Biuletyn WAT
Vol. LXVII, Nr 1, 2018

Zabezpieczanie haseł w systemach informatycznych

PRZEMYSŁAW RODWALD[1], BARTOSZ BIERNACIK[2]

[1]Akademia Marynarki Wojennej, Wydział Nawigacji i Uzbrojenia Okrętowego,
Instytut Uzbrojenia Okrętowego i Informatyki, ul. Śmidowicza 69, 81-103 Gdynia,
p.rodwald@amw.gdynia.pl,
[2]Akademia Sztuki Wojennej, Wydział Wojskowy, Instytut Działań Informacyjnych,
al. gen. A. Chruściela 103, 00-910 Warszawa, b.biernacik@akademia.mil.pl

Streszczenie. Celem artykułu jest usystematyzowanie metod zabezpieczania statycznych haseł przechowywanych na potrzeby systemów informatycznych, w szczególności serwisów internetowych, wskazanie słabych stron zaprezentowanych metod oraz wyciągnięcie wniosków w postaci zaleceń dla projektantów systemów informatycznych. Na wstępie przedstawiono pojęcie kryptograficznej funkcji skrótu, a następnie omówiono kolejne metody przechowywania haseł, pokazując ich ewolucję oraz podatności na współczesne ataki. Pokazano wyniki badań nad hasłami maskowanymi w polskich bankach oraz przedstawiono najciekawsze przykłady współczesnych funkcji adaptacyjnych. Następnie dokonano autorskiej systematyzacji metod zabezpieczania haseł oraz wskazano kierunki dalszych badań.
Słowa kluczowe: informatyka, hasła, uwierzytelnianie, zabezpieczanie danych, funkcje skrótu
DOI: 10.5604/01.3001.0011.8036

1. Wstęp

Nasze podejście do bezpieczeństwa informatycznego ciągle ewoluuje. Jeszcze kilka lat temu, gdy dane przechowywaliśmy przeważnie na lokalnych komputerach, broniliśmy się głównie przed wszelkiego rodzaju szkodliwym oprogramowaniem (wirusy, robaki itp.). Dzisiaj ciężar przenosi się na przetwarzanie danych w chmurach obliczeniowych (ang. *cloud computing*), za pomocą których przechowujemy dane (Dropbox, OneDrive, Google Drive) czy wykonujemy codzienne czynności (Office 365). Nasze dane, znajdujące się we wspomnianych chmurach, najczęściej

chronione są tylko za pomocą hasła, i to właśnie bezpieczeństwo haseł zaczyna mieć priorytetowe znaczenie.

Projektanci systemów informatycznych, w tym serwisów internetowych, w których wymagany jest uwierzytelniony dostęp do określonych zasobów, stoją przed problemem przechowania haseł użytkowników. Hasła jako mechanizm uwierzytelnienia zostały użyte po raz pierwszy w roku 1960 i do dziś cieszą się ogromną popularnością [1]. Hasła powinny być przechowywane w taki sposób, aby w przypadku włamania się do systemu przez intruza i uzyskania nieautoryzowanego dostępu do bazy danych, w której hasła są najczęściej przechowywane, ich odtworzenie było obliczeniowo trudne[1]. Przed omówieniem poszczególnych metod zabezpieczania haseł pokazane zostaną przykłady wycieków haseł oraz wyjaśnione zostanie pojęcie kryptograficznej funkcji skrótu, jako techniki najczęściej wykorzystywanej do przechowywania haseł.

2. Bezpieczeństwo haseł

Można zadać sobie pytanie, w jakim celu w ogóle zabezpieczać same hasła, skoro najczęściej stosowane są mechanizmy zabezpieczające przed nieautoryzowanym dostępem do miejsc (na przykład baz danych), w których hasła są przechowywane. Otóż zawsze istnieje ryzyko włamania do systemu i wycieku bazy zawierającej hasła użytkowników. Włamania takie, także do systemów dużych firm, nie należą w dzisiejszych czasach do rzadkości. Przykładowe, najbardziej spektakularne, informacje o włamaniach z ostatnich lat:

- Uber ujawnił informacje, że zapłacił okup atakującym, którzy w 2016 roku wykradli dane 57 milionów użytkowników z dnia 22.11.2017 [2];
- ujawnienie użytkowników serwisu Ashley Madison z dnia 11.07.2015 [3];
- wyciek danych z Sony z dnia 25.11.2014 [4];
- nieautoryzowany dostęp do baz danych PKW z dnia 18.11.2014 [5];
- wykradziono hasła i dokumenty z GPW z dnia 23.10.2014 [6];
- wyciek 7 milionów haseł do serwisu Dropbox z dnia 14.10.2014 [7];
- opublikowanie 450 000 haseł użytkowników Yahoo z dnia 13.07.2012 [8].

Niebezpieczeństwa, jakie może nieść wyciek haseł do danego systemu, to:

- uzyskanie nieautoryzowanego dostępu do danego systemu i wynikające z tego konsekwencje (na przykład możliwość dokonania nieautoryzowanej transakcji internetowej);

[1] Atak obliczeniowo trudny rozumiany jest tutaj jako atak, którego złożoność obliczeniowa w praktyce jest tak duża (teoretycznie: wykładnicza), że przy współcześnie istniejącej technice oraz stanie wiedzy staje się on praktycznie niewykonalny w rozsądnym czasie.

— uzyskanie nieautoryzowanego dostępu do innego systemu i wszelkie konsekwencje z tego płynące — wynika to z faktu, że wielu użytkowników używa tego samego loginu i hasła do wielu systemów.

W niniejszym artykule skupiono uwagę tylko na systemach, w których uwierzytelnienie następuje za pomocą podania statycznego hasła. Nie rozważa się innych mechanizmów uwierzytelniających wykorzystujących na przykład: hasła jednorazowe, karty magnetyczne czy techniki biometryczne.

3. Kryptograficzne funkcje skrótu

Pod pojęciem kryptograficznej funkcji skrótu *h* rozumie się odwzorowanie skończonego ciągu znaków (np. hasła) *pass* w ciąg bitów o określonej stałej długości *n*:

$$h\colon \{0,1\}^* \rightarrow \{0,1\}^n, \quad \text{gdzie: } 0,1\}^* = \bigcup_{i\in \mathrm{I}} \{0,1\}^i, \mathrm{N} = \{0,1,2,\ldots\}, n \in \mathrm{N}. \qquad (1)$$

Podstawowe wymagania stawiane funkcjom skrótu można przedstawić następująco:

— **Nieodwracalność** (ang. *non-invertibility*, *preimage resistance*): dany jest skrót *h*(*pass*), hasło *pass* jest nieznane. Znalezienie hasła *pass* jest obliczeniowo trudne.

— **Słaba bezkolizyjność** (ang. *weak collision resistance*, *2nd preimage resistance*): dany jest skrót *h*(*pass*) i odpowiadające mu hasło *pass*. Znalezienie innego hasła *pass*'≠*pass*, takiego że *h*(*pass*) = *h*(*pass*'), jest obliczeniowo trudne.

— **Silna bezkolizyjność** (ang. *strong collision resistance*, *collision resistance*): obliczeniowo trudne jest znalezienie dowolnej pary różnych haseł *pass*' i *pass*, takich że *h*(*pass*) = *h*(*pass*').

Aktualnie do powszechnie używanych do przechowywania haseł funkcji skrótu ciągle należą funkcje MD5, SHA-1, mimo że ich używanie nie jest już zalecane przez NIST[2] ze względu na istniejące skuteczne praktyczne ataki przeciwko nim [9, 10]. O ile w kryptografii pożądana właściwość funkcji skrótu, jaką jest ich szybkość (w szczególności w porównaniu z szyfrowaniem asymetrycznym), jest niemal niezbędna, o tyle przy przechowywaniu wartości funkcji skrótu jako metody przechowywania haseł wspomniana zaleta staje się jej największą wadą. Dzieje się tak dlatego, gdyż atakujący może bardzo efektywnie dokonywać prób łamania haseł poprzez wykorzystanie różnorodnych ataków (brutalnych, słownikowych, popularnych haseł itp.). Współczesne rozwiązania oparte na kartach graficznych osiągają

[2] NIST (ang. *National Institute of Standards and Technology*) — Narodowy Instytut Standaryzacji i Technologii w USA.

wydajności rzędu setek miliardów hashy na sekundę (na przykład Brutalis [11] firmy Sagitta złożony z ośmiu GPU[3] osiągający wydajność 200 GH/s dla funkcji MD5 [12]).

4. Metody przechowywania haseł

Najogólniej rzecz ujmując, statyczne hasła można przechowywać w dwóch postaciach: w postaci jawnej i w postaci niejawnej. Przechowywanie haseł w postaci niejawnej implikuje konieczność wykorzystania pewnych mechanizmów kryptograficznych (funkcji skrótu, algorytmów szyfrujących). Użycie tych algorytmów na potrzeby przechowywania haseł także ewoluowało na przestrzeni lat. Początkowo używane kryptograficzne funkcje skrótu zostały wzbogacone o pewną wartość losową, zwaną solą (ang. *salt*), następnie funkcje wzbogacono o możliwość zwielokrotnienia iteracji (wydłużenie czasu generowania hasła), w kolejnym kroku ewolucji stworzono specjalne algorytmy łączące zalety obu podejść (bcrypt, PBKDF), aż finalnie powstały przeznaczone do tego funkcje pamięciowo trudne (scrypt, ARGON2). Poniżej przedstawiono szczegółowo kolejne metody wraz z przykładowymi tabelami przechowującymi dwa hasła: ProdW@10, A1M3DakA oraz metody ich generacji w języku PHP.

4.1. Hasło przechowywane w postaci jawnej

Przechowywanie haseł w postaci jawnej polega na zapisaniu hasła użytkownika w jego oryginalnej postaci (czyli jako ciąg znaków wpisany przez użytkownika) na przykład w bazie danych. Podejście takie jest w dzisiejszych czasach niedopuszczalne, gdyż osoba mająca dostęp do repozytorium uzyskuje dostęp do haseł wszystkich użytkowników. Niestety, jak pokazują wycieki danych [13], do dzisiaj istnieją serwisy internetowe przechowujące hasła w ten sposób. W tabeli 1 pokazano sposób przechowywania haseł dla tej metody.

Tabela 1
Tabela haseł przechowywanych w postaci jawnej

uid	plaintext_password
1	ProdW@10
2	A1M3DakA

```
<?php
 $plaintext_password = $password;
?>
```

Kod 1. Przechowywanie hasła w postaci jawnej

[3] GPU (ang. *Graphics Processing Unit*) — procesor graficzny.

4.2. Hasło przechowywane w postaci zaciemnionej

Część badaczy [14] wskazuje jeszcze pewną modyfikację przechowywania haseł w postaci jawnej, a mianowicie przechowywanie ich w postaci „zaciemnionej" (ang. *obfuscation*). W podejściu tym albo staramy się ukryć samo miejsce przechowywania hasła, albo do samego hasła wprowadzamy pewien „szum" informacyjny (na przykład poprzez wprowadzenie do pierwotnego hasła losowych znaków między kolejnymi znakami hasła).

4.3. Hasło przechowywane w postaci zaszyfrowanej

Naturalną metodą zabezpieczenia się przed przechowywaniem haseł w postaci jawnej jest ich zaszyfrowanie. Szyfrowanie wymaga klucza szyfrującego, który musi być gdzieś przechowany w systemie. Jeśli atakujący uzyska dostęp do repozytorium haseł, to może również mieć dostęp do klucza(y) szyfrującego. Rozwiązanie to, mimo że bezpieczniejsze od poprzedniego, nie zabezpiecza jednak haseł przed odtworzeniem w przypadku kompromitacji repozytorium i klucza(y) szyfrującego. W tabeli 2 pokazano sposób przechowywania haseł zaszyfrowanych za pomocą algorytmu AES-128 w trybie ECB kluczem „password01234567".

Tabela 2
Tabela haseł przechowywanych w postaci zaszyfrowanej

uid	encrypted_password
1	96b00abf383cb0ca12f13d28d33fbe41
2	bfcaffa12a4701689ed066d92574de68

```
<?php
 $encrypted_password = bin2hex(mcrypt_encrypt( MCRYPT_RIJNDAEL_128,
 ,password01234567', $password, MCRYPT_MODE_ECB ));
?>
```

Kod 2. Przekształcanie hasła na postać zaszyfrowaną

4.4. Hasło przechowywane w postaci skrótu

Kolejną metodą przechowywania haseł jest wygenerowanie i przechowywanie skrótu hasła z wykorzystaniem kryptograficznej funkcji skrótu. Jednak i w tym przypadku atakujący może próbować odtworzyć oryginalne hasła. Ze względu na fakt, że kryptograficzną funkcję skrótu powinna cechować nieodwracalność, nie da się przeprowadzić ataku bezpośrednio, próbując niejako „odwrócić" skrót. Ale atakujący, wiedząc, że skróty haseł przechowywane w repozytorium utworzone zostały za pomocą konkretnego algorytmu (na przykład funkcji MD5), może skorzystać z jednego z podejść: łamania brutalnego (polegającego na sprawdzaniu wszystkich

możliwych haseł o określonej długości) lub wcześniej przygotować tablicę przechowującą pary: *pass* — *MD5*(*pass*) dla wszystkich możliwych haseł o określonej długości. Taka tablica skrótów może mieć wielkość setek terabajtów, przez co jej generowanie i przeszukiwanie staje się nieefektywne. Przykładowo dla wszystkich co najwyżej ośmioznakowych haseł złożonych ze znaków ze zbioru 62-elemetowego [a-zA-Z0-9] można utworzyć 221919451578090 różnych haseł (2).

$$\sum_{k=1}^{8} 62^k = 221919451578090. \tag{2}$$

Wykorzystując funkcję MD5 generującą 128-bitowe skróty i zakładając, że przechowywać będziemy tylko skróty bez haseł (hasła możemy generować za pomocą pewnego uporządkowanego[4] algorytmu), otrzymujemy tablicę skrótów o rozmiarze ponad 3550 TB (terabajtów). Przy dzisiejszym stanie techniki przetwarzanie takiej tablicy (wygenerowanie, zapisanie, przeszukiwanie) jest jak na razie zagadnieniem mało praktycznym.

Istnieje jednak znacznie bardziej efektywna metoda składowania skrótów nosząca w literaturze nazwę kompromisu czasowo-pamięciowego. Metoda zaprezentowana po raz pierwszy przez Hellmana [15] w 1980 roku przeżyła swój rozkwit w roku 2003 za sprawą pracy Oechslina [16]. Autor przedstawił jej zmodyfikowaną wersję opartą na tęczowych tablicach (ang. *rainbow tables*) oraz udostępnił oprogramowanie (ophcrack [17]) wraz z gotowymi tęczowymi tablicami pozwalającymi na łamanie haseł. Dla zaprezentowanego powyżej zbioru znaków i funkcji MD5 tablice pozwalające złamać hasło z prawdopodobieństwem 99,9% w średnim czasie kilku minut zajmują zaledwie 160 GB [18], co czyni ten typ ataku relatywnie łatwym do wykonania i podważa bezpieczeństwo tego sposobu przechowywania haseł.

Łamanie „brutalne" jest ściśle skorelowane z możliwościami obliczeniowymi dostępnego sprzętu. Należy zwrócić uwagę na fakt, że zgodnie z prawem Moore'a moc obliczeniowa nieustannie rośnie, a wykorzystanie kart graficznych do generowania skrótów i łamania haseł pozwala nam obecnie wygenerować 221919451578090 skrótów MD5 w czasie: 18 minut na przeznaczonym do tego urządzeniu (wspomniany Brutalis), czy też niespełna 2,5 godziny na komputerze klasy PC (karta graficzna GeForce 1080 — wydajność 250 GH/s[5]). Dodatkowo popularność kryptowalut, których większość

[4] Przykładowym uporządkowanym algorytmem może być generowanie kolejnych haseł, zmieniając kolejno poszczególne znaki w kolejności rosnącej, na przykład dla haseł czteroznakowych kolejne hasła miałyby postać:
aaaa, aaab, aaac, …, aaaz, aaaA, aaaB, …, aaaZ, aaa0, …, aaa9,
aaba, aabb, aabc, …, aabz, aabA, aabB, …, aabZ, aab0, …, aab9,
…
999a, 999b, 999c, …, 999z, 999A, 999B, …, 999Z, 9990, …, 9999.

[5] GH/s — 10^9 skrótów (hashy) na sekundę.

wydobywana jest, bazując na obliczaniu funkcji skrótu, także wpływa na postęp w projektowaniu coraz bardziej efektywnych urządzeń wyspecjalizowanych dla danego zadania. I tak przykładowo urządzenie Antminer S9 [19] służące do wydobywania bitcoinów charakteryzuje się wydajnością nawet 14 TH/s[6] dla funkcji skrótu SHA-256. Pozwala to na przeprowadzenie ataku na wszystkie hasła ośmioznakowe (zbudowane na 62-znakowym alfabecie) w czasie około 15 sekund. W tabeli 3 przedstawiono czas niezbędny do przeprowadzenia ataku na trzech wyżej wymienionych urządzeniach dla kilku różnych długości haseł na dwóch 62- oraz 95-znakowych (cyfry, litery i znaki specjalne) alfabetach.

Tabela 3

Czas łamania brutalnego dla haseł przechowywanych w postaci skrótu

długość hasła	liczba znaków	czas łamania brutalnego [y:ddd:hh:mm:ss][7]		
		GeForce 1080	Brutalis	Antminer S9
8	62	0:000:02:25:34	0:000:00:18:12	0:000:00:00:16
8	95	0:003:01:42:48	0:000:09:12:51	0:000:00:07:54
10	62	1:023:13:32:55	0:048:13:42:37	0:000:16:39:10
10	95	75:344:07:15:57	0:179:21:54:30	0:049:11:58:12
12	62	4092:062:00:28:16	511:190:06:03:32	7:112:05:13:46
12	95	685388:087:06:01:46	85673:193:09:45:13	1223:331:06:18:39

W tabeli 4 pokazano sposób przechowywania haseł z wykorzystaniem funkcji skrótu *MD5* dla tej metody.

Tabela 4

Tabela haseł przechowywanych w postaci skrótu

	hash [hex]
1	29f7322f2be0953544f60c663b54a845
2	5d311cccbea8690bafdb4f65a5d2e125

```
<?php
 $hash = md5($password);
?>
```

Kod 3. Przekształcanie hasła na skrót

[6] TH/s — 10^{12} skrótów (hashy) na sekundę.

[7] [y:ddd:hh:mm:ss] – liczba lat : dni : godzin : minut : sekund

4.5. Hasło przechowywane w postaci skrótu z solą

Metodą eliminującą niebezpieczeństwo płynące z zastosowania tęczowych tablic jest tak zwane „solenie" skrótów (ang. *salted hash*), którą dla przykładowej funkcji skrótu MD5 można zapisać w postaci *MD5*(*pass . salt*)[8]. Metoda ta, zaprezentowana w 1979 roku przez Morrisa i Thompsona [20], polega na dodaniu do każdego hasła tak zwanej „soli" — unikalnego ciągu losowego znaków o odpowiedniej długości — i wygenerowaniu skrótu dla konkatenacji soli i hasła. Implikuje to fakt, że te same hasła, dzięki zastosowaniu unikalnych wartości soli, będą miały inny skrót i zastosowanie tęczowych tablic będzie praktycznie niemożliwe przy dostatecznie długich wartościach soli (konieczne byłoby wygenerowanie tęczowych tablic dla wszystkich możliwych wartości soli). W tabeli 5 pokazano sposób przechowywania haseł z wykorzystaniem funkcji skrótu MD5 dla losowych wartości soli o długości 64 bitów dla tej metody.

Tabela 5

Tabela haseł przechowywanych w postaci solonego skrótu

uid	salt Hex	hash [Hex]
1	0123456789abcdef	0a27a226f3d5c225df71c2bfc619add6
2	3148efa092ab2ec0	3595dba5f67845cca3734b24ac902e93

```
<?php
 $hash = md5($password . $salt);
?>
```

Kod 4. Przekształcanie hasła na solony skrót

Przy znanej wartości soli złożoność ataku brutalnego jest identyczna jak w poprzednim wariancie przechowywania haseł (bez soli). Jeśli jednak atakujący nie zna wartości soli, wówczas złożoność ataku wzrasta o 2^s, gdzie s jest długością soli w zapisie binarnym (dla danych z tabeli 4 złożoność ta wzrosłaby o 2^{64}).

Pewną modyfikacją metody „solenia" jest przechowywanie skrótów z pieprzem (ang. *pepper hash*). Idea użycia pieprzu jest podobna do idei użycia soli, różnica wynika tylko z miejsca przechowywania tej losowej wartości. O ile sól przechowywana jest fizycznie w tym samym miejscu co skrót hasła (repozytorium, bazie danych, pliku), narażając ją na ujawnienie w przypadku włamania do systemu, o tyle pieprz musi spełniać jeden z warunków: powinien być przechowywany oddzielnie od skrótu hasła lub nigdy nie powinien być przechowywany [21].

[8] Symbol . oznacza tutaj konkatenację dwóch ciągów.

4.6. Hasło przechowywane w postaci iteracyjnego skrótu

Inną metodą zmniejszającą niebezpieczeństwo płynące z zastosowania tęczowych tablic jest wielokrotne iteracyjne skracanie kolejnych skrótów hasła. Podobnie jak w poprzedniej metodzie dla każdego hasła generowana jest losowa wartość, ale tym razem jest to wartość liczbowa określająca liczbę iteracji. Wartość ta określa liczbę iteracji skracania, a algorytm dla funkcji MD5 można zapisać w postaci *MD5*(*MD5*(… *MD5*(*pass*))). W tabeli 6 pokazano sposób przechowywania haseł z wykorzystaniem funkcji skrótu MD5 dla losowych wartości liczby iteracji dla tej metody.

Tabela 6

Tabela haseł przechowywanych w postaci iteracyjnego skrótu

uid	count	hash [Hex]
1	21029	69102bcee6365e6094bd907084481e62
2	99237	e704ece788493040f05f319ca5dcde18

```
<?php
 for ( $i = 0; $i < $count; $i++ ) $password = md5($password);
 $hash = $password;
?>
```

Kod 5. Przekształcanie hasła na iteracyjny skrót

Podejście to częściowo eliminuje główną słabość płynącą z wykorzystania kryptograficznych funkcji skrótu przy przechowywaniu haseł, a więc ich szybkość. A ponieważ łamanie „brutalne" jest ściśle skorelowane z możliwościami obliczeniowymi dostępnego sprzętu, liczbę iteracji należy dostosowywać do zakładanego poziomu bezpieczeństwa. Przykładowe wartości liczby iteracji w rzeczywistych systemach to: 20000 w BlackBerry Backup, czy nawet 220 w programie BitLocker To Go.

4.7. Hasło przechowywane w postaci iteracyjnego solonego skrótu

Naturalną ewolucją jest metoda łącząca w sobie zalety obu poprzednich metod: wykorzystania soli oraz wielokrotnych iteracji. Algorytm ten może przyjąć postać *MD5*(*MD5*(…*MD5*(*pass.salt*))) lub też *MD5*(*MD5*(…*MD5*(*pass.salt*)…).*salt*). Podejście to z jednej strony chroni przed użyciem tęczowych tablic (sól), a z drugiej utrudnia, a właściwie wydłuża, ataki brutalne (iteracje). W tabeli 7 pokazano sposób przechowywania haseł z wykorzystaniem funkcji skrótu *MD5* dla losowych wartości zarówno soli, jak i liczby iteracji dla tej metody.

Tabela 7

Tabela haseł przechowywanych w postaci iteracyjnego solonego skrótu

uid	salt [Hex]	count	hash [Hex]
1	0123456789abcdef	21029	0f906634f663cfe8e1f1c9bc5a9103cf
2	3148efa092ab2ec0	99237	42caea5cdeda91489177b185530922cb

```
<?php
 $hash = $password . $salt;
 for ( $i = 0; $i < $count; $i++ ) $hash = md5($hash);
?>
```

Kod 6. Przekształcanie hasła na iteracyjny solony skrót

W nurcie tego podejścia powstały algorytmy przeznaczone do przechowywania haseł, takie tak: *bcrypt* czy *PBKDF2*, które zostały opisane w dalszej części artykułu.

4.8. Hasło przechowywane w postaci trudnego pamięciowo skrótu

Rozwój narzędzi i technik do łamania haseł, wykorzystujących procesory graficzne GPU, układy ASIC[9] czy też przeznaczone do tego układy FPGA[10], spowodował, że podejście oparte na przechowywaniu haseł w postaci iteracyjnego solonego skrótu przestało być wystarczające. Kolejnym konceptem mającym chronić przez atakami brutalnymi było wzbogacenie tej metody o element powodujący konieczność wykorzystania dużych zasobów pamięciowych w celu wygenerowania skrótu (ang. *memory-hard hash function*). Podejście to znacząco osłabia ataki brutalne oparte na zrównoleglonych obliczeniach ze względu na niewystarczające zasoby pamięci do ich wykonania. W podejściu tym oprócz wykorzystania soli i złożoności obliczeniowej (liczby iteracji) podaje się także konieczne zasoby pamięci niezbędne do utworzenia skrótu hasła. Sama forma przechowywania haseł nie różni się od tej zaprezentowanej w tabeli 7. Do funkcji o parametryzowanej wielkości używanej pamięci przy generowaniu skrótu należą między innymi *scrypt*, *ARGON2* czy *Balloon*.

4.9. Hasło przechowywane w postaci skrótów dla wszystkich masek hasła

W systemach informatycznych wymagających podwyższonego poziomu bezpieczeństwa, na przykład w niektórych systemach bankowości internetowej, stosuje się jeszcze inne podejście. Podczas logowania użytkownik nie wprowadza całego hasła, lecz tylko jego wybrane znaki, tak zwaną maskę hasła. Przykładową maską hasła może być jego pierwszy, trzeci, piąty, siódmy znak, wówczas długość takiej

[9] ASIC (ang. *Application Specified Integrated Circuit*).
[10] FPGA (ang. *Field-Programmable Gate Array*).

maski wynosi 4. W podejściu takim w repozytorium dla każdej maski hasła przechowywany musi być skrót wyliczany za pomocą kryptograficznej funkcji skrótu wzbogaconej często o unikalną wartość „soli" dla danego użytkownika (tab. 8).

Tabela 8

Tabela haseł przechowywanych w postaci maskowanej

uid	mask	salt	password [Hex]
1	x_x_x_x_	0123456789abcdef	7dc000a92b1bdc998f388b09253a5dd3
1	xxxx____	wghyropld8dr3m0w	1e1be792b824aa7957fb3e5c2f05080e
1	…		
2	x_xx__x_	8efhn7qrjq9qjx9f	fe8dd4b37e583eeb98115b0cf5569c22
2	x__x__xx	79481jriuf7491na	6cc66a8d7132a3731c385a02cd025352
2	…		

Główną zaletą tej metody są stosunkowo niegroźne konsekwencje w przypadku przechwycenia pojedynczego maskowanego hasła przez nieuprawnioną osobę (np. poprzez podpatrzenie wpisywanego hasła lub przy wykorzystaniu keyloggera w kafejce internetowej). Poznanie tylko kilku znaków z całego hasła nie umożliwi jeszcze atakującemu włamania się do systemu. Kolejne maski wybierane przez system informatyczny powinny być dobierane tak, aby maksymalnie wydłużyć proces poznania całego hasła. Inną zaletą tej metody może być trudność w poznaniu rzeczywistej długości hasła przez atakującego. Jednak jej główną wadą jest potencjalna wielkość repozytorium.

Próbując oszacować liczbę rekordów konieczną do przechowania wszystkich możliwych masek haseł dla dwóch największych banków w Polsce spośród tych, które stosują hasła maskowane (Bank Pekao SA [22] oraz Alior Bank [23]), i zakładając, że:

— szacunkowa liczba użytkowników [24]: 3000000,
— minimalna i maksymalna długość hasła: 8 i 16 znaków,
— maska: ma zawsze długość $n/2$ dla n parzystego, ma długość $(n\text{-}1)/2$ lub $(n+1)/2$ dla n nieparzystego, gdzie n jest długością hasła,

można wyliczyć maksymalną liczbę rekordów przypadającą na pojedynczego użytkownika według wzoru (3)

$$C_{16}^{8} = \frac{16!}{8!\times(16-8)!} = 12870 \tag{3}$$

(liczba wszystkich możliwych 8-znakowych masek dla 16-znakowego hasła). To w rezultacie implikuje konieczność przechowywania ponad 38 miliardów rekordów, co nawet dla funkcji skrótu *MD5* powoduje konieczność użycia 575 GB pamięci na przechowanie samych skrótów. Inną wadą tej metody jest konieczność

wprowadzenia ograniczenia co do maksymalnej długości hasła, czego nie mamy w żadnej z wcześniej prezentowanych metod.

Autor dokonał badania polegającego na sprawdzeniu, jak w praktyce hasła maskowane przechowywane są w systemach transakcyjnych dwóch wyżej wymienionych banków [25]. Eksperyment polegający na wielokrotnym logowaniu się do systemu wykazał następujące własności dla systemu transakcyjnego Banku Pekao SA:

— niezależnie od długości hasła (sprawdzono hasła długości 8-15 znaków) system przechowuje co najwyżej **20** różnych masek,
— każda kolejna maska jest dobierana tak, aby był przynajmniej jeden nowy znak w stosunku do poprzedniej maski oraz aby maksymalnie wydłużyć liczbę logowań niezbędną do poznania całego hasła.

Podobny eksperyment dla banku Alior Bank wykazał, że liczba masek także jest prawdopodobnie ograniczona (mniejsza niż wszystkie możliwe maski): na 120 logowań uzyskano 86 unikalnych masek.

Resumując, przebadane systemy bankowe oparte na hasłach maskowanych nie korzystają ze zbioru wszystkich możliwych masek hasła, lecz ograniczają go do określonego podzbioru.

5. Wybrane funkcje adaptacyjne

W tym punkcie zostaną przedstawione najbardziej znane adaptacyjne funkcje do przechowywania haseł. Funkcje adaptacyjne (ang. *adaptive function*) to funkcje, w których użytkownik może modyfikować pewne ustawienia, dostosowując je do zakładanego poziomu bezpieczeństwa, najczęściej kosztem wydajności algorytmu. W funkcjach tych użytkownik może na przykład zwiększać z czasem liczbę iteracji (ang. *computation-hard*) czy też używane przez algorytm zasoby pamięci (ang. *memory-hard*) tak, aby wykonanie funkcji było wolniejsze, a tym samym atak brutalny trudniejszy do przeprowadzenia, nawet przy zwiększonej mocy obliczeniowej.

5.1. bcrypt

Funkcja *bcrypt* stworzona przez Provosa i Mazieresa w 1999 r. [26] należy do funkcji iteracyjnych z solą. Jej wewnętrzna struktura została oparta na algorytmie Blowfish, a użytkownik ma możliwość definiowania liczby powtórzeń tego etapu tworzenia skrótu, w którym występuje właśnie procedura rozszerzania klucza algorytmu Blowfish (określanych jako stopień złożoności obliczeniowej — ang. *cost*). Ogólna postać funkcji wygląda więc następująco: *bcrypt* (*pass*, *cost*, *salt*). Parametr *cost* określa liczbę iteracji algorytmu rozszerzania klucza, podawany jest jako wykładnik potęgi liczby 2. Ten stopień złożoności obliczeniowej może przyjmować wartości z zakresu od 4 do 31, zwiększenie go o 1 podwaja złożoność

obliczeniową i tym samym dwukrotnie wydłuża czas generowania skrótu. Przykładowo dla wartości 12 (212 = 4096 iteracji) generowanie skrótu na komputerze klasy PC[11] zajęło 0,3 sekundy, natomiast po zwiększeniu tej wartości do 15 niemal sześciokrotnie dłużej, czyli około 2,4 sekundy.

Tabela 9

Tabela haseł przechowywanych dla funkcji bcrypt

uid	hash [Hex]
1	$2y$12$0123456789abcdefghijk**eZ92.tue9QaSNYC0GFAXCPmvbcOhhU4u**
2	$2y$12$0123456789abcdefghijk**eRho5oRPoGxSY/rAIO6xdD9lKiVlZyJC**

W tabeli 9 przedstawiono sposób przechowywania haseł z wykorzystaniem funkcji *bcrypt* dla wartości soli wynoszącej 0123456789abcdefghijk oraz złożoności obliczeniowej (*cost*) wynoszącej 12. Jak widać, sam skrót reprezentowany jest za pomocą 32 znaków (kodowanie Base64), co odpowiada 192 bitom skrótu.

```
<?php
 $hash = password_hash($password, PASSWORD_BCRYPT, [,cost' => 12]);
?>
```

Kod 7. Przekształcanie hasła dla funkcji bcrypt

Aktualnie jako zalecany poziom bezpieczeństwa, spowalniający znacząco przed atakami brutalnymi, przyjmuje się złożoność obliczeniową (*cost*) o wartości 10. Jest to także domyślna obecnie wartość dla funkcji *password_hash w* języku PHP.

5.2. PBKDF2

Funkcja *PBKDF2* (ang. *Password-Based Key Derivation Function*) została opracowana w 2000 roku jako część standardu klucza publicznego (PKCS — Public-Key Cryptography Standards) [27] firmy RSA Laboratories. Podobnie jak *bcrypt* należy do funkcji iteracyjnych z solą, ale zamiast używać jednego konkretnego algorytmu do generowania hasła (w *bcrypt* jest to algorytm *Blowfish*), funkcja *PBKDF2* pozwala na wybór algorytmu, na przykład: *MD5*, *SHA-1*, *SHA256* i inne. Ogólna postać funkcji wygląda następująco: *PBKDF2* (*PRF*, *pass*, *salt*, *count*, *length*), gdzie: *PRF* jest funkcją pseudolosową wybieraną przez użytkownika (funkcja ta bazuje na kodzie HMAC [28] dla wybranej funkcji skrótu), *count* liczbą iteracji, a *length* opcjonalnym parametrem określającym pożądaną długość generowanego skrótu hasła (domyślna długość jest zależna od długości skrótu generowanego przez wybraną funkcję skrótu).

[11] Intel® Core™ i7-4600U 2.1 GHz, 8GB RAM.

W tabeli 10 przedstawiono sposób przechowywania haseł z wykorzystaniem funkcji *PBKDF2* dla funkcji *MD5* użytej jako funkcja pseudolosowa.

Tabela 10

Tabela haseł przechowywanych dla funkcji PBKDF2

uid	salt [Hex]	count	hash [Hex]
1	0123456789abcdef	21029	8eee792ac755eca338222fd08c7dd42f
2	3148efa092ab2ec0	99237	930b4ffb2095f107f7e9bb64a1c5214a

```
<?php
 $hash = hash_pbkdf2(„md5”, $password, $salt, $count);
?>
```

Kod 8. Przekształcanie hasła dla funkcji PBKDF2

Należy podkreślić, że mimo popularności i dostępności funkcji *PBKDF2*, przez wykorzystanie w jej strukturze klasycznych kryptograficznych funkcji skrótu, które są podatne na ataki z wykorzystaniem GPU, jej używanie powinno być poprzedzone analizą pożądanego poziomu bezpieczeństwa. Jako minimum zaleca się obecnie używanie tej funkcji z co najmniej 100 000 iteracjami i z wykorzystaniem funkcji skrótu z rodziny SHA-2. Dla takich parametrów skrót hasła na komputerze klasy PC generowany jest w czasie 0,2 sekundy.

5.3. scrypt

Funkcja *scrypt*, zaprezentowana w 2009 roku [29], a następnie w roku 2016 opublikowana przez IETF[12] jako RFC 7914 [30], jest pierwszą funkcją adaptacyjną, która oprócz możliwości dostrojenia liczby iteracji pozwala także na wykorzystanie określonej wielkości pamięci do wygenerowania skrótu. Funkcje tego typu nazywamy funkcjami pamięciowo trudnymi, a ich budowa zwiększa odporność na ataki z wykorzystaniem przeznaczonych do tego układów sprzętowych, na przykład ASIC. Uproszczona postać funkcji wygląda następująco: *scrypt* (*pass*, *salt*, *cost*, *length*), gdzie parametry wejściowe odpowiadają parametrom omówionym wcześniej. Wymagania pamięciowe w algorytmie wynikają z użycia dużego ciągu pseudolosowych bitów, które są generowane podczas wyliczania skrótu.

Wadą algorytmu *scrypt* jest jego podatność na ataki kanałem bocznym (ang. *side channel attack*), to znaczy, że informacje o haśle mogą być pozyskane z analizy użytych w procesie generowania skrótu komórek pamięci.

Algorytm ten nie był zaimplementowany natywnie w języku PHP, jednak od wersji 7.2 jest dostępny za pośrednictwem biblioteki libsodium.

[12] Internet Engineering Task Force.

```
<?php
 $hash = sodium_crypto_pwhash_scryptsalsa208sha256
 ($length , $password , $salt, $opslimit, $memlimit);
?>
```

Kod 9. Przekształcanie hasła dla funkcji scrypt

5.4. ARGON2

Funkcja ARGON2 to zwycięzca konkursu Password Hashing Competition [31] (PHC) wyłoniony w połowie 2015 roku [32]. Jej adaptacyjność, w przeciwieństwie do funkcji *bcrypt* czy *PBKDF2*, które wykorzystywały jeden czynnik złożoności obliczeniowej, polega na możliwości ustawienia aż trzech parametrów: liczby iteracji, wielkości wykorzystywanej pamięci oraz tak zwanego paralelizmu (liczby równoległych wątków). Ponadto można sprecyzować dodawaną sól, długość wyjściowego skrótu czy też dodatkowy opcjonalny klucz. Ogólna uproszczona postać funkcji wygląda następująco: *ARGON2* (*pass*, *salt*, *parallelism*, *length*, *memory*, *count*, *key*), gdzie: *parallelism* oznacza poziom zrównoleglenia obliczeń (na przykład liczbę wątków), *memory* to ilość pamięci niezbędnej do obliczenia skrótu), *key* jest opcjonalnym dodatkowym kluczem, a pozostałe parametry — *salt*, *length*, *count* — zostały omówione wcześniej. Wnętrze algorytmu bazuje na kryptograficznej funkcji skrótu BLAKE2 [33] będącej jednym z finalistów konkursu na standard SHA-3 [34]. Algorytm ten został wprowadzony do języka PHP od wersji 7.2.

```
<?php
 $hash = password_hash($password, PASSWORD_ARGON2I,
 [,memory_cost' => 1<<17, ,time_cost' => 4, ,threads' => 3] );
?>
```

Kod 10. Przekształcanie hasła dla funkcji ARGON2

5.5. Balloon

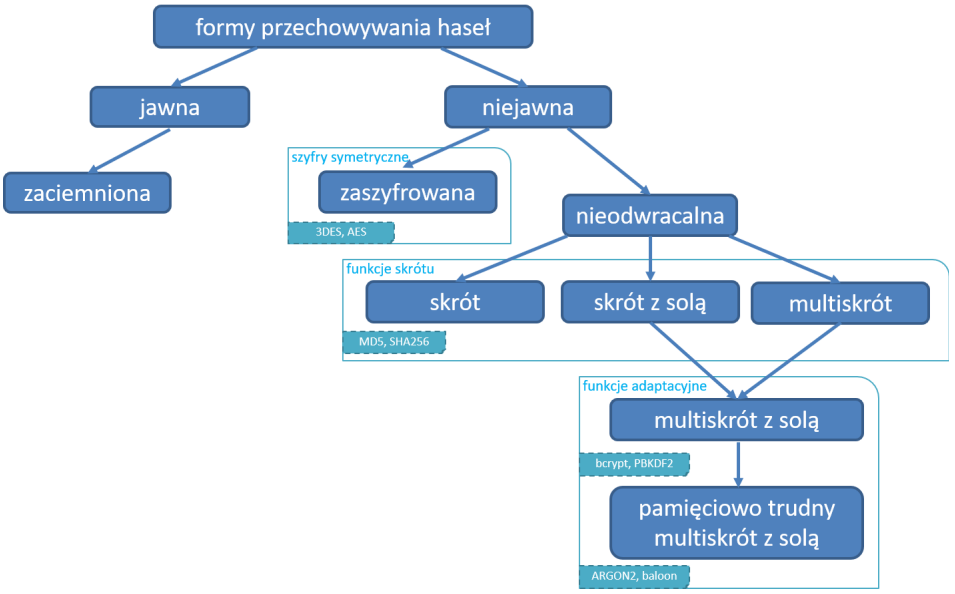
Funkcja Balloon [35, Boneh] została przedstawiona przez autorów, wskazując na pewne słabości funkcji ARGON2, podważając jednocześnie jej użyteczność ze względu na brak formalnej analizy bezpieczeństwa, którą sama funkcja Balloon posiada. Ogólna postać funkcji wygląda następująco: *Balloon* (*pass*, *salt*, *space*, *rounds*), gdzie: *space* oznacza poziom zrównoleglenia obliczeń (na przykład liczbę wątków), parametr *rounds* to liczba iteracji wykonywanych wewnątrz funkcji. Algorytm ten nie jest zaimplementowany natywnie w języku PHP. Aby z niego skorzystać, należy użyć zewnętrznych bibliotek.

5.6. Inne funkcje

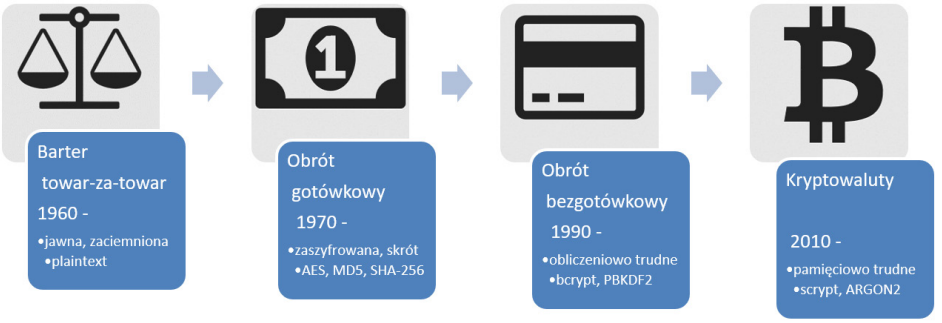
Wspomniany konkurs PHC przyniósł także kilka innych ciekawych funkcji adaptacyjnych: Catena, Lyra2, yescrypt [36, Peslyak], czy też Makwa.

6. Systematyzacja

Podsumowując powyższe rozważania, systematyzując wiedzę i wprowadzając autorski podział porównujący użycie haseł do użycia pieniędzy (w różnych jego formach), można przedstawić omówione formy przechowywania haseł w systemach informatycznych w postaci graficznej (rys. 1) i ich ewolucję (rys. 2).



Rys. 1. Formy przechowywania haseł



Rys. 2. Ewolucja form przechowywania haseł

7. Kierunki dalszych badań

W dalszej części prac nad przechowywaniem haseł w systemach informatycznych zostaną przeprowadzone badania polskich serwisów internetowych pod względem polityki haseł oraz sposobów ich przechowywania. Innym kierunkiem badawczym będzie próba zaprojektowania autorskiej adaptacyjnej funkcji do przechowywania haseł.

8. Podsumowanie

Hasła statyczne, należące do metod zwanych „**coś co wiesz**" (ang. *something you know*), nie są najbezpieczniejszą metodą uwierzytelniania użytkowników. Bezpieczniejsze rozwiązania oparte są na metodach typu „**coś co masz**" (ang. *something you have*) czy też „**coś czym jesteś**" (ang. *something you are*). Te pierwsze to na przykład hasła jednorazowe generowane za pomocą przeznaczonych do tego urządzeń (tokenów) lub też przekazywane przez wiadomości SMS. Te drugie oparte są na technikach biometrycznych (siatkówka oka, linie papilarne itp.). Mimo ich niedoskonałości są jednak najczęściej stosowaną techniką uwierzytelniania. Za tą popularnością stoją takie czynniki jak: niskie koszty wdrożenia, prostota rozwiązania czy też duża akceptacja społeczna.

Projektując bezpieczny system informatyczny, należy dołożyć wszelkich starań, aby hasła użytkowników były zabezpieczone w sposób uniemożliwiający, a przynajmniej skutecznie utrudniający ich odtworzenie w przypadku wycieku bazy danych. Pamiętać jednak trzeba o tym, że bezpieczne przechowanie haseł nie jest wystarczające dla zapewnienia ich poufności.

Hasło powinno być odpowiednio długie i złożone, na przykład powinno mieć co najmniej dwanaście znaków i wymóg występowania w nim kombinacji małych liter, wielkich liter, cyfr, znaków specjalnych (chociaż najnowsze zalecenia NIST-u odchodzą od tej praktyki [37]). Hasło nie powinno znajdować się w bazach

popularnych haseł czy słownikach, nie powinno też stanowić sekwencji znaków (np.: '11111111', '1234abcd'). Zaleca się nieograniczanie maksymalnej długości hasła, aby dawać użytkownikom możliwość wpisywania dowolnie długich fraz hasłowych (ang. *pass-phrase*) i akceptować „spację". Także sam proces logowania powinien odbywać się z wykorzystaniem bezpiecznych protokołów, na przykład protokołu TLS w wersji 1.2, zamiast SSL w wersji 3.0, który okazał się podatny na atak o akronimie POODLE [38]. W celu zabezpieczenia się przed atakiem brutalnym lub słownikowym polegającym na kolejnych próbach podawania danych do logowania, zaleca się blokowanie dostępu do systemu po kilku nieudanych próbach logowania (np. fail2ban), czy też stopniowe wydłużenie czasu odpowiedzi podczas kolejnych nieudanych logowań, lub zabezpieczenia typu „CAPTCHA". Ważna jest także edukacja użytkowników podkreślająca co najmniej kilka dobrych praktyk: ochronę komputerów/smartfonów przed wirusami, używanie różnych haseł dla różnych serwisów, nieudostępnianie nikomu swoich haseł, nieużywanie publicznych komputerów dostępnych w hotelach/ kafejkach do logowania, nieużywanie otwartych sieci Wi-Fi do logowania.

Niestety, jak pokazują wycieki haseł [39], do których w zasadzie zdążyliśmy się już przyzwyczaić, hasła w systemach informatycznych nadal bardzo często są przechowywane w postaci funkcji MD5. Funkcji, która poza swoimi słabościami kryptograficznymi (praktyczne kolizje dla ciągów znaków [40], plików w formacie jpg [41], exe [42] czy pdf [43]), praktycznie nie nadaje się do przechowywania haseł ze względu na wydajność, jaką osiąga się współcześnie przy łamaniu dzięki zastosowaniu układów GPU czy ACIS. Bolączka ta dotyczy też innych klasycznych kryptograficznych funkcji skrótu (MD4, SHA-1, SHA-256 i in.).

Czas najwyższy zmienić to przyzwyczajenie projektantów systemów informatycznych, w których uwierzytelnienie oparte jest na statycznych hasłach. W dzisiejszych czasach powinniśmy wyjść z założenia, że atakujący wcześniej czy później uzyska dostęp do wrażliwych informacji uwierzytelniających, jakimi są skróty haseł. Poziom bezpieczeństwa systemu będzie już wtedy zależał tylko od złożoności ataku, polegającego najczęściej na łamaniu brutalnym lub słownikowym, którego skuteczność uzależniona będzie od wykorzystanego algorytmu przechowywania hasła. Należy więc definitywnie odejść od klasycznych kryptograficznych funkcji skrótu i używać funkcji adaptacyjnych.

Źródło finansowania pracy — środki własne autorów.

Artykuł wpłynął do redakcji 29.12.2017 r. Zweryfikowaną wersję po recenzjach otrzymano 5.03.2018 r.

LITERATURA

[1] www.wired.com/2012/01/computer-password/ [dostęp 30.11.2017]

[2] www.uber.com/newsroom/2016-data-incident/ [dostęp 30.11.2017]

[3] www.hydraze.org/2015/08/ashley-madison-full-dump-has-finally-leaked/ [30.11.2017]

[4] www.reddit.com/r/hacking/comments/2n9zhv/ [dostęp 30.11.2017]

[5] www.niebezpiecznik.pl/post/wlamanie-na-serwery-panstwowej-komisji-wyborczej-wykradziono-hashe-hasel-i-klucze-urzednikow/ [dostęp 30.11.2017]

[6] www.niebezpiecznik.pl/post/gielda-papierow-wartosciowych-zhackowana/ [30.11.2017]

[7] www.niebezpiecznik.pl/post/wyciek-7-milionow-hasel-do-dropboxa/ [dostęp 30.11.2017]

[8] www.cnn.com/2012/07/12/tech/web/yahoo-users-hacked [dostęp 30.11.2017]

[9] Wang X., Feng D., Lai X., Yu H., *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, 2004, online: http://eprint.iacr.org/ 2004/199.

[10] Stevens M., Bursztein E., Karpman P., Albertini A., Markov Y., Bianco A.P., Baisse C., *Announcing the first SHA1 collision*, Google Security Blog, 2017, online: https://security.google-blog.com/2017/02/announcing-first-sha1-collision.html

[11] www.sagitta.pw/hardware/ [dostęp 30.11.2017]

[12] www.gist.github.com/epixoip/a83d38f412b4737e99bbef804a270c40 [dostęp 30.11.2017]

[13] www.vigilante.pw [dostęp 30.11.2017]

[14] Kenan K., *Kryptografia w bazach danych. Ostatnia linia obrony*, Wydawnictwo Mikom, 2007.

[15] Hellman M.E., *A cryptanalytic time-memory trade-off*, IEEE Transactions on Information Theory, vol. 26, no. 4, 1980, 401-406.

[16] Oechslin P., *Making a faster Cryptanalytic Time-Memory Trade-Off*, Advances in Cryptology — CRYPTO 2003, LNCS 2729, Springer-Verlag, 2003, online: lasec.epfl.ch/pub/lasec/doc/oech03.pdf

[17] www.ophcrack.sourceforge.net [dostęp 30.11.2017]

[18] www.project-rainbowcrack.com/table.htm [dostęp 30.11.2017]

[19] www.shop.bitmain.com/antminer_s9_asic_bitcoin_miner.htm [dostęp 30.11.2017]

[20] Morris R., Thompson K., *Password Security: A Case History*, Communications of the ACM, 22(11), 1979, pp. 594-597, online: http://www.cs.unibo.it/~sacerdot/doc/papers/Morris-PasswordSecurity.pdf

[21] Manber U., *A simple scheme to make passwords based on one-way functions much harder to crack*, Computers & Security, 15(2), 1996, 171-176.

[22] www.rodwald.pl/blog/314/polityka-hasel-pekao24-pl [dostęp 30.11.2017]

[23] www.rodwald.pl/blog/316/polityka-hasel-aliorbank-pl [dostęp 30.11.2017]

[24] www.prnews.pl/raporty/raport-rynek-bankowosci-6553183.html [dostęp 30.11.2017]

[25] www.rodwald.pl/blog/550/hasla-maskowane-w-polskich-bankach [dostęp 30.11.2017]

[26] Provos N., Mazières D., *A Future-Adaptable Password Scheme*. Proceedings of 1999 USENIX Annual Technical Conference, online: https://www.usenix.org/event/usenix99/provos/provos.pdf

[27] Kaliski B., *PKCS #5: Password-based cryptography specification, version 2.0*, IETF Network Working Group, RFC 2898, 2000.

[28] https://tools.ietf.org/html/rfc2104 [dostęp 30.11.2017]

[29] Percival C., *Stronger Key Derivation via Sequential Memory-Hard Functions*, BSDCan, 2009, online: https://www.tarsnap.com/scrypt/

[30] https://tools.ietf.org/html/rfc7914 [dostęp 30.11.2017]

[31] www.password-hashing.net/index.html [dostęp 30.11.2017]

[32] Biryukov A., Dinu D., Khovratovich D., *Version 1.2 of Argon2*, https://password-hashing.net/submissions/specs/Argon-v3.pdf, 2015.

[33] https://tools.ietf.org/html/rfc7693 [dostęp 30.11.2017]

[34] www.csrc.nist.gov/projects/hash-functions/sha-3-project [dostęp 30.11.2017]

[35] Boneh D., Henry Corrigan-Gibbs H., Schechter S., *Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks*, https://eprint.iacr.org/2016/027.pdf, 2016.

[36] Peslyak A., *yescrypt: large-scale password hashing*, BSides Ljubljana, 2017, online: https://bsidesljubljana.si/yescrypt-large-scale-password-hashing-alexander-peslyak

[37] https://pages.nist.gov/800-63-3/sp800-63b.html [dostęp 30.11.2017]

[38] www.openssl.org/~bodo/ssl-poodle.pdf [dostęp 30.11.2017]

[39] www.centermast.files.wordpress.com/2017/03/hashingalgo.png [dostęp 30.11.2017]

[40] www.mathstat.dal.ca/~selinger/md5collision [dostęp 30.11.2017]

[41] natmchugh.blogspot.com/2015/02/create-your-own-md5-collisions.html [30.11.2017]

[42] www.mathstat.dal.ca/~selinger/md5collision [dostęp 30.11.2017]

[43] www.win.tue.nl/hashclash/Nostradamus [dostęp 30.11.2017]

P. RODWALD, B. BIERNACIK

Password protection in IT systems

Abstract. The aim of the article is to systematise the methods of securing static passwords stored in IT systems. Pros and cons of those methods are presented and conclusions as a recommendation for IT system designers are proposed. At the beginning, the concept of cryptographic hash function is presented, following discussion of methods of storing passwords showing their evolution and susceptibility to modern attacks. Results of research on masked passwords of Polish banks IT systems are presented, as well as the most interesting examples of adaptive password functions are given. Then, the systematisation of password protection methods was carried out. Finally, the directions for further research are indicated.

Keywords: computer security, password, authentication, hash function
DOI: 10.5604/01.3001.0011.8036