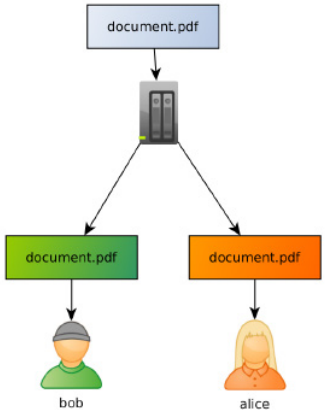
Jak schwytać złodzieja danych

Co łączy wielkie korporacje, służby specjalne oraz gildie w grach komputerowych? Na pierwszy rzut oka niewiele, ale jest co najmniej jedna wspólna cecha: wszystkie mają swoje sekrety oraz chcą je ukryć przed resztą świata. Nieważne, czy mówimy o projekcie nowego produktu, szczegółach operacji antyterrorystycznej albo planach ataku – niektóre informacje powinny zostać w zaufanej grupie.

Niestety, wycieki danych się zdarzają. Nawet NSA[1] czy CIA[2] nie są od nich wolne. Bo jak zabezpieczyć dane przed wyciekiem w cyfrowym świecie? W przypadku ekstremalnie wrażliwych danych możemy trzymać je na komputerach odizolowanych od sieci, uniemożliwić podłączanie do komputera dysków USB oraz nagrywarek CD, a nawet korzystania z telefonów w pobliżu. Ciężko jednak wyobrazić sobie narzucanie takich ograniczeń ludziom w biurze, albo w przypadku organizacji działających głównie w Internecie.

Jest jednak metoda, która może znacząco utrudnić upublicznienie danych. Wystarczy upewnić się, że osoba wynosząca informacje nie pozostanie anonimowa. W końcu mało kto zaryzykowałby pracą i odpowiedzialnością karną, nawet w słusznej sprawie. Na przykład, jeśli kontrolujemy serwer z dokumentami, możemy napisać małe przezroczyste proxy, które automatycznie dokleja do każdego dokumentu informację, który użytkownik go pobrał (Rysunek 1).



Rysunek 1. Interakcja z serwerem

Bob może o tym wiedzieć albo nie, ale w tym momencie, kiedy document.pdf znajdzie się w Internecie, będzie łatwo wskazać winnego.

Cały ten artykuł jest napisany z punktu widzenia osoby chcącej zaimplementować takie zabezpieczenie – wydaje się to naturalne, skoro czasopismo jest skierowane dla programistów. Jednak te same informacje są przydatne także dla ludzi, którzy chcą ominąć takie systemy. W szczególności dla (różnie ocenianych) demaskatorów (ang. *whistleblowers*), czyli ludzi, którzy upubliczniają nielegalnie zdobyte informacje, żeby zwiększyć świadomość publiczną na temat nieuczciwych lub niemoralnych działań wielkich organizacji.

1. https://pl.wikipedia.org/wiki/Edward_Snowden
2. https://www.spidersweb.pl/2017/03/wikileaks-cia.html

1. UKRYWANIE INFORMACJI W PLIKACH

Mniej poetycko ujmując, ten artykuł będzie o tym, jak można sprytnie ukryć pewne dane (na przykład nazwę użytkownika) w plikach. Nie chodzi jednak o zwykłe doklejenie informacji. Zależy nam na tym, by dodane dane nie rzucały się w oczy oraz aby było je jak najtrudniej usunąć. Inaczej mówiąc, chcemy stworzyć cyfrowy **znak wodny**. Temat jest bardzo szeroki, więc w tym artykule skoncentrujemy się tylko na dwóch przykładach – plikach graficznych oraz tekście.

Ale nie będzie to zwykły artykuł na temat steganografii. Wiele cyfrowych metod steganograficznych koncentruje się na detalach formatu pliku – czyli np. wykorzystania detali tego, jak działa wewnętrznie format .png. Jest tutaj jednak jeden problem – z definicji bardzo łatwo usunąć takie informacje, nawet przypadkowo (np. konwertując obraz do formatu .jpg). Tymczasem my chcemy zapisać dane w maksymalnie trwały sposób. Z tego powodu zignorujemy zupełnie te metody (zostało na ich temat napisane wystarczająco dużo) i zastanowimy się, jak schować dane „głębiej".

2. UKRYWANIE INFORMACJI W OBRAZACH

Zacznijmy od plików graficznych, ponieważ są chyba najbardziej popularnym przykładem.

Klasycznym sposobem na ukrycie informacji w obrazku jest zapisanie informacji na najniższych bitach każdego piksela. W celu zakodowania wiadomości zmieniamy ją na bity i zapisujemy subtelnie wartość każdego piksela. W najprostszej wersji w celu zakodowania „0" zmieniamy piksel na parzysty, a w celu zakodowania „1" zmieniamy piksel na nieparzysty.

Ta metoda jest tak popularna, że (jako jedyna) została nawet wspomniana na polskiej Wikipedii[3].

Weźmy pożyczony bezpośrednio z Wikipedii przykład: niepozorne drzewo (Rysunek 2). Okazuje się jednak, że jeśli przeczytamy najniższe dwa bity każdego piksela, otrzymamy zdjęcie kota (Rysunek 3).

Na pierwszy rzut oka ta metoda spełnia wszystkie nasze oczekiwania. Patrząc na Rysunek 2, trudno się domyślić, że ukryte w nim są jakieś dane. Dodatkowo gęstość zakodowanych informacji jest bardzo wysoka – udało się zakodować aż 6 bitów na każdym pikselu. Co więcej, zapisane dane są w stanie przetrwać pewne drobne modyfikacje – na przykład skonwertowanie obrazu do innego, bezstratnego formatu albo screen-shot ekranu. Niestety, trwałość zakodowanych danych nie jest dla nas satysfakcjonująca. Żeby je

3. https://pl.wikipedia.org/wiki/Steganografia

Rysunek 2. Niepozorny obrazek (źródło: https://pl.wikipedia.org/wiki/Plik:Steganography_original.png, użytkownik Cyp, licencja CC-BY 2.0)

Rysunek 3. Ukryta grafika (źródło: https://pl.wikipedia.org/wiki/Plik:Steganography_recovered.png, użytkownik Cyp, licencja CC-BY 2.0)

wymazać, wystarczy nawet nieświadomie zapisać obraz w stratnym formacie (tak jak np. jpg) albo lekko przeskalować. Tymczasem my jako cel postawiliśmy sobie zakodowanie informacji tak, żeby usunięcie ich było maksymalnie trudne.

Czy jednak możemy wymyślić coś znacznie lepszego? Odpowiedź brzmi: oczywiście tak, inaczej spora część tego artykułu nie miałaby sensu. Wcześniej jednak musimy cofnąć się o kilka kroków i przerobić trochę teorii.

2.1 TRANSFORMACJA FOURIERA

Konkretnie, w dalszej części artykułu wykorzystamy Dyskretną Transformację Fouriera[4]. Większość czytelników prawdopodobnie miała (nie)przyjemność zapoznać się z nią już na wczesnych latach studiów. W takim przypadku zazwyczaj prezentowana jest przyjazna definicja w rodzaju:

$$A_n = \sum_{n=0}^{N-1} a_n e^{\frac{2\pi i}{N} kn}$$

Wzór 1. Klasyczna postać Dyskretnej Transformaty Fouriera

Albo równoważnie:

$$A_n = \sum_{n=0}^{N-1} a_n \left[\cos(\frac{2\pi kn}{N}) - i \sin(\frac{2\pi kn}{N}) \right]$$

Wzór 2. Rozwinięcie wzoru 1

4. https://en.wikipedia.org/wiki/Discrete_Fourier_transform

Wspomina się też wtedy, że reprezentuje ona ciąg próbek wejściowych w przestrzeni częstotliwości (ang. *frequency domain*). Ale co to naprawdę znaczy? I do czego można to wykorzystać? Odpowiedź na to pytanie nie zawsze jest podawana bezpośrednio.

Tymczasem transformacja Fouriera jest niezwykle przydatna przy dowolnym przetwarzaniu sygnałów (np. obrazów, dźwięku, wibracji lub sygnałów elektrycznych), więc warto rozumieć ją bardziej intuicyjnie. O co tu właściwie chodzi? W dużym uproszczeniu – Fourier pozwala nam zapisać dowolną funkcję jako sumę cosinusów (i sinusów, jeśli przejmujemy się wartościami złożonymi). Ponieważ artykuł nie traktuje o matematyce, jak również bardziej chodzi mi o pokazanie intuicji niż dokładnej definicji, popatrzmy na przykłady działania algorytmu w praktyce:

Listing 1. Prosty Przykład transformacji oraz jej odwrotności

```
> import scipy.fftpack as fp
> print fp.rfft([1, 1, 1, 1])  # wykonaj FFT na liście [1, 1, 1, 1]
[ 4.  0.  0.  0.]  # wynik: lista w przestrzeni częstotliwości

> print fp.irfft([4, 0, 0, 0])  # operacja odwrotna
[ 1.  1.  1.  1.]  # wynik: oryginalna lista
```

Ten wynik oznacza, że sygnał [1, 1, 1, 1] można zapisać jako 4*cos(2*pi*k*0)/4 (patrz: wzór 2), czyli po prostu 1*cos(0) = 1. Spróbujmy pobawić się trochę z odwrotną transformacją:

Listing 2. Więcej przykładów dla pierwszego elementu

```
> print fp.irfft([4, 0, 0, 0])
[ 1. 1. 1. 1.]

> print fp.irfft([5, 0, 0, 0])
[ 1.25 1.25 1.25 1.25]

> print fp.irfft([8, 0, 0, 0])
[ 2. 2. 2. 2.]

> print fp.irfft([8, 0, 0, 0, 0, 0, 0, 0])
[ 1. 1. 1. 1. 1. 1. 1. 1.]
```

Jak widać, pierwszy element odpowiada za stały czynnik dodawany do każdego elementu. Rodzi się oczywiście pytanie, za co odpowiada drugi element? Można łatwo wydedukować to ze wzoru, ale jeszcze prościej sprawdzić interaktywnie:

Listing 3. Działanie drugiego elementu

```
> print fp.irfft([0, 4, 0, 0])
[ 2.  0. -2.  0.]
```

Ciekawe – w końcu widać, że to faktycznie cosinus. Jeszcze łatwiej to zauważyć kiedy zwiększymy ilość elementów:

Listing 4. Cosinus na całym zakresie danych

```
print fp.irfft([0, 8, 0, 0, 0, 0, 0, 0])
[ 2, 1.4142, 0, -1.4142, -2, -1.4142, 0, 1.4142]
```

Im dalszy element, tym mniejszy cykl funkcji cosinus, aż przy ostatnim elemencie:

Listing 5. Cosinus o dużej częstotliwości

```
> print fp.irfft([0, 0, 0, 4])
[ 1. -1.  1. -1.]
> print fp.irfft([0, 0, 0, 0, 0, 0, 0, 8])
[ 1. -1.  1. -1.  1. -1.  1. -1.]
```

Mówimy cały czas o odwrotnej transformacji. Prawdziwa magia polega na transformacji we właściwą stronę, czyli szybkiej zmianie dowolnego sygnału na sumę cosinusów:

Listing 6. Transformata dla dowolnych danych

```
> print fp.rfft([1337, 1234, 13, 42])
[ 2626.  1324. -1192. 74.]
```

No dobrze, ale jaki to ma związek z właściwym tematem tego artykułu? Okazuje się, że można tę operację zastosować też w przypadku większej ilości wymiarów[5]. W szczególności w przypadku dwóch wymiarów oznacza to, że możemy konwertować obrazy do przestrzeni częstotliwości i z powrotem. Zobaczmy, jak wygląda to w praktyce.

Temat samodzielnej implementacji DFT wykracza poza ramy tego artykułu. O ile relatywnie prosto jest napisać kod prosto z definicji, to zaimplementowanie szybkiego i pewnego algorytmu jest trudne i czasochłonne. Zamiast tego skorzystamy z gotowych bibliotek naukowych numpy oraz scipy:

Listing 7. Szablon kodu do modyfikacji obrazów

```
import numpy as np
import scipy.fftpack as fp
import sys
from PIL import Image

def im2freq(data):
  """ Skonwertuj obraz do przestrzeni częstotliwości """
  return fp.rfft(fp.rfft(data, axis=0), axis=1)

def freq2im(data):
  """ Konwersja "z powrotem", do pikseli """
  return fp.irfft(fp.irfft(data, axis=1), axis=0)

def touint8(data):
  """ Skalowanie wartości tablicy z powrotem do (0, 255) """
  data = data - np.amin(data, axis=(0,1), keepdims=True)
  data = data / max(data.max(), 1)
  return (data * (256-1e-4)).astype(int)

def save(data, fname):
  """ Zapisanie tablicy jako obrazu na dysk """
  out = Image.new('RGB', data.shape[1::-1])
  out.putdata(map(tuple, data.reshape(-1, 3)))
  out.save(fname)

def main():
  freq = im2freq(np.array(Image.open(sys.argv[1])))
  # freq to tablica reprezentująca obraz w przestrzeni
  # częstotliwości. W tym miejscu wykonamy na niej
  # jakieś operacje, a następnie skonwertujemy z powrotem
  save(touint8(freq2im(freq)), sys.argv[2])
```

Weźmy na przykład zupełnie czarny obraz – jak na Rysunku 4. Wykonajmy na na nim DFT. Co będzie wynikiem? Oczywiście – dalej ten sam zupełnie czarny obraz. Dane wejściowe składają się z samych zer, więc wynik również będzie zerami (reprezentowanymi przez czarne piksele).

Rysunek 4. Obrazek z czarnym tłem

5. Samo FFT generalizuje się do wielu wymiarów. Mimo tego w tym artykule używamy prostszego do implementacji i zrozumienia jednowymiarowego FFT, za to wykonujemy go dwa razy.

Wykonajmy teraz prostą operację w przestrzeni częstotliwości – zmieńmy jakąś wartość na znacznie wyższą. Graficznie mówiąc, rozjaśnijmy znacznie jeden piksel:

Listing 8. Pierwsza próba – zmiana jednego piksela

```
freq[5][10][0] = 50000
# ustawienie czerwonego kanału „piksela” (5, 10)
# na wysoką wartość
```

Po wykonaniu transformacji odwrotnej pojawią się odpowiednie „cosinusy”. Wynikiem będzie „szachownica”, zademonstrowana na Rysunku 5.

Rysunek 5. Szachownica cosinusów

Pokazuje to, jak faktycznie wygląda pojedyncza „składowa” w przestrzeni częstotliwości. Nic nie stoi na przeszkodzie, żebyśmy ustawili więcej składowych jednocześnie, tworząc psychodeliczną tęczę – Rysunek 6.

Listing 9. Trzy piksele jednocześnie.

```
freq[5][10][0] = 50000  # pozycja (5, 10), czerwony kanał
freq[7][8][1] = 40000  # pozycja (7, 8), zielony kanał
freq[9][4][2] = 30000  # pozycja (9, 4), niebieski kanał
```

Rysunek 6. Cosinusowa tęcza

Jak widać, wyprodukowane w ten sposób obrazy są bardzo abstrakcyjne. Jednakże teoria Fouriera mówi, że ustawiając odpowiednio dużo punktów, otrzymamy dowolnie dokładne przybliżenie

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

dla każdego obrazu! Na przykład dla Rysunku 7 możemy za pomocą pokazanego kodu wykonać transformację w obie strony i wrócić do stanu nierozróżnialnego od oryginału (dla ludzkiego oka).

Rysunek 7. Zdjęcie kaczki, na „której" testowaliśmy prezentowane algorytmy

Jak wygląda graficzna reprezentacja tego obrazu w przestrzeni częstotliwości? Mało ciekawie. Nawet po sztucznym zwiększeniu kontrastu dwudziestokrotnie cała składa się z jednolitego koloru. Jedynie po przybliżeniu obrazu, w lewym górnym rogu widać jakąś entropię: Rysunek 8. Mimo wszystko da się z takiej reprezentacji otrzymać z powrotem oryginalną kaczkę, wykonując odwrotną transformację.

Rysunek 8. Dyskretna Transformata Fouriera z Rysunku 7

Wychodzi na to, że po DFT większość informacji znajduje się przy początku układu współrzędnych (tzn. w lewym górnym rogu obrazka), a reszta nie „wnosi" wiele. Okazuje się, że to całkiem praktyczne spostrzeżenie. Sprytny sposób na kompresję obrazów to właśnie transformacja obrazów do przestrzeni częstotliwości, a następnie odrzucenie wszystkiego poza tym interesującym kawałkiem. Możemy łatwo zaimplementować to w Pythonie:

Listing 10. Wyzerowanie prawie całego obrazka

```
width, height = freq.shape[0:2]
for x in range(width):
  for y in range(height):
    if x > width/10 or y > height/10:
      freq[x][y] = (0, 0, 0)
```

Jak widać, zerujemy wszystko, poza małym prostokątem szerokim i wysokim na 10% oryginalnego rozmiaru (czyli zostaje ledwo 1% danych). Po uruchomieniu go na naszej kaczce dostaniemy bardzo rozmazaną, ale wciąż rozpoznawalną grafikę (Rysunek 9).

Rysunek 9. Mało wyraźna kaczka z Rysunku 7. Zakodowana za pomocą tylko 1% informacji z oryginału!

Przy zachowaniu 20% obrazu wynik byłby już praktycznie nierozróżnialny dla ludzkiego oka – a o ile mniej przestrzeni dyskowej by wymagał! Oczywiście, ktoś już na to wpadł – tak właśnie działa (w wielkim uproszczeniu) format jpg.

2.2 PRAKTYCZNE UKRYWANIE DANYCH

Ale nie jest to artykuł o przetwarzaniu obrazów (było już ich parę na łamach „Programisty"). Do czego zmierzam? Skoro duża część danych w obrazie jest prawie niewidoczna dla ludzkiego oka, to po co się ma marnować? Możemy ukryć tam nasze dane!

Inaczej mówiąc, zamiast ukrywać dane bezpośrednio w obrazie, możemy ukryć je w niewidzialnej dla człowieka przestrzeni częstotliwości. Spróbujmy zaimplementować to w języku Python:

Listing 11. Enkodowanie i dekodowanie

```
def main():
  op = sys.argv[1]
  input = sys.argv[2]
  if op == 'decode':
    freq = im2freq(np.array(Image.open(input)))
    for x, y, z in select_pixels(int(sys.argv[3])):
      value = freq[x][y][z]
      print '{:10.0f} ({})'.format(value, int(value > 50000))
      # wypisz surowe wartości zakodowanej wiadomości
  elif op == 'encode':
    freq = im2freq(np.array(Image.open(input)))
    output = sys.argv[3]
    bits = bin2bits(sys.argv[4])  # zmień wiadomość na bity
    for bit, (x, y, z) in zip(bits, select_pixels(len(bits))):
    print bit

      freq[x][y][z] = int(bit) * 100000

      # zmień pixel na 0 dla bitów 0, i 100000 dla bitów 1
    save(touint8(freq2im(freq)), output)
  else:
  print "Unknown operation"
```

I to w zasadzie wszystko, poza dwoma funkcjami pomocniczymi: `select_pixels`, która wybiera, na jakich współrzędnych zapisać nasze n bitów danych:

Listing 12. Funkcja pomocnicza wybierająca n mało znaczących pikseli

```
def select_pixels(n):
  # distance - jak „daleko” ukryć dane.
  # im wyższe, tym więcej danych można ukryć
  # bez widocznej zmiany obrazka, ale też
  # powoduje, że łatwiej je uszkodzić.
  distance = 10
  while True:
    for y in range(distance + 1):
      for channel in range(3):
        yield  (distance-y, y, channel)
        n -= 1
        if n <= 0:
          return
    distance += 1
```

A także `bin2bits`, która zamienia bajty na bity (konkretnie napis składający się z zer i jedynek):

Listing 13. Zmiana bajtów na ciąg bitów

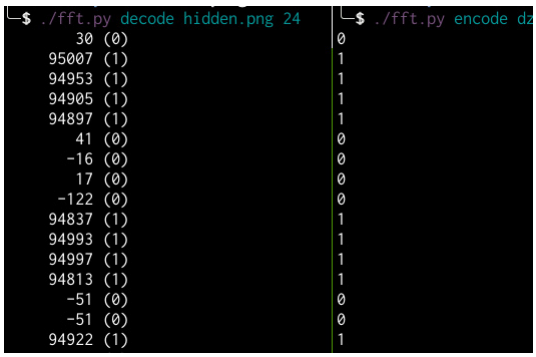
```
def bin2bits(data):
  out = ''
  for byte in data:
    out += '{:08b}'.format(ord(byte))
  return out
```

Oczywiście jest to bardzo prymitywny kod, tak zwany „proof of concept”. Można go użyć do zakodowania wiadomości, a później przeczytania wyników:

Listing 14. Prosta interakcja z programem

```
$ python fft.py encode dziob.png hidden.png xyz
# (program wypisuje bity wiadomości)
$ python fft.py decode hidden.png 24
# (program wypisuje kolejne zdekodowane wartości)
```

Przykład wykonania powyższych komend można zaobserwować na Rysunku 10. Po lewej stronie wynik dekodowania: pierwsza liczba to surowa wartość w przestrzeni częstotliwości, a druga to zdekodowany bit. Po prawej stronie dla porównania poprawne wartości wyliczone podczas kodowania. Jak widać, metoda działa – oryginalna wiadomość została spokojnie i bezbłędnie odtworzona.



Rysunek 10. Interakcja z programem kodującym

Czas porównać ją z przedstawionym na początku ukrywaniem informacji w najniższych bitach obrazka. Czy faktycznie coś zyskujemy?

Okazuje się, że dane ukryte w ten sposób są **znacznie** bardziej odporne na manipulacje. W testach laboratoryjnych, które przeprowadziłem, zakodowane dane przetrwały bez większych problemów:

- Zapisanie obrazka w stratnym formacie .jpg,
- Rozjaśnienie, przyciemnienie i zmianę kontrastu obrazka,
- Przeskalowanie obrazka do 60% rozmiaru i z powrotem,
- Zakrycie niewielkiej części obrazka (np. narysowanie kilku kresek programem graficznym),
- A nawet wyświetlenie zdjęcia na monitorze komputera, zrobienie zdjęcia komórką, poprawienie perspektywy zdjęcia w programie graficznym i dopiero zdekodowanie!

W momencie kiedy nasz znak wodny da się, przy pewnej staranności i odrobinie szczęścia, odtworzyć nawet z samego zdjęcia kiepskiej jakości, możemy uznać, że odnieśliśmy sukces[6]. Do osiągnięcia tak ekstremalnego efektu trzeba niestety zapisywać dane blisko początku układu współrzędnych (niski parametr `distance`), co z kolei powoduje, że można zakodować jedynie kilkanaście-kilkadziesiąt bitów bez zmian widocznych dla ludzkiego oka – ale tyle spokojnie wystarczy do naszych zastosowań (na przykład unikalny identyfikator użytkownika powinien zajmować najwyżej 32 bity).

Trochę historii

Pierwotnie w artykule miały znaleźć się również informacje o ukrywaniu danych w formacie .pdf, ale ostatecznie nie starczyło ani miejsca, ani czasu. W ramach ciekawostki można tu nawiązać do historii „Programisty”. Jako że wersja elektroniczna miała tendencję do wyciekania tuż po premierze, autor niniejszego artykułu uczestniczył swego czasu w nieformalnej rozmowe na temat możliwości fingerprintowania elektronicznych wydań magazynu. Powstał nawet prototyp! Niestety (albo na szczęście) system nigdy nie został wykorzystany w praktyce, ale można się zastanawiać, czy istnieją czasopisma, które praktykują takie działania.

3. UKRYWANIE INFORMACJI W CZYSTYM TEKŚCIE

Drugim ciekawym przypadkiem, który chciałbym omówić, jest ukrywanie danych w tekście. Będzie tu znacznie mniej matematyki niż w przypadku grafiki, ale zaprezentowane podejścia są równie sprytne.

Ukrywanie danych w tekście niepokoi mnie szczególnie dlatego, że jako programista spędzam większość swojego czasu, pracując właśnie z tekstem. Wydaje się, że nie da się nic ukryć bez zwracania uwagi – w końcu łatwo zauważyć dodatkowe znaki, słowa lub zdania. No i faktycznie – w większości byłaby to prawda… gdyby nie standard zwany Unicode.

Unicode to standard kodowania oraz zestaw znaków, z którego korzystają wszystkie współczesne programy przetwarzające tekst. Okazuje się też, że jest fantastycznie skomplikowany i bardzo trudno jest napisać program, który działa z nim dobrze w 100% przypadków. Ale nie będziemy tu narzekać na komplikacje kodowania znaków (konsorcjum Unicode i tak świetnie sobie poradziło z tematem, biorąc pod uwagę, jak chaotyczne są naturalne języki). Przeciwnie – wykorzystamy je do własnych celów.

3.1. Niewidzialne znaki

Jedna z prostszych metod ukrycia danych w tekście to użycie niewidzialnych znaków. W Unicode jest ich dostatek – na przykład znaki „Zero-width non-joiner” oraz „Zero-width space”. Oba znaki mają zerową szerokość (jak sama nazwa wskazuje), przez co są niewidoczne w większości sytuacji. Pokazują się one jedynie w niektórych specjalizowanych edytorach dla programistów, oraz oczywiście widać je przy przeglądaniu bajtów w hexedytorze.

6. W zasadzie tak zakodowane dane przetrwałyby też wydrukowanie obrazka w gazecie i zeskanowanie go…

Jak można ich użyć? Wystarczy skonwertować wiadomość, którą chcemy zakodować, na bity (używając np. poprzednio przedstawionej metody `bin2bits`), zamienić każde zero na „Zero-width non-joiner", a każdą jedynkę na „Zero-width space" i dokleić taki znacznik gdzieś do tekstu (może nawet kilka razy). Tak przedstawiony algorytm jest możliwy do zaimplementowania nawet w kilku linijkach Pythona:

Listing 15. Ukrywanie danych za pomocą niewidocznych znaków

```
def main():
  ZWSP = u'\u200B'
  ZWNJ = u'\u200C'

  input = sys.stdin.read()
  secret = sys.argv[1]
  encoded_secret = ''.join(
    ZWSP if bit == '0' else ZWNJ
    for bit in bin2bits(secret)
  )
  output = input[0] + encoded_secret + input[1:]
  sys.stdout.write(output.encode('utf-8'))
```

Czy to faktycznie działa? Łatwo sprawdzić w konsoli (Rysunek 11):

```
msm@europa /home/msm/demo
$ echo "plain text"
plain text
msm@europa /home/msm/demo
$ echo "plain text" | xxd
00000000: 706c 6169 6e20 7465 7874 0a
msm@europa /home/msm/demo
$ echo "plain text" | python hide.py magic
plain text
msm@europa /home/msm/demo
$ echo "plain text" | python hide.py magic | xxd
00000000: 70e2 808b e280 8ce2 808c e280 8be2 808c
00000010: e280 8ce2 808b e280 8ce2 808b e280 8ce2
00000020: 808c e280 8be2 808b e280 8be2 808b e280
00000030: 8ce2 808b e280 8ce2 808c e280 8be2 808b
00000040: e280 8ce2 808c e280 8ce2 808b e280 8ce2
00000050: 808c e280 8be2 808c e280 8be2 808b e280
00000060: 8ce2 808b e280 8ce2 808c e280 8be2 808b
00000070: e280 8be2 808c e280 8c6c 6169 6e20 7465
00000080: 7874 0a
```

Rysunek 11. Niewidzialne dane ukryte w tekście

Mamy tu po kolei cztery komendy. Najpierw wypisywany jest tekst „plain text". Wynik jest zgodny z oczekiwaniami. Następnie konwertujemy poprzedni tekst na bajty – ponownie bez zaskoczeń: każdy znak napisu odpowiada jednemu bajtowi. Dopiero w trzecim poleceniu uruchamiamy nasz skrypt. Nie ma on jednak żadnych widocznych efektów – tekst wygląda dokładnie tak samo jak na początku. Ukryte dane pojawiają się dopiero wtedy, kiedy przedstawimy wynik jako bajty – od razu widać, że danych jest więcej niż powinno być dla tak krótkiej wiadomości.

Tego typu metody są o tyle sprytne, że mało kto spodziewa się ukrytych danych w czymś tak prostym jak tekst. Nie jest to też czysta teoria – istnieją źródła, które mówią, że dokładnie ta metoda była wykorzystywana w praktyce do namierzania „źródeł" dziennikarzy. Są też udokumentowane historie wykorzystywania jej w równie poważnej sprawie, czyli… prywatnych forach gildii w grze EVE Online. Jak widać, niektórzy gracze podchodzą do grania bardzo poważnie.

Główną wadą jest to, że stosunkowo łatwo pozbyć się takiego znaku wodnego, jeśli ktoś wie, czego się spodziewać. Jest skończona liczba niewidzialnych znaków w Unicode i żaden z nich nie powinien się znajdować w zwykłym tekście. Wystarczy więc usunąć wszystkie podejrzane znaki i problem zostanie rozwiązany.

Homografy: a czy a?

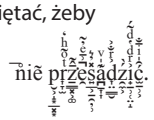
Pewną wariacją tej metody jest używanie tak zwanych homografów. Są to pary znaków, które wyglądają tak samo w większości czcionek, ale z punktu widzenia komputera są innymi znakami. Przykładem jest rosyjska litera „a" (bajty: 0xd0, 0xb0), która wygląda często dokładnie tak samo jak łacińska litera „a" (bajty: 0x61). Czyli zamiast dorzucać sztuczny znak wodny, możemy podmieniać niektóre znaki w tekście na inne. Ta metoda ma dużą zaletę – ciężko się jej pozbyć w sposób automatyczny (nie wiedząc dokładnie, jakie homografy są używane). Jedyny pewny sposób to „opcja nuklearna", czyli usunięcie wszystkich znaków spoza zakresu ASCII – ale w ten sposób usunięte zostaną też na przykład polskie znaki, co przysporzy sporo dodatkowej pracy.

W praktyce ten sposób jest wykorzystywany znacznie częściej do ataków na niewinnych klientów banków i znany pod nazwą IDN homograph attack[1]. Sztuczka polega na tym, że użytkownik znajduje w złośliwym e-mailu link do www.dobrybank.pl i nie spodziewa się, że jedna literka w adresie nie jest znakiem, na jaki wygląda… Dobrze, że przeglądarki starają się (z sukcesami) wykrywać takie ataki i bronić nas przed nimi, bo inaczej nie można by było już ufać nawet paskowi adresu.

1. https://en.wikipedia.org/wiki/IDN_homograph_attack

3.2. Prawdziwe i fałszywe ogonki

Jak wspominałem, Unicode bywa dziwne. Jedną z ciekawostek są tak zwane *combining characters*. To grupa specjalnych kodów, które nie reprezentują konkretnych liter alfabetu, a modyfikują litery przed nimi. Jednym z nich jest na przykład swojsko brzmiący U+0328 COMBINING OGONEK[7]. Jak nazwa wskazuje, służy on do dodawania ogonków do znaków – dzięki temu technicznie nic nie stoi na przeszkodzie, żeby poza ą i ę używać w tekście litery „d z ogonkiem" – d̨. Możemy w ten sposób dowolnie „wzbogacać" tekst – trzeba tylko pamiętać, żeby



Jednocześnie jednak większość typowych kombinacji ma swoje dedykowane znaki. Co oczywiście oznacza, że „ą" można zapisać na dwa sposoby – ze znakiem łączącym oraz bezpośrednio. Jak zwykle – takie niejednoznaczności możemy wykorzystać do naszych celów (patrz też: ramka o homografach). Wystarczy np. znaleźć wszystkie znaki „ę" w wiadomości, zmienić kodowaną wiadomość na bity, i jeśli kolejny bit to „0", zostawić oryginalne „ę", a w przeciwnym wypadku zamienić na „e" z doklejonym ogonkiem.

Jaka jest zaleta tej metody? Głównie taka, że wszystkie szanujące się edytory muszą obsługiwać poprawnie *combining characters*. Jeszcze ciekawsze jest to, że technicznie te dwa znaki są sobie równoważne. To znaczy, że poprawnym zachowaniem dla języka programowania byłoby zwrócenie `true` dla porównania `"ę"` z `"e[COMBINING OGONEK]"`.

Wadą jest to, że relatywnie łatwo pozbyć się takich artefaktów, używając normalizacji Unicode – ale trzeba wcześniej wiedzieć o tym, że istnieje.

3.3. Zastosowanie dla synonimów

Na koniec chyba najbardziej trwała metoda (ale jednocześnie najtrudniejsza w implementacji). Co jeśli zamiast koncentrować się na poszczególnych słowach, będziemy podmieniać całe słowa? Na przykład słowo „szybko" można w większości przypadków zamie-

7. https://codepoints.net/U+0328

nić na „błyskawicznie". Można napisać prosty program wyświetlający wszystkie możliwe podmiany:

Listing 16. Wyszukiwarka synonimów

```
import sys, re
ansi = {
  "GREEN": "",
  "RESET": "",
}

def replace(test, word, part):
  repl = '{' + ','.join(
    "\x1b[32m{}\x1b[39m".format(p, **ansi) for p in part
    ) + '}'  # kolorowanie na zielono
  return re.sub(
    "\\b{}\\b".format(word), repl, test)

def replace_all(test, word, part):
  test = replace(test, word, part)
  test = replace(test, word + 's', part)
  if word.endswith('e'):
    test = replace(test, word + 'e', part)
  else:

    test = replace(test, word + 'ed', part)
  return test

def main():
  synonyms = eval(open('synonyms.txt').read())
  test = sys.stdin.read().lower()
  for part in synonyms:
    for synonym in part:
      test = replace_all(test, synonym, part)
  print test

main()
```

Przy odpowiednim słowniku synonimów[8] można osiągnąć nawet dobre rezultaty (Rysunek 12). Na trzecim przykładzie widać też jednak, że naiwne podmienianie słów potrafi dać bezsensowne rezultaty, więc trzeba z nim uważać.

8. Ja do testów skorzystałem z https://www.englisch-hilfen.de/en/words/synonyms.htm, ale w Internecie znajduje się wiele bogatszych źródeł.

```
xmsm@xmsm1 /tmp/test
$ cat test.txt
you did a great job on your report.
when did you start?
you won't fool me.
xmsm@xmsm1 /tmp/test
$ cat test.txt | python syn.py
you did a {fantastic,great} job on your report.
when did you {begin,start}?
you won't {fool,idiot} me.
```

Rysunek 12. Trzy miejsca na ukrycie informacji

W tym przypadku można zapisać ledwo trzy bity, ale w przypadku dłuższej wiadomości i lepszego słownika (np. mając po 4-5 synonimów do jednego słowa) spokojnie doszlibyśmy do potrzebnych nam około 30 bitów.

4. PODSUMOWANE

Powyższym artykułem chciałem zwrócić uwagę na to, jak łatwo ukryć dane w trudny do wykrycia i usunięcia sposób. Myślę, że może się to przydać zarówno osobom, które próbują zabezpieczyć swoje informacje, jak i tym, którzy mają dobry powód, żeby je nagłaśniać. Prywatnie nie stoję po żadnej stronie, za to jestem pod wrażeniem złożoności nawet najprostszych otaczających nas cyfrowych elementów, jak obrazki i tekst.

JAROSŁAW JEDYNAK
msm@tailcall.net

Rozpoczynał karierę jako programista, wyspecjalizował się w security. Do niedawna jako analityk malware w CERT Polska zajmował się między innymi inżynierią wsteczną złośliwego oprogramowania i automatyzacją jego analizy. Obecnie pracuje jako security engineer w Google, spędza czas na reverse-engineeringu oraz prowadzi badania nad nowymi sposobami wykrywania złośliwego oprogramowania. W wolnym czasie programuje do szuflady, gra w konkursy CTF i ogląda koty w Internecie.