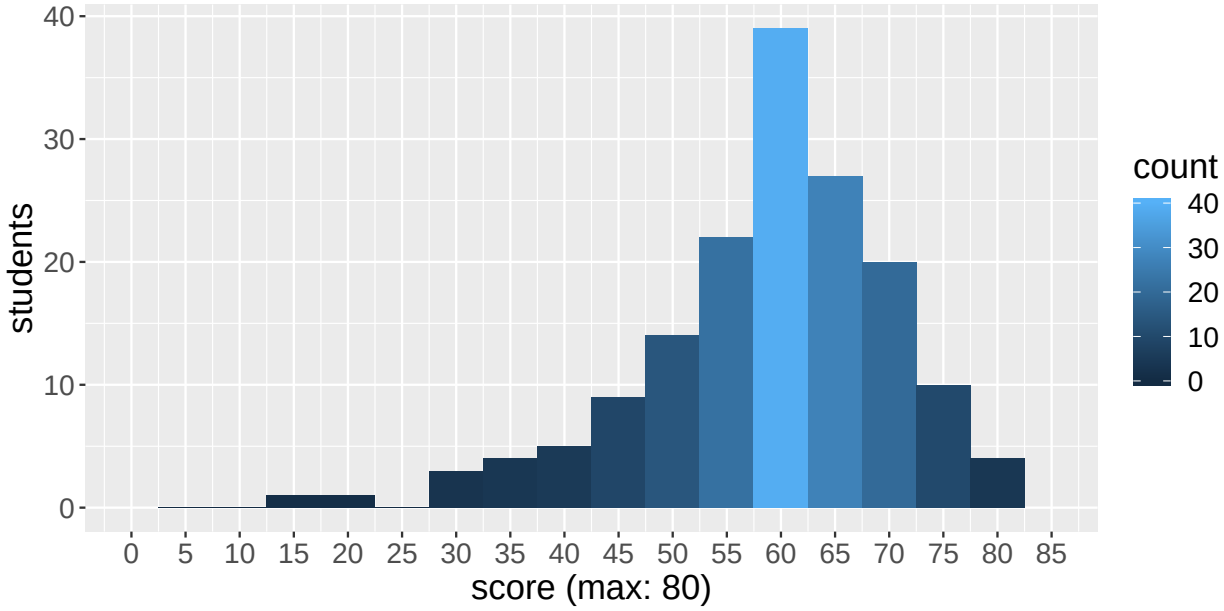
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Spring 2019

Quiz II Solutions

Mean 58.62 Standard deviation 11.45 Kurtosis 1.583558



I Multiple-choice questions

1. [3 points]: Based on Mark Silis, Garry Zacheiss, and Jessica Murray's guest lecture, which of the following statements are true?

(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False The one DDoS attack on MIT in the last year flooded a network link.

B. True / False Email forwarding through MIT resulted in much email being marked as spam, because after forwarding the DKIM signature failed for many messages.

C. True / False Chinese hackers stole military maritime secrets through a breach at MIT.

Answer: A is false because the attack ran the server that accepts incoming email out of TCP sockets; it didn't flood a network link and so Akamai couldn't defend against it. B is true; emails were broken up into pieces and then reassembled slightly differently for forwarding, causing the signature to fail at the receiver. C is false; the report was inaccurate.

2. [4 points]: Based on Max Burkhardt's guest lecture, which of the following statements are true?

(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False Security solutions scale well with the growth of the Internet/Web.

B. True / False Mutual TLS authenticates the server to the client and the client to the server.

C. True / False Network segmentation limits the services a compromised machine can talk to.

D. True / False The global map of which services talk to each other changes so slowly that the map can be maintained manually.

Answer: A is false, and a good reason to seek a job in security, according to Max. B is true. C is true, and was the main point of Max's project. D is false; airBnB has an automatic way of computing a global map based on reading configuration files of individual services.

3. [4 points]: Based on the Tor readings and Nick Mathewson's guest lecture on Tor, which of the following statements are true?

(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False Tor sacrifices some security properties that are desirable in an anonymous communication system.

B. True / False An attacker who can observe all traffic going into entry points and coming out of exit points can learn the IP address of the machine that opened a TCP connection over TOR.

C. True / False Tor traffic has grown so much that the directory server and protocol of Section 6.3 in the paper may become impractical, because the complete list of relays is too large.

D. True / False If an attacker compromises one relay in the middle of a circuit (i.e., not the first relay), then the attacker can learn the IP address of the machine that originated the TCP connection that is running over the circuit.

Answer: A is true. Tor, for example, doesn't delay message or use cover traffic, to improve usability. B is true. If an attacker can observe all incoming and outgoing traffic, then the attacker can correlate the two and deduce who is connecting to Web site. C is true; Nick spent a good amount time explaining a new proposal to fix this potential problem (https://lists.torproject.org/pipermail/tor-dev/2019-February/013666.html). D is false; this is one of the primary attacks that Tor defends against.

II Symbolic/concolic execution

Ben implements `concolic_force_branch` (that explores the other side of branch *b*) for Lab 3 the following way:

```
def concolic_force_branch(b, branch_conds, branch_callers):
    constraint = const_bool(True)

    cond_list = [ x for i, x in enumerate(branch_conds) if i != b]
    for branch in cond_list:
        constraint = sym_and(constraint, branch)
    constraint = sym_and(constraint, sym_not(branch_conds[b]))

    return constraint
```

4. [5 points]: Would Ben's implementation exhaust all possible control flow decisions in the program? Why/Why not? If not, please correct the code above.

Answer: No. The code above keeps *all* constraints except `i != b`, but we want it to follow the same path all the way till right *before* `b` and go the other way just on the b^{th} branch. None of the branches past `b` are of relevance. The conditional should read `i < b`.

Consider the following snippet, similar to the code in `check-symex-int.py`:

```
def f(x):
    if x == 0:
        return 3
    elif x == 1:
        return 6
    elif x == 2:
        return 8
    elif x == 3:
        return 7
    return -1

def test_f():
    i = fuzzy.mk_int('i')
    v = f(i)
    return v

f_results = fuzzy.concolic_execs(test_f)
```

With a correct implementation of the concolic execution system, `f_results` contains [3, 6, 8, 7, -1], all possible results of executing the function `f`.

Suppose `f` were replaced with a functionally equivalent implementation `g` as follows:

```
def g(x):
    if 0 <= x < 4:
        return [3, 6, 8, 7][x]
    return -1
```

With this implementation, the concolic execution system from Lab 3 only finds the results [3, -1].

5. [5 points]: Explain why the concolic execution system as implemented in Lab 3 fails to find the rest of the possible results.

Answer: There is no support for symbolic indexing into a concrete array. In the first execution, the indexing operation uses the concrete value of x (0) to index into the array, returning the value 3, but this is not recognized as a branch. That is, it doesn't produce any additional path constraint (e.g. x != 0) that would enable discovering the rest of the results for the (0 <= x < 4) path.

III TCP hijacking

As described in the paper *A Look Back at "Security Problems in the TCP/IP Protocol Suite"*, an attacker on machine X can pretend to be a trusted machine T while connecting with TCP/IP to machine S. The attack is as follows and relies on guessing the initial sequence number ISN_s:

A. X $\rightarrow$ S: SYN(ISN_x), SRC=T

B. S $\rightarrow$ T: SYN(ISN_s), ACK(ISN_x)

C. X $\rightarrow$ S: ACK(ISN_s), SRC=T

D. X $\rightarrow$ S: ACK(ISN_s), SRC=T, data

6. [4 points]: The attacker must suppress message B; why? (Briefly explain your answer.)

Answer: If T receives message B, it will respond to S with a connection reset (RST) packet, which will cause S to terminate the connection that X is trying to establish.

7. [4 points]: Assume that the attacker has no control of the network between T and S. How can the attacker suppress message B? (Briefly explain your answer.)

Answer: Wait until T is down, or DoS T (potentially using TCP SYN flooding).

IV Secure handshake

Ben uses the following protocol to set up a secure channel over the Internet from a client (C) to a server (S):

A. $C \rightarrow S$: make a connection with S

B. $C \leftarrow S$: PK_t, Sign(SK_S, $\{PK_t\}$)

C. $C \rightarrow S$: Encrypt(PK_t, $\{K, SN\}$)

D. $C \rightarrow S$: c_1 = Encrypt(K, $\{msg_1, SN\}$), t = MAC(K, c_1)

E. $C \rightarrow S$: c_2 = Encrypt(K, $\{msg_2, SN+1\}$), t = MAC(K, c_2)

F. ...

PK_S / SK_S are the server's permanent public/private key pair. The client knows PK_S. The server creates a fresh public/private key pair PK_t / SK_t using a secure random number generator just before step B. The client creates a fresh symmetric key K using a secure random number generator just before step C. The client generates a random initial sequence number SN with a secure random number generator just before step C.

The ciphers and random-numbers are strong and correctly implemented. The adversary does not know SK_S.

Immediately after processing message C, S deletes SK_t from memory. When C and S are done communicating and close the secure channel, they also delete the shared symmetric key K.

Steps D and E provide authenticated encryption: an adversary cannot determine the contents of the message, or construct another valid message, without knowing the shared key K. The MAC must match in order for S to accept a message. S will reject a message if it doesn't have the expected next sequence number, so that the attacker cannot replay messages D and E.

8. [4 points]: Suppose an adversary A can guess SN. Can the adversary hijack an on-going conversation between C and S? That is, can A insert a message in step F that S will accept? (Explain your answer briefly.)

Answer: The adversary does not have the key K and thus cannot construct a message with a MAC that will pass the server's check.

9. [4 points]: Can S establish the identity of C? That is, if C sets up a new secure channel to S, can S establish that it is the same C as in the earlier secure channel? Explain your answer briefly.

Answer: C never signs any message with a private key and S cannot rely on the source IP address as reliable identifier for C.

10. [4 points]: Does the protocol provide forward secrecy? That is, if the attacker learns SK_S after the connection has finished and the attacker has a recording of all the encrypted messages sent, is it difficult for the attacker to learn the content of those messages? (Briefly explain your answer.)

Answer: Yes, because K is encrypted with PK_t, which is freshly generated for this session, and SK_t has been deleted.

V Web security

Ben Bitdiddle did not do lab 4, and decides to use the 6.858 Zoobar application without modifications to launch his own bank: Ben's Bucks. Ben knows from skimming the 6.858 notes that the same-origin policy protects web sites from each other, so when Alyssa asks him about the security of his bank, Ben reassures her that it is completely safe. Alyssa, having done lab 4, finds this hard to believe given all the issues she knows about in Zoobar.

11. [2 points]: Briefly explain one JavaScript-based attack that Alyssa can perform against Ben's bank that the same-origin policy does not protect against.

Answer: Basically any attack from lab4. Cross-site scripting allows you to run code in the same origin as the target, which circumvents same-origin restrictions. Cross-site request forgery allows requests to be posted to a different origin, and while the attacker cannot read the response, they can cause the request to be issued.

Following his discussion with Alyssa, Ben realizes that his bank may have some security shortcomings. He carefully walks through his code and ensures that he never emits unsanitized user-controlled input, and that every transfer request requires a unique token embedded into the transfer form. Afterwards, he tells Alyssa that the site is now 100% secure.

Alyssa has been paying close attention to web security research over the past few years, and is still not convinced. She decides to try a "clickjacking" attack; she loads the transfer page from Ben's bank in an iframe on a page she controls, makes it transparent, and then adds a button that says "Free apples" directly underneath the "Transfer" button on the form. This way, when a victim clicks the "Free apples" button, their browser will instead interpret it as a click of the "Transfer" button.

12. [5 points]: Assuming the fields of the transfer form have already been filled out, would Alyssa's clickjacking attack indeed perform an unauthorized transfer? Briefly explain how the same-origin policy prevents this attack, or why it does not.

Answer: The attack would work. The same-origin policy has no bearing on this attack, since no cross-site access is happening: the iframe is running in Ben's Bucks' origin, but Alice's site never attempts to interact with the contents of that iframe (which would be a SOP violation). Since the form is the true form loaded from Ben's Bucks, any CSRF tokens Ben has added will also be present. When the victim clicks "Free apples", the "Transfer" button is really what is pressed, which will look to Ben's Bucks as a legitimate transfer request. Note that Alice's site does not contain a form, and thus does not need the cookie or any CSRF tokens.

Alyssa tells Ben that he really ought to set the `SameSite: strict` option on the Zoobar session cookie. When this option is set, modern browsers will not include cookies set by origin X when making requests generated by pages that are not themselves from origin X.

13. [5 points]: Assuming all his users use modern browsers, briefly describe an attack that is no longer possible against Ben's bank, and why the header prevents it.

Answer: CSRF attacks from other domains are no longer possible, as the browser would not include any cookies necessary to authenticate the CSRF request. Note that attacks from the Bank's own origin are not defeated by this header, since the attacker could simply read out the tokens using a JavaScript request in that case. The clickjacking attack from the previous is question is not directly defeated: nothing prevents the transfer form from being submitted, since the transfer form itself is in Ben's Bucks' origin, and thus the submission of the form is a same-site request, which will include cookies. If *rendering* the form requires cookies, however, then the attack would fail, since the request for the transfer form would not succeed.

VI Side channels

Consider the Spectre example implementation shown in Appendix A of the paper "Spectre attacks: exploiting speculative execution" by Kocher et al. The attack assumes that the `victim_function` function exists in the address space of the victim. The attack measures accesses to `array2` to determine the secret `secret`.

14. [3 points]: If Intel turned off speculative execution on their processors would this attack still work? (Briefly explain your answer.)

Answer: If Intel turned off speculative execution, the attacker would not be able to get the code in the branch to be executed with an `x` that causes the access into `array1` to reach the secret and cause the indexed element of `array2` to be loaded in the cache.

15. [3 points]: If the attacker cannot cause `array2` to be evicted from the processor cache (i.e., there is no instruction to flush the cache and the attacker cannot evict the cache indirectly), is the Spectre attack still possible? (Briefly explain your answer)

Answer: No, because then the attacker cannot repeatedly measure which elements of `array2` have been accessed by `victim_function` and which haven't, so it cannot learn the secret.

16. [3 points]: Ben runs a memory-intensive program on one core and the example program of appendix A on another core. The two cores use a single shared cache. How will this impact the program's ability to identify the secret? (Briefly explain your answer.)

Answer: The memory-intensive program is likely to load or cause eviction of cache lines that the attacker is assuming contain elements of `array2`, so the attacker will have less confidence in the result of each measurement. Ben most likely must do more measurements to get a reliable answer.

Ben finds a code fragment similar to the `victim_function`:

```
void function1(size_t x) {
  if (x < array1_size) {
     temp &= array1[x] * 512;
  }
}
```

Ben also modifies the measurement code to flush and measure `array1`.

17. [3 points]: Can Ben learn the secret? (Briefly explain your answer.)

Answer: No. Measuring the presence/absence of cache lines in `array1` only reveals the value of `x`, not the secret.

VII Untrusted Version Server

Alice is designing a new internet "Version Server" to help people keep the software on their computers up to date. The idea is that software vendors would register the latest version number of each software package with the server. Peoples' computers would periodically run an "update client" that would ask the server for the latest version of each installed package, and tell the user which needed to be upgraded. The larger goal is to help people avoid running old software versions that have known bugs and vulnerabilities.

Alice is inspired by the 6.858 guest lectures about Keybase and Key Transparency and the "SoK: Secure Messaging" paper's Trust Management section. She would like her design to make it difficult for the Version Server to get away with maliciously providing incorrect answers.

For these questions, assume that the cryptographic primitives are used correctly, that the cryptographic algorithms cannot be broken, and that attackers cannot learn anyone else's private key.

Design D1. Here is Alice's first design, D1. Note that a real implementation might not follow the design faithfully, and might even be malicious. Alice's design specifies that the Version Server should keep a database of version records, one for each package identifier. Each version record is created by the package's vendor, and contains the following information:

```
struct VersionRecord {
    id: PackageId;    // Contains vendor ID
    version: int;     // Increased by one for a new release
    sig: Signature;   // signature over id and version, signed by vendor
}
```

A package ID contains the vendor ID so that clients know which public key to use to verify a version record. The version numbers for a package are increasing integers.

The D1 design provides two RPC calls:

- `put(packageID, versionRecord)`. A vendor calls `put(packageID, versionRecord)` whenever the vendor releases a new version or creates a new package. The server should validate the request: it should check that the versionRecord is correctly signed, and that its version number is higher than that packageID's version number already in the database (if any). If both are true, the server should update its database entry for the packageID.
- `get(packageID)`. A user's update client calls `get(packageID)` on each installed package. The RPC return value is a version record. The update client ignores the return value if it is not signed by the package's vendor.

Each user's computer knows the correct public key of the vendor of each installed package. The Version Server knows the public keys of all vendors.

18. [3 points]: Which of the following deceptions could a malicious server carry out without detection, given the above design?

(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False Cause a client to think a package ID isn't registered at all.

B. True / False Cause a client to think version $X+1$ is the latest, when actually the the highest version submitted by the vendor is X.

C. True / False Cause a client to think version $X-1$ is the latest, when actually the the highest version submitted by the vendor is X.

Answer: A is true, since the design doesn't have a way for the client to verify a "no such package" reply. B is false, since the version server would not be able to produce the required vendor signature. C is true, since the server could return the previously registered version record for the older version.

Design D2. Alice thinks she can do better, and creates a second design, D2. In this design, the Version Server should arrange all received version records in a log. The log should be a sequence of log records with the following fields in each log record:

```
struct LogRecord {
   versionR: VersionRecord;  // version record (as in D1)
   hash: Hash;               // the cryptographic hash of the previous log record
   sig: Signature;           // signature over versionR and hash, signed by Version Server
}
```

The first (oldest) log record contains zeros instead of a cryptographic hash. The Version Server should maintain a single log containing all submitted version records for all packages. The most recent log record is called the *tail*.

The design supports two RPCs:

- `put(packageID, versionRecord)`. When the Version Server receives `put(packageID, versionRecord)`, it validates the version record (as in D1), creates a new log record with the version record and the hash of the previous tail, signs it, and adds it to the log as the new tail.
- `fetch()`. The `fetch()` RPC returns the entire log – all of the log records.

The server should process incoming RPCs one at a time, completing each before starting the next one.

When a user's update client needs to check installed packages, it calls `fetch()`. The client checks the log that `fetch()` yields as follows:

A. It checks that the log records form a proper log: that there is some total order of log records for which each log record in the order is pointed to by the hash in its successor.

B. The client checks that every log record is correctly signed by the Version Server. All clients know the Version Server's public key.

C. The client checks that the version record in each log record is correctly signed by the relevant vendor.

D. The client checks that the hash in the earliest log record is zero.

The update client rejects the `fetch()` result if any of these checks fail.

If the server correctly follows Alice's design, then (in the absence of `put()`s) all clients will see the same log from `fetch()`, reflecting the sequence of `put()`s processed by the server. In the presence of `put()`s, later fetches may see a longer log, so a more accurate statement is that for any two logs seen by clients or stored on the server, the shorter log will be a *prefix* of the longer log: they will be identical up to the length of the shorter of the two logs. If this prefix property doesn't hold, then the server is not following Alice's design, and therefor may be malicious.

Consider the following sequence of events:

- SecureChat v2 (version 2) is released, and its vendor calls `put` to add its version record to the Version Server's log.
- After fixing a bug, the vendor calls `put` to add the version record for SecureChat v3.
- Other vendors call `put`.
- A vendor releases the brand-new game Angry Turtles v1 and calls `put()` for it.

Bob's update client calls `fetch()`, and the resulting log contains a record for SecureChat v3 and, later in the log, Angry Turtles v1. The `fetch()` results pass all the client checks explained above.

However, when Alice calls `fetch()`, the server maliciously omits the SecureChat v3 record, though it does include SecureChat v2. The server includes Angry Turtles v1; everyone has heard of this game, and Alice would be suspicious if `fetch()` omitted it. The `fetch()` results pass all the client checks explained above.

19. [5 points]: Bob and Alice both see a log record for Angry Turtles v1 in their `fetch` results. Are these log records identical, or different? Briefly explain.

Answer: The two Angry Turtles v1 log records must be different. Each log record contains the hash of the preceding log record, which itself depends (via each record's hash) on the content of all the preceding records in the log. Since the logs that the server gives Alice and Bob differ, the hashes that they see in the Angry Turtles log records must differ too.

Alice would like a fast "cross-validation" technique to help detect if the Version Server is hiding version records; she wants to do this without the expense of sending complete logs between clients. Every once in a while, Alice sends Bob h_a, the cryptographic hash of the tail log record from Alice's most recent call to `fetch()`. After Bob receives Alice's h_a, Bob calls `fetch()`. Bob computes the cryptographic hash h_b of the tail log record he got from his `fetch()`. Both `fetch()` results pass all the client checks explained above. Note that Bob does not have a copy of Alice's log. Bob and Alice are entirely trustworthy.

20. [4 points]: Which of the following are true?
(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False If h_a != h_b, the server must be intentionally hiding version records from either Alice or Bob.

B. True / False If $h_a = h_b$, then the Version Server cannot be hiding version records from Alice.

C. True / False If $h_a = h_b$, then Alice's and Bob's `fetch()`s must have returned identical logs.

D. True / False If Bob's `fetch()` result doesn't contain a record that hashes to h_a, then Alice's log is not a prefix of Bob's.

Answer:

A is false, because new put()s might have arrived between when Alice and Bob fetch()ed.

B is false, because the server might be hiding the same records from both Alice and Bob.

C is true, since if they differed at some point, all subsequent log records would also differ in their `hash` elements.

D is true. Suppose the log that the server sent Alice has n records. Bob calls fetch() after Alice. If Bob got a log that has fewer than n records, it's clear Alice's log isn't a prefix of Bob's. If Bob got a log with at least n records, then by assertion the hash of Bob's n'th record is not h_a. That means Bob's n'th record must differ from Alice's n'th record. That means that Alice's log isn't a prefix of Bob's log. This is a useful property because it gives Alice and Bob a quick way to discover some scenarios in which the server is dishonestly showing them different information.

VIII 6.858

We'd like to hear your opinions about 6.858. Any answer, except no answer, will receive full credit.

21. [2 points]: Are there any papers in the second part of the semester that you think we should definitely remove next year? If not, feel free to say that.

Answer: Tangled Web (23), Secure Messaging (13), TCP/IP (12), SSL (10), Spectre (9), HTTPS/SSL (8), Bitcoin (7), Tor (5), Keybase (4), Android (2), OWASP (2), EXE (2)

22. [1 points]: Are there topics that we didn't cover this semester that you think 6.858 should cover in future years?

Answer: Current Events/Recent Attacks (12), Cryptography (8), Botnets (7), Cryptocurrencies (7), Hardware Security (6), Malware (6), Kerberos (5), OS Security (5), More Network Security (5), IoT (5), Firewalls (3), Meltdown (3), Machine Learning (3), Personal Web Security Strategies (3), Penetration Testing (3), Ethical Hacking (3), Forced HTTPS (2), Worms (2), Social Engineering Attacks (2), Anonymity/Privacy (2), Filesystems (2), SQL Injection (2), Timing Attacks (2), Distributed System Security (1), DNSSEC (1), Network Security (1), SUNDR (1), Static Analysis (1), Key Management (1), API Security (1), More Device Security (1), Javascript (1), What's Next (1), Homomorphic Encryption (1), Quantum Computing (1), TLS (1), Database Security (1)

End of Quiz