Praktyczna kryptografia: Hashowanie, podpisy cyfrowe i wyprowadzanie klucza

W dzisiejszych czasach bezpieczeństwo każdej istniejącej aplikacji internetowej silnie polega na skrótach kryptograficznych. Co więcej, znalezienie praktycznej metody ich odwracania wywołałoby globalny kryzys w całej informatyce. To argument, który sprawia, że warto się z nimi zapoznać. W tym artykule pokażemy, jak poprawnie używać hashy oraz jak skonstruowane są algorytmy HMAC i PBKDF2.

WPROWADZENIE: HASHOWANIE

Artykuł ten kierowany jest głównie do osób, które posiadają podstawowe doświadczenie z używaniem hashy kryptograficznych. W tym wprowadzeniu skupimy się na krótkim przypomnieniu podstawowych właściwości funkcji skrótu (ang. *hash*).

Hash kryptograficzny jest jednokierunkową funkcją jednego argumentu o następującej sygnaturze: $bytearray\ \ hash(bytearray\ input)$, gdzie wejście jest jakimś ciągiem bajtów o arbitralnej długości, a wyjście – ciągiem bajtów o stałej długości (np. 128 bitów w przypadku MD5). Funkcje te są deterministyczne (zawsze zwracają to samo dla tego samego wejścia) i cechują się tzw. efektem lawinowym – zmiana zaledwie jednego bitu na wejściu powoduje diametralną zmianę wartości całego hasha:

```
>>> hashlib.sha256('bezpieczenstwo').hexdigest()
'8c133246c0c91fc9df1f7f25e475d1e2626362cf0492df8015a79b847da196e4'

# dwukrotne obliczenie hasha tej samej wartości daje ten sam
wynik
>>> hashlib.sha256('bezpieczenstwo').hexdigest()
'8c133246c0c91fc9df1f7f25e475d1e2626362cf0492df8015a79b847da196e4'

# ale zmiana chociaż jednego znaku kompletnie go modyfikuje
>>> hashlib.sha256('bezpieczenstwa').hexdigest()
'd8ca11a4a877464d99218e84b28cd84371901a433edb93d8fac50e565ab6e122'
```

Co najważniejsze, możliwe jest szybkie obliczenie wartości funkcji $hash(x)$, natomiast funkcja $hash^{1}(x)$ (wyliczenie oryginalnej wiadomości z hasha) jest zbyt kosztowna obliczeniowo, aby zrealizować ją w praktyce. Mówi się również, że hash jest funkcją pseudolosową (PRF), albowiem w przypadku idealnej funkcji wyniki nie powinny być możliwe do odróżnienia od czystej losowości. Dodatkowo funkcje skrótu powinny uniemożliwiać wygenerowanie tzw. kolizji, czyli dwóch różnych wiadomości, dla których funkcja skrótu daje tą samą wartość hasha.

To tyle, jeśli chodzi o przypomnienie tematu funkcji skrótu, przejdźmy teraz do omówienia popularnej abstrakcji nad nimi, czyli podpisów elektronicznych.

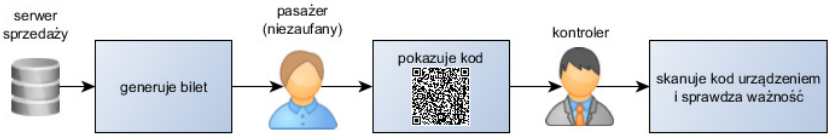
CZĘŚĆ I: PODPISY ELEKTRONICZNE

Protokoły oparte na podpisach

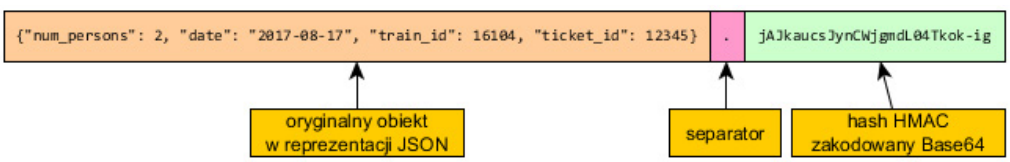
…w dzisiejszych czasach wykorzystujemy je niemal wszędzie. Jeden z intuicyjnych przykładów mogą stanowić chociażby elektroniczne bilety na pociąg (Rysunek 1). Jak to działa?

Załóżmy, że kod generowany dla użytkownika zawiera wszystkie kluczowe informacje, takie jak data i godzina odjazdu, stacja początkowa, końcowa, liczba osób itp. Dlaczego systemu nie da się oszukać? Prawidłowo zaprojektowane rozwiązanie tego typu wykorzystuje podpisy cyfrowe, najczęściej oparte albo na stosunkowo prostej technice „współdzielonego sekretu" (ang. *shared secret*), albo na kryptografii asymetrycznej[1]. Pierwsza, mniej skomplikowana metoda wymaga tego samego klucza do wystawiania i do weryfikowania podpisów, natomiast druga posługuje się dwoma osobnymi kluczami służącymi do przeprowadzania tych czynności.

1. Nieprawidłowo zaprojektowane rozwiązanie tego typu wykorzystuje szyfrowanie. W poprzednim numerze „Programisty" w artykule „Praktyczna kryptografia: Szyfry blokowe" pokazywaliśmy, jak można łamać systemy tego typu na wszelkie możliwe sposoby.



Rysunek 1. Schemat działania biletów elektronicznych



Rysunek 2. Wynik działania funkcji Serializer.dumps

W tej części skupimy się na podpisach ze „współdzielonym sekretem", które są stosunkowo proste i szybkie w użyciu (o ile wiemy, jak to zrobić dobrze!), jednocześnie zapewniając satysfakcjonujący poziom bezpieczeństwa w wielu zastosowaniach.

Jak zastosować taki podpis w naszej aplikacji? Python posiada do tego godny pochwały moduł `itsdangerous`:

Listing 1. Wykorzystanie itsdangerous[2] do podpisywania obiektu biletu (po stronie serwera wystawiającego bilet)

```
from itsdangerous import Serializer
s = Serializer('secret-key-123')
s.dumps({'ticket_id': 12345, 'train_id': 16104, 'date': '2017-
08-17', 'num_persons': 2})
# wynik: '{"num_persons": 2, "date": "2017-08-17", "train_id":
16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'
```

Co się stało? Utworzyliśmy obiekt `Serializer`, przekazując mu sekretny klucz `secret-key-123`. Następnie przekazaliśmy obiekt biletu do metody `dumps`, która skonwertowała go na JSON, a następnie wynikowy ciąg znaków podpisała. Podpis został doklejony na końcu tekstowej reprezentacji obiektu (Rysunek 2).

Następnym krokiem byłoby wygenerowanie kodu QR zawierającego podpisany bilet i przesłanie go do użytkownika, ale tę część intencjonalnie pominiemy (istnieją do tego odpowiednie biblioteki).

Czytnik biletów, po odczytaniu tekstu znajdującego się w kodzie QR, powinien wykonać następujące czynności:

Listing 2. Wykorzystanie itsdangerous do odczytania obiektu biletu wraz z weryfikacją poprawności podpisu

```
>>> from itsdangerous import Serializer

>>> # wartosc odczytana z kodu QR
>>> ticket = '{"num_persons": 2, "date": "2017-08-17", "train_
id": 16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'

>>> s = Serializer('secret-key-123')
>>> s.loads(ticket)
{'num_persons': 2, 'train_id': 16104, 'ticket_id': 12345,
'date': '2017-08-17'}
```

Co dokładnie zrobiła metoda `loads`? Ciąg znaków przekazany do niej został rozbity na dwa pod-ciągi (`message` oraz `signature`), odpowiednio: od początku do ostatniej kropki w ciągu oraz od ostatniej kropki do końca. Następnie wykonane zostało sprawdzenie, czy `signature` jest prawidłowym podpisem ciągu `message` dla klucza `secret-key-123`. Okazało się, że tak, w związku z tym pod-ciąg `message` został zdeserializowany do postaci obiektu.

Jeżeli w międzyczasie użytkownik dokonał nieautoryzowanej modyfikacji swojego biletu, otrzymamy wyjątek:

Listing 3. Nieautoryzowana modyfikacja użytkownika

```
>>> ticket = '{"num_persons": 7, "date": "2017-08-20", "train_
id": 16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'
>>> s.loads(ticket)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/itsdangerous.py",
  line 582, in loads
    return self.load_payload(self.make_signer(salt).unsign(s))
  File "/usr/local/lib/python3.5/dist-packages/itsdangerous.py",
  line 374, in unsign
    payload=value)
itsdangerous.BadSignature: Signature
b'jAJkaucsJynCWjgmdL04Tkok-ig' does not match
```

2. Warto podkreślić, że istnieje społecznościowy port tego modułu do PHP: https://github.com/mattbasta/itsdangerous-php oraz innych technologii.

Typowym przypadkiem wykorzystania podpisów cyfrowych jest sytuacja, kiedy potrzebujemy przekazać informację z zaufanego punktu A do zaufanego punktu B (tutaj: z serwera biletów do czytnika), ale poprzez niezaufany punkt C (czyli przez potencjalnie złośliwego użytkownika). Olbrzymią zaletą jest tutaj fakt, że weryfikacja poprawności podpisu może nastąpić offline – mobilny skaner biletów nie musi łączyć się z centralą, aby sprawdzić, czy bilet jest ważny.

W analogiczny sposób można zaimplementować kody rabatowe, kwity magazynowe, znaczki pocztowe i wiele innych rozwiązań.

Wiemy już, jak działają podpisy, widzieliśmy, jak działa biblioteka do podpisywania… pora zaimplementować to samodzielnie (np. w naszym nowym rewolucyjnym frameworku).

Hash to nie uwierzytelnianie, czyli dlaczego MD5(secret || data) to błąd

Posłużmy się teraz nieco prostszym przykładem: serwer A wystawia token upoważniający do zakupu jabłek z rabatem 10% i wysyła go klientowi na maila w postaci linku do serwisu hostowanego przez serwer B. Klient może kliknąć i skorzystać z rabatu. Serwer B zna współdzielony sekret użyty do wygenerowania tokenu, więc bez łączenia się z serwerem A weryfikuje poprawność tokenu znajdującego się w adresie URL i przyznaje rabat. Ruch użytkowników jest tak duży, że takie rozwiązanie jest znacznie bardziej korzystne niż utrzymywanie aktywnego połączenia pomiędzy serwerami A i B.

Jak stworzyć najbardziej intuicyjną i zarazem błędną implementację takiego rozwiązania?

Listing 4. Błędna implementacja podpisu cyfrowego za pomocą hasha (PHP)

```
$secret = 'pH2ef4OhUdec3eSPdrujAy4gdec3eSPdrudec3eSPdru';
$promotion = 'name=rabat_na_jablka&amount=10'; // rabat 10%
$token = md5($secret . ':' . $promotion);

echo('Link promocyjny: http://serwer_b/rabat/' . $promotion .
'/' . $token);
```

Sekret o takiej długości jest praktycznie niemożliwy do złamania, używamy funkcji hashującej, gdzie jest problem?

Pomijając wykorzystanie przestarzałego algorytmu MD5 (nigdy nie róbmy tego w nowych projektach!), jest to ewidentny przykład błędnego zrozumienia i błędnego wykorzystania prymitywu kryptograficznego, jakim jest funkcja hashująca. Czym w ogóle jest funkcja hashująca? Odwołajmy się do definicji.

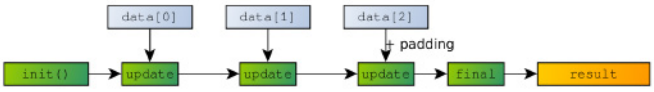
Kryptograficzna funkcja hashująca musi:

- powodować, że próba wygenerowania wiadomości mającej wybraną wartość hasha jest niepraktyczna (tzw. *pre-image resistance*);
- powodować, że próba znalezienia dwóch różnych wiadomości z tą samą wartością hasha jest niepraktyczna (tzw. *collision resistance*);
- być szybka do policzenia dla dowolnej zadanej wiadomości (z powodów praktycznych; wyłączając niektóre przypadki, gdzie kosztowność obliczania hasha jest pożądana, np. hashowanie haseł).

Funkcja skrótu sprawdza się świetnie, jeżeli naszą intencją jest sprawdzenie integralności pliku (np. pobraliśmy go od innej osoby, która wcześniej wysłała nam SMSem hash SHA256). Jeżeli po pobraniu pliku policzymy hash i będzie on pasował do tego, który

otrzymaliśmy, to możemy być niemal pewni, że plik został pobrany prawidłowo, w całości i nie został w trakcie przez kogokolwiek zmodyfikowany. Wiele osób uważa, że w takim razie można wykorzystać je również bezpośrednio do podpisywania wiadomości, ale podejście to jest błędne – jak się zaraz przekonamy.

Funkcje hashujące MD5, SHA-1 i SHA-2 zbudowane są przy użyciu konstrukcji Merklego–Damgårda, co sprawia, że wewnętrznie operują one na N-bajtowych blokach. Wykonywanie takiej funkcji hashującej zawsze rozpoczyna się od pewnego stanu początkowego, który jest ustalony na stałe w implementacji, a początkowo hashowanie dotyczy jedynie pierwszych N bajtów, a w wyniku tego zmienia się wewnętrzny stan funkcji hashującej. Dalej oddzielnie konsumowane i hashowane jest kolejne N bajtów wejścia, tym razem już z nowym stanem utworzonym przez poprzednią operację, i tak aż do momentu przetworzenia całej wiadomości.



Rysunek 3. Uproszczony schemat działania konstrukcji Merklego–Damgårda

Co jednak najważniejsze, wynikowy hash eksponuje cały stan końcowy, który został wyliczony w wyniku przetworzenia wiadomości. Nic nie powstrzymuje nas przed załadowaniem tego stanu z powrotem i doliczeniem dodatkowego bloku do istniejącego hasha.

Powróćmy do wspomnianego przykładu z rabatem. Zamiast rabatu 10% zażyczymy sobie rabat 1000%. Nie ma konieczności łamania hasha ani sekretu. Możemy dodać kolejny blok do wiadomości i wyliczyć kolejny stan, w jakim znajdzie się funkcja hashująca.

Wykorzystajmy poprzedni przykład:

```
>>> hashlib.md5('pH2ef4OhUdec3eSPdrujAy4gdec3eSPdrudec3eSPdru' +
                ':name=rabat_na_jablka&amount=10').hexdigest()
'8b990fdf27e79353a7fadf639e5fa7c4'
```

W tym przypadku znany użytkownikowi podpis to `'8b990fdf27e79353a7fadf639e5fa7c4'`, zna też treść podpisanej wiadomości, nie zna natomiast sekretu. Atak automatyzuje narzędzie hashpump:

```
$ hashpump -d ':name=rabat_na_jablka&amount=10' -s
'8b990fdf27e79353a7fadf639e5fa7c4' -k 44 -a '&amount=100'
ef7ca4ca05a171525a668d23ac6c639a
:name=rabat_na_jablka&amount=10\x80\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00X\x02\x00\x00\x00\x00\x00\
x00&amount=100
```

Druga linia wyniku oznacza naszą nową wiadomość, a pierwsza podpis dla niej. Łatwo sprawdzić, czy atak byłby skuteczny:

```
>>> hashlib.md5('pH2ef4OhUdec3eSPdrujAy4gdec3eSPdrudec3eSPd
ru:name=rabat_na_jablka&amount=10\x80\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00X\x02\x00\x00\x00\x00\x00\
x00&amount=100').hexdigest()
'ef7ca4ca05a171525a668d23ac6c639a'
```

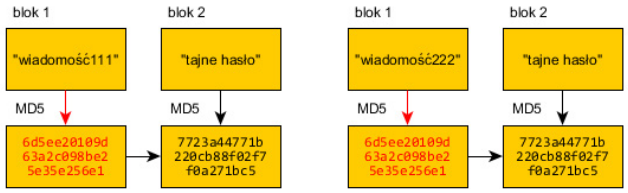
Faktycznie, podpis się zgadza. W ten sposób dokleiliśmy parametr `amount` drugi raz do wiadomości. Z uwagi na specyfikę mechanizmów parsowania parametrów w serwerach webowych wzięta pod uwagę będzie druga wartość i `amount` wyniesie 100.

Trudniejsze do złamania, wciąż niepoprawne: MD5(message || secret)

W tym miejscu czas na pytanie: czy przestawienie kolejnością sekretu oraz wiadomości rozwiąże problem? Mamy na myśli użycie: `hash(message || secret)`[3]. Niestety, nie. Uważny czytelnik po lekturze poprzednich akapitów zauważy, że konstrukcja Merklego-Damgårda umożliwia wystąpienie kolizji nie tyle w obrębie hasha jako całości, co w poszczególnych blokach hashowanej wiadomości.

Znając wiadomość $message$ oraz wartość $hash(message)$, jeżeli znajdziemy wiadomość $message'$, taką że $message \neq message'$ oraz $hash(message) = hash(message' \mid\mid secret)$, to implikuje, że $hash(message \mid\mid secret) = hash(message' \mid\mid secret)$.

Inaczej mówiąc, jeśli atakujący zna jakąś kolizję w funkcji hashującej – czyli zna dwie wiadomości *A* i *B*, dla których wartość hasha jest taka sama – może „namówić" serwer do podpisania wiadomości *A* i będzie od razu znał poprawny podpis dla wiadomości *B*. Nie brzmi to równie źle jak problem z poprzedniego podpunktu.



Rysunek 4. Demonstracja kolizji na poziomie pierwszego bloku

Jak poważny jest to atak? Po pierwsze, osłabia siłę podpisu z 2^n (pełny atak siłowy na hash) do $2^{n/2}$ (atak urodzinowy w celu wygenerowania kolizji). Po drugie, każda znaleziona kolizja może zostać wykorzystana przeciwko nam. Na przykład firma Google niedawno opublikowała pierwszą znaną kolizję na algorytm SHA-1 – oznacza to, że teraz każdy może wykorzystać tę kolizję do przeprowadzenia ataku na taki mechanizm podpisywania.

Istnieje również bardziej praktyczny atak. Jeżeli znamy całą wiadomość oprócz sekretu, a jej długość jest wielokrotnością rozmiaru bloku, możliwe jest obliczenie wewnętrznego stanu, który zostanie osiągnięty po zhashowaniu wiadomości, ale przed zhashowaniem sekretu. Problem redukuje się wtedy do brute-force'owania funkcji `MD5'(secret)` z innym stanem początkowym, a to oznacza, że taki hash można łamać bez żadnego dodatkowego narzutu. Osiem kart Nvidia GTX 1080 jest w stanie łamać MD5 z szybkością 200 miliardów hashy na sekundę, co w praktyce oznacza, że ośmioznakowy sekret składający się z małych liter i cyfr da się na tym sprzęcie złamać w niecałe 6 sekund!

CTF

Czytelników zachęcamy do samodzielnego spróbowania swoich sił z włamywaniem się do aplikacji zawierającej podobne błędy: https://var.tail-call.net/hmac

Prawidłowe rozwiązanie: HMAC

Czas na przedstawienie prawidłowego rozwiązania. Do podpisywania dedykowane są tzw. Message Authentication Codes (MAC),

3. Podobny błąd w implementacji popełnili niektórzy polscy operatorzy serwisów z płatnościami online. Błędów tych nie da się łatwo wykorzystać wyłącznie przez szczęśliwy zbieg okoliczności (protokoły wymuszają dodatkowe sprawdzenia danych, więc „złośliwe" operacje zostaną prędzej czy później wykryte i odrzucone).

które gwarantują prawidłowe uwierzytelnienie. Istnieje również podzbiór MACów skonstruowanych na bazie funkcji skrótu, czyli HMAC. Są to algorytmy zaprojektowane z myślą o podpisywaniu, które mają wyraźnie sprecyzowane osobne argumenty wiadomości oraz sekretu (hasła).

Operatory

Operatory i funkcje pomocnicze:

- $a \oplus b$ – wynik xora bitowego a i b;
- $a \parallel b$ – złączenie ze sobą ciągów a i b (długość wynikowego ciągu jest sumą długości a i b);
- *int*32*BigEndian*(*int*) – liczba zapisana w postaci `uint32_t`, poczynając od najbardziej znaczącego bajtu.

Przyjrzyjmy się wewnętrznej konstrukcji HMAC, która nie jest skomplikowana:

$$HMAC(hash, message, secret) = hash\left((secret \oplus opad) \parallel hash((secret \oplus ipad) \parallel message)\right)$$

$$opad = 0x5c5c5c \ldots 5c5c$$

$$ipad = 0x363636 \ldots 3636$$

…gdzie *opad* i *ipad* oznaczają kolejno *outer* i *internal padding* i są to stałe o długości takiej, jak długość bloku używanego hasha. Stałe są dobrane w taki sposób, aby dwie wersje klucza: $secret \oplus opad$ oraz $secret \oplus ipad$ miały jak najmniej wspólnych bitów. Z punktu widzenia HMAC ważne jest, aby dwie wersje klucza różniły się między sobą przynajmniej jednym bitem.

„Zagnieżdżona" konstrukcja HMACa (hash z hasha) oraz manipulacje z kluczem dodatkowo zabezpieczają używany schemat hashowania przed wspomnianymi wcześniej atakami związanymi z kolizjami. Inaczej mówiąc: używanie tego schematu istotnie łagodzi konsekwencje, które mogą wystąpić w wyniku znalezienia słabości w funkcji hashującej znajdującej się pod spodem.

Prawidłowe konstruowanie payloadu do HMACa

Do tego momentu omawialiśmy metody prawidłowego podpisywania jakiegoś abstrakcyjnego ciągu *message*, który zgodnie z definicją musi być niczym innym jak po prostu ciągiem bajtów. To zadowala nas od strony teoretycznej, ale w niektórych zastosowaniach wyjątkowo rzadko będziemy w posiadaniu bufora, który można by natychmiastowo podpisać.

Wróćmy do poprzedniego przykładu z biletami kolejowymi, gdzie naszą intencją jest podpisanie typu złożonego (obiektu, słownika). Programiści często są zmieszani faktem, że znana im literatura nie wspomina, jak prawidłowo przygotować „wsad" do funkcji HMAC, bowiem skupia się ona wyłącznie na wewnętrznej budowie funkcji.

Nieprawidłowe przygotowanie ciągu wejściowego stanowi świetny sposób na wytworzenie nowych luk bezpieczeństwa, które umożliwią ominięcie całego mechanizmu podpisywania. Przekonała się o tym firma MasterCard[4] [5], kiedy indonezyjski badacz Yohanes Nugroho opracował sposób zmiany kwoty transakcji na 0 złotych, który nie powodował unieważnienia podpisu. Jak to możliwe, przecież używamy superbezpiecznego schematu podpisywania?

Taka sytuacja jest możliwa za każdym razem, kiedy wykorzystamy niejednoznaczny sposób kodowania danych. Uproszczony przykład podobnego ataku:

„Poprzez HTTP POST należy przekazać parametry: «numer_konta», «numer_przedmiotu», «ilosc_przedmiotu», «kwota». Oprócz tego należy dołączyć parametr «podpis», wyliczony zgodnie ze wzorem:

$$\text{HMAC}(\text{sekret},\ \text{str}(\text{numer\_konta}) + \text{str}(\text{numer\_przedmiotu}) + \text{str}(\text{ilosc\_przedmiotu}) + \text{str}(\text{kwota}))$$

… gdzie str() oznacza konwersję wartości na ASCII string, a operator plus oznacza złączenie ze sobą dwóch stringów".

Po wykonaniu tej procedury tworzymy mniej więcej takie żądanie:

```
numer_konta=07290109873664746444176233&numer_
przedmiotu=123&ilosc_przedmiotu=1&kwota=10&podpis=cf23df2207d99
a74fbe169e3eba035e633b65d94
```

… gdzie obliczenie podpisu wygląda tak:

```
HMAC(sekret, b"07290109873664746444176233123110") =
cf23df2207d99a74fbe169e3eba035e633b65d94
```

… ale nic nie stoi na przeszkodzie, aby użytkownik posunął się do następującej manipulacji:

```
numer_konta=07290109873664746444176233&numer_
przedmiotu=123&ilosc_przedmiotu=11&kwota=0&podpis=cf23df2207d99
a74fbe169e3eba035e633b65d94
```

4. http://tinyhack.com/2017/09/05/mastercard-internet-gateway-service-hashing-design-flaw/
5. Jeden z autorów tego artykułu, Michał Leszczyński, w lutym 2015 roku znalazł podobny błąd w implementacji bramki płatności firmy Blue Media; błąd został poprawiony w ciągu kilku godzin.

Tzn. przesunięcia „jedynki" z pola `kwota` do pola `ilosc_przedmiotu`. W związku z tym zamiast 1 przedmiotu za 10 złotych otrzymujemy 11 przedmiotów za 0 złotych. Obliczenie podpisu zgodnie z podaną procedurą:

```
HMAC(sekret, b"0729010987366474644417623312 3110") =
cf23df2207d99a74fbe169e3eba035e633b65d94
```

… daje dokładnie ten sam wynik, ponieważ procedura niejednoznacznie koduje oba żądania w ten sam sposób. Wyciągnąć z tego można następującą konkluzję:

Patrząc jedynie na ciąg bajtów przekazany do funkcji HMAC, powinniśmy być w stanie jednoznacznie odtworzyć oryginalny obiekt, z którego został on wygenerowany. Co prawda nie jest to niezbędne, ale „odwracalność" jest bardziej praktycznym pojęciem niż „jednoznaczność". Prawidłowo zmodyfikowany protokół będzie wyglądał tak:

„Poprzez HTTP POST należy przekazać dwa parametry:

1. «zapytanie» (obiekt JSON zawierający wszystkie pola opisujące transakcję: «numer_konta», «numer_przedmiotu», «ilosc_przedmiotu», «kwota»),
2. «podpis», czyli HMAC obliczony z parametru «zapytanie» reprezentowanego bajtowo jako UTF-8".

Oczywiście nie jest to jedyne poprawne rozwiązanie problemu kodowania obiektów, ale warto je rozważyć – wykorzystujemy tutaj stosunkowo prosty i sprawdzony format JSON, który posiada właściwości wystarczające do zapewnienia bezpieczeństwa podpisu.

Wnioski

Najważniejszy z nich brzmi: „hashowanie to nie uwierzytelnianie". Funkcja hashująca jest jedynie prymitywem kryptograficznym, a zapewniane przez nią bezpieczeństwo ściśle zależy od kontekstu jej użycia. Do podpisywania danych zawsze wykorzystujmy dobrze zbadane (powszechne) odmiany algorytmów (H)MAC.

CZĘŚĆ II: HASŁO UŻYTKOWNIKA A KLUCZ KRYPTOGRAFICZNY

Drugą kwestią, która zostanie poruszona w tym artykule, jest często pomijana różnica pomiędzy hasłem i kluczem. Ponownie posłużymy się tutaj hashami, podpisami, a nawet szyfrowaniem symetrycznym. Pora na kolejny, ilustracyjny przykład intuicyjnego użycia kryptografii:

Projektujemy prosty protokół do bezprzewodowego przesyłania danych. W naszym prostym WLANie potrzebujemy szyfrowania symetrycznego oraz uwierzytelniania. Załóżmy następujący sposób działania: użytkownicy znają współdzielone hasło w postaci ASCII, które jednocześnie jest kluczem szyfrowania dla AES256 (jeżeli hasło ma mniej niż 32 znaki, to jest dopełniane zerami do 32 bajtów). Pakiety są numerowane, aby zapobiec atakom powtórzenia (jeżeli któraś ze stron komunikacji otrzyma dwa razy pakiet z tym samym numerem, to go po prostu ignoruje). Każdy pakiet wysłany w eter jest szyfrowany, a wynikowy szyfrogram dodatkowo jest podpisywany za pomocą HMAC (identycznym kluczem jak ten służący do szyfrowania).

Podsumujmy: pakiety są zaszyfrowane i uwierzytelnione. Korzystamy z silnych i sprawdzonych algorytmów. Gdzie jest problem?

1. Każdy bajt może przyjąć jedną z 256 wartości…

…ale typowy użytkownik będzie wykorzystywał jedynie 36 z nich, a konkretniej kody ASCII odpowiedzialne za małe litery oraz cyfry. Niektóre białe znaki, takie jak `\x00` czy `\x15`, są trudne do wprowadzenia i praktycznie nikt nie zdecyduje się na ich wykorzystanie.

2. Długość typowego hasła wynosi około 8 znaków

Atakujący może założyć, że pozostałe 24 znaków to zera. Powoduje to, że do złamania mamy wyłącznie 8 z 32 znaków hasła.

3. Ataki statystyczne i słownikowe

Dalsza redukcja przestrzeni jest możliwa dzięki zastosowaniu wspomnianych technik. Bardzo pesymistycznie załóżmy, że dzięki temu do sprawdzenia jest dwa razy mniej kombinacji haseł.

Dokonajmy więc obliczenia wynikowej przestrzeni haseł, która ma rozmiar:

$$\frac{36^8}{2} = 1410554953728$$

Dla komputera jest to bardzo niewiele. Jeden rdzeń procesora AMD Opteron jest w stanie zrealizować 10^8 deszyfrowań AESa na sekundę. Atakujący dysponujący takim rdzeniem byłby więc w stanie wykonać miliard deszyfrowań w czasie poniżej 11 sekund. W praktyce atak zająłby więcej czasu (z uwagi na dodatkowe operacje, jak key scheduling, i sprawdzanie, czy deszyfrowanie się udało), ale ciągle jest wykonalny, nawet na zwykłym domowym sprzęcie.[6]

Tak naprawdę jest jeszcze gorzej. Jeżeli zamiast łamać szyfrogram AESa, spróbujemy tego samego z podpisem HMAC-SHA256, dobry GPU jest w stanie policzyć około 2.9 miliarda hashy na sekundę. Ponieważ jeden HMAC składa się z dwóch zagnieżdżonych hashy, w ciągu niecałej sekundy, dysponując takim sprzętem, da się złamać jedno hasło.[7]

Jak widać, zaproponowany sposób działania sieci WLAN jest implementowalny, ale z punktu widzenia bezpieczeństwa nie ma żadnego sensu. W tym momencie konieczne jest podjęcie decyzji o „spowolnieniu" któregoś z elementów całej machiny. Przeanalizujmy:

Nie możemy wykonywać żadnych kosztownych obliczeniowo transformacji na oryginalnej wiadomości, która jest szyfrowana i podpisywana, ponieważ obie strony komunikacji będą musiały wykonywać intensywne obliczenia związane z każdym wysłanym lub otrzymanym pakietem, a to w nieakceptowalny sposób obniży przepustowość sieci.

Analogiczne konsekwencje miałoby wprowadzenie iterowanego szyfrowania/podpisywania, albo jakakolwiek inna operacja wprowadzająca istotny „koszt" związany z wysłaniem/odczytaniem pakietu.

Można natomiast wykonać bardzo kosztowną obliczeniowo operację na haśle, a jej wynik uznać za klucz do szyfrowania. Oficjalny termin opisujący taką czynność brzmi „wyprowadzanie klucza w oparciu o hasło" (*ang. Password-Based Key Derivation*).

Korzystne właściwości wynikające z takiego wyprowadzenia:

» dobra transformacja powinna spowodować, że wszystkie wynikowe bity wyprowadzonego klucza będą w tym samym

6. https://security.stackexchange.com/a/9083
7. https://gist.github.com/epixoip/a83d38f412b4737e99bbef804a270c40

stopniu pseudolosowe, co powoduje, że atak brute force przeciwko kluczowi jest całkowicie niepraktyczny;
» nawet jeżeli sieć używa słabego hasła, atakujący, chcąc wykonać atak brute force przeciwko hasłu, musi każdorazowo przeprowadzać bardzo kosztowną operację wyprowadzenia klucza, co obniża prędkość łamania hasła o kilka rzędów wielkości;
» funkcje z możliwością regulacji kosztu będą użyteczne również w przyszłości – wraz ze wzrostem mocy obliczeniowej możliwe będzie podwyższenie tego współczynnika, aby uzyskać ten sam poziom ochrony, co kiedyś.

CTF

Jeżeli rozumiecie błąd wynikający z takiej implementacji protokołu do szyfrowania pakietów, możecie spróbować go złamać samodzielnie: https://var.tailcall.net/wlansec

Wyprowadzanie klucza

Jak można taką funkcję utworzyć? Przeanalizujmy, jak działa algorytm PBKDF2[8] wynaleziony przez RSA Laboratories:

Definicja [pbkdf2]. Pełny, generyczny wzór funkcji PBKDF2

$$U_{i,0} = salt \parallel int32BigEndian(i)$$

$$U_{i,n} = PRF(password, U_{n-1})$$

$$T_i = U_{i,1} \oplus U_{i,2} \oplus U_{i,3} \oplus \ldots \oplus U_{i,cost}$$

$$PBKDF2(PRF, password, salt, cost, dkLen) = T_1 \parallel T_2 \parallel \ldots \parallel T_{dkLen/hLen}$$

Gdzie *PRF* to „dwuargumentowa funkcja pseudolosowa" (dobrym kandydatem jest HMAC), *password* to hasło otrzymane od użytkownika, *salt* to losowa sól kryptograficzna, *cost* to regulowany koszt wyprowadzenia pojedynczego klucza, a *dkLen* to oczekiwana długość wyprowadzonego klucza. Do parametru *salt* doklejamy jest numer iteracji „*i*", aby poszczególne iteracje różniły się czymś od siebie.

„Solenie" hashy

Dosyć poważnym atakiem przeciwko hashom generowanym z krótkich ciągów wejściowych jest wykorzystanie wcześniej wyliczonych tabel lookupu. Prawidłowe hashowanie krótkich wiadomości polega na wykorzystaniu techniki „solenia", czyli dodawaniu na jej końcu losowego ciągu bajtów, który jest potem zapisywany w formie jawnej. Nawet jeżeli potencjalny atakujący wie, że wiadomość ma długość 3 bajtów, zna dodany do niej *salt* o długości 64 bajtów oraz hash wiadomości, to nie jest w stanie posłużyć się tabelami lookupu, ponieważ wcześniejsze wygenerowanie tabel dla ciągów 69-bajtowych jest wysoce niepraktyczne. Oczywiście zawsze istnieje możliwość złamania takiego hasha metodą brute force, ale wymaga to znacznie większych nakładów mocy obliczeniowej.

Jeżeli założymy, że funkcja *PRF* to *HMAC-SHA256* oraz że koszt wynosi 3, a $dkLen = hLen = 32\ bajty$ (tzn. chcemy wyprowadzić klucz o takiej samej długości, jaką ma pojedynczy wynik SHA256), otrzymujemy bardziej czytelną postać:

Definicja [pbkdf_reduced]. Zredukowana wersja PBKDF2

$$U_1 = HMAC_{SHA256}(password, salt \parallel int32BigEndian(1))$$

$$U_2 = HMAC_{SHA256}(password, U_1)$$

$$U_3 = HMAC_{SHA256}(password, U_2)$$

$$PBKDF2'(password, salt) = U_1 \oplus U_2 \oplus U_3$$

Jak widać, cały pomysł polega na „zagnieżdżonym" obliczaniu wyniku funkcji *HMAC-SHA256*, a potem xorowaniu ze sobą wszystkich pośrednich wyników. Polecamy również zbadać inny aspekt tego algorytmu, poprzez zredukowanie oryginalnej definicji równaniami:

$$PRF = HMAC_{SHA256};\ cost = 1;\ dkLen = 3 \cdot hLen = 96 \text{ bytes}$$

Przedstawione powyżej wzory dotyczą algorytmu PKBDF2. Istnieją również inne powszechnie znane algorytmy o podobnym zastosowaniu, między innymi bcrypt, scrypt oraz Argon2 (zwycięzca konkursu na funkcję do hashowania haseł[9]).

Wspomniane algorytmy umożliwiają regulowanie kosztu obliczeniowego (wszystkie trzy), pamięciowego (scrypt oraz Argon2) oraz stopnia równoległości (tylko Argon2). Różnorodne parametry kosztu są wprowadzane celem obniżenia podatności na ataki z użyciem GPU albo dedykowanych układów scalonych. Jest to odpowiedzią na problem z algorytmem PBKDF2, który jest trudny dla CPU, ale nie wymaga zbyt dużych zasobów pamięciowych i łatwo go zrównoleglić na GPU.

Algorytm PKBDF2 wciąż jest używany, chociaż nie jest zalecanym wyborem do nowych projektów. Wspominamy jego detale implementacyjne w celu ułatwienia zrozumienia, jak działa wyprowadzanie klucza (inne algorytmy są nieco trudniejsze). Ponadto algorytm scrypt wewnętrznie wykorzystuje PBKDF2.

Warto zapamiętać, że hasło to ciąg znaków pochodzący od użytkownika, który najczęściej nie spełnia restrykcyjnych wymagań stawianych wobec klucza szyfrującego. Należy poważnie rozważyć wykorzystanie algorytmu wyprowadzania klucza, kiedy zamierzamy szyfrować dane hasłem.

Istotne jest, że wspomniane wyżej algorytmy świetnie sprawdzą się również w zakresie przechowywania haseł użytkownikow – tak jak pokazano powyżej, wykorzystanie powolnego algorytmu może „uratować" nawet osoby z dosyć słabymi hasłami przed atakiem brute force.

PODSUMOWANIE

Celem artykułu było podniesienie świadomości programistów związanej z zastosowaniami poszczególnych rozwiązań kryptograficznych, takich jak hash, podpis i algorytmy wyprowadzania klucza. Podając praktyczne przykłady, udowodniliśmy, że projektowanie zabezpieczeń z wykorzystaniem intuicji bywa zgubne, a niekiedy potrafi obniżyć bezpieczeństwo do zera, nawet pomimo użycia sprawdzonych, „niełamalnych" narzędzi.

Michał Leszczyński, Jarosław Jedynak

8. Pewne zamieszanie wprowadza fakt, że termin PBKDF (albo samo KDF) odnosi się do samego konceptu, a prawie identycznie brzmiące PBKDF2 to już nazwa konkretnego algorytmu.

9. https://password-hashing.net/