CS 450: Midterm (Spring 2019)

Black Hat

Please Read All Questions Carefully!

There are sixteen (16) total numbered pages, with eight (8) questions

Please put your FULL NAME (mandatory) on THIS page only.

Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		10
Q4		10
Q5		10
Q6		10
Q7		10
Q8		10
Total		80

Directions

> *A hacker is someone who thinks outside the box. It's someone who discards conventional wisdom, and does something else instead. It's someone who looks at the edge and wonders what's beyond. It's someone who sees a set of rules and wonders what happens if you don't follow them. A hacker is someone who experiments with the limitations of systems for intellectual curiosity.*
>
> *—Bruce Schneier*

Welcome to the exam. Actually, this isn't just an exam. It *may* also be a document that we send along to the NSA to search for potential recruits. *Maybe*. If you don't know what the NSA is, that's okay—I imagine that's just what they want. You're going to be putting on your *black hat* and applying the knowledge you've acquired thus far to think like a `l33t h4ck3r`. I promise that this exam is not wiretapped.

Good luck! A good strategy is to start with the easy problems (not necessarily the order that they're given in) and work your way to the harder ones. Hint: this requires that you flip through every page first. Pay close attention and think like a *hacker*…

P.S. please don't give answers in ciphertext. We have lives too.

1. D3adl0ck H0liday

You're an undercover agent working for a foreign software company with known ties to arms dealers. They're trying to scale up their illegal arms dealing website (hosted on the dark web, of course) to handle more customers, and they need to use concurrency. Your goal is to insert a deadlock bug into their code that will be hard to detect, and that will harm their services and cost them money.

Here's a piece of backend code for their marketplace website that determines if two customer carts contain the same items:

```
cart_t * shared_items (cart_t * customer1, cart_t * customer2) {
      cart_t * c = malloc(sizeof(*c));
      for (int i = 0; i < customer1->len; i++) {
            if (in_cart(customer2, customer1->items[i])
                  cart_add(c, customer1->items[i]);
}
```

This code needs to be modified so that multiple threads can execute it concurrently. What needs to be added? **Hint**: you'll want to use mutexes.

Why will this addition be hard to identify as a *purposefully* injected bug?

Show a case where the new code will definitely result in deadlock.

If someone were to discover the bug, how would they fix it?

2. The Dr34d Pyr4t R0berts and His B00tstraps

Bootstrap attacks are some of the most sophisticated, hardest to detect, and downright annoying. Recall that the boot sequence generally goes: BIOS -> bootloader -> OS.

If we wanted to corrupt a system with a malicious bootloader, where on the disk would we need to write?

Among the many tools in a hacker's toolkit is the dreaded *rubber ducky*, a USB drive designed to inject some malicious code into a victim system when plugged in. You might sprinkle them in a parking lot hoping that someone unwittingly picks one up and uses it, or if you have physical access to a target machine you can insert it yourself. Many BIOSes allow you to boot from such a USB drive. How might we use a rubber ducky to install our own evil bootloader?

How would a forensics agent detect that the bootloader has been modified?

3. Hiding in plain sight (STUXNET lite)

You've been charged with developing a *rootkit*, piece of kernel code which can do many things given the privilege level it runs at. Your rootkit has one goal: *to inject and hide exploit code in a process's memory.* How the malicious exploit code got there in the first place, and how it is revealed again when it needs to execute isn't your concern, you just have to conceal it so that an enterprising sysadmin doesn't come along and ruin your fun.

Assume you have a 15-bit virtual address, with page size = 32 bytes. Assume further a two-level page table, with a page directory which points to pieces of the page table. Each page directory entry is 1 byte, and consists of a valid bit and a PFN of the page of the page table. Each page table entry is similar: a valid bit followed by the PFN of the physical page where the desired data resides.

The page directory is at page 82. Consider the following physical memory layout for a victim process:

```
page  33: 7f 7f 7f 7f f0 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
          7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f

page  38: 02 09 0b 12 01 0c 1e 05 10 0c 08 16 06 1c 14 0f
          0b 1d 10 00 0f 1a 08 04 1e 06 0f 0c 1b 0d 00 19

page  82: 7f 7f 7f 7f f6 7f 7f 92 7f 90 7f 7f 7f 7f 7f 7f
          7f 7f 7f a1 7f 7f 7f 7f 7f 7f 7f 7f 7f ef 7f 7f

page 112: 1b 1d 07 16 15 03 04 1e 18 00 1d 00 18 00 16 1d
          14 13 0a 02 17 0a 00 06 15 14 02 10 14 10 0c 1e

page 118: a6 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
          7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
```

We first need to figure out where to stash the exploit code, without modifying the address space. We need to find a page of memory that holds actual application data, and overwrite it with our payload. Let's assume we use page 112. Now we want to overwrite that with the payload, but anybody who scans memory shouldn't be able to see it. Instead, we'll make it look like *some other page of actual application data*.

Which page can we use for this?

OK, now if someone tries to access data at page 112, they should actually see the data at this page. What strategy can we use here?

Which page needs to be modified? Hint: 112 in hex is 0x70. What will that become with a valid bit?

How does it need to be modified?

How do we make the code visible again when it's time to execute our exploit?

4. B0mbs 4way

A *logic bomb* is a piece of malicious code that runs at a predetermined time or when a predetermined condition is met. Hacker Joe has injected a OS-level logic bomb into our kernel.

There are three processes in the system, p0, p1, and p2. These processes can be in one of three states: READY, WAITING, RUNNING, or DONE. The kernel scheduler will switch processes when a process goes into the WAITING state or DONE state. Processes are either doing work (denoted cpu) or are performing IO (denoted io). Assume cpu work takes one scheduler tick, and the scheduler tick is our unit of time. Assume that when a process does IO, it takes 3 scheduler ticks to complete (not including the tick where the process was in the RUNNING state). Consider the following process trace:

```
P0: io io io cpu
P1: cpu io cpu cpu cpu
P2: io io
```

Show the state of each process for the first 10 scheduler ticks. Assume the initial ready queue ordering of the processes is p0,p1,p2.

Hacker Joe's logic bomb is triggered when P1 enters the waiting state. At which tick number does this occur?

5. Who broke my spinlock?

You work for a new and successful (totally legitimate) software company in Ukraine called PewPewPewtex. After a recent nightly build, things have been breaking, deadlocking, and crashing. You've been tasked with figuring out what's going on, and you're looking at a two-thread spinlock trace in your webserver code.

Here, "c" means a thread is computing, "S" means a thread is spinning on a lock, "A" means a thread acquired the lock, and "R" means a thread released the lock. Look at each of the following traces and determine if it looks suspicious or not, and explain why:

T1: cccccccSSSS AccccRcc
T2: ccccAccc ccccccRccc

T1 : ccccA Rccc
T2 : A ccccRccc

T1 : SSSSSSSS AR
T2 : Rccccccc

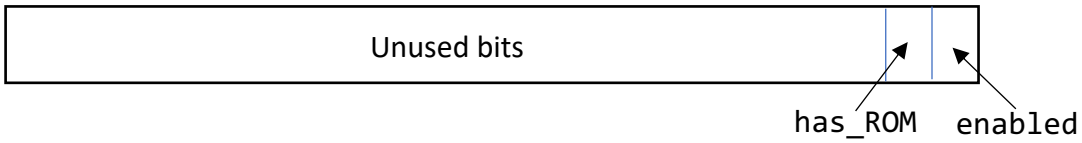
T1 : c cAcccc
T2 : c c RccccccA

6. GodMode: DEITYBOUNCE

Intel is considering a new simplified I/O device standard called SPI (Simple Peripheral Interface). Each SPI device must expose 3 special 4-byte registers that give the device ID, device information flags, and the (physical) address of a routine which must be called by the BIOS when the machine boots up (this routine is called an *expansion ROM*). This register space looks like so:

Device ID
Flags
Expansion ROM

The zoomed in flags structure looks like this



On the SPI bus, there can be a maximum of 64 devices connected. The SPI device specification has a section for BIOS developers that reads:

> *When the operating system boots up, it must scan the SPI bus and enable every valid device. BIOS software will check for a valid device by reading the* `device ID` *register for a SPI bus number between 0 and 63. A valid device will return a value !=* `0xffff`*. The BIOS can enable the device by writing a one to the enabled bit of the flags register. If that device has an expansion ROM (denoted by the* `has_ROM` *bit of the flags register), the expansion ROM must be executed (by jumping to the address in the Expansion ROM register).*

Assume your BIOS code has existing utility functions that looks like the following which will let you read and write an SPI device's SPI-specific registers:

```
unsigned int spi_dev_read_reg(int dev_num, int reg_num);
void         spi_dev_write_reg(int dev_num, unsigned int value);
```

And a function which invokes an expansion ROM for you given its address:

```
void spi_invoke_exp_rom(unsigned int exp_rom_addr);
```

For example,

`spi_dev_read_reg(0, 0)` will return the value of the device ID register for the first device on the SPI bus.

Write some BIOS driver code to initialize the SPI subsystem according to the specification outlined above. Syntax is not important.

Dell sells an SPI-compatible disk whose firmware can be updated. The firmware is some code sitting on a flashable ROM (EEPROM) inside the disk controller. This firmware appears to SPI as an expansion ROM. Explain how this Dell disk is vulnerable.

Explain how you might use this vulnerable disk to take over a system.

How might the SPI standard be changed to close the vulnerability?

7. Escalate.exe

As we saw before, a rootkit is some malicious kernel code which enjoys all the privileges of ring 0. For the following attack vectors, choose whether you'd need a rootkit or a regular user-level exploit to carry them out. If you're uncertain, (e.g. you might think it could be done with *both*) explain your reasoning below.

`fork()` an evil process	User exploit	Rootkit
Invoke a system call	User exploit	Rootkit
Modify a page table entry	User exploit	Rootkit
Overwrite my process stack	User exploit	Rootkit
Change the value of a general-purpose register	User exploit	Rootkit
Overwrite *any* process stack	User exploit	Rootkit
Turn off interrupts	User exploit	Rootkit
Hide an evil process in the process table (e.g. so `ps` and `top` don't show it)	User exploit	Rootkit
Modify internal memory of an IO device	User exploit	Rootkit
Give data to an input routine longer than it expects	User exploit	Rootkit

8. Denying Your Service

We're trying to analyze a scheduler trace to see if someone is performing a denial of service attack on our scheduler. **If the turnaround time of a job is greater than 4, we mark a trace as suspicious**. The following timelines show a set of jobs arriving, and then being executed on a processor. Which job traces are suspicious? Mark either "yes" or "no."

Assume A arrives at the beginning of the time unit where the * is on the timeline, A ends at the end of the time unit where "x" is, and that each tick moves time forward one unit.

Example:

ABAAB
* x

This means that A starts at time t=0, runs right away, and finishes at time 4. B starts at time 1, runs until time 2, then starts again at time 4 and finishes at time 5.

ABABABABABBB _____
* x

AAAAABABBBBB _____
* x

ABABBBBBBBBB _____
* x

BBAAAABBBABB _____
  * x

For the following, calculate the **response time** (in time units) for A:

BAAAAAAAAAB _____
  * x

BBBBAAAAAAA _____
  * x

ABBABABBBBBB _____
* x

Blank page

Extra Credit (10 points)

Decipher the code.

Blank Page