Criptografie și Securitate

- Prelegerea 15.1 -
Atacuri de timp pentru MAC-uri

Adela Georgescu, Ruxandra F. Olimid

Facultatea de Matematică și Informatică
Universitatea din București

Cuprins

Atac de timp

- Prezentăm un atac general care afectează multe implementări ale algoritmilor MAC;

Atac de timp

- Prezentăm un atac general care afectează multe implementări ale algoritmilor MAC;
- Studiem verificarea unui HMAC implementată în libraria criptografică KeyCzar (Python);

Atac de timp

- Prezentăm un atac general care afectează multe implementări ale algoritmilor MAC;
- Studiem verificarea unui HMAC implementată în libraria criptografică KeyCzar (Python);
- Codul de mai jos verifică un tag generat de HMAC

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

Atac de timp

- Prezentăm un atac general care afectează multe implementări ale algoritmilor MAC;
- Studiem verificarea unui HMAC implementată în libraria criptografică KeyCzar (Python);
- Codul de mai jos verifică un tag generat de HMAC

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

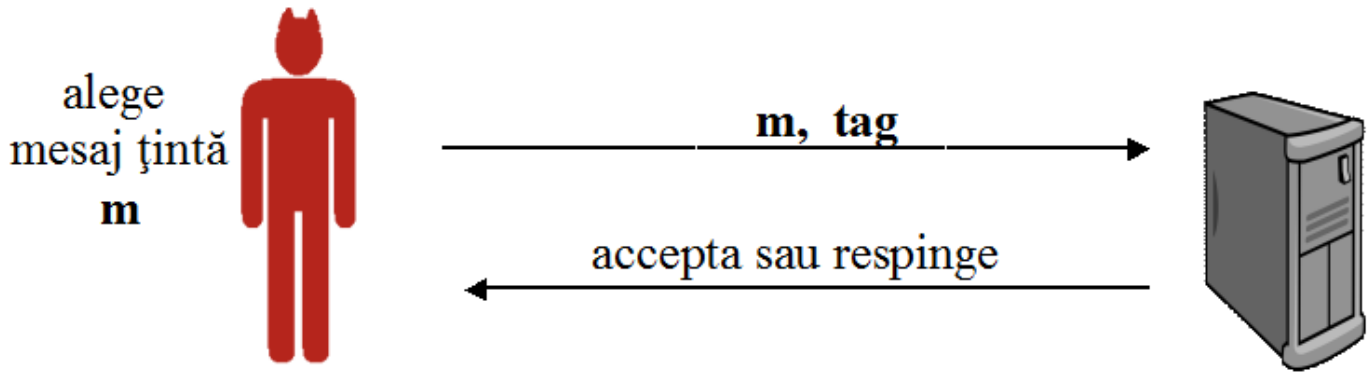
- Problemă: comparația "==" se face octet cu octet și întoarce *false* atunci când prima inegalitate este găsită;

Atac de timp

- Scopul atacului: Adversarul $\mathcal{A}$ vrea să calculeze un tag pentru un anume mesaj m

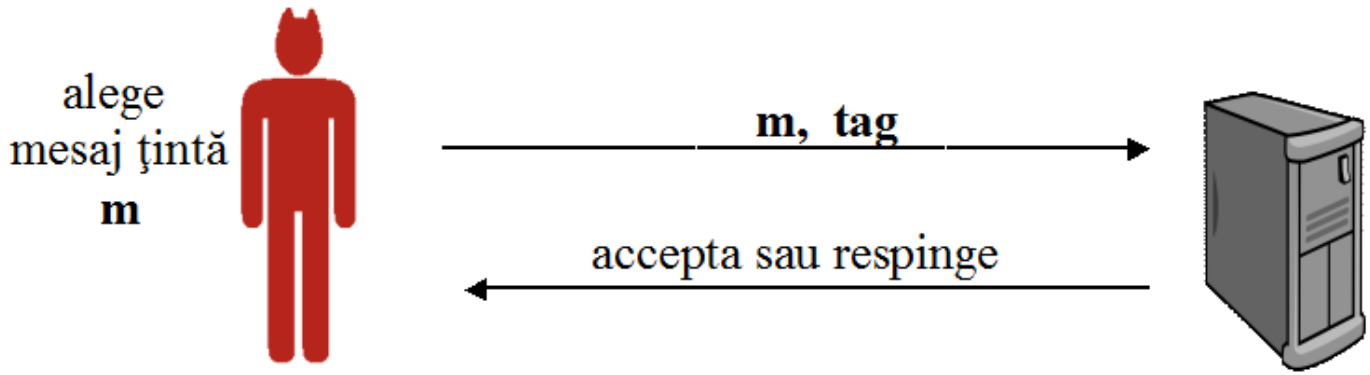
Atac de timp

- Scopul atacului: Adversarul $\mathcal{A}$ vrea să calculeze un tag pentru un anume mesaj m



Atac de timp

- Scopul atacului: Adversarul $\mathcal{A}$ vrea să calculeze un tag pentru un anume mesaj m



- $\mathcal{A}$ trimite multe cereri către server pentru același mesaj m și tag-uri diferite.

Atac de timp

Pașii atacului:

Atac de timp

Pașii atacului:

1. $\mathcal{A}$ trimite cerere către server pentru m si un tag aleator $t = B_1\ B_2 ... B_n$ (t este reprezentat pe octeți) și măsoara timpul până server-ul răspunde;

Atac de timp

Pașii atacului:

1. $\mathcal{A}$ trimite cerere către server pentru m si un tag aleator $t = B_1\ B_2 ... B_n$ (t este reprezentat pe octeți) și măsoara timpul până server-ul răspunde;

2. $\mathcal{A}$ trimite cerere către server pentru fiecare $t_i{}' = i\ B'_2 ... B'_n$ pentru $i = 1, 2, 3...$ ($B'_2, ..., B'_n$ sunt octeti arbitrari);

Atac de timp

Pașii atacului:

1. $\mathcal{A}$ trimite cerere către server pentru m si un tag aleator $t = B_1\ B_2 ... B_n$ (t este reprezentat pe octeți) și măsoara timpul până server-ul răspunde;

2. $\mathcal{A}$ trimite cerere către server pentru fiecare $t_i' = i\ B_2' ... B_n'$ pentru $i = 1, 2, 3...$ ($B_2', ..., B_n'$ sunt octeti arbitrari);

3. $\mathcal{A}$ se oprește când timpul de verificare e puțin mai mare decât la pasul 1;

Atac de timp

Pașii atacului:

1. $\mathcal{A}$ trimite cerere către server pentru m si un tag aleator $t = B_1\ B_2 ... B_n$ (t este reprezentat pe octeți) și măsoara timpul până server-ul răspunde;

2. $\mathcal{A}$ trimite cerere către server pentru fiecare $t_i' = i\ B_2' ... B_n'$ pentru $i = 1, 2, 3...$ ($B_2', ..., B_n'$ sunt octeti arbitrari);

3. $\mathcal{A}$ se oprește când timpul de verificare e puțin mai mare decât la pasul 1;

4. In acest caz, a găsit primul octet și continuă atacul pentru următorii octeți, pe rând, până când găsește tag-ul valid.

Soluții posibile

- Soluția nr. 1: comparația pe cele două stringuri trebuie să necesite mereu același timp:

```
def Verify(key,msg,tag_bytes):
   if len(tag_bytes) != len(correct_tag) return false;
   result = 0
   for x, y in zip( HMAC(key,msg), tag_bytes):
      result |= ord(x) ^ ord(y)
   return result == 0
```

Soluții posibile

- Soluția nr. 2: comparația se face pe valorile HMAC ale celor 2 stringuri - adversarul pierde accesul direct la valorile comparate:

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

Important de reținut!

- Nu folosiți propriile voastre implementări criptografice, ci doar pe cele standardizate!