CSE 127

Week 9 Discussion

PA5: Cryptography

Sumanth Rao

Overview

https://zzjas.github.io/cse127sp22/pa/pa5.html

- Due date - Wednesday, June 1st @ 11:59 PM

- Groups of up to 4

- Five parts
  - Vigenère Cipher
  - MD5 Length Extension
  - MD5 collisions
  - RSA signature forgery
  - Writeup

Ceasar Ciphers

Shift letters of plaintext by fixed amount to get ciphertext

```
Plaintext: ATTACKATDAWN

Ciphertext: DWWDFNDWGDZQ
```

A + 3 → D
T + 3 → W
C + 3 → F
…

Part 1: Vigenère Ciphers

UCSDCSE
Computer Science and Engineering

The combination of several Caesar Ciphers

```
 Plaintext: ATTACKATDAWN
       Key: BLAISEBLAISE
Ciphertext: BETIUOBEDIOR
```

Key 'A' means no shift
Key 'B' means shift by 1
Key 'C' means shift by 2
...

Each of you should see a *PA5: Ciphertext* assignment on Gradescope

PID: ....

```
ASABREVLDNXGSVWBVBIHWVXXCTLMUYALCIKUTV
JJNQUFCFDNPSANQGAVKKXOBELGZAPDCQ...
```

- Be careful, when copying the *ciphertext* from gradescope to your local system.
- It is a single string of alphabets with no spaces or newlines in between.
- If working in Use any one of the team-members

Part 1: Vigenère Ciphers

HINTS

- Caesar Cipher is vulnerable to *frequency analysis*
- Vigenère Cipher is composed of |**Key**| Caesar Ciphers that can be defeated individually
- How can you figure out |**Key**| ?
    - https://inventwithpython.com/hacking/chapter21.html
    - Or maybe just bruteforce??
- How do you know you got the correct key?

```
def vigDecrypt(ciphertext, key):
    decrypted = ''
    for i, ch in enumerate(ciphertext):
        decrypted += unshiftLetter(ch, key[i % len(key)])
    return decrypted

def unshiftLetter(letter, keyLetter):
    letter = ord(letter) - ord("A")
    keyLetter = ord(keyLetter) - ord("A")
    new = (letter - keyLetter) % 26
    return chr(new + ord("A"))
```

Part 2: MD5 Length Extension

Generate an URL where the token is the valid MD5 hash of extended parameters

http://bank.cse127.ucsd.edu/pa5/api?token=**6c256f4a53dd0068b2d82306d9c09d1c**&user=george&command1=ListSquirrels&command2=NoOp

where token is MD5(*user's 8-character password* || user=...)

- 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

pymd5.py

```
from pymd5 import md5, padding

print(md5(m).hexdigest())

padding(count)

h = md5(state=bytes.fromhex("3ec…"), count=512)

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

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.

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 its not available on your OS so there is also a makefile to help
- Key idea: once you have a collision, you can use your previous part to add identical suffixes to them and they will continue to collide

prefix

```
#!/bin/bash
cat << "EOF" | openssl dgst -sha256 > DIGEST
```

suffix

```
<BLANK LINE>
EOF
digest=$(cat DIGEST | sed 's/(stdin)= //' )
echo "The sha256 digest is $digest"
```

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 $x^{(de)} = x \mod N$
- To sign a message m, Alice computes $s = m^d$ and Bob can verify by checking that $s^e = m \mod N$
- Eve can trivially generate a signed message (m=s^e, s), where s^e is the message and s the signature
- Bob verifies the signature by checking by s^e=m! Uh oh...

Part 4: RSA Signature

- To combat the previous problem, structure is added to the message
- A k-bit RSA key used to sign a SHA-1 hash digest will generate the following padded value of m:

```
00 01 FF...FF 00 3021300906052B0E03021A05000414 XX...XX
      ^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ^^^^^^^

  k/8 - 38 bytes wide         ||           20-byte SHA-1 digest
                    ASN.1 "magic" bytes
```

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

Part 4: RSA Signature Forgery

- So now Eve can't compute just any s^e because it needs to match the format

- Note that number of FF bytes is determined in specification

- **What happens if this is not checked? (i.e. implementation just discards FF bytes until reaches a 00 byte)**

- Instead of generating a signature s such that s^e is of the form on the previous slide, it only needs to match on a certain number of high order bytes with any number of FF padding bytes

- Remember e=3 makes things simpler vs e=65537

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

```
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())
```

Part 5: Writeup

- 7 questions
  - 4 from part 3
  - and 3 from part 5

- Answers should be concise and complete

- Write a comment if you used your code from previous classes (e.g. CSE 107)

UCSDCSE
Computer Science and Engineering

Thank you