

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. Alexandra 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 dyplому lub uzyskaniem tytułu zawodowego.

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

Data Podpis autora pracy

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

Spis treści

1 Wstęp	9
1.1 Cel pracy	9
1.2 Pojęcie uwierzytelnienia wielopoziomowego	9
1.3 Korzyści płynące z używania uwierzytelnienia wielopoziomowego	9
2 Elementy kryptografii	10
2.1 Kryptografia symetryczna oraz asymetryczna	10
2.1.1 Szyfrowanie symetryczne	10
2.1.2 Szyfrowanie asymetryczne	10
2.2 Szyfry blokowe	11
2.2.1 Tryby pracy szyfrów blokowych	11
2.2.2 Data Encryption Standard	14
2.2.3 Advanced Encryption Standard	15
2.3 Szyfry strumieniowe	16
2.3.1 RC4	17
2.3.2 Salsa20	18
2.4 Kryptograficzna funkcja skrótu	19
2.4.1 Message Digest 5	19
2.4.2 Secure Hash Algorithm 1	21
2.4.3 Secure Hash Algorithm 2	23
2.4.4 Secure Hash Algorithm 3	25
2.5 Uwierzytelnienie i integralność	25
2.5.1 Kod uwierzytelnienia wiadomości	25
2.5.2 MAC bazujący na funkcji skrótu	26
2.6 Generatory liczb losowych	27
2.6.1 Pojęcie entropii	27
2.6.2 Generatory prawdziwie losowych liczb	28
2.6.3 Kryptograficzne generatory liczb pseudolosowych	28
3 Kryptografia w praktyce	29
3.1 Pojęcia pomocnicze	29
3.1.1 Kodowanie transportowe	29
3.1.2 Czas uniksowy	30
3.1.3 Ujednolicony identyfikator zasobów	30
3.2 Hasło jednorazowe	31
3.2.1 Hasło oparte o HMAC	31
3.2.2 Hasło oparte o czas	32
3.2.3 Porównanie HOTP i TOTP	32

3.2.4	Porównanie kanałów dostarczania	32
3.3	Interfejs Windows Data Protection	34
4	PicnicAuth	35
4.1	Architektura projektu	35
4.1.1	REST API	35
4.1.2	Frontend	37
4.1.3	Biblioteki klienckie	37
4.2	Tworzenie konta podmiotu	38
4.3	Generacja sekretu użytkownika	39
4.4	Pobranie użytkowników powiązanych z podmiotem	39
4.5	Dostarczanie sekretu na urządzenie mobilne	40
4.6	Generowanie OTP po stronie serwera	43
4.6.1	Generowanie haseł typu HOTP	43
4.6.2	Generowanie haseł typu TOTP	44
4.7	Generowanie OTP po stronie użytkownika	44
4.8	Przechowywanie sekretu użytkownika	45
4.9	Walidacja hasła jednorazowego	46
4.10	Zmiana sekretu użytkownika	46
4.11	Przykład użycia projektu	46
4.12	Planowane ulepszenia	46
4.12.1	Generowanie hasła po stronie użytkownika	46
4.12.2	Biblioteki klienckie	46
4.12.3	Architektura	48
4.12.4	Konteneryzacja części serwerowej	48
4.12.5	Przechowywanie sekretu	49
5	Zakończenie	53
5.1	Podsumowanie i wnioski	53
5.2	Podziękowania	53
Spis rysunków		54
6	Bibliografia	55

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 - publiczny oraz prywatny. 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 jest możliwe odzyskanie danych wejściowych tylko za pomocą 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 rozsądny 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. Na ogół wykorzystywane są one tylko do bezpiecznego przesyłania klucza symetrycznego, który jest używany do szyfrowania pozostały komunikacji. Spowodowane jest to tym, że algorytmy symetryczne są dużo bardziej wydajne niż te 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

Niezwyczajną 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 czyli 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, należałyby sprawdzić aż $(2^{128})!$ możliwych kluczy.

2.2.1 Tryby pracy szyfrów blokowych

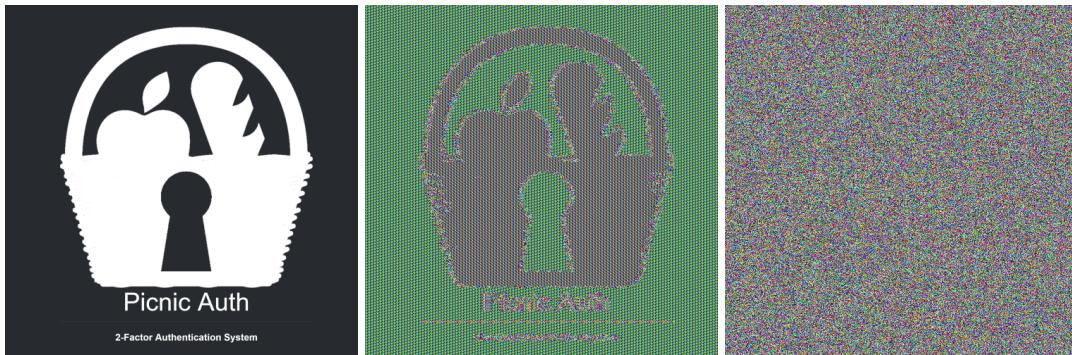
Nietrudno zauważyc jedną wadę szyfrów blokowych, która znacząco ogranicza proces szyfrowania. A mianowicie używając szyfrów blokowych, jesteśmy w stanie zaszyfrować dane o długości równej długości jednego bloku. Zwykle chcielibyśmy jednak szyfrować wiadomości dłuższe niż 16 czy 32 bajty. Z tego powodu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

ECB

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

Tryb ten nosi nazwę *ECB (elektroniczna książka kodowa)*.

Posiada on niezwykle niebezpieczną wadę. Dany blok wiadomości zostanie zaszyfrowany



(a) Logo projektu.

(b) AES w trybie ECB.

(c) AES w trybie CBC.

Rysunek 2.1. Wizualizacja trybów szyfrowania.

do zawsze tego samego bloku szyfrogramu.

Przykładowo mając wiadomość:

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

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

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

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*.

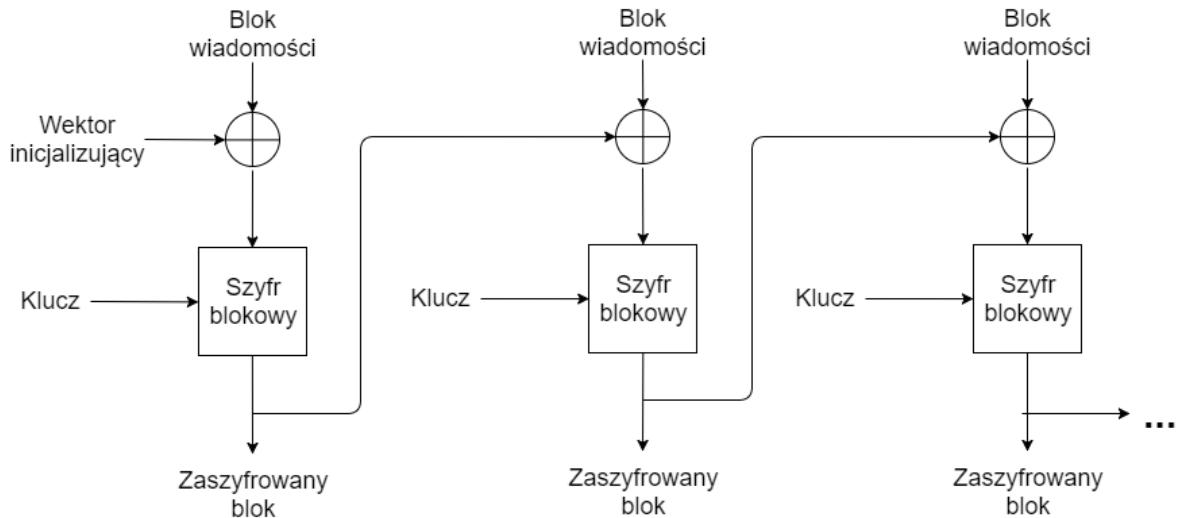
CBC

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*.

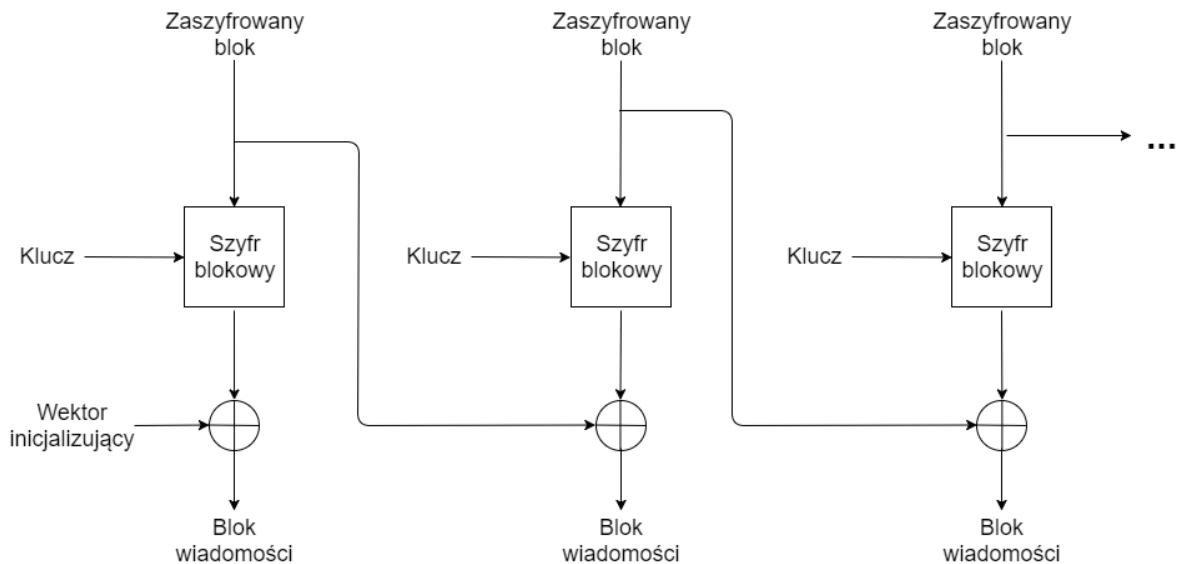
Wektor ten powinien być niemożliwy do zgadnięcia a w idealnej sytuacji powinien on być kryptograficznie losowy. Nie jest konieczne, aby wektor inicjalizujący był tajny, zwykle jest wysyłany razem z zaszyfrowaną wiadomością. Ważne jest jednak, aby atakujący nie był w stanie go przewidzieć przed samym 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.

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



Rysunek 2.2. Schemat szyfrowania w trybie CBC.



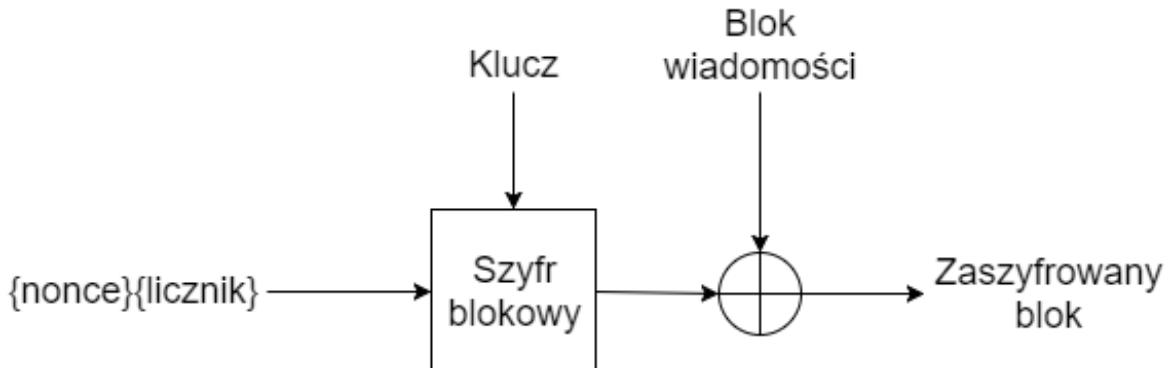
Rysunek 2.3. Schemat deszyfrowania w trybie CBC.

1. Używanie tego samego wektora inicjalizującego dla więcej niż jednej wiadomości.
2. Używanie przewidywalnych wektorów inicjalizujących, na przykład bazujących na liczniku.
3. Używanie klucza jako wektora inicjalizującego.

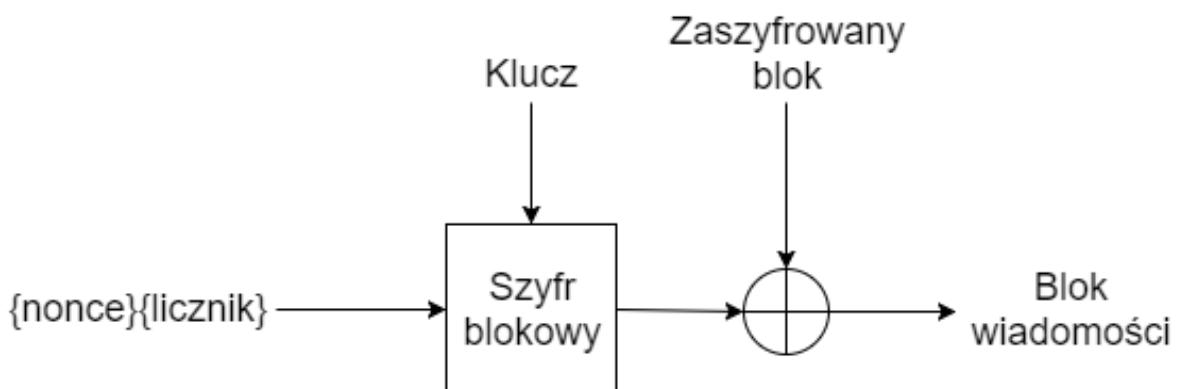
Jednym z najbardziej znanych ataków na system bazujący na trybie CBC był atak o nazwie *BEAST* [18]. Błędem, który został wykorzystany, było użycie zaszyfrowanego bloku poprzedniej wysłanej wiadomości jako wektora inicjalizującego.

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 wspomniana liczba połączona



Rysunek 2.4. Schemat szyfrowania w trybie CTR.



Rysunek 2.5. Schemat deszyfrowania w trybie CTR.

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 Data Encryption Standard

Szyfr blokowy DES był pierwszym algorytmem, zatwierdzonym przez *Narodowe Biuro Standaryzacji* (obecnie *Narodowy Instytut Standaryzacji i Technologii*), jako bezpieczny standard szyfrowania symetrycznego. Przez ponad 20 lat stowarzyszenia naukowe poddawały go intensywnym analizom.

W tym czasie został wielokrotnie złamany, głównie z powodu długość klucza jaką posiada szyfr - zaledwie 56 bitów.

Jednym ze skutecznych ataków na DES było przedsięwzięcie, dokonane w styczniu 1999 roku, składające się z kooperacji platformy *distributed.net* oraz urządzenia stworzonego przez *Electronic Frontier Foundation* o nazwie *Deep Crack*. Atak bazował wyłącznie na metodzie siłowej, należało więc sprawdzić $2^{56} = 72\ 057\ 594\ 037\ 927\ 936$ różnych kluczy. W mniej niż 24 godziny szyfr DES został sukcesywnie złamany.

Dla porównania przy dzisiejszej mocy obliczeniowej może on zostać złamany przez jedną maszynę w około jeden dzień [19].

Próbą uratowania szyfru była propozycja algorytmu 3DES. Polegał on na zaszyfrowaniu danych szyfrem DES, odszyfrowaniu ich, a następnie ponownym zaszyfrowaniem. Zamiast 3-krotnego szyfrowania, 3DES został wybrany schemat operacji umożliwiający zachowanie

kompatybilności z systemami, w których możliwe było użycie jedynie wersji podstawowej algorytmu.

Pomimo zwiększonej długości klucza w przypadku 3DES, w porównaniu do AES jest mniej bezpieczny i nadzwyczajnie wolny (AES potrzebuje 12.6 cykli procesora do zaszyfrowania bajtu danych, 3DES aż 134.5 [13]).

2.2.3 Advanced Encryption Standard

Po problemach z DES agencja NIST ogłosiła konkurs na algorytm, który mógłby stać się nowym standardem. Do konkursu zgłoszone zostały między innymi: *Rijndael*, *Serpent*, *Twofish*, *MARS* czy *RC6*. Zostały one poddane intensywnej kryptoanalizie. Ostatecznie wygrał algorytm stworzony przez Vincenta Rijmena oraz Joana Daemena - *Rijndael*.

Rijndael jest rodziną szyfrów, których rozmiar bloku oraz rozmiar klucza może wynosić: 128, 160, 192, 224 i 256 bitów. Za standard (AES) uznany został *Rijndael* z długością bloku równą 128 bitów oraz rozmiarze klucza równym 128, 192 lub 256 bitów.

Algorytm składa się z wielu przebiegów. Liczba przebiegów zależy od rozmiaru klucza. Pierwszym etapem algorytmu jest rozszerzenie klucza. Jako że, w każdym przebiegu potrzebny jest 128-bitowy klucz, zachodzi konieczność wygenerowania wielu kluczy, na podstawie głównego klucza.

W pierwszej kolejności główny klucz jest ładowany do macierzy 4 na 4 bajty. Ostatnia kolumna jest obracana (bajt z góry jest wstawiany na dół) a następnie wartość każdej komórki z obróconej kolumny podawana jest do nieliniowej funkcji zamiany, specyficznej dla algorytmu AES, zamieniającej podany bajt na inny. Następnie kolumna ta jest XORowana ze stałą przebiegu (inną dla każdego przebiegu). Na koniec XORowana jest ona z pierwszą kolumną klucza poprzedniego przebiegu.

Pozostałe kolumny liczone są XORując poprzednią kolumnę z kolumną na tym samym miejscu, ale z klucza poprzedniego przebiegu.

Po wygenerowaniu klucza, algorytm rozpoczyna przebiegi.

Każdy przebieg składa się z następujących kroków:

1. Zamiana bajtów.

Funkcja zamiany algorytmu AES jest złożeniem dwóch funkcji: $f(g(a_{xy}))$.

Wynikiem wewnętrznej funkcji g jest liczba odwrotna do zadanej nad ciałem skończonym. Jest to równoznaczne z zamianą zadanej liczby do postaci wielomianowej (wielomian o maksymalnym stopniu równym 7, w którym współczynnikami są bity liczby) a następnie znalezienie takiego wielomianu, który po przemnożeniu przez wielomian liczby, modulo $x^8 + x^4 + x^3 + x + 1$, da w wyniku 1.

Funkcja f jest przekształceniem wyrażanym następująco:

Możliwe jest też jednorazowe wygenerowanie tablicy odwzorowania dla zadanych wielomianów, dzięki czemu zamiast za każdym razem obliczać wartość tej funkcji, wystarczy odczytać wartość tablicy pod odpowiednim indeksem. Przykładowo bajt 0xc4, za pomocą powyższej tablicy, konwertowany jest do wartości 28.

2. Przesuwanie wierszy.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Rysunek 2.6. Przekształcenie f .

Każdy z wierszy jest przesuwany o odpowiednio 0, 1, 2, 3 komórki w lewo. Komórki macierzy, które wyszły poza macierz są dostawiane z prawej strony.

3. Mieszanie kolumn.

Kolumna zapisywana jest w postaci wielomianu, następnie wielomian ten jest mnożony przez specjalny wielomian $03x^3 + 01x^2 + 01x + 02$. Na koniec brana jest reszta z dzielenia przez wielomian $x^4 + 1$.

Całość powyższych operacji można uproszczyć do mnożenia macierzowego:

$$\begin{bmatrix} 2113 \\ 3211 \\ 1321 \\ 1132 \end{bmatrix} \times \begin{bmatrix} b_{01} \\ b_{12} \\ b_{23} \\ b_{30} \end{bmatrix} = \begin{bmatrix} c_{01} \\ c_{11} \\ c_{21} \\ c_{31} \end{bmatrix}$$

W przypadku ostatniego przebiegu krok ten jest pomijany.

4. Dodanie klucza przebiegu.

Ostatnim etapem przebiegu jest użycie funkcji XOR pomiędzy każdym elementem macierzy a każdym elementem klucza przebiegu.

W celu odszyfrowania danych cały proces jest powtarzany w odwrotnej kolejności.

Na chwile obecną nie są znane żadne praktyczne ataki na AES a co do szybkości działania na współczesnych komputerach, znaczco przyczynił się producent procesorów, implementując w nich natywne instrukcje procesora, których jedynym zadaniem jest realizacja algorytmu 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 blokowych, 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 po-

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x00	99	124	119	123	242	107	111	197	48	1	103	43	254	215	171	118
0x10	202	130	201	125	250	89	71	240	173	212	162	175	156	164	114	192
0x20	183	253	147	38	54	63	247	204	52	165	229	241	113	216	49	21
0x30	4	199	35	195	24	150	5	154	7	18	128	226	235	39	178	117
0x40	9	131	44	26	27	110	90	160	82	59	214	179	41	227	47	132
0x50	83	209	0	237	32	252	177	91	106	203	190	57	74	76	88	207
0x60	208	239	170	251	67	77	51	133	69	249	2	127	80	60	159	168
0x70	81	163	64	143	146	157	56	245	188	182	218	33	16	255	243	210
0x80	205	12	19	236	95	151	68	23	196	167	126	61	100	93	25	115
0x90	96	129	79	220	34	42	144	136	70	238	184	20	222	94	11	219
0xa0	224	50	58	10	73	6	36	92	194	211	172	98	145	149	228	121
0xb0	231	200	55	109	141	213	78	169	108	86	244	234	101	122	174	8
0xc0	186	120	37	46	28	166	180	198	232	221	116	31	75	189	139	138
0xd0	112	62	181	102	72	3	246	14	97	53	87	185	134	193	29	158
0xe0	225	248	152	17	105	217	142	148	155	30	135	233	206	85	40	223
0xf0	140	161	137	13	191	230	66	104	65	153	45	15	176	84	187	22

Rysunek 2.7. Macierz odwzorowania funkcji zamiany.

między 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.

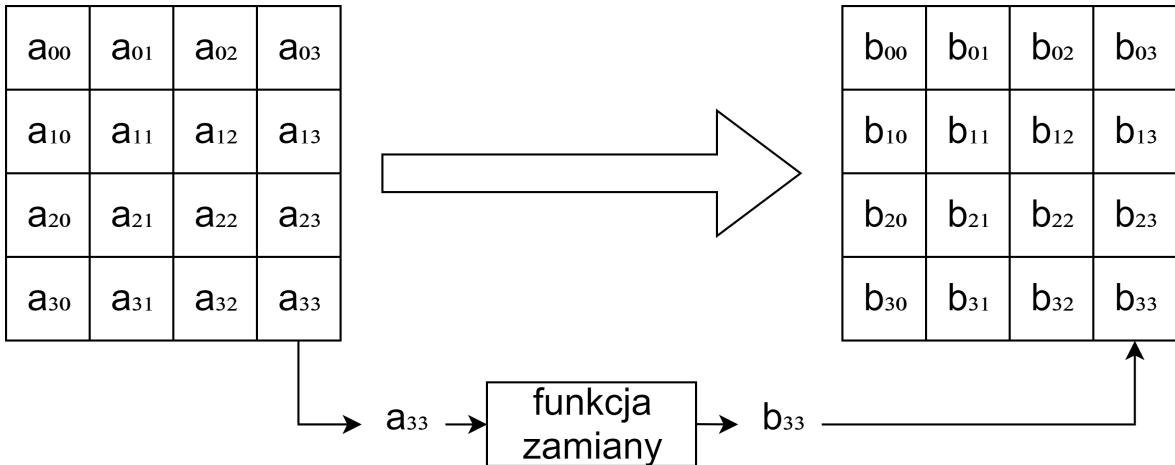
2.3.1 RC4

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.

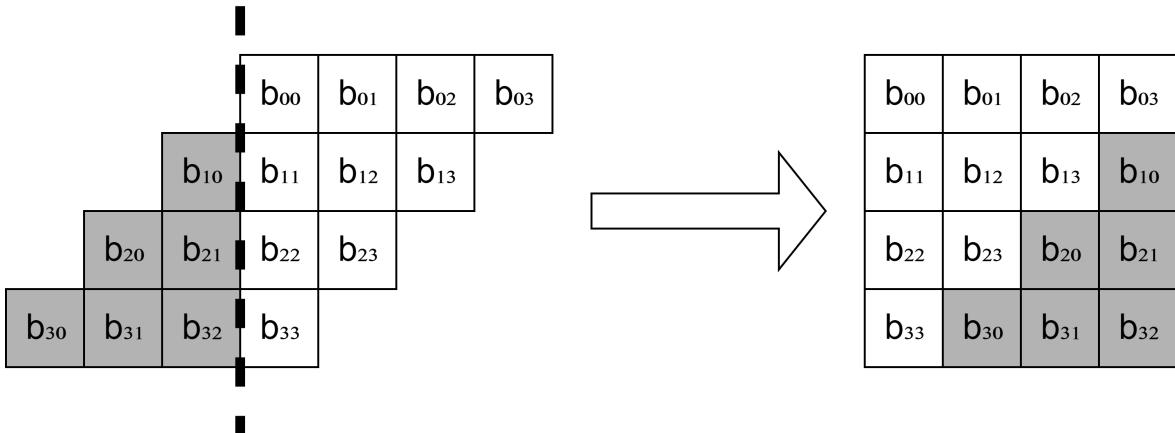
Schemat działania algorytmu jest niezwykle prosty. Składa się z inicjalizacji klucza oraz generacji pseudolosowego ciągu.

W etapie inicjalizacji klucza tworzona jest tablica permutacji S, składającej 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 indeksów 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.



Rysunek 2.8. Wizualizacja zamiany bajtów.



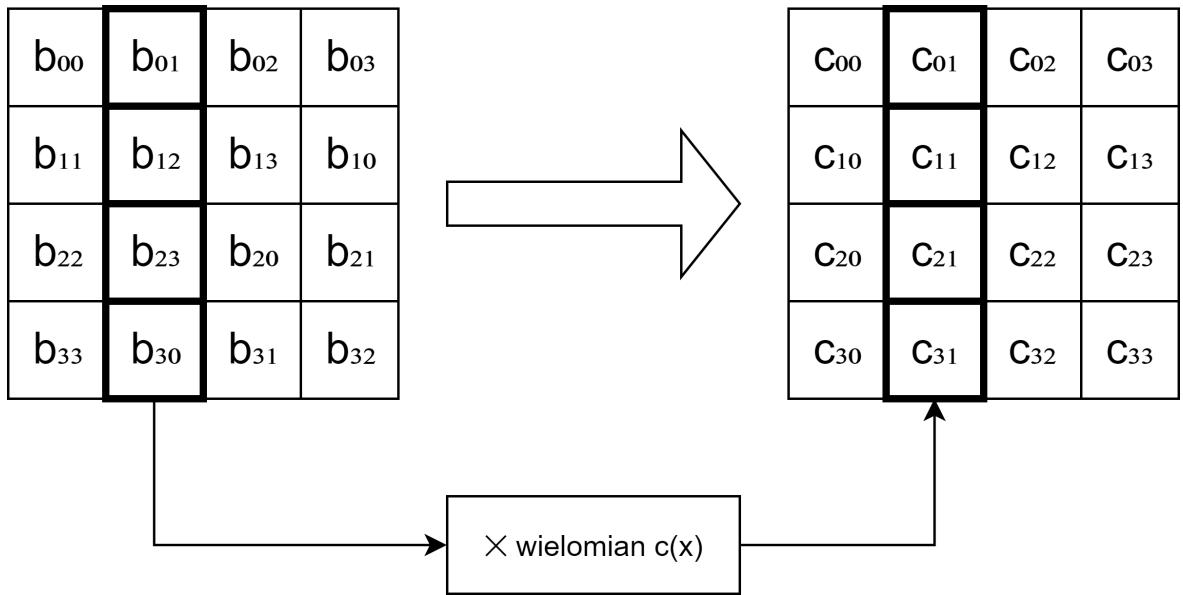
Rysunek 2.9. Wizualizacja przesunięcia wierszy.

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 operacjach 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 i uzyskiwanie informacji na podstawie wyników pomiarów.

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.



Rysunek 2.10. Wizualizacja mieszania kolumn.

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 Bloom'a), algorytmy dopasowania wzorca (algorytm Karpa-Rabina) czy też w kriptografii.

Aby funkcja skrótu mogła zostać użyta w systemach kryptograficznych musi posiadać szereg parametrów.

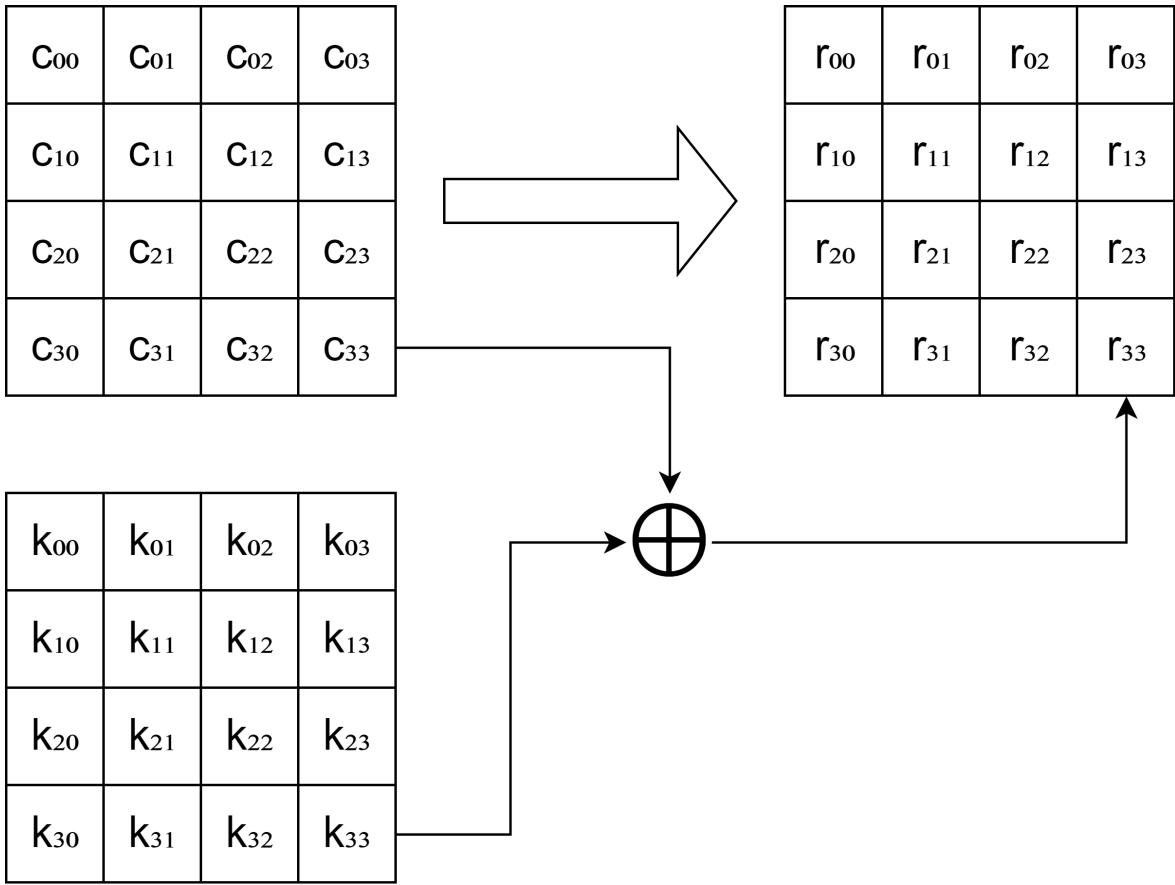
Jednym z nich jest odporność na kolizje. Kolizją nazywamy przypadek, gdy dla dwóch różnych argumentów funkcja skrótu zwraca ten sam wynik. Nie jest oczywiście możliwe, aby móc całkowicie uniknąć kolizji, gdyż zbiór danych o dowolnym rozmiarze jest mapowany na zbiór skończony, zależy nam jednak aby proces 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.



Rysunek 2.11. Wizualizacja dodania klucza przebiegu.

W uproszczeniu, algorytm MD5 można przedstawić w następujących krokach [13]:

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 dopełnienia) 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 algorytmu (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 stan zmiennych.

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

Szczegółowa specyfikacja algorytmu znajduje się w dokumencie RFC 1321 [5].

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].

2.4.2 Secure Hash Algorithm 1

SHA-1 jest funkcją bazującą na MD4 (podobnie jak MD5), o długości skrótu wynoszącej 160 bitów [15].

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:

$$A := 1732584193$$

$$B := 4023233417$$

$$C := 2562383102$$

$$D := 271733878$$

$$E := 3285377520$$

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:

$$w[i] := (w[i - 3] \oplus w[i - 8] \oplus w[i - 14] \oplus w[i - 16]) \ll 1$$

(c) Inicjalizacja zmiennych pomocniczych dla danego bloku

$$\begin{aligned}a &:= A \\b &:= B \\c &:= C \\d &:= D \\e &:= E\end{aligned}$$

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

i. jeżeli $0 \leq i \leq 19$

$$\begin{aligned}f &:= (b \& c) \mid ((\sim b) \& d) \\k &:= 0x5A827999\end{aligned}$$

ii. jeżeli $20 \leq i \leq 39$

$$\begin{aligned}f &:= b \oplus c \oplus d \\k &:= 0x6ED9EBA1\end{aligned}$$

iii. jeżeli $40 \leq i \leq 59$

$$\begin{aligned}f &:= (b \& c) \mid (b \& d) \mid (c \& d) \\k &:= 0x8F1BBCDC\end{aligned}$$

iv. jeżeli $60 \leq i \leq 79$

$$f := b \oplus c \oplus d \quad k := 0xCA62C1D6$$

v. następnie

$$\begin{aligned}t &:= (a <<< 5) + f + e + k + w[i] \\e &:= d \\d &:= c \\c &:= b <<< 30 \\b &:= a \\a &:= t\end{aligned}$$

(e) Aktualizacja wewnętrznego stanu funkcji

$$\begin{aligned}A &:= A + a \\B &:= B + b \\C &:= C + c \\D &:= D + d \\E &:= E + e\end{aligned}$$

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 [12].

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 [16].

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. [17]

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
- 256 bitów

SHA-256

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

1. Inicjalizacja stanu wewnętrznego funkcji:

$$A := 1779033703$$

$$B := 3144134277$$

$$C := 1013904242$$

$$D := 2773480762$$

$$E := 1359893119$$

$$F := 2600822924$$

$$G := 528734635$$

$$H := 1541459225$$

2. Inicjalizacja pomocniczych stałych.

$$\begin{aligned}
& 1116352408, 1899447441, 3049323471, 3921009573, \\
& 961987163, 1508970993, 2453635748, 2870763221, \\
& 3624381080, 310598401, 607225278, 1426881987, \\
& 1925078388, 2162078206, 2614888103, 3248222580, \\
& 3835390401, 4022224774, 264347078, 604807628, \\
& 770255983, 1249150122, 1555081692, 1996064986, \\
& 2554220882, 2821834349, 2952996808, 3210313671, \\
k[0..63] := & 3336571891, 3584528711, 113926993, 338241895, \\
& 666307205, 773529912, 1294757372, 1396182291, \\
& 1695183700, 1986661051, 2177026350, 2456956037, \\
& 2730485921, 2820302411, 3259730800, 3345764771, \\
& 3516065817, 3600352804, 4094571909, 275423344, \\
& 430227734, 506948616, 659060556, 883997877, \\
& 958139571, 1322822218, 1537002063, 1747873779, \\
& 1955562222, 2024104815, 2227730452, 2361852424, \\
& 2428436474, 2756734187, 3204031479, 3329325298
\end{aligned}$$

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:

$$\begin{aligned}
s0 &:= (w[i-15] \ggg 7) \oplus (w[i-15] \ggg 18) \oplus (w[i-15] \gg 3) \\
s1 &:= (w[i-2] \ggg 17) \oplus (w[i-2] \ggg 19) \oplus (w[i-2] \gg 10) \\
w[i] &:= w[i-16] + s0 + w[i-7] + s1
\end{aligned}$$

- (c) Inicjalizacja zmiennych pomocniczych dla danego bloku

$$\begin{aligned}
a &:= A \\
b &:= B \\
c &:= C \\
d &:= D \\
e &:= E \\
f &:= F \\
g &:= G \\
h &:= H
\end{aligned}$$

- (d) Dla każdej z 64 części $w[i]$ należy wykonać:

$$\begin{aligned}
S1 &:= (e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25) \\
ch &:= (e \& f) \oplus ((\sim e) \& g) \\
t1 &:= h + S1 + ch + k[i] + w[i] \\
S0 &:= (a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22) \\
m &:= (a \& b) \oplus (a \& c) \oplus (b \& c) \\
t2 &:= S0 + m \\
h &:= g
\end{aligned}$$

```

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

```

- (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:

```

 $(A << 224) | (B << 192) | (C << 160) |$ 
 $(D << 128) | (E << 96) | (F << 64) | (G << 32) | H$ 

```

W porównaniu do SHA-1, funkcje SHA-2 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 [14].

2.5 Uwierzytelnienie i integralność

2.5.1 Kod uwierzytelnienia wiadomości

W celu zagwarantowania prywatności kanału komunikacji samo szyfrowanie nie jest wystarczające. Potwierdzeniem tej tezy są ataki typu „*Man in the middle*”, w których atakujący

podsywa się pod osobę, do której przesyłamy wiadomość. Wysyłając wiadomość jesteśmy przekonani, że wysyłamy ją do docelowej osoby, gdy tak naprawdę wysyłamy ją do atakującego. Odbiorca również sądzi, że otrzymana wiadomość jest wysłana przez spodziewanego nadawcę, mimo że wysłał ją atakujący. Pomimo tego, że atakujący nie jest w stanie przeczytać przechwyconej wiadomości, gdyż jest zaszyfrowana, to może próbować ją modyfikować. Kody uwierzytelnienia wiadomości (Message Authentication Code ang.) są jedną z metod na uwierzytelnienie i zapewnienie integralności danych. Przesyłane są razem z wiadomością w formie niewielkiej porcji danych. Algorytmy generujące MAC przyjmują klucz kryptograficzny o z góry określonej długości oraz dowolne dane o dowolnej długości (podobnie jak w przypadku funkcji skrótu) i w wyniku zwracają kod o ustalonej długości. Algorytmy te umożliwiają również weryfikację danego kodu na podstawie klucza oraz wiadomości. Po wygenerowaniu kodu uwierzytelniającego pozostaje jeszcze kwestia sposobu dołączenia go do wiadomości. Sposobów tych je wiele.

Przykładowo możemy w pierwszej kolejności wygenerować MAC z danej wiadomości a następnie ją zaszyfrować. Wysyłana jest wtedy zaszyfrowana wiadomość oraz wygenerowany kod. Jednym z protokołów, w których MAC jest wykorzystywany w taki sposób jest protokół SSH.

Możliwe jest też wygenerowanie MAC, doklejenie go do wiadomości i zaszyfrowanie całości. Ten schemat wykorzystywany jest w protokole TLS.

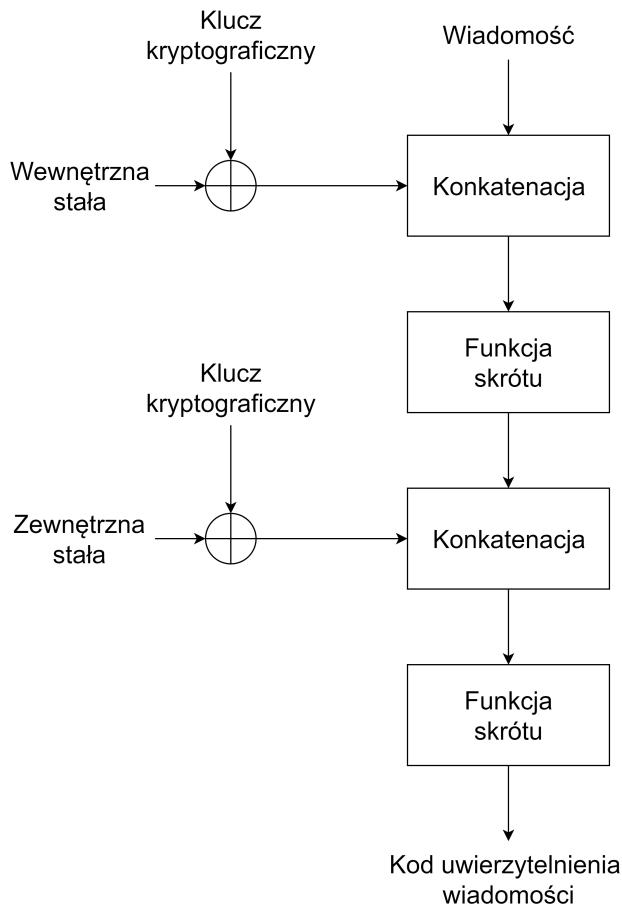
Trzecim sposobem jest zaszyfrowanie wiadomości, po czym wygenerowanie MAC z zaszyfrowanej wiadomości. Tego sposobu używa na przykład protokół IPSec.

Okazuje się, że ostatni wspomniany sposób jest najbezpieczniejszy, należy sprawdzić kod uwierzytelnienia wiadomości przed próbą jej odszyfrowania. W przypadku protokołu SSH, który wykorzystuje pierwszy sposób, istnieje atak, który umożliwia atakującemu odszyfrowanie pierwszych czterech bajtów zaszyfrowanego bloku. Jest to spowodowane specyficzną konstrukcją pakietu SSH (pole reprezentujące długość pakietu jest również zaszyfrowane). Na drugi sposób również istnieje atak zaproponowany przez Serge'a Vaudenaya [22].

2.5.2 MAC bazujący na funkcji skrótu

Praktycznie jedynym poprawnie działającym mechanizmem generującym kody uwierzytelnienia wiadomości są algorytmy MAC bazujące na funkcji skrótu (Hash-based Message Authentication Code ang.). Oprócz klucza kryptograficznego oraz wiadomości algorytm HMAC pobiera dodatkowo funkcję skrótu. Jako argument może zostać użyta dowolna funkcja skrótu: SHA1, SHA-256, SHA-3 czy nawet MD5. Nie jest tutaj wymagane aby użyta funkcja skrótu była odporna na kolizję więc HMAC zbudowany na funkcji MD5 jest uważany za bezpieczny [23], chociaż nie jest to zalecane przy tworzeniu nowych systemów kryptograficznych.

Schemat generacji HMAC można przedstawić następująco: Wartości stałych nie mają tutaj większego znaczenia. Ważne jest jednak aby miały długość równą długości bloku użytej funkcji skrótu oraz aby były od siebie różne. Na etapie konkatenacji wynik działania funkcji XOR jest dostawiany z lewej strony.



Rysunek 2.12. Schemat HMAC.

2.6 Generatory liczb losowych

2.6.1 Pojęcie entropii

Entropia jest miarą średniej ilości informacji w danej wiadomości. Im więcej informacji jest zawartych w wiadomości, tym większa jest jej entropia.

Przykładowo dla wiadomości składającej się z samych bitów o wartości 1, entropia wynosi 0 bitów, gdyż wybierając losowo jeden z bitów, ze stuletnim prawdopodobieństwem otrzymamy bit o wartości 1. Mając wiadomość o długości 64 bity, w której każdy bit ma losową wartość, entropia wynosi 64 bity. W przypadku wiadomości o długości 32 znaki, w której mogą pojawić się litery: a, b, c i d, a prawdopodobieństwo wystąpienia każdej z liter jest równe 25% entropia wynosi 2 bity.

Wartość entropii jest liczona za pomocą wzoru:

$$H(X) := - \sum_x P(X = x) \log_2 P(X = x)$$

$P(X = x)$ jest prawdopodobieństwem, że zmienna X przybierze wartość x .

W zastosowaniach kryptograficznych konieczne jest aby entropia kluczy była wysoka. W prze-

ciwnym razie pojawi się możliwość złamania kluczy metodą siłową.

2.6.2 Generatory prawdziwie losowych liczb

Oczywistym jest fakt, że przy użyciu wyłącznie deterministycznych metod nie jesteśmy w stanie stworzyć generatorów prawdziwie losowych liczb. Do stworzenia takich generatorów konieczne jest bazowanie na zjawiskach fizycznych takich jak procesy termiczne czy procesy kwantowe.

Jednym z takich generatorów dostępnych publicznie jest platforma *RANDOM.ORG*, wykorzystująca szумy atmosferyczne do generacji liczb prawdziwie losowych.

2.6.3 Kryptograficzne generatory liczb pseudolosowych

Idealnym byłoby używanie generatorów prawdziwie losowych liczb do zastosowań kryptograficznych, takich jak na przykład generacja klucza symetrycznego. Często jednak nie mamy dostępu do takich generatorów lub użycie ich byłoby zbyt kosztowne. Szczęśliwie okazuje się, że do codziennych zastosowań wystarczającym jest użycie kryptograficznie bezpiecznych generatorów liczb pseudolosowych.

Algorytmami używanymi w kriptografii do generacji liczb pseudolosowych są:

- *Yarrow*
- *Blum Blum Shub*
- *Dual_EC_DRBG*
- *Mersenne Twister*

Warto zaznaczyć, że samodzielna implementacja algorytmów jest wyjątkowo niewskazana. Jest to obszar kriptografii, w którym niezwykle łatwo jest popełnić błąd, który będzie powodował złamanie całego systemu kryptograficznego.

W razie potrzeby wygenerowania liczb pseudolosowych zalecane jest użycie interfejsu wystawionego przez używany system operacyjny. Na systemach z rodziny *Unix* interfejsem tym jest */dev/urandom*, zaś na systemach *Windows* należy korzystać z *CryptGenRandom*.

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 funkcję 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

Hasło jednorazowe jest jedną z możliwych form uwierzytelnienia przy używaniu uwierzytelnienia wieloetapowego.

W odróżnieniu od zwykłego, statycznego hasła, haseł jednorazowych można użyć tylko raz. Dzięki czemu przechwycenie przez atakującego, raz użytego hasła, nie daje możliwości uwierzytelnienia.

Hasła jednorazowe eliminują również problem zbyt niskiej entropii w hasłach. Często zdarza się, że hasła tworzone przez użytkowników są zbyt słabe, przez co nie zapewniają wystarczającego poziomu bezpieczeństwa. Hasła jednorazowe powstają przy użyciu bezpiecznych generatorów liczb pseudolosowych, dzięki czemu zapewniony jest wystarczający poziom entropii (pod warunkiem, że generatory te są w odpowiedni sposób używane).

Istnieje wiele metod generacji haseł jednorazowych o różnym stopniu bezpieczeństwa. Niektóre z tych metod to:

1. Użycie łańcucha wyników funkcji skrótu.
2. Użycie mechanizmu HMAC w połączeniu z synchronizowanym licznikiem.
3. Użycie mechanizmu HMAC z synchronizacją czasu.

Standard uwierzytelniania za pomocą haseł jednorazowych zdefiniowany jest w dokumencie *RFC 2289* [6].

3.2.1 Hasło oparte o HMAC

Do generacji haseł tego typu wykorzystywana jest funkcja HMAC. Może być tu użyta dowolna funkcja skrótu, jednak musi być ona taka sama po stronie generującej jak i weryfikującej. Na wejście funkcji podawane są dwa argumenty, symetryczny klucz oraz licznik. Konieczne jest aby obie wartości były zsynchronizowane pomiędzy użytkownikiem a stroną uwierzytelniającą. Przy każdej udanej walidacji hasła jednorazowego licznik zwiększany jest o 1. Przykładowo używając funkcji skrótu SHA-1, wynik HMAC-SHA-1 o długości 160 bitów jest obcinany do kilku znakowego hasła. Hasło powinno być używane w postaci numerycznej a za minimalną bezpieczną długość hasła uznawane jest sześć cyfr. Minimalną długością klucza kryptograficznego jest 128 bitów, rekomendowane jest jednak używanie przynajmniej 160-bitowego klucza.

Proces generacji hasła jednorazowego można przedstawić następująco:

$$HOTP(K, C) = Przytnij(HMAC-SHA-1(K, C))$$

Przycięcie przebiega w dwóch etapach. W pierwszym etapie 160-bitowy wynik funkcji HMAC skracany jest do czterech bajtów. Z 160 bitów pobierane są 4 ostatnie bity. Wykorzystywane są one jako indeks, od którego pobieramy 4 bajty z wyniku funkcji HMAC. Drugi etap polega na obliczeniu reszty z dzielenia z poprzedniego wyniku przez 10^n , gdzie n jest liczbą cyfr jaką chcemy ostatecznietrzymać. Algorytm jak i przykładowa implementacja w języku Java znajduje się w dokumencie *RFC 4226* [7].

3.2.2 Hasło oparte o czas

Algorytm generacji haseł opartych o czas jest ulepszoną wersją poprzedniego. Ponownie używana jest funkcja HMAC, jednak zamiast licznika wykorzystywany jest czas uniksowy. Przed podaniem czasu uniksowego do funkcji HMAC jest on dzielony przez okno czasowe czyli przez liczbę sekund, jaką dane hasło jest ważne. Okno czasowe zwykle jest ustawiane na 30 lub 60 sekund.

Wymogi dotyczące klucza kryptograficznego jak również proces przycinania wyniku funkcji HMAC do postaci hasła jednorazowego jest identyczny jak w poprzednim algorytmie. W przypadku haseł opartych o czas konieczna jest synchronizacja zegarów pomiędzy stroną generującą hasło oraz stroną uwierzytelniającą.

Dokumentem, w którym opisany jest algorytm jest *RFC 6238* [8].

3.2.3 Porównanie HOTP i TOTP

Przewagą haseł opartych o czas jest ich krótki czas ważności. Ich użycie jest możliwe tylko w z góry określonym oknie czasowym. Ważność haseł opartych o licznik jest praktycznie nieograniczona, są ważne aż do momentu ich użycia.

W przypadku podejrzenia hasła opartego o czas, wygenerowanego na czymś telefonie, atakujący nie jest w stanie pójść do domu i spróbować użyć wykradzione hasło. Przechwycone hasło oparte o licznik jest możliwe do użycia w dowolnej chwili.

Drugim aspektem przemawiającym za używanie haseł opartych o czas jest kwestia ataków siłowych. Ataki siłowe skierowane bezpośrednio na hasło jednorazowe są nieco mniej efektywne, gdyż szukane hasło w każdym oknie czasowym jest inne.

3.2.4 Porównanie kanałów dostarczania

„Zdrapka”

Zwykle wysyłane pocztą w postaci dokumentu z wydrukowaną listą haseł lub zdrapki w rozmiarze karty bankomatowej. W celu potwierdzenia transakcji użytkownik proszony jest o podanie kolejnego z haseł zapisanych na zdrapce lub jedno z haseł ze zdrapki o numerze wybranym przez serwer w pseudolosowy sposób.

Zgubienie takiej karty z hasłami jednorazowymi jest równoznaczne ze złamaniem jednego z etapów w uwierzytelnianiu wieloetapowym. Również sam sposób dostarczania zdrapki w postaci listu nie jest uważany za całkowicie bezpieczny.

SMS

Obecnie najczęściej spotykanym kanałem dostarczania haseł jednorazowych są wiadomości SMS (Short Message Service). W momencie wykonywania akcji wymagającej dodatkowego uwierzytelnienia, na telefon komórkowy użytkownika wysyłane jest pojedyncze hasło jednorazowe.

Pomijając koszty związane z wysyłaniem wiadomości SMS, kanał ten posiada zastrzeżeń w kwestii bezpieczeństwa. Standard szyfrowania A5/x używany w wiadomościach został wielokrotnie złamany a wady protokołu SS7 (*Signaling System No. 7*), odpowiedzialnego za przekierowywanie wiadomości pozwalają na przekierowanie konkretnej wiadomości do atakującego.

W lipcu 2016 roku *Narodowy Instytut Standaryzacji i Technologii* wydał oświadczenie, w którym zalecane jest nieużywanie tego kanału przy implementacji uwierzytelnienia dwuetapowego.

Aplikacja mobilna

Możliwe jest też wygenerowanie haseł jednorazowych korzystając z aplikacji mobilnych dostępnych na smartfonach. Kod źródłowy takich aplikacji zwykle jest otwarty, dzięki czemu łatwo jest sprawdzić czy hasła generowane w aplikacji nie są wysyłane gdzieś poza telefon. Aplikacje te działają bez dostępu do sieci komórkowej czy do Internetu.

Bezpieczeństwo tego kanału jest silnie zależne od stanu bezpieczeństwa samego telefonu. Przykładowo blokada ekranu za pomocą silnego hasła zabezpiecza również dostęp do aplikacji generującej hasła jednorazowe.

Istnieje duża liczba aplikacji mobilnych, które można tutaj wykorzystać:

- *Google Authenticator*
- *Microsoft Authenticator*
- *Authy 2-Factor Authentication*
- *Duo Mobile*
- *HDE OTP*
- *Authenticator Plus*
- *FreeOTP*

Token sprzętowy

Token jest niewielkim urządzeniem z wyświetlaczem, na którym pojawiają się kolejne hasła jednorazowe.

Nieco bardziej bezpieczne tokeny oprócz wyświetlacza posiadają jeszcze klawiaturę numeryczną. Wymagane jest wtedy podanie PINu przed uzyskaniem hasła jednorazowego. Zgubienie lub kradzież tokenu nie jest wtedy problemem.

Tokeny sprzętowe pomimo posiadania wysokiego poziomu bezpieczeństwa wiążą się z dość wysokimi kosztami użytkowania w porównaniu do aplikacji mobilnych. Pojawia się również problem synchronizacji zegara w przypadku haseł opartych o czas.

Ołówek i kartka papieru

Pomimo, że może wydawać się to niedorzeczne, możliwe jest wygenerowanie hasła jednorazowego, opartego na liczniku, przy użyciu wyłącznie ołówka i kartki papieru. Z badań przeprowadzonych przez Kena Shirriffa [11] wynika, że obliczenie jednego kroku funkcji skrótu SHA-256 zajmuje około 16 minut i 45 sekund. Używając konstrukcji HMAC funkcja skrótu jest używa dwa razy. Nietrudno więc policzyć, że sam proces obliczania wyniku funkcji HMAC wyniósł by około 1.49 dnia.

Nie jest to efektywny sposób, lecz jeśli nie ufamy wszystkim poprzednim metodom i dysponujemy wystarczającą ilością wolnego czasu, możemy wygenerować hasło jednorazowe bez pomocy zewnętrznych systemów obliczeniowych.

3.3 Interfejs Windows Data Protection

„Jeśli wpisujesz do swojego kodu źródłowego litery A-E-S to robisz coś źle” [10]. Używając kryptografii zalecane jest pracować na jak najwyższym poziomie abstrakcji jaki tylko jest dostępny. Im głębiej wchodzi się w detale implementacyjne tym większa szansa, że gdzieś zostanie popełniony błąd. A w tej dziedzinie, jeden błąd wystarczy by zaburzyć bezpieczeństwo całego systemu kryptograficznego.

Windows Data Protection jest prostym, wysokopoziomowym interfejsem dostępnym w systemach *Windows*, począwszy od wersji *Windows 2000*. Podstawowym jego zadaniem jest zarządzanie kluczami kryptograficznymi, poprzez szyfrowanie symetryczne, wykorzystując klucze systemowe.

Z każdą kolejną wersją systemu *Windows* interfejs *Interfejs Windows Data Protection* jest wykorzystywany coraz powszechniej, obecnie wykorzystywany jest między innymi w takich miejscach jak:

- *Microsoft Outlook* przy standardzie *S/MIME*
- *Credential Manager*
- Przestrzeń *Microsoft.Owin*
- *Windows Live Mail*
- Hasła Zdalnego Pulpitu
- Hasła Sieci bezprzewodowych
- Szyfrowanie bazy danych *Microsoft SQL Server*
- Szyfrowany system plików EFS (Encrypting File System)

Bezpieczeństwo danych zaszyfrowanych tym interfejsem w znacznym stopniu zależne jest od hasła systemu *Windows*. W przypadku ujawnienia tego hasła lub ataku na bazę kont użytkowników system, zaszyfrowane dane narażone są na możliwe ataki.

Używane algorytmy kryptograficzne różnią się w zależności o wersji systemu operacyjnego. W systemie operacyjnym *Windows 7* do szyfrowania symetrycznego wykorzystywane jest szyfr *AES* w trybie *CBC* z kluczem o długości 256 bitów. Do generowania skrótów używana jest funkcja *SHA512* [9].

4 PicnicAuth

4.1 Architektura projektu

Projekt składa się z trzech komponentów: serwera w architekturze REST (Representational state transfer) API, informacyjnej strony internetowej oraz bibliotek klienckich. Komponenty te są od siebie niezależne, dzięki czemu łatwa jest rozbudowa projektu, jak również wersjonowanie każdego z nich.

4.1.1 REST API

Pierwszym z nich jest serwer implementujący logikę aplikacji. Jest on stworzony w technologii *ASP.NET Web API 2* w języku programowania *C#*.

Przy jego tworzeniu zostały zachowane zasady architektury REST, co pozwala na łatwą i sprawną integrację platform klienckim, niezależnie od użytej technologii.

Zasoby udostępniane przez REST API to:

- **[GET] /api/Companies/Me/AuthUsers**
Zwraca listę użytkowników dla aktualnie zalogowanego podmiotu.
- **[POST] /api/AuthUsers**
Tworzy nowego użytkownika i dodaje go do kolekcji użytkowników zalogowanego podmiotu. W odpowiedzi zwracany jest sekret stworzonego użytkownika, jak również link do kodu QR kompatybilnego z aplikacjami mobilnymi.
- **[PATCH] /api/AuthUsers/userId/secret**
Generuje nowy sekret dla użytkownika o podanym *userId*.
- **[GET] /api/Companies/Me**
Zwraca dane zalogowanego podmiotu, takie jak login, adres poczty elektronicznej oraz unikalny identyfikator.
- **[POST] /api/Companies**
Tworzy nowe konto podmiotu.
- **[GET] /api/AuthUsers/userId/hotp**
Zwraca hasło jednorazowe typu *HOTP* dla użytkownika o podanym *userId*.
- **[GET] /api/AuthUsers/userId/totp**
Zwraca hasło jednorazowe typu *TOTP* dla użytkownika o podanym *userId*.
- **[GET] /api/AuthUsers/userId/hotp/hotp**
Zwraca wynik weryfikacji podanego hasła jednorazowego typu *HOTP*
- **[GET] /api/AuthUsers/userId/totp/totp**
Zwraca wynik weryfikacji podanego hasła jednorazowego typu *TOTP*

PicnicAuth.Api

AuthUsers

Show/Hide | List Operations | Expand Operations

GET /api/Companies/Me/AuthUsers Get AuthUsers logged company.

POST /api/AuthUsers Create new AuthUser and add it to logged company's collection.

AuthUsersSecrets

Show/Hide | List Operations | Expand Operations

Companies

Show/Hide | List Operations | Expand Operations

Hotps

Show/Hide | List Operations | Expand Operations

GET /api/AuthUsers/{userId}/hotp Get Hotp for given AuthUser.

Response Class (Status 200)
AuthUser's Hotp

Model Example Value

```
{
  "OtpValue": "string"
}
```

Response Content Type application/json ▾

Parameters

Parameter	Value	Description	Parameter Type	Data Type
userId	(required)	AuthUser's id in Guid format.	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Company is not logged in.		
404	AuthUser has not been found in logged company's collection.		

Try it out!

Rysunek 4.1. Interaktywna dokumentacja API.

- [POST] /api/tokens Zwraca klucz API, który służy do uwierzytelnienia podmiotu. (Równoznaczne z logowaniem, zgodnym ze standardem *OAuth*.)

Zasoby na których operuje API są w formacie JSON (JavaScript Object Notation), zarówno przy metodach, które przyjmują dane, jak i tych, które zwracają dane.

Za pomocą pakietu *Swashbuckle* skonfigurowana została automatyczna generacja dokumentacji API na podstawie publicznie wystawionych kontrolerów. Oprócz podstawowych informacji o metodach znajdujących się w kontrolerach, takich jak nazwy czy typy parametrów, w dokumentacji umieszczane są także komentarze, którymi opisana jest dana metoda. Oprócz funkcji informacyjnej, wygenerowana dokumentacja pozwala na wygodne testy każdej z metod. Końcowy efekt wygenerowanej dokumentacji przedstawiony jest na Rysunku 4.1.

Logika kryptograficzna aplikacji, którą wykorzystuje REST API, pokryta jest testami jednostkowymi w stu procentach. Aplikacja przeznaczona jest do działania na serwerze *IIS* (*Internet Information Service*) w wersji 8.5.



Rysunek 4.2. Informacyjna strona internetowa projektu.

4.1.2 Frontend

Strona internetowa projektu (Rysunek 4.2) stworzona została w technologii *Angular* w wersji 5 w konwencji *Single Page Application*. Znajduje się na niej instrukcja użycie projektu jak również informacje o aktualnej liczbie dostępnych bibliotek. Zawiera ona także przydatne odnośniki do repozytoriów, w których znajduje się kod źródłowy oraz do strony na której znajduje się dokumentacja API.

Dodatkowymi funkcjonalnościami jakie oferuje strona jest stworzenie nowego konta dla podmiotu (przedstawione na Rysunku 4.3), jak również uzyskanie klucza API, umożliwiającego użycie bibliotek klienckich.

4.1.3 Biblioteki klienckie

W celu ułatwienia integracji z serwerem zaimplementowane zostały biblioteki klienckie. Umożliwiają one w prosty sposób użycie funkcjonalności serwera bez konieczności studiowania dokumentacji i pisania kodu odpowiedzialnego za integrację.

Przykładowo biblioteka w języku C# udostępnia klasę *PicnicAuthClient*, posiadającą metody:

- Login
- GetAuthUsers
- AddAuthUser
- GenereteNewSecret
- GetLoggedCompany

The screenshot shows a 'Create Account' form. At the top, there's a navigation bar with links: Home, How-to, Create Account, Get API key, Project Information, and API Documentation. Below the navigation bar is a large dark area with the title 'Create Account' and the sub-instruction 'Create a new company account.'. There are four input fields: 'Login:' (with an input box), 'Email address:' (with an input box), 'Password:' (with an input box), and 'Confirm password:' (with an input box). At the bottom right of this dark area is a 'Register' button.

Rysunek 4.3. Tworzenie konta podmiotu.

- AddCompany
- GetHotpForAuthUser
- ValidateHotpForAuthUser
- GetTotpForAuthUser
- ValidateTotpForAuthUser

Na chwilę obecną gotowe do użycia są biblioteki w następujących technologiach:

1. C#
2. Visual Basic
3. TypeScript
4. Python 3.6
5. Python 2.7
6. Ruby

4.2 Tworzenie konta podmiotu

Podmiotem jest nazywana instytucja, posiadająca użytkowników i chcącą wykorzystać w swoim systemie uwierzytelnienie dwuetapowe.

Podmiot jest opisany bazodanowym modelem *CompanyAccount*, bazującym na klasie *IdentityUser* z przestrzeni *Microsoft.AspNet.Identity*. Oprócz standardowych pól odziedziczących z klasy *IdentityUser*, model ten posiada także kolekcję kolekcję użytkowników *AuthUser*.

Sam proces tworzenia nowego konta jak i autoryzacji jest zgodny ze standardowym zarządzaniem zwykłymi użytkownikami w projekcie typu *WebAPI* w technologii *.Net*.

W celu stworzenia nowego konta, metodą *HTTP POST*, na adres */api/Companies*, wysyłane są dane konta, takie jak adres poczty elektronicznej, nazwa konta oraz hasło. Format wysyłanych danych w postaci obiektu JSON jest przedstawiony poniżej:

```
{
    "Email": "string",
    "UserName": "string",
    "Password": "string",
    "ConfirmPassword": "string"
}
```

Proces logowania również jest standardowy i zgodny ze standardem *OAuth*. Na adres */api/tokens*, metodą *POST* wysyłane są dane takie jak nazwa konta oraz hasło a w odpowiedzi zwracany jest token typu *JWT* (JSON Web Token), wykorzystywany do autoryzacji podmiotu. Otrzymany token jest wysyłany przy każdym kolejnym zapytaniu w nagłówku w postaci *Authorization: Bearer {JWT}*.

4.3 Generacja sekretu użytkownika

Po utworzeniu konta podmiotu możliwe jest dodanie do kolekcji użytkowników nowego użytkownika. Wysyłane jest wtedy zapytanie typu *POST* na adres */api/AuthUsers*. Zasób pod tym adresem dostępny jest wyłącznie po zalogowaniu się kontem podmiotu.

Dane mają postać obiektu:

```
{
    "ExternalId": "string",
    "UserName": "string",
    "Email": "string"
}
```

Po przesłaniu żądania tworzone jest konto użytkownika a następnie zapisywane jest ono do bazy. W procesie tym ważnym etapem jest generacja sekretu użytkownika.

Do tego celu stworzona została klasa *SecureRandomNumberGenerator*, której kod widoczny jest na Rysunku 4.4. Wykorzystywana jest w niej klasa *RNGCryptoServiceProvider* z przestrzeni *System.Security.Cryptography*. Stworzona klasa wykorzystywana jest do wygenerowania kryptograficznie bezpiecznego, pseudolosowego ciągu bajtów, zwracanego w formie tablicy.

Bardziej wyspecjalizowaną klasą jest klasa *SecretGenerator*, przedstawiona na Rysunku 4.5. Jest w niej użyta funkcjonalność poprzedniej klasy. Zwraca ona wygenerowany sekret użytkownika, czyli ciąg bajtów o długości zgodnej z zaleceniami standardu, równej dziesięć.

4.4 Pobranie użytkowników powiązanych z podmiotem

Podmiot może w dowolnym momencie uzyskać kolekcję użytkowników, których stworzył. Aby tego dokonać należy wysłać zapytanie na adres */api/Companies/Me/AuthUsers*. Zapytanie musi być typu *GET* i może przyjmować parametry takie jak *page* oraz *pageCount*, wykorzystywane do paginacji. W przypadku niepodania któregoś z parametrów w zapytaniu

```

public class SecureRandomNumberGenerator : ISecureRandomNumberGenerator, IRequestDependency
{
    private const int MinimalNumberOfBytes = 1;
    private readonly RNGCryptoServiceProvider cryptoServiceProvider;

    public SecureRandomNumberGenerator()
    {
        cryptoServiceProvider = new RNGCryptoServiceProvider();
    }

    public byte[] GenerateRandomBytes(int numberofBytes)
    {
        if (numberofBytes < MinimalNumberOfBytes || numberofBytes > sbyte.MaxValue)
            throw new ArgumentOutOfRangeException();

        var buffer = new byte[numberofBytes];
        cryptoServiceProvider.GetBytes(buffer);
        return buffer;
    }

    public void Dispose()
    {
        cryptoServiceProvider?.Dispose();
    }
}

```

Rysunek 4.4. Klasa zwracająca kryptograficznie bezpiecznie pseudolosowe dane.

przyjmują one domyślne wartości. Dla parametru *page* domyślną wartością jest 1, a dla parametru *pageCount* jest to 10.

Przykład zapytania z parametrami przedstawia się następująco:

<https://picnicauth.gear.host/api/Companies/Me/AuthUsers?page=2&pageCount=15>

4.5 Dostarczanie sekretu na urządzenie mobilne

Po wysłaniu danych do stworzenia użytkownika w odpowiedzi wysyłany jest obiekt reprezentujący nowo stworzonego użytkownika:

```

{
    "ExternalId": "string",
    "UserName": "string",
    "Email": "string",
    "HotpQrCodeUri": "string",
    "TotpQrCodeUri": "string",
    "SecretInBase32": "string",
    "Id": "00000000-0000-0000-0000-000000000000"
}

```

W kwestii dostarczania sekretu użytkownika na jego urządzenie mobilne interesujące są tutaj pola zawierające adresy do kodów QR (Quick Response) oraz pole zawierające sekret zakodowany używając kodowania transportowego *Base32*. Istnieją więc dwie metody dostarczenia sekretu.

```

public class SecretGenerator : ISecretGenerator, IRequestDependency
{
    private const int PreferredNumberOfBytes = 10;
    private readonly ISecureRandomNumberGenerator randomNumberGenerator;

    public SecretGenerator(ISecureRandomNumberGenerator randomNumberGenerator)
    {
        this.randomNumberGenerator = randomNumberGenerator;
    }

    public byte[] GenerateSecret()
    {
        byte[] randomBytes = randomNumberGenerator.GenerateRandomBytes(PreferredNumberOfBytes);
        return randomBytes;
    }
}

```

Rysunek 4.5. Klasa generująca sekret użytkownika.

Pierwsza z nich polega na przepisaniu sekretu do urządzenia mobilnego, nadaniu nazwy konta oraz wybraniu typu hasła jednorazowe. Ekran używany do wprowadzania tych danych przedstawiony jest na Rysunku 4.6.

Jeśli urządzenie mobilne użytkownika jest wyposażone w aparat fotograficzny możliwe jest natychmiastowe dostarczenie sekretu, poprzez zeskanowanie jednego z kodów QR. W kodach tych zakodowane są dane w formacie *Key Uri*. Format ten jest zgodny ze standardem *URI*.

Można go przedstawić w następujący sposób:

otpauth://{{TYP}}/{{ETYKIETA}}?{{PARAMETRY}}

Typ może przybierać tutaj wartości *hotp* lub *totp*, w zależności jaki rodzaj hasła jednorazowego chcemy stosować.

Etykieta używana jest do identyfikacji konta, dla którego generowane są hasła jednorazowe. Zwykle składa się z nazwy konta poprzedzonej nazwą podmiotu. Przykładową etykietą może być tutaj:

Podmiot1:Jan.Kowalski

Jednym z parametrów jest parametr *secret*, w którym musi znajdować się sekret użytkownika zakodowany kodowaniem Base32.

Niewymaganym parametrem jest tutaj parametr *issuer*. Jest równoznaczny z prefiksem etykiety, będącym nazwą podmiotu. Jeśli jest on obecny powinien przyjmować taką samą wartość jak wcześniej wspomniany prefiks.

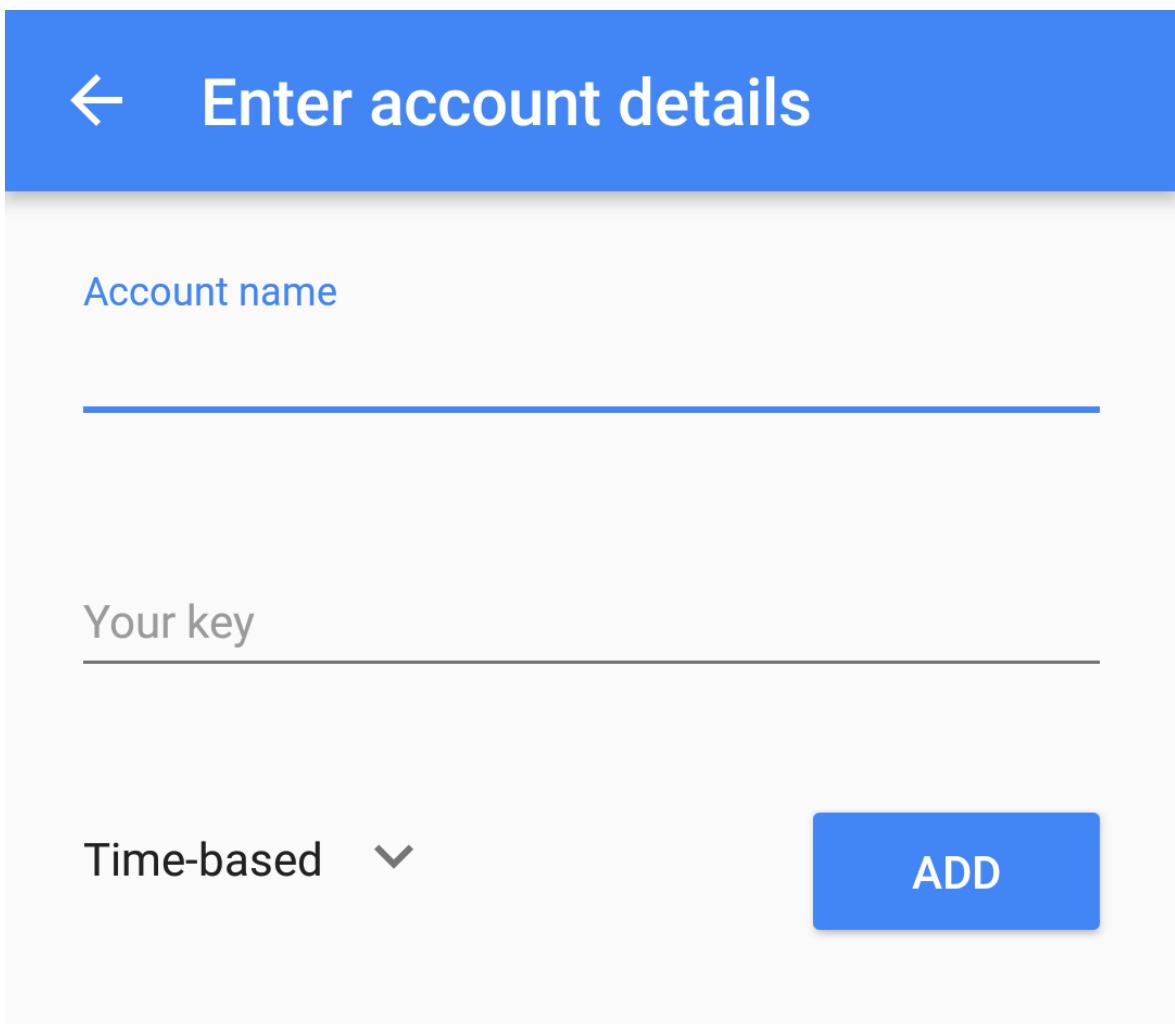
Pozostałe parametry są ignorowane przez najnowszą implementację aplikacji mobilnej *Google Authenticator*. Ignorowanymi parametrami są:

- *Algorithm* determinujący jaka funkcja skrótu będzie używana przy tworzeniu haseł jednorazowych.

Może przyjmować wartości:

- SHA1 (Domyślna wartość).
- SHA256.
- SHA512.

- *Digits* determinujący długość hasła jednorazowego. Może przyjmować wartości 6 lub 8.



Rysunek 4.6. Ekran tworzenia nowego konta w aplikacji Google Authenticator.

Hasło jednorazowe będzie wtedy skracane do sześciu lub ośmiu cyfr.

- *Counter* ustalający początkową wartość licznika w przypadku haseł opartych o licznik.
- *Period* określający okres ważności haseł opartych o czas. Domyślną wartością jest tutaj 30 sekund.

Klasa stworzona do generacji łańcucha znaków w formacie *Key Uri* przedstawiona jest na Rysunku 4.7. Wykorzystuje ona formatowanie łańcuchów znaków na podstawie z góry zadanego szablonu.

Przykładowe kody QR o typach HOTP oraz TOTP znajdują się odpowiednio na Rysunku 4.8 oraz Rysunku 4.9. Niezwykle ważne jest tutaj bezpieczeństwo samego kanału przekazywania sekretu do użytkownika końcowego. Konieczne jest aby cała komunikacja odbywała się z wykorzystaniem protokołu HTTPS. Projekt oczywiście wspiera użycie protokołu HTTPS, jak również zapobiega używaniu niezabezpieczonego kanału poprzez zablokowanie możliwości logowania w przypadku nieszyfrowanego połączenia.

```

public class KeyUriCreator : IKeyUriCreator, IRequestDependency
{
    private const string KeyUriTemplate = "otpauth:///{0}/{1}:{2}?secret={3}&issuer={1}";
    public string CreateKeyUri(string issuer, string user, string secret, OtpType otpType = OtpType.Totp)
    {
        if (AnyArgumentIsNull(issuer, user, secret))
            throw new ArgumentNullException();
        if (AnyArgumentIsEmpty(issuer, user, secret))
            throw new ArgumentException();

        return string.Format(KeyUriTemplate, otpType.ToString().ToLower(), issuer, user, secret);
    }

    private bool AnyArgumentIsNull(string issuer, string user, string secret)
    {
        return issuer == null || user == null || secret == null;
    }

    private bool AnyArgumentIsEmpty(string issuer, string user, string secret)
    {
        return string.IsNullOrEmpty(issuer) ||
               string.IsNullOrEmpty(user) ||
               string.IsNullOrEmpty(secret);
    }
}

```

Rysunek 4.7. Klasa generująca KeyUri.

4.6 Generowanie OTP po stronie serwera

Mając już wspólną wartość sekretu po obu stronach można przejść do procesu generacji hasła jednorazowego. Niezależnie od wybranego typu hasła jednorazowego, wykorzystywana jest funkcja HMAC, której jednym z argumentów jest sekret użytkownika. Klasa zaimplementowana w projekcie, wykorzystywana do uzyskania wyniku funkcji HMAC, przy użyciu funkcji skrótu SHA1 przedstawiona jest na Rysunku 4.10.

W implementacji tej argument *key* jest równoznaczny z sekretem użytkownika.

Wynik funkcji HMAC jest następnie przycinany do postaci hasła jednorazowego. Hasło może zostać przycięte do sześciu lub ośmiu cyfr w zależności od preferencji podmiotu. Sam proces przycięcia opisany został w sekcji 3.2.1.

Zaimplementowana funkcja przycinająca przedstawiona jest na Rysunku 4.11. Funkcja ta pobiera wcześniej otrzymany wynik funkcji HMAC oraz liczbę cyfr, do jakiej zostanie przycięte hasło. Domyślną liczbą cyfr, która zostanie wybrana jeśli ten argument nie zostanie podany funkcji, jest sześć.

Jako że prawidłową długością długością wyniku funkcji HMAC jest 20 bajtów, funkcja przycinająca sprawdza długość przekazanego argumentu i rzuca wyjątek w przypadku, gdy długość danych jest różna od 20 bajtów.

4.6.1 Generowanie haseł typu HOTP

Do generacji haseł typu HOTP stworzona została klasa *HotpGenerator*, przedstawiona na Rysunku 4.12. Funkcja *GenerateHotp* w niej zawarta na podstawie licznika użytkownika oraz jego sekretu generuje hasło jednorazowe. Funkcjonalność tej klasy bazuje na dwóch



Rysunek 4.8. Kod QR umożliwiający generację haseł typu HOTP.

poprzednio omawianych klasach, odpowiedzialnych za wyliczenie wyniku funkcji HMAC oraz przycięcie wyniku funkcji HMAC do postaci hasła.

4.6.2 Generowanie haseł typu TOTP

W przypadku klasy *TotpGenerator*, przedstawionej na Rysunku 4.13, zamiast licznika, do funkcji HMAC przekazywana jest wartość aktualnego czasu uniksowego, zwrócona przez funkcję *GetUnixTimestamp*. Przed podaniem zmiennej *currentTimestamp* do funkcji HMAC, jest ona uprzednio dzielona przez stałą reprezentującą okres ważności hasła jednorazowego, przyjmującą wartość zgodną ze standardem, równą trzydzieści sekund.

4.7 Generowanie OTP po stronie użytkownika

Generacja haseł jednorazowych po stronie użytkownika odbywa się przy użyciu aplikacji mobilnych. Na tym etapie zakładamy, że sekret został już w bezpieczny sposób dostarczony na urządzenie mobilne.

Jedną z wielu aplikacji, których można użyć jest *Google Authenticator*. Na Rysunku 4.14 przedstawiony jest zrzut ekranu z tej aplikacji. Widnieją na nim trzy konta użytkownika, wszystkie powiązane z podmiotem *sggw.pl*. Pierwsze od góry konto jest skonfigurowane do generacji haseł typu HOTP, opartych o licznik, pozostałe dwa widoczne hasła, są hasłami typu TOTP, opartymi o czas uniksowy. W celu wygenerowania kolejnego hasła typu HOTP konieczne jest na ikonę *zwiniętej strzałki*. Po kliknięciu zostanie zwiększyły licznik dla tego konta a na ekranie pojawi się nowe hasło jednorazowe. Hasła typu TOTP odświeżane są automatycznie wraz z upływem ich ważności. Ikona w kształcie koła, znajdująca się na prawo od hasła symbolizuje pozostały czas ważności hasła.

Bezpieczniejszą alternatywą do aplikacji *Google Authenticator* jest *Authy 2-Factor Authentication*. Posiada ona możliwość stworzenia szyfrowanych kopii zapasowych, użytkowych



Rysunek 4.9. Kod QR umożliwiający generację haseł typu TOTP.

w przypadku zgubienia telefonu oraz wyposażona jest w dodatkową warstwę ochrony w postaci kodu PIN, wymaganego do uzyskania haseł jednorazowych.

Jedną z cech podnoszących bezpieczeństwo aplikacji, odkrytych podczas testów jest zablokowana możliwość robienia zrzutów ekranów, jak również nagrywania ekranu z wyświetlonym hasłem jednorazowym.

Ekran aplikacji przedstawiony jest na Rysunku 4.15. Widoczne jest na nim wygenerowane hasło typu TOTP dla użytkownika *Przykładowy użytkownik*.

4.8 Przechowywanie sekretu użytkownika

Sekrety użytkowników przechowywane są w bazie danych w postaci zaszyfrowanej. Do konwersji sekretu użytkownika do postaci zaszyfrowanej wykorzystywany jest interfejs *Windows Data Protection*, scharakteryzowany w sekcji 3.3.

Klasą odpowiedzialną za szyfrowanie sekretów użytkownika jest klasa *DpapiEncryptor*, której kod źródłowy przedstawiony jest na Rysunku 4.16. Posiada ona dwie metody *Encrypt* pobierające odpowiednio tablicę bajtów oraz ciąg znaków, których można używać naprzemienne w zależności od typu danych jaki chcemy zaszyfrować. Sam proces szyfrowania odbywa się *w miejscu*, oznacza to że funkcja pobiera dane do zaszyfrowania i bezpośrednio zwraca dane w formie zaszyfrowanej, bez zapisu ich w jakimkolwiek miejscu w pamięci.

Komplementarną do powyższej klasy jest klasa *DpapiDecryptor*, przedstawiona na Rysunku 4.17. Umożliwia ona deszyfrowanie danych, zaszyfrowanych interfejsem *Windows Data Protection* do postaci łańcucha znaków lub do postaci tablicy bajtów.

```

public class HmacSha1Generator : IHmacSha1Generator, IRequestDependency
{
    private readonly IULongConverter uLongConverter;
    private readonly IUtf8Converter utf8Converter;

    public HmacSha1Generator(IULongConverter uLongConverter, IUtf8Converter utf8Converter)
    {
        this.uLongConverter = uLongConverter;
        this.utf8Converter = utf8Converter;
    }

    public byte[] GenerateHmacSha1Hash(byte[] input, byte[] key)
    {
        CheckArguments(input, key);
        var hmacsha1 = new HMACSHA1(key);

        return hmacsha1.ComputeHash(input);
    }
}

```

Rysunek 4.10. Klasa generująca HMAC.

4.9 Walidacja hasła jednorazowego

4.10 Zmiana sekretu użytkownika

4.11 Przykład użycia projektu

4.12 Planowane ulepszenia

4.12.1 Generowanie hasła po stronie użytkownika

Za najbezpieczniejszą metodę generacji haseł jednorazowych po stronie użytkownika uważany jest token sprzętowy, zabezpieczony kodem PIN. Jest to dosyć kosztowna metoda ze względu na koszty produkcji takiego tokenu, jak również jest mało uniwersalna, gdyż zwykle token powiązany jest z pojedynczym kontem.

Wychodząc na przeciw tym ograniczeniom, mogą tutaj zostać zaproponowane rozwiązania oparte o mikrokontroler *Arduino Uno* lub o mini-komputer *Raspberry Pi Zero*. Projekt ten mógłby mieć formę DIY (Do it yourself) czyli być przepisem jak z dostarczonego oprogramowania i części stworzyć własny token sprzętowy, mogący generować hasła jednorazowe do wielu serwisów. Token posiadałby ekran, na którym wyświetlane były wygenerowane hasło jedno razowe oraz klawiaturę alfanumeryczną, wykorzystywaną do podawania kodu PIN oraz do dodania nowego konta do bazy tokenu.

4.12.2 Biblioteki klienckie

Użycie REST API jest możliwe w każdej technologii umożliwiającej wykorzystanie protokołu HTTP wraz z formatem JSON. Dużo wygodniejsze jest jednak korzystanie z gotowych

```

public class OtpTruncator : IOtpTruncator, IRequestDependency
{
    private const int ValidSha1SignatureLength = 20;
    private const int TruncateMask = 0b1111;
    private const int OtpBase = 10;

    private readonly IArraySlicer arraySlicer;
    private readonly IUIntConverter uIntConverter;

    public OtpTruncator(IArraySlicer arraySlicer, IUIntConverter uIntConverter)
    {
        this.arraySlicer = arraySlicer;
        this.uIntConverter = uIntConverter;
    }

    public string Truncate(byte[] hmacSignature, OtpLength otpLength = OtpLength.SixDigits)
    {
        if (hmacSignature == null) throw new ArgumentNullException();
        if (hmacSignature.Length != ValidSha1SignatureLength)
            throw new ArgumentException(Properties.Resources.InvalidSha1LengthExceptionMessage);

        var otpDigits = (int)otpLength;

        byte lastByte = hmacSignature.Last();
        int randomStartIndex = lastByte & TruncateMask;
        byte[] resultBytes = arraySlicer.Slice(hmacSignature, randomStartIndex, randomStartIndex + sizeof(int));
        uint resultInt = uIntConverter.ConvertToIntBigEndian(resultBytes);

        resultInt &= int.MaxValue;
        resultInt %= (uint) Math.Pow(OtpBase, otpDigits);

        return resultInt.ToString().PadLeft(otpDigits, '0');
    }
}

```

Rysunek 4.11. Klasa odpowiedzialna za przycięcie wyniku HMAC do postaci hasła.

bibliotek, stworzonych pod daną technologię.

W tym celu, w kolejnych wersjach projektu, planowana jest implementacja bibliotek klienckich w następujących językach programowania:

- C
- C++
- Clojure
- D
- Elixir
- Erlang
- Groovy
- Java
- JavaScript
- Kotlin
- Nim
- Objective-C
- Perl
- PHP

```

public class HotpGenerator : IHotpGenerator, IRequestDependency
{
    private readonly IHmacSha1Generator hmacSha1Generator;
    private readonly IOtpTruncator otpTruncator;

    public HotpGenerator(IHmacSha1Generator hmacSha1Generator, IOtpTruncator otpTruncator)
    {
        this.hmacSha1Generator = hmacSha1Generator;
        this.otpTruncator = otpTruncator;
    }

    public string GenerateHotp(long counter, byte[] secret)
    {
        if (secret == null)
            throw new ArgumentNullException();
        if (secret.Length == byte.MinValue)
            throw new ArgumentException();

        byte[] hmac = hmacSha1Generator.GenerateHmacSha1Hash(counter, secret);
        string otp = otpTruncator.Truncate(hmac);
        return otp;
    }
}

```

Rysunek 4.12. Klasa generująca hasło jednorazowe oparte o licznik.

- Scala
- Swift

4.12.3 Architektura

Możliwe jest także ulepszenie architektury części REST API projektu pod kątem większego odseparowania od siebie komponentów. Szczególnie dotyczy to odseparowania części przechowującej sekrety użytkowników i walidującej hasła jednorazowe od reszty aplikacji. W idealnym przypadku odseparowana część umieszczona by była na osobnym systemie komputerowym, bez dostępu do Internetu. Komunikacja z główną częścią aplikacji powinna odbywać się wtedy poprzez minimalny interfejs, pozwalający wyłącznie na uwierzytelnienie podmiotu oraz walidację podanego przez użytkownika hasła jednorazowego. Po przesłaniu podanego przez użytkownika hasła jednorazowego, zwrócony zostałby jedynie wynik walidacji w postaci *true* lub *false*.

4.12.4 Konteneryzacja części serwerowej

Poza podstawowym przypadkiem użycia opartym o REST API wystawione pod adresem <https://picnicauth.gear.host/swagger/ui/index>, nic nie stoi na przeszkodzie wykorzystaniu kodu źródłowego aplikacji i stworzenia REST API na własnym serwerze, na przykład w firmowej sieci lokalnej. Ten sposób wykorzystania projektu może okazać się bardziej bezpieczny, jeśli jest przez dany podmiot umiejętnie administrowany oraz zupełnie niebezpieczny w przeciwnym przypadku.

Aby ułatwić procedurę wdrożenia REST API na serwer można zaproponować tutaj użycie

```

public class TotpGenerator : ITotpGenerator, IRequestDependency
{
    private const int TotpTimeWindow = 30;

    private readonly IUnixTimestampGetter unixTimestampGetter;
    private readonly IHmacSha1Generator hmacSha1Generator;
    private readonly IOtpTruncator otpTruncator;

    public TotpGenerator(IUnixTimestampGetter unixTimestampGetter, IHmacSha1Generator hmacSha1Generator,
        IOtpTruncator otpTruncator)
    {
        this.unixTimestampGetter = unixTimestampGetter;
        this.hmacSha1Generator = hmacSha1Generator;
        this.otpTruncator = otpTruncator;
    }

    public string GenerateTotp(byte[] secret)
    {
        if (secret == null)
            throw new ArgumentNullException();
        if (secret.Length == byte.MinValue)
            throw new ArgumentException();

        long currentTimestamp = unixTimestampGetter.GetUnixTimestamp();
        byte[] hmac = hmacSha1Generator.GenerateHmacSha1Hash(currentTimestamp / TotpTimeWindow, secret);
        string otp = otpTruncator.Truncate(hmac);

        return otp;
    }
}

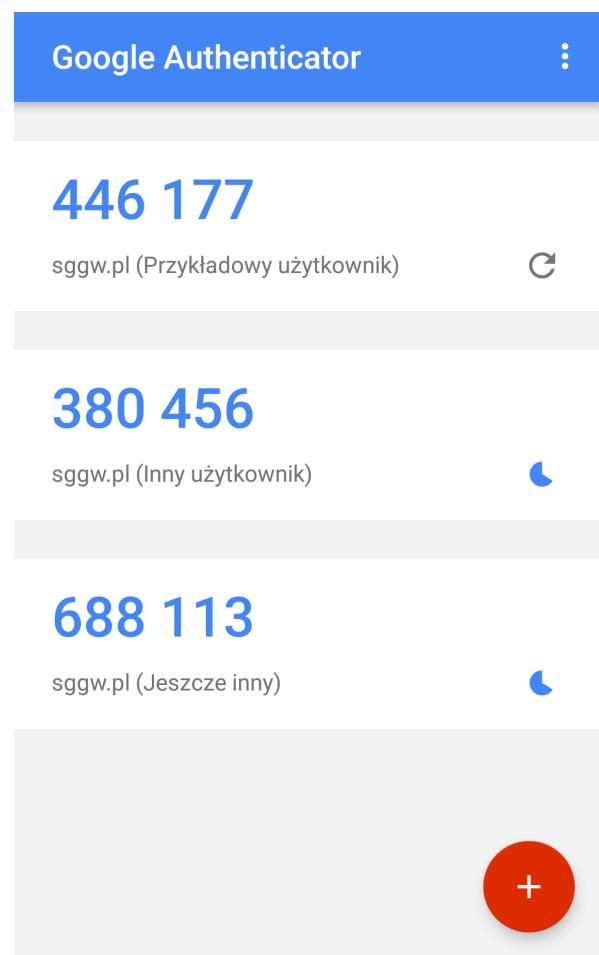
```

Rysunek 4.13. Klasa generująca hasło jednorazowe oparte o czas.

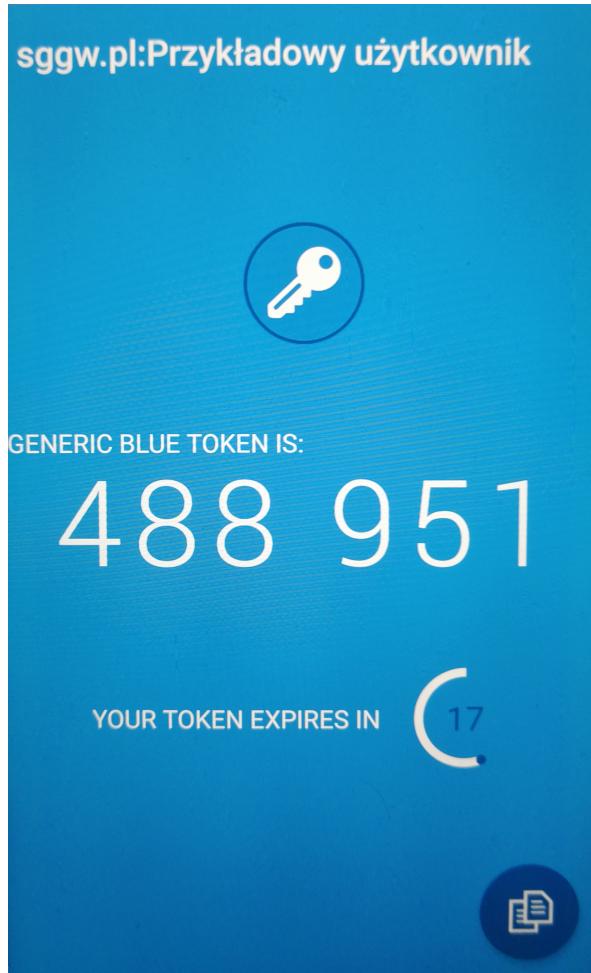
kontenerów. Dzięki nim aplikacja nie jest zależna od systemu, który jest zainstalowany na serwerze a sam proces wdrożenia jej jest niezwykle prosty i szybki. Może zostać użyte oprogramowanie *Docker*.

4.12.5 Przechowywanie sekretu

Obecnie do szyfrowania sekretów użytkowników wykorzystywany jest interfejs *Windows Data Protection*. Posiada on kilka wad, między innymi to, że jest silnie zależny od hasła użytkownika systemu. Lepszym ale i bardziej kosztownym rozwiązaniem, byłoby użycie szyfrowania sprzętowego zamiast wspomnianego interfejsu.



Rysunek 4.14. Użycie aplikacji Google Authenticator.



Rysunek 4.15. Użycie aplikacji Authy.

```
public class DpapiEncryptor : IDpapiEncryptor, IRequestDependency
{
    public byte[] Encrypt(byte[] input, byte[] optionalEntropy = null,
        DataProtectionScope scope = DataProtectionScope.CurrentUser)
    {
        byte[] protectedBytes = ProtectedData.Protect(input, optionalEntropy, scope);

        return protectedBytes;
    }

    public byte[] Encrypt(string input, byte[] optionalEntropy = null,
        DataProtectionScope scope = DataProtectionScope.CurrentUser)
    {
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(input);

        return Encrypt(bytes, optionalEntropy, scope);
    }
}
```

Rysunek 4.16. Klasa odpowiedzialna za szyfrowanie.

```

public class DpapiDecryptor : IDpapiDecryptor, IRequestDependency
{
    public string DecryptToString(byte[] input, byte[] optionalEntropy = null,
        DataProtectionScope scope = DataProtectionScope.CurrentUser)
    {
        byte[] decryptedBytes = DecryptToBytes(input, optionalEntropy, scope);
        return System.Text.Encoding.UTF8.GetString(decryptedBytes);
    }

    public byte[] DecryptToBytes(byte[] input, byte[] optionalEntropy = null,
        DataProtectionScope scope = DataProtectionScope.CurrentUser)
    {
        byte[] decryptedBytes = ProtectedData.Unprotect(input, optionalEntropy, scope);

        return decryptedBytes;
    }
}

```

Rysunek 4.17. Klasa odpowiedzialna za deszyfrowanie.

```

public class TotpValidator : ITotpValidator, IRequestDependency
{
    private const int ValidTimeStepSize = 1;
    private const int TotpTimeWindow = 30;

    private readonly IUnixTimestampGetter unixTimestampGetter;
    private readonly IHotpGenerator hotpGenerator;

    public TotpValidator(IUnixTimestampGetter unixTimestampGetter, IHotpGenerator hotpGenerator)
    {
        this.unixTimestampGetter = unixTimestampGetter;
        this.hotpGenerator = hotpGenerator;
    }

    public bool IsTotpValid(byte[] secret, string totp)
    {
        if (secret == null || totp == null)
            throw new ArgumentNullException();
        if (secret.Length == byte.MinValue || string.IsNullOrWhiteSpace(totp))
            throw new ArgumentException();

        long currentTimestampInput = unixTimestampGetter.GetUnixTimestamp() / TotpTimeWindow;
        IEnumerable<long> consideredTimestampInputs = GetConsideredTimestampInputs(currentTimestampInput);

        IEnumerable<string> totps = consideredTimestampInputs
            .Select(input => hotpGenerator.GenerateHotp(input, secret));

        return totps.Contains(totp);
    }

    private IEnumerable<long> GetConsideredTimestampInputs(long currentTimestampInput)
    {
        return new[]
        {
            currentTimestampInput,
            currentTimestampInput - ValidTimeStepSize,
            currentTimestampInput + ValidTimeStepSize
        };
    }
}

```

Rysunek 4.18. Klasa odpowiedzialna za walidację hasła typu TOTP.

5 Zakończenie

5.1 Podsumowanie i wnioski

5.2 Podziękowania

Spis rysunków

2.1	Wizualizacja trybów szyfrowania.	12
2.2	Schemat szyfrowania w trybie CBC.	13
2.3	Schemat deszyfrowania w trybie CBC.	13
2.4	Schemat szyfrowania w trybie CTR.	14
2.5	Schemat deszyfrowania w trybie CTR.	14
2.6	Przekształcenie f	16
2.7	Macierz odwzorowania funkcji zamiany.	17
2.8	Wizualizacja zamiany bajtów.	18
2.9	Wizualizacja przesunięcia wierszy.	18
2.10	Wizualizacja mieszania kolumn.	19
2.11	Wizualizacja dodania klucza przebiegu.	20
2.12	Schemat HMAC.	27
4.1	Interaktywna dokumentacja API.	36
4.2	Informacyjna strona internetowa projektu.	37
4.3	Tworzenie konta podmiotu.	38
4.4	Klasa zwracająca kryptograficznie bezpiecznie pseudolosowe dane.	40
4.5	Klasa generująca sekret użytkownika.	41
4.6	Ekran tworzenia nowego konta w aplikacji Google Authenticator.	42
4.7	Klasa generująca KeyUri.	43
4.8	Kod QR umożliwiający generację haseł typu HOTP.	44
4.9	Kod QR umożliwiający generację haseł typu TOTP.	45
4.10	Klasa generująca HMAC.	46
4.11	Klasa odpowiedzialna za przyjęcie wyniku HMAC do postaci hasła.	47
4.12	Klasa generująca hasło jednorazowe oparte o licznik.	48
4.13	Klasa generująca hasło jednorazowe oparte o czas.	49
4.14	Użycie aplikacji Google Authenticator.	50
4.15	Użycie aplikacji Authy.	51
4.16	Klasa odpowiedzialna za szyfrowanie.	51
4.17	Klasa odpowiedzialna za deszyfrowanie.	52
4.18	Klasa odpowiedzialna za walidację hasła typu TOTP.	52

6 Bibliografia

- [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.pdf>, 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. *Cryptography ePrint Archive*, Report 2005/067, 2005.
- [5] R. Rivest, *The MD5 Message-Digest Algorithm*
<https://tools.ietf.org/pdf/rfc1321.pdf>, MIT, 1992
- [6] N. Haller, C. Metz, P. Nesser, M. Straw, *A One-Time Password System*
<https://tools.ietf.org/pdf/rfc2289.pdf>, MIT, 1998
- [7] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen, *HOTP: An HMAC-Based One-Time Password Algorithm*
<https://tools.ietf.org/pdf/rfc4226.pdf>, 2005
- [8] D. M'Raihi, S. Machani, M. Pei, J. Rydell, *TOTP: Time-Based One-Time Password Algorithm*
<https://tools.ietf.org/pdf/rfc6238.pdf>, 2011
- [9] Passscape Software, *DPAPI Secrets. Security analysis and data recovery in DPAPI*
<https://www.passscape.com/index.php?section=docs&cmd=details&id=28#14>
- [10] Thomas Ptacek *If You're Typing the Letters A-E-S Into Your Code You're Doing It Wrong*
<https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2009/july/if-youre-typing-the-letters-a-e-s-into-your-code-youre-doing-it-wrong/>, 2009
- [11] Ken Shirriff, *Mining Bitcoin with pencil and paper: 0.67 hashes per day*
<http://www.righto.com/2014/09/mining-bitcoin-with-pencil-and-paper.html>, 2014
- [12] D. Eastlake, 3rd, P. Jones *US Secure Hash Algorithm 1 (SHA1)*
<https://tools.ietf.org/pdf/rfc3174.pdf>, 2001

- [13] Laurens Van Houtven (lvh), *Crypto 101*
<https://www.crypto101.io>, 2017
- [14] Keccak Team, *Strengths of Keccak - Design and security*
https://keccak.team/keccak_strengths.html, 2017
- [15] Niels Ferguson, Bruce Schneier, Todayoshi Kohno *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010
- [16] Marc Stevens, Pierre Karpman, Thomas Peyrin, *The SHAppening: freestart collisions for SHA-1*.
<https://sites.google.com/site/itstheshappening>, 2015
- [17] 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
- [18] Thai Duong, Juliano Rizzo, *Here Come The \oplus Ninjas*.
<https://bug665814.bmoattachments.org/attachment.cgi?id=540839>, 2011
- [19] SciEngines GmbH *Break DES in less than a single day*.
<https://www.voltage.com/technology/rivyera-from-sciengines/>, 2008
- [20] *DES-III contest*.
<http://www.distributed.net/DES>, 1999
- [21] Thomas Ptacek, *If You're Typing the Letters A-E-S Into Your Code You're Doing It Wrong*.
<https://people.eecs.berkeley.edu/~daw/teaching/cs261-f12/misc/if.html>, 2009
- [22] Serge Vaudenay, *Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, TLS....*
<https://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf>, 2002
- [23] Mihir Bellare, *New proofs for NMAC and HMAC: Security without collision-resistance*.
<http://cseweb.ucsd.edu/~mihir/papers/hmac-new.html>, 2006

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

.....
(czytelny podpis autora pracy)

