Cryptography Week 2

Introduction to Offensive Security

Misc.

- This is the last week of homework
- Master challenge
  - Will open after next class
  - Extra credit (replaces 1 homework)
  - Multi-stage challenge incorporating all 4 CTF categories we've talked about

Misc.

- Options for next week
  - Q&A "lecture"
    - What do you want to learn more about?
    - Impromptu 45-ish minute lectures on whatever
  - SECCON this weekend
    - 24 hour − starts at 1a Saturday
    - Come to the lab, play, we'll be casting to the TV as we work through harder chals
    - If you haven't done a write-up, you can group up with others who haven't either and stake out a challenge
    - Not required

- Opinions?

Misc.

- https://superuser.com/questions/1268868/can-i-save-these-documents-on-a-dying-machine-from-oblivion

HW Overview

- Quick demo

RSA Attacks (Cont'd)

- Low e – see last lecture
- Common modulus
  - Same message encrypted with the same n, but with different e
  - We'll discuss this in detail
- Wiener's attack
- Coppersmith's
- Franklin-Reiter related message
- And more!

RSA Common Modulus Attack

- Same m, n different e
- Public exponents have to be relatively prime
- To solve:
  - Compute a,b such that $e_1 a * e_2 b = 1$ (extended Euclidean algorithm)
  - Then $c_1^a * c_2^b = m \pmod n$

- Example on board

Hashes

- Given arbitrary input bytes, generate a constant length "hash"
  - Ideally the hash changes a lot for a small input change
- Ex.
  - md5("Hello") -> 8b1a9953c4611296a827abf8c47804d7
  - md5("Hellp") -> 62ac2dcdae264b4aac4e9b2631692514
- Variety of algorithms
  - MD5
  - SHA1,2,3
  - BLAKE2

Cryptographic Hash Properties

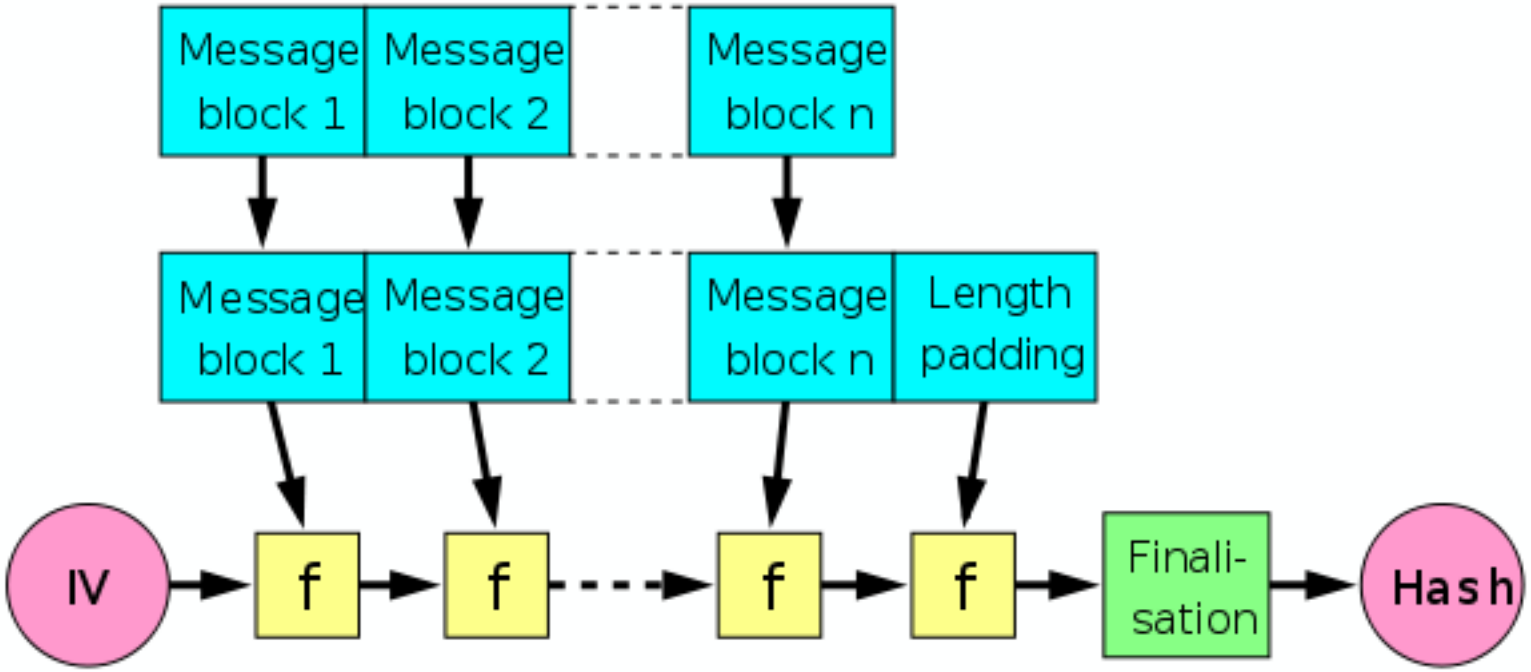
- Want three features:
  - Preimage reistance: given H(x), it's hard to find x
  - Second-preimage resistance: given x1, H(x1), it's hard to find x2 where H(x1) = H(x2)
  - Collision resistance: it's hard to find *any* pair x1, x2 where H(x1) = H(x2)
- Note 1: these properties are listed in order of how hard they are to achieve (easiest to hardest)
- Note 2: collisions are *guaranteed* to exist (why?)
  - Goal of a cryptographic hash function is to make them hard to find

MD5

- 32 byte hash output
- Merkle–Damgård construction (more on this in a minute)

- (Effectively) completely broken
  - Collisions (2 input strings with the same hash) are easy to find
  - No practical preimage attack though
    - Given hash, efficiently find input that hashes to the given value

Merkle–Damgård Hash Functions



- MD5 and SHA1 are both constructed this way

Merkle–Damgård Hash Functions

- The input is chunked into blocks of a constant size, padding if needed
  - MD5 uses 512 bit blocks and NULL padding
- These are then "compressed" with the `f` function
- And the results are mixed together
  - Could be XOR, could be bitwise masking, etc.

Merkle–Damgård Padding

- MD based algorithms have to work on blocks
- These blocks are the full internal state of the algorithm
- The padding is predictable
  - Has to be otherwise hashing the same input twice would produce a different result!

- Imagine a construction like this:
  - `md5(SECRET + "username=asdf&is_admin=0")`
  - Seemingly could be used as a MAC

Hash Length Extension

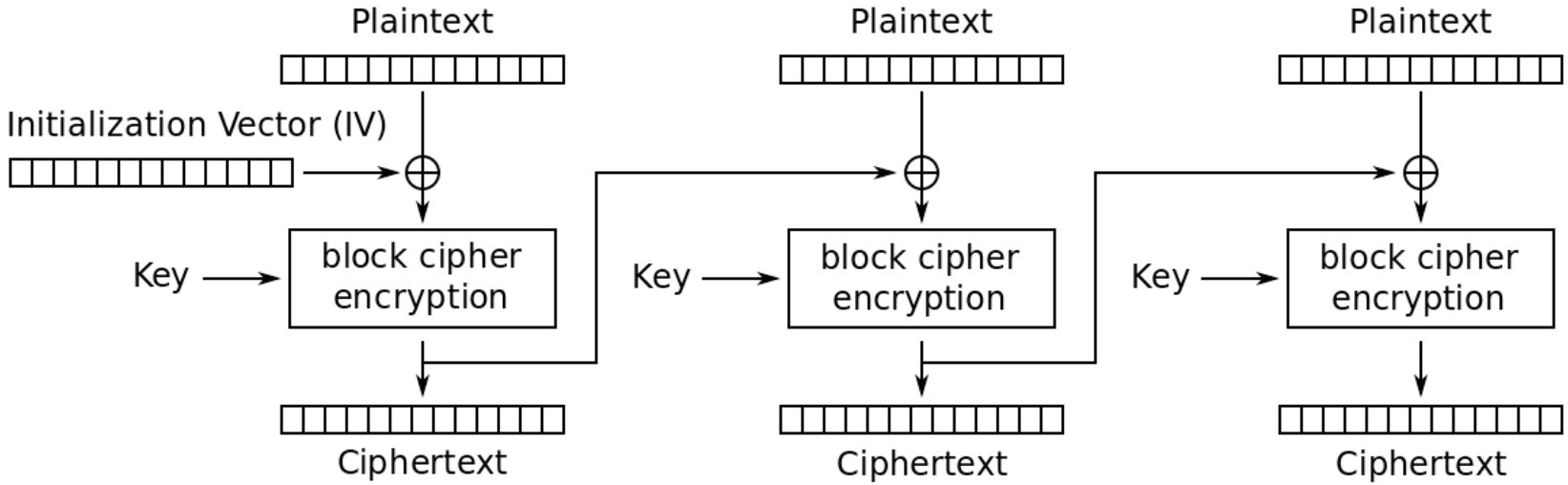
- `md5(SECRET + "username=asdf&is_admin=0")`
- The result of the hash fully determines the internal state of the algorithm
  - -> we can fully recover the internal state from the hash
- Since we know the full internal state, we can "reset" to that point and add on whatever data we want
  - i.e. we can easily compute:
  - `md5(SECRET + "username=asdf&is_admin=0" + "\x80\x00\x00…" + "arbitrary_data")`
  - without knowing SECRET!
  - Relies on SECRET being *prepended* to the input though

Hash Length Extension Tooling

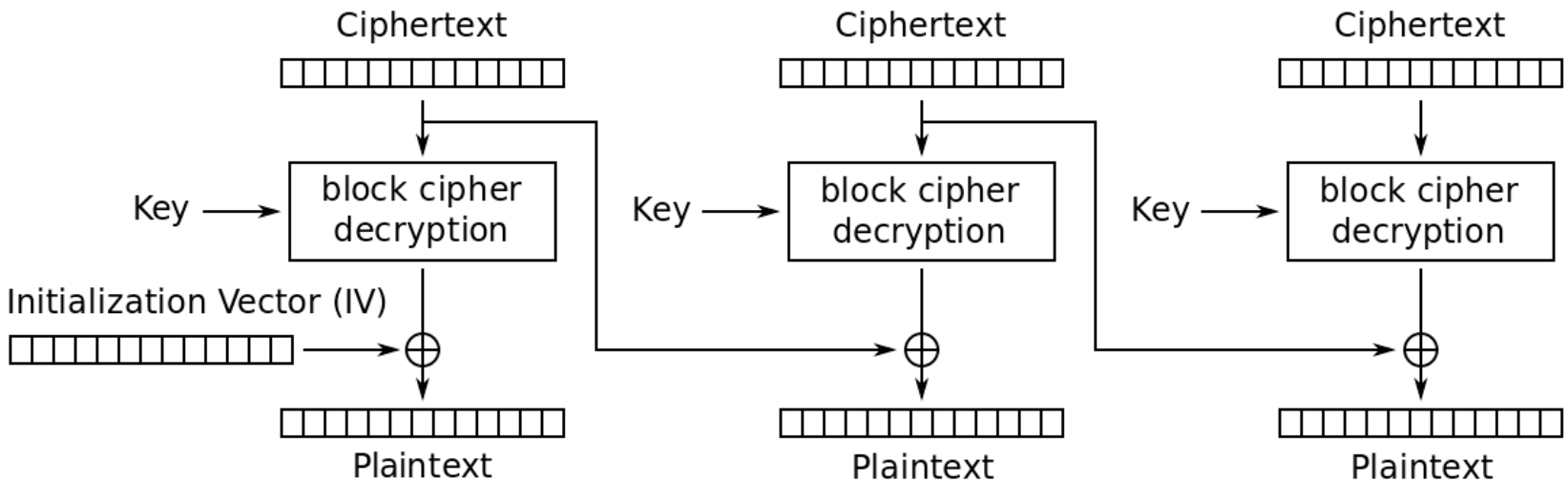
- https://github.com/iagox86/hash_extender

CBC Padding Oracles

- Last time we saw that to use a block cipher on arbitrary length data, you need to split it into blocks
- Then use a construction like CBC to encrypt



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Padding

- Most messages are not exact multiples of the block length
- So we need to *pad* the last block out with extra data to make a full block
- This data can be discarded after decryption

PKCS#7

- There many ways messages could be padded, but one of the most popular is called PKCS#7
  - It's a standard: Public Key Crypto Standard #7
- The padding character gives the number of bytes of padding used
- Example: block length 4, message = "hello"
  - Padded: "hello\x03\x03\x03"
- Padding is *always* used – if the message is an exact multiple, you add an entire block of padding
- Example: block length 4, message = "blah"
  - Padded: "blah\x04\x04\x04\x04"

Validating Padding

- When decrypting a message, we can check if the padding is correct
  - Strip off the padding and check that all pad bytes are equal to the pad length
- What should we do if we discover bad padding?
  - Probably means something went wrong decrypting, or the message was corrupted somehow
  - Maybe we should present an error message to the user?

CBC Padding Oracle Attack

- It turns out if we do *anything different* when the padding is incorrect vs correct, we can decrypt the message entirely!
- How?
- Basic idea:
  - If we *change* the second-to-last ciphertext block, we can control what the last plaintext block decrypts to
  - We will get *different messages* if that plaintext has correct padding or not
  - When we get the padding correct, we can use this information to figure out a bit about the message!
- Let's work this out on the board…

Repeated Unmasking

- If we know flag format and where it appears, we can discover part of the key
- If that key happens to also be at a known place in the plaintext, we can recover another part of the key
- Repeat until everything is decrypted

- Walkthrough
  - https://github.com/isislab/CSAW-CTF-2017-Quals/tree/master/crypto/another_xor