Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2011

Quiz I

You have 80 minutes to answer the questions in this quiz. In order to receive credit you must answer the question as precisely as possible.

Some questions are harder than others, and some questions earn more points than others. You may want to skim 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 and legible. If we can't understand your answer, we can't give you credit!

Write your name on this cover sheet.

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/20)	II (xx/10)	III (xx/16)	IV (xx/22)	V (xx/10)	VI (xx/16)	VII (xx/6)	Total (xx/100)

Name:

Username from handin site:

I XFI

Consider the following assembly code, which zeroes out 256 bytes of memory pointed to by the EAX register. This code will execute under XFI. XFI's allocation stack is not used in this code.

You will need fill in the verification states for this code, which would be required for the verifier to check the safety of this code, along the lines of the example shown in Figure 4 of the XFI paper. Following the example from the paper, possible verification state statements include:

```
valid[regname+const, regname+const)
origSSP = regname+const
retaddr = Mem[regname]
```

where `regname` and `const` are any register names and constant expressions, respectively. Include all verification states necessary to ensure safety of the subsequent instruction, and to ensure that the next verification state is legal.

```
1   x86 instructions        Verification state
2
3   mrguard(EAX, 0, 256)
4                           (1)
5   ECX := EAX      # current pointer
6   EDX := EAX+256  # end of 256-byte array
7
8 loop:
9                           (2)
10  Mem[ECX] := 0
11  ECX := ECX+4
12                          (3)
13  if ECX+4 > EDX, jmp out
14                          (4)
15  jmp loop
16
17 out:
18  ...
```

1. **[5 points]:** What are the verification states needed at location marked (1)?

2. **[5 points]:** What are the verification states needed at location marked (2)?

3. **[5 points]:** What are the verification states needed at location marked (3)?

4. **[5 points]:** What are the verification states needed at location marked (4)?

II ForceHTTPS

5. [10 points]: Suppose `bank.com` uses and enables ForceHTTPS, and has a legitimate SSL certificate signed by Verisign. Which of the following statements are true?

A. True / False ForceHTTPS prevents the user from entering their password on a phishing web site impersonating `bank.com`.

B. True / False ForceHTTPS ensures that the developer of the `bank.com` web site cannot accidentally load Javascript code from another web server using `<SCRIPT SRC=...>`.

C. True / False ForceHTTPS prevents a user from accidentally accepting an SSL certificate for `bank.com` that's not signed by any legitimate CA.

D. True / False ForceHTTPS prevents a browser from accepting an SSL certificate for `bank.com` that's signed by a CA other than Verisign.

III Zoobar security

Ben Bitdiddle is working on lab 2. For his privilege separation, he decided to create a separate database to store each user's zoobar balance (instead of a single database called `zoobars` that stores everyone's balance). He stores the zoobar balance for user `x` in the directory `/jail/zoobar/db/zoobars.x`, and ensures that usernames cannot contain slashes or null characters. When a user first registers, the login service must be able to create this database for the user, so Ben sets the permissions for `/jail/zoobar/db` to 0777.

6. [4 points]: Explain why this design may be a bad idea. Be specific about what an adversary would have to do to take advantage of a weakness in this design.

Ben Bitdiddle is now working on lab 3. He has three user IDs for running server-side code, as suggested in lab 2 (ignoring transfer logging):

- User ID 900 is used to run dynamic python code to handle HTTP requests (via zookfs). The database containing user profiles is writable only by uid 900.
- User ID 901 is used to run the authentication service, which provides an interface to obtain a token given a username and password, and to check if some token for a username is valid. The database containing user passwords and tokens is stored in a DB that is readable and writable only by uid 901.
- User ID 902 is used to run the transfer service, which provides an interface to transfer zoobar credits from user *A* to user *B*, as long as a token for user *A* is provided. The database storing zoobar balances is writable only by uid 902. The transfer service invokes the authentication service to check whether a token is valid.

Recall that to run Python profile code for user *A*, Ben must give the profile code access to *A*'s token (the profile code may want to transfer credits to visitors, and will need this token to invoke the transfer service).

To support Python profiles, Ben adds a new operation to the authentication service's interface, where the caller supplies an argument `username`, the authentication service looks up the profile for `username`, runs the profile's code with a token for `username`, and returns the output of that code.

7. [4 points]: Ben discovers that a bug in the HTTP handling code (running as uid 900) can allow an adversary to steal zoobars from any user. Explain how an adversary can do this in Ben's design.

8. [8 points]: Propose a design change that prevents attackers from stealing zoobars even if they compromise the HTTP handling code. Do not make any changes to the authentication or transfer services (i.e., code running as uid 901 and 902).

IV Baggy bounds checking

Consider a system that runs the following code under the Baggy bounds checking system, as described in the paper by Akritidis et al, with slot_size=16:

```
1  struct sa {
2    char buf[32];
3    void (*f) (void);
4  };
5
6  struct sb {
7    void (*f) (void);
8    char buf[32];
9  };
10
11 void handle(void) {
12   printf("Hello.\n");
13 }
14
15 void buggy(char *buf, void (**f) (void)) {
16   *f = handle;
17   gets(buf);
18   (*f) ();
19 }
20
21 void test1(void) {
22   struct sa x;
23   buggy(x.buf, &x.f);
24 }
25
26 void test2(void) {
27   struct sb x;
28   buggy(x.buf, &x.f);
29 }
30
31 void test3(void) {
32   struct sb y;
33   struct sa x;
34   buggy(x.buf, &y.f);
35 }
36
37 void test4(void) {
38   struct sb x[2];
39   buggy(x[0].buf, &x[1].f);
40 }
```

Assume the compiler performs no optimizations and places variables on the stack in the order declared, the stack grows down (from high address to low address), that this is a 32-bit system, and that the address of `handle` contains no zero bytes.

9. [6 points]:

A. True / False If function `test1` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

B. True / False If function `test2` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

C. True / False If function `test3` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

D. True / False If function `test4` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

For the next four questions, determine what is the minimum number of bytes that an adversary has to provide as input to cause this program to likely crash, when running different test functions. Do not count the newline character that the adversary has to type in to signal the end of the line to `gets`. Recall that `gets` terminates its string with a zero byte.

10. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test1` to crash?

11. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test2` to crash?

12. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test3` to crash?

13. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test4` to crash?

V Browser security

The same origin policy generally does not apply to images or scripts. What this means is that a site may include images or scripts from any origin.

14. [3 points]: Explain why including images from other origins may be a bad idea for user privacy.

15. [3 points]: Explain why including scripts from another origin can be a bad idea for security.

16. [4 points]: In general, access to the file system by JavaScript is disallowed as part of JavaScript code sandboxing. Describe a situation where executing JavaScript code will lead to file writes.

VI Static analysis

Consider the following snippet of JavaScript code:

```
1  var P = false;
2
3  function foo() {
4    var t1 = new Object();
5    var t2 = new Object();
6    var t = bar(t1, t2);
7    P = true;
8  }
9
10 function bar(x, y) {
11   var r = new Object();
12   if (P) {
13     r = x;
14   } else {
15     r = y;
16   }
17
18   return r;
19 }
```

A flow sensitive pointer analysis means that the analysis takes into account the order of statements in the program. A flow insensitive pointer analysis does not consider the order of statements.

17. [4 points]: Assuming no dead code elimination is done, a flow-insensitive pointer analysis (i.e., one which does not consider the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

18. [4 points]: Assuming no dead code elimination is done, a flow-sensitive pointer analysis (i.e., one which considers the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

19. [2 points]: At runtime, variable `t` in function `foo` may only be observed pointing to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

20. [2 points]: Do you think a sound analysis that supports the `eval` construct is going to be precise? Please explain.

21. [4 points]: What is one practical advantage of the bottom-up analysis of the call graph described in the PHP paper by Xie and Aiken (discussed in class)?

VII 6.858

We'd like to hear your opinions about 6.858, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

22. [2 points]: How could we make the ideas in the course easier to understand?

23. [2 points]: What is the best aspect of 6.858 so far?

24. [2 points]: What is the worst aspect of 6.858 so far?

End of Quiz