CS628A: Midterm Examination

Computer Systems Security (100 points)

23-Feb-2019 (2 Hours)

Instructions: Please be brief and to the point in your answers. Don't just write a lot of vaguely relevant stuff in the hope that some of it will be close to the right answer; we will take points off for the parts that are not correct. You will have to write your answers in the exam booklet, so be sure to write the question number as well.

Do not open the exam booklet until the instructor announces that the exam has started. You **must put down your pens/pencils** when the instructor says that time is up. Disobeying either of these commands will result in getting zero credit for this and all subsequent exams and pop quizzes.

Honor Pledge: Copy the academic integrity pledge below into your answer booklet and sign your name below it. You **can write** the honor pledge before the exam starts.

I pledge my honor that I have faithfully followed the institute's academic integrity policy during this examination.

Good luck!

DO NOT TURN OVER UNTIL THE EXAM STARTS

Q1. An Unfortunate Pair (6)

In this hypothetical story, Bhondoo works for the company DesiPenguin which produces a fortified version of the Linux kernel for use by the Indian defence forces (among others). DesiPenguin has a winter internship program for undergraduate students. This academic year, Bhondoo has been supervising an intern, Rondoo. Rondoo has been learning how to modify the Linux kernel and as part of his experiments, he created these two toy system calls: setint and getint.

```
int x;
void sys_setint(int *p) {
  memcpy(&x, p, sizeof(x));
}
void sys_getint(int *p) {
  memcpy(p, &x, sizeof(x));
}
```

Bhondoo inadvertently merged Rondoo's code with DesiPenguin's main release branch. This resulted in DesiPenguin shipping a version of the Linux kernel with these two system calls added and this modified kernel was being used by the defence forces.

It turns out some auditor in one of the National Cyber Defence Research Centres (NCDRC) found Rondoo's modifications and notified his superiors. For reasons not clearly explained to Bhondoo and Rondoo, the whole cyber-security establishment is now furious with them. So much so that some NCDRC head honchos have appeared on TV and accused them of being Chinese spies. Bhondoo and Rondoo, for their part, can't understand what the fuss is about. This code doesn't seem to do anything at all, and the other syscalls and the rest of the kernel work exactly as before. Sure, the Linux kernel is some 0.0001% slower because of the system call table's larger cache footprint but is that the reason people are angry?

So, who is right? Are these system calls anything to worry about? Explain all reasons for why or why not.

Answer: These two functions are a security disaster. Neither function validates the pointers (p) that it gets from user space. A syscall must make sure that any pointers received from a user process belong to the process's virtual addresses space and that the process has permissions to read/write to these addresses. If you have seen Linux kernel code, you might have seen references to the functions copy_from_user and copy_to_user – these functions perform the above checks. Every user pointer access in the kernel is wrapped by these two functions.

In the absence of these checks, since the pointers can be set to arbitrary values by a malicious process, a combination of setint and getint can be used to read arbitrary kernel memory locations. A malicious process can also use getint to write to arbitrary kernel memory locations. As a result of these syscalls, every security feature in the kernel is now useless.

Q2. do_fallocate (18)

The function below is a modified snippet of code based on the Linux kernel.

```
int do_fallocate(int offset, int len)
{
    static const int MAX_BYTES = INT_MAX;
    if (offset < 0 || len <= 0) return -1;
    if ((offset + len > MAX_BYTES) || (offset + len < 0)) return -2;
    /* do the real work ... (snipped) */
    return 0;
}
```

This code was compiled using:

```
$ gcc -c -O3 open.c
```

The resulting object file was disassembled using:

```
$ objdump -d -M intel open.o
```

Here is the output from objdump.

```
open.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <do_fallocate>:
   0:    c1 ef 1f                shr    edi,0x1f
   3:    85 f6                   test   esi,esi
   5:    0f 9e c0                setle  al
   8:    09 f8                   or     eax,edi
   a:    0f b6 c0                movzx  eax,al
   d:    f7 d8                   neg    eax
   f:    c3                      ret
```

a. There are seven instructions in the compiled code. Explain what each instruction does. (No more than one sentence for each instruction.) (5)

Hints: test <arg>, <arg> performs a bitwise AND operation on the arguments and sets the flags based on the result. It always clears the OF and sets the SF if the MSB of the result is 1. setle's result is 1 if ZF = 1 (thanks to Harikrishnan Balagopal for catching this typo) or OF != SF.

Answer:

- shr is shift right
- movzx is mov with zero extend; used when copying a small bit length to a larger bit length
- neg is negate

The instructions are equivalent to:

```
edi = edi >> 31              // preserves only the sign bit of edi
al = esi <= 0 ? 1 : 0        // checks if esi <= 0
eax = eax | edi
```

```
eax = ( 0xff & eax ) | al          // clear the upper 24 bits of EAX
eax = -eax
return
```

The first five instructions set eax = 1 if edi < 0 || esi <= 0. Remember that edi and esi are the two arguments to the function (offset and len respectively). Hence, the first six instructions correspond to the first if statement.

And then we return! (Return value is in eax.)

Grading note: Since there was a typo in the hint for this question, everybody will get 2 points.

b. What is the return value of do_fallocate(0x7fffffff, 0x7fffffff)? (6)

Answer: This is a trick question because the answer is different for the C and the assembly code.

The problem is that the compiler is smart enough to figure out that when the second if statement is executed, offset and len are both >= 0. The compiler now deduces that the only way offset + len < 0 can occur is if the result overflows. According to the C standard, the result of integer overflow is undefined behaviour.

Therefore, the correct answer, according to the C standard, is that result of do_fallocate(0x7fffffff, 0x7fffffff) is undefined behaviour. That said, I will give credit to those who say that the result should be -2 according to the C code. But even that is only half the answer!

As we saw in part (a) above, the second if has disappeared in the assembly code. So according to the assembly code the result will be 0.

c. Do you see anything surprising in the compiled code? If so, explain what it is that surprises you and why this behavior occurred. (5)

Answer: Yes! The second if statement has disappeared in the compiled code because of undefined behaviour of integer overflows.

d. Does the above code have any vulnerability? If so, name its type. (2)

Answer: It would seem that the programmer was careful to check for overflows and so the code should be secure. But as we saw above, undefined behaviour has eliminated one of our checks and made a seemingly secure program insecure! The compiled code has an integer overflow vulnerability.

Q3. Of cats and mem (7)

The standard C library provides memcmp, memchr and memcpy as analogues of strcmp, strchr and strcpy. Unfortunately, there is no memcat. Our friend Bhondoo searched on the internet for implementations of memcat and found these two snippets of code.

```
void* memcat1(void *dst, size_t dstsz,
              const void* buf1, size_t sz1,
              const void* buf2, size_t sz2)
```

```
{
    if(sz1 < dstsz - sz2) {
        memcpy(dst, buf1, sz1);
        memcpy((char*)dst+sz1, buf2, sz2);
        return dst;
    }
    return NULL;
}

void* memcat2(void *dst, size_t dstsz,
              const void* buf1, size_t sz1,
              const void* buf2, size_t sz2)
{
    if(sz1 + sz2 < dstsz)  {
        memcpy(dst, buf1, sz1);
        memcpy((char*)dst+sz1, buf2, sz2);
        return dst;
    }
    return NULL;
}
```

a. Are both implementations equivalent? (1)
Answer: No

b. If they are not, are both equally secure? (2)

Answer: No. 1 is good, 2 is bad. 2 has integer and buffer overflow vulnerabilities. According to the C standard, the result of unsigned subtraction is well-defined and has wrap-around behaviour, so memcat1 is secure.

c. If one (or both!) of the options are insecure, briefly explain why. (2)

Answer: memcat2 has an integer overflow problem. If we have sz1=sz2=0x8000000000000040, and dstsz=0x100, memcat1 will have a massive buffer overflow.

d. If one (or both!) of the options are secure, will there be a security issue introduced if the cast to char* is changed to be a cast to int* in it? Explain why or why not. (2)

Answer: memcat1 will also become insecure if (char*) is changed to (int*). Pointer arithmetic rules mean that for some type T, (T*)ptr + sz is the address ptr + sizeof(T)* sz. If T is int, we could be writing past the end of the destination buffer.

Q4. RedHat lpr Redux (10)

Recall that the printing program lpr shipped with Red Hat Linux versions 4.2 to 6.1 had a bug which is shown in the following snippet of code. Note: lpr used to run as suid root and could be invoked by arbitrary users to print documents.

```
if (access(filename, R_OK) == 0) {
    int fd = open(filename, O_RDONLY);
    send_to_printer(fd);
}
```

Hint: access checks whether the real UID associated with the process has the permissions to open filename, while open uses the effective UID.

a. What was the bug? (2)

Answer: Time of check to time of use. The attacker could create a symbolic link from /tmp/a to /tmp/b.txt, and then later change the link to point to /etc/shadow. If the attacker is able to time these operations such that the check in access uses the old link, while the open uses the new link, they will be able to print arbitrary files.

b. Here is one proposal to fix the bug. Comment on the proposal. (3)

```
pthread_mutex_lock(&lock);
if (access(filename, R_OK) == 0) {
    int fd = open(filename, O_RDONLY);
    pthread_mutex_unlock(&lock);
    send_to_printer(fd);
}
```

Answer: The locks are useless and cannot prevent the TOCTOU bug. They are not even being used correctly, but that doesn't matter because they wouldn't do anything even if they were used correctly.

c. Explain how you would fix the bug. (See the hint above!) (5)

Answer: Just use open with reduced privileges.

1. First save the effective uid in the saved uid.
2. Set the effective uid to be same as the real uid.
3. Open the file. This fails if the invoking user does not have permission to access the file.
4. If the open succeeds, restore the saved uid to the effective uid. Otherwise quit.

Q5. OTP Woes (8)

We return to help our friend Bhondoo who has now been given the task of implementing a one-time password (OTP) that will be sent via SMS when logging into DesiPenguin's webmail. He googles around and finds that OTPs are implemented by: (i) having a counter on the server that starts at 1 and increments

for each login, and (ii) hashing some combination of a secret (possibly based on the user's password) and the counter to generate an OTP for each login.

Based on the above, Bhondoo has come up with three design choices for an OTP.

i. $OTP = SHA256(secret \,||\, str(counter))$

Notation $A \,||\, B$ means A concatenated with B.

ii. $OTP = SHA256^{counter}(secret)$

Notation $SHA256^n(x)$ means the function is applied repeatedly n times. Precisely stated, $SHA256^1(x) = SHA256(x)$ while $SHA256^n(x) = SHA256(SHA256^{n-1}(x))$ when $n > 1$.

iii. $OTP = SHA256^{max-counter}(secret)$

Here max is some very large number (say 1 million) and the assumption is that max is the maximum number of logins (i.e., more than 1 million logins is not allowed – maybe after 1 million logins, the user is forced to set a new password/secret, and the counter is reset).

Evaluate the security of each of these options. If any of the options are insecure, discuss how they can be attacked. If the options are secure, briefly discuss why.

Note: Don't worry about the length of the hash. DesiPenguin will be sending the hash/OTP as a QR code and a webcam will scan the QR code, so the user doesn't need to type in 64 hex characters/256 bits.

Answer: i and ii are insecure. iii is fine.

What is the purpose of an OTP? The clue is in the name: one-time password. The password must be usable only once, and never again. From that perspective, (i) is problematic because if the adversary somehow gets hold of one hash, they can use length-extension attacks to compute other OTPs.

The second option is also problematic because again if an adversary gets holds of one OTP, they can compute all subsequent OTPs.

The third one is secure. Even if the adversary gets hold of an OTP which is $SHA256^n(secret)$. They have no way of computing $SHA256^{n-1}(secret)$ or $SHA256^{n-2}(secret)$ or any other subsequent OTP because hash functions are one-way (i.e., they are resistant to preimage attacks).

Q6. ptrace for evil (6)

We learnt in class that the ptrace system call can be used by a parent process to intercept all the system calls of a child process. In fact, ptrace can be used for much more than that. Among other things, it can also be used to read/write a child process' memory.

So, here's an idea. We'll create a program which forks, enables tracing using ptrace, and thens run `execve("/usr/bin/sudo", …, …)` in the child. Later, when the sudo child process reads `/etc/shadow` and compares the typed-in password with the stored password, the parent will modify child memory (i.e., sudo's memory) to say that the comparison succeeded. This attack seemingly allows anyone to become root.

a. Does the above attack work? (Yes/No) (2)

Answer: No.

b. If it doesn't work, which of the below steps will fail: (4a)

i. execve of sudo
ii. modification of sudo's memory
iii. something else entirely (explain what)

Answer: If you try the attack, you will find that sudo prints an error message about not being effective root and then exits.

What is happening underneath?

1. When execve runs a setuid program, and tracing is enabled, the uid doesn't change and the process runs with the same uid as the original program.
2. This means sudo doesn't get effective root. Sudo has code to detect this so it prints an error message saying "`effective uid is not 0. Is sudo installed setuid root?`" and exits.
3. Even if sudo didn't detect that it wasn't setuid root, no damage is done because the call to open /etc/shadow would fail, the calls to setuid to escalate privilege later in the program would also fail.

The relevant parts from the `execve` manpage (http://man7.org/linux/man-pages/man2/execve.2.html) are:

If the set-user-ID bit is set on the program file pointed to by filename, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, when the set- group-ID bit of the program file is set the effective group ID of the calling process is set to the group of the program file.

The aforementioned transformations of the effective IDs are not performed (i.e., the set-user-ID and set-group-ID bits are ignored) if any of the following is true:

- the no_new_privs attribute is set for the calling thread (see prctl(2));
- the underlying filesystem is mounted nosuid (the MS_NOSUID flag for mount(2)); or
- **the calling process is being ptraced**.

c. If the above attack works, how are such attacks prevented in modern systems? (4b)

i. sudo has code to detect if it is being traced
ii. in the default configuration, only root can execute ptrace
iii. something else entirely (explain what)

Answer: Not applicable

Q7. Racing and Systrace (24)

A system call interposition framework has to deal with many kinds of race conditions caused by multithreaded and multiprocess applications. Two specific types of races to worry about are symbolic link races and current directory races.

a. Explain these two types of races and how they lead to attacks (8)
   - Draw a timeline showing how these races could potentially be exploited.
   - Be sure to clearly mention which entities are racing (processes or threads).

   Answer: Symbolic link races involve two processes. Suppose a process is only allowed to read files within /tmp/ but wants to somehow open /etc/passwd. A symlink race attack would work in the following way:

1. Process A creates a symlink from /tmp/a to /tmp/b.txt	
2. Process A opens /tmp/a	
3. The syscall interposition framework checks whether opening /tmp/a is allowed. Reponse is “yes” because the target is /tmp/b.txt.	
	4. Process B changes /tmp/a to point to /etc/passwd
5. The syscall is now executed and opens /etc/passwd.	

   Current directory races involve two different threads within the same application. (Why?) They work like this:

Thread 1 sets current directory to /tmp	Thread 2
It creates a subdirectory /tmp/etc and a file /tmp/etc/passwd	
It then attempts to open “./etc/passwd”	
Syscall interposition framework does a policy check and allows the call because /tmp/etc/passwd can be opened by the application	
	Thread 2 changes current directory to /
Now the open is actually executed and ends up opening /etc/passwd	

b. How are these two types of races prevented in systrace? (6)

   Answer:

- Both types of races are prevented by normalization: this involves resolving all symbolic links and then expanding relative paths to absolute paths.
- Current directory races have an additional layer of protection. For each system call, systrace fully finishes executing it before the next system call is processed. In the above code, there are actually two syscalls – open and change directory, and they are being executed concurrently, which leads to the race condition. Systrace will do the change directory only after the open syscall has finished, so there won't be a race condition.

c. The systrace papers says: "Instead of placing the rewritten arguments on the stack as done in MAPbox [1], we provide a read-only look-aside buffer in the kernel." What is this buffer for? Why is it required? (4)

Answer: Arguments are copied into this buffer in the kernel to prevent the process from changing them after the policy check and before the execution of the syscall.

d. When are arguments placed into this buffer? Before or after the policy check? (2)

Answer: Must be copied before, otherwise it would serve no purpose.

e. Is systrace still vulnerable to any race conditions? (4)

Hint: Think about what arguments are passed to the syscall in `open("/etc/passwd", O_RDWR)`. Is there some way of exploiting this?

Answer: Yes, see the paper "Exploiting Concurrency Vulnerabilities in System Call Wrappers" for details. Slides are here: http://www.watson.org/~robert/2007woot/20070806-woot-concurrency.pdf. The short version is that although the arguments are copied, what if the arguments are pointers? We could change what the pointer points to, even if the pointer itself is kept the same.

Q8. Software Fault Isolation (10)

a. Rewrite this function so that it can be executed in a Native Client sandbox. (3)

```
do_fallocate:
      shr    edi,0x1f
      test   esi,esi
      setle  al
      or     eax,edi
      movzx  eax,al
      neg    eax
      ret
```

Answer: We only need to change the return instruction to use the nacljmp pseudoinstruction. The function now looks like:

```
do_fallocate:
      shr    edi,0x1f
```

```
test   esi,esi
setle  al
or     eax,edi
movzx  eax,al
neg    eax
pop    ecx
and    ecx, 0xfffffffe0
jmp    ecx
```

b. Why is the code in sub-part (a) not directly allowed in NaCl? (2)

Answer: All jumps (this includes returns) in NaCl must be to 16b aligned targets.

c. Name three types of instructions that can't be executed inside of NaCl sandboxes. (3)

Answer:

- Returns
- Segment register modifications
- System calls
- Privileged instructions (mov cr0, etc.)

d. Mention one operation that can be done within a chroot jail, but not in a NaCl sandbox. (2)

Answer: two examples are system calls, segment register modifications.

Q9. Rapid Fire Round (11)

a. When running within a VMM, how is a malicious operating system prevented from modifying the page tables to access to some other victim operating system's memory? (2)

Answer: There are a couple of different techniques used. One is to make the page tables write protected, so that when the OS writes to them, it results in a trap which is handled by the VMM. The second is the idea of nested page tables, where the OS's page tables go through another level of translation which is controlled by the VMM.

b. True/False: A format string vulnerability only allows attacker to read victim memory. In other words, only confidentiality can be breached, the control-flow can't be modified. (2)

Answer: False. The "%n" format specifier can write values onto the stack.

c. What is the significance of the canary value `0x000aff0d` used by StackGuard v2.0.1? (2)

Answer: The 0x00 stops `strcpy` etc. from copying past this delimiter. The 0xa stops the function `gets` from copying past this delimiter. If an attacker is trying to overwrite the return address by overflow using a buffer which also contains a correct canary, this will not work because both `strcpy` and `gets` will stop copying beyond the canary. (Recall that the canary is at a lower address value than the return address, so it needs to be copied first.)

d. Give one example of how a user-mode rootkit would be implemented. (2)

Answer: You could replace ls with a program that hides certain files. You could replace ps with a program that doesn't show certain running processes. You could replace a commonly used program (e.g. web browser) with a compromised version that logs all user input (e.g., passwords).

e. Why do kernel rootkits modify ("hook") the system call table? (2)

Answer: They need to intercept system calls in order to modify their results. They do this by making the system call table point to rootkit code; the rootkit code in turn calls the "real" system call to do the work, examines the results, potentially modifies them and returns.

f. One suggestion for improving the class. (1)

(END OF EXAM)