理及实现

✦实现（ARM 32bits）（续3）

3. Instrumentation(续)

a.手动地为memset/memmove/memcpy 做 Instr

① arch-dependent，asm，non_instrable

② weaken original, use our instred version instead

```
.weak memcpy
ENTRY(__memcpy)
ENTRY(memcpy)

#include "copy_template.S"

ENDPROC(memcpy)
ENDPROC(__memcpy)
```

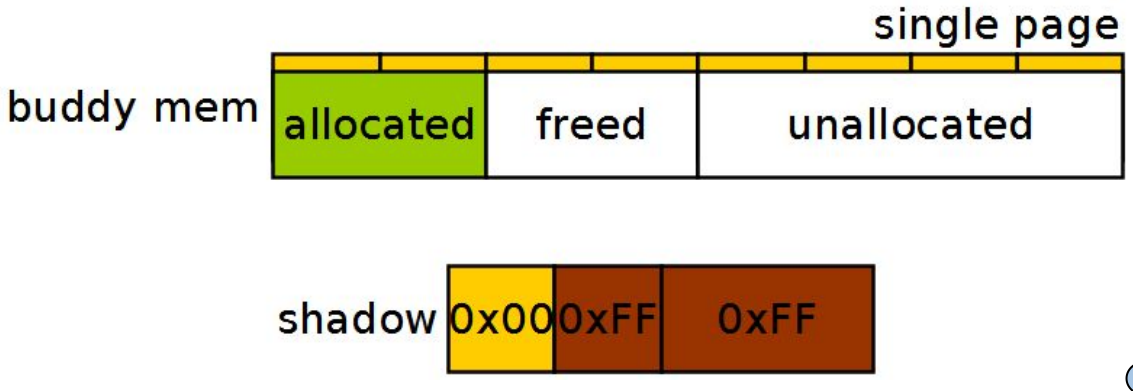
```
#undef memcpy
void *memcpy(void *dest, const void *src, size_t len)
{
        __asan_loadN((unsigned long)src, len);
        __asan_storeN((unsigned long)dest, len);

        return __memcpy(dest, src, len);
}
```

原理及实现

✦实现（ARM 32bits）（续4）

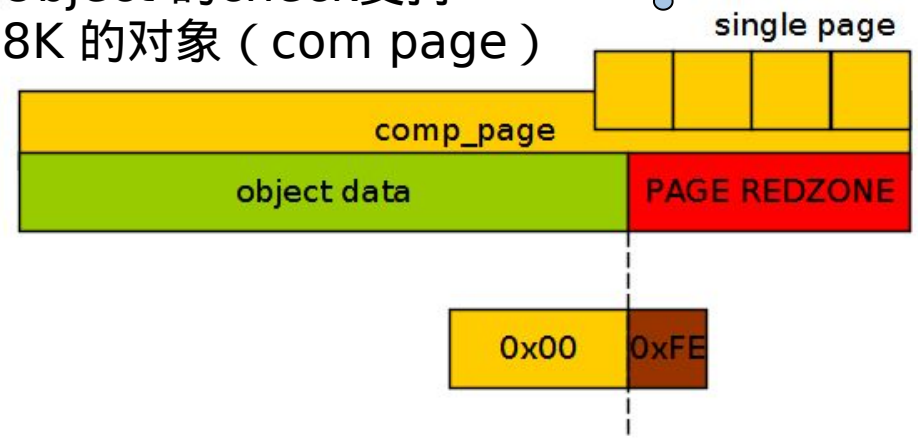
4. 对 Low Mem Page 的check支持



思考：
这里可能
存在漏检么？

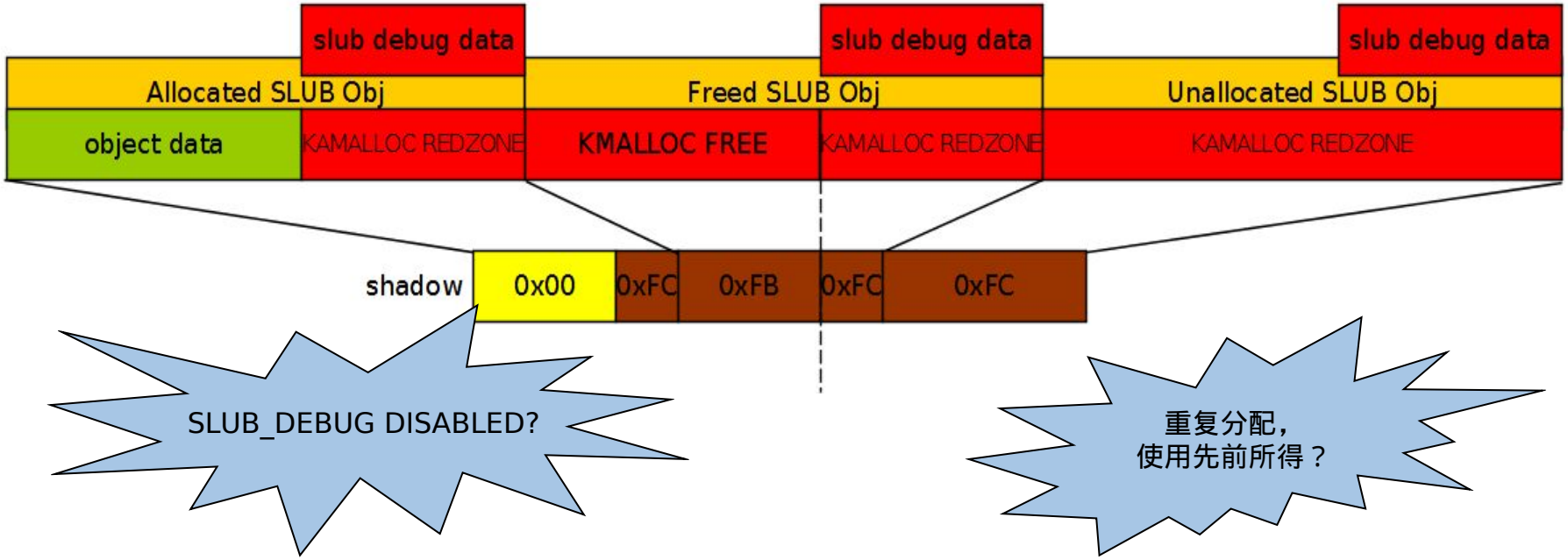
5. 对 SLUB Object 的check支持

a. 大于 8K 的对象（com page）



原理及实现

- 实现（ARM 32bits）（续5）
  - 5. 对 SLUB Object 的check支持(续)
    - b. 小于等于 8K 的对象



原理及实现

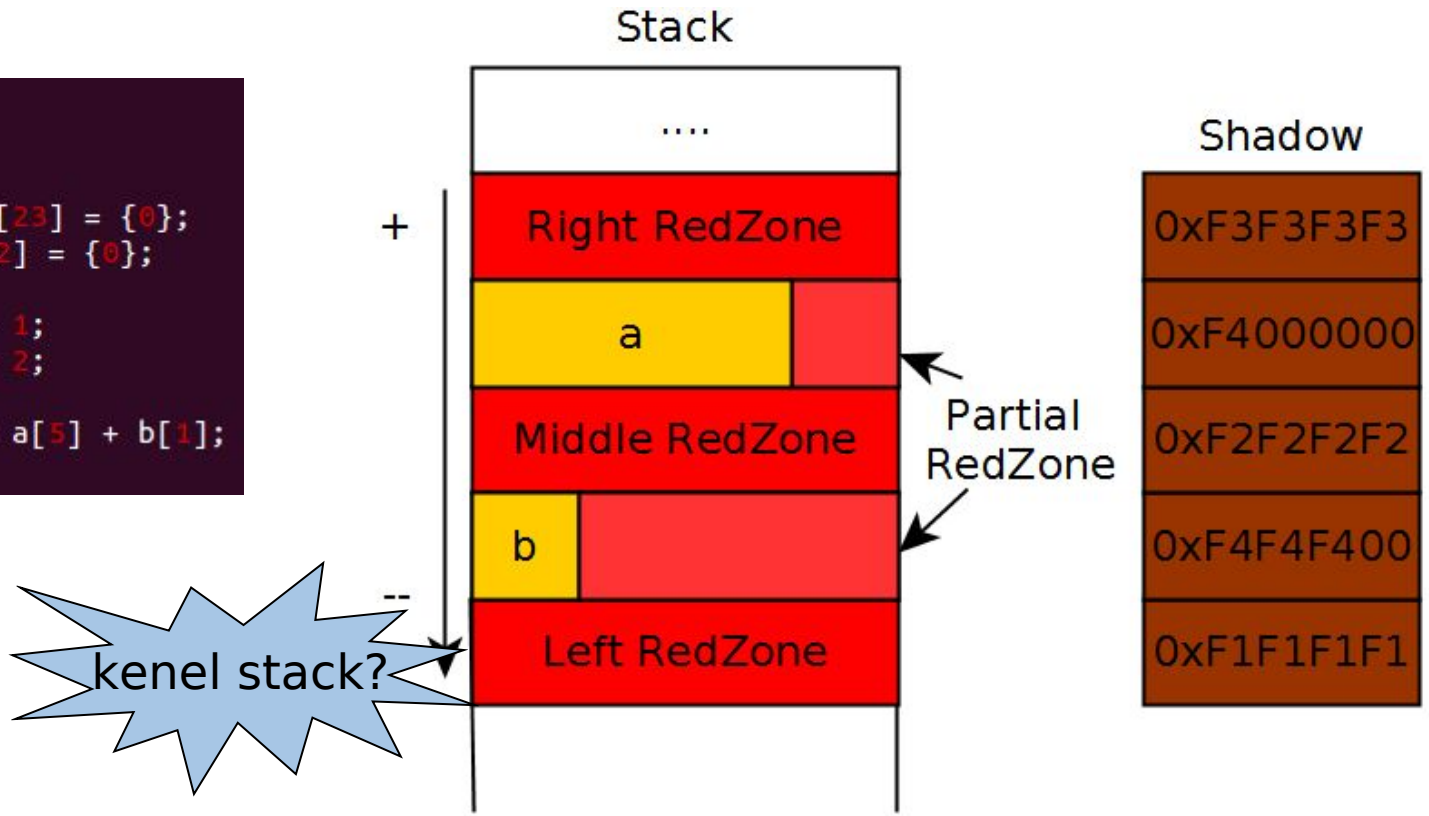
✦实现（ARM 32bits）（续6）

6. 对stack中变量的check支持

```
int
foo ()
{
  char a[23] = {0};
  int b[2] = {0};

  a[5] = 1;
  b[1] = 2;

  return a[5] + b[1];
}
```



原理及实现

- 实现（ARM 32bits）（续7）
  7. 对全局变量的check支持(!)
     1. 变量要求 32Bytes 对齐;
     2. 在变量中间插入 Redzone (0xFA)

```
void __asan_register_globals(struct kasan_global *globals, size_t size)
{
        int i;

        for (i = 0; i < size; i++)
                register_global(&globals[i]);
}
EXPORT_SYMBOL(__asan_register_globals);
```

```
/* Call all constructor functions linke
static void __init do_ctors(void)
{
#ifdef CONFIG_CONSTRUCTORS
        ctor_fn_t *fn = (ctor_fn_t *) __ctors_start;

        for (; fn < (ctor_fn_t *) __ctors_end; fn++)
                (*fn)();
#endif
}
```

```
arm_crash_gcc5> struct kasan_global c1071830 -x
struct kasan_global {
  beg = 0xc0a71040 <__func__.16354>,
  size = 0x11,
  size_with_redzone = 0x40,
  name = 0xc0d811b4,
  module_name = 0xc0da7f83,
  has_dynamic_init = 0x0,
  location = 0xc1071800
}
```

```
c097ee04 <_GLOBAL__sub_I_65535_1_km_flag>:
c097ee04:       e1a0c00d        mov     ip, sp
c097ee08:       e92dd800        push    {fp, ip, lr, pc}
c097ee0c:       e24cb004        sub     fp, ip, #4
c097ee10:       e3a01005        mov     r1, #5
c097ee14:       e59f0004        ldr     r0, [pc, #4]    ; c097ee20 <_GLOBAL__sub_I_65535_1_km_flag+0x1c>
c097ee18:       ebdf87f5        bl      c0160df4 <__asan_register_globals>
c097ee1c:       e89da800        ldm     sp, {fp, sp, pc}
c097ee20:       c1071830        .word   0xc1071830
```

原理及实现

- 实现（ARM 32bits）（续8）

8. 与其他内存调试工具的比较

Kasan 与其他工具的比较				
	kmemcheck	kasan	SLUB_DEBUG	DEBUG_PAGEALLOC
实现策略	Page Fault Dynamically	Compiler Instrumentation Statically / Redzone	Redzone / Poison / Use-tracing	Compound Page Poison / Unmap
功能	1, Uninited Read 2, UAF 3, OOB	1, UAF 2, OOB	1, UAF 2, OOB 3, User-tracing	1, UAF
速度/Overhead/缺点	最慢 最费内存	较慢 较费内存 不能检查某些 oob	较快 费较少内存 不能检查某些 bad reads 出错点和报告点 在时空上有脱节	较快 不费内存 不能检查细粒度错误

patch 整合

- 合适的 Toolchain 版本
  1. 最少要求 >= 4.9.2
  2. 若需支持以下功能，则要求 >= 5.0
     a. check OOB of stack/global vars
     b. inline instrumentation

Sabermod Toolchain / Outline

- Flash 中 boot 分区大小
  1. boot.img 大小
     a. ~21M (outline)
     b. ~32M (inline)

- U-boot 相关影响？

使用

- 配置
  1. 要求：
     CONFIG_KASAN=y
     CONFIG_KASAN_[OUTLINE|INLINE]
  2. 建议：
     CONFIG_STACKTRACE=y
- 命令行
  1. 最起码设置 slub_debug=U

  在 dts 中设置 bootargs

- 屏蔽 Kasan
  1. 单个文件：
     KASAN_SANITIZE_main.o := n
  2. 整个目录：
     KASAN_SANITIZE := n

使用

- 例子

```
static noinline void __init kmalloc_uaf(void)
{
        char *ptr;
        size_t size = 10;
        unsigned int a,b;

        pr_info("use-after-free\n");
        ptr = kmalloc(size, GFP_KERNEL);
        if (!ptr) {
                pr_err("Allocation failed\n"
                return;
        }

        kfree(ptr);
        *(ptr + 8) = 'x';
}
```

```
c0ed6a38 <kmalloc_uaf>:
c0ed6a38:       e1a0c00d        mov     ip, sp
c0ed6a3c:       e92dd830        push    {r4, r5, fp, ip, lr, pc}
c0ed6a40:       e24cb004        sub     fp, ip, #4
c0ed6a44:       e59f1050        ldr     r1, [pc, #80]   ; c0ed6a9c <kmalloc_uaf+0x64>
c0ed6a48:       e2810040        add     r0, r1, #64     ; 0x40
c0ed6a4c:       ebea3db8        bl      c0966134 <printk>
c0ed6a50:       e59f0048        ldr     r0, [pc, #72]   ; c0ed6aa0 <kmalloc_uaf+0x68>
c0ed6a54:       ebca2954        bl      c0160fac <__asan_load4>
c0ed6a58:       e59f3044        ldr     r3, [pc, #68]   ; c0ed6aa4 <kmalloc_uaf+0x6c>
c0ed6a5c:       e3a0200a        mov     r2, #10
c0ed6a60:       e3a010d0        mov     r1, #208        ; 0xd0
c0ed6a64:       e5930018        ldr     r0, [r3, #24]
c0ed6a68:       ebca1dc6        bl      c015e188 <kmem_cache_alloc_trace>
c0ed6a6c:       e2504000        subs    r4, r0, #0
c0ed6a70:       1a000003        bne     c0ed6a84 <kmalloc_uaf+0x4c>
c0ed6a74:       e59f1020        ldr     r1, [pc, #32]   ; c0ed6a9c <kmalloc_uaf+0x64>
c0ed6a78:       e28100a0        add     r0, r1, #160    ; 0xa0
c0ed6a7c:       ebea3dac        bl      c0966134 <printk>
c0ed6a80:       e89da830        ldm     sp, {r4, r5, fp, sp, pc}
c0ed6a84:       ebca221e        bl      c015f304 <kfree>
c0ed6a88:       e2840008        add     r0, r4, #8
c0ed6a8c:       ebca291c        bl      c0160f04 <__asan_store1>
c0ed6a90:       e3a03078        mov     r3, #120        ; 0x78
c0ed6a94:       e5c43008        strb    r3, [r4, #8]
c0ed6a98:       e89da830        ldm     sp, {r4, r5, fp, sp, pc}
c0ed6a9c:       c0a71080        .word   0xc0a71080
c0ed6aa0:       c12168a0        .word   0xc12168a0
c0ed6aa4:       c1216888        .word   0xc1216888
```

```
struct kmem_cache *kmalloc_caches[KMALLOC_SHIFT_HIGH + 1];
EXPORT_SYMBOL(kmalloc_caches);
```

```
[   14.654296] c0 kasan test: kmalloc_uaf use-after-free
[   14.654602] c0 ================================================================
[   14.654693] c0 BUG: KASan: use after free in kmalloc_uaf+0x58/0x70 at addr ecac2808
[   14.654785] c0 Write of size 1 by task swapper/0/1
[   14.654846] c0 ================================================================================
[   14.654937] c0 BUG kmalloc-64 (Not tainted): kasan: bad access detected
[   14.654968] c0 --------------------------------------------------------------------------------

[   14.655059] c0 Disabling lock debugging due to kernel taint
[   14.655181] c0 INFO: Allocated in kmalloc_uaf+0x34/0x70 age=0 cpu=0 pid=1
[   14.655303] c0       kmem_cache_alloc_trace+0x80/0x22c
[   14.655395] c0       kmalloc_uaf+0x34/0x70
[   14.655487] c0       kmalloc_tests_init+0x10/0x34
[   14.655578] c0       ...... //省略
[   14.655975] c0       ret_from_fork+0x14/0x20
[   14.656097] c0 INFO: Freed in kmalloc_uaf+0x50/0x70 age=0 cpu=
[   14.656188] c0       kfree+0x204/0x214
[   14.656280] c0       kmalloc_uaf+0x50/0x70
[   14.656372] c0       kmalloc_tests_init+0x10/0x34
[   14.656463] c0       ...... //省略
[   14.656860] c0       ret_from_fork+0x14/0x20
[   14.656921] c0 INFO: Slab 0xc192e840 objects=16 used=15 fp=0xecac2800 flags=0x0080
[   14.657012] c0 INFO: Object 0xecac2800 @offset=2048 fp=0x  (null)

[   14.657135] c0 Bytes b4 ecac27f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[   14.657257] c0 Object ecac2800: 00 00 00 00 6e 65 6c 2f 73 6c 61 62 2f 63 66 71  ....nel/slab/cfq
[   14.657379] c0 Object ecac2810: 5f 69 6f 5f 63 71 00 00 00 00 00 00 00 00 00 00  _io_cq..........
[   14.657470] c0 Object ecac2820: ...... //省略
[   14.657806] c0 Padding ecac28f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[   14.657928] c0 CPU: 0 PID: 1 Comm: swapper/0 Tainted: G    B        3.10.65-00021-g802eaee-dirty #55
[   14.658111] c0 [<c001ac34>] (unwind_backtrace+0x0/0x200) from [<c0016d1c>] (show_stack+0x20/0x24)
[   14.658264] c0 [<c0016d1c>] ...... //省略
[   14.658874] c0 [<c015cc58>] (object_err+0x3c/0x44) from [<c0161c28>] (kasan_report_error+0x1d4/0x2ac)
[   14.659057] c0 [<c0161c28>] (kasan_report_error+0x1d4/0x2ac) from [<c0161dfc>] (kasan_report+0x4c/0x54)
[   14.659210] c0 [<c0161dfc>] (kasan_report+0x4c/0x54) from [<c0160f38>] (__asan_store1+0x34/0x38)
[   14.659362] c0 [<c0160f38>] (__asan_store1+0x34/0x38) from [<c0ed6a90>] (kmalloc_uaf+0x58/0x70)
[   14.659515] c0 [<c0ed6a90>] (kmalloc_uaf+0x58/0x70) from [<c0ed6f9c>] (kmalloc_tests_init+0x10/0x34)
[   14.659698] c0 [<c0ed6f9c>] (kmalloc_tests_init+0x10/0x34) from [<c0009860>] (do_one_initcall_debug+0x80/0x114)
[   14.659881] c0 [<c0009860>] ...... //省略
[   14.660430] c0 [<c095d738>] (kernel_init+0x1c/0xf8) from [<c0011388>] (ret_from_fork+0x14/0x20)
[   14.660491] c0 Memory state around the buggy address:
[   14.660614] c0  ecac2700: 00 00 00 00 00 00 00 00 fc fc fc fc fc fc fc fc
[   14.660705] c0  ecac2780: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[   14.660797] c0 >ecac2800: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
[   14.660858] c0               ^
[   14.660949] c0  ecac2880: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[   14.661041] c0  ecac2900: 00 00 00 00 00 00 00 00 fc fc fc fc fc fc fc fc
[   14.661102] c0 ================================================================
```

```
arm_crash_gcc5> struct kmem_cache ec002140 -x | grep 'size\|name'
  size = 0x100,
  object_size = 0x40,
  name = 0xec001f00 "kmalloc-64",
    name = 0xece47500 "kmalloc-64",
```

参考

- kasan 介绍
  https://lwn.net/Articles/612153/
- https://github.com/torvalds/linux
- https://github.com/aryabinin/linux
- Porting to GCC 5.0
  https://gcc.gnu.org/gcc-5/porting_to.html
- http://www.sabermod.com