Week 9 Discussion

PA5: Cryptography
Reminder

https://cseweb.ucsd.edu/classes/wi23/cse127-a/pa/pa5.html

- Due date - March 14, 2023
- Groups of up to 2
- Four parts
  - Vigenère Cipher
  - MD5 Length Extension
  - MD5 collisions
  - RSA signature forgery
Part 1: Vigenère Ciphers

The combination of several Caesar Ciphers

<table>
<thead>
<tr>
<th>Plaintext:</th>
<th>ATTACKATDAWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key:</td>
<td>BLAISEBLAISE</td>
</tr>
<tr>
<td>Ciphertext:</td>
<td>BETIUOBEDIOR</td>
</tr>
</tbody>
</table>

Key ‘A’ means no shift
Key ‘B’ means shift by 1
Key ‘C’ means shift by 2
...

Idea:
- We know Caesar Cipher is vulnerable to frequency analysis
- However we can’t do frequency analysis on ciphertext of Vigenere Cipher, because each key is different
- Can we reduce Vigenere Cipher into several Caesar Ciphers?
- If we line up the ciphertext in n columns, where n is the length of the keyword. We can do frequency analysis on the n groups.
Kasiski Examination

To know the length of the key we do Kasiski Examination:

1. Look for strings of characters that are repeated in the ciphertext.
2. Find the distances between those repeated characters.
3. Since those distances are multiples of the key length, we find the greatest common divisor of those distances, which will give us the length of key.
4. The longer the string the better, the more distances you find the better.

Plaintext: crypto is short for cryptography. (dist is 20)
Key: abcdea bcdea abcdea abcdea abcdea abc
Ciphertext: csasxo kv siqux gqu csasxo htdthz.
Frequency analysis

After knowing the length, we need to do frequency analysis on each group of ciphertext that was applied the same key.

- The English language have more frequently used letter and less frequently used letters.
- In each group in your ciphertext, count the frequency of each letter.
- Since they are shifted by the same amount, this pattern of frequency would emerge.
- Map the shifted letters to the plaintext letters.
- Then you know how much each letter is shifted. (aka. the key)
Part 2: MD5 Length Extension

Given $H(\text{secret} || m)$
Create correct hash with $x$ appended, without knowing secret

http://bank.cse127.ucsd.edu/pa5/api?token=d6613c382dbb78b5592091e08f6f41fe&user=nadiah&command1=ListSquirrels&command2=NooOp

where token is MD5($user's \ 8$-character password $|| user=...$)
Part 2: MD5 Length Extension

- For this part it is pymd5.py which has some functions to get at individual steps of md5 hashing
- Key idea: **padding** is 1 followed by necessary number of zeros at end of message, but you need to be able to have a 1 followed by zeros as part of the message as well
- *Part 2: Experimenting* in the assignment walks you through this and should make the attack understandable
Part 2: MD5 Length Extension

Example: want to generate correct hash with “Good advice” appended to the end, without knowing preimage of that hash.

```python
m = "Use HMAC, not hashes"
print(md5(m).hexdigest())
3ecc68efa1871751ea9b0b1a5b25004d

bits = (len(m) + len(padding(len(m) * 8))) * 8

h = md5(state=bytes.fromhex("3ecc68efa1871751ea9b0b1a5b25004d"), count=bits)

x = "Good advice"
hs.update(x)
print(hs.hexdigest())
```

```
e1ca9db8eae1b8cbfacc63de828af6d0
```
Part 2: MD5 Length Extension

Example: want to generate correct hash with “Good advice” appended to the end, without knowing preimage of that hash.

```
x = "Good advice"
h.update(x)
print(h.hexdigest())
```

```
e1ca9db8eae1b8cbfacc63de828af6d0
```

```
result = m.encode("utf-8") + padding(len(m)*8) + x.encode("utf-8")
h_new = md5()
h_new.update(result)
print(h_new.hexdigest())
e1ca9db8eae1b8cbfacc63de828af6d0
```

- Why can’t we just do m+x? What is the padding for?

```
m = "Use HMAC, not hashes"
print(md5(m).hexdigest())
```

```
3ecc68efa1871751ea9b0b1a5b25004d
```

So $H(m \| padding)$
Part 2: MD5 Length Extension

http://bank.cse127.ucsd.edu/pa5/api?token=d6613c382dbb78b5592091e08f6f41f
e&user=nadiah&command1=ListSquirrels&command2=NoOp

where token is $\text{MD5(user's 8-character password || user=...)}$

Without knowing the password, we want to append 
&command3=UnlockAllSafes to the end of URL, and get the correct hash.

In other words, want to get:
$H(pwd \ || user=nadiah&command1=ListSquirrels&command2=NoOp&command3=UnlockAllSafes)$

So restore the state using the hash provided, 
update($x$)
Then you get $H(pwd \ || user=... \ || \ padding \ || \ x)$
Part 2: MD5 Length Extension

HINTS

- python3 len_ext_attack.py "http://.........NoOp"
- Only use urllib.parse.quote() for the padding
- Use the Gradescope autograder for testing if your attack works.
- https://deeprnd.medium.com/length-extension-attack-bff5b1ad2f70
Part 3: MD5 collisions

Two programs with different behavior that hash to the same thing

- We provide fastcoll which generates MD5 collisions
- You might need to build this code if it's not available on your OS so there is also a makefile to help
- Key idea: once you have a collision,
  - Adding identical suffixes to them will also collide because of the length extension property of MD5
- Explanation of prefix suffix

```
#!/bin/bash

cat << "EOF" | openssl dgst -sha256 > DIGEST
<BLANK LINE>
EOF

digest=$(cat DIGEST | sed 's/(stdin)= //')

echo "The sha256 digest is $digest"
```
Part 3: MD5 collisions

- How can you change suffix so that the program behaves like in the write up (print different stuff)?
- Remember the two files have different SHA256 hashes!
- You can reuse and modify the prefix and suffix we gave. (The provided code is just printing out the SHA256 value of the blobs.)
Part 3: MD5 collisions

HINT

● Think about how you can hide junk you are creating, will be useful later as well

● Use `openssl dgst -sha256 file1 file2` and `openssl dgst -md5 file1 file2` to verify

● Remember to submit `good` and `bad`, not `good.sh` or `bad.sh`, not `good.py` or `bad.py`

```
good
#!/bin/bash
...
```

submission file example
Part 4: RSA Signature - Textbook

- Alice has public key \((N, e)\) and private key \(d\) where \(x^{de} = x \mod N\)

Public Key \(pk\)

\[
N = pq \text{ modulus} \\
\text{e public exponent}
\]

Secret Key \(sk\)

\[
p, q \text{ primes} \\
d \text{ private exponent} \\
(d = e^{-1} \mod (p - 1)(q - 1))
\]

\[pk = (N, e)\]

\[m, s = \text{Sign}(m) = m^d \mod N\]

\[
\text{Verify}(m, s): m = s^e \mod N
\]

Works for the same reason RSA encryption does.
Part 4: RSA Signature

- To combat vulnerabilities, RSA is used with paddings
- A k-bit RSA key used to sign a SHA-1 hash digest will generate the following padded value of m:

\[
\text{Sig} = \text{padding(SHA1(m))^d mod N}
\]

\[
\text{Verify} = (\text{strip_padding(Sig^e mod N)} == \text{SHA1(m)})
\]
Part 4: RSA Signature

Verifier:
- Applies the RSA public exponent to reveal PKCS-1 padded data ($s^e \mod N$)
- Checks and removes the PKCS-1 padding
- Compares the hash with its own hash value computed over the signed data.

Sig = padding(SHA1(m))^d \mod N
Verify = (strip_padding(Sig^e \mod N) == SHA1(m))

00 01 FF...FF 00 30213009060520E03021A05000414 XX...XX
~~~~~~~~~~ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
k/8 - 38 bytes wide         ||         20-byte SHA-1 digest
ASN.1 "magic" bytes
Part 4: RSA Signature

- If verifier don’t check the length of the hash, and don’t check the SHA-1 is right-justified, We can use Bleichenbacher’s forgery attack. (only works for $e=3$)

- The ASN.1 data encodes the length of the hash within it, so this tells them how big the hash value is. Therefore some implementations don’t bother with length.
Part 4: RSA Signature

We can construct
\[ x = 00\ 01\ FF\ ...\ FF\ 00\ 3021300906052B0E03021A05000414\ XX\ ...\ XX \]

This gives us freedom to put anything we want to the right of the hash.
So we can arrange the value to be a perfect cube
\[ s = \sqrt[3]{x} \]

This way
\[ s = \sqrt[3]{x} \]
Remember the validator will check \[ m = ? s^e \mod N \], where \( e = 3 \) in our case.
Part 4: RSA Signature

- Do we need the exact root?
- How many bytes are we checking here?
- Remember the validator only cares about the hash value matching
Part 4: RSA Signature Forgery

HINTS

- If got stuck finding a valid root, think about how many higher bytes in the signature the verification process should recover?

- Don’t use openssl to test your solution. Write your own validation code that doesn’t check the length of FF s

roots.py

```python
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA
from roots import *
import sys

message = sys.argv[1]
# Your code to forge a signature goes here.

# some example functions from roots

root, is_exact = integer_nthroot(27, 3)
print(integer_to_base64(root).decode())
```
Thank you