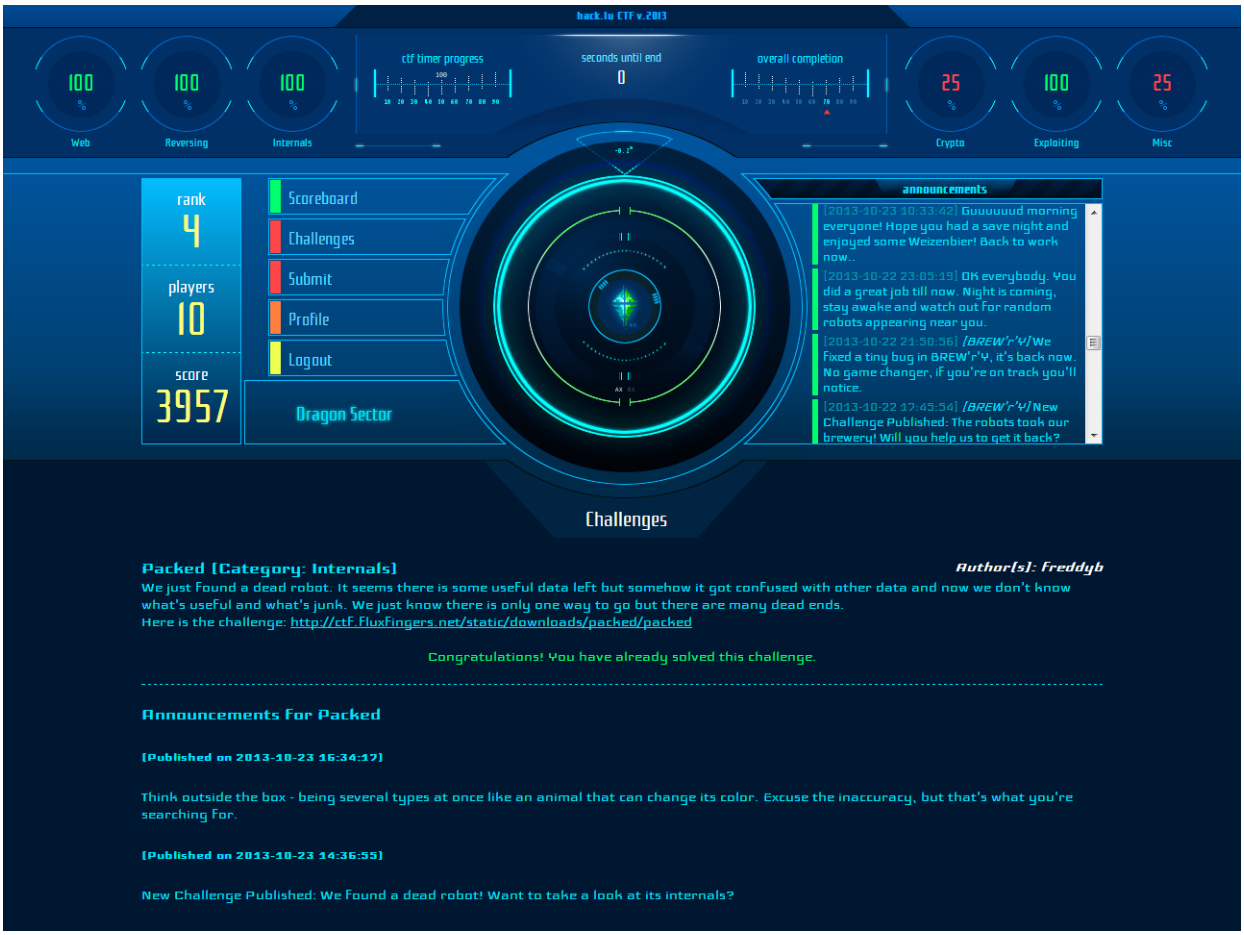
Gynvael Coldwind

Zdobyć flagę...

hack.lu CTF 2013 – Packed

Średnio co około dwa tygodnie gdzieś na świecie odbywają się komputerowe Capture The Flag – zawody, podczas których kilku/kilkunasto osobowe drużyny starają się rozwiązać jak najwięcej technicznych zadań z różnych dziedzin informatyki: kryptografii, steganografii, programowania, informatyki śledczej, bezpieczeństwa aplikacji internetowych itd. Chcielibyśmy zaproponować Wam nowy dział w *Programiście* – Strefę CTF – w którym co miesiąc będziemy publikować wybrane zadanie pochodzące z jednego z minionych CTFów wraz z jego rozwiązaniem.



CTF	hack.lu CTF 2013 http://ctf.fluxfingers.net
Waga CTFtime.org	70 (https://ctftime.org/event/97)
Organizator	FluxFingers (http://fluxfingers.net/)
Liczba drużyn (z niezerową liczbą punktów)	412
System punktacji zadań	Od 100 (proste) do 500 (trudne) punktów.
Liczba zadań	21
Podium	1. Plaid Parliament of Pwning (USA) – 4450 pkt. 2. More Smoked Leet Chicken (Rosja) – 4149 pkt. 3. Stratum Auhuur (Niemcy) – 3967 pkt.
Zadanie	Packed (200 pkt.)

O CTF-IE

Hack.lu CTF to organizowane corocznie zawody przy okazji konferencji Hack.lu odbywającej się w Luksemburgu (możliwy jest również udział przez Internet). Tegoroczny CTF był organizowany przez FluxFingers – zespół z Uniwersytetu Ruhry w Bochum. Z uwagi na wymagające zadania, silną konkurencję oraz wysoką wagę w ogólnym rankingu prowadzonym przez serwis CTFtime.org, jest jednym z najważniejszych CTF-ów w ciągu roku. W tej edycji wzięło udział 412 drużyn z całego świata. Zadania miały oczywiście zróżnicowany poziom trudności, jednak ogólnie był to jeden z bardziej wymagających CTF-ów w tym roku.

ZADANIE

Zadanie, które chcielibyśmy przedstawić w tym numerze, nosiło nazwę Packed i za jego rozwiązanie można było dostać 200 punktów. W ramach zadania do pobrania był 14-kilobajtowy plik o nazwie *packed*, w którym należało znaleźć ukrytą flagę – czyli hasło, po którego wprowadzeniu w systemie hack.lu CTF otrzymywało się punkty.

Zachęcamy czytelników do pobrania pliku *packed* z http://gynvael.coldwind.pl/ctf-mirror/ i prześledzenia jego analizy wspólnie z nami.

Ramka 1. Istotne fragmenty pliku *packed*

```
.....
.....
.....#disabled-encoding:._rot_____
_____
_____
_____13.|.... ..........
.~+-+& . .'~ . .|...#.'~&+.)..(..$%.+......*$..%$.. .(.{...-..}-*.!.*.
!$...".{.......#.....#.}..}|$....!..-}.........'..%.(%.",.~......!(|.
.&#,...(.-...'...+.. -.-.").|..~....#.......-....~.'~* ..*.(...#{$...*
|$'........ .#.{..(.(.....&."""%PDF-102.98.1 0 obj <</Type /Catalog /P
ages 2 0 R>>.endobj.2 0 obj <</Type /Pages /Kids [3 0 R] /Count 1>>.en
[...]
000 n.0000000317 00000 n.trailer <</Size 7/Root 1 0 R>>.startxref.406.
%%EOF.""".pvcure="U51\\\'Hk2W&+(3M;Hkpk0Kkf\k13u\k014$I!E($E>\g/)E!\k0
[...]
yr a <(5 *10 **6 ):xrl =(s (xrl ).qvtrfg ());a =a +1 .xrl =xrl [:5 ].h
ccre ().juvyr yra (xrl )<yra (pvcure ):xrl =xrl *2 .cynva ="".wbva (zn
c (pue ,[beq (n )^beq (o )sbe n ,o va mvc (pvcure ,xrl )])).gel :rkrp.
cynva .rkprcg :cevag "k\k9p\k0o/\kpn\kpsXJ\ks0A\knqG\k04\k00\k14q\k03k
".qrpbqr ("zip"), erce(cynva).'''UEsDBBQAAAgAAL1jikFexjIMJwAAACcAAAAIA
AAAbWltZXR5cGVhcHBsaWNhdGlvbi92bmQub2Fz.aXMub3BlbmRvY3VtZW50LnRleHRQSw
[...]
```

REKONESANS

Już na pierwszy rzut oka można było zauważyć kilka różnych części składowych pliku (istotne fragmenty pliku w postaci ASCII są przedstawione w Ramce 1):

- **PDF** (kolor zielony) – można rozpoznać go po charakterystycznym ciągu znaków %PDF, oznaczeniach kolejnych obiektów (np. `1 0 obj`) oraz specyficznych tagach umieszczonych pomiędzy << a >> (np. `<</Type /Catalog /Pages 2 0 R >>`).
- **Kod zaszyfrowany ROT13** (kolor żółty). ROT13 jest bardzo prostym szyfrem podstawieniowym, który w zasadzie spotyka się jedynie w zadaniach CTF-owych oraz podczas nauki kryptografii. Cechą charakterystyczną większości jego implementacji jest szyfrowanie jedynie liter oraz przepisanie bez jakiegokolwiek szyfrowania pozostałych znaków wiadomości. Stąd widząc w szyfrogramie ciąg np. `a =a +1` czy `xrl =xrl[5:]`, mogliśmy się domyślić, że mamy do czynienia ze skryptem napisanym w Pythonie.
- **Fragment zakodowany base64** (na niebiesko). Base64 jest bardzo popularnym kodowaniem, służącym do transportu danych binarnych w protokole tekstowym (przykładem zastosowania mogą być załączniki w wiadomościach e-mail). Dane nim zakodowane są na tyle charakterystyczne, że nawet przy niewielkim doświadczeniu bardzo łatwo można je rozpoznać (długi ciąg złożony z liter, cyfr oraz co najwyżej dwóch innych znaków, często zakończony jednym lub dwoma znakami równości).

Poza wskazanymi wyżej fragmentami, plik zawierał jeszcze 256-bajtowy blok bajtów o podwyższonej entropii, niemniej jednak nie wyglądał on szczególnie interesująco w stosunku do wyżej wymienionych fragmentów, więc jego analizę pominęliśmy.

ŚLEPY ZAUŁEK #1

Zaczęliśmy od przyjrzenia się dokumentowi PDF. Format PDF, podobnie jak ZIP, nie musi rozpoczynać się wraz z początkiem pliku – a więc by wyświetlić dokument, wystarczy dodać do nazwy pliku rozszerzenie `.pdf` i otworzyć go w dowolnym programie wspierającym ten format. W tym wypadku ujrzeliśmy następującą wiadomość:

no hint given (brak podpowiedzi)

A więc nie tędy droga. Na wszelki wypadek rzuciliśmy jeszcze okiem na ogólną strukturę dokumentu, czy przypadkiem pomiędzy tagami nie kryją się jakieś podpowiedzi, ale... nie, nic ciekawego tam nie znaleźliśmy.

ŚLEPY ZAUŁEK #2

W drugiej kolejności rozkodowaliśmy ciąg *base64* z końca pliku (korzystając ze standardowego modułu Pythona – *base64*, oraz funkcji `b64decode`). Otrzymany ciąg rozpoczynał się od dość charakterystycznych liter PK, wskazujących na format ZIP (PK pochodzi od inicjałów Phila Katza – twórcy formatu ZIP). Oczywiście, o ile plik ZIP nie musi rozpoczynać się od tych liter (jak wspomnieliśmy wcześniej, plik ZIP jest jednym z formatów, który nie musi rozpoczynać się wraz z początkiem pliku), to faktycznie zazwyczaj można je znaleźć na samym początku pliku.

Wewnątrz archiwum znaleźliśmy m.in. plik *META-INF/manifest.xml*, w którym można znaleźć informacje o tym, czym w zasadzie analizowany ZIP był:

```
application/vnd.oasis.opendocument.text
```

Mieliśmy zatem do czynienia z ODT (dokumentem OpenOffice), który można obejrzeć np. za pomocą WordPada lub przez Google Drive. Po otwarciu dokumentu okazało się ponownie, że nie tędy droga:

still no hint given (nadal brak podpowiedzi)

Na wszelki wypadek sprawdziliśmy jeszcze samą strukturę archiwum, jak i same pliki tworzące ODT, ale jedyne, co udało się znaleźć, to tytuł dokumentu – *mad forensic skillz* – w pliku *meta.xml*.

ROT 13 I SKRYPT W PYTHONIE

Pozostała nam więc analiza skryptu w Pythonie. Po rozkodowaniu ROT13 (korzystając ze strony rot13.com) otrzymaliśmy skrypt przedstawiony na Listingu 1.

Listing 1. Rozszyfrowany skrypt

```
cipher="H51\\\'Ux2J&+(3Z;Uxcx0Xxs\x13h\x014$V!R($R>\t/)R!\
x01<.\x13,N-aP4M4aRuG1-VuU0 GuH+a@0W=3R9\x01>(_0\x01,8C0Rx
GuN6\"V|\x1ezKZ3\x014$]}R!2\x1d4S?7\x1au\x1fxs\t_\x01xa\x13<Gx)
R&Ip2J&\x0f93T#zj\x1c\x1ap\x13rk\x00g\x01e|\x13g\x19ju\x0ba\
x18jt\x02o+xa\x13u\x01xa\x13%S1/Gu\x03\x1b.\\:N7.\\:N4o\x13\
x0cN-3\x133M9&\x13<Rx A2WjiZ{DvaX0Xjh\x136N6\"R!\x01\x07rC0p\
x138a\x1dc22ieu\x161Fw+=-@0\x1bRa\x13u\x01(3Z;UxcR\'F.s\
x1c>D!s\x13<Rx,Z&R1/Tw+R"
n =0 ;import hashlib ,sys ;
try :key =sys .argv [1 ]
except IndexError :sys .exit ("x\x9c\xf3N\xadT0T\xc8\xcd,.\xce\
xccKW\xc8\xccSH,J/\x03\x00M\x97\x07\\".decode ("mvc"))
f =getattr (hashlib ,"x\x9c\xcbM1\x05\x00\x02G\x01\x07".decode
("mvc"))
while n <(5 *10 **6 ):key =(f (key ).digest ());n =n +1
key =key [:5 ].upper ()
```

```
while len (key )<len (cipher ):key =key *2
plain ="".join (map (chr ,[ord (a )^ord (b )for a ,b in zip
(cipher ,key )]))
try :exec plain
except :print "x\x9c\x0b/\xca\xcfKW\xf0N\xadT\x04\x00\x14d\
x03x".decode ("mvc"), repr(plain)
```

Jak się szybko okazało, skrypt nie wykonywał się poprawnie:

```
Traceback (most recent call last):
  File "skrypt.py", line 5, in <module>
    except IndexError :sys .exit ("x\x9c\xf3N\xadT0T\xc8\xcd,.\
xce\xccKW\xc8\xccSH,J/\x03\x00M\x97\x07\\".decode ("mvc"))
LookupError: unknown encoding: mvc
```

Należy zauważyć, że argument metody decode w zaszyfrowanej wersji – "zip" – ma więcej sensu niż "mvc", które uzyskaliśmy po rozkodowaniu – prawdopodobnie twórcy zadania chcieli poprzez tę zamianę utrudnić odrobinę analizę. Zatem, aby powyższy skrypt zmusić do poprawnego działania, należało we wszystkich miejscach zmienić argument decode z "mvc" na "zip".

Kodowanie "zip" w powyższym skrypcie zostało użyte do zaciemnienia tekstu, który zazwyczaj podczas analizy często daje bardzo dużo informacji. Warto więc rozkodować powyższe ciągi i zobaczyć, co się pod nimi kryje. Rozkodowany tekst wygląda następująco (w kolejności występowania w skrypcie):

```
'Key 1 missing in argv'
'md5'
'Wrong Key!'
```

ANALIZA SKRYPTU

Co więc robił powyższy kod? Przede wszystkim obliczał klucz z hasła podanego w argumencie skryptu:

```
key =sys .argv [1 ]
f =getattr (hashlib , 'md5')
while n <(5 *10 **6 ):
  key =(f (key ).digest ())
  n =n +1
key =key [:5 ].upper ()
```

Ostateczny klucz był uzyskiwany poprzez wielokrotne wyliczenie sumy MD5 z podanego argumentu, a następnie obcięcie uzyskanego ciągu do 5-bajtów. Czyli potencjalnie mieliśmy do czynienia z kluczem o wielkości 40 bitów, a więc dość niewielkim – w najgorszym wypadku podejście siłowe powinno dość szybko sobie z nim poradzić. Dalej...

```
while len (key )<len (cipher ):key =key *2
plain ="".join (map (chr ,[ord (a )^ord (b )for a ,b in zip
(cipher ,key )]))
```

Klucz był używany w prostym szyfrze korzystającym z XOR, co znacznie upraszczało sprawę, ponieważ oznaczało to, że nie mamy do czynienia z 40-bitowym kluczem, a z pięcioma 8-bitowymi kluczami, gdzie każdy klucz odpowiedzialny był za co piąty bajt (czyli bajt zerowy klucza służył do odszyfrowania bajtu zerowego, piątego, dziesiątego, piętnastego itd). Popatrzmy, co było dalej.

```
exec plain
```

Po poprawnym rozszyfrowaniu powinniśmy więc dostać kolejny skrypt w Pythonie, który zostałby w tym miejscu wykonany. Na tym etapie znaliśmy już wszystkie istotne elementy układanki.

ŁAMANIE XOR

Przede wszystkim, mogliśmy z dużym prawdopodobieństwem założyć, iż w poprawnie rozszyfrowanej wiadomości pojawią się takie ciągi jak "print" lub "import" – w końcu wiedzieliśmy, że wiadomość jest poprawnym skryptem w Pythonie. Idąc dalej, mogliśmy założyć, że występowanie obu tych ciągów może być dobrym kryterium do stwierdzenia, czy wiadomość została poprawnie rozszyfrowana (a więc czy znaleźliśmy pasujący klucz).

Do samego znalezienia klucza można było podejść na kilka sposobów:

- **Sposób 1**: Sposób siłowy (*brute force*) – sprawdzić wszystkie możliwości – liczba testów: 2^{40}.
- **Sposób 2**: Dla każdego bajtu klucza znaleźć wszystkie wartości, które po rozkodowaniu co piątego bajtu dają jedynie drukowalne znaki (a konkretniej, typowe znaki, które można znaleźć w skryptach napisanych w Pythonie) – liczba operacji: 5 * 256.
- **Sposób 3**: Rozpoczynając od N-tego znaku, obliczyć wynik operacji XOR dla kilku kolejnych znaków ze znakami ze słowa "import" (albo innego ciągu, którego się spodziewamy). Mając tak powstały "klucz", należy sprawdzić, czy zerowy oraz piąty znak "klucza" są równe. Jeśli nie, odrzucamy "klucz" i idziemy dalej. Jeśli tak, otrzymany klucz przycinamy do pięciu znaków i używamy go do rozszyfrowania całego ciągu. I tak dla kolejnych N, rozpoczynając od 0, aż do końca danych – liczba operacji: 287 (o ile w oryginalnej wiadomości faktycznie wystąpiło słowo "import").

Ponieważ trochę obawialiśmy się, że słowa "import" lub "print" mogą nie wystąpić w rozszyfrowanym ciągu, postanowiliśmy zaimplementować sposób drugi, licząc na to, że będzie na tyle niewiele kombinacji klucza, że uda się manualnie przejrzeć wszystkie rozszyfrowane ciągi i zdecydować, który jest poprawnym.

W tym celu stworzyliśmy kod zaprezentowany na Listingu 2.

Listing 2. Łamanie XOR

```
import os
cipher=map(ord, "H51\\\'Ux2J&+(3Z;Uxcx0Xxs\x13h\
x014$V!R($R>\t/)R!\x01<.\x13,N-aP4M4aRuG1-VuU0 GuH+a@0W=3R9\
x01>(_0\x01,8C0Rx GuN6\"V|\x1ezKZ3\x014$]}R!2\x1d4S?7\x1au\
x1fxs\t_\x01xa\x13<Gx)R&Ip2J&\x0f93T#zj\x1c\x1ap\x13rk\x00g\
x01e|\x13g\x19ju\x0ba\x18jt\x02o+xa\x13u\x01xa\x13%S1/Gu\x03\
x1b.\\:N7.\\:N4o\x13\x0cN-3\x133M9&\x13<Rx A2WjiZ{DvaX0Xjh\
x136N6\"R!\x01\x07rC0p\x138a\x1dc22ieu\x161Fw+=-@0\x1bRa\x13u\
x01(3Z;UxcR\'F.s\x1c>D!s\x13<Rx,Z&R1/Tw+R")
PYTHON_CHARSET = map(ord, """
\t\r\n
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
[]=_(){}:.<>!#%^&*()_";'?,/\\
1234567890
""")
keys = [[],[],[],[],[]]
for key_i in xrange(5):
   for test in xrange(256):
      all_ok = True
      i = key_i
      while i < len(cipher):
         d = cipher[i] ^ test
         if d not in PYTHON_CHARSET:
            all_ok = False
         break
      i += 5
   if all_ok:
      keys[key_i].append(test)

print keys
```

Warto zwrócić uwagę, że zestaw znaków w PYTHON_CHARSET nie zawiera wszystkich znaków, które faktycznie mogłyby wystąpić w poprawnym kodzie w Pythonie, więc jeśli dla danej pozycji nie zostałaby znaleziona żadna możliwa wartość klucza, należałoby dodać do zestawu brakujące znaki (np. tyldę i at).

Na szczęście, wynik działania kodu okazał się jednoznaczny:

```
[[33], [88], [65], [51], [85]]
```

Dla każdej pozycji klucza skrypt znalazł dokładnie jedną wartość. Nie znaczy to jednak, że musiał to być poprawny klucz – natomiast rozszyfrowanie ciągu za jego pomocą rozwiało wątpliwości: wynik okazał się być poprawny, a kolejny skrypt wyglądał następująco:

```
import sys
print "Key 2 = leetspeak(what do you call a file that is
several file types at once)?"
if len(sys.argv) > 2:
   if hash(sys.argv[2])%2**32 == 2824849251:
      print "Cooooooooool. Your flag is argv2(i.e. key2) concat
      _3peQKyRHBjsZ0TNpu"
else:
   print "argv2/key2 is missing"
```

Jak widać, nasze obawy o brak słowa "import" czy "print" okazały się bezpodstawne.

ANALIZA DRUGIEGO SKRYPTU

Drugi skrypt rozpoczynał się od zagadki: "jak nazwiesz plik, który jest kilkoma typami plików jednocześnie?" (w dość luźnym tłumaczeniu). Odpowiedź na tę zagadkę należało podać zapisaną w leetspeaku ("1337 5p34K" – patrz http://pl.wikipedia.org/wiki/Leet_speak) jako argument wywołania skryptu.

Weryfikacja odpowiedzi była zrealizowana analogicznie do sposobu, w jaki weryfikuje się hasła w serwisach internetowych czy w systemach operacyjnych – z odpowiedzi jest liczony hash (w tym wypadku jest to standardowy hash stosowany w różnych miejscach w Pythonie), a następnie wyliczona wartość jest porównywana ze znaną wartością (w tym wypadku: zapisaną w kodzie):

```
if hash(sys.argv[2])%2**32 == 2824849251:
   print "Cooooooooool. Your flag is argv2(i.e. key2) concat
   _3peQKyRHBjsZ0TNpu"
```

A więc wiedzieliśmy, że musimy najpierw znaleźć odpowiedź na zagadkę, a następnie odpowiedni zapis w leetspeaku.

ODPOWIEDŹ NA ZAGADKĘ

Zagadka okazała się najtrudniejszą częścią zadania. Narzucającą się odpowiedzią był „polyglot" (patrz np. *CorkaMIX binary polyglot* – https://code.google.com/p/corkami/wiki/mix), jednak żadna wersja zapisu nie okazała się być tą poprawną.

Po niedługim czasie ukazała się podpowiedź od organizatorów, w której przeprosili za nieścisłość i dopowiedzieli, że chodzi o pewnego rodzaju zwierzę, które potrafi zmienić kolor skóry.

Z tak sformułowanej podpowiedzi łatwo było domyślić się, że chodzi o kameleona (*chameleon*); pozostało więc znaleźć odpowiedni zapis w leetspeaku.

WYSZUKIWANIE ZAPISU

W celu sprawdzenia wszystkich możliwości posłużyliśmy się skryptem przedstawionym na Listingu 3.

Listing 3. W poszukiwaniu prawidłowego kameleona

```
import itertools
word = [[ 'c' ],
       [ 'h' ],
       [ 'a', '4' ],
       [ 'm' ],
       [ 'e', '3' ],
       [ 'l', '1' ],
       [ 'e', '3' ],
       [ 'o', '0' ],
       [ 'n' ]]
for test in itertools.product(*word):
   test = ''.join(test)
   if hash(test)%2**32 == 2824849251:
      print test
```

Wynikiem działania skryptu jest:

```
ch4m3l30n
```

A więc udało się rozwiązać zagadkę!

FLAGA I PODSUMOWANIE

Zgodnie z instrukcją z drugiego skryptu w celu uzyskania ostatecznej flagi należało połączyć odpowiedź na zagadkę z wymienionym w wiadomości ciągiem. Wynikiem tej operacji była upragniona flaga, czyli:

```
ch4m3l30n_3peQKyRHBjsZ0TNpu
```

Podsumowując, zadanie to było o tyle ciekawe, że wymagało zarówno podstawowej wiedzy z kryptografii, programowania, jak i analizy plików binarnych. Ostatecznie zadanie zostało rozwiązane przez 67 drużyn.

Rozwiązanie zadania Packed zostało nadesłane przez Dragon Sector – jedną z Polskich drużyn CTF-owych. http://dragonsector.pl/