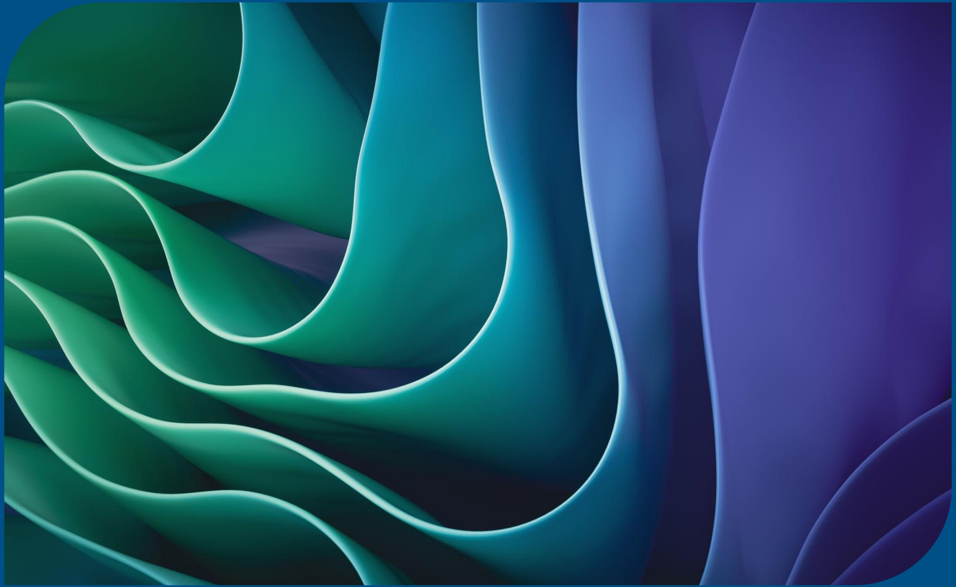
Implementing (and cracking) SHA1

COMP6441, F15A - Friday 3PM



Richard Tong, z5610997

Agenda

1. **What is SHA1**
2. **The High Level Design**
3. **The Code**
4. **Breaking SHA1**
5. **Conclusion**

What is SHA1?

- Securing Hashing Algorithm 1(SHA1)
- A cryptographic hashing algorithm(designed to be the standard hashing algorithm for the US Government) developed in 1995
- Generates a 160 bit message
- Widely used before for a variety of security based applications i.e password hashing, cryptographic signing etc.
- Cracks started showing around 2005 for well funded attackers with theoretical cracks showing on smaller versions of SHA1
- Was considered generally insecure past 2010 due to a near-collision attack which was able to crack SHA1 at the time with a cost around $2.77 Million per hash
- Was curious to see how hard would it be to crack a semi-modern hash (**spoilers: it was pretty tough**)

Agenda

1. What is SHA1
2. The High Level Design
3. The Code
4. Breaking SHA1
5. Conclusion

High Level Design

- The SHA1 Algorithm has a couple of high level steps:
- **Step 1:** Pad and Split the data into 512 bit chunks, Add Padding if not cleanly breakable into 512 bits
- **Step 2:** Preprocess the data into 512-bit chunks
    - *Step 2.a:* Convert chunks into 16 32-bit words with big endian order
    - *Step 2.b:* Extend the 16 words to 80 words
- **Step 3:** Initialize working variables
- **Step 4:** 80 rounds of mixing:
    - *Step 4a:* choose bits from b, c, d
    - *Step 4b:* XOR words together
    - *Step 4d:* again XOR words together
- **Step 5:** Add the mixed words to the working variables to receive the hash

Agenda

1. **What is SHA1**
2. **The High Level Design**
3. **The Code**
4. **Breaking SHA1**
5. **Conclusion**

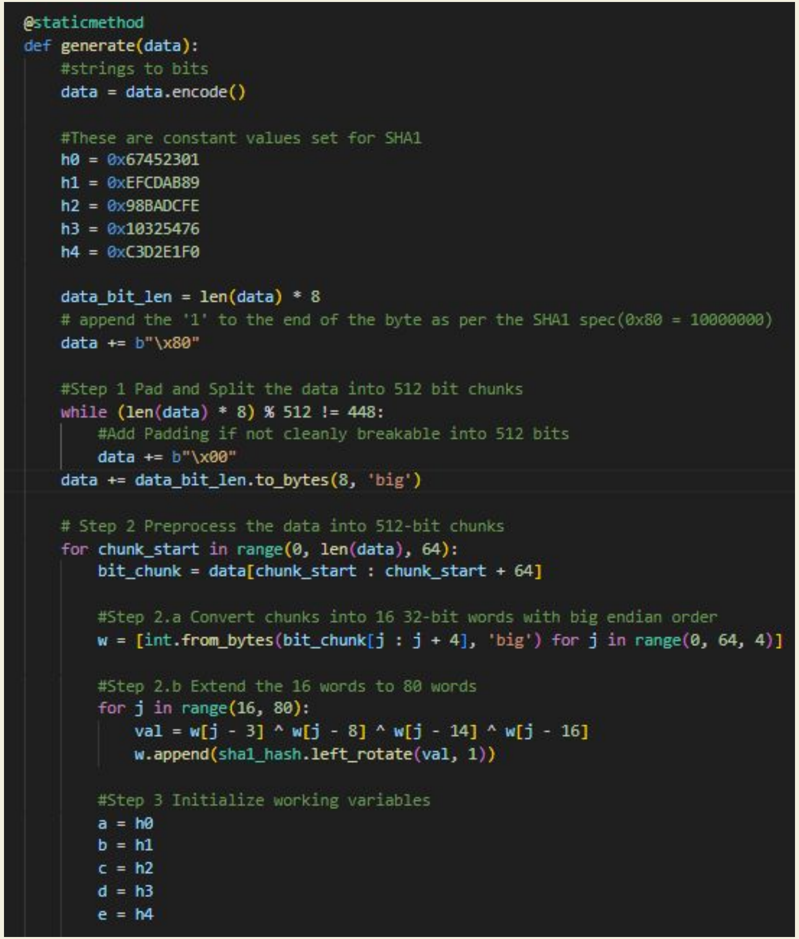
Thc Code

- The code can be broadly broken up into 3 parts:

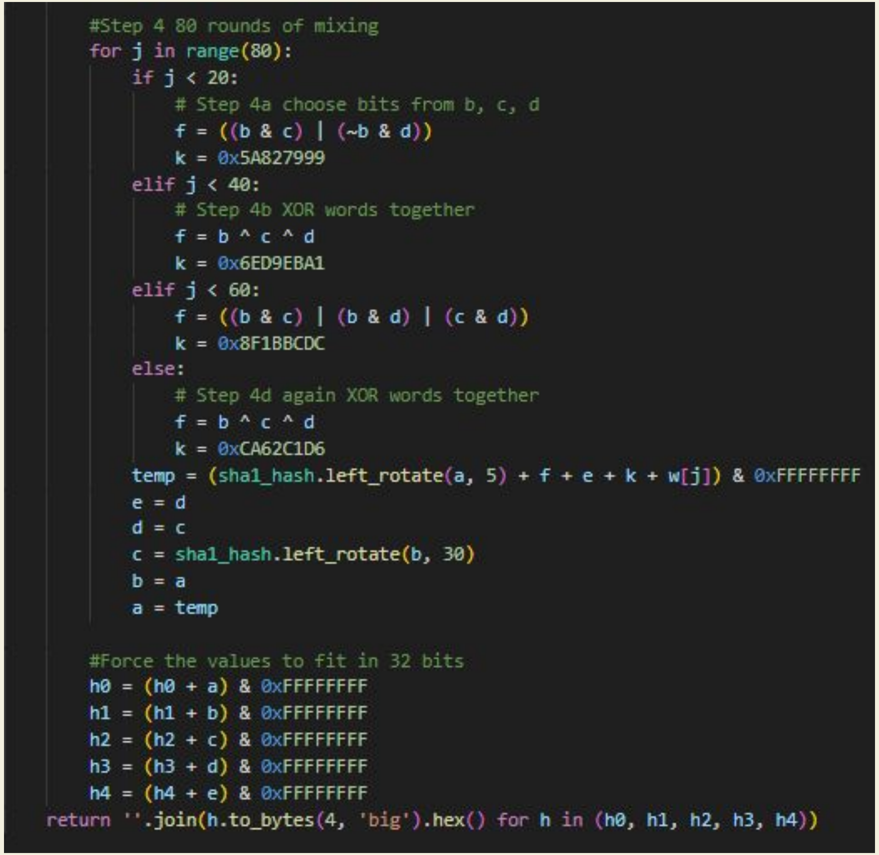
1. The SHA1 Hash
2. The code cracking the hash
3. The caller code

- We’ll explore each section of the code separately

Thc Code – SHA1 Hash

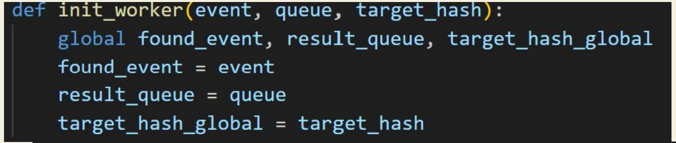


```
@staticmethod
def generate(data):
    #strings to bits
    data = data.encode()

    #These are constant values set for SHA1
    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0

    data_bit_len = len(data) * 8
    # append the '1' to the end of the byte as per the SHA1 spec(0x80 = 10000000)
    data += b"\x80"

    #Step 1 Pad and Split the data into 512 bit chunks
    while (len(data) * 8) % 512 != 448:
        #Add Padding if not cleanly breakable into 512 bits
        data += b"\x00"
    data += data_bit_len.to_bytes(8, 'big')

    # Step 2 Preprocess the data into 512-bit chunks
    for chunk_start in range(0, len(data), 64):
        bit_chunk = data[chunk_start : chunk_start + 64]

        #Step 2.a Convert chunks into 16 32-bit words with big endian order
        w = [int.from_bytes(bit_chunk[j : j + 4], 'big') for j in range(0, 64, 4)]

        #Step 2.b Extend the 16 words to 80 words
        for j in range(16, 80):
            val = w[j - 3] ^ w[j - 8] ^ w[j - 14] ^ w[j - 16]
            w.append(sha1_hash.left_rotate(val, 1))

        #Step 3 Initialize working variables
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4
```

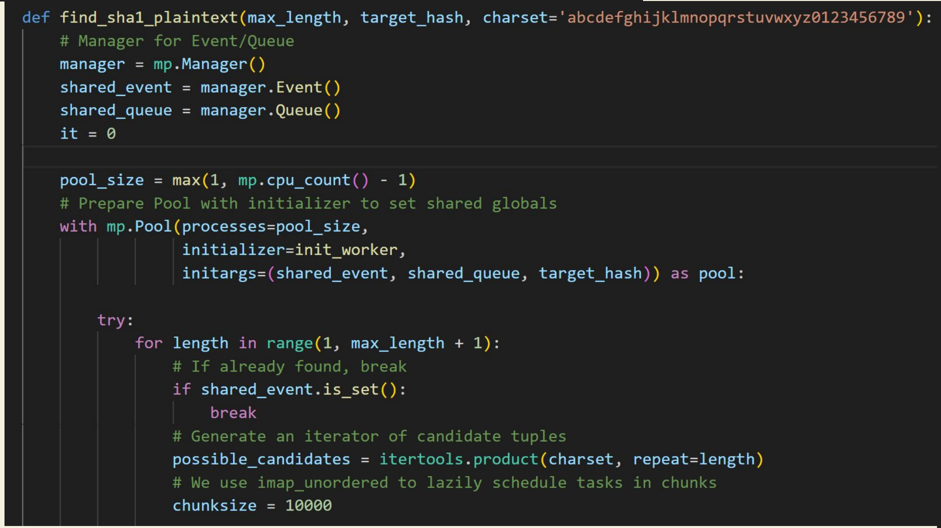


```
        #Step 4 80 rounds of mixing
        for j in range(80):
            if j < 20:
                # Step 4a choose bits from b, c, d
                f = ((b & c) | (~b & d))
                k = 0x5A827999
            elif j < 40:
                # Step 4b XOR words together
                f = b ^ c ^ d
                k = 0x6ED9EBA1
            elif j < 60:
                f = ((b & c) | (b & d) | (c & d))
                k = 0x8F1BBCDC
            else:
                # Step 4d again XOR words together
                f = b ^ c ^ d
                k = 0xCA62C1D6
            temp = (sha1_hash.left_rotate(a, 5) + f + e + k + w[j]) & 0xFFFFFFFF
            e = d
            d = c
            c = sha1_hash.left_rotate(b, 30)
            b = a
            a = temp

        #Force the values to fit in 32 bits
        h0 = (h0 + a) & 0xFFFFFFFF
        h1 = (h1 + b) & 0xFFFFFFFF
        h2 = (h2 + c) & 0xFFFFFFFF
        h3 = (h3 + d) & 0xFFFFFFFF
        h4 = (h4 + e) & 0xFFFFFFFF
    return ''.join(h.to_bytes(4, 'big').hex() for h in (h0, h1, h2, h3, h4))
```

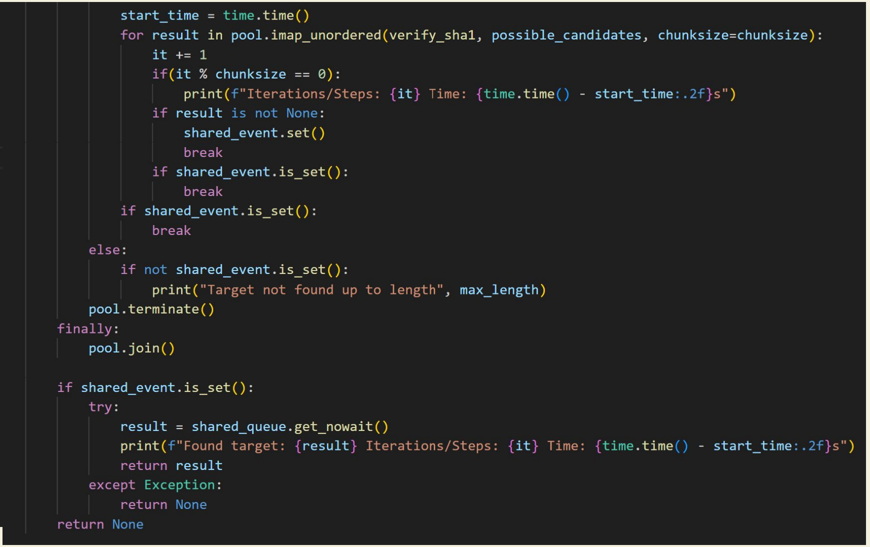
Thc Code – Cracking Hash



```python
def init_worker(event, queue, target_hash):
    global found_event, result_queue, target_hash_global
    found_event = event
    result_queue = queue
    target_hash_global = target_hash
```

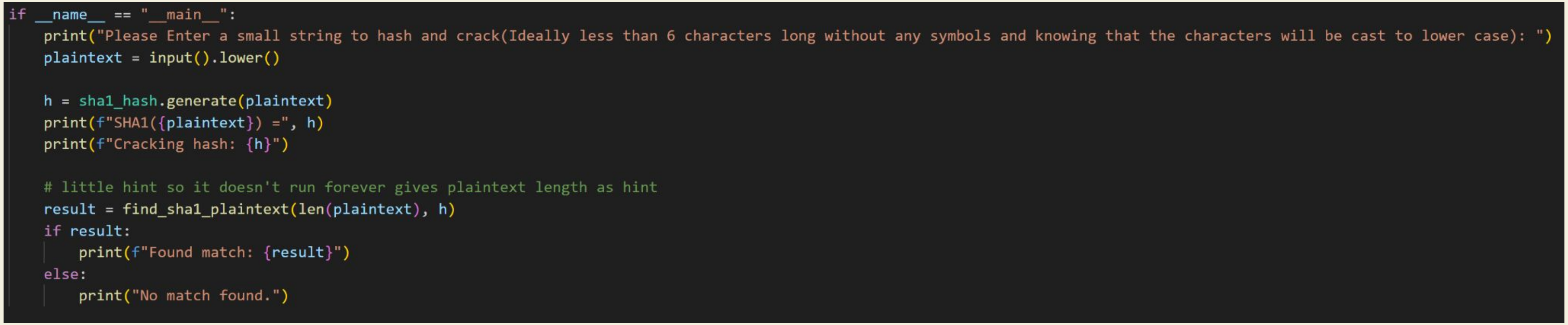


```python
def find_sha1_plaintext(max_length, target_hash, charset='abcdefghijklmnopqrstuvwxyz0123456789'):
    # Manager for Event/Queue
    manager = mp.Manager()
    shared_event = manager.Event()
    shared_queue = manager.Queue()
    it = 0

    pool_size = max(1, mp.cpu_count() - 1)
    # Prepare Pool with initializer to set shared globals
    with mp.Pool(processes=pool_size,
                 initializer=init_worker,
                 initargs=(shared_event, shared_queue, target_hash)) as pool:

        try:
            for length in range(1, max_length + 1):
                # If already found, break
                if shared_event.is_set():
                    break
                # Generate an iterator of candidate tuples
                possible_candidates = itertools.product(charset, repeat=length)
                # We use imap_unordered to lazily schedule tasks in chunks
                chunksize = 10000
```



```python
                start_time = time.time()
                for result in pool.imap_unordered(verify_sha1, possible_candidates, chunksize=chunksize):
                    it += 1
                    if(it % chunksize == 0):
                        print(f"Iterations/Steps: {it} Time: {time.time() - start_time:.2f}s")
                    if result is not None:
                        shared_event.set()
                        break
                    if shared_event.is_set():
                        break
                if shared_event.is_set():
                    break
            else:
                if not shared_event.is_set():
                    print("Target not found up to length", max_length)
            pool.terminate()
        finally:
            pool.join()

        if shared_event.is_set():
            try:
                result = shared_queue.get_nowait()
                print(f"Found target: {result} Iterations/Steps: {it} Time: {time.time() - start_time:.2f}s")
                return result
            except Exception:
                return None
        return None
```

Thc Code – Caller



```python
if __name__ == "__main__":
    print("Please Enter a small string to hash and crack(Ideally less than 6 characters long without any symbols and knowing that the characters will be cast to lower case): ")
    plaintext = input().lower()

    h = sha1_hash.generate(plaintext)
    print(f"SHA1({plaintext}) =", h)
    print(f"Cracking hash: {h}")

    # little hint so it doesn't run forever gives plaintext length as hint
    result = find_sha1_plaintext(len(plaintext), h)
    if result:
        print(f"Found match: {result}")
    else:
        print("No match found.")
```

Agenda

1. What is SHA1
2. The High Level Design
3. The Code
4. **Breaking SHA1**
5. Conclusion

Breaking SHA1

- The approach I went with was a bit of an “optimized brute force” approach i.e:
  - 1. Generate all combinations of possible characters lazily (i.e iterate on demand)
  - 2. Split these combinations up into chunks of 10000
  - 3. Assign a Python Process to search these 10000 combinations
  - 4. If found raise a shared event so the other processes know to stop running
- This however is a bit clunky and is a linear speed up to a exponential search space
- Chosen-Prefix attacks are now also viable
- But given the complexities(and the fact that it still takes a lot of time to create a duplicate hash) this method is left out

Agenda

1. What is SHA1
2. The High Level Design
3. The Code
4. Breaking SHA1
5. **Conclusion/Results**

Results

Plaintext	Cracking Time (seconds)
123	1.00
1234	45.30
food	8.8
hello	474.67
hell0	633.55

Conclusion

- I found out that semi-modern hashes are hard to exploit from scratch and that we tend to be forward looking when decommissioning/assuming the worst case
- Best practices for passwords(i.e include numbers and letters in your password) help to drastically improve security regardless of algorithm
- The scary thing about Algorithm insecurity is that even with complex methods that are hard to reproduce due to the fact that code is so easy to abstract/copy even 1 exploit makes an algorithm insecure