Computer Science 5271
Fall 2017
Midterm exam
October 16th, 2017
Time Limit: 75 minutes, 1:00pm-2:15pm

- This exam contains 9 pages (including this cover page) and 4 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 2:15pm. 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	30	
2	24	
3	24	
4	22	
Total:	100	

1. (30 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) ____ Program for changing terminal settings

   (b) ____ Property that should hold whenever a program point executes

   (c) ____ System call to run a new program

   (d) ____ Reuse of code segments each ending in `0xc3`

   (e) ____ Isolation mechanism based on instruction rewriting

   (f) ____ VM implemented underneath a normal kernel

   (g) ____ Choosing random locations for memory regions

   (h) ____ Possible when the attacker already has an account

   (i) ____ Pointers to code called when a program exits

   (j) ____ Pointers used to implement shared library calls

   (k) ____ A common kind of race condition vulnerability

   (l) ____ Used when attacker can't control a pointer value

   (m) ____ When set, CPU allows all instructions to execute

   (n) ____ Attack against availability

   (o) ____ Requires that indirect jumps go only to intended targets

   (p) ____ Damage caused by attacks in one year

   (q) ____ Program that runs with the binary owner's privilege

   (r) ____ Another name for UID 0

   (s) ____ False positive and false negative rate when they are equal

   (t) ____ Used to connect networks of different classifications

   A. ALE B. ASLR C. CFI D. data diode E. DoS F. `.dtors` G. EER H. `execve` I. GOT J. heap spray K. hypervisor L. invariant M. local exploit N. ROP O. setuid P. SFI Q. `stty` R. superuser S. supervisor bit T. TOCTTOU

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

   (a) Suppose we are designing a MLS system with four levels (unclassified, confidential, secret, and top secret), and five specialized compartments. How many possible classifications are there?

   A. $4 \cdot 5 = 20$
   B. $2^4 + 2^5 = 48$
   C. $5 \cdot 2^4 = 80$
   D. $4 \cdot 2^5 = 128$
   E. $2^4 \cdot 2^5 = 512$

   (b) In a 32-bit Linux/x86 program, which of these objects would have the highest address (numerically largest when considered as unsigned)?

   A. An environment variable
   B. A local `float` variable in a function called from `main`
   C. A global array
   D. A local array in `main`
   E. The function `printf`

   (c) Depending on how they are used, all of these functions could cause a buffer overflow that replaces a return address, *except*:

   A. `strcmp`
   B. `strlcpy`
   C. `read`
   D. `strcat`
   E. `gets`

   (d) Consider an attack that uses a stack buffer overflow to replace a return address with the address of shellcode in an environment variable. Which of the following defenses could block the attack?

   A. W $\oplus$ X
   B. a shadow stack
   C. CFI
   D. Control-Flow Guard
   E. All of the above

   (e) Consider an attack that uses a stack buffer overflow to replace a local `int` variable on the stack, to cause the program to skip a permissions check. Which of the following defenses could block the attack?

   A. Code-pointer Integrity
   B. ASLR
   C. W $\oplus$ X
   D. CFI
   E. None of the above

(f) Suppose that `a` is a global variable that is an array of characters. Which of the following relationships always holds?

A. `strlen(a) < sizeof(a)`
B. `strlen(a) <= sizeof(a)`
C. `strlen(a) == sizeof(a)`
D. `strlen(a) != sizeof(a)`
E. None of the above

(g) 32-bit x86 systems have the following features:

1. Many legal addresses do not contain 0x00 bytes
2. 32-bit stores are allowed to unaligned addresses
3. Instructions are variable length and can start at any byte

Which of these features make format string vulnerabilities easier to exploit?

A. 1 and 2
B. 1 and 3
C. 2 and 3
D. 1, 2, and 3
E. None of these features help

(h) The ASLR system on a 32-bit system chooses the stack location uniformly at random in the high half of the address space (`0x80000000` to `0xfffff000`), with the limitation that the location is always a multiple of 2^{12} (`0x1000`, 4096). If you carry out a brute-force attack against this defense where the vulnerable program is a server that restarts 10 times per second, what's the maximum time before your attack is successful?

A. 2^{32} s $\approx 4.3 \cdot 10^9$ s $\approx$ 136 years
B. $(2^{32} - 2^{31} - 2^{12})/10$ s $\approx 2.1 \cdot 10^8$ s $\approx$ 6.8 years
C. $(2^{32} - 2^{31}) \cdot 10/2^{12}$ s $\approx 5.2 \cdot 10^6$ s $\approx$ 61 days
D. $2^{32-1-12}$ s $\approx 5.2 \cdot 10^5$ s $\approx$ 6.1 days
E. $(2^{32} - 2^{31})/(2^{12} \cdot 10)$ s $\approx 5.2 \cdot 10^4$ s = 15 hours

(i) Traditionally NOP sleds on x86 are made using the single byte no-op instruction 0x90, which is equivalent to `xchg %eax, %eax` (swapping `%eax` with itself). But you might consider making a NOP sled out of a longer instruction. Which of the following x86 instructions, if repeated many times, would make the best NOP sled?

A. `8d b6 00 00 00 00` (`lea 0x0(%esi),%esi`; add 0 to `%esi`)
B. `66 90` (`xchg %ax, %ax`; swap `%ax` with itself)
C. `b6 00` (`mov %0x0, %dh`; move 0 into `%dh`)
D. `00 00` (`add %al, (%eax)`; add `%al` to the location `%eax` points to)
E. `00 8d b6 00 00 00` (`add %cl, 0xb6(%ebp)`; add `%cl` to the location `0xb6` bytes beyond `%ebp`)

(j) Terri is the TA for 5271, and responsible for maintaining a directory on a Unix system containing the solutions for the hands-on assignments (this directory is fictional). There is a top-level directory named `solutions`, with subdirectories `ha1` and `ha2`. `ha1` in turn has subdirectories `week1` through `week5`, and the leaf directories contain text files with solutions. Based on Terri's default `umask` settings, the directories mostly have permissions `0750`, and the files mostly have permissions `0640`, owned by the `terri` user with group owner `CSEL-student`, a group containing all the students with accounts on the CSE Labs machines. Because 5271 students should not be able to read the solutions, Terri performs the command `chmod og-r solutions`, changing the permissions of the `solutions` directory to `0710`.

Only one of the following operations by 5271 students would be prevented by this change: which is it? (Recall that the `-d` option to `ls` causes it to print information about a directory instead of listing the directory's contents.)

A. `ls -ld solutions`
B. `ls solutions`
C. `ls -ld solutions/ha1`
D. `ls -l solutions/ha1`
E. `less solutions/ha1/week1/exploit.sh`

(k) Terri would like to argue that a contributing factor to the security failure in the previous question was poor design of the Unix file permissions system. Which one of Saltzer and Schroeder's principles was arguably not observed in the Unix design, leading to this failure?

A. Psychological acceptability
B. Open design
C. Compromise recording
D. Least common mechanism
E. Complete mediation

(l) Which one of the following relationships is always true, when the variables are interpreted as 32-bit unsigned C values? (The $\Rightarrow$ operator is logical implication.)

A. `(8*x == 0)` $\Rightarrow$ `(x == 0)`
B. `x + 1 > 0`
C. `(x == -x)` $\Rightarrow$ `(x == 0)`
D. `x + y - x = y`
E. `x + 5 >= x`

3. (24 points) Recognizing attack techniques. Each of the following sequences of bytes was recovered from the network traffic, logs, or memory of a Linux/x86 system under attack. Match each sequence of bytes (in which non-printable characters are represented by backslash escapes, as in C), with the attack technique it was likely used in. Each answer is used exactly once.

(a) ____ `\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90`

(b) ____ `\x80\xe8\xd6\xff\xff\xff//bin/sh\x9b\xd9`

(c) ____ `name="fred'; chmod 666 /etc/passwd; echo '"`

(d) ____ `There is a major center of economic activity`

(e) ____ `\x35\x90\x90\x90\x3c\x35\x90\x90\x90\x3c\x35`

(f) ____ `%08x%08x%08x%08x%42x%n%137x%n%246x%n%57x%n`

(g) ____ `\x01\x00\x00\x04`

(h) ____ `../../../../../../../../../../../etc/sudoers`

A. directory traversal
B. English shellcode
C. `execve` shellcode
D. format string attack
E. integer overflow
F. JIT spray
G. NOP sled
H. shell script injection

4. (22 points) Shellcoding. Your coworker on a penetration testing team previously attacked a setuid-root Linux/x86 binary by using the following shellcode to run the shell `/bin/sh`:

```
31 c9           xor    %ecx, %ecx    # ecx = 0
51              push   %ecx          # \0\0\0\0
68 2f 2f 73 68  push   $0x68732f2f   # "//sh"
68 2f 62 69 6e  push   $0x6e69622f   # "/bin"
89 e3           mov    %esp, %ebx    # ebx = "/bin//sh\0\0\0\0"
51              push   %ecx
53              push   %ebx
89 e1           mov    %esp, %ecx    # ecx = {"/bin//sh", 0}
31 c0           xor    %eax, %eax
b0 0b           mov    $0xb, %al     # eax = 11 (execve)
31 d2           xor    %edx, %edx    # edx = 0
cd 80           int    $0x80         # execve("/bin/sh", ["/bin/sh"], 0)
```

However, to attack a new version of the system, you've realized you need to make some changes to your shellcode:

- In order to frustrate off-the-shelf shellcode, the sysadmin has removed the shell `/bin/sh`; instead you need to run a different shell, `/bin/dash`.
- The `/bin/dash` shell drops privileges if it can tell it is being run setuid, using code like the following:

  ```
  if (getuid() != geteuid())
      setuid(getuid());
  ```

  Therefore, before calling the shell, your shellcode needs to erase the evidence that the process was setuid by changing the real and saved UIDs to root. Looking at the man pages, you figure out you can use the `setresuid` system call to do this:

  ```
  SYNOPSIS
      int setresuid(int ruid, int euid, int suid);
  DESCRIPTION
      setresuid() sets the real user ID, the effective user ID, and the
      saved set-user-ID of the calling process.
  ```

  On Linux/x86, `setresuid` is system call number 208 (0xd0).
- Because of a strange check in the vulnerable program, you can no longer use the `xor` instruction. In fact, only a limited set of instructions (shown on a later page) will work.
- Because the shellcode needs to be passed in an environment variable, it cannot contain any 0x00 bytes.

On the brighter side, because this is a local attack, you can run a non-privileged shell command before the vulnerable program, if that helps.

Your job (on the next page) is to figure out the hex bytes of an appropriately updated shellcode.

(Optional) shell command: _____

____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
____	____	____	____	____	
89	e1				mov %esp, %ecx
____	____	____	____	____	
b0	0b				mov $0xb, %al
____	____	____	____	____	
cd	80				int $0x80

Fill in the blanks with hex codes for bytes. You can use the space on the right to show assembly code or to comment on what the instructions are doing, which may be good for partial credit. But getting the right hex bytes is necessary and sufficient for full credit. Write one instruction, up to 5 bytes, per line. You don't need to use all the rows, and you can leave any unused rows blank. We've filled in three rows to get you started.

The next page has the list of instruction you can use, and some other reference information that may be helpful.

Allowed x86 instructions:

Hex bytes	assembly	comments
01 [regs]	`add %sreg, %dreg`	
09 [regs]	`or %sreg, %dreg`	
21 [regs]	`and %sreg, %dreg`	
29 [regs]	`sub %sreg, %dreg`	
~~31 [regs]~~	~~`xor %sreg, %dreg`~~	no longer allowed
50	`push %eax`	
51	`push %ecx`	
52	`push %edx`	
53	`push %ebx`	
68 [32bits]	`push $0x12345678`	note little-endian
89 [regs]	`mov %sreg, %dreg`	
b0 [byte]	`mov $0x42, %al`	`%al` is the low byte of `%eax`
cd 80	`int $0x80`	

In the AT&T syntax used here, the second operand is the destination. So `mov %eax, %ebx` is like `ebx = eax`, and `sub %eax, %ebx` is like `ebx = ebx - eax`.

Bytes specifying two registers ("[regs]" in table above). Row is the source register, column is the destination register:

	eax	ecx	edx	ebx	esp	ebp	esi	edi
eax	c0	c1	c2	c3	c4	c5	c6	c7
ecx	c8	c9	ca	cb	cc	cd	ce	cf
edx	d0	d1	d2	d3	d4	d5	d6	d7
ebx	d8	d9	da	db	dc	dd	de	df
esp	e0	e1	e2	e3	e4	e5	e6	e7
ebp	e8	e9	ea	eb	ec	ed	ee	ef
esi	f0	f1	f2	f3	f4	f5	f6	f7
edi	f8	f9	fa	fb	fc	fd	fe	ff

ASCII/hex conversion table:

20 SPC	30 0	40 @	50 P	60 ‘	70 p
21 !	31 1	41 A	51 Q	61 a	71 q
22 "	32 2	42 B	52 R	62 b	72 r
23 #	33 3	43 C	53 S	63 c	73 s
24 $	34 4	44 D	54 T	64 d	74 t
25 %	35 5	45 E	55 U	65 e	75 u
26 &	36 6	46 F	56 V	66 f	76 v
27 ’	37 7	47 G	57 W	67 g	77 w
28 (	38 8	48 H	58 X	68 h	78 x
29)	39 9	49 I	59 Y	69 i	79 y
2A *	3A :	4A J	5A Z	6A j	7A z
2B +	3B ;	4B K	5B [	6B k	7B {
2C ,	3C <	4C L	5C \	6C l	7C |
2D -	3D =	4D M	5D]	6D m	7D }
2E .	3E >	4E N	5E ^	6E n	7E ~
2F /	3F ?	4F O	5F _	6F o	

The Linux/x86 system call convention is that the system call number goes in `%eax`, the first argument is in `%ebx`, the second argument is in `%ecx`, and the third argument is in `%edx`.

Your shellcode should work correctly regardless of the initial values of registers, except that you may assume that `%esp` points to a memory area usable as a stack.

The arguments to `execve` are as follows:

```
int execve(char *filename, char **argv, char **envp);
```

`argv` is a list of string pointers to arguments, terminated by a null pointer. `envp` can be null.