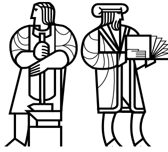


Department of Electrical Engineering and Computer Science

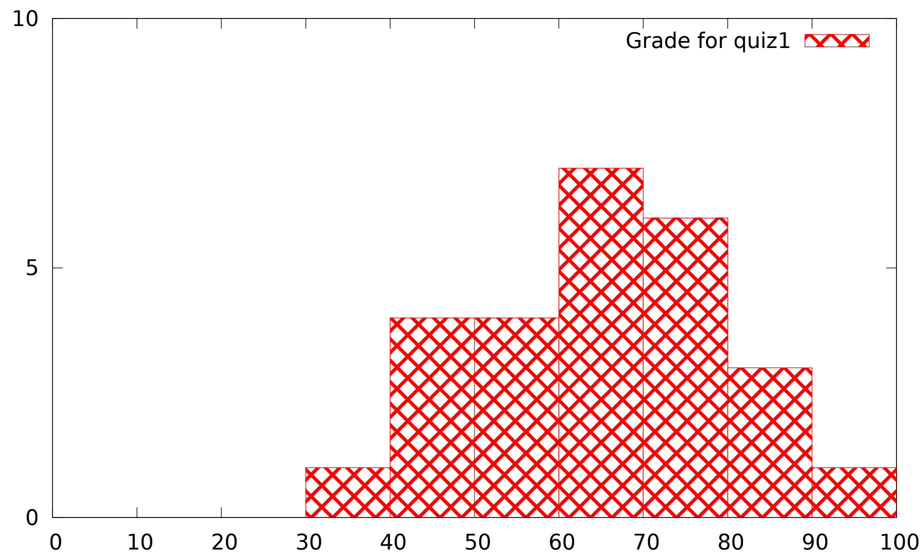
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2011

Quiz I: Solutions

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)



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)?

Answer:

- valid[EAX-0, EAX+256) is the only verification state that can be inferred at this point.

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

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-0, ECX+4), to satisfy the subsequent write to 4 bytes at ECX.
- valid[ECX-0, EDX+0), to represent the loop condition.

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

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-4, EDX+0), the loop condition updated with new value of ECX.

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

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-4, EDX+0), from above.
- valid[ECX-0, ECX+4), inferred from the check just before.

Note that these verification states must imply (i.e., be at least as strong) as the verification states at (2).

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`.

Answer: False.

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=...>`.

Answer: False.

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

Answer: True.

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

Answer: False.

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.

Answer: Since the directory is world-writable, an adversary could replace the contents of an arbitrary database, by first renaming the existing database's subdirectory to some unused name, and then creating a fresh directory (database) with the desired name of the database. For example, the adversary could replace all passwords with ones that the adversary chooses.

Answer: If an attacker can compromise any service, he can rename the `zoobars.x` file, since the directory is world-writable and not sticky, and replace it with a new one. (He can also replace the file with a symbolic link to an interesting other file that the zoobar-handling user can write to, and mount something along the lines of a confused-deputy attack.)

Full credit was also given for creating a directory before the user gets created; partial credit was given for removing a directory (since you cannot remove a non-empty directory you don't have permissions on).

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.

Answer: An adversary can modify an arbitrary user's profile and inject Python code that will transfer all of the user's zoobars to the adversary's account.

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).

Answer: Use a separate service, running as a separate uid, to edit profiles. Make sure the profile database is writable only by this new service's uid. Require the user's token to be passed to this service when editing a user's profile. Have this profile-editing service check the token using the authentication service.

Note that this only prevents attacking users who never log in, as the HTTP service can get the token of any user who does log in. An argument that compromising the HTTP service gets you wide latitude in compromising any user's activity would have been accepted for full credit.

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.

Answer: True. (If you overflow x.buf into x.f, you remain within the allocation bounds of x.)

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.

Answer: False. (If you overflow x.buf into any higher location, like the return pointer, you exceed the allocation bounds of x.)

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.

Answer: False. (If you overflow x.buf into any higher location, like y, you exceed the allocation bounds of x.)

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.

Answer: True. (If you overflow x[0] into x[1], you remain within the allocation bounds of the array x.)

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?

Answer: 32 (by overwriting x.f with a NUL byte, and jumping to it). Overwriting 64 bytes would cause a baggy bounds exception, but you can crash the program earlier.

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?

Answer: 60 (via a baggy bounds exception).

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?

Answer: 64 (via a baggy bounds exception).

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?

Answer: 32 (by overwriting x[1].f with a NUL byte, and jumping to it). Overwriting 124 bytes would cause a baggy bounds exception, but you can crash the program earlier.

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.

Answer: The other origin's server can track visitors to the page embedding images from that server.

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

Answer: The other origin's server must be completely trusted, since the script runs with the privileges of the embedding page. For example, the script's code can access and manipulate the DOM of the embedding page, or access and send out the cookies from the embedding page.

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.

Answer: Setting a cookie in Javascript typically leads to a file write, since the browser usually stores cookies persistently. Loading images can cause the image content to be saved in the cache (in some local file).

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

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: True.

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

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: False.

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

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: False.

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

Answer: No, because it is difficult to statically reason about the code that may be executed at runtime when eval is invoked, unless the analysis can prove that arbitrary code cannot be passed to eval at runtime, and can statically analyze all possible code strings that can be passed to eval.

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)?

Answer: Performance and scalability, by not analyzing functions that are not invoked by application code, and by summarizing the effects of the function once and reusing that information for inter-procedural analysis.

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?

Answer: Any answer received full credit.

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

Answer: Any answer received full credit.

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

Answer: Any answer received full credit.

End of Quiz