VE482 Homework 6

Liu Yihao 515370910207

Ex. 1 — Simple questions

1. First fit:

   (i) 20 KB
   (ii) 10 KB
   (iii) 18 KB

   Best fit:

   (i) 12 KB
   (ii) 10 KB
   (iii) 9 KB

   Quick fit:

   (i) 12 KB
   (ii) 10 KB
   (iii) 9 KB

2. $$\frac{10k}{10k+n}$$

3. Counter 0: 01101110
   Counter 1: 01001001
   Counter 2: 00110111
   Counter 3: 10001011

Ex. 2 — Page tables

- inverted page tables

  The inverted page table (IPT) is best thought of as an off-chip extension of the TLB which uses normal system RAM. Unlike a true page table, it is not necessarily able to hold all current mappings. The OS must be prepared to handle misses, just as it would with a MIPS-style software-filled TLB.

  The IPT combines a page table and a frame table into one data structure. At its core is a fixed-size table with the number of rows equal to the number of frames in memory. If there are 4000 frames, the inverted page table has 4000 rows. For each row there is an entry for the virtual

page number (VPN), the physical page number (not the physical address), some other data and a means for creating a collision chain, as we will see later.

To search through all entries of the core IPT structure is inefficient, and a hash table may be used to map virtual addresses (and address space/PID information if need be) to an index in the IPT - this is where the collision chain is used. This hash table is known as a hash anchor table. The hashing function is not generally optimized for coverage - raw speed is more desirable. Of course, hash tables experience collisions. Due to this chosen hashing function, we may experience a lot of collisions in usage, so for each entry in the table the VPN is provided to check if it is the searched entry or a collision.

In searching for a mapping, the hash anchor table is used. If no entry exists, a page fault occurs. Otherwise, the entry is found. Depending on the architecture, the entry may be placed in the TLB again and the memory reference is restarted, or the collision chain may be followed until it has been exhausted and a page fault occurs.

A virtual address in this schema could be split into two, the first half being a virtual page number and the second half being the offset in that page.

A major problem with this design is poor cache locality caused by the hash function. Tree-based designs avoid this by placing the page table entries for adjacent pages in adjacent locations, but an inverted page table destroys spatial locality of reference by scattering entries all over. An operating system may minimize the size of the hash table to reduce this problem, with the trade-off being an increased miss rate. There is normally one hash table, contiguous in physical memory, shared by all processes. Memory fragmentation makes per-process page tables impractical, so a per-process identifier is used to disambiguate the pages of different processes from each other. It is somewhat slow to remove the page table entries of a process; the OS may avoid reusing per-process identifier values to delay facing this or it may elect to suffer the huge waste of memory associated with pre-allocated (necessary because of fragmentation) per-process hash tables.

- multilevel page tables

  The inverted page table keeps a listing of mappings installed for all frames in physical memory. However, this could be quite wasteful. Instead of doing so, we could create a page table structure that contains mappings for virtual pages. It is done by keeping several page tables that cover a certain block of virtual memory. For example, we can create smaller 1024-entry 4K pages that cover 4M of virtual memory.

  This is useful since often the top-most parts and bottom-most parts of virtual memory are used in running a process - the top is often used for text and data segments while the bottom for stack, with free memory in between. The multilevel page table may keep a few of the smaller page tables to cover just the top and bottom parts of memory and create new ones only when strictly necessary.

  Now, each of these smaller page tables are linked together by a master page table, effectively creating a tree data structure. There need not be only two levels, but possibly multiple ones.

  A virtual address in this schema could be split into three parts: the index in the root page table, the index in the sub-page table, and the offset in that page.

  Multilevel page tables are also referred to as hierarchical page tables.

Reference: https://en.wikipedia.org/wiki/Page_table

Ex. 3 — Research

The buffer overflow attack was discovered in hacking circles. It uses input to a poorly implemented, but (in intention) completely harmless application, typically with root / administrator privileges. The buffer overflow attack results from input that is longer than the implementor intended. To understand its inner workings, we need to talk a little bit about how computers use memory.

The stack is a region in a program's memory space that is only accessible from the top. There are two operations, push and pop, to a stack. A push stores a new data item on top of the stack, a pop removes the top item. Every process has its own memory space (at least in a decent OS), among them a stack region and a heap region. The stack is used heavily to store local variables and the return address of a function.

We look at the following example program, taken from Howard and LeBlanc:

```
1  /*
2    StackOverrun.c
3    This program shows an example of how a stack-based
4    buffer overrun can be used to execute arbitrary code.  Its
5    objective is to find an input string that executes the function bar.
6  */
7
8  #pragma check_stack(off)
9
10 #include <string.h>
11 #include <stdio.h>
12
13 void bar(void)
14 {
15     printf("Augh! I've been hacked!\n");
16 }
17
18 void foo(const char* input)
19 {
20     char buf[10];
21
22     printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
23     fflush(stdout);
24
25     strcpy(buf, input);
26     //printf("%s\n", buf);
27     //fflush(stdout);
28
29     printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
30     fflush(stdout);
31 }
32
33 int main(int argc, char* argv[])
34 {
35     //Blatant cheating to make life easier on myself
36     printf("Address of foo = %p\n", foo);
37     printf("Address of bar = %p\n", bar);
```

```
38      if (argc != 2)
39      {
40          printf("Please supply a string as an argument!\n");
41          return -1;
42      }
43      foo(argv[1]);
44      return 0;
45  }
```

Now if we compile it with

```
1  gcc -m32 -o ex3 ex3.c
```

And run it with the argument "Hello" we can get some result (for example)

```
1  Address of foo = 0x565fa69b
2  Address of bar = 0x565fa670
3  My stack looks like:
4  0xffe82280
5  0xf760d811
6  0x565fa6a7
7  0xffe822b0
8  0xf7781000
9  0xffe82298
10 0xffe83655
11 0xf7781d60
12 0x565fa908
13
14 Now the stack looks like:
15 0xffe83655
16 0xf760d811
17 0x565fa6a7
18 0xffe822b0
19 0xf7781000
20 0xffe82298
21 0xffe83655
22 0x65481d60
23 0x6f6c6c
```

It seems like "0x565fa6a7" is the return address of function foo, and if we can construct some argument so that it is altered as "0x565fa670", the function bar will be called, and we can do something which cause some secure problem in that.

Fortunately, in the new gcc versions (4.x+), the compiler will protect the stack with GCC Stack-Smashing Protector, which can add a guard variable to functions with vulnerable objects. The guard variable will be checked before return the function. It is effective because the attacker don't know the value of it, the variable will be altered before the return address is altered.

Reference:
https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#Instrumentation-Options
http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html

Ex. 4 — Minix 3

1. The files are in “minix/servers/vm”, such as

```
minix/servers/vm/vm.h
minix/servers/vm/proto.h
minix/servers/vm/pt.h
minix/servers/vm/arch/i386/pagetable.h
minix/servers/vm/pagetable.c
```

2. We can find

```
#define I386_PAGE_SIZE^^I^^I4096
#define ARM_PAGE_SIZE^^I^^I4096 /* small page on ARM  */
```

   So on both i386 and arm architecture, the page table size is 4096.

3.
```
/* A pagetable. */
typedef struct {
    /* Directory entries in VM addr space - root of page table.  */
    u32_t *pt_dir;      /* page aligned (ARCH_VM_DIR_ENTRIES) */
    u32_t pt_dir_phys;  /* physical address of pt_dir */

    /* Pointers to page tables in VM address space. */
    u32_t *pt_pt[ARCH_VM_DIR_ENTRIES];

    /* When looking for a hole in virtual address space, start
     * looking here. This is in linear addresses, i.e.,
     * not as the process sees it but the position in the page
     * page table. This is just a hint.
     */
    u32_t pt_virtop;
} pt_t;
```

4.
```
void pt_init(void);
void vm_freepages(vir_bytes vir, int pages);
void pt_init_mem(void);
void pt_check(struct vmproc *vmp);
int pt_new(pt_t *pt);
void pt_free(pt_t *pt);
int pt_map_in_range(struct vmproc *src_vmp, struct vmproc *dst_vmp,
 ↪  vir_bytes start, vir_bytes end);
int pt_ptmap(struct vmproc *src_vmp, struct vmproc *dst_vmp);
int pt_ptalloc_in_range(pt_t *pt, vir_bytes start, vir_bytes end, u32_t
 ↪  flags, int verify);
void pt_clearmapcache(void);
int pt_writemap(struct vmproc * vmp, pt_t *pt, vir_bytes v, phys_bytes
 ↪  physaddr, size_t bytes, u32_t flags, u32_t writemapflags);
int pt_checkrange(pt_t *pt, vir_bytes v, size_t bytes, int write);
int pt_bind(pt_t *pt, struct vmproc *who);
void *vm_mappages(phys_bytes p, int pages);
void *vm_allocpage(phys_bytes *p, int cat);
```

```
16  void *vm_allocpages(phys_bytes *p, int cat, int pages);
17  void *vm_allocpagedir(phys_bytes *p);
18  int pt_mapkernel(pt_t *pt);
19  void vm_pagelock(void *vir, int lockflag);
20  int vm_addrok(void *vir, int write);
21  int get_vm_self_pages(void);
22  int pt_writable(struct vmproc *vmp, vir_bytes v);
```

Ex. 5 — Linux

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include <time.h>
4
5  #define SIZE SHRT_MAX
6
7  int main() {
8      static int arr1[2048];
9      double start = clock();
10     for (int i = 0; i < SIZE * SIZE / 2048; ++i) {
11         for (int j = 0; j < 2048; ++j) {
12             arr1[j] = i * j;
13         }
14     }
15     double end = clock();
16     printf("No thrashing: %lfs\n", (end - start) / CLOCKS_PER_SEC );
17
18     static int arr2[SIZE][SIZE];
19     start = clock();
20     for (int i = 0; i < SIZE; ++i) {
21         for (int j = 0; j < SIZE; ++j) {
22             arr2[i][j] = i * j;
23         }
24     }
25     end = clock();
26     printf("Thrashing: %lfs\n", (end - start) / CLOCKS_PER_SEC );
27     return 0;
28 }
```

Ex. 6 — Dirty COW

Dirty COW (Dirty copy-on-write) is a computer security vulnerability for the Linux kernel that affects all Linux-based operating systems including Android. It is a local privilege escalation bug that exploits a race condition in the implementation of the copy-on-write mechanism in the kernel's memory-management subsystem. The vulnerability was discovered by Phil Oester. Because of the race condition, with the right timing, a local attacker can exploit the copy-on-write mechanism to turn a read-only mapping of a file into a writable mapping. Although it is a local privilege escalation, remote attackers can use it in conjunction with other exploits that allow remote execution of non-privileged

code to achieve remote root access on a computer. The attack itself does not leave traces in the system log.

The Dirty COW vulnerability has many perceived use cases including proven examples, such as obtaining root permissions in Android devices, as well as several speculated implementations. There are many binaries used in linux which are read-only, and can only be modified or written to by a user of higher permissions, such as the root. When privileges are escalated, whether by genuine or ingenuine means – such as by using the Dirty COW exploit – the user can modify usually unmodifiable binaries and files. If a malicious individual could use the Dirty COW vulnerability to escalate their permissions, they could change a file, such as /bin/bash, so that it performs an additional, unexpected functions, such as a keylogger. When a user starts a program which has been infected, they will inadvertently allow the malicious code to run. If the exploit targets a program which is run with root privileges, the exploit will enjoy those same privileges.

Reference: https://en.wikipedia.org/wiki/Dirty_COW