2018 CNS Homework 2

1. Super Cookie

1) HSTS: To prevent SSL stripping

```
Strict-Transport-Security: max-age=31536000;
includeSubDomains
```

When it's set, the browser will always send HTTPS request, and HSTS-related errors cannot be ignored.

2) Super cookies

By Leveraging the one-bit infomation stored in the client side, we can efficiently track the user.

For example, visiting `example.com`:

- Set HSTS in bit01.example.com
- Do nothing in bit02.example.com
- Set HSTS in bit03.example.com

...

3) Mitigation

1. Only permit HSTS state to be set for the loaded hostname.
2. Only permit HSTS state to be set for Top Level Domain + 1 (set HSTS of `example.com` from `abc.example.com`)
3. Ignore the HSTS state when sending subresource requests
    1. Visiting `example.com`

2. Load `http://bit01.example.com`, `http://bit02.example.com`
3. The browser will ignore HSTS and send request via HTTP.

Reference: https://webkit.org/blog/8146/protecting-against-hsts-abuse/

2. BGP

1) BGP Prefix Hijacking

Split 10.10.220.0/22 into (10.10.220.0/23, 10.10.222.0/23), and advertise them.

If we just advertise 10.10.220.0/22, it's possible the hijacking fails, because AS 1000 will also advertise the same IP range.

2) BGP MitM

According RFC 4271:

> AS loop detection is done by scanning the full AS path (as specified in the AS_PATH attribute), and checking that the autonomous system number of the **local system** does not appear in the AS path.

AS 1, 2, 1000 should not be hijacked. Therefore, we can prepend them in the AS path to trigger the loop detection. AS 1, 2 , 1000 will simply drop the route.

That is, to advertise:

- (10.10.220.0/23, [1000, 1, 2, 999])
- (10.10.222.0/23, [1000, 1, 2, 999])

(AS 1000 is optional here.)

Pros: The traffic from AS 3, 4, 5 can be forwarded to AS 1000 via AS 1, 2. The attacker can eavesdrop the traffic rather than simply dropping it.

Cons: AS 1000 can detect the MitM attack, because AS 1000 can also receive the advertisement.

3. PIN Authentication

$$
\begin{aligned}
&S \to C : HMAC_{K1}(RS_1 \| PIN_1) && (1)\\
&S \to C : HMAC_{K1}(RS_2 \| PIN_2) && (2)\\
&C \to S : HMAC_{K1}(RC_1 \| PIN_1) && (3)\\
&C \to S : HMAC_{K1}(RC_2 \| PIN_2) && (4)\\
&C \to S : E_{K2}(RC_1) && (5)\\
&S \to C : E_{K2}(RS_1) && (6)\\
&C \to S : E_{K2}(RC_2) && (7)\\
&S \to C : E_{K2}(RS_2) && (8)
\end{aligned}
$$

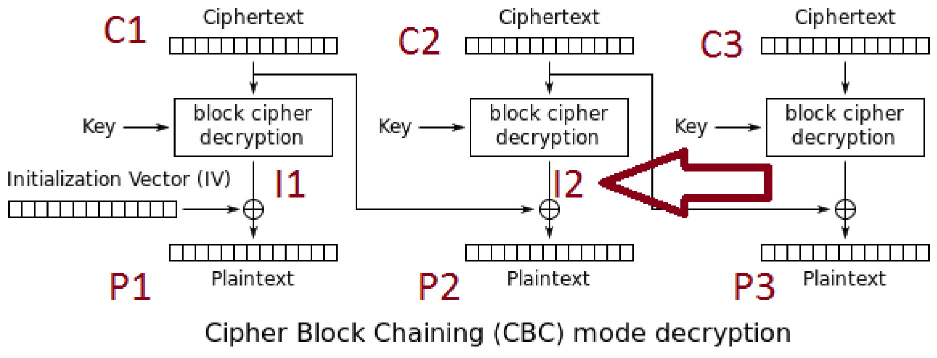
Possible values for `PIN1` and `PIN2` are only from `0000` to `9999`, we can brute force and guess the correct `PIN`.

We can use the server as an oracle, if we guess the wrong `PIN1` in (3) and send the wrong `RC1` in (5), the server will immediately abort, which means we know that `PIN1` is wrong. Otherwise, if you receive `RS1` (6) from the server, then you can confirm that you correctly guess `PIN1`. Similar approach can be done for `PIN2`. This is a well-known bug in WPS: https://dankaminsky.com/2012/01/26/wps2/

4. Can't Beat CBC

++cbc1++

> Reference: http://ctfsolutions.blogspot.com/2016/10/hitcon-ctf-2016-lets-decrypt.html



We already know the ciphertext and plaintext.

```
m = 'QQ Homework is too hard, how2decrypt QQ'
c =
'296e12d608ad04bd3a10b71b9eef4bb6ae1d697d1495595a5f5b98e409d7
a7c437f24e69feb250b347db0877a40085a9'
```

Seperate `m` and `c` into 3 parts, then we have `c1`, `c2`, `p1`, `p2`. So `i2 = c1 ^ p2`

Now we forge a ciphertext `c2+c2+c3` and let the server decrypt. At this point, `i1` is same as `i2`. Then `p1 = i1 ^ IV`. Since we already know what `i2` is and `i2 == i1`, then we can calculate `IV = p1 ^ i2`. You got the flag.

```python
#!/usr/bin/env python

from pwn import *

host = '140.112.31.96'
port = 10124

m = 'QQ Homework is too hard, how2decrypt QQ'
c =
'296e12d608ad04bd3a10b71b9eef4bb6ae1d697d1495595a5f5b98e409d7
a7c437f24e69feb250b347db0877a40085a9'

c1 = c[:32]
c2 = c[32:64]
c3 = c[64:]

p2 = m[16:32]
i2 = xor(c1.decode('hex'), p2)

cfake = c2+c2+c3

# r = process(['python', 'chal1.py'])

r = remote(host, port)
```

```
r.recvuntil('> ')
r.sendline('1')
r.recvuntil(': ')
r.sendline(cfake)
pfake = r.recvline()

pfake1 = pfake[:16]
flag = xor(i2, pfake1)
print flag

r.interactive()
```

++cbc2++

Just implement your Padding Oracle Attack...

```
#!/usr/bin/env python

from pwn import *

host = '140.112.31.96'
port = 10125

# r = process(['python', 'chal2.py'])
r = remote(host, port)

sleep(2)

r.recvuntil('> ')
r.sendline('1')

r.recvuntil('IV:\n')
IV = r.recvline().strip()
r.recvuntil('Encrypted_FLAG:\n')
enc_flag = r.recvline().strip()
print enc_flag
enc_flag = enc_flag.decode('hex')
```

```
print len(enc_flag)

split_flag = []
split_flag.append(IV)

for i in range(int(len(enc_flag)/16)):
    split_flag.append(enc_flag[i*16:i*16+16])

r.recvuntil('How2decrypt? QQ\n\n')

flag = ''
guess = ['\x00'] * 16
for numBlock in range(len(split_flag) - 3, -1, -1):
    for i in range(15, -1, -1):
        count = -1 + -16 * (len(split_flag) - numBlock - 3)
        for j in range(15, i, -1):
            guess[j] = chr(ord(split_flag[numBlock][j]) ^ 
ord(flag[count]) ^ (16 - i))
            count -= 1
        for c in range(256):
            guess[i] = chr(ord(split_flag[numBlock][i]) ^ c ^ 
(16 - i))
            tmpMsg = IV + ''.join(guess) + 
split_flag[numBlock + 1]
            tmpMsg = tmpMsg.encode('hex')
            r.sendline(tmpMsg)

            if r.recvline().strip() == 'Success':
                flag = chr(c) + flag
                print 'Success: ' + flag
                break

            if c == 255:
                assert False
```

++cbc3++

> Reference: https://l4w.io/2015/01/tetcon-ctf-2015-crypto200-the-poodle-attack/

Details on how the exploit works is linked above.

```
c[0] = 0acbad9b05fbc03551afdf147446a49b -> IV
c[1] = 88754146ca526818f5d61a14b4b3fc5d -> AAAAAAAAAAAAAAAA
(ENCRYPTED)
c[2] = 73e12492fbf42c7d326d21a5329dc673 -> FLAG (ENCRYPTED)
c[3] = ddae3c3ca20d8509dc899f3e3a50a43d -> FLAG (ENCRYPTED)
c[4] = 2983f99e0b6b0e28ee3270f42b9ce770 -> AAAAAAAAAAAAAAAA
(ENCRYPTED)
c[5] = c74bf206eb25e67371b2dab91af52fc3 -> HMAC (ENCRYPTED)
c[6] = ffffffffffffffffffffffffffffffXX -> APPEND THIS BLOCK
c[7] = ddae3c3ca20d8509dc899f3e3a50a43d -> FLAG (ENCRYPTED)
```

Forge the ciphertext by appending `c[6]` and `c[7]`. After decryption, if `XX ^ YY = 0x1f`, then last 2 blocks `c[6]` and `c[7]` will be unpad so that you can bypass the MAC authentication, and you can guess the flag by calculating `p[3][-1] = c[2][-1] ^ XX ^ 0x1f`.

```python
#!/usr/bin/env python

import copy
from pwn import *

host = '140.112.31.96'
port = 10126

BS = 16

# r = process(['python', 'chal3.py'])
r = remote(host, port)

sleep(2)


def split_block(data):
    split_data = []

    for i in range(int(len(data) / 16)):
        split_data.append(data[i*16:i*16+16])

    return split_data

def encrypt(n):
    r.recvuntil('> ')
    r.sendline('1')

    m1 = 'A'*16 + 'B'*(n % BS)
    m2 = 'A'*16 + 'B'*(16 - (n % BS))

    r.recvuntil('m1: ')
    r.sendline(m1)
    r.recvuntil('m2: ')
    r.sendline(m2)

    r.recvuntil('Encrypted_message:\n')
    data = split_block(r.recvline().strip().decode('hex'))

    return data

def decrypt(c_block, m):
    flag = ''

    for i in range(256):
        xor_byte = ord(c_block[m][-1])
        new_block = copy.copy(c_block)
        new_block.append(new_block[m + 1])
        new_block[len(c_block) - 1] = '\xff'*15 + chr(i)

        r.recvuntil('> ')
        r.sendline('2')
        r.recvuntil('m: ')
        r.sendline(''.join(new_block).encode('hex'))
```

```
            res = r.recvline().strip()
            if res == 'Success':
                c = 31 ^ i ^ xor_byte
                flag += chr(c)
                print flag
                break

    return flag

def exploit():
    flag = ''
    for m in range(1, 5):
        for n in range(BS-1, -1, -1):
            data = encrypt(n)
            flag += decrypt(data, m)
        print 'Partial Flag: ' + flag
    print 'FLAG: ' + flag

if __name__ == '__main__':
    exploit()
```

5. Man-in-the-middle Attack

1. Guess the unknown password pwd in one round of DH key exchange and pass the correct number generated by two servers for the other two rounds.
2. If your guess is correct => flagA ^ keyA == flagB ^ keyB
3. Guess three pwds (should within 60 guess) then execute a correct DH with one server to get the flag.

Basic idea explained:

1. MitM (reflection) in the first round: guess the g_1', receive server A public key g_1^a and server B g_1^b
2. Send $g_1'^e$ to both server A and B. server A and B compute $(g_1')^a$, $(g_1')^b$ respectively. e can be arbitrary.
3. Exchange the two server's public key in the remaining 2 rounds:

$$serverAflag : F_A = (g_1'^e)^a \oplus (g_2^b)^a \oplus (g_3^b)^a$$

$$serverBflag : F_B = (g_1'^e)^b \oplus (g_2^a)^b \oplus (g_3^a)^b$$

4. If $g_1' = g_1$, then the two equations are equal.
5.

$$(g_1)^b \oplus F_A = (g_1^e)^b \oplus (g_1)^a \oplus (g_2^b)^a \oplus (g_3^b)^a$$

$$(g_1)^a \oplus F_B = (g_1^e)^b \oplus (g_1)^a \oplus (g_2^b)^a \oplus (g_3^b)^a$$

```python
from pwn import *
import hashlib

p =
2626034878161944881812583523269882322103765919961462525429196
0587880500546969378231271874991509984140890844676040448123664
6436295067318626356598442952156854984209550714670817589388406
0592850645429057187104757751215659835867801368256002643808687
7002968092561858839199793447319105459081225619780603461815775
1903
ip = '140.112.31.96'
port = 10127
# get the public key of a server
def getPub(server):
  questionLine = server.recvuntil('server:').split('\n')
  return int(questionLine[1][14:])
# exchange public key for several round
def exchange(serverA, serverB, roundNum):
  while roundNum > 0:
    serverA.sendline(str(getPub(serverB)))
    serverB.sendline(str(getPub(serverA)))
    roundNum -= 1
# get flag and close server connection

def getFlag(server):
  flag = server.recvline()[9:-1]
  server.close()
  return int(flag)

roundNum = 3
solvedNum = 0
gs = []
pwds = []
key = 0
c = 10
while solvedNum < 3:
  # try my g ^ c
  for pwd in range(2, 21):
    serverA = remote(ip, port)
    serverB = remote(ip, port)
    exchange(serverA, serverB, solvedNum)

    print 'trying pwd{} = {}'.format(solvedNum, pwd)
    t = pwd
    pwd = int(hashlib.sha512(str(pwd)).hexdigest(), 16)
    g = pow(pwd, 2, p)
    C = pow(g, c, p)
    A = getPub(serverA)
    B = getPub(serverB)
    serverA.sendline(str(C))
    serverB.sendline(str(C))
    KA = pow(A, c, p)
    KB = pow(B, c, p)
    keyA = int(hashlib.sha512(str(KA)).hexdigest(), 16)
    keyB = int(hashlib.sha512(str(KB)).hexdigest(), 16)

    exchange(serverA, serverB, roundNum - solvedNum - 1)
    flagA = getFlag(serverA)
    flagB = getFlag(serverB)
    # if my g is correct, K should = gi ^ ac = A ^ c
    # flag xor gi ^ ac should be the same
    if flagA ^ keyA == flagB ^ keyB:

      gs.append(g)
```

```
            pwds.append(t)
            solvedNum += 1
            break

server = remote(ip, port)
key = 0
for g in gs:
  C = pow(g, c, p)
  A = getPub(server)
  server.sendline(str(C))
  K = pow(A, c, p)
  key ^= int(hashlib.sha512(str(K)).hexdigest(), 16)
encryptedFlag = getFlag(server)
hexFlag = hex(encryptedFlag ^ key)
flag = hexFlag[2:].decode('hex')
print flag
```

6. Cloudburst

The server is using the SSL certificate, with `the-real-reason-to-not-use-sigkill.csie.org` in the Common Name field. Thus, if we access the correct origin IP, the server will return the same certificate.

Another clue in the problem description is "The geographical location is in CSIE building, NTU". Due to IPv4 locality, the possible IP range can be highly reduced.

- NTU campus: 140.112.0.0/16: ~65536 IP addresses.
- CSIE building: According to the list, ~1600 IP addresses.

After brute force all the possible IP addresses, the origin IP is `140.112.91.250`.

Reference: Vissers, Thomas, et al. "Maneuvering around clouds: Bypassing cloud-based security providers." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

7. One-time Wallet

1. Use previous random numbers to restore internal state of python PRNG.
2. Predict future random numbers by internal state.
3. MT19937 PRNG requires 624 consecutive numbers to predict the next state.

```python
from pwn import *
from mt19937predictor import MT19937Predictor

def str2Nums(string, byte):
    nums = []
    for i in range(byte / 4):
        s = string[8 * i: 8 * (i + 1)]
        nums.append(int(s, 16))
    return nums

def genHexString(byte):
    global predictor
    s = ''
    for i in range(byte / 4):
        r = predictor.getrandbits(32)
        s += '{0:0{1}x}'.format(r, 8)
    return s

def genPassword():
    return genHexString(20)

server = remote('140.112.31.96', 10128)
data = server.recvuntil('?')
lines = data.split('\n')
i = 0
randomNumbers = []
while True:
    walletNum = lines[i][7:]
    address = lines[i + 1][9:]

    randomNumbers.extend(str2Nums(address, 12))
```

```
        password = lines[i + 3][10:]
        randomNumbers.extend(str2Nums(password, 20))
        if (walletNum == '100'):
            break
        i += 5

    data = server.recvuntil('?')
    lines = data.split('\n')
    address = lines[2][9:]
    randomNumbers.extend(str2Nums(address, 12))

    predictor = MT19937Predictor()
    for num in randomNumbers:
        predictor.setrandbits(num, 32)

    server.sendline(genPassword())
    server.interactive()
```

8. TLS Certificate

Just utilize the versatile OpenSSL.

https://gist.github.com/fntlnz/cf14feb5a46b2eda428e000157447309