ICS Homework 8

Cao Shengcao

> 编写虚页页号遍历程序，该程序打印出本进程所能够访问的所有虚页页号。

除了下面将要详细介绍的方法之外，同学们提出的一些解决方案主要如下：

- 直接或间接读取系统提供的文件`/proc/$PID/maps`，从中获取虚拟地址映射的信息。我们将用它来作为验证。
- 试图利用该进程的`task_struct`中的信息。但是很遗憾这是由Linux系统内核维护的，目前没有找到什么可以访问它的办法。
- 在主进程中每次`fork()`一个子进程来试图访问一个虚拟页，通过子进程是否能够正常结束来判断能否访问。思路其实和下面的方法类似，这个方法是可行的，不足之处在于效率较低。

这里给出一种利用异常控制流的方法。我们每次尝试读取一个虚拟页中的内容，如果当前进程不能访问的话会触发段错误。但是我们可以为段错误设置异常处理函数，让它能够继续运行。需要注意的是如果处理函数是直接返回，还会回到触发异常的同一条语句。

如果考虑在处理函数中修改我们所访问的那个虚拟地址，会发现继续执行那条语句时虚拟地址并没有被改变。其原因要到汇编代码的层面才能理解，比如说要首先得到那个虚拟地址（无论是从内存中取还是计算得到），存放在寄存器`%rax`中，然后通过`movl (%rax) %edx`来试图访存，触发异常。在异常处理程序中，即使是修改了要访问的虚拟地址的值，也不能改变原来的`%rax`这个寄存器里的值，因此返回到该条语句之后仍然是试图访问同一个虚拟地址。

这里我们使用`sigsetjmp()` / `siglongjmp()`来完成一个特殊的跳转。具体如下：

```
#include "csapp.h"
typedef uint64_t va_t;                                  // 64-bit address space
const va_t va_start = 0x400000, va_end = 0x100000000;   // start and end of traversal
va_t va;
int test;
sigjmp_buf buf;
void handler(int sig)                                   // SIGSEGV handler, jump forward
{
    siglongjmp(buf, 2);
}
int main()
{
    switch (sigsetjmp(buf, 1))
    {
    case 0:                                             // first time, set handler and va
        Signal(SIGSEGV, handler);
        va = va_start;
    case 1:
        test = *(int*)va;                               // try to access the page
        printf("VPN: %llx VA: %llx-%llx\n",
               va >> 12, va, va + (1 << 12));           // if nothing goes wrong then print info
    case 2:
        va += 1 << 12;                                  // try next page
        if (va == va_end)
            break;
        siglongjmp(buf, 1);                             // jump back
    }
    printf("Finished!\n");
    printf("PID: %d\n", getpid());                      // print $PID and wait
    scanf("%d", &test);
    return 0;
}
```

程序运行的流程大概是这样的：每次通过`test = *(int*)va`试图访问一个虚拟页，如果成功的话就把这一页的信息打印出来。如果失败会触发段错误，会有信号`SIGSEGV`被我们所设置的处理函数处理。它会跳转到`case 2`，把打印的语句跳过去，然后无论是否访问成功都会更新虚拟地址，尝试下一页。

关于我们所需遍历的区域，书上已知的信息是可以直接从`0x400000`开始。我们知道在Intel Core i7 / 64-bit Linux环境下实际的虚拟地址空间是48位，高的16位要么全为0要么全为1。对于48位地址空间，存在2^36个虚拟页，要逐一尝试显然是不可接受的，即使是从`0x400000`开始。此外，不同的内存管理也会有不同的布局，似乎也并不存在什么标准能够告诉我们进程的哪一部分具体应该在哪些地址。因此，在上面的程序中只遍历了前32位地址空间。不过我们也可以把代码进行一些修改，并且用参数`-m32`来编译为32位的程序，这样是可以很快找到所有的虚拟页的。

上面程序的运行结果如下：

```
VPN: 400 VA: 400000-401000
VPN: 401 VA: 401000-402000
VPN: 402 VA: 402000-403000
VPN: 403 VA: 403000-404000
VPN: 404 VA: 404000-405000
VPN: 604 VA: 604000-605000
VPN: 605 VA: 605000-606000
VPN: 95f VA: 95f000-960000
VPN: 960 VA: 960000-961000
VPN: 961 VA: 961000-962000
VPN: 962 VA: 962000-963000
VPN: 963 VA: 963000-964000
VPN: 964 VA: 964000-965000
VPN: 965 VA: 965000-966000
VPN: 966 VA: 966000-967000
VPN: 967 VA: 967000-968000
VPN: 968 VA: 968000-969000
VPN: 969 VA: 969000-96a000
VPN: 96a VA: 96a000-96b000
VPN: 96b VA: 96b000-96c000
VPN: 96c VA: 96c000-96d000
VPN: 96d VA: 96d000-96e000
VPN: 96e VA: 96e000-96f000
VPN: 96f VA: 96f000-970000
VPN: 970 VA: 970000-971000
VPN: 971 VA: 971000-972000
VPN: 972 VA: 972000-973000
VPN: 973 VA: 973000-974000
VPN: 974 VA: 974000-975000
VPN: 975 VA: 975000-976000
VPN: 976 VA: 976000-977000
VPN: 977 VA: 977000-978000
VPN: 978 VA: 978000-979000
VPN: 979 VA: 979000-97a000
VPN: 97a VA: 97a000-97b000
VPN: 97b VA: 97b000-97c000
VPN: 97c VA: 97c000-97d000
VPN: 97d VA: 97d000-97e000
VPN: 97e VA: 97e000-97f000
VPN: 97f VA: 97f000-980000
Finished!
PID: 5021
```

退出之前，我们另开一个终端，用`cat /proc/5021/maps`看看系统提供的映射信息：

```
00400000-00405000 r-xp 00000000 08:01 271222
/home/friedrich/Documents/pre/va
00604000-00605000 r--p 00004000 08:01 271222
/home/friedrich/Documents/pre/va
00605000-00606000 rw-p 00005000 08:01 271222
/home/friedrich/Documents/pre/va
0095f000-00980000 rw-p 00000000 00:00 0                                  [heap]
7fd893c3d000-7fd893dfd000 r-xp 00000000 08:01 398781                     /lib/x86_64-linux-
gnu/libc-2.23.so
7fd893dfd000-7fd893ffd000 ---p 001c0000 08:01 398781                     /lib/x86_64-linux-
gnu/libc-2.23.so
7fd893ffd000-7fd894001000 r--p 001c0000 08:01 398781                     /lib/x86_64-linux-
gnu/libc-2.23.so
7fd894001000-7fd894003000 rw-p 001c4000 08:01 398781                     /lib/x86_64-linux-
gnu/libc-2.23.so
7fd894003000-7fd894007000 rw-p 00000000 00:00 0
7fd894007000-7fd89401f000 r-xp 00000000 08:01 398927                     /lib/x86_64-linux-
gnu/libpthread-2.23.so
7fd89401f000-7fd89421e000 ---p 00018000 08:01 398927                     /lib/x86_64-linux-
gnu/libpthread-2.23.so
7fd89421e000-7fd89421f000 r--p 00017000 08:01 398927                     /lib/x86_64-linux-
gnu/libpthread-2.23.so
7fd89421f000-7fd894220000 rw-p 00018000 08:01 398927                     /lib/x86_64-linux-
gnu/libpthread-2.23.so
7fd894220000-7fd894224000 rw-p 00000000 00:00 0
7fd894224000-7fd89424a000 r-xp 00000000 08:01 398753                     /lib/x86_64-linux-
gnu/ld-2.23.so
7fd894428000-7fd89442b000 rw-p 00000000 00:00 0
7fd894447000-7fd894449000 rw-p 00000000 00:00 0
7fd894449000-7fd89444a000 r--p 00025000 08:01 398753                     /lib/x86_64-linux-
gnu/ld-2.23.so
7fd89444a000-7fd89444b000 rw-p 00026000 08:01 398753                     /lib/x86_64-linux-
gnu/ld-2.23.so
7fd89444b000-7fd89444c000 rw-p 00000000 00:00 0
7fff7d585000-7fff7d5a6000 rw-p 00000000 00:00 0                          [stack]
7fff7d5ea000-7fff7d5ec000 r--p 00000000 00:00 0                          [vvar]
7fff7d5ec000-7fff7d5ee000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

最左一列是虚拟地址，然后是权限位，其中r代表readable，w代表writeable，x代表executable，p/s代表private/shared。至少可以确认在前32位的地址空间中我们的程序是正确的。我们也可以探究一下接下来的内容都是什么。

64位Linux对使用四级页表的内存布局如下：

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffffffffff (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1526 MB) module mapping space (variable)
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

高16位是根据第47位进行符号扩展得到的（如果是0则高16位为0000，如果是1则高16位为ffff），所以中间会有一个巨大的“洞”。洞以下为用户空间，其布局是由具体的内存管理器规定的，大概是这样：

```
Starts from 0x400000
ELF and Program and Section Headers
Program Text (.text)
Initialised Data (.data)
Uninitialised Data (.bss)
Heap
    |
    v
Memory Mapped Region for Shared Libraries or Anything Else
    ^
    |
User Stack
```

我们也可以对应到上面所看到的结果，在程序的堆和栈之间的确是一些共享库的映射。

理解了这些大概的布局之后，其实我们也可以想办法把所有需要的虚拟页都找出来。主要分为以下三个部分：

- 用户地址空间最下方，包括`.text` `.data`以及堆等，其位置可以用`main`函数的地址估计，该区域占用空间不会太大，前后各扫描8G就足够了。
- 用户地址空间最上方，包括栈、`vvar` `vdso`等，其位置可以用一个局部变量的地址估计，同样前后各扫描8G。
- 中间区域加载的动态链接库等，位置不定且范围较大。可以用GNU库函数`dl_iterate_phdr`找出它们的地址范围。

以下代码来自*@Jet Zhang*助教：

```c
#define _GNU_SOURCE  //For tricky dl_iterate_phdr

#include <link.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>

typedef unsigned long long va_t;

sigjmp_buf buf;
volatile char test;
const va_t scan_range = 0x100000000;
const size_t page_size = 1 << 12;

//SIGSEGV handler
void sigsegv_hdlr(int sig)
{
    siglongjmp(buf, 2);
}

//Callback function for dl_iterate_phdr
//Estimate the bound of mmap-ed areas
va_t mmap_low = -1;
va_t mmap_high = 0;

int callback(struct dl_phdr_info *info, size_t size, void *data)
{
    if (strcmp("", info->dlpi_name) == 0)
        return 0;

    va_t max_offset = 0;
    for (int i = 0; i < info->dlpi_phnum; i++)
    {
        va_t v_offset = info->dlpi_phdr[i].p_vaddr;
        if (v_offset > max_offset)
            max_offset = v_offset;
    }
    if (mmap_low > info->dlpi_addr)
        mmap_low = info->dlpi_addr;
    if (mmap_high < info->dlpi_addr + max_offset)
        mmap_high = info->dlpi_addr + max_offset;

    // If you need detail segment info:
    /*
    //    printf("filename: %s, %d segments\n", info->dlpi_name,
    //           info->dlpi_phnum);
    //
    //    for (int i = 0; i < info->dlpi_phnum; i++)
    //        printf("    header %2d: address=%p\n", i,
    //               (void *) (info->dlpi_addr + info->dlpi_phdr[i].p_vaddr));
```

```
     */
    return 0;
}

//Scan existing pages from start to end
//Return value: page count
int scan(va_t start, va_t end)
{
    volatile int cnt = 0;
    volatile va_t va = start;
    switch (sigsetjmp(buf, 1))
    {
        case 0:
        case 1:                         //Case 1: unvisited page, try visiting
            test = *(char *) va;        //If nothing went wrong, print info
            printf("VPN: %llx VA: %llx-%llx\n", va >> 12, va, va + page_size);
            cnt++;
        case 2:                         //Case 2: longjmp back from SIGSEGV
            va += page_size;            //then try next page
            if (va >= end)
                break;
            siglongjmp(buf, 1);         //Goto case 1
    }

    printf("Finish Scanning %llx-%llx, %d Pages Scanned\n\n", start, end, cnt);
    return cnt;
}

int main()
{
    signal(SIGBUS, SIG_IGN);
    signal(SIGSEGV, sigsegv_hdlr);

    char cmd[100];
    int page_cnt = 0;

    //Scan lower pages (i.e. .text, .data etc.)
    //We use main to locate the segments,
    // in case of possible text-shifting (on Ubuntu 17.04)
    printf("Scanning lower pages:\n");
    va_t text_scan_start = (va_t) &main > scan_range ?
                           ((va_t) &main - scan_range) : 0;
    page_cnt += scan(text_scan_start, text_scan_start + scan_range * 2);

    //Scan higher pages (i.e. stack, vvar, vdso etc.)
    //We use local variable to locate the higher segments
    printf("Scanning higher pages:\n");
    va_t stack_scan_start = ((va_t) &cmd / page_size) * page_size - scan_range;
    page_cnt += scan(stack_scan_start, stack_scan_start + scan_range * 2);

    //We use dl_iterate_phdr to iterate loaded objects,
    //so we can estimate the range of mmap-ed areas

    printf("Scanning middle pages (mmap-ed segments):\n");
```

```
    dl_iterate_phdr(callback, NULL);
    usleep(2000000);
    page_cnt += scan((mmap_low / page_size) * page_size - scan_range,
                     (mmap_high / page_size) * page_size + scan_range);

    printf("\nFinish Scanning, %d pages scanned\n", page_cnt);

    //Print /proc/pid/maps as our baseline
    printf("\n\n---------------Results From maps (baseline)---------------\n");
    sprintf(cmd, "cat /proc/%d/maps", getpid());
    system(cmd);

    return 0;
}
```

关于这部分的更详细的介绍，可以阅读书上9.7的内容以及以下参考资料：

- https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4
- https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.txt
- http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
- https://codeyarns.com/2015/04/27/memory-layout-of-a-process-in-linux/