P5: xv6 Memory Encryption

Due Tuesday by 11:59pm **Points** 100

Updates:

- Changed `dump_rawphymem`'s physical_addr parameter from `char *` to `uint`
- Added Makefile instructions
- Corrected ppage example (should not have zeroes for lower order 12 bits)

Objectives

- To learn about the virtual memory system in xv6
- To understand page table entries in detail
- To modify page table entries to be able to detect the current state of a page
- To modify the trap handler to be able to handle the page fault

Background

The page tables, traps, and interrupts of xv6 are described in Chapters 2 and 3 of the **xv6 book. (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)**

You can work on this project with one other partner. P5 is due Tuesday, April 6th, 11:59PM CST.

You will be starting from unmodified xv6 code. You can find a copy of xv6 source code in `~cs537-1/projects/xv6.tar.gz`. Copy it into your private directory and untar it.

```
prompt> cp ~cs537-1/projects/xv6.tar.gz /path/to/your/private/dir
prompt> tar -xvzf xv6.tar.gz
```

As usual, set CPUS to 1 and change the compiler flag from O2 to Og in you Makefile.

Main Idea

As you know, the abstraction of an **address space** per process is a great way to provide **isolation** (i.e., protection) across processes. However, even with this abstraction, attackers can still try to modify or access portions of memory that they do not have permission for. For example, in a Rowhammer attack, a malicious process can repeatedly write to certain addresses in order to cause bit flips in DRAM in the nearby (physical) addresses that the process does not have permission to write directly.

If pages are left in clear text in DRAM, it may be possible for a clever malicious process to read those pages (e.g., by using a Rowhammer attack to modify the page tables or by leveraging physical access to the hardware). Therefore, for extra security, a running process may wish for its pages to be stored in an encrypted format in memory -- as long as those pages are not being accessed frequently.

To simplify this project, you will approximate the benefits of encryption by **flipping all bits in a page** (i.e., xor every bit with 1 or just use ~); actually performing encryption would require more computation and recording a corresponding key, but those are not relevant to this project.

User-level Memory Encryption

Page Encryption

We will push the decision to encrypt a virtual page to the user: the user process will be able to encrypt a range of virtual pages with the new system call

int mencrypt(char *virtual_addr, int len)

The virtual page associated with the parameter **virtual_addr** will be the starting virtual page. The parameter **len** specifies how many pages will be encrypted. You should **not assume virtual_addr is always page-aligned**. A successful call to **mencrypt** will encrypt the virtual addresses ranging from [PGROUNDDOWN(virtual_addr), PGROUNDDOWN(virtual_addr) + len * PGSIZE) and returns 0. For instance, suppose the page size is 4KB, a successful call to mencrypt(0x3000, 2) will encrypt the virtual addresses in the range [0x3000, 0x5000). A call to mencrypt(0x3050, 2) will do the same.

Your implementation of **mencrypt(char *virtual_addr, int len)** should execute successfully in the following cases and return 0:

1. When the parameter **len** equals **0,** your implementation should do nothing. This should happen before doing any error checking.

2. When part of or all the pages in the range have already been encrypted. **Encrypted pages and their corresponding page table entries should remain unchanged.** All the unencrypted pages should be encrypted and the function should return 0.

Your implementation should return -1 in the following cases:

1. The calling process does not have permission or privilege to access or modify some pages in the range. Your implementation should return -1 without encrypting any page in this case. To be specific, either all the pages in the range are successfully encrypted or none of them is encrypted. You can use uva2ka() for this.

2. The parameter **virtual_addr** points to an invalid address (e.g., out-of-range value). Depending on your implementation, this might be the same as case 1.

3. The parameter **len** is a negative value or a very large value that will let the page range exceed the upper bound of the user virtual space.

Implementation Hints:

In order to encrypt the page residing in the physical memory, you will need to access and modify the physical memory from the kernel. For doing this, you might want to understand the layout of the virtual address space of a process. Specifically, xv6 places the kernel in the virtual address space of each process from KERNBASE to KERNBASE + PHYSTOP; these addresses are mapped in physical memory from 0 to PHYSTOP. In other words, virtual address KERNBASE + **pa** is mapped to physical address **pa**. You might find the macro and constants defined in the **memlayout.h** can help you to do the translation between virtual and physical address. A good reference for the xv6 memory layout is Figure 2-2 (Page 31) from the xv6 reference **book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)**.

You might want to look through the macros defined in **mmu.h** and **memlayout.h** in detail. For instance, PGROUNDDOWN and PGROUNDUP can help you round down and up the virtual address to the nearest page-aligned address. Note that you calculate a particular virtual address's page number by rounding down.

In addition, understanding the functions defined in **vm.c** (e.g., **walkpgdir(), uva2ka(), and copyout()**) might be helpful because you may need to either use or modify those routines, or implement similar functionality. **uva2ka()** can help with error checking. You can pass a virtual address to this function and it will return a null pointer or pointer to 0 if there is an error. Just remember to modify uva2ka to handle the PTE_P bit being clear for encrypted pages.

One more detail:

The TLB caches virtual address translations. If you edit the page table in memory, the CPU has no way of knowing that the page table in memory has changed. Unless you flush the TLB, it's possible that the CPU will not see those changes immediately. So make sure to **flush the TLB after modifying the page table.** One way to do this is to overwrite the `CR3` register (page table base register) even with the same value by calling switchuvm() to the same process, and thus flush the TLB.

Page Decryption

An encrypted page must be decrypted whenever it is accessed. To implement this functionality, the kernel must catch a user's access to any encrypted page. To do this, the first step is modifying the kernel to use one bit in page table entries (PTEs) to record whether or not the corresponding page is currently encrypted (fortunately, there are plenty of unused bits in the current PTEs). Maybe call this new bit **PTE_E**. Set this bit to 1 in the PTEs when the corresponding pages are encrypted.

The second step is to make your xv6 kernel get control when a user tries to access an encrypted page. Remember, xv6 is running on emulated x86 hardware, and every time a user process tries to access some virtual address, the hardware walks the page tables to find the PTE and grab the corresponding physical address translation. The OS usually isn't involved when doing the address translation -- except when there is a page fault (i.e., PTE_P bit is not set). So, the trick is, **clear the PTE_P bit when you set**

the PTE_E bit. Then, when a user process tries to access this page, a page fault will be triggered (one of the "default" case not currently handled in **trap.c** by xv6 that you can refine with the definitions in **traps.h**), you can look at the faulting address; if the faulting address occurred for a page where PTE_P was cleared and PTE_E was set, you need to decrypt the page, reset the appropriate bits in the PTE, and return from the trap. You should **not assume the virtual address that causes the page fault is always page-aligned**.

Note that after a page has been decrypted, it will stay decrypted until being encrypted again. Repeated accesses to that (previously encrypted) page should not cause additional page faults into xv6; after a page has been decrypted, the hardware should be able to walk the page tables without involving the OS. In addition, when a child process is created, its initial memory state (including whether a page is encrypted or not), should match that of its parent.

Keep in mind that page faults and memory errors, in general, are still possible - not all traps that are marked as page faults are necessarily decryption requests. In these cases, you should just silently exit() from trap.c. You can use the function rcr2() to get the virtual address in question and pass it to an internal decryption function, which can then determine whether the page in question was valid and encrypted or simply invalid.

Implementation Hints:

To implement this, you might want to start by looking through the information in **mmu.h** in detail. For instance, you will see the format for 32-bit virtual addresses (defined by the x86 architecture): 10 bits for the page directory index, 10 bits for the inner page table index, and then 12 bits for the offset within a page. Next in **mmu.h,** you will see the format of a **PTE** itself (again defined by the x86 architecture). From the macro PTE_ADDR, you can see that the upper 20 bits designate the address (the physical page) stored in the PTE; from PTE_FLAGS, you can see that the lower 12 bits designate the flags in the PTE. From PTE_P, PTE_W, and PTE_U you can see that the 3 least-significant bits record whether or not the corresponding page is **present** (which we would have called "valid"), **writable** , and part of **user** address space. Pick an unused bit amongst the flags and consider that as your PTE_E bit. You might find Figure 2-1 (Page 30) in the xv6 reference **book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** is helpful for you to identify which bit is unused.

You might also want to learn how to set and clear certain bits in the PTE by reading through the code of other functions (e.g., **mappages() or clearpteu()**) defined in **vm.c**.

Other Parts of the Kernel

Remember that trick where we clear the PTE_P bit on an encrypted page table entry to trigger a page fault when it's accessed? That PTE_P bit is used by other parts of xv6 for various reasons. Originally, PTE_P equals 0 meant that this page is not present or is otherwise invalid. But now if PTE_P equals 0, that page could be an encrypted page, which is a valid, in-use page. Therefore, we need to change some of the original code that does the following kind of check:

```
if (*pte & PTE_P) // Check whether this pde is valid or in-used.
```

Hint: Specifically, you might want to check uva2ka(), copyuvm(), deallocuvm(), freevm() and mappages() in **vm.c**. For example, copyuvm() is called when fork() creates a child process. That child process should inherit the encrypted pages, so you'll want to setup the child's page table appropriately.

Statistics

In order to gather statistics about your memory system and test your implementation. You need to implement the following two syscalls:

int getpgtable(struct pt_entry* entries, int num)

will allow the user to gather information about the state of the page table. The parameter **entries** is an array of pt_entries with **num** elements that should be allocated by the user application and filled up by the kernel.
The header file which defines **struct pt_entry** should be copied from `~cs537-1/projects/memory/ptentry.h`.
Do not edit `ptentry.h`.

```
struct pt_entry {
    uint pdx : 10; // page directory index of the virtual page (see PDX macro defined in mmu.h)
    uint ptx : 10; // page table index of the virtual page (see PTX macro defined in mmu.h)
    uint ppage : 20; // physical page number
    uint present : 1; // Set to 1 if PTE_P == 1, otherwise 0. THIS IS A SIMPLE DUMP OF THE PTE_P BIT!
    uint writable : 1; // Set to 1 if PTE_W == 1, otherwise 0
    uint encrypted : 1; // Set to 1 if this page is currently encrypted, otherwise 0
};
```

Note that struct pt_entry uses bitfields to conserve space. Those fields that have a ': 1' next to them have a size of 1 bit. Attempting to set a value greater than 1 will cause an **overflow error**.
The kernel should fill up the entries array using the information from the page table of the currently running process. Only valid (allocated) virtual pages belong to the user will be considered. When the actual number of valid virtual pages is greater than the **num**, filling up the array starts from the allocated virtual page with the highest page numbers and returns **num** in this case. You might find **sz** field in the proc structure of each process is useful to identify the most top user page. For instance, if one process has allocated 10 virtual pages with page numbers ranging from 0x0 - 0x9 and page 0x9 is encrypted, then page 0x9 - 0x7 should be used to fill up the array when **num** is 3. The array should look as follows (ppage might be different):

```
0: pdx: 0x0, ptx: 0x9, ppage: 0xC3, present: 0, writable: 1, encrypted: 1
1: pdx: 0x0, ptx: 0x8, ppage: 0xC2, present: 1, writable: 1, encrypted: 0
2: pdx: 0x0, ptx: 0x7, ppage: 0xC1, present: 1, writable: 1, encrypted: 0
```

When the actual number of valid virtual pages is less than or equals to the **num,** then only fill up the array using those valid virtual pages. Return the actual number of elements that are filled in **entries**. The only error defined for this function is if **entries** is a null pointer, in which case you should return -1. Return -1 if you encounter any other error, too.

int dump_rawphymem(uint physical_addr, char * buffer)

allows the user to dump the raw content of one physical page where **physical_addr** resides (This is very dangerous! We're implementing this syscall only for testing purposes.). The kernel should fill up the buffer with the current content of the page where **physical_addr** resides -- it should not affect any of the page table entries that might point to this physical page (i.e., it shouldn't modify PTE_P or PTE_E) and it shouldn't do any decryption or encryption. Note that **physical_addr** may not be the first address of a page (i.e., may not be page aligned). **buffer** will be allocated by the user and have the size of **PGSIZE**. You are not required to do any error handling here. Note that **argptr()** will do a boundary check, which would cause an error for the pointer **physical_addr.** Therefore, when you grab the value of **physical_addr** from the stack, use **argint()** instead of **argptr().**
dump_rawphymem should return 0 on success and -1 on any error.
If you do this function right, it will only be a couple of lines of code (see copyout).
UPDATE: physical_addr is now a uint instead of a char*. You must use argint to parse it, and you cannot dereference it until you translate it to a kernel virtual memory address. How do you do that?

Hints

Before you start with the coding part, make sure you understand the layout of the virtual memory and how to index and manipulate page table in xv6. This includes:

1. How to grab certain entry from the page table

2. How to change a certain bit in the page entry

3. How to access the physical memory from the kernel

Reading through the existing codebase is a good start for you to figure out the above question.

Code Delivery

Handing in Your Code

EACH project partner should turn in their joint code to each of their handin directories.

So that we can tell who worked together on this project, each person should place a file named **partners.txt** in their handin/p5 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

To repeat, both project partners should turn in their code and both should have this **partners.txt** file in their handin/p5 directory.

Within your p5 directory, make the following directory and place your xv6 code in it as follows:

```
~cs537-1/handin/<login>/p5/ontime/src/<xv6 files>
```

If you wish to use slip days in this project, then you should submit your code to the corresponding slip directory: **slip1**, **slip2**, or **slip3. slip1** indicates that you wish to use one slip day in this project. We will use the latest submission for grading.

Testing

We strongly recommend you first write a few small user programs to test various aspects of this project. A simple user application could look as following

```
char *ptr = sbrk(PGSIZE); // Allocate one page
mencrypt(ptr, 1); // Encrypt the pages
struct pt_entry pt_entry;
getpgtable(&pt_entry, 1); // Get the page table information for newly allocated page
```

Ensure correct behavior from these tests before moving on to our tester. The tester is at `~cs537-1/tests/p5/run-tests.sh` If you want to run just test n, you can run `~cs537-1/tests/p5/run-tests.sh -t n` On any CSL machine, use `cat ~cs537-1/tests/p5/README.md` to read more details about how to list the tests and how to run the tests in batch. Note that there will be a small number of hidden test cases (25%).

In your Makefile, just like in previous PAs, set CPUS = 1 and change the compilation flag from O2 to Og.

Slip Day Policy

A maximum of 3 slip days can be used for this project no matter you are working with a partner or not. Additional 2 slip days for each one have been added as described in this [post (https://piazza.com/class/kjn4sz4kq7t2d2?cid=596)](https://piazza.com/class/kjn4sz4kq7t2d2?cid=596).

If you are working with a partner, then

- Each of you will only need to contribute 1/2 of any slip days you use; for example, if you use 1 slip day, each of you is charged 1/2 of a day.
- If only one of you runs out of slip days, the needed slip days will be taken from the partner who still has them.
- A 1/2 slip day can't be used (unless you are combining with a 1/2 slip day from your partner). We will assume you are aware of your partner's slip days and the implications.