Alexander Lotero
Offensive Security
Spring 2023

Week 13 - Crypto 2

This week's challenges were "RSA1," "RSA2," and "Cool Block Challenge."

RSA1

The first challenge that I completed this week was "RSA1." Like the challenges from "Crypto 1," we are provided with a prompt "a simple RSA challenge. Can you crack it?" and a ciphertext file *ciphertext.txt*. The ciphertext file contains the following RSA parameters used in the following equation:

c = (m ^ e) mod n

- n: the modulus, a public parameter, the product of two large prime numbers "p" and "q"
- e: the exponent, another public parameter, a small prime number, in this case "5"
- c: the ciphertext, technically also a public parameter, the encrypted message
- m: not provided, the original plaintext

We begin by creating a solver script, *rsa1solver.py* that reads these values in from the ciphertext file and assigns them to variables "n," "e," and "c." Next, for the attack itself, we note that given that "e" is 5, almost the smallest possible option, we suspect that this ciphertext may be vulnerable to the small exponent attack (1). While further researching this attack, we encountered an open-source tool "RsaCtfTool" available on GitHub (2). RsaCtfTool, a python script, is capable of implementing several well-known RSA attacks including "same n, huge e," "Wiener's attack," and, lucky for us, the small exponent attack.

We clone the repository and start by examining the tool's help page. Doing so reveals that while we can specify an attack type via the "–attack" option, we can also leave the option blank to attempt every attack in the tool's database. Further, among the many input options for the script is the option to just provide "n," "e," and "c."

Thus, because our script already extracts the parameters from the provided

ciphertext file, we create a subprocess (3) within our script to run RsaCtfTool and print its output upon completion:

```python
#!/usr/bin/env python3

import subprocess

# open ciphertext2 file
file = open("ciphertext.txt", "r")
input = file.read()
file.close()

#extract important values from provided files
input_list = input.split(" = ")
n = int(input_list[1][:-2])
e = int(input_list[2][:-2])
c = int(input_list[3][:-1])

result = subprocess.run(["./RsaCtfTool.py", "-n", str(n), "-e", str(e), "--uncipher", str(c)], capture_output=True)

output = result.stderr.decode("utf-8")

print(output)
```

This results in the following terminal command being run as a subprocess:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week13/rsa1]
└─$ ./RsaCtfTool.py -n 1283873041255102302747890710141229008409895085885531108011838170521155218939209148481810690985701975480970357250608155028888338730
003751860331840812985057911887180686518878474440113045267709992884952605180309669514152326408985993262727944123979125033841673976339109070277817828778611
998965776004747806653073211454300822610180033484116696610032950943035208739404310770805549768934478995341807589136074534979725941332499888157775366330050
829275536363027995169964085822620805847302087604428638673906085981824957408969490494225718921953622352676401123285151767360415190826499827899704904141877
161097560742871493115634720927238611276098281988337346625988691041925410093236136561133076133856408246458982564716985861348283708378633534978247660734186
439465501376145543074132038982148234344705664267825870783553531044963082911890652419801050967847166321034723164987505586760668023052348442353038881478678
127863347205233226941478065912414734604111683805667470019087273101316603121700513877611186852218494578683139987137227366647556610951180513465957418280185
952747777178828395262232889090331969913287858008224302067318846442752865298231469757953741948690492798499374260650121371330063379053546822899537279598481
29334820782861235805967388539831 -e 5 --uncipher 80535491957867449270763317789170401825716325711608849070785044020759937558942548546182118415359720375052
231303566951380377942299288333418385585815509728060122158128606211908384528275897985828819332463031434961253794230786516769640622256173200701183491429905
385824613173592581864865244592471045926113902685049289048845551939867833142706361742149655484285093954528501575970791767218262850671596858142076414053390
71091506873082916985182670458954820033818744270368011527294047666804951966588317621357926108000341426799341717769333930772518715180452122545293
private argument is not set, the private key will not be displayed, even if recovered.
```

The script reports which attack it is currently attempting, any errors encountered while attempting the attack, the number of attempts completed, and the time taken to complete each attempt.

We allow the script to run for several minutes until a successful attack is identified and with a print statement we can view the decrypted message:

```
Results for /tmp/tmplfol8zo8:

Unciphered data :
HEX : 0x666c61677b795f63616e745f70797468306e5f64305f6e74685f72303074733f5f6132373330323833373431377d
INT (big endian) : 240545625414752547968424892933083827388131291531760396579969476988304166763533697205953996292739835852538066813
INT (little endian) : 294074155811479350382632064326225735625939457473626963524631868846893297792145661694746809470788038968257637478
utf-8 : flag{y_cant_pyth0n_d0_nth_r00ts?_a27302837417}
utf-16 : 汦条祻损湡彴祰桴渰摟弰瑮彨〲琰㽳慟㜲〳㠲㜳ㄴ紷
STR : b'flag{y_cant_pyth0n_d0_nth_r00ts?_a27302837417}'
```

As seen in the image above, the RsaCtfTool script identified an attack to discover the key and complete the decryption process, needing only “n,” “e,” and “c.” The output reveals our flag, indicating that we have successfully completed the challenge.

RSA2

The second challenge I completed this week was “RSA2.” In this case, we are

provided with a hint “this one requires some math, gmpy2 is a nice Python library” and two ciphertext files, *ciphertext1.txt* and *ciphertext2.txt*. Each ciphertext file, like the one found in “RSA1,” includes the three public RSA parameters: the modulus “n,” the exponent “e,” and the ciphertext “c.” Opening the two files, we see that while the exponent “e” and the ciphertext “c” are unique between the two, they share the same modulus “n.” This likely means that the RSA implementation used is vulnerable to the “same modulus attack” as referenced in this week’s slides. Further research into the attack (4, 5), we find a more detailed explanation of the function shown in the slides:

$$
\begin{aligned}
(C1)^{\alpha} \bmod n * (C2)^{\beta} \bmod n &= ((P^{e1})^{\alpha} * (P^{e2})^{\beta}) \bmod n \\
&= (P^{(\alpha * e1 + \beta * e2)}) \bmod n \\
&= (P^{1}) \bmod n \\
&= P
\end{aligned}
$$

Where “*α*” and “*β*” are the outputs of the extended Euclidean algorithm (6), “*n*” is the is the shared public modulus, “*e1*” and “*e2*” are the public exponents for each ciphertext, respectively, and “*c1*” and “*c2*” are the two ciphertexts, respectively.

We now have the steps required to “solve for” the plaintext, we just need to implement this solution. To do so, we create a script *rsa2solver.py*. Like “RSA1,” we start by reading the provided parameters into our script from *ciphertext1.txt and ciphertext2.txt*. We read in the values, convert them from strings to integers, and store them in the following variables: *n* - the shared modulus, *e1* - the exponent for *ciphertext1.txt*, *c1* - the encrypted message from *ciphertext1.txt*, *e2* - the exponent for *ciphertext2.txt*, and *c2* - the encrypted message from *ciphertext2.txt*.

Next, we perform the attack with a function titled *com_mod_atk()*, passing each of the parameters listed above as arguments: *(c1, c2, e1, e2, n)*. First, because this attack requires that “e1” and “e2” are coprime (7) with one another, we implement the *math.gcd()* function found within the python “math” library (8) so as to not “reinvent the wheel.” The *math.gcd()* function, taking *e1* and *e2* as arguments, returns their greatest common divisor. We then check that this value is in fact 1, indicating that they are coprime, and print an error message if they are not. Otherwise, we proceed.

The next step in our solution is to find “α” and “β” using the Extended Euclidean algorithm (5, 6). To do so, once again not reinventing the wheel, we implement the existing *libnum.xgcd(a, b)* function from within the python “libnum” library (9, 10). This function, implementing Extended Euclidean, takes *e1* and *e2* as arguments and returning (x, y, g) such that:

$$a * x + b * y = gcd(a, b) = g$$

In our case, we call the function with *libnum.gcd(e1, e2)* and it returns (α, β, g) such that:

$$e1 * \alpha + e2 * \beta = gcd(e1, e2) = g$$

We receive the following values when run on the challenge’s ciphertexts:

$$\alpha = 32641; \quad \beta = -128; \quad g = 1;$$

With *α* and *β* identified, we now have all of the necessary parameters to solve the equation defined above. First, we compute *(C1)^α mod n* with python’s built-in *pow(base, exp, mod)* function that computes *base* to the power of *exp* modulo *mod*. For our purposes, *C1_to_α = pow(c1, α, n)*, which returns a very large integer:

```
C1_to_α = pow(c1, α, n) =  32790717564017574122844866048840552353775465380500662178295737397423160390317004622544855951679636331504241666276726093212077612688207422
60048841323531043360092179777880922736317052070920978582961523907557917317259259193030531438300922617053439568252573740282390703344892349962476554579158145006068456
86347359905410940333401936553931010216578206703840967181838990543714616323171090181904725524602389343419267859466641682414150176057785707586879847545167679080163334
65051818341874855048741244406367547795386005299944339727241458276474714639579825648751104067705008354273284918947598321631543049518881154404544070186263992475352967
92867358397806784586848728812956194268781674571066072757686416331250936938520355882367784602375937568463422829883480970550101990344981458886788787272337841893023299
96940656568619086284313904713929322999279483882788469503206271043582269234763564370623739445133279622274365191789898341492133731311158080090527599424871620452440289
68488719750873310028722435102439584928349536391897663701153725955953237374577772096839664101351489911460302840614488392490613123889318880050892988083447621581419279
89847923153898591280620081223132908313253272995983154026227426999720861497281845725947842238120024882789946030 5
```

Next, we repeat the process for *(C2)^β mod n*. Our code, *C2_to_β = pow(c2, β, n)* which once again computes a very large integer:

```
C2_to_β = pow(c2, β, n) =  79806517759515317693302414754804931744607838264476417504845799529621369937845317959992303022348317968052391506062891547634415924705260345
39546753655232368579656445907911993920308032040867450538538658629978030449103480893938608010946752168186417149668889687128450933490111430514623525775146332673982689
89336589516816524005762966008371772733092802740180293302952835034715026748120563314260133669256526849498377005140705741677350128468626982838433070247250464032951716
84535725626518913120391100968567902813437722674012628749194174687495645005867391548611545458247323455186458194274822162125566810936179273802143376835764868307967008
33060754201269302442423488994589202882471807967475585895089284825882528878249580263034923394353967474799722221658217440447610900637978253043952051169609932226888423
15742360436200219724768537412145657858374203420930252743578424176727062674261586233192124881040775989336649028850514821973002125515453669029194328728581054885127607
07952690117308591880712649144910154234412122848144119210258109603035689883040061153939465156974012214911729595810689189933772308579968533837081828785346735419990803
63903081366582513032013768750365374169487475687599288486196966354309021230691713332327253543184838223824742750
```

Finally, as the last step of computation, we multiply these two values and modulo the shared modulus, “n.” Now, while the output of this process should be the plaintext, it remains in the form of a very large integer. Thus, we still need to convert this value to something human-readable. To do so, we deploy python’s built-in *to_bytes()* function to convert the integer to its byte representation (12). Finally, we print the result of this conversion:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week13/rsa2]
└─$ python3 rsa2solver2.py
b'flag{c0ngr@ts!_youre_d0n3_with_h0meworks!_8114f0df3299}'
```

As shown in the image above, the bytes object printed to our screen is the flag itself, indicating that we have successfully completed the challenge.

Cool Block Challenge

The third and final challenge this week was “Cool Block Challenge” or “CBC”. This challenge provides us with a short hint “this challenge is off the chain,” a *ciphertext.txt* file, and finally, a netcat command to connect to the challenge running on the live CTFd server. Opening the *ciphertext.txt* file, we see that we are provided with an initialization vector (IV) (13), as well as the encrypted ciphertext itself.

We run the provided netcat command to receive the following message and prompt:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week13/cbc]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1478
Please input your NetID (something like abc123): aal562
hello, aal562. Please wait a moment ...
Messages should be hex strings, in the format [IV][Ciphertext].
We're using PKCS#7 padding.
Gimme a message:
```

In addition to learning that the encryption process utilized the PKCS#7 padding scheme (14), we can see that the server accepts “messages” in the form of a hex string of the IV and ciphertext concatenated together, and reports back whether it is a “valid” message:

```
Gimme a message: 7dc4885cc38ef6fd0181b78f8f484873f4a60fdec121df64f0dca2abcb8cb82507b20ee6da804da255714b7b588329b807b5dc63fa498063600da8eb0e343cf56e086336
3330e8de6e4c22dc2a8dcbb6febf36f164293b89e015a34c6572a80083f210969b1d8d4529ce0bd0fd0786e4e035696cee347097f3081912524d0a2eea26e3a33902c37bf94a5507e1e337c59
4ee326bfcdfbd2a70c6b8c5d8c78ca609515616514a25ac21ad2168cd53f01ad402d352c0deaa8d14f406c4448f151f0e945d7aaa7b708ebd0094d54025ac3b870781e49c9f21bdd4cdfa895e
c5c151bcbcfd76256ee2ec1c000f4bff1907fc1c4d8cbad87a6fa3e9e82bdb38fc3dbea51e0a74b517c87a30689f7a4757c70f6dd7f4f510dd2d8ed84e158d2d3e67831f0fd9a13350c49eabc
fe0a6a4a29c80500527bcf243e1a80ea533b67d82ef29e6ddabae38a23ee6d56060e259a5f00b3c13377301a77fd488e5bbe8af04f986f7e375c1e692f65c8380a78032d88a6bfdbe602a1797
1e6c4004629db6e9601cebe6dd4435283eabca39b2d7510929d63ff30fa3228839db0a7b14b42e22d314277e5fa728e03ed075af49bb00b27e9377b71e7ec475c20f4f7aeb3aea50e0685485a
27c53789d87234cf4ba0ca4d53490eadc7f17a3924023483692473e1dbf0679c1834b4b80ec7237f9380fbe26b3ab1e9c13a514e3431e852e56fd0bb44c8c9ac7d45c8cfb208ddac4be7c41ef
4879112c81ea7a3353d85b8ff153e4a26078ee472a3cf4551c3b07b0fa35909c222c6b6ffa34dce603101303117 79ba1368e5db101804a4577512b10640a121551ead575ee2aa0243a476e83c
a261f6ce4b3f0fa1d30e3822e71434e0d9bfb28e636e745d11d650f5debc937d15df213e98cbfa0442817 4a02dab3556f0d605293e23df46123adb2c6ca4aeeb1399cbf970fda409fc77c3595
0d728c0db5e65693a02039948988be4f37b0c69cc69fac5c5a5d96092912fca0c3e36ce1a5046f9da0dc794fcb3453326ef66afc3f27c70610a4302fb4002fda516c95ecf70135023e25b8c5b
a2db82adff9303fd1e76ce7b0d1ce3c9ef5b69ba791e87a60cec554b08e28f4392482581865 3080e845c334ce60b4cd94886fc57e09fc2c2c8291505bead2142c3d868ae35fd634f623ea4c67
251952ed375efad15fa2a443c0178f78421bdec84c091e5a8cbdc3d1542cf2a22585e4032dbab343f416b7fe3d9c49032f3ee1d073bcaa31908c3e619507401a0cfaf6847045625c92ed4037a
0927bae583ff79f9c0db7c3f3c859b116a38b49426f9548701ab715cddcbebc2383a3429ce9100e53ca3bf025901e1c3bdc86
 yep, that's a valid message
```

By this point, given this week’s content slides, we can conclude that we are dealing with the AES Cipher Block Chaining (CBC) mode encryption algorithm, using PKCS#7 padding, with the CTFd server acting as a padding oracle. Thus, we are performing a padding oracle attack (15).

To perform the attack, we create a python script *cbcsolver.py*. Note, because there is no shortage of literature on the details of the padding oracle attack, we have chosen not to explicitly explain the attack itself within this write-up. Instead, we will step

through our solver script that implements the attack as it is described elsewhere (15, 16). First, as with the RSA challenges above, we read the IV and ciphertext in from *ciphertext.txt* and store them separately as strings, bytes objects, and lists of individual bytes. All of which are individual variables. Additionally, we store a copy of the list of ciphertext bytes to retain those values after we make changes to the list.

Using the “pwn tools” python library we create a remote session to the CTFd server to exchange messages back and forth. Namely, we will send the IV/ciphertext combo and receive confirmation of whether it includes valid padding or not. Next, we define three additional variables, *solved = b””*, *plaintext_full = []*, and *intermed_full = b””*. Finally, we reach the most important part of our attack script:

```
# loop to iterate backwards over every element in c_bytes, a list of every 16-byte block in the ciphertext
for n in range(-2, -len(c_bytes)-1, -1):
        intermed = b""
        plaintext = b""

        original_block = original_c_bytes[n]

        # loop to iterate through each byte in the 16-byte block
        for m in range(1, 17):

                # iterate through every possible single byte value until the oracle reports a valid padding
                for i in range(0x00, 0xff+1):

                        new_block = rand[:-m] + bytes([i]) + bytes(z ^ m for z in intermed[::-1])

                        c_bytes[n] = new_block

                        short_c_bytes = c_bytes[:n + 2]
                        if (n + 2) == 0:
                                new_cipher = b"".join(c_bytes)
                        else:
                                new_cipher = b"".join(short_c_bytes)

                        temp_msg = bytes(iv, "utf-8") + bytes(new_cipher.hex(), "utf-8") + b"\n"
                        s.send(temp_msg)

                        out = s.recvuntil(b"Gimme a message:")
                        if b"valid message" in out:
                                solved += bytes([i])
                                intermed += bytes([i ^ m])
                                plaintext += bytes([(i ^ m) ^ original_block[-m]])
                                break

        # append the intermediate values and plaintext for each block once computed
        intermed_full += intermed
        plaintext_full.append(plaintext[::-1])
```

Three nested for loops, the outermost loop iterating backwards through each 16-byte block within the list of ciphertext blocks, the middle loop iterating through each individual byte in the current ciphertext block, and the innermost loop iterating through every possible value between 0x00 and 0xff - a single byte. This innermost loop is where we are brute-forcing, one byte at a time, until the oracle reports a valid padding. More

specifically, in the case of brute-forcing the final byte of the final block of plaintext, we replace the second to last block of ciphertext with 15 random bytes followed by the brute-force byte (0x00-0xff). In standard decryption, this value would be XORed against the output of the block cipher decryption process of the final block, but in our case, because we “know” the last byte of the plaintext here is 0x01 (15, 16, 17), we can solve for the “intermediate value”: ***intermediate[-1] = edited_ciphertext[-17] XOR 0x01***. Note the -17 index points to the last byte of the second-to-last block. We do this in our code with *intermed += bytes([i ^ m])* where *i* is the brute-forced value as confirmed by the oracle, and *m* is the current byte-in-block offset, in this case “1”. From there, we can easily recover the plaintext by XORing this intermediate value with the original byte from the ciphertext - the one in the position of the value that we are brute-forcing: ***plaintext[-1] = intermediate[-1] XOR original_ciphertext[-17]***. We perform this process in our script with *plaintext += bytes([(i ^ m) ^ original_block[-m]])* where *i* and *m* remain the same as described above, and *original_block[-m]* is the original ciphertext byte of the same position in the proceeding block. This process is repeated for each remaining byte in the block. However, note that as we solve for bytes, during the next iteration, we must XOR the solved intermediate bytes against the current block offset and append them onto our edited bytes for the payload, done with this code:

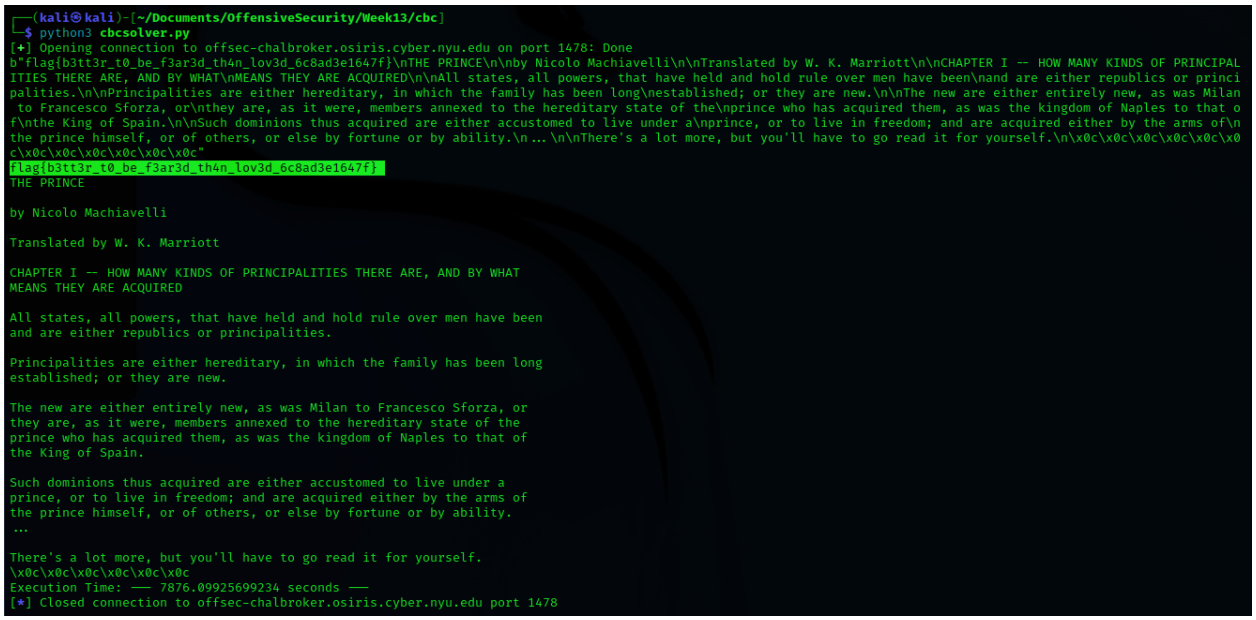
```
new_block = rand[:-m] + bytes([i]) + bytes(z ^ m for z in intermed[::-1])
```

In the case of the next iteration following the one solved above, our edited block payload becomes 14 random bytes + the byte we are brute-forcing + the intermediate value of the previous byte XORed with 0x02. Then, the next will be 13 random bytes + the byte we are brute-forcing + both intermediate values of the previous bytes XORed with 0x03, and so on.

This process is repeated for every byte in the block. Once a block of plaintext is solved, we remove the corresponding block of ciphertext from our payload and repeat the process again for the next block. On the final iteration, when solving for the final block of plaintext, the IV is used instead of the current_ciphertext_block - 1. We accounted for this detail by appending the IV to the ciphertext list that we are iterating through. So, while it is technically a different process in concept, programmatically our

script just treats the IV as another ciphertext block. This strategy did work and reduced the need for extra if statements within our innermost loop.

Finally, once all plaintext blocks have been solved, we combine them all into a single bytes object and print it to the screen:



As seen in the image above, in addition to an excerpt from Machiavelli's "The Prince," the plaintext includes our flag, indicating that we have successfully completed the challenge.

References

1. *https://crypto.stackexchange.com/questions/6713/low-public-exponent-attack-for-rsa*
2. *https://github.com/RsaCtfTool/RsaCtfTool*
3. *https://docs.python.org/3/library/subprocess.html*
4. *http://www.math.umd.edu/~immortal/MATH406/lecturenotes/ch8-Additional.pdf*
5. *https://www.youtube.com/watch?v=uX2z4fZYYkQ*
6. *https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm*
7. *https://www.cuemath.com/numbers/coprime-numbers/*
8. *https://docs.python.org/3/library/math.html*
9. *https://pypi.org/project/libnum/*
10. *https://github.com/hellman/libnum*
11. *https://pycryptodome.readthedocs.io/en/latest/src/util/util.html*
12. *https://docs.python.org/3/library/stdtypes.html*
13. *https://www.techtarget.com/whatis/definition/initialization-vector-IV*
14. *https://medium.com/asecuritysite-when-bob-met-alice/so-what-is-pkcs-7-daf8f4423fd1*
15. *https://robertheaton.com/2013/07/29/padding-oracle-attack/*
16. *https://book.hacktricks.xyz/crypto-and-stego/padding-oracle-priv*
17. *https://www.youtube.com/watch?v=4EgD4PEatA8*
18.