

Computer Science 5271
Fall 2014
Midterm exam
October 14th, 2014
Time Limit: 75 minutes, 2:30pm-3:45pm

- This exam contains 9 pages (including this cover page) and 5 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TA, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- By signing below you certify that you agree to follow the rules of the exam, and that the answers on this exam are your own work only.

The exam will end promptly at 3:45pm. Good luck!

Your name (print): _____

Your UMN email/X.500: _____@umn.edu

Number of rows ahead of you: _____ Number of seats to your left: _____

Sign and date: _____

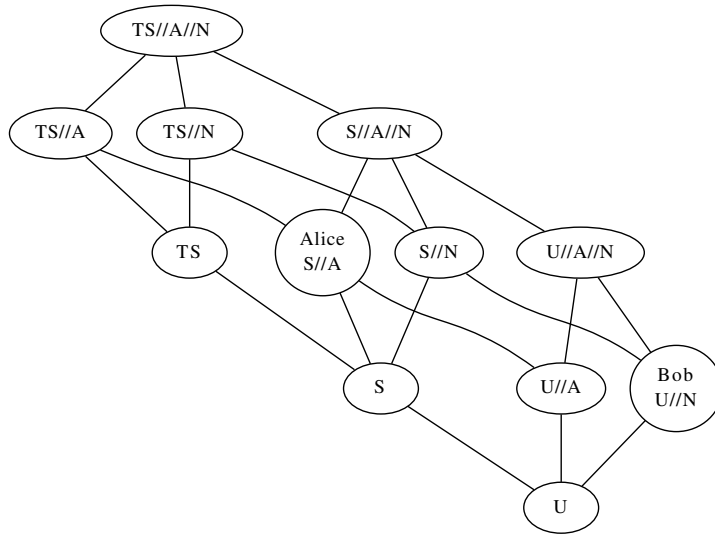
Question	Points	Score
1	20	
2	24	
3	20	
4	20	
5	16	
Total:	100	

1. (20 points) Matching, specific to general. Fill in each blank next to a more specific term or concept with the letter of a more general term or concept it is an instance of. (Don't mix this up with any other relationships, like "used together with" or "sounds like".) Each answer is used at least once, some more than once.

- (a) ___ Unix file permissions
- (b) ___ DEP
- (c) ___ `chown`
- (d) ___ Bell-LaPadula
- (e) ___ Linux kernel
- (f) ___ Unix file descriptor
- (g) ___ L4 microkernel
- (h) ___ `printf`
- (i) ___ non-executable stack
- (j) ___ function pointer overwrite
- (k) ___ `execve`
- (l) ___ `system`
- (m) ___ precondition
- (n) ___ Facebook Connect
- (o) ___ return to `libc`
- (p) ___ TOCTTOU bug
- (q) ___ Biba
- (r) ___ ROP
- (s) ___ stack smashing
- (t) ___ Microsoft Passport

A. access control list B. capability C. centralized authentication attack D. code injection
E. code reuse attack F. invariant G. `libc` function H. MAC model
I. race condition J. reference monitor K. system call L. $W \oplus X$

2. (24 points) Multiple choice. Each question has only one correct answer: circle its letter.



- (a) The military systems of a small island nation use a BLP security policy according to the lattice pictured above: there are three levels for unclassified (U), secret (S), and top secret (TS) information, and specialized compartments for secrets specific to the army (A) and navy (N); the country has no air force. Suppose that the email system is configured to allow emails to be delivered only if BLP would allow information flow from the sender to recipient. If Alice is a lieutenant in the army with clearance S//A, and Bob is a private in the navy with clearance U//N, which of these describes how they can communicate?
- They can send emails in both directions
 - Alice can send email to Bob, but not Bob to Alice
 - Bob can send emails to Alice, but not Alice to Bob
 - Neither Alice nor Bob can send emails to the other
- (b) Continuing with the lattice from the previous question, suppose that intelligence gathered by both Alice and Bob is put together in a combined report. What is the correct classification of this report?
- TS//A//N
 - S//A
 - S//A//N
 - S//N
 - U//A//N
- (c) This feature of the x86 architecture makes overlapping instructions possible:
- Instructions have variable length
 - Instruction lengths are not multiples of 8 bits
 - There is no PC-relative addressing
 - There are few general-purpose registers
 - Most byte sequences are legal instructions
- (d) In the classic style of ROP for x86, this register plays a role analogous to the program counter `%eip`:
- `%esi`
 - `%gs`
 - `%esp`
 - `%ebp`
 - `%eax`

- (e) Which of these intended instructions could **not** supply the end of a gadget in an ROP or JOP attack?
- A. `xor %eax, %ebx (0x31 0xc3)`
 - B. `add %dh, %ah (0x00 0xf4)`
 - C. `jmp *%ebx (0xff 0xe3)`
 - D. `add %ax, -0x1d(%edi,%edi,8) (0x66 0x01 0x44 0xff 0xe3)`
 - E. `ret (0xc3)`
- (f) Which of these security mechanisms is **not** deployed in most modern systems?
- A. Stack canaries
 - B. $W \oplus X$
 - C. CFI
 - D. Access control lists
 - E. ASLR
- (g) In a 32-bit Linux/x86 program, which of these objects would have the lowest address (numerically least when considered as unsigned)?
- A. A global array
 - B. A local array in `main`
 - C. An environment variable
 - D. A command-line argument in `argv`
 - E. A local `int` variable in a function
- (h) This function has so few secure uses that it has been removed from the latest version of the C standard:
- A. `system`
 - B. `gets`
 - C. `setuid`
 - D. `strcat`
 - E. `sprintf`
- (i) Which of these systems is **not** designed using privilege separation?
- A. Android
 - B. `qmail`
 - C. OpenSSH
 - D. The Linux kernel
 - E. Google Chrome
- (j) Which of these C expressions does **not** evaluate to 0 on a system where ints are 32 bits?
- A. `5 / 10`
 - B. `(int)(signed char)'\xa2' - (unsigned)(unsigned char)'\xa2'`
 - C. `0x80000000u + -(0x80000000u)`
 - D. `0x01000000 << 8u`
 - E. `10 >> 4`
- (k) This format specifier allows a format string vulnerability to be used to overwrite memory:
- A. `%d`
 - B. `%x`
 - C. `%s`
 - D. `%c`
 - E. `%n`
- (l) The following set of 8 instruction types is Turing complete, even if self-modifying code is prohibited. Removing which instruction type would leave the set not Turing complete?
- A. Decrement a memory location
 - B. Unconditional direct jump
 - C. Store register to memory
 - D. Set register to constant
 - E. Increment a memory location
 - F. Load register from memory
 - G. Branch if register is zero
 - H. Exchange register with memory location

3. (20 points) Attacks versus defenses. We've discussed a number of defense techniques that try to make C programs less vulnerable to attacks, but not every defense helps against every attack technique. In the following matrix, the rows correspond to techniques an attacker might use, and the columns correspond defense techniques. Fill in each entry of the matrix according to whether that defense helps stop that attack technique. Each entry should be a letter C, P, or N, with the following meanings:

C = Complete protection: the attack technique is no longer possible

P = Partial protection: the attack technique blocked in some but not all circumstances, or more difficult but still possible

N = No protection: the defense does not affect the attack technique

Answer for each attack technique on its own, not counting other attack techniques it might sometimes be combined with. We've already filled in some of the entries for you.

	ASLR	Non-executable stack	Coarse-grained CFI	Adjacent stack canaries	Shadow stack
Shellcode in environment variable	P		C		
Return address overwrite	P				
Directory traversal		N			
ROP using <code>ret</code>					
Non-control data overwrite				N	

4. (20 points) Race conditions. Below are four code samples from C programs interacting with a Unix filesystem. Two of the samples suffer from race condition vulnerabilities. For each vulnerable sample, describe a possible attack in three phases: select a point in the execution of the vulnerable code where an attack action should interleave by checking one of the boxes. Then, specify any preparation action, the interleaved action, and any final action for the attacker to take. If the code is not vulnerable, you can simply write “not vulnerable” in the preparation action line and leave the rest blank. Specify the actions clearly: for instance shell commands or short code snippets might work well.

Similar to the situation in HA1, you can assume that the listed code is running with root privileges, while the attacker is unprivileged; unqualified paths refer to a directory owned by the attacker. An attack can succeed if it writes to or removes the file `/etc/passwd`, or if it can read a secret file `/root/.ssh/key`; you do not need to include details of what to write to or read from these files. The attacker can also succeed by reading data described in the code as “secret.”

Your attacks are worth 7 points each, while correctly identifying the non-vulnerable code segments is worth 3 points each. If you want to optimize for partial credit, you can fill in either more or less than two attacks if you want.

```

int fd = open("file", O_WRONLY);           Preparation action:
assert(fd != -1);

int res = fstat(fd, &st_buf);
assert(!res);
 Interleaved action:
if ((st_buf.st_mode & 0222) != 0222) {
(a)   /* Not enough write privileges */
      abort();
}
 Final action:
write(fd, data, data_size);

close(fd);

```

```

 Preparation action:
int res = stat("/etc/hostname", &st_buf);
assert(!res);

char *buf = malloc(st_buf.st_size);
assert(buf);
 Interleaved action:
(b) int fd = open("/etc/hostname", O_RDONLY);
assert(fd != -1);
 Final action:
read(fd, buf, st_buf.st_size);

write(1, buf, st_buf.st_size);

```

Preparation action:

```
int res = stat("file", &st_buf);
assert(!res);

if ((st_buf.st_mode & 0222) != 0222) {
    /* Not enough write privileges */
    abort();
}
```

Interleaved action:

```
(c) 
int fd = open("file", O_WRONLY);
assert(fd != -1);

write(fd, data, data_size);

close(fd);
```

Final action:

Preparation action:

```
int fd = open("file",
    O_CREAT|O_WRONLY|O_TRUNC, 0666);
assert(fd != -1);

```

Interleaved action:

```
(d) 
int res = fchmod(fd, 0600);

/* readable only by root */

write(fd, secret, 1024);

close(fd);
```

Final action:

5. (16 points) Call-preceded ROP. Suppose you want to subvert the execution of a program with a code-reuse attack, but it uses a defense mechanism that constrains return instructions to jump only to locations right after legitimate calls. The task of creating an attack under these constraints is called call-preceded ROP. This also generally implies that the gadgets consist of intended instructions, so you can plan out your attack at the source-code level.

For this question, your job is to formulate the plan for a call-preceded ROP attack against the program shown on the next page. The function `vulnerable` contains a buffer overflow vulnerability that will let you overwrite the stack with a sequence of return addresses of your choosing, plus you also have control of the inputs to the program. Similar to HA1, you can assume that this program is running as root, so your goal as an attacker is to get it to run a shell such as `/bin/sh`.

We've marked a number of locations in the program right after function calls with labels running from A: through K:. This isn't quite legal C syntax, but the way we used the comma operator is intended to remind you that by returning to a different return site like this, you can replace the value that the calling function considers to be the return value of the function. For figuring out what you want to overwrite the stack with, you need to decide what sequences of return addresses you want the program to follow, starting with where you want the `vulnerable` function to return to, if not K. Write these down by letter in order, left to right. In the final attack there may need to be some extra space between the return addresses to account for other stack usage in the functions, but you don't need to worry about that.

You can assume that ASLR is disabled, so code and global data in the program have fixed addresses. In your plan for what to supply as the arguments to the program, you can write a formula for the value used in the attack, using the `&` operator as in C to refer to addresses of code and/or data. The first two arguments to the program are floating point numbers and the third is a string.

As a hint, we've only included in the program source the parts that you will need for your attack, and we've written the GOT/PLT machinery as if it were source code so it's easier to read. But we've included more return address spaces than you will probably need. The source code is on the next page so you can look at it all at once, but write your answers below.

Value for program argument 1 (number):

Value for program argument 2 (number):

Value for program argument 3 (string):

Return addresses (first to execute on the left):

--	--	--	--	--	--	--	--	--	--


```

typedef int (*func_ptr)(const char *);

func_ptr puts_got = puts;
int puts_plt(const char *s) {
    return ((*puts_got)(s), A:);
}

func_ptr system_got = system;
int system_plt(const char *s) {
    return ((*system_got)(s), B:);
}

struct list_node {
    struct list_node *next; /* offset 0 */
    struct list_node *prev; /* offset 4 */
    char *element; /* offset 8 */
};

struct point_info {
    int x; /* offset 0 */
    int y; /* offset 4 */
    char *descr; /* offset 8 */
};

void list_delete(struct list_node *p) {
    p->next->prev = p->prev;
    (free(p), C:);
}

char *process_ent(void) {
    struct list_node *n;
    char *s;
    n = (lookup_node(), D:);
    s = n->element;
    (list_delete(n), E:);
    return s;
}

void print_message(int msg_num) {
    const char *msg;
    msg = (lookup_msg(msg_num), F:);
    (printf("Message %d is: ", msg_num), G:);
    (puts_plt(msg), H:);
}

struct point_info pt;

int which_point = 0;

struct point_info *select_point(void) {
    which_point++;
    (printf("Selecting point\n"), I:);
    if (which_point == 0) {
        return &pt;
    } else {
        /* ... */
    }
}

void vulnerable(const char *s) {
    char buf[10];
    /* ... */
    (strcpy(buf, s), J:);
}

int main(int argc, char **argv) {
    assert(argc == 4);
    pt.x = 10000000.0*strtod(argv[1], 0);
    pt.y = 10000000.0*strtod(argv[2], 0);
    pt.descr = argv[3];
    (vulnerable(getenv("VAR")), K:);
    return 0;
}

```

In case you've forgotten, here's a reminder about some of the standard functions used in this example. `puts` prints a string to the standard output. `system` uses a string as a shell command. `strcpy` copies a null-terminated string without checking the size of the destination. `strtod` converts a string into a floating point number with type `double`; the first argument is the string and the second argument is not important for this question. If you assign a `double` value to an `int`, it will be rounded to a whole number, if it's within range. `getenv` returns the value of an environment variable, which in a case like this is under the control of an attacker.