VERSCHLÜSSELUNGSKONZEPTE ERKLÄRT

Grundlagen der Kryptografie

Die Kryptografie ist ein komplexes Thema. Das Verständnis einiger Begriffe macht das Entwicklerleben leichter.

Bei der Kryptografie geht es darum, Daten zu verschlüsseln, um zu verhindern, dass diese für nicht autorisierte Parteien zur Verfügung stehen. Hierzu werden die Ausgangsdaten – bezeichnet als Klartext – von einem Algorithmus unter Verwendung eines Schlüssels so umgewandelt, dass das Ergebnis – der sogenannte Geheimtext – nicht mehr lesbar ist. Nur wer im Besitz des passenden Schlüssels ist, kommt wieder an die Daten. Dieser Artikel stellt einige wichtige Konzepte, Begrifflichkeiten und Zusammenhänge vor.

Zufallszahlen

Den Schlüssel, der für die Ver- und Entschlüsselung benötigt wird, sollte es möglichst nur einmalig geben. Um dies sicherzustellen, werden Zufallszahlen generiert. Wer nun jedoch seine Gedanken in Richtung des .NET-Typs *System.Random* [1] lenkt, sollte vorsichtig sein, da dieser bei der Instanzierung einen Seed verwendet, das heißt einen Wert, der für die Initialisierung benötigt wird. In Bild 1 ist ein Beispiel zu sehen, wie zwei Instanzen mit dem identischen Seed generiert werden (rot markiert). Die zugehörigen rot markierten Ausgaben zeigen, dass hierdurch auch zweimal der identische „Zufallswert" generiert wurde. Die grünen Markierungen dagegen zeigen, wie mit einem anderen Seed ein anderer Zufallswert berechnet wurde.

Der parameterlose Standard-Konstruktor ermittelt einen Seed anhand der Systemuhr, was bedeutet, dass es durchaus passieren kann, dass bei mehreren nacheinander erstellten *System.Random*-Instanzen diese den gleichen „Zufallswert" produzieren. Da die Zufallszahlen für die Key-Generierung zwingend benötigt werden, ist somit *System.Random* für den Einsatz in der Kryptografie ungeeignet.

```
var random1 = new Random(Seed: 10);
var random2 = new Random(Seed: 10);
var random3 = new Random(Seed: 11);

ShowRandom(nameof(random1), random1);
ShowRandom(nameof(random2), random2);
ShowRandom(nameof(random3), random3);

3 references
static void ShowRandom(string description,
    Random random)
{
    Console.WriteLine(description);
    byte[] values = new byte[5];
    random.NextBytes(values);
    foreach (byte value in values)
        Console.Write("{0, 5}", value);
    Console.WriteLine();
}
```

```
random1
  205  173  195   47   63
random2
  205  173  195   47   63
random3
  249  113    3   66   35
```

Der .NET-Typ System.Random (Bild 1)

```
using System.Security.Cryptography;

for (int counter = 0; counter < 5; counter++)
{
    var numbers = RandomNumberGenerator.GetBytes(32);
    Console.WriteLine($"{counter} - {Convert.ToBase64String(numbers)}");
}
```

```
No issues found
0 - 2uUeA7BmpGjnOi1V9Takr0xtqcoD8+Qfx1kdVI4vSqQ=
1 - H8YEENaiJ/4MgSxavKfBm7O8ckZm0+zz5uVOZHLIInM=
2 - 8VbvmXcB/YQYLc0oehK+7FvkXaHCDjKz73LqWOemaaA=
3 - rHarm/BJ9IuCmzkKegru8h4XjqSiO0OPEK75qN/YlYU=
4 - QZ9z7xVCmSO5E0v5tzd7ak6N8UdjycQlxayKpublWas=
```

Verwendung des RandomNumberGenerator (Bild 2)

.NET stellt für die Generierung von Zufallszahlen im Kontext der Kryptografie den Typ *RandomNumberGenerator* zur Verfügung, der bei Generierung zwar langsamer ist, aber gegenüber dem Stellenwert der Absicherung von Daten fällt dieser Nachteil nicht ins Gewicht. Bild 2 zeigt die Verwendung dieses Typs. Die Ausgabe enthält die fünf generierten Zufallswerte, es gibt keinerlei Übereinstimmung.

Mit dem Parameterwert *32* beim Aufruf von *RandomNumberGenerator* wird angegeben, dass 32 Bytes von Zufallswerten zu generieren sind.

```
RandomNumberGenerator.GetBytes(32)
```

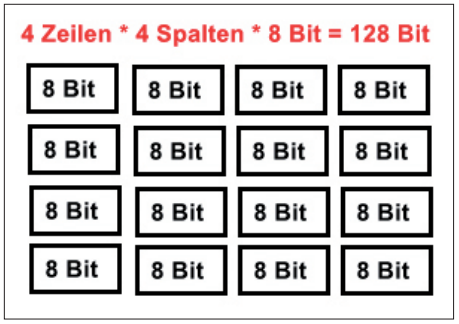
Um die generierten Zufallswerte in einem lesbaren Format zu erhalten, werden diese in Base64 konvertiert. Obwohl auch für diesen Vorgang der Begriff Encodieren verwendet wird, handelt es sich hier nicht um eine Verschlüsselung; der Base64-String kann problemlos wieder in das ursprüngliche Byte-Array-Format zurückverwandelt (decodiert) werden.

Symmetrische Verschlüsselung – AES

Bei der symmetrischen Verschlüsselung kommt lediglich ein Schlüssel zum Einsatz. Dieser eine Schlüssel wird sowohl zum Verschlüsseln als auch zum Entschlüsseln verwendet.

Die symmetrische Verschlüsselung hat folgende Merkmale:

- Sie ist sehr sicher. Der Advanced Encryption Standard (AES [3]) Standard wurde von Joan Daemen und Vincent Rijmen unter der Bezeichnung „Rijndael" entwickelt und löste den Data Encryption Standard (DES [4]) ab, der nur eine geringe Schlüssellänge von 56 Bit unterstützte.
- Die symmetrische Verschlüsselung ist im Vergleich zur asymmetrischen Verschlüsselung, die später erläutert wird, deutlich schneller.
- Den Schlüssel sicher auszutauschen ist eine Herausforderung. Sollte dieser in die falschen Hände geraten, könnte



AES-Tabelle mit 128 Bit (Bild 3)

```
internal static class EncryptionAes
{
    1 reference
    public static byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] initVector)
    {
        using var aes = Aes.Create();
        aes.Key = key;
        return aes.EncryptCbc(dataToEncrypt, initVector);
    }

    1 reference
    public static byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] initVector)
    {
        using var aes = Aes.Create();
        aes.Key = key;
        return aes.DecryptCbc(dataToDecrypt, initVector);
    }
}
```

Klasse zur AES-Verschlüsselung (Bild 4)

zwischen den rechtmäßigen Sender und Empfänger ein „Man in the middle" installiert werden, der die versendeten Daten vor dem Empfänger entgegennimmt, entschlüsselt und sie bei Bedarf manipuliert, verschlüsselt und weiterleitet. Der Empfänger könnte somit mit falschen Informationen versorgt werden.

AES verwendet folgende Parameter:

- Eine Schlüssellänge von 128, 192 oder 256 Bit.
- Einen 16-Byte-Initialisierungsvektor. Dieser Wert wird sowohl für die Ver- als auch die Entschlüsselung eingesetzt, um eine gewissen Entropie, das heißt eine Unordnung, zu erreichen.

Beim AES-Verfahren handelt es sich um eine sogenannte „Blockchiffre" oder auch „block cipher". Damit wird ein deterministisches Verschlüsselungsverfahren bezeichnet, das einen Klartextabschnitt fester Länge (die zu verschlüsselnden Daten in lesbarer Form) in einen Geheimtext abbildet. Wie bereits beschrieben kann beim verwendeten Schlüssel zwischen einer Größe von 128, 192 oder 256 Bit gewählt werden.

Beim AES wird zunächst jeder Block in eine Tabelle mit vier Zeilen geschrieben. Die Anzahl der Spalten hängt von der Größe ab. Jede Zelle in der Tabelle enthält 8 Bit; bei 128 Bit hat die Tabelle also vier Spalten (Bild 3). Bei einer Länge von 256 Bit hätte die Tabelle acht Spalten.

Der Vorgang der Verschlüsselung durchläuft mehrere Runden. Jede Runde vollzieht hierbei folgende Schritte:

- **Substitution:** Jedes Byte wird unter Verwendung einer Substitutionsbox, auch S-Box genannt, verschlüsselt. Enthielt beispielsweise eine Zelle ursprünglich das Symbol *A(0,0)*, enthält sie danach das Symbol *G(0,0)*. Jedes Byte wird durch einen anderen Wert ersetzt.
- **Shift Row:** Jede Zeile in der Tabelle wird um eine bestimmte Anzahl von Spalten nach links verschoben.
- **Mix Column:** Die Spalten werden mithilfe von Matrixmanipulationen vermischt.
- **Key Addition:** Die Matrix wird mit einem Rundenschlüssel, der aus dem Eingabeschlüssel generiert wird, mit der *XOR*-Operation verknüpft.

Diese Runden werden mehrfach durchlaufen. Das Verfahren ist vollständig umkehrbar, das heißt, die verschlüsselten Daten können tatsächlich auch wieder entschlüsselt werden.

So viel zur Theorie. In der Praxis kann dies, wie in Bild 4 zu sehen, in C# umgesetzt werden. Die Klasse *EncryptionAes* bietet die beiden Funktionen *Encrypt* und *Decrypt* an. Erstere wird mit den zu verschlüsselnden Daten, dem Schlüssel, der sowohl dem Sender als auch dem Empfänger bekannt sein muss, und dem Initialisierungsvektor versorgt. Die zweite Funktion erwartet dagegen die verschlüsselten Daten, den gemeinsamen Schlüssel und den Initialisierungsvektor.

Die Verwendung dokumentiert Bild 5. Zunächst werden der gemeinsame Verschlüsselungs-Key und der Initialisierungsvektor mit Aufrufen an

```
RandomNumberGenerator.GetBytes
```

erzeugt, welche für die Verschlüsselung (rot markiert) und Entschlüsselung (grün) angewendet werden.

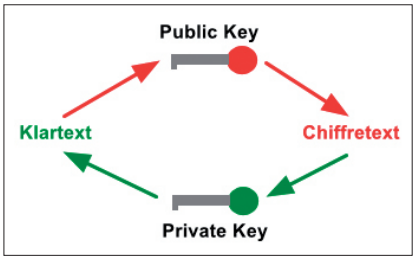
```
originalMessage: DotNetPro ist klasse!
Encrypted message: ZQXwrjhk5XWIySRzFVoXOEi69+KN4vv8aD0pBYi/4yE=
Decrypted message: DotNetPro ist klasse!

ryptographie - Program.cs

tographie
const string originalMessage = "DotNetPro ist klasse!";
var key = RandomNumberGenerator.GetBytes(32);
var initVector = RandomNumberGenerator.GetBytes(16);

var encrypted = EncryptionAes.Encrypt(Encoding.UTF8.GetBytes(originalMessage), key, initVector);
var decrypted = EncryptionAes.Decrypt(encrypted, key, initVector);

var decryptedMessage = Encoding.UTF8.GetString(decrypted);

Console.WriteLine($"originalMessage: {originalMessage}");
Console.WriteLine($"Encrypted message: {Convert.ToBase64String(encrypted)}");
Console.WriteLine($"Decrypted message: {decryptedMessage}");
```

Daten mittels AES ver- und entschlüsseln (Bild 5)

Asymmetrische Verschlüsselung – RSA

Die symmetrische Verschlüsselung ist eine gute Sache. Die Herausforderung ist jedoch, sicherzustellen, dass der Key sicher zwischen den Kommunikationspartnern ausgetauscht wird; dies wurde bereits zuvor mit dem Hinweis auf ▶

den „Man in the middle“ erklärt. Hier kommt die asymmetrische Verschlüsselung ins Spiel, die zwei unterschiedliche Schlüssel nutzt. Sie arbeitet mit einem „Public Key“, der prinzipiell jedem zur Verfügung steht und zur Verschlüsselung zum Einsatz kommt (Bild 6). Die Entschlüsselung kann nur durch den „Private Key“ erfolgen, der mathematisch mit dem Public Key verknüpft ist. Der Private Key steht nur dem Empfänger zur Verfügung, um die verschlüsselte Nachricht zu entschlüsseln.



Asymmetrische Verschlüsselung (Bild 6)

Dieses Verfahren wird als asymmetrisch bezeichnet, da im Gegensatz zur symmetrischen Verschlüsselung zwei Schlüssel mit unterschiedlichen Aufgaben zum Einsatz kommen: der Public Key zum Verschlüsseln und der Private Key zum Entschlüsseln. Bei der symmetrischen Verschlüsselung dagegen wird lediglich ein Schlüssel genutzt, der sowohl für die Verschlüsselung als auch für die Entschlüsselung Verwendung findet.

Einige Merkmale der asymmetrischen Verschlüsselung:

- Im Vergleich zur symmetrischen Verschlüsselung ist der Vorgang, Nachrichten zu verschlüsseln und zu entschlüsseln, deutlich langsamer.
- Auch die Menge der auf einmal zu verschlüsselnden Daten ist limitiert.
- Daher wird sie gerne verwendet, um symmetrische Schlüssel zu verschlüsseln.
- Die Schlüssellänge kann zwischen 1024, 2048 und 4096 Bit variieren, wobei 1024-Bit-Schlüssel heute als nicht mehr sicher gelten.

Bild 7 zeigt den ersten Abschnitt der C#-Klasse. Zunächst wird im Konstruktor eine Instanz des .NET-Typs *RSA* erstellt. Die Factory-Funktion wird mit der Information der Schlüssellänge von 2048 aufgerufen. Als Nächstes folgt die Funktion *Encrypt*, sie wandelt vor der Verschlüsselung die zu verschlüsselnde Zeichenkette zunächst in ein Byte-Array um. Das Gegenstück hierzu, die Funktion *Decrypt*, dient dazu, ein Byte-Array mit verschlüsselten Daten wieder zu entschlüsseln.

In Bild 8 sind die Funktionen der RSA-Klasse zu sehen, die für den Export und den Import des Public und des Private Keys genutzt werden. Bild 9 veranschaulicht, wie die Klasse zur Ver- und Entschlüsselung verwendet wird.

Wie die Schlüsselinformationen exportiert und wieder importiert werden, zeigt Bild 10. Die erste Instanz der RSA-Klasse (grün markiert) exportiert diese und verschlüsselt anschließend die Daten. Die grünen Teile entsprechen somit dem Versender. Die zweite RSA-Klasse (rot) steht für den Empfänger. Sie importiert beide Schlüsselinformationen und entschlüsselt die verschlüsselte Information.

Hybride Verschlüsselung mit AES und RSA

Der Inhalt der vorangegangenen Abschnitte kann damit wie folgt zusammengefasst werden:

- Die symmetrische Verschlüsselung ist recht flott im Vergleich zur asymmetrischen Verschlüsselung.
- Der sichere Austausch eines symmetrischen Schlüssels ist schwierig und birgt Sicherheitsrisiken.
- Der Schlüsselaustausch der asymmetrischen Verschlüsselung ist deutlich einfacher. Der Zugriff auf den Public Key kann grundsätzlich jedem gewährt werden. Nur der Private Key muss sicher verwahrt werden.

Daher bietet es sich an, das Beste aus beiden Welten miteinander zu kombinieren, was als hybride Verschlüsselung bezeichnet wird. Symmetrische Schlüssel werden mit RSA verschlüsselt und können dadurch sicher ausgetauscht werden.

Als Beispiel soll der sichere Nachrichtenaustausch zwischen zwei Chat-Partnern, Frank und Martina, dienen. Frank

```
internal class EncryptionRSA
{
    private readonly RSA _rsa;
    0 references
    public EncryptionRSA()
    {
        const int keySize = 2048;
        _rsa = RSA.Create(keySize);
    }

    0 references
    public byte[] Encrypt(string dataToEncrypt)
    {
        return _rsa.Encrypt(Encoding.UTF8.GetBytes(dataToEncrypt),
            RSAEncryptionPadding.OaepSHA256);
    }

    0 references
    public byte[] Decrypt(byte[] dataToDecrypt)
    {
        return _rsa.Decrypt(dataToDecrypt, RSAEncryptionPadding.OaepSHA256);
    }
```

ctor und erste Funktionen der RSA-Klasse (Bild 7)

```
public byte[] ExportPrivateKey(int iterationCount, string password)
{
    var keyParams = new PbeParameters(
        PbeEncryptionAlgorithm.Aes256Cbc, HashAlgorithmName.SHA256, iterationCount);

    var encryptedPrivateKey =
        _rsa.ExportEncryptedPkcs8PrivateKey(Encoding.UTF8.GetBytes(password), keyParams);
    return encryptedPrivateKey;
}

1 reference
public void ImportEncryptedPrivateKey(byte[] encryptedPrivateKey, string password)
{
    _rsa.ImportEncryptedPkcs8PrivateKey(Encoding.UTF8.GetBytes(password), encryptedPrivateKey, out _);
}

1 reference
public byte[] ExportPublicKey()
{
    return _rsa.ExportRSAPublicKey();
}

1 reference
public void ImportPublicKey(byte[] publicKey)
{
    _rsa.ImportRSAPublicKey(publicKey, out _);
}
```

Private und Public Key der RSA-Klasse (Bild 8)

möchte eine Nachricht an Martina senden und vollzieht dazu für den Versand folgende Schritte:

- Generierung eines AES-Session-Keys (symmetrische Verschlüsselung).
- Generierung eines Initialisierungsvektors (Zufallswert, dient der Entropie).
- Verschlüsselung der Nachricht mit dem AES-Schlüssel und dem Initialisierungsvektor.
- Verschlüsselung des AES-Session-Keys mit dem Public Key von Martina (RSA).
- Versand (1.) der verschlüsselten Daten, (2.) des verschlüsselten Session Keys und (3.) des Initialisierungsvektors an Martina.

Auf der Empfangsseite muss Martina folgende Schritte durchlaufen:

- Entschlüsselung des empfangenen, mit dem RSA-Public-Key verschlüsselten AES-Session-Keys mit dem zugehörigen RSA-Private-Key.
- Entschlüsselung der Nachricht mit dem entschlüsselten AES-Session-Key und dem Initialisierungsvektor.

Nachdem Martina die Nachricht gelesen hat, möchte sie Frank antworten und vollzieht hierfür folgende Schritte:

- Generierung eines neuen AES-Session-Keys (symmetrische Verschlüsselung). Der empfangene Session Key wird nicht wieder verwendet.
- Generierung eines neuen Initialisierungsvektors.
- Martina verschlüsselt die Nachricht für Frank mit dem neuen Session Key mittels RSA (asymmetrisch), aber diesmal mit Franks Public Key.
- Martina versendet (1.) die verschlüsselten Daten, (2.) den verschlüsselten Session Key und (3.) den Initialisierungsvektor.

Um die von Martina empfangene Nachricht zu lesen, durchläuft Frank folgende Sequenz:

- Entschlüsselung des mit dem RSA-Public-Key verschlüsselten AES-Session-Keys mit dem eigenen RSA-Private-Key.
- Mit dem entschlüsselten AES-Session-Key kann die Nachricht entschlüsselt werden.

Die Implementierung beginnt mit dem „Datenpaket“, das in Bild 11 dargestellt ist und über die folgenden drei Member verfügt:

- einen AES-Key in verschlüsselter Form,
- die Nachricht, die mit dem AES-Key verschlüsselt wurde,
- den Initialisierungsvektor.

Das hybride Verschlüsselungsverfahren wird in einer Klasse gekapselt. In Bild 12 ist die Funktion zu sehen, mit der die ▶

```
Original: DotNetPro ist klasse!

Encrypted: uiX8uGKIUiSrgLqJp6u9WXT3Ny25WskXe7ZGlNOHzb+ccqYfAbrUBZyMiNiTwZ4vLTC5euDnlNReRQVR754iVlaAIJ
+lkz3kWfgTbNTbz/C223bi7iNte4sHFiAhSPrdfhWHoPWFpPCqmnHZCxJw6KK85Kqj4RStBuQ7qZK4AE5pWLUrzgR0bukkE4s0Cuv
aYHol+junI5rV0tI5Eq77Yi4dVpZo4Ccv6EQ1RpCECpzAh1djMniWHBZfSiOE02bjNOSoodSLF1JBgTHaUtmerg9EKtTM/ckCPUaF
h9p16mYaSuCzU9BXMDdK1V+rmnsDjEfdowgq7bh9s0r0SFw==

Decrypted: DotNetPro ist klasse!
```

```
DotNetPro.Cryptographie - Program.cs
ram.cs
tNetPro.Cryptographie
5   const string originalMessage = "DotNetPro ist klasse!";
6   var rsa = new EncryptionRSA();
7
8   var encrypted = rsa.Encrypt(originalMessage);
9   var decrypted = rsa.Decrypt(encrypted);
10
11  Console.WriteLine($"Original: {originalMessage}{Environment.NewLine}");
12  Console.WriteLine($"Encrypted: {Convert.ToBase64String(encrypted)}{Environment.NewLine}");
13  Console.WriteLine($"Decrypted: {Encoding.Default.GetString(decrypted)}{Environment.NewLine}");
```

RSA-Klasse in Verwendung (Bild 9)

```
Original: DotNetPro ist klasse!

Encrypted: g1KCWAqagB+7qA1rTEDmnerDuAfPDcRZoAn8S/tmHmxnNRarWWgwNoC3rLn0oax3tYivaAhqy4oH+dsETAzC
n6P/JisdQ/Vap29tXAuysDcxE3q4caLhI+PoOsOopgKQAstwjZRG8VVr/kDWmBCVHbdfS31tgrfKBlIr7vYcV24Zm7kWa+7
EIahoAF8jRe4jk3amUSgCKJRSNPssl0WTJF59/mc8do4FY/GbmKAey79YdvhgoQXdaR7pUUqGRB7/zoOlZcgPGWUY4U8W6S
s7pN9D0983gFMIALKjAxGb/C/7NjUBTJAxEtITrMqFpVrtqZa4XxE+u4VFDo4okq6C1g==

Decrypted: DotNetPro ist klasse!
```

```
Cryptographie - Program.cs
ptographie
var rsa = new EncryptionRSA();
byte[] encryptedPrivKey = rsa.ExportPrivateKey(iterationCount: 1_000, "mypw");
byte[] publicKey = rsa.ExportPublicKey();

const string originalMessage = "DotNetPro ist klasse!";
var encrypted = rsa.Encrypt(originalMessage);

var rsa2 = new EncryptionRSA();
rsa2.ImportPublicKey(publicKey);
rsa2.ImportEncryptedPrivateKey(encryptedPrivKey, "mypw");

var decrypted = rsa2.Decrypt(encrypted);

Console.WriteLine($"Original: {originalMessage}{Environment.NewLine}");
Console.WriteLine($"Encrypted: {Convert.ToBase64String(encrypted)}{Environment.NewLine}");
Console.WriteLine($"Decrypted: {Encoding.Default.GetString(decrypted)}{Environment.NewLine}");
```

RSA-Klasse mit Import/Export der Schlüssel (Bild 10)

```
internal class EncryptedMessagePacket
{
    /// <summary>
    /// Der verschlüsselte AES Key (symmetrischer Schlüssel).
    /// </summary>
    public byte[] EncryptedAESSessionKey;
    /// <summary>
    /// Die mit dem AES Key verschlüsselten Daten.
    /// </summary>
    public byte[] DataEncryptedWithAESSessionKey;
    /// <summary>
    /// Zufallswert
    /// </summary>
    public byte[] InitializationVector;
}
```

Das Datenpaket mit seinen Membern (Bild 11)

Daten verschlüsselt werden. Die Bestandteile, die die symmetrische Verschlüsselung behandeln, sind rot markiert, während die asymmetrischen Verschlüsselungsbestandteile grün markiert sind.

Zunächst wird der AES-Session-Key erstellt und in das Datenpaket gereicht. Die eigentlichen Daten werden im nächsten Schritt mit diesem AES-Session-Key verschlüsselt und ebenfalls dem Datenpaket übergeben. Final wird der symmetrische Schlüssel asymmetrisch verschlüsselt.

Bild 13 zeigt die Funktion, die der hybriden Entschlüsselung dient. Auch hier wurden die symmetrischen Anweisungen (rot markiert) und die asymmetrischen (grün) Anweisungen farblich gekennzeichnet. Der asymmetrisch verschlüsselte symmetrische Schlüssel wird zunächst entschlüsselt. Anschließend wird mit diesem die symmetrisch verschlüsselte Nachricht entschlüsselt.

Bild 14 dokumentiert die Anwendung. Im Konstruktor der *EncryptionRSA*-Klasse wird das für die asymmetrische Verschlüsselung benötige Objekt erstellt, siehe Listing 1. Als Nächstes wird dieses Objekt gemeinsam mit der zu verschlüsselnden Nachricht hybrid verschlüsselt und im Anschluss wieder entschlüsselt.

```
public static EncryptedMessagePacket Encrypt(byte[] dataToEncrypt, EncryptionRSA rsaParams)
{
    var aesSessionKey = RandomNumberGenerator.GetBytes(32);
    var messagePacket = new EncryptedMessagePacket
    {
        InitializationVector = RandomNumberGenerator.GetBytes(16)
    };

    messagePacket.DataEncryptedWithAESSessionKey =
        EncryptionAes.Encrypt(dataToEncrypt, aesSessionKey, messagePacket.InitializationVector);

    messagePacket.EncryptedAESSessionKey = rsaParams.Encrypt(aesSessionKey);
    return messagePacket;
}
```

Hybride Verschlüsselung (Bild 12)

```
public static byte[] DecryptData(EncryptedMessagePacket encryptedMessagePacket, EncryptionRSA rsaParams)
{
    var aesSessionKey = rsaParams.Decrypt(encryptedMessagePacket.EncryptedAESSessionKey);

    var decryptedData = EncryptionAes.Decrypt(encryptedMessagePacket.DataEncryptedWithAESSessionKey,
        aesSessionKey, encryptedMessagePacket.InitializationVector);

    return decryptedData;
}
```

Hybride Entschlüsselung (Bild 13)

```
const string originalMessage = "DotNetPro ist klasse!";

var rsaParams = new EncryptionRSA();

var encryptedBlock = EncryptionHybrid.Encrypt(Encoding.UTF8.GetBytes(originalMessage), rsaParams);
var decrypted = EncryptionHybrid.DecryptData(encryptedBlock, rsaParams);

Console.WriteLine($"Original: {originalMessage}");
Console.WriteLine($"Encrypted: {Encoding.UTF8.GetString(decrypted)}");
```

Verwendung der hybriden Verschlüsselung (Bild 14)

Hashing

Bei dem Thema Kryptografie kommt man unweigerlich mit dem Begriff „Hashing“ in Berührung. Bei einem Hash handelt es sich um einen digitalen Fingerabdruck von Daten. Wird für eine Textnachricht der Hash berechnet, wird eine scheinbar zufällig zusammengewürfelte Zeichenfolge generiert. Das Hashing hat bestimmte Merkmale:

- Die Berechnung eines Hash ist recht einfach.
- Es ist nicht möglich, eine Nachricht mit einem bestimmten Hash-Wert zu berechnen.
- Eine Änderung der Nachricht, ohne den Hash zu verändern, ist nicht möglich.
- Es ist nicht möglich, unterschiedliche Nachrichten mit dem gleichen Hash zu erhalten.
- Hashing ist somit eine „One way operation“. Hingegen ist die Verschlüsselung eine „Two way operation“.

Wird vor dem Versand einer Nachricht deren Hash berechnet und mit der Nachricht an den Empfänger geliefert, kann der Empfänger selbst den Hash der empfangenen Nachricht berechnen und mit dem übertragenen Hash vergleichen. Sind beide Hash-Werte identisch, kann davon ausgegangen werden, dass die Nachricht unverändert empfangen wurde.

Bild 15 zeigt, wie von zwei nahezu identischen Nachrichten der Hash berechnet wird. Der Unterschied ist das letzte Zeichen, auf welches der rote Pfeil hinweist. Zu sehen ist, dass die berechneten Hash-Werte absolut unterschiedlich sind.

```
Message1: v5JTLrQcmj+i4Mn7MzeYYlx7J0TbGi/8dG+iYVfIkRMqX7l4AKT0XxA2rNWG60YkTos2sZ4+nTXaQwJU0LHxYw==
Message2: pDoPamQjUvgA3YHLIcOo6F1PLdAgRDNDbLnObD/opxt2lwRFwpw8QEDYGQd1fXfxs259vU0CUcsdC4vRdQI19Q==
NetPro.Cryptographie - Program.cs
Pro.Cryptographie
1   using DotNetPro.Cryptographie;
2   using System.Text;
3
4   const string message1 = "DotNetPro ist klasse!";
5   const string message2 = "DotNetPro ist klasse1";
6
7   var sha512Message1 = Hash.HashSha512(Encoding.UTF8.GetBytes(message1));
8   var sha512Message2 = Hash.HashSha512(Encoding.UTF8.GetBytes(message2));
9
10  Console.WriteLine($"Message1: {Convert.ToBase64String(sha512Message1)}");
11  Console.WriteLine($"Message2: {Convert.ToBase64String(sha512Message2)}");
```

Hash-Berechnung für zwei sehr ähnliche Nachrichten (Bild 15)

HMAC – Hashed Message Authentication Code

Die Kombination aus einem „One way“-Hash und einem kryptografischen Schlüssel wird als HMAC bezeichnet. Der HMAC lässt sich ebenfalls wie ein Hashcode für die Prüfung der Nachrichtenintegrität verwenden. Der HMAC kann zudem zur Authentifizierung der Nachricht verwendet werden – nur wer im Besitz des kryptografischen Schlüssels ist, kann den HMAC berechnen.

Listing 1: Konstruktor der RSA-Kapselung

```
private readonly RSA _rsa;
public EncryptionRSA()
{
  const int keySize = 2048;
  rsa = RSA.Create(keySize);
}
```

```
internal class EncryptedMessagePacket
{
    /// <summary> Der verschlüsselte AES Key (symmetrischer Schlüssel).
    public byte[] EncryptedAESSessionKey;
    /// <summary> Die mit dem AES Key verschlüsselten Daten.
    public byte[] DataEncryptedWithAESSessionKey;
    /// <summary> Zufallswert
    public byte[] InitializationVector;
    /// <summary>
    /// Authentifizierte Hash der verschlüsselten Daten
    /// </summary>
    public byte[] Hmac;
}
```

Für den HMAC erweitertes Datenpaket (Bild 16)

Listing 2 zeigt, wie die Berechnung eines HMAC in einer Funktion gekapselt wird. Übergeben werden die Daten, für die der Hash berechnet wird, und der kryptografische Schlüssel. Die Verwendung der zugehörigen .NET-Klasse ist trivial.

Auf das Beispiel von Frank und Martina bezogen wäre noch ein zusätzlicher Schritt beim Versand von Frank zu Martina nötig: Der HMAC der verschlüsselten Daten unter Verwendung des AES-Session-Keys müsste berechnet werden.

Zunächst wird hierzu das Datenpaket, wie es in Bild 16 zu sehen, ist erweitert. Die hybride Verschlüsselung ist in der Klasse *EncryptionHybrid* gekapselt. Sie ähnelt dem bisher gesehenen Code, jedoch ergänzt um die Erweiterung zur Berechnung des HMAC (Bild 17). Der HMAC berechnet den Wert aus den symmetrisch verschlüsselten Daten und dem Initialisierungsvektor. Beide werden in der Funktion *Combine* zusammengeführt.

```
internal class EncryptionHybrid
{
    1 reference
    public static EncryptedMessagePacket Encrypt(byte[] dataToEncrypt, EncryptionRSA rsaParams)
    {
        var aesSessionKey = RandomNumberGenerator.GetBytes(32);
        var messagePacket = new EncryptedMessagePacket
        {
            InitializationVector = RandomNumberGenerator.GetBytes(16)
        };

        messagePacket.DataEncryptedWithAESSessionKey =
            EncryptionAes.Encrypt(dataToEncrypt, aesSessionKey, messagePacket.InitializationVector);

        messagePacket.EncryptedAESSessionKey = rsaParams.Encrypt(aesSessionKey);

        using var hmac = new HMACSHA256(aesSessionKey);
        messagePacket.Hmac = hmac.ComputeHash(
            Combine(messagePacket.DataEncryptedWithAESSessionKey, messagePacket.InitializationVector));

        return messagePacket;
    }
    messagePacket {DotNetPro.Cryptographie.EncryptedMessagePacket}
      DataEncryptedWithAESSessionKey  View {byte[32]}
      EncryptedAESSessionKey  View {byte[256]}
      Hmac  View {byte[32]}
      InitializationVector  View {byte[16]}
    2 references
    private static by
    1 reference
```

Berechnung des HMAC beim Verschlüsseln (Bild 17)

Bei der Entschlüsselung kann nun der HMAC wie bereits zuvor aus den symmetrisch verschlüsselten Daten und dem Initialisierungsvektor berechnet werden und mit dem HMAC der empfangenen Daten verglichen werden (Bild 18).

```
public static byte[] DecryptData(EncryptedMessagePacket encryptedMessagePacket, EncryptionRSA rsaParams)
{
    var aesSessionKey = rsaParams.Decrypt(encryptedMessagePacket.EncryptedAESSessionKey);
    using var hmac = new HMACSHA256(aesSessionKey);
    var hmacToCheck = hmac.ComputeHash(
        Combine(encryptedMessagePacket.DataEncryptedWithAESSessionKey, encryptedMessagePacket.InitializationVector));

    if (!Compare(encryptedMessagePacket.Hmac, hmacToCheck))
    {
        throw new CryptographicException("HMAC check failed");
    }
```

Berechnung des HMAC beim Entschlüsseln (Bild 18)

Fazit

Die Kryptografie ist ein wichtiges Thema. Dieser Beitrag hat hoffentlich zum Verständnis einiger Begriffe aus dieser Domäne beigetragen. ■

[1] Microsoft Learn, System Random Constructors, www.dotnetpro.de/SL2411Krypto1

[2] Microsoft Learn, RandomNumberGenerator, www.dotnetpro.de/SL2411Krypto2

[3] AES bei Wikipedia, www.dotnetpro.de/SL2411Krypto3

[4] DES bei Wikipedia, www.dotnetpro.de/SL2411Krypto4

Listing 2: HMAC berechnen

```
public static byte[] HmacSha512(byte[] toGetHash,
byte[] key)
{
  using var hmac = new HMACSHA512(key);
  return hmac.ComputeHash(toGetHash);
}
```

Christian Havel
wohnt in einem Vorort von München und ist seit mehr als zwanzig Jahren in der Softwareentwicklung tätig. Er programmiert hauptsächlich in C#. Sie erreichen ihn unter **christian.havel@googlemail.com**.

dnpCode A2411Krypto