عنوان: **نحوه‌ی محاسبه‌ی هش کلمات عبور کاربران در ASP.NET Identity**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۲۸ ۱۴:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Security, Cryptography, ASP.NET Identity

روش‌های زیادی برای ذخیره سازی کلمات عبور وجود دارند که اغلب آن‌ها نیز نادرست هستند. برای نمونه شاید ذخیره سازی کلمات عبور، به صورت رمزنگاری شده، ایده‌ی خوبی به نظر برسد؛ اما با دسترسی به این کلمات عبور، امکان رمزگشایی آن‌ها، توسط مهاجم وجود داشته و همین مساله می‌تواند امنیت افرادی را که در چندین سایت، از یک کلمه‌ی عبور استفاده می‌کنند، به خطر اندازد.

در این حالت هش کردن کلمات عبور ایده‌ی بهتر است. هش‌ها روش‌هایی یک طرفه هستند که با داشتن نتیجه‌ی نهایی آن‌ها، نمی‌توان به اصل کلمه‌ی عبور مورد استفاده دسترسی پیدا کرد. برای بهبود امنیت هش‌های تولیدی، می‌توان از مفهومی به نام Salt نیز استفاده نمود. Salt در اصل یک رشته‌ی تصادفی است که پیش از هش شدن نهایی کلمه‌ی عبور، به آن اضافه شده و سپس حاصل این جمع، هش خواهد شد. اهمیت این مساله در بالا بردن زمان یافتن کلمه‌ی عبور اصلی از روی هش نهایی است (توسط روش‌هایی مانند brute force یا امتحان کردن بازه‌ی وسیعی از عبارات قابل تصور).

اما واقعیت این است که حتی استفاده از یک Salt نیز نمی‌تواند امنیت بازیابی کلمات عبور هش شده را تضمین کند. برای مثال نرم افزارهایی موجود هستند که با استفاده از پرداش موازی قادرند بیش از 60 میلیارد هش را در یک ثانیه آزمایش کنند و البته این کارآیی، برای کار با هش‌های متداولی مانند MD5 و SHA1 بهینه سازی شده‌است.

روش هش کردن کلمات عبور در ASP.NET Identity

ASP.NET Identity 2.x که در حال حاضر آخرین نگارش تکامل یافته‌ی روش‌های امنیتی توصیه شده‌ی توسط مایکروسافت، برای برنامه‌های وب است، از استانداردی به نام RFC 2898 و الگوریتم PKDBF2 برای هش کردن کلمات عبور استفاده می‌کند. مهم‌ترین مزیت این روش خاص، کندتر شدن الگوریتم آن با بالا رفتن تعداد سعی‌های ممکن است؛ برخلاف الگوریتم‌هایی مانند MD5 یا SHA1 که اساسا برای رسیدن به نتیجه، در کمترین زمان ممکن طراحی شده‌اند.

PBKDF2 یا password-based key derivation function جزئی از استاندارد RSA نیز هست (PKCS #5 version 2.0). در این الگوریتم، تعداد بار تکرار، یک Salt و یک کلمه‌ی عبور تصادفی جهت بالا بردن انتروپی (بی‌نظمی) کلمه‌ی عبور اصلی، به آن اضافه می‌شوند. از تعداد بار تکرار برای تکرار الگوریتم هش کردن اطلاعات، به تعداد باری که مشخص شده‌است، استفاده می‌گردد. همین تکرار است که سبب کندشدن محاسبه‌ی هش می‌گردد. عدد معمولی که برای این حالت توصیه شده‌است، 50 هزار است.

این استاندارد در دات نت توسط کلاس Rfc2898DeriveBytes پیاده سازی شده‌است که در ذیل مثالی را در مورد نحوه‌ی استفاده‌ی عمومی از آن، مشاهده می‌کنید:

```
using System;
using System.Diagnostics;
using System.Security.Cryptography;
using System.Text;

namespace IdentityHash
{
    public static class PBKDF2
    {
        public static byte[] GenerateSalt()
        {
            using (var randomNumberGenerator = new RNGCryptoServiceProvider())
            {
                var randomNumber = new byte[32];
                randomNumberGenerator.GetBytes(randomNumber);
                return randomNumber;
            }
        }

        public static byte[] HashPassword(byte[] toBeHashed, byte[] salt, int numberOfRounds)
        {
            using (var rfc2898 = new Rfc2898DeriveBytes(toBeHashed, salt, numberOfRounds))
            {
                return rfc2898.GetBytes(32);

            }
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            var passwordToHash = "VeryComplexPassword";
            hashPassword(passwordToHash, 50000);
            Console.ReadLine();
        }

        private static void hashPassword(string passwordToHash, int numberOfRounds)
        {
            var sw = new Stopwatch();
            sw.Start();
            var hashedPassword = PBKDF2.HashPassword(
                                        Encoding.UTF8.GetBytes(passwordToHash),
                                        PBKDF2.GenerateSalt(),
                                        numberOfRounds);
            sw.Stop();
            Console.WriteLine();
            Console.WriteLine("Password to hash : {0}", passwordToHash);
            Console.WriteLine("Hashed Password : {0}", Convert.ToBase64String(hashedPassword));
            Console.WriteLine("Iterations <{0}> Elapsed Time : {1}ms", numberOfRounds,
sw.ElapsedMilliseconds);
        }
    }
}
```

شیء Rfc2898DeriveBytes برای تشکیل، نیاز به کلمه‌ی عبوری که قرار است هش شود به صورت آرایه‌ای از بایت‌ها، یک Salt و یک عدد اتفاقی دارد. این Salt توسط شیء RNGCryptoServiceProvider ایجاد شده‌است و همچنین نیازی نیست تا به صورت مخفی نگه‌داری شود. آن‌را می‌توان در فیلدی مجزا، در کنار کلمه‌ی عبور اصلی ذخیره سازی کرد. نتیجه‌ی نهایی، توسط متد rfc2898.GetBytes دریافت می‌گردد. پارامتر 32 آن به معنای 256 بیت بودن اندازه‌ی هش تولیدی است. 32 حداقل مقداری است که بهتر است انتخاب شود.

پیش فرض‌های پیاده سازی Rfc2898DeriveBytes استفاده از الگوریتم SHA1 با 1000 بار تکرار است؛ چیزی که دقیقا در ASP.NET Identity 2.x بکار رفته‌است.

تفاوت‌های الگوریتم‌های هش کردن اطلاعات در نگارش‌های مختلف ASP.NET Identity

اگر به سورس نگارش سوم ASP.NET Identity مراجعه کنیم، یک چنین کامنتی در ابتدای آن قابل مشاهده است:

```
/* =======================
* HASHED PASSWORD FORMATS
* =======================
*
* Version 2:
* PBKDF2 with HMAC-SHA1, 128-bit salt, 256-bit subkey, 1000 iterations.
* (See also: SDL crypto guidelines v5.1, Part III)
* Format: { 0x00, salt, subkey }
*
* Version 3:
* PBKDF2 with HMAC-SHA256, 128-bit salt, 256-bit subkey, 10000 iterations.
* Format: { 0x01, prf (UInt32), iter count (UInt32), salt length (UInt32), salt, subkey }
* (All UInt32s are stored big-endian.)
*/
```

در نگارش دوم آن از الگوریتم PBKDF2 با هزار بار تکرار و در نگارش سوم با 10 هزار بار تکرار، استفاده شده‌است. در این بین، الگوریتم پیش فرض HMAC-SHA1 نگارش‌های 2 نیز به HMAC-SHA256 در نگارش 3، تغییر کرده‌است.

در یک چنین حالتی بانک اطلاعاتی ASP.NET Identity 2.x شما با نگارش بعدی سازگار نخواهد بود و تمام کلمات عبور آن باید مجددا ریست شده و مطابق فرمت جدید هش شوند. بنابراین امکان انتخاب الگوریتم هش کردن را نیز پیش بینی کرده‌اند .

در نگارش دوم ASP.NET Identity، متد هش کردن یک کلمه‌ی عبور، چنین شکلی را دارد:

```
public static string HashPassword(string password, int numberOfRounds = 1000)
{
```

```
    if (password == null)
        throw new ArgumentNullException("password");

    byte[] saltBytes;
    byte[] hashedPasswordBytes;
    using (var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, 16, numberOfRounds))
    {
        saltBytes = rfc2898DeriveBytes.Salt;
        hashedPasswordBytes = rfc2898DeriveBytes.GetBytes(32);
    }
    var outArray = new byte[49];
    Buffer.BlockCopy(saltBytes, 0, outArray, 1, 16);
    Buffer.BlockCopy(hashedPasswordBytes, 0, outArray, 17, 32);
    return Convert.ToBase64String(outArray);
}
```

تفاوت این روش با مثال ابتدای بحث، مشخص کردن طول salt در متد Rfc2898DeriveBytes است؛ بجای محاسبه‌ی اولیه‌ی آن. در این حالت متد Rfc2898DeriveBytes مقدار salt را به صورت خودکار محاسبه می‌کند. این salt بجای ذخیره شدن در یک فیلد جداگانه، به ابتدای مقدار هش شده اضافه گردیده و به صورت یک رشته‌ی base64 ذخیره می‌شود. در نگارش سوم ، از کلاس ویژه‌ی RandomNumberGenerator برای محاسبه‌ی Salt استفاده شده‌است.

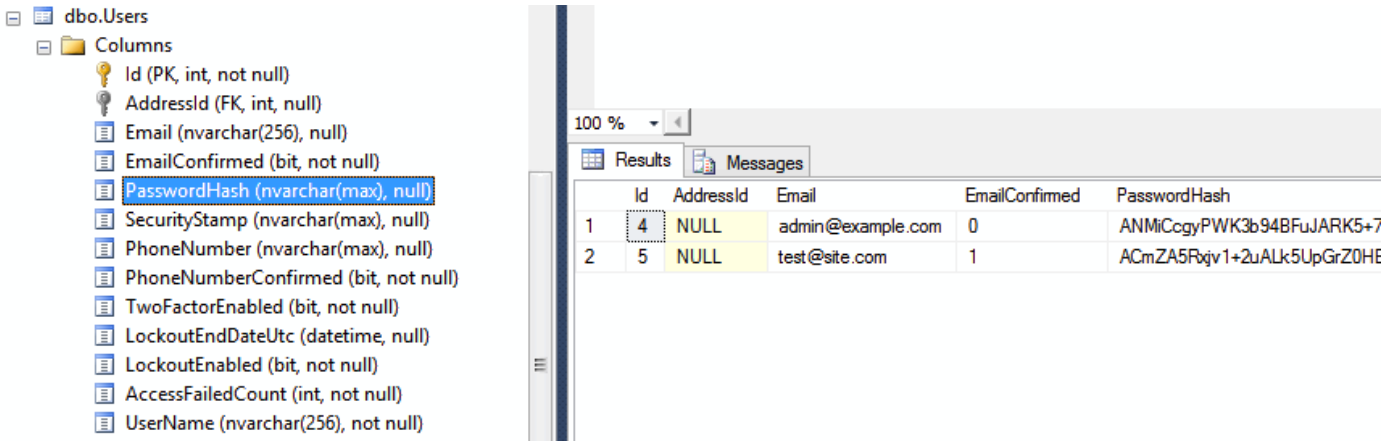
نظرات خوانندگان

نویسنده: امیر صیدی لو
تاریخ: ۱۳۹۴/۰۴/۱۵ ۶:۳۴

ممنون از مطلب خوبتون
ولی یه مشکلی که من موقع تست برخوردم این بود که زمان تبدیل آرایه تولید شده به وسیله تابع HashPassword به معادل رشته ای اون برای ذخیره در دیتابیس و بازیابی اون رشته به معادل آرایه اون برای چک کردن صحت کلمه عبور هر دو مقدار قبل از تبدیل و بعد از تبدیل با هم برابر بودن و مشکلی نداشتن ولی هنگام همین عمل تبدیل برای مقدار salt و بازیابیش از دیتا بیس مقدار قبل تبدیل و بعدش یکسان نبودن به همین خاطر مجبور شدم مقدار salt رو به صورت آرایه توی دیتابیس ذخیره کنم، خروجی حاصل از salt هم چک کردم نمی‌دونم چرا آرایه حاصل بیشتر از 32 خانه بود؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۰:۴

در ASP.NET Identity جمع هش و salt با فرمت base64 در بانک اطلاعاتی به صورت رشته‌ای با طول max ذخیره می‌شوند (هر دو با هم در یک فیلد). همچنین در اینجا طول salt به صورت صریح به 16 بایت تنظیم شده‌است (متد آخر مطلب).



نویسنده: امیر صیدی لو
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۰:۳۵

تو این حالت (یکی کردن salt و hashPassword) چطوری می‌تونیم مقدار salt رو از دیتا بیس بخونیم و با کلمه عبور ورودی کاربر جمع بزنیم و با مقدار hashPassword اولیه مقایسه کنیم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۴/۱۵ ۱۱:۹

از متدهای HashPassword و VerifyHashedPassword سورس ASP.NET Identity ایده بگیرید. مورد اول برای ذخیره سازی اطلاعات در بانک اطلاعاتی است. مورد دوم در حین لاگین، جهت تعیین اعتبار کلمه‌ی عبور کاربر استفاده می‌شود.