Popa & Wagner
Spring 2016

CS 161
Computer Security

Midterm 2

Problem 1 ***True or False*** **(10 points)**

Circle True or False. Do not justify your answer.

(a) TRUE or FALSE: It is safe (IND-CPA-secure) to encrypt multiple messages in CBC mode with a constant IV of 0, using the same encryption key each time.

> **Solution:** If you encrypt the same message twice, you'll get the same ciphertext both times, so this can't be IND-CPA-secure.

(b) TRUE or FALSE: It is safe (IND-CPA-secure) to encrypt multiple messages in CBC mode with a constant IV of 0, as long as the encryption key is different for each message sent.

> **Solution:** Using a different random key each time provides the randomization needed to ensure that each message encrypts to a different random-looking ciphertext—even if you encrypt the same message twice, you'll get two different unrelated ciphertexts.

(c) TRUE or FALSE: Encrypting a message with CBC mode protects the integrity of the message.

> **Solution:** Encryption doesn't provide integrity, as you saw in HW3 Problem 2(c).

(d) TRUE or FALSE: It's ok for multiple people using El Gamal public key encryption to use the same modulus p.

> **Solution:** Knowing p doesn't help you compute Bob's private key (b) from his public key ($g^b \bmod p$).

(e) TRUE or FALSE: It's ok for multiple people using RSA signatures to use the same modulus n.

> **Solution:** If two people have the same n, then they'll have the same d, i.e., the same private key. This means Alice will be to sign messages that look like they came from Bob.

Problem 2 ***More True or False*** **(8 points)**

In this question, H refers to a secure cryptographic hash function and len(x) is a 128-bit int storing the length of x. You can assume that x and y are at most one million bytes long. Circle True or False. Do not justify your answer.

(a) [True] or False: Let $F(x, y) = H(x||y)$. Given x, y, and $F(x, y)$, it is easy for an attacker to find x', y' such that $F(x', y') = F(x, y)$ and $x \neq x'$.

> **Solution:** Just shift the boundary. For instance if $x =$ `builtin` and $y =$ `securely`, you could use $x' =$ `built` and $y =$ `insecurely`.

(b) True or [False]: Let $F(x, y) = H(\text{len}(x)||x||y)$. Given x, y, and $F(x, y)$, it is easy for an attacker to find x', y' such that $F(x', y') = F(x, y)$ and $x \neq x'$.

> **Solution:** The input to H is uniquely decodable: given len(x)$||x||y$, x and y are uniquely determined. In other words, if len(x)$||x||y$ = len(x')$||x'||y'$, then we must have len(x) = len(x') (since the two input strings match in their first 128 bits), and thus $x = x'$ and $y = y'$ (there is no opportunity to shift the boundary, since x has to have the same length in both input strings).
>
> Thus: if $x \neq x'$, then len(x)$||x||y \neq$ len(x')$||x'||y'$ Since H is collision-resistant, this means $H(\text{len}(x)||x||y) \neq H(\text{len}(x')||x'||y')$.

(c) True or [False]: Let $F(x, y) = H(\text{len}(y)||x||y)$. Given x, y, and $F(x, y)$, it is easy for an attacker to find x', y' such that $F(x', y') = F(x, y)$ and $x \neq x'$.

> **Solution:** The same reasoning as in part (b): the input to H is uniquely decodable. If two inputs to H are equal, then len(y) must be the same in both, and there's no opportunity to shift the boundary.
>
> We felt it was sufficiently clear that len(y) is a 128-bit int containing the length of y. However, we gave credit for True for students who explicitly wrote "I assume len(y) is variable-length."

(d) [True] or [False]: Let $F(x, y) = H(x||\text{len}(x)||y)$. Given x, y, and $F(x, y)$, it is easy for an attacker to find x', y' such that $F(x', y') = F(x, y)$ and $x \neq x'$.

> **Solution:** We gave points to both answers here, based on student feedback.
>
> We can shift the boundary if x is chosen cleverly. Let x' be arbitrary, and suppose we choose x so that $x = x'||\text{len}(x')$. Set $y' = \text{len}(x)||y$. Then you can verify that $x||\text{len}(x)||y = x'||\text{len}(x')||y'$, so $H(x||\text{len}(x)||y) = H(x'||\text{len}(x')||y')$.
>
> However some students pointed out that they interpreted the question as asking

whether or not such an attack can always be done, for all x, y. Indeed, it can't be done for all x, y; only for x, y with a special structure. Therefore, we gave credit for either True or False on this question.

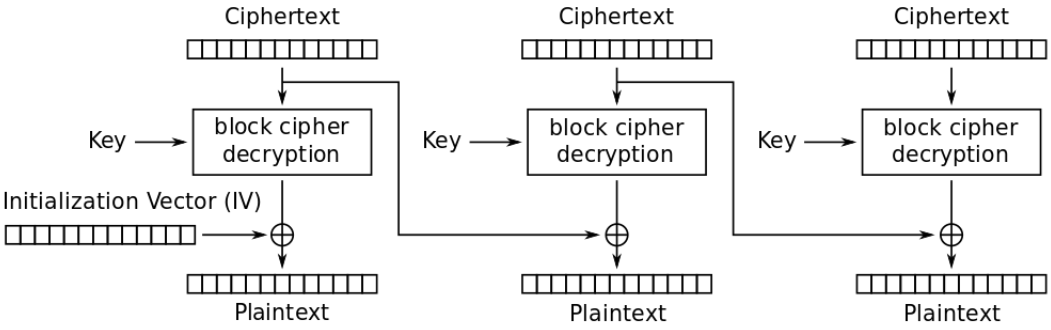
Problem 3 ***Multiple choice*** **(6 points)**

Circle all the options that apply.

(a) When using the CBC block chaining mode, if the IV is modified by an attacker during transmission (so the correct IV was used during encryption, but the receiver receives the modified value), the recipient can still successfully decrypt:

1. the entire ciphertext
2. none of the ciphertext
3. only the first block
4. [all but the first block]
5. other set of blocks than above

Solution: CBC decryption looks like this:



Cipher Block Chaining (CBC) mode decryption

(source: Wikipedia)

An incorrect IV will lead to an incorrect first plaintext block, but the rest will actually be correct, because they do not depend on the IV at all, only the previous ciphertext block.

Post-exam clarification: option (5) was intended as a "none of the above".

Mathematically: (note that the IV here is denoted C_0)
So we receive $C_0', C_1, C_2, \ldots, C_n$ instead of $C_0, C_1, C_2, \ldots, C_n$. We decrypt using the equation $M_i = D_k(C_i) \oplus C_{i-1}$. This means the recipient will get M_1 wrong, but $M_2, M_3, \ldots, M_n$ will all be correct.

(b) Which of the following are properties of CTR mode?

1. [encryption can be parallelized]
2. [decryption can be parallelized]
3. [the nonce does not have to be random, as long as it never repeats]
4. [it turns a block cipher into a stream cipher]
5. it is more secure than CBC mode
6. it provides integrity and authentication for the message

Solution: Note: Item 3 was ignored for the purposes of grading, because in lecture we specified that the nonce *should* be random. Also, the nonce does not actually have to be random, as long as it does not ever repeat for any block encrypted using a given key. (There are two variants of CTR mode, and the requirement on the nonce depends which variant you are using. One variant uses $C_i = M_i \oplus E_k(IV||i)$, and for that variant, the IV doesn't need to be random; it just needs to never repeat. Another variant uses $C_i = M_i \oplus E_k(IV + i)$; for that variant, the IV needs to be random, or at least to ensure that you never have $IV + i = IV' + i'$ for any pair of messages.)

CTR isn't more secure than CBC mode: they're both secure if they are used and implemented correctly. If used and implemented correctly, there are no known attacks on either.

Problem 4 *TLS* (16 points)

An attacker is trying to attack the company Wahoo and its users. Assume that users always visit Wahoo's website with an HTTPS connection, using RSA and AES encryption (no Diffie-Hellman). (You may assume that Wahoo does not use certificate pinning—if you don't know what that is, you can ignore it.) For each of the following attack scenarios, circle all of the options that an attacker could achieve in that attack scenario.

(a) If the attacker obtains a copy of Wahoo's certificate, the attacker could:

1. impersonate the Wahoo web server to a user
2. discover some of the plaintext of data sent during a past connection between a user and Wahoo's website
3. discover all of the plaintext of data sent during a past connection between a user and Wahoo's website
4. replay data that a user previously sent to the Wahoo server over a prior HTTPS connection
5. [none of the above]

Solution: The certificate is public. Anyone can obtain a copy simply by connecting to Wahoo's webserver. So, learning the certificate doesn't help the attacker.

(b) If the attacker obtains the private key of a certificate authority trusted by users of Wahoo, the attacker could:

1. [impersonate the Wahoo web server to a user]
2. discover some of the plaintext of data sent during a past connection between a user and Wahoo's website
3. discover all of the plaintext of data sent during a past connection between a user and Wahoo's website
4. replay data that a user previously sent to the Wahoo server over a prior HTTPS connection
5. none of the above

Solution: The attacker can't decrypt past data, because the attacker doesn't learn Wahoo's private key—only the CA's private key. All that the CA's private key can be used for is to create bogus certificates, which can be used to fool the client into thinking it is talking to Wahoo—but doesn't allow learning past data. Replays aren't possible, due to the nonces in the TLS handshake.

(c) If the attacker is a man in the middle on a HTTPS connection between a user and Wahoo's website, the attacker could:

1. impersonate the Wahoo web server to this user
2. discover some of the plaintext of data sent during *this* connection
3. discover all of the plaintext of data sent during *this* connection
4. discover all of the plaintext of data sent during a *past* connection between a user and Wahoo's website
5. replay data that a user previously sent to the Wahoo server over a prior HTTPS connection
6. [none of the above]

Solution: TLS is secure against man-in-the-middle attacks.

(d) Suppose the attacker obtains the private key that was used by Wahoo's server during a past connection between a victim and Wahoo's server, but not the current

private key. Also, assume that the certificate corresponding to the old private key has been revoked and is no longer valid. This attacker could:

1. impersonate the Wahoo web server to this user
2. discover all of the plaintext of data sent during a *current* connection (one where the current private key is used) between a user and Wahoo's website
3. discover all of the plaintext of data sent during a *past* connection (one where the old private key was used) between a user and Wahoo's website
4. none of the above

Solution: Since the server is using RSA, an attacker who learns the RSA private key can decrypt past sessions (the attacker can decrypt to learn the pre-master secret, derive the symmetric keys, and decrypt all of the data). This can't be used to impersonate the Wahoo server, because the attacker doesn't have a valid certificate for the RSA public key that was compromised.

Problem 5 ***Iliad Identification Integrity scheme*** **(15 points)**

Jeff is hired by the big computer company, Iliad, to do a security review of their devices. All of Iliad's devices are assigned a unique *identification number*. Iliad's devices use the Iliad Identification Integrity (I^3) scheme to protect the integrity of identification numbers. Each of Iliad's devices has a unique key K embedded in the hardware that can only be used for verification with the I^3 scheme and can't be extracted by any means. In the I^3 scheme, when the device is manufactured, it does the following:

1. Generate a random 16-byte identification number N. Output N via a direct physical link to the factory machine.
2. Generate a random 16-byte IV.
3. Define the plaintext P to be N followed by 16 zero-bytes. Encrypt P with AES-CBC using the IV and device's embedded key K, resulting in ciphertext C.
4. Store N, IV, and C on the device's flash storage.

When Iliad's products are powered up, they will obtain N as follows:

1. Read IV and C from flash storage.
2. Decrypt C using IV and embedded key K, to get the plaintext P'.
3. If the last 16 bytes of P' are 0 and the first 16 bytes of P' match the N stored in flash, return N. Otherwise, signal an error.

The flash storage has no other protections, and an attacker could potentially tamper with the data stored on flash.

Answer the following questions.

(a) Jeff has a hunch that the scheme does not actually protect the integrity of N. Briefly, what's the reason for Jeff's hunch? One sentence should be enough.

Solution: Encryption doesn't provide integrity.

(b) Upon further investigation, Jeff determines that the I^3 scheme is in fact insecure. Describe how you can modify N, IV and C to have the verification process accept at least one other N.

Solution:
The original encryption process is:

$$C_1 = \mathrm{E}(IV \oplus N)$$

$$C_2 = \mathrm{E}(C_1 \oplus 0)$$

Therefore you also know:

$$\mathrm{D}(C_1) = IV \oplus N$$

$$\mathrm{D}(C_2) = C_1$$

According to the problem statement "key K embedded in the hardware can only be used for verification with the I scheme and can't be extracted by any means." Therefore, you are not allowed to compute any E() or D() with inputs other than the ones above as part of your solution. Additionally, your modifications must maintain the following invariants to be accepted by the verification process as described:

$$0 = \mathrm{D}(C_2') \oplus C_1'$$

$$N' = \mathrm{D}(C_1') \oplus IV'$$

The following answers are definitely correct:

Modified value of $N = N'$ (where N' can be arbitrary $\neq N$)
Modified value of $IV = IV \oplus N \oplus N'$
Modified value of $C = C$

Alternate answer:
Modified value of $N = N \oplus X$ (X can be anything other than 0)
Modified value of $IV = IV \oplus X$
Modified value of $C = C$

Alternate answer: swap N and IV, leave C unchanged.

(c) Jeff now needs to recommend a change to the scheme to make it secure, so that Iliad won't lose face. Describe a change to the I^3 scheme so that it will actually protect the integrity of N.

Solution: Store a MAC of N, instead of encrypting.

Or: store $N, \text{MAC}_k(N)$.

Or, you could use a digital signature. You'd need to modify the key generation / key handling to make it work. The factory could generate a RSA keypair (needs to be unique for each device), embed the public key in hardware (so it can't be changed), and store N and a signature on N in the flash memory of the device. Then the device can verify using the embedded public key.

Problem 6 ***Name confidentiality in Project 2*** **(12 points)**

Jamie is working on Project 2 (Part 1) and had several ideas for how to keep the filename secret. Jamie will store the file at id i, where i is computed from the filename n in some way. For each approach listed below, circle either "secure" or "broken" according to whether it would meet the requirements for filename confidentiality from Project 2 or not.

You can assume CBC mode encryption uses a symmetric key that is generated and stored securely and not known to the attacker. H represents the SHA256 cryptographic hash function. Don't justify your answer.

(a) SECURE or BROKEN: $i = n$.

Solution: Reveals the name to the storage server.

(b) SECURE or BROKEN: $i = H(n)$.

Solution: Not secure when n has low entropy, due to dictionary attacks. See the Project 2 Part 1 solutions for more explanation.

(c) SECURE or BROKEN: i is a RSA signature on n, using Jamie's private RSA key.

Solution: Not secure, since the attacker can test a guess at n using the verification algorithm and Jamie's public key. See the Project 2 Part 1 solutions for more explanation.

(d) SECURE or BROKEN: Encrypt n using AES in CBC mode, with all-zeros IV, and use the resulting ciphertext (excluding the IV) as i.

Solution: Not secure, since the first 16 bytes of the ciphertext depends only on the first 16 bytes of n. Consequently, it is vulnerable to chosen-name attacks. See the Project 2 Part 1 solutions for more explanation.

(e) [SECURE] or [BROKEN]: Encrypt n using AES in CBC mode, with all-zeros IV, and use the last 16 bytes of the resulting ciphertext as i.

> **Solution:** We gave full credit for both Secure and Broken.
>
> This is equivalent to AES-CBC-MAC with a single key (like AES-EMAC, but without the final computation using the second key). As Prof. Popa mentioned in lecture, AES-CBC-MAC is not secure when used with variable-length messages (as the case here). See also the Wikipedia page on CBC-MAC for more explanation. As a result, there is a complicated attack that can be used to defeat name confidentiality, if there is no restriction on the characters in filenames.
>
> However, it was pointed out that in Project 2 we promised that filenames would use only alphanumeric characters. As a result, that attack wouldn't be possible, within the filename constraints listed in Project 2. Therefore, we decided to give credit for both answers.

(f) [SECURE] or BROKEN: Encrypt n using AES in CBC mode, with all-zeros IV, and let j denote the resulting ciphertext. Use $i = H(j)$.

> **Solution:** Secure, because j has high entropy, and hashing a high-entropy value is safe (it's not vulnerable to dictionary attacks).
>
> Or, another way to think about it: j depends on both n and k (the AES key), so this is much like $H(n||k)$, which is secure, as explained in the Project 2 Part 1 solutions.

Problem 7 *Computing on encrypted data* (10 points)

A cool property of some encryption schemes is that they allow you to compute on encrypted data! Let $\mathsf{Enc}(M)$ denote the encryption of message M. Given C_1 and C_2 (the ciphertexts for messages M_1 and M_2), anyone (even without the decryption keys) can compute a ciphertext C_3 that will decrypt to the product $M_1 \times M_2$.

The idea is that we want a server in the cloud to do some work for us, but we don't want the cloud to see our data. So we give encrypted data to the cloud (without giving the decryption key to the cloud), and the cloud gives us back the encrypted computation result. Let's figure out how the cloud can perform this computation.

(a) Recall the El Gamal scheme: The El Gamal public key is (p, g, h), where x is the private key and $h = g^x \bmod p$. The encryption of a message M is $\mathsf{Enc}(M) = (g^r \bmod p, M \times h^r \bmod p)$, for a random r.

Given only $C_1 = \mathsf{Enc}(M_1) = (s_1, t_1)$ and $C_2 = \mathsf{Enc}(M_2) = (s_2, t_2)$ and the El Gamal public key, show how the cloud can compute a ciphertext $C_3 = \mathsf{Enc}(M_1 \times M_2 \bmod p)$. In other words, show how the cloud can compute a ciphertext C_3 that will decrypt to $M_1 \times M_2 \bmod p$. Show an equation that the cloud can use to compute C_3:

Solution: $C_3 = (s_1 s_2 \bmod p, t_1 t_2 \bmod p)$

(b) Suppose the cloud has two plaintext values u_1, u_2, each a 1024-bit number. Suppose the cloud also has two ciphertexts $C_1 = \mathsf{Enc}(x_1)$ and $C_2 = \mathsf{Enc}(x_2)$ that were computed using RSA encryption, and the cloud knows the RSA public key, but the cloud doesn't know x_1 or x_2 or the RSA private key. How can the cloud compute a ciphertext $C_3 = \mathsf{Enc}(x_1^{u_1} \times x_2^{u_2} \bmod n)$ efficiently? You don't need to justify your answer or explain why your solution works.

Solution: $C_3 = C_1^{u_1} \times C_2^{u_2} \bmod n$

This question was poorly drafted. We didn't teach you RSA encryption in this class, so we shouldn't have asked you about this. Therefore, we gave full credit to everyone on this question.

In fairness to those who spent a significant amount of time working on this question and did figure out how to answer it, we gave bonus points to anyone who answered this question correctly. The bonus points allow to earn back points you lost on other questions. (The total score on the midterm was still capped at 100.)

Problem 8 *Protocol analysis* (15 points)

Alice and Bob want to simulate flipping a fair coin. Ideally, each of them would like to be sure that the other can't "cheat" and force the coin to be heads or tails.

For each of the following schemes, determine whether the scheme is secure or not and then circle "Secure" or "Broken". If you circle secure, you don't need to justify your answer. If you circle broken, give the attack that either Alice or Bob would mount to give them better than 50% chance of obtaining heads.

In the following, let p be a 2048-bit prime number, g a generator modulo p, and H a secure cryptographic hash function; these are fixed in advance and known to everyone. You can assume that both parties complete the protocol. That is, neither party will refuse to finish the protocol.

(a) Alice randomly picks a such that $0 < a < p$ and sends $g^a \bmod p$ to Bob. Then, Bob randomly picks b such that $0 < b < p$ and sends $g^b \bmod p$ to Alice. Then, both compute $g^{ab} \bmod p$. If $g^{ab} \bmod p$ is even, the coin flip is heads; if odd, the coin flip is tails.

SECURE or BROKEN

Solution: Attack: Before Bob sends anything, he can check whether his choice of b would cause the coin flip outcome to be heads; if not, he can pick a new b and try again.

Alternate solution: a malicious Alice chooses $a = p - 1$, then $g^{ab} = 1 \bmod p$ regardless of b, so the coin flip always comes up tails. (The problem statement said to make it come out heads, but we accepted this answer as well.)

(b) Alice randomly picks a such that $0 < a < p$ and sends $g^a \bmod p$ to Bob. Then, Bob randomly picks b such that $0 < b < p$ and sends b to Alice. Then, Alice sends a to Bob and Bob checks that it matches what Alice sent earlier. If $H(a||b)$ is even, the coin flip is heads; if odd, the coin flip is tails.

SECURE or BROKEN

Solution: Explanation: Once Alice has sent $g^a \bmod p$, she is committed: there is only a single value a she'll be able to send later, and she can't "change her mind" or lie about what value of a she had in mind, without being detected. Therefore, there's no way for Alice to cheat Bob.

Next consider Bob. His only opportunity to try anything malicious is after Alice has sent $g^a \bmod p$ but before he has sent b (once he sends b, the outcome is determined and there's no going back). But as we saw when studying Diffie-Hellman, knowing $g^a \bmod p$ doesn't help you find a. So, when Bob receives $g^a \bmod p$, he can't find a, so he won't be able to predict the value of $H(a||b)$ at that point.

(c) Alice randomly picks a to be either 0 or 1 and sends $H(a)$ to Bob. Then, Bob randomly picks b to be either 0 or 1 and sends b to Alice. Then, Alice reveals a to Bob and Bob checks that it matches what Alice sent earlier. If both picked 0 or both picked 1, the coin flip is heads. If one picked 0 and the other picked 1, the coin flip is tails.

SECURE or BROKEN

Solution: Attack: Given $H(a)$, Bob can find a, since there are only two possibilities for a. Then he sends $b = a$.

(d) Alice randomly picks a such that $0 < a < 2^{128}$ and sends $H(a)$ to Bob. Then, Bob randomly picks "even" or "odd" and sends that to Alice. Then, Alice reveals a to Bob and Bob checks that it matches what Alice sent earlier. If Bob's guess about a was right (e.g., Bob picked "even" and a is even, or Bob picked "odd" and a is odd), the coin flip outcome is heads, otherwise it is tails.

SECURE or BROKEN

Solution: Explanation: Similar to part (b), once Alice has sent $H(a)$, she's committed—she won't be able to later lie about what value of a she had in mind, because that would require finding a collision in A. Therefore, there's no way for Alice to cheat Bob.

The situation for Bob is similar to part (b). The preimage resistance of H means that, after seeing $H(a)$, Bob can't learn a, and Bob can't predict whether a is even or odd—so he has only a 50% chance of guessing right.

(e) Alice randomly picks a such that $0 < a < 2^{128}$ and randomly picks an AES key k. Alice computes $c = E_k(a)$ [the AES-CBC encryption of a, under a random IV; c includes the IV] and sends c to Bob. Then, Bob randomly picks "even" or "odd" and sends that to Alice. Then, Alice reveals a and k to Bob and Bob checks that it matches what Alice sent earlier. If Bob's guess about a was right (e.g., Bob picked "even" and a is even, or Bob picked "odd" and a is odd), the coin flip outcome is heads, otherwise it is tails.

SECURE or BROKEN

Solution: Attack: After Alice receives Bob's guess, Alice can check whether the outcome will be heads if she continues honestly. If yes, she can reveal her a and k. If not, she can pick another key k', compute $a' = D_{k'}(a)$ [the AES-CBC decryption of c, under key k'], and check if a', k' would lead to an outcome of heads. She keeps picking k' randomly until she finds a', k' that would lead to an outcome of heads, and she sends this to Bob. Note that everything will look OK as far as Bob can tell, since a', k' are consistent with c.

In other words, Alice can "change her mind" about what a was after seeing Bob's guess. This makes it easy for her to cheat and ensure the outcome is heads.

Problem 9 ***RSA keypairs*** **(8 points)**

You want to generate a RSA keypair and store the private key on two different servers, one in New York and the other in California.

You have a high-security key storage device that can store up to 150 bits of secret key material, and will be highly resistant to tampering or reverse engineering. Corporate policy says that secret key material must not leave those two servers, with the sole exception that it can be copied between a server and the key storage device via USB. You are not allowed to transport or communicate any secret key material except on this key storage device. (For instance, storing a copy of your private key on your laptop's hard drive is prohibited. Sending your private key over the Internet is prohibited, even if it is encrypted before transmission.) The key storage device must never leave your personal possession. You have only one key storage device and can only afford to take one trip from California to New York.

Describe how you can securely generate such a keypair and install it on both servers, with only one trip from California to New York.

Solution: Use a cryptographically secure PRNG. Store the seed on the key storage device. Generate a random RSA keypair, but whenever the key generation algorithm asks for random bits, instead use the next pseudorandom bit from the CSPRNG. Then both NY and CA will generate the same keypair, since they start from the same seed.

Or: Pick a 150-bit for a CSPRNG; use the output of the CSPRNG as the randomness for the RSA key generation algorithm.

Or: Pick a random AES key k and store it on the key storage device. Generate the infinite stream of pseudorandom bits $\text{AES}_k(0), \text{AES}_k(1), \text{AES}_k(2), \ldots$ (like in counter mode). Do RSA key generation, but use pseudorandom bits from the stream in place of the random bits needed by key generation.

Note: This is similar to how TLS derives symmetric keys from the pre-master secret (seed), except here we are deriving asymmetric keys from the seed, so it's a bit more challenging.

The challenging part of this problem is that RSA keys are 2048 bits long. A 150-bit RSA key (e.g., 75-bit primes p, q) is insecure: it's easy to factor numbers of that size. Therefore, simply storing the private key on the storage device doesn't work.

Using the output of a CSPRNG directly as the private key doesn't quite work. You can't use the output of a CSPRNG directly as p, q, as likely the resulting numbers won't be prime. With the form of RSA we showed you in class, the public exponent is 3; as a result, using the output of a CSPRNG directly as d won't work, because we need d to satisfy $3d = 1 \pmod{(p-1)(q-1)}$, and a random d probably won't satisfy that equation.

If you read about RSA elsewhere, you might have read about another variant of RSA where the public exponent e can be arbitrary, so long as $ed = 1 \pmod{(p-1)(q-1)}$. With this variant of RSA, it's possible to put together a solution that almost works: store a CSPRNG seed on the storage device, then use the first 2048 bits of output of the CSPRNG as d. The server in CA generates random primes p, q, computes $n = pq$ and then $e = 1/d \pmod{(p-1)(q-1)}$ and publishes e, n. We travel to NY with the storage device, and the server in NY rederives d, then downloads the public key e, n from the CA server over the Internet. This almost works, but not quite: not all values of d will work; we need d to be relatively prime to $(p-1)(q-1)$ for this to work, and a random d might or might not satisfy those equations. (For instance, d needs to be odd; that happens with probability only 1/2.) You could fix up this solution by having the server in CA pick a new seed repeatedly until it finds one where the corresponding d is OK, though then the solution ends up being a bit complicated to describe.

You can't compress the private key. The private key will look random, and random data is incompressible.