Pico CTF 24 - rsa_oracle

AUTHOR
Mike (Rem)
GCIA, GCIH, GSEC | Founder, Vipyr Security

Introduction

rsa_oracle was a challenge in PicoCTF 2024 in the cryptography category. As the name suggests, you are challenged with abusing an rsa oracle to decode a password and decrypt a ciphertext.

The password is encoded *with the private key* and the oracle will not decrypt the password. The ciphertext is AES encrypted, and the password contains the key to that file.

We're given a few hints here before starting:

1. Cryptography Threat models: chosen plaintext attack.
2. OpenSSL can be used to decrypt the message. e.g. `openssl enc -aes-256-cbc -d ...`
3. The key to getting the flag is by sending a custom message to the server by taking advantage of the RSA encryption algorithm.
4. Minimum requirements for a useful cryptosystem is CPA security.

Let's unpack these a bit. We're actually just handed our attack vector here, a chosen plaintext attack on the RSA oracle. We also know that the ciphertext is encrypted with AES-256 in CBC mode. Not wildly useful, but saves us some steps at the end. We also know that we need to send a message to the oracle to derive some information about the RSA algorithm.

Solving

Research

RSA encryption is very public and well published. With a little bit of research, we can determine the following: Given a Message M, Ciphertext C is determined by: `C = M^e mod N` I am not a math guy. I will never claim to be. So there's a lot going on here that I really do not understand. Fortunately, we can actually just kinda... `c = pow(m, e, n)` Oh, that's Python, and it's very easy. As with all things in Math, these have some specific names. Specifically, e is the public exponent, and n is the modulus. It is typically an enormously long (2048 bit) prime number, derived from multiplying two smaller prime numbers together.

Through my research, I also found out that a common public exponent is used, 65537. So with all that in hand, we actually have enough to solve this now.

Cracking RSA

```
┌──(rem λ redmint)-[~/CTF/Pico/rsa oracle]
└─$ nc titan.picoctf.net 62026
*****************************************
****************THE ORACLE***************
*****************************************
what should we do for you?
E --> encrypt D --> decrypt.
e
enter text to encrypt (encoded length must be less than keysize): 1
1

encoded cleartext as Hex m: 31

ciphertext (m ^ e mod n) 4374671741411819653095065203638363839705760144
```

We can actually just start to plug in some information here. M is our message, in this case we sent `1`, which is `0x31`. That value was raised to the **public exponent**, and we're going to start with 65537 here.

We can always check our assumptions when we derive an N value by using the formula `pow(m, e, n)` and our target here is the n value, for a step we'll use later. I found a snippet of code to simplify this.

```python
def recover_n(pairings, e):
    pt1, ct1 = pairings[0]
    N = ct1 - pow(pt1, e)

    # loop through and find common divisors
    for pt,ct in pairings:
        val = gmpy2.mpz(ct - pow(pt, e))
        N = gmpy2.gcd(val, N)
return N
```

The above function has been slightly modified for brevity, but the original function, found online, took multiple pairings to ensure a discrete value was being obtained. If we pass in our pairing, which is our plaintext `0x30` and the ciphertext we received, we can derive the modulus with that formula.

RSA encryption is based on modular exponentiation, which involves raising a plaintext message to a certain power (the public exponent) and then taking the remainder when divided by a large modulus.

The provided function aims to recover this modulus by exploiting a property of RSA encryption. When a plaintext message is encrypted using RSA, the difference between the ciphertext and the plaintext raised to the power of the public exponent shares a common factor with the modulus.

The function takes pairs of plaintext and ciphertext values and calculates this difference for each pair. By finding the greatest common divisor (GCD) among these differences, the function effectively identifies a factor of the modulus. Since the modulus is the product of two prime numbers, finding just one factor is enough to reveal the modulus itself.

So with that information in hand (and some writing liberties from ChatGPT to explain this better than I think I attempted to the first few times I wrote this) we should be able to derive the modulus N.

```
pow(0x30, 65537, 550759845235642222575519402088087645258846354344599522

>>> 4374671741411819653095065203638363839705760144524191633605358134684
```

And that is our original ciphertext! Alright so, what do we actually do with this? Well, I found a nice little Github repository called RSHack which contains a particularly useful tool for us. However, it's about 7 years old, so it didn't quite work out of the box.

```
# RSA Chosen Plaintext Attack
# Zweisamkeit - zweisamkeit.fr
# 07/05/17
# GNU GPL v3

import codecs

class Chopla(object):
  def __init__(self, n, e, c):
    c_bis = c * pow(2,e,n) % n
    print("\t[*] Please send the following ciphertext to the server: {}
    out = int(input("\t[*] What's the result? "))
    p = out // 2
    print("\t[+] The plaintext is: {}\n".format(p))

    try:
      p_text = codecs.decode(hex(p)[2:].replace('L',''), "hex_codec").d

      print("\n\t[+] The interpreted plaintext: {}\n".format(p_text))

    except:
      print("\t[-] The plaintext is not interpretable\n")
```

We see some very familiar variables here. We can pass the public exponent, modulus, and ciphertext to Zweisamkeit's RSHack with this module. Since we have all the information from how a ciphertext is generated, we can actually just step through here with given values. If we pass in a value of 2 instead of our ciphertext in the RSA formula, we can factor out the ciphertext contents later by sending a custom message that will result in the ciphertext simply being multiplied by two. We then divide that by two, and we're given our plaintext.

Plugging in our information, we are given a message to send to the rsa oracle. When we receive our response, we need to pass back the **base 10** or decimal converted value back to RSHack.

```
[*] Please send the following ciphertext to the server: 241054478552663
[*] What's the result? 491317127782
[+] The plaintext is: 245658563891
[+] The interpreted plaintext: 92d53
```

We can then use this plaintext key to decrypt our file.

```
┌──(rem λ redmint)-[~/CTF/Pico/rsa oracle]
└─$ openssl enc -aes-256-cbc -d -in secret.enc
enter AES-256-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

Summary

There's still a lot of math I don't really understand about RSA. But it's old, and it's well documented enough that stumbling blindly through it can often produce the intended results. I think in the event that my initial public exponent hadn't worked, I probably would've beat the server up a bit with some public exponent values, and maybe rewrap RSHack and the modulus derivation formulas into a script with Pwntools to automate the entire process.

Even writing this article has taught me a little bit as I check assumptions, and dig a bit more into the code that was previously just a means to an end.