

Gegevensbeheer en beveiliging – Les 8

In deze les bekijken we hoe we een programma kunnen afschermen met een login-venster. De gebruiker moet een bestaande gebruikersnaam en bijhorend paswoord ingeven om de applicatie te kunnen starten. We gebruiken hiervoor een MySql database met een tabel *Users*.



We starten met een vereenvoudigd voorbeeld, we slaan het paswoord ongeëncrypteerd op in de tabel. **Dit is in een realistisch scenario geen goed idee.** Iedereen met toegang tot de tabel zou dan de paswoorden kunnen lezen. Later in dit document bekijken we enkele technieken om het paswoord veiliger op te slaan.


Het programma werkt zo:

1. Het programma toont een klein loginvenster en vraagt hier de gebruikersnaam en het wachtwoord.
2. De ingevoerde naam en wachtwoord wordt vergeleken met de database.
 - a. Als wachtwoord en gebruikersnaam niet juist zijn wordt een MessageBox getoond.
 - b. Als wachtwoord en gebruikersnaam juist zijn wordt een nieuw venster geopend en sluit het loginvenster.

Het programma maken

De database

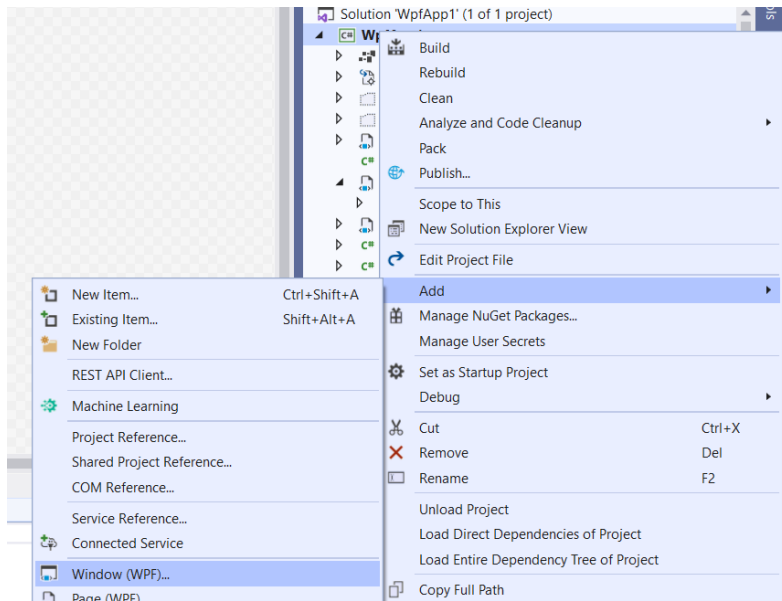
Maak een tabel *Users* ziet er zo uit:

#	Naam	Type
1	Id 	int(11)
2	Naam	varchar(255)
3	Password	varchar(128)

Zet via je database programma (phpMyAdmin, MySql Workbench of iets dergelijks) een gebruiker in de tabel met een niet geëncrypteerd paswoord 😬

Login Window

In Visual Studio maken we een nieuw *Window* aan door rechts te klikken in de Solution Explorer en te kiezen voor Add > Window.



Bewaar je bestand als LoginScreen.cs en maak ongeveer volgende layout:

Gebruik voor het wachtwoordveld het PasswordBox element. Hierdoor worden de karakters niet getoond tijdens het typen.

```
<PasswordBox x:Name="PasswordBox"></PasswordBox>
```

Scaffolden

Installeer de juiste package om met MySQL te kunnen communiceren. Installeer de tools om de nodige files te scaffolden en voer het *Scaffold-DbContext* commando uit om de juiste bestanden aan te maken (zie vorige les).

Code

Hieronder zie je voorbeeld code die de ingevoerde gebruikersnaam (tbxUsername.Text) vergelijkt met de namen van de users in de tabel (FirstOrDefault aanroep). Als er een user

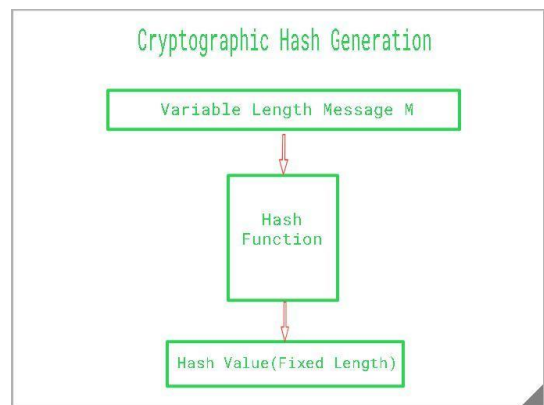
gevonden is wordt het ingegeven wachtwoord vergeleken. Als het wachtwoord correct is wordt MainWindow getoond en het loginvenster afgesloten met *this.Close()*.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    using (var db = new ShopContext())
    {
        var user = db.Users.FirstOrDefault(user => user.Naam == tbxUsername.Text);
        if (user == null)
        {
            MessageBox.Show("Username does not exist or password doesn't match.");
        }
        else
        {
            if (PasswordBox.Password == user.Password)
            {
                MainWindow program = new MainWindow();
                program.Show();
                this.Close();
            }
            else
            {
                MessageBox.Show("Username does not exist or password doesn't match.");
            }
        }
    }
}
```

Het wachtwoord *hashen*

Een niet versleuteld (“plain text”) wachtwoord in een database opslaan is een slecht idee. Iedereen die leestoeegang heeft tot de database (geoorloofd of ongeoorloofd) kan dan de wachtwoorden bekijken. Ook bij een lek, waarbij de databank tegen onze wil publiek gemaakt wordt, hebben we een groot probleem als de wachtwoorden in plain text in de database staan.

Hoe lossen we dit probleem? Een versleutelingsalgoritme zoals AES lost het probleem niet op want dan moeten we weer een sleutel opslaan die we geheim moeten houden...



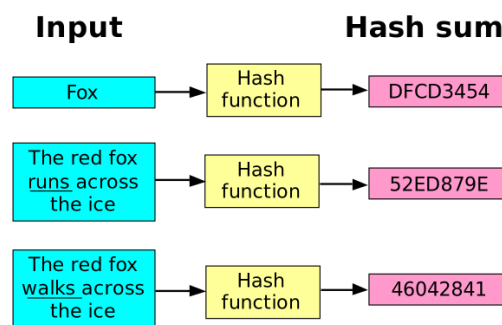
Hashing-functie

Een geschikt hashing-algoritme is een oplossing. Een hashing-algoritme of hashing-functie is een speciale functie: het heeft een input en een output. Als input geef je de functie data (zoals een stuk tekst, een wachtwoord of een heel bestand) en als output krijg je een willekeurig uitziende

reeks van bits: de hash. De hashing-functie maakt altijd een even lang reeks bits (of hash) *ongeacht de lengte van de input data*. Het algoritme SHA3-512 zet bijvoorbeeld data om in een hash van 512 bits. Het doet er niet toe of je het verzamelde werk van Shakespeare hasht of je email-adres, je krijgt altijd 512 bits. Hier kan je SHA3-512 en andere hash-functies uitproberen: https://emn178.github.io/online-tools/sha3_512.html

Een goede hash-functie berekent voor elke data-input een vrijwel unieke waarde. Je krijgt dus als het ware een *vingerafdruk* van je tekst of bestand dat je door de hash-functie stuurt. Elke input creëert een (zo goed als) unieke hash/vingerafdruk. Puur theoretisch is het mogelijk dat het hashen van 2 inputs dezelfde hash produceren. Vergelijk het met echte vingerafdrukken, het is puur theoretisch ook mogelijk dat 2 mensen dezelfde vingerafdruk kunnen hebben. Een goede hash-functies is echter zo gemaakt dat die kans verwaarloosbaar klein is. Je kan een hash dus bekijken als een unieke reeks bits voor een stuk data (in ons geval een wachtwoord).

Bijzonder aan een goede hash-functie is dat je heel moeilijk (of onmogelijk) van een hash terug kan rekenen naar de input-data. Een kleine wijziging in de inputdata geeft dan ook een totaal andere hash. Laat de tool https://emn178.github.io/online-tools/sha3_512.html eens de hash berekenen van 123456789 en van 123456788 (enkel het laatste cijfer is verschillend). Je zal zien dat je totaal verschillende hashes als resultaat krijgt.



Hashing voor wachtwoorden

Met een hash-functie kan je ervoor zorgen dat je geen wachtwoorden in de database meer moet opslaan. Je bewaart in plaats van het wachtwoord de *hash van het wachtwoord*.

De authenticatie gebeurt dan als volgt: de gebruiker geeft bij het inloggen zijn wachtwoord in, dat ingegeven wachtwoord wordt door het programma on-the-fly gehasht en vergeleken met de hash van het wachtwoord in de database.

Het wachtwoord van de gebruiker komt dus nooit op de harde schijf van de computer te staan. Om het wachtwoord dat de gebruiker intypt te achterhalen zou een aanvaller in het werkgeheugen van de computer moeten geraken wat niet evident is (of iets zoals een keylogger moeten gebruiken, wat dan weer niet zo moeilijk is als je fysieke toegang hebt tot de computer).

Evil Bob

Hashes zijn al een stap in de goede richting, maar zijn op zich niet voldoende om de wachtwoorden in een database te beveiligen. Er is nog een probleem: gebruikers kiezen vaak eenvoudige en makkelijk te raden wachtwoorden.



The infographic features a green background with a key icon on the left and a padlock icon on the right. The title 'Top 30 Most Used Passwords in the World' is prominently displayed in the center. Below the title, a table lists the top 30 most common passwords, numbered 1 through 30. The passwords are arranged in three columns: the first column contains passwords 1-10, the second column contains passwords 11-20, and the third column contains passwords 21-30.

Rank	Password	Rank	Password	Rank	Password
1	123456	11	abc123	21	princess
2	password	12	1234	22	letmein
3	123456789	13	password1	23	654321
4	12345	14	iloveyou	24	monkey
5	12345678	15	1q2w3e4r	25	27653
6	qwerty	16	000000	26	1qaz2wsx
7	1234567	17	qwerty123	27	123321
8	111111	18	zaq12wsx	28	qwertyuiop
9	1234567890	19	dragon	29	superman
10	123123	20	sunshine	30	asdfghjkl

Een hash-functie zal bij dezelfde input steeds dezelfde hash geven en dus bij het wachtwoord *123456* steeds dezelfde hash tonen. Als een aanvaller, laten we deze Bob noemen, de hashes berekent van de vaak voorkomende wachtwoorden (of wachtwoorden uit een woordenboek) kan Bob deze gaan gebruiken om te vergelijken met de wachtwoorden in de database. Als Bob de data van een grote database weet te bemachtigen (of kan downloaden van het internet) is dat een eenvoudige klus. Vele wachtwoorden zullen dezelfde hash hebben en dat zijn dan de meest gebruikte wachtwoorden zoals *123456* of *password*.

Hieronder zie je een voorbeeld van gelekte data van Adobe in 2013 waarbij de wachtwoorden van 38 miljoen actieve gebruikers (en 150 miljoen gebruikers in totaal) publiek werden gemaakt. Er zaten in de database ook “handige” hints bij 😲:

Adobe password data		Password hint
110edf2294fb8bf4	->	numbers 123456
110edf2294fb8bf4	->	==123456
110edf2294fb8bf4	->	c'est "123456"
8fda7e1f0b56593f	e2a311ba09ab4707	numbers
8fda7e1f0b56593f	e2a311ba09ab4707	1-8
8fda7e1f0b56593f	e2a311ba09ab4707	8digit
2fca9b003de39778	e2a311ba09ab4707	the password is password
2fca9b003de39778	e2a311ba09ab4707	password
2fca9b003de39778	e2a311ba09ab4707	rhymes with assword
e5d8efed9088db0b	->	q w e r t y
e5d8efed9088db0b	->	ytrewq tagurpidi
e5d8efed9088db0b	->	6 long qwert
ecba98cca55eabc2	->	sixxone
ecba98cca55eabc2	->	1*6
ecba98cca55eabc2	->	sixones

De site <https://scotthelme.co.uk/the-adobe-hack/> toont het resultaat van enkele SQL-queries op de gelekte Adobe-database . De conclusie is dat er bijna 2 miljoen identieke hashes in de database staan... dat zal hoogstwaarschijnlijk het wachtwoord 123456 zijn.

```
select password, count(*) from
adobe_creds.data group by password
order by count(*) desc limit 100;
```

PASSWORD	COUNT
EQ7flpT7i/Q=	1,911,522
j9p+HwtWWT86aMjgZFLzYg==	446,072
L8qbAD3jl3jioxG6CatHBw==	345,758
BB4e6X+b2xLioxG6CatHBw==	211,629
j9p+HwtWWT/ioxG6CatHBw==	201,546

Rainbow tables en salt

Ook wat moeilijkere wachtwoorden kunnen gekraakt worden met *rainbow tables*. Een rainbow table is een grote tabel waarbij een aanvaller voorberekende hashes gebruikt om wachtwoorden te kunnen kraken. Daardoor kan evil Bob al op voorhand een heleboel hashes berekenen en kan hij als hij een database in handen krijgt de hashes opzoeken. Wiskundige technieken worden gebruikt om de tabel beperkt te houden in omvang en het kraken te versnellen.

Bij een lang en ingewikkeld wachtwoord met willekeurige tekens wordt het kraken van wachtwoorden met rainbow tables erg moeilijk tot onmogelijk. Het lastige is dat je niet van alle gebruikers kan verwachten dat ze lange, ingewikkelde en voor elk programma of website unieke wachtwoorden onthouden (die ze dan nog liefst elke paar weken veranderen).

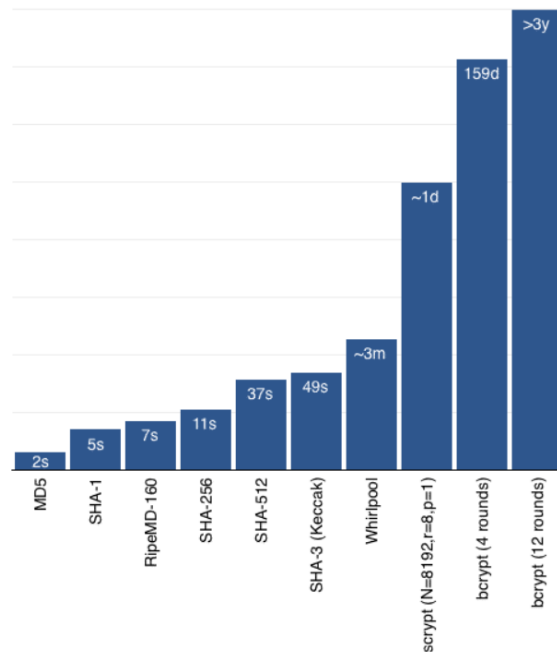
Een oplossing is om ons programma zelf een unieke reeks willekeurige bits te laten toevoegen aan het wachtwoord van de gebruiker: de *salt*. De unieke salt wordt toegevoegd aan het wachtwoord en daarvan wordt dan de hash berekend en opgeslagen. Voor elk wachtwoord krijgen we nu een unieke hash (ook voor al de gebruikers die 123456 gebruiken als wachtwoord,

dit wachtwoord zal natuurlijk wel gemakkelijk door een dictionary attack gevonden kunnen worden). De salt wordt enkel gebruikt om een rainbow table attack tegen te gaan en hoeft daarom niet geheim te worden gehouden (handig!), maar moet wel voor elk wachtwoord uniek zijn. Het geheim houden van de salt geeft wel een extra laag beveiliging maar zorgt voor een complexer systeem. Een geheime salt wordt een secret salt of ook wel pepper genoemd.

Password hashing in C# met bcrypt

Hoe doen we dit nu in de praktijk in C#? We gebruiken hiervoor een hashing-functie die goed werkt voor het hashen van wachtwoorden. Het populaire hashing-algoritme SHA3-512 is hiervoor niet zo geschikt omdat een aanvaller hier snel heel veel verschillende hashes mee kan berekenen wat het kraakproces vergemakkelijkt. Dat maakt het geen goed systeem voor als de salt gelekt is (bij een datalek). Het bcrypt-algoritme is een goed alternatief, dit is een trage hash: het berekenen kost meer tijd. De blogger

<https://www.novatec-gmbh.de/en/blog/choosing-right-hashing-algorithm-slowness/> deed een test en kwam op deze resultaten uit voor het kraken van een kort wachtwoord.



Time to brute force the clear text password "Pw#1!" hashed by a specific hashing algorithm on a NVIDIA Quadro M2000M GPU

Een ander voordeel van bcrypt is dat je de sterkte van het algoritme kan aanpassen zodat het in de toekomst kan meeschalen met de stijgende rekenkracht van de computers. In de figuur hierboven wordt het aantal rondes van bcrypt verhoogd (4 rondes en 12 rondes) wat resulteert in een tragere hash-berekening.

Bcrypt.NET

Gelukkig is het gebruik van bcrypt in C# eenvoudig 🤗. We gebruiken hiervoor een bestaand pakket (*don't roll your own!*): Bcrypt.NET. Dit is, zoals Entity Framework, een NuGet-pakket. Je vindt de installatie-regel hier: <https://www.nuget.org/packages/BCrypt.Net-Next/>

Als Bcrypt.NET geïnstalleerd is kan je deze using-regel gebruiken om de prefix BC te gebruiken in plaats van steeds BCrypt.Net.BCrypt te moeten typen.

```
using BC = BCrypt.Net.BCrypt;
```

Een wachtwoord hashen kan nu zo:

```
string passwordHash = BC.HashPassword("my password");
```

De variabele *passwordHash* bevat nu een wachtwoord met een salt er aan vast. Je hoeft je dus geen zorgen te maken over het opslaan van de salt, bcrypt doet dat voor ons. Je kan dit wachtwoord bewaren in een database en vergelijken met een ingegeven string van een gebruiker met de *Verify* methode:

```
if(BC.Verify("my password", passwordHash)) { ... }
```

Verify geeft een waarde true of false terug.

Oefening: Bcrypt implementeren

Kies één oefening van de vorige oefening van de opleiding uit en voorzie hiervoor een loginscherm.

Kan je een *Admin* knop maken in het programma waarmee je wachtwoorden kan instellen voor gebruikers? Probeer dit te maken. Denk je dat er security issues zijn met zo'n systeem? Hoe zou je ze kunnen oplossen?