Midnight Sun CTF 2018 – Badchair

16 czerwca tego roku odbyła się pierwsza edycja konkursu Midnight Sun CTF[1], w którym uczestniczyliśmy razem z kilkoma innymi członkami p4. Został on zorganizowany wspólnie przez KTH (Królewski Instytut Technologiczny w Sztokholmie), szwedzki zespół HackingForSoju oraz firmę SaaB. Termin oraz nazwa CTFa nie jest przypadkowa. Midnight Sun[2] to zjawisko naturalne, które można zaobserwować w letnich miesiącach w krajach leżących blisko koła podbiegunowego. Jak można się domyślić, polega ono na tym, że o północy słońce zamiast być za horyzontem znajduje się ciągle na niebie. I faktycznie, dokładnie to stało się podczas trwania CTFa – w teorii. W praktyce rzeczywiście noc była bardzo krótka, a północ jasna, co znacznie ułatwiło nam wytrwanie całej nocy pełnej łamania zadań.

1. https://www.midnightsunctf.se/
2. https://en.wikipedia.org/wiki/Midnight_sun

CTF	Midnight Sun CTF 2018 https://www.midnightsunctf.se/
Waga CTFtime.org	25 (https://ctftime.org/event/635)
Liczba drużyn (z niezerową liczbą punktów)	14
System punktacji zadań	Od 5 punktów (bardzo proste) do 468 punktów
Liczba zadań	26
Podium	1. LC/BC (Rosja) – 6687 pkt. 2. scryptos (Japonia) – 6660 pkt. 3. TokyoWesterns (Japonia) – 5239 pkt.
Zadanie	Badchair (Crypto, 161 pkt)

O ZADANIU

Zadanie *Badchair* jest ciekawe, ponieważ prawie identyczny problem pojawił się na dwóch różnych CTFach w prawie identycznej postaci. Pierwszy raz zmierzyliśmy się z nim na WCTF w Chinach, gdzie występowało pod dużo bardziej kontrowersyjną nazwą (jako „gestapo"). Niestety, nie opublikowaliśmy wtedy nigdzie naszego rozwiązania dla tego zadania. Żałowaliśmy tego, kiedy pojawiło się drugi raz, bo musieliśmy przypomnieć sobie niektóre kroki, jakie podjęliśmy.

W zadaniu dostajemy zaszyfrowaną w ciekawy sposób flagę:

```
{
  "split": [
    "jUEumBCZY6GRlXbB/uobM53gis/RldnMAfBAnkg=",
    "e3WhwYy6YUQGedMXkGnxJO6v0ov4cgteapL17wI="
  ], "shares": 9, "threshold": 5
}
{
  "split": [
    "mrygQzFoasgDY23te4MGqTFXjpS/pMalQiN9Sks=",
    "XjpXSuBzyfRAbKj7hODzcf0cv0NZsXQDEQiIdtA="
  ], "shares": 9, "threshold": 5
}
{
  "split": [
    "U64tceO4Ddtr4V6FMXSTJre4f6t4nPczWgtIkfo=",
    "t3hrQmsqonoBCl7T4f3bLqMShchtOtF3WesMGEs="
  ], "shares": 9, "threshold": 5
}
{
  "split": [
    "TcOzpm1jNtwsj0cdiLfd6oVHLGMMKRt52t9kU4E=",
    "RqJ1WKlufadMFwFLbIjvBs82BH/yP8CAT6kyv3M="
  ], "shares": 9, "threshold": 5
}
```

Wraz z kodem, który został użyty do szyfrowania:

```
class KeySplitter:
    def __init__(self, numshares, threshold):
        self.splitter = Shamir(numshares, threshold)
        self.numshares = numshares
        self.threshold = threshold

    def split(self, key):
        xshares = [''] * self.numshares
        yshares = [''] * self.numshares
        for char in key:
            xcords, ycords = self.splitter.split(ord(char))
            for idx in range(self.numshares):
                xshares[idx] += chr(xcords[idx])
                yshares[idx] += chr(ycords[idx])
        return zip(xshares, yshares)

    def jsonify(self, shares, threshold, split):
        data = {
            'shares': shares,
            'threshold': threshold,
            'split': [
                base64.b64encode(split[0]),
                base64.b64encode(split[1])
            ]
        }
        return json.dumps(data)

if __name__ == "__main__":
    splitter = KeySplitter(9, 5)
    splits = splitter.split(FLAG)
    for i in range(0, 4):
        print splitter.jsonify(9, 5, splits[i])
```

Po nazwie klasy szyfrującej (`Shamir`) łatwo domyślić się, że zastosowany został algorytm dzielenia sekretów Shamira (ang. *Shamir Secret Sharing*). Jest to algorytm, który pozwala podzielić wiadomość (sekret) na dowolną ilość fragmentów tak, aby dało się ją odzyskać, tylko mając co najmniej „N" z nich. Na przykład możemy podzielić nasz sekret na cztery fragmenty tak, żeby można go było odzyskać, tylko posiadając co najmniej trzy.

Do czego można takie coś zastosować? Jako dość ekstremalny przykład weźmy kody upoważniające do użycia broni nuklearnej w Stanach Zjednoczonych (tzw. Golden Codes). Zarządzenie ataku bronią atomową to dość poważna sprawa i przypadkowe wysłanie rakiet byłoby dość sporą szkodą dla wizerunku kraju. Z tego powodu nawet prezydent nie może zrobić tego samodzielnie – rozkaz musi potwierdzić naczelny dowódca sił zbrojnych. Jest to znacznie bezpieczniejszy system, ale dalej jest jedna rzecz, która może pójść nie tak. Broń atomowa najbardziej przydaje się podczas wojny, a wojny mają to do siebie, że często panuje podczas nich chaos. Co, jeśli zdarzy się, że trzeba pilnie autoryzować atak, ale prezydent lub dowódca sił zbrojnych nie jest dostępny? Rozwiązań jest wiele[3]. Jednym z nich byłoby podzielenie kodów pozwalających na detonację na trzy części i rozdanie ich trzech ważnym osobom – w taki sposób, że do autoryzacji rozkazu wystarczą dwa z nich. Dzięki temu, nawet jeśli jedna z tych osób będzie niedostępna, kraj nie będzie bezbronny.

Jako bardziej przyziemny przykład można podać Facebooka. Pozwala on dostarczyć znajomym „fragmenty", których możemy później użyć do odzyskania zgubionego hasła. W ten sposób żaden fragment z osobna nie wystarczy do resetu, więc nikt nie będzie w stanie samodzielnie włamać się na nasze konto. Ale w razie potrzeby możemy odezwać się do naszych powierników, udowodnić swoją tożsamość i zebrać potrzebną nam ilość fragmentów hasła.

W tym konkretnym przypadku sekret (czyli oczywiście flaga) jest podzielony na 9 fragmentów, a żeby go odzyskać, potrzebujemy co najmniej 5 z nich. Brzmi dobrze, gdzie jest więc problem? No tak – w treści zadania dostajemy tylko cztery z nich. Więc nie ma łatwo, będziemy musieli wykonać jakiś atak zanim zdobędziemy flagę – było to do przewidzenia. A żeby coś złamać, trzeba najpierw zrozumieć, jak działa.

SHAMIR SECRET SHARING

Algorytm dzielenia sekretów Shamira może i brzmi enigmatycznie, ale pomysł nie jest wcale skomplikowany. Opiera się na relatywnie prostej matematyce, którą wszyscy (prawdopodobnie) poznali dobrze w szkole – wielomianach.

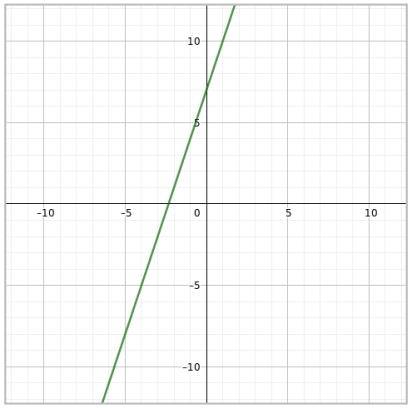
Ustalmy, że naszym sekretem zawsze będzie liczba – co jest z reguły prawdziwe w cyfrowym świecie (ponieważ każdą sekwencję bitów możemy potraktować jako odpowiednio dużą liczbę). Załóżmy na razie dla uproszczenia, że chcemy ją podzielić tylko na dwa fragmenty, z których oba będą potrzebne do odzyskania oryginału. Da się to robić na wiele sposobów (chociażby starą, dobrą, uniwersalną operacją xor), ale wyobraźmy sobie, że zamiast tego przedstawimy naszą liczbę jako prostą. Na przykład, jeśli chcemy „ukryć" liczbę 7, ustalamy prostą:

$$f(x) = a*x + b$$

gdzie $b = 7$, natomiast a jest losową liczbą z wybranego przedziału. Na przykład:

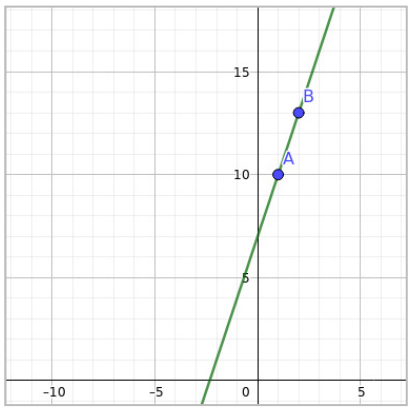
$$f(x) = 3x + 7$$

3. Autor tego artykułu jest przekonany, że departament obrony Stanów Zjednoczonych znalazł lepsze rozwiązanie tego problemu. Mimo wszystko uważa swój przykład za edukacyjny i ciekawy.



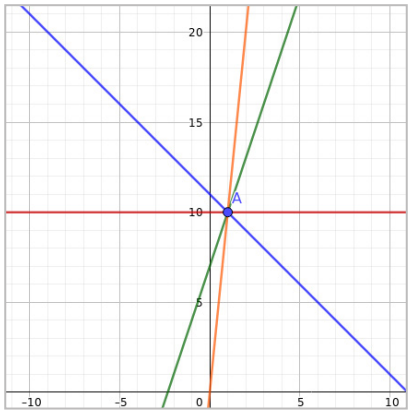
Rysunek 1. Prosta enkodująca nasz sekret. Ten i następne wykresy zostały wygenerowane przy użyciu narzędzia https://www.math10.com

Co nam to daje? Weźmy teraz dwa punkty na tej prostej – na przykład wartości dla 1 i dla 2:



Rysunek 2. Dwa „udziały" naszej prostej.

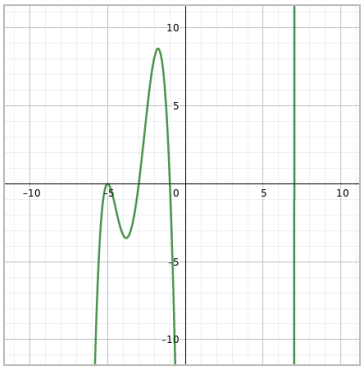
Możemy teraz jeden z tych punktów przekazać naszemu pierwszemu przyjacielowi, a drugi punkt dać drugiemu – oraz wytłumaczyć im, jak zostały wygenerowane. Co nam to da? Otóż mając oba punkty, odzyskanie oryginalnego wzoru jest banalne (przez dowolne dwa punkty przechodzi tylko jedna prosta) – wystarczy rozwiązać układ 2 równań z 2 niewiadomymi. Natomiast mając tylko jeden z nich, jest to niemożliwe – mamy nieskończenie wiele możliwości (patrz Rysunek 3).



Rysunek 3. Mając tylko jeden punkt, nie ma sposobu odgadnąć, która z nieskończoności możliwych prostych jest tą poprawną

Jaki ma to związek z naszym oryginalnym problemem? Okazuje się, że tą samą metodę można wykorzystać przy dowolnym wielomianie, nie tylko prostej. Na przykład dla funkcji (Rysunek 4):

f(x) = -52.5 - 83.5 x - 34.6 x^2 - 3. x^3 + 0.7 x^4 + 0.1 x^5



Rysunek 4. Funkcja f(x)=-52.5 - 83.5 x - 34.6 x^2 - 3.0 x^3 + 0.7 x^4 + 0.1 x^5

możemy wybrać 5 punktów, które jednoznacznie będą ją identyfikowały. Jeśli ktoś pozna tylko 4 punkty, nie będzie w stanie jej odtworzyć. W ogólności do odzyskania wielomianu stopnia k potrzeba nam k+1 punktów. To oczywiście można zastosować bezpośrednio do naszego problemu – jeśli chcemy podzielić sekret na N fragmentów, z których K jest potrzebne do odtworzenia go, generujemy odpowiedni wielomian stopnia K-1 i wybieramy na nim N punktów (tradycyjnie punkty dla x=1, x=2, x=3...). Żeby wrócić z listy punktów do oryginalnego wzoru, wystarczy wykorzystać jeden z wielu algorytmów interpolacji – np. algorytm Lagrange'a (ang. *Lagrange Interpolation*).

Mocną stroną algorytmu jest to, że jest prawdziwie bezpieczny z punktu widzenia teorii informacji (tzw. *Information theoretic security*). Można matematycznie udowodnić (a może intuicyjnie „widać"), że nie mając odpowiedniej ilości udziałów, nie da się go złamać. Niezależnie, jaką mocą obliczeniową dysponują hakerzy, NSA albo obcy – nie odzyskają sekretu, nie mając odpowiedniej ilości jego fragmentów, podobnie jak nie odzyskają prostej, znając tylko jeden punkt.

Inną ciekawą cechą jest minimalność (nie da się stworzyć kodu zapewniającego podobny poziom bezpieczeństwa, ale używającego mniejszej ilości bajtów), dzięki sztuczce przedstawionej w kolejnym podrozdziale. Shamir Secret Sharing jest też bardzo elastyczny, to znaczy można łatwo implementować z jego pomocą bardziej skomplikowane mechanizmy bezpieczeństwa (np. dawanie bardziej zaufanym podmiotom 2 fragmentów zamiast jednego albo generowanie ich dynamicznie w razie potrzeb). Te wszystkie zalety połączone z łatwością i wydajnością implementacji powodują, że w zasadzie nie ma potrzeby szukania alternatywnego algorytmu i algorytm Shamira jest powszechnie stosowany, od kiedy został opracowany w 1979 roku (!)[4], do dzisiaj.

MATEMATYCZNE DETALE TECHNICZNE

Jeśli ktoś nie cierpi matematyki (albo odwrotnie – kocha matematykę i nie może znieść niskiej precyzji moich wyjaśnień), może spokojnie pominąć ten rozdział. Nie jest potrzebny do zrozumienia reszty artykułu – wystarczy wiedzieć, że „liczby" w naszych wielomianach są nie do końca typowe, mimo że zachowują się tak, jakbyśmy się spodziewali.

Jest tylko jeden problem – liczby rzeczywiste nie są zbyt przyjazne dla komputera. Dla programisty znacznie lepiej jest korzystać wyłącznie z liczb naturalnych, które bardziej pasują do rejestrów procesora (wydajność) oraz nie tracą dokładności z czasem (precyzja numeryczna). Rozwiązaniem bardzo popularnym w kryptografii jest użycie jakiegoś innego pierścienia, który ma wymagane własności. Często używaną opcją są liczby modulo N (formalnie pierścień $\mathbb{Z}n$), które są dobrze znane wszystkim, którzy np. mieli do czynienia z RSA. Niestety, mają swoje wady. Na przykład, jeśli chcemy zakodować bajt (czyli liczbę z przedziału 0-255), musimy wykonywać operacje modulo liczba pierwsza większa niż 255[5]. To oznacza, że wynik naszych operacji może nie mieścić się w zakresie bajta, więc trzeba go zapisywać na dwóch bajtach! Brzmi to bardzo… nieoptymalnie.

Da się lepiej. Rozwiązaniem jest grupa nazywana GF(2^n). Jest to bardzo specyficzna grupa – dodawanie w niej jest równoważne operacji... xor!

```
class GF:
    def __init__(self, value):
        self.value = value % 256

    def __add__(self, other):
        return GF(self.value ^ other.value)
```

Jako że xor dowolnych dwóch wartości mieszczących się w bajcie również będzie bajtem, nie mamy tutaj problemu z przekraczaniem wielkości typów prostych. Mnożenie i dzielenie jest trochę bardziej skomplikowane. Wykorzystujemy fakt, że mnożenie można zastąpić operacjami potęgowania i logarytmowania (na przykład *a * b = exp(ln(a) + ln(b))*).

Więc, po wyliczeniu tablic logarytmów i potęg dla ustalonej bazy, możemy je zaimplementować w prosty i wydajny sposób:

```
class GF:
    # ...
    def __mul__(self, other):
        if other.value == 0 or self.value == 0:
            return GF(0)
        return GF(power[(logarithm[self.value] +
logarithm[other.value]) % 255])

    def __div__(self, other):
         if other.value == 0:
             raise ArithmeticError('Division by zero')
        return GF(power[(255 + logarithm[self.value] -
logarithm[other.value]) % 255])
```

Mając taką klasę, możemy używać jej zamiast zwykłych liczb (np. floatów) i będzie zachowywała się zgodnie z naszymi oczekiwaniami (formalnie: będzie ciałem). W szczególności możemy na niej używać klasycznych algorytmów interpolacji, jak np. algorytmu Lagrange'a.

WŁAŚCIWY ATAK

Koniec przerwy. Wiemy już mniej więcej (mam nadzieję), jak działa Shamir Secret Sharing, ale nie jesteśmy ani o krok bliżej do zaatakowania go. Przeciwnie – kilka razy podkreśliłem jego zalety i siłę.

Jak zwykle w takich przypadkach okazuje się, że problemem jest niepoprawne użycie kryptografii, a nie słabość samego algorytmu[6]. I faktycznie, błąd jest, chociaż łatwy do przeoczenia:

```
self.splitter = Shamir(numshares, threshold)
```

4. Shamir, Adi (1979), *How to share a secret*, Communications of the ACM, 22

5. Najmniejsza liczba pierwsza większa niż 255 to 257

6. Złe używanie dobrych algorytmów to, swoją drogą, pewnego rodzaju konik autora. Więcej materiałów na podobny temat można znaleźć na przykład w prezentacji *Jak źle użyć kryptografii* z konferencji SBS 2017 albo artykule *Praktyczna kryptografia* z Programisty 08/2017 (63).

13-14 WRZEŚNIA 2018
WARSZAWA
Hotel Sound Garden

SECURITY CASE 2018 STUDY

V DOROCZNA KONFERENCJA IT SECURITY

SCS EXPO, SCS FREE, ELEVATOR PITCH – Wstęp wolny
ZAREJESTRUJ SIE JUŻ DZIŚ

CZYM JEST KONFERENCJA SCS?

SECURITY CASE STUDY to spotkanie społeczności IT Security przyciągające setki uczestników, 5 ścieżek tematycznych, konkurs CTF (Capture the Flag), specjalistyczne warsztaty oraz EXPO ze stoiskami wiodących firm branży IT Security.

Ścieżki konferencji SCS 2018:

- **SCS PRO** – IT Security, cyberbezpieczeństwo
- **SCS URDI** – informatyka śledcza
- **SCS INFOOPS** – ścieżka dedykowana zagadnieniom operacji informacyjnych w cyberprzestrzeni
- **SCS FREE** – prelekcje, część wystawiennicza i stoiska partnerskie **(część bezpłatna)**
- **ELEVATOR PITCH** czyli otwarta scena, na której będzie można w luźniejszej formie przedstawić swój projekt, produkt czy też ideę **(część bezpłatna)**
- **konkurs CTF (Capture the Flag)**, każda drużyna może spróbować swoich sił, dla trzech pierwszych drużyn nagrody pieniężne **(część bezpłatna)**.

www.securitycasestudy.pl
#SCSconference

Instancja klasy `Shamir` jest tworzona w konstruktorze, a to oznacza, że wszystkie wygenerowane punkty dzielą ten sam wielomian. Zmienia się jedynie wyraz wolny wielomianu (jest równy literze, którą szyfrujemy). Gdyby udało nam się odzyskać ten wielomian, moglibyśmy odzyskać flagę metodą siłową – dla każdego znaku sprawdzać wszystkie 256 możliwości i sprawdzać, kiedy wynik się zgadza (kiedy pozostałe punkty dla danej litery leżą na testowanym wielomianie). Ale jak odzyskać wielomian, skoro potrzebujemy 5 próbek, a mamy tylko 4?

Jak często w CTFach bywa, wiemy coś o zaszyfrowanej wiadomości – konkretnie wiemy, że jest flagą, która przestrzega pewnego ustalonego formatu. W tym przypadku było to `midnight{...}`. Przydatną cechą tego formatu jest to, że znak „i" powtarza się tutaj dwukrotnie.[7] Pomyślmy, jak możemy to wykorzystać...

Wiemy, że wszystkie znaki są szyfrowane prawie identycznym wielomianem – różni się w nim tylko czynnik stały, który jest zależny jedynie od szyfrowanego znaku. A skoro znak „i" powtarza się dwukrotnie, to znaczy, że dla pewnego wielomianu mamy nie 4, a aż 8 fragmentów! W dodatku wcześniej wspominałem, że zazwyczaj przy generowaniu punktów wybiera się po prostu kolejne wartości x (x=1, 2, 3...). Gdyby tak było i w tym przypadku, nasze „i" zaszyfrowałoby się po prostu dwa razy do tego samego. Na szczęście, jeśli popatrzymy w implementację klasy `Shamir`, zauważymy, że autorzy zadania zdecydowali się wybierać losowe wartości:

```
def split(self, secret):
    coeffs = [secret] + self.base_poly
    coords = []
    result = []
    while len(coords) < self.shares:
        drawn = self.rng.randint(1, 255)
        if not drawn in coords:
            coords += [drawn]
    # (...)
```

To oznacza, że (z bardzo dużym prawdopodobieństwem) otrzymaliśmy wartości wielomianu dla więcej niż czterech różnych x-ów. Ponieważ wielomian jest piątego stopnia, to żeby go odzyskać, wystarczy teraz rozwiązać układ pięciu równań i otrzymamy rozwiązanie – albo użyć gotowego algorytmu interpolacji wielomianów i pozwolić na wykonanie tej pracy komputerowi.

Wystarczy tylko zaimplementować ten atak. Zaczynamy od parsowania danych wejściowych:

```
def recover_splits(input_file_data):
    splits = []
    for (s0, s1) in re.findall('"split": \["(.*?)", "(.*?)"\]',
input_file_data):
        splits.append((base64.b64decode(s0), (base64.
b64decode(s1))))
    return splits

def recover_points_per_key_character(data):
    splits = recover_splits(data)
    points_per_char = []
    for point_id in range(len(splits[0][0])):
        points = []
        for idx in range(numshares):
            split = splits[idx]
            points.append((ord(split[0][point_id]),
ord(split[1][point_id])))
        points_per_char.append(points)
    return points_per_char
```

Dzięki temu dostajemy listę punktów dla każdego zakodowanego znaku flagi. Weźmy te, które nas interesują, czyli te odpowiadające powtarzającej się literze „i":

```
[(65, 117), (188, 58), (174, 120), (195, 162)]
[(16, 140), (49, 224), (227, 107), (109, 169)]
```

Jeśli wybierzemy dowolne 5 z tych punktów i wykonamy na nich interpolację, powinniśmy dostać oryginalny wielomian, który pozwoli nam odszyfrować wiadomość i odzyskać flagę!

Na podstawie podanych punktów możemy wykonać interpolację wielomianu stopnia 4, w efekcie czego dostaniemy wielomian 31x4+173x3+111x2+219x+C.

Następnie możemy dla każdego kolejnego znaku zaszyfrowanej flagi przetestować wszystkie możliwości i sprawdzić, dla której z nich pozostałe punkty pasują do wielomianu:

```
def main():
    result = ""
    with codecs.open("shares.txt") as input_data:
        points_per_character = recover_points_per_key_
character(input_data.read())
        print("\n".join(map(str, points_per_character)))
        for points_for_single_char in points_per_character:
            for c in string.printable:
                # 31x^4+173x^3+111x^2+219x+C
                p = Polynomial(
                    [IntegerInRing(ord(c)), IntegerInRing(219),
IntegerInRing(111), IntegerInRing(173),
                     IntegerInRing(31)])
                failed = False
                for x,y in points_for_single_char:
                    if p(IntegerInRing(x)).value != y:
                        failed = True
                        break
                if not failed:
                    result += c
    print(result)
```

I… to tyle – otrzymujemy sensownie odszyfrowaną wiadomość: `midnight{ehhh_n0t_3ven_cl0se}`.

PODSUMOWANIE

Midnight Sun był bardzo ciekawym CTFem o charakterystycznym klimacie i mamy nadzieję, że wrócimy tam jeszcze (może w przyszłym roku?). Jeśli chodzi o zadanie, było ono dość typowe, ale po latach grania cieszymy się zawsze, kiedy widzimy zadanie z kryptografii polegające na czymś innym niż łamanie RSA lub AES po raz setny. A skoro pojawiło się już dwa razy, to może się pojawić i trzeci – więc dla swojego i innych CTFowiczów dobra postanowiliśmy w końcu je opisać.

Jarosław Jedynak

Rozwiązanie zadania *Badchair* zostało nadesłane przez p4, polski zespół CTFowy, który w chwili redagowania tego artykułu zajmował 5. miejsce w generalnej klasyfikacji CTFTime.org. https://ctftime.org/team/5152

7. Kiedy zadanie pierwszy raz się pojawiło na WCTF, format flagi nie miał tej własności. Trzeba było wykonać dodatkowy brute-force w celu znalezienia takiej pary, która oczywiście „zupełnie przypadkowo" występowała dalej w tekście flagi. Nie utrudniało to zadania, za to dodawało kolejny krok – kolejny powód, żeby użyć zadania w wersji z Midnight Sun CTF.