

Joona Korkeamäki

**SALASANOJEN VARASTOINTI JA TO-
DENNUS**
Katsaus salasanatiivisteisiin

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2024

TIIVISTELMÄ

Joona Korkeamäki: Salasanojen varastointi ja todennus
Kandidaatintutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2024

Salasanoja ei kannata tallentaa suoraan mihinkään, sillä tietovuodon tapahtuessa kaikki käyttäjätilit voitaisiin suoraan kaapata. Salasanat pitää kuitenkin tallentaa jossain muodossa todentamista varten, minkä takia salasanat syötetään yksisuuntaisille funktioille, joiden tarkoituksena on tuottaa pseudosatunnainen tuloste, jonka perusteella syötettä ei voi saada selville. Tulostetut tiivisteet voidaan tallentaa, sillä niiden murtaminen vaatii käytännössä kaikkien mahdollisten syötteiden kokeilemista. Työ on kirjallisuuskatsaus, jonka tavoite on saada katsaus salasanaatiivisteisiin, niihin liittyviin perusmekanismeihin sekä niiden yhteydessä käytettäviin algoritmeihin ja standardeihin.

Tärkein työssä käsiteltävä perusmekanismi on suolaaminen. Suolaaminen on toiminto, jossa salasanaan yhdistetään ei-salainen satunnainen merkkijono joko ennen kuin se syötetään millekään funktiolle, tai vaihtoehtoisesti funktio yhdistää salasanan ja suolan suorituksen aikana. Suolan tarkoitus on yksilöidä salasanaatiivisteitä ja tehdä valmiiksi laskettujen tiivistetaulujen tekemisestä mahdollisimman raskasta.

Perusmekanismien esittelyn jälkeen työssä siirrytään algoritmeihin, joilla salasanaatiivisteitä lasketaan. Nykyään ainoa turvallinen ratkaisu salasanaatiivisteiden laskemiseen on salasanan derivointifunktiot, jotka käyttävät kryptografisia tiivistefunktioita sisäisesti, mutta niillä on säädetty kustannustekijä, joka määrää yhden tiivisteiden laskemiseen tarvittavan työmäärään. Vanhemmissa salasanan derivointifunktioissa kustannustekijänä on yksinkertainen iteraatiomäärä, mutta rinnakkaisuutta hyödyntäen hyökkääjät voivat laskea tiivisteitä niin nopeasti, että iteraatiomäärän kasvattaminen on raskaampaa oikealle käyttäjälle kuin hyökkääjälle. Rinnakkaisuudella on kuitenkin heikkous, joka on jaettu muisti. Tätä heikkoutta hyödyntäen on kehitetty muistitiraskaita salasanan derivointifunktioita, joissa voi säätää iteraatiomäärän lisäksi myös suoritukseen vaadittua muistia.

Työn havainnot osoittavat, että kryptografian kannalta salasanaatiivisteiden turvallisuuteen on olemassa erittäin vahvoja ratkaisuja, joiden työmäärää voi säätää sitä mukaa kun laskentateho kehittyy. Tutkimukset kuitenkin osoittavat, että turvallinen tiivisteiden salaus ei enää ole tärkein tekijä salasanaatiivisteiden turvallisuudessa, sillä funktioiden skaalautuvuus mahdollistaa teoriassa niiden turvaamisen nyt ja tulevaisuudessa. Ongelmaksi jää ihmisten yleinen salasanaikäyttäytyminen, kuten salasanojen uudelleenkäyttö, joka mahdollistaa sen, että hyökkääjien ei tarvitse yrittää murtaa salasanaatiivisteitä ollenkaan. Tämän lisäksi organisaatioiden haluttomuus ottaa turvallisia primitiivejä käyttöön aiheuttaa yhteisvaikutuksen, jossa käyttäjien tilit ovat vaarassa myös palveluissa, jotka käyttävät turvallisimpia ratkaisuja salasanaatiivisteiden laskemiseen.

Avainsanat: salasana, tiiviste, derivointifunktio, suolaus, todentaminen

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYS

1. JOHDANTO	1
2. SALASANOJEN VARASTOINTI	2
3. SALASANOJEN SUOLAUS	3
3.1 Suolan ja salasanan yhdistäminen.....	4
3.2 Pippurointi	4
4. DERIVOINTIFUNKTIOT.....	5
4.1 Perinteiset funktiot	6
4.2 Modernit funktiot.....	8
5. KESKUSTELU	10
6. YHTEENVETO.....	11
LÄHTEET	13

1. JOHDANTO

Salasanat ovat olleet vuosikymmeniä tehokkain tapa todentaa käyttäjiä eri järjestelmiin. Salasanojen turvallisuuteen liittyy monia ominaisuuksia, kuten salasanan vahvuus ja salassapito, todentajan ja käyttäjän välisen kommunikoinnin salaus ja salasanatiivisteiden turvaaminen. Tässä tutkielmassa keskitytään turvallisuuteen vielä syvemmällä tasolla, eli kun salasanatiivisteet ovat jo vuotaneet. Tässä tilanteessa turvallisuus perustuu erityisesti tiivisteiden vahvuuteen ja siihen, että niitä on mahdollisimman vaikea yhdistää yksittäisiin salasanoihin. Mediassa kerrotaan jatkuvasti tietovuodoista, joissa jonkun palveluntarjoajan salasanatiivisteet ovat vuotaneet. Kuitenkin herää kysymys, että miten suuressa vaarassa salasanojen paljastuminen on tällaisessa tilanteessa ja se riippuu suurelta osin tiivisteiden laskemiseen käytetyistä kryptografisista primitiiveistä.

Tämä tutkielma on kirjallisuuskatsaus, jonka tavoitteena on selvittää moderneja salasanatiivisteiden turvallisuuteen liittyviä perusmekanismeja sekä käytäntöjä, ja samalla saada katsaus siihen miten tärkeälle sijalle salasanatiivisteiden turvallisuus sijoittuu salasanaturvallisuuden kokonaiskuvassa. Tarkemmin tutkielma tavoittelee vastausta kysymykseen: ”Miten salasanatiivisteistä tehdään turvallisia?” Vastaus tähän kysymykseen lyhyesti on, että salasanatiivisteiden laskemiseen käytetään säädettäviä salasanan derivointifunktioita, joiden yhteydessä käytetään perustoimintoja, kuten suolaus ja pippurointi. Tutkielman löydökset kuitenkin osoittavat, että salasanatiivisteiden turvallisuus on enemmän käyttäjän salasanakäyttäytymisestä kiinni kuin kryptografisten primitiivien turvallisuudesta. Tutkielma alkaa selvittämällä salasanojen turvalliseen varastointiin liittyviä perusmekanismeja, jonka jälkeen siirrytään tutkimaan käytössä olevia primitiivejä ja niiden turvallisuutta toisiinsa verrattuna. Lopuksi tarkoituksena on saada katsaus turvalliseen salasanojen varastointiin nyt ja miltä se mahdollisesti näyttää tulevaisuudessa.

2. SALASANOJEN VARASTOINTI

Kun mihin tahansa palveluun luodaan uusi käyttäjä, on palveluntarjoajan pystyttävä myöhemmin todentamaan sama käyttäjä uudestaan. Kun käyttäjä myöhemmin kirjautuu tililleen käyttäen käyttäjätunnistettaan ja salasanaansa, palveluntarjoajan on pystyttävä todentamaan, että annettu salasana vastaa annettua käyttäjätunnistetta. Tämä onnistuu käytännössä vain siten, että palveluntarjoaja tallentaa käyttäjätunnisteen ja salasanan jossain muodossa.

Kaikista yksinkertaisin tapa on, että uuden käyttäjän luonnin yhteydessä palveluntarjoaja tallentaa tietokantaansa tiedot {Käyttäjätunnus, Salasana} ihmisluettavana selvätekstinä ja kun käyttäjä haluaa kirjautua sisään, hänen syöttämiään tunnuksia verrataan suoraan tallennettuihin arvoihin. Ongelma tässä tavassa kuitenkin on se, että mikäli palveluntarjoajan tietokanta vuotaa, hyökkääjä saa suoraan käsiinsä kaikki tietokantaan tallennetut käyttäjätunnus- ja salasananayhdistelmät. Tätä uhkaa pahentaa vielä se, että käyttäjillä on tapana käyttää samoja salasanoja useissa eri palveluissa (Hanamsagar et al., 2016; Pal et al., 2019). Nyt yksi vuotanut tietokanta on potentiaalisesti vaarantanut kaikkien sen palvelun käyttäjien tunnukset useissa eri palveluissa.

Edellisessä kappaleessa tuli siis todettua, että salasanoja ei kannata varastoida sellaisenaan. Ratkaisu tähän ongelmaan on kryptografisten tiivistefunktioiden käyttäminen. Tiivistefunktiot ovat yksisuuntaisia funktioita, joilla voidaan muuttaa dataa tietyn kokoiseen muotoon. Kryptografisilla tiivistefunktioilla on kolme tärkeää ominaisuutta, jotka tekevät niistä hyviä salasanojen varastoinnissa. Ensimmäinen ominaisuus on alkukuvan resistanssi, jonka mukaan tiivisteelle h on vaikea löytää sellainen syöte x , jolla $h = H(x)$. Tämä ominaisuus on se, joka tekee funktiosta yksisuuntaisen. Toinen ominaisuus on toinen alkukuvan resistanssi, jonka mukaan syötteelle x_1 pitäisi olla vaikeaa löytää toinen syöte x_2 , siten, että $H(x_1) = H(x_2)$. Kolmas ominaisuus on törmäysresistanssi, jonka mukaan pitäisi olla vaikeaa löytää kaksi eri syötettä x_1 ja x_2 siten, että $H(x_1) = H(x_2)$. Tässä on hyvä huomioida, että kun törmäysresistanssi pätee, niin toinen alkukuvan resistanssi pätee.

Nyt sen sijaan, että salasanat tallennettaisiin suoraan tietokantaan, ne syötetään ensin kryptografiselle tiivistefunktiolle ja pelkästään tiivisteet tallennetaan tietokantaan, esimerkiksi muodossa {käyttäjätunnus, $H(\text{salasana})$ }. Nyt tiiviste-funktioiden ominaisuuksien ansiosta, jos tietokanta vuotaa, on hyökkääjän vaikea saada selville alkuperäisiä salanoja. Todentaminen onnistuu siten, että todentaja laskee salasanatiivisteiden uudestaan $h = H(\text{salasana})$ ja vertaa sitä tietokantaan tallennettuun arvoon. Mikäli tiivisteet ovat samat, käyttäjä on todennettu. Tässäkin ratkaisussa on kuitenkin ongelma: samasta salasanasta tulee aina sama tiiviste. Samoilla salasanatiivisteillä voi esimerkiksi yksilöidä käyttäjiä eri tietokantavuodoista tai nähdä kuinka monella käyttäjällä on sama salasana. Hyökkääjä voi esimerkiksi laskea tiivisteitä valmiiksi yleisistä salanoista ja verrata niitä vuotaneisiin tietokantoihin murtaakseen salanoja pienellä vaivalla. Salasanatiivisteiden yksilöinnin ongelmallekin on ratkaisu: suolaus.

3. SALASANOJEN SUOLAUS

Salasanojen suolaus on toiminto, jonka ensimmäisen kerran esittelivät Robert Morris ja Ken Thompson (1979). Suolatessa salasanaan lisätään ylimääräinen ei-salainen merkkijono eli suola ennen kuin se syötetään tiivistefunktiolle, jotta syötteistä tulisi yksilöiviä, eikä kahdella samalla salanasasyötteellä saataisi samaa tiivistettä. Suolan ei ole tarkoitus olla salainen arvo, vaan se tallennetaan salasanatiivisteiden yhteyteen tietokannassa, esimerkiksi muodossa {käyttäjätunnus, $H(\text{salasana}, \text{suola})$, suola}.

Yksilöivä suolaus varmistaa sen, että vuotaneita salasanatiivisteitä ei voi suoraan yhdistää toisiinsa, eikä siis yksittäinen vuotanut salasana vaaranna muita käyttäjiä, jotka käyttävät samaa salasanaa. PKCS#5 ja NIST suosittelevat, että suolaksi valitaan satunnainen bittijono (Kaliski, 2000; Turan et al., 2010). Mikäli suolaan tarvitaan jokin tunnistetieto, esimerkiksi jotta suola ei voisi käyttää muihin tarkoituksiin, voidaan suolaksi valita tunnistemerkkijono, johon on yhdistetty satunnainen osa.

3.1 Suolan ja salasanan yhdistäminen

Modernit salasanojen hajautusalgoritmit, kuten Scrypt ja Argon2 (Birykov et al., 2016; Percival, 2009) ottavat suolan suoraan parametrina salasanan kanssa. Vanhempia tiivistefunktioita käytettäessä, kuten MD5, salasana ja suola tulee kuitenkin yhdistää ennen kuin se syötetään funktiolle. Alun perin Morris ja Thompson ehdottivat (1979), että suola lisätään salasanan perään muodossa "salasana||suola", joka sitten syötetään tiivistefunktiolle. On kuitenkin löydettävissä hieman ristiriitaista tietoa siitä, että mikä tapa on turvallisin.

Artikkelissa "Security Analysis of salt||password Hashes" Gauravaram (2012) esitteli tavan hyökätä Merkle-Damgård rakennetta käyttäviin tiivistefunktioihin, kun suolaus on tehty muodossa "suola||salasana", hyödyntäen Merkle-Damgård rakenteen alttiutta pituuden pidennyshyökkäyksille. Hyökkäys ei kuitenkaan toimi, jos käytetään alkuperäistä "salasana||suola" tyyliä. Boonkrong ja Somboonpattanakit (2016) taas väittävät, että jos hyökkääjä tietää suolan sijainnin suhteessa salasanaan, laskee sen turvallisuus huomattavasti. Tälle väitteelle ei kuitenkaan ole löydettävissä lähdettä. Intuitiivisesti voi kuitenkin ajatella, että suolan sijainnin tietäminen vähentää tarkistettavia permutaatioita, kun tehdään juuri sateenkaaritauluhyökkäyksiä. Artikkelissa ehdotetaan siis tapaa yhdistää salasana suolaan dynaamisesti, jolloin hyökkääjän on mahdotonta tietää miten suola ja salasana on yhdistetty.

3.2 Pippurointi

Pippurointi on toiminto, jossa salasanan todentaja lisää salasanaan salaisen lisäosan eli pippurin ennen kuin se syötetään tiivistefunktiolle. Tämän tarkoitus on lisätä ylimääräinen kerros turvallisuutta varastoituihin salasanatiivisteisiin. Pippurin käsite ei kuitenkaan ole yksiselitteinen, vaan se jakautuu pääosin kahden määritelmään.

Ensimmäisen määritelmän esitteli U. Manber (1996). Nimi pippuri ei ollut vielä käytössä vaan ehdotuksesta käytettiin nimeä salainen suola. Manber ehdotti, että uudelle käyttäjälle salasanan ja suolan lisäksi tiivistefunktiolle syötetään satunnainen kokonaisluku väliltä $[1, n]$, jota ei tallenneta mihinkään. Bai et al. (2024) selittivät sen toiminnan näin: Tietokantaan tallennettu lisäys on muotoa $\{\text{käyttäjätunniste, suola, } h = H(\text{salasana, suola, } x)\}$, missä x on satunnainen luku

väliltä $[1, n]$. Salasanaa todentaessa todentajan täytyy laskea annetusta salasanaasta tiivisteet, $h_1 = H(\text{salasana}', \text{suola}, 1)$, ..., $h_n = H(\text{salasana}', \text{suola}, x_n)$, sillä salasana voidaan todentaa oikeaksi tiivisteellä h_n . Väärällä salasanalla taas tiivisteet pitää laskea kaikille mahdollisille kokonaisluvuille n . Tässä tilanteessa hyökkääjän tulee laskea tiivisteitä maksimi n määrän voidakseen olla varma, että kokeiltu salasana ei ole oikea. Tämä tekniikka siis tuo salasanatiivisteiden laskemiseen epätasaisuutta todentajan ja hyökkääjän välille, sillä yhden tiivisteiden laskeminen on keskimäärin nopeampaa todentajalle.

Toinen määritelmä pippurille on, että se on yksinkertaisesti salainen suola, joka on joko yhteinen koko tietokannalle tai yksilöllinen jokaiselle käyttäjälle. Tämä pippuri kuitenkin varastoidaan, mutta ei salasanan yhteyteen suolan kanssa (Whited, 2021). Yleisessä käytössä pippurin käsite ei kuitenkaan rajoitu edes siihen, että pippuri missään muodossa syötetään tiivistefunktiolle. Esimerkiksi Dropbox kuvailee käyttävänsä tietokannan yhteistä pippuria salausavaimena, jolla salataan salasanatiivisteet AES:llä (engl. Advanced Encryption Standard) (Akhav, 2016). Tässä tapauksessa pippuri on vain synonyymi salausavaimelle.

4. DERIVOINTIFUNKTIOT

Kryptografiset tiivistefunktiot, kuten md5 ja sha-1, on ensisijaisesti tarkoitettu viestien eheyden todentamiseen, jossa tiivisteiden laskentanopeus on etu (Boonkrong & Somboonpattanakit, 2016). Nopeus toimii kuitenkin salasanatiivisteiden laskemista vastaan, sillä oikean käyttäjän tai todentajan täytyy laskea tiivisteitä vain kerran, mutta väsytyshyökkäyksessä tai sanakirjahyökkäyksessä hyökkääjän tulee laskea suuria määriä tiivisteitä. Jos yhden funktion tiivisteiden laskemisessa kestää $1 \mu\text{s}$ ja toisen funktion laskemisessa 1 ms , ero yksittäiselle käyttäjälle ei ole merkittävä, mutta hyökkääjälle se merkitsee 1000-kertaista työ määrän. Modernit tietokoneet pystyvät laskemaan vanhempia tiivistefunktioita niin nopeasti, että yksittäinen iteraatio ei riitä turvalliseen salasanojen varastointiin. Esimerkiksi RTX 3060 Ti näytönohjain pystyy laskemaan Hashcat-ohjelmaa käyttäen noin $3,5 * 10^{10}$ md5 tiivistettä sekunnissa.

Tämän takia salasanatiivisteiden laskemiseen ei käytetä enää yksittäisiä tiiviste-funktioita vaan salasanan derivointifunktioita. Salasanan derivointifunktiot ottavat parametrinaan salasanan, suolan sekä iteraatiomäärän tuottaakseen salasanatiivisteen (Kaliski, 2000). Iteraatiomäärän tarkoitus on säädellä sitä, kauanko yksittäisen salasanatiivisteen laskemisessa kestää. Yksinkertaisimmillaan iteraatiomäärä määrittää sen, kuinka monta kertaa salasanan derivointifunktion sisäistä kryptografista tiivistefunktiota iteroidaan. Iteraatiomäärän sijasta nykyään käytetään kuitenkin ilmaisua kustannustekijä, sillä kaikki salasanan derivointifunktiot eivät käytä iteraatiomäärää säätämään sitä kauanko yksittäisen tiivisteen laskemisessa kestää. Salasanan derivointifunktiot siis tuottavat tiivisteen $h = K(\text{salasana}, \text{suola}, \text{cf})$, missä K on salasanan derivointifunktio ja cf on kustannustekijä.

Kustannustekijän säädettävyyksy tekee salasanojen derivointifunktioista joustavia, sillä käyttökohteen turvallisuusvaatimusten mukaan kustannustekijän voi asettaa siten, että yhden tiivisteen laskeminen on suhteellisen nopeaa tai mahdollisimman hidasta. Esimerkiksi kustannustekijä voidaan säätää pienemmäksi verkkosovelluksessa, jossa turvallisuus ei ole käyttäjillä yhtä tärkeää kuin nopeus. Toisaalta sovelluksessa, jossa nopeus ei ole merkityksellistä, saatetaan säätää raskas kustannustekijä paremman turvallisuuden takaamiseksi.

Seuraavissa alakappaleissa esitellään neljä salasanan derivointifunktiota, joita suositellaan käytettäväksi salasanatiivisteiden laskemiseen. Ensimmäisessä alakappaleessa esitellään kaksi vanhempaa salasanan derivointifunktiota, PBKDF2 ja Bcrypt, joita pidetään vielä turvallisina ja hyvinä vaihtoehtoina erityisesti sovelluksissa, joissa nopeus on tärkeää käyttäjäkokemukselle. Toisessa alakappaleessa esitellään kaksi modernia salasanan derivointifunktiota, Scrypt ja Argon2, jotka ovat säädettävämpiä ja joilla on ominaisuuksia rinnakkaishyökkyä vastaan.

4.1 Perinteiset funktiot

Burt Kaliskin (2000) esittelemä PBKDF2 (Password-Based Key Derivation Function 2) on yhä yleisessä käytössä oleva salasanan derivointifunktio. PBKDF2 ottaa parametreina salasanan, suolan, iteraatiomäärän sekä lopullisen tiivisteen pituuden. Siis tiiviste $h = \text{PBKDF2}(\text{salasana}, \text{suola}, \text{cf}, \text{dkLen})$, jossa cf on iteraatiomäärä ja dkLen on haluttu tiivisteen pituus tavuina. PBKDF2 käyttää sisäisesti pseudosatunnaista funktiota, joka on

oletusarvoisesti HMAC-SHA-1, mutta käytännössä mikä tahansa pseudosatunnainen funktio käy. PBKDF2 laskee tiivistelohkoja, joissa jokaisessa satunnaisfunktiota iteroidaan cf osoittaman määrän. Tiivistelohkojen pituus on siis riippuvainen satunnaisfunktion tulosten pituudesta. Tiivistelohkoja lasketaan sen verran, että niiden yhteispituus on enemmän kuin $dkLen$ ja PBKDF2:n lopullinen tuloste on yhdistettyjen lohkojen ensimmäiset $dkLen$ tavua. Esimerkiksi, jos satunnaisfunktio on HMAC-SHA-1, jonka tulosten pituus on 20 tavua ja $dkLen$ on 64 tavua, niin PBKDF2 laskee neljä tiivistelohkoa, jotka yhdistetään 80 tavun pituiseksi tiivisteeksi. Lopullisena salasana tiivisteenä käytetään yhdistettyjen lohkojen ensimmäiset 64 tavua.

PBKDF2 käyttää kustannustekijänään iteraatiomäärää. Kaliskin (2000) alkuperäinen suositus iteraatiomääräksi oli vähintään 1000 kierrosta, mutta iteraatiomäärä on mielivaltainen luku, joten on oletettavaa, että sitä säädetään tilanteen mukaan. OWASP suosittelee PBKDF2-HMAC-SHA-1:n iteraatiomääräksi vähintään 1,3 miljoonaa iteraatiota ja PBKDF2-HMAC-SHA-256:n 600 000 iteraatiota (Password storage, n.d.).

Niels Provosin ja David Mazièresin (1999) esittelemä Blowfish (Schneier, 1994) lohkosalausalgoritmiin pohjautuva Bcrypt on toinen suosittu salasanan derivointifunktio. Se ottaa parametrinaan salasanan, suolan ja kustannustekijän, joka on Bcryptin tapauksessa kaksikantainen logaritmi halutusta iteraatiomäärästä tai toisin sanoen luku n , jolla 2^n on haluttu iteraatiomäärä. Siis tiiviste $h = \text{Bcrypt}(\text{salasana}, \text{suola}, cf)$, jossa cf on aiemmin kuvattu kustannustekijä. Bcrypt ei ota parametrinaan halutun tiivisteiden pituutta, sillä lopullinen tiiviste on aina 24 tavua pitkä, mikä tarkoittaa, että Bcrypt ei ole määritelmän mukaan derivointifunktio.

Salasanatiivisteiden laskennassa toivottava hitaus Bcryptin kohdalla perustuu sen sisäiseen EksBlowFishSetup funktioon, joka luo salasanan ja suolan pohjalta Blowfishia varten avainakataulun iteroiden 2^{cf+1} kertaa raskasta ExpandKey funktiota. Lopullinen tiiviste saadaan iteroimalla Blowfish algoritmia 64 kertaa staattisella tekstisyötteellä sekä salasanan ja suolan avulla luodulla avainakataululla.

Provos ja Mazières (1999) suunnittelivat Bcryptin siten, että sen suoritusnopeutta olisi vaikea kasvattaa sen laskemista varta vasten rakennetuilla laitteistoilla (Provos, 2023). Tämän takia Bcryptin pohjustaksi valittiin Blowfish lohkosalausalgoritmi, sillä sen käyttämä ExpandKey funktio käyttää 4 kb muistia. Tämän muistirajoitteen ansiosta Bcrypt on jokseenkin tehokas rinnakkaistamista vastaan, mikä tekee siitä myös turvallisemman kuin PBKDF2, kun hyökkääjä hyödyntää rinnakkaislaskentaa (Hatzivasilis, 2017; Malvoni et al., 2014; Percival, 2009).

4.2 Modernit funktiot

Proessorien ja erityisesti näytönohjainten sekä ohjelmoitavien porttimatriisien laskentatehon kehittyessä on tullut tarve löytää parempia ratkaisuja salasanaatiivisteiden laskemiseen. Vanhat ratkaisut, kuten PBKDF2 ja Bcrypt eivät enää riitä, sillä rinnakkaislaskentaa hyödyntävät tekniikat mahdollistavat sateenkaari-
taulujen laskemisen nopeasti (Dürmuth et al., 2012; Malvoni et al., 2014). Tärkein tekijä rinnakkaisessa salasanojen murtamisessa on jaettu muisti, jota pidetään kalliina (Hendarto & Kurniawan, 2017). Tämä tekee jaetusta muistista pulonkaulan rinnakkaisen salasanojen murron tehokkuudelle. Näytönohjainten avulla rinnakkaislaskentaa vastaan artikkelissa suositellaan iteraatiomäärän kasvattamista sitä mukaa kun näytönohjainten teho kehittyy. Iteraatiomäärän kasvattaminen kuitenkin hidastaa myös palveluun todentamista oikeille käyttäjille, joten liiallinen iteraatiomäärän kasvaminen voi heikentää käytettävyyttä.

Toinen vaihtoehto rinnakkaislaskentaa vastaan on hyödyntää sen muistirajoitetta. Tätä varten on kehitetty muistiraskaita salasanan derivointifunktioita, jotka vaativat paljon muistia, jotta niitä voitaisiin laskea nopeasti (Alwen et al., 2017). Normaalille käyttäjälle suuri muistivaatimus ei ole ongelma, mutta väsytyshyökkäyksen näkökulmasta tiivisteiden laskeminen rinnakkaisesti ei ole juurikaan tehokkaampaa kuin tavallinen tiivisteiden laskeminen yksittäisillä ytimillä.

Termin muistiraskas funktio määritteli Colin Percival (2009). Hänen määritelmänsä mukaan muistiraskaan algoritmin on käytettävä lähes yhtä paljon muistiosoitteita kuin se tekee operaatioita. Tässä määritelmässä on kuitenkin ongelma, että vaikka se käyttää paljon muistia, ei se suoraan estä rinnakkaislaskentaa. Percival antoi toisen määritelmän peräkkäisesti muistiraskaille funktioille, jotka vaativat muistiosoitteita käyttävien operaatioiden peräkkäistä laskemista, tehden rinnakkaislaskennasta vähemmän tehokasta. Tämä jälkimmäinen määritelmä on se, jota yleensä tarkoitetaan, kun puhutaan muistiraskaista funktioista.

Samassa artikkelissa (Percival, 2009) esiteltiin Scrypt, joka on kolmas laajassa suosiossa oleva salasanan derivointifunktio. Scrypt on peräkkäisesti muistiraskas funktio, joka tuottaa salasanaatiivisteen muodossa $h = \text{Scrypt}(\text{salasana}, \text{suola}, n, r, p, \text{dkLen})$, jossa n on yleinen kustannustekijä, joka kasvattaa vaadittua laskentatehoa, r säätelee kuinka paljon muistia vaaditaan ja p säätelee sitä,

kuinka montaa ydintä voidaan hyödyntää rinnakkaiseen laskemiseen. Siis n ja r parametreilla säädetään suhde muistikäytön ja vaaditun laskentatehon välillä. Scryptin $dkLen$ parametrilla voidaan päättää minkä pituisen tiivisteen se tulosta samoin kuin PBKDF2:n kohdalla.

Percivalin (2009) esittelemän Scryptin suorituksen alussa kutsutaan PBKDF2:ta yhdellä iteraatiolla luodakseen tiivistelohkoja salasanasta ja suolasta, jotka syötetään erikseen ROMix tyyppiselle sekoitusalgoritmille. Percival kuvailee ROMix algoritmia siten, että se laskee suuren määrän satunnaisia arvoja ja sitten käyttää niitä satunnaisesti, mutta järjestyksessä. Scryptin käyttämän ROMix funktion nimi on Smix, joka on siis se funktio, joka tekee Scryptistä muistiraskaan. Smix:n muistiraskaus yksinkertaistettuna perustuu siihen, että siinä kutsutaan ja sekoitetaan paljon muistiosoitteita, millä saadaan pidettyä paljon muistia varattuna suorituksen aikana. Sekoituksen jälkeen PBKDF2:ta kutsutaan vielä sekoitetuille lohkoille yhdellä iteraatiolla, josta saadaan lopullinen tiiviste.

Vuonna 2013 järjestettiin avoin kilpailu uuden salasanatiivistestandardin luomiseksi (Password hashing competition, n.d.). Tarkoituksena oli löytää uusia ratkaisuja salasanatiivisteiden laskemiseen turvallisesti. Yhtenä suurena huolenä oli rinnakkaisuuden hyödyntäminen salasanojen murtamisessa, joten kilpailussa kiinnitettiin erityistä huomiota rinnakkaisuuden vastustukseen (Hatzivasilis, 2017). Vuonna 2015 kilpailun lopulliseksi voittajaksi julistettiin salasanan derivointifunktio Argon2, jota kilpailun järjestäjät suosittelivat ensisijaisesti käytettäväksi vanhempien algoritmien sijasta.

Argon2 on muistiraskas funktio, joka suunniteltiin olemaan mahdollisimman yksinkertainen ja nopea, mutta silti raskas laskea rinnakkaisesti (Biryukov et al., 2016). Argon2:sta esiteltiin alun perin kaksi versiota Argon2d ja Argon2i, joista ensimmäinen on nopeampi ja parempi rinnakkaisuutta vastaan, mutta altis sivukanavahyökkäyksille (Hatzivasilis, 2017). Argon2i taas on hitaampi, mutta ei omaa samaa alttiutta kuin Argon2d. Tämän takia Argon2:n suunnittelijat suosittelivat Argon2d:tä käytettäväksi pääosin kryptovaluuttojen kanssa ja Argon2i:tä salasanatiivisteiden laskemiseen. Myöhemmin on kuitenkin esitelty kolmas versio Argon2id, joka hyödyntää molempien edellisten versioiden etuja. Argon2id on se versio, jota yleisesti suositellaan käytettäväksi (Password storage, n.d.).

Argon2 ottaa parametrinaan salasanan, suolan, rinnakkaisuusasteen p , minimi muistimäärän m , tiivisteeseen haluttu pituus $dklen$ sekä kustannustekijän t , joka Argon2:n tapauksessa on iteraatiomäärä (Biryukov et al., 2016). Lopullinen tiiviste saadaan siis muodossa $h = \text{Argon2}(\text{salasana}, \text{suola}, p, m, t, dklen)$, jossa m ja t yksinkertaistettuna säätelevät muistinkäyttöä ja vaadittua laskentatehoa ja p säättää sitä, kuinka monta ydintä tiivisteeseen laskemisessa voidaan käyttää samoin kuin Scrypt:n parametri p .

5. KESKUSTELU

Kaikki edellisessä osiossa esitellyt funktiot, PBKDF2, Bcrypt, Scrypt ja Argon2, ovat laajassa käytössä. Yleisesti kuitenkin salasana tiivisteiden laskemiseen suositellaan PHC voittajaa Argon2:ta ja erityisesti sen Argon2id variaatiota (Password storage, n.d.; Whited, 2021). Mikäli Argon2 ei ole saatavilla, Owasp suosittelee ensisijaisesti Scrypt:iä ja toissijaisesti Bcrypt:iä. PBKDF2:ta suositellaan vain, jos on tarve noudattaa NIST:n määrittämää FIPS-140 standardia, joka vaatii, että derivointifunktio käyttää sisäisesti hyväksytyjä tiivistefunktioita (NIST, 2001), jotka ovat käytännössä SHA-perheen funktiot.

Modernit salasanan derivointifunktiot vaativat suolan, sillä se annetaan suoraan parametrinä. Riippuen toteutuksesta, kirjastot saattavat luoda suolan sisäisesti, eikä suolaa tarvitse luoda erikseen. Esimerkiksi argon2-cffi python kirjasto luo suolan sisäisesti (argon2-cffi, n.d.). Pippuria taas ei ole pakollista käyttää, mutta se on suositeltavaa ylimääräisen suojakerroksen luomiseksi.

Salasanojen derivointifunktiot ovat kuitenkin nykyään niin hyviä estämään väsytyshyökkäyksiä, että ne eivät enää ole suurin vaara salasanaturvallisuudelle. Suurin syy käyttäjätilien vaarantumiselle ovat hyökkäykset, joissa kokeillaan jo vuotaneita salasanoja mahdollisimman monessa eri palvelussa, sillä käyttäjillä on tapana käyttää samoja salasanoja kaikkialla (Hanamsagar et al., 2016; Pal et al., 2019). Tähänkin ongelmaan on jo olemassa tekninen ratkaisu, joka on kaksivaiheinen tunnistautuminen. Kaksivaiheinen tunnistautuminen vähentää salasanaturvallisuuden merkitystä, sillä hyökkääjä ei pysty vielä ottamaan käyttäjät-

liä haltuun pelkällä käyttäjätunnuksella ja salasanalla. Yksittäisen palvelun kaksivaiheinen tunnistautuminen ei kuitenkaan estä toisen palvelun tilin valtaamista, jossa ei ole kaksivaiheista tunnistautumista.

Bcryptin kehittäjän Niels Provosin (2023) mielestä salasanaturvallisuus on jo ratkaistu tekniseltä kannalta. Ongelmaksi jää se, että organisaatioiden tulisi ottaa käyttöön turvallinen teknologia, joka on jo olemassa. Uuden teknologian käyttöönotto voi kuitenkin olla kallista ja siksi moni organisaatio saattaa mieluummin ottaa riskin, että tietovuoto tapahtuu. Ehkä tärkein tekijä on kuitenkin ihmisten turvallisuuskäyttäytyminen. Salasanatiivisteiden murtamisen haasteellisuudella ei ole mitään merkitystä, jos hyökkääjä saa käyttäjän tai todentajan antamaan kirjautumistunnukset vapaaehtoisesti.

6. YHTEENVETO

Salasanatiivisteiden turvallisuus kryptografian näkökulmasta on hyvällä tasolla. Tämän työn tutkimuskysymykseen ”miten salasanatiivisteistä tehdään turvallisia” saatiin vastaus. Salasanatiivisteiden turvallisuuteen liittyy kaksi tärkeää perusmekanismia: suolaus ja pippurointi. Suolatessa salasanatiivisteiden laskemisen yhteydessä salasanaan lisätään satunnainen merkkijono, jonka luottamuksellisuudella ei ole väliä. Merkkijonon tarkoitus on yksilöidä salasanatiivisteitä sekä vaikeuttaa salasanatiivisteiden laskemista etukäteen. Pippurointi taas on toiminto, jolla ei ole yleispätevää määritelmää, mutta sitä voidaan kuvailla salaisena suolana, jonka tarkoitus on tuoda ylimääräinen kerros turvallisuutta salasanatiivisteisiin, sillä nyt hyökkääjän tulisi kokeilla kaikkia mahdollisia pippureita, mikäli se pysyy salaisena. Salasanatiivisteet itsessään tulee laskea salasanan derivointifunktioilla, jotka ovat yksisuuntaisia funktioita, joissa työmäärää sekä muita ominaisuuksia voidaan säätää turvallisuusvaatimusten mukaan. Näistä salasanan derivointifunktioista Argon2:ta pidetään turvallisimpana ja sitä suositellaan ensisijaisesti käytettäväksi salasanatiivisteiden laskemiseen.

Olemassa olevan kryptografian näkökulmasta salasanatiivisteiden turvallisuus on siis hyvällä tasolla, mutta kryptografia ei ole ainoa salasanatiivisteiden turvallisuuteen vaikuttava tekijä, sillä myös ihmisellä on osansa salasanatiivisteiden

turvallisuudessa. Esimerkiksi salasanojen uudelleenkäyttö tekee salasanatiivisteiden turvallisuudesta merkityksetöntä, sillä hyökkääjän ei tarvitse edes yrittää murtaa salasanaa, mikäli se on jo aiemmin vuotanut. Organisaatioiden huonot turvallisuuskäytännöt sekä haluttomuus investoida turvallisimpaan teknologiaan vaarantavat myös käyttäjätilejä. Turvallisia ratkaisuja on siis olemassa, mutta niitä ei käytetä oikein tai niitä ei käytetä ollenkaan.

LÄHTEET

- Akhawe, D. (2016). *How Dropbox securely stores your passwords*. Dropbox. <https://dropbox.tech/security/how-dropbox-securely-stores-your-passwords>
- Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., & Tessaro, S. (2017). Script Is Maximally Memory-Hard. In J.-S. Coron & J. B. Nielsen (Eds.), *Advances in Cryptology -- EUROCRYPT 2017* (pp. 33–62). Springer International Publishing. https://doi.org.lib-proxy.tuni.fi/10.1007/978-3-319-56617-7_2
- argon2-cffi. (n.d.). *API Reference - Argon2-cffi 23.1.0 documentation*. Argon2-Cffi. Retrieved April 15, 2024, from <https://argon2-cffi.readthedocs.io/en/stable/api.html>
- Bai, W., Blocki, J., & Ameri, M. H. (2024). Cost-asymmetric memory hard password hashing. *Information and Computation*, 297, 105134. <https://doi.org/10.1016/j.ic.2023.105134>
- Biryukov, A., Dinu, D., & Khovratovich, D. (2016). Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 292–302. <https://doi.org/10.1109/EuroSP.2016.31>
- Boonkrong, S., & Somboonpattanakit, C. (2016). Dynamic salt generation and placement for secure password storing. *IAENG International Journal of Computer Science*, 43(1), 27–36. https://www.iaeng.org/IJCS/issues_v43/issue_1/IJCS_43_1_04.pdf.
- Dürmuth, M., Güneysu, T., Kasper, M., Paar, C., Yalcin, T., & Zimmermann, R. (2012). Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In S. Foresti, M. Yung, & F. Martinelli (Eds.), *Computer Security -- ESORICS 2012* (pp. 716–733). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33167-1_41
- Gauravaram, P. (2012). Security Analysis of salt||password Hashes. *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, 25–30. <https://doi.org/10.1109/ACSAT.2012.49>
- Hanamsagar, A., Woo, S., Kanich, C., & Mirkovic, J. (2016). *How Users Choose and Reuse Passwords*. Information Sciences Institute. <https://www.isi.edu/publications/trpublic/pdfs/isi-tr-715.pdf>
- Hatzivasilis, G. (2017). Password-Hashing status. *Cryptography*, 1(2), 10. <https://doi.org/10.3390/cryptography1020010>
- Hendarto, I. L. S., & Kurniawan, Y. (2017). Performance factors of a CUDA GPU parallel program: A case study on a PDF password cracking brute-force algorithm. *2017 International Conference on Computer, Control, Informatics and Its Applications (IC3INA)*, 35–40. <https://doi.org/10.1109/IC3INA.2017.8251736>
- Kaliski, B. (2000). *RFC 2898: PKCS #5: Password-Based cryptography specification version 2.0*. IETF Datatracker. <https://datatracker.ietf.org/doc/html/rfc2898#section-4.1>

- Malvoni, K., Designer, S., & Knezovic, J. (2014). Are your passwords safe: energy-efficient bcrypt cracking with low-cost parallel hardware. *Proceedings of the 8th USENIX Conference on Offensive Technologies*, 10. <https://www.usenix.org/system/files/conference/woot14/woot14-malvoni.pdf>
- Manber, U. (1996). A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2), 171–176. [https://doi.org/10.1016/0167-4048\(96\)00003-x](https://doi.org/10.1016/0167-4048(96)00003-x)
- Morris, R., & Thompson, K. (1979). Password security: a case history. *Commun. ACM*, 22(11), 594–597. <https://doi.org/10.1145/359168.359172>
- NIST. (2001). *Security requirements for cryptographic modules*. National Institute of Standards and Technology. <http://dx.doi.org/10.6028/nist.fips.140-2>
- Pal, B., Daniel, T., Chatterjee, R., & Ristenpart, T. (2019). Beyond Credential Stuffing: Password Similarity Models Using Neural Networks. *2019 IEEE Symposium on Security and Privacy (SP)*, 417–434. <https://doi-org.libproxy.tuni.fi/10.1109/SP.2019.00056>
- Password hashing competition*. (n.d.). Retrieved April 15, 2024, from <https://www.password-hashing.net/>
- Password storage*. (n.d.). OWASP Cheat Sheet Series. Retrieved April 4, 2024, from https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- Percival, C. (2009). *Stronger Key Derivation Via Sequential Memory-Hard Functions*. https://www.bsdcn.org/2009/schedule/attachments/87_scrypt.pdf
- Provos, N. (2023, May 16). *Bcrypt at 25: A retrospective on password security*. USENIX. https://www.usenix.org/publications/loginonline/bcrypt-25-retrospective-password-security?trk=public_post_comment-text
- Provos, N., & Mazières, D. (1999, 0). A Future-Adaptable Password Scheme. *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. <https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme>
- Schneier, B. (1994). Description of a new variable-length key, 64-bit block cipher (Blowfish). In R. Anderson (Ed.), *Fast Software Encryption* (pp. 191–204). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-58108-1_24
- Turan, M. S., Barker, E. B., Burr, W. E., & Chen, L. (2010). *Recommendation for password-based key derivation*: National Institute of Standards and Technology. <http://dx.doi.org/10.6028/nist.sp.800-132>
- Whited, S. (2021). *Best practices for password hashing and storage, Section 4.2*. IETF. <https://www.ietf.org/archive/id/draft-ietf-kitten-password-storage-04.html#section-4.2>