New York University
CSCI-UA.480-008: Advanced Computer Systems: Spring 2016

Midterm Exam

- This exam is **75 minutes**. Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.
- There are **10** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is open notes, as long as your notes do not include the posted class notes and the lab source code. You may not use electronics: phones, tablets, calculators, laptops, etc.**
- If you find a question unclear or ambiguous, state your assumptions.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and NetId on this cover sheet and on the bottom of every page of the exam.**

Do not write in the boxes below.

I (xx/8)	II (xx/23)	III (xx/32)	IV (xx/18)	V (xx/19)	Total (xx/100)

Name: Solutions

NYU NetId:

I Networking (8 points total)

1. **[8 points]** A server executes the code below (assume the code has the correct headers):

```
int fd1, fd2, fd3;
struct sockaddr_in server_addr;

if ( (fd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      exit(-1);

memset(&server_addr, 0, sizeof(struct sockaddr_in));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(50002);

if ( bind(fd1, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0)
      exit(-1);

if ( listen(fd1, 50) < 0)
      exit(-1);

fd2 = accept(fd1, NULL, NULL);
if (fd2 < 0)
      exit(-1);

/* Marker A */

fd3 = accept(fd1, NULL, NULL);
if (fd3 < 0)
      exit(-1);

/* Marker B */
```

Which of the following statements is correct?

Circle the BEST answer:

A The flow of execution cannot reach `Marker A`.

B The flow of execution can reach `Marker A` but not `Marker B`.

C The flow of execution can reach `Marker B`; at that point, the variables `fd1`, `fd2`, and `fd3` are all equal.

D The flow of execution can reach `Marker B`; at that point, the variables `fd2` and `fd3` are equal to each other but different from `fd1`.

E The flow of execution can reach `Marker B`; at that point, the variables `fd1` and `fd2` are equal to each other but different from `fd3`.

F The flow of execution can reach `Marker B`; at that point, the variables `fd1`, `fd2`, and `fd3` are all different.

F.

II Buffer overflow attacks and defenses (23 points total)

2. [8 points] In the function `translate()` below, `src` is guaranteed to be null-terminated. However, the content and length of `src` are determined by a network client (and thus potentially supplied by an attacker). `translate()` has a buffer overflow vulnerability. Your job is to fix it.

Fix the vulnerability, by crossing out and rewriting. Note that you may need to change the interface to the function. You should preserve the original purpose of the function as much as possible.

```
void translate(char* dst, const char* src, unsigned int dst_len  )
{
     unsigned int i;
     while (1) for (i = 0; i < dst_len-1; i++)   {


        if (*src == 'a')
            *dst = 'A';
        else
            *dst = *src;

        if (*dst == 0)
            break;

       src++;
       dst++;

    }

    if (i == dst_len - 1)
        *dst = 0;

}
```

The above null terminates dst, but we will give full credit for solutions that do not do that. The requirement is that the code takes a length, null terminates if there is room, and stops short of writing `dst + dst_len`.

3. [15 points] In the multiple choice questions below, unless specified otherwise, assume a single process, single-threaded server written in C, with no defenses (stack canaries, `W^X`, and so forth).

If the server is bug-free, an adversary can successfully do which of the following?

Circle the BEST answer below:

A Mount a stack smashing attack that crashes the server

B Mount a stack smashing attack that `execs` a shell

C Both **A** and **B**

D Neither **A** nor **B**

D

Now assume that the server has a stack overflow vulnerability. The server runs with a non-executable stack. Using the kind of attacks described in AlephOne's paper, an adversary can successfully do which of the following?

Circle the BEST answer below:

A Mount an attack that crashes the server

B Mount an attack that `execs` a shell

C Both **A** and **B**

D Neither **A** nor **B**

A

Assume a multi-threaded server. Stack canaries can potentially be defeated by which of the following?

Circle ALL that apply:

A ASLR (address space layout randomization)

B Exploiting a bug in which a heap-allocated buffer can overflow

C Exploiting a bug in which an attacker can control the arguments to `memcpy`

D Stack reading, provided the server does not rerandomize its canaries after a crash and restart

E Stack reading, provided the server does rerandomize its canaries after a crash and restart

B, C, D.

III Return-oriented programming (32 points total)

4. [22 points] You will use return-oriented programming (ROP) to exploit a server. Although the server is configured to never execute code located on the stack, it has a vulnerability that allows you, as the attacker, to replace the contents of its stack, beginning at a location (call it X) where the server is expecting a return address, and continuing indefinitely. Thus, if you overwrite this stack location with your chosen code address, you will alter the programmer's intended control flow. Also, the server is running as root and can thus can be made to execute privileged operations.

Your goal is to cause the server to replace the machine's password file, print an obnoxious message, and then exit cleanly. Specifically, you need to cause the following to execute:

```
unlink("/etc/passwd");
link("/eviluser/passwd", "/etc/passwd");
printf("don't run as root\n");
exit(0);
```

Although we used C syntax above, you will be coding with the building blocks listed below. Note that we are assuming that you, as the attacker, have precise knowledge of the server's address space (so only ROP is needed here, not ***B***ROP). `c1`, . . . , `c4` are the addresses of machine instruction sequences within the server's text (code) section; we have represented these instructions in terms of their function, rather than their byte representation. `d1`, . . . , `d3` are the addresses of useful data (you can imagine that the attacker earlier placed these strings in memory); we have represented this data in terms of the ASCII characters, with `\0` representing the null terminator. `p1` is the address of `printf()`.

c1	`pop %eax; ret`	d1	`/ e t c / p a s s w d \0`
c2	`pop %ebx; ret`	d2	`/ e v i l u s e r / p a s s w d \0`
c3	`pop %ecx; xorl %eax, %eax; ret`	d3	`d o n ' t   r u n   a s   r o o t \n \0`
c4	`int 0x80; pop %edx; ret`	p1	`pushl %ebp; ...; popl %ebp; ret`

Important conventions:

- The platform is Linux, running on the 32-bit x86 architecture; this is the same as in our labs.
- Calling convention:
  - `int 0x80` traps into the kernel to make a system call
  - `%eax` holds the system call number (see the table below)
  - `%ebx` holds the first argument to the system call
  - `%ecx` holds the second argument to the system call
  - `%edx` holds the third argument to the system call
- Relevant system call numbers:

`exit()`	1
`unlink()`	9
`link()`	10

Name: Solutions NYU NetId:

In the space below, write down what the attacker should place on the stack, beginning from location X, to cause the execution of `unlink("/etc/passwd")`.

$X+24$		
	junk value	
	c4	
	9	
$X+8$	arg2 (overwrite this) c1	
$X+4$	arg1 (overwrite this) d1	
X	return address (overwrite this) c2	
$X-4$	saved `%ebp`	`<--- %ebp`

Below, draw the rest of the payload, beginning from location $X+20$ (you may wind up copying over some of what you have above).

	c4
	1
	c1
	0
	c2
	d3
	c1
	p1
	junk
	c4
	10
	c1
	d1
	c3
	d2
$X+24$	c2
$X+20$	junk

Name: Solutions **NYU NetId:**

5. [10 points] The operator of a server reads the BROP paper and grows concerned. The operator adjusts the logic of the server so that, if the server crashes, it does not restart automatically.

Does this adjustment stop the BROP attack? Justify. Write no more than two sentences.

Yes, it stops BROP. BROP relies on a server that keeps restarting, which gives the client a chance to learn about the server's address space.

Does this adjustment have side effects? Justify. Write no more than two sentences.

The side effect is that any crash in the server takes the server offline.

IV Passwords and privilege (18 points total)

6. [8 points] Consider the following threat model. There is a Web site (meaning a Web server and its associated logic) on which you have previously set up a login id and password. An attacker knows your login id but not your password, and is trying to impersonate you. The attacker is remote and can interact with the Web site only through the site's intended interface, meaning that the attacker can submit (login, password) pairs. The attacker cannot subvert the control flow or logic of the Web site.

Does the mechanism of *salting* provide protection against this attacker? If so, state what protection it provides. If not, explain why not. Write no more than three sentences.

The salt does not help with this attacker. This attacker is limited to guessing your password, and the salt does not make such guesswork harder or easier. The purpose of the salt is to defend against an attack in which the adversary obtains the password file (which was ruled out by the threat model).

7. [10 points] Assume a Unix system with a `root` user, who has uid and gid 0; further assume that `root` is not compromised, and that all programs that run as `root` are bug-free. Assume that the system has a binary, `svc`, and that the operator's intent to run this specific binary as user id (uid) `63001` and group id (gid) `63001`. There is also a user account whose associated user id and group id are `300`.

Consider four alternatives for the permissions and ownership of `svc`:

```
                   perms      uid    gid
                  -------------------------
Alternative A:    r-xr-x---    0      0
Alternative B:    r-xr-xr-x    0      0
Alternative C:    r-xr-x---    0      63001
Alternative D:    r-xr-x---   63001   63001
```

Between alternatives A and B, which of the two corresponds to less privilege in this system? Justify. Write no more than two sentences.

Alternative A has less privilege because it does not allow non-root users to read or execute the binary.

Between alternatives C and D, which of the two corresponds to less privilege in this system? Justify. Write no more than two sentences.

Alternative C has less privilege because alternative D gives 63001 the ability to modify the permissions (perhaps making the binary writable), which could enable a service running as 63001 to overwrite the binary with something else.

Name: Solutions **NYU NetId:**

V SFI and feedback (19 points total)

In the questions below, assume a system that uses SFI on a 32-bit RISC architecture, with 4-bit segment Ids. Make the following assumptions about this architecture:

- All instructions are four bytes.
- The instruction pointer is always four-byte aligned (that is, the CPU raises a fault if code tries to jump into the middle of an instruction).
- There are no implicit stack manipulations: instructions like `call`, `ret`, `push`, `pop` do not exist. (To implement stack frames, the compiler designates a register as the stack pointer. For example, when returning from a function, code moves the return address into a register, increments the stack pointer, and jumps to the address in the register.) If the parenthetical confuses you, ignore it.

8. [9 points] Recall that, in SFI, an indirect store instruction, such as `Store R1, R0` (which in x86 syntax would be `movl %r1, (%r0)`), is sandboxed as follows, where `Ra`, `Re`, and `Rf` are dedicated registers (which you should assume are initialized correctly on entry to a fault domain):

```
Ra <- R0 & Re  // Re is 0x0fffffff, because we assume 4-bit segment Ids
Ra <- Ra | Rf  // Rf is 0xI0000000, where I identifies the data segment
STORE R1, Ra
```

Above, the first line clears the top four bits of `R0`, placing the result in `Ra`; the second line places the correct segment Id in `Ra`. (Note that sandboxing indirect jump instructions uses different registers in place of `Ra` and `Rf`.)

Wishing to cut overhead from SFI, your friend proposes to eliminate the first line above. Specifically, your friend proposes the following alternative sandboxing of `Store R1, R0`:

```
Ra <- Ra | Rf  // Rf is 0xI0000000, where I identifies the data segment
STORE R1, Ra
```

What claim or claims are true of the proposed replacement? Circle the BEST answer.

A This approach would be accepted by the existing verification algorithm.
B All stores are guaranteed to remain within the data segment.
C This approach is an acceptable replacement.
D Claims **A** and **B** are correct.
E Claims **A** and **C** are correct.
F Claims **B** and **C** are correct.
G Claims **A**, **B**, and **C** are correct.
H All of the claims are incorrect.

D. This meets the requirements of the verification algorithm. The problem is that the store no longer happens to the correct address because register `R0` (more precisely, its bottom bits) have been forgotten about.

Name: Solutions NYU NetId:

9. [8 points] This question asks whether and how SFI protects against various misbehaviors in the untrusted fault domain. Make the following assumptions:

- The untrusted fault domain runs in the same OS process as the trusted domain.
- The untrusted fault domain accepts external input (which could be supplied by an attacker).
- The stack used by the untrusted fault domain is non-executable, and the `W^X` policy is in effect.
- *stores* are sandboxed, but *loads* are not. (This is the configuration in many of the authors' experiments.)

Fill in the blanks below with a few words indicating whether the SFI authors propose to protect against the misbehavior and, if so, what the proposed protection is. If there is no protection for the given issue, write NONE.

Untrusted fault domain reads the trusted domain's memory ________________________________NONE; loads aren't sandboxed.

Untrusted fault domain dereferences a null pointer ________________________________Unix signals; see last paragraph of §4 of paper.

Untrusted fault domain's control flow subverted by an attacker ________________________________NONE; attacker can buffer overflow and use ROP, etc. This is a subversion of control flow.

10. [2 points] This is to gather feedback. Any answer, except a blank one, will get full credit.

Please state the topic or topics in this class that have been least clear to you.

Please state the topic or topics in this class that have been most clear to you.