

p4-team / ctf

Watch78

★ Star296

🍴 Fork63

<> Code

🔔 Issues 0

🔗 Pull requests 0

📁 Projects 0

📈 Pulse

📊 Graphs

Branch: master ▾

ctf / 2015-10-18-hitcon / crypto_314_rsabin /






Create new file

Find file

History

 Pharisaeus more typos in rsabin

Latest commit 9c7422c on Oct 19 2015

..		
 README.md	more typos in rsabin	2 years ago
 cipher.txt	Add crypto 314 rsabin	2 years ago
 rsa_rsabin.py	Added missing description for rsabin and eng version	2 years ago
 rsabin.py	Add crypto 314 rsabin	2 years ago
 rsabin.tgz	Add crypto 314 rsabin	2 years ago

 README.md

Rsabin (crypro, 314p, 37 solves)

Classical things?

[rsabin-a51076632142e074947b08396bf35ab2.tgz](#)

PL

ENG

Z dołączonego do zadania pliku tgz wypakowujemy dwa kolejne - [rsabin.py](#) (kod szyfrujący coś) oraz [cipher.txt](#) (wynik wykonania go).

Kod szyfrujący wygląda tak:

```
from Crypto.Util.number import *

p = getPrime(157)
q = getPrime(157)
n = p * q
e = 31415926535897932384

flag = open('flag').read().strip()
assert len(flag) == 50

m = int(flag.encode('hex'), 16)
c = pow(m, e, n)
print 'n =', n
print 'e =', e
print 'c =', c
```

A dane tak:

```
n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307
e = 31415926535897932384
c = 19103602508342401901122269279664114182748999577286972038123073823905007006697188423804611222902
```

N jest na tyle małe, że można go sfaktoryzować (albo znaleźć w factordb):

```
p = 123722643358410276082662590855480232574295213977
q = n / p
```

Na pierwszy rzut oka szyfrowanie wygląda na klasyczne RSA. Uwagę przykuwa jednak dość nietypowy wybór eksponenty szyfrującej e . Jej dalsza analiza pozwala stwierdzić, że całe szyfrowanie jest wykonane niepoprawnie, ponieważ eksponenta nie spełnia założenia $\gcd(e, (p-1)(q-1)) = 1$. Brak tego założenia sprawia, że RSA nie może poprawnie zdekodować zaszyfrowanej wiadomości. Faktoryzacja e pozwala stwierdzić że ma ona 2^5 w swoim rozkładzie na czynniki i z tego powodu ma wspólne dzielniki z $(p-1)(q-1)$. Musimy się więc pozbyć tego czynnika z wykładnika. Kilka godzin czytania na temat RSA oraz arytmetyki modularnej pozwala nam dojść do wniosku, że możemy wykorzystać Algorytm Rabina (stąd też zapewne gra słów w tytule zadania) aby pozbyć się niechcianego czynnika z eksponenty.

RSA koduje dane poprzez:

```
cipher_text = (plain_text^e)%n
```

Algorytm Rabina szyfruje jako:

```
cipher_text = (plain_text^2)%n
```

W naszym przypadku możemy przyjąć, że $e = e' \cdot 32$ i uzyskujemy dzięki temu:

```
cipher_text = (plain_text^e)%n = (plain_text^(e'*32))%n
```

Teraz jeśli oznaczymy $x = \text{plain_text}^{e' \cdot 16}$ to nasze równanie będzie miało postać $\text{cipher_text} = (x^2)\%n$ czyli dokładnie postać znaną z Algorytmu Rabina! Oznacza to, że dekodując nasz zaszyfrowany tekst Algorytmem Rabina możemy uzyskać potencjalne wartości x . Warto wspomnieć, że takich potencjalnych wartości będzie 4 bo algorytm jest niejednoznaczny.

Możemy zaprezentowaną metodę stosować wielokrotnie, a każde zastosowanie szyfrowania z Algorytmu Rabina spowoduje "usunięcie" jednego czynnika 2 z wykładnika. To oznacza, że jednokrotne deszyfrowanie da nam możliwe wartości $\text{plain_text}^{e' \cdot 16}$, kolejne $\text{plain_text}^{e' \cdot 8}$, ..., a piąte $\text{plain_text}^{e'}$.

Widzimy więc, że po pięciu deszyfrowaniach uzyskamy wreszcie postać którą będziemy mogli odszyfrować za pomocą RSA, pamiętając przy tym, że nasza eksponenta uległa zmianie i wynosi teraz $31415926535897932384/32$.

Wykonujemy więc:

```
n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307L
p = 123722643358410276082662590855480232574295213977L
q = n / p
e = 31415926535897932384L
e_p = e / 32
ct = 19103602508342401901122269279664114182748999577286972038123073823905007006697188423804611222902
d = get_d(p, n, e_p)

rabin = Rabin(p, q)
partially_decoded_ct = [ct]
for i in range(5):
    new_partially_decoded_ct = []
    for ct_p in partially_decoded_ct:
        new_ct_p = rabin.decrypt(ct_p)
        new_partially_decoded_ct.extend(list(new_ct_p))
    partially_decoded_ct = set(new_partially_decoded_ct)

potential_plaintext = []
for potential_rsa_ct in partially_decoded_ct:
    pt = pow(potential_rsa_ct, d, n)
    potential_plaintext.append(pt)
print(potential_plaintext)
```

Kompletny kod dla tego kroku jest dostępny [tutaj](#).

Teoretycznie każde deszyfrowanie Rabina mogło dać nam 4 różne potencjalne plaintexty, w praktyce niektóre są identyczne więc ich liczba finalnie wyniosła 16.

Tak więc otrzymujemy w końcu listę liczb które mogły potencjalnie być szyfrowaną wiadomością. Nie jest ona specjalnie długa:

```
11781957604393222865231495052284116318876440682267206215409582095778432861874348660845251848045
4169292882246487226436372571580328445408188970955313275707356349635909107039805208587333631745
1170873348295885335059944818034561278496937179514286352764111586578035361653036340797178495797
13871573946024796279189581177591269513969694950673020202114872377044854770083045515434824811804
6441791373850850303735177662669226594971314531797606586938114159564488393181507111460739720503
19142491971579761247864814022225934830444072302956340436288874950031307801611516286098386036510
16144072437629159356488386268680167663532820511515313513345630186973434056224747418308230900562
```

```
8531407715482423717693263787976379790064568800203420573643404440830910301390203966050312684262
1055425717878104375386098355306535487232069922237396940514031160338383741028199997840854113277
13756126315607015319515734714863243722704827693396130789864791950805203149458209172478500429284
11591134129841181208716021381503690146413855638482506249018420423689972012921602773678731165798
3978469407694445569920898900799902272945603927170613309316194677547448258087059321420812949498
1633489591218120101300385993946059383599540555300013479736791859061894905177493305474751582809
8722231190034465374208737458756805962527153843988120540034566112919371150342949853216833366509
6557239004268631263409024125397252386236181789074495999188194585804140013806343454417064103023
19257939601997542207538660484953960621708939560233229848538955376270959422236352629054710419030
```

Więc co pozostaje? Tylko odczytać je jako tekst i uzyskać flagę? Jest mały problem - w źródle widzimy:

```
assert len(flag) == 50
```

A jednocześnie `len(long_to_bytes(n))` jest równe 40 - to oznacza, że nasza szyfrowana wiadomość o długości 50 bajtów jest przycinana przez modulo. Co z tym zrobić? Jedyne co się da to bruteforce - ale niestety, 10 bajtów to dużo za dużo na jakikolwiek bruteforce. Na szczęście - wiemy coś o wiadomości. Konkretnie, jak się zaczyna i kończy. Na pewno jest to flaga, więc pierwsze bajty to `hitcon{` a ostatni bajt to `}`.

Rozpiszmy sobie, jak wygląda flaga, oraz kilka naszych pomocniczych zmiennych (o nich później):

```
flag = hitcon{ABCxyzxyzxyzxyzxyzxyzxyzxyzxyz}
const = hitcon{                                     }
brute =      ABC
msg =      xyzxyzxyzxyzxyzxyzxyzxyzxyzxyz
```

Po co taki podział? Otóż znamy `flag % n`, ale niestety, jest to niejednoznaczne (bo `flag` ma 50 bajtów, a `n` 40). Ale gdybyśmy poznali samo "msg", to `msg % n` jest już jednoznaczne (bo jest krótsze niż `n`). A jak go poznać? Na pewno znamy `const` - możemy odjąć je od flagi. Możemy też brutować `brute` - to tylko trzy bajty, i sprawdzać każde możliwe `msg` po kolei.

Piszemy kod łamiący/bruteforcujący flagę:

```
def crack_flag(ct, n):
    n_len = 40
    flag_len = 50
    const = int('hitcon{'.encode('hex'), 16) * (256**(flag_len - 7))
    for a in range(32, 128):
        for b in range(32, 128):
            for c in range(32, 127):
                brute = (a * 256 * 256 + b * 256 + c) * (256**(flag_len - 10))
                flag = (ct - const - brute) % n
                flag = (flag - ord('{')) * modinv(256, n) % n
                flagstr = long_to_bytes(flag)
                if all(32 <= ord(c) <= 128 for c in flagstr):
                    print chr(a) + chr(b) + chr(c) + flagstr

n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307

for msg in possible_inputs:
    print 'testing', msg
    crack_flag(msg, n)
```

Po uruchomieniu go, ostatecznie dostajemy flagę w swoje ręce:

```
Congratz~~! Let's eat an apple pi <3.14159
```

Pamiętamy żeby dopisać obciętą stałą część:

```
hitcon{Congratz~~! Let's eat an apple pi <3.14159}
```

Jesteśmy 314 punktów do przodu.

ENG version

From the task archive we extract two files: [rsabin.py](#) (cipher code) and [cipher.txt](#) (results of the cipher code).

Cipher code is:

```

from Crypto.Util.number import *

p = getPrime(157)
q = getPrime(157)
n = p * q
e = 31415926535897932384

flag = open('flag').read().strip()
assert len(flag) == 50

m = int(flag.encode('hex'), 16)
c = pow(m, e, n)
print 'n =', n
print 'e =', e
print 'c =', c

```

And the result data:

```

n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307
e = 31415926535897932384
c = 19103602508342401901122269279664114182748999577286972038123073823905007006697188423804611222902

```

N is small enough to factor (or check in factordb):

```

p = 123722643358410276082662590855480232574295213977
q = n / p

```

At the first glance the cipher resembles classical RSA. What catches attention is an unusual choice of cipher exponent e . Exponent analysis leads us to conclusion that the whole cipher is *incorrect* as RSA since the exponent violates the assumption $\gcd(e, (p-1)(q-1)) = 1$. Lack of this property means that RSA cannot correctly decrypt the message simply by applying the decryption exponent d .

Factorization of e shows us that it has 2^5 as factor and therefore it has common divisors with $(p-1)(q-1)$. We need to get rid of this factor from exponent. Few hours reading about RSA and modular arithmetics leads us to conclusion that we can use Rabin Algorithm (this is probably the reason for word play in the task title) to remove the unwanted part of exponent.

RSA encodes the data by:

```
cipher_text = (plain_text^e)%n
```

Rabin Algorithm does this by:

```
cipher_text = (plain_text^2)%n
```

If we introduce a new variable e' such that $e = e' * 32$ we get:

```
cipher_text = (plain_text^e)%n = (plain_text^(e'*32))%n
```

Now if we introduce a new variable x such that $x = \text{plain_text}^{e'*16}$ then our equation will be $\text{cipher_text} = (x^2)\%n$ which is exactly the ciphertext formula from Rabin Algorithm! This means that we can use decrypt function from Rabin Algorithm on our ciphertext and get potential x values. It's worth noting that there will be 4 potential values since this algorithm is ambiguous.

We can use the presented method multiple times and each Rabin decryption run will "remove" a 2 factor from exponent. This means that single decryption will give us potential values of $\text{plain_text}^{e'*16}$, next one $\text{plain_text}^{e'*8}$, ..., and the fifth will give $\text{plain_text}^{e'}$.

We can see that after five consecutive decryptions we will finally get a ciphertext that can be decoded by RSA. We need to keep in mind here that the exponent for RSA is now changed because we removed the 32 so it is now:

```
31415926535897932384/32
```

We execute the code:

```

n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307L
p = 123722643358410276082662590855480232574295213977L
q = n / p
e = 31415926535897932384L
e_p = e / 32

```

```

ct = 19103602508342401901122269279664114182748999577286972038123073823905007006697188423804611222902
d = get_d(p, n, e_p)

rabin = Rabin(p, q)
partially_decoded_ct = [ct]
for i in range(5):
    new_partially_decoded_ct = []
    for ct_p in partially_decoded_ct:
        new_ct_p = rabin.decrypt(ct_p)
        new_partially_decoded_ct.extend(list(new_ct_p))
    partially_decoded_ct = set(new_partially_decoded_ct)

potential_plaintext = []
for potential_rsa_ct in partially_decoded_ct:
    pt = pow(potential_rsa_ct, d, n)
    potential_plaintext.append(pt)
print(potential_plaintext)

```

Complete code for this step is available [here](#).

Theoretically each Rabin decryption could give 4 different potential plaintexts, however in reality a lot of them were identical so finally there were only 16 to check.

So we end up with a list of potential plaintext values. It's not very long:

```

11781957604393222865231495052284116318876440682267206215409582095778432861874348660845251848045
4169292882246487226436372571580328445408188970955313275707356349635909107039805208587333631745
1170873348295885335059944818034561278496937179514286352764111586578035361653036340797178495797
13871573946024796279189581177591269513969694950673020202114872377044854770083045515434824811804
6441791373850850303735177662669226594971314531797606586938114159564488393181507111460739720503
19142491971579761247864814022225934830444072302956340436288874950031307801611516286098386036510
16144072437629159356488386268680167663532820511515313513345630186973434056224747418308230900562
8531407715482423717693263787976379790064568800203420573643404440830910301390203966050312684262
1055425717878104375386098355306535487232069922237396940514031160338383741028199997840854113277
13756126315607015319515734714863243722704827693396130789864791950805203149458209172478500429284
11591134129841181208716021381503690146413855638482506249018420423689972012921602773678731165798
3978469407694445569920898900799902272945603927170613309316194677547448258087059321420812949498
16334895912181201013003859939460593835995405555300013479736791859061894905177493305474751582809
8722231190034465374208737458756805962527153843988120540034566112919371150342949853216833366509
6557239004268631263409024125397252386236181789074495999188194585804140013806343454417064103023
19257939601997542207538660484953960621708939560233229848538955376270959422236352629054710419030

```

So what is left? Just read the values as text? Unfortunately there is a problem - in the cipher code we have:

```
assert len(flag) == 50
```

But at the same time `len(long_to_bytes(n))` equals 40 - this means that our message was 50 bytes long and got "cut" by modulo operation. What can we do? The only way is a brute-force - but brute-forcing 10 bytes is a lot. Fortunately we know some things about the message. We know the prefix and suffix. It has to be the flag therefore it starts with `hitcon{` and ends with `}`.

Let's start with drawing how the flag and our additional variables look like:

```

flag = hitcon{ABCxyzxyzxyzxyzxyzxyzxyzxyzxyz}
const = hitcon{
brute =      ABC
msg      =      xyzxyzxyzxyzxyzxyzxyzxyzxyz

```

Why this division? We know the value of `flag % n` but it is ambiguous (since `flag` has 50 bytes and `n` has 40). However we know only "msg" part then `msg % n` is not ambiguous (since it is shorter than `n`). But how to get it? We know `const` so we can cut this out of the flag. We can also brute-force the `brute` part - this is only 3 bytes, and check every possible `msg`.

We write the code to crack the flag:

```

def crack_flag(ct, n):
    n_len = 40
    flag_len = 50
    const = int('hitcon{'.encode('hex'), 16) * (256**(flag_len - 7))
    for a in range(32, 128):
        for b in range(32, 128):
            for c in range(32, 127):

```

```

brute = (a * 256 * 256 + b * 256 + c) * (256**(flag_len - 10))
flag = (ct - const - brute) % n
flag = (flag - ord('}')) * modinv(256, n) % n
flagstr = long_to_bytes(flag)
if all(32 <= ord(c) <= 128 for c in flagstr):
    print chr(a) + chr(b) + chr(c) + flagstr

n = 20313365319875646582924758840260496108941009482470626789052986536609343163264552626895564532307

for msg in possible_inputs:
    print 'testing', msg
    crack_flag(msg, n)

```

After we run it we finally get the flag from one of ciphertxts:

Congratz~~! Let's eat an apple pi <3.14159

We also remember that we removed the const part:

hitcon{Congratz~~! Let's eat an apple pi <3.14159}

And we have +314 points.

