Kryptografia i bezpieczeństwo danych
- odporność na kolizje II

Sławomir Samolej
ssamolej.kia.prz.edu.pl
ssamolej@prz.edu.pl

Odporność na kolizje: przegląd

Niech $H: M \rightarrow T$ będzie funkcją hash ($|M| >> |T|$)

Kolizja dla funkcji H jest parą $m_0, m_1 \in M$ taką, że:

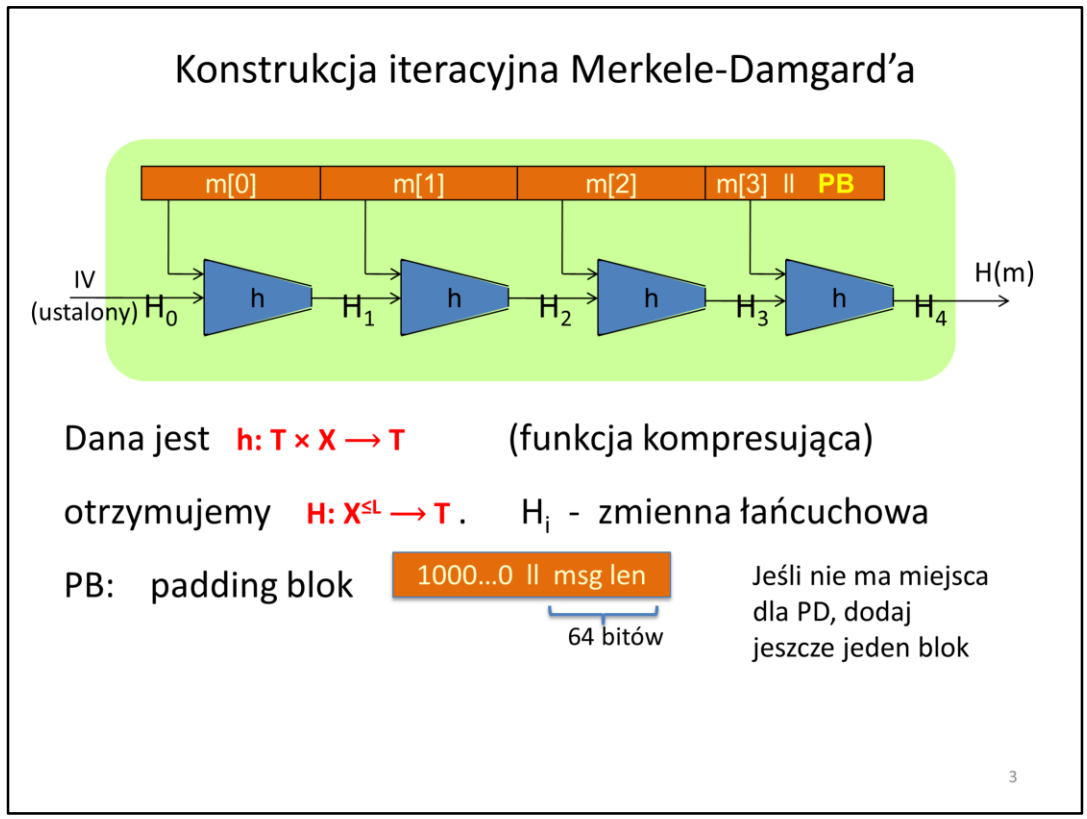
$$H(m_0) = H(m_1) \quad \text{and} \quad m_0 \neq m_1$$

Cel: opracować funkcję hash odporną na kolizje

Krok 1: mając funkcję odporną na kolizję dla **krótkich** wiadomości,
konstruujemy funkcję odporną na kolizję dla **długich** wiadomości

2

Mamy funkcję hashującą, która przekształca długie wiadomości w małe tagi. Kolizja następuje wtedy, gdy dla dwóch różnych wiadomości otrzymuje się takie same funkcje hash. Warto pamiętać, że chodzi tu o znalezienie efektywnych algorytmów, które taką kolizję znajdą a nie tylko stwierdzenie, że takich kolizji jest „dużo".
Teraz zajmiemy się opracowaniem funkcji hash według następującego pomysłu. Zakładając, że dysponujemy funkcją obliczającą hash dla krótkich wiadomości, rozszerzymy jej funkcjonalność dla wiadomości dłuższych. Później pokażemy, jak zdefiniować taką funkcję HASH dla krótkich wiadomości i wiedza się nam uzupełni.



Rozważmy konstrukcję jak na rysunku. Jest to ogólna konstrukcja, według której buduje się funkcje skrótu. Zakładamy, że dysponujemy funkcją h (hash) odporną na kolizje dla małych wiadomości. Funkcja nazywana jest też funkcją kompresującą. Wiadomość jest dzielona na bloki. W konstrukcji stosowany jest również IV (ang. Initialisation Vector), tym razem ustalany jednokrotnie raz na zawsze i wbudowywany w program (jego wartość jest ustalana na poziomie definiowania standardu). Wyjście funkcji kompresji (H_n nazywane zmienną łańcuchową) jest przekazywane na wejście kolejnej funkcji kompresji, która przekształca także kolejny blok wiadomości. Łańcuch przekształceń kolejnych porcji wiadomości jest kontynuowany do momentu osiągnięcia ostatniego bloku. Do ostatniego bloku musi zostać dodany tzw. „padding blok". Ostatnia część widomości wraz z paddingiem jest przekształcana z zastosowaniem funkcji h i otrzymujemy wyliczony hash dla długiej wiadomości. Padding blok składa się z pola 10000…0 uzupełniającego wiadomość do odpowiedniej długości oraz 64 bitowego pola zawierającego długość wiadomości. Rozmiar wiadomości jest kodowany w polu o ustalonej długości. Przykładowo, we wszystkich funkcjach SHA długość wiadomości jest ograniczona do 2^{64}-1 (ostatni blok może zwierać sam padding i długość wiadomości, jeśli wiadomość ma długość równą całkowitej wielokrotności bloku). Ograniczenie wiadomości do podanego rozmiaru w rzeczywistości nie stanowi zbyt wielkiego ograniczenia (ok. 18 tys. TB).

Kolizja w MD

Tw: jeśli h jest odporna na kolizję to H jest także odporna na kolizje.
Dowód można przeprowadzić, przez wykazanie, że jeśli istnieje kolizja w H, to istnieje kolizja w h.

Wniosek: do skonstruowania efektywnej funkcji hash dla długich wiadomości wystarczy pokazana wcześniej konstrukcja i „mała" funkcja hash (kompresująca) nie zawierająca kolizji.

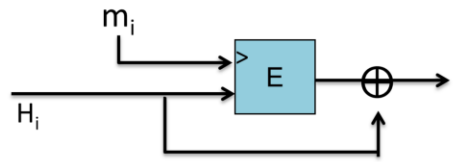
4

Jest udowodnione twierdzenie, że jeśli funkcja h jest odporna na kolizje, to funkcja H jest również odporna na kolizje. Wystarczy więc opracować algorytm funkcji h (odpornej na kolizje) aby móc konstruować funkcje hash dla długich wiqadomości.

Funkcja kompresji na bazie szyfru blokowego

Niech **E: K× {0,1}^{n} ⟶ {0,1}^{n}** będzie szyfrem blokowym.

Funkcja kompresji **Davies-Meyer'a** ma postać: **h(H, m) = E(m, H)⊕H**



Tw: Załóżmy, że E jest idealnym szyfrem (kolekcja |K| losowych permuTacji.). Znalezienie kolizji **h(H,m)=h(H',m')** zajmuje wtedy **O(2$^{n/2}$)** sprawdzeń (E,D).

Najlepsze z możliwych !!!

5

Na początku rozważmy, czy dysponując prymitywami, które już znamy (np. alg. szyfrowania) możemy zbudować odporną na kolizje „małą" funkcję hash. Odpowiedź brzmi tak.
Zakładamy, że mamy pewien szyfr blokowy pracujący na n-bitowych blokach (przekształcający n-bitowy blok w inny n-bitowy blok danych). Blok szyfrujący możemy przekształcić w funkcję hash dla małej porcji danych stosując konstrukcję jak na slajdzie (konstrukcja Davies-Meyer'a). Zmienną łańcuchową z poprzedniej fazy algorytmu H_i (lub IV jeśli jest to pierwszy blok algorytmu) traktujemy jako wiadomość do zaszyfrowania, a kluczem jest blok wiadomości m_i. Dodatkowo na otrzymanym szyfrogramie wykonujemy xor z wartością H_i.
Może to wygląda trochę dziwacznie, bo blok wiadomości, którą może dowolnie manipulować atakujący traktujemy jako klucz. Można jednak udowodnić, że taka konstrukcja (jeśli E jest idealnym szyfrem) jest odporna na kolizje w takim stopniu, jak założyliśmy. Znalezienie kolizji wymagałoby przeprowadzenia $O(2^{n/2})$ szyfrowania i deszyfrowania.
Twierdzenie mówi, że jeśli E jest idealnym szyfrem, czyli składa się z |K| losowych permutacji, to znalezienie kolizji zajmie czas $O(2^{n/2})$. Wracając do paradoksu dnia urodzin, to podana konstrukcja jest tak bezpieczna jak tylko może być.

Co się stanie, gdy zrezygnujemy w konstrukcji „ostatni” xor z H?

- Zmodyfikowana konstrukcja: **h(H, m) = E(m, H)**
- Okazuje się, że nie jest bezpieczna
  Wystarczy wybrać losowo: (H, m, m’) i skonstruować H’ = D(m’,E(m,H))

$E(m', H') = E(m', D(m', E(m, H)))$ / $E(k, D(k, m) = m$

$E(m', H') = E(m, H)$ ⇝ kolizja

6

Pokazana na poprzednim slajdzie konstrukcja musi być traktowana w całości. Jeśli usuniemy z niej wykonanie „ostatniej” operacji xor, to szybko uzyskujemy konstrukcję, która nie jest odporna na kolizję. Dla losowych H, m i m’, gdzie m ≠ m’.
Wystarczy, że jedno wejść bloku szyfrowania zostanie skonstruowane z zależności H’ = D(m’,E(m,H)), wtedy wykonanie na nim szyfrowania: E(m’, H’), czyli po rozwinięciu: E(m’, D(m’, E(m, h))). Z właściwości szyfrów z kluczem symetrycznym: E(k, D(k, M) = m. Gdy spojrzymy na wypracowaną formułę w podobny sposób otrzymujemy zależność: E(m’,H’)=E(m, k). Następuje kolizja.

Inne konstrukcje oparte o szyfry blokowe

Niech **E: $\{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^n$** (w uproszczeniu)

Miyaguchi-Preneel: **h(H, m) = E(m, H)⊕H⊕m** **(Whirlpool)**
h(H, m) = E(H⊕m, m)⊕m

W sumie istnieje 12 wariantów rozwiązań…

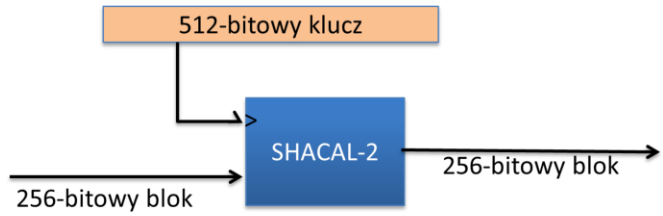
Okazuje się jednak, że inne, jak się wydaje naturalne warianty, np.:
h(H, m) = E(m, H)⊕m
nie są bezpieczne.

7

Konstrukcja Davies-Meyer'a nie jest jedyną stosującą szyfrowanie blokowe do opracowania funkcji kompresujących. Na slajdzie pokazano dwa przykładowe rozwiązania. Pierwsze, (Miyaguchi-Preneel) jest stosowane w rozwiązaniu funkcji hash Whirlpool. Jest w sumie opublikowanych 12 wariantów obliczania funkcji kompresującej z zastosowaniem szyfrów blokowych. Warto zwrócić uwagę, że nie wszystkie, jak się wydaje, naturalne konstrukcje okazują się bezpieczne. W dolnej części slajdu przytoczyłem jedną z nich.

Realny przykład: SHA-256

- Konstrukcja Merkle-Damgard'a
- Funkcja kompresji Davies-Meyer'a function
- Szyfr blokowy: SHACAL-2



8

Mamy więc wszystkie składowe, które pozwalają opisać jeden z najpopularniejszych algorytmów funkcji hash – SHA-256. Okazuje się, że w algorytmie zastosowano konstrukcję Merkle-Damgard'a. Wykorzystuje on funkcję kompresji Davies-Meyer'a. Algorytmem szyfrowania blokowego, jaki zastosowano jest SHACAL-2. Parametry algorytmu szyfrowania są następujące: klucz szyfrowania ma 512 bitów długości. Przy czym należy pamiętać, że kluczem są tu poszczególne bloki wiadomości(m_i). Dane do szyfrowania (H_{i-1} - zmienna łańcuchowa) w jednej turze mają długość 256 bitów i oczywiście ostatecznie długość otrzymanego hash wynosi 256 bitów (H_i). Algorytm SHACAL-2 nie będzie omawiany.

Inne funkcje kompresji odporne na kolizje

Wybierzmy 2000-bitową liczbę pierwszą p i dwie losowe wartości u i v z przedziału $1 \le u, v \le p$.

Dla m, H ∈ {0,…,p-1} definiujemy **$h(H,m) = u^H \cdot v^m \pmod p$**

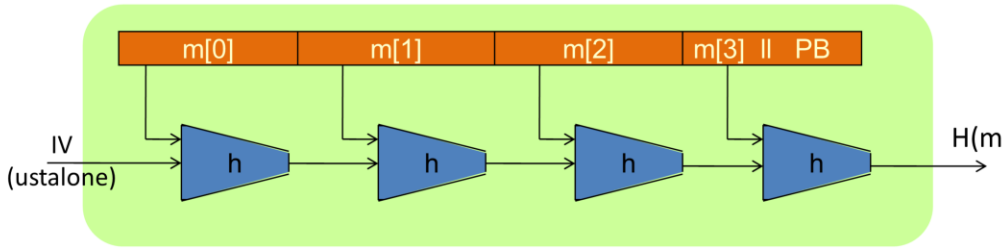
Fakt: znalezienie kolizji dla tak skonstruowanej funkcji h(.,.) jest tak trudne jak rozwiązanie problemu dyskretnego logarytmu modulo p.

Problem: Wolna.

9

Jedna z klas funkcji kompresji jest budowana w oparciu o szyfry blokowe. Okazuje się, że jest inna klasa mająca swoje podstawy w teorii liczb. Przedstawiony jest tu jeden przykładowy algorytm. Kolizja może być tu znaleziona, jeśli uda się rozwiązać bardzo trudny problem z teorii liczb.
Działanie algorytmu zaczyna się od wybrania bardzo dużej liczby pierwszej **p**, zapisywanej w systemie dwójkowym na 2000 bitach, co daje około 700 cyfr w systemie dziesiętnym. Następnie wybiera się dwie losowe liczby **u** i **v** z przedziału **1** do **p-1**. Wartości m i H muszą należeć do przedziału {0,…,p-1}. Funkcja kompresująca jest opisana zależnością: $h(H, m) = u^H \bullet v^m \pmod p$. Bierze ona 2 liczby z podanego przedziału i zwraca jedną (zastosowana jest operacja reszta z dzielenie przez p).
Znalezienie kolizji w takim systemie jest tak trudne, jak rozwiązanie problemu dyskretnego algorytmu (znaleźć a, b, c całkowite spełniające zależność $a^c=b$, zwłaszcza dla dużych liczb), modulo p.
Rozwiązanie jest rzadko stosowane w praktyce ponieważ proces obliczeniowy trwa długo, w porównaniu z algorytmami opartymi na szyfrach blokowych. Jeśli mamy do dyspozycji standardowy komputer, to możemy za jego pomocą obliczyć skrót dla długiej wiadomości z zastosowaniem tego algorytmu, ale zajmie to około jednego dnia…

Czy można skonstruować MAC z zastosowaniem konstrukcji Merkle-Damgard'a?



Tw: h jest odporne na kolizje ⇒ H jest odporne na kolizje

Czy możemy zastosować H(.) do bezpośredniego zbudowania MAC?

10

Przypomnijmy konstrukcję Merkle-Damgard'a. Jest ona odporna na kolizje, jeśli sama funkcje h jest odporna na kolizje. Odpowiednio zastosowane szyfry blokowe (ujęte w odpowiednią konstrukcję) mogą praktycznie posłużyć jako funkcje h (funkcje kompresujące). Konstrukcja pozwala na obliczenie hash dla dużych wiadomości (praktycznie o rozmiarach do 2^{64} bloków). Powstaje pytanie, czy mając funkcję H(.) możemy bezpośrednio zbudować MAC (ang. Message Authentication Code) bez opierania się na PRF (ang. Pseudo Random Function)?

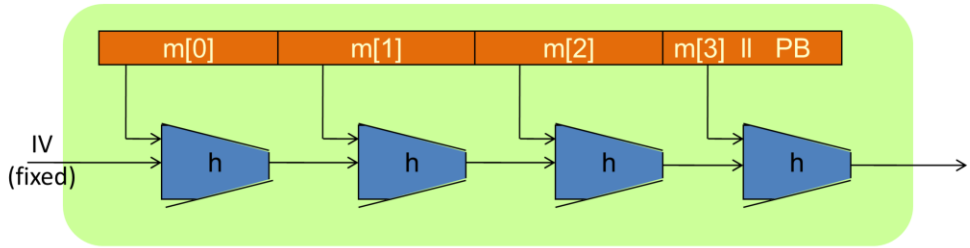
Hash i MAC - pierwsze podejście

Dysponujemy **H: $X^{\leq L} \longrightarrow T$** odporną na kolizje funkcją hash zbudowaną z zastosowaniem konstrukcji Merkle-Damgard'a

Podejście #1: **S(k, m) = H(k || m)**

Niestety to rozwiązanie nie jest bezpieczne:

Mając H(k || m) możemy obliczyć H(k || m || PB || w) dla każdego w.



11

W pierwszym odruchu moglibyśmy spróbować zbudować MAC poprzez połączenie klucza i wiadomości w jedną wiadomość i obliczenie dla nich funkcji hash (H(.)). Takie rozwiązania niestety nie jest bezpieczne ponieważ jest podatne na atak na rozszerzenie wiadomości. W takiej konstrukcji możemy dodać do wiadomości jeszcze jeden blok (w) i zażądać wykonania jeszcze jednej rundy funkcji h. Otrzymalibyśmy poprawny wynik generujący hash, ale dla sfałszowanej (rozszerzonej przez atakującego) wiadomości. Okazuje się, że istnieje wiele produktów kryptograficznych właśnie w tak błędny sposób stosujący tę konstrukcję. Nie jest to rozwiązanie bezpieczne i nigdy nie powinno być aplikowane.

Ustandaryzowana metoda HMAC (Hash MAC)

Najczęściej stosowana metoda obliczania MAC w Internecie.

H: funkcja hash.
przykład: SHA-256 ; wyjście: 256 bitów

Zasada budowania MAC na bazie funkcji hash:

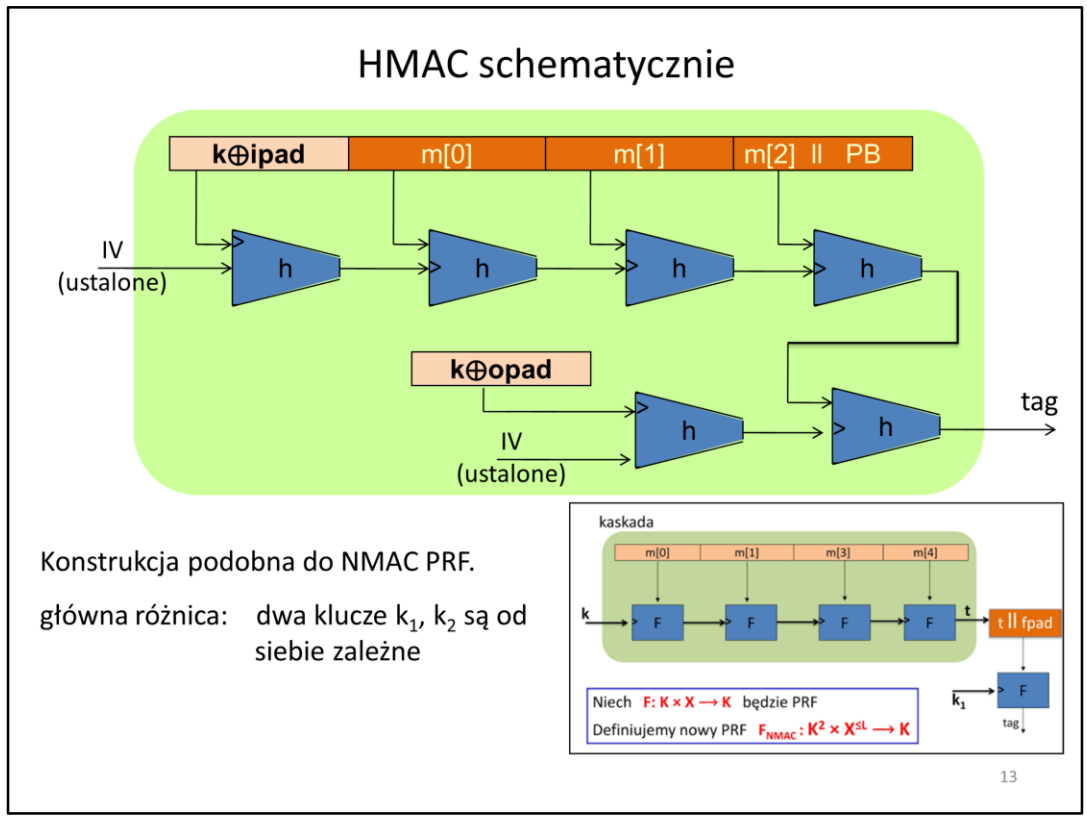
HMAC: $S(k, m) = H(k \oplus opad \;||\; \mathbf{H(k \oplus ipad \;||\; m)})$

12

Do zbudowania MAC na bazie funkcji hash można przykładowo zastosować funkcję SHA-256, która zwraca wartość 256 bitową. Uznaje się, że wyjście SHA-256 jest tożsame z PRF (pseudo random function), w związku z tym spełnione jest podstawowe wymaganie co do MAC. Konstrukcja HMAC opisana symbolicznie ma postać:

$S(k, m) = H(k \oplus opad \;||\; \mathbf{H(k \oplus ipad \;||\; m)})$.

Najpierw brany jest klucz **k** i dołączany do niego **ipad** (ang. Internal Pad). Powoduje to przygotowanie pierwszego bloku danych („wiadomośći") w konstrukcji Merkle-Damguard'a (jeśli stosowany jest alg. SHA-256, to blok ma długość 512 bitów). Blok ten jest dołączany z przodu wiadomości i wykonywana jest na tym połączeniu funkcja hash (H). Operacja ta nie jest bezpieczna, ale to co jest wykonywane w algorytmie HMAC to ponowne wzięcie klucza **k**, połączenie go z **opad** (ang. Outer Pad), a następnie przyłączenie do wcześniej wygenerowanego $H(k \oplus ipad \;||\; m)$ <256 bitów> i wykonaniu na tak skonstruowanym 512-bitowym bloku ponownie funkcji HASH (H).



Slajd pokazuje zasadę działania HMAC w sposób schematyczny. Jako pierwszy blok wiadomości tworzone jest połączenie klucza z wartością ipad. Następuje przetworzenie łańcucha Merkle-Damgard'a. Wyjście łańcucha jest podane na wejście innego łańcucha zainicjowanego wartością **k⊕opad**.

Z podobną konstrukcją już się spotkaliśmy. Jeśli potraktujemy pierwsze wykonania funkcji h w każdym z łańcuchów jako generowanie kluczy, oraz funkcję h jako generator PRF to otrzymamy konstrukcję NMAC. Jedyna różnica polega na tym, że oba klucze są generowane z jednego bazowego i są od siebie zależne. Ważną własnością jest fakt, że funkcje h tworzą PRF nawet z kluczy ze sobą powiązanych.

Właściwości HMAC

Zbudowany z implementacji SHA-256 potraktowanej jako „czarna skrzynka”.

HMAC jest uznawany jako bezpieczna PRF

- Może to być udowodnione pod warunkiem założenia właściwości PRF dla funkcji h(.,.)
- Ograniczenia bezpieczeństwa są podobne, jak dla NMAC
  - Potrzebujemy aby wyrażenie $q^2/|T|$ pozostawało pomijalnie małe, przy założeniu, że ($q << |T|^{½}$)

W protokole TLS: musi być wspierane HMAC-SHA1-96

14

Konstrukcja HMAC jest uznawana za bezpieczną. Nie potrzeba zmieniać klucza dopóki liczba wiadomości jest znacznie mniejsza od pierwiastka kwadratowego z ilości wszystkich możliwych tagów. W przypadku SHA-256 liczba wiadomości ma być mniejsza od 2^{128}. To w praktyce oznacza, że można korzystać z tej konstrukcji bez ograniczeń z zachowaniem bezpieczeństwa.

Standard TSL zakłada, że w jego implementacjach musi być zastosowane HMAC-SHA1-96, czyli konstrukcja HMAC zbudowana w oparciu o SHA1 z wyjściem „przyciętym” do 96 bardziej znaczących bitów (SHA1 ma wyjście 160 bitowe). Pamiętamy, że SHA1 nie jest już uznawane za bezpieczne, więc dlaczego jest stosowane w HMAC? Otóż w tej konstrukcji funkcja h nie musi być odporna na kolizje, ma mieć tylko właściwość PRF (przypomnijmy sobie, w jaki sposób badane były właściwości konstrukcji NMAC).

Ostrzeżenie: atak czasowy w trakcie weryfikacji [L'09]

Przykład: biblioteka Keyczar (Python) [uproszczone]

```
def Verify(key, msg, sig_bytes):
    return HMAC(key, msg) == sig_bytes
```

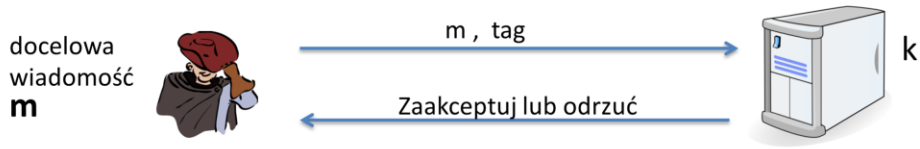
Problem: '==' zaimplementowane jako porównanie bajt po bajcie

- Komparator zwraca błąd kiedy zostanie napotkana pierwsza niezgodność

15

Rozważmy możliwe ataki na HMAC. Przedmiotem rozważań będzie biblioteka Keyczar napisana w języku Python. Wejściami funkcji Veryfi(.) są klucz, wiadomość i tag. Weryfikacja polega na przeliczeniu tagu z wiadomości **m** obliczonego z zastosowaniem klucza **key** za pomocą algorytmu HMAC. Wyliczone 16 bajtów jest porównywane z otrzymanym z danymi tagiem **sig_bytes**.
Rozwiązanie wydaje się poprawne, zwłaszcza, jeśli popatrzymy na nie z punktu widzenia matematycznego. Problem polega na zasadzie działania operatora '=='. Porównywanie odbywa się bajt po bajcie w pętli. Jeśli wykryta zostanie niezgodność na którymś z analizowanych bajtów, to funkcja zwraca błąd.
Zastosowanie takiej metody porównywania pozwala na atak czasowy.

Ostrzeżenie: atak czasowy w trakcie weryfikacji [L'09]



Atak czasowy: aby obliczyć tag dla docelowej wiadomości m wykonuj:

Step 1: Wyślij wiadomość z losowym tagiem

Step 2: Wyślij tę samą wiadomość i jej tag wiele razy za każdym razem zmieniając tylko pierwszy bajt tagu, tak aby „przejść" po wszystkich możliwych wariantach tego pierwszego bajtu.

zatrzyma je się, gdy weryfikacja będzie trwała trochę dłużej niż w pierwszym kroku

Step 3: powtarzaj analizę dla wszystkich bajtów tagu do momentu odgadnięcia całego tagu

16

Zakładamy, że jesteśmy atakującymi i mamy wiadomość **m**, dla której chcemy otrzymać poprawny tag. Naszym celem jest serwer, na którym działa HMAC z ukrytym kluczem **k**. Zakładamy następujące działanie serwera. Pobiera on parę (wiadomość, tag). Jeśli tag jest prawidłowy, to wiadomość jest przetwarzana, jeśli nie to serwer zgłasza do nadawcy błąd.
Atakujący może więc wysłać do serwera wiele razy tę samą wiadomości i spróbować wydedukować jej tag.
Sposób jest bardzo prosty. Cały czas wysyłamy tę samą wiadomość, ale z różnymi tagami. Atak zaczyna się od wysłania wiadomości z dołączonym do niej losowym tagiem. Mierzy się czas potrzebny serwerowi do wygenerowania odpowiedzi. Następnie wysyłamy za każdym razem tę samą wiadomość i wszystkimi możliwościami pierwszego bajta tagu. W pewnym momencie zauważymy, że czas weryfikacji był nieco dłuższy. Oznacza to, że zgadliśmy pierwszy bajt tagu. Zatrzymujemy wysyłanie wiadomości, przyjmujemy, że znamy pierwszy bajt tagu. Rozpoczynamy więc wysyłanie par wiadomość i tag, ale w niej pierwszy bajt nie ulega już zmianom, za to rozważane są wszystkie możliwości drugiego bajtu. Znowu czekamy na chwilę, gdy odpowiedź będzie trwała odrobinę dłużej. Wtedy odgadujemy drugo bajt tagu dla naszej wiadomości. Czynność przeprowadzamy do momentu otrzymania informacji, że wiadomość została poprawnie zweryfikowana…
Pokazany atak ma Wam uświadomić, jak ważna jest nawet sama implementacja konstrukcji kryptograficznej.

Obrona nr 1

Wymuś działanie części kodu porównującej łańcuchy w taki sposób, aby zawsze działała tak samo długo (Python) :

```
return false if  sig_bytes  has wrong length
result = 0
for x, y in zip( HMAC(key,msg) , sig_bytes):
    result |= ord(x) ^ ord(y)
return result == 0
```

Może być trudne do zapewnienia ze względu na działania optymalizacyjne kompilatora.

17

Pierwszym sposobem na obronienie się przed omówionym wcześniej atakiem jest zorganizowanie kodu porównującego łańcuchy bajtów tak, aby zawsze porównanie trwało tyle samo. Podany przykład w języku Python pochodzi już z zaktualizowanej biblioteki Keyczar, w której zaimplementowano ochronę na atak czasowy.
W pierwszym etapie działania kodu następuje sprawdzenie, czy długość nadesłanego tagu ma właściwą wartość (co czasami się zdarza w przypadku konstruowania ataków). Zapewnienie tego samego czasu porównywania łańcuchów odbywa się przez wprowadzenie własnego jawnego kodu zastępującego operator porównania. W rozwiązaniu zastosowano funkcję zip, która generuje listę par bajtów pobranych z odpowiednio pierwszego i drugiego argumentu funkcji (zip nie jest tu algorytmem kompresującym, a tylko tworzącym nową strukturę danych z kolejnych par bajtów słów wejściowych). W pętli za każdym razem pobierana jest jedna para, za pomocą funkcji ord znaki tekstu są przekształcane na ich reprezentacje liczbowe. Na parze bajtów wykonywana jest operacja xor i dodawana bitowo (or bitowe) do wartości result. Jeśli kiedykolwiek w którejś z par operacja xor da wynik inny niż 0, to oznacza, że napotkano różnicę pomiędzy wygenerowanym tagiem a otrzymanym. Jeśli na otrzymanej liczbie wykonamy operację or bitowe z wartością result, początkowo ustawioną na 0, to do końca przeprowadzania obliczeń wykrycie różnicy choćby w jednym bicie zostanie w wartości result zapamiętane. Ostatecznie funkcja porównuje wartość result z 0 i odpowiada, czy tagi były identyczne, czy nie. Za każdym razem w takiej konstrukcji czas porównywania łańcuchów bajtów jest identyczny i nie można przeprowadzić na nią ataku czasowego.
Problemem w takim rozwiązaniu może być jednak kompilator, który może „spróbować” zoptymalizować opracowany kod, tak, żeby skrócić jego działanie zatrzymując działanie pętli wcześniej, co może w konsekwencji zrujnować nasze wysiłki ukrycia czasu prowadzonych obliczeń.

Obrona nr 2

Inny sposób zapewnienia takiego samego czasu obliczeń (Python) :

```
def Verify(key, msg, sig_bytes):
    mac = HMAC(key, msg)
    return HMAC(key, mac) == HMAC(key, sig_bytes)
```

Atakujący nie zna porównywanych wartości.

18

Innym sposobem na obronę jest ukrycie przed atakującym informacji, które łańcuchy bajtów są porównywane. Jest to rozwiązanie używane rzadziej.
Mamy tutaj algorytm weryfikujący, na którego wejście podawany jest klucz, wiadomość i tag. W pierwszym kroku obliczamy MAC dla wiadomości. Następnie obliczamy jeszcze raz funkcje hash, ale tym razem z obliczonego MAC i z otrzymanego MAC z wiadomością. Obliczone wartości porównujemy operatorem „==”. Obliczone funkcje hash z hash, jeśli tagi są identyczne, też powinny być identyczne. Tutaj porównujemy bajt po bajcie łańcuchy po przekształceniu i atakujący nawet jeśli znajdzie różnice w czasie, to nie odpowiadają one kształtowi tagu obliczonego dla wiadomości.
To podejście nie skazuje nas na „miłosierdzie” algorytmów optymalizujących kompilatora.

Z tych rozważań wynika jeszcze jedna lekcja. Nawet eksperci od kryptografii tworzący biblioteki potrafią popełnić błędy. Kod realizujący precyzyjnie opracowaną konstrukcję kryptograficzną, ale jest podatny na ataki czasowe kompletnie ruminuje bezpieczeństwo systemu. Jeszcze raz zachęcam do stosowania dobrze „prześwietlonych” i sprawdzonych bibliotek kryptograficznych, takich jak OpenSSL. Na marginesie aktualna wersja biblioteki Keyczer także uznawana jest już za „przyzwoitą” i na pewno jej zastosowanie ograniczy wrażliwość Waszych systemów na tego typu ataki.