Praktyczna kryptografia: Szyfry blokowe

„Ktokolwiek, poczynając od najbardziej bezmyślnego amatora, na najlepszym kryptografie kończąc, może stworzyć algorytm, którego on sam nie będzie mógł złamać". Właśnie tak brzmi Prawo Schneiera – amerykańskiego badacza bezpieczeństwa i kryptografa. Postaramy się je zweryfikować i przekonać przy tym czytelników, że nawet korzystając z silnych, nowoczesnych algorytmów, można popełnić błąd, który kompletnie rujnuje bezpieczeństwo całej aplikacji.

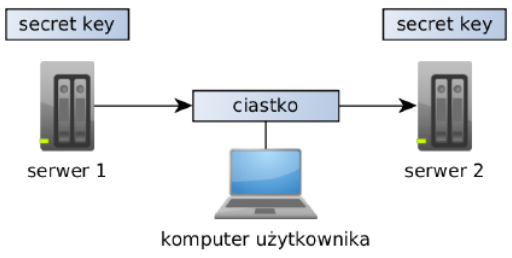
Wspomniane prawo odnosi się konkretnie do tworzenia własnych kryptosystemów, ale w praktyce równie mocno dotyczy całych programów, które korzystają bezpośrednio z prymitywów kryptograficznych. Jest to szczególnie niepokojące, bo o ile błędy w działaniu zostaną wyłapane przez testy albo przez użytkowników aplikacji, to słaba kryptografia nie będzie budziła żadnych podejrzeń – do momentu, kiedy ktoś nie wykorzysta jej, żeby okraść naszych klientów lub wykraść nasze dane.

W tym artykule skoncentrujemy się na szyfrach blokowych. Jako przykładu będziemy używać krótkich programów napisanych w języku Python (2.7), ale wszystkie przedstawione ataki są niezależne od języka. Co ważniejsze – dla demonstracji używamy AES, ale ataki byłyby dokładnie takie same na każdy inny szyfr blokowy (np. 3DES, Blowfish, Serpent, a nawet przyszłe, jeszcze nie wymyślone algorytmy).

SZYFROWANIE NIE SŁUŻY DO UWIERZYTELNIANIA

Wstęp

Wyobraźmy sobie następującą sytuację: mamy dwie niezależne aplikacje webowe, które z jakiegoś powodu muszą uwierzytelniać się sobie wzajemnie. Na przykład, jedna strona to forum internetowe, a druga to chat internetowy[1]. Jeśli chcemy, żeby można było się logować na chat za pomocą konta z forum, jedną z najprostszych opcji jest ciastko (lub token) współdzielone między tymi dwoma aplikacjami. Pierwszy serwer wystawia jakieś ciastko/token, a użytkownik później używa tego ciastka, żeby wykonać jakieś akcje na drugim serwerze.



Rysunek 1. Serwer 1 przekazuje zaszyfrowane dane użytkownikowi, a użytkownik do serwera 2

1. Przykład inspirowany forum 4progrmmers.net i chatem, który był tam instalowany przez autora niniejszego artykułu. Chociaż oczywiście bez popełnienia opisanych tu błędów.

Potrzebujemy jakiegoś prostego formatu serializacji do współdzielenia danych – standardem obecnie jest JSON (jeśli ktoś woli XML, urlencoding lub inne – omawiane metody, po drobnych modyfikacjach, zadziałają dla dowolnego generycznego sposobu serializacji).

```
def serialize_cookie_0(name, has_admin):
  return json.dumps({
    'name': name,
    'has_admin': has_admin
  })

def deserialize_cookie_0(cookie):
  return json.loads(cookie)
```

Przetestujmy kod:

```
cookie_msm = serialize_cookie_0('msm', False)
print cookie_msm
print deserialize_cookie_0(cookie_msm)
# {"name": "msm", "has_admin": false}
# {u'name': u'msm', u'has_admin': False}

cookie_monk = serialize_cookie_0('monk', False)
print cookie_monk
print deserialize_cookie_0(cookie_monk)
# {"name": "monk", "has_admin": false}
# {u'name': u'monk', u'has_admin': False}
```

I tak zakodowane ciastka wysyłamy użytkownikowi. Czy jest tu jakiś problem? Oczywiście, ktoś, kto wie, jak działają ciastka w przeglądarkach, już po kilku sekundach rozpozna zagrożenie – użytkownik ma pełną władzę nad własną przeglądarką, więc może po prostu zmienić sobie wartość swojego ciastka na:

```
{"name": "msm", "has_admin": true}
```

A następnie zalogować się jako administrator na chat. Jest to zdecydowanie niepożądane – użytkownik powinien mieć dostęp tylko do swojego konta. W jaki sposób temu zapobiec?

Jeden z pomysłów, na który łatwo wpaść i który jest czasami (niestety) stosowany: zaszyfrować ciastko kluczem dzielonym między dwoma serwerami.

Trochę teorii o szyfrowaniu

Napiszmy więc odpowiedni kod. JSON pozostanie bez zmian (już do końca artykułu w zasadzie), natomiast dopiszmy odpowiednie szyfrowanie do serializacji.

Najpierw kilka koniecznych słów teorii. Rozróżniamy dwa rodzaje szyfrów symetrycznych:

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

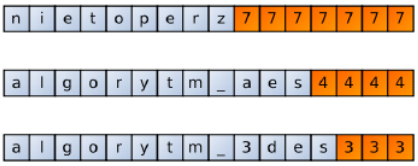
Patroni medialni: sekurak.pl rozwal.to

» Szyfry strumieniowe, gdzie szyfrujemy dane bit po bicie (w praktyce bajt po bajcie).
» Szyfry blokowe, które szyfrują dane „blokami", czyli np. po 16 bajtów. Długość szyfrowanych danych musi być wielokrotnością długości bloku (np. 32, 48… bajtów).

Większość stosowanych obecnie szyfrów jest blokowa – w szczególności AES, jeden z najpopularniejszych algorytmów szyfrowania, działa blokowo. Wiąże się z tym fakt, że jeśli mamy dane o długości np. 7 bajtów, przed zaszyfrowaniem musimy je w jakiś sposób „wydłużyć" do wielokrotności długości bloku. Operacja ta profesjonalnie nosi nazwę „padding" (czasami tłumaczona jako „wyrównanie"/„wypełnienie") i musi być odwracalna – żeby można było przy deszyfrowaniu bezstratnie ją odwrócić. Proste ćwiczenie dla chętnych programistów – wymyślić samodzielnie, jak coś takiego zrobić. Rozwiązań jest wiele[2], ale najpopularniejszym z nich jest PKCS#7 padding.

Zasada działania jest prosta. Jeśli chcemy mieć blok o długości 16 bajtów, to:

» Jeśli do wielokrotności 16 bajtów brakuje 1 bajta, dodajemy na koniec danych `"\x01"` (jeden bajt o wartości 1).
» Jeśli do wielokrotności 16 bajtów brakuje 2 bajtów, dodajemy `"\x02\x02"`.
» Jeśli do wielokrotności 16 bajtów brakuje 3 bajtów, dodajemy `"\x03\x03\x03"`.
» …
» Jeśli długość danych już jest wielokrotnością 16 bajtów, na koniec dodajemy 16 bajtów `"\x10"` (16 bajtów równych 16 – inaczej dekodowanie niestety nie byłoby jednoznaczne).



Rysunek 2. Sposób działania paddingu PKCS#7. Dodawane jest N bajtów o wartości N

Implementacja również nie jest skomplikowana:

```
def pkcs7_pad(data):
  pad_len = 16 - (len(data) % 16)
  pad_len = 16 if pad_len == 0 else pad_len
  return data + chr(pad_len) * pad_len

def pkcs7_unpad(data):
  pad_len = ord(data[-1])
  data, pad = data[:-pad_len], data[-pad_len:]
  assert all(c == pad[0] for c in pad)
  return data
```

Testy:

```
msg = 'msm_lubi_koty'
padded_msg = pkcs7_pad(msg)
print repr(padded_msg)
print pkcs7_unpad(padded_msg)
# 'msm_lubi_koty\x03\x03\x03'
# msm_lubi_koty
```

Szyfrowanie 1: ECB

Ok, wyobraźmy sobie, że szef kazał nam zaimplementować takie właśnie zabezpieczenie dla ciastek. Czas zacząć pisać kod:

```
from Crypto.Cipher import AES

# openssl rand -base64 16 - 16 bajtów entropii powinno uchronić
przed atakami brute-force do końca świata
SHARED_KEY = 'KDbWBsg11cC8vAAmIT/+Ig==''.decode('base64')

def serialize_cookie_1(name, has_admin):
  raw_cookie = json.dumps({
    'name': name,
    'has_admin': has_admin
  })
  cipher = AES.new(SHARED_KEY)
  raw = pkcs7_pad(raw_cookie)
  return cipher.encrypt(raw).encode('hex')

def deserialize_cookie_1(cookie):
  cipher = AES.new(SHARED_KEY)
  raw_cookie = cipher.decrypt(cookie.decode('hex'))
  cookie = pkcs7_unpad(raw_cookie)
  return json.loads(cookie)
```

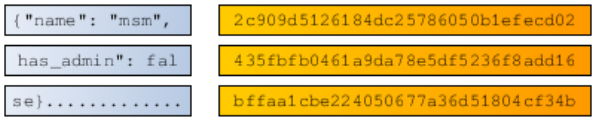
Zaznaczę tutaj, że ten kod jest poprawny[3] – w tym sensie, że programista nie popełnił żadnego błędu przy implementacji szyfrowania, klucz jest silny, padding wykonany poprawnie, a wszystko inne korzysta z wartości domyślnych biblioteki PyCrypto.

Testy kodu:

```
cookie = serialize_cookie_1('msm', False)
print cookie
print deserialize_cookie_1(cookie)
# fbbb0abb8df1565687a99ca5c3990a4bc37d2c03494118c1574af74fa26cfaf8
# {'user_name': 'msm', 'has_admin': 'False'}
```

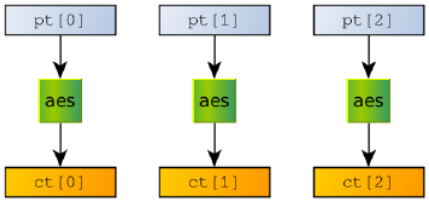
Nasze ciastko jest bezpieczne! Użytkownik otrzymuje tylko zaszyfrowany ciąg `fbbb0abb8df1565687a99ca5c3990a4bc37d2c03494118c1574af74fa26cfaf8`, którego nie jest w stanie zdeszyfrować (w koncu to AES, prawda?).

No, w sumie... to nie. Sam z siebie AES jest silnym algorytmem, jak na razie nie złamanym. Ale po co miałby deszyfrować, skoro i tak wie, co się w ciastku znajduje (każdy zna swoją nazwę użytkownika i wie, czy jest adminem). Popatrzmy więc dokładniej, co się dzieje – w jaki sposób działa szyfrowanie w tym momencie? Domyślnym sposobem działania wielu bibliotek szyfrujących jest, niestety, tak zwany tryb ECB (Electronic Codebook). Działa on w najprostszy możliwy sposób – każdy blok jest szyfrowany niezależnie od siebie, a wyniki sklejane ze sobą:



Rysunek 3. Wiadomość zaszyfrowana w trybie ECB, z podziałem na bloki

Co można przedstawić w postaci grafu:



Rysunek 4. Sposób działania trybu ECB

2. https://en.wikipedia.org/wiki/Padding_(cryptography)

3. A przynajmniej, jeśli są w nim jakieś błędy, są niezamierzone – autor też jest człowiekiem, a w końcu przewodnią myślą artykułu jest „przy kryptografii łatwo się pomylić".

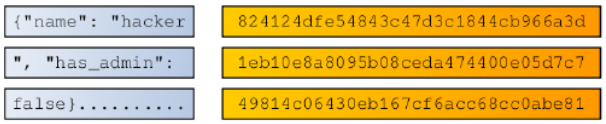
Pseudokod tej metody jest bardzo prosty – jeśli oznaczymy przez `pt[0]`, `pt[1]`, `pt[2]`… kolejne bloki plaintextu, przez `ct[0]`, `ct[1]`, `ct[2]`… kolejne bloki ciphertextu, a przez `aes()` operację szyfrowania, mamy:

```
ct[0] = aes(pt[0])
ct[1] = aes(pt[1])
ct[2] = aes(pt[2])
```

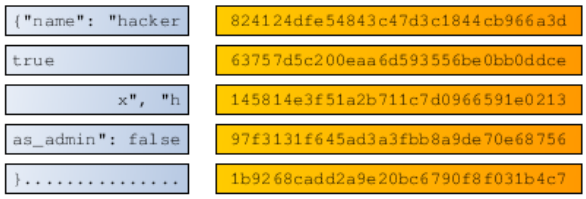
Czyli najpierw szyfrujemy `,{"name": "msm",'`, następnie `,has_admin": fal'` itd.

Czy jest tu jakiś problem? Oczywiście jest, dlatego też powstał ten artykuł.

Jak to na forum, użytkownik może założyć konto z dowolną (w miarę sensowną) nazwą, więc wyobraźmy sobie, że nasz atakujący założył konto z nazwą `"hacker"`. Nie zadowoliło go to jednak, więc założył kolejne, tym razem o nazwie `"hackertrue\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20x"` (gdzie `\x20` oznacza spację). Jak wyglądać będą ciastka tych użytkowników?



Rysunek 4. Zaszyfrowane ciastko użytkownika hacker



Rysunek 5. Zaszyfrowane ciastko użytkownika hackertrue.....x

Hmm... jak wcześniej widzieliśmy, użytkownik zna i może „bezkarnie" edytować swoje tokeny. Nic więc nie powstrzymuje go przed „zabawą nożyczkami" i sklejeniem bloków szyfru z poprzednich tokenów w taki sposób, aby utworzyć zupełnie nowy token:

Rysunek 6. Ciastko-potwór frankensteina, poskładane z dwóch poprzednich

I faktycznie:

```
print deserialize_cookie_1(
  '824124dfe54843c47d3c1844cb966a3d'+
  '1eb10e8a8095b08ceda474400e05d7c7'+
  '63757d5c200eaa6d593556be0bb0ddce'+
  '1b9268cadd2a9e20bc6790f8f031b4c7')
# {u'name': u'hacker', u'has_admin': True}
```

Co tu się właściwie stało? Atakujący w taki sposób skopiował kawałki tokena z obu użytkowników, że wygenerował sobie „sztuczny" token dający mu prawa admina.

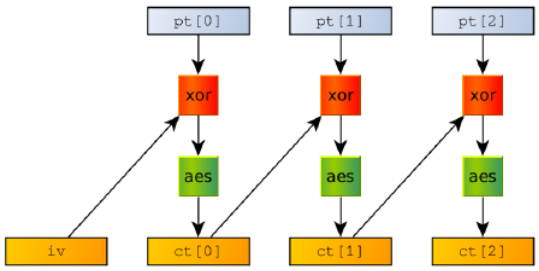
Kto zawinił? Ostatecznie na to pytanie odpowiemy pod koniec tej części artykułu. Krótka odpowiedź brzmi oczywiście: programista – ale jest to niezbyt konstruktywne stwierdzenie. Wiele osób, które czytało trochę o kryptografii, odruchowo odpowie: „błędem było użycie szyfrowania w trybie ECB", ale w tym przypadku nie będzie to słuszna uwaga. Owszem, stosowanie go powoduje bardzo wiele problemów i nie powinien być używany prawie nigdy[4], ale tym razem nasze problemy spowodowało co innego. Jak przekonamy się za chwilę, podobne ataki można wykonać i na inne tryby.

Złam to sam: ECB

Jeśli ktoś czuje się na siłach, by zmierzyć się z trybem ECB samodzielnie, zapraszamy do rozwiązania zadania dostępnego pod adresem: https://var.tailcall.net/ecb.

Szyfrowanie 2: CBC

Terminy gonią nadal, a szef jest niezadowolony z bezpieczeństwa strony. Sfrustrowany problemami z trybem ECB programista uruchamia przeglądarkę i za pomocą swojej ulubionej wyszukiwarki szuka alternatywy. W ten sposób dowie się o trybie CBC – który jest najpopularniejszym trybem stosowanym w szyfrach blokowych. Działa on tak:



Rysunek 7. Sposób działania trybu CBC

Lub bardziej programistycznie, zapisując (operator ^ oznacza operację xor):

```
ct[0] = aes(pt[0] ^ iv)
ct[1] = aes(pt[1] ^ ct[0])
ct[2] = aes(pt[2] ^ ct[1])
...
```

Jak widać, dochodzi tu dodatkowy element – tak zwany IV (wektor inicjalizacyjny). Jest to jeden blok składający się w całości z losowych danych, którego celem jest wprowadzenie losowości do ciphertextu. Podczas deszyfrowania kolejne bloki szyfrowanej wiadomości są xorowane z wynikami deszyfrowania poprzedniego bloku, a dla bloku pierwszego z IV, co uniemożliwia trywialne ataki, jak ten pokazany przed chwilą, ponieważ szyfrogram dla danego bloku zależy już nie tylko od treści wiadomości, ale także od poprzedniego zaszyfrowanego bloku.

Czytając o trybie CBC, nasz programista dowie się, że nie cierpi on na takie problemy jak ECB i jest powszechnie uznawany za bezpieczny (bo to prawda – jest bezpieczny). Szybko przerobi więc

4. Można się zastanawiać, dlaczego tryb ECB tak często jest domyślnym trybem. Jako programiści zazwyczaj oczekujemy, że biblioteka ma sensowne wartości domyślne i używając jej na domyślnych ustawieniach, nie strzelimy sobie w stopę. Jak widać, w kryptografii nie zawsze tak jest.

odpowiednie funkcje na nową, lepszą wersję:

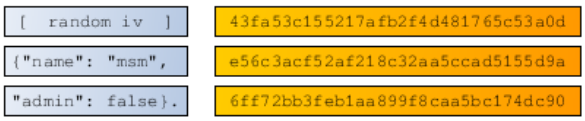
```
from Crypto import Random

def serialize_cookie_2(name, has_admin):
  raw_cookie = json.dumps({
    'name': name,
    'admin': has_admin
  })
  iv = Random.new().read( AES.block_size )
  cipher = AES.new(SHARED_KEY, AES.MODE_CBC, iv)
  raw = pkcs7_pad(raw_cookie)
  return (iv + cipher.encrypt(raw)).encode('hex')

def deserialize_cookie_2(cookie):
  cookie = cookie.decode('hex')
  iv, cookie = cookie[:16], cookie[16:]
  cipher = AES.new(SHARED_KEY, AES.MODE_CBC, iv)
  raw_cookie = cipher.decrypt(cookie)
  cookie = pkcs7_unpad(raw_cookie)
  return json.loads(cookie)
```

Ponownie podkreślmy – szyfrowanie jest zaimplementowane poprawnie. IV jest generowany za pomocą generatora liczb prawdziwie losowych, padding wykonano zgodnie ze sztuką itd. Celem artykułu nie jest pokazanie, jak można popełnić błąd podczas implementacji (a można, na naprawdę wiele sposobów). Sprawdźmy więc dla pewności, czy algorytm daje oczekiwane wyniki:

```
cookie_msm = serialize_cookie_2('msm', False)
print cookie_msm
print deserialize_cookie_2(cookie_msm)
# 43fa53c155217afb2f4d481765c53a0de56c
  3acf52af218c32aa5ccad5155d9a6ff72bb3fe
  b1aa899f8caa5bc174dc90
# {u'name': u'msm', u'admin': False}
```



Rysunek 8. Wiadomość zaszyfrowana w trybie CBC, z podziałem na bloki

Działa. Programista jest szczęśliwy, szef daje premię, a haker gryzie paznokcie ze złości.

Ale czy na pewno? W końcu dalej użytkownik może dowolnie edytować swoje cookie. Zobaczmy, co się stanie, jeśli trochę namieszamy w zaszyfrowanym ciastku. Stwórzmy najpierw ciastko dla naszego hakera:

```
print serialize_cookie_2('hacker', False)
# 1cd0539e5d370993a62b7c30ffe72f81e6ed642301d
  a252c4e900fab1d95680d4e0f6ee8e336ccd3296801
  8bbd65cddb8292ead021be75767ad03f9f96e76e83
```

Zmieńmy trzeci bajt w zaszyfrowanym ciastku i zobaczmy, co się stanie podczas deszyfrowania. Logika sugeruje, że powinny wyjść nam kompletne śmieci:

```
cookie = '1cd0519e5d370993a62b7c30ffe72f81e6ed642301da2
52c4e900fab1d95680d4e0f6ee8e336ccd32968018bbd65cddb8292
ead021be75767ad03f9f96e76e83'
# trzeci bajt zmieniony z 53 na 51
print deserialize_cookie_2(cookie)
# {u'admin': False, u'lame': u'hacker'}
```

Zaraz, co się stało? Dlaczego deszyfrowanie powiodło się? Dlaczego klucz „name" zmienił się w „lame"? Wiele pytań, czas na odpowiedzi – spójrzmy jeszcze raz na pseudokod przedstawiający sposób działania CBC:

```
ct[0] = aes(pt[0] ^ iv)
ct[1] = aes(pt[1] ^ ct[0])
ct[2] = aes(pt[2] ^ ct[1])
...
```

No tak – na początku szyfrogramu znajduje się IV, a jego modyfikacja wpływa tylko na pierwszy blok. W dodatku wpływa w sposób przewidywalny – konkretnie jeśli zamienimy `iv` na `iv ^ x` (iv xorowany z wartością x), po deszyfrowaniu otrzymamy `pt ^ x` (czyli plaintext xorowany z x). I faktycznie, zobaczmy, co stało się z naszym ciastkiem:

```
plaintext: '{"name": "hacker", "admin": false}'
xor:          2
wynik:     '{"lame": "hacker", "admin": false}'
```

Konkretnie, „n" zamieniło się na „l", ponieważ znak „n" xorowany z liczbą 2 daje znak „l":

```
print chr(ord('n') ^ 2)  # 'l'
```

Łatwo domyślić się, co można zrobić w ten sposób – skoro dalej możemy modyfikować zawartość ciastka, to wystarczy wykonać kilka obliczeń i można będzie łatwo wejść na chat jako dowolny użytkownik. Spróbujmy zaimplementować atak:

```
cookie = serialize_cookie_2('haker', False).decode('hex')
change = '\x00' * 10 + xor('haker', 'admin') + '\x00'
cookie = xor(cookie[:16], change) + cookie[16:]
print deserialize_cookie_2(cookie.encode('hex'))
# {u'admin': False, u'name': u'admin'}
```

Udało się – możemy teraz udawać prawie dowolnego użytkownika. Jest tylko jeden problem – dalej nie mamy praw admina, jedynie jego nick. Niestety, zmiana „false" na „true" będzie trochę trudniejsza – wróćmy raz jeszcze do pseudokodu:

```
ct[0] = aes(pt[0] ^ iv)
ct[1] = aes(pt[1] ^ ct[0])
ct[2] = aes(pt[2] ^ ct[1])
```

Jeśli zmienimy coś w `ct[1]`, to owszem, `ct[2]` zmieni się w taki sposób jak byśmy chcieli, ale podczas deszyfrowania drugi blok zmieni się w losowe (w znaczeniu „nieprzewidywalne") dane. Na przykład, jeśli zaszyfrujemy `{"name": "admin", "admin": false}`, otrzymamy ciastko:

```
9fca5cdfcf45de713fc0fa02887af5c1a794c1f3330a25710a651c51905
0b99352034310ff53d07bab13597f3ec1f6b591c930f085bf4556b21dfd
e16387d37a
```

Jeśli teraz zmienimy jeden bajt w drugim bloku (czyli `ct[0]`), zdeszyfruje się ono do:

$$\text{Æu}\llcorner\text{A'E+}\ \text{❄*!, "atmin": false}\}$$

Faktycznie, jakiś znak się zmienił („admin" na „atmin"), ale cały pierwszy blok został zniszczony – prawidłowy JSON to już nie jest. Oczywiście, skoro o tym wspominam, jest i sposób na to – wystarczy, że spreparujemy dane tak, żeby śmieci trafiły gdzieś, gdzie są „niegroźne". Spróbujmy – tym razem nasz haker musi zarejestrować konto o nazwie np. `"hacker x"`.

[random iv]	593361eeafe757e421836c0a01c3d6ea
{"name": "hacker	0f53fdc786a629247ef60cf264b271af
x	f5b035ebd307311ed6fdffbf8cb7e246
", "admin": fals	28092ff40ef6f05620880acedd1317a7
e}..............	ad98aa5050c0c81e1043300c24bde345

Rysunek 9. Zaszyfrowane ciastko dla nowego użytkownika

Co nam to dało? Że teraz trzeci blok (ten zawierający ciąg spacji oraz „x") może w zasadzie zawierać prawie dowolne dane – jedyne ograniczenie jest takie, że bajty muszą poprawnie zdekodować się jako utf-8 oraz nie zawierać znaku cudzysłowu („) – żeby stanowiły poprawny JSON.

Ten algorytm można zapisać np. tak:

```
from random import randint
while True:
  try:
    cookie = serialize_cookie_2('hacker              x', False)
    cookie = [ord(c) for c in cookie.decode('hex')]
    cookie[44] ^= ord('f') ^ ord(' ')  # zmiana znaku "f" na " "
    cookie[45] ^= ord('a') ^ ord('t')  # zmiana znaku "a" na "t"
    cookie[46] ^= ord('l') ^ ord('r')  # zmiana znaku "l" na "r"
    cookie[47] ^= ord('s') ^ ord('u')  # zmiana znaku "s" na "u"
    cookie = ''.join(chr(c) for c in cookie).encode('hex')
    deserialize_cookie_2(cookie)
  except:
    pass  # deserializacja nie powiodła się
  else:
    print cookie  # udało się
    print deserialize_cookie_2(cookie)
    break
```

Prawdziwy atakujący oczywiście nie dysponowałby funkcją `deserialize_cookie` bezpośrednio – musiałby sprawdzać, czy atak się udał, wchodząc na naszą stronę z ustawionym ciastkiem i obserwując, czy występuje błąd.

Po wielu próbach atak się udaje – otrzymujemy ciastko, które poprawnie dekoduje się:

```
cookie = "bfe7c87cb5df56f225cef6d8d4a1ad61cbe017e14b
2dd767d3b31772148b593ddb34c1c042b4a781c0fa37b4ff490c
b40d78cc15644a2607af1387ed8fc63af2a57c08e03a2fe35417
655a39d61bc3c4"
print deserialize_cookie_2(cookie)
# {u'admin': True, u'name': u'hacker8+!E.2C1a)*>>#BJ'}
```

Nazwa użytkownika `hacker8+!E.2C1a)*>>#BJ` nie jest może idealna, ale też nie przeszkadza – ważne jest, że mamy prawa administratora.

I ponownie, nasza metoda zawiodła. Jak to się stało, że AES/CBC, metoda wykorzystywana przez wiele silnych protokołów kryptograficznych, została przez nas właśnie bez większego problemu „złamana"? Do tego pytania wrócimy pod koniec rozdziału.

Swoją drogą – atak, który właśnie wykonaliśmy, jest na tyle popularny, że ma nawet swoją nazwę. Konkretnie mówi się o „bit flippingu" albo „byte flippingu"[5] [6].

Złam to sam: CBC

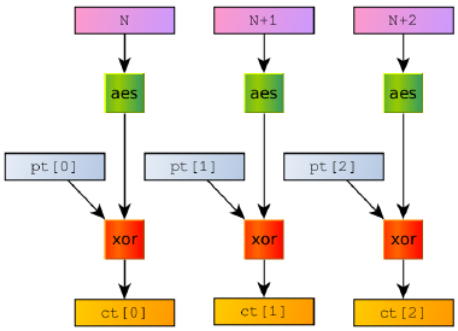
Jeśli komuś tryb CBC niestraszny, może zmierzyć się z praktycznym zadaniem, wchodząc na stronę https://var.tailcall.net/cbc.

5. Teoretycznie „bit-flipping" dokładniej opisuje charakter ataku, ale termin ten ma już wiele innych znaczeń, więc prywatnie autorzy preferują jednoznaczny termin „byte flipping".
6. https://en.wikipedia.org/wiki/Bit-flipping_attack

Szyfrowanie 3: CTR

A na razie nasz nieszczęśliwy programista musi ponownie naprawiać kod. Tryby ECB i CBC go zawiodły – ale wyczytał w sieci, że jest jeszcze jeden, silniejszy: tryb CTR[7].

Działa on tak:



Rysunek 10. Sposób działania trybu CTR

W pseudokodzie:

```
ct[0] = aes(n+0) ^ pt[0]
ct[1] = aes(n+1) ^ pt[1]
ct[2] = aes(n+2) ^ pt[2]
...
```

Co tu się dzieje? Nie szyfrujemy tu plaintextu, a jakąś losowo wybraną liczbę N (tzw. nonce), następnie N+1, N+2, N+3 itd. Wynik tego szyfrowania (zwany keystream) tylko xorujemy z plaintextem, żeby otrzymać ciphertext.

Dlaczego w ogóle coś takiego robić? Okazuje się, że szyfrowanie w ten sposób jest równie bezpieczne jak CBC (a nawet minimalnie bardziej), a ma nad nim kilka zalet:

- Można zrównoleglić szyfrowanie i deszyfrowanie, co może mieć znaczenie przy współczesnych wieloprocesorowych maszynach.
- Padding staje się niepotrzebny – długość plaintextu nie musi być wielokrotnością długości bloku. Formalnie rzecz ujmując, zmieniamy szyfr blokowy w szyfr strumieniowy.

Czy ma jakieś wady? Głównie jedną – jest nowszy, gorzej wspierany w bibliotekach i trudniej go dobrze użyć. Nie brzmi to zbyt problematycznie, ale w praktyce błędy bezpieczeństwa w aplikacjach spowodowane złym użyciem trybu CTR są dużo częstsze niż w przypadku CBC. Wynika to z tego, że nonce (skrót od *number used once*) naprawdę musi być unikalne. W momencie, gdy nonce powtórzy się chociaż raz, całe bezpieczeństwo szyfrowania spada do zera – atakujący może łatwo odzyskać *keystream*, na przykład xorując swój ciphertext ze swoim plaintextem (jeśli go zna), albo wykonać jeden z wielu podobnych ataków.

Zaimplementujmy więc tryb CTR:

```
def int_to_16bytes(n):
  return '{:032x}'.format(n)[-32:].decode('hex')

def counter(nonce):
```

7. Faktycznie, tryb CTR jest w pewien sposób matematycznie silniejszy niż tryb CBC. Jednak różnica w poziomie bezpieczeństwa jest w praktyce pomijalna (rozbija się o teoretyczne pojęcie *rozróżnialności od losowych danych*, przy ponad 2^{64} bajtów zaszyfrowanych jednym kluczem), za to dużo łatwiej krytycznie pomylić się w implementacji (np. wielokrotnie używając jednego nonce), a konsekwencje pomyłki są gorsze. Z tego powodu trudno wskazać jeden z tych dwóch trybów jako obiektywnie bezpieczniejszy.

```
  return Counter.new(128, initial_value=nonce)

from Crypto.Random import random
from Crypto.Util import Counter

def serialize_cookie_3(name, has_admin):
  raw_cookie = json.dumps({
    'name': name,
    'admin': has_admin
  })
  raw = pkcs7_pad(raw_cookie)
  nonce = random.randint(0, 2**128)
  cipher = AES.new(SHARED_KEY, AES.MODE_CTR,
  counter=counter(nonce))
  return (int_to_16bytes(nonce) + cipher.encrypt(raw)).
  encode('hex')

def deserialize_cookie_3(cookie):
  nonce, cookie = int(cookie[:32], 16), cookie[32:].decode('hex')
  cipher = AES.new(SHARED_KEY, AES.MODE_CTR,
  counter=counter(nonce))
  raw_cookie = cipher.decrypt(cookie)
  cookie = pkcs7_unpad(raw_cookie)
  return json.loads(cookie)
```

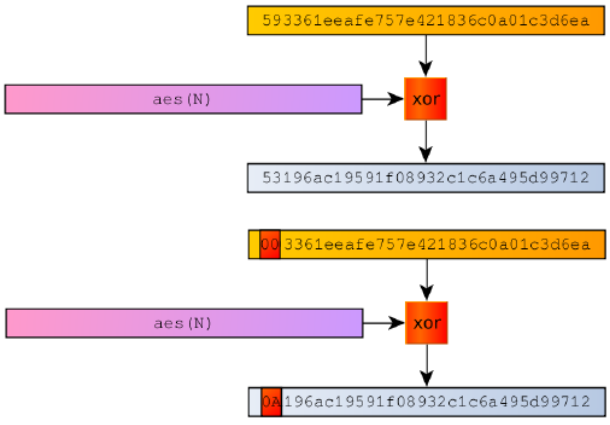
Jak widać, mieliśmy rację, że w CTR łatwo się pomylić – trzeba było napisać całkiem sporo kodu jak na szyfrowanie standardową metodą. A potencjalnie w każdej linijce kodu czai się błąd…[8]

Upewnijmy się, że przynajmniej funkcje działają zgodnie z naszymi oczekiwaniami:

```
cookie = serialize_cookie_3('msm', False)
print cookie
print deserialize_cookie_3(cookie)
# ab56feafd4a1a2762c24c1ee7e4540413150551507ca8f5c7717e
  516addde3d4cfb8a3d01e778961105e67a2e56af458
# {u'admin': False, u'name': u'admin'}
```

Już dwa razy udało nam się „złamać" program wykorzystujący jakąś metodę szyfrowania. Czy i tym razem atak również będzie możliwy? Jeśli tak, jak trudny będzie? Zachęcam wszystkich do chwili namysłu – co stanie się, jeśli zmodyfikujemy fragment ciphertextu?

No to popatrzmy:



Rysunek 11. Efekt modyfikacji szyfrogramu zaszyfrowanego w trybie CTR

No tak. Ostatecznie szyfrowanie i deszyfrowanie polega tylko na xorowaniu ze strumieniem klucza, więc każda zmiana jednego oktetu w szyfrogramie spowoduje deterministyczną i łatwą do przewidzenia zmianę jednego oktetu w deszyfrowanym plaintexcie. Korzystając z tego faktu, jeśli wiemy, jakie dane są zaszyfrowane, możemy je trywialnie zmodyfikować na cokolwiek tylko chcemy. Atak jest bardzo prosty, ale dla porządku zaimplementujmy go:

```
print deserialize_cookie_3(cookie)
# {u'admin': False, u'name': u'msm'}
cookie = cookie.decode('hex')  # get raw bytes

# (...32 bytes...)  <"admin": false}.>
old = '\x00' * 32 + ".........false..".replace('.', '\x00')
new = '\x00' * 32 + ".........true ..".replace('.', '\x00')

mod = xor(old, new)  # remove old bits, add new ones
new_cookie = xor(cookie, mod)  # apply modifier
new_cookie = new_cookie.encode('hex')  # go back to hex
print deserialize_cookie_3(new_cookie)
# {u'admin': True, u'name': u'msm'}
```

W tym momencie nasz programista prawdopodobnie kompletnie się załamuje. Kolejna metoda szyfrowania nie sprawdziła się w programie. Dlaczego?

Złam to sam: CTR

Zjadłeś zęby na kryptografii? Tryby ECB i CBC łamiesz na kartce? Zostało ostatnie wyzwanie w temacie: https://var.tailcall.net/ctr.

Encryption is not authentication

No właśnie, w końcu czas na odpowiedź – dlaczego? Często powtarzane motto w kryptografii brzmi: *Encryption is not authentication*, czyli „szyfrowanie nie służy do uwierzytelniania". Problemem tutaj nie były błędy w implementacji – bo tych nie było. Zrobiliśmy coś gorszego – użyliśmy szyfrowania do celu, do którego nie jest przeznaczone.

Celem szyfrowania jest ukrycie treści wiadomości przed osobami, dla których nie jest przeznaczona. Czy faktycznie mieliśmy coś do ukrycia w tym przypadku? Oczywiście nie – zarówno nick, jak i bycie administratorem to publiczne atrybuty każdego użytkownika – szyfrowanie tutaj jest w zasadzie zupełnie niepotrzebne. Owszem, wydaje się, że jeśli użytkownik dostanie zaszyfrowane tzw. „losowe śmieci", nie będzie w stanie wpłynąć na nie w żaden sposób. Jak, mam nadzieję, udowodniliśmy, jest to zupełnie błędne założenie. Szyfrowanie nie służy do uwierzytelniania!

Zamiast tego powinniśmy popatrzeć na tak zwane podpisy… Ale o tym przy innej okazji.

ZŁE SZYFROWANIE TO NIE SZYFROWANIE

Na razie powrócimy do naszych szyfrów i sprawdzimy, czy przynajmniej spełniają swoje podstawowe zadanie – czyli uniemożliwiają odczytanie danych osobom postronnym.

W naszych poprzednich przykładach zakładaliśmy, że wszystkie dane są jawne. Na potrzeby tej części wyobraźmy sobie, że z jakiegoś powodu użytkownik nie powinien móc odczytać ze swojego ciastka, czy jest administratorem (owszem, jest to zupełnie abstrakcyjna sytuacja – robimy tak dla uproszczenia. Ale łatwo wyobrazić sobie, że np. w ciastku zaszyty jest jakiś sekret potrzebny do komunikacji między serwerami, którego użytkownik nie powinien móc poznać).

Utrzymamy ten sam model ataku co w poprzedniej części – czyli użytkownik może dowolnie zmieniać swój szyfrogram oraz obserwować odpowiedzi, jakie dostaje od serwera po modyfikacjach.

8. Swoją drogą użycie wartości domyślnych obiektu `Counter` dostarczanego przez PyCrypto często może łatwo doprowadzić do dziurawego kodu – parametr `initial_value` ma domyślną wartość 1 i łatwo używając go w ten sposób, spowodować powtórzenie się nonce i problemy z bezpieczeństwem. Jest to dobry przykład na to, jak trzeba uważać nawet korzystając z bibliotek.

Szyfrowanie 1: ECB

Zacznijmy ponownie od trybu ECB. Internet jest pełen ostrzeżeń, że ten tryb jest niebezpieczny, ale w zasadzie dlaczego?

Odpowiedź jest prosta – wyobraźmy sobie, że mamy zaszyfrowane ciastka trzech użytkowników:

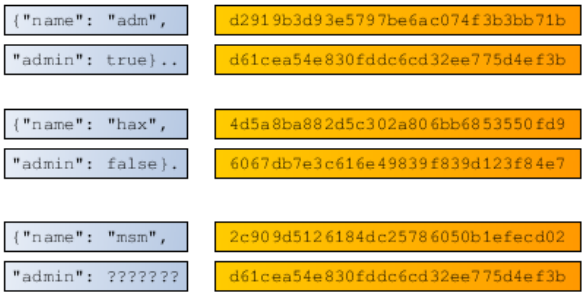
» adm, o którym wiemy, że jest administratorem,
» hax, nasz użytkownik – wiemy, że nie jest administratorem,
» msm, użytkownik, o którym nie wiemy, czy jest administratorem, i chcemy się dowiedzieć.

Gdyby szyfrowanie faktycznie dobrze zabezpieczało treść wiadomości, dowiedzenie się czegokolwiek o „msm" powinno być niemożliwe.

Załóżmy, że mamy następujące ciastka:

```
- adm: d2919b3d93e5797be6ac074f3b3bb71bd61cea54e830fddc6c
  d32ee775d4ef3b
- hax: 4d5a8ba882d5c302a806bb6853550fd96067db7e3c616e4983
  9f839d123f84e7
- msm: 2c909d5126184dc25786050b1efecd02d61cea54e830fd dc6
  cd32ee775d4ef3b
```

Przyjrzyjmy się dokładniej zaszyfrowanym wartościom:



Rysunek 12. Porównanie zaszyfrowanych wiadomości

Czy można ustalić grupę, do której przynależy użytkownik „msm"? Po chwili wpatrywania się w rysunek 12 można zauważyć, że owszem, można – drugi blok ciphertextu jest identyczny dla użytkowników „msm" i „adm", co oznacza, że drugi blok plaintextu również musi być identyczny.

Oznacza to, że tryb ECB zawiódł nas nawet w swoim podstawowym zadaniu – ukrywaniu treści wiadomości przed postronnymi[9].

Bywa nawet gorzej – jeśli atakujący jest w stanie szyfrować dowolne dane i kontroluje jakiś ich fragment (np. swój nick), istnieje atak, który pozwoli mu również odszyfrować dowolne dane za kontrolowanym kawałkiem (kosztem 256 zapytań do serwera na każdy bajt szyfrogramu).

Rysunek 13. Atakujący kontroluje swój nick i chciałby przeczytać „tajne informacje"

W takiej sytuacji należy tak wybrać wstawiane dane, żeby ostatni blok zaszyfrowanych danych zawierał tylko jeden bajt faktycznych danych oraz 15 bajtów paddingu.

9. Czasami w niektórych materiałach dedykowanych dla programistów pojawiają się zdania w rodzaju „trybu ECB nie powinno się nigdy używać". Niezupełnie tak jest. To bardziej uproszczenie, oparte na założeniu, że lepiej być „za bardzo zabezpieczonym" niż „za mało zabezpieczonym". W szczególności, kiedy wiemy, że zawsze będziemy szyfrować dokładnie jeden blok, tryb ECB nie ma przewagi nad innymi.

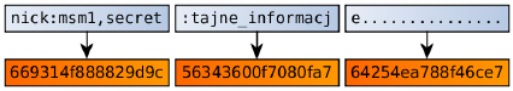
ECB, CBC, CTR, WTF?

Dlaczego niezrozumiałe trzyznakowe nazwy? Dlaczego nie można było nazwać trybów szyfrowania np. „niebezpieczny", „standardowy" i „licznikowy"? W zasadzie nie wiemy – być może to tajny plan kryptografów na zapewnienie sobie tak zwanego *job security*. Sprawa ma się jeszcze gorzej, bo trybów jest więcej:

- Tryb OFB, podobny do CTR, ale zamiast `aes(n)`, `aes(n+1)`, `aes(n+2)` używany jest `iv`, `aes(iv)`, `aes(aes(iv))`. Jego niewielką zaletą jest to, że deszyfrowanie działa dokładnie tak samo jak szyfrowanie (więc można zaoszczędzić na bramkach w hardware), ale ma dużo gorsze parametry bezpieczeństwa niż tryb CTR i z tego powodu nie jest polecany
- Tryb OCB, relatywnie nowe odkrycie, które zapewnia jednocześnie szyfrowanie, uwierzytelnianie i wysoką wydajność. Oznacza to, że nie trzeba dodatkowo podpisywać wiadomości. Ten schemat to jeden z nowoczesnych trybów gwarantujących tak zwane *authenticated encryption*[1]. Dlaczego więc nie jest używany powszechnie? Niestety – ciąży na nim patent[2], co historycznie skutecznie zniechęcało potencjalnych użytkowników i twórców standardów.
- Tryby GCM, CCM, EAX będące próbami stworzenia wolnej alternatywy dla trybu OCB. Niestety, wszystkie mają jakieś ograniczenia lub problemy – z tego powodu trudno wyróżnić jeden z nich (najpopularniejszy obecnie jest GCM). Mimo wszystko, jeśli zależy nam na uwierzytelnionym szyfrowaniu, każdy z nich ma przewagę nad szyfrowaniem i podpisywaniem wiadomości osobno. Wspólną wadą jest średnie wsparcie ze strony bibliotek i frameworków, szczególnie tych starszych.

W praktyce nie trzeba się tym za bardzo przejmować – tryb CBC w zupełności wystarcza do szyfrowania danych, a jeśli potrzebujemy dodatkowo uwierzytelnić dane, to możemy albo dodatkowo podpisać dane, albo użyć trybu GCM.

1. https://en.wikipedia.org/wiki/Authenticated_encryption
2. https://en.wikipedia.org/wiki/OCB_mode



Rysunek 14. Atakujący tak dopasował ostatni blok, że zawiera tylko jeden bajt danych

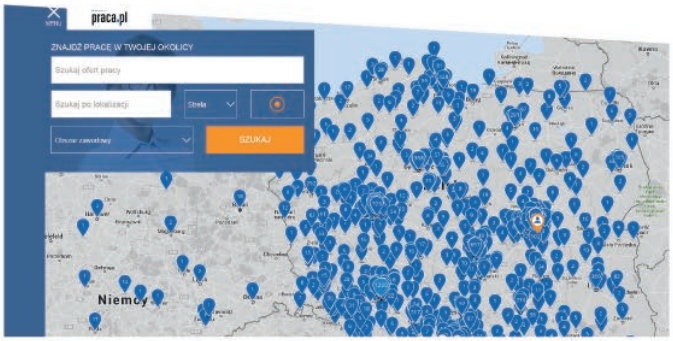
Można wtedy ten jeden bajt znaleźć za pomocą metody siłowej (skoro możemy szyfrować dowolne dane, możemy spróbować wszystkich jednobajtowych wiadomości). Kiedy to się uda, należy wygenerować kolejny szyfrogram, żeby w ostatnim bloku były dwa bajty danych oraz 14 bajtów paddingu. Można wtedy znowu zastosować metodę siłową – jako że ostatni bajt już znamy, wystarczy kolejne 255 zapytań. Tę procedurę należy powtarzać, aż poznamy wszystkie bajty całej wiadomości.

Rysunek 15. Atakujący szyfruje jednobajtowe bloki, aż uda mu się trafić na pasujący

Jako że atak ten jest dość rzadko wykorzystywany w praktyce (tryb ECB, na szczęście, prawie wymarł), podarujemy sobie implementację. Zamiast tego przejdźmy do ciekawszego ataku.

praca.pl

Chcesz dobrze zarobić?
Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

Lokalna.praca.pl

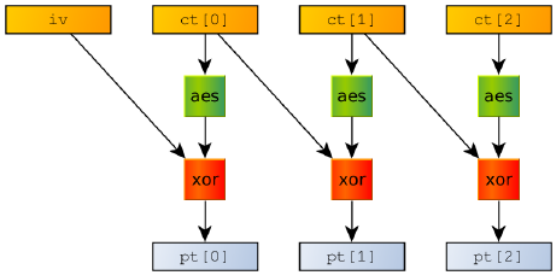
Poleć znajomego do pracy i zgarnij 1 000 zł

Praca.pl/rekomendacje.html

Szyfrowanie 2: CBC

Szyfrowanie w trybie CBC zepsuć równie łatwo, a atak ma podobnie (albo nawet bardziej) drastyczne skutki.

Widzieliśmy wcześniej, jak działa szyfrowanie w trybie CBC, ale teraz ważniejsze będzie dla nas deszyfrowanie:



Rysunek 16. Deszyfrowanie w trybie CBC

Ta metoda szyfrowania „matematycznie" jest silna, ale ma jedną pułapkę. Popatrzmy na nasz kod deszyfrujący:

```
def deserialize_cookie_2(cookie):
  cookie = cookie.decode(,hex')
  iv, cookie = cookie[:16], cookie[16:]
  cipher = AES.new(SHARED_KEY, AES.MODE_CBC, iv)
  raw_cookie = cipher.decrypt(cookie)
  cookie = pkcs7_unpad(raw_cookie)  # throws on invalid padding
  return json.loads(cookie)  # throws on invalid json
```

W przypadku aplikacji webowych błędy często idą bezpośrednio do użytkownika. Jeśli atakujący dowie się, jaki wyjątek został rzucony, będzie w stanie rozpoznać, czy deszyfrowanie nie udało się z powodu błędnego paddingu (`AssertionError` w `pkcs7_unpad`), czy z powodu nieprawidłowego JSON (`ValueError` w `json.loads`). Metoda pozwalająca rozpoznawać, czy padding dowolnego szyfrogramu jest poprawny, nazywana jest w kryptografii *Padding Oracle*. Nie brzmi groźnie, prawda?

```
# w praktyce, wyrocznią jest oczywiście zazwyczaj zdalny serwer
# np. strona deszyfrująca ciastko
# dla celów demonstracyjnych, "lokalna wyrocznia"
def padding_oracle(ciphertext):
  try:
    deserialize_cookie_2(ciphertext.encode('hex'))
  except AssertionError:
    return False  # invalid padding
  except ValueError:
    return True  # invalid json
  return True  # all ok
```

Niestety, w praktyce to duży problem. Jeśli popatrzymy jeszcze raz na rysunek 16 (albo przypomnimy sobie poprzednie ataki), widzimy, że atakujący może łatwo wpływać na zdeszyfrowane dane, modyfikując bloki szyfrogramu. W szczególności atakujący może spróbować wszystkich 256 możliwości na ostatni bajt, aż uda mu się tak trafić, że ostatni zdeszyfrowany bajt będzie równy 0x01. Wtedy szyfrogram będzie kończył się jednym bajtem równym 0x1 i ten ostatni bajt, uznany przez algorytm za padding, zostanie usunięty:

```
def attack_1(prev_block, last_block):
  for i in range(256):
    new_block = prev_block[:15] + chr(i)
    if padding_oracle(new_block + last_block):
      print 'padding is correct for prev[15] =', i
      return
# padding is correct for prev[15] = 221
```

I znowu – co z tego wynika? Dochodzimy tu do „punktu kulminacyjnego" naszego ataku. Warto spojrzeć jeszcze raz na Rysunek 16 pokazujący, jak działa deszyfrowanie.

Ustalmy terminologię:

- `prev` to oryginalna wartość przedostatniego bloku,
- `intermediate` to wartość ostatniego bloku po deszyfrowaniu, ale przed operacją xor,
- new to zmodyfikowana przez nas wartość ostatniego bloku,
- `plain` to plaintext (odszyfrowany blok, który chcemy odzyskać).

Wiemy, że:

- znamy ostatni bajt new, który deszyfrowaliśmy (to wartość `i`, którą wypisaliśmy w programie),
- Wiemy,[10] że ostatni bajt `plain` dla naszego zmodyfikowanego `prev` jest równy 1,
- znamy oryginalną wartość `prev` (zanim ją zmodyfikowaliśmy).

Zapiszmy to jako równanie:

```
prev[15] ^ intermediate[15] = plain[15]
new[15] ^ intermediate[15] = 1

# z czego wynika:
intermediate[15] = new[15] ^ 1
plain[15] = prev[15] ^ intermediate[15]
```

Matematyka nie kłamie – za pomocą dwóch operacji xor możemy odzyskać w ten sposób ostatni bajt plaintextu (czyli inaczej odszyfrować ostatni bajt).

Przetestujmy:

```
def attack_2(prev, last):
  intermediate = [0] * 16
  plain = [0] * 16
  for i in range(256):
    new = prev[:15] + chr(i)
    if i != ord(prev[15]) and padding_oracle(new + last):
      intermediate[15] = ord(new[15]) ^ 1
      plain[15] = ord(prev[15]) ^ intermediate[15]
      print 'last byte of plaintext is', plain[15]
      return

# last byte of plaintext is 4
```

Czyli udało nam się zdeszyfrować ostatni bajt wiadomosci, do wartości „4". Wygląda to na poprawną wartość (jako że jest stosowany standard PKCS#7, oznacza to, że cztery ostatnie bajty wiadomości to padding).

Faktycznie, coś zdeszyfrowaliśmy, ale jeszcze nie wygląda to drastycznie. Przełom następuje, kiedy zorientujemy się, że możemy rozszerzyć ten atak – i pójść dalej. Skoro znamy ostatni bajt bloku `intermediate`, możemy tak ustawić ostatni bajt bloku new, żeby plaintext kończył się na 0x2 – i zgadywać przedostatni bajt:

```
def attack_3(prev, last):
  intermediate = [0] * 16
  plain = [0] * 16
  new = list(ord(c) for c in prev)
  for i in range(256):
    new[15] = i
    if i != ord(prev[15]) and padding_oracle(to_str(new) + last):
      intermediate[15] = new[15] ^ 1
      plain[15] = ord(prev[15]) ^ intermediate[15]
      print 'plaintext[15] is ', plain[15]
      break
```

10. A dużym prawdopodobieństwem jest ryzyko, że tekst zdeszyfruje się do czegoś kończącego się np. na dwa bajty o wartości 0x2.

```
  for i in range(256):
    new[15] = 2 ^ intermediate[15]
    new[14] = i
    if padding_oracle(to_str(new) + last):
      intermediate[14] = new[14] ^ 2
      plain[14] = ord(prev[14]) ^ intermediate[14]
      print 'plaintext[14] is ', plain[14]
      break
# plaintext[15] is 4
# plaintext[14] is 4
```

Wygląda dobrze.

Jak łatwo się domyśleć, atak generalizuje się też na trzy bajty, i na cztery, i na więcej. Można by iść dalej tą drogą i skopiować pętlę 16 razy, ale lepiej użyć trochę bardziej zaawansowanych technik programistycznych i zapisać kod w postaci jednej, dość krótkiej pętli:

```
def attack_4(prev, last):
  intermediate = [0] * 16
  plain = [0] * 16
  new = list(ord(c) for c in prev)
  result = ''
  for j in range(16)[::-1]:  # from 15 to 0
    for k in range(j+1, 16):
      new[k] = intermediate[k] ^ (16-j)

    for i in range(256):
      if j == 15 and i == ord(prev[j]):
        continue

      new[j] = i
      if padding_oracle(to_str(new) + last):
        intermediate[j] = new[j] ^ (16 - j)
        plain[j] = ord(prev[j]) ^ intermediate[j]
        print 'plaintext[{}] is {}'.format(j, plain[j])
        result = chr(plain[j]) + result
        break
  print 'plaintext:', result

# plaintext[15] is 4
# plaintext[14] is 4
# plaintext[13] is 4
# plaintext[12] is 4
# plaintext[11] is 125
# plaintext[10] is 101
# plaintext[9] is 117
# plaintext[8] is 114
# plaintext[7] is 116
# plaintext[6] is 32
# plaintext[5] is 58
# plaintext[4] is 34
# plaintext[3] is 110
# plaintext[2] is 105
# plaintext[1] is 109
# plaintext[0] is 100
# plaintext: dmin": true}
```

Gotowe – możemy odszyfować dowolne dane, mając do dyspozycji jedynie wyrocznię. Takie ataki to jeden z powodów, dla których kryptografia bywa „straszna", a implementowanie powiązanych rzeczy odradzane. W końcu w samym trybie CBC nie ma niczego, co sugerowałoby, że zdradzanie informacji o poprawności paddingu prowadzi do tak krytycznych problemów. Szczególnie że zazwyczaj podczas tworzenia oprogramowania priorytety są wręcz odwrotne – dostarczanie dokładnych informacji o błędzie jest uważane za dobrą praktykę. Twórcy API kryptograficznych zauważyli w końcu ten problem i w obecnych wysokopoziomowych bibliotekach (np. WebCrypto API[11] [12]) ukrywane są szczegółowe informacje o przyczynie błędu.

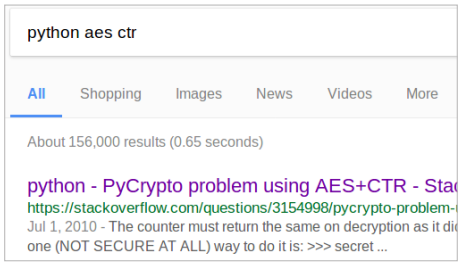
11. https://szukaj.programistamag.pl/uuid/c6976487a4749c6e8620aa6db5540dd9fde372e5
12. https://www.w3.org/TR/WebCryptoAPI/

Metoda *Padding Oracle* została odkryta w 2002 roku i spowodowała małe trzęsienie ziemi – była na nią wówczas podatna większość frameworków webowych, w tym Ruby on Rails, JavaServer Faces oraz ASP.NET. Do dzisiaj zdarza się, że w popularnych i stosowanych powszechnie protokołach ktoś znajdzie błąd tego typu – np. atak Lucky Thirteen na TLS[13] z 2013 roku, który łączy opisany tu atak z atakiem czasowym (mierzeniem czasu, jaki zajmuje zapytanie)[14].

Szyfrowanie 3: CTR

Na koniec, ku przestrodze: smutno – straszna historia.

Tak jak wspominaliśmy, tryb CTR jest w różnym stopniu wspierany przez biblioteki. Z tego powodu użycie go zazwyczaj nie jest trywialne. Programiści często w takim przypadku szukają pomocy w Internecie. Pierwszy wynik po zapytaniu „Python AES CTR" w moim przypadku prowadzi do StackOverflow[15]:

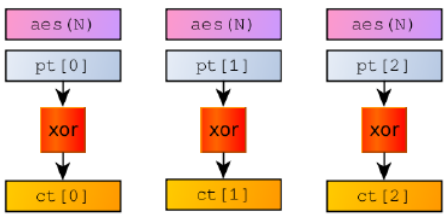


Rysunek 17. Pierwszy wynik po zapytaniu o „szyfrowanie AES CTR w pythonie"

Gdzie najwyżej punktowaną odpowiedzią jest następujący przykład od użytkownika będącego TOP 15 pod względem liczby punktów reputacji[16]:

```
>>> secret = os.urandom(16)
>>> crypto = AES.new(os.urandom(32), AES.MODE_CTR,
counter=lambda: secret)
>>> encrypted = crypto.encrypt("aaaaaaaaaaaaaaa")
>>> print crypto.decrypt(encrypted)
aaaaaaaaaaaaaaa
```

Ponownie zadajmy retoryczne pytanie – czy jest tu jakiś problem? Jak właściwie działa parametr `counter`? Jest to funkcja, która wywoływana jest dla każdego bloku i za każdym razem powinna zwrócić kolejną wartość licznika (autor jest tego świadomy). Niestety, w tym przypadku przekazana została funkcja, która zawsze zwraca tę samą wartość (!), co prowadzi do pewnych komplikacji:



Rysunek 17. Deszyfrowanie w trybie CTR, przy stałym nonce

Czyli całe „szyfrowanie AESem" sprowadza się do prostej operacji xor ze stałym, 16-bajtowym kluczem! Jest to bardzo dobrze znana kryptografom konstrukcja, nie dostarczająca w zasadzie żadnego bezpieczeństwa.

W większości przypadków można ją złamać nawet ręcznie (używając tylko ołówka i kartki papieru), w najgorszym przypadku poradzi sobie z nią komputer w milisekundy za pomocą prostej analizy statystycznej – takie ataki jednak pominiemy ze względu na ograniczone miejsce. Zobaczmy jednak przynajmniej, jak trywialny jest atak, jeśli znamy choć jeden blok ciphertextu. To bardzo realistyczna sytuacja – na przykład, jeśli zaszyfrowane dane to plik .png, pierwsze 16 bajtów będzie zawsze takie samo. Wiemy, że:

```
ct_0 = pt_0 ^ secret
ct_1 = pt_1 ^ secret
ct_2 = pt_2 ^ secret
# ...
ct_n = pt_n ^ secret
```

Teraz, jeśli wiemy, że zaszyfrowano obraz .png, możemy podstawić znaną wartość pod `pt_0` i obliczyć `secret`:

```
# constant header from .png file
pt_0 = '\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR'
secret = ct_0 ^ pt_0
```

W ten sposób dysponujemy wystarczającą ilością danych, żeby odzyskać cały ciphertext:

```
pt_1 = ct_1 ^ secret
pt_2 = ct_2 ^ secret
# ...
ct_n = pt_n ^ secret
```

Jeśli ktoś skopiował kod ze strony Stack Overflow, to cały „atak" na jego program to kilka do kilkunastu linijek kodu.

Ale to przecież nie problem – przecież nikt[17] nigdy[18] nie[19] kopiuje[20] bezmyślnie[21] kodu ze Stack Overflow…

ZAKOŃCZENIE

W artykule zaprezentowano kilka stosunkowo prostych ataków, które można przeprowadzić wobec prawidłowo zaimplementowanych, ale błędnie użytych algorytmów szyfrowania. Nie należy zapominać, że prawo do wypowiedzi w Internecie posiada każdy. Skutkuje to tym, że osoby niemające wiedzy z zakresu kryptografii udzielają związanych z nią porad na popularnych portalach. Przed wykorzystaniem jakiegokolwiek prymitywu kryptograficznego konieczne jest osiągnięcie pełnego zrozumienia sposobu jego działania, a to można zrobić, wyłącznie posługując się stosowną, zweryfikowaną literaturą.

13. https://en.wikipedia.org/wiki/Lucky_Thirteen_attack
14. Dla uczciwości trzeba dodać, że ten atak wymaga bardzo dokładnego pomiaru czasu zapytania, przez co jest mało praktyczny poza sieciami lokalnymi (albo środowiskami laboratoryjnymi).
15. https://stackoverflow.com/a/3155175
16. Ta odpowiedź została napisana w 2010 roku. Dwa tygodnie temu dokładnie ten kod został wykorzystany w konkursie SHA2017 CTF jako najprostsze (!) zadanie z kryptografii. Dopiero wtedy ktoś zlitował się nad innymi i dopisał do odpowiedzi „(NOT SECURE AT ALL)", (patrz Rysunek 17).
17. https://goo.gl/smRNCe (W krytycznej funkcji projektu nazywającego się *Confidentiality as a Service*)
18. https://goo.gl/y9cJeg (Szyfrowanie w *off the wire crypto, otr with pgp + aes-ctr 256 instead of diffie hellman*)
19. https://goo.gl/w2dEE1 (*A rebuild of marionette, encrypted proxy that simulates general webtraffic. (...) Provides strong end to end encryption with PGP + AES-CTR + HMACSHA256*)
20. https://goo.gl/oHjvjA (Zabezpieczanie plików w *GUI Text Editor with Crypto operations in Python*)
21. https://goo.gl/EAkJ5b (Biblioteka do szyfrowania)

JAROSŁAW JEDYNAK
msm@tailcall.net

Na co dzień broni bezpieczeństwa polskiego Internetu, pracując jako Security Engineer w Cert Polska. Do jego specjalności należy niskopoziomowa analiza malware oraz inżynieria wsteczna protokołów sieciowych złośliwego oprogramowania. W wolnym czasie programuje, pomaga w administracji serwisem 4programmers.net i nałogowo gra w konkursy CTF.