

SIXUENET

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(<http://www.mysixue.com/>)

MEMORY MANAGEMENT (5): MEMORY DEBUGGING

📅 April 20, 2019 ([Http://Www.Mysixue.Com/?P=119](http://Www.Mysixue.Com/?P=119)) 👤 JL
([Http://Www.Mysixue.Com/?Author=1](http://Www.Mysixue.Com/?Author=1)) 💬

5 Memory debugging

Many system stability issues are related to memory , especially memory cross-border access . This section introduces several kernel native memory debugging techniques

5.1 Page_Owner

5.1.1 Principle Introduction

The purpose of page_owner is to store the call stack information when the page is allocated , so that we can know who allocated each page .

To achieve this goal , you have to answer three questions : how to store, where to store, and when to store .

How to save

How to save means how to get the call stack information . This is easy to handle , and the kernel officially provides [save_stack_trace](http://lxr.free-electrons.com/ident?v=4.4;i=save_stack_trace) (http://lxr.free-electrons.com/ident?v=4.4;i=save_stack_trace) This API, we only need to call this function , you can get the call stack information .

Where to save

Where is the call stack information stored ? Since this information is related to each page , it should be reasonable to store it in the struct page structure . But the page structure is created during the memory initialization phase, and each physical page is allocated a page structure ; we only need to save the page call stack information has

been allocated for , if directly in the `page` structure which is the call stack information to allocate storage space , will increase the `page` 's size, and thus a waste of memory (refer back to `page` structure introduction known kernel in order to reduce the structure of this size done a lot of work) .

Thus kernel call stack information is stored in the **struct `page_ext`** (http://lxr.free-electrons.com/ident?v=4.4;i=page_ext) in , `page_ext` structure also each `page` corresponds to one , in addition to avoiding the wasted memory , `mm / page_ext.c` comment section also describes the introduction of other `page_ext` reasons . Struct `page` and struct `page_ext` There is no direct relationship , they are indexed to the data structure corresponding to each physical page through the page number .

When to save

Call stack information stored is of course the best time when a page assigned . `Page_owner`. H defines **`set_page_owner`** (http://lxr.free-electrons.com/ident?v=4.4;i=set_page_owner) function , this function is responsible for obtaining the call stack and store information in `page_ext` in ; and `page_alloc.c` in **`prep_new_page`** (http://lxr.free-electrons.com/ident?v=4.4;i=prep_new_page) `Set_page_owner` is called in the function.

more info

Currently Internet does not seem too much `page_owner` explained .

The official Documentation/vm/[page_owner.txt](http://lxr.free-electrons.com/source/Documentation/vm/page_owner.txt?v=4.4) (http://lxr.free-electrons.com/source/Documentation/vm/page_owner.txt?v=4.4) contains an introduction to how to enable & use the `page_owner` mechanism .

In addition, if you want to understand the code history , you can also use `git log mm/page_owner.c` to see which codes have been modified .

5.1.2 Example

In the example chapter, we intend to enable the `page_owner` function of the kernel , then write a kernel module to allocate a page, and then view the call stack information .

enable `page_owner`

Two conditions :

- Enable `CONFIG_PAGE_OWNER` when compiling
- Pass the parameter " `page_owner=on` " in uboot bootargs

Write kernel module sample code

https://gitlab.com/study-kernel/memory_management/tree/master/page_owner (https://gitlab.com/study-kernel/memory_management/tree/master/page_owner)

operation result

The following steps are completed on the board

- in `smud page_owner_test.ko`
- `cat /sys/kernel/debug/page_owner> page_owner_full.txt`

The following steps are done on the PC :

- `cd tools/vm`
- `make page_owner_sort`

- `grep -v ^PFN page_owner_full.txt> page_owner.txt`
- `./page_owner_sort page_owner.txt sorted_page_owner.txt`

Then check `sorted_page_owner.txt` , search for "`do_init_module`" in it , you can see the following information :

```
Page allocated via order 0, mask 0x24200c0
PFN 635287 Block 310 type 0 Flags
[<bf84803d>] 0xbf84803d
[<c0009713>] do_one_initcall+0x9b/0x198
[<c00fb215>] do_init_module+0x4d/0x310
[<c00a116b>] load_module+0x16eb/0x1b80
[<c00a17c7>] SyS_finit_module+0x77/0x9c
[<c000ed21>] ret_fast_syscall+0x1/0x52
[<ffffffff>] 0xffffffff
```

If you read the sample code written , you know that we have specially written an allocation function "`alloc_page_owner`" , hoping to see the function name in the call stack , but unfortunately only the address of `bf84803d` can be seen . The reason is that when getting the call stack information , **The `save_stack_trace`** function will determine whether the address is kernel_text_address (http://lxr.free-electrons.com/ident?v=4.4;i=kernel_text_address) , and if it is , it will find the function name corresponding to the address , so the function name in the kernel module cannot be displayed . (http://lxr.free-electrons.com/ident?v=4.4;i=kernel_text_address)

5.1.3 Section

The `page_owner` function can be compiled into the kernel , and then turned on by the `page_owner=on` switch when needed , which is more convenient .

`page_owner` only for `page` -level trace the call stack information , if a slab allocation , can not do anything .

In addition, through my own experiments , I found that this `page_owner` function does not seem to be very big , and I don't know how it helps to track and debug memory errors for the time being .

5.2 Kasan

5.2.1 Introduction

Kasan is the abbreviation of Kernel Address Sanitizer . It is a tool for dynamically detecting memory errors. Its main function is to check for problems such as memory out-of-bounds access and use of released memory . Kasan is integrated in the Linux kernel, released with the Linux kernel code, and maintained and developed by the kernel community

Kasan can be traced back to LLVM 's sanitizers project (<https://github.com/google/sanitizers>) . Andrey Ryabinin borrowed the idea of AddressSanitizer and implemented Kernel Address Sanitizer in the Linux kernel . So Kasan can also be seen as an Address Sanitizer for kernel space

5.2.2 Use

5.2.2.1 Prerequisites

GCC version

KASAN uses compile-time instrumentation for checking every memory access, therefore , the GCC version of the compiled kernel needs to be $\geq 4.9.2$

GCC 5.0 or later is required for detection of out-of-bounds accesses to stack or global variables.

Kernel version

Kasan is a part of the kernel and needs to be reconfigured, compiled and installed when used. Kasan in Linux kernel 4.0 was introduced when the kernel version, then only supports X86, at 4.4 introduced ARM64 support .

Kernel config

- To enable KASAN configure kernel with:
 - `CONFIG_KASAN = y`
- Choose one of `CONFIG_KASAN_OUTLINE` and `CONFIG_KASAN_INLINE`
 - Outline: produces smaller binary
 - INLINE: is 1.1 – 2 times faster, it requires a GCC version 5.0 or later
- Currently KASAN works only with the SLUB memory allocator
- For better bug detection and nicer reporting, enable `CONFIG_STACKTRACE`
- To disable instrumentation for specific files or directories, add a line similar to the following to the respective kernel Makefile:
 - For a single file (eg main.o):
`KASAN_SANITIZE_main.o := n`
 - For all files in one directory:
`KASAN_SANITIZE := n`

5.2.2.2 Test

Compile the kernel

According to the aforementioned conditions , configure, compile and start the kernel .

Compile the test program

The source code of the Linux kernel already contains the test code for Kasan , and its location is `linux/lib/test_kasan.c`

Compile it into ko, and when this module is loaded , the corresponding test function will be run .

Error report interpretation

`test_kasan.c` There are test functions for various situations , such as `kmalloc_oob_right` to simulate the cross-border memory: the application of the 123 bytes of space, but write access to the first 124 content bytes, the problem of cross-border access will result.


```
20 static noinline void __init kmalloc_oob_right(void)
21 {
22     char *ptr;
23     size_t size = 123;
24
25     pr_info("out-of-bounds to right\n");
26     ptr = kmalloc(size, GFP_KERNEL);
27     if (!ptr) {
28         pr_err("Allocation failed\n");
29         return;
30     }
31
32     ptr[size] = 'x';
33     kfree(ptr);
34 }
```

When running the above test code, the following content will be printed in detail in the kernel log:

Analysis tool : To simplify reading the reports you can use our symbolizer script :

<https://github.com/google/kasan/wiki#reports> (<https://github.com/google/kasan/wiki%23reports>)

The function of this tool is to convert the information in the form of "funcname+offset" in "Error reports" to "filename: line number". When using this tool , there are a few things to note

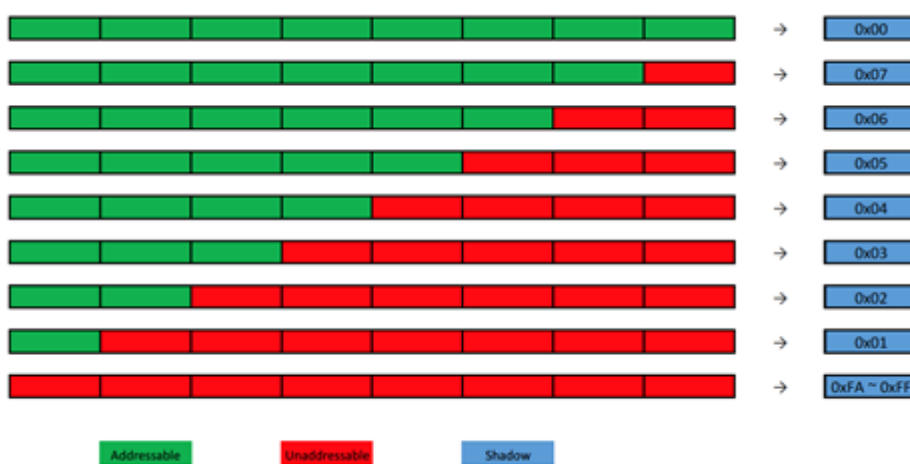
- The script using regular expressions to match each line , in order to obtain funcname and offset, so the output "Error reports" to conform to the script specification , concrete can read the script code and viewing of wiki examples log
-  script uses addr2line and nm, which need to use the version corresponding to the compiler . The script uses the addr2line and nm provided under the \$PATH path by default , which may not correspond .

5.2.3 Principle

5.2.3.1 Shadow area

Kasan 's principle is to use "extra" memory to mark the state of memory that can be used. The marking area is known as the shadow area (Shadow Region). Readers who understand Linux memory management know that each physical page in memory will have a structure like struct page in the memory to represent, that is, every 4KB page requires a 40B structure, and about 1% of the memory is used to represent the memory itself. Kasan is similar but "waste" is more serious, the ratio of the shadow area is 1:8 , that is, one-ninth of the total memory will be "waste".

The marking method is relatively simple. The available memory is grouped according to the size of 8 subsections. If all 8 bytes in each group are accessible, the corresponding place in the shadow memory is represented by all zeros (0x00); if the available memory is The first N (between 1 and 7) bytes are available, and the corresponding location in the shadow memory is represented by N ; in other cases, the shadow memory uses a negative number to indicate that the memory is not available, and the value range is 0xFA ~ 0xFF, and the specific meaning can be queried mm/kasan/kasan.h .



5.2.3.2 Address Translation

The so-called address translation refers to the conversion from the actually accessed memory address to the shadow area memory address . In the ARM64 architecture , in the memory virtual address space , a virtual address ($VA_START \sim (VA_START + KASAN_SHADOW_SIZE)$) is allocated to the shadow area . Of course , This virtual address will eventually allocate the corresponding physical page and establish the page table mapping .

The kernel code provides a function to do address conversion :

```
//include/linux/kasan.h

static inline void * kasan_mem_to_shadow ( const void * addr )
{
    return ( void * ) ( ( unsigned long ) addr >> KASAN_SHADOW_SCALE_SHIFT )
        + KASAN_SHADOW_OFFSET ;
}
```

The logic of the conversion is very simple , here $KASAN_SHADOW_SCALE_SHIFT = 3$, first set $addr \gg 3$, and then use the obtained value as an index to find the corresponding memory from the shadow area .

5.2.3.3 Initialization of the shadow area

Initialization function name shadow area is `kasan_init`. 4.4 of the kernel , `arch / arm64 / mm / kasan_init.c` and `arch / x86 / mm / kasan_init_64.c`. Read the initialization code , can be a general understanding of the process of initializing the shadow area . Overview of them also It is to allocate physical page frames (of course it is impossible to pass the partner system , at this time the entire memory system of the kernel has not been up , arm64 is through `memblock_region`) , and establish the page table mapping .

`git log -p arch/arm64/mm/kasan_init.c`, you can see the author's description of the initialization process in the submission history .

At early boot stage the whole shadow region populated with just one physical page (`kasan_zero_page`). Later, this page reused as readonly zero shadow for some memory that KASan currently don't track (`vmalloc`). After mapping the physical memory, pages for shadow memory are allocated and mapped.

After the shadow area initialization is complete , the next step is in the process of allocation and release of the kernel , to the shadow area is marked , mark which is occupied by the memory , which is free memory .

About the author `kasan` in a code commit message (<https://lwn.net/Articles/611410/> (<https://lwn.net/Articles/611410/>)) , we can see the author `mm / page_alloc.c` and `mm / slab_common.c`, `mm / slub.c` do modified , with `-p git log` to view history modify these codes , you can know the memory allocation and release of the API in , have joined the operation to the shadow area . this also shows that either directly by `page_alloc` allocation entire page , or by `slab` allocate small memory , `kasan` can handle .

5.2.3.4 How to detect illegal access

Compile-time instrumentation used for checking memory accesses. Compiler inserts function calls (`__asan_load*(addr)`, `__asan_store*(addr)`) before each memory access of size 1, 2, 4, 8 or 16. These functions check whether memory access is valid or not by checking corresponding shadow memory.

GCC 5.0 has possibility to perform inline instrumentation. Instead of making function calls GCC directly inserts the code to check the shadow memory. This option significantly enlarges kernel but it gives x1.1-x2 performance boost over outline instrumented kernel.

In addition, in the submission information of `git log -p arch/arm64/mm/kasan_init.c`, such a paragraph was also found :

Functions like `memset/memmove/memcpy` do a lot of memory accesses.

If bad pointer passed to one of these function it is important to catch this. Compiler's instrumentation cannot do this since these functions are written in assembly.

KASan replaces memory functions with manually instrumented variants.

Original functions declared as weak symbols so strong definitions in `mm/kasan/kasan.c` could replace them. Original functions have aliases with `'__'` prefix in name, so we could call non-instrumented variant if needed.

Some files built without kasan instrumentation (eg `mm/slub.c`).

Original `mem*` function replaced (via `#define`) with prefixed variantsto disable memory access checks for such files.

5.2.3.5 Historical information

Please refer to the author's original words : <https://lwn.net/Articles/611410/> (https://lwn.net/Articles/611410/) : Changes since v1:... (https://lwn.net/Articles/611410/)

5.2.3.6 Comparison of advantages and disadvantages

The following is excerpted from the author's original words : <https://lwn.net/Articles/611410/> (https://lwn.net/Articles/611410/) (https://lwn.net/Articles/611410/)

A lot of people asked about how kasan is different from other debuggin features, so here is a short comparison:

KMEMCHECK: – KASan can do almost everything that `kmemcheck` can. KASan uses compile-time instrumentation, which makes it significantly faster than `kmemcheck`. The only advantage of `kmemcheck` over KASan is detection of uninitialized memory reads.

DEBUG_PAGEALLOC: – KASan is slower than `DEBUG_PAGEALLOC`, but KASan works on sub-page granularity level, so it able to find more bugs

SLUB_DEBUG (poisoning, redzones): – `SLUB_DEBUG` has lower overhead than KASan.

– `SLUB_DEBUG` in most cases are not able to detect bad reads, KASan able to detect both reads and writes.

– In some cases (eg redzone overwritten) `SLUB_DEBUG` detect bugs only on allocation/freeing of object. KASan catch bugs right before it will happen, so we always know exact place of first bad read/write.

5.2.3.7 Reference link

https://www.ibm.com/developerworks/cn/linux/1608_tengr_kasan/index.html
(https://www.ibm.com/developerworks/cn/linux/1608_tengr_kasan/index.html)
<https://lwn.net/Articles/611410/> (<https://lwn.net/Articles/611410/>)
<http://lxr.free-electrons.com/source/Documentation/kasan.txt?v=4.4> (<http://lxr.free-electrons.com/source/Documentation/kasan.txt?v=4.4>)
<https://github.com/google/kasan/wiki> (<https://github.com/google/kasan/wiki>)

5.3 Asan

5.3.1 Basic principles

The principle of Asan is similar to Kasan and is divided into two parts :

- A RUN-Time Library, to replace the default malloc / free functions , so that to allocate and free memory when the shadow area marked
- Compiler , used to add detection code before memory access code .

But from the principle described by Kasan , it seems that it can only detect the anomaly of accessing unallocated memory .

For the detection of out-of-bounds memory , Asan adopts another method , which is to add some reserved memory before and after the detected memory , and poison these memories . In this way, it can be detected when an out-of-bounds access occurs .

For a detailed description of Asan's principles , refer to :
<https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>
(<https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>)
(<https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>)

5.3.2 Conditions of Use

LLVM >= 3.1 or GCC >= 4.8 or Android >= 4.2

5.3.3 Types of errors that Asan can detect

AddressSanitizer (aka ASan) is a memory error detector for C/C++. It finds:

- Use after free (<https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterFree>) (dangling pointer dereference)
- Heap buffer overflow (<https://github.com/google/sanitizers/wiki/AddressSanitizerExampleHeapOutOfBounds>)
- Stack buffer overflow (<https://github.com/google/sanitizers/wiki/AddressSanitizerExampleStackOutOfBounds>)

- [Global buffer overflow](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleGlobalOutOfBounds) (https://github.com/google/sanitizers/wiki/AddressSanitizerExampleGlobalOutOfBounds)
- [Use after return](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterReturn) (https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterReturn)
- [Use after scope](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterScope) (https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterScope)
- [Initialization order bugs](https://github.com/google/sanitizers/wiki/AddressSanitizerInitializationOrderFiasco) (https://github.com/google/sanitizers/wiki/AddressSanitizerInitializationOrderFiasco)
- [Memory leaks](https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer) (https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer)

5.3.4 Comparison of Asan with other similar tools

<https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools>
(https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools)

5.3.5 In Linux using the AddressSanitizer

- Clang: <https://github.com/google/sanitizers/wiki/AddressSanitizer#using-addresssanitizer>
(https://github.com/google/sanitizers/wiki/AddressSanitizer#using-addresssanitizer)
(https://github.com/google/sanitizers/wiki/AddressSanitizer#using-addresssanitizer)
- GCC: being good is not found link

5.3.6 In Android on the use of Asan

- Use Asan based on NDK : <https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid>
(https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid)
(https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid)
- Use Asan based on Android system : <https://source.android.com/devices/tech/debug/asan.html>
(https://source.android.com/devices/tech/debug/asan.html)
(https://source.android.com/devices/tech/debug/asan.html)

5.3.7 Adjust the call stack output by Asan

<https://github.com/google/sanitizers/wiki/AddressSanitizerCallStack>
(https://github.com/google/sanitizers/wiki/AddressSanitizerCallStack)