Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2013

Quiz II Solutions

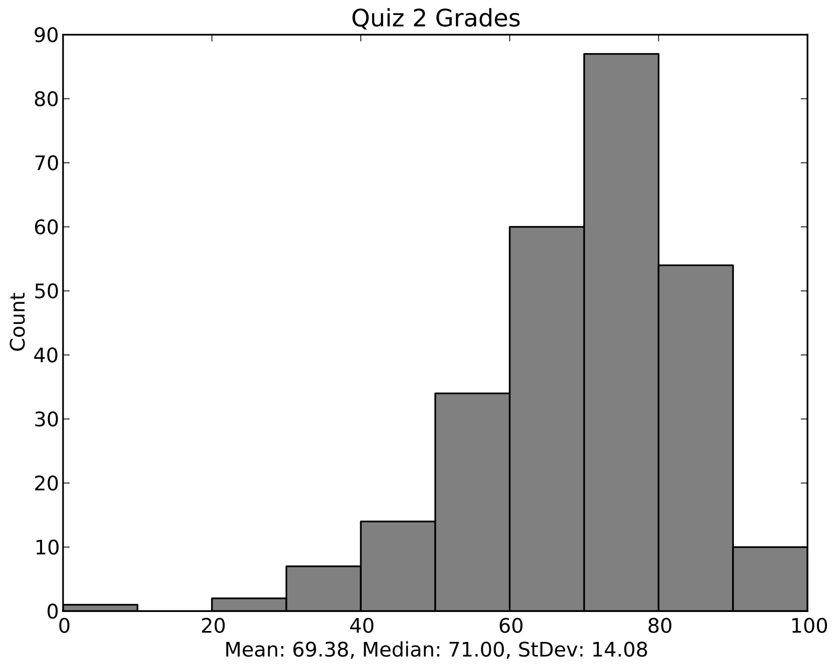
There are 12 questions and 11 pages in this quiz booklet. Answer each question according to the instructions given. The quiz is designed to be do-able in 90 minutes, but you have **120 minutes** to answer the questions.

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

For true/false and yes/no questions, you will receive 0 points for no answer, and negative points for an incorrect answer. We will round up the score for every *numbered* question to 0 if it's otherwise negative (i.e., you cannot get less than 0 on a numbered question).

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!

Grade distribution histogram:



I Lectures

The following questions refer to material discussed in lectures.

1. [12 points]: Indicate whether the following statements (or *Claims* therein) are correct.
(Circle True or False for each choice.)

A. True / False Using a random *salt* while hashing a user's password would make it more difficult for an attacker who gains access to the password database to determine whether two users chose the same password.

Answer: True. The attacker would not be able to merely compare different user's hashed passwords.

B. True / False Diffie–Hellman key exchange protects against eavesdropping attacks but not Man-in-the-middle attacks.

Answer: True. D-H can bootstrap a secure channel, but cannot authenticate the endpoints of the channel to ensure it isn't an attacker.

C. True / False If onion routing used a single shared secret key, it would be easier to compromise the system by compromising one key. *Claim*: Assuming that the shared key is not compromised, it would give onion routing the same level of anonymity as a per-node public-private key.

Answer: False. Each node will be able to decrypt the whole onion and discover the whole path.

D. True / False Using a certificate authority facilitates public key distribution. However, there is a bootstrapping difficulty: the first time Bob communicates with Alice, he still does not have a certificate listing her public key. *Claim*: Bob can only retrieve Alice's certificate directly from the certificate authority.

Answer: False: Alice can send the certificate to Bob without Bob talking to the certificate authority.

E. True / False The *second system effect* refers to the tendency to try to add every possible feature that didn't end up in the first version into the next version, often making a project too complex to succeed.

Answer: True

F. True / False The legal issues around whether speech is protected by the First Amendment make the design of computers systems more complex.

Answer: True

II RAID

2. [6 points]: Which of the following statements are true and which ones are false?
(Circle True or False for each choice.)

A. True / False When using interleaving over N data disks, even considering synchronization slow-down, read throughput is *guaranteed* to be $\frac{N}{2}$ times that of the throughput of a single disk.

Answer: False. The presence and types of writes, and granularity of the read requests are two possible reasons for slower read throughput.

B. True / False In RAID 4, if the probability that a disk is faulty on a read is p, then the expected number of reads given N **data** disks is $1 + p \cdot (N - 1)$.

Answer: False. The non-faulty case requires 1 read. The faulty case requires 1 read to the faulty disk, followed by $N - 1$ reads to the other data disks, plus 1 read to the parity disk, so the total expected number of reads is $(1 - p) \cdot 1 + p \cdot (1 + (N - 1) + 1) = 1 + p \cdot N$.

3. [3 points]: The minimum number of disks for RAID 5 is
(Circle the BEST answer)

A. 1

B. 2

C. 3

D. 4

Answer: 3

4. [3 points]: Which levels of RAID use striping (i.e., interleaving)?
(Circle ALL that apply)

A. 1

B. 2

C. 3

D. 4

E. 5

Answer: 2, 3, 4, and 5

5. [3 points]: In a RAID 4 system with N data disks and one parity disk, writing a sector on a data disk requires

(Circle the BEST answer)

A. 2 reads and 2 writes.

B. 2 reads and 1 write.

C. N reads and 2 writes.

D. N reads and 1 write.

Answer: A. It needs to read the old data and parity, calculate the new parity and data, and write the new parity and data.

III System R

6. [9 points]: For each statement below indicate whether it is true or false

(Circle True or False for each choice.)

A. True / False System R stores the log in a separate disk partition managed as a ring buffer. In a ring buffer newly stored log entries may overwrite old log entries: If the ring buffer has N sectors on disk and if the most recent log entries L_1, L_2, …, L_k, fit in N sectors but there is no additional space for entry L_{k+1}, then the older log entries L_{k+1}, L_{k+2}, etc. are lost. To be able to recover from crashes, it is *sufficient* to store at least one checkpoint among the k most recent log entries that are stored within the ring buffer.

Answer: False. In order to recover from a crash, it needs not only the log entries up to and including the most recent checkpoint, but also the log entries up to and including the last incomplete transaction. To make sure that the last incomplete transaction is stored in the ring buffer, System R also needs to abort transactions that take too long.

B. True / False Archiving in System R is implemented as copying the data base system from disk to tape. If System R wishes to restore an old archived version of the data base, it simply performs its crash recovery protocol on the archived version using the current log.

Answer: False. The old log (at the time of the archive) is needed, not the current log.

C. True / False System R manages its disk buffer (cache) using an LRU eviction mechanism during which transaction data is written to shadow/backup files. Suppose System R only does FILESAVEs at every checkpoint at which all transaction data become permanent in that shadow/backup files becomes current/primary files. *Claim*: If System R adopts the policy to write the log buffer (cache) to disk before starting a checkpoint, then it will implement write-ahead logging correctly.

Answer: True. System R also forces the log whenever a commit record is written to the log, but this is not necessary as an additional requirement for write-ahead logging.

7. [4 points]: Suppose the log buffer (cache) is written to disk using a FIFO (first in first out) policy whenever the buffer is full. Recall that storage for the buffer is not permanent and is lost during a crash. To ensure correct recovery, system R must also:

(Circle True or False for each choice.)

A. True / False Write the log buffer to disk whenever a commit record is written

Answer: True and False. Writing the log in FIFO order means that any recovery action will restore a consistent point, but effectively moves the commit point of the transaction to when a log flush occurs. Transactions won't be durable across recovery unless the database delays returning from commit requests until log flush points.

IV PNUTS

8. [12 points]: For each statement below indicate whether it is true or false

(Circle True or False for each choice.)

A. True / False Ben Bitdiddle proposes that instead of using PNUTS for replication, we just have each application accessing a record send update messages to all of its replicas, and wait to get an answer back. The main reason for not doing this is because, compared to PNUTS, it will increase the overall number of messages we need to send per record update.

Answer: False. This change would also break PNUTS' consistency property.

B. True / False In "per-record timeline consistency," once an update is committed, every client read will return the newly committed value.

Answer: False. Only clients which request the latest version of a record will see the recent changes.

C. True / False In PNUTS, if the "Tablet Controller" crashes, then it will recover a version of the "Interval Map" from one of the Routers in its region.

Answer: False. Routers only hold cached copies—a crashed tablet controller recovers a version from the standby tablet controller.

D. True / False In PNUTS, to improve performance, the location of the master copy is fixed in the region corresponding to the application that first calls it.

Answer: False. The tablet controller can change the location.

V Trusting trust

The following questions refer to the paper "Reflections on Trusting Trust" by Ken Thompson.

9. [8 points]: For each *Claim* below indicate if it is true or false

(Circle True or False for each choice.)

A. True / False The vertical tab character, `'\v'`, was used to control some old-fashioned printers. All C compilers long ago learned to recognize `'\v'`, as explained in the Trusting Trust paper. However, no one uses vertical tabs anymore, so Lem E. Tweakit proposes eliminating it. Soon thereafter, all C compilers remove support for vertical tabs from their source code.

Claim: Since new C compilers are written in C and are compiled with existing C compilers that still recognize `'\v'`, we cannot get rid of it; these new compilers will still recognize it.

Answer: False. The new compiler will not recognize it.

B. True / False Consider (1) a program which contains the UNIX `login` code, and also prints out its own source code, and (2) a compiler which might or might not contain the specific Trojan horse described in the paper that miscompiles `login`.

Claim: If this program's output matches the source code to `login`, then the compiler does not contain the Trojan.

Answer: False. The binary may be affected but not the strings in the program's output.

Perhaps one way to get rid of compilers that might contain the backdoor described in the Trusting Trust paper is to add another "bug" or "feature" to the compiler. If the UNIX login pattern is detected, then replace whatever code is generated with correct code. This will be the last pattern detected performed after any other detected pattern. Call this compiler $C_{FrontDoor}$. Compile $C_{FrontDoor}$ with any existing C compiler (that may have the login+compiler hack as described in the paper in it).

10. [8 points]: For each *Claim* below indicate if it is true or false

(Circle True or False for each choice.)

A. True / False *Claim*: Compiling OS code with $C_{FrontDoor}$ results in an OS without the compiler-inserted backdoor Thompson described in the paper.

Answer: True and False. Either answer is possible depending on what assumptions one makes about how the new "feature" interacts with an existing Trojan in the compiler.

B. True / False Compile $C_{FrontDoor}$ twice. First with any compiler, and then with itself (the $C_{FrontDoor}$ compiled after step one) yielding $C^2_{FrontDoor}$. Replace all existing compilers with $C^2_{FrontDoor}$.

Claim: Recompiling all existing C source compilers with $C^2_{FrontDoor}$ will eliminate the problem once and for all and we can trust the new compilers.

Answer: False. We only protected against the backdoor login Trojan, but not the compiler trojan which may still be part of $C^2_{FrontDoor}$.

Student Initials:

VI Transactions

A local consulting firm hires Ben Bitdiddle to build a task management system. The firm wants to have each partner check the number of unfinished jobs when deciding whether to accept new jobs, and avoid overbooking by declining new work if that would result in more than 3 unfinished jobs across the firm.

Ben builds a database with a table listing unfinished jobs for each of the two partners, Alice and Bob, and writes the following procedure to add a new job:

```
tables = ("alice_tasks","bob_tasks")

def newjob(table, task):
  database.execute("begin;")
  database.execute("insert into %s values ('%s');" % (table, task))
  total_count = 0
  for t in tables:
     # "select count(*) from X" returns the number of rows in table X
     total_count += database.execute("select count(*) from %s;" % t)
  if total_count > 3:
     database.execute("abort;")
  else:
     database.execute("commit;")
```

Alyssa P. Hacker reviews Ben's code. She has several comments, two of which are: (1) she asks Ben to modify his code to guard against SQL injection attacks (which we will ignore for the rest of this question), and (2) she asks Ben what degree of isolation between transactions his database server provides.

Ben runs some tests by executing the following three calls concurrently from different client machines.

```
newjob("alice_tasks","C")      newjob("bob_tasks","D")      newjob("alice_tasks","E")
```

Before running these calls, the initial database state is:

table: alice_tasks	table: bob_tasks
A	B

Student Initials:

11. [18 points]: For each of the following end database states, indicate whether that outcome is possible if the database server provides the given type of isolation guarantee.

(Circle Yes or No for each choice.)

Final Database State 1:

table: alice_tasks
A

table: bob_tasks
B
D

A. Yes / No : **Serializable**

Answer: Yes. One of the three requests will commit first, and the other two will abort.

B. Yes / No : **Snapshot Isolation**

Answer: Yes, for the same reason as above.

Final Database State 2:

table: alice_tasks	table: bob_tasks
A	B
E	D

C. Yes / No : **Serializable**

Anwser: No. This outcome requires two requests to commit, which cannot happen with a serial order.

D. Yes / No : **Snapshot Isolation**

Anwser: Yes. Two requests each see a snapshot of the initial DB state, and insert their new record to that state, which does not result in a write-write conflict. They each then count the entires in their modified snapshot, and proceed to commit. The third request arrives after the first two commit, and aborts.

Final Database State 3:

table: alice_tasks	table: bob_tasks
A	B

E. Yes / No : **Read Uncommitted**, where a read can return values written by another transaction before that transaction commits or aborts.

Answer: Yes. In this case, all three requests insert their new rows. All three count the resulting number of rows, discover that there are too many, and abort their transactions.

F. Yes / No : **Serializable**

Answer: No.

Student Initials:

VII Buffer Overrun Attacks

The authors of the Bitdiddle C Compiler (BCC) are fed up dealing with programs like this one:

```
void process_input(char *input) {
  char buf[1024];

  strcpy(buf, input);  /* copies string input into buf */

  /* ...compute compute compute... */
}

int main(int argc, char *argv[]) {
  /* ...compute compute compute... */
  process_input(input);
  /* ...compute compute compute... */
}
```

We've seen how this sort of program is vulnerable to *buffer overrun* attacks, since the `strcpy()` invocation does no bounds checking and may "write off the end of" the buffer `buf`, if the string `input` is long enough. When users are allowed to provide `input`, they may be able to subvert the intended control flow of the program.

BCC as originally implemented follows the same stack layout convention as GCC for x86 Linux, so that running the above program until just after entering `process_input`, we might see the top few stack slots looking like this:

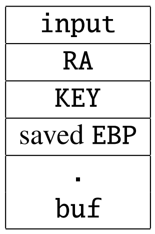
highest address	`input`
	`RA` (return address)
	saved `EBP` (base pointer)
lowest address	. `buf`

The BCC team decides to change their compilation scheme to make buffer overrun attacks harder to pull off. Some of their most important users run a version of Linux from 2000, which makes the stack fully executable and doesn't randomize stack addresses; so the change should be effective even in that tricky setting. Some of the proposed solutions involve picking random numbers or using cryptography, so assume that there is a perfect random number generator, which generates length-32 bitstrings uniformly at random, and assume crypto operations at the usual level of effectiveness.

12. [14 points]: Which of the following alternate compilation schemes will, with high probability (say, 99% or higher), foil buffer overrun attacks that try to get `process_input()` to return to somewhere beside the intended return address in `main()`? Mark **T** when the scheme works and **F** when it doesn't.

Student Initials:

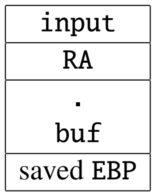
A. **T F**



When the program starts, choose a random secret key `KEY` and store it in a global variable, such that a buffer overrun could never overwrite the key. After pushing a return address during a function call, also push `KEY`. When returning from a function, pop the value from the key position on the stack and make sure it still matches `KEY`.

Answer: T. True for the given process_input (no other function pointer locals), not true in general.

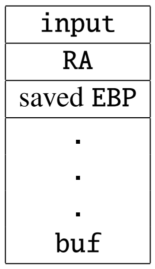
B. **T F**



In the callee, push the `EBP` value to save *after* allocating local variable positions.

Answer: F. The attacker can still overwrite the return address.

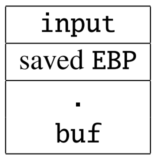
C. **T F**



Conservatively double the number of bytes allocated for each string buffer.

Answer: F. This approach only changes by how many bytes the attacker needs to overfill `buf`.

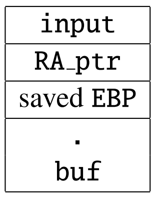
D. **T F**



Skip storing return addresses on the main stack, instead keeping return addresses on a separate stack, in some part of memory that it is safe to assume can't be touched by a buffer overrun. Index this separate stack with a global variable that is similarly safe to assume is protected.

Answer: T

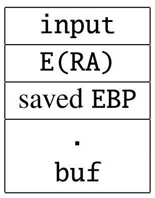
E. **T F**



For each return address needed in a compiled program, allocate a read-only global variable storing that return address. Instead of saving the return address directly on the stack, instead save the address of the global variable that itself holds the proper address. At return time, pop the pointer, read the return address that it points to, and jump to that address.

Answer: F. The attacker can overwrite `RA_ptr` with a reference to a pointer to somewhere else.

F. **T F**

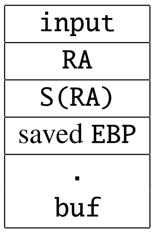


When the program starts, generate a random encryption/decryption key and store it in a global variable that is safe to assume protected from overruns. When making a function call, instead of pushing the return address, push the result of encrypting it with the key (denoted to the left with the encryption function `E()`). At return time, decrypt the popped value to get the real return address to jump to.

Answer: F. The attacker can overwrite the RA slot with any value; on return it will be "decrypted" to create a random return address.

Student Initials:

G. **T F**



When the program starts, generate a random key for crytographic signatures and store it in a global variable that is safe to assume protected from overruns. When making a function call, push both the return address and its signature with the key (denoted to the left with the signing function `S()`). At return time, pop both values and check that the signature matches the return address before jumping to it.

Answer: T

End of Quiz II

Please double check that you wrote your name on the front of the quiz, and circled your recitation section number. Enjoy the summer!