

CS 380S - 0x1A Great Papers in Computer Security
Fall 2011

MIDTERM

October 27, 2011

DO NOT OPEN UNTIL INSTRUCTED

YOUR NAME: _____

Collaboration policy

No **collaboration** is permitted on this assignment. 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.

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 2 (2 points)

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

Problem 3 (2 points)

In Native Client, *springboards* are snippets of trusted code which are located in the memory of the untrusted binary module. Their purpose is to enable control transfers from the trusted runtime environment to untrusted code. Because the springboard code is trusted, it may include privileged instructions that are not normally available to the untrusted code. What prevents untrusted code from executing these instructions by passing control—via either a jump, or sequential execution—to the springboard code located in its memory?

Problem 4

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 4a (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 4b (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 4c (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 4d (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 5 (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 6 (6 points)

Imagine a static-analysis tool for checking source C 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 (8 points)

Integrity is an important element of an information flow policy. Suppose there are two levels of integrity, T for Trusted and U for Untrusted. Intuitively, untrusted data should not be allowed to corrupt trusted data. That is, data from untrusted variables should not be allowed to flow to trusted variables.

Examine the following four statements, which have integrity labels as subscripts on variables. Explain which statements are secure, which are insecure, and why.

1. $X_T := Y_T + Z_U$

2. $V_U := Y_T + Z_U$

3. if X_T then $Y_T := X_T$ else $V_U := Z_U$

4. if V_U then $Y_T := X_T$ else $V_U := Z_U$

Problem 8 (4 points)

Denning and Sacco proposed the following protocol, which enables Alice and Bob to establish a shared symmetric key K with the help of a trusted public-key certificate directory S . Alice contacts the directory and obtains the certificates for her own public key $pk(A)$ and Bob's public key $pk(B)$. Alice then generates a fresh random key K , signs it together with the current timestamp, and sends the result to Bob, encrypted with Bob's public key and accompanied by the two certificates. Bob decrypts the message and verifies Alice's signature using the public key contained in Alice's certificate. If verification succeeds and the timestamp is recent, Bob concludes that he now shares key K with Alice.

The protocol is summarized below:

<i>Alice</i>	\rightarrow	<i>Directory</i>	A, B
<i>Directory</i>	\rightarrow	<i>Alice</i>	$cert_A, cert_B$
<i>Alice</i>	\rightarrow	<i>Bob</i>	$cert_A, cert_B, encrypt_{pk(B)}(sig_{pk(A)}(K, time_A))$

You can assume that public-key certificates are signed by a trusted certificate authority, and that Alice's and Bob's clocks are synchronized.

How can this protocol be used to impersonate Alice to a third party, Charlie?

Problem 9 (4 points)

Tatebayashi, Matsuzaki, and Newman (TMN) proposed the following protocol, which enables Alice and Bob to establish a shared symmetric key K with the help of a trusted server S . Both Alice and Bob know the server's public key K_S . Alice randomly generates a temporary secret K_A , while Bob randomly generates a new key K to be shared with Alice. The protocol then proceeds as follows:

$$\begin{array}{lll} \textit{Alice} & \rightarrow & \textit{Server} \quad \textit{enc}_{K_S}(K_A) \\ \textit{Bob} & \rightarrow & \textit{Server} \quad \textit{enc}_{K_S}(K) \\ \textit{Server} & \rightarrow & \textit{Alice} \quad K \oplus K_A \\ & & \textit{Alice recovers key } K \textit{ as } K_A \oplus (K \oplus K_A) \end{array}$$

In this protocol, Alice sends her secret to the Server encrypted with the Server's public key, while Bob sends to the Server the new key, also encrypted with the Server's public key. The Server XORs the two values together and sends the result to Alice. Therefore, both Alice and Bob know K .

Suppose that evil Charlie eavesdropped on Bob's message to the Server. How can he, with the help of his equally evil buddy Don, extract the key K that Alice and Bob are using to protect their communications?

Assume that Charlie and Don can engage in the TMN protocol with the Server, but they don't know the Server's private key.