CNS HW2

B03902082 資工四 江懿友

1. Super Cookie

1.1 HSTS

HSTS is a mechanism to protect against "protocol downgrade attack" and "cookie hijacking attack". It works by forcing the client (web browser) to communicate with the server only over https and never via http. Upon a client's first visit to a website, a HSTS header will be sent over https, instructing the client that "any future connection to this domain of http should be automatically converted to https connection" and "supply a error message to the user if https connection cannot be established when visiting this domain".

1.2 Super Cookie via HSTS

The tracker can assign for each user a unique ID, say a 32-bit integer. Then on the first visit of the user, the tracker sets the HSTS policy of his 32 subdomains according to the user ID, and the 32 subdomains are accessed by the user by 32 images embedded in the webpage. Thus 32 1-bit super cookies can be stored on the user's browser. When the user visits the site again, he would access the 32 images of different subdomains, and access them with http or https according to the users HSTS record. So by checking if his 32 subdomains are accessed via http or https, the tracker can recover the user's unique ID.
Reference: https://thehackernews.com/2018/03/hsts-supercookie-tracking.html

1.3 Mitigations

Only allow setting HSTS for the visiting hostname. For example if I'm visiting "www.example.org" then only the HSTS setting of "www.example.org" should be set but not "sub.www.example.org" nor "sub.example.org". This way the tracker cannot abuse HSTS for tracking by browser automatically loading resources from tracker's subdomains.

2. BGP

2.1 Prefix Hijacking

It is likely that the attacker is performing "BGP prefix hijacking".
The attacker announces itself to be the owner of "10.10.220.0/23" and "10.10.222.0/23". Since BGP routing prefers the longest matching prefix, the other ASs would route destinations of "10.10.220.0/22" to AS999 instead of AS1000 because AS999 has a longer match than AS1000 (23 > 22).

2.2.a Spurious BGP Update

UPDATE {"10.10.220.0/23", {AS999, AS2, AS1, AS1000}}
UPDATE {"10.10.222.0/23", {AS999, AS2, AS1, AS1000}}

2.2.b Explain 2.2.a

For AS3, AS4 and AS5, they would route destination "10.10.220.0/22" to AS999 because the route to AS999 have a longer matching prefix.
For AS1 and AS2, they won't accept AS999's spurious BGP update according to the "loop prevention" rule.

2.2.c

Advantage: AS999 doesn't have to pretend to be the owner of "10.10.220.0/22" like in 2.1. So if AS3, AS4 or AS5 were to verify the owner of "10.10.220.0/23" and "10.10.222.0/23" then they would find out that AS1000 indeed owns "10.10.220.0/23" and "10.10.222.0/23".
Disadvantage: The attacker cannot tamper with packets originating from AS1 and AS2.

3. PIN Authentication

The server S can be used as a oracle that answers "Does X equals PIN1?" where X is the guess of PIN1.
First let's guess PIN1 to be X. At step (3) we send HMAC_K1(RC1 || X) to server. If X does not equal PIN1 then the server would close the connection after step (5). If X does equal PIN1 then the server would proceed to step (6) and send some message back, thus indicating I made a right guess. The total combination of PIN1 is 10^4 = 10000 so I can use this oracle to brute force PIN1 in feasible time.

After obtaining PIN1 a similar process can be used to search PIN2. Only this time I use "whether or not server close connection at step (8)" as a indicator of whether or not I made a right guess. Thus I can recover the whole PIN in feasible time.
The vulnerability of the protocol is that it aborts immediately when a verification fails, and this leaks information to the attacker. Another vulnerability is that the PIN is split in half and the two halves are used separately. This reduced the effective key length from 8 digits to 4 digits and made my brute force attack possible.

4. Can't Beat CBC

4.1

Flag = BALSN{IV=KEY=GG}

Suppose
m = 'QQ Homework is too hard, how2decrypt QQ'
c =
'296e12d608ad04bd3a10b71b9eef4bb6ae1d697d1495595a5f5b98e409d7a7c437f2
4e69feb250b347db0877a40085a9'
where m is the plain text and c is cipher text

Then I partition the plain and cipher text into three blocks
m = M1 || M2 || M3
c = C1 || C2 || C3
Then I ask the server to decrypt the message M1 || C1 || C2 || C3.
So when CBC operation mode tries to decrypt this message it does:
D(M1) ^ IV = P1
D(C1) ^ M1 = P2
D(C2) ^ C1 = P3
D(C3) ^ C2 = P4

P1 is some random value that we don't care.
Because D(C1) = IV ^ M1, so P2 = IV which is the flag.
And D(C3) = C2 ^ M3 so P4 = M3. Thus the decrypted message have proper padding.

4.2

Flag =
BALSN{1T_15_V3RY_FUN_T0_533_TH3_FL4G_4PP34R_0N3_BY_0N3_R1GHT_
XD}

This problem can be solved by CBC padding oracle attack. Searching for the plaintext byte by byte is time consuming, so the connection to server may timeout, thus I have to reconnect to the server periodically.

4.3

Flag = BALSN{N33D_B3TT3R_P4DD1NG_M3TH0D_T0_PR3V3NT_P00DL3_4TT4CK_SSLv3}

The vulnerability is that the MAC is checked after the padding has been removed, so I can alter the ciphertext without being detected by MAC checking by carefully choosing the padding length so that when the server unpads it removes my altered message.
My attack roughly works like:

1. Ask server to encrypt FLAG, so I get (C1 || C2 || MAC(C1 || C2) || padding), where C1 || C2 is the encrypted FLAG.
2. Ask server to decrypt (C1 || C2 || MAC(C1 || C2) || padding || C1' || C2), where C1' is C1 block modified using a way similar to CBC padding oracle attack. I try to guess the last byte of C1's plaintext, if my guess is correct then the "padding || C1' || C2" can be correctly removed when server unpads and the MAC checking can pass.
3. After I find the last byte of C1's plaintext, I ask server to encrypt 'a' || FLAG, so I get ('a' || C1 || C2 || MAC('a' || C1 || C2) || padding). The ciphertext is shifted by one byte, so when I use the same procedure to recover the last byte of C1's plaintext I actually get the second to last byte of the plaintext.

So by repeating this procedure I can eventually recover the whole FLAG.

5. Man-in-the-Middle Attack

Flag = BALSN{Wow_you_are_really_in_the_middle}

This problem is vulnerable to reflection attack.
Let pwd1, pwd2, pwd3 be the three unknown parts of the password.
First I initiate two connections with the server, and I receive B11=g1^b11 and B21=g1^b21 from the two connections, where b11, b21 and g is unknown by me.
Then I send B21 to connection 1 and B11 to connection 2. This way connection 1 and 2 would both derive the secret B21^b11 = g1^(b11*b21) = B11^b21.
The same goes for the second key exchange, I forward B12=g2^b12 to connection 2 and B22=g2^b22 to connection 1. So now connection 1 and connection 2 both derives the same secret.

Then I will receive B13 = g3^b13, B23 = g3^b23 from the two connections. I then try to guess the value of g3 and send A = g3^1 to both connections. So connection 1 derives the secret g3^b13 and connection 2 derives g3^b23.
Then I'll receive two encrypted messages. The messages are the FLAG xor masked with the established shared secrets. Because connection 1 and connection 2's shared secret only differ at the last part (the part derived from g3), so when I xor the two messages together I get (A^b13 xor A^b23). If my guess of g3 is correct then this value will equal to (B13 xor B23) else there is a high chance that (A^b13 xor A^b23) =/= (B13 xor B23). So now I have a way to verify my guess, I can search the value of g3 by trying pwd3 from 1 to 20, as g3 is calculated from pwd3.
Then I can use the same procedure to search g2 and g1. After finding all of g1, g2 and g3, I can recover the encrypted FLAG.

6. Cloudburst

Flag = BALSN{what_a_C1oudPiercer}
Origin IP: 140.112.91.250

Inspired by the Cloudpiercer paper (link), I scan the csie department IP region (according to this table http://ccnet.ntu.edu.tw/NTU/IP/IPserver.html) and dumped all of the hosts' SSL certificate. My assumption is that knowing the origin host is listening on port 443, it is likely that origin host's https service uses a certificate whose CN includes "the-real-reason-to-not-use-sigkill.csie.org".
This assumption is proved to be correct as I dumped a SSL certificate from the IP 140.112.91.250 that shows:

subject=/CN=the-real-reason-to-not-use-sigkill.csie.org
issuer=/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3

7. One-time Wallet

Flag = BALSN{R3tir3_4t_tw3nty}

The vulnerability is that the address / password generator uses random.getrandbits() to generate pseudo random bit sequences, however the PRNG is actually predictable.
The PRNG used by random.getrandbits() is Mersenne twister. Mersenne twister's internal state can be determined by 624 consecutive outputs. In other words, knowing the first 624 outputs of a Mersenne twister, one can predict the 625-th and the rest of output.

I used this tool to predict Mersenne twister:
pip3 install mersenne-twister-predictor

8. TLS Certificate

Flag = BALSN{t1s_ChAiN_0f_7ru5t}

This problem is rather straightforward. I read OpenSSL's documentation to learn how to sign a X509 certificate.
First I downloaded csie_cert from https://www.csie.ntu.edu.tw/ .
Then I craft a new X509 certificate, and I copy csie_cert's subject and publickey into my certificate. I also set my certificate's serial number, not_before and not_after to some arbitrary value. At last I set my certificate's issuer to BALSN rootCA's subject. Finally I sign my certificate with the provided RSA private key and send the signed CA to server.