Uwierzytelnianie dwuskładnikowe

Na przykładzie Google Authenticator

Login lub email oraz hasło to nieśmiertelny mechanizm uwierzytelniania, którego w różnych postaciach używamy już od przeszło 60 lat. I o ile w większości przypadków jest on wystarczający, aby skutecznie zabezpieczyć wrażliwe dane, to jednak często konieczne jest zastosowanie mechanizmu zapewniającego większe bezpieczeństwo. Wówczas do gry wchodzi tak zwane uwierzytelnianie wieloskładnikowe.

SEKRETY

Koncepcja sekretu jest fundamentem bodaj połowy tematów wchodzących w skład kryptografii. Sekrety stanowią szczególny rodzaj informacji, które ułatwiają weryfikację tożsamości różnych osób (i nie tylko), w sytuacji gdy dysponujemy ograniczonymi możliwościami przeprowadzenia takiego procesu.

Idea jest prosta: załóżmy, że ja i Alicja znamy pewien sekret, którego nie zna nikt inny. Jeżeli teraz będę rozmawiał z kimś anonimowym, kto wykaże się znajomością tego sekretu, mogę z dużą dozą prawdopodobieństwa założyć, że jest to właśnie Alicja.

Stopień pewności weryfikacji tożsamości opartej na sekrecie zależy w dużej mierze od jego jakości. Jednym z najbezpieczniejszych sekretów mógłby być na przykład unikalny dla każdego kod genetyczny, którego podrobienie – wedle mojej wiedzy – na chwilę obecną nie jest możliwe. Jednak sprawdzenie tego kodu – szczególnie w sytuacji, gdy nie mamy fizycznego dostępu do osoby ubiegającej się o weryfikację tożsamości, jest przynajmniej niepraktyczne, żeby wręcz nie powiedzieć: całkowicie niemożliwe.

W praktyce, jeżeli kontakt z rozmówcą odbywa się poprzez jakieś medium (telefon czy Internet), rolę sekretu może pełnić jakiś strzępek informacji, która znana jest obu stronom – na przykład wspólne hasło. I faktycznie, hasło stanowi formę prostej, ale jednak w miarę skutecznej formy weryfikacji tożsamości rozmówcy. Niestety jednak jest ono jednocześnie takim rodzajem sekretu, który relatywnie łatwo może wycieknąć i dostać się w ręce osób niepowołanych, co umożliwi im nieuprawniony dostęp do chronionych zasobów. Jeżeli więc dane, które chcemy zabezpieczyć, są szczególnie wrażliwe, konieczne jest zastosowanie bardziej zaawansowanych metod uwierzytelniania. A co jest lepsze niż sekret? Oczywiście dwa sekrety.

Sprawa nie jest jednak tak prosta, jak mogłoby się to wydawać. Jeżeli bowiem do hasła dodamy drugi sekret będący również hasłem, to potencjalna poprawa ochrony danych będzie śladowa, jeżeli w ogóle jakakolwiek. Wynika to z faktu, iż wycieknięcie jednego hasła jest w praktyce równie prawdopodobne jak wycieknięcie obu – atakujący może przecież posłużyć się dokładnie tym samym mechanizmem, aby zdobyć oba z nich. Dlatego bezpieczniejsze niż sekret są nie tyle dwa sekrety, co raczej dwa *rodzaje* sekretów.

Wśród popularnych mechanizmów pełniących rolę drugiej formy uwierzytelnienia znajdziemy na przykład wysłanie wiadomości tekstowej, e-maila, wyświetlenie prośby o potwierdzenie autoryzacji w dedykowanej aplikacji mobilnej lub korzystanie z rozwiązań sprzętowych (tokeny, klucze USB, passkeys itp.). Tyle tylko, że wysyłanie SMSów kosztuje, e-maile są niewygodne w użytku (i osobiście ich nie lubię, bo zaśmiecają skrzynkę odbiorczą, jeśli nie usuwamy ich na bieżąco), napisanie dedykowanej aplikacji mobilnej dla aplikacji często nie wchodzi w grę – szczególnie jeśli miałaby służyć tylko jednemu celowi - a rozwiązania sprzętowe trzeba jakoś końcowym użytkownikom dostarczyć: często może to być rozwiązanie, które nie mieści się w budżecie projektu.

Rozwiązaniem tego problemu jest mechanizm kodów jednorazowych, który zaimplementowany jest w takich aplikacjach jak Google Authenticator czy Microsoft Authenticator. I okazuje się, że skorzystanie z nich nie będzie nas kosztowało ani grosza.

Implementacja takiego dwuskładnikowego uwierzytelnienia jest bardzo łatwa, ale zanim do niej dojdziemy, skorzystajmy z okazji, aby opowiedzieć, w jaki sposób jest ona pod spodem realizowana.

KODY JEDNORAZOWE

Uwierzytelnianie kodami jednorazowymi opiera się na bardzo podobnym pomyśle, jak uwierzytelnianie za pomocą hasła, ponieważ obie strony muszą być w posiadaniu wspólnego sekretu – klucza, będącego ciągiem bajtów. Różnica polega na tym, że weryfikacji nie podlega sam sekret, ale wartość, która jest wygenerowana za pomocą tego sekretu. Jeżeli w obu przypadkach wygenerujemy tę samą wartość, możemy domniemywać, że sekrety są takie same.

Ponieważ wygenerowane kody składają się przeważnie z sześciu lub ośmiu cyfr, mamy do czynienia z milionem lub stoma milionami możliwych kombinacji. To mało, sześcioznakowe hasło składające się z samych małych liter generuje już więcej możliwości.

Aby podnieść bezpieczeństwo tego rozwiązania, do wygenerowania hasła używany jest nie tylko klucz, ale również i licznik, czyli wartość, która zwiększana jest po wygenerowaniu każdego jednorazowego kodu. W ten sposób kod staje się jednorazowy, co sprawia, że próba złamania go metodę *brute-force* stanie się mało skuteczna.

Proces generowania w ten sposób jednorazowych kluczy opisany jest w dokumencie RFC 4226 pod nazwą „HOTP: An HMAC-Based One-Time Password Algorithm".

ALGORYTM HOTP

Algorytm HOTP korzysta z kryptograficznego konstruktu zwanego HMAC-SHA1. Samo SHA1 jest algorytmem generującym skróty, podobnym do CRC32, MD5 i im podobnych: jego zadaniem jest przyporządkowanie każdemu, dowolnie dużemu ciągowi bajtów unikalnej wartości o stałym rozmiarze. Kolejne wywołania SHA-1 dla tego samego ciągu zwrócą zawsze ten sam wynik, ale niewielka zmiana danych wejściowych (nawet o jeden bit) sprawia, że wygenerowana wartość będzie się znacząco różnić od poprzedniej.

Przedrostek HMAC algorytmu HMAC-SHA1 rozwija się jako „Hash-based Message Authentication Code" i oznacza proces polegający na generowaniu skrótu z użyciem klucza. W ramach tego procesu skracana wiadomość łączona jest najpierw z kluczem, a następnie wykonywany jest skrót tego złączenia. W kolejnym kroku wartość tego skrótu ponownie łączona jest z kluczem i w takiej formie haszowana po raz drugi. Ostatni skrót stanowi końcowy wynik działania algorytmu.

Ogólnie mechanizm ten stanowi coś w rodzaju uproszczonej wersji podpisu cyfrowego i pozwala na stwierdzenie, czy przesłane przez niezabezpieczone medium dane nie zostały zmodyfikowane – pod warunkiem, że obie strony znajdują się w posiadaniu tego samego klucza. Wykonany powyższym algorytmem skrót wystarczy przesłać razem z wiadomością: atakujący znajdujący się na drodze przesyłanych danych będzie miał wtedy tylko dwie opcje. Jedną z nich będzie usunięcie skrótu, co w oczywisty sposób wskaże na potencjalne manipulacje w wiadomości, drugą zaś – wygenerowanie nowego skrótu po zmodyfikowaniu przesyłanych danych, ale tego nie uda się zrobić bez klucza, który użyty został na początku całego procesu.

Wróćmy jednak do generowania jednorazowych haseł.

Algorytm HOTP wymaga zdefiniowania następujących wartości:

1. Licznika (`C`). Licznik musi być synchronizowany pomiędzy serwerem oraz klientem (co oznacza, że musi być przechowywany per-klient). W przypadku rozsynchronizowania kłopotliwe może okazać się doprowadzenie do ponownej synchronizacji.
2. Klucza (`K`) – współdzielonego przez serwer i klienta sekretu. Stanowi on ciąg bajtów o dowolnej długości, aczkolwiek dokument RFC 4226 zaleca minimum 128 bitów, a rekomenduje przynajmniej 160.
3. Liczba cyfr kodu jednorazowego (`d`) – zwykle 6 lub 8. Wartości te stanowią wypośrodkowanie pomiędzy bezpieczeństwem (oczywiście im dłuższy kod, tym lepszy) a wygodą stosowania. Wśród osób zajmujących się kryptografią krąży bowiem powiedzenie: „Security at the expense of usability comes with the expense of security" (bezpieczeństwo kosztem wygody użytkowania przyjdzie kosztem bezpieczeństwa).
4. Parametr resynchronizacji (`s`). Aby zmniejszyć prawdopodobieństwo rozsynchronizowania licznika, serwer może weryfikować nie tylko bieżący kod, ale również i `s` następnych. Zmniejsza to nieco bezpieczeństwo rozwiązania, ale też ogranicza skutki potencjalnych problemów z połączeniem, które mogłyby rozsynchronizować liczniki.
5. Progu akceptacji prób nieudanych uwierzytelnień (`T`). Jeżeli klient podejmie `T` występujących po sobie nieudanych prób uwierzytelnień, serwer odmówi dalszych prób połączenia. Jest to proste zabezpieczenie przed próbą ataku typu *brute-force*.

Sam algorytm HOTP realizowany jest następująco:

1. Generujemy HMAC-SHA1 za pomocą klucza `K`. Skracaną wiadomością w tym przypadku będzie po prostu wartość licznika. W efekcie otrzymamy ciąg 20 bajtów reprezentujących wartość skrótu.
2. Pobieramy ostatni bajt i maskujemy 4 najstarsze bity (czyli np. `0xef` zamieni się w `0x0f`). W ten sposób otrzymujemy wartość, którą nazwiemy *offsetem*.
3. Pobieramy ze skrótu cztery bajty, począwszy od offsetu. Jeżeli przyjmiemy, że naszym offsetem było `0x0f`, czyli 15, będą to bajty o indeksach: 15, 16, 17 i 18.
4. Scalamy pobrane 4 bajty w liczbę całkowitą 32-bitową w formacie *big-endian* (najbardziej znaczący bajt jest pierwszy).
5. Wygaszamy najstarszy bit otrzymanej wartości. W założeniu twórców algorytmu ma to sprawić, że liczba we wszystkich architekturach będzie traktowana jako dodatnia, co ograniczy potencjalne problemy związane z zachowaniem operatora modulo na ujemnych wartościach.
6. Na otrzymanej liczbie wykonujemy operację modulo 10^d, czyli efektywnie pobieramy d najmniej znaczących cyfr. Na przykład, jeżeli d = 6, a otrzymany skrót to 1203956243, w wyniku otrzymamy 956243.
7. Wynik operacji modulo stanowi nasz kod jednorazowy.

ALGORYTM TOTP

Choć algorytm HOTP jest uznawany za bezpieczny (efektywnie jedyną formą ataku jest brute-force), to jego słabą stroną zdecydowanie jest konieczność synchronizowania liczników pomiędzy klientami oraz serwerem. Cóż; prawa Murphy'ego powiedzą nam, że jeżeli tylko istnieje możliwość rozsynchronizowania, to z całkowitą pewnością kiedyś do niej dojdzie.

Dlatego też zaproponowana została niewielka modyfikacja algorytmu HOTP zwana TOTP: „Time-Based One-Time Password Algorithm" (RFC 6238). Bazuje on na algorytmie HOTP, ale zastępuje licznik wartością generowaną na podstawie bieżącego czasu.

Jako wartość zerową przyjmujemy początek epoki linuksowej, czyli 1 stycznia 1970 roku. Następnie liczbę sekund, które upłynęły od tamtego czasu (ang. *epoch timestamp*), dzielimy na odcinki o stałej wielkości – przeważnie 30 sekund lub minutę. Później pozostaje już tylko wyznaczyć, w którym odcinku się obecnie znajdujemy, aby wartość tę móc wykorzystać jako licznik.

Dla przykładu, w czasie pisania tego artykułu bieżącą wartością epoch timestamp było 1745536812. Jeżeli podzielimy tę wartość przez 30, otrzymamy 58184560,4. Liczbę tę zaokrąglamy teraz w dół i otrzymujemy 58184560 – co przekazujemy następnie algorytmowi HOTP jako wartość licznika.

RFC opisujące algorytm TOTP rozszerza również możliwość stosowania funkcji haszujących o HMAC-SHA-256 lub HMAC-SHA-512, które wewnętrznie używają algorytmów, odpowiednio, SHA-256 i SHA-512. Zmiana ta podyktowana jest faktem, iż od 2010 roku NIST uznało algorytm SHA-1 za niewystarczająco bezpieczny.

Spróbujmy teraz zaimplementować algorytm TOTP w C#. Z uwagi na fakt, iż implementacja HMAC-SHA1 znajduje się w bibliotekach standardowych platformy .NET, nie będzie to zbyt skomplikowany proces:

APLIKACJE SAMOCHODOWE ANDROID AUTO · Z ARCHIWUM CVE

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

3/2025 (118)

Cena 32.90 zł (w tym VAT 8%)

CZY LLM TO SZTUCZNA INTELIGENCJA?

Praktyczny przewodnik po Web Extension API

Uwierzytelnianie dwuskładnikowe na przykładzie Google Authenticator

„Rust to rak", czyli rzecz o software maintainability

MEGA65 – komputer o wielu obliczach

Nowy numer juz w empikach