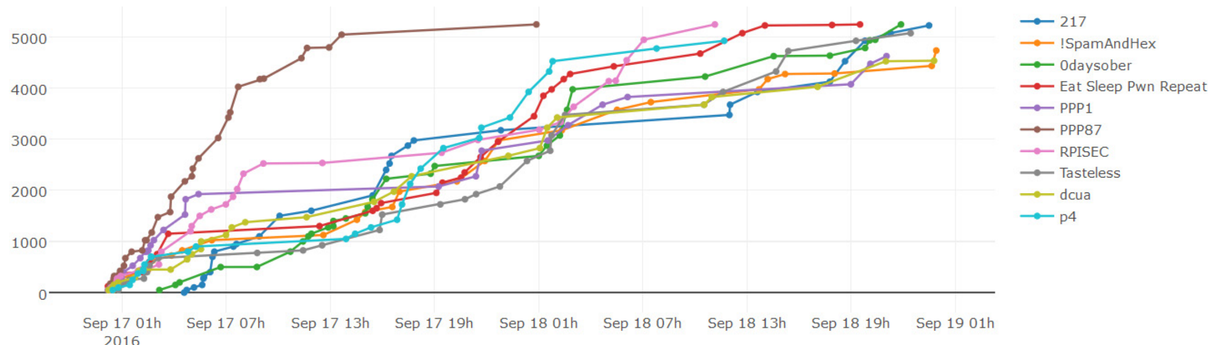
CSAW 2016 Quals – (Still) Broken Box

CSAW to jedne z największych zawodów CTF odbywających się online, organizowane w ramach konferencji Computer Security Awareness Week na Uniwersytecie Nowojorskim. W tym roku udział wzięły 1274 ekipy, zwyciężyła amerykańska drużyna PPP, a my zakończyliśmy zmagania na pozycji 7., będąc tym samym najwyżej sklasyfikowanym polskim zespołem. Postanowiliśmy podzielić się z czytelnikami rozwiązaniami dwóch interesujących zadań z dziedziny kryptografii asymetrycznej. Są one o tyle ciekawe, że sam algorytm został bezbłędnie zaimplementowany, a wektor ataku stanowiły symulowane błędy sprzętowe, możliwe do wywołania także w realnej sytuacji.



CTF	CSAW 2016 Quals: https://ctf.csaw.io/
Waga CTFtime.org	24,58 (https://ctftime.org/event/347)
Liczba drużyn (z niezerową liczbą punktów)	1274
System punktacji zadań	Od 10 punktów (proste) do 500 punktów (trudne)
Liczba zadań	31
Podium	1. PPP (Stany Zjednoczone) – 5246 pkt. 3. RPISEC (Stany Zjednoczone) – 5246 pkt. 3. Eat Sleep Pwn Repeat (Niemcy) – 5246 pkt.
Zadanie	Broken Box (Crypto 300p) + Still Broken Box (Crypto 400p)

O ZADANIU

Opisujemy dwa zadania, ponieważ Still Broken Box jest „rozszerzoną wersją" zadania Broken Box – konieczne było wykonanie najpierw ataku analogicznego do tego dla Broken Box, warto więc zacząć od wyjaśnienia najpierw tego zadania.

Oba problemy należały do dziedziny kryptografii asymetrycznej i jedyną dostępną informacją był adres serwera, z którym należało się komunikować. Opis zadania jasno stwierdzał, że mamy do czynienia z aplikacją generującą podpisy kryptograficzne za pomocą algorytmu RSA, ale z powodu wadliwego sprzętu zwracane wyniki nie zawsze są poprawne.

Brak kodu źródłowego aplikacji szyfrującej sugerował, że podatność nie jest związana z niepoprawną implementacją, a informacja o problemach sprzętowych przywodzi na myśl klasę ataków znanych jako „fault attacks". Są to ataki oparte o przekłamania w danych, jak na przykład zmiana losowego bitu na przeciwną wartość w trakcie wykonywania obliczeń. Oprócz oczywistego skutku, jakim jest niepoprawny końcowy wynik obliczeń, może to także otwierać luki związane z bezpieczeństwem.

Warto mieć na uwadze, że sytuacja, o której mówimy, nie jest zupełnie abstrakcyjna. W roku 2014 opisany został atak „Row hammer", który prezentował metodę zmiany stanu wybranego bitu za pomocą wielokrotnych odczytów z pamięci RAM, co wynika z fizycznej niedoskonałości układów pamięci. Dodatkowo ataki tego typu są najskuteczniejszą metodą eksploitowania urządzeń kryptograficznych – błędy mogą być wtedy wprowadzane na przykład przez używanie lasera skierowanego na odpowiedni fragment układu scalonego lub przez przegrzewanie urządzenia.

ANALIZA DZIAŁANIA APLIKACJI SERWEROWEJ

Działanie serwera było dość proste: po nawiązaniu połączenia zostajemy poproszeni o podanie liczby z zakresu 0-9999, a następnie w odpowiedzi dostajemy podpis RSA dla tej liczby. W trakcie jednego połączenia możemy podpisywać wiele liczb. Ograniczenie na rozmiar podpisywanej liczby wynika z faktu, że otrzymujemy również zaszyfrowane flagi, a operacja podpisywania jest równoważna z deszyfrowaniem. Brak tego ograniczenia oznaczałby, że moglibyśmy wysłać zaszyfrowaną flagę do podpisu i dostać w odpowiedzi odszyfrowaną wartość.

Ponieważ otrzymujemy w zadaniach zaszyfrowane flagi, jasne jest, że celem jest ich odszyfrowanie, najpewniej poprzez poznanie klucza prywatnego.

Serwer podaje nam informacje o wykładniku szyfrującym e, a do każdego podpisu dodawana jest informacja o modulusie n, który ma 1024 bity.

Można zaobserwować dziwne zachowanie serwera po kilkukrotnym wysłaniu tej samej liczby do podpisania – uzyskany podpis nie zawsze jest taki sam. Co więcej, skoro znamy wykładnik szyfrujący e oraz modulus n, możemy odtworzyć klucz publiczny RSA, a co za tym idzie – zweryfikować podpis. Faktycznie czasami podpis jest niepoprawny.

```
Input a number(0~9999) to be signed:2
signature:22611972523744021864587913335128267927131958989869
436027132656215690137049354670157725347739806657939727131080
334523442608301044203758495053729468914668456929675330095440
863887793747492226635650004672037267053895026217814873840360
359669071507380945368109861731705751166864109227011643600107
409036145468092331,
N:1727946914720528916061230268738049088280416696916095758792
188391033127255755392745101460723149725951035142052664177604
253990219241012130434760749467877970270009465943520738299757
800015003657745534884709672613074283664614334415941966304948
342606530222380455408393001904446860460168943563837490669664
16917513737
Sign more items?(yes, no):yes
Input a number(0~9999) to be signed:2
signature:15263446719776629494452837846238921844187158344383
434551949637332903242235983113539049924841524545077178891831
920687050917778404061903941900483472844720619854978355231620
383164645300909460529138845851966062192718333675629775058916
784861191397026401375370180869256855783800260563439133794113
2554468117369960245,
N:1727946914720528916061230268738049088280416696916095758792
188391033127255755392745101460723149725951035142052664177604
253990219241012130434760749467877970270009465943520738299757
800015003657745534884709672613074283664614334415941966304948
342606530222380455408393001904446860460168943563837490669664
16917513737
```

Można wywnioskować z tego, że w trakcie obliczania podpisu kryptograficznego następuje losowa zmiana w danych. Dostajemy jednak za każdym razem wartość modulusa i zawsze jest on poprawny, co oznacza, że przekłamanie wartości musi dotyczyć innego parametru.

Istnieją dwie główne metody podpisywania za pomocą algorytmu RSA:

1. Metoda klasyczna, korzystająca bezpośrednio z jednej operacji potęgowania modularnego.
2. Metoda szybka RSA-CRT, korzystająca z Chińskiego Twierdzenia o Resztach (ang. *Chinese Remainder Theorem*).

Wersja RSA-CRT pozwala przyspieszyć obliczenia, jednocześnie nie zwiększając znacznie poziomu skomplikowania algorytmu. Ma ona jednak pewien mankament – jeśli w trakcie obliczeń wystąpi przekłamanie w danych, podobnie jak w naszym zadaniu, można w dość trywialny sposób, za pomocą Największego Wspólnego Dzielnika (ang. *Greatest Common Divisor*), rozłożyć modulus n na czynniki pierwsze, mając do dyspozycji zaledwie jeden poprawny oraz jeden uszkodzony podpis.

W opisywanym zadaniu metoda ta nie odniosła skutku, co oznacza, że obliczenia realizowane były za pomocą algorytmu klasycznego.

DZIAŁANIE ALGORYTMU RSA

Zanim przejdziemy do wyjaśnienia, w jaki sposób na podstawie wadliwych podpisów można odzyskać klucz prywatny, warto przypomnieć, jak dokładnie działa podpis RSA.

RSA to algorytm kryptografii asymetrycznej, gdzie do szyfrowania i deszyfrowania używa się innych kluczy. Szyfrowanie wymaga posiadania klucza publicznego złożonego z dwóch liczb: e oraz n. Natomiast w deszyfrowaniu używa się klucza prywatnego złożonego z pary liczb: d i n. Operacje matematyczne stojące za szyfrowaniem oraz odszyfrowywaniem wiadomości to odpowiednio:

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

oraz:

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

Algorytm nakłada pewne ograniczenia na potencjalne wartości e, d oraz n:

1. Liczba n, zwana modulusem, powinna być odpowiednio duża oraz trudna w rozkładzie na czynniki pierwsze. Rozmiar tej liczby determinuje długość wiadomości, z którymi można pracować. W klasycznym modelu liczba n jest iloczynem 2 dużych liczb pierwszych p oraz q.
2. Liczba e, zwana wykładnikiem szyfrującym, musi spełniać zależności:

$$\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$$

$$gcd(\varphi(n), e) = 1$$

Gdzie gcd oznacza Największy Wspólny Dzielnik, a funkcja $\varphi(n)$ to Funkcja Eulera (tocjent). Czyli liczba e musi być względnie pierwsza z $\varphi(n)$.

3. Liczba d, zwana wykładnikiem deszyfrującym, musi spełniać zależność:

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

Ze względu na powyższą zależność na mocy twierdzenia Eulera zachodzi równość:

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

W naszym zadaniu nie interesuje nas co prawda operacja szyfrowania, a podpis kryptograficzny, ale w przypadku RSA jest on jednak tożsamy z deszyfrowaniem, natomiast weryfikacja podpisu z szyfrowaniem. Jest tak, ponieważ kolejność potęgowania w przytoczonym wyżej równaniu nie ma znaczenia dla końcowego wyniku.

Nazywamy więc wartość:

$$signature = message^{d}\ (mod\ n)$$

podpisem kryptograficznym wiadomości message. Taki podpis może złożyć jedynie posiadacz klucza prywatnego. Jednocześnie każdy posiadacz klucza publicznego jest w stanie wykonać operację:

$$message = signature^{e}\ (mod\ n)$$

Jeśli uzyskana w ten sposób wiadomość jest zgodna z wiadomością otrzymaną w postaci tekstu jawnego, oznacza to, że tekst jawny nie został zmodyfikowany, a wiadomość została faktycznie wysłana przez posiadacza klucza prywatnego.

Warto zauważyć, że w trakcie obliczania podpisu kryptograficznego pod uwagę brane są jedynie 3 parametry: wiadomość, modulus n oraz wykładnik d. W zadaniu zaobserwowaliśmy już, że modulus n dla każdego podpisu jest jednakowy. Możemy więc domniemywać, że przekłamanie następuje dla wykładnika d, ponieważ przekłamanie dla samej wiadomości w niczym by nam nie pomogło. Możemy to zresztą szybko zweryfikować, o czym szerzej w dalszej części artykułu.

ODZYSKIWANIE KLUCZA Z BŁĘDNYCH PODPISÓW

W tej sekcji skupimy się na wyniku podpisywania danych za pomocą RSA, kiedy wykładnik d jest uszkodzony. Jak wspomnieliśmy wcześniej, podpis jest generowany poprzez operację:

$$signature = message^{d}\ (mod\ n)$$

Rozważmy teraz, co będzie się działo, gdy jeden z bitów liczby d ulegnie przekłamaniu i zmieni wartość na przeciwną. Oznaczmy przez `bits[i]` wartość `i`-tego bitu liczby d. Przy takim oznaczeniu liczbę d możemy przedstawić jako sumę:

$$d = bits[0] * 2^0 + bits[1] * 2^1 + bits[2] * 2^2 + \ldots + bits[n] * 2^n$$

Wynika to bezpośrednio z zapisu naturalnego kodu binarnego.

Jeśli teraz k-ty bit wykładnika d zmieni wartość z 0 na 1, liczba d zostanie powiększona o 2^k, a jeśli k-ty bit zmieni wartość z 1 na 0, liczba d zostanie zmniejszona o 2^k.

Ponieważ modulus ma 1024 bity, wiemy, że wykładnik d ma nie więcej niż 1024 bity. Możemy więc dla każdego błędnego podpisu `signature’` przetestować każdy z 1024 bitów i sprawdzić, który z nich zmienił wartość:

» Jeśli $signature' + 2^k mod\ n = signature$ dla pewnego k z zakresu <0,1024>, oznacza to, że k-ty bit wykładnika d zmienił wartość z 0 na 1. Z czego wynika, że w oryginalnej liczbie d k-ty bit ma wartość 0.

» Jeśli $signature' - 2^k mod\ n = signature$ dla pewnego k z zakresu <0,1024>, oznacza to, że k-ty bit wykładnika d zmienił wartość z 1 na 0. Wiemy dzięki temu, że w oryginalnej liczbie d k-ty bit ma wartość 1.

» Jeśli żadna z powyższych własności nie zaszła, oznacza to, że uszkodzeniu uległa sama podpisywana wiadomość albo przekłamane zostało kilka bitów wykładnika d.

W analizowanych zadaniach, dla ułatwienia, każdy błędny podpis był wynikiem zmiany wartości dokładnie jednego bitu wykładnika d. Dzięki temu, aby poznać oryginalną wartość klucza prywatnego, potrzebne było pobranie z serwera 1024 różnych podpisów dla tych samych danych. Każdy z tych podpisów pozwalał na odkrycie wartości jednego bitu.

```
# good - poprawny podpis dla wiadomości
print 'precomputing g**(d+2**k) == good * g**2**k...'
good_plus = [(good * pow(2, pow(2, bit, mod), mod)) % mod for
bit in range(1024)]
print 'precomputing g**(d-2**k) == good / g**2**k...'
good_minus = [(good * invmod(pow(2, pow(2, bit, mod), mod),
mod)) % mod for bit in range(1024)]

bits = {}
for bad in bad_sigs:
  for i in range(1024):
    # bad == g' == g^d' == g^(d+2**k) == good * g^2**k
    if bad == good_plus[i]:
      bits[i] = 0  # i-ty bit klucza to 0
    # bad == g' == g^d' == g^(d-2**k) == good / g^2**k
    if bad == good_minus[i]:
      bits[i] = 1 # i-ty bit klucza to 1
```

W ten sposób odzyskujemy wartość wykładnika d i stosując przepis przytoczony wcześniej:

$$plaintext = ciphertext^d (mod\ n)$$

możemy odszyfrować pierwszą flagę i tym samym zakończyć zadanie Broken Box.

ODZYSKIWANIE KLUCZA NA PODSTAWIE JEGO FRAGMENTU

Drugie zadanie, o którym postanowiliśmy napisać, wyglądało z pozoru dokładnie tak samo. Znów mieliśmy dostęp do serwera, który podpisywał dane za pomocą RSA, i ponownie część podpisów była niepoprawna. Zadanie różniło się od wcześniej opisywanego problemu dwoma szczegółami:

1. Zmienił się wykładnik szyfrujący e i zamiast standardowej wartości 65537, wynosił on w tej części zaledwie 97. Niemniej to samo w sobie nie powoduje luki bezpieczeństwa – z powodzeniem można stosować nawet wykładnik 3 przy zachowaniu pewnych zasad.
2. Liczba unikalnych błędnych podpisów była wyraźnie mniejsza.

Mniejsza liczba błędnych podpisów, które mogliśmy uzyskać, oznaczała automatycznie mniejszą liczbę bitów wykładnika d możliwych do odkrycia. W tym przypadku z 1024 bitów mogliśmy poznać zaledwie 295. Nie były to jednak bity przypadkowe – były to dokładnie 295 najniższe bity liczby d.

```
$ python solver.py signatures
precomputing +...
precomputing -... ??????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????00000110000110
1000010001101110010111100000000101100101100010101011110010011101
1111011101010111010110010111111001001100111001010000011111010001
0011001100001001110000000101110110000010011111010001100000010110
0011111011001011010001011100011001110100011101001101110011100110
1110001110110011111010111011101101
```

Można by odnieść wrażenie, że brak znajomości pozostałych 700 bitów powinien zapewnić wystarczające bezpieczeństwo klucza prywatnego, jednak Dan Boneh, Glenn Durfee i Yair Frankel już w 1998 r. w artykule „An attack on RSA given a small fraction of the private key bits" dowiedli, że można dokonać faktoryzacji modulusa, a tym samym także odzyskać klucz prywatny, posiadając informacje zaledwie o ¼ najniższych bitów wykładnika d. Złożoność obliczeniowa przedstawionej przez nich metody jest liniowa względem wykładnika szyfrującego e, stąd też w zadaniu Still Broken Box wykładnik ten został zmniejszony do wartości 97.

Algorytm opisany we wspomnianej wyżej publikacji ma dość złożone podstawy matematyczne, więc jeśli kogoś nie interesuje skomplikowana matematyka, może przeskoczyć od razu do dalszej części artykułu.

Przed przejściem do samego algorytmu musimy omówić twierdzenie Coppersmitha. Mamy wielomian $f(x,y)$ na liczbach całkowitych, gdzie maksymalny stopień każdego czynnika to δ i współczynniki f są względnie pierwsze. Ustalamy teraz X i Y jako przedziały na rozwiązania wielomianu (x0, y0) i definiujemy $f^*(x) = f(Xx, Yy)$. Niech D będzie największym współczynnikiem f*. Jeśli $XY < D^{\frac{2}{3\delta}}$, to możemy „szybko" (w czasie wielomianowym) znaleźć wszystkie rozwiązania (x0, y0) dla f(x, y), takie że x<X oraz y<Y. Dowodu tego twierdzenia nie przedstawimy, bo jest bardzo długi i nieładny (opiera się na technice znanej jako *lattice basis reduction*) – można go znaleźć np. pod adresem http://web.eecs.umich.edu/~cpeikert/lic13/lec04.pdf.

Sam problem odzyskiwania wyższej części klucza można przedstawić matematycznie w następujący sposób: dla RSA, gdzie $N = pq = n$ bitowy modulus, podane jest $r \geq 2^{\frac{n}{4}}$ i znamy $p_0 = p\ mod\ r$. Możemy z tego wyliczyć:

$$q0 = \frac{N}{p0}\ mod\ r$$

W tym momencie można zdefiniować wielomian:

$$f(x, y) = (rx + p0)(ry + q0) - N$$

Rozwiązanie (x0, y0) tego równania pozwoli nam trywialnie odzyskać faktoryzację N (p = rx + p0, oraz q = ry + q). Rozwiązać to równanie możemy bezpośrednio z twierdzenia Coppersmitha z X i Y ustalonym na:

$$\frac{2^{n/2+1}}{r}$$

IMPLEMENTACJA ALGORYTMU

Uruchamiamy wspomniany algorytm przygotowany dla środowiska obliczeniowego Sage:

```
def partial_p(p0, kbits, n):
  PR.<x> = PolynomialRing(Zmod(n))
  nbits = n.nbits()

  f = 2^kbits*x + p0
  f = f.monic()
  roots = f.small_roots(X=2^(nbits//2-kbits), beta=0.3) # szukamy
  pierwiastków < 2^(nbits//2-kbits) z czynnikiem >= n^0.3
  if roots:
    x0 = roots[0]
    p = gcd(2^kbits*x0 + p0, n)
    return ZZ(p)

def find_p(d0, kbits, e, n):
  X = var('X')

  for k in xrange(1, e+1):
    results = solve_mod([e*d0*X - k*X*(n-X+1) + k*n == X], 2^kbits)
    for x in results:
      p0 = ZZ(x[0])
      p = partial_p(p0, kbits, n)
      if p:
        return p

if __name__ == '__main__':
  print "[+] Start"
  n = 123541066875660402939610015253549618669091153006444623444
  0816487986129314268044740972499836229081317710266533226014 6
  6480170685973651622700515979315988600405563682920330486664 84
  5273165214922371767569956347920192959023447480720231820595 5
  9000359680240983293591190952704871706121993481942612800689 5
  966231433690709
  e = 97
  beta = 0.5
  epsilon = beta^2/7
  d0 = 0b00000110000110100001000110111001011110000000010110010110
  0010101011110010011101111101110101011101011001011111001001100
  1110010100001111101000100110011000010011100000001011101100000 10
  0111110100011000000101100011110110010110100010111000110011101 0
  0011101001101110011100110111000111011001111101011101110110 1
  nbits = n.nbits()
  kbits = d0.nbits()
  print "lower %d bits (of %d bits) is given" % (kbits, nbits)
  p = find_p(d0, kbits, e, n)
  print "[+] Found p: %d" % p
```

I w dość krótkim czasie uzyskujemy wynik:

```
[+] Start
lower 295 bits (of 1024 bits) is given
[+] Found p: 10734991637891904881084049063230500677461594645206
40095591612930789268466507434132424531182846720643944357091117 7
6976154738469557875377495266473525537100 47
```

Stosując omówiony algorytm, możemy poznać faktoryzację modulusa na liczby pierwsze, a dzięki temu za pomocą rozszerzonego algorytmu Euklidesa obliczyć wartość wykładnika d i wykorzystać go do odszyfrowania drugiej flagi:

```
import gmpy2

def decode_rsa(ct, d, n):
  return pow(ct, d, n)

def long_to_bytes(integer_data): # zamiana integera na bajty
  integer_data = str(hex(long(integer_data)))[2:-1]
  return "".join([chr(int(integer_data[i:i + 2], 16)) for i in
  range(0, len(integer_data), 2)])

def phi(p, q):
  return (p - 1) * (q - 1)

def solve():
  p = 115082592556095281787829856723844891818817809694237593729 6
  2395789423779211087080016838545204916636221839732993706338791 5
  71211260830264085606598128514985547
  n = 123541066875660402939610015253549618669091153006444623444
  0816487986129314268044740972499836229081317710266533226014 6
  6480170685973651622700515979315988600405563682920330486664 84
  5273165214922371767569956347920192959023447480720231820595 5
  9000359680240983293591190952704871706121993481942612800689 5
  966231433690709
  q = n / p # n = p*q
  e = 97
  d = gmpy2.invert(e, phi(p, q)) # za pomocą rozszerzonego
  algorytmu Euklidesa EGCD
  flag = 963243286517902867887788560465718850851171292484401 64
  8199086297618996849921871998820969123860203514863471191022 15
  9303016183442675422385168171015942260317151064369817997256 01
  9782321243499671330561860196893589739537540211539349537450 37
  8280150771547407210291106509065747806192326917228493557131 63
  780985059673037
  pt = decode_rsa(flag, d, n)
  print(long_to_bytes(pt))
```

Można jeszcze dodać, że przedstawiony algorytm odzyskiwania klucza prywatnego, mając tylko najniższe ¼ bitów, wydaje się czysto teoretyczny, ale również ma swoje zastosowania. Na przykład ataki czasowe na RSA mogą zostać użyte do odzyskania wybranych bitów, ale są bardzo kosztowne czasowo. Używając pokazanej metody, można ten czas zmniejszyć nawet czterokrotnie.

PODSUMOWANIE

Opisane przez nas zadania były bardzo ciekawym przykładem na prezentację praktycznych podatności jednego z najpopularniejszych algorytmów kryptografii asymetrycznej. W przeciwieństwie do przeważającej większości zadań kryptograficznych spotykanych podczas CTFów nie mieliśmy tu do czynienia z błędami implementacyjnymi ani konfiguracyjnymi, a z problemami na etapie wykonania.

Kluczowe dla zadania Broken Box było zaobserwowanie związku pomiędzy zmianą wartości pojedynczego bitu a zmianą wartości podpisu kryptograficznego. Zadanie Still Broken Box wymagało dodatkowo odnalezienia pozycji literaturowych omawiających szczególną sytuację, z którą się spotkaliśmy.

Stanisław Podgórski, Jarosław Jedynak

Rozwiązanie zadań „Broken Box" i „Still Broken Box" zostało nadesłane przez zespół p4, który robi, co może, żeby polskie flagi były wysoko w rankingach drużyn CTFowych.

- https://ctftime.org/team/5152