عنوان: **تبدیل فایل‌های pfx به snk**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۷/۲۹ ۱۱:۴۳
آدرس: www.dotnettips.info
گروه‌ها: Security, Certificate, PKCS, X509, Cryptography

مرسوم است و توصیه شده است که جهت ارائه کتابخانه‌های دات نتی خود از امضای دیجیتال استفاده کنید. VS.NET برای این منظور در برگه signing خواص یک پروژه، چنین امکانی را به صورت توکار ارائه می‌دهد.
حال اگر بخواهیم همین پروژه را به صورت سورس باز ارائه دهیم، استفاده کنندگان نهایی به مشکل برخواهند خورد؛ زیرا فایل pfx حاصل، توسط کلمه عبور محافظت می‌شود و در سایر سیستم‌ها بدون درنظر گرفتن این ملاحظات قابل استفاده نخواهد بود.
معادل فایل‌های pfx، فایلهایی هستند با پسوند snk که تنها تفاوت مهم آن‌ها با فایل‌های pfx، عدم محافظت توسط کلمه عبور است و ... برای کارهای خصوصا سورس باز انتخاب مناسبی به شمار می‌روند. اگر دقت کنید، اکثر پروژه‌های سورس باز دات نتی موجود در وب (مانند NHibernate، لوسین، iTextSharp و غیره) از فایل‌های snk برای اضافه کردن امضای دیجیتال به کتابخانه نهایی تولیدی استفاده می‌کنند و نه فایل‌های pfx محافظت شده.
در اینجا اگر فایل pfx ایی دارید و می‌خواهید معادل snk آن‌را تولید کنید، قطعه کد زیر چنین امکانی را مهیا می‌سازد:

```
using System.IO;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;

namespace PfxToSnk
{
    class Program
    {
        /// <summary>
        /// Converts .pfx file to .snk file.
        /// </summary>
        /// <param name="pfxData">.pfx file data.</param>
        /// <param name="pfxPassword">.pfx file password.</param>
        /// <returns>.snk file data.</returns>
        public static byte[] Pfx2Snk(byte[] pfxData, string pfxPassword)
        {
            var cert = new X509Certificate2(pfxData, pfxPassword, X509KeyStorageFlags.Exportable);
            var privateKey = (RSACryptoServiceProvider)cert.PrivateKey;
            return privateKey.ExportCspBlob(true);
        }

        static void Main(string[] args)
        {
            var pfxFileData = File.ReadAllBytes(@"D:\Key.pfx");
            var snkFileData = Pfx2Snk(pfxFileData, "my-pass");
            File.WriteAllBytes(@"D:\Key.snk", snkFileData);
        }
    }
}
```

نظرات خوانندگان

نویسنده: Mohsen
تاریخ: ۱۳۹۱/۰۷/۳۰ ۱۱:۹

آقای نصیری ممنون از لطف شما.
ممکنه بیشتر درمورد این امضا و نحوه‌ی کاربرد اون صحبت کنید؟(مثلا بنده یک کتابخانه‌ی آزمایشی را با استفاده از امضای موجود در بخش Signing امضا نموده و فایل pfx مربوطه را ساختم.اما اسمبلی مربوطه به سادگی در سایر پروژه‌ها قابل استفاده و حتی قابل مشاهده است(از طریق metadata)).

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۷/۳۰ ۱۱:۱۲

بله. این امضای دیجیتال، فقط به این معنا است که کار تولید شده متعلق به شما می‌باشد. هیچ نوع محدودیت دیگری را اعمال نمی‌کند.
+ وجود آن اندکی patch کردن برنامه‌ها رو مشکل می‌کنه. خصوصا در مورد برنامه‌های WPF و سیلورلایت.

نویسنده: سام ناصری
تاریخ: ۱۳۹۱/۱۲/۱۶ ۶:۲

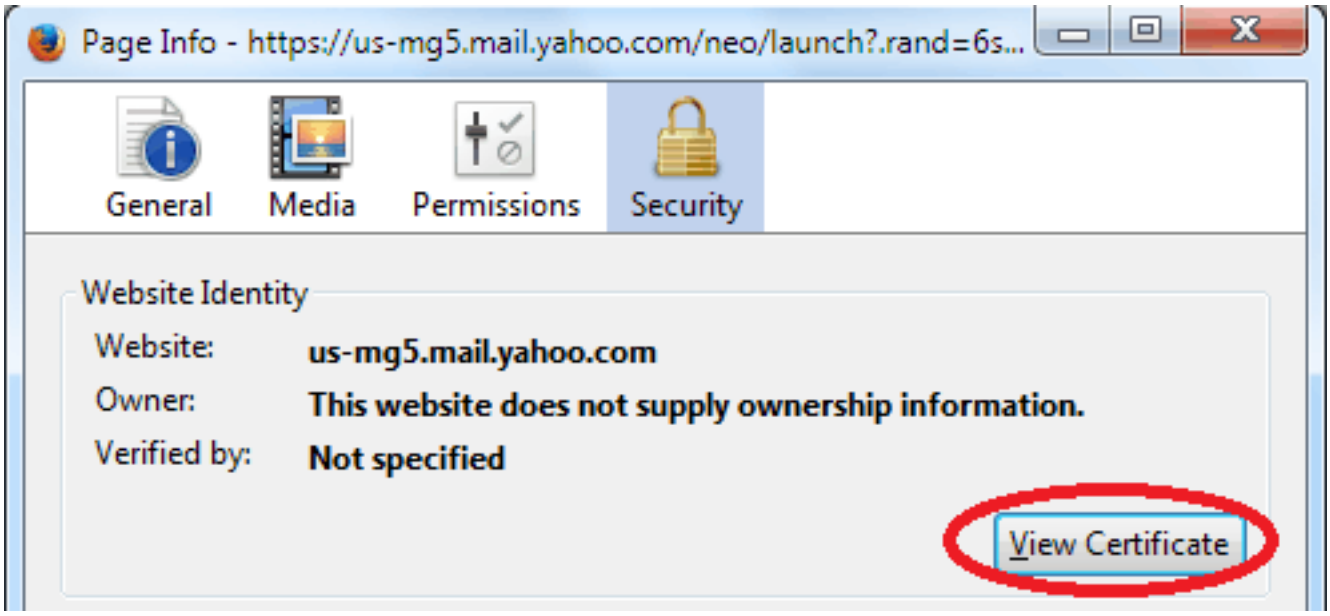
مطلب خوبی بود وحید جان. ممنونم.
البته من بعد از اینکه مطلب شما رو خوندم و متوجه شدم که دو نوع فایل pfx و snk هست که با اون میشه sign کرد اندکی تو اینترنت گشتم و متوجه یک نکته شدم که گفتم بد نیست اینجا مطرح کنم.
هر چند مطلب شما درباره تبدیل فایل pfx به snk است اما متنی که نوشتید این موضوع را القا میکند که نمیشود به سادگی این فایل رو ساخت.
به هر روی، میتوان فایل snk را از طریق فایل زبانه signing در خواص پروژه ساخت. برای این کار کافیست که گزینه Protect my key file with a password را آنتیک کرد و در این حالت به جای اینکه فایل pfx ساخته شود فایل snk ساخته میشود.
مطلب دیگر اینکه من پروژه‌های متن باز دیگری را دیده ام که الان حضور ذهن ندارم بگم(احتمالاً یکیشون RavenDB بود) که از طریق خواص پروژه ویژوال استودیو کار signing را انجام نمیدهند یعنی در آنجا گزینه sign کردن را انتخاب نکرده اند. چون فایل snk را اگر منتشر کنیم همه میتونند با اون اسمبلی‌ها را sign کنند و معنای strong name بودن اسمبلی به طور کلی میره زیر سوال.
در عوض از یک customized build استفاده میکنند که فقط توسط خودشون(مالکان پروژه) قابل فراخوانی است و توسط اون اسمبلی‌های release را میسازند. البته در اینباره باید بیشتر بررسی کنم و شاید دقیقاً ماجرا 100 درصد به این شکل که گفتم نیست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۱۲/۱۶ ۹:۳۱

- علت اینکه این مطلب رو نوشتم مربوط به زمانی بود که پروژه‌ای از قبل موجود بود با فایل pfx آن و قصد داشتم معادل محافظت نشده فایل pfx آن‌را تولید کنم.
- در مورد تولید فایل‌های pfx و snk یک مطلب نسبتا جامع در سایت داریم .
- به نظر من زمانیکه یک پروژه سورس باز است، امضا کردن اسمبلی‌های آن آنچنان مفهومی ندارد چون دسترسی به سورس و حتی ارائه آن بر اساس اطمینان به جامعه مصرف کننده صورت می‌گیرد. خیلی خیلی کم هستند موارد سوء استفاده از اسمبلی‌های امضاء شده به این صورت. مگر اینکه بحث پروژه کرنل لینوکس با تعداد مصرف کننده بالا و اهمیت امنیتی آن مطرح باشد که نیاز به امضای فایل‌های باینری آن وجود داشته باشد.

عنوان: دانلود مجوز SSL یک سایت HTTPS
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۷/۲۱ ۰:۵۵
آدرس: www.dotnettips.info
گروه‌ها: system.net, Cryptography, https, SSL

اگر به مرورگرها دقت کرده باشید، امکان نمایش SSL Server Certificate یک سایت استفاده کننده از پروتکل HTTPS را دارند. برای مثال در فایرفاکس اگر به خواص یک صفحه مراجعه کنیم، در برگه امنیت آن، امکان مشاهده جزئیات مجوز SSL سایت جاری فراهم است:



سؤال: چگونه می‌توان این مجوزها را با کدنویسی دریافت یا تعیین اعتبار کرد؟

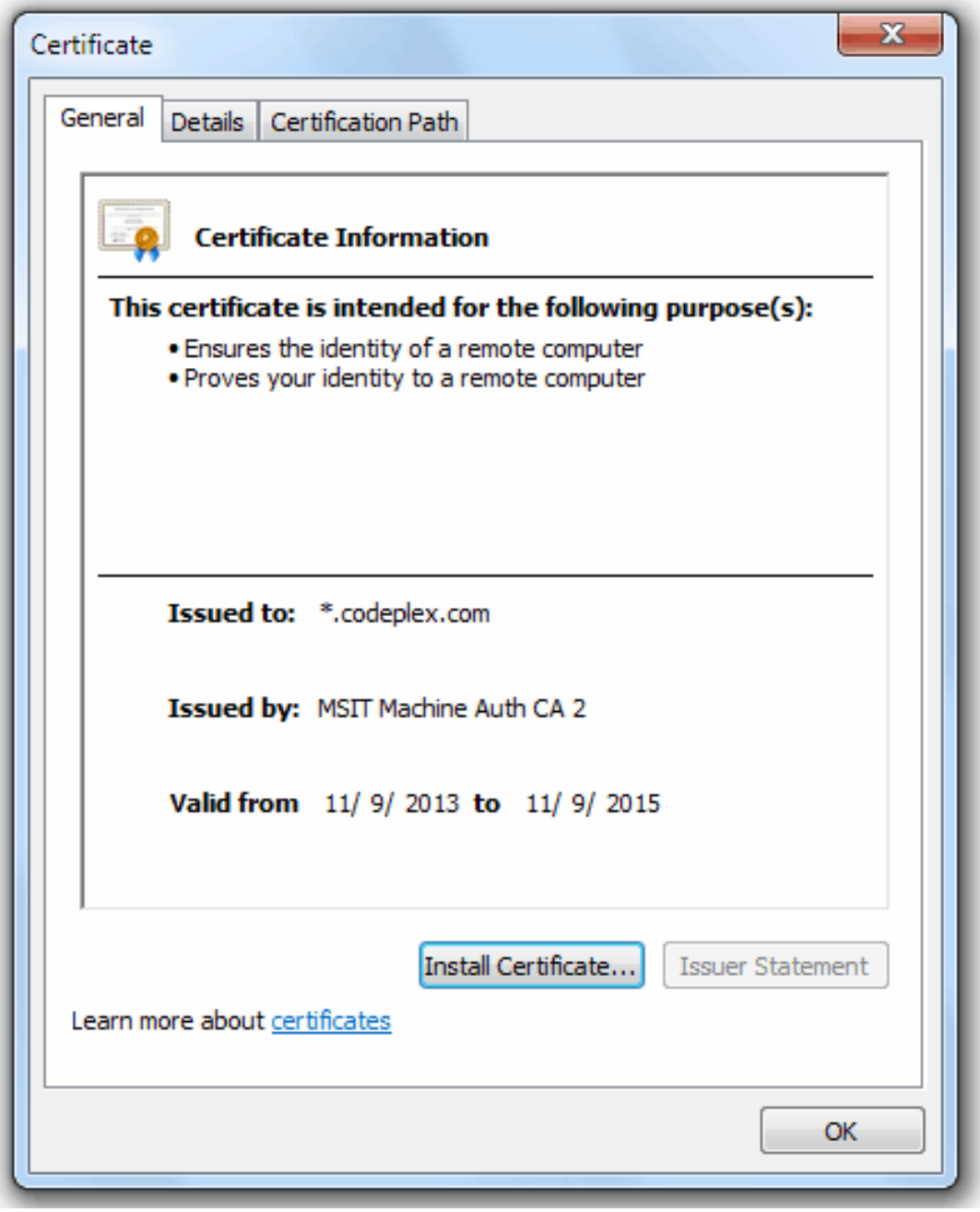
قطعه کد زیر، نحوه دریافت مجوز SSL یک سایت را نمایش می‌دهد:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Net;
using System.Security.Cryptography.X509Certificates;

namespace DownloadCerts
{
    class Program
    {
        static void Main(string[] args)
        {
            // صرفنظر از خطاهای احتمالی مجوز
            ServicePointManager.ServerCertificateValidationCallback = delegate { return true; };

            var url = "https://pdfreport.codeplex.com";
            var request = WebRequest.Create(url) as HttpWebRequest;
            request.Method = WebRequestMethods.Http.Head;
            using (var response = request.GetResponse())
            { /* در اینجا مجوز، در صورت وجود دریافت شده */  }

            if (request.ServicePoint.Certificate == null)
                return;

            // ذخیره سازی مجوز در فایل
            var cert = new X509Certificate2(request.ServicePoint.Certificate);
            Console.WriteLine("Expiration Date: {0}", cert.GetExpirationDateString());
            var data = cert.Export(X509ContentType.Cert);
```

```
            File.WriteAllBytes("site.cer", data);
            Process.Start(Environment.CurrentDirectory);
        }
    }
}
```

ممکن است مجوز یک سایت معتبر نباشد. کلاس WebRequest در حین مواجه شدن با یک چنین سایت‌هایی، یک WebException را صادر می‌کند. از این جهت که می‌خواهیم حتما این مجوز را دریافت کنیم، بنابراین در ابتدای کار، ServerCertificateValidation را غیرفعال می‌کنیم.
سپس یک درخواست ساده را به آدرس سرور مورد نظر ارسال می‌کنیم. پس از پایان درخواست، خاصیت request.ServicePoint.Certificate با مجوز SSL یک سایت مقدار دهی شده است. در ادامه نحوه ذخیره سازی این مجوز را با فرمت cer مشاهده می‌کنید.

Name	Date modified	Type
DownloadCerts.exe	۲۱ مهر ۱۳۹۲ ۱۲:۲۷ ق.ظ	Application
DownloadCerts.pdb	۲۱ مهر ۱۳۹۲ ۱۲:۲۷ ق.ظ	PDB File
DownloadCerts.vshost.exe	۲۱ مهر ۱۳۹۲ ۱۲:۲۹ ق.ظ	Application
DownloadCerts.vshost.exe.manifest	۲۶ اسفند ۱۳۸۸ ۰۹:۳۹ ب.ظ	MANIFEST File
site.cer	۲۱ مهر ۱۳۹۲ ۱۲:۲۹ ق.ظ	Security Certificate



نظرات خوانندگان

نویسنده: حمید حسین وند
تاریخ: ۱۳۹۳/۰۱/۲۲ ۱۶:۲۸

سلام؛ وقتی این گواهی یا certificate رو دانلود کردیم به چه دردمون میخوره؟ یعنی کارااییش برای ما چیه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۲۲ ۱۶:۵۸

جهت بررسی اعتبار آن می‌تواند مفید باشد. مثلا نوشتن برنامه‌ای مانند SSL Certificate Verifier

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.