Zgoščevanje gesel z algoritmom Argon2

Matej Bizjak*

9. september 2020

1 Uvod

Zgoščevanje gesel je temeljni pristop varovanja gesel, ki jih uporabniki uporabljajo za preverjanje pristnosti pri raznoraznih sistemih in aplikacijah. Gesla se zaradi varnosti na napravah ne hranijo v čistopisu, temveč se hrani njihova zgoščena vrednost. To pa zato, da napadalec ob uspešnem vdoru v sistem ne bi razkril gesel, ampak le njihove zgoščene vrednosti, kar mu oteži delo. Vendar v primeru ko branitelj gesla zavaruje le z osnovnimi kriptografskimi zgoščevalnimi funkcijami, jih lahko napadalec vseeno razkrije, saj uporabniki pogosto izberejo precej enostavna in prekratka gesla, katere lahko prefinjen napadalec z uporabo namenske strojne opreme in raznih metod enostavno pridobi. Takšni metodi sta, na primer, preiskovanje vseh možnih rešitev in uporaba mavričnih tabel (angl. rainbow tables), kjer ima napadalec shranjene že izračunane vrednosti zgoščevalne funkcije in tako ne potrebuje vsakič vsega na novo računati, ampak le primerja vrednosti v tabeli. Takšne napade lahko precej otežimo z uporabo soljenja (angl. salting) gesel, kar doda določeno stopnjo unikatnosti vsaki zgoščeni vrednosti gesla in tako preprečuje, da bi se enakim geslom pripisala enaka zgoščena vrednost. Vendar to zaradi hitrega tehnološkega napredka in vedno boljše strojne in programske opreme ni dovolj, da bi napadalcem preprečilo razkritje gesel [9]. Cilj je torej upočasniti računanje napadalca. Ena izmed metod, ki to dosežejo je raztegovanje gesel (angl. key stretching). To deluje tako, da zgoščevalno funkcijo uporabimo večkrat zapored in se tako izognemo morebitnim šibkim geslom, saj je zgoščena vrednost daljša od samega gesla in tako napadalec potrebuje veliko več časa, da pride do gesla [17]. Z uporabo namenske strojne opreme kot so, na primer, grafične kartice (angl. graphics processing unit - GPU), programabilni čipi (angl. field-programmable gate array - FPGA) in posebno namenska integrirana vezja (angl. application-specific integrated circuit - ASIC), lahko zgoščene vrednosti iz le procesorsko težke (angl. CPU-hard) zgoščevalne funkcije napadalcu še vedno uspe razkriti. Zato so bile vpeljane pomnilniško težke (angl. memory-hard) zgoščevalne funkcije, ki zelo otežijo uporabo takšnih namenskih strojnih enot.

Do nedavnega nismo imeli veliko algoritmov za varno zgoščevanje gesel, kar se je poznalo na velikem številu razkritij gesel pri podatkovnih vdorih po svetu. Edini takšen standardizirani algoritem je bil *PBKDF2*, alternativi za to pa sta še *bcrypt* in *scrypt*. Kljub temu, da so ti algoritmi za hrambo gesel veliko bolj varni kot običajne zgoščevalne funkcije, imajo kar nekaj slabosti. Vse trije algoritmi so ranljivi proti napadalcem z optimizirano namensko strojno opremo, saj napadi na zgoščene vrednosti teh algoritmov niso tako zelo pomnilniško težki, *scrypt* pa je ranljiv tudi na napade po stranskih kanalih (angl. side channel attack) [7, 9]. Zaradi vseh teh problemov je bilo organizirano tekmovanje *Password Hashing Competition* (PHC) [13], kjer so iskali varnejše rešitve za zgoščevanje gesel, ki bi bile varne tudi pred najmodernejšimi napadalci. To tekmovanje bomo predstavili v poglavju 3, pred tem pa si bomo v poglavju 2 pogledali, kako učinkovito lahko napadalci razbijejo gesla, ki so bila zgoščena z najbolj uporabljenimi algoritmi za zgoščevanje. V poglavju 4 bomo predstavili strukturo in delovanje zmagovalnega algoritma na tem tekmovanju, ki se imenuje *Argon2*, v poglavju 5 bomo preverili, kako varen je algoritem, v poglavju 6 pa si bomo pogledali, kako se algoritem uporablja.

*Kriptografija in računalniška varnost - Fakulteta za računalništvo in informatiko, Univerza v Ljubljani

2 Učinkovitost razbijanja gesel

V tem poglavju si bomo torej pogledali, kako učinkovito lahko napadalci razbijejo gesla. V članku [8] iz leta 2009 avtor oceni, koliko bi napadalca stalo razbijanje gesel, ki so bila zgoščena s takrat najbolj pogosto uporabljenimi algoritmi za zgoščevanje. Te ocene so prikazane v tabeli 1. V njej je prikazana primerjava cen takratne strojne opreme, ki je pričakovana za razbitje enega gesla v enem letu. Primerja se glede na uporabljen algoritem za zgoščevanje in na velikost vhodnega gesla. Pri algoritmih *PBKDF2*, *bcrypt* in *scrypt* je možno s parametri vplivati na čas izvajanja in tako tudi na varnost, zato je v tabeli prikazana ocena tako za manj kot tudi bolj varno različico. Hitro lahko opazimo, da algoritmi *DES CRYPT*, *MD5* in *MD5 CRYPT* pred napadi niso varni, bolje se odreže *PBKDF2*, za tem je *bcrypt*, za daleč najbolj varnega v tistem času pa je veljal *scrypt*, ki je bil tudi priporočljiv za hrambo gesel, četudi ni bil ustvarjen izrecno za to.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	< $1	< $1	< $1	< $1	< $1	< $1
MD5	< $1	< $1	< $1	$1.1k	$1	$1.5T
MD5 CRYPT	< $1	< $1	$130	$1.1M	$1.4k	1.5×10^{15}
PBKDF2 (100 ms)	< $1	< $1	$18k	$160M	$200k	2.2×10^{17}
bcrypt (95 ms)	< $1	$4	$130k	$1.2B	$1.5M	$48B
scrypt (64 ms)	< $1	$150	$4.8M	$43B	$52M	6×10^{19}
PBKDF2 (5.0 s)	< $1	$29	$920k	$8.3B	$10M	11×10^{18}
bcrypt (3.0 s)	< $1	$130	$4.3M	$39B	$47M	$1.5T
scrypt (3.8 s)	$900	$610k	$19B	$175T	$210B	2.3×10^{23}

Tabela 1: Pričakovane cene strojne opreme do leta 2009 za razbitje enega gesla v enem letu. Vir: [8]

Vendar v času pisanja noben izmed primerjanih algoritmov ni več priporočljiv za hranjenje gesel. Tudi *scrypt* je lahko ranljiv pred zelo prefinjenimi napadalci, ki uporabljajo drage naprave ASIC. V tabeli 2 je dobro prikazano, kako učinkovite so naprave ASIC (zedboard) v primerjavi z GPU (GTX 480). Rezultati so iz leta 2015, ko sta avtorja članka [6] nizko cenovno napravo ASIC primerjala z običajno grafično kartico pri razbijanju gesel algoritma *bcrypt*. Ugotovila sta, da so že nizko cenovne naprave ASIC bistveno zmogljivejše glede na porabljeno energijo, kar je v veliko korist modernim napadalcem. S časom se zmogljivost in število takšnih naprav enormno povečuje, kar pomeni, da je dolgoročno ogrožena tudi varnost *scrypt-a*, o čemer se je pisalo že leta 2014 [14]. Varne rešitve je, kot smo že omenili, torej treba iskati v pomnilniško zelo težkih funkcijah, ki razbijanje gesel praktično onemogoči tudi zelo hitrim in učinkovitim napravam ASIC.

		Target (CPU) runtime			
	Energy	1ms	10ms	100ms	1000ms
bcrypt					
– zedboard	4.2 W	2 198 H/Ws	218.15 H/Ws	23.52 H/Ws	2.36 H/Ws
– GTX 480	430 W	6.67 H/Ws	0.74 H/Ws	0.08 H/Ws	0.01 H/Ws
scrypt					
– GTX 480	430 W	99.19 H/Ws (t=1)	5.43 H/Ws (t=2)	0.11 H/Ws (t=8)	0.00 H/Ws (t=4)

Tabela 2: Primerjava porabe energije za testiranje gesel med GPU in ASIC. Vir: [6]

3 Tekmovanje PHC

Na tekmovanje, ki se je zaključilo leta 2015, se je prijavilo 24 kandidatov, med katerimi so izbrali le en najboljši algoritem - *Argon2*. Avtorji algoritma so raziskovalci Alex Biryukov, Daniel Dinu in Dmitry Khovratovich iz Univerzve v Luksemburgu.

Za vhodne parametre so prijavljeni algoritmi morali imeti:

- geslo poljubne dolžine med 0 in 128 bajtov ne glede na kodirno shemo,
- vrednost soli dolžine 16 bajtov,
- velikost izhoda do 32 bajtov in
- vsaj en parameter za nastavljanje časovnih ali prostorskih zahtev [13].

Kriterij za izbiro zmagovalca pa je bil sestavljen iz naslednjih lastnosti:

- kriptografska varnost: Odpornost praslik, odpornost drugih praslik, odpornost na trke, odpornost na druge slabosti Merkle-Damgårdove konstrukcije, kot so, napad s podaljševanjem dolžine (angl. length extension attack), trk delnih sporočil (angl. partial message collistion) itd.
- varnost pred napadi TMTO (angl. time/memory/data tradeoff attack): Napadi z vnaprej izračunanimi vrednostmi v tabelah (mavrične tabele) ne smejo biti mogoči.
- odpornost proti napadalcem z veliko procesorsko močjo: Algoritem mora biti procesorsko težek, tako da napadalec s sočasnostjo in velikim številom procesorskih jeder ne doseže bistvene pohitritve.
- odpornost proti napadalcem z namensko strojno opremo: Algoritem mora biti pomnilniško težek, tako da napadalec z uporabo namenske strojne opreme, kot so, GPU, FPGA in ASIC, ne doseže bistvene pohitritve.
- odpornost proti napadom po stranskih kanalih: Na primer, napada *cache-timing* in *Garbage Collector Attack* (GCA) [7, 9].

Poleg tega pa so gledali tudi na jasnost algoritma, število uporabljenih drugih konstruktov, enostavnost implementacije itd.

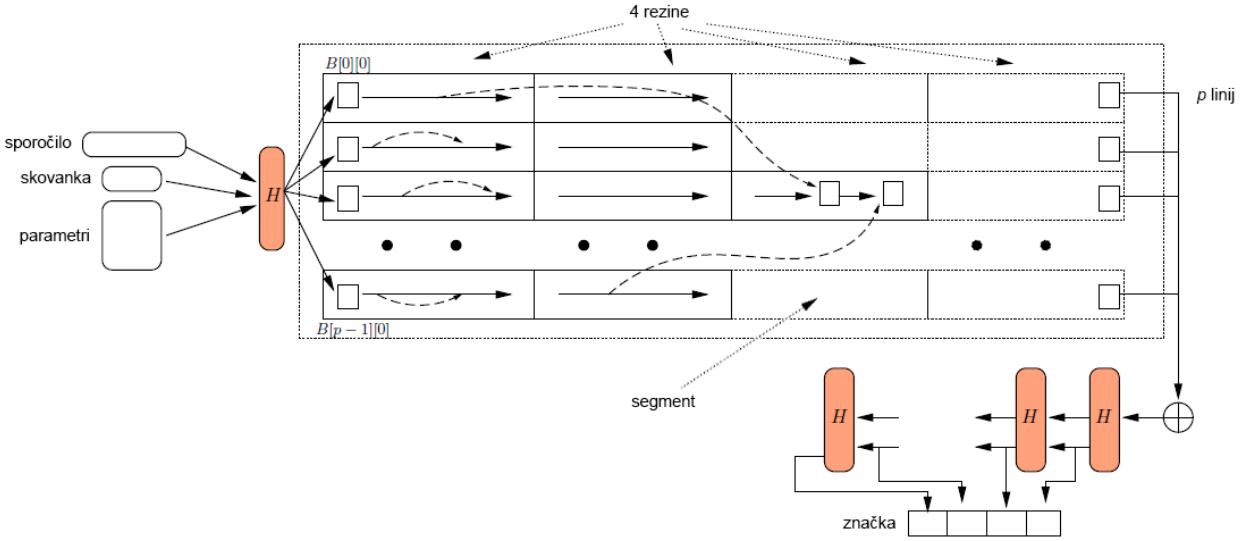
4 Struktura in delovanje algoritma

V tem poglavju bomo na kratko opisali delovanje algoritma *Argon2*, ki je optimiziran za arhitekturo x86. Algoritem podpira sočasnost, je razteglјiv tako po času kot po porabi pomnilnika in se deli na dve osnovni različici *Argon2d* in *Argon2i*, uporablja se pa tudi hibrid obeh *Argon2id*. Prva je optimizirana za hitro delovanje. Njeni pomnilniški dostopi so odvisni od vhodnih podatkov, tako da je odporna na napade z namensko strojno opremo, ampak je ranljiva na napade po stranskih kanalih, zato je bolj primerna za aplikacije, kjer to ni problem, na primer, pri zalednih strežnikih in kriptovalutah. V nasprotju s prvo, *Argon2i* deluje počasneje, pri delu uporablja od podatkov neodvisne pomnilniške dostope in postopek nad pomnilnikom ponovi trikrat, da je odporna tudi na napade TMTO. Slednji je bolj primeren za zgoščevanje gesel in izpeljavo varnostnih ključev, mogoče pa je uporabiti tudi hibrid obeh osnovnih različic - *Argon2id*, ki podeduje nekaj prednosti iz obeh. Ta za prvo polovico prehoda čez pomnilnik uporablja način različice *Argon2i*, za drugo polovico pa *Argon2d* [5, 9, 11, 15].

Argon2 za vhodne podatke lahko prejme: vhodno sporočilo P (geslo), skovanko S (sol), stopnjo sočasnosti p, dolžino značke T, velikost pomnilnika m (kjer je število 1024-bajtnih pomnilniških blokov $m' = \lfloor \frac{m}{4p} \rfloor$), število iteracij t (omogoča nastavljanje časovne zahtevnosti neodvisno od pomnilniške) in še nekaj ostalih manj pomembnih parametrov.

Na sliki 1 lahko vidimo eno iteracijo algoritma. Iteracij je lahko več, kar določimo s parametrom t. Najprej se sporočilo P skupaj s skovanko S in ostalimi parametri z zgoščevalno funkcijo H ustvari 512-bitno zgoščeno vrednost. Nato se začne polnjenje m' 8192-bitnih pomnilniških blokov. Pomnilnik je organiziran

kot $p \times q$ matrika *B[i][j]*, kjer je $q = \frac{m'}{p}$. Za računanje blokov se uporablja zgoščevalna funkcija spremenljive dolžine *H'*, ki je zgrajena na podlagi funkcije *H(X)*, kompresijska funkcija *G(X,Y)*, kazalca *i'* in *j'* pa sta določena s strani indeksne funkcije $\theta(i, j)$.



Slika 1: Ena iteracija algoritma *Argon2* s p linijami in štirimi rezinami. Vir: [3]

Indeksna funkcija se razlikuje glede na različico algoritma. Pri verziji *Argon2d* so kazalci odvisni od vhodnih podatkov, med tem ko pri verziji *Argon2i* niso. Funkciji H in *H'* sta definirani kot zgoščevalni funkciji *BLAKE2b* [1], kompresijska funkcija G pa deluje na principu permutacij, zato da se znebimo problemov, ki nastanejo pri iterativnih zgoščevalnih funkcijah [4, 9].

Da algoritem omogoča sočasno računanje posameznih blokov je pomnilnik poleg delitve po linijah razdeljen še na $S = 4$ rezin, kar nam skupaj določa segment dolžine $\frac{q}{S}$. Vsi segmenti znotraj ene rezine se izvedejo sočasno. Ko se vse iteracije zaključijo imamo poračunan zadnji blok B_m. Na koncu se nad zadnjim stolpcem izvede funkcija bitni XOR, s čimer dobimo izhodno značko $h = H'(B_m)$ [3]. Bolj podrobno je postopek algoritma predstavljen kot psevdokoda v izseku izvorne kode 1.

```
Function Argon2
    Vhodi:
      password (P): Bytes (0..2^32-1)
      salt (S): Bytes (8..2^32-1)
      parallelism (p): Number (1..2^24-1)
      tagLength (T): Number (4..2^32-1)
      memorySizeKB (m): Number (8p..2^32-1)
      iterations (t): Number (1..2^32-1)
      version (v): Number (0x13)
      key (K): Bytes (0..2^32-1)
      associatedData (X): Bytes (0..2^32-1)
      hashType (y): Number (0=Argon2d, 1=Argon2i, 2=Argon2id)
    Izhod:
      tag: Bytes (tagLength)

    // ustvarjanje zacetne 64-bajtne zgoscene vrednosti H0 s funkcijo BLAKE2b
    buffer = parallelism|tagLength|memorySizeKB|iterations|version|hashType|
          Length(password)|Password|Length(salt)|salt|Length(key)|key|
          Length(associatedData)|associatedData
    H0 = Blake2b(buffer, 64)
```

```
// izracun stevila 1 kB blokov pomnilnika
blockCount = Floor(memorySizeKB, 4*parallelism)

// alociranje 2d tabele z 1kB bloki
columnCount = blockCount / parallelism;
B[][] = Init(parallelism, columnCount)

// izracun prvih dveh blokov ((1024 B zgoscena vrednost)) vsake linije
for i = 0 to parallelism-1 do // za vsako linijo
  B[i][0] = Hash(H0|0|i, 1024)
  B[i][1] = Hash(H0|1|i, 1024)

// izracun ostalih blokov vsake linije
for i = 0 to parallelism-1 do // za vsako linijo
  for j = 2 to columnCount-1 do // za vsak blok od tretjega naprej
     index_i, index_j = GetBlockIndexes(i, j) // indeksna funkcija
     B[i][j] = G(B[i][j-1], B[index_i][index_j]) // zgoscevalna funkcija G

// ponovitev prejsnje tocke za morebitne preostale iteracije
for nIteration = 2 to iterations do
  for i = 0 to parallelism-1 do
    for j = 0 to columnCount-1 do
       index_i, index_j = GetBlockIndexes(i, j)
       if j == 0 then
         B[i][0] = B[i][0] xor G(B[i][columnCount-1], B[index_i][index_j])
       else
         B[i][j] = B[i][j] xor G(B[i][j-1], B[index_i][index_j])

// izracun zadnjega bloka C (XOR zadnjega bloka vsake linije)
C = B0[columnCount-1]
for i = 1 to parallelism-1 do
  C = C xor B[i][columnCount-1]

// izracun izhodne znacke (koncna zgoscena vrednost)
return Hash(C, tagLength)
```

Izsek izvorne kode 1: Psevdokoda algoritma *Argon2*. Povzeto po virih [11, 15].

5 Varnostna analiza

V tem poglavju si bomo algoritem pogledali iz vidika varnosti, torej kako dobro ustreza zahtevam tekmovanja, ki smo jih opisali v poglavju 3.

5.1 Kriptografska varnost

Argon2 velja za kriptografsko varen algoritem, kljub temu da kompresijska funkcija G, ki jo uporablja, ni dokazano kriptografsko varna. To bomo pokazali pri dokazovanju naslednjih trditev:

Trditev 5.1. *Argon2 je odporen na trke.*

Dokaz. Ker napadalec ne more spreminjati vhodov v G, so trki zelo malo verjetni in se jih ne da kontrolirati. Denimo, da napadalec uspe najti trk za G za dva poljubna m_1, m_2, tako da je $G(m_1) = G(m_2)$. Če bi nato želel najti trk za celoten algoritem, bi moral najti še pripadajočo prasliko za $H(x)$, kjer je x neka vrednost,

iz katere sta bila dobljena m_1 ali m_2. To pa vemo, da ni mogoče, ker je funkcija H definirana kot funkcija *BLAKE2b*, ki pa velja za kriptografsko varno [9]. □

Trditev 5.2. *Argon2 je odporen na praslike.*

Dokaz. Zaradi enakega razloga velja *Argon2* odporen na praslike, saj začetno zgoščevanje uporabniških vhodnih podatkov s kriptografsko varno funkcijo H naredi celoten algoritem odporen na praslike. Na primer, če bi napadalec za poljuben h uspel najti tak m, da velja $G(m) = h$, bi potem še vedno moral najti prasliko p, da velja $H(p) = m'$. □

Trditev 5.3. *Argon2 je odporen na druge praslike.*

Dokaz. Na enak način lahko dokažemo, da je *Argon2* odporen na druge praslike. Recimo da napadalcu uspe najti tak m_2, da velja $G(m_1) = G(m_2), m_1 \neq m_2$ za poljuben m_1 in tako izvede napad na G. Potem bi moral še vedno najti nek p, da bi veljalo $H(p) = m_2{}'$, če bi želel napad prenesti na cel algoritem. □

Ker *Argon2* ni osnovan na Merkle-Damgårdovi konstrukciji, je prav tako varen pred napadi, ki so značilni za to konstrukcijo, kot so, napad s podaljševanjem dolžine (angl. length extension attack), trk delnih sporočil (angl. partial message collistion) itd. V času pisanja tudi še ni prišlo do odkritij kakšnih varnostnih lukenj, tako pri funkciji BLAKE2b, kot pri samemu *Argon2*, tako da algoritem za enkrat velja za zelo varnega.

5.2 Varnost pred napadi TMTO

Argon2 se pred napadi TMTO rešuje z uporabo skovank (soli), tako da nobeni sporočili ne bosta nikdar imeli enake zgoščene vrednosti. To velja v primeru, da skovanke uporabljamo na pravilen način, in sicer tako da skovanke generiramo na pravilen način, priporočljivo z uporabo kriptografsko varnega generatorja naključnih števil, kot je CSPRNG (A Cryptographically Secure Pseudorandom Number Generator) [16]. Priporočena velikost skovanke je 16 bajtov, kar pomeni, da bi si napadalec moral za vsako skovanko preračunati in si shraniti izračune v tabelo, kar bi skupaj zneslo kar 2^{128} različnih tabel, vsaka pa bi bila velika vsaj nekaj gigabajtov [3, 9]. V primeru, da skovanke ne bi uporabljali na pravilen način, tako da bi lahko napadalec predvidel katera skovanka se uporablja, bi bil napad bolj izvedljiv. Vendar bi tudi to za večina napadalcev bilo izjemno težko, ker je razbitje *Argon2* zgoščenih vrednosti procesorsko in pomnilniško težek problem in bi bilo že samo preračunanje tabel zelo težko izvedljivo.

5.3 Varnost pred napadalci z optimizirano namensko strojno opremo

Kako dobro so zgoščene vrednosti odporne proti zelo močnim napadalcem, je odvisno od parametrov, ki jih uporabnik nastavi pri uporabi algoritma. Zato je zelo pomembno, da uporabnik *Argon2* pravilno uporablja in nastavi parametre primerne za svoj primer. Več o priporočljivih nastavitvah v poglavju 6, zdaj pa si poglejmo prednosti procesorsko in pomnilniško težke narave algoritma.

Avtorji v članku [3] omenijo, da je vsak algoritem, ki za delo potrebuje že nekaj sto megabajtov, skoraj gotovo neučinkovit za delo na GPU in FPGA. *Argon2* ima možnost nastavljanja porabe pomnilnika in ker lahko nastavimo, da za testiranje vsakega gesla porabi tudi po nekaj gigabajtov pomnilnika, visoka stopnja sočasnosti pri takih napravah ne pride do izraza, saj jim zelo hitro začne primanjkovati pomnilnika in se jim cenovno ne izplača. Da bi lahko ocenili moč napadalca (predvsem napadalcev z napravami ASIC, saj so te lahko izjemno zmogljive (in tudi zelo drage)) in stopnjo pomnilniške težkosti se uporablja metrika *time-area product*. Dobra ocena te metrike branitelju omogoča, da nastavi parametre zahtevnosti algoritma tako, da zadostuje za njegov primer uporabe. Pri tej metriki gledamo, kako visoko stopnjo zmanjšanja pomnilnika lahko napadalec doseže, ne da bi pri tem povečal produkta *time-area*. Če je ta stopnja dovolj nizka, pravimo, da je algoritem pomnilniško težek. Avtorji algoritma *Argon2* trdijo, da pri *Argon2-u* ne moremo doseči višje stopnje zmanjšanja pomnilnika kot je 3 [3]. In da pri obeh osnovnih različicah algoritma s privzetim številom prehodov čez pomnilnik napadalec z ASIC ne more znižati produkta *time-area*, če je stopnja zmanjšanja pomnilnika 3 ali več. In večje kot je število prehodov, višje kazni pri izračunih napadalec dobi [9]. Zato algoritem velja za pomnilniško težkega in je varen pred napadalci z optimizirano namensko strojno opremo.

Argon2 je optimiziran za delo na arhitekturi x86 in je bil posebej načrtovan, da ne bi bil dobro učinkovit za delo z GPU, FPGA in ASIC. Vendar bi lahko zaradi učinkovitosti dela s modernimi procesorji algoritem bil ranljiv za napadalce, ki imajo na voljo zelo veliko procesorske moči. Načrtovalci so *Argon2* naredili učinkovit za večjedrne procesorje zato, da bi povečali zmogljivost zgoščevanja pri branitelju, ampak so hkrati onemogočili učinkovite napade TMTO, ker so sočasnost znotraj algoritma omejili na segmente. Za število rezin so izbrali 4, ker nam to ne prinese veliko dodatnih režijskih stroškov, ampak napadalcu, ki želi izvesti napad TMTO, doda veliko časovno-prostorske kazni (angl. time-area penalty). Zato se veliko število procesorskih jeder pri tem problemu zelo slabo skalira glede na ceno opreme in kmalu za napadalca postane predrago [3, 9].

5.4 Varnost pred napadi po stranskih kanalih

Kot smo že povedali v poglavju 4, se indeksna funkcija θ razlikuje med različicama *Argon2i* in *Argon2d*. Pri prvi različici, je dostop do pomnilniških blokov definiran vnaprej (neodvisno od vhodnih podatkov), med tem ko je pri drugi dostop do pomnilnika določen sproti (odvisen od vhodnih podatkov). Druga onemogoča napade TMTO, ker napadalec ne more vnaprej pripraviti izračune, vendar se je izkazalo, da je zaradi odvisnosti od vhodnih podatkov lahko ranljiva na napade po stranskih kanalih, kot je, na primer, napad *cache-timing attack* [2]. Pri tem se lahko zgodi, da bi napadalec zajel določene informacije (v povezavi z vhodnimi podatki), s katerimi bi si zmanjšal količino dela pri testiranju kandidatov.

Ker privzeto *Argon2d* med delovanjem gre le enkrat čez pomnilnik in ga na koncu ne prepiše, je ta različica ranljiva na napad *garbage collector attack*, saj bi lahko napadalec prišel tako do začetne zgoščene vrednosti, kot do končnega rezultata in si tako olajšal delo. Druga različica (*Argon2i*) pa gre čez celoten pomnilnik trikrat, kar pomeni da delovni pomnilnik dvakrat prepiše in tako napadalcu skoraj nič ne olajša dela. *Argon2* zaradi nevarnosti pred takimi napadi uporabniku omogoča, da v parametrih algoritma nastavi zastavico, ki algoritmu pove, da naj bo končanem delu celoten uporabljen pomnilnik za seboj počisti. Tako se algoritem popolnoma izogne napadom take vrste.

6 Uporaba

Argon2 je v času pisanja še vedno priporočljiv zgoščevalni algoritem za hranjenje gesel. Kot smo že omenili imata obe glavni različici algoritma nekaj prednosti in nekaj slabosti, zato je v splošnem priporočljivo uporabiti njun hibrid - *Argon2id*, razen ko vemo, da nam za naš primer slabosti ene izmed osnovnih različic ne predstavljajo nevarnosti [15].

Naš cilj je čim bolj povečati čas in porabo računskih virov, ki ga napadalec porabi pri napadu na naša gesla, ampak hkrati ne preveč upočasniti delovanja braniteljevega strežnika, ko se uporabnik v aplikacijo želi prijaviti. Ravno zaradi želje po prilagodljivosti algoritma za naše potrebe, imamo pri *Argon2-u* na voljo tri parametre, s katerimi to nastavljamo. To so:

- Število prehodov, ki jih algoritem naredi čez pomnilnik (večje število pomeni boljšo varnost pred napadi TMTO).
- Količina pomnilnika, ki ga bo algoritem porabil (več pomeni boljšo varnosti pred naprednimi napadalci z optimizirano namensko strojno opremo).
- Število sočasnih niti, ki jih bo algoritem uporabil pri računanju.

Preden lahko nastavljamo te parametre, se moramo odločiti, koliko časa so uporabniki pripravljeni čakati na prijavo v sistem, saj vsi tej parametri vplivajo na čas izvajanja. V viru [12] priporočajo čakanje na prijavo do največ 1 sekunde za spletne aplikacije in do največ 5 sekund za namizne.

Avtorji algoritma priporočajo naslednji postopek pri določanju parametrov. Najprej izberemo število sočasnih niti. To je priporočljivo nastaviti na čim večjo vrednost (toliko kot jih je naš procesor sposoben uporabljati). Nato nastavimo količino pomnilnika. Tudi tu je dobro začeti s čim večjo vrednostjo (toliko kolikor ga ima naš sistem prostega). Nazadnje izberemo število iteracij. Večje število je boljše, vendar moramo razmišljati tudi o času izvajanju, zato to vrednost nastavimo na čim večje možno, tako da je čas

izvajanja še vedno pod zastavljeno mejo. Če čas izvajanja preseže mejo tudi pri eni iteraciji, primerno znižamo porabo pomnilnika, dokler se ne čas ustrezno zmanjša [3].

Algoritem lahko poleg generatorja na spletni strani [10] enostavno uporabljamo preko knjižnic, ki so na voljo za večino programskih jezikov, ali pa ga namestimo v okolje Unix. Sam sem preizkusil zadnjo možnost in si ga namestil iz uradnega repozitorija Github [11]. S spreminjanjem parametrov in upoštevanjem priporočil sem ugotovil, da na virtualnem stroju z 8 GB DDR4 3200MHz pomnilnika in 8 nitmi procesorja AMD Ryzen 5 1600 na operacijskem sistemu Ubuntu 20.04 lahko dosežem naslednje najboljše nastavitve algoritma *Argon2i*, ne da bi čas preverjanja narasel nad 1 sekundo:

- 8 niti
- 2^{18} $kB = 256$ MB pomnilnika
- 2 iteraciji

Čas preverjanja pravilnosti v tem primeru znaša 0,77 sekunde. Za preverjanje vseh treh različic algoritma se porabi približno enako časa.

7 Zaključek

Argon2 je torej hiter, varen in zelo prilagodljiv algoritem za zgoščevanje. Vendar je ravno ta prilagodljivost lahko precej nevarna za manj vešče programerje. Tej morajo dobro poznati delovanje in arhitekturo svoje aplikacije ter se na podlagi tega in dokumentacije algoritma odločiti, katero različico bi bilo primerno uporabiti. Povrh tega pa se morajo znati odločiti tudi za pravilno generiranje skovank, izbiro dolžine zgoščene vrednosti, stopnjo sočasnosti, število iteracij in porabo pomnilnika. Veliko spremenljivk, katere lahko ob nepravilnih nastavitvah ogrozijo varnost sistema.

V prihodnosti bo zaradi algoritma *Argon2* zagotovo prihajalo do manj razkritij gesel ob podatkovnih vdorih, vendar to vseeno ni popolna rešitev za varnost uporabnikov. Za zelo iznajdljive napadalce sama hramba gesel verjetno niti ni največji problem iz stališča varnosti uporabnikov, ker lahko napadalec gesla pridobi tudi na druge načine, s pomočjo socialnega inženiringa. Zaradi tega je zelo pomembna izobraženost samih uporabnikov in za največjo stopnjo varnosti uporabljati večstopenjsko overjanje (angl. multi-factor authentication), ki napadalcu prepreči prijavo v uporabniški račun, tudi če pridobi njihovo geslo.

Literatura

[1] Jean-Philippe Aumasson in sod. "BLAKE2: simpler, smaller, fast as MD5". V: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, str. 119–135.

[2] Daniel J Bernstein. "Cache-timing attacks on AES". V: (2005).

[3] Alex Biryukov, Daniel Dinu in Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". V: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, str. 292–302.

[4] Alex Biryukov in Dmitry Khovratovich. "Tradeoff cryptanalysis of memory-hard functions". V: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2015, str. 633–657.

[5] Dan Boneh, Henry Corrigan-Gibbs in Stuart Schechter. "Balloon hashing: A memory-hard function providing provable protection against sequential attacks". V: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, str. 220–248.

[6] Markus Dürmuth in Thorsten Kranz. "On password guessing with GPUs and FPGAs". V: *International Conference on Passwords*. Springer. 2014, str. 19–38.

[7] George Hatzivasilis, Ioannis Papaefstathiou in Charalampos Manifavas. "Password Hashing Competition-Survey and Benchmark." V: *IACR Cryptol. ePrint Arch.* 2015 (2015), str. 265.

[8] Colin Percival. *Stronger key derivation via sequential memory-hard functions*. 2009.

[9] Jos Wetzels. "Open sesame: the password hashing competition and Argon2". V: *arXiv preprint arXiv: 1602.03097* (2016).

[10] *Argon2*. Dosegljivo: `https://argon2.online`. [Dostopano: 1. 8. 2020].

[11] *Github: Argon2*. Dosegljivo: `https://github.com/P-H-C/phc-winner-argon2`. [Dostopano: 3. 8. 2020].

[12] *Libsodium documentation*. Dosegljivo: `https://libsodium.gitbook.io/doc/password_hashing/default_phf`. [Dostopano: 5. 8. 2020].

[13] *Password Hashing Competition*. Dosegljivo: `https://password-hashing.net/cfh.html`. [Dostopano: 1. 8. 2020].

[14] *Why I Don't Recommend Scrypt*. Dosegljivo: `https://blog.ircmaxell.com/2014/03/why-i-dont-recommend-scrypt.html`. [Dostopano: 9. 9. 2020].

[15] *Wikipedia: Argon2*. Dosegljivo: `https://en.wikipedia.org/wiki/Argon2`. [Dostopano: 2. 8. 2020].

[16] *Wikipedia: Cryptographically secure pseudorandom number generator*. Dosegljivo: `https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator`. [Dostopano: 5. 8. 2020].

[17] *Wikipedia: Key stretching*. Dosegljivo: `https://en.wikipedia.org/wiki/Key_stretching`. [Dostopano: 2. 8. 2020].