CS 380S - Theory and Practice of Secure Systems
Fall 2009

MIDTERM

October 22, 2009

DO NOT OPEN UNTIL INSTRUCTED

YOUR NAME: _____

Collaboration policy

No collaboration is permitted on this midterm. Any cheating (*e.g.*, submitting another person's work as your own, or permitting your work to be copied) will automatically result in a failing grade. The Computer Sciences department code of conduct can be found at `http://www.cs.utexas.edu/users/ear/CodeOfConduct.html`

Midterm (45 points)

Problem 1 (3 points)

Why is calling `free()` twice on the same memory object in a C program a potential security problem?

Problem 2

Consider the following password-checking code written in C (similar code was used in early versions of Unix). Assume that all passwords are 7 characters, and the code is compiled using StackGuard.

```
#define LEN 8

void check_pwd(char *name, pwd) {
   int flag=0;
   char buf1[LEN];
   char buf2[LEN];
   /* code that places actual pwd for given name into buf1 (omitted) */
   strcpy(buf2,pwd);
   if (!memcmp(buf1,buf2,LEN)) {
          /* allow login */ }
   else { /* deny login */ }
}
```

Problem 2a (2 points)

Explain a simple attack that allows anyone to log in under a given username.

Problem 2b (2 points)

If this code is executed with standard Libsafe, will the attack be prevented? Explain.

Problem 3

In the DEC Alpha assembly language, all instructions are 4-bytes wide and must start on an aligned 4-byte boundary. Here are some examples:

- `br` *Ra*, *disp*
  An unconditional relative branch. This instruction stores the address of the next instruction in *Ra* and then skips *disp* instructions, where *disp* may be negative. For example, `br r13, -5` jumps back 5 instructions (this may happen in a loop, for example).

- `jmp` *Ra*, (*Rb*)
  Jump to register. Stores the address of the next instruction in *Ra*, then jumps and starts executing code at address *Rb*.

- `ldq` *Rv*, *disp* (*Ra*)
  Load. Takes the memory address contained in register *Ra*, adds *disp* to it, and loads the value of the memory location at this address into register *Rv*.

- `stq` *Rv*, *disp* (*Ra*)
  Store. Takes the memory address contained in register *Ra*, adds *disp* to it, and stores the value of register *Rv* into the memory location at this address.

- `bis` *Ra*, *Rb*, *Rc*
  Compute bitwise OR of *Ra* and *Rb* and store it into *Rc*.

- `and` *Ra*, *Rb*, *Rc*
  Compute bitwise AND of *Ra* and *Rb* and store it into *Rc*.

Problem 3a (3 points)

Fault isolation requires inserting special checking code before every *unsafe* instruction, *i.e.*, an instruction that may potentially write or execute memory outside the fault domain. For example, a store instruction `stq` *Ra*, 0(*Rb*) is unsafe if it cannot be statically checked that the address contained in *Rb* is within the fault domain's data segment.

In the following list, circle the instruction(s) which can be unsafe:

- `br` *Ra*, *disp* where *disp* falls within the fault domain's code segment.
- `jmp` *Ra*, (*Rb*)
- `ldq` *Rv*, *disp* (*Ra*)
- `bis` *Ra*, *Rb*, *Rc*

Problem 3b (2 points)

Suppose that the unsafe store instructions are "sandboxed" as follows. We use dedicated registers $r20$ and $r21$ to store, in the positions corresponding to the segment identifier part of a memory address, all-zero bits and the segment ID bits, respectively. If the code contains an unsafe store instruction `stq r2, 0(r1)`, it is replaced by the following three instructions:

```
and r1, r20, r1
bis r1, r21, r1
stq r2, 0(r1)
```

How can you subvert the safety of the system that uses this sandboxing mechanism?

Problem 3c (3 points)

Suppose communication between fault domains is implemented as follows. For each fault domain, the trusted execution environment inserts special "stubs" (little snippets of code) into a special region of that domain's code segment. Because the code of the stubs is trusted, it may contain unsafe instructions. Furthermore, the stubs are the only part of the fault domain's code segment that is allowed to have instructions branching outside of this code segment.

When a trusted caller calls an untrusted function, it branches to the "entry" stub, which copies arguments, saves registers that must be changed when switching fault domains, and passes control to the untrusted code. When the untrusted code returns, it jumps directly to the "return" stub in its code segment, which restores the context and returns to the caller.

How can you subvert the safety of the system that uses this cross-domain communication mechanism?

Problem 3d (3 points)

How should you implement the "stubs" for cross-domain communication so that they cannot be subverted? You may explain or draw a picture.

Problem 4 (4 points)

Consider a host-based intrusion detection system (IDS) that combines stack monitoring with path profiling. During the training phase, the IDS observes a large number of program executions and collects a library of all pairs of sequential function calls ever made by the program (*e.g.*, it knows that function `mysetuid` often calls function `setuid`).

After the IDS is activated, it monitors the stack during program execution. If the IDS detects that the program made a sequence of function calls that was never observed during the training phase, it raises an alarm.

Give a snippet of C code that the attacker can exploit using a standard return address overflow without triggering an alarm. Your code must include at least two user-defined functions. Explain how the attack manages to evade the IDS.

Problem 5

A Unix process may call another process without fully trusting it. In this situation, the caller may want the called process to have access only to the objects that the caller explicitly passes to it, and not to arbitrary files owned by the caller. One possible solution is to create a restricted user ID, and execute the called process under this restricted UID.

Problem 5a (2 points)

Every UNIX process has a Real UID (RUID) and an Effective UID (EUID). What is the difference between the RUID and the EUID? How is each one used by the OS?

Problem 5b (2 points)

In some flavors of UNIX, any process can use `setuid()` to set its EUID to RUID. Are there any security implications for the situation described above, where one process calls another under a restricted UID? Assume that either the caller, or the callee may be malicious.

Problem 6 (6 points)

Imagine a MOPS-like tool for checking source code to ensure that it satisfies a certain set of rules. Each rule is expressed by a finite-state automaton, with a special ERROR state. As the checker scans the code, it keeps track of the current state in the automaton. If a state labelled ERROR is ever reached, then the checker reports an error in the code.

Draw finite-state automata representing the following security rules. If you believe the rule cannot be expressed by a finite-state automaton, explain why.

- Immediately before each call to `strcpy(dest,src)`, the program must check the length of `src` by calling `strlen(src)`.

- Each temporary file used by the program must be created using `mkstemp()`, written, and eventually closed.

- The return value of every call to `malloc` must be immediately checked to ensure that it is not NULL.

Problem 7

Consider the following PHP script for logging into a website:

```
$username = $_GET[user];
$password = $_GET[pwd];
$sql = "SELECT * FROM usertable
        WHERE username= '$username' AND password = '$password' ";
$result = $db->query($sql);
if ($result->num_rows > 0) { /* successful login */ }
```

```
else                          { /* login failed */ }
```

Problem 7a (2 points)

Give an example of a username that will successfully subvert the above authentication code.

Problem 7b (3 points)

The PHP function `addslashes` makes a pass through the string and adds a slash before every quote it encounters (unless it is already preceded by a single slash). For example, `addslashes(x'y)` outputs the string `x\'y`.

Suppose user's input is sanitized as follows:

```
$username = addslashes($_GET[user]);
$password = addslashes($_GET[pwd]);
```

In the Chinese, Korean, and Japanese unicode character sets, some characters are encoded as single bytes, while others are double bytes. For example, the database interprets `0x5C` as \, `0x27` as ', `0x5C27` as \', but `0xBF5C` is interpreted as a single Chinese character. For unicode inputs, `addslashes` adds `5C` before every `27` it encounters (unless it's already there).

Give an example of a username that will successfully subvert the above authentication code even if the input is sanitized using `addslashes`.

Problem 7c (3 points)

How should `addslashes` be implemented to prevent SQL injection attacks?

Problem 8 (3 points)

What is the right way for the user to think of the browser lock icon, *i.e.*, what kind of attacks should not be possible if the lock icon is displayed?

Problem 9 (2 points)

Write a program in pseudo-code that (i) will pass verification in the Myers-Liskov model for decentralized information flow control (as described in their SOSP 1997 paper), yet (ii) contains an information channel which leaks the value of some sensitive variable. Show the labels associated with variables at each point of your program.