

CS 380S - Theory and Practice of Secure Systems
Fall 2008

MIDTERM

October 28, 2008

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

Consider the following C code fragment:

```
int foo(int i, double *data1, double data2) {
    double *p = data1;
    double *datavec[10];
    if ((i < 0) || (i > 10)) return;
    datavec[i] = data1;
    *p = data2;
}
```

Problem 1a (2 points)

Explain how to stage a control-hijacking attack on this code.

Problem 1b (2 points)

If this code is compiled with StackGuard, will the attack be prevented? Explain.

Problem 1c (2 points)

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

Problem 2 (5 points)

For the following snippet of C code, write the constraints on `len` and `alloc` of all string variables that might be inferred by the BOON tool (described in the Wagner *et al.*'s paper "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities"). Write each constraint next to the line where it is inferred.

```
1:  int lg=0;
2:  char msg[9] = "DEFAULT";
3:  char log[256] = "";
4:  if (lg) {
5:      strcpy(log, msg);
6:  } else {
7:      log[0] = '\0';
8:  }
9:  if (!lg) {
10:     strcpy(msg, log);
11:  }
```

Will BOON indicate that this code is potentially vulnerable to a buffer overflow attack? If you think it will, which line in the code will BOON consider potentially vulnerable? Why?

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. 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 (2 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 system call monitor enforcing the following policy for multi-threaded applications, where some of the threads may be untrusted. When a thread tries to open a file, the monitor looks for the file name in the list of files that the application is allowed to access (this list is securely kept in the kernel memory). If the name is not on the list, the monitor blocks the call. Otherwise, the monitor executes the call and returns the file descriptor to the thread.

Can the application subvert this reference monitor? Explain why or why not.

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)

What happens to the RUID and EUID when the caller's process activates an executable file whose uid is set to the callee's user ID?

Problem 5c (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()`, opened, 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` adds a slash before every quote. 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.

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?