SE 4472 - Information Security

Study Guide (Fall 2018)

Security Notions

Concepts

- Rules to live by:
    - Kerkhoff's principle
        - The security of an encryption scheme should be based on the secrecy of the key, not the secrecy of the encryption algorithm. More specifically: always assume the encryption scheme is publicly known.
    - Don't roll your own
        - Crypto algorithms and software implementations are really really ridiculously easy to get wrong in subtle ways that can easily turn into big vulnerabilities. It only takes one line of code (e.g., Apple's goto fail, or Heartbleed) to mean the difference between secure and vulnerable.
    - Don't assume something has a certain form unless you check it
- Brute-force attack
    - Try every key and/or message until one "works"
- Bits of security
    - Exponential value describing how many operations (encryption, hashes, etc) are necessary to recover the message and/or key in a particular cryptosystem. E.g. 128 bits of security means the attacker has to do on the order of 2^128 operations to succeed
    - If a system can be broken in approximately 2^b operations, we say it has b-bits of security.

- Negligible quantity
    - A value that is less than one over any polynomial function with degree less than or equal to the security parameter
- Indistinguishability
    - The probability that you can tell the difference between two things is less than a negligible quantity
- Pseudo-random functions
    - A random looking mapping of inputs to outputs
    - A potentially many-to-one mapping
    - Inverse does not necessarily exist
- Pseudo-random permutations
    - One-to-one mapping
    - Image and pre-image sets are equivalent
    - An unique inverse exists for every element
- Oracles
    - An abstract black box. Ask a question, get an answer
    - Oracles considered:
        - Encryption Oracle
            - Here's the encryption of your query
        - Decryption Oracle
            - Here's the decryption of your query
        - Random Oracle
            - Here's a fixed-length random value that I associate with this particular query
        - Padding Oracle
            - Yes or No: Your query is the encryption of a plaintext with valid padding

Attack Games

Attack games are a way to qualitatively addressing the following question: how much information can about a plaintext by seeing its associated ciphertext.

Features common to all games

Games involve two players A and B. The game begins with B generating an independent random secret encryption key (n.b., every time the game is played, B picks a fresh key). A chooses two messages m1 and m2 and sends them to B. B flips a coin and chooses one of the messages, mb, and encrypts it. This value c=Enc(mb) is called the *challenge*. Challenge c is returned to A. The game concludes with A guessing if b=1 or b=2, i.e., if c=Enc(m1) or c=Enc(m2). If A guesses correctly, she is said to win the game. The encryption scheme is said be indistinguishable if A has negligible advantage over a random guess of guessing b.

Game types / Security levels

1. Eavesdropping attack (EAV)
   - A doesn't get to make any queries.

2. Chosen plaintext attack (CPA)
   - B behaves as an encryption oracle. That is, at any point in the game, B will encrypt any plaintext and return the ciphertext to A. A variant is called an adaptive chosen plaintext attack in which B continues to behave as an encryption oracle

3. Chosen ciphertext attack (CCA1)
   - Prior to the challenge B will decrypt any ciphertext and return the plaintext to A

4. Adaptive chosem ciphertext attack (CCA2)
   - B with decrypt any ciphertext and return the plaintext to A both before and after the challenge. The only limitation is that A cannot submit the challenge to be decrypted (otherwise the game is trivial).

Implications of Security Levels

- IND-CCA2 implies IND-CCA1
- IND-CCA1 implies IND-CPA
- IND-CPA implies IND-EAV

How to achieve security levels (informal)

- IND-CCA2
    - It should not be possible to create a valid ciphertext without knowledge of a private/secret key
    - Achieved in practice through the use of message authentication codes
- IND-CPA
    - Encrypting the same message twice should give two totally different looking ciphertexts
    - Achieved in practice by using randomized encryption
- IND-EAV
    - You should have negligible advantage telling the difference between ciphertext, and random noise

---

Symmetric-key Primitives

Block Ciphers

Used for efficient bulk encryption of data. Encryption takes message (plaintext) and key and produces encryption (ciphertext). Decryption takes a ciphertext and key and produces a plaintext. One secret key used for both encryption and decryption.

- Ideal functionality:
    - Pseudo-random permutation
        - Imagine shuffling a deck of cards
        - For a given key, every message has a unique ciphertext, and vice versa
    - Secret key determines the particular permutation
    - Fixed length input maps to fixed-length output
- Feistel Network
    - A simple, provable way to take a pseudo-random function and turn it into a pseudo random permutation
- Cipher modes of operation examined:
    - Electronic Codebook Mode (ECB)
        - Not IND-EAV secure

- Cipher Block Chaining (CBC)
    - Requires an initialization vector (IV)

- Counter Mode (CTR)
    - A block cipher that acts like a stream cipher
    - Also requires an IV
    - Pros: random access, no decryption function needed

- Initialization vectors (IVs)

  - Works toward CPA security goal by making encryption of related messages different
  - Public value transmitted with ciphertext, necessary for decryption
  - Must be independently and randomly generated
  - Must be unpredictable to adversary, otherwise encryption oracle attacks are possible

- Block ciphers examined:

  - DES
    - Feistel network
    - 56 bit key, 64-bit block

  - AES
    - A combination of byte-level Galois Field arithmetic operations in GF(2^8)
    - 128-bit block, 128-, 192-, and 256-bit key options

Sample Problems

- Fortuna is a pseudo-random number generator based on AES in counter mode. Recall that CTR mode encrypts a successively incrementing counter value and xors it with a message. Suppose you don't xor the key stream with any message, you just output it. This could form a useful pseudo-random number generator. This is a actually a pretty decent approach, but it's not perfect. After a certain amount of output you can distinguish between Fortuna's output and that of a perfectly random sequence. How much output would you need to expect to be able to do this?

  - A: recall a block cipher is a pseudo-random permutation, i.e., every plaintext maps to a unique ciphertext. That means if you were using counter mode, you'd never see a ciphertext block repeat until the counter rolled over. AES has a 128-bit block, meaning you wouldn't see a block repeat until 2^128 blocks were generated. A truly random sequence, however, does not have this property. Each block is randomly generated, and thus has a small probability of matching a previously generated block. So how many blocks would it take before you'd expect to see a repeat (i.e., where the probability was >=1/2)?

Hash Functions

Used for producing a "fingerprint" or "digest" of a message. Hashing accepts a message and produces a hash (doesn't use a key in its basic form). Used for checking file integrity, storing passwords, and for making certain public-key operations more efficient.

- Ideal functionality:
    - Random oracle
        - Input: arbitrary length message
        - Output: fixed length string, l bits long where l is the output length
        - Given an input message m, the oracle flips l coins and associates m with the result in a giant lookup table
        - Every time you input m again, it gives you back the same l-bit result
        - Cannot exist in practice (requires infinite memory)
        - Differs from a real hash function in that the hashes of two messages m and m' are chosen by completely independent coin tosses, whereas a real hash function generates the hashes using a deterministic (though highly non-linear) function.
- Terms:
    - Pre-image: input to hash function (message)
    - Image: output of hash function (sometimes called message digest, fingerprint or simply "hash")
    - Hash length: the length l (in bits) of the hash
    - Collision: two pre-images hash to the same value. Since pre-image space is unbounded, but there are 2^l images, collisions must exist due to the pigeon hole principle
- Necessary properties:
    - Pre-image resistance
        - Given a hash y, it should be hard to find an x s.t. h(x)=y
        - If hash function is indistinguishable from a random oracle, difficulty is 2^(l),

i.e., l-bits of security

- Second pre-image resistance
    - Given a pre-image x, it should be hard to find a second pre-image y such that h(x)=h(y)
    - If hash function is indistinguishable from a random oracle, difficulty is 2^(l)

- Collision resistance
    - It should be hard to find *any* pair x,y such that h(x)=h(y)
    - If hash function is indistinguishable from a random oracle, difficulty should be 2^(l/2), i.e., (l/2)-bits of security, due to birthday paradox

- Hash functions examined:

    - MD5

        - l = 128
        - Is no longer indistinguishable from random oracle
        - **NOT collision resistant**
        - Collisions can be generated in much less than 2^(128/2), as demonstrated in assignment 1
        - Still technically pre-image resistant, but the best practice is to not use it at all anymore

    - SHA1

        - l = 160
        - Collision resistant to 2^80
        - No longer meets NIST's minimum 112-bit security level for collision resistance
        - Can still be used for pre-image resistance, but SHA-256 is the current minimum overall best practice

    - SHA-256

        - l = 256
        - 128 bits of collision resistant

Sample Problems

- Suppose there are two files f1 and f2 and suppose SHA1(f1) = SHA1(f2). Are these files identical?

    - A: Possibly. If they were the same, they would definitely have the same hash value since hash functions are deterministic. If they are different, they still could possibly have the same hash (called a collision)

- Suppose two files f1 and f2 and suppose SHA1(f1) != SHA1(f2). Are these files identical?
    - A: No. As before, SHA1 is deterministic, meaning the same input always gives the same output. So if the outputs are different, its because the inputs are different

Message Authentication Codes (MACs)

Used for verifying the integrity of data by associating a fixed-length value called a 'tag' with a given message. The tag is derived from a message and a secret key.

- Tag creation
    - Input: message m, secret key k
    - Tag t = MAC_k(m)
    - Output t

- Tag verification

    - Input: message m', secret key k, tag t
    - Tag t' = MAC_k(m')
    - Output t' == t

- Ideal functionality:

    - Like a keyed hash function
    - Variable length input maps to fixed length output

- MACs examined:

    - HMAC, a MAC built from a hash function
    - GHASH, the MAC portion of AES-GCM
        - Essentially a polynomial evaluated over a Galois field

Authenticated Encryption (AE)

A means of securely packaging a cipher with a MAC under one common interface. Simplifies (i.e., protects developers from themselves) by preventing the plaintext from being returned if the MAC tag was invalid. Uses the encrypt-then-mac strategy.

- Authenticated Encryption:
    - Accepts: message m, encryption key ke, mac key km
    - c = Enc_ke(m)
    - t = MAC_km(c)
    - Output: (c,t)
- Authenticated Decryption:
    - Accepts: ciphertext c, tag t, decryption key ke, mac key km
    - Check tag: t' = MAC_km(c)
    - If t' != t
        - Output tag error
    - Else:
        - Return Dec_ke(c)

  Authenticated encryption function accepts a message, an encryption key, and a MAC key and produces a ciphertext and authenticator tag. Authenticated decryption accepts a ciphertext, an authenticator tag, a decryption key, and a MAC key. If the authenticator tag is valid, the function returns the plaintext, otherwise it returns an error condition.

- Modes:
    - Encrypt-then-MAC
        - MAC on ciphertext
        - Preferred choice
        - Used in AES/GCM
    - MAC-then-Encrypt
        - Append MAC to plaintext before encryption
        - Used in TLS
        - Not optimal (subtle attacks possible in some cases)
    - Encrypt-and-MAC
        - MAC plaintext, append to ciphertext
        - Used in SSH
- Encryption, MAC keys, and IVs must be independently generated

- Provides IND-CCA2 security Using encryption without a MAC is exploitable
    - In the CCA2 game, the attacker can use the decryption oracle to indirectly determine m
    - In practice a padding oracle attack can be used to bootstrap a decryption oracle to determine m
    - A MAC scheme prevents an attacker from being successful, because they cannot produce another valid ciphertext without knowledge of the MAC key.
- AE's examined:
    - AES/GCM
        - Encrypt-then-MAC
        - Encryption = AES in CTR mode. MAC = GHASH function, i.e., a MAC function that iteratively multiplies each ciphertext block by the MAC key in GF(2^128)

Sample Problems

- Prove AES-CTR is not IND-CCA2 secure. Argue why AES-CTR becomes IND-CCA2 secure when the ciphertext is MAC'd.

Asymmetric-key (Public-key) Primitives

Asymmetric-key primitives have two keys: one key is for performing *public* operations (called the public key), the other is for performing *private* operations (called the private key). Anyone can perform public operations, but only the key holder can perform the private operation.

Key-Exchange

Diffie-Hellman Exchange (DHE)

- Based on the hardness of solving the Discrete-logarithm problem, i.e., given a=g^x mod p, find x. The problem is hard if g generates a cyclic group of large, prime order q. Note if we're talking about discrete logarithms and we write g^x, the "mod p" is implied.

- DH domain parameters:
    - large prime p
    - prime q (such that q divides p-1)
    - generator g of order q (i.e., g^i mod p "generates" new numbers in the range i=0 to i=q-1. Once i>=q, the it repeats)
- DH keys:
    - private key x, a randomly generated number in the range 1<x<q
    - public key g^x
- Diffie-Hellman problem:
    - given g, g^a, g^b, compute g^ab
    - Hard to do if p,g,q are sufficiently large (see key lengths below)
- Diffie-Hellman key agreement:
    - A picks a random number a in the range 1<a<q and sends g^a to B
    - B picks a random number b in the range 1<b<q and sends g^b to A
    - A computes shared secret = (g^b)^a = g^ab
    - B computes shared secret = (g^a)^b = g^ab
    - Shared secret can now be used to derive symmetric keys, like AES encryption keys, MAC keys, etc.
- Ephemeral Diffie-Hellman
    - Denoted DHE/ECDHE
    - Fresh exponents (i.e., private keys) are generated for each connection
    - Once you've calculated the shared secret, you can delete the private key
    - Provides what's called "forward secrecy"
    - If you're a popular website that always used the same key, and gets hacked, and your private key is compromised, then all the messages you sent in the past could be decrypted. But if you use a new key every time, it limits the damage such an attack can achieve.
- Man-in-the-middle attacks
    - Attacker intercepts g^a and sends its own value g^c to B
    - Attacker intercepts g^b and sends its own value g^d to A
    - Attacker now has a shared secret with B: g^bc

  - Attacker now has a shared secret with A: g^ad
  - When A sends a message, it unknowingly uses the shared secret with attacker to encrypt.
  - The attacker receives this ciphertext, decrypts it, and reencrypts this plaintext using the shared secret with B
  - B receives this value, and decrypts it thinking it came from A
  - Attack also works in reverse (from B to A)
  - Need some kind of authentication mechanism on the public key to prevent this

Sample Problems

- Write the steps of the Diffie-Hellman protocol
- Show how an attacker can man-in-the-middle Diffie-Hellman
- Show how a man-in-the-middle attack would be detected by the client when the server uses a signature (assuming the client has the server's signature verification key)
- TLS only uses signatures on the server's DHE public key, but not on the client's. Why is this sufficient to prevent a MITM attack?

RSA Encryption

- Public key encryption
  - Everyone knows the public key and can use it to do encryption. Only the key holder knows the private key, and therefore only the key holder can do decryption.
  - Mental model: I give an open padlock to anyone that wants one, but only I know the combination. They write a private message to me, put in a box, and lock it with the padlock. Anyone can create a locked box, but only I can unlock the padlock to receiver the message.

- RSA encryption based on the hardness of factoring the product of two large prime numbers, i.e., given n=pq (for large primes p,q), find p, or q.

- Key Generation

  - RSA public encryption key: (n, e)
  - RSA private decryption key: (n, d) where ed = 1 mod (p-1)*(q-1)

- Encryption:

  - Input: Public key (n,e), message m (where 1 < m < n)
  - c = m^e mod n

    - Output c

- Decryption:

    - Input: Private key (n,d), ciphertext c
    - m = c^d = (m^e)^d = m^ed = m^1 = m mod n

- Why it works:

    - Recall by Euler's theorem a^phi = 1 mod n, for any 1 < a < n
    - As a consequence, a^e mod n is congruent to a^(e mod phi) mod n, i.e., the produce the same result
    - Recall decryption is c^d
        - which equals (m^e)^d
        - which equals m^(ed)
        - which, recall is congruent to m^(ed mod phi) mod n
        - recall e*d was specially chosen at key generation time to equal 1 mod phi
        - therefore m^(ed mod phi) mod n = m^1 mod n = m

- Textbook RSA is multiplicatively homomorphic: the product of two ciphertexts equals the encryption of the product of the corresponding plaintexts:

    - c1 * c2 = m1^e * m2^e = (m1*m2)^e
    - Clearly not IND-CCA2 secure

- Commonly used in conjunction with a padding scheme such as OAEP to make it IND-CCA secure.

Signatures

Used to link an identity to a message. Consists of two keys: a *signing* key and a *verification* key. The signing key is private: only the key holder should be able to sign messages associated with their key pair. The verification key is public: anyone should be able to verify that a signature is valid relative to a party's verification key.

- Functionality:

    - Signing accepts a message and a signing key and outputs a signature. Verifying accepts a message, a signature and a verification and outputs *success* if the signature is valid relative to verification key and message, and outputs *fail* otherwise.

- Properties:
  - It should be hard to create a valid signature on a message without the signing key. In simple terms: it should be hard to forge a signature
  - Types of forgeries
    - Universal forgery
      - Attacker can produce a valid signature on any arbitrary message

    - Selective forgery
      - Attacker can produce a valid signature on a particular message that was chosen head of time

    - Existential forgery
      - Attacker is able to produce a valid signature on *some* message, but they do not necessary have control over what the message is, and the message may necessarily make sense

- Use of hash functions
  - Signatures are usually performed on the hash of a message, not the message itself
  - Done for efficiency

Unpadded RSA Signatures

- Similar to RSA encryption, but "backwards"

- Key Generation
  - RSA public verification key: (n, e)
  - RSA private signing key: (n, d) (where ed = 1 mod (p-1)*(q-1))

- Signing
  - Input: signing key (d,n) and message m (where 1 < m < n)
  - s = m^d mod n
  - Outputs s

- Verification
  - Input: verification key (e,n), signature s, and message m
  - m' = s^e mod n

    - Output m' == m

- Vulnerable to existential forgeries

    - Given a valid signature pair (m,s), an attacker can create a second valid signature pair without knowing the private signing key:
        - Attacker computes (m^2 mod n, s^2 mod n)
        - Valid because: (s^2)^e = ((m^d)^2)^e) = m^(2ed) mod n = m^2

    - Can create a valid signature pair without knowing anything except public key (n,e):
        - Attacker picks an arbitrary number 1<x<n and outputs (m = x^e, s=x)
        - Valid because: s^e = (x)^e = x^e = m

Padded RSA (PKCS 1.5)

Padding prevents existential forgeries by making the signature non-malleable, i.e., by preventing linear operations on ciphertexts from having linear affects on the plaintext.

- Key generation: as before
- Signing
    - Input: signing key (d,n) and message m (where 1 < m < n)
    - h = hash(m) *// hash message*
    - p = pad(h) *// add padding*
    - s = p^d mod n *// sign padding*
    - Outputs s

- Verification

    - Input: verification key (e,n), signature s, and message m'
    - h' = hash(m') *// hash message*
    - p' = pad(h') *// add padding*
    - p = s^e mod n
    - Return p' == p

- PKCS 1.5 Padding function

    - Input: hexidecimal encoded message hash h
    - Output: p = 0001FFFF ... FFFF00 || h
    - Add as many FF's as possible so that p and n have the same length in bytes
    - TLS also adds an ASN.1 header A to identify the padding and hash function
        - i.e., p = 0001FFFF ... FFFF00 || A || h

- Why it works
  - Suppose we have a valid signature s=(001FFFF ... FF00 || A || h)^d mod n
  - If we try to produce an existential forgery as before, e.g., s^2 mod n, then essentially we have ((001FFFF ... FF00 || A || h)^2)^d mod n
  - But with high probability (001FFFF ... FF00 || A || h)^2 is going to look like a random number, i.e., is not going to have this "nice" expected structure (001FFFF ... FF00 || A || h') for some other message hash h'
- Signatures schemes discussed
  - Digital signature algorithm (DSA) (cyclic group implemented in a finite field like DHE)
  - Elliptic curve digital signature algorithm (ECDSA) (cyclic group implemented in over an elliptic curve like ECDHE)

Sample Problems

Suppose you had two valid PKCSv1.5 message/signature pairs (m1,s1) and (m2,s2). Explain how padding prevents existential forgery attacks. Hint: What would happen if you multiplied pad(hash(m1)) * pad(hash(m2)) mod n? Would the result also have valid padding?

Elliptic Curve Cryptography

An alternative means of implementing a prime order cyclic groups. Elliptic curve crypto (ECC) can be used as a drop-in replacement for cryptosystems based on the hardness of solving the discrete logarithm problem.

- Elements of group are points on elliptic curve instead of integers
  - Analog of modular exponentiation, e.g., h=g^a mod p, is point multiplication, e.g., H = a*P, where H and P are points on the curve
- ECDHE is the elliptic curve version of DHE
- ECDSA is the elliptic curve version of DSA
- Two flavors:
  - Elliptic curves over GF(2^m)

    - Fast in hardware

  - Elliptic curves over GF(p)
    - Fast in software

- ECC Pros:

  - Point multiplication is faster than the analog modular exponentiation
  - Public-keys in ECC setting are smaller than their integer counterparts

- ECC Cons:

  - More complex to implement, harder to understand
  - Concern about potential for backdoors in some common curve parameters

Practical Applications

Key Lengths

NIST requires a minimum security level of 112-bits, meaning an attacker must have to do at least 2^112 operations to break a particular primitive. The implications for various primitives:

- Symmetric key:
  - Block cipher / MAC keys must be >= 112 bits

- Hash functions:
  - Output length >= 112 bits for pre-image and second pre-image resistance
  - Output length >= 224 bits for collision resistance

- DHE and DSA
  - Prime modulus p >= 2048 bits
  - Group order q >= 224 bits

- ECDHE and ECDSA
  - Prime modulus p >= 224 bits
  - Group order q >= 224 bits

- RSA (encryption / signatures)

    - n >= 2048 bits, i.e., p, q >= 1024 bits

- Some real-world speeds:

    - AES / SHA hashes can be done on a standard GPU at a rate of about 2^30 per second
    - MD5 collisions can be found in a few minutes on a laptop (remember, it's broken!)
    - 512-bit RSA and DHE, and DSA can be cracked on Amazon EC2 for about $100
    - The largest known factorization was 768 bits, and 1024 bits may one day soon be within reach of the NSA.

Sample Problems

- How many bits of collision resistance does MD5 have?

    - A: Less than the expected 2^64

- If the Bitcoin network can do 2^64 SHA-1 hashes per second, how long would it take for a network of equivalent computing power to find a SHA-1 collision?

    - A: About 18 hours (i.e., 2^16 seconds)

Certificates and PKI

A document used to authenticate a signature verification key. Used to prevent man-in-the-middle attacks. Includes:

- Certificate structure:
    - Serial number
    - Subject's identity and common name
    - Subject's signature verification key
        - If RSA: public verification key (n,e), padding scheme and hash function
        - If DSA: Domain parameters (g,p), public verification key g^x and hash function
        - If ECDSA: Which NIST curve used (e.g., SECP-256r1), public key xG, hash function

    - Issuer's identity and common name
    - Validity period
    - Basic constraints, such as what the keys in the certificate may be used for (e.g., signatures, key exchange, etc) and if the subject is a certificate authority (i.e.,

    may issue certificates)
  - Subject Alternative Name (SAN)
  - Link to issuer's certificate revocation list
  - Signature of everyone above using issuer's private signing key

- Certificate revocation list
  - A signed list of certificates maintained by the certificate authority that have been revoked (i.e., deemed invalid) prior to the expiration.
  - Common reasons:
    - Needing to update an aspect of the certificate, e.g., changing key lengths or signature algorithms
    - When an entity shuts down
    - When a server's private key is compromised

  - Before client checks the validity of a server certificate, it mush first check that it not on a CRL
  - Online Certificate Status Protocol (OSCP) stapling
    - instead of a client making a separate connection to the CAthe server can contact the CA at fixed intervals (e.g., every hour)
    - receives a signed statement saying "I'm the CA and this subject is not currently on my CRL as of one hour ago"
    - 'staples', i.e., includes this signed statement to the TLS handshake

Sample Problems

- Suppose a powerful state-level actor coerced a root CA into revealing its private signing key. How can they use this to man-in-the-middle people in their country? Is it possible for the client to detect this attack?

  - A: The browser implicitly trusts any certificate chain that leads back to a root certificate in its trust store. If an adversary has the root CAs signing key, it can perform a man-in-the-middle attack and replace the server's public key with its own, and then sign it using a fake key for the server. It can then force the browser to accept this fake key by building a fake certificate chain leading back to the root CA. Since the root certificate is the the client's trust store, the client will trust it.
  - A: It maybe possible for the client to detect that something's wrong. For example, if the attacker replaced Google's certificate chain (which is rooted by Geotrust Global CA), and was using a different root certificate, e.g., Hongkong Post Root CA1, the client could in principle could check the certificate chain and see the root CA was different.

- What's the first thing you should do with your certificate if you discover your server was hacked
    - A: Ask your CA to place it on its CRL
- Collision resistant hash functions are extremely important to certificates. Why?
    - A: if an attacker could break collision resistance, then it could produce two certificates that hashed to the same value. It could make the certificate hash to different values by e.g., specifying different server signing keys. One certificate would be a for a legitimate website controlled by the hacker. The other would be for a victim website (e.g., google.com). The attacker would make a certificate signing request for its legitimate website. The server signs the hash of the certificate for the legitimate site. But since the hash of the good certificate equals the hash of the evil certificate, the attacker can take the CAs signature and stick it at the end of the evil certificate. It can then man-in-the-middle google.com, since it presents the client with an (evil) certificate for google.com that contains a valid signature.

SSL/TLS

- Client Hello
    - Random nonce and list of supported cipher suites
- Server Hello
    - Random nonce and list of chosen cipher suite
- Certificate chain
    - A chain of certificates, typically consisting of:
        - Certificate of server's identity and public signature verification key
            - Signed by intermediate CA
        - Certificate of intermediate CA's identity and public signature verification key
            - Signed by root CA
        - Root CA's identity and public signature verification key
            - Self signed
- Server key exchange message
    - Key exchange parameters (e.g., p, g)
    - Server public key (e.g., g^x)

    - Signature on this items and the client/server randoms

- Client key exchange message
    - Client's public key (e.g., g^y)

- Client finished message
    - Client derives pre-master secret (e.g., g^xy)
    - Uses PRF to derive master secret
    - Sends a client-finished message which is essentially a MAC of all the handshake messages it saw up to this point
    - Uses master secret in a PRF to derive
        - Client send encryption key
        - Client send MAC key
        - Server send encryption key
        - Server send MAC key

- Server finished message

    - Server derives pre-master secret (e.g., g^xy)
    - Uses PRF to derive master secret
    - Checks MAC on client finished message
    - If valid, server creates a MAC on all handshake messages it has seen up to this point (incl. client finished message) and sends to client
    - Uses PRF and master secret to derive symmetric keys (see above)

- Client and server can now communicate using block cipher and MAC (or alternatively authenticated encryption)

- TLS cipher suite specifies

    - Key exchange algorithm (DHE, ECDHE, or RSA)
    - Signature algorithm (DSA, ECDSA, or RSA)
    - Cipher and mode of operation (e.g., AES-CBC, AES-GCM)
    - Hash function (e.g., SHA-256)
        - Used to define pseudo-random function (PRF)
        - If using a non authenticated encryption scheme (e.g., something other than AES-GCM), hash function is also used as an HMAC

Sample Problems

- The server typically doesn't send the root CAs certificate. Why not?

    - A: It doesn't need to. The client already has the root CAs cert

Passwords

- Entropy
    - How many bits of information it takes to encode a particular password
    - Considers not just how many possibilities there are, but also what the probability is of each password occurring
    - Low entropy passwords are more easily guessable, and high entropy passwords are harder to guess
    - Computing entropy
        - Uniform distribution (i.e., all possible passwords occur with equal probability)
            - If s represents the set of possible passwords entropy is log_2(1/s)
            - E.g., a 6-character password chosen at random from the base-64 characters set (consisting of 2^6 characters), the set of possible passwords would be (2^6)^6=2^36, and therefore the password has 36 bits of entropy

- Storing passwords
    - We need to store passwords on a web server in order to check them when a user tries to log in
    - What happens if this server was ever compromised and this password database stolen? It's a risk, so we need to protect our passwords
    - Password hashing. Instead of storing passwords in the clear, we can hash them and store the hash instead. When a user tries to log in, you hash their password and check it against the stored hash in the password database
        - Pro: Attacker can't get passwords outright. It has to make a guess, hash it, and check it
        - Con: Guessing is sped up because the same password maps to the same hash, so if 100 people have the same hash, the attacker only has to guess one password to recover it for 100 people.
        - Con: Attack can pre-build large dictionaries or rainbow tables (a dictionary with a time-memory tradeoff)

    - Salt. Instead of just hashing the password, generate a random value called a salt and hash the password and the salt. Store the password and salt together
        - Pro: People with the same password will still very likely have different salts, and therefore different password hashes, limiting the use of dictionaries

        - Con: Guessing is still efficient (i.e., one hash)

    - Key stretching. Instead of storing a salted hash of the password, for which guesses can be efficiently checked, design a special kind of "slow" hash function that takes a long time to compute, making guessing harder
        - Iteratively hash
            - Begin by hashing the password and salt. Then hash this result. Then has this result. And so on.
            - Recall our 6-character random base-64 password required an expected 2^36 guesses. If we iteratively hashed 2^10 times, then checking a single guess would require 2^10 hashes, and thus recovering the password would require an expected 2^36*2^10 = 2^46 hashes.

- Example iterated password hashing algorithms

    - PBKDF2 (salted, iterated applications of a hash function, eg SHA1)
    - bcrypt (similar to PBKDF2 but uses a particular hash function)
    - scrypt (which also makes the attacker have to use a lot of memory)

Sample Problems

- An attacker steals a password database from a website using user-chosen passwords. Suppose the database contains 2^20 (about a million) passwords, but 2^10 people (about 1000) people were using the most common password "123456". The passwords are salted and hashed using SHA-256. The attacker knows "123456" is the most common password, but doesn't know who's using it. How does he check? How many SHA-256 operations would it take him to find all 1000 accounts using this password?

    - A: For a given password hash and salt (h,s), the attacker checks if SHA-256("123456"||s) = h.
    - A: There are 2^20 password hashes, and checking each hash requires one execution of SHA-256, therefore the attacker must perform at most 2^20 hashes to recover all accounts with "123456" as their password.

- An attacker owns a GPU cluster with 2^6 GPUs that can perform 2^30 SHA-1 hashes per second each. The attacker stole a password database from a victim website which was using 8-character random system assigned passwords from the base-64 alphabet hashed with PBKDF2 with 1000 iterations of SHA-1. How long would it take the adversary to recover a single password?

- A: 2^22 seconds.