Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.893 Fall 2009

Quiz I Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/18)	II (xx/16)	III (xx/18)	IV (xx/8)	V (xx/18)
VI (xx/8)	**VII (xx/8)**	**VIII (xx/6)**	**Total (xx/100)**	

The mean score on the quiz was 67, median was 62, and standard deviation 15.

Name:

I Buffer Overflows

Ben Bitdiddle is building a web server that runs the following code sequence, in which `process_req()` is invoked with a user-supplied string of arbitrary length. Assume that `process_get()` is safe, and for the purposes of this question, simply returns right away.

```
void process_req(char *input) {
    char buf[256];
    strcpy(buf, input);
    if (!strncmp(buf, "GET ", 4))
        process_get(buf);
    return;
}
```

1. [6 points]: Ben Bitdiddle wants to prevent attackers from exploiting bugs in his server, so he decides to make the stack memory non-executable. Explain how an attacker can still exploit a buffer overflow in his code to delete files on the server. Draw a stack diagram to show what locations on the stack you need to control, what values you propose to write there, and where in the input string these values need to be located.

Answer: An attacker can still take control of Ben's server, and in particular, remove files, by using a "return-to-libc" attack, where the return address is overflowed with the address of the `unlink` function in libc. The attacker must also arrange for the stack to contain proper arguments for `unlink`, at the right location on the stack.

2. [6 points]: Seeing the difficulty of preventing exploits with a non-executable stack, Ben instead decides to make the stack grow up (towards larger addresses), instead of down like on the x86. Explain how you could exploit `process_req()` to execute arbitrary code. Draw a stack diagram to illustrate what locations on the stack you plan to corrupt, and where in the input string you would need to place the desired values.

Answer: The return address from the `strcpy` function is on the stack following the `buf` array. If the attacker provides an input longer than 256 bytes, the subsequent bytes can overwrite the return address from `strcpy`, vectoring the execution of the program to an arbitrary address when `strcpy` returns.

3. [6 points]: Consider the StackGuard system from the "Buffer Overflows" paper in the context of Ben's new system where the stack grows *up*. Explain where on the stack the canary should be placed, at what points in the code the canary should be written, and at what points it should be checked, to prevent buffer overflow exploits that take control of the return address.

Answer: The canary must be placed at an address immediately before each function's return address on the stack. Because the stack grows up, this space must be reserved by the caller (although it's OK if the callee puts the canary value there, before executing any code that might overflow the stack and corrupt the return address). The callee must verify the canary value before returning to the caller.

II XFI

4. [2 points]: Suppose a program has a traditional buffer overflow vulnerability where the attacker can overwrite the return address on the stack. Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

Answer: Because XFI has two stacks, the attacker will not be able to exploit a traditional buffer overflow (corrupting the return address). The attacker may still be able to corrupt other data or pointers on the allocation stack; see below.

5. [4 points]: Suppose a program has a buffer overflow vulnerability which allows an attacker to overwrite a function pointer on the stack (which is invoked shortly after the buffer is overflowed). Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

Answer: The attacker can cause the module to start executing the start of any legal function in the XFI module, or any legal stub that will in turn execute allowed external functions.

6. [4 points]: Suppose a malicious XFI module, which is not allowed to invoke `unlink()`, attempts to remove arbitrary files by directly jumping to the `unlink()` code in `libc`. What precise instruction will fail when the attacker tries to do so, if any?

Answer: The CFI label check before the jump to `unlink` will notice that the `unlink` function does not have the appropriate CFI label, and abort execution.

7. [6 points]: Suppose a malicious XFI module wants to circumvent XFI's inline checks in its code. To do so, the module allocates a large chunk of memory, copies its own executable code to it (assume XFI is running with only write-protection enabled, for performance reasons, so the module is allowed to read its own code), and replaces all XFI check instructions in the copied code with `NOP` instructions. The malicious module then calls a function pointer, whose value is the start of the copied version of the function that the module would ordinarily invoke. Does XFI prevent an attacker from bypassing XFI's checks in this manner, and if so, what precise instruction would fail?

Answer: XFI assumes and relies on the hardware/OS to prevent execution of data memory (e.g. the NX flag on recent x86 CPUs).

III Privilege Separation

8. [4 points]: OKWS uses database proxies to control what data each service can access, but lab 2 has no database proxies. Explain what controls the data that each service can access in lab 2.

Answer: Lab 2 relies on file permissions (and data partitioning) to control what service can access what data.

9. [8 points]: In lab 2, logging is implemented by a persistent process that runs under a separate UID and accepts log messages, so that an attacker that compromises other parts of the application would not be able to corrupt the log. Ben Bitdiddle dislikes long-running processes, but still wants to protect the log from attackers. Suggest an alternative design for Ben that makes sure past log messages cannot be tampered with by an attacker, but does not assume the existence of any long-running user process.

Answer: One approach may be to use a setuid binary that will execute the logging service on-demand under the appropriate user ID.

10. [6 points]: Ben proposes another strawman alternative to OKWS: simply use `chroot()` to run each service process in a separate directory root. Since each process will only be able to access its own files, there is no need to run each process under a separate UID. Explain why Ben's approach is faulty, and how an attacker that compromises one service will be able to compromise other services too.

Answer: Processes running under the same UID can still kill or debug each other, even though they cannot interact through the file system.

IV Information Flow Control

11. [8 points]: This problem was buggy; everyone received full credit.

V Java

12. [4 points]: When a privileged operation is requested, extended stack introspection walks up the stack looking for a stack frame which called `enablePrivilege()`, but stops at the first stack frame that is not authorized to call `enablePrivilege()`. Give an example of an attack that could occur if stack inspection did not stop at such stack frames.

Answer: A luring attack, whereby trusted code that has called `enablePrivilege()` accidentally invokes untrusted code, which can then perform privileged operations.

13. [8 points]: Suppose you wanted to run an applet and allow it to connect over the network to web.mit.edu port 80, but nowhere else. In Java, opening a network connection is done by constructing a Socket object, passing the host and port as arguments to the constructor. Sketch out how you would implement this security policy with extended stack introspection, assuming that the system library implementing sockets calls `checkPrivilege("socket")` in the Socket constructor. Explain how the applet must change, if any.

Answer: Something like the following code:

```
public class MitSocketFactory {
    public static Socket getSocket() {
        enablePrivilege("socket");
        return new Socket("web.mit.edu", 80);
    }
}
```

The applet's code would need to invoke `MitSocketFactory.getSocket()` instead of using the `Socket` constructor directly.

14. [6 points]: Sketch out how you would implement the same security policy as in part (b), except by using name space management. Explain how the applet must change, if any.

Answer: Replace the Socket object with MitSocket in the applet's namespace:

```
public class MitSocket extends Socket {
    public MitSocket(String host, int port) {
        super();
        if (!host.equal("web.mit.edu") || port != 80)
            throw SecurityException("only web.mit.edu:80 allowed");
        connect(host, port);
    }

    ...
}
```

The applet would not have to change. Note that `MitSocket`'s constructor does not call `super(host, port)`. Instead, it invokes `super()` and calls `connect(host, port)` later. Invoking the `super(host, port)` constructor would have allowed the applet to open connections to arbitrary hosts (which would then be immediately closed), by constructing an `MitSocket` object with the right `host` and `port` arguments, since the security check comes after the superclass constructor.

VI Browser

15. [8 points]: The paper argues that the child policy (where a frame can navigate its immediate children) is unnecessarily strict, and that the descendant policy (where a frame can navigate the children of its children's frames, and so on) is just as good. Give an example of how the descendant policy can lead to security problems that the child policy avoids.

Answer: Consider a web site that contains a login frame, where users are expected to input passwords. Under the descedant policy, an attacker can put the web site in a frame, and navigate the login frame (a grandchild) to http://attacker.com, which looks similar to the original one, to steal passwords.

VII Resin

16. [8 points]: Sketch out the Resin filter and policy objects that would be needed to avoid cross-site scripting attacks through user profiles in `zoobar`. Assume that you have a PHP function to strip out JavaScript.

Answer: There are several possible solutions. One approach is to define two "empty" policies, *UnsafePolicy* and *JSSanitizedPolicy*, the *export_check* functions of which do nothing. Input strings are tagged *UnsafePolicy*, and the PHP function to strip out JavaScript attaches *JSSanitizedPolicy* to resulting strings. The standard output filter checks that strings must contain neither or both policies.

VIII 6.893

We'd like to hear your opinions about 6.893, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

17. [2 points]: How could we make the ideas in the course easier to understand?

More and simpler examples to illustrate the problem;
More labs.

18. [2 points]: What is the best aspect of 6.893?

Labs;
Recent papers.

19. [2 points]: What is the worst aspect of 6.893?

Long, conceptual papers;
Repetitive, time-consuming labs.

End of Quiz