Programowanie Defensywne

Kryptgrafia, czyli co może pójść nie tak

Arkadiusz Chrobot

Katedra Systemów Informatycznych

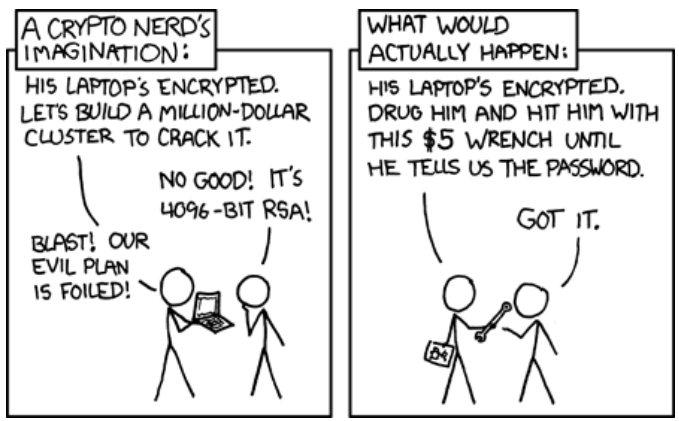
26 kwietnia 2024

Plan

1. Wprowadzenie
2. Wartości losowe
3. Skróty kryptograficzne
4. Szyfrowanie
   - Ochrona komunikacji
   - Ochrona danych składowanych

Wprowadzenie

Kryptografia jest podstawowym elementem służącym do zapewniania poufności i integralności. Celem tego wykładu nie jest wyjaśnienie zasad działania poszczególnych metod kryptograficznych, ale sposobu ich zastosowania w oprogramowaniu. Niestety, ale dosyć często się zdarza, że dokumentacja do bibliotek kryptograficznych jest napisana mało klarownie, a podane w niej przykłady zawierają mało bezpieczne lub w ogóle niebezpieczne ustawienia domyślne. Dodatkowo kryptografia bywa stosowana w sytuacjach, w których niewiele może pomóc, jak na zamieszczonym rysunku.



Autor: Randall Munroe, Źródło: https://xkcd.com/538/

Wartości losowe

Liczby (wartości) losowe są niezbędne do zapewnienia prawidłowego działania rozwiązań kryptograficznych. Kłopot w tym, że komputery są urządzeniami deterministycznymi, zatem trudno w nich uzyskać wartości, które byłby w pełni nieprzewidywalne. Najczęściej języki programowania dostarczają *generator liczb pseudolosowych* (ang. *Pseudorandom Number Generator*, PRNG), działający na bazie ciągów kongruencyjnych, który może być zastosowany np. w symulacjach opartych na metodach Monte Carlo[1], ale zupełnie nie nadaje się do użycia w kryptografii. Tutaj potrzebne są *kryptograficznie bezpieczne generatory liczb pseudolosowych* (ang. *Cryptographically Secure Pseudorandom Number Generator*, CSPRNG).

[1] Tutaj też należy zachować ostrożność. W drugim tomie „Sztuki programowania" autor, czyli prof. Donald E. Knuth opisuje historię, w której wiele prac naukowych musiało zostać wycofanych po tym, gdy odkryto, że użyto w nich metody generowania liczb, która dawała wartości zupełnie nielosowe.

Wartości losowe

CSPRNG są wolniejsze od PRNG, ale generują bardziej nieprzewidywalne wartości. Ich działanie oparte jest na losowej informacji pobieranej z otoczenia (np. częstotliwość i kolejność naciskania przez użytkownika na klawisze, ruch myszy, częstotliwość przerwań pochodzących od urządzenia pamięci masowej, wartości określonych rejestrów CPU, itd.). Może ono także być wspomagane przez sprzętowe generatory liczb losowych (ang. *hardware random number generators*), w które wyposażane są współczesne procesory. Z ciekawych rozwiązań sprzętowych, które nie są powszechnie stosowane, należy wymienić Dice-O-Matic, czyli ▸ maszynę, która fizycznie „rzuca" kostkami do gry i dzięki kamerom oraz oprogramowaniu OCR rozpoznaje wyrzucone liczby.

Wartości losowe

W praktyce dostęp do liczb losowych, które można zastosować w kryptografii otrzymuje się za pośrednictwem funkcji i metod udostępnianych przez odpowiednie biblioteki. W przypadku Javy należy skorzystać z klasy `java.security.SecureRandom`. Z kolei Python zapewnia pakiet `secrets`. W przypadku języka C należy użyć funkcji `getrandom()` lub `getentropy()`. Przykłady dla Pythona i C podano na następnych slajdach.

Wartości losowe

Python

```python
#!/usr/bin/env python3
import secrets

if __name__ == "__main__":
    print(secrets.randbelow(1500))
```

Wartości losowe

C

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/random.h>

int main(void)
{
        unsigned long int buffer;
        for(int i=0;i<50;i++) {
            ssize_t number_of_bytes =
↪   getrandom(&buffer,sizeof(buffer),0);
            if(number_of_bytes == -1)
                perror("getrandom");
            printf("Number of bytes: %lu, random number:
↪   %lu\n",number_of_bytes, buffer);
            while(sleep(60));
        }
    return 0;
}
```

Wartości losowe

Funkcja `getrandom()` zapisuje liczbę w pierwszym argumencie wywołania, którym powinien być adres odpowiedniej zmiennej. Drugim argumentem jest rozmiar tej zmiennej. Trzecim jest 0 lub następujące flagi:

`GRND_RANDOM` funkcja będzie korzystała z pliku urządzenia `/dev/random`, zamiast domyślnego `/dev/urandom`,

`GRND_NONBLOCK` funkcja nie będzie wstrzymywała działania w oczekiwaniu na pojawienie się nowej entropii w puli.

Obie flagi mogą być połączone sumą bitową. Pliki `/dev/random` i `/dev/urandom` są urządzeniami znakowymi i można je odczytywać bezpośrednio, ale nie jest to możliwe w przypadku każdego oprogramowania. Urządzenie `/dev/random` powinno być stosowane tylko kiedy liczby losowe są potrzebne zaraz po starcie systemu. W większości sytuacji `getrandom()` powinna być używana tak, jak zaprezentowano w listingu. Ponadto należy pamiętać, aby nie pobierać przy jej pomocy więcej niż 32 bajty i nie częściej niż 1 minutę.

Wartości losowe

C

Inną funkcją, która pozwala uzyskać wartości losowe nadające się do zastosowań kryptograficznych jest `getentropy()`. Wymaga ona tylko dwóch argumentów wywołania, którymi są adres zmiennej, do której ma być zapisana wartość i rozmiar tej wartości wyrażony w bajtach, który nie może przekraczać 256. Funkcja zwraca -1, jeśli nie może uzyskać prawidłowej wartości losowej. Jeśli jest ona używana przy starcie systemu, to jej wykonanie może ulec wstrzymaniu w oczekiwaniu na napełnienie przez jądro puli entropii, która stanowi źródło danych losowych.

Wartości losowe

```c
#include<stdint.h>
#include<stdio.h>
#include<unistd.h>

int main(void)
{
    uint64_t random_number;

    if(getentropy(&random_number,sizeof(random_number))==-1)
        perror("getentropy");
    else
        printf("Losowa 64-bitowa liczba naturalna: %lu\n",
↪   random_number);

    return 0;
}
```

Skróty kryptograficzne

Funkcje skrótu (ang. *hash functions*) są jednokierunkowe, tzn. na podstawie ich wartości nie można odtworzyć argumentu. Do zastosowań kryptograficznych nadają się funkcje, które cechuje trudność w znalezieniu *kolizji*. Kolizją nazywamy zjawisko polegające na tym, że różne argumenty dają tę samą wartość funkcji skrótu. Istnienie kolizji jest *pewne dla wszystkich funkcji skrótu*, bo zazwyczaj ich argumenty są dużo większe niż wartości. Jednak dla niektórych z nich trudno jest je znaleźć. Takie funkcje, niestety, jest trudno zaprojektować. Przykładowo, opracowana przez Ronalda Rivesta, współautora algorytmu RSA, funkcja MD5 uważana była bezpieczną, dopóki Wang Xiaoyun ze współpracownikami, a potem Vlastimil Klíma i inni nie odkryli efektywnej metody znajdywania dla niej kolizji.

Skróty kryptograficzne

Inna funkcja, SHA-1, przestała być bezpieczna z powodu rozwoju mocy obliczeniowej komputerów. Algorytmy znajdywania jej kolizji był wcześniej znany, ale nie było systemów komputerowych, które efektywnie mógłby go wykonać. Współcześnie zalecane jest stosowanie funkcji SHA-2 i SHA-3.

Zwróćmy uwagę, że te funkcje same z siebie nie gwarantują poufności danych, a jedynie ich integralność — gdyby intruz wiedział jakim algorytmem został wygenerowany skrót, to mógłby „podrzucić" swoje dane, wraz z poprawnym skrótem. Aby zapewnić poufność stosuje się funkcje skrótu z kluczem, które generują kody uwierzytelniania wiadomości (ang. *Message Authentication Code*, MAC). Ich argumentem, oprócz danych, dla których skrót ma być wygenerowany, jest także tajny klucz. Ten klucz, podobnie jak inne klucze, **nie może być na stałe zapisany w kodzie programu**.

Skróty kryptograficzne

Funkcje skrótu stosowane są także do przechowywania haseł w bezpiecznej postaci. Obecnie zaleca się stosowanie funkcji Argon2id. Posiada on trzy parametry, które można konfigurować, wpływając na jego działanie: `m` — minimalny rozmiar pamięci, `t` — minimalna liczba iteracji i `p` — stopień zrównoleglenia. Zalecane wartości to:

- m=12288 (12 MiB), t=3, p=1,
- m=9216 (9 MiB), t=4, p=1,
- m=7168 (7 MiB), t=5, p=1.

Jeśli ten algorytm nie jest dostępny, lub nie można go zastosować ze względu na jego wymagania, to zaleca się stosowanie `scrypt`, `bcrypt` lub PBKDF2. Ten ostatni jest wymagany przez standard FIPS 140-2. Następne slajdy zawierają kod programu w języku Python obrazującego użycie algorytmu Argon2id, ale bez ustawiania jego parametrów pracy.

Skróty kryptograficzne

Argon2id — Python

```python
#!/usr/bin/env python3
from argon2 import PasswordHasher

def hash_password(password):
    return PasswordHasher().hash(password)
```

Skróty kryptograficzne

Argon2id — Python

```
def verify_password(hash, password):
    password_hasher = PasswordHasher()
    try:
        if password_hasher.verify(hash, password):
            print("Hasło poprawne")
    except:
        print("Hasło niepoprawne")
```

Skróty kryptograficzne

Argon2id — Python

```python
if __name__ == "__main__":
    first = input("Podaj hasło: ")
    hash = hash_password(first)
    print(hash)
    second = input("Podaj hasło: ")
    verify_password(hash=hash, password=second)
```

Szyfrowanie

Szyfrowanie jest operacją polegającą na zamianie tekstu jawnego, na *szyfrogram*. Operacją do niej odwrotną jest *deszyfrowanie*. Szyfry możemy podzielić na:

strumieniowe klucz jest generowany przez specjalną funkcję i łączony z tekstem jawnym, bajt po bajcie — nadaje się do szyfrowania komunikacji odbywającej się w trybie ciągłym, ale klucz i tekst jawny mogą się powtarzać.

blokowe tekst jawny jest dzielony na *bloki*, które łączone są z kluczem — istnieje kilka trybów pracy takich algorytmów szyfrujących, nie wszystkie są bezpieczne.

Ponadto wyróżniamy szyfry:

symetryczne używany jest ten sam klucz do szyfrowania i deszyfrowania, który musi pozostać tajny,

asymetryczne szyfrowanie wykonywane jest przy pomocy klucza publicznego (jawnego), a deszyfrowanie przy pomocy prywatnego (tajnego).

Szyfrowanie

Siła szyfrowania opiera się nie na tajności jego projektu, ale na tajności niektórych parametrów wejściowych. Najlepiej jest stosować te algorytm, które są dobrze znane i zostały sprawdzone przez społeczność zajmującą się kryptoanalizą. **Pod żadnym pozorem nie należy opracowywać „własnych" algorytmów szyfrowania, zwłaszcza nie mając odpowiedniej wiedzy kryptograficznej**. Dodatkowo, należy także używać **sprawdzonych implementacji tych algorytmów**. Własne implementacje można stosować jedynie w celach edukacyjnych. Opracowanie bezpiecznej implementacji algorytmu szyfrującego jest bardzo trudne, z uwagi na możliwość powstania, tzw. *kanałów bocznych* (ang. *side channels*), przez które intruz może uzyskać informacje na temat szyfrowanych danych lub samego procesu szyfrowania.

Ochrona komunikacji

W zastosowaniach związanych z komunikacją należy używać sprawdzonych rozwiązań, takich jak TLS 1.2 i nowszych. W wymianie danych są stosowane zarówno szyfry asymetryczne, np. do uwierzytelniania wiadomości i wymiany kluczy dla szyfrów symetrycznych, jak i te ostatnie. Szyfry asymetryczne są wolniejsze od symetrycznych, zatem używa się je głównie do bezpiecznego przesyłania klucza dla algorytmu symetrycznego, przy rozpoczęciu transmisji lub do uwierzytelniania danych. Ta ostatnia operacja jest odwrotnością szyfrowania — strona podpisująca szyfruje wiadomość kluczem prywatnym, a weryfikująca deszyfruje publicznym. W komunikacji używany jest też algorytm Diffie-Hellmana do bezpiecznej wymiany kluczy przez niezabezpieczony kanał. Używane algorytmy szyfrowania to: AES, RSA i ECC.

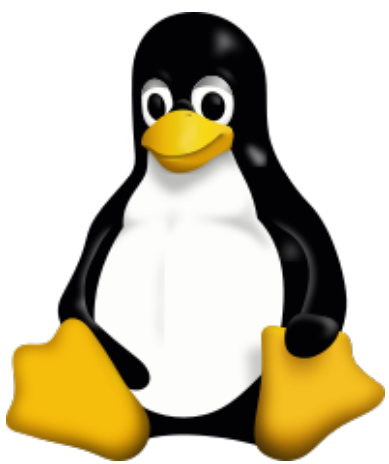
Szyfry asymetryczne

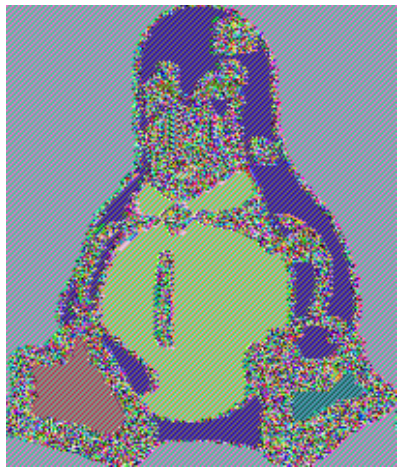
Najstarszym szyfrem asymetrycznym jest opracowany w 1977 roku RSA. Jego bezpieczeństwo oparte jest na trudności rozkładu iloczynu dwóch dużych liczb pierwszych. Niestety, stworzenie poprawnej implementacji algorytmu RSA jest trudne. Większość implementacji jest podatna na ataki z wykorzystaniem ▸ kanałów bocznych, lub bazujące na nietrafnym ▸ wyborze liczb pierwszych. Dodatkowo RSA wymaga ▸ dużych kluczy. Szacuje się, aby uzyskać siłę szyfru RSA odpowiadającą sile szyfru AES z kluczem 80-bitowym należy użyć kluczy 1024-bitowych. Obecnie powszechnie używanym rozmiarem kluczy w RSA jest 2048 bitów. Alternatywą dla tego szyfru są szyfry oparte na krzywych eliptycznych (ECC). W powszechnym użyciu jest należący do tej rodziny *Curve25519*.

Prace teoretyczne wykazały, że algorytmy asymetryczne nie będą odporne na kryptoanalizę przeprowadzoną z użyciem komputerów kwantowych.

Ochrona danych składowanych

Dla danych, które mogą być składowane w niezaufanych zasobach można użyć zarówno szyfrowania symetrycznego, jak i asymetrycznego, do zapewnienia ich poufności i integralności. W przypadku algorytmów symetrycznych, czyli głównie AES, należy pamiętać o wybraniu odpowiedniego trybu ich pracy. Współcześnie zalecany jest GCM (*Galois Counter Mode*). **Nie należy** używać trybu ECB, poza pewnymi szczególnymi okolicznościami. Aby się przekonać dlaczego, proszę spojrzeć na obraz w postaci jawnej i „zaszyfrowanej" z użyciem tego trybu (źródło: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#ECB).





Pytania

?

Koniec

Dziękuję Państwu za uwagę!