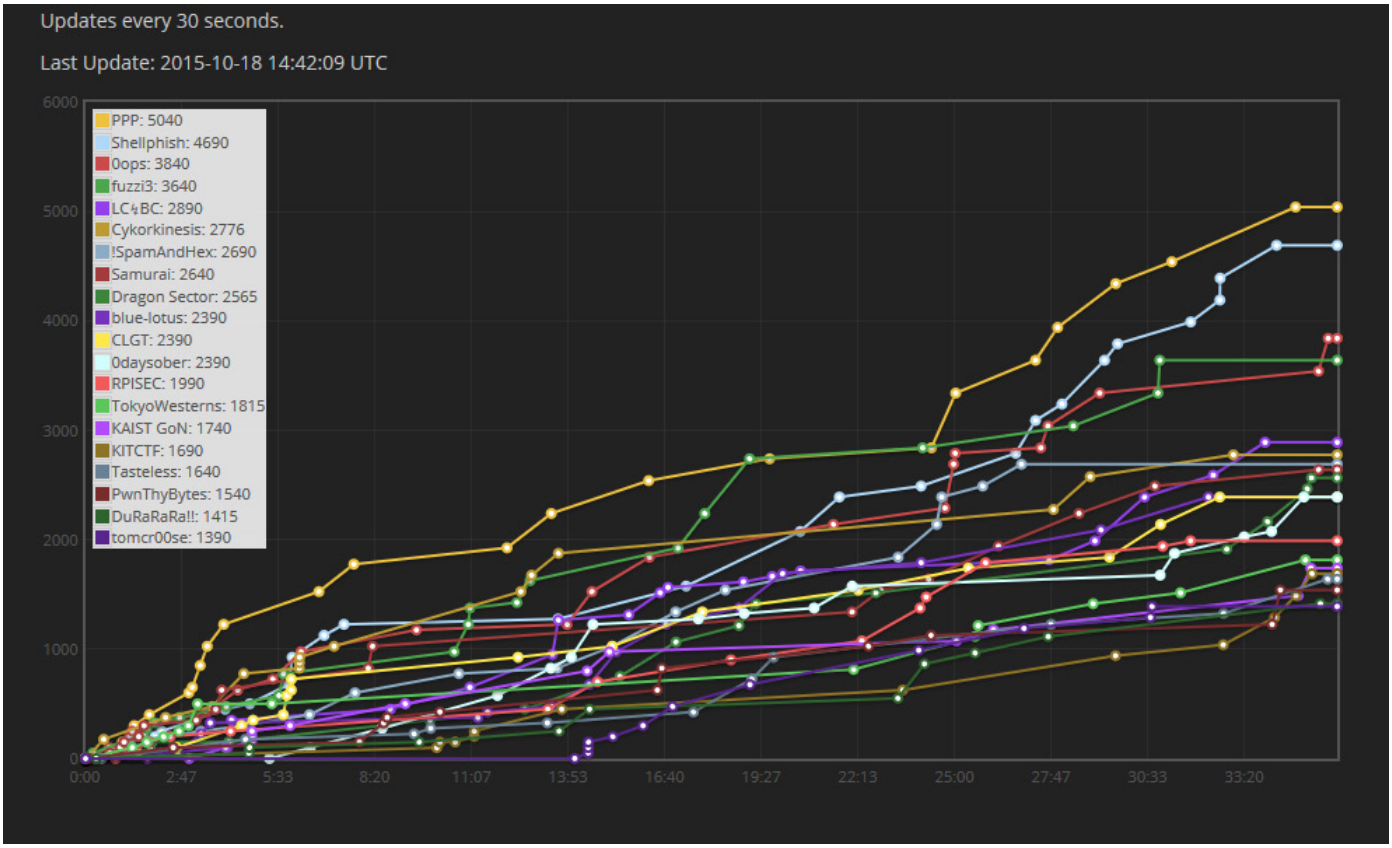
HITCON CTF 2015 – Rsabin

HITCON to tajwańska konferencja poświęcona szeroko pojętej dziedzinie bezpieczeństwa informatycznego, a jednocześnie CTF odbywający się w trakcie konferencji. Jest ona organizowana przez czołowy zespół o tej samej nazwie. Przed właściwymi zawodami odbywają się kwalifikacje w Internecie, w których nagrodą dla najlepszych zespołów jest możliwość wzięcia udziału w finałach w Tajlandii. Podczas tegorocznych kwalifikacji zwyciężył zespół PPP, polska drużyna Dragon Sector zajęła 9., a nasza ekipa 22. miejsce.



CTF	HITCON CTF 2015 http://ctf.hitcon.org/
Waga CTFtime.org	50 (https://ctftime.org/event/245)
Liczba drużyn (z niezerową liczbą punktów)	382
System punktacji zadań	Od 1 punktu (bardzo proste) do 500 punktów (trudne).
Liczba zadań	32
Podium	1. PPP (Stany Zjednoczone) – 5040 pkt. 2. Shellphish (Stany Zjednoczone) – 4690 pkt. 3. 0ops (Chiny) – 3840 pkt.
Zadanie	Rsabin (Crypto 314)

O ZADANIU

Wybrane zadanie, które chcielibyśmy przedstawić, pochodzi z dziedziny współczesnej kryptografii matematycznej. W przeciwieństwie do historycznych szyfrów nie operujemy tutaj na znakach alfabetu, tylko na liczbowej reprezentacji szyfrowanych danych. Same operacje kodowania i dekodowania sprowadzają się do przekształceń z zakresu matematyki dyskretnej.

Jako parametry wejściowe otrzymaliśmy zaszyfrowany ciąg oraz kod źródłowy użytego szyfru, od którego rozpoczniemy naszą analizę:

```
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
```

Kod rozpoczyna się od wylosowania dwóch 157-bitowych liczb pierwszych, które następnie są mnożone, a wynik mnożenia przypisany jest do zmiennej n. Następnie utworzona zostaje zmienna e, której przypisana zostaje pewna duża liczba. Jej bazą w tym przypadku jest liczba `pi`.

Zapraszamy na autorskie szkolenia
z zakresu bezpieczeństwa IT

{ Bezpieczeństwo aplikacji WWW }

{ Offensive HTML, SVG, CSS and other Browser-Evil }

{ Wprowadzenie do bezpieczeństwa IT }

{ Szkolenie przygotowujące do egzaminu CEH }
(Certified Ethical Hacker)

www.securitum.pl/oferta/szkolenia

Patroni medialni: sekurak.pl rozwal.to

Kolejnym krokiem jest wczytanie flagi i sprawdzenie, czy ma na pewno 50 znaków.

Najpierw flaga przekształcana jest do postaci liczbowej. Konkretnie każdy znak tekstu jest zamieniany na odpowiadający mu bajt, a uzyskany ciąg binarny jest traktowany jako jedna wielka liczba. Dopiero później wykonane zostaje właściwe szyfrowanie.

Szyfrowanie polega na potęgowaniu modularnym: tekst do zaszyfrowania zostaje podniesiony do potęgi o wykładniku e, a reszta z dzielenia wyniku tej operacji przez liczbę n stanowi zaszyfrowane dane:

$$ciphertext = plaintext^{e} \ (mod\ n)$$

Osoby mające pewne doświadczenie z popularnymi algorytmami kryptografii asymetrycznej powinny w tym momencie rozpoznać schemat szyfrowania RSA. Niemniej o ile sama operacja kodowania danych jest tożsama z tą wykorzystywaną w RSA, nie oznacza to jeszcze, że właśnie z tym algorytmem mamy do czynienia. Istnieją pewne dodatkowe warunki, które muszą zostać spełnione, aby RSA działało w sposób poprawny. W celu ich weryfikacji musimy przyjrzeć się bliżej danym, na których operował otrzymany algorytm:

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

Znamy liczbę n, czyli iloczyn dwóch 157-bitowych liczb pierwszych, jak również liczbę e, czyli wykładnik szyfrowania, a także liczbę c, czyli zaszyfrowaną flagę. Nasze e jest dobrane dość nietypowo, gdyż, ze względu na pewne szczególne własności, zwykle jest to liczba 65537.

Zakładamy chwilowo, że faktycznie mamy do czynienia z algorytmem RSA, więc naszym pierwszym krokiem jest faktoryzacja liczby n w celu uzyskania liczb pierwszych użytych w algorytmie.

Liczba n jest na tyle mała, że można dokonać jej faktoryzacji, np. za pomocą narzędzia yafu, albo znaleźć w serwisie factordb.com:

```
p = 123722643358410276082662590855480232574295213977
q = 164184701914508585475304431352949988726937945291
```

Istnieją dwa podstawowe ograniczenia nałożone przez algorytm RSA:

1. Funkcja Eulera (tocjent) $\varphi(n)$ musi być względnie pierwsza z wykładnikiem szyfrowania e:

$$\varphi(n) = \varphi(p)\,\varphi(q) = (p-1)(q-1)$$
$$gcd(\varphi(n), e) = 1$$

Gdzie *gcd* oznacza funkcję greatest common divisor, czyli największy wspólny dzielnik.

2. Liczba n, względem której liczymy operację reszty z dzielenia, musi być odpowiednio duża: musi mieć przynajmniej tyle bajtów, ile wiadomość, którą szyfrujemy.

Niestety, w opisywanym zadaniu nie jest spełniony żaden z podanych warunków. Największy wspólny dzielnik funkcji Eulera oraz wykładnika szyfrowania to $2^4 = 16$, ponieważ liczba e ma w swoim rozkładzie na czynniki pierwsze $2^5 = 32$.

Dodatkowo liczba n ma zaledwie niecałe 40 bajtów, podczas gdy z kodu wynika, że flaga ma dokładnie 50 bajtów.

Z naszej krótkiej analizy wynika, że nie jest to klasyczne RSA i nie możemy wprost zastosować deszyfrowania przy pomocy tego algorytmu w celu uzyskania flagi.

RSA I KRYPTOSYSTEM RABINA

Problemem brakujących bajtów zajmiemy się na końcu, a rozpoczniemy od pokonania samego szyfru. Zanim przejdziemy do przedstawienia algorytmu łamania szyfru, przytoczymy pewne definicje, które będą niezbędne do zrozumienia zastosowanych później metod.

Jak wspomnieliśmy w poprzednim paragrafie, operacja szyfrowania algorytmem RSA to:

$$ciphertext = plainttexte \ (mod\ n)$$

Parę liczb `(e,n)` nazywamy kluczem publicznym. Taki klucz możemy udostępnić osobom, z którymi chcemy się komunikować. Przy odpowiednio dobranych parametrach RSA nie ma potrzeby chronić tego klucza: może być publicznie dostępny. Dane zaszyfrowane za jego pomocą mogą zostać odczytane tylko przez posiadacza klucza prywatnego.

Operacja dekodowania wygląda identycznie jak kodowanie, z tą różnicą, że wykładnikiem jest liczba d nazywana też wykładnikiem deszyfrującym. Liczba d musi spełniać zależność:

$$d \cdot e \equiv 1 \ mod\ (\varphi\ (n))$$

dzięki czemu: $(plaintext^{e})^{d}\ (mod\ n) = plaintext$

gdzie: $plaintext^{e}\ (mod\ n) = ciphertext$

więc: $ciphertext^{d}\ (mod\ n) = plaintext$

Wartość wykładnika d możemy wyliczyć bezpośrednio, stosując Rozszerzony Algorytm Euklidesa egcd przy założeniu, że znamy rozkład liczby n na czynniki pierwsze.

Parę liczb `(d, n)` nazywamy kluczem prywatnym i powinniśmy go chronić i nie udostępniać nikomu. Każdy posiadacz klucza prywatnego może deszyfrować wiadomości zakodowane kluczem publicznym.

Warto zauważyć, że liczby pierwsze wybrane przez autorów zadania nie są dobrymi kandydatami na parametry algorytmu RSA, ponieważ są za małe. Bezpieczeństwo RSA opiera się o trudny obliczeniowo problem faktoryzacji liczb, ale dla odpowiednio małych danych można z powodzeniem rozkładać liczby na czynniki pierwsze. W związku z tym dla małych wartości p oraz q liczba n jest na tyle mała, że można ją faktoryzować. W ten sposób można uzyskać wykładnik d, a tym samym klucz prywatny, mając do dyspozycji jedynie klucz publiczny. W związku z tym o ile sam algorytm RSA jest bezpieczny, niepoprawne dobranie rozmiaru klucza może skutkować jego złamaniem.

Niestety jak zauważyliśmy podczas analizy danych dla algorytmu, w naszym przypadku liczba $\varphi(n)$ nie jest względnie pierwsza z e, więc nie istnieje liczba d, którą moglibyśmy wykorzystać do deszyfrowania. Jednak jeśli zapiszemy naszą operację szyfrowana w nieco inny sposób, możemy zauważyć dość ciekawą własność. Przyjmijmy na chwilę oznaczenie $e = e' \cdot 2^5$. Wtedy szyfrowanie możemy zapisać jako:

$$ciphertext = plaintext^{e}\ (mod\ n) = plaintext^{e' \cdot 2^5}\ (mod\ n)$$

Jednocześnie liczba `e'` nie posiada w swoim rozkładzie żadnych 2, więc jest względnie pierwsza z $\varphi(n)$ i dla `e'` istnieje liczba d, która potrafiłaby dokonać operacji deszyfrowania wiadomości `plaintext'` zaszyfrowanej jako:

$$ciphertext = plaintext'^{\,e'}\ (mod\ n)$$

Niemniej wynik deszyfrowania dałby nam jedynie wartość:

$$plaintext' = plaintext^{2^5}\ (mod\ n)$$

Z pomocą przychodzi nam tutaj kryptosystem Rabina, który jest podobny do RSA, aczkolwiek znacznie mniej popularny. Stąd też zapewne nazwa zadania – Rsabin = RSA + Rabin. Kryptosystem Rabina także szyfruje dane za pomocą

modularnego potęgowania, jednak w jego przypadku wykładnikiem potęgi zawsze jest liczba 2. Oznacza to, że algorytm Rabina wykonuje operację:

$$ciphertext = plaintext^2 \pmod n$$

Operacja deszyfrowania opiera się na dość złożonej operacji pierwiastkowania modularnego. Dodatkowo jest to operacja niejednoznaczna, ponieważ pierwiastkowanie może dać nam cztery potencjalne rozwiązania. Wynika z tego, że stosując algorytm Rabina, musimy być w stanie jednoznacznie odróżnić poprawnie zdeszyfrowane dane od pozostałych, błędnych rozwiązań, które po zaszyfrowaniu dają identyczny wynik (`ciphertext`).

SPOSÓB NA ZŁAMANIE SZYFRU

Oznaczmy na chwilę wprowadzony wcześniej `plaintext'` jako:

$$plaintext' = plaintext^{2^5} \pmod n = (plaintext^{2^4})^2 \pmod n$$

A $plaintext^{2^4} \pmod n$ oznaczmy jako `plaintext''` i uzyskujemy wtedy:

$$plaintext' = plaintext''^2 \pmod n$$

Czyli postać identyczną jak w algorytmie Rabina! Wracając do pierwotnych oznaczeń, mamy:

$ciphertext = plaintext^{e' \cdot 2^5} \pmod n$
i jeśli wprowadzimy oznaczenie: $x = plaintext^{e' \cdot 2^5} \pmod n$
uzyskujemy: $ciphertext = x^2 \pmod n$

Równanie tej postaci możemy z powodzeniem deszyfrować za pomocą modularnego pierwiastkowania z algorytmu Rabina. Jednokrotne pierwiastkowanie da nam cztery potencjalne wartości x. Jak nietrudno zauważyć, możemy zastosować identyczne rozumowanie, aby pozbyć się kolejnych 2 z wykładnika. Mając potencjalne wartości dla:

$x = plaintext^{e' \cdot 2^4} \pmod n$
możemy oznaczyć: $x' = plaintext^{e' \cdot 2^3} \pmod n$
uzyskując: $x = x'^2 \pmod n$

które znów możemy pierwiastkować. Każdorazowe pierwiastkowanie usuwa jedną 2 z wykładnika, więc po wykonaniu operacji pierwiastkowania 5 razy uzyskamy potencjalne wartości dla $plaintext^{e'} \pmod n$, które potrafimy zdekodować za pomocą klasycznego RSA.

Każda operacja pierwiastkowania może dać nam do 4 możliwych wartości, więc pesymistycznie uzyskamy $4^5 = (2^2)^5 = 2^{10} = 1024$ wartości do zdekodowania za pomocą RSA.

Musimy koniecznie pamiętać, że wykładnik szyfrujący e uległ zmianie i będziemy liczyć d dla wykładnika $e' = e/2^5 = e/32 = 981747704246810387$

IMPLEMENTACJA

Pierwszym krokiem łamania szyfru jest przeprowadzenie pięciokrotnego pierwiastkowania lub, jak można to inaczej rozumieć – pięciokrotnego deszyfrowania algorytmu Rabina:

```
rabin = Rabin(p, q)
partially_decoded_ciphertexts = [ciphertext]
for i in range(5):
  new_partially_decoded_ciphertexts = []
  for ciphertext_partial in partially_decoded_ciphertexts:
    new_ciphertexts_partial = rabin.decrypt(ciphertext_partial)
    new_partially_decoded_ciphertexts.
    extend(list(new_ciphertexts_partial))
  partially_decoded_ciphertexts = set(new_partially_decoded_ciphertexts)
```

Zaczynamy algorytm z jednym zaszyfrowanym ciągiem – odczytanym bezpośrednio z danych do zadania. Następnie pięciokrotnie dla każdego zaszyfrowanego ciągu wykonujemy deszyfrowanie algorytmem Rabina. Wszystkie wyniki tego kroku stanowią listę zaszyfrowanych ciągów dla kolejnej iteracji.

Ciągi przechowujemy w zbiorze, ponieważ nie wszystkie muszą być unikalne, a nie ma sensu wielokrotnie pierwiastkować tych samych danych.

Drugim krokiem jest przeprowadzenie dekodowania wszystkich uzyskanych w poprzednim kroku ciągów za pomocą RSA:

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

Gdzie wykładnik deszyfrujący d został wyliczony jako:

```
d = egcd(e_prim, (p-1)*(q-1))
```

W wyniku wykonania powyższego kodu uzyskujemy 16 potencjalnych wartości dla tekstu jawnego.

ODZYSKIWANIE BRAKUJĄCYCH BAJTÓW

Tak więc mamy 16 liczb, z których 15 to nieprawidłowe rezultaty, wynikające z niejednoznaczności modularnego pierwiastkowania. Wiemy również, że jedna z nich jest zdeszyfrowaną flagą. Oznaczmy tę liczbę jako `M`. Wyrażając to jako pseudokod, wiemy, że zachodzi następująca zależność:

```
M = rsaDecrypt(rsaEncrypt(flag));
```

W ten sposób zdeszyfrowaliśmy flagę, ale czy to już koniec zadania?

Niestety, twórcy zadania przygotowali na nas jeszcze jedną pułapkę. W RSA wszystkie operacje są wykonywane modulo pewne n. Natomiast n, które zostało użyte do zaszyfrowania flagi, ma niecałe 40 bajtów. Jednocześnie w dostarczonym kodzie widzimy:

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

To bardzo bezpośrednia wiadomość od autorów zadania: flaga ma 50 bajtów. Tak więc `M` wcale nie jest flagą – jest flagą modulo n, czyli w zapisie matematycznym:

$$flag \equiv M \pmod n$$

Co możemy w tym momencie zrobić? Możemy spróbować zgadnąć za pomocą metody bruteforce brakujące bajty, przekształcając poprzednie równanie modularne (z definicji) do takiej postaci:

$flag = M + n \cdot k$ (dla pewnego k będącego liczbą całkowitą)

Czyli istnieje takie k, że $flag = M + n \cdot k$

Naiwny bruteforce wyglądałby tak:

```
def is_ascii_string(string):
  return all(32 <= ord(c) <= 128 for c in string)

def string_to_long(string):
  return int(string.encode('hex'), 16)

def naive_bruteforce(ct, n):
  k = 0
  while True:
    possible_flag = ct + n * k
    flag_string = long_to_bytes(possible_flag)
    if is_ascii_string(flag_string):
      print possible_flag
```

Wykorzystujemy tutaj fakt, że flaga zmieniona na tekst składa się z samych drukowalnych znaków ascii (sprawdza to funkcja `is_ascii_string`).

Niestety, pojawiają się dwa duże problemy z taką funkcją:

- wartości do sprawdzenia jest tak dużo, że prosta heurystyka `is_ascii_string` da bardzo wiele false-positive'ów.
- wartości do sprawdzenia jest o wiele za dużo jak na bruteforce – 10 bajtów to 256^10 = 2^80 możliwości.

Na szczęście wiemy o fladze coś jeszcze, a mianowicie, że zaczyna się od 'hitcon{' , a kończy na '}'. Jeśli będziemy w stanie wykorzystać tę wiedzę, zredukujemy przestrzeń poszukiwań z dziesięciu do dwóch bajtów. Wystarczy wtedy bruteforce'ować trzy bajty (trzy zamiast dwóch, bo n brakuje kilku bitów do równych 40 bajtów, więc musimy je też bruteforce'ować).

Okazuje się, że można dość łatwo odciąć ten „prefiks" i „sufiks" flagi za pomocą odrobiny arytmetyki modularnej. Spróbujmy najpierw poeksperymentować na prostszym przykładzie, dla takich oto danych:

```
plaintext = 732926 # oryginalna wiadomość
n = 543 # modulo
m = plaintext % n # 419
prefiks = 73
sufiks = 26
```

(każda litera reprezentuje dowolną cyfrę, konkretne liczby podstawimy później).

W tym przypadku znamy prefiks (73) tekstu jawnego oraz jego sufiks (26). Nie znamy natomiast „środka" (29), i to do niego chcemy się dostać za pomocą operacji matematycznych.

Zacznijmy od prostszej części, czyli usuwania prefiksu z wiadomości. Jak zrobilibyśmy w przypadku znanych liczb? Np. jak pozbyć się pierwszych dwóch cyfr z liczby 123456? Oczywiście – wystarczy odjąć 120000. Analogicznie dla naszych danych odjąć 730000:

$$m \equiv plaintext \pmod n$$
$$m \equiv 732926 \pmod n$$
$$30000\ m - 7 \equiv 732926 - 730000 \pmod n$$
$$m - 730000 \equiv 2926 \pmod n$$

Ponieważ `m` jest dane, a 730000 możemy wyliczyć z prefiksu (który jest dany), jesteśmy też w stanie policzyć prawą część równości. Świetnie, potrafimy już pozbyć się prefiksu. Co z sufiksem? Bardzo podobnie – odejmujemy go, a następnie dzielimy wynik przez odpowiednią potęgę dziesiątki, żeby usunąć zera na końcu:

$$m \equiv plaintext \pmod n$$
$$m \equiv 732926 \pmod n$$
$$(m - 26)/100 \equiv (732926 - 26)/100 \pmod n$$
$$(m - 26)/100 \equiv 7329 \pmod n$$

Mając sposób, wystarczy teraz zaimplementować go w Pythonie:

```
def string_to_long(string):
  return int(string.encode('hex'), 16)

def cut_prefix(num, flag_len, prefix, m):
  return (num - string_to_long(prefix)
    * 256**(flag_len - len(prefix))) % m

def cut_suffix(num, suffix, m):
  return ((num - string_to_long(suffix))
    * modinv(256**len(suffix), m)) % m
```

Kod jest trochę bardziej zagmatwany niż wzory powyżej – bo działa na liczbach zapisanych w podstawie 256, zamiast znanego nam systemu dziesiętnego (bajty to inaczej liczby zapisane w podstawie 256), oraz zamiast dzielić bezpośrednio, musimy mnożyć przez odwrotność modularną.

Przykład działania:

```
>>> n = 2**50-1
>>> flag = string_to_long('flag{test}') % n
>>> long_to_bytes(flag)
'\x03{t\x7f\x0e\x8c\xd6'   # flaga mod n - śmieci
>>> flag_no_prefix = cut_prefix(flag, 10, 'flag{', n)
>>> long_to_bytes(flag_no_prefix)
'test}'
```

W tym momencie możemy połączyć wszystkie kroki i zaimplementować bruteforce dla ostatecznej flagi:

```
import itertools

def bruteforce_flag(ct, n):
  flag_len = 50
  printable = map(chr, range(32, 128))
  # zgadujemy trzy bajty
  for c1, c2, c3 in itertools.product(printable, repeat=3):
    # obcinamy prefiks
    flag = cut_prefix(ct, flag_len, 'hitcon{'+c1+c2+c3, n)
    # obcinamy sufiks
    flag = cut_suffix(flag, '}', n)
    # zamieniamy liczbę na tekst
    flag_string = long_to_bytes(flag)
    # i jeśli jest printowalnym napisem...
    if is_ascii_string(flag_string):
      # to może być flaga - wypisujemy
      print c1+c2+c3+flag_string
```

Pozostaje przetestować wszystkie możliwości na flagę:

```
for ciphertext in possible_flags:
  bruteforce_flag(ciphertext, n)
```

Liczenie tego zajęło na naszym komputerze jakieś 30 sekund na literę, więc już po kilku minutach otrzymaliśmy na ekranie flagę:

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

Trzeba dodać do niej jeszcze prefiks i sufiks, których wcześniej się pozbyliśmy:

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

PODSUMOWANIE

Zadanie, które opisaliśmy, było naszym zdaniem jednym z ciekawszych problemów z dziedziny kryptografii w ostatnim czasie. Kluczem do rozwiązania zadania było zaobserwowanie związku łączącego algorytmy RSA i Rabina oraz to, jak je wykorzystać w postawionym problemie, w czym pomógł trochę tytuł zadania. Pod koniec musieliśmy za to odkurzyć trochę arytmetykę modularną, żeby móc efektywnie bruteforce'ować brakujący fragment flagi.

Jarosław Jedynak, Stanisław Podgórski

O drużynie

Rozwiązanie zadania Rsabin zostało nadesłane przez p4, zespół CTFowy, który sprawia, że nawet smoki drżą (twierdzą, że to ze śmiechu, ale my swoje wiemy), a korzenie chowają się głęboko w ziemi.

W połowie listopada tego roku zespół p4 połączył się z Amber Chamber – innym wysoko plasowanym polskim zespołem.

▶ https://ctftime.org/team/5152