Mason Competitive Cyber

Hash Functions and Hash Attacks

This week in Cyber...

- No news of major attacks on the election!
  - We have a new secretary of defense!
- Checking Out?: Exposed S3 bucket leaks 10m+ records from hotel booking sites
  - https://threatpost.com/millions-hotel-guests-worldwide-data-leak/161044/
- Having a bad time: Voter fraud investigation website accidentally leaks voter data
  - https://threatpost.com/trump-site-alleging-az-election-fraud-exposes-voter-data/161068/
  - Names, Addresses, Last 4 of SSN

Upcoming Competitions

- MasonCC Members participating in HackTheBuilding as part of HTCPCP://
  - Team is full!
  - There’s an external discord for this, make sure you join it as well as the discord for the competition
- C2C CTF: December 6th @ 0000Z (timezones are weird)
  - If you signed up, check your email! A lot of MasonCC guys were selected as team leads (cuz we the best), so if that’s you make sure to reach out to your team members!
  - #c2c_2020

On to the actual content!

Agenda

- Hash Functions 101
  - Why should I care about hash functions? (Hint: not bitcoin)
  - Bonus Material: *Entropy isn't just for physicists, but it is just for nerds*
- Hash Function Identification
- CTF problems involving hashes
  - Brute Force
  - Iterative Hashing
- Advanced Hash Attacks (i.e. real world attacks which are more complicated than brute force)
  - Fast MD5 Collisions
  - Merkle-Damgard Length Extension Attacks

Hash Functions 101

- A Hash Function, represented as H(*x*) = *y*
  - *x* is the “message” or “plaintext”
  - *y* is the “digest” or “hash”
  - A hash is typically* a one-way function
- Given only *y*, it should be very difficult to get back to *x*
- If you want to get pedantic, there’s really two types of hash functions
  - Non-Cryptographic Digests
    - Checksums, Java hashes, etc
    - Finding an *x* for a given *y* is inconvenient and probably means you’re doing something wrong
  - Cryptographic Digests
    - MD5, SHA-family, etc
    - Finding an *x* for a given *y* should be impossible*

Why do I care?

- ~~Those sweet, sweet BTC gainz~~
- It is often convenient to have a **one-way function**
  - For instance, a password
- Often convenient to “digest” a large piece of data into a small piece of data that characterizes it
  - For instance, a checksum on a downloaded file
- Often convenient to have an indicator of change (similar to that ^ example)
  - For instance, a cryptographic signature, which is a can of worms for a different talk

The 3-ish commandments of a good cryptographic hash function

1. **Thou shalt not be reversible**
    a. AKA *preimage resistance*
    b. Given a hash, you shouldn't be able to find the message that produced the hash easily
2. **Thou shalt not have collisions**
    a. AKA *second preimage resistance* (or *weak collision resistance* to friends and lovers)
    b. **Given a thing,** it should be hard to find another thing which has the same hash value
3. **Thou shalt NOT HAVE COLLISIONS.**
    a. AKA *collision resistance* (or *strong collision resistance*)
    b. It should be hard to find **any** two things that share the same hash value

Heuristics that indicate a good hash function

- A small change in the input results in a large change in the output
    - *The Avalanche Effect* in literature, but I prefer the *Skittle in a Handful of M&Ms Effect*
    - A change in one input bit should change approximately 50% of output bits
- The output should have relatively high entropy
    - Coorelary: The output's entropy should be independent of the entropy of the input

But why are you talking about entropy? Isn't that physics?

- I promise there will be no mention of thermodynamics or any scary integration...
- Entropy is a concept which can be easily stolen and applied as a useful heuristic for cryptanalysis
- In its most basic form: Shannon Entropy
  - Butchered: "What is the average value of this set of bits?"
  - The closer to 0.5 (i.e. 50% 1s and 50% 0s), the higher the entropy
  - We can use this simple form of entropy to characterize how "random" a sample appears
  - Cleartext and plaintext are generally quite low entropy (4.5-5 bits per byte) whereas ciphertext and compressed data are generally high entropy (>7 bits per byte)
  - Cyberchef has a built in module for calculating entropy for a sample

Identifying Hash Functions

- Either the first step to a longer problem or a low-point crypto question
- The theory: Hash functions produce fixed length output, so you can get an idea of what kind of hash you're looking at just by counting how many hex digits are present
- For instance:
    - MD5: Always 32 hex digits (128 bits)
    - SHA-256: Always 64 hex digits (256 bits)
- Cyberchef has a hash identifier, but honestly after you've gotten some experience you can start to pick them out easier.
- Live demo of that!

Basic CTF Problems: Hash Cracking

- The easiest way to attack a hash (and the way you usually see hashes attacked in practice) is to run a brute force attack on the digest
- Basically, try every word in a given wordlist or based on some ruleset and see if you can generate a matching hash
- The optimal technique depends on three factors
    - Is the hash salted?
    - Do you suspect that the password is short?
    - Do you have a wordlist you think might work?

Hash Cracking

- If the hash is unsalted:
  - You're in luck! Drop it into https://crackstation.net and hope for the best.
  - This technique is so quick to do, it's often the first thing I check when I get a new hash
- If the hash is salted:
  - You're going to need to get that salt. Then, we're going to proceed on to one of the next methods, making sure we keep the salt handy for when we need it

Computing a normal hash: H(x) = y
Computing a salted hash: H(salt+x)=y store the salt along with y

In practice, a salt just tends to be a random string of characters to defeat lookup attacks like crackstation

If you the password is short...

- You can do a brute-force approach that iterates through every possible character
- Hashcat or John are the tools of choice for something like this
- If you have a salt, make sure to include that as input to your tool (there’s probably a command line flag or special format you have to use)

Using a Wordlist

- This is usually my second stop for hash attacks
- If you have a hash that can’t be found using a look-up table, you might want to try building a wordlist or using one of the wordlists included in Kali (located in /usr/share/wordlists)
- Again, use Hashcat or John for this, and remember to use your salt if you recovered one
- You can also use tools like CeWL to scrape webpages to generate wordlists (you see this in HTB sometimes but generally it’s not super realistic)

Iterative Hashing

- Made an appearance at MetaCTF Cybergames
- Generally exclusively confined to CTFs, but a neat scripting challenge
- Basically:
  - Hash the empty string
  - Hash the first character of the flag
  - Hash the first two characters of the flag
  - Hash the first three characters of the flag
  - …
  - Hash the whole flag

Cracking an iterative hash

- Generally, you'll see something that looks like a file full of hashes. Throw the first few into crackstation and see if anything notable falls out
- Then, do the same process in reverse:
  - Brute force one character until you get a matching hash. Store the character which produced the match
  - Brute force the second character, appending it to the first, until you get a match. Store the two characters.
  - Brute force the third character, appending it to the first two, until you get a match. Store the three characters
  - …
  - Brute force the final character, appending it to the end of the leaked flag. You will have produced the flag when you find the last character.

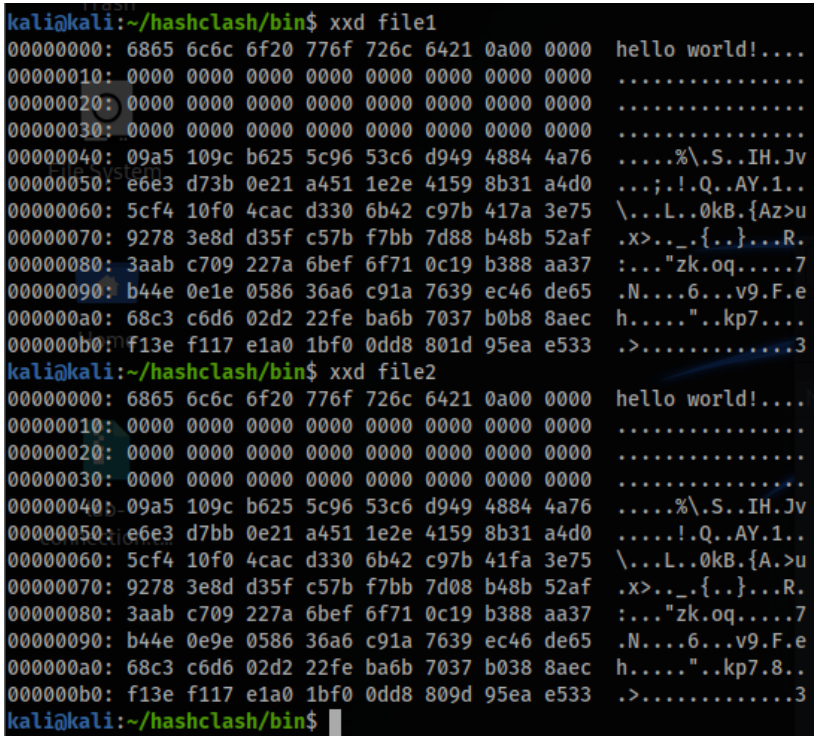
- Demo Time!

Creating an MD5 Collision

- MD5 has been known to be broken for a long time
    - Specifically, it's *weak collision resistance* property has been found to be faulty
- It may be possible, for certain messages, to produce an MD5 collision in a trivial amount of time
- The attack demonstrated here is a "chosen-prefix" collision, meaning that we have two messages with an identical prefix but different suffix which have the same md5 value
- MD5(m||s1)==MD5(m||s2)

Using HashClash

- https://github.com/cr-marcstevens/hashclash
- Demo!
- https://asciinema.org/a/4LUZuNjFD3otNIINIbi1AwuD4



Length Extension Attacks

- Actually pretty straightforward once you understand the internals
  - Remember: Hashes act on blocks of data, so input data needs to be padded in order to be processed
- Let's say we have an application which takes data in the form:

```
user=[username]&user_id=[number]&rand=[random]
```

- Also suppose there's a bug (or a feature?) which means that if a field appears twice, the second appearance's value is used
- So if we had a situation like this:

```
user=Bob&user_id=1000&rand=\xde\xad\xbe\xef&user=Alice
```

- The data processed would see Alice as the user instead of Bob

Message Authentication Codes

- A neat way to prove authenticity using hash functions
- For this recipe, you need:
  - A message that you want to prove came from a trusted party
  - A shared secret
  - A hash function
- Distribute the shared secret to every trusted party
- When you want to send a message, you also send an authentication block which is calculated as

  Hash(Secret||Message)
- You can then send the authentication block along with the message
- When the recipient gets the message, they simply calculate the authentication block using the message and shared secret and verify that the hash matches

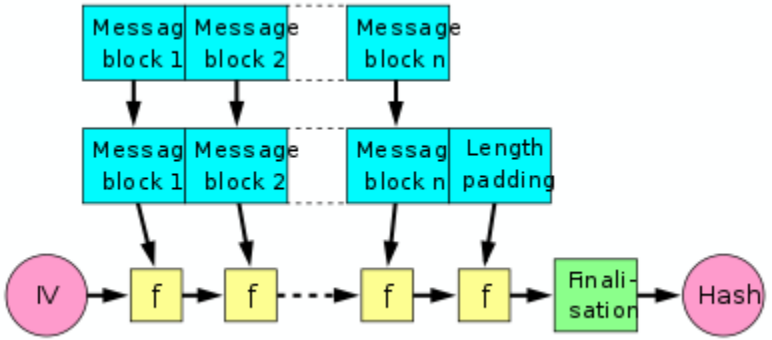
For Example:

- Alice and Bob have a shared secret of “SuperSecretText”
- Alice wants to send the message “Hello” to Bob
- Alice calculates sha256(“SuperSecretTextHello”)=be3a3568ef767c5dfe641f3d3368df0c176e85e9b527abf990689250a9348525
- Alice sends Bob “Hello | be3a3568ef767c5dfe641f3d3368df0c176e85e9b527abf990689250a9348525”
- Bob is able to calculate the same hash from the message and the shared secret, meaning whoever sent the message must have known the shared secret

- Questions?

Back to the attack...

- DO NOT USE SHA-256 AS A MAC
- MACs are good, but SHA & MD5 use a construction called Merkle-Damgard
- This means that one block feeds directly into the next block (the last block may be padded)
- If we can guess the length of the secret, we can compute a valid hash with some appended data

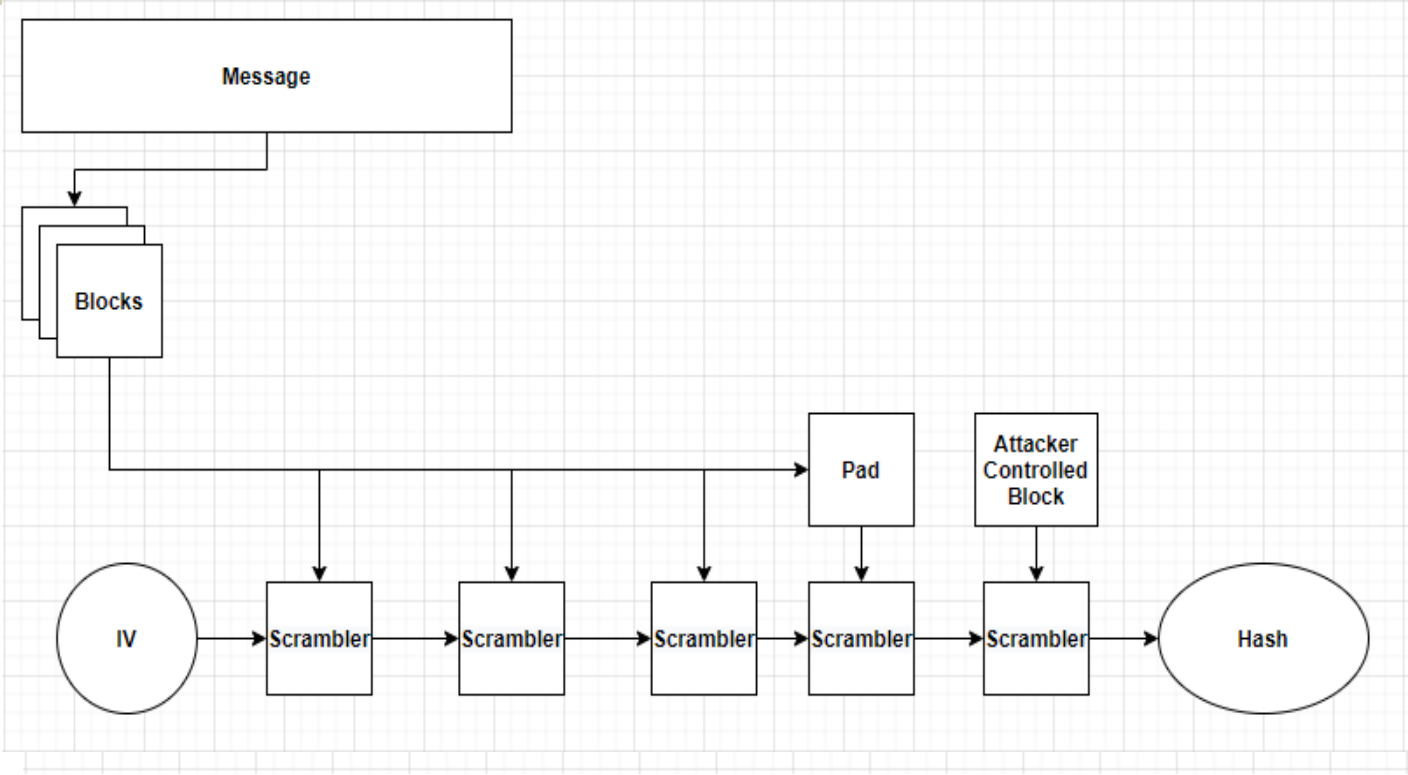


- How?

Generating valid data

- In practice, we can jump right back into the hashing state and insert another block with custom data and calculate a valid hash

- There's tools that can do this for us!



Attacking our example system

- We capture a valid message and signature

  `user=Bob&user_id=1000&random=a6c5328f1b`

  `auth=ba79b7413c33342b23aa06659e99e2bd35670aef533b53fefbc5cee7de93fe7b`
- We know that the signature is computed using SHA256
- We also assume that the “random” field accepting of bad data (in practice this may or may not be true)
- We can then use Ron Bowes’ hash extender to perform a length extension attack
- https://github.com/iagox86/hash_extender
- In practice, we’d have a service to check each of our length guesses against until we got it right, but for this demo I don’t have a service so just for now let’s say we’re given the length of the secret (14 bytes)

Attacking our example system

- https://asciinema.org/a/0xfdayDrevuZ4U0TXLfHAA9qX

- New Signature: 683b046c6ff0a40f50aa9349fd6683a1249fbfd4e77b7b0b25693f586a6c2756
- New Data:
  757365723d426f6226757365725f69643d313030302672616e646f6d3d6136633533332386631628000000000000000001a826757365723d416c696365

- Secret: SuperSecretKey

Proud Sponsors

CACI

EVER VIGILANT

VOLGENAU SCHOOL

MASON

ENGINEERING