# Implementacija lozinki u računalnim sustavima

**Španić, Matija**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Matija Španić**

# IMPLEMENTACIJA LOZINKI U RAČUNALNIM SUSTAVIMA

**ZAVRŠNI RAD**

**Varaždin, 2021**.

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Matija Španić**

**Matični broj: 45064/16-R**

**Studij: Poslovni sustavi**

# IMPLEMENTACIJA LOZINKI U RAČUNALNIM SUSTAVIMA

**ZAVRŠNI RAD**

**Mentor:**

**Doc. dr. sc. Nikola Ivković**

**Varaždin, srpanj 2021.**

*Matija Španić*

**Izjava o izvornosti**

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

*Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi*

_____

# Sažetak

U ovom radu analizirana je implementacija lozinki u različitim operacijskim i ostalim računalnim sustavima. Naglasak je na autentifikaciji kao načinu ulaska u neki sustav. Objašnjeno je kreiranje i spremanje lozinki u operacijske i ostale računalne sustave, kao i manipulacija koja se nad njima može činiti.

Kako bi se lozinke zaštitile od toga da ih napadač može pročitati (u izvornom obliku), sustavi temeljeni na lozinkama koriste kriptografske funkcije sažimanja koje su u radu objašnjene. Isto tako navedeni su i različiti načini kamufliranja tih lozinki pomoću različitih funkcija koje danas postoje.

Nakon funkcija sažimanja koje šifriraju lozinke, objašnjeni su načini kako se te lozinke mogu probiti i kako ih korisnici mogu dodatno osigurati.

Nakon teoretskog dijela slijedi aplikacija kod koje se vrši autentifikacija korisnika. Ona ima mogućnost registracije pa pritom prijave, gdje se podaci korisnika kriptiraju nekim funkcijama. Nakon kriptiranja, podaci registracije spremaju se u bazu podataka. Nakon te aplikacije slijedi još jedna, pomoću koje se pokušava razbiti lozinka nad kojom smo prije proveli kriptografske funkcije.

Ključne riječi: Autentifikacija, lozinka, funkcije sažimanja, sol, napad grubom silom.

# Sadržaj

# 1. Uvod

Lozinka ili zaporka oblik je tajnog zapisa koji je potreban kako bi se moglo pristupiti određenim podacima, informacijama ili sustavima. To je niz znakova pomoću kojih pristupamo željenom sadržaju na računalu ili nekom drugom mediju putem našeg korisničkog računa. On se sastoji od nekog imena pomoću kojega se u nekom sustavu identificiramo kao osoba, te lozinke koja nam služi pristupanju tom korisničkom računu. Lozinke zapravo predstavljaju mehanizam za potvrdu identiteta prilikom autentifikacije, odnosno ona je preduvjet za kontrolu pristupa, gdje npr. brane da netko s našeg računala ne napada nekog drugog. Lozinke su se koristile još u davnim vremenima kada su stanovnici gradova i sela imali određenu komunikaciju među sobom kako bi im život bio što sigurniji. Rimljani su tako naprimjer tijekom smjena stražara, svakodnevno mijenjali kodne riječi kako bi se prepoznao potencijalni autsajder.

Danas, živimo u računalnom dobu u kojemu svaka osoba želi iskoristi tehnologiju za nešto što ju zanima i što joj se pruža. Socijalizacijom ljudi i pojavom društvenih mreža javili su se različiti pristupi sadržajima u kojima sudjelujemo kao individua, koja ima svoj korisnički račun na kojem se nalaze osobni podaci i slično. Od operacijskih sustava, društvenih mreža, platformi, aplikacija, foruma i sličnih mogućnosti koje se mogu koristiti na različitim uređajima, korisnicima tih usluga je omogućeno da imaju svoju bazu podataka („profil“), sa kojim pristupaju različitim sadržajima radi zanimacije, zabave ili posla. Lozinke služe kao provjera valjanosti identiteta osobe koja pristupa nekom sustavu. Svaka osoba je zapravo varijabla u bazi podataka kojoj se pristupa preko korisničkog imena i lozinke. U pravilu bi samo ta osoba trebala imati pravo pristupa svojoj „bazi podataka“, ali u nekim slučajevima može doći do probijanja sigurnosne razine čime naši podaci postaju kompromitirani.

Zbog toga, razvijaju se razni programi koji nam pomažu u očuvanju naših podataka. Ti programi mogu biti implementirati direktno u sustav ili mogu doći kao nekakva vrsta nadogradnje na sustav. Razni operacijski i računalni sustavi koriste različite načine zaštite korisničke privatnosti prilikom instaliranja na računalo. To su većinom sigurnosni programi kojima se pristupa preko lozinke. Može se reći da su današnji takvi sustavi veoma sigurni, no još uvijek bivaju provaljeni. Možemo li se onda pouzdati u sigurnost takovih sustava ako postoji povijest u kojoj su oni probijeni? Odgovor na ovo pitanje nije jednostavan iz razloga što ne postoji stopostotno učinkovite obrane od onih koji žele doći do naših podataka. Iz toga razloga, implementiraju se lozinke u različite sustave kako bi barem otežali posao takvim osobama, koje bi naše podatke koristili u svoje svrhe. Upravo ta uloga lozinki predstavlja: prvu, zadnju,

a u nekim slučajevima i jedinu vrstu obrane od neželjenog zločinca. Pitanje na koje treba dati odgovor je kako napraviti takav sigurnosni sustav koji ima: jaku sigurnost, lagan je za korištenje i težak za probijanje. Neovisno o kakvom se sigurnosnom sustavu radi, on mora ispuniti svoj primarni posao, a to je da zaštiti korisnika.

S toga, lozinke u operacijskim i računalnim sustavima su zapravo prva crta obrane nad kojom se može detektirati, obraniti ili izvršiti napad. Naravno, one nisu svemoguće pa kao i ljudi mogu imati svoje propuste. U nastavku slijedi više o lozinkama u računalnim sustavima, kao barijeri koja sprečava ulazak nelegalnih korisnika.

# 2. Autentifikacija

Već se u samim početcima korištenja računalnih sustava moglo pretpostaviti kako će osiguranje sustava biti neizostavan čimbenik svakog tog sustava. Razvitkom operacijskih i ostalih računalnih sustava paralelno su se razvijali i obrambeni mehanizmi koji bi te sustave činili što otpornijim na napade. Danas, takovih mehanizama ima puno. Od autentifikacijskih metoda, enkripcija informacija, kriptografskih algoritama pa sve do različitih programa koji podatke čine sigurnima. Sigurnosnim mehanizmima je zadaća da samo ovlaštenim osobama omoguće ulazak u sustave, koji se tada služe informacijama dostupnim od strane sustava.

Može se reći kako je autentifikacija jedna od najčešće korištenih metoda zaštite informacijskog sustava. Ona brani ulazak od bilo kakvog neovlaštenog pristupa kako bi informacije ostale sigurne odnosno netaknute od vanjskog utjecaja. Kod autentifikacije s lozinkom se prijavljujemo u sustav sa informacijama koje su poznate korisniku, a to su lozinka i korisničko ime. Autentifikacija se izvršava na principu provjere upisanih podataka i podataka koji su prije bili memorirani u računalu prilikom kreiranja korisničkog računa. Danas je uporaba autentifikacije uobičajena procedura pojedinca, bilo da se radi u osobne ili poslovne svrhe. Jedan od glavnih aktera u tome ima i razvoj mobilne tehnologije, gdje pomoću pametnog uređaja pristupamo raznim aplikacijama i gadgetima koji nam život čine jednostavnijim, pa i zabavnijim.

U ovom poglavlju objašnjena je autentifikacija, kao mehanizam ulaska u neki sustav korištenjem lozinke. Objašnjeno je kako i zašto je potrebna autentifikacija prilikom korištenja operacijskih sustava, kao i načini kako se ona može izvršiti.

## 2.1. Vrste autentifikacija

Prilikom kreiranja korisničkog računa bitna je autentifikacija pojedinca. Autentifikacija se izvršava na principu provjere upisanih podataka i podataka koji su prije bili memorirani u računalu prilikom kreiranja korisničkog računa. To je proces kojim se utvrđuje identitet korisnika. Danas je to neizostavan čimbenik prilikom korištenja distribuiranih informacijskih sustava ili korištenja internetskih aplikacija.

Kao što osoba prepoznaju drugu osobu na temelju karakteristika lica ili boje glasa, tako se računalni korisnici prepoznaju na temelju korisničkog imena i lozinke. Autentifikacijski proces se sastoji od identifikacije korisnika kod koje se provjerava prava pristupa na sustav, te autorizacije kojom se utvrđuju moguća ovlaštenja korisnika nakon identifikacije. Drugim riječima, korisnik se mora identificirati sa korisničkim imenom te autentificirati sa lozinkom.

Uspješna autentifikacija nam omogućuje ono što želimo, a to je korištenje resursa sustava i manipulaciju podatcima.

„Autentifikacija provjerava dali je osoba taj ili ta koja tvrdi da je.“ [2, str 16]. Većina operacijskih sustava kao i ostalih računalnih sustava donosi restriktivne mjere prijava u sam sustav kako bi se određenim subjektima ograničili pristupi, ili pak nelegalnim korisnicima zabranili. Menkus navodi kako se autentifikacija korisnika može se izvršavati korištenjem nekoliko metoda [2, str.14]:

- Lozinka (nešto što znamo) – ako znamo lozinku od korisničkog računa, tada u teoriji smo mi i vlasnici računa. Ipak, može se javiti problem ako izgubimo lozinku ili ju nam netko ukrade.
- Tokeni (nešto što posjedujemo) – to se odnosi na razne predmete koje posjedujemo a kojima pristupamo određenim sustavima. Kao što u kućanstvu vrata otvaramo ključem kojeg samo mi posjedujemo, tako u neke računalne sustave možemo ući sa posebnim alatima koje sustav prepoznaje. Kao i kod lozinki, ovaj način ulaska u sustav se može kompromitirati na način da se ukrade ili duplicira „token“.
- Osobne karakteristike (identifikacijske oznake osoba) – sustavima pristupamo na način da se preko tjelesnih karakteristika prijavimo u sustav kao: otisci prstiju, potpis, prepoznavanje pomoću glasa, prepoznavanje očiju i slično. Iako su ovakvi sustavi u većini slučajeva točni i pouzdani, nisu puno rasprostranjeni zbog financijskih izdataka koje donose.

Na temelju ovih podjela može se reći da postoje načini identifikacije pomoću lozinki, tokena ili biometrijskih karakteristika koji su nadalje objašnjeni.

### 2.1.1. Autentifikacija pomoću lozinki

Kao što je već spomenuto, danas je korištenje lozinke u svrhu prijave u sustav jedno od najčešćih mehanizama zaštite. U sustavima baziranim na prijavu preko lozinke, potrebno je prvo napraviti korisnički račun koji se sastoji od ID-a osobe, odnosno identifikacijskog obilježja (ime pod kojim ćemo se prijavljivati) i lozinke korisnika. Na takvu autentifikaciju nailazimo prilikom spajanja na Google račun, mail, društvene mreže i slično.

Pod autentifikacije pomoću lozinke smatra se da je to nešto što znamo. Drugim riječima, samo mi znamo lozinku kojom smo se prvobitno registrirali u sustav. Prilikom izbora lozinki trebali bi se pouzdati u smjernice koje nam taj sustav pruža kao: određena količina

alfanumeričkih znakova, korištenje velikih i malih slova, brojeva i znakova. Ako se ovi aspekti uzimaju u obzir prilikom kreiranja lozinki, tada bi ona trebala biti snažna te otporna na potencijalne napade. Poželjno je da se za različite sustave koje koristimo, koriste i različite lozinke. To nekada može biti korisnicima naporno i teško za zapamtiti, ali u suprotnom ako napadač sazna lozinku iz jednog sustava, može ju koristiti kao resurs za napade i na druge. Iz toga razloga, javili su se razni alati koji pomažu pri radu sa lozinkama kao alati za upravljanje lozinkama (*eng. Password manager*). Upravljanje lozinkama omogućuje korisnicima organiziranje lozinki. To su zapravo programi koji skladište lozinke u bazama podataka i kriptiraju ih kako bi sustav bio sigurniji. Postoje nekoliko vrsta takovih programa kao što su:

- Upravitelji za radnu površinu gdje se podaci pohranjuju na čvrsti disk računala
- Portabilni upravitelji koji se koriste kod pohrane podataka na pametnim uređajima
- Online upravitelji koji omogućuju spremanje i kriptiranje lozinki na poslužitelju pomoću algoritama.

Primjeri takvih komercijalnih alata koji su danas u upotrebi su: *Password safe, KeePass, Password Gorilla, RoboForm, Sxipper*. Većina ovakvih alata kompatibilna je sa svim poznatijim operacijskim sustavima, a kriptirani su Blowfish algoritmom o kojemu će biti riječi kasnije kao i o implementacijama lozinki u računalnim sustavima. Autentifikacija je još dodatno objašnjena i u aplikaciji koju sam napravio kao praktični zadatak.

### 2.1.2. Autentifikacija pomoću tokena

Tokeni se koriste kod naprednijih sigurnosnih sustava kao autentifikacijski mehanizam. Prema Woodu to bi bilo nešto što posjedujemo. To je zapravo fizički uređaj kojim korisnik identificira sebe kao ona za koju se predstavlja. Oni se dijele prilikom prvobitne registracije na neki sustav.

Tokeni se zapravo koriste još od davnine gdje je uobičajeno bilo nositi oznake (prsten, broš, lančić, narukvica) kojima se dokazivalo da glasnici koji prenose kraljeve poruke, zaista budu njegovi poslanici [3, str. 166.]. Danas se oni koriste svakodnevno, kao naprimjer kod uzimanja novaca iz bankomata, davanju osobnih isprava nadležnim tijelima za identifikaciju, pa i u poslovne svrhe gdje se bilježi radno vrijeme zaposlenika pomoću njih.

Uobičajeno korištenje tokena se odvija u dva koraka. Prvi korak je taj da se token stavi u ili na neki uređaj te se pomoću njega dođe do autentifikacije. Tada sustav zna o kojoj se osobi radi. Ako sustav prepozna token tada slijedi uobičajena identifikacija gdje korisnik unosi identifikacijsku oznaku i lozinku. U nekim slučajevima je potrebna samo lozinka ako sustav

prepozna korisnika, a u nekim nije potrebna ni lozinka već odmah nakon prvog koraka ulazimo u sustav. Ukoliko osoba uspije proći ove korake, tada ima pristup sustavu. Pod token mogu spadati pametne kartice ili USB token, ili razne druge stvari koje u sebi sadrže neke atribute pomoću kojih ih računalni sustav može detektirati. Problem ovakve vrste autentifikacije je taj što ukoliko se izgubi ili ukrade token, može doći do probijanja u sustav.

### 2.1.3. Biometrija

Još jedna tehnika autentifikacije aktualna u današnjem svijetu je korištenjem biometrijske tehnologije. Prema Websteru biometrija je: „Grana biologije koja se statistički bavi podacima i kvantitativno ih analizira“ [4].

Biometrijska autentifikacija u računalnim sustavima automatski identificira korisnike na temelju fizičkih karakteristika osobe kao naprimjer otisak prsta. Preko sustava koji je napravljen od senzora, mjere se karakteristike odnosno „predispozicije“ korisnika i uspoređuju se sa onima pod kojim bi trebao imati pravo ulaska u sustav. Postoje dva načina autentifikacije pomoću biometrije. Prvi način se bazira na fizičkim karakteristikama osobe (izgled), to jest na principu geometrije tijela kao naprimjer otisak prsta ili geometrija lica. Drugi princip se odnosi na ponašanje korisnika kao što je: glas korisnika, njegov potpis, način udarca na tipkovnicu, te svi ostali oblici ponašanja na temelju kojih sustav može prepoznati da se radi o pravom korisniku. Postoji nekoliko načina autentifikacije pomoću biometrije te se oni koriste ovisno o potrebama koje zahtijevaju sustavi. Tako postoje autentifikacija pomoću: prepoznavanja lica, otiska prsta, skeniranja oka, skeniranja glasa i slično. Iako ovaj način autentifikacije korisnika podiže razinu sigurnosti sustava, novčano je skupa za primjenu pa se ne koristi toliko često u operacijskim sustavima.

# 3. Implementacija lozinki u operacijskim sustavima

Za informacije se može reći da su jedne od, ako ne i najbitnije stavke koje se koriste prilikom korištenja nekog pametnog uređaja. Uz što veći broj korisnika koji mogu pristupati informacijama, javila se potreba da se takve informacije zaštite kako ne bi došlo do nepotrebnog curenja istih. Iz toga razloga, implementiraju se sigurnosni sustavi pomoću kojih korisnici sigurno pristupaju željenom sadržaju. Tako je prvi takav sustav, kod kojeg se prvi puta javila zaštita korisničkih podataka, napravio Dr. Fernando Corbato,. Njegov operativni sustav s podjelom vremena napravljen na američkom sveučilištu MIT (*eng. Massachusetts Institute of Technology*), zvan CTSS (*eng. Compatible Time – Sharing System*), omogućio je da računalu bude dodijeljeno više korisnika istovremeno. Taj sustav se koristio u znanstvene svrhe za izračunavanje matematičkih problema, a kako se u početku mogao dati pristup samo jednoj osobi, on je podijelio snagu procesora na više dijelova kako bi se više poslova moglo izvoditi istovremeno, odnosno više osoba je moglo koristiti sustav istovremeno. Taj podijeljeni sustav napravljen je na način da svaki korisnik mora unijeti oznaku pod kojom se prijavljuje, te lozinku. Fernando je za Engleske novine BBC izjavio kako: „Stavljanje lozinki za svaku individualnu osobu čini se kao jedino normalno rješenje“ [5]. Već onda se moglo pretpostaviti da će se u budućnosti morati čuvati integritet podataka kako oni ne bi „došli“ u krive ruke.

Rizik ulaska neadekvatnog korisnika mora se spriječiti bez obzira na financijske troškove koje nose takove implementacije. Iz razloga što se želi zaštiti privatnost korisnika pa time i sami podaci koje sustav nosi, implementiraju se lozinke u operacijske i računalne sustave kao barijera od neželjenih ulazaka u sustav. Prednosti lozinki su takve da nisu skupe, lako se implementiraju i imaju podršku operacijskih sustava. One bi trebale biti fleksibilne, pouzdane, sigurne i jednostavne za upotrebu [6].

U nastavku se baziramo na jednostrukim lozinkama kao načinom zaštite operacijskih i računalnih sustava. Iako lozinke predstavljaju barijeru kojom se sprečava ulazak ilegalnim korisnicima, postoje situacije gdje ipak dođe do ilegalnog ulaska u sustav. Kod prvobitnog kreiranja računa, lozinka koju smo unijeli, negdje mora biti spremljena. One su najčešće spremljene u bazi podataka koja se koristi u tom sustavu. Kada unesemo lozinku kojom se prijavljujemo u sustav, tada se taj unos koji smo unijeli, uspoređuje sa onim pohranjenim u bazi podataka. Ukoliko se unosi poklapaju, tada dolazi do ulaska u sustav. Zato je prvo potrebno objasniti kako se te lozinke kreiraju, te spremaju u bazu podataka i na koji način su one tamo zapravo pohranjene, te kako se one tamo mogu pronaći. Ovisno o računalnom sustavu, manipulacija lozinkama je drugačija.

## 3.1. Kreiranje lozinki

Postoji veliki broj načina kako se lozinka može implementirati u sustav. Tako ima i raznih tipova lozinke koje se spominju u različitim literaturama. Implementacija lozinki zapravo se zasniva na primarnim i sekundarnim lozinkama. Primarne se koriste kod autentifikacije na računalnim sustavima kroz operacijski sustav. Sekundarne su specifične i one se koriste prilikom pristupa osjetljivijim podacima unutar sustava. Menkus je naveo kako se lozinke mogu okarakterizirati na dva načina upotrebe: generalizacijsko i korisničko. Generalizacijske uključuju sustav, korisnika i sistemsko okruženje dok korisničke uključuju primarnu, sekundarnu i takozvanu „prisilnu“ lozinku [7]. Lozinke se u različitim informacijskim sustavima različito i primjenjuju. Ovdje će biti fokus na primarnim lozinkama pa ćemo načine kreiranja lozinke podijeliti na: sustavno, tvornički i korisnički generirane lozinke.

### 3.1.1. Sustavno generirane lozinke

Sustavno generirane lozinke (*eng. System – Generated Passwords*) su lozinke kojima upravlja sustavno sigurnosni upravitelj. Uloga upravitelja je da kreira i selektira nove lozinke te ih distribuira bazi podataka. Također, nadgleda korisnika kako bi se osiguralo pravilno korištenje lozinke kao i uklanjale stare odnosno istekle lozinke [7]. Takova vrsta lozinke generirana je programima za izbacivanje nasumičnih brojeva, kao i ostalim programima koji nasumično generiraju pa upisuju nesmislene oznake. Ovaj način generiranja lozinke spada u generalizacijski način upotrebe.

Prednost takovih sustava je u njegovoj nepredvidivosti za potencijalne napadače. One se sastoje od nasumičnih znakova koji ne moraju na ni jedan način biti povezane sa korisnikom i njegovim razmišljanjem glede kreiranja lozinke.

Nedostatci sustavno generiranih poruka su uglavnom vezani oko korisničkog pamćenja lozinke. One mogu biti nesmisleno postavljene pa su time teže za memorirati. Može doći do toga da se ista lozinka nekim slučajem ponovi ali to je ipak rijedak slučaj.

### 3.1.2. Tvornički generirane lozinke

Tvornički generirane lozinke (*eng. Manufacturer – Generated Passwords*) su lozinke koje se dobije bez uloge korisnika prilikom njihova odabira. Umjesto korisnika, zaporku bira sustav koji to obavlja preko nekog algoritma. To su nesmislene lozinke koje korisnik dobijemo prilikom prve upotrebe te se kasnije mogu izmijeniti. One se upotrebljavaju kada upravitelji sami kreiraju korisnički račun na zahtjev korisnika. Takve lozinke se većinom koriste kod sustava u kojima je potrebna stalna ljudska nadgledanost. To može biti na principu cijelog sustava gdje naprimjer tehničari, kojima bi bilo naporno da koriste različite lozinke u različitim

razinama sustava kada moraju raditi preinake kod održavanja nekog informacijskog sustava. Isto tako se u većini obrazovnih ustanova na principu nekog obilježja osobe (ime, prezime, grupa), osoba dobije smisleno ime te njegova lozinka bude generirana od strane sustava.

Nedostatak je taj da sustav ipak zna za lozinku pa bi ju valjalo promijeniti prilikom korištenja, a u slučaju da ju tehničar sazna, lozinka bi se mogla proširiti dalje na druge osobe te bi sustav mogao postati kompromitiran.

### 3.1.3. Korisnički generirane lozinke

Ova način (*eng. User-Generated Passwords*) je ujedno i najlakši način implementacije lozinki. Lozinke se implementiraju od strane korisnika na način da se prilikom samog kreiranja u obzir uzimaju podaci korisnika, njemu veoma poznati. Takva vrsta lozinki je danas najčešća. Većina subjekata za lozinke uzimaju nešto što neće tako lako zaboraviti, kao naprimjer: nadimci, datumi rođenja, poznatu osobu ili slično. Na taj način nesvjesno koristimo lozinku laganu za pogoditi, koja samim time i našu privatnost izlaže opasnosti. Opasnost može biti prisutna pri korištenju društvenih mreža, a o sustavima gdje obavljamo novčanu transakciju da i ne pričamo.

Koliko je ovakav način kreiranja lozinke loš, govore istraživanja koje su Morris i Thompson proveli prilikom ovakvog načina implementacije lozinki [8, str. 596]. Rezultati su bili poražavajući. Od 3289 ispitanika, čak u 86 posto njih lozinke su se mogle provaliti surovim pogađanjem, dok u ostalih 14 posto lozinke su pogođene *brute force* napadom koji je objašnjen u razbijanju lozinki.

Prednosti koje postoje kod ovakvog načina kreiranja lozinke su te da su lagane za zapamtiti dok su mane upravo to što su prejednostavne (trivijalne). Radi nedostataka koje pružaju ovakve lozinke javljaju se ipak neki način kako ih zakomplicirati, a da budu pamtljive. U nastavku slijedi nekoliko takovih načina [8].

- Parafraziranje - Parafraza je način izražavanja gdje neku rečenicu uzmemo i prepričamo ju svojim riječima. Drugim riječima, prepišemo neki tekst u kojem možemo a i ne moramo dodati riječi, a korisnik će tako lakše zapamtiti lozinku. Parafraziranjem možemo izabrati lozinku koja je lagana za zapamtiti a teška za pogoditi. Što je takva rečenica duža, teže ju je za pogoditi. To može biti citat iz neke knjige, filma ili sličnih Vama poznatih rečenica. Najučestalije se koriste riječi pjesama, gdje se za lozinku uzme stih ili slično.

- Kognitivne lozinke - način autentifikacije u sustav gdje se korisnika pita niz pitanja na koje samo oni znaju dati odgovor. Prilikom kreiranja računa korisnik odgovara na nekoliko pitanja koja se prilikom ulaska u sustav slučajnim odabirom postavljaju. Ukoliko

korisnik odgovori točno, dozvoljen mu je ulazak u sustav. Ta pitanja su većinom vezana uz osobni život korisnika, kao naprimjer: ime prvog kućnog ljubimca, djevojačko prezime majke, ime prve ljubavi i slično. Takve lozinke su pogodne za korisnika jer je mala vjerojatnost da se nećemo sjetiti odgovora o stvarima iz svojeg života. Za probijanje takvih lozinki trebalo bi privatno znati osobu što smanjuje mogućnost probijanja zaporki. Osim kao lozinka za ulazak u sustav, ovaj način upotrebe lozinka se upotrebljava kao i backup. Često se ovaj način autentifikacije može naći kao pomoćni alat nekog računalnog sustava pri zaboravi lozinke. Tada nas se onda pita od strane sustava da odgovorimo na takve odgovore kako bi on prepoznao da smo to stvarno mi te nam odobrio zahtjev za zamjenom lozinke. Većina društvenih mreža uspostavlja upravo ovakva pitanja samo radi tog backupa, zbog čestih zaborava lozinki.

Jačina lozinki je stavka koja se uzima u obzir prilikom kreiranja lozinki. Sadržaji kojima se pristupa na nekom računalnom sustavu žele se zaštiti od neželjenih korisnika kako bi i sama zaštita privatnosti bila neupitna. Ona nam zapravo govori o kompleksnosti sadržaja kojega stavimo kao lozinku, a to mogu biti: slova, brojevi i/ili znakovi. Jačina lozinke najviše ovisi o ljudskom faktoru koji zapravo odlučuju o njezinom nazivu. Iz toga razloga, jačinu lozinki možemo podijeliti na dva tipa: ona može biti ili jaka ili slaba. Također, trebali bi se pridržavati nekih smjernica prilikom izbora lozinke kao što su: minimalna količina znakova kao i njihova kombinacija, sposobnost memoriranja lozinke (koja najviše ovisi o mentalnim sposobnostima korisnika), kao i korištenje različitih lozinki za različite sustave. Operacijski kao i ostali računalni sustavi sami postavljaju uvjete koje lozinka mora sadržavati prilikom autentifikacije, ovisno o osjetljivosti podataka koje taj sustav nosi.

## 3.2. Spremanje lozinki u računalnim sustavima

Prilikom kreiranja korisničkog računa, podaci koji su se unijeli kao ime i lozinka spremaju se u bazu podataka. Kada se prilikom autentifikacije unesi ti podaci kako bi se prijavili u sustav, uspoređuju se sa onima koji se nalaze u bazi podataka te se dopušta ulazak u sustav ukoliko postoji takav račun sa tom lozinkom. Nažalost, postojanje lozinki u bazi podataka predstavlja prijetnju sustavu. Moguća otkrivanja takovih podataka iz baze podataka omogućuju zlonamjernim korisnicima uvid u korisničko ime i lozinku. Iz tog razloga, smišljeni su neki načini kako bi otežali poslovi potencijalnim napadačima. Lozinka u bazi podataka može biti pohranjena na tri načina: kao otvoreni tekst, šifrirana ili sažeta lozinka [9, str. 13 i 14].

- Otvoreni tekst (*eng. Plaintext password*) – najjednostavniji i najrašireniji način spremanja lozinke je otvorenog teksta. To znači da je lozinka spremljena u bazu podataka na način da ju osoba može pročitati i shvatiti njezin sadržaj. Naprimjer, lozinka „test123“ bi ovakvom metodom spremanja lozinke izgledala upravo tako i u bazi podataka, „test123“. Ovo je najgori način spremanja zaporki što se tiče sigurnosti. Veliki broj uglednih sustava kao i web stranica ne koristi ovaj način prilikom pohranjivanja lozinki jer ukoliko dođe do napada, napadač vrlo jednostavno dobije pristup lozinkama, pa time nije samo jedan korisnik kompromitiran, već svi.

- Šifrirana lozinka (*eng. Encrypted password*) – da bi se unaprijedio zapis pohranjene lozinke kako ona ne bi bila zapisana identično kao u bazi podataka, sustavi su se morali prilagoditi sigurnosnim zahtjeva, pa su se lozinke kao takve počele spremati u obliku šifri. Kod šifriranja se koristi takozvani tajni ključ kako bi se otvoreni tekst transformirao u slučajni zapis brojeva, znakova i slova. Drugim riječima, lozinka koja se prije spremala onakva kakva zaista je, sada je spremljena potencijalnim napadačima na nečitak tekst. Sada je ona vidljiva samo u šifriranom stanju, a ne originalnom. Problem kod ove metode je taj da ukoliko napadač uspije dešifrirati tajni ključ, koji je uobičajeno spremljen na računalni server ili na server gdje se nalaze lozinke (baze podataka), tada ukoliko napadač uspije dešifrirati jednu lozinku, može dešifrirati sve u tom sustavu. Iz toga razloga, ova metoda takozvanog skrivanja lozinke nije sigurna za današnje računalne sustave.

- Sažeta lozinka (*eng. Hashed password*) – *hashing* radi na sličan način kao i šifrirana lozinka. Prilikom spremanja, zaporka se pohranjuje u slučajan niz znakova, brojeva ili slova fiksne dužine. Za razliku od šifriranih lozinki, kod funkcija sažimanja, lozinka se ne može dekriptirati odnosno vratiti u prvobitni oblik. Više o tome kako funkcioniraju i za što točno služe objašnjeno je u sljedećem poglavlju. U radu će se često pojavljivati *hash* i *hashing* umjesto sažimanja lozinke radi jednostavnosti.

Operacijski i ostali računalni sustavi ogradili su se oko sigurnosti korisnika na način da se lozinke šifriraju te spremaju na mjesta ovisno o sustavu koji se koristi. Prilikom ulaska, operacijski sustav „šifrira“ lozinku te ju uspoređuje sa onom spremljenom, kako bi se dopustio ili ne dopustio ulazak u sustav. Sustavi se razlikuju jedan od drugog u načinima spremanja i dešifriranja lozinki, te koriste različite programske alate prilikom baratanja sa njima. Neki od njih su poprilično sigurni dok su neki manje. Razvijeniji sustavi jednostavno ne smiju pohranjivati podatke u sustav kao tablicu korisničkih imena i lozinki. Čak i kada bi se ta tablica zaštitila na neki način, informacije će u nekom trenutku procuriti. Isto tako, zaporke nije

poželjno pohraniti samo u šifriranom obliku jer se one ipak može dešifrirati. Upravo zbog toga, trebamo jednosmjerne *hash* funkcije kod kojih se ne može iz *hasha* izvući prvobitni input. U pravilu, operacijski sustavi nemaju spremljenu originalnu lozinku, već oni samo izvode matematičke izračune nad unosom, te ga uspoređuju sa *hash* vrijednosti koja se nalazi u bazi podataka. U nastavku je objašnjeno kako operacijski sustavi „kamufliraju“ lozinku, te kako možemo vidjeti gdje su oni na računalu smješteni.

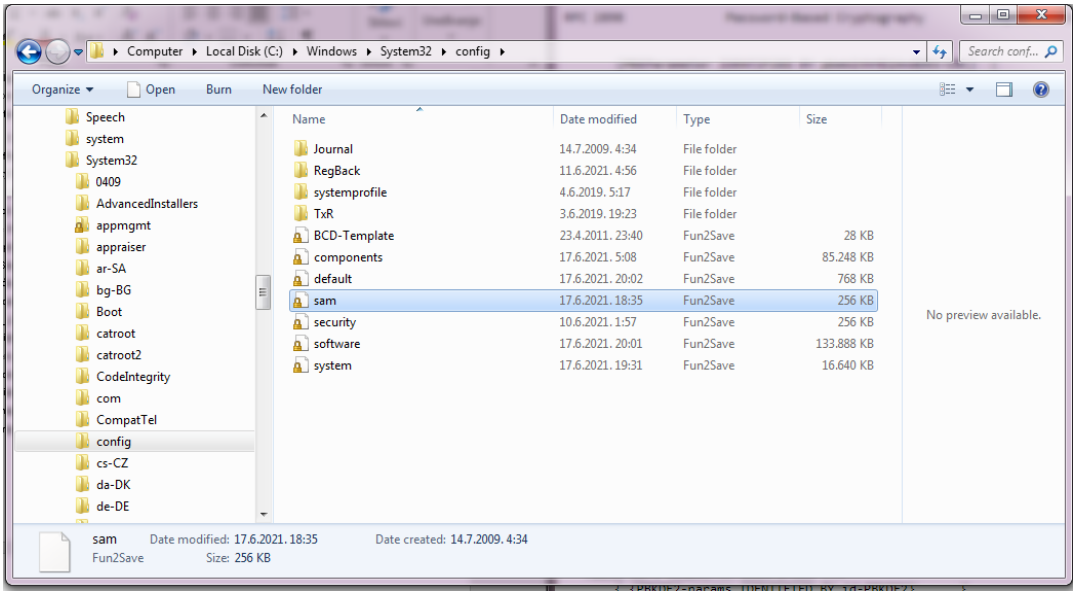
### 3.2.1. Windows OS

Windows operacijski sustavi jedni su od korisnički najčešće korištenih operacijskih sustava. Autentifikacija se zasniva na unošenju korisničkog imena i lozinke. U starijim inačicama ovog sustava, lozinke su bile spremljene u jednostavnom tekstualnom obliku. One su prvobitno bile spremane u LanManager formatu, takozvanom LANMAN-u. Struktura ovog algoritma sadržavala je lozinku sa samo 14 znakova, s time da su sva slova morala biti velika te nije sadržavala takozvanu sol, koja je kasnije u radu detaljnije objašnjena. Zbog sigurnosnih propusta koje je sadržavala, danas više nije u upotrebi.

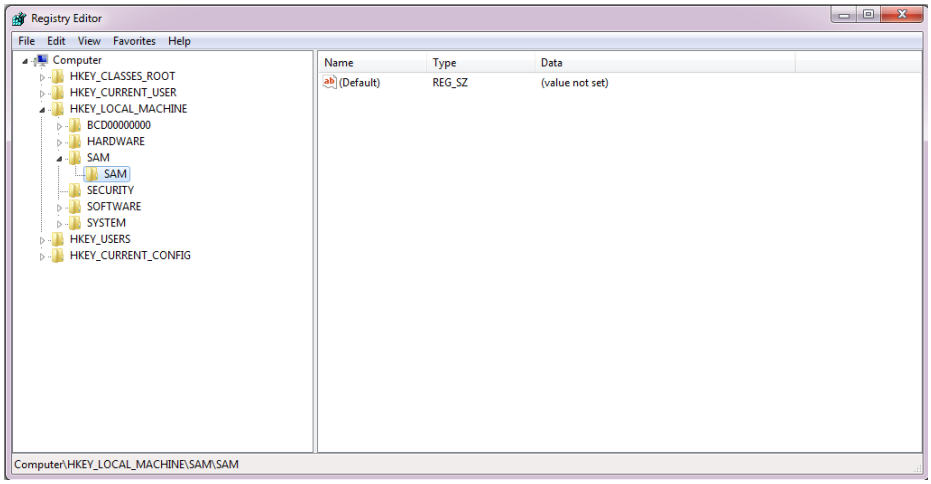
U Windows operacijskim sustavima postoji set protokola koji se izvršavaju prilikom autentifikacije korisnika. Oni uključuju Kerberos i TLS/SSL (*eng. Transport Layer Security/Secure Socket Layer*). Novije verzije Windows operacijskih sustava kao: *2000, XP, Vista, 7 i 10,* koriste još i NTLM (*eng. New Technology LAN Manager*). To je Microsoftov autentifikacijski protokol koji je naslijedio LM. Protokol uzima lozinku te ju kriptira MD5 algoritmom. Sada, lozinke nisu osjetljive na velika slova te mogu ići do dužine od 127 znakova. Nakon provedbe NTLM protokola, lozinka se sprema kao *hash* vrijednost u sigurnosnu bazu podataka – SAM (*eng. Security account manager*) [10, str.2]. SAM je smješten na disku na kojem se nalaze Windowsi. Najčešće se nalazi u: C:\Windows\system32 datoteci koja je zaključana tijekom rada Windowsa. Prilikom ponovnog ulaska u sustav, korisničko ime i lozinka se uspoređuju sa onima koji se nalaze u SAM-u. Lozinka se uspoređuje na način da se lozinka koju unosimo kod prijave, algoritmom transformira u *hash* vrijednost, te uspoređuje sa onom *hash* vrijednosti koja se nalazi u SAM-u. Osim *hash* vrijednosti pohranjenih na particiji diska, postoji verzija pohranjena u samom registru, naziva HKEY_LOCAL_MACHINE\SAM, koji sadrži informacije o korisničkom računu kojeg je također nemoguće pročitati prilikom rada operacijskog sustava. Također, jedna kopija lozinke pohranjena je u Windows-directory\repair folderu, ali kao i kod prijašnjih, nema mogućnosti za čitanje [10, str.2]. Ukoliko zlonamjerni korisnik uspije kopirati vrijednost pohranjenu unutar SAM-a na portabilni disk, USB ili CD, ili pak preko drugog operacijskog sustava uspije izvući te podatke, tada može saznati *hash* vrijednost lozinke nad kojom onda može izvršiti napad. Isto može napraviti i korisnik koji je

zaboravio lozinku pa koristi ove metode kako bi probao ući u sustav. Sada, pomoću nekog programskog alata za razbijanje lozinke možemo saznati istu na temelju tih *hash* vrijednosti.

Kako ne bi stalno morali upisivati ime i lozinku prilikom pristupa nekim sustavima, Windowsi prilikom autentifikacije korisnika u različite sustave spremaju detalje prijave u takozvani Credential Manager. To je zapravo skrivena datoteka u računalu u koju se spremaju podaci prijave, te ih operacijski sustav tada koristi prilikom prijave na mrežu, server ili internetske stranice. Autentifikacijske podatke koje sadrže „credentiali“, nalaze se u takozvanom Windows Vault-u (trezor), te su ti podaci šifrirani kako bi se zaštitili. Tu se koriste tri načina za spremanje tih podataka: Windows Credentials za spremanje korisničkih imena i lozinki kod prijave na internet i internetske stranice; Certificate – Based Credientials za uređaje koji spajamo na računalo; Generic – Based Credentials za Microsoft programe koji se mogu koristiti i zaštiti, kao: *word*, *powerpoint*, *outlook* i slično.




Slika 1: SAM datoteka pohranjena u memoriji




Slika 2: SAM datoteka pohranjena u registru

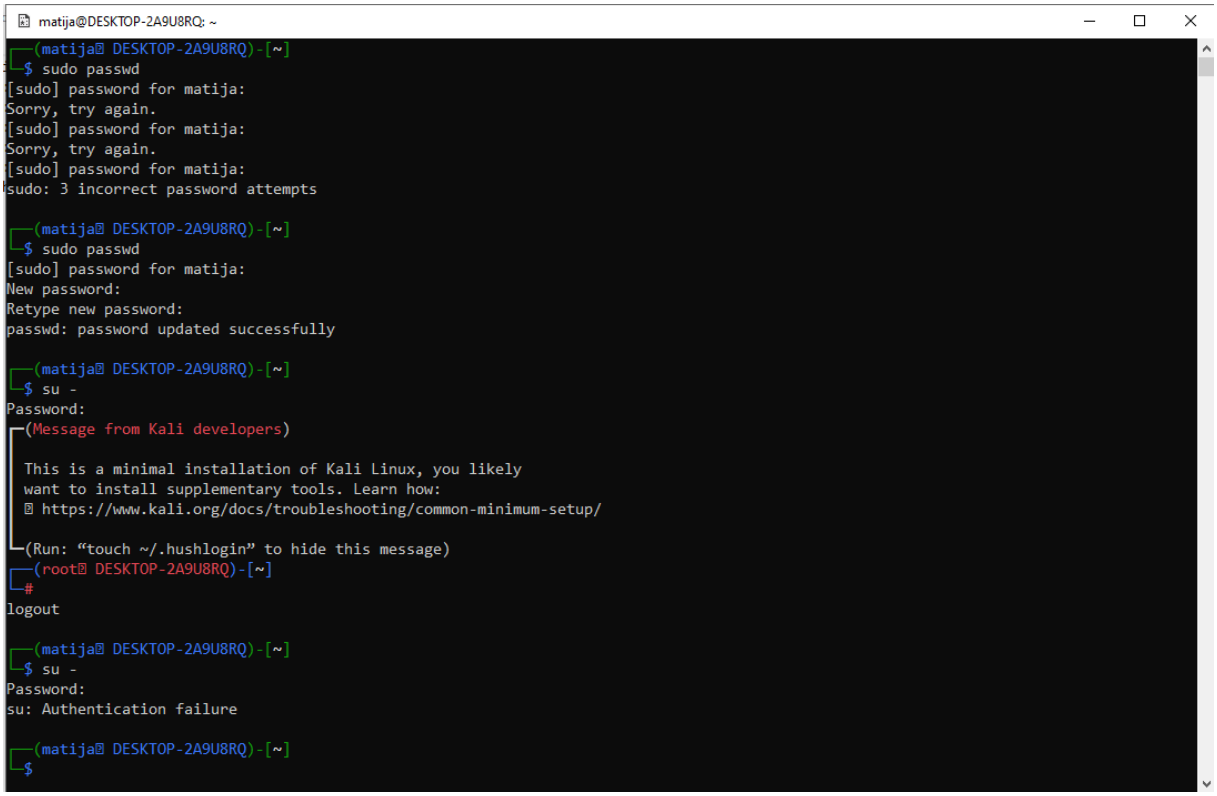
### 3.2.2. Unix/Linux OS

Modernije verzije Linuxa danas koriste priključne autentifikacijske module kako bi se spojili u bazu podataka. Najčešće se koristi zadana verzija modula PAM (*eng. Pluggable Authentication Module*). Početne inačica ovih operacijskih sustava spremale su lozinke u kriptografskom obliku u takozvane */etc/passwd* datoteke [10, str.2]. Tu se nalaze informacije o korisničkim računima koji se nalaze u sustavu. Lozinke su se kriptirale jednosmjernom crypt() funkcijom te se tako spremale u taj folder. Crypt funkcija je modifikacija DES algoritma (*eng. Data Encryption Standard*) koji je prihvaćen standard Američke vlade za zaštitu podataka, te se koristi prilikom SSL-TLS protokola [11]. Ovo je prilično sigurno kriptirana verzija lozinke gdje se ona može razbiti samo *brute force* napadom.

Moderni Linux sustavi, danas koriste */etc/shadow* ili */etc/master.passwd* foldere za spremanje *hash* vrijednosti kako bi ih samo sustav mogao pročitati, ili čak korisnik pomoću glavnog (*eng. root*) računa. Shadow datoteka zapravo sadrži informacije o korisniku. To su korisničko ime, sol te kriptirana lozinka. Kod lozinke se nude informacije o tome kada je zadnji puta promijenjena, vrijeme kada se smije promijeniti, vrijeme kada je potrebna promjena, vrijeme kada ističe, te takozvano rezervirano vrijeme. Osim ovih dviju datoteka, prilikom registracije korisnika, može se kreirati još jedna datoteka, */etc/group*, koja definira grupe u sustavu.

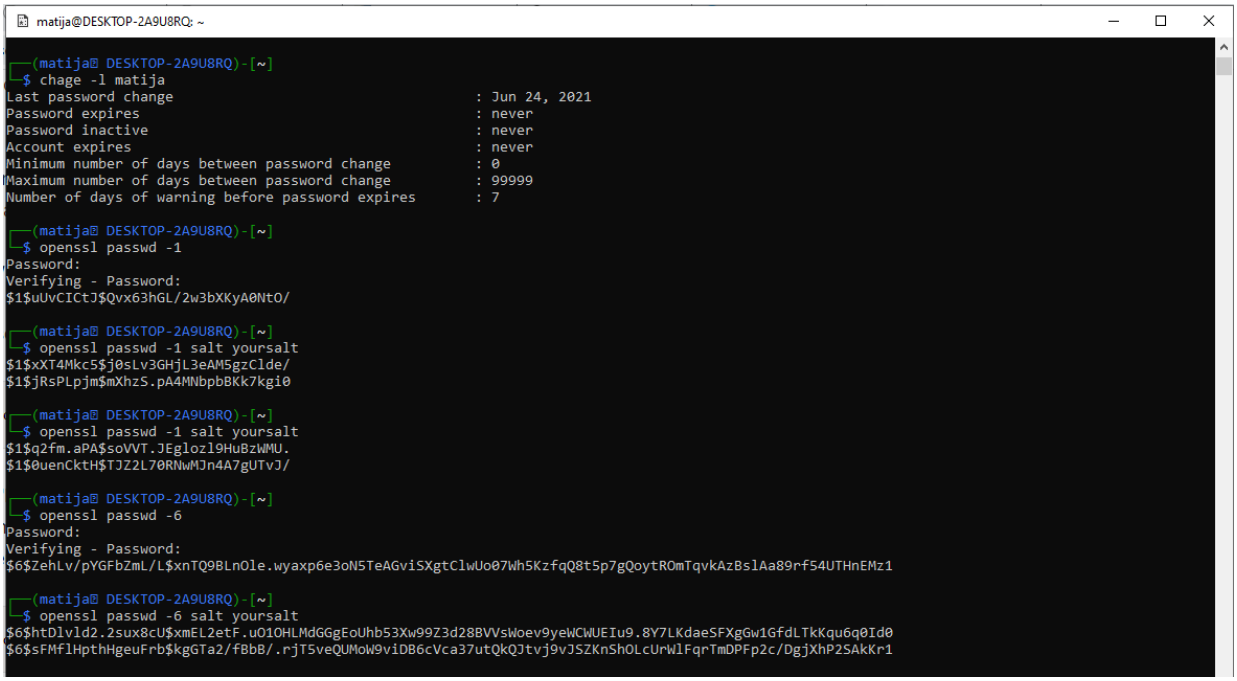
Važna stavka ovih operacijskih sustava je ta da se prilikom kriptiranja lozinke dodaje takozvana sol (*eng. salt*). Ona nije tajno pohranjena već se nalazi zajedno sa lozinkom, a ti podaci mogu se pročitati ukoliko smo glavni korisnik sustava. Ukoliko se radi o DES algoritmu, prilikom početnog kreiranja lozinke izvodi se 25 iteracija u algoritmu kako bi se dobila lozinka u šifriranom obliku što pojačava sigurnost sustava. U pravilu, administratori ovih sustava trebali

bi koristiti MD5 ili Blowfish umjesto zadane DES funkcije, malo su sporije pa time i sigurnije za baratanje lozinkama, što je detaljnije objašnjeno u funkcijama sažimanja. Lozinka u Shadow folderu na Linux sustavu izgleda kao $id$sol$hash. Id se odnosi na algoritam koji koristimo prilikom kriptiranja lozinke, kao naprimjer: za MD5 algoritam koristimo $1$, za Blowfish $2a$ i $2y$, za SHA-256 $5$, za SHA-512 $6$ [12]. Nakon kriptiranja lozinke nekom funkcijom sažimanja, dodaje joj se sol i nakon toga se sprema u sigurnosnu datoteku kao *hash* vrijednost.




Slika 3: Mijenjanje lozinke glavnog korisnika u Kali Linux-u

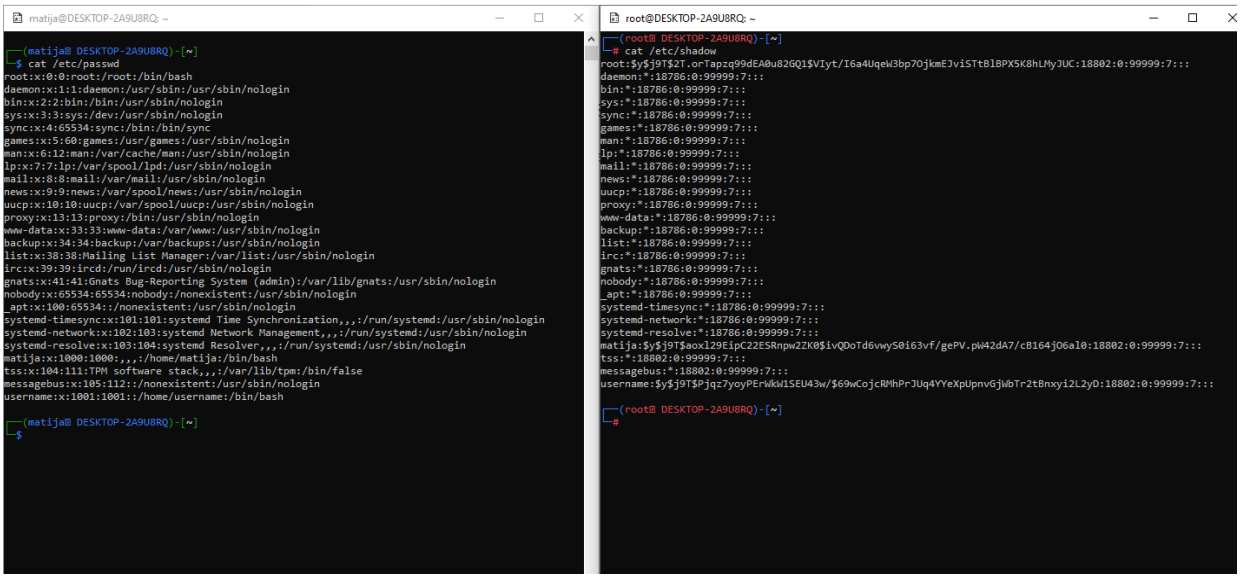
Da prikažemo kako izgledaju te datoteke u koje se spremaju lozinke, pokazani su načini kako se ulazi u sustav i kako se može vidjeti i *hashirati* lozinka. Ukoliko želimo promijeniti lozinku glavnog korisnika, to možemo uraditi preko *sudo passwd* funkcije. Prije nego unesemo novu lozinku, moramo potvrditi staru za što imamo 3 pokušaja, u protivnom moramo ponovno unijeti funkciju. Ukoliko je autentifikacija uspješna, vrati nam se potvrda o ažuriranju lozinke. Sada, kako bismo ušli kao glavni korisnik, koristi se *su* – naredba koja nas pita za lozinku, te omogućava ulazak ovisno o uspješnosti prijave.




Slika 4: Informacije o lozinki, sažimanje i dodavanje soli na lozinku

Linux nudi opciju uvida u stanje lozinke sa naredbom *chage –l username* (ime korisnika). Tom naredbom možemo vidjeti kada smo zadnji puta promijenili lozinku, kada istječe, kada će postati neaktivna, kada se gasi račun, minimalan i maksimalan broj dana između promjene lozinke, te broj dana kada dođe obavijest prije isteka iste.

Isto tako, lozinke se mogu *hashirati* na nekoliko načina naredbama *openssl passwd –broj* (*eng. number*), gdje broj predstavlja algoritam koji želimo koristit. U ovom primjeru koristi se broj 1 za MD5 algoritam i broj 6 za SHA512 algoritam koji zauzima 64 bajta. Kako bi dodali još jedan sloj sigurnosti, na ovu naredbu možemo dodati još i sol. Sada je lozinka kompliciranija i otpornija na napade *rainbow* tablicama.




Slika 5: Passwd i Shadow datoteke

Lozinke imaju mogućnost pohrane u dvije datoteke. Ukoliko smo obični korisnik, tada datoteku možemo samo iščitati, ne znajući vrijednost lozinke. To možemo napraviti naredbom *cat /etc/passwd* (slika 5). Ukoliko smo glavni (*eng. root*) korisnik, tada možemo saznati i lozinku sažetu algoritmom, naredbom *cat /etc/shadow*, koja prikazuje takozvani shadow folder koji se također nalazi u bazi podataka. Sva polja su odvojena dvotočkom (:), te tu možemo vidjeti: ime korisnika, kriptiranu lozinku sa soli (ovisno o algoritmu koji ju izvodi), te podatke o vremenskom stanju lozinke. Ukoliko se sazna ova vrijednost u shadow datoteci, može se izvršiti napad na nju.

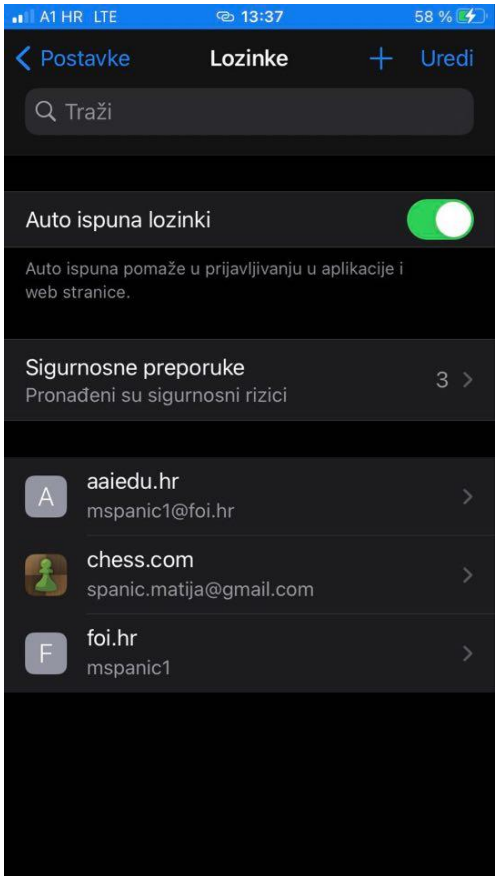
### 3.2.3. MAC operacijski sustav X

MAC operacijski sustavi u početnim inačicama ovog sustava za razliku od Unix-a, gdje se korisnički podaci spremaju u shadow datoteku, informacije o korisniku spremaju direktno u bazu podataka koja se zove Netinfo [13]. U naprednijim verzijama ovog sustava (*OS X10, OS X10.1 i 10.2*), prilikom kriptiranja lozinki, koristio se Unix-ov DES algoritam koji je kasnije zamijenjen radi propusta koji su se nalazili u njemu. Kasnije inačice sustava (od *OS X 10.3* pa nadalje), počele su kamuflirati lozinke i spremati ih izvan Netinfo baze podataka. Sada su lozinke pohranjene u /var/db/shadow/hash direktoriju koji je sličan /etc/shadow–u, te je taj direktorij vidljiv samo glavnom korisniku. Lozinke koje se ovdje nalaze, prvo su sažete SHA-1 algoritmom, a kasnije LANMAN-om.

Današnje verzije ovog sustava koriste PBKDF2-SHA512 algoritam koji sadrži sol, a detaljnije je objašnjen u funkcijama sažimanja. Kako se za pristup raznim aplikacijama i web stranicama koristi korisnički podaci prilikom ulaska, OS X koristi takozvane privjeske ključeva

(*eng. Keychains*) kako bi pomogao zabilježiti lozinke i ostale povjerljive informacije na računalu. Ti privjesci zapravo služe kao kriptirani spremnik za lozinke i nazive računa od operacijskog sustava, aplikacija, internetskih stranica, pa čak i lozinki kreditnih kartice [13]. Upravitelj lozinkama sličan ovome je i Apple ID koji umjesto korisničkog imena zahtjeva e-mail adresu. Privjesci glavnog korisnika smješteni su u /System/library/Keychain folderu. Tu se nalaze svi osjetljivi podaci korisnika. Osim toga, privjesci nude mogućnost uvida i izmjena podataka koji se nalaze u privjesku, korištenjem pristupa privjesku (*eng. Keychain Access utility*).

Lozinku iz ovog operacijskog sustava možemo saznati i preko Property list podataka (.plist). To su podaci koji pohranjuju *hash* vrijednost lozinke, ali su nedostupni prilikom rada operacijskog sustava, a nalaze se u /var/db/dslocal/nodes/default direktoriju. Ukoliko uspijemo prebaciti ove podatke na drugi disk, CD ili USB, tada možemo ovoj naredbi dodati i korisničko ime, čime možemo dobiti uvid u *hash* lozinku.



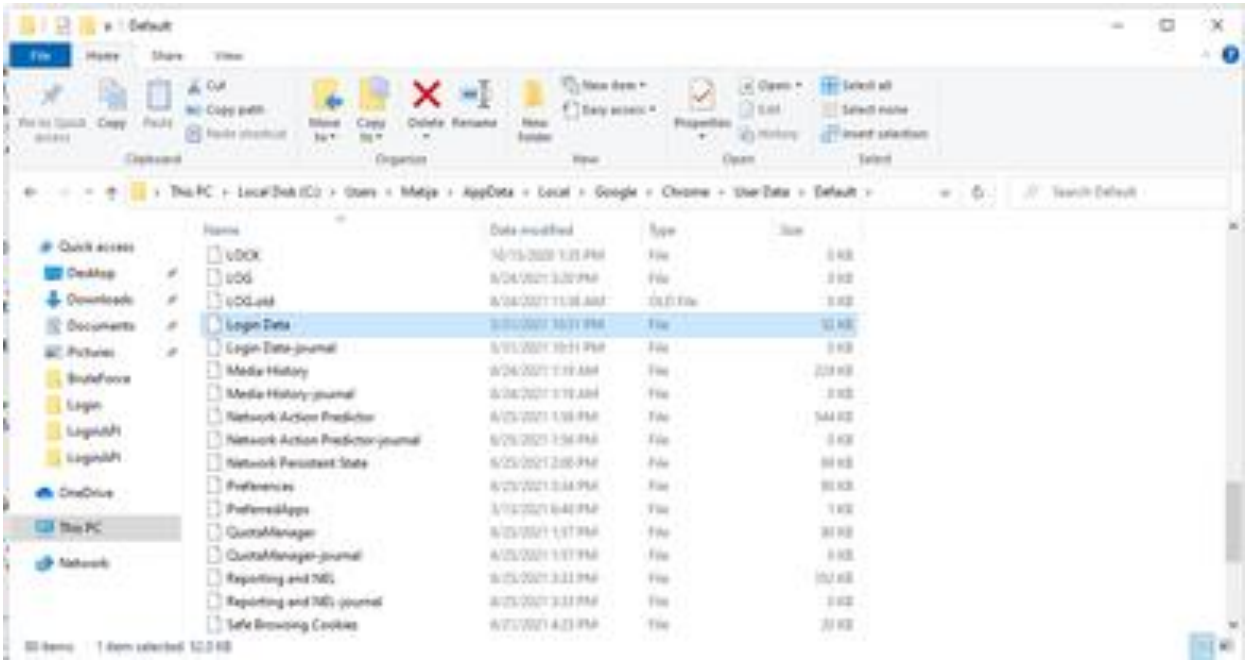

Slika 6:Uvid u lozinke na IOS operacijskom sustavu prikazan na *Iphone 6 SE2* uređaju

Sličan MAC-u, iOS operacijski sustavi također koriste privjeske. Lozinke koje su pohranjene na njima se mogu pronaći u postavkama pod imenom korisnika. Tu se nalaze lozinke kojima se pristupa različitim web stranicama i aplikacijama te su pohranjene u obliku

teksta. Samo glavni korisnik može manipulirati njima na način da dođe do njih sa svojom lozinkom, odnosno Touch ID privjeskom.

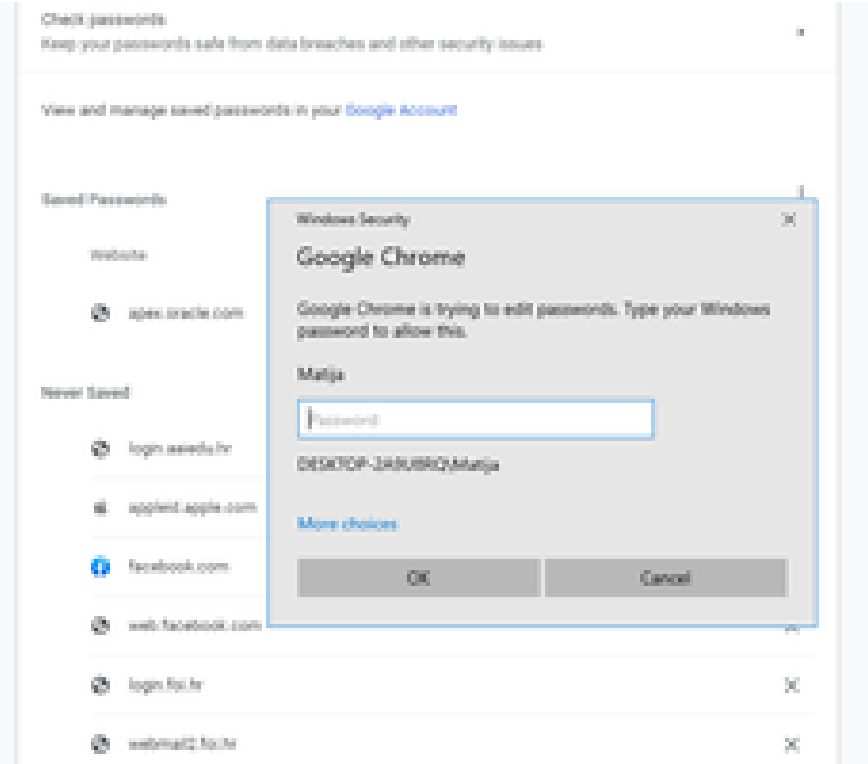
### 3.2.4. Chrome operacijski sustavi

U Chrome operacijskim sustavima prilikom login korisnika na neku stranicu, generira se takozvani „otisak prsta“ lozinke. To se radi na način da se pomoću scrypt-a *hashira* lozinka koja se tada skrati na 37 bitova, te sprema na lokalno računalo korisnika. Spremaju se *hash* podaci lozinke, duljina lozinke, datum i vrijeme kada se zadnji puta lozinka upotrebljavala te informacije o korisničkom mail-u. Podaci se uobičajeno spremaju na lokalnom disku računala. Kada se pokušamo ulogirati, Chrome šifrira lozinku i korisničko ime sa tajnim ključem kojeg samo računalo zna. Tada se šalje sigurnosna kopija na Google. Razlog šifriranja korisničkog imena dolazi zbog anonimnosti, kako napadač ne bi znao kojim stranicama pristupamo i slično. Lozinke se spremaju i u sqllite bazu podataka gdje se one šifriraju sa CryptProtectData, što je API funkcija za šifriranje. Lozinka se sada može dešifrirati samo na istom računalu od strane istog korisnika [14].




Slika 7: Google Chrome datoteka sa lozinkama

Kako bismo pronašli spremljene lozinke na računalo koje Chrome koristi, moramo ići u računalne podatke i otvorit Chrome podatke. Domenu možemo naći preko putanje C:\Users\$username\AppData\Local\Google\Chrom\UserData\Default (slika 6). Tu se nalazi Login Data dokument u kojemu možemo pronaći web stranice koje smo posjećivali, kao i login korisnička imena. Također se može vidjeti koji su se sve korisnici prijavljivali na različite stranice pod stvarnim a ne šifriranim korisničkim imenom.




Slika 8: Prikaz i mijenjanje lozinke pomoću lozinke računala

Chrome još ima mogućnost implementacije nove lozinke (slika 8.). To se može pronaći u Chrome postavkama gdje ima mogućnost manipulacije sa lozinkama. Ukoliko želimo promijeniti lozinku spremljenu od strane *Chrome-a*, tada moramo unijeti lozinku računala pod kojim smo i prvi puta spremili lozinku [14].

# 4. Funkcije sažimanja

Funkcija sažimanja bi bilo cijepanje nečega u manje dijelove kako bi to „nešto“, izgledalo potpuno neprepoznatljivo. Kod lozinki, funkcija sažimanja (*eng. Hashing*) je postupak kojim se lozinka iz originalnog oblika kako je napisana, pretvara u drugačiji zapis slova, znakova i/ili brojeva pomoću različitih matematičkih funkcija. U njemu se unos bilo koje dužine „znakova“ pretvara u ispis fiksne dužine, odnosno, funkcija h pridružuje znakovima proizvoljne dužine nizove znakova konačne duljine što se naziva kompresija [15, str.115]. Za domenu D i kodomenu K definiramo preslikavanje $h : D \rightarrow K$, gdje je $|D| > |R|$ što znači da se sa manjim brojem znakova treba prikazati veći broj znakova. Iz toga razloga dolazi do kolizija o kojima ima više riječi u nastavku.

Naša lozinka koju unosimo u sustav zapravo je zapisana kao potpuno nova abeceda pomoću koje se i ona pamti na računalu. To omogućava da lozinka bude jača, to jest teže ju je probiti. *Hashing* zapravo omogućuje korisnicima veću dozu sigurnosti jer lozinka spremljena u bazi podataka izgleda kao niz vrijednosti (*eng. hash*) koji je teško prepoznatljiv ljudima. Postupak djeluje u nekoliko koraka. Kada ispunimo potrebne podatke, lozinka koju smo unijeli prolazi kroz algoritam koji zamjenjuje postojeću lozinku i daje joj potpuno novi izgled te ju sprema u bazu podataka. Kada se ponovno ulogiramo na sustav, lozinka koju unosimo uspoređuje se sa *hash* vrijednosti spremljenoj u bazi podataka. Ukoliko su sadržaji jednaki, odnosno daju se pročitati, omogućuje se pristup sustavu. Vrijednosti koje vraća *hash* funkcija nazivamo *hash* vrijednostima, *hash* kodom ili najjednostavnije *hashom* (u radu se nadalje spominje *hash*). Pomoću njih se indeksira tablica sa određenom veličinom *hash* vrijednosti.

## 4.1. Kriptografske hash funkcije za lozinke

Prema Hrvatskoj enciklopediji, kriptografija je znanost prevođenja razgovijetnog teksta ili nekog drugog skupa podataka u nerazgovijetan, kriptirani tekst koji se može i kojega može odgonetnuti samo onaj koji posjeduje ključ. Zadaća kriptografije je da se omogući pošiljatelju i primatelju neke poruke tajnost sadržaja tih poruka od treće strane, koja bi te poruke mogla presresti. Kada bismo neku proizvoljnu lozinku kao naprimjer „Matana123“ preveli u *hash* vrijednost pomoću MD5 algoritma, dobili bismo vrijednost: 6ad70cc591e7f81576a5c786b6ed3490. Ukoliko promijenimo jedan znak u lozinki kao naprimjer „matana123“, dobili bismo potpuno drugačiju *hash* vrijednost: 4b8b4907e8c68ff247ade9d48114c793.

Za razliku od običnih *hash* funkcija, kriptografske *hash* funkcije za lozinke imaju posebne uvjete kako bi bile sigurne za upotrebu. Bilo koji tekst odnosno input koji unosimo u algoritam nam daje *hash* vrijednost. Cilj svake funkcije sažimanja je ta da ona bude unikatna, odnosno da ne dolazi do kolizija gdje se sa dva različita inputa dođe do istog outputa, te da svaka vrijednost unesena ponovno daje istu vrijednost, a mala preinaka u inputu mijenja u potpunosti vraćenu vrijednost. Iz toga razloga, *hash* algoritmi se koriste kod kreiranja lozinke kako bi potencijalni napadač teško došao do izvorne poruke. Također, brzina konverzacije inputa kroz algoritam u *hash* vrijednost je veoma bitna kako bi zaporka bila vremenski uporabljiva. Brzina zapravo može biti dvosjekli mač. Za sažimanje poruka i sličnih tekstualnih oblika trebala bi biti brza, no ukoliko se radi o lozinkama, tada bi one trebale biti spore kako bi napad, kao naprimjer *brute force*, dugo trajao te ga se vremenski ne bi isplatilo raditi. Na posljetku, bitno je da funkcija sažimanja bude jednosmjerna pa time sigurna, odnosno da ispunjava svoj zadatak a to je da treća strana ne može dobivenu *hash* funkciju prebaciti u prvobitni input.

Glavni prioritet *hash* funkcije kako je i prije rečeno, je podizanje sigurnosti podataka. Kriptografske funkcije su nepotrebne i nezadovoljavajuće ukoliko postoje dva ili više inputa koji će dati iste *hash* vrijednosti te ukoliko se može naći input na temelju outputa, odnosno *hash* vrijednosti dobivene algoritmom. Ako napadač može napraviti nešto od navedenog, integritet podataka postaje narušen. Iz toga razloga postoje važna svojstva za izračunavanje sažetka poruke. Nadalje su navedena ta svojstva [16]:

1. Otpornost na izračunavanje originala (*eng. Preimage resistance*) – za izlaze je nemoguće pronaći ulaz koji prolaskom kroz algoritam (funkciju) daje taj izlaz odnosno za H = h(M) ne postoji inverzna funkcija M=h-1(H).

2. Otpornost na izračunavanje poruke koja daje isti sažetak (*eng. Second – preimage resistance*) – računski je nemoguće pronaći ulaz koji ima isti izlaz kao neki drugi ulaz, odnosno pronaći drugi original. Drugim riječima, za poznati M i H = h(M) je nemoguće pronaći $M^i$ koji daje isti H.

3. Otpornost na kolizije (*eng. collision resistance*) – nemoguće je pronaći bilo koje dvije poruke M1 i M2 za koje se dobiva isti sažetak h(M1) = h(M2). U slučaju da ne postoji kolizija, tada kažemo da je to perfektno sažimanje (*eng. perfect hashing*).

Ukoliko se žele očuvati podaci od potencijalnog napadača, funkcija treba sadržavati prijašnja svojstva. Osim ovih svojstava koje moraju sadržavati *hash* funkcije, njezinu podjelu možemo svrstati u dvije skupine. To su funkcije bez ključa i funkcije sa ključem [15, str. 115]. Kod funkcija bez ključa u obzir se uzima jedan parametar, to jest poruka, dok kod funkcija sa ključem postoje dva parametra, a to su poruka i tajni ključ. One se pak mogu svrstati u dva tipa *hash* funkcija. Jedan tip su kodovi za otkrivanje preinaka, koji se zovu MDC (*eng. Modification detection codes*) koji koriste samo jedan parametar, a to je ulazna poruka, dok drugi tip o kojem će biti više riječi, su kodovi za autentifikaciju poruke, takozvani MAC (*eng. Message Authentification Codes)* koji koriste dva parametra – ulaznu poruku i ključ.

Valja još spomenuti *hash* tablice koje se koriste za izračunavanje sažetka kod koje ne postoji, ili pak postoji vrlo malen broj kolizija. Svaki podatak u tablici ima svoj jedinstveni ključ. Na taj način je mogući brzi dohvat podataka iz malog broja pokušaja. Najpoznatiji takav algoritam koji se koristi u *hash* tablicama je LOOKUP2 algoritam koji je izuzetno brz, ali je iz tog razloga jednostavan te se ne koristi kod baratanja sa lozinkama zato što je reverzibilan, to jest nije siguran za korisnika i integritet njegovih podataka. Za razliku od digitalnog potpisa koji treba brze *hash* funkcije, lozinke trebaju spore [15, str.115 i 116].

### 4.1.1. MD5 algoritam

MD5 je naziv za kriptografsku funkciju koju je dizajnirao Ronald Riverst 1991. godine. MD je akronim za Message-Digest (*hrv. sažimanje poruke*). Sve većim zahtjevima sustava koji su se morali oduprijeti provalama u sustave, doveli su do napretka u računalnoj kriptografiji. „Ovaj algoritam uzima kao unos poruku proizvoljne duljine i vraća 128 – bitni „otisak prsta“ ili „sažetak poruke“ unosa [17]. Nagađa se da je kompjuterski neizvedivo napraviti dvije poruke koje imaju iste sadržaje ili napraviti bilo kakvu poruku kojoj je cilj unaprijed specificiran sadržaj poruke. MD5 je algoritam koji je namijenjen digitalnom potpisu aplikacija gdje se veća datoteka „stisne“ radi sigurnosnih razloga prije enkripcije sa privatnim (tajnim) ključem. MD5 stvara 128–bitni sažetak ulaznog dijela poruke te je dizajniran da bude brz. Kod njegovih prijašnjih inačica kao MD2 i MD4, koje su još i brže od MD5, uočeni su nedostaci te ih se nakon toga više ne koristi. Ronald je došao do pretpostavke da je susretanjem dviju poruka koje imaju isti sadržaj jednak redu od $2^{64}$ operacija, dok je mogućnost dolaska do istog sadržaja jednak redu $2^{128}$ operacija. Koristi se u različitim operacijskim sustavima zbog svoje brzine i jednostavnosti. Time se može reći da je dosta pouzdan ali ima i on svojih mana koje su naposljetku otkrivene pa trenutno nije toliko u upotrebi.

### 4.1.2. SHA algoritam

SHA je siguran algoritam sažimanja (*eng. Secure Hash Algorithm*). Izrađen je od strane Nacionalnih Instituta Standarda i Tehnologije (*eng. National Institutes of Standards and Technology*) i nekih državnih te privatnih stranaka. Originalna verzija, SHA-0, sadržavala je 160-bitnu *hash* funkciju, no nije mogla držati korak sa tehnologijom, pa je ubrzo slijedila nova verzija ovog algoritma, SHA-1. Daljnjim pronalaskom propusta u ovom algoritmu, javile su se i SHA-2 i SHA-3 verzije. Razlika između SHA-2 i SHA-1 algoritma je ta da novija verzija daje duži sažetak, pa je samim time i otpornija na napade grubom silom. Tako postoje 256 i 512 bitna verzija ove inačice SHA algoritma koja se većinom koristi u kriptovalutama, kao SHA-256 u Bitcoin-u. Premda je novija verzija sigurnija od prijašnjih, i dalje je pratila sličan koncept pretvaranja lozinke, pa se tako kao standard za novi SHA-3 algoritam uzeo Keccak algoritam. On nudi odličnu performansu i jak otpor na napade [18].

### 4.1.3. PBKDF2

PBKDF2 (*eng. Password Key derivation function)* je funkcija deriviranja ključa lozinke koja je osmišljena da bude algoritamski spora kako bi učinkovitost „Rainbow“ tablica bila reducirana. Ovaj algoritam je osmišljen na način da proizvoljno odaberemo broj iteracija koji će se provoditi kroz algoritam, te nam nakon toga broja iteracija, vraća lozinku kao *hash* vrijednost. Ona zaustavlja programe za razbijanje šifre na način da se smanji iskoristivost grafičke procesorske jedinice pa time dolazi do predugog vremenskog roka razbijanja zaporke. Ovaj algoritam je opisan i uveden u internetski standard RFC 2898 (PKCS #5) [19]. Najčešće dolazi u kombinacijama sa SHA algoritmima gdje se može postići i do preko sto tisuća iteracija, kao naprimjer u PBKDF2-SHA512 inačici, koja koristi 512-bitni ključ a primjenjuje se u Dropbox i iCloud sustavima [20, str. 8,9 i 10]. Isto tako se primjenjuju u MAC operacijskim sustavima kod kriptiranja lozinki, a upotrebljava se i kod upotrebe privjesaka (*eng. Keychains*).

### 4.1.4. Salting

Soljenje (*eng. Salting*) je koncept koji se koristi kod lozinki sa sažetkom. To je jedinstvena vrijednost (slova, znakova ili broja/brojeva) koja može biti dodana na početak ili kraj lozinke kako bi se kreirala drugačija *hash* vrijednost. Ta dobivena vrijednost se naziva sol (*eng. Salt*) i spremljena je u bazi podataka zajedno sa *hashom*. Ovisno o programu, ona može biti unesena od strane korisnika ili izabrana slučajnim izborom od strane algoritma. Čak iako je ta vrijednost spremljena na isto mjesto gdje i lozinka, veoma je teško naći takvu lozinku u „Rainbow tablici“, o kojoj će biti više govora u razbijanju lozinki. Ovaj koncept koristi se u Unix operacijskim sustavima u kojima je prvobitno korišten kako ne bi došlo do pojavljivanja dviju

istih lozinki, no uočilo se kako suzbijaju napade na sustav, pa se počela koristiti i u drugim računalnim sustavima.

To je na neki način dodatan sloj sigurnosti koji pomaže protiv napada grubom silom (*eng. Brute force*). Tim postupkom sakrivamo stvarnu *hash* vrijednost tako što joj dodamo bit podataka te ga tako izmijenimo. Dodana vrijednost utječe na korisnika na način da niti jedan drugi korisnik neće imati istu *hash* vrijednost ukoliko koriste iste ili čak različite lozinke, odnosno neće doći do kolizije. Kada u algoritam koji sadrži „salt“ koncept stavimo proizvoljnu lozinku „Matana123“, za rezultat dobivamo *hash* vrijednost „Ntf3MMdSa“. Ukoliko ponovimo proceduru, tada za rezultat dobijemo potpuno novu vrijednost „Bc1XFU1WM“ [21]. Dužina soli može biti proizvoljna ili generirana od strane algoritma no poželjno je da bude barem jednako dugačka kao i lozinka. Nakon što lozinki dodamo sol, lozinka postaje vremenski neisplativa za probijanje.

### 4.1.5. Peppering

Osim „soljenja“, postoji koncept takozvane „tajne soli“ kojoj vrijednost za razliku od nje, nije spremljena zajedno sa lozinkom već na nekoj drugoj lokaciji, pa je samim time teža za pristup napadaču. Taj postupak se naziva „paprenje“ (*eng. peppering*), a ta vrijednost uobičajeno je spremljena u izvorni kod online stranice ili računalnog sustava. Prednosti ovog koncepta, za razliku od „soljenja“, je ta što nije spremljena u bazi podataka, pa samim time ukoliko napadač želi saznati tu vrijednost mora izvesti *brute force* napad koji bi u slučaju spore *hash* funkcije poprilično dugo trajao [22]. S druge strane, za napredne *hash* algoritme namijenjeno je korištenje „soli“, pa se tako implementacija „papra“ smatra ne praktičnom. Kod paprenja se prije spremanja lozinki, ispred nje stavlja jedan ili više znakova koji se odabiru slučajno nekim algoritmom. Na taj način se produžuje vrijeme napada kao kod *brute force* napada, gdje se za svaku lozinku mora još provoditi i iteracija sa znakovima, kako bi došli do lozinke.

### 4.1.6. Bcrypt

Bcrypt je algoritam baziran na enkripcijskom algoritmu nazvanom Blowfish. Koristi se u Linux i BSD operacijskih sustava gdje lozinku pretvara u *hash* vrijednost.. Za razliku od drugih algoritama kao naprimjer MD5 ili SHA-1 koji su brzi, ovaj je napravljen da bude spor. Razlog zašto je napravljen da bude spor je taj što naša baza podataka teško može postati kompromitirana. Recimo da naprimjer posjedujemo web stranicu i da se na nju želi prijaviti neki korisnik. On mora unijeti svoje korisničko ime i lozinku. Bycrpt algoritam transformira lozinku u *hash* vrijednosti određene duljine (najčešće oko 30 do 50 znakova) te mu dodaje sol (*eng. Salt*) na kraju, kako u slučaju dviju istih lozinka ne bi postojala ista *hash* vrijednosti,

odnosno kako ne bi došlo do kolizija. Sada kada bi napadač želio izvesti *brute force* napad ili napad rječnikom, koje ćemo kasnije detaljnije objasniti, trebala bi im cijela vječnost. Naime, Bycrpt algoritam je s namjerom napravljen da bude spor, upravo radi takovih napada kod kojih trajanje izvođenja traje dugo, pa je napad samim time vremenski neisplativ. Napadi kod Bcyrpt algoritma trebaju veću količinu dostupne memorije koju zapravo grafički procesori ne posjeduju, pa je tako brzina grafičke procesorske jedinice limitirana njezinom veličinom [23, str. 7 i 8].

### 4.1.7. Scrypt

Scrypt je algoritam sličan bycrpt-u koji je dizajniran na način da se prilikom napada na korisnikovu lozinku koristi veća količina kompjuterske memorije, ili pak u isto vrijeme procesor izvodi više izračuna kako bi se ukralo vrijeme napada. Izvorno ga je smislio Colin Percival, te 2009.-te godine pustio u slobodno korištenje. Koristi se kod kriptovaluta kao naprimjer litecoin, dogecoin, syscoin i slično. Sastoji se od nekoliko parametara. Ti parametri su: broj iteracija (n) koji utječu na memoriju i procesor, veličina bloka (r), broj dretvi koje se izvode istovremeno (p), lozinka, sol i duljina ključa. Memorija koja je potrebna da se spremi lozinka jednaka je umnošku svih parametara [24]. Za razliku od ostalih algoritama, ovome je potrebno korištenje većeg broja memorije. Kada napadač povećava brzinu napada, tada algoritam troši enormno više memorije, samim time napad postaje vremenski a i novčano neučinkovit jer je napadač limitiran veličinom memorije koju posjeduje.

### 4.1.8. Argon2

Argon2 je sigurnosni *hash* algoritam za lozinke dizajniran od strane Sveučilišta Luxemburg na način da korisnik sam bira vrijeme potrebno za pretvorbu lozinke u *hash*, kao i količinu memorije koje će zauzeti. U mogućnosti je da se primjeni na x86 arhitekturi i može se kompajlirati na Linux, OS X i Windows operacijskim sustavim. Dolazi u tri varijante: Argon2d (većinom se koristi za kriptovalute i aplikacije iz razloga što je malo brži), Argon2i (koristi se za kreiranje lozinki zato što je poprilično spor, od 0,5 sekundi za kreiranje i više od sekundu za autentifikaciju) te Argon2id koji je zapravo kombinacija prijašnjih dva. Argon2 ima nekoliko parametara [25]:

- Memorija – lozinka zauzima podosta memorije kako bi provaljivanje bilo čim „skuplje“ odnosno zahtjevnije za potencijalnog napadača.
- Iteracija – broj iteracija koji se nalazi u algoritmu te koji samo postavljamo kako bi vrijeme pronalaska *hasha* bilo čim dulje.

- Paralelizam – broj dretvi koji se koristi bi trebao biti što veći kako bi i procesor izvršavao što više zadataka. Poželjno je da bude duplo veći od dostupnih jezgri procesora.
- Duljina soli (*eng. salt length*) – preporuča se da duljina bude 128 bita ali može i 64 bita.
- Dužina ključa (*eng. key length*) – kao i kod soli, preporuča se da bude 128 bita.

Algoritam Argon2 je dobio priznanje za najsigurniji algoritam što se tiče *hash* funkcija na „Password Hashing“ natjecanju koje se održavalo od 2012.-te do 2015.-te godine. Pobjedu je odnio zbog vremena koje je potrebno da se izvrši napada na njega, kao i količini memorije koja se pritom koristi, te ogromnom broju dretvi koje procesor mora pritom izvršavati.

# 5. Razbijanje lozinki

Iako stalnim napretkom tehnologije, u smislu poboljšanja sigurnosti sustava na način da lozinke budu zaštićene i prevedene raznim algoritmima, ista tehnologija unazaduje to isto. Napredak procesora i grafičkih kartica pomaže algoritmima koji „kamufliraju“ lozinku ali i odmažu ukoliko ih napadač posjeduje i zna što činiti sa njima. On te alate koristi kako bi pokušao razbiti lozinku te time kompromitirati korisnika. Napadač je osoba koja svjesno ili nesvjesno koristi naše podatke u svrhu vlastitih potreba. Zatim te podatke koristi u privatne svrhe radi novčane vrijednosti koje time može postići ili čak iz svoje vlastite zanimacije.

Razbijanje lozinki je zapravo proces kojim se zaporka obnavlja iz podatka u kojem je pohranjena na nekom računalnom sustavu. U prošlom poglavlju su navedeni algoritmi za lozinke koji služe kao zadnja linija protiv napada na njih. Ukoliko napadač ukrade *hash* vrijednost pohranjenu na serveru za autentifikaciju, tada može izvršavati offline napad koji može biti samo novčano limitiran cijenom alata koji se koriste za njega (snaga računala, posjedovanje algoritama i slično). Kod online principa napada, postoji višestruko pogađanje lozinke kojeg korisnik koristi u slučaju zaboravljene zaporke. S namjerom da se razbije lozinka, napravljeni su razni algoritmi koji prilikom autentifikacije ponavljaju različite lozinke kako bi neovlašteno ušli u sustav. Iz toga razloga, administratori sustava donose preventivne mjere kao naprimjer brojanje pokušaja ulaska u sustav čijim se prekoračenjem isključuje sustav radi onemogućavanja izvođenja potencijalnog napada.

U prošlom poglavlju smo naveli funkcije sažimanja koje se koriste za zaporke koje su u pravilu dosta brze (premda postoje spore funkcije sažimanja koje su na neki način ipak brze). One se mogu provesti vrlo brzo, što daje napadaču mogućnost da testira veći broj zaporki dok ne pronađe odgovarajuću. Lozinke u operacijskim sustavima spremljene su bazu podataka u obliku šifriranog teksta (*eng. Encrypted passwords*). Podaci za provjeru zaporke generiraju se takozvanim jednosmjernim funkcijama (*eng. One-way functions*). Jednosmjerne funkcije za spremanje zaporki u *hash* obliku u pravilu su sigurne ukoliko se nalaze u bazi podataka, no ukoliko napadač posjeduje takav oblik lozinke, dolazi do mogućnosti njezinog razotkrivanja. U nastavku su objašnjeni neki načini kako napadač može doći do naše lozinke.

## 5.1. Načini otkrivanja lozinki

Kako je već prije objašnjeno da se za lozinke trebaju koristiti spore *hash* funkcije, kako se napad na otkivanje lozinke ne bi isplatio, razvile su se nove metode koje napadaču nude drugačije mogućnosti dolaska do lozinke. Tim metodama napadač može pokušati pristupiti

operacijskom sustavu bez da zna *hash* vrijednost kojom je lozinka pohranjena u sustav. Iako kod dobro dizajniranog sustava zaštite gdje se prilikom prekoračenog broja pristupa u sustav obavještava administratora kako se pokušava ući u sustav, te sigurnosne mjere nekada ne predstavljaju prepreku. Lozinke se mogu nabaviti i nekim drugim metodama kao [26]:

- wiretapping – praćenje internetske ili telefonske konverzacije od strane napadača a da pritom nismo ni svjesni da nas netko prisluškuje,
- snimanje unosa u tipkovnicu (*eng. keystroke logging*) – gdje se snima unos tipkovnice prilikom ulogiravanja u računalni sustav, te šalje napadaču koji tada vidi sadržaj koji smo unijeli prilikom ulaska u sustav,
- shoulder surfing – način špijuniranja kod kojeg napadač prati pokrete korisnika koji unosi lozinku te time pokušava „provaliti“ korisnikove pokrete prilikom upisivanja šifre,
- Phishing – napadač usmjerava korisnika na lažne stranice kako bi upisali lozinku
- Dumpster diving – pronalaženje informacija koje je korisnik odbacio u smeće kako bi došao do korisničkog imena i zaporke,
- Timing attack – korisnika se ugrožava na način da se zna vrijeme potrebno za izvršavanje kriptografskog algoritma pa tako napadač dolazi do saznanja o kojem je algoritmu riječ,
- Virusi - korištenje zlonamjernog programa kao naprimjer trojanskog konja u svrhu da se zavara korisnika koji na taj način svojim ne znanjem pušta napadača u sustav.

Osim ovih načina gdje saznajemo lozinku preko korisnika, zaporku možemo saznati još nekim metodama kojima ne moramo tražiti korisnikove sigurnosne propuste, već lozinku tražimo takozvanim „grubljim načinom“ sljedećim metodama: pogađanje zaporki, napad rječnikom (*eng. dictionary attack*), napad grubom silom (*eng. brute force*) i napad „Rainbow“ tablicama. Nadalje će biti objašnjene ove metode napada.

### 5.1.1. Pogađanje lozinki

Kod računalnih sustava u kojima korisnik sam odabire lozinku koju će koristit tijekom logina, dolazi do problema jednostavnosti lozinke gdje korisnik bira onu zaporku koja će se njemu najlakše zapamtiti kako ju ne bi zaboravio prilikom ponovnog ulaska u sustav. Iz toga razloga, takve zaporke je moguće pogoditi na temelju privatnih saznanja o vlasniku lozinki. Na primjer, ako korisnik radi u banci i ima lozinku „MATIJAERSTE97“ zato što se zove Matija i rođen je 1997.-te godine te radi u Erste banci, njegove kolege i prijatelji mogu jednostavno pogoditi njegovu zaporku.

To je samo jedan od primjera gdje korisnik unosi prejednostavnu lozinku te time svoje podatke stavlja u opasnost. Takvo predvidljivo stavljanje zaporke postavlja problem gdje

napadač sa malo domišljatosti može provaliti lozinku. Najveći problem dolazi kada napadač zna o kojoj je osobi riječ, odnosno kada osobno poznaje tu osobu, pa na temelju toga može isprobavati razne kombinacije lozinki. Najčešće se do pogađanja lozinke dolazi kada za lozinku uzmemo jednostavne pojmove kao što je: ime druge osobe, ime prve ljubavi, datum rođenja, ime neke slavne osobe, jednostavan niz brojeva (npr. od 1 do 9), ime ljubimca, korisničko ime i slično.

Kako bi se riješio taj problem, administratori računalni sustava uvode kriterije kao što su: određen broj znakova koja lozinka mora sadržavati (najčešće od 8 na više), barem jedno malo i veliko slovo uz barem jedan broj ili znak, razlikovanje od korisničkog imena i slično.

### 5.1.2. Napad rječnikom

Najbrži a i najjednostavniji način otkrivanja lozinke je upravo napad rječnikom (*eng. dictionary attack*). Ova tehnika razbijanja lozinki zasniva se na isprobavanju lozinki prethodno pripremljenih rječnika. Rječnik je zapravo baza podataka u kojoj se nalazi ogroman broj lozinki. Postoje dva načina provođenja napada, a to su online i offline. Kod offline dictionary metode, uspoređuju se *hash* vrijednosti nepoznate lozinke spremljene u bazi podataka sa *hash* vrijednostima koji se nalaze u rječnik. Online napad koristi riječi kod same prijave u računalni sustav pri čemu je potrebno znati samo korisničko ime. Napad rječnikom najčešće izvršava program koji ima ugrađen rječnik i implementiran algoritam za pretraživanje

Ovaj način napada na lozinku je zapravo među prvima koji se izvršava kako bi ju razbili, međutim danas nije toliko u upotrebi zbog ograničenog broja lozinki koje se nalaze u rječniku. Razlog tomu je što računalni sustavi pohranjuju sažetak poruke, pa offline način pretraživanja lozinke nije toliko uspješno, jer nema toliko lozinki u bazi podataka. Osim toga, postoje načini kako spriječiti ovakav napad, a to je da se odgodi odgovor poslužitelju te da se zaključa korisnički račun. Kod odgode odgovora sprječava se napadaču provjeravanje većih količina lozinki kroz kratko vrijeme. Osim toga, napadača se može spriječiti kada sustav prepozna previše neispravnih pokušaja pa se time zaključa račun. Postoje razne verzije rječnika koje se mogu pronaći na internetu pa tako i hrvatska verzija.

### 5.1.3. Napad grubom silom

Napad grubom silom (*eng. brute force*) jedan je od načina razbijanja lozinke gdje se pokušava isprobavanje svake moguće kombinacije znakova. Premda bi ovaj način provaljivanja trebao biti sto posto uspješan, veličina zaporke produžuje vrijeme traženja odgovarajuće lozinke. Moglo bi se reći da je ova metoda prilično nepraktična te bi trebala biti zadnja opcija za razbijanje lozinke. Što je računalo koje izvršava napad snažnije, to je i metoda uspješnija. *Brute force* je zapravo kompletnija verzija napada rječnikom no za razliku od njih

on provjerava svako slovo, broj ili znak dok ne dođe do željene kombinacije. Današnji algoritmi za traženje lozinke su daleko napredniji od naprimjer DES-a (*eng. Data Encryption Standard*) koji koristi 56 bitnu duljinu ključa, to jest kod kojeg postoji $2^{56}$ mogućih kombinacija. U slučaju offline napada, napadač neograničeno može pogađati zaporku dok kod online napada postoje metode koje to sprječavaju, kao naprimjer CAPTCHA (*eng. Completely Automated Public Turing test to tell Computers and Humans Apart*). Tu se korisnika prilikom prijave pita nešto što računalo ne može, kao naprimjer da očita znak ili neku drugi sadržaj sa slike te ga unese prije upisa lozinke.

Kako bi se izvršio ovaj napad, napadač mora saznati *hash* vrijednost lozinke pohranjene u sustavu. Nakon toga, *hash* se uspoređuje sa svim ostalim hash vrijednostima tog algoritma (naprimjer u MD5 algoritmu, lozinka zapisana kao *hash* vrijednost se uspoređuje sa svim ostalim MD5 *hash* vrijednostima, jedna po jednu). Ukoliko se te vrijednosti poklapaju, pronašla se odgovarajuća lozinka. Postavlja se pitanje koliko dugo takav napad traje? Ako se uzme lozinka koja sadrži 8 malih slova engleske abecede, tada bi napad grubom silom sadržavao $(26)^8$ kombinacija što je jednako 208827064576 kombinacija. Ukoliko pritom računalo ima snage da provede 1000 kombinacija u sekundi, tada bi taj proces trajao oko 58 tisuća sati. Na internetu postoje različite aplikacije za računanje vremena potrebnog za izvođenje *brute force* napada kao naprimjer *Password Calculator*.

Tablica 1: Vrijeme potrebno za izvođenje napada grubom silom

| Dužina znakova | Mala slova | Velika slova | Brojevi | Mala i velika slova | Svi ASCII znakovi | Broj računala |
|---|---|---|---|---|---|---|
| <=4 | minuta | minuta | minuta | Nekoliko minuta | 3 minute | 1 |
| 7 | 5 sati | 5 sati | minuta | 24 dana | 5 godina | |
| 9 | 5 mjeseci | 5 mjeseci | 34 minute | 179 godina | 44531 godina | |
| <=4 | Sekunda | sekunda | sekunda | Nekoliko sekundi | 2 minute | 2 |
| 7 | 134 minute | 134 minute | minuta | 12 dana | 29 mjeseci | |
| 9 | 63 dana | 63 dana | 17 minuta | 90 godina | 222660 godina | |

Iz tablice se može uočiti kako dužina znakova, kao i njihova kombinacija povećava vrijeme traženja lozinke. Isto vrijedi i za snagu računala koja se povećava ovisno o brzini procesora, to jest koliko operacija on može izvoditi u sekundi (u tablici iznad, brzina iznosi 500000). U ovoj vrsti napada, količina RAM-a ne utječe na brzinu napada. Ukoliko se napad podijeli na više računala tada možemo skratiti vrijeme izvođenja napada skoro za dvostruko. Ukoliko se radi o zaporci koja ima kombinacije različitih znakova kao i veću dužinu, tada ovakva metoda razbijanja lozinke nije praktična već vremenski neisplativa. Odlični savjeti za otežavanje ovakve vrste napada je povećanje dužine i kompleksnosti lozinki, kao i ograničavanje broja pokušaja kod prijavljivanja te korištenje dvo-faktorske autentifikacije.

### 5.1.4. Rainbows tablice

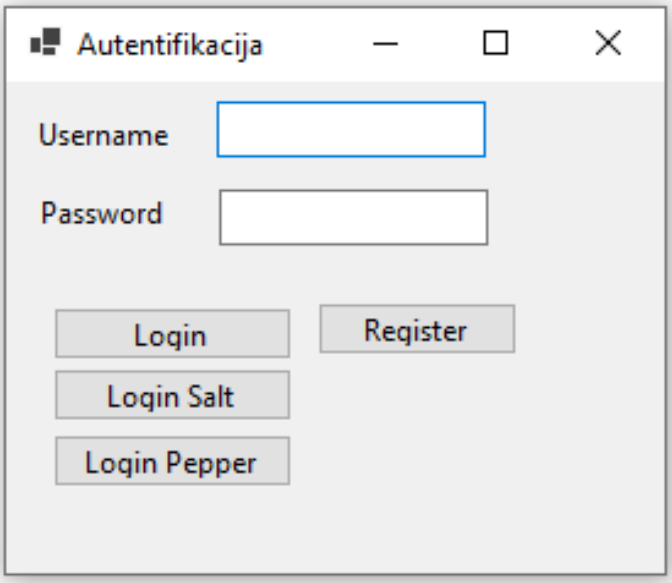
Dugine tablice (eng. Rainbow tables) koriste time-memory trade of princip za razbijanje zaporki. Time-memory trade of tehniku (TMTO) osmislio je Ronald Riverst kako bi ubrzao kriptoanalizu unaprijed izračunatih vrijednosti koje se nalaze u memoriji. Kako je za pohraniti zaporke i njihove *hash* vrijednosti potrebna velika količina memorije, Rivest je došao do ideje da se umjesto cijelih lozinki i njihovih *hash* vrijednosti pohranjuje samo jedan dio iz kojeg je moguće dobiti ostatak. Postoji veliki broj lozinki koji se nalaze u bazi podataka pa ih iz toga razloga koriste napadači. *Rainbow* tablica nam prikazuje povezanost lozinki (lanac). Svaki taj lanac počinje sa početno lozinkom iz koje se izračunaju ostale. Njima se izračunava sažetak koji se zatim reducira kako bi se iz heksadekadskog niza pretvorila u ASCII. Taj proces se ponavlja te se dobiveni sadržaj pohranjuje u *rainbow* tablicu. Povećanjem iteracija i daljnjom redukcijom *hash* funkcija smanjuje se i tablica, no vrijeme potrebno da se ispita *hash* funkcija se povećava.

Lanac je niz zaporki koja započinje sa početnom. Početna zaporka se smanjuje te se prolazi na sljedeću. Nakon što je postignut odgovarajući broj zaporki tada se početna i konačna spremaju u tablicu. Nakon toga pretražuje se zaporka u tablici. Ukoliko se ne pronađe u tablici, tada se proces ponavlja. Uspješnost napada ovisi o duljini zaporke, duljini lanca, broju redaka tablice kao i korištenim skupom znakova. Isto tako dodavanje soli na kraj ili početak lozinke znatno otežava taj proces zato što *rainbow* tablice ne sadrže zaporke iste duljine i složenosti pa ju nije moguće pronaći. Ukoliko se i pronađe, zaporki se prije korištenja mora ukloniti „salt“. *Rainbow* tablice moguće je samostalno napraviti pomoću RainbowCrack alata. Također dolazi besplatan u alatu *Ophrack*, razvijen specifično za operacijske sustave. Operacijski sustavi kao Windows ne koriste kriptografske algoritme sa „salt“ primjesama, pa je pogodan za razbijanje lozinki pomoću *rainbow* tablica. Ukoliko zaboravimo lozinku potrebnu da uđemo u Windows operacijski sustav, tada se možemo koristit ovom metodom.

# 6. Aplikacija

## 6.1. Autentifikacija korisnika kriptografskim funkcijama

Kao praktični zadatak sam napravio nekoliko aplikacija kako bih prikazao autentifikaciju korisnika u neki sustav. Prva je LoginAPI koja pruža API metode za registraciju i prijavu. Koristio sam ASP.NET Core kako bi napravio API servis preko *Microsoft Visual Studia* koji koristi programski jezik C#. Aplikacija ima samo jedan kontroler–AccountController, koji ima jednu metodu za registriranje i tri za login (login bez soli i papra, login sa soli i login sa paprom). Aplikacija koristi SHA-256 algoritam kako bi lozinku pretvorio u *hash* vrijednost. Kao bazu koristi jednu datoteku tako da će svi korisnici biti tamo spremljeni nakon registracije . Druga aplikacija daje pristup tom API-u za registraciju i prijavu preko Winformsa, a sastoji se od metoda za pozivanje API-a. Treća aplikacija služi za *brute forcanje* tog API–a pomoću rječnika.



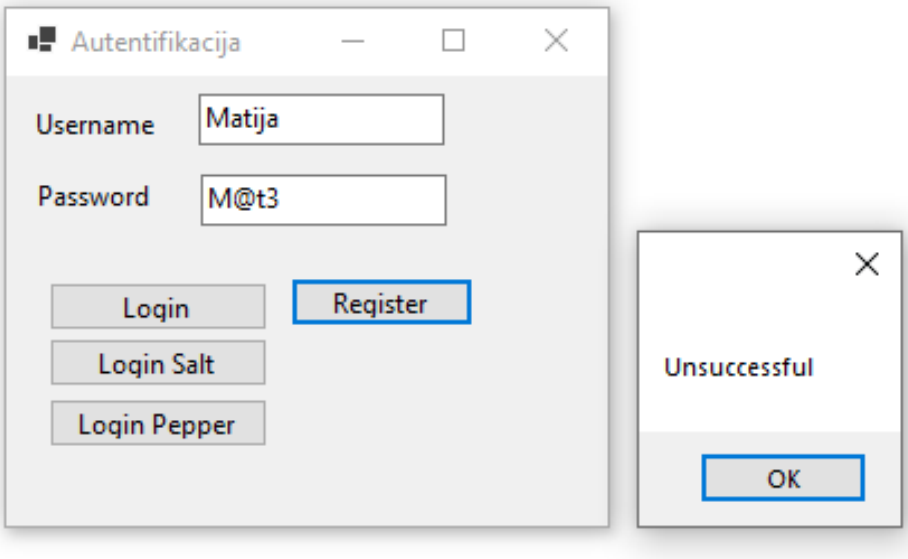

Slika 9: Izgled aplikacije za autentifikaciju

Aplikacija se sastoji od polja za unos korisničkog imena (*eng. username*) i polja za unos lozinke (*eng. password*). Osim toga, postoje mogućnosti za: registraciju, login sa *hashom*, login sa soli i login za paprom. Aplikacija radi na principu korisničke registracije gdje se unosi korisničko ime i lozinka. Ukoliko je registracija uspješna imamo mogućnost prijave u aplikaciju.

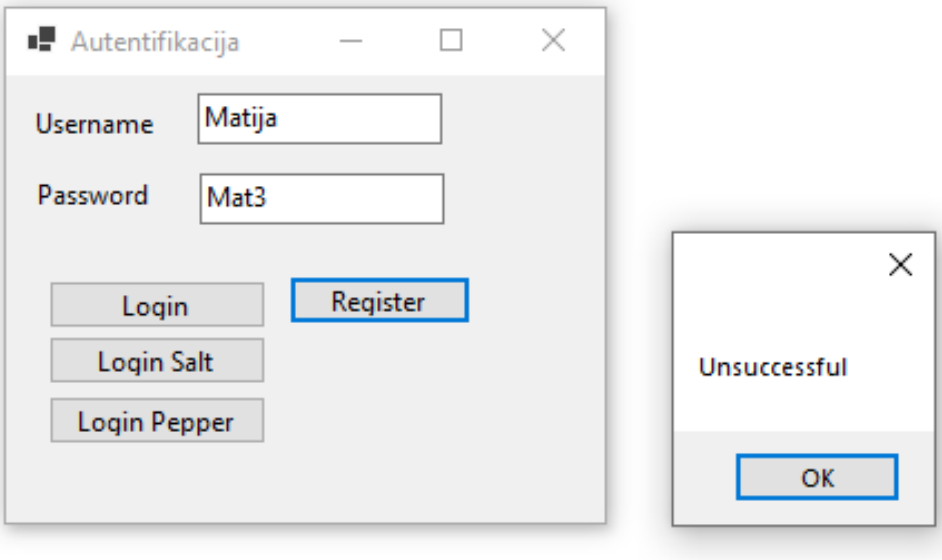
Kod početne forme za registraciju i login izgleda ovako:

```
namespace Login
{
    public partial class Form1 : Form
    {
        private const string _url = "https://localhost:44365/Account/";
        public Form1()
        {
            InitializeComponentAsync();
        }
        private void btnLogin_ClickAsync(object sender, EventArgs e)
        {
            var url = _url + $"Login?username={txtUsername.Text}&hash={GetHashString(txtPassword.Text)}";
            CallApiAsync(url);
        }
        private void btnRegister_ClickAsync(object sender, EventArgs e)
        {
            var url = _url + $"Register?username={txtUsername.Text}&password={txtPassword.Text}";
            CallApiAsync(url);
        }
        private void btnLoginSalt_ClickAsync(object sender, EventArgs e)
        {
            CallSaltApiAsync(txtUsername.Text, txtPassword.Text);
        }
        private void btnLoginPepper_ClickAsync(object sender, EventArgs e)
        {
            CallPepperApiAsync(txtUsername.Text, txtPassword.Text);
        }
```

Kako bi bila sigurna, lozinka mora sadržavati neke uvjete koje sam postavio. Ti uvjeti su: barem jedno veliko i malo slovo, barem jedan broj te barem jedan znak. Zato je minimalan broj karaktera koji se koristi 4, kako bi kasnije i *brute force* imao smisla. Provjera lozinke se vrši preko *Regexa [27]*. Ukoliko ona sadrži sve od navedenog, tada je registracija uspješna i sprema se u bazu podataka. U protivnom, ukoliko se korisničko ime ponavlja, ili pak lozinka ne sadrži navedene kriterije, aplikacija vrati grešku odnosno prozorčić (*eng. messge box*) koji kaže da ona nije uspješna (slika 10 i 11).




Slika 10: Uspješna registracija




Slika 11: Neuspješna registracija

Kod za uspješnu ili neuspješnu registraciju prilikom poziva API-a izgleda:

```
private async Task CallApiAsync(string url)
        {
            HttpClient client = new HttpClient();
            var response = await client.PostAsync(url, null);
            var result = await response.Content.ReadAsStringAsync();
            if (result == "true")
            {
                MessageBox.Show("Successful");
            }
            else
            {
                MessageBox.Show("Unsuccessful");
            }
        }
```

Kod prijave sa paprom, poziva se API i generiraju se sve varijacije za papar te se izbacuje prozorčić o uspješnosti. U nastavku slijedi prijava sa paprom:

```
        private async Task CallPepperApiAsync(string username, string
password)
        {
            for (var i = 65; i <= 91; i++)
            {
                var url = _url +
$"LoginPepper?username={username}&hash={GetHashString(password + (char)i)}";
                HttpClient client = new HttpClient();
                var response = await client.PostAsync(url, null);
                var result = await response.Content.ReadAsStringAsync();
                if (result == "true")
                {
                    MessageBox.Show("Successful");
                    return;
                }
            }
            MessageBox.Show("Unsuccessful");
        }
```

Kod prijave sa soli, poziva se API i dohvaća se sol sa API-a i spaja ga sa lozinkom, te se izbacuje prozorčić ovisno o uspješnosti. To izgleda ovako:

```
            private async Task CallSaltApiAsync(string username, string
password)
        {
            HttpClient client = new HttpClient();
            var responseSalt = await client.PostAsync(_url +
$"GetSalt?username={username}", null);
            var salt = await responseSalt.Content.ReadAsStringAsync();
            var url = _url +
$"LoginSalt?username={username}&hash={GetHashString(password + salt)}";
            client = new HttpClient();
            var response = await client.PostAsync(url, null);
            var result = await response.Content.ReadAsStringAsync();
            if (result == "true")
            {
                MessageBox.Show("Successful");
            }
            else
            {
                MessageBox.Show("Unsuccessful");
            }
        }
```

Kako bi napravio registraciju korisnika, napravio sam još tri dodatna modela koja uključuju metode za rad sa datotekama koje su potrebne za aplikaciju, pretvaranje stringa u *hash*, te model klasu za korisnika.

Kod za provjeru lozinke i spremanje u bazu podataka:

```
namespace LoginAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class AccountController : ControllerBase
    {
        [HttpPost]
        [Route("[action]")]
        public bool Register(string username, string password)
        {
            Regex regex = new Regex(@"^(?=.*[a-z])(?=.*[A-
Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{4,}$");
```

```
            Match match = regex.Match(password);
            if (!match.Success || !FileHelper.CheckUsername(username))
            {
                return false;
            }
            var passwordHashed = HashHelper.GetHashString(password);
            var salt = GenerateSalt();
            var passwordHashedSalt = HashHelper.GetHashString(password +
salt);
            var passwordHashedPepper = HashHelper.GetHashString(password +
FileHelper.GetPepperValue());

            User user = new User
            {
                UserName = username,
                PwdHash = passwordHashed,
                PwdHashSalt = passwordHashedSalt,
                PwdHashPepper = passwordHashedPepper,
                Salt = salt
            };
            FileHelper.StoreUser(user);
            return true;
        }
```

Kako bismo spremili lozinku, napravio sam model (HashHelper) za pretvaranje stringa (lozinke) u *hash*, kao i metode dobivanja bajtova iz stringa te dobivanje *hasha* iz stringa. U aplikaciji se koristi SHA-256 algoritam koji pretvara niz znakova u niz bajtova. Kod za to izgleda:

```
namespace LoginAPI.Models
{
    public class HashHelper
    {
        private static byte[] GetHash(string inputString)
        {
            using (HashAlgorithm algorithm = SHA256.Create())
                return
algorithm.ComputeHash(Encoding.UTF8.GetBytes(inputString));
        }
        public static string GetHashString(string inputString)
        {
```

```
            StringBuilder sb = new StringBuilder();
            foreach (byte b in GetHash(inputString))
                sb.Append(b.ToString("X2"));
            return sb.ToString();
        }
    }
}
```

Osim toga, napravio samo još model klasa za korisnika (User) koji sadrži sol i *hash* za sva 3 načina *hashiranja*. Kod za to slijedi:

```
namespace LoginAPI.Models
{
    public class User
    {
        public string UserName { get; set; }
        public string PwdHash { get; set; }
        public string PwdHashSalt { get; set; }
        public string Salt { get; set; }
        public string PwdHashPepper { get; set; }
    }
}
```

Napravio sam i model FileHelper, koji sadrži metode za rad sa datotekom koja je potrebna za aplikaciju te se pozivaju prilikom registriranja i logina. To su metode koje spremaju podatke u bazu podataka na kraj datoteke, te pretvaraju podatke u string kako bi odgovarale formatu datoteke. Također, ovdje se generira i vrijednost papra koji je se dobije slučajnim izborom jednog slova od A-Z. Ujedno se i provjerava postoji li korisnik u bazi podataka prilikom prijave. Kod za ovaj model izgleda:

```
namespace LoginAPI.Models
{
        public class FileHelper
    {
        const string fileName = "database.txt";
        public static string GetPepperValue()
        {
            return ((char)(65 + (GetCount() % 26))).ToString();
```

```
        }
        public static int GetCount()
        {
            return File.ReadLines(fileName).Count();
        }
        public static void StoreUser(User user)
        {
            using (StreamWriter sw = File.AppendText(fileName))
            {
                sw.WriteLine(GenerateLine(user));
            }
        }
        private static string GenerateLine(User user)
        {
            return
$"{user.UserName},{user.Salt},{user.PwdHash},{user.PwdHashSalt},{user.PwdHa
shPepper}";
        }
        public static User GetUserFromLine(string line)
        {
            var values = line.Split(',');
            return new User
            {
                UserName = values[0],
                Salt = values[1],
                PwdHash = values[2],
                PwdHashSalt = values[3],
                PwdHashPepper = values[4],
            };
        }
        internal static bool CheckUsername(string username)
        {
            using (var fileStream = File.OpenRead(fileName))
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    string line;
                    while ((line = streamReader.ReadLine()) != null)
                    {
                        if (line.ToUpper().StartsWith(username.ToUpper()))
                        {
```

```
                        return false;
                    }
                }
            }
        }
        return true;
    }
    internal static bool CheckLogin(string username, string hash,
string type)
    {
        using (var fileStream = File.OpenRead(fileName))
        {
            using (var streamReader = new StreamReader(fileStream))
            {
                string line;
                while ((line = streamReader.ReadLine()) != null)
                {
                    if (line.ToUpper().StartsWith(username.ToUpper()))
                    {
                        var user = GetUserFromLine(line);
                        if(type == "login")
                        {
                            return user.PwdHash == hash;
                        }
                        if(type == "salt")
                        {
                            return user.PwdHashSalt == hash;
                        }
                        if(type == "pepper")
                        {
                            return user.PwdHashPepper == hash;
                        }
                    }
                }
            }
        }
        return false;
    }
    public static string GetSalt(string username)
    {
        using (var fileStream = File.OpenRead(fileName))
```

```
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    string line;
                    while ((line = streamReader.ReadLine()) != null)
                    {
                        if (line.ToUpper().StartsWith(username.ToUpper()))
                        {
                            var user = GetUserFromLine(line);
                            return user.Salt;
                        }
                    }
                }
            }
            return "";
        }
    }
}
```

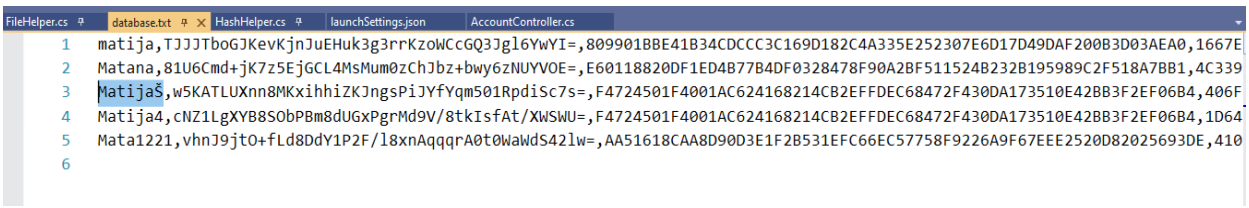
Kako bismo spremili sol i *hash* sa soli, moramo ju prvo generirati. To je napravljeno u Account kontroleru preko RNGCryptoServiceProvider klase koja vraća sol duljine 5 [28]. Kod izgleda:

```
        private string GenerateSalt()
        {
            var value = "";
            while (true)
            {
                var random = new RNGCryptoServiceProvider();
                int max_length = 32;
                var salt = new byte[max_length];
                random.GetNonZeroBytes(salt);
                value = Convert.ToBase64String(salt);
                if (value.Contains(','))
                {
                    continue;
                }
                break;
            }
            return value;
        }
```

Kontroler osim registracije sadrži još i metode za: prijavu korisnika sa običnim *hashom*, prijava korisnika sa *hashom* i soli, prijava korisnika sa *hashom* i paprom.

To izgleda ovako:

```
[HttpPost]
[Route("[action]")]
public bool Login(string username, string hash)
{
    return FileHelper.CheckLogin(username, hash, "login");
}
[HttpPost]
[Route("[action]")]
public bool LoginSalt(string username, string hash)
{
    return FileHelper.CheckLogin(username, hash, "salt");
}
[HttpPost]
[Route("[action]")]
public bool LoginPepper(string username, string hash)
{
    return FileHelper.CheckLogin(username, hash, "pepper");
}
```

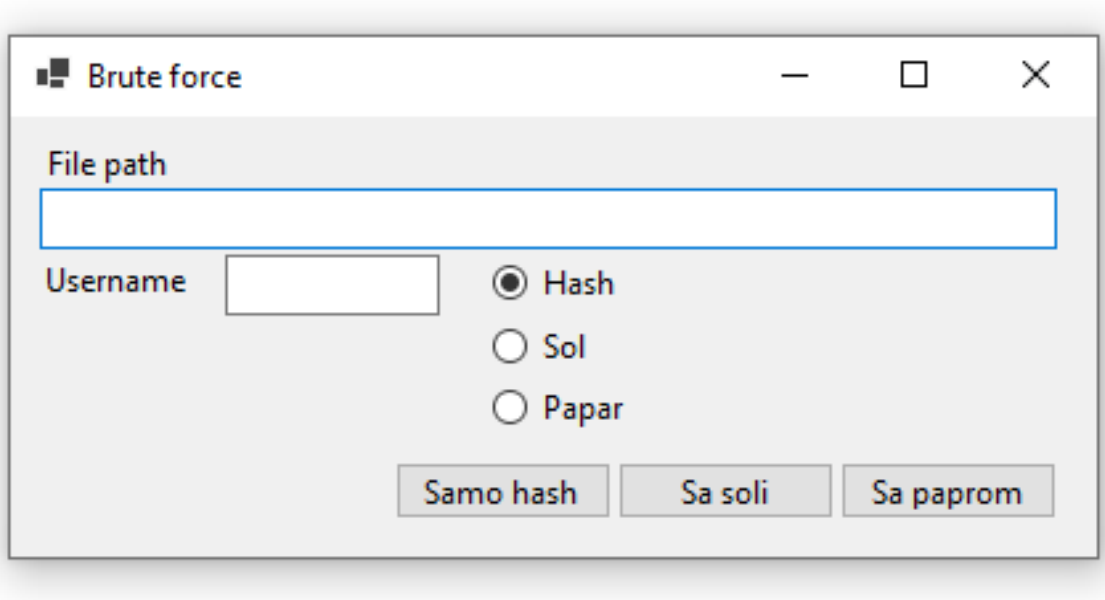



Slika 12: Baza podataka

Kada se registriramo, korisničko ime se zajedno sa kriptiranom lozinkom sprema u bazu podataka (slika 12.) u tekstualnom obliku (database.txt). Isto tako sprema se sol, *hash* sa soli i *hash* sa paprom. Prvo se sprema korisničko ime, pa sol, pa samo *hash*, pa *hash* i sol, pa *hash* i papar. Sve verzije sa *hashom* spremaju se u heksadekadskom obliku, duljine 64 znaka.

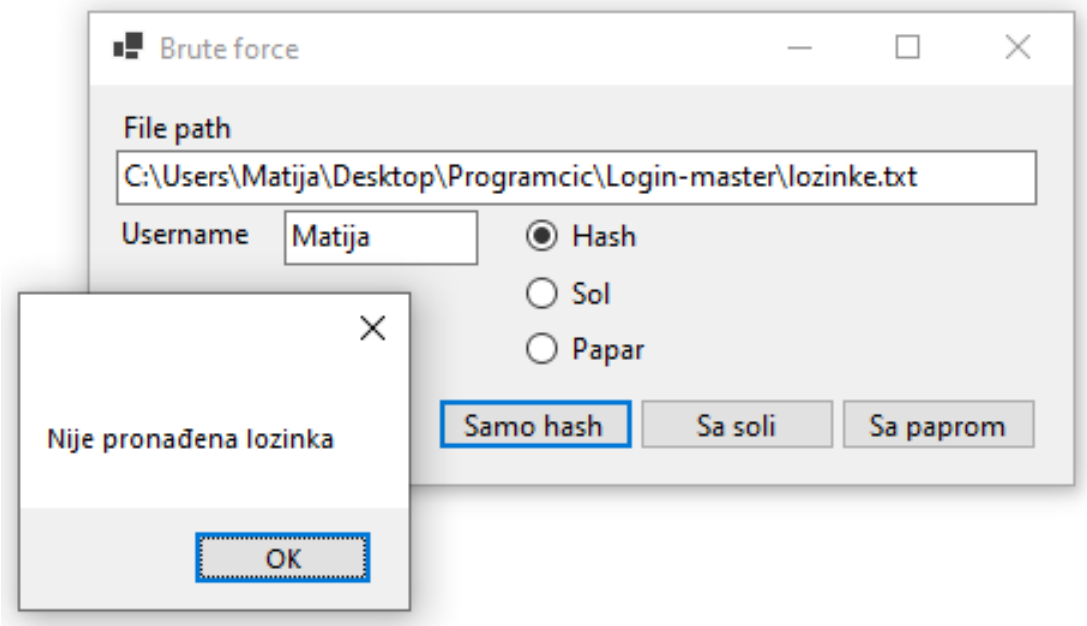
## 6.2. Napad grubom silom

Treća aplikacija služi za napad grubom silom prijašnjeg API-a pomoću rječnika. Kako bi se ovaj napad mogao izvesti, potrebno je napraviti tekstualnu datoteku sa lozinkama koju ćemo unijeti kao putanju pomoću koje će se pokušati razbiti lozinka iz baze podataka. Proizvoljno sam napravio takvu datoteku naziva „lozinke.txt“ iz razloga je ovaj primjer nije velik za razliku od pravih datoteka od nekoliko gigabajta, sa kojima bi napad predugo trajao. Treća aplikacija izgleda ovako:



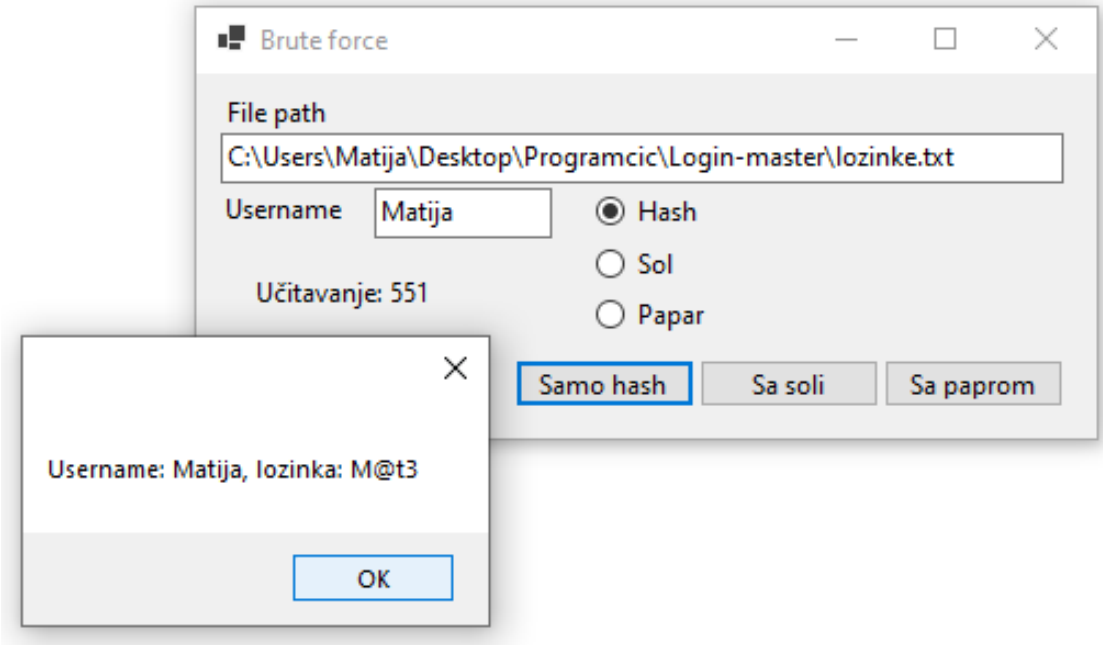

Slika 13: Izgleda aplikacije za napad grubom silom (*eng. brute force*)

Lozinku možemo saznati na tri načina: pomoću običnog *hasha*, soli i papra. Aplikacija će uspoređivati vrijednosti, te izbaciti ime lozinke ukoliko se ona nalazi u datoteci. Sada će biti prikazano kako aplikacija funkcionira, na primjeru kojeg smo uzeli za korisničku registraciju kod autentifikacije pomoću kriptografskih funkcija.




Slika 14: Neuspješni napad na lozinku sa *hashom*

Kako bi koristili aplikaciju potrebno je staviti putanju datoteke sa lozinkama, te korisničko ime. Ime datoteke je lozinke.txt i nalazi se u korijenskom direktoriju. Pošto naša lozinka se ne nalazi u datoteci, nije ju moguće saznati. Kada bismo željeli saznati lozinku pomoću *hasha*, morali bismo ju ručno unijeti u tu tekstualnu datoteku iz razloga što ova datoteka ne sadrži veliki broj lozinki. Sada ćemo ju unijeti i ponovno pokrenuti napad.




Slika 15: Uspješan napad na lozinku sa *hashom*

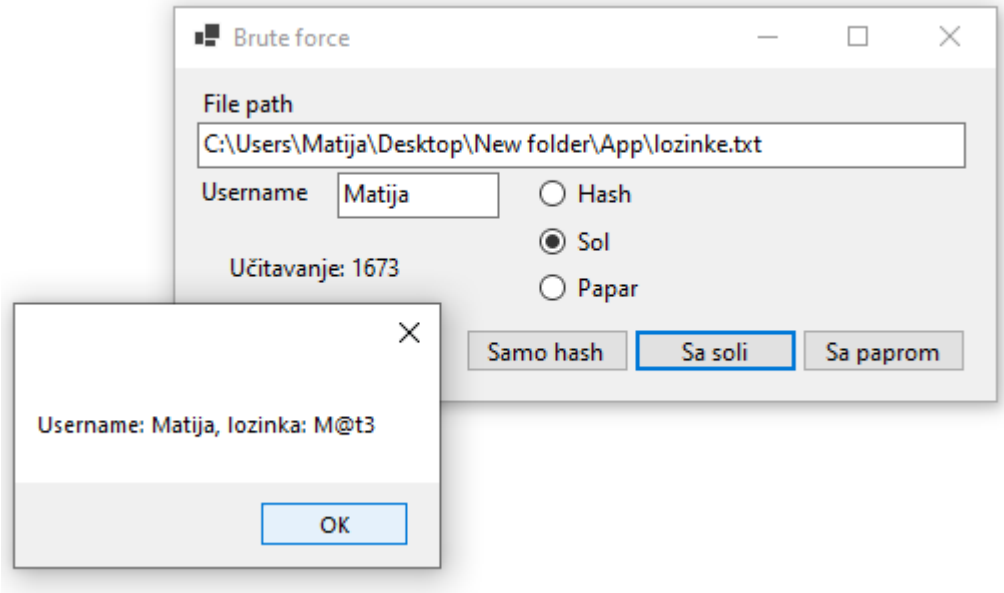
Napad se izvodi na način da funkcija iterira kroz datoteku i poziva API za svaku liniju, te ne koristi sol i papar. Iteracija se izvodi na način da se svaka lozinka u datoteci pretvara u *hash* vrijednost, te se ta učitana vrijednost uspoređuje sa *hash* vrijednosti korisničkog imena kojega napadamo. Naša lozinka koja se nalazi u lozinke.txt se pretvara u *hash* istim algoritmom kojeg smo koristili u API-u. Kada se pronađe lozinka, izbaci se potvrda o uspješnosti. Aplikacija još posjeduje „učitavanje“ gdje se može vidjeti na kojoj poziciji je lozinka smještena (u ovom slučaju 551 u dokumentu).

Kod za *brute force* pomoću *hasha* izgleda:

```
namespace BruteForce
{
    public partial class Form1 : Form
    {
        private const string _url = "https://localhost:44365/Account/";

        public Form1()
        {
            InitializeComponent();
        }
        private void btnHash_Click(object sender, EventArgs e)
        {
            BruteHash();
        }
        private async Task BruteHash()
        {
            lblLoading.Text = "Učitavanje";
            using (var fileStream = File.OpenRead(txtPath.Text))
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    string line;
                    string username = txtUsername.Text;
                    int i = 1;
                    string urlExtension = "Login";
                    if (rdbSalt.Checked)
                    {
                        urlExtension += "Salt";
                    }
                    if (rdbPepper.Checked)
                    {
```

```
                    urlExtension += "Pepper";
                }
                while ((line = streamReader.ReadLine()) != null)
                {
                    lblLoading.Text = $"Učitavanje: {i}";
                    var url = _url +
$"{urlExtension}?username={username}&hash={GetHashString(line)}";
                    HttpClient client = new HttpClient();
                    var response = await client.PostAsync(url, null);
                var result = await response.Content.ReadAsStringAsync();
                    if (result == "true")
                    {
                        MessageBox.Show($"Username: {username}, lozinka:
{line}");
                        lblLoading.Text = "";
                        return;
                    }
                    i++;
                }
                lblLoading.Text = "";
                MessageBox.Show($"Nije pronađena lozinka");
            }
        }
    }
```



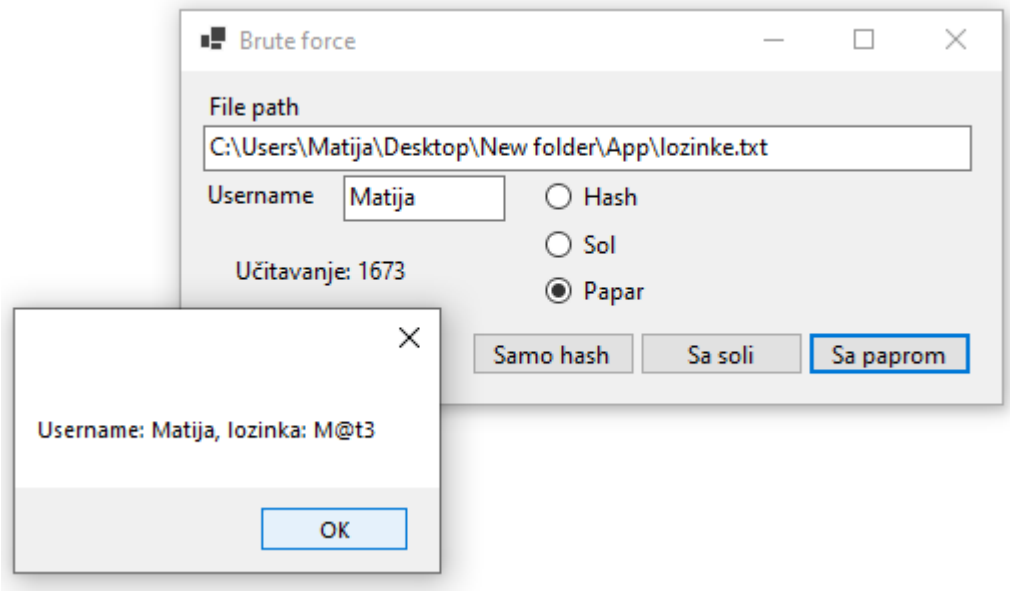

Slika 16: *Brute force* sa soli

Osim razbijanja običnog *hasha*, postoji mogućnost razbijanja lozinke pomoću soli (slika 16). To se radi pomoću funkcije koja iterira kroz datoteku i poziva API za svaku liniju te koristi sol. Dohvaća se sol iz baze podataka te se zajedno sa lozinkom iz rječnika pretvara u *hash* sa soli. Prolazi se linija po linija te kad se nađu iste *hash* sa soli vrijednosti, izbacuje se lozinka. Kod za to izgleda ovako:

```
private async Task BruteSalt()
        {
            lblLoading.Text = "Učitavanje";
            using (var fileStream = File.OpenRead(txtPath.Text))
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    string line;
                    string username = txtUsername.Text;
                    int i = 1;
                    string salt = await GetSalt();
                    string urlExtension = "Login";
                    if (rdbSalt.Checked)
                    {
                        urlExtension += "Salt";
                    }
                    if (rdbPepper.Checked)
                    {
                        urlExtension += "Pepper";
                    }
                    while ((line = streamReader.ReadLine()) != null)
                    {
                        lblLoading.Text = $"Učitavanje: {i}";
                        var url = _url +
$"{urlExtension}?username={username}&hash={GetHashString(line + salt)}";
                        HttpClient client = new HttpClient();
                        var response = await client.PostAsync(url, null);
                        var result = await
response.Content.ReadAsStringAsync();
                        if (result == "true")
                        {
                            MessageBox.Show($"Username: {username},
lozinka: {line}");
                            lblLoading.Text = "";
                            return;
```

```
                    }
                    i++;
                }
                lblLoading.Text = "";
                MessageBox.Show($"Nije pronađena lozinka");
            }
        }
    }
```

Ova aplikacija ima još mogućnosti i dobivanja lozinke korištenjem papra. Postoji funkcija koja iterira kroz datoteku i poziva API za svaku liniju i koristi papar. Iteracija se odvija na način da se svakoj lozinki doda papar (slučajno slovo od A-Z), te se onda *hashira* i uspoređuje sa *hashom* i paprom koji se nalaze u bazi podataka. Za razliku od *hasha* i soli, ovaj način razbijanja lozinke traje puno duže iz razloga što se za svaku lozinku u datoteci vrti 26 iteracija, kako bi se dobivena usporedila sa *hashom* i paprom koji se nalaze u bazi podataka.




Slika 17: *Brute force* papra

Kod za to izgleda ovako:

```
private async Task BrutePepper()
        {
            lblLoading.Text = "Učitavanje";
            using (var fileStream = File.OpenRead(txtPath.Text))
```

```
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    string line;
                    string username = txtUsername.Text;
                    int i = 1;
                    string urlExtension = "Login";
                    if (rdbSalt.Checked)
                    {
                        urlExtension += "Salt";
                    }
                    if (rdbPepper.Checked)
                    {
                        urlExtension += "Pepper";
                    }
                    while ((line = streamReader.ReadLine()) != null)
                    {
                        lblLoading.Text = $"Učitavanje: {i}";
                        for (var j = 65; j <= 91; j++)
                        {
                            var url = _url +
$"{urlExtension}?username={username}&hash={GetHashString(line + (char)j)}";
                            HttpClient client = new HttpClient();
            var response = await client.PostAsync(url, null);
                            var result = await
response.Content.ReadAsStringAsync();
                            if (result == "true")
                            {
                                MessageBox.Show($"Username: {username},
lozinka: {line}");
                                lblLoading.Text = "";
                                return;
                            }
                        }
                        i++;
                    }
                    lblLoading.Text = "";
                    MessageBox.Show($"Nije pronađena lozinka");
                }
            }
        }
```

## 6.3. Rezultati grubog napada na lozinke

*Brute force* napad pomoću rječnika ovisi o algoritmu pretvaranja lozinke u *hash* vrijednost, veličini lozinke, kao i o veličini rječnika. Ukoliko se radi o MD ili SHA algoritmu, tada ovaj napad ima smisla zbog njihove brzine. Ukoliko se radi o sporim *hash* funkcijama, napad bi predugo trajao pa samim time nije prikladan za prethodnu aplikaciju *brute force*. U aplikaciji se koristi SHA-256 algoritam koji pretvara lozinku od 4 znaka u *hash*, razlog tome je prikladna brzina pretvaranja lozinke u *hash* korištenjem manjeg broja znakova, kako bi se praktično prikazalo razbijanje lozinke. Isto tako koristi se i biblioteka od preko 2000 lozinki koja je ručno napravljena iz razloga što rječnici koji sadrže velik broj lozinki zauzimaju puno prostora i nisu praktični za ovakav prikaz napada, radi dugog čekanja pronalaska lozinke u rječniku.

Ukoliko se u aplikaciji koristi samo *hash* prilikom napada, vrijeme pronalaska je izrazito brzo, čita se riječ po riječ u rječniku (linija po liniju), te pretvara u *hash* i zatim uspoređuje sa *hash* vrijednosti korisničkog imena, koji se prilikom registracije spremio u bazu podataka. Ukoliko se koristi *hash* sa soli, napad traje duže zbog dohvaćanja soli, što u aplikaciji iznosi otprilike sekundu duže za pretraživanje lozinki u rječniku, u odnosu na samo *hash*. Vremenska razlika razbijanja lozinke sa soli, za razliku od razbijanja samo sa *hashom* nije značajna. Kada se dinamički računa *hash*, tada sol ne otežava previše posao. Ukoliko se kod soli koristi veća količina znakova, napad se može produžiti. Glavna karakteristika soli je ta da se ne mogu koristiti unaprijed izračunate tablice s *hash* vrijednostima. Dodavanje papra tu vremensku razliku znatno produljuje, razlog tomu je što se za svaku lozinku u rječniku dodaje 26 znakova na kraj lozinke kako bi se ispitala podudarnost istih. Zato, vrijeme traženja *hash* vrijednosti sa paprom traje do 26 puta duže za svaku pojedinu lozinku u rječniku nego sa običnim *hashom*, pa je tako razbijanje lozinki koja sadrži papar vremenski neučinkovito, odnosno ne isplati ga se raditi.

Prilikom testiranja aplikacije unio sam nekoliko istih lozinki za različite korisnike kako bih potvrdio da *hash* dobro radi. Ručno sam unio nekoliko različitih lozinki kako bih vidio postoji li šansa da dođe do kolizija, no do njih nije došlo iz razloga što je mala vjerojatnost pojavljivanja kolizije ručnim unošenjem lozinki. Bitno je bilo i prijašnje spremanje samo soli u bazu podataka, iz razloga što se svakim novim unosom generira nova sol, pa se u protivnom ne bi mogla pronaći lozinka ukoliko se ona ne može dohvatiti iz baze podataka. Sol i papar daju određenu dozu sigurnosti autentifikacijskom mehanizmu, no nema potrebe koristiti i jedno i drugo. Sol je teška za dohvatiti što napadaču dodatno otežava posao, ukoliko želi provaliti u sustav. S druge strane, napad na lozinku koja sadrži papar traje ekstremno dugo, pa je korištenje ove metode odlična obrana protiv napada rječnikom ili *rainbow* tablicama.

# 7. Zaključak

Autentifikacija pomoću lozinke kao rješenje pristupanja nekim informacijskim sustavima je najjeftiniji mehanizam, koji se lako implementira, te sadrži dovoljnu razinu sigurnosti za korisnika. Upravo zbog te jednostavnosti implementacije i malih novčanih izdavanja, lozinke su najčešći mehanizam ulazka u računalne i ostale operacijske sustave. Kod ovog principa autentifikacije gdje je potrebno unijeti lozinku, najčešće se pojavljuju zaporke generirane od strane korisnika koje odabiremo prilikom registracije u sustav. Sustav sam nalaže sigurnosne mjere koje lozinka mora sadržavati, kao određenu količinu znakova i njihovu kombinaciju kako bi korisnički račun bio otporniji na napade. Kada se prvi puta želimo prijaviti u sustav, potrebna je registracija korisnika gdje on unosi ime i lozinku pod kojom će se daljnje prijavljivati u sustav. Ti podaci spremaju se u bazu podataka ili čak u sam računalni kod sustava, a najčešće su kriptirani nekim algoritmima kako potencionalnom napadaču ne bi bili vidljivi i čitljivi.

Ovisno o sustavu u kojem se nalazimo, koriste se različiti algoritmi za hashiranje lozinke. Isto tako, uvid u bazu podataka omogućen je samo glavnim korisnicima što napadaču sprječava uvid u *hash* vrijednost lozinke pohranjene u bazi podataka. Kao što postoje brze *hash* funkcije za pretvaranje lozinki, poruka i teksta, tako postoje i spore, specijalizirane za lozinke gdje se produljuje vrijeme napada na korisnika. Razlika između brzih i sporih funkcija najviše ovisi o iteracijama lozinke kroz algoritam, kao i količini resursa kojom pritom računalo mora koristiti kako bi razbilo lozinku. U radu su istražene i objašnjene brze funckije sažimanja poput MD5 i SHA, koje koriste operacijski sustavi prilikom šifriranja lozinki, objašnjene su i spore funcije kao Argon i PBKDF koje se koriste kod osjetljivijih računalnih sustava kod kojih bi vrijeme napada kao *brute force*, predugo trajalo.

Kao dodatnu dozu sigurnosti unose se noviteti u šifriranju lozinki poput paprenja i soljenja. Ti koncepti pomažu na način da za svakim ponovnim unosom lozinke prilikom registriranja dobijemo potpuno novi izgled *hash* vrijednosti. Osim autentifikacije pomoću lozinke i funkcija sažimanja, u radu su objašnjeni i načini razbijanja lozinke kao i njihovo funkcioniranje kojima se pretežito služe napadači. U zadnjem poglavlju objašnjena je aplikacija koja se sastoji od registracije i prijave korisnika gdje se lozinka *hashira* SHA algoritmom, te se tom *hashu* dodaje sol i papar. Nakon ove aplikacije slijedi još jedna gdje se izvršava *brute force* napad, a podaci i rezultati regstracije, prijave i napada su detaljno objašnjeni u poglavlju aplikacije. Svakodnevna upotreba autentifikacije od strane korisnika prilikom korištenja računalnih sustava doveli su do toga da šifriranje lozinki više nije “novost” u osiguravanju sustava, već je to postalo standard kojim se sprečavaju provale u sustav.

# Popis literature

[1] Smith, S. L., Authenticating Users by Word Association," Computers and Security: Časopis o računalnoj tehnologiji i sigurnosti, izd.6, str. 100-470, 1987. [Na internetu] Dostupno: Password research, http://passwordresearch.com. [pristupano 22.01.2021].

[2] B. Menkus,“ Various User Authentification Mechanisms“, The EDP Audit: Control and Security Newsletter, broj 23, izdanje 9, str. 14 - 16 , siječanj 2015, [na internetu] Dostupno: Tandfonline, https://www.tandfonline.com. [pristupano 26.04.2021].

[3] D. Russel, i G.T.Gangemi Sr., Computer Security Basics, 1. izd., Cambridge, 1991.

[4] „Meriam–Webster Dictionary“, (bez dat.) dostupno na internetu: www.merriam-webster.com, [posjećeno 22.4. 2020].

[5] BBC News, „Computer Password inventor dies aged 93“, BBC online novine, izdano: 15.06.2019. Dostupno na: https:/www.bbc.com/news, https://www.bbc.com/news/technology-48988091, [pristupano: 12.03.2021].

[6] J.A. Cooper, Computer and Communications Security, Strategies for the 1990's, McGraw Hill Inc., New York, NY, 1989.

[7] B. Menkus, „Understanding the Use of Passwords“, Computers and Security: časopis o računalnoj tehnologiji , broj 2, izdanje 7, str. 132-136, travanj 1987. [na internetu] Dostupno: sciencedirect, htttp://www.sciencedirect.com, [pristupano 25.04.2021].

[8] R. Morris i K.Thompson „ Password Security: A case History,“ Communications of the ACM, broj 11, izdanje 22, str. 594-597, prosinac 1979. [Na internetu] , Dostupno: tech.cornell, http:/rist.tech.cornell.edu, [pristupano 02.02.2021].

[9] J.M.Johansson, Perfect Password: Selection, Protection, Authentification. O'Reill Media US, Canada: Syngress Publishing Inc. 2002.

[10] W.C. Summers, i E. Bosworth „Password Policy: The Good, The Bad, and The Ugly,“ 2015. [Na internetu] Dostupno: https://www.researchgate.net/publication/234799064_Password_policy_The_good_the_bad_and_the_ugly [pristupano: 20.03.2021].

[11] IBM – The Data Encryption Algorithm and the Data Encryption Standard (DES) [Na internetu]. DES OS 2.2.0 (2015) Dostupno: https://www.ibm.com/docs/en/zos/2.2.0?topic=cryptography-data-encryption-algorithm-data-encryption-standard [pristupano 28.06.2021.].

[12] Apple support (bez dat.) macOS User Guide [Na internetu] Dostupno: https://support.apple.com/guide/mac-help/use-keychains-to-store-passwords-mchlf375f392/mac [pristupano: 20.03.2021].

[13] S. Pillai „How are passwords stored in Linux (Understanding hashing with shadow utils)“, [Blog post]. 2013. [Na internetu]. Dostupno: https://www.slashroot.in/how-are-passwords-stored-linux-understanding-hashing-shadow-utils [pristupano 20.03.2021].

[14] Google Chrome Enterprise Help (bez dat.) *Monitor and prevent password reuse: Data privacy and security* [Na internetu]. Dostupno: https://support.google.com/chrome/a/answer/9696904?hl=en [pristupano 20.03.2021].

[15] S. Boonkrong, „Security of Passwords“, Information Technology: Časopis za informatičke napretke i istraživanja, sve. 2, izd 8, str. 112-116, prosinac 2012, [Na internetu] Dostupno: Information Technology Journalm, https://ph01.tci-thaijo.org/index.php/IT_Journal/ [pristupano: 21.03.2021]

[16] B. Preneel, *Analysis and design of cryptographic hash functions* [Doktorska disertacija]. Katolički Fakultet Leuven, Katholike Universitet te Leuven, Leuven 1993.

[17] R.Rivest, „The MD5 Message-Digest Algorithm“, travanj 1992. [Na internetu]. Dostupno: https://www.ietf.org/rfc/rfc1321.txt [pristupano 12.04.2021]

[18] „Cryptography Hash functions“ (03.02.2007). Tutorialspoint [Na internetu]. Dostupno: https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm [pristupano 12.04.2021].

[19] Network working group – Password-Based Crptography Specifiation – RFC (*Request for Comments*) standard [Na internetu]. Dostupno: https://www.ietf.org/standards/rfcs/ [pristupano 28.06.2021.]

[20] B. Kaliski, „PKCS #5: Password-Based Cryptography Specification Version 2.0“, rujan 2000. [Na internetu] Dostupno: https://www.ietf.org/rfc/rfc2898.txt [pristupano 12.04.2021].

[21] Symbionts (1996-2020) [Internetska stranica]. Dostupno: https://www.symbionts.de/tools/random-password-salt-generator.html

[22] S. Gibbs (2016, 15. pro.) „Password and hacking: the jargon of hashing, salting and SHA-2 explained,“ The Guardian. Dostupno: https://www.theguardian.com/technology/2016/dec/15/passwords-hacking-hashing-salting-sha-2 [pristupano 12.04.2021.].

[23] N. Provos, i D. Mazieres, „A Future-Adaptable Password Scheme“, The OpenBSD Project, str. 1-12, 2003. [Na internetu] Dostupno: OpenBSD http://openbst.org.com/. [pristupano 14.04.2021]

[24] F. Valsorda, „The Scrypt Parameters“, [Blog post]. 2017. [Na internetu]. Dostupno: https://blog.filippo.io/the-scrypt-parameters/ [pristupano 13.04.2021.].

[25] A. Biryukov, D. Dinu i D. Khovratovich [2015]. „Argon2: New Generation of Memory-Hard functions for Password Hashing and other Applications“, A. Biryukov, D.Dinu, 2016 iEEE European Symposium on Security and Privacy (EuroS&P), Saarbruecken, Njemačka.

[26] M. Razza, M. Iqbal, M. Sharif, i W. Haider, „A Survey of Password Attacks and Comparative Analysis on Methods for Secure Authentification“, Svjetski znanstveni časopis: World Applied Sciences Journal, sve. 19, izd. 4, str. 7, travanj 2012, [Na internetu]. Dostupno: IDOSI Publications [pristupano: 19.04.2021.].

[27] Ocpsoft [bez dat.] Password Regular Expression [Na internetu]. Dostupno: https://www.ocpsoft.org/tutorials/regular-expressions/password-regular-expression/ [pristupano 15.06.2021.].

[28] Stack Exchange (15.06.2015) *Salt generation in C#* [Na internetu]. Dostupno: https://codereview.stackexchange.com/questions/93614/salt-generation-in-c?fbclid=IwAR1xGgXgCM7-CG_MXqffSoqxWfH7UOwQ8J4VspDBjvnVSil1AO5QwJ7p3vc [pristupano 16.05.2021.].

# Popis slika

# Popis tablica