Szkoła Główna Gospodarstwa Wiejskiego
w Warszawie
Wydział Zastosowań Informatyki i Matematyki

Mateusz Tracz
172391

Implementacja serwisu umożliwiającego dwuetapową weryfikację użytkownika

Implementation of two factor authentication service
at the Warsaw University of Life Sciences – SGGW

Praca dyplomowa inżynierska
na kierunku Informatyka

Praca wykonana pod kierunkiem
dr. hab. Alexandera Prokopenya, prof. SGGW

Wydział Zastosowań Informatyki i Matematyki

Katedra Zastosowań Informatyki
Zakład Modelowania i Analizy Systemów

Warszawa 2017

Oświadczenie promotora pracy

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia tej pracy w postępowaniu o nadanie tytułu zawodowego.

Data ..... Podpis promotora pracy .....

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności z ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. Nr 90 poz. 631 z późn. zm.)

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną. Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data ..... Podpis autora pracy .....

Spis treści

Streszczenie

TODO: POLSKI TYTUŁ
TODO: POLSKIE STRESZCZENIE

Słowa kluczowe – TODO: POLSKIE TAGI implementacja, SGGW, Szkoła Główna Gospodarstwa Wiejskiego

Summary

TODO: ANGIELSKIE TYTUŁ
TODO: ANGIELSKIE STRESZCZENIE

Keywords – TODO: ANGIELSKIE TAGI thesis, implementation, SGGW, Warsaw University of Life Sciences

1 Wstęp

1.1 Cel pracy

1.2 Pojęcie uwierzytelnienia wielopoziomowego

1.3 Korzyści płynące z używania uwierzytelnienia wielopoziomowego

2 Elementy kryptografii

2.1 Kryptografia symetryczna oraz asymetryczna

Współczesną kryptografię można podzielić na kryptografię symetryczną oraz kryptografię asymetryczną. W dużym uproszczeniu, w przypadku kryptografii symetrycznej używany jest jeden, wspólny klucz kryptograficzny a w kryptografii asymetrycznej obecna jest para kluczy kryptograficznych - publiczny oraz prywatny. Przez co kryptografia asymetryczna często nazywana jest kryptografią klucza publicznego.

2.1.1 Szyfrowanie symetryczne

Szyfrowanie symetryczne przy użyciu klucza kryptograficznego „miesza" dane w taki sposób, że nie jest możliwe odzyskanie danych wejściowych bez użycia tego samego klucza. Algorytmy szyfrowania znacznie różnią się między sobą, jednak można pośród nich wyróżnić szyfry blokowe oraz szyfry strumieniowe.
Szyfry symetryczne idealnie nadają się do szyfrowania dużych ilości danych.

2.1.2 Szyfrowanie asymetryczne

Algorytmy używane w szyfrach asymetrycznych opierają się na rozwiązywaniu bardzo trudnych matematycznych problemów. Konstrukcja tych szyfrów pozwala niemal natychmiastowo rozwiązać zadany problem (i co za tym idzie odszyfrować dane) mając odpowiedni klucz, a w przypadku nieposiadania owego klucza, problem staje się niemożliwy do rozwiązania w rozsądnym czasie. Zwykle wykorzystywane jest tutaj zagadnienie faktoryzacji dużych liczb oraz znajdywania dyskretnych logarytmów.
Algorytmy asymetryczne działają na danych o ustalonej długości - zwykle od 1024 do 2048 bitów. Wykorzystywane są one zwykle tylko do przesłania klucza symetrycznego, który jest używane do szyfrowania pozostałej komunikacji. Spowodowane jest to tym, że algorytmy symetryczne są dużo bardziej wydajne niż asymetryczne.
Szyfrowanie asymetryczne poprawia nieco problem dystrybucji klucza. W odróżnieniu od szyfrowania symetrycznego, wymagane jest użycie $O(N)$ kluczy zamiast $O(N^2)$ kluczy.
Przykładami algorytmów asymetrycznych jest RSA lub ElGamal.

W praktyce rzadko kiedy korzysta się z wyłącznie kryptografii symetrycznej lub wyłącznie asymetrycznej. Zwykle w skład systemów kryptograficznych wchodzą elementy kryptografii symetrycznej jak również asymetrycznej.

2.2 Szyfry blokowe

Jednym z podstawowych elementów systemów kryptograficznych jest szyfr blokowy. Jest to algorytm operujący na bloku danych o ustalonej długości. Długość wynikowego bloku danych jest równa długości bloku podanego na wejściu.
Szyfry blokowe oparte są o tzw. „sieć substytucji-permutacji". Oznacza to, że dla każdego możliwego bloku jest ustalany dokładnie jeden blok mu odpowiadający. Za pomocą klucza kryptograficznego determinowane jest, który blok odpowiada któremu.
Dla zobrazowania tej definicji poniżej znajduje się szyfr o długości bloku równej 4 bity:

Tabela 2.1. Przykładowy szyfr blokowy o 4-bitowym bloku

0000: 0010	1000: 1111
0001: 0000	1001: 0011
0010: 0110	1010: 1011
0011: 1101	1011: 0001
0100: 1110	1100: 1100
0101: 0101	1101: 1001
0110: 0100	1110: 0111
0111: 1000	1111: 1010

Niezwykle ważną cechą jaka wynika z powyższej charakterystyki szyfru blokowego jest fakt, że znając pewną liczbę par (wiadomość, zaszyfrowana wiadomość), nie ujawnia to żadnych informacji o innych parach, szyfrowanych tym samym kluczem. Jedyną możliwą formą odszyfrowania wiadomości, nie posiadając klucza, jest wtedy metoda siłowa czyli wypróbowanie wszystkich możliwych kombinacji. Biorąc pod uwagę powyższy przykład, mając 16 różnych bloków, liczba ich permutacji wynosi 16! = 20922789888000. Prawdziwe szyfry posiadają znacznie większy rozmiar bloku, zwykle od 128 do 256 bitów. Przykładowo dla szyfru z rozmiarem bloku równym 128 bitów, należałoby sprawdzić aż $(2^{128})!$ możliwych kluczy.

2.2.1 Tryby pracy szyfrów blokowych

Nietrudno zauważyć jedną wadę szyfrów blokowych, która znacząco ogranicza proces szyfrowania. A mianowicie przy użyciu szyfrów blokowych, jesteśmy w stanie zaszyfrować dane o długości równej długości bloku. Zwykle chcielibyśmy jednak szyfrować wiadomości dłuższe niż 16 czy 32 bajty. Z tego powodu zostały stworzone uniwersalne konstrukcje, możliwe do użycia z dowolnym szyfrem blokowym, pozwalające zaszyfrować dane o dowolnej długości.

ECB

Rozwiązaniem przychodzącym natychmiast na myśl jest podzielenie wiadomości na bloki i zaszyfrowanie każdego bloku oddzielnie.

(a) Logo projektu. (b) AES w trybie ECB. (c) AES w trybie CBC.

Rysunek 2.1. Wizualizacja trybów szyfrowania.

Tryb ten nosi nazwę *ECB (elektroniczna książka kodowa)*.
Posiada on niezwykle niebezpieczną wadę. Dany blok wiadomości zostanie zaszyfrowany do zawsze tego samego bloku szyfrogramu.
Przykładowo mając wiadomość:

$$|\mathbf{ABCD}|EFGH|\mathbf{ABCD}|IJKL|$$

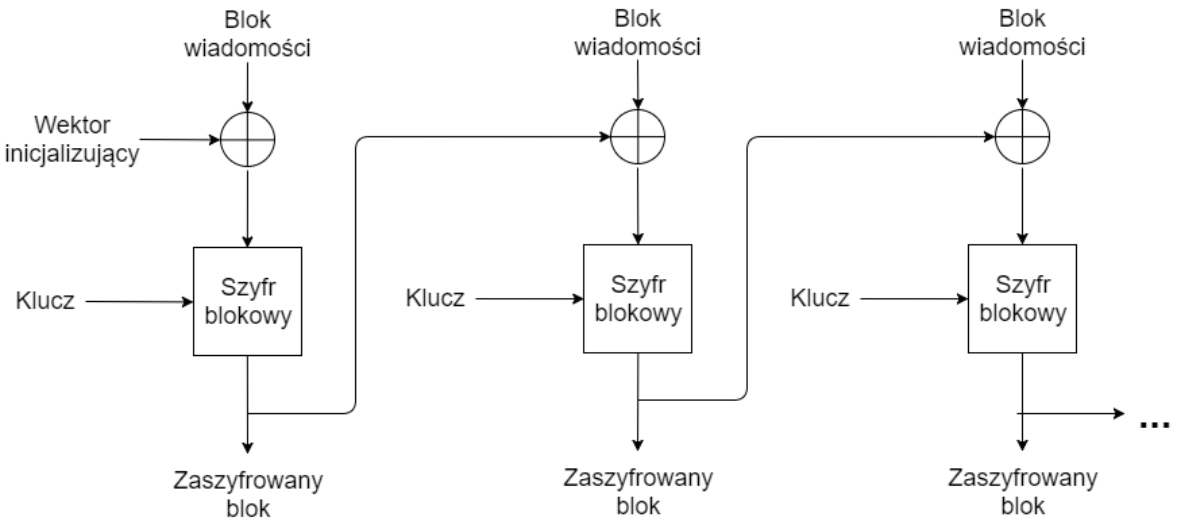
oraz szyfr blokowy o długości bloku 4 bajty, zaszyfrowana wiadomość będzie wyglądać następująco:

$$|\mathbf{IEKG}|BKLD|\mathbf{IEKG}|FHDL|$$

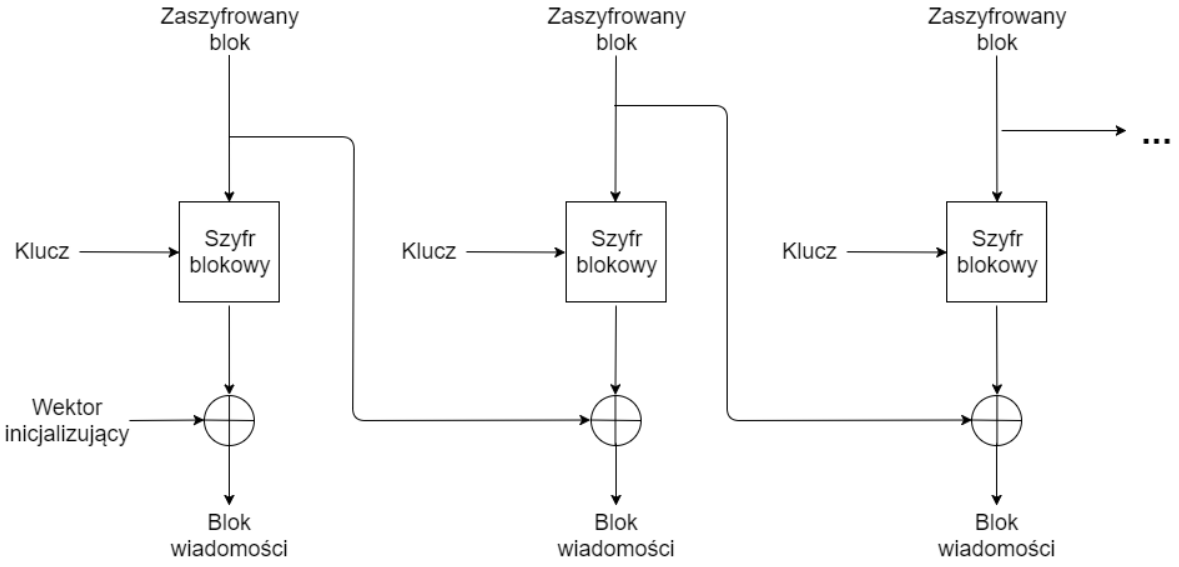
Wadę tę można zobrazować szyfrując w trybie ECB dane dowolnej grafiki. Grafika po zaszyfrowaniu nie jest w pełni czytelna ale z pewnością można stwierdzić co na niej jest.
Tryb ECB podatny jest również na atak typu *oracle*.

CBC

Najczęściej używanym trybem jest *CBC - Cipher Block Chaining (ang.)*. W tym trybie, blok wiadomości najpierw jest XORowany z poprzednim zaszyfrowanym blokiem a następnie on sam jest poddawany procesowi szyfrowania. W przypadku pierwszego bloku, dla którego nie ma bloku poprzedniego, wprowadzane jest pojęcie *wektora inicjalizującego*.
Wektor ten powinien być niemożliwy do zgadnięcia a w idealnej sytuacji powinien on być kryptograficznie losowy. Nie jest konieczne, aby wektor inicjalizujący był tajny, zwykle jest wysyłany razem z zaszyfrowaną wiadomością. Ważne jest jednak, aby atakujący nie był w stanie go przewidzieć przed samym procesem szyfrowania.
Proces deszyfrowania jest analogiczny. W pierwszej kolejności, zaszyfrowany blok poddawany jest działaniu szyfru blokowego, po czym jest on XORowany z poprzednim zaszyfrowanym blokiem. Na pierwszy blok musi zostać użyta funkcja XOR z takim samym wektorem inicjalizującym, jaki został użyty podczas szyfrowania.



Rysunek 2.2. Schemat szyfrowania w trybie CBC.



Rysunek 2.3. Schemat deszyfrowania w trybie CBC.

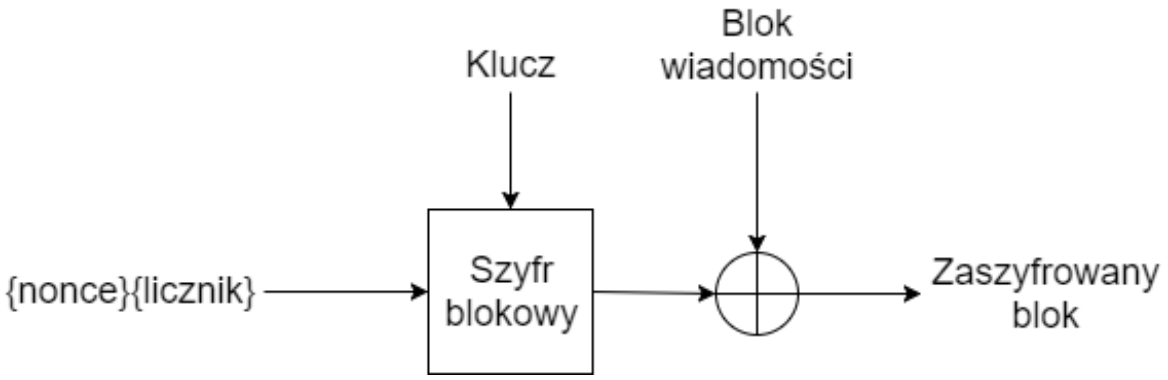
Tryb CBC sam w sobie można uznać za bezpieczny (w odróżnieniu od ECB), należy jednak o paru najczęściej popełnianych błędach:

1. Używanie tego samego wektora inicjalizującego dla paru wiadomości.

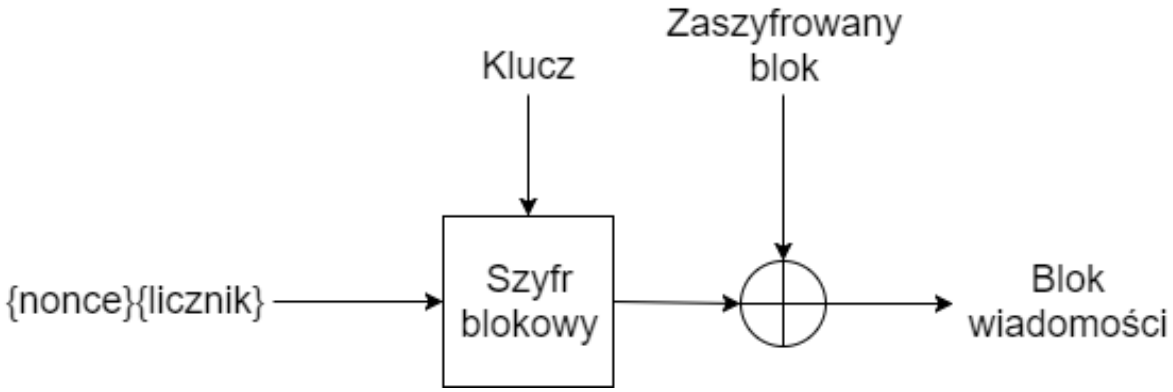
2. Używanie przewidywalnych wektorów inicjalizujących, na przykład bazujących na liczniku.

3. Używanie klucza jako wektora inicjalizującego.

Jednym z najbardziej znanych ataków na system bazujący na trybie CBC był atak o nazwie *BEAST* [13]. Błędem, który został wykorzystany, było użycie zaszyfrowanego bloku poprzedniej wysłanej wiadomości jako wektora inicjalizującego.



Rysunek 2.4. Schemat szyfrowania w trybie CTR.



Rysunek 2.5. Schemat deszyfrowania w trybie CTR.

CTR

Warto wspomnieć jeszcze o trybie *CTR - Counter (ang.)*, gdyż sposób jego działania jest zbliżony do szyfrów strumieniowych. Wykorzystywany jest w nim *liczba używana jednorazowo (z angielskiego „nonce")* oraz licznik. Licznik jest zwiększany dla każdego z bloków wiadomości. Na wejście szyfru blokowego podawana jest wyżej wymieniona liczba połączona z licznikiem. Następnie wynik działania szyfru jest XORowany z blokiem wiadomości. Krytyczne jest tutaj generowanie nowej liczby *nonce* dla każdej z wiadomości.

2.2.2 DES

2.2.3 AES

2.3 Szyfry strumieniowe

Pisząc o szyfrach strumieniowych mam tutaj na myśli natywne szyfry strumieniowe, które zostały stworzone z myślą o pracy w trybie strumieniowym. Pomimo istnienia dwóch typów szyfrów strumieniowych - synchroniczny oraz asynchronicznych, w praktyce używany jest prawie wyłącznie pierwszy typ.
W odróżnieniu od szyfrów blokowy, które szyfrowały cały blok danych, szyfry strumieniowe szyfrują każdy bit osobno. Przy użyciu klucza kryptograficznego generują one wystarczająco

długi, pseudolosowy ciąg bitów. Następnie wykonywana jest operacja XOR pomiędzy wygenerowanym ciągiem a wiadomością jaką chcemy zaszyfrować. Deszyfrowanie polega na ponownym wygenerowaniu ciągu bitów na podstawie klucza a następnie wykonanie operacji XOR na bitach zaszyfrowanej wiadomości oraz tych wygenerowanych z klucza.

2.3.1 RC4

Obecnie najczęściej spotykanym szyfrem strumieniowym jest RC4. Mimo, że wielokrotnie wykazano wiele podatności w konstrukcji RC4, szyfr ten jest ciągle używany w protokołach takich jak TLS czy WEP.
Schemat działania algorytmu jest niezwykle prosty. Składa się z inicjalizacji klucza oraz generacji pseudolosowego ciągu.
W etapie inicjalizacji klucza tworzona jest tablica permutacji S, składają się z 256 bajtów. Początkowo znajdują się w niej kolejne liczby od 0 do 255. Następnie dokonywane jest przejście po elementach tablicy, korzystając z indeksu i. Na każdym kroku pętli obliczany jest indeks j, poprzez dodanie aktualnej wartości j (zaczynającej się od 0) do wartości klucza znajdującej się pod indeksem i oraz do wartości tablicy S znajdującej się pod indeksem i. W przypadku, gdyby któryś z indeksu wyszedł poza zakres rozpatrywanych tablic, używa się operacji modulo długość tablicy. Mając obliczony indeks j zamienia się element $S[i]$ z elementem $S[j]$.
Do wygenerowania pseudolosowego ciągu wykorzystywana jest wcześniej stworzona tablica permutacji S. W pierwszej kolejności dla każdego indeksu i, obliczany jest indeks $j := j + S[i]$. Następnie zamienione są wartości $S[i]$ oraz $S[j]$. W celu otrzymania bajtu, który zostanie wykorzystany do zaszyfrowania wiadomości, pobierana jest wartość $S[S[i]+S[j]]$ z tablicy S.

2.3.2 Salsa20

W porównaniu do RC4, Salsa20 jest relatywnie nowym szyfrem strumieniowym, stworzonym przez Daniela J. Bernsteina. Bazuje on na operacjach *ARX (add–rotate–xor)*, to jest operacji składających się z dodawania modulo, obrotów bitowych oraz funkcji *XOR*. Zaletą stosowania szyfrów *ARX* jest to, że odporne są one na ataki czasowe, polegające na mierzeniu czasu, jaki potrzebny jest do przeprowadzenia operacji kryptograficznych.
Niezwykle interesującą cechą szyfru Salsa20 jest możliwość rozpoczęcia procesu odszyfrowywania od dowolnego miejsca w strumieniu danych. Mając więc duży plik, możliwe jest odszyfrowywanie tylko tej jego części, która nas interesuje.
Na chwilę obecną nie są znane żadne praktyczne ataki na szyfr Salsa20, może on zostać użyty jako alternatywa do szyfru blokowego AES.

2.4 Kryptograficzna funkcja skrótu

Funkcją skrótu nazywana jest funkcja, która dla danych wejściowych o dowolnym rozmiarze zwraca dane o z góry ustalonej długości. Funkcje skrótu znajdują zastosowanie w takich dziedzinach jak struktury danych (tablice mieszające, filtr Blooma), algorytmy dopasowania

wzorca (algorytm Karpa-Rabina) czy też w kryptografii.

Aby funkcja skrótu mogła zostać użyta w systemach kryptograficznych musi posiadać ona szereg parametrów.
Jednym z nich jest odporność na kolizje. Kolizją nazywamy przypadek, gdy dla dwóch różnych argumentów funkcja skrótu zwraca ten sam wynik. Nie jest oczywiście możliwe, aby całkowicie uniknąć kolizji, gdyż zbiór danych o dowolnym rozmiarze jest mapowany na zbiór skończony, zależy nam jednak aby proces znajdywania kolizji dla określonych danych był uważany za „trudny". (Przez „trudny" należy tutaj rozumieć problem, który nie jest możliwy do rozwiązania w rozsądnej ilości czasu.)
Kolejny z parametrów jest częściowo związany z poprzednim. Zależy nam, aby rozpatrywana funkcja była funkcją jednokierunkową. Oznacza to, że dla danego wyniku funkcji skrótu, znalezienie argumentu jest również problemem „trudnym". (Sam fakt istnienia funkcji jednokierunkowych nie został formalnie udowodniony. [3])
Niezwykle ważne jest też, aby nawet niewielka zmiana danych wejściowych, spowodowała znaczną zmianę danych otrzymanych na wyjściu (wymagane jest aby przynajmniej połowa bitów uległa zmianie).

2.4.1 Message Digest 5

Funkcja MD5 jest wykorzystywana do generowania 128-bitowego skrótu. Została stworzona przez Rona Rivesta w 1991 roku.
W uproszczeniu, algorytm MD5 można przedstawić w następujących krokach [8]:

1. Dodanie dopełnienia. W pierwszej kolejności dopisywany jest jeden bit o wartości 1, a następnie dopisywane są zera, aż do momentu, gdy długość danych wynosić będzie 448 bitów modulo 512. Dopełnienie dopisywane jest nawet w przypadku, gdy długość danych wynosi 448 bitów.

2. Pozostałe 64 bity wypełniane są liczbą reprezentującą długość wiadomości (sprzed wypełnienia) modulo 2^{64}.

3. Inicjalizacja stanu MD5 w postaci czterech 32-bitowych zmiennych A, B, C i D. Są one inicjalizowane stałymi zdefiniowanymi w specyfikacji (przedstawione w systemie szesnastkowym):
   $A := 01234567$
   $B := 89abcdef$
   $C := fedcba98$
   $D := 76543210$

4. Dane wejściowe dzielone są na bloki po 512 bitów. Kolejno na każdym z bloków wykonywane są operacje bitowe zmieniające zmiennych.

5. Wynikiem działania algorytmu jest 128-bitowa wartość składająca się z omawianych czterech zmiennych w kolejności A, B, C, D.

Szczegółowa specyfikacja algorytmy znajduje się w dokumencie RFC 1321 [6].
Analiza kryptograficzna funkcji MD5 wykazała wiele podatności i błędów przez co obecnie nie jest wskazane używanie MD5 w zastosowaniach kryptograficznych. W roku 2004 została opublikowana praca wykazująca podatność funkcji MD5 na ataki kolizyjne (ang. collision attack) [4]. Cztery lata później został znaleziony atak na kody uwierzytelnienia wiadomości bazujące na funkcji MD5 [5].

2.4.2 Secure Hash Algorithm 1

SHA-1 jest funkcją bazującą na MD4 (podobnie jak MD5), o długości skrótu wynoszącej 160 bitów [10].
Wewnętrzny stan funkcji składa się z pięciu zmiennych: A, B, C, D oraz E, każda o rozmiarze 32 bitów.
Pseudokod algorytmu można przedstawić w następujących krokach:

1. Inicjalizacja zmiennych następującymi stałymi (przedstawione w systemie szesnastkowym):
   $A := 0x67452301$
   $B := 0xEFCDAB89$
   $C := 0x98BADCFE$
   $D := 0x10325476$
   $E := 0xC3D2E1F0$

2. Zaaplikowanie dopełnienia:

   (a) Dopisanie na koniec danych bitu o wartości 1.
   Przykładowo jeśli przetwarzane są dane *01011001*, to są one dopełniane do *010110011*.

   (b) Następnie dopisywane są bity o wartości 0, aż do momentu, gdy długość danych wynosić będzie 448 bitów modulo 512.
   Mając dane *00101101 01000010 10011101 10001010 00101101 10011101*, po zastosowaniu dopełnienia z kroku (a) otrzymujemy *00101101 01000010 10011101 10001010 00101101 10011101 1*. Jako że długość danych wynosi 49 bitów, wymagane jest dopisanie 399 zer. Po zastosowaniu dopełnienia otrzymujemy (reprezentacja heksadecymalna):
   2d429d8a 2d9d8000 00000000 00000000
   00000000 00000000 00000000 00000000
   00000000 00000000 00000000 00000000
   00000000 00000000

   (c) W ostatnim kroku na miejsce ostatnich 64 bitów dopisywana jest długość danych.
   2d429d8a 2d9d8000 00000000 00000000
   00000000 00000000 00000000 00000000
   00000000 00000000 00000000 00000000
   00000000 00000000 00000000 00000031

3. Podzielenie wiadomości na 512-bitowe bloki.

4. Dla każdego z bloków należy wykonać następujące operacje:

   (a) Podzielenie bloku na szesnaście 32-bitowych części $w[i], 0 \leq i \leq 15$.

   (b) Rozszerzenie 16 części do 80 korzystając ze wzoru:
   $w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) <<< 1$

   (c) Inicjalizacja zmiennych pomocniczych dla danego bloku
   $a := A$
   $b := B$
   $c := C$
   $d := D$
   $e := E$

   (d) Dla każdej z 80 części $w[i]$ należy wykonać:

      i. jeżeli $0 \leq i \leq 19$
      $f := (b \,\&\, c) \mid ((\sim b) \,\&\, d)$
      $k := 0x5A827999$

      ii. jeżeli $20 \leq i \leq 39$
      $f := b \oplus c \oplus d$
      $k := 0x6ED9EBA1$

      iii. jeżeli $40 \leq i \leq 59$
      $f := (b \,\&\, c) \mid (b \,\&\, d) \mid (c \,\&\, d)$
      $k := 0x8F1BBCDC$

      iv. jeżeli $60 \leq i \leq 79$
      $f := b \oplus c \oplus d \; k := 0xCA62C1D6$

      v. następnie
      $t := (a <<< 5) + f + e + k + w[i]$
      $e := d$
      $d := c$
      $c := b <<< 30$
      $b := a$
      $a := t$

   (e) Aktualizacja wewnętrznego stanu funkcji
   $A := A + a$
   $B := B + b$
   $C := C + c$
   $D := D + d$
   $E := E + e$

5. Zwrócenie wyniku w postaci:
   $(A << 128) \mid (B << 96) \mid (C << 64) \mid (D << 32) \mid E$

W dokumencie RFC 3174 zostały zawarte szczegóły dotyczące algorytmu, jak również przykładowa implementacja [7].

Podobnie jak MD5, funkcja SHA-1 również nie jest uważana za bezpieczną.
Do roku 2017 wszystkie przedstawione ataki uznawane były za niepraktyczne z uwagi na zbyt dużą moc obliczeniową, jaka byłaby potrzebna do ich wykonania. Przykładem może być tutaj atak opublikowany w październiku 2015 roku o nazwie *The SHAppening*. Koszt wynajęcia sprzętu potrzebnego do przeprowadzenia ataku (wygenerowania jednej kolizji) estymowany był na 75 000 - 120 000 dolarów amerykańskich [11].
Pierwszy praktyczny atak na SHA-1 został ogłoszony w lutym 2017 roku. Zostały wygenerowane dwa pliki w formacie PDF, dla których wynik funkcji skrótu SHA-1 jest taki sam. Wszystkie systemy, w których wykorzystywana jest funkcja SHA-1 są narażone na ten atak, przykładowo umożliwia on fałszowanie podpisów cyfrowych dokumentów, czy też certyfikatów HTTPS. [12]

2.4.3 Secure Hash Algorithm 2

SHA-2 jest rodziną funkcji składającą się z SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. Funkcje te generują skróty o długościach odpowiednio: 224, 256, 384, 512, 224 oraz 256 bitów.

SHA-256

Jedną z najczęściej używanych funkcji z rodziny SHA-2 jest SHA-256. Proces uzyskiwania skrótu SHA-256 jest następujący:

1. Inicjalizacja stanu wewnętrznego funkcji (przedstawione w systemie szesnastkowym):
   $A := 0x6a09e667$
   $B := 0xbb67ae85$
   $C := 0x3c6ef372$
   $D := 0xa54ff53a$
   $E := 0x510e527f$
   $F := 0x9b05688c$
   $G := 0x1f83d9ab$
   $H := 0x5be0cd19$

2. Inicjalizacja pomocniczych stałych.

$$k[0..63] := \begin{matrix} 0x428a2f98, & 0x71374491, & 0xb5c0fbcf, & 0xe9b5dba5, \\ 0x3956c25b, & 0x59f111f1, & 0x923f82a4, & 0xab1c5ed5, \\ 0xd807aa98, & 0x12835b01, & 0x243185be, & 0x550c7dc3, \\ 0x72be5d74, & 0x80deb1fe, & 0x9bdc06a7, & 0xc19bf174, \\ 0xe49b69c1, & 0xefbe4786, & 0x0fc19dc6, & 0x240ca1cc, \\ 0x2de92c6f, & 0x4a7484aa, & 0x5cb0a9dc, & 0x76f988da, \\ 0x983e5152, & 0xa831c66d, & 0xb00327c8, & 0xbf597fc7, \\ 0xc6e00bf3, & 0xd5a79147, & 0x06ca6351, & 0x14292967, \\ 0x27b70a85, & 0x2e1b2138, & 0x4d2c6dfc, & 0x53380d13, \\ 0x650a7354, & 0x766a0abb, & 0x81c2c92e, & 0x92722c85, \\ 0xa2bfe8a1, & 0xa81a664b, & 0xc24b8b70, & 0xc76c51a3, \\ 0xd192e819, & 0xd6990624, & 0xf40e3585, & 0x106aa070, \\ 0x19a4c116, & 0x1e376c08, & 0x2748774c, & 0x34b0bcb5, \\ 0x391c0cb3, & 0x4ed8aa4a, & 0x5b9cca4f, & 0x682e6ff3, \\ 0x748f82ee, & 0x78a5636f, & 0x84c87814, & 0x8cc70208, \\ 0x90befffa, & 0xa4506ceb, & 0xbef9a3f7, & 0xc67178f2 \end{matrix}$$

3. Zaaplikowanie dopełnienia. Proces ten przebiega identycznie jak w przypadku SHA-1.

4. Podzielenie wiadomości na 512-bitowe bloki.

5. Dla każdego z bloków należy wykonać następujące operacje:

   (a) Podzielenie bloku na szesnaście 32-bitowych części $w[i], 0 \leq i \leq 15$.

   (b) Rozszerzenie 16 części do 64 korzystając ze wzoru:
   $s0 := (w[i-15] >>> 7) \oplus (w[i-15] >>> 18) \oplus (w[i-15] >> 3)$
   $s1 := (w[i-2] >>> 17) \oplus (w[i-2] >>> 19) \oplus (w[i-2] >> 10)$
   $w[i] := w[i-16] + s0 + w[i-7] + s1$

   (c) Inicjalizacja zmiennych pomocniczych dla danego bloku
   $a := A$
   $b := B$
   $c := C$
   $d := D$
   $e := E$
   $f := F$
   $g := G$
   $h := H$

   (d) Dla każdej z 64 części $w[i]$ należy wykonać:
   $S1 := (e >>> 6) \oplus (e >>> 11) \oplus (e >>> 25)$
   $ch := (e \,\&\, f) \oplus ((\sim e) \,\&\, g)$
   $t1 := h + S1 + ch + k[i] + w[i]$
   $S0 := (a >>> 2) \oplus (a >>> 13) \oplus (a >>> 22)$
   $m := (a \,\&\, b) \oplus (a \,\&\, c) \oplus (b \,\&\, c)$

$t2 := S0 + m$
$h := g$
$g := f$
$f := e$
$e := d + t1$
$d := c$
$c := b$
$b := a$
$a := t1 + t2$

(e) Aktualizacja wewnętrznego stanu funkcji
$A := A + a$
$B := B + b$
$C := C + c$
$D := D + d$
$E := E + e$
$F := F + f$
$G := G + g$
$H := H + h$

6. Zwrócenie wyniku w postaci:
$(A << 224) \mid (B << 192) \mid (C << 160) \mid$
$(D << 128) \mid (E << 96) \mid (F << 64) \mid (G << 32) \mid H$

W porównaniu do SHA-1 są znacznie odporniejsze na kolizję, dzięki czemu zalecane jest ich używanie w zastosowaniach takich jak podpisy cyfrowe czy uwierzytelnianie wiadomości.

2.4.4 Secure Hash Algorithm 3

Wewnętrzna budowa funkcji z rodziny SHA-3 znacząco się różni w odniesieniu do omawianych wcześniej. Poprzednie funkcje oparte były o schemat Merkle–Damgård. Funkcje z rodziny SHA-3 oparte są o tak zwaną „architekturę gąbki". W obu przypadkach funkcja zawiera wewnętrzny stan. Różnica jest jednak w sposobie otrzymywania wyniku na podstawie tego stanu. Poprzednio zwrócenie przez funkcję wyniku było równoznaczne ze zwróceniem wewnętrznego stanu funkcji. W przypadku SHA-3 tak się nie dzieje. Wewnętrzny stan przepuszczany jest kolejne cykle algorytmu. Na wynik funkcji składają się części wewnętrznego stanu, pobierane w kolejnych cyklach. Skutkuje to tym, że nigdy nie jest ujawniany cały wewnętrzny stan funkcji.
Ulepszona wewnętrzna konstrukcja algorytmu zapobiega atakom typu „length extension", na który podatne są funkcje MD5, SHA-1 oraz SHA-2 [9].

2.5 Kod uwierzytelnienia wiadomości

2.6 MAC bazujący na funkcji skrótu

2.7 Pojęcia entropii

3 Kryptografia w praktyce

3.1 Pojęcia pomocnicze

Przed przystąpieniem do opisu praktycznych aspektów kryptografii użytych w projekcie, wymagane jest wyjaśnienie pojęć wykorzystywanych w mechanizmie haseł jednorazowych, lecz które nie są bezpośrednio związane z kryptografią.

3.1.1 Kodowanie transportowe

Kodowanie transportowe wykorzystywane jest w przypadku, gdy zachodzi potrzeba transferu danych w środowiskach, które pozwalają na przesyłanie wyłącznie znaków ASCII.

Użycie kodowania transportowego jest konieczne w celu zachowania kompatybilności przy pracy z protokołami, które przystosowane są do pracy na danych 7-bitowych. W takim przypadku najstarszy bit jest zerowany, co mogłoby uszkodzić przesyłane dane. W przypadku przesyłania wyłącznie znaków ASCII zerowanie najstarszego bitu nie jest problemem, gdyż wszystkie znaki w podstawowej tablicy ASCII mają ten bit wyzerowany.
Bardziej współczesnym przykładem wykorzystania kodowania transportowego jest osadzanie danych graficznych bezpośrednio w kodzie HTML. Konieczne jest wówczas zakodowanie danych w celu wyeliminowania ryzyka pojawienia się znaków ’<’ oraz ’>’, które mogłyby być zinterpretowane jako tagi HTML.

Aby ujednolicić implementacje kodowania transportowego został stworzony dokument RFC 4648 [2], w którym opisany jest prawidłowy sposób implementacji oraz to jaki typ kodowania wybrać w zależności od nałożonych wymagań.

Kodowanie Base64

Najczęściej spotykanym typem kodowania transportowego jest kodowanie Base64. Kodowanie to konwertuje dowolny ciąg bajtów do postaci ciągu złożonego z małych i wielkich liter, cyfr oraz znaków ’+’ i ’/’. Jeżeli po zakodowaniu końcowa część danych jest mniejsza niż 24 bity używany jest także znak ’=’ jako dopełnienie.
Sam proces kodowania polega na pobraniu 24 bitów danych a następnie podzieleniu ich na 4 grupy po 6 bitów. Każda z grup jest interpretowana jako indeks tablicy ustalonego alfabetu Base64. Dla każdej z grup za pomocą indeksu odczytywany jest znak a następnie dopisywany jest on do ciągu zakodowanego.
Istnieje również odmiana kodowania Base64 przystosowana do użycia w przypadku adresów URL czy nazw plików. W alternatywie tej zamiast znaków ’+’, ’/’, które mogłyby zostać błędnie zinterpretowane np w środowisku systemu plików, używane są znaki ’-’ oraz ’_’.

Kodowanie Base32

W porównaniu do kodowania Base64, dane zakodowane w Base32 są dużo bardziej czytelne dla ludzi. Właściwość ta spowodowana jest faktem, że w kodowaniu Base32 nie ma znaczenia wielkość liter, dzięki czemu przykładowo nie ma problemu z rozróżnieniem małej litery 'L' z wielką literą 'I' ('l' oraz 'I').
Alfabet kodowania Base32 składa się z 32 znaków ASCII oraz znaku '=' pełniącego funkcje dopełnienia. Proces kodowania polega na pobraniu 40 bitów danych a następnie ustawienie ich w osiem 5-bitowych grup. Każda z 8 grup interpretowana jest jako jeden ze znaków alfabetu Base32.
Podobnie jak przy kodowaniu Base64, wymagane jest tutaj wstawienie dopełnienia w sytuacji, gdy długość ostatniej z grup jest mniejsza od 40 bitów.

Tabela 3.1. Alfabet w kodowaniu Base32

Indeks	Znak	Indeks	Znak	Indeks	Znak	Indeks	Znak
0	A	9	J	18	S	27	3
1	B	10	K	19	T	28	4
2	C	11	L	20	U	29	5
3	D	12	M	21	V	30	6
4	E	13	N	22	W	31	7
5	F	14	O	23	X		
6	G	15	P	24	Y		
7	H	16	Q	25	Z		
8	I	17	R	26	2		

3.1.2 Czas uniksowy

Czas uniksowy jest sposobem na reprezentację punktu w czasie, polegającym na mierzeniu sekund, które upłynęły od daty 1 stycznia 1970 (UTC). W systemach uniksowych zwykle reprezentowany jest w postaci 32-bitowej liczby całkowitej ze znakiem.
W przypadku architektur typu serwer-klient wskazane jest synchronizowanie czasu wykorzystując czas uniksowy, gdyż nie zależy on od lokalizacji w której jest mierzony. Właściwość ta eliminuje problem synchronizacji czasu pomiędzy strefami czasowymi.

3.1.3 Ujednolicony identyfikator zasobów

Ujednolicony identyfikator zasobów (ang. Uniform Resource Identifier, URI) jest ciągiem znaków jednoznacznie identyfikującym dany zasób.
Składnia identyfikatora jest wyrażana następująco:

schemat ":" ścieżka ["?" zapytanie] ["#" fragment]

Warto zauważyć, że składnia ta determinuje schemat (protokół), jaki wykorzystywany jest przy interakcji z identyfikowanym zasobem.

Przykłady identyfikatorów:

- ftp://randomftp.com/files/file.docx
- https://www.randomwebsite.pl/index.html
- mailto:jan.nowak@wp.pl
- tel:+48-25-123-88

Szczegóły dotyczące standardu URI są opisane w dokumencie RFC 3986 [1].

3.2 Hasło jednorazowe

3.3 Interfejs Windows Data Protection

4 Ataki na mechanizm OTP

4.1 Atak urodzinowy

4.2 Side-channel attacks

4.3 Atak przez powtórzenie

4.4 Atak „Man in the middle”

4.5 Phishing

5 PicnicAuth

5.1 Architektura projektu

5.2 Generowanie OTP po stronie użytkownika

5.3 Przechowywanie sekretu użytkownika

5.4 Przykład użycia projektu

5.5 Planowane ulepszenia

6 Zakończenie

6.1 Podsumowanie i wnioski

6.2 Podziękowania

7 Spis literatury

[1] T. Berners-Lee, R. Fielding, L. Masinter., *Uniform Resource Identifier (URI): Generic Syntax.*
`https://tools.ietf.org/pdf/rfc3986.pdf`, 2005

[2] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*
`https://tools.ietf.org/pdf/rfc4648`, SJD, 2006

[3] Oded Goldreich, *Foundations of Cryptography: Volume 1, Basic Tools* Cambridge University Press. ISBN 0-521-79172-3., 2001

[4] Arjen Lenstra, Xiaoyun Wang,and Benne de Weger. Colliding x.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005.

[5] Xiaoyun Wang, Hongbo Yu,Wei Wang, Haina Zhang,and Tao Zhan. *Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC*. EUROCRYPT 2009

[6] R. Rivest, *The MD5 Message-Digest Algorithm*
`https://tools.ietf.org/pdf/rfc1321.pdf`, MIT, 1992

[7] D. Eastlake, 3rd, P. Jones *US Secure Hash Algorithm 1 (SHA1)*
`https://tools.ietf.org/pdf/rfc3174`, 2001

[8] Laurens Van Houtven (lvh), *Crypto 101*
`https://www.crypto101.io`, 2017

[9] Keccak Team, *Strengths of Keccak - Design and security*
`https://keccak.team/keccak_strengths.html`, 2017

[10] Niels Ferguson, Bruce Schneier, Todayoshi Kohno *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010

[11] Marc Stevens, Pierre Karpman, Thomas Peyrin, *The SHAppening: freestart collisions for SHA-1*.
`https://sites.google.com/site/itstheshappening`, 2015

[12] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, Yarik Markov *The first collision for full SHA-1*.
`https://shattered.io/static/shattered.pdf`, CWI Amsterdam, Google Research, 2017

[13] Thai Duong, Juliano Rizzo, *Here Come The $\oplus$ Ninjas*.
`https://bug665814.bmoattachments.org/attachment.cgi?id=540839`, 2011

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW w tym w Archiwum Prac Dyplomowych SGGW.

. . . . . .

(czytelny podpis autora pracy)