MORNING:

Motto: ‘Don’t roll your own’ – use crypto functions built by the pros, yours will be bad

Cryptographic Protocols:

WEP – Attempting to make WIFI communication secure

* Most flawed protocol ever – tried to write their own, instead of a cryptographer
  + Could listen passively to packets and find the key to the WIFI network
* Once it was discovered that it was flawed, people kept using it and could be hacked
* WEP: bits are sent in sequence. Laptops need to know which bits are for them. Packets are given identifiers at the start, and the laptops with that identifier can get onto that WEP line
  + How to encrypt: the sender and receiver generate a random bit sequence, which is placed below the bit stream and XOR’d to give the cipher bit string, and then deciphered with the same random bit sequence at the other end.
    - This is a good idea in principle, they just implemented it shitly
  + XOR with 1 random string will generate a cipher with good randomness properties, so no frequency analysis and such can crack it
* MAIN FLAW: encrypt the entire TCP/IP packet with XOR; at the start of the packet is the header, which contains source and destination IP addresses (who sent and who is receiving).
  + We know the location of each chunk in the header, so we know which bits (though they are encrypted) are where. We can listen to a packet and resend it as a duplicate, which the protocol doesn’t manage well.
  + We intercept a message, do magic to change the destination address to point to us, and send it back to the access point who decrypts it and passes it on to us. Since it is only decrypted on the channel between sender and receiver, once received, it’s decrypted before sent off to its final destination (different layers on network, only 1 is encrypted).
  + Brute force the 256 location addresses and the 1 is ours and the 255 others are dropped by receivers who weren’t expecting the packet
  + **Don’t mix data and control – but this is impossible, they always get mixed**

HASHING for integrity last week

HASHING for authentication this week –

* Breaking a hash: we can find a preimage/undesirable result of the hash faster than brute forcing, even if it’s only 1 bit faster. I.e. you can do a preimage of a 256-bit hash in 255 bits.
  + Although this might not have practical implications, it’s an indication of future security compromises, “building on sand”.
  + MD5 was broken, and no-one was building a new hash function, and md5 was getting more and more broken
  + Lead to NSA inventing SHA, and then SHA1 and then SHA2, and people don’t really trust them. Recently from a competition we got SHA3.
  + MD5 still used; last year a bunch of passwords got compromised since they were hashed with MD5
  + MD5, SHA0 and SHA1 have been broken. SHA2 not yet but SHA3 is better so it is the Industry standard.
* MD5 -> SHA2 incl. all use ***Merkle–Damgård construction*,** which uses a 1-way compression function (take 2 fixed length inputs to create 1 fixed length output) to produce a collision resistant cryptographic hash function; How:
  + use MD-compliant padding -> to stretch the input into an acceptable size (since compression functions cannot handle arbitrarily sized input)
    - MD-compliant padding conditions:
  + Chunk the input -> sequentially compress and hash the blocks. E.x. compress & hash block 1 to get first output. Add block 2 to the result so far, & compress & hash the combined. Repeat until the final chunk. Then 0’s and bits to represent the message length are appended to get correct-size result.
* MAC: *refer to last week’s notes*
* **Length extension attack:** hash(key|message) -> works for M-D Construction
  + Hashes split into chunks and then hash the chunks. Mix the first chunk with the IV and hash. Then add the next chunk and hash.
  + To attack, you can add another chunk, hash it (hash function is public) and XOR it with the previously hashed chunk and you’ll get a valid hash and message combo
* HMAC = hash(key2 | hash(key1 | message)) -> prevents length extension attacks
  + The inner function can be length attacked in that you can append to the message and generate a valid MAC for it. But then that output is of fixed length, since it was hashed, so the input into the out function is always the same length. So without knowing the keys, you can’t extend the message since you don’t have control over intermediate hash(key1|message) output

Passwords:

**Password Attacking:**

* /etc/shadow – contains username :: password hash :: password salt
* <login> -> looks for your name in etc/shadow -> takes the salt -> prepends to your password -> hashes it -> compares to stored hash to authenticate
* The best way to hack passwords is to gain access to someone’s computer, access the password/shadow password directory, write the password hashes to somewhere else and take it to your local machine to crack later.

**Rainbow tables:** <https://security.stackexchange.com/questions/379/what-are-rainbow-tables-and-how-are-they-used>

**Password salting:**

* Background: I don’t want anyone to know my password, but I want you to be able to authenticate my login. To do this, when I type my password, it is hashed and then compared to the hash that is stored for it in the database. This way, no one can know my password (assuming Preimage resistance), but I can still be authenticated
* Can be brute forced by checking the hash of everything e.x. if you know the hash is a dictionary word, hash the entire dictionary
* SALT: like food, to make you password richer and more random. You get a different salt value (e.x. 7) for every person and put it at the start of the password, and then hash. The database stores your hash, and the plaintext salt. Then when you type the password, the database is queried for the salt, the whole thing is hashed, and compared to the hash in the database.
* With enough salts, a dictionary attack can only ever be used against 1 person, since the dictionary for each person is different. They can only attack 1 password, not a suite of them since they are from different dictionaries.

**Key Stretching:** longer to compute 1 password, another way of slowing down cracking.

* Key stretching adds 2 a ***sleep (for 2 seconds) function*** to this, so big woop for a successful use case, you wait 2 seconds
* But for a million-hash brute forcing, it adds 2mill seconds = 3 weeks hash time to brute force a million hashes.