HMAC-based one-time password algoritme

`http://tools.ietf.org/html/rfc4226` HOTP: An HMAC-Based One-Time Password Algorithm
`http://tools.ietf.org/html/rfc6238` TOTP: Time-Based One-Time Password Algorithm

HOTP

HOTP is een HMAC based one-time password algoritme dat voor 2-factor authenticatie gebruikt wordt..

Definitie

- K is een *secret key*.
- C is een *counter*.
- HMAC(K,C) is een HMAC functie, in rfc4226 staat vermeld dat hier SHA1 moet worden gebruikt, maar voor een HMAC kan je een willekeurige hashing algoritme gebruiken. Als de lengte van de key (*shared secret*) te groot is zal deze gewoon worden getrunceerd.
- Selecteer 4 bytes van de resulterende HMAC. We noemen deze functie $T()$. De 4-bytes moeten altijd op de zelde manier worden gekozen.
  HMAC(K,C) = $H((K \oplus opad)||H((K \oplus ipad)||C))$
- HOTP(K,C) = T(HMAC(K,C)) & 0x7FFFFFFF. De reden dat we hier een AND-mask toepassen is om de MSB weg te werken enzo meer compatibel te zijn tussen verschillende processen.
- Op het gewenste aantal digits te verkrijgen trunceren we gewoon de output op de volgende manier: HTOP = HTOP(K, C) mod 10^d waar d het aantal digits is.

Voorwaarden en opmerkingen

Zoals vermeld moet er worden gebruikt gemaakt van een *secret key*. Deze key moet gekend zijn door de client als door de server, alsook de counter moet door beide partijen gekend zijn. De counter moet nooit worden gecommuniceerd met de server. De *secert key* wel. Na het genereren van de HMAC moet de counter worden verhoogd.
De RFC voor dit algoritme vertelt ons dat we SHA1 moeten gebruiken. Maar SHA1 heeft technische problemen, nl er kunnen collisions worden gevonden in 2^69 operaties. Dit is argumenteerbaar of het echt een probleem zou zijn omdat de counter die hier zou worden gebruikt blijft optellen. Dus voorspellen waar de counter zou zijn is moeilijk. Let wel op, moeilijk. Niet onmogelijk.

TOTP

Time-based one-time password algoritme, dit is een extentie van het HOTP, het grote verschil met HTOP is dat hier de "tijd" een factor speelt.

Een ander verschil hier is dat de standaard RFC-6238 beschrijft dat ook andere hashing algoritmes mogen worden gebruikt, nl de algoritmes uit de SHA2-familie. Die om dit moment betere hashing algoritmes zijn.

Definitie

Citatie uit de rfc: " Basically, we define TOTP as TOTP = HOTP(K, T), where T is an integer and represents the number of time steps between the initial counter time T0 and the current Unix time."

- K is een *shared secret key*
- M, we defineren M als de huidige unix-time met een fout marge. Om een fout marge toe te laten zullen we de unix timestamp in zijn binaire vorm bekijken B, de n minst significante bits zullen we logische AND van de vorm 0xFFFF...FF00..00 op 0 zetten, de reden hiervoor is omdat we instaat willen zijn een ẅindowẗe introduceren waarin de TOTP output het zelfde zal zijn, ookal drift de unix time stamp. We negeren dus het verschil.
- TOTP = HOTP(K,M) (De layout van HOTP blijft het zelfde).

Voorwaarde en opmerkingen

Onze implementatie hier maakt opnieuw gebruikt van een *Shared secert* Beide definities van TOTP en HOTP zijn volgens de definitie van de bovenstaande RFC, de RFC's bevatten voorbeeld code in java. Een implementatie van een HMAC in python kan je vinden in deze github repo onder v̈oorbeeld_code:

HMAC-based one-time password algoritme

De opgave vertelt ons ook dat we moeten instaat zijn om transacties te verificeren. Dus er moet een notie zijn het plaatsen van een digitale handtekening. De manier waarop we dit gaan doen zal terug gebasserd zijn op een HMAC en zal ook terug gebruik maken van een *shared secert*, nl het zelfde *shared secert* dat we gebruiken tijdens het 2-factor inloggen.

De manier waarop dit zal werken zal de volgende zijn: De client stelt zijn verichting op via de web-interface. Eenmaal dit inorde is zal hij klikken op "uitvoeren". Opdat moment zal de server die verichting nemen en een hash berekenen. Daarna zal de server deze hash trunceren tot 6 digits. Dit is onze M. De server vraagt aan de client om dit bericht digitaal te ondertekenen met zijn smartphone.
De client typt de M (6 digits lang) in op zijn smartphone. Daarna nemen we opnieuw op de smartphone de current unix timestamp en beschouwen we die in zijn binaire vorm B, daarna voeren we terug een logische AND uit om de n minst

significante bits op 0 te zetten (we doen dit om opnieuw om een ẅindowv̈an tijd te introduceren waarin de HMAC van onze bericht M het zelfde zal zijn). Dus de input van onze HMAC zal het volgende zijn: $HMAC(shared\ secert, M||B$ AND 0xFFFF...FF00..00), waar *shared secret* de zelfde key is als tijdens de 2-factor authenticatie.
De output van deze HMAC zal dan terug worden getrunceerd naar 6 digits en de gebruiker tikt deze over op zijn computer. We noemen de output van onze HMAC R, R wordt terug naar de server gestuurd en controleerd deze. De server beschikt over de unix timestamp (uiteraard met de logische AND om een window in de tijd te introduceren). Over het *shared secert*, want die is geassocieerd met de user zijn account. en over M want de server heeft deze aan de client gegeven. Op deze manier hebben we authenticatie en verficatie. M en R samen met de gebruikte unix timestamp worden in de database opgeslagen.