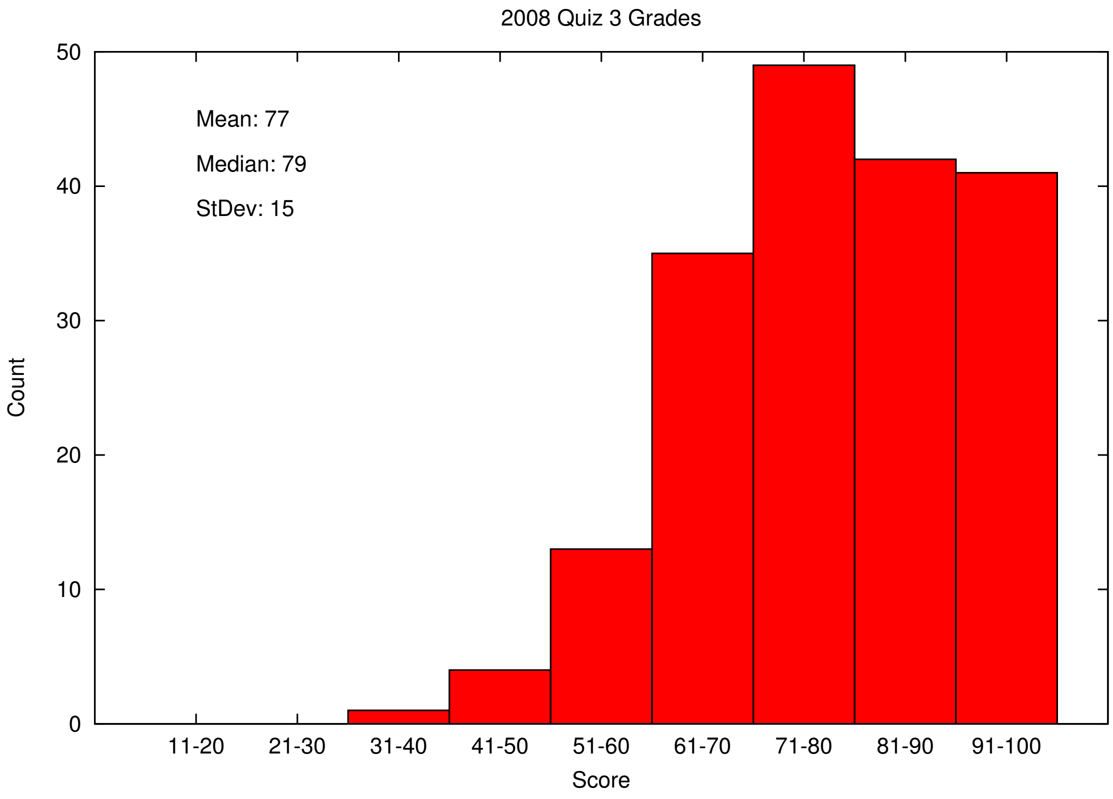
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2008

Quiz III - Solutions



I Warm-up

1. [6 points]: Which of the following statements about the paper *Why Cryptosystems Fail* by Ross Anderson (reading #17) is true?

(Circle True or False for each choice.)

A. True / False The paper claims that most frauds exploit cryptanalysis or other technical attacks.

FALSE. Most frauds exploit implementation errors and management failures.

B. True / False The paper implies that if UK law were changed such that disputed transactions are the bank's liability unless the bank can prove fraud, the likely result would be a large increase in fraud.

FALSE. The paper notes that in the US, where banks are liable by default, the resulting fraud is insignificant.

C. True / False The paper implies that if you keep your ATM card in your possession at all times, then no-one can use the ATM system to steal money from your account.

FALSE. The paper includes examples of ATM theft that did not require use of the customer's card.

2. [6 points]: Which of the following statements are true of Unison (as described in *How to Build a File Synchronizer*, from recitation)?

(Circle True or False for each choice.)

A. True / False If Unison finds that the content of a file on one host differs from the same file on another host, it indicates a conflict.

FALSE. It only indicates a conflict when the difference is due to separate modifications on both hosts.

B. True / False You create a file `X` with content `"a"` on host 1, then create a file `X` with content `"b"` on host 2. You then delete `X` from host 1, and run Unison to synchronize hosts 1 and 2. After Unison finishes you'll see a file `X` with content `"b"` on host 1.

TRUE.

C. True / False If you accidentally delete Unison's archive files on both machines, you are likely to need to resolve many more conflicts when you synchronize than if you didn't delete them.

TRUE. Unison then won't be able to distinguish between a file modified on one host versus a file modified on both hosts, so it will indicate a conflict for every file that's not identical on the two hosts.

II System Design

3. [4 points]: Ben Bitdiddle is building a file system, using a B-tree to represent each directory. As he works out his design he realizes that some changes to the B-tree need to write several disk pages, and he worries about what happens if there's a crash in the middle of these writes. He starts to work out the details, but Pam Principled says he should first read Lampson's *Hints* paper (reading #20). Which hints directly relate to Ben's problem?

(Circle ALL that apply)

A. Divide and conquer

B. Dynamic translation

C. Log updates

D. Make actions atomic

Answer: C and D.

4. [4 points]: Ben finishes his implementation and deploys it. He now starts worrying that B-trees have too much overhead for small directories, and decides to deploy a second implementation, using linear search, for that case. Which of the following hints directly support Ben's decision to deploy a second implementation?

(Circle ALL that apply)

A. Keep secrets

B. Do one thing well

C. Separate normal and worst case

Answer: C (A and B were also accepted.)

III Trusting Trust

5. [6 points]: You've written the source code of a compiler `C` for a new system. You compiler is written in the language Blub, and compiles the same language. Since you're a good MIT student, your compiler is completely bug-free.

Now you need to compile your source. You're given two executable Blub compilers, `a` and `b`, on the new system, but warned that one might be buggy or contain a Trojan horse. Except for this potential problem in `a` or `b`, the three compilers all implement Blub according to its specification, which is complete and exact.

You compile `C` using both `a` and `b` to produce executables `ca` and `cb` respectively. You then compile `C` using `ca` to produce `cca`, and compile `C` using `cb` to produce `ccb`. Finally, you compile `C` using `cca` to produce `ccca`, and compile `C` using `ccb` to produce `cccb`. You now compare the binaries produced at various stages of the compilation. Which of the following statements is true?

(Circle True or False for each choice.)

A. True / False If `ca` and `cb` differ then one of `a` and `b` is necessarily buggy/Trojaned.

FALSE. Even if `a` *and* `b` *are bug-free, they will likely compile the same input code to different output machine instructions.*

B. True / False If `cca` and `ccb` differ then one of `a` and `b` is necessarily buggy/Trojaned.

TRUE. `C` *should always produce the same output for a given input. That is, any two correct compilations of* `C`*'s source should behave the same.*

C. True / False If `ccca` and `cccb` differ then one of `a` and `b` is necessarily buggy/Trojaned.

TRUE

D. True / False If `cca` and `ccca` differ then `a` is necessarily buggy/Trojaned

TRUE

E. True / False If `b` contains a Trojan horse, then `ccb` and `cccb` will necessarily differ.

FALSE. Perhaps the operation of the Trojan horse is not triggered in this particular case.

IV Worms

Ben Bitdiddle, the new head of Cyber-Security for the Department of Homeland Security, wants to build a system that will detect and stop a *Witty* (reading #18) attack before it can reach 50% of the vulnerable hosts.

We make the following assumptions about Witty:

- Each Witty instance sends 512 (2^9) packets per second
- Witty's software has been fixed to probe *all* IP addresses rather than only a subset
- There are 32768 (2^{15}) vulnerable hosts on the Internet
- The initial hitlist contains a single vulnerable host

6. [6 points]: Given the assumptions above, roughly how many seconds will it take for the size of the infected population to double, during the early stages of a Witty outbreak?

(Circle the BEST answer)

A. 16 seconds

B. 256 seconds

C. 1024 seconds

Answer: B 256. There are 2^{32} possible IP addresses, and 2^{15} vulnerable hosts, hence roughly one in 2^{17} probes will be directed at a vulnerable host. It takes each Witty 2^8 seconds to generate 2^{17} probes at 2^9/second. That is, a given small population of Wittys will double roughly every 256 (2^8) seconds.

Ben convinces a consortium of router vendors to develop a new, remotely-configurable packet-filtering feature, and develops a system that can propagate filter updates to all routers in the Internet within 15 minutes (900 seconds) of a detected outbreak. Once all routers have the filter, the filters will prevent all further Witty infections.

Ben's detection mechanism is connected to a class-A network telescope (meaning that it sees $1/256$th of the Internet address space). His system is automatically triggered, propagating a packet filter update, whenever Witty traffic on the network telescope reaches a predefined threshold.

7. [8 points]: From the traffic thresholds shown below, which is the largest that would be expected to result in stopping Witty before it reaches 50% of the vulnerable hosts? (You may use your approximate answer from the previous question to compute this.)

(Circle the BEST answer)

A. 10 packets/second

B. 100 packets/second

C. 1000 packets/second

D. 10000 packets/second

E. 100000 packets/second

Answer: 1000. The threshold must be crossed 15 minutes before the outbreak reaches 50% of vulnerable hosts. 15 minutes is roughly 4 doublings, so the threshold should be set for the traffic level generated by infection of 3% of the vulnerable hosts. That's about 1000 hosts, which would generate 512,000 packets/second, of which Ben's telescope would see $1/256$th, or 2000/second. So the threshold must be set to a value smaller than 2000.

V Beyond Stack Smashing

You are hired by a well-known OS vendor to help them defend their products against buffer overrun attacks. Their team presents several proposed strategies to foil buffer exploits:

- **Random stack:** Place the stack in an area of memory randomly chosen for each new process, rather than at the same address for every process.
- **Non-executable stack:** Set the permissions on the virtual memory containing the stack to RW (read and write but not execute). Set the permissions on the memory containing the program instructions to RX (read and execute but not write).
- **Bounds checking:** Use a language such as Java or Scheme that checks that all array/buffer indices are valid.

You are aware of several buffer overrun attacks from your reading of *Beyond Stack Smashing* by Pincus and Baker (reading #16). They are summarized as follows:

- **Simple buffer overrun:** The victim program has an array on the stack as in Figure 1(a) of the paper. The attacker overruns the array with data that includes both some new instructions and a return program counter (PC) that points to these instructions.
- **Trampoline:** The victim program has an array on the stack as in Figure 1(a) of the paper, but the attacker cannot predict its address. However, the attacker knows that subroutine `f1a()` leaves a pointer into the array in register `R5` when it returns, and that the instructions at address `x` jump to wherever `R5` points to. The attacker overruns the array, overwriting the return PC with a pointer to address `x`.
- **Arc injection (return-to-libc):** Similar to the above, except that the attacker overwrites the return PC to point to an `exec` call in the libc library, indicated by `target` in the figure on page 22 of the paper.

In the following questions, an attack is considered *prevented* if the attacker can no longer execute the intended malicious code, even if an overflow can still overwrite data or crash or disrupt the program.

8. [6 points]: Which of the following attack methods are prevented by the use of the *random stack* technique?

(Circle ALL that apply)

A. Simple buffer overrun

Yes. The simple buffer overrun requires the malicious instructions in the buffer to be at a predictable address; the buffer also contains that address at a place which will overwrite the return PC on the stack.

B. Trampoline

No. Trampoline lets the attack work regardless of stack location, since it uses a register that points into the stack.

C. Arc injection (return-to-libc)

No. Arc injection doesn't involve executing instructions in the buffer overwritten by the attacker, so the fact that the buffer's address isn't predictable doesn't matter.

9. [6 points]: Which of the following attack methods are prevented by the use of the *non-executable stack* technique?

(Circle ALL that apply)

A. Simple buffer overrun

Yes, since the attack tries to execute instructions in a buffer on the stack.

B. Trampoline

Yes, since the attack tries to execute instructions in a buffer on the stack.

C. Arc injection (return-to-libc)

No, since the attack executes only instructions that are already present in the victim program.

10. [6 points]: Which of the following attack methods are prevented by the use of the *bounds checking* technique?

(Circle ALL that apply)

A. Simple buffer overrun

B. Trampoline

C. Arc injection (return-to-libc)

Yes for all three, since they all involve writing beyond the end of an array on the stack in order to change the return PC.

VI Two-Phase Commit

Here's a summary of the two-phase commit protocol example outlined in Chapter 9-F-3 of the book, where Alice is the coordinator for Bob and Charles:

1 Alice → Bob: `please do X`

2 Alice → Charles: `please do Y`

3 Bob → Alice: `done with X`

4 Charles → Alice: `done with Y`

5 Alice → Bob: `PREPARE to commit or abort`

6 Alice → Charles: `PREPARE to commit or abort`

7 Bob → Alice: `PREPARED`

8 Charles → Alice: `PREPARED`

9 Alice → Bob: `COMMIT`

10 Alice → Charles: `COMMIT`

11. [8 points]: At which points is it OK for Bob to abort his part of the transaction, i.e., undo any changes he has made to his data in response to Alice's requests, release any locks he holds, and forget all information he stores about those requests?

(Circle ALL that apply)

A. After receiving message 1 but before sending message 3.

B. After sending message 3 but before receiving message 5.

C. After receiving message 5 but before sending message 7.

D. After sending message 7 but before receiving message 9.

E. After receiving message 9.

Answer: it's OK for Bob to abort any time before sending PREPARED (message 7). If Alice has not received Bob's PREPARED, she will not tell Charles to commit. If Alice has received Bob's PREPARED, she may tell Charles to COMMIT. Since the goal is to ensure that either both or neither of Bob and Charles commits, Bob must be willing to commit once he sends PREPARED; he cannot unilaterally abort.

VII One-time Pad

Alice wants to communicate with Bob over an insecure network. She learned about one-time pads in 6.033, and decides to use a one-time pad to secure her communications. Since Alice wants to send a k-bit message to Bob in the future, she generates a random k-bit key `r` and hands it to Bob in person.

When Alice comes to send Bob her message, she XORs the message `m` with the key `r` to produce a cyphertext `c`, and sends this on the network. Bob XORs `c` with `r` to retrieve `m`.

12. [6 points]: Assume that Alice's message `m` is a concatenation of a header followed by some data. Consider an eavesdropper Eve who snoops on Alice's conversation. If Eve can correctly guess the value of the header in Alice's message, which of the following are correct?

(Circle ALL that apply)

A. Eve's ability to decrypt the data bits in `m` is not improved by her knowledge of the header bits.

Yes. The one-time pad encrypts each bit independently, hence there is no correlation between the encryption of the data bits and the header bits.

B. The data bits in Alice's message are confidential.

Yes. Since each bit in `m` *is XORed with the corresponding bit in the random key, it is impossible to determine the correct decryption without knowledge of the key (or* `m`*).*

C. The data bits in Alice's message are securely authenticated.

No. An active attacker could change bits in `c` *and the one-time scheme would not alert Bob that he was not reading the information Alice sent.*

Alice rapidly grows tired of the effort in exchanging one-time pads with Bob, and has an idea to simplify the key distribution process. Alice's idea works as follows:

To send a k-bit message `m1` to Bob, Alice picks a k-bit random number `r1`, computes cyphertext `c1 = m1 XOR r1`, and sends `c1` to Bob. Bob then picks his own k-bit random number `r2`, computes `c2 = c1 XOR r2`, and sends `c2` to Alice. Alice finally computes `c3 = c2 XOR r1` and sends `c3` to Bob.

13. [6 points]: Which of the following are correct of Alice's scheme?

(Circle ALL that apply)

A. Bob can correctly decrypt Alice's message `m1`, without receiving `r1` ahead of time, assuming all messages between Alice and Bob are correctly delivered.

Yes, since `c3 = r2 XOR m1` *and Bob knows* `r2`.

B. An active attacker Lucifer (who can intercept, drop, and replay messages) can decrypt the message.

Yes. Lucifer can use the same technique as Eve (see next answer).

C. A passive eavesdropper Eve can decrypt the message.

Yes. Eve can find out `r2` *using* `c2 XOR c1`*, and then find* `m1` *using* `c3 XOR r2`.

VIII Locking for Transactions

Assume a database with logs as described in Chapter 9-C and locks as described in Chapter 9-E. The logging and recovery works as shown in Figure 9-21 (the in-memory database with write-ahead logging). Programmers write transaction functions, and the database automatically adds acquire and release calls to the transaction functions in a way that achieves serializability, avoids deadlocks caused by locking, and allows the transactions to have all-or-nothing atomicity in the face of crashes.

As a reminder, concurrently executing transactions are serializable if the result is the same as if they had executed one at a time in some order. A transaction has all-or-nothing atomicity if, after a failure and recovery, the transaction appears either to have completely executed or to have never started.

Consider the following transaction. The database system automatically inserted the `acquire()` and `release()` calls.

```
t1():
  BeginTransaction
  acquire(X.lock)
  acquire(Y.lock)
  X = X + 1
  if(X == 1):
    Y = Y + 1
  CommitTransaction
  release(X.lock)
  release(Y.lock)
```

`X` and `Y` denote particular database entries (they are *not* parameters to the transaction).

The above transaction works correctly even if multiple instances of the transaction execute concurrently. That is, it is serializable, won't deadlock, and has all-or-nothing atomicity if there is a crash and recovery.

14. [6 points]: The database starts with contents `X=0` and `Y=0`. Two instances of `t1()` are started at about the same time. There are no crashes, and no other activity. After both transactions have completely finished, which of the following are possible database contents?

(Circle ALL that apply)

A. `X=1 Y=1`

B. `X=2 Y=0`

C. `X=2 Y=1`

D. `X=2 Y=2`

Answer: C. The locks cause the two transactions to execute one at a time. Since they are identical, the result is the same regardless of order.

Suppose the code for `t1()` was changed such that the two releases are just before the `CommitTransaction`:

```
t1b():
  BeginTransaction
  acquire(X.lock)
  acquire(Y.lock)
  X = X + 1
  if(x == 1):
    Y = Y + 1
  release(X.lock)
  release(Y.lock)
  CommitTransaction
```

This version is *not* correct.

15. [8 points]: The database starts with contents `X=0` and `Y=0`. Two instances of `t1b()` are started at about the same time. There is a crash, a restart, and recovery runs. After recovery completes, which of the following are possible database contents?

(Circle ALL that apply)

A. `X=1 Y=1`

Yes. The crash may occur after one transaction commits, but before the other one starts.

B. `X=2 Y=0`

Yes. One transaction might get as far as releasing locks, but not as far as committing. Then the second transaction runs and commits, writing only a new `X=2` *to the log. Then the system crashes, loses its state, and recovery re-does only the second transaction's write to* `X` *(and not Y), since the first transaction did not commit.*

C. `X=2 Y=1`

Yes, if the crash occurs after both transactions commit.

D. `X=2 Y=2`

No. No execution can produce `Y=2`*.*

Here are three transactions. The locking in `t2()` is not correct.

```
t2():
  BeginTransaction
  acquire(M.lock)
  temp = M
  release(M.lock)
  acquire(N.lock)
  N = N + temp
  CommitTransaction
  release(N.lock)

t3():
  BeginTransaction
  acquire(M.lock)
  M = 1
  CommitTransaction
  release(M.lock)

t4():
  BeginTransaction
  acquire(M.lock)
  acquire(N.lock)
  M = 1
  N = 1
  CommitTransaction
  release(M.lock)
  release(N.lock)
```

16. [8 points]: The initial values of `M` and `N` in the database are `M=2 N=3`. Two of the above transactions are executed at about the same time. There are *no* crashes, and there is no other activity. For each of the following pairs of transactions, decide whether concurrent execution of that pair could result in a non-serializable result. If not, write `none`. If a non-serializable result could occur, write an example of such a result (an `M` and an `N`).

(Please write your answers in the spaces below)

`t2()` and `t2()`: *None. Since* `M` *doesn't change, we can ignore the first half of* `t2()`*. The second halves are forced to run one at a time by the lock on* `N`*, which means the result must be serializable by definition.*

`t2()` and `t3()`: *None. The only potential problem case is if* `t3()` *runs after* `t2()`*'s release of the* `M` *lock, but in that case the result is the same as if* `t3()` *ran entirely after* `t2()`*.*

`t2()` and `t4()`: `M=1 N=3`, *if* `t4()` *runs after* `t2()`*'s release of* `M` *and before its acquire of* `N`*. This result not serializable because it is equal to neither the result of running* `t2()` *then* `t4()`*, nor running* `t4()` *then* `t2()`*.*

End of Quiz III.

Have a great summer!