

APPLICATION SECURITY IN .NET

SUCCINCTLY

BY **STAN
DRAPKIN**

Application Security in .NET Succinctly

By
Stan Drapkin

Foreword by Daniel Jebaraj



Copyright © 2017 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Stephen Haunts

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	8
About the Author	10
Preface	11
Chapter 1 .NET Security	12
Cryptographic primitives	12
FIPS mode	12
Random number generators	13
System.Random	13
RNGCryptoServiceProvider	15
Random integers in min-max range	17
Chapter 2 Hashes and MACs	19
Object.GetHashCode	19
FNV-1a hash	20
Hash flooding	21
FNV-1a with entropy	22
SHA-1 hash	23
SipHash	23
Modern non-cryptographic hashes	24
Cryptographic hashes	24
HMAC (including hash/HMAC factories)	25
HMAC in .NET	25
Chapter 3 Key Derivation	29
PBKDF2	29
HKDF and SP800_108	31

PBKDF2 combined with HKDF	32
Salt	33
Client-side KDF	33
Key separation	34
Chapter 4 Comparing Byte Arrays	35
Direct comparison (insecure)	35
AND comparison	35
XOR comparison	36
Double-HMAC comparison	37
Chapter 5 Binary Encodings	38
Base64	38
Base32	39
Base16	41
Chapter 6 Text Encodings	42
UTF-32	42
UTF-16	42
UTF-8	43
UTF-7	43
UTF comparison	43
Safe construction	44
Serialization	45
Chapter 7 Symmetric Encryption	47
AES	48
Key	49
Cipher mode	49
Padding mode	49
Initialization vector (IV)	50

AES in counter mode (CTR)	50
Chapter 8 Authenticated Encryption.....	52
EtM Key derivation	53
Primitive choices	53
Length leaks	54
Common mistakes	54
Chapter 9 Asymmetric Cryptography	55
RSA key management	56
RSA signatures	58
RSA key exchange	59
RSA encryption	61
Diffie-Hellman key agreement	64
Perfect forward secrecy	65
Elliptic Curve Integrated Encryption Scheme (ECIES)	65
Key separation	66
Key wrap	67
Chapter 10 Two-Factor Authentication (2FA).....	68
HOTP	68
TOTP.....	68
U2F	69
Chapter 11 Web Security.....	70
ASP.NET security	70
Session state	71
CSRF	75
Forms authentication	80
Membership	83
Insider threats	84

Credential storage.....	85
Improving forms authentication.....	87
New ASP.NET crypto stack	90
ASP.NET CSRF API	91
Client-side PBKDF	94
Password reset	94
Strict Transport Security (STS)	96
X-Frame-Options (XFO).....	97
Content-Security-Policy (CSP)	98
Subresource Integrity (SRI)	99
JSON Object Signing and Encryption (JOSE) framework	100
Cookies	101
HTTP/2.....	102

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Stan Drapkin ([@sdrapkin](#), [GitHub](#)) is a hands-on technical leader, manager, and security and compliance expert. He is focused on hiring, training, and leading top-talent engineering teams using .NET to build highly secure cloud-based solutions for strongly regulated environments.

Stan has computer science and MBA degrees, as well as more than 16 years of .NET Framework experience. He picked up .NET 1.0-beta in 2001, while working as an engineer at Microsoft, and has been immersed in .NET evolution ever since.

Preface

Who is this book for?

This book is intended for architects, developers, and other information technology professionals who design and build security components and layers of .NET solutions.

Why is this book relevant today?

Three main problems that plague developers writing security-focused .NET code today are:

- Many security developers are not as familiar or comfortable with the material in this book as they ideally should be, regardless of seniority or job title. This book tries to fill some of the common knowledge gaps, or at least provide a good refresher of .NET security practices.
- What many security developers believe to be the right thing to do can sometimes be wrong, insecure, or both. This book highlights some of these cases, and presents them as a learning opportunity to the reader.
- Even the most knowledgeable and aware security developers are often unable to produce quality .NET implementations that are sane, simple, reusable, and secure. This book addresses many widespread security knowledge gaps and provides practical guidance and high-quality .NET implementations of cryptographic primitives via the [Inferno](#) crypto library.

[Inferno](#) is a modern, open-sourced cryptographic library for .NET, available on [GitHub](#) and [NuGet](#). Inferno does .NET crypto right, and offers a misuse-resistant API with safe algorithms, modes, and choices. We encourage you to read Inferno's design goals.

What makes this book different?

- Original and relevant content, which has not been covered properly in other books.
- Succinctness. There are many excellent books and other learning materials dedicated to introductory and foundational topics. Detailed coverage of many concepts not directly in-scope is not provided, but links are to Wikipedia or other relevant sources are included when such peripheral concepts appear.
- The only math required is arithmetic (we promise).
- It's based on real-world experience implementing .NET security. Sometimes we bend the rules or try an unusual technique, but we pick our battles and compromises carefully.

Chapter 1 .NET Security

Cryptographic primitives

The .NET Framework security foundation is made of various cryptographic building blocks, which, in turn, rely on various cryptographic primitives. These primitives are supposed to be well-defined and well-understood cryptographic algorithms and services that follow the single responsibility principle, meaning they should do one thing only, and do it well. Cryptographic primitives are typically combined to form more complex cryptographic building blocks, which can be further combined to form security protocols. Not all primitives provided by Microsoft are well-defined or well-understood, but before we start improving them, we should be clear about basic security rules.

The first rule of sane security architecture is never design your own cryptographic primitives or protocols. The second rule of sane security architecture is never implement cryptographic primitives or protocols—even when you follow an established design, since your implementation will be flawed. Implementation flaws are a much more common cause of practical vulnerabilities than actual cryptographic weaknesses. There are no exceptions to the first rule—it is absolute. There are some exceptions to the second rule, however. One acceptable reason to break the second rule (outside of learning purposes) is to deal with situations when reliable implementations are either not available or are sorely lacking.

You will be breaking the second rule in upcoming chapters, but first you need an understanding of what primitives are available in .NET out of the box, and the strengths and deficiencies of their implementations. So let's get on with it.

FIPS mode

[Federal Information Processing Standards \(FIPS\) 140](#) are U.S. government standards for accreditations of cryptographic modules. The Windows operating system already comes with certain FIPS-validated cryptographic modules. The OS-level FIPS mode itself is advisory only; it does not take any proactive steps to enforce application compliance and prevent non-FIPS-validated modules. The .NET-level FIPS mode, however, will prevent any non-validated cryptographic implementations that .NET itself is aware of (the ones that are part of the .NET Framework) from running by throwing an **InvalidOperationException**. Various regulations might require FIPS mode to be enabled in your .NET environment, so designing your solutions to be FIPS-mode-compliant is a good idea.

The .NET Framework comes with several cryptographic API types:

- 100 percent managed implementations, typically ending with **Managed**.
- APIs that call external, [unmanaged implementations](#) of the underlying Windows OS:
 - The “old” [CryptoAPI](#) providers, typically ending with **CryptoServiceProvider**.

- The “new” [crypto-next-generation](#) (CNG) APIs, typically starting or ending with **CNG**.

The managed APIs **are not** FIPS-validated (the first type), while both the “old” and “new” unmanaged APIs (the second type) **are** FIPS-validated. The main reason to use the managed APIs is start-up speed—they are much faster to construct than their unmanaged equivalents. The downside of managed APIs is their raw processing speed—they are slower at processing than native alternatives. By Microsoft’s own guidance, the CNG APIs are preferred over **CryptoServiceProvider** APIs. The “what do I use?” decision for modern 64-bit .NET deployments is thus very simple:



Tip: Use CNG APIs.

In some scenarios that only process very few bytes of data (16–20 bytes or fewer), managed APIs might be slightly faster than CNG APIs due to faster start-up. Some older Windows platforms do not support CNG APIs. This should be less of a problem now, however, since all these “legacy” Windows platforms are no longer supported by Microsoft (for example, Windows XP or Windows Server 2003). We will see other reasons to choose CNG over **CryptoServiceProvider** APIs when we discuss HMAC.

Random number generators

The .NET Framework provides two random number generators: the **System.Random** class (we’ll call it **RND**) and the **System.Security.Cryptography.RNGCryptoServiceProvider** class (we’ll call it **CSP**), both of which reside in **mscorlib.dll**.

System.Random

RND’s purpose is generation of pseudo-random **int** and **double** types in a defined range. According to MSDN, RND implementation follows Knuth’s [subtractive generator](#) algorithm from Chapter 3 of [The Art of Computer Programming](#). Knuth’s algorithm keeps a circular buffer of 56 random integers, which all get initialized with a deterministic permutation based on the provided seed value. Two indices into this buffer are kept 31 positions apart. A new pseudo-random number is calculated as the difference of the values at these two indices, which is also stored in the buffer. Unfortunately, Microsoft’s implementation of Knuth’s algorithm has a bug that keeps the indices 21 positions apart. Microsoft has [acknowledged](#) this issue, but refused to fix it, citing “backward compatibility” concerns. RND is a seeded deterministic generator, which means it provides the same pseudo-random sequence for a given seed. Fixing the bug would cause the algorithm to produce a different sequence for any given seed, and Microsoft apparently decided that not breaking applications that relied on specific pseudo-random sequences (contrary to MSDN warning that RND implementation is subject to change) was more important than getting the algorithm right.

This oversight, however, does not pose a practical problem for the vast majority of RND uses for the simple reason that RND has far more serious issues that are likely to cause you grief a lot sooner than exhausting the intended or actual sequence period. [Mono](#) (an alternative CLR implementation) got the algorithm right, by the way (which also implies that Mono-generated sequences of **Random** are not compatible with Microsoft ones).

RND could have been a decent choice for many scenarios requiring randomness, since high-quality randomness is often **not** a must-have requirement. “Quality” of random number generators refers to their ability to pass a battery of statistical tests designed to detect non-randomness (for example, [NIST](#) and [diehard](#) tests).

Unfortunately, a lot of these common scenarios are multi-threaded (for example, IIS/ASP.NET threads, or developer-created parallelism to utilize all available CPUs). One way to make RND thread-safe is to derive from it, overload all virtual methods, and wrap base calls in a “lock” block ([Monitor](#)). This adds thread safety but does not solve the API flaws, which can be illustrated by the following:

```
int A = new System.Random().Next();
int B = new System.Random().Next();
```

What do you know about **A** and **B**? We know that they are equal. The problem here is that RND must be instantiated with an integer seed, in the absence of which (i.e. the default constructor) RND uses [Environment.TickCount](#), which has millisecond precision. Basic operations, however, take nanoseconds, and both RND objects get seeded with the same **TickCount** value. MSDN [documents](#) this behavior, and recommends to “*apply an algorithm to differentiate the seed value in each invocation*,” but offers no clues on how to actually do it.

One approach is to have a static integer value which is initialized to **Environment.TickCount** inside a static constructor (which is thread-safe), and then is read for seeding (inside instance constructor) with **var seed = System.Threading.Interlocked.Increment(ref _staticSeed)**. The seed will wrap around when it reaches the max integer value. The downside of this approach is that it requires keeping state, which is often not convenient.

Another approach is to use **var seed = Guid.NewGuid().GetHashCode()**, which is fast and thread-safe, and produces integers that are reasonably well-distributed in **int** space. The upside of this approach is that it is stateless. The downside is that different GUIDs can produce the same hashcode, which will cause identical seeds and identical random sequences.

MSDN makes another suggestion: “*call the [Thread.Sleep](#) method to ensure that you provide each constructor with a different seed value.*” This is a bad idea—do not do this. There are very few reasons to ever call **Thread.Sleep**, and this is not one of them.

Even if you manage to initialize **System.Random** with a consistently unique seed, you must still deal with the fact that this class is not thread safe. If you accidentally use its methods from multiple threads, the internal state will get silently corrupted and you will start seeing a not-so-random sequence of all zeroes or negative ones.

A third common approach to making **System.Random** safe is to never create more than one instance of it, and invoke all **System.Random** methods on that single instance inside a lock. You can see one example [here](#). One downside of this approach is that a single random sequence becomes the provider of randomness for all components that draw upon it, which makes it easier to reach the sequence period. While this might not pose a problem in many scenarios, it is important to be aware of. Another downside is that all that thread-safe **lock/Monitor**

synchronization not only takes about 20 nanoseconds on every call, but also makes the generator inherently non-parallel.

A fourth common approach is to use the `[ThreadStatic]` attribute or `ThreadLocal<Random>` class to create a separate instance of `System.Random` per thread ([example](#)). This is a very popular approach. It is also dangerous because it gives a false sense of addressing the core issues. It is simply a different take on how to deal with parallel access: you either share a single instance among many threads (which requires locking), or you create a separate instance per thread. The `System.Random` construction and the seeding issues did not go away.

`ThreadLocal<Random>` makes construction explicit—for example, `new ThreadLocal<Random>(() => new Random(/* what seed do we use here ?? */))`—while the `[ThreadStatic]` attribute moves the construction logic to a method or property getter that you use to retrieve the instance. In either case, none of the seeding problems are resolved.

The RND algorithm's randomness properties are supposed to hold within a specific, reasonably long cycle period. The instance-per-thread approach completely disregards these mathematical properties because instead of consuming the RND sequence itself (as designed), this approach cycles the seed instead, and consumes only some short number of cycles after the initial seeding. Such misuse affects the pseudo-randomness quality that RND was mathematically designed to provide, since the seed is 2^{32} , but the period is supposed to be 2^{56} .

Another common pitfall is that the `ThreadStatic/ThreadLocal` instance is usually exposed as an RND-typed property, which makes it easy to use in all places that expect an RND-typed instance. However, it seems reasonable for a developer to write the following code: `static Random safeRnd = SafeRandom.Rnd;` and then use `safeRnd` in parallel code. The problem, of course, is that the property returns a thread-safe value for the *currently executing* thread, so saving the returned value in a static variable would save a specific instance from a specific thread. When the same static value is accessed from another thread, multiple threads start trampling all over each other because RND is not thread-safe. The core flaw here is that this approach is not foolproof enough for developers who might not be familiar with internal implementation nuances.

Let's pause and reflect for a moment. You just want to generate some random numbers using the documented APIs—how hard should it be? Why do you have to be aware of all these workarounds and complicated approaches which are all bad at the end of the day?

`System.Random` is a quagmire of bad choices. It is a pit of failure, regardless of how you try to engineer your way out of it. Our advice is to avoid it entirely. Let's go over CSP next, which is not ideal either.



Tip: Avoid using `System.Random` if you can—there are better alternatives. Be aware of the pitfalls if you cannot (for example, if you require deterministic pseudo randomness).

RNGCryptoServiceProvider

CSP's purpose is the generation of [cryptographically-secure](#) random *bits*. Unlike RND, CSP is not specifically designed for random *number* generation, and thus RND and CSP have very different APIs.

It's tempting to assume that random bits can be trivially combined to generate random numbers. This is certainly true in the case of *integral* types: random **byte** is 8 random bits, random **int** is 4 random bytes, and random **long** is 8 random bytes. *Floating-point* types, however, are not that simple to randomize. For example, **double** in .NET is an 8-byte, [IEEE-754](#) floating-point structure, which essentially means that its precision varies with magnitude (precision goes down when magnitude goes up). Simply generating 8 random bytes and casting them into **double** will generate a non-uniform random distribution over a given min-max range. The proper way to achieve range uniformity with doubles is to randomize the significand separately for some pre-set fixed exponent. However, that would require control over bit-construction of **double**, which is not trivial in .NET.

RND.NextDouble() returns a **double** in [0,1) range. Its implementation “cheats” (in order to ensure uniformity of distribution) by dividing an internal 4-byte **int** state by **int.MaxValue+1** using floating-point division (+1 ensures that the upper bound is exclusive). RND's internal sample size is 4 bytes, and it should be apparent that 64 bits of randomness (which you might expect for a **double**) cannot be manufactured from 4 bytes of internal sample. This approximation, however, suffices for the vast majority of uses that RND was intended for. We could use the same trick to make doubles out of CSP integers. Let's compare RND and CSP attributes that matter.

Table 1: RND and CSP attributes of interest

	Thread safety	Performance	Quality	Requires seeding
RND	No	Very fast	Poor	Yes (int)
CSP	Yes	Depends	High quality	No

CSP certainly looks very appealing given its thread safety, lack of seeding, cryptographically strong quality, and ease of implementing RND APIs on top of CSP. Performance of CSP is probably its most misunderstood aspect. CSP often gets accused of poor performance, especially compared to RND.

The truth is that CSP can be very fast—much faster than RND—when used appropriately. CSP has only one useful method, **GetBytes()**, which populates a byte array (the other method, **GetNonZeroBytes()**, works the same way). This method has a substantial invocation cost; that is, it's a lot more expensive to call it 100 times to populate a 4-byte array than it is to call it once to populate a 400-byte array. If you measure performance beyond the method invocation cost (which is much more expensive for CSP than RND methods), you will realize that the random-byte-generation performance itself is much better for CSP than RND. (We ran our tests against x64 .NET 4.5 runtime on Intel Core-i7 running Windows 2008R2.) For example, populating a 4-kilobyte array is twice as fast with CSP as it is with RND.

CSP and RND have a performance inflection point (in terms of the size of the passed-in byte array) when CSP performance becomes equal to RND performance, and then surpasses it for all larger array sizes. That point is a 245-byte array on our machine. For small byte arrays in the

1–200 range, CSP performance is significantly worse than RND. For example, a 4-byte array (a random integer) generation could be about 40 times slower with CSP.

Fortunately, we can substantially improve the small-range performance of CSP through buffering. The **CryptoRandom** class of the [Inferno](#) library implements the familiar RND [interface](#), but uses buffered CSP. **CryptoRandom** is only two to three times slower than RND in the worst case, which should be quite acceptable for most small-range scenarios. .NET 4.5 itself uses the buffered-CSP approach internally (for example, the internal **System.Collections.HashHelpers.GetEntropy()** method, which uses CSP to populate a 1024-byte array and then consume it in 8-byte chunks).

While the RND approach is a pit of failure, the buffered-CSP/**CryptoRandom** approach is a mountain of success because it offers an acceptable-to-excellent performance range, high-quality cryptographic randomness, and frictionless thread safety, which makes it virtually impossible to use **CryptoRandom** incorrectly. RND can still be useful in certain scenarios that require deterministic reproduction of low-quality random sequences.

CSP implementation itself is a wrapper around the [CryptGenRandom](#) Windows API. **CryptGenRandom** is famous enough to have its own [Wikipedia](#) entry, but it's not up to date. Starting with Windows Server 2008/Vista SP1, **CryptGenRandom** uses a well-known, approved, secure algorithm by default (CTR_DRBG from [NIST SP 800-90A](#)).

CSP can be instantiated in several ways:

Table 2: CSP instantiation options

Instantiation code	Performance
<code>var csp = new RNGCryptoServiceProvider();</code>	1x
<code>var csp = RNGCryptoServiceProvider.Create();</code>	~20 times slower

Using the default CSP constructor for instantiation and not the factory method gives much better performance. It is also worth noting that CSP is a **Disposable** instance, so if you plan on allocating lots of them (although we cannot think of a good reason), you might want to clean up proactively (for example, **using** block) rather than leaving the cleanup to the garbage collector.

Random integers in min-max range

Randomly-generated integers often need to be bound within a specific min-max range. RND has the `int Next(int minValue, int maxValue)` method, which you might find yourself re-implementing (for an RND-derived class like **CryptoRandom**, for example). Another common requirement is the `long NextLong(long minValue, long maxValue)` method, which extends the min-max range to 64-bit integers (18 full decimal digits).

The naïve implementation approach is to generate a full-range number, mod it by the range length, and add it to the `minValue`. This will produce a skewed random distribution because the full range will not be evenly divisible into range-length chunks. Be aware of this common pitfall (one correct implementation can be found in the [CryptoRandom](#) class).

Generating a random `long` with CSP or RND is easy—you just generate eight random bytes and convert them to `long` (for example, using the `BitConverter.ToInt64()` method). If for some reason you want to generate a `long` with the `RND.Next()` method only, you need to do more work. `Next()` returns positive integers only, which means that it can only provide 31 bits of entropy. You require 64 bits for a `long`, however, so you need to call `Next()` three times to obtain intermediate `int1`, `int2`, and `int3` values. You need 64/3, so about 22 random bits from each intermediate `int`. There have been reports that the RND algorithm has a bias in the least significant bit, so you should prefer its high-order bits over its low-order bits. The C# implementation could be:

Code Listing 1

```
public static long NextLong(Random rnd)
{
    long r = rnd.Next() >> 9; // 31 bits - 22 bits = 9
    r <<= 22;
    r |= rnd.Next() >> 9;
    r <<= 22;
    r |= rnd.Next() >> 9;
    return r;
}
```

Chapter 2 Hashes and MACs

Hash functions (hashes) map values from domain A to values from domain B. Domain A is typically a large, variable-length domain, while domain B is typically a much smaller, fixed-length domain. B can often be a subdomain of A (for example, a “string-to-string” mapping). Hash functions have various important properties which you need to be aware of to be able to judge the suitability of any given hash for a particular purpose.

- **Determinism:** A deterministic hash function must consistently return the same domain B value given the same domain A value. Determinism is a property that you almost always want, with the exception of few special cases that we will cover later.
- **Uniformity:** A hash function with good uniformity characteristics is expected to produce domain B values that are uniformly spread among all possible domain B values; any domain B value should have an equal probability of being returned.
- **Perfection:** A perfect (injective) hash function maps each domain A value to a *different* domain B value. Perfection quality minimizes the number of domain B collisions.

Cryptographic hash functions have the following important properties:

- **First-preimage resistance:** Given any domain B value, it is computationally infeasible to find a domain A value that produced it. ([Preimage](#) is a simple mathematical concept.)
- **Second-preimage resistance:** Given any domain A value, it is computationally infeasible to find another domain A value that produces the same domain B value.
- **Collision resistance:** It is computationally infeasible to find any two domain A values that map to the same domain B value. Collision resistance implies second-preimage resistance, but does not imply first-preimage resistance.

Object.GetHashCode

Most .NET developers have to choose and implement a hash function when overriding the [Object.GetHashCode\(\)](#) method. The only requirement for `GetHashCode()` is determinism, with “sameness” of domain A values handled by the [Object.Equals\(\)](#) method. Uniformity is not required, but is recommended to improve performance (and, as we will see later, security). Most .NET Framework types have `GetHashCode()` implementations that are fast and reasonably uniform for general use. Many trivial input scenarios, however, can cause a serious uniformity degradation of built-in .NET `GetHashCode()` implementations. For example, generating 10 million GUID strings with `Guid.NewGuid().ToString()` leads to about 11.5k `GetHashCode()` collisions (0.1%). Doing the same test for the GUID string concatenated with itself leads to about 750k `GetHashCode()` collisions (7.5%)—an unexpected 75x increase in the number of collisions. One MSDN blog called it a “[curious property](#).”

You might want to use a different `GetHashCode()` implementation in order to avoid unexpected surprises of default implementations. Most .NET containers that rely heavily on uniqueness characteristics of `GetHashCode()` can also take an [IEqualityComparer<T>](#) instance as part of their constructor (`Dictionary`, `HashSet`, `SortedSet`, `KeyedCollection`, `ConcurrentDictionary`, and others, as well as most LINQ operations). Providing a custom implementation of `IEqualityComparer<T>` is your chance to override the `Equals()` and `GetHashCode()` methods with your custom logic. The next important consideration is what to override `GetHashCode()` with, so let's cover a few good options you should be aware of.

FNV-1a hash

[FNV-1a](#) is a widely used non-cryptographic hash function that is fast, simple to implement, and exhibits low collision rates as well as good [avalanche effect](#) behavior (after a small mod). Here is a 32-bit FNV-1a implementation with an additional post-processing step for better avalanche behavior (based on tests by [Bret Mulvey](#)):

Code Listing 2

```
public static int GetHashCodeFNV1a(byte[] bytes)
{
    const uint fnv32Offset = 2166136261;
    const uint fnv32Prime = 16777619;

    unchecked
    {
        uint hash = fnv32Offset;
        for (var i = 0; i < bytes.Length; ++i)
            hash = (uint)(hash ^ bytes[i]) * fnv32Prime;

        hash += hash << 13;
        hash ^= hash >> 7;
        hash += hash << 3;
        hash ^= hash >> 17;
        hash += hash << 5;
    }
}
```

```
        return (int)hash;
    }
}
```

Hash flooding

Hash flooding attacks are a form of [algorithmic complexity attack](#). Hash functions are commonly used in hash table implementations, which have $O(1)$ lookup and insertion on average but $O(n)$ in the worst case, which is achieved when all inputs hash to the same value. An insert or lookup operation over n inputs can therefore degenerate into an $O(n^2)$ situation if worst-case behavior could somehow be triggered. Any hash function that has poor second-preimage resistance is thus vulnerable to hash flooding.

Algorithmic complexity attacks and, specifically, hash flooding attacks are old attacks that have lately come back with a vengeance due to widespread use of vulnerable default hash function implementations in many popular development frameworks (Python, Ruby, .NET, Java, etc.)—low-hanging fruit for [denial-of-service](#) attacks. Neither the FNV-1a hash, nor the default .NET hashes are second-preimage resistant, meaning they are all vulnerable to hash flooding.

Two defenses against hash flooding attacks are typically employed: entropy injection and switching to a second-preimage-resistant hash function.

Entropy injection is injecting some variability or randomness into an existing non-cryptographic hash function algorithm to make it more difficult to exploit the weak second-preimage resistance of the algorithm itself, as well as to prevent scalability of an attack.

Entropy injection is very similar in concept to the use of [salt](#). It merely prevents attack scalability, and, perhaps, buys you some time, but does not address the weakness itself. The proper defense is switching to a second-preimage-resistant hash function, which can also be complemented with entropy injection.

The biggest challenge with using a proper second-preimage-resistant hash function for general-purpose 32-bit hashing is that most of the available well-known functions are a lot slower than the fast non-cryptographic hash functions. Part of the reason is that cryptographically strong hash functions are designed to provide first-preimage, second-preimage, and collision resistance, while we only care about second-preimage resistance for the purposes of hash flood protection. Slow performance of cryptographic hash functions (compared to the non-cryptographic ones) is likely the main reason for the default hash function choices of most popular development frameworks.

Microsoft has added a new .NET 4.5 configuration element, [UseRandomizedStringHashAlgorithm](#), which changes the behavior of `string.GetHashCode()`. When the `UseRandomizedStringHashAlgorithm` element is enabled, `string.GetHashCode()` calls into a new, private, external (unmanaged) method “`int InternalMarvin32HashString(string s, int sLen, long additionalEntropy)`” and returns whatever that new method returns. Microsoft offers no insight whatsoever about design

or implementation (or even existence) of this method. We can only hope that this information void is temporary (intended, perhaps, to buy Microsoft some time to produce an implementation that might actually survive peer review). As it stands, though, “Marvin32” has to be treated as security by obscurity.

At this point, however, it is very unlikely that the Marvin32 algorithm makes the already weak default .NET string-hashing logic any weaker than it already is. In fact, it is apparent—both from MSDN documentation of the new config element and from the Marvin32 signature and the invoking code calling it—that it uses AppDomain-specific entropy injection (a 64-bit static CSP-generated value), which can only be a good thing.

Other changes due to **UseRandomizedStringHashAlgorithm** include internal switching of **Dictionary**, **HashSet**, and **Hashtable** comparers to new, internal, entropy-injected **RandomizedObjectEqualityComparer** or **RandomizedStringEqualityComparer** (which internally call Marvin32) when the number of loop iterations during element insertion exceeds 100. This defensive switch will only be triggered if the starting comparer is a well-known (default) comparer; custom-provided comparers do not trigger it.

While we believe that enabling **UseRandomizedStringHashAlgorithm** is still a good idea (given the alternative weak default logic), these new Marvin32-related APIs are built as an internal .NET defense, which you cannot control or consume directly. The only direct access is via **string.GetHashCode()**, which will return new, Marvin32-based hash values when enabled in the config. This defense is turned off when you have to provide your own custom comparer, however. There are many other common scenarios when you need proper control over the hash code generating logic, none of which can benefit from the supposed benefits of Microsoft’s new Marvin32 hash function.

FNV-1a with entropy

You might want to inject some entropy into FNV-1a to make it a bit harder to exploit via hash flooding attacks. One simple way to do that is to replace the retrieval of a single byte of input within the loop, **data[i]**, with a permutation table lookup **permutationTable[data[i]]**, which could be constructed as follows.

Code Listing 3

```
static byte[] permutationTable = new byte[byte.MaxValue];  
  
...  
  
for (byte i = 0; i < byte.MaxValue; ++i) permutationTable[i] = i;  
permutationTable = permutationTable.OrderBy(  
    b => Guid.NewGuid()  
) .ToArray();
```

permutationTable is a randomly shuffled one-to-one (bijective) mapping of every byte to some other byte. You could do it with full rigor by using CSP for randomization and [Fisher-Yates](#) for

shuffling, but even a simple sort-by-GUID approach will produce a uniform shuffling with acceptable unpredictability. The FNV-1a algorithm itself does not change—only the input bytes change as they go through the **permutationTable** before being consumed by FNV-1a. **permutationTable** would be calculated once (for example, in a static constructor) and would retain its mapping over the life of AppDomain.

This approach will allow you to keep using FNV-1a while raising the difficulty barrier for mounting a successful hash flood attack against it. Naïve brute-forcing to guess the permutation is infeasible due to the 1684-bit permutation space [$\log_2(256!)$], so an attacker would have to do differential cryptanalysis.

SHA-1 hash

Despite **SHA-1** being cryptographically retired (significantly less than brute-force attacks are known), there are still compelling reasons to consider using it for certain purposes:

- SHA-1 is still practically secure with respect to both first-preimage resistance (no collisions are published yet) as well as second-preimage resistance, which is the only property we care about for hash flooding defense. In fact, NIST's own SP800-131A guidance declares SHA-1 to be acceptable indefinitely for all non-signature applications.
- SHA-1 is the fastest SHA-family hash function, and, along with MD5, is probably one of the fastest among all other “standardized” cryptographic hash functions. In our non-scientific tests, SHA1Cng was about four to ten times slower than FNV-1a, depending on input length, which might be acceptable in your scenario (FNV-1a is very fast). You might also be able to cache the calculated SHA-1 hash in scenarios where your input is either immutable (strings, anonymous types) or changes infrequently.
- SHA-1 implementation is available in the .NET Framework.

These reasons make us suggest SHA1Cng as an alternative general-purpose 32-bit hash function able to resist hash flood attacks. SHA-1 is 160 bit; it produces 20 bytes of output. You can take any four bytes of that output and convert them to **Int32**. If you cannot overcome regulations or policies or any other stigma associated with use of SHA-1, you might as well use SHA512Cng. You can also substitute SHA-1 with HMAC-SHA-1 (covered later), which will be twice as slow as SHA-1 and on par with SHA-512, which is about twice as fast as SHA-1 in 64-bit environments.

SipHash

You might be facing scenarios when neither compromising hash function security at the expense of performance (the status quo) nor compromising performance at the expense of proper security is acceptable. Indeed, why should it be acceptable? Why can't cryptography experts come up with a design that is fast, relatively simple, and non-collision-resistant yet still second-preimage resistant? One such design is [SipHash](#).

SipHash achieves speeds comparable to modern state-of-the-art non-cryptographic functions, yet does not compromise on security. It can even run as a **message authentication code (MAC)**, with some injected entropy acting as a key, for example. Among the downsides of using SipHash is the fact that it is relatively new, and thus has not received as much scrutiny as older designs. A design needs to become popular in order to attract a lot of scrutiny, which is also a prerequisite for the design to become popular—it takes time to break out of this cycle. Another downside is that SipHash is not built into .NET, and you would be taking a risk trusting any implementation you can find out there by Googling “SipHash c#”, or by implementing it yourself. You would also be taking this risk with a custom FNV-1a implementation. Unlike FNV-1a, however, SipHash cannot be implemented in five lines of code.

Even though there is a lot to like about SipHash, we would recommend against using it for now due to a lack of trusted, time-tested, and maintained implementation in the .NET Framework. We sincerely hope that SipHash is as good as it claims to be, and that tested and vetted .NET implementations will come with time—perhaps even as part of .NET Framework (Perl, Ruby, Rust, and Haskell have already switched to SipHash). In the meantime, you should only consider SipHash if you must have your cake (hashing speed) and eat it too (second-preimage resistance).

SipHash is not the only lightweight cryptographic hash function design—there are other, arguably better options, such as [BLAKE2](#) (also from 2012). They are all too new, and too understudied—user beware.

Modern non-cryptographic hashes

The [MurmurHash](#) family, [SpookyHash](#), and [CityHash](#) are some examples of modern non-cryptographic hash functions. They are all fast, and are likely closer to perfection than FNV-1a due to extra complexity. None of them are designed to resist hash flooding attacks, however ([CVE-2012-6051](#), [oCERT](#)), and none of them have built-in .NET Framework implementations, which leaves little reason to consider any of them over SipHash or BLAKE2.

Cryptographic hashes

It is hard to justify anything other than **SHA-2** hash function family for scenarios requiring full-fledged cryptographic hash properties. SHA-2 is time-tested, standardized, NIST-approved, and implemented in .NET. **SHA-3** should be treated as a “backup” for SHA-2 in case some major SHA-2 weakness is discovered, but should not be the primary choice. Even when SHA-3 implementation becomes part of the .NET Framework, SHA-2 should still be preferred since SHA-3 software performance is on par with or worse than SHA-2.

You should prefer **SHA-512** over **SHA-256** because SHA-512 is typically 30–50% [faster](#) than SHA-256 in software on 64-bit systems (your typical deployment environment) while being a lot slower than SHA-256 on dedicated hardware, optimized for brute-forcing. The fact that SHA-512 is more secure than SHA-256 due to extra digest length and complexity is not really material, since SHA-256 is secure enough. Many frameworks with built-in support for the SHA-2 hash family (such as .NET 4.5) use SHA-256 by default. We recommend that you change these

defaults to use SHA-512 in 64-bit environments (not for security reasons, but for performance reasons).

The best general-purpose SHA-2 family hash, however, is not SHA-512, but **SHA-384**, which is a truncated version of SHA-512 (i.e. has the same performance as SHA-512) computed with different initial values. The truncated nature of SHA-384 protects it from a length-extension vulnerability affecting all non-truncated SHA-2 hashes.



Tip: Use SHA-384 as your default general-purpose cryptographic hash.

HMAC (including hash/HMAC factories)

Hash-based message authentication code (HMAC) is a specific form of MAC, which, like all MACs, is used for verifying message authenticity. It can be used with any cryptographic hash function, and requires no initialization vector (IV). HMAC has three inputs: (1) the cryptographic hash function to use, (2) message, and (3) key.

The MAC security model allows the attacker to observe lots of $\text{MAC}(m,k)$ results for a fixed secret key k and any message m , which the attacker can choose arbitrarily. The MAC is secure if the attacker cannot compute $\text{MAC}(M,k)$ where M is a new message distinct from all previous m messages. Such a new message, if computed, would be a forgery.

The MAC security model is not meant for message confidentiality. Some MAC algorithms might leak quite a lot of information about the message, while other MAC algorithms, such as HMAC, tend to be quite effective at preserving message confidentiality, as well as key confidentiality. It is prudent, however, to use HMAC within the security model it was designed for. Therefore, you should not assume that HMAC (or any other MAC) will provide message confidentiality (but if you screw it up and happen to be using HMAC, it might save you).

HMAC was proven in 2006 (Bellare) to be secure even when the underlying cryptographic hash function is weakly collision-resistant (i.e. collisions in an n -bit hash function could be found with less than $n/2$ -bit complexity), which might explain why no serious weaknesses have been found in HMAC-MD5 and HMAC-SHA1 constructions so far, even though the underlying MD5 and SHA-1 hash functions have serious weaknesses.

HMAC in .NET

.NET has had built-in HMAC support since .NET 2.0. There is a base abstract **HMAC** class with the core HMAC transformation logic, and several derived classes—**HMACMD5**, **HMACRIPEMD160**, **HMACSHA1**, **HMACSHA256**, **HMACSHA384**, and **HMACSHA512**—that implement a particular HMAC-hash combination. This is a tightly coupled design—if Microsoft wants to add a new hash function implementation (e.g., SHA-3), they will have to create two classes: **NewHash** and **HMACNewHash**. Adding another MAC implementation, **NewMAC**, would necessitate a multitude of additional classes for every **NewMAC**-hash combination. If you accept this design, you will be in the same situation as Microsoft.

A better design would have been to have a concrete **HMAC** class that takes any **HashFunction** factory as part of its constructor. Such a hash factory could be smart enough to return either a FIPS-compliant implementation (when needed), or a managed implementation. Alternatively, the **HMAC** constructor could have been accepting two **HashFunction** factories—one that produces FIPS-compliant implementations, and another that produces managed implementations—with all the FIPS-checking logic (which never changes) within the **HMAC** class itself rather than within the factory.

This factory-based design is, in fact, already inside a private, internal static HMAC method:

Code Listing 4

```
internal static HashAlgorithm GetHashAlgorithmWithFipsFallback(
    Func<HashAlgorithm> createStandardHashAlgorithmCallback,
    Func<HashAlgorithm> createFipsHashAlgorithmCallback);
```

This method is called by each deriving HMAC-hash class, which provides its own factories. Unfortunately, this method is only accessible to .NET's own HMAC-hash implementations; Microsoft does not want you to use it. What Microsoft is effectively doing here is forcing you to use an inheritance-based implementation injection instead of a constructor-based injection. That is okay—we will try to make the best of what is available by using the conveniently provided **HMACSHA256** class. There is a problem, however. Run the following test in your .NET environment (we are using [LINQPad](#)):

Code Listing 5

```
void Test(Func<HashAlgorithm> hashFactory)
{
    var data = new byte[1024];
    int cnt = 1000*100;
    var sw = Stopwatch.StartNew();
    for (int i = 0; i < cnt; ++i)
        using (var hash = hashFactory())
            { hash.ComputeHash(data); }
    sw.Elapsed.Dump(hashFactory().GetType().Name);
}

Test(() => new SHA512Managed());
Test(() => new SHA512Cng());
Test(() => new SHA512CryptoServiceProvider());
```

Typical results are:

SHA512Managed:	00:00:02.4827152
SHA512Cng:	00:00:00.9306346
SHA512CryptoServiceProvider:	00:00:04.8959034

As you can see, the CSP-based hash construction is about five times slower than the CNG-based construction, and about two times slower than managed. All factories in .NET-provided HMAC hash implementations follow the following logic: if FIPS is not required, use managed, otherwise use CSP (and not CNG). Microsoft is being conservative by ensuring that their implementations are guaranteed to work on all legacy platforms. You, however, probably do not want to pay a five-times penalty for FIPS deployments, because you do not plan to run your production environment on a legacy Windows OS. In fact, .NET 4.5 *requires* Vista/Windows 2008 or higher, so paying this “legacy tax” makes no sense. HMAC requires two hash function instantiations; Microsoft’s HMAC implementation refers to them as **hash1** and **hash2**. That doubles the actual construction time, while preserving the four-times performance penalty over the managed API.

How significant is this performance penalty in practical scenarios? It is very significant. Many applications of HMAC operate on short message sizes, and require large numbers of quick HMAC calculations. The invocation cost of any HMAC hash in such scenarios will be significant, or, perhaps, even greater than the cost of message processing and result generation. In an extreme example of hashing an empty byte array, 100,000 iterations of SHA-512 take 0.2, 0.5, and four seconds for managed, CNG, and CSP implementations respectively. You might recall similar trade-offs in our discussion of random number generation. Unlike the **RNGCryptoServiceProvider** class, which is thread safe, **HMAC** and its derived implementations are not thread safe, and thus have to be constructed for every HMAC computation, amplifying any high invocation costs to a significant level.

In order to get acceptable HMAC performance in FIPS deployments, you have no choice but to make your own HMAC-derived implementations that use improved hash factory logic. The moment you try to provide a concrete implementation of the abstract **HMAC** class, you will realize that **HMAC** cannot be sub-classed by external implementations in any meaningful way, because key internal fields such as **hash1** and **hash2** are internal and not accessible to you. In other words, the public-base **HMAC** class is built by Microsoft for Microsoft. .NET Framework authors are effectively saying “if you don’t like our HMAC design, build your own.”

Another issue with .NET-provided HMAC-salt implementations is that their default (parameterless) constructor creates a random, CSP-generated 512-bit key for all implementations using 256-bit or lower hash functions (e.g., **HMACSHA1**, **HMACSHA256**), or a 1024-bit key for all implementations using hash functions above 256 bits (e.g., **HMACSHA384**, **HMACSHA512**). This auto-generated key is then passed to the internal **InitializeKey()** method, which sets up internal buffers and does various checks. The key auto-generation and the **InitializeKey()** call are both initialization activities that require computation and time.

A common HMAC usage scenario, however, is using it as a cryptographic primitive in other, more complex cryptographic primitives, such as PBKDF2 or HKDF (covered later). In these scenarios, the HMAC-consuming primitives need to provide their own key to HMAC hash

implementations. There are two approaches to achieve that: either the HMAC is instantiated with the default constructor and then re-keyed to the desired key, or HMAC is instantiated with another constructor that takes the key directly. The first approach results in a waste of computation and time spent on auto-generating a strong default key in order to immediately discard it and re-initialize with the provided key. The second approach does not waste computation and time, but it tightly couples the *creation* of the HMAC-hash primitive to its *keying*. Such tight coupling is simply a bad design for reusable cryptographic primitives that are supposed to act as building blocks for other, more complex constructs.

In order to generically consume specific method signatures, they need to be either a part of an interface, a base class, or a default constructor (C#). The key-accepting constructor on .NET-provided HMAC hash classes fits neither of these cases, and thus cannot be generically consumed (without tight coupling the consuming code to the specific HMAC hash implementation). Always generating a strong HMAC key in the default constructor is a “security-by-default” approach; Microsoft is being conservative, which would normally be a good idea. In this case, however, the default HMAC constructor is the only way to decouple HMAC-hash creation and keying.

An improved **HMAC2** class implementation can be found in the [Inferno](#) crypto library. **HMAC2** is a lot faster in FIPS and non-FIPS deployments, and is a little faster when consumed by other, more complex primitives that provide their own key. It also avoids tight coupling and reimplementations of the **HMAC** class.

Chapter 3 Key Derivation

The preceding HMAC discussion has touched on the notion of a **secret key**, which determines the output of a keyed cryptographic primitive. Secret keys should ideally contain enough [entropy](#) to render brute-force attacks impractical. That typically calls for MAC keys of at least 256 bits and encryption keys of at least 128 bits. The ideal way to generate such keys is to use either a true random number generator (rarely available), or a cryptographically-strong pseudo-random number generator (based on CSP, for example). There are very common scenarios, however, where such randomly generated keys are either impractical or insufficient (and can cause either usability or security flaws). **Key derivation functions (KDFs)** are often used in these scenarios to derive one or more strong keys from a single, secret value.

Some cryptographic algorithms and protocols can have keys that are known to be [weak](#), and thus should be avoided; KDFs can help avoid weak keys in such systems. Another common use of KDFs is to extract entropy from user-provided keys (which we often call passwords or passphrases) and derive the actual key via a deliberately slow computation process intended to thwart brute-force password guessing attacks. KDFs are also often used as part of a [defense in depth](#) security strategy. We will not cover the weak key avoidance uses of KDFs since you will not encounter any cryptographic algorithms with known weak keys in this book, but we will cover the other two common KDF scenarios next.

KDFs are often designed as a two-part extract-then-expand process. The “extract” part extracts as much entropy as possible from the input key and generates a fixed-length intermediate key. This intermediate key is supposed to be a short but cryptographically strong, uniformly distributed representation of the entropy gathered from the input key. The second, “expand” part of the process takes that intermediate key and expands it into a final key or keys based on specific objectives that each KDF is designed to achieve.

PBKDF2

Password-based key derivation function version 2 (PBKDF2), covered by [RFC-2898](#) is a specific KDF designed for user-provided password cryptography. Password-based KDFs like PBKDF2 typically employ three defense mechanisms to generate a strong derived key:

- **Condensing** all available password entropy into a uniform intermediate key (the “extract” process).
- [Salting](#) the “extract” process to prevent scalability of brute-force attacks against multiple keys.
- **Expanding** the intermediate key via a slow computation process that’s very costly to make faster.

The slow computation of the third item will affect both legitimate users and brute-forcing attackers, but not in equal proportions. A legitimate user’s password check might be subject to

one such slow computation on every check, which is likely to be fast enough not to affect usability, while the attacker would have to do this computation for every new password guess.

It is also prudent to assume that a resourceful, driven attacker will have access to much better hardware (likely custom-built and optimized for brute-forcing) than a legitimate installation. Modern cloud computing services can also be easily utilized to spin up large, powerful server farms dedicated to brute-force guessing, which can achieve the same results with large numbers of commodity hardware as fewer, more powerful, custom-built hardware resources.

The third mechanism would not have been necessary had passwords contained sufficient entropy to begin with. Unfortunately, high-entropy passwords are typically at odds with usability because they are hard for humans to come up with quickly and reliably. They are also hard to memorize, and difficult and time-consuming to type and re-type without typos.

Three password-based KDF algorithms are worth considering: **PBKDF2**, **bcrypt**, and **scrypt**.

Table 3: Comparison of PBKDF2, bcrypt, and scrypt

	PBKDF2	bcrypt	scrypt
Maturity	1999	1999	2009
Variable CPU hardness	yes	yes	yes
Variable memory hardness	no	no	yes
Memory requirement	tiny	small	variable
IETF spec	yes	no	yes (draft)
Weird assumptions/limitations	no	yes	no
Variable output length	yes	no (192 bits)	no (256 bits)
NIST-approved	yes	no	no
MS-supported .NET implementation	yes (subpar)	no	no

Table 3 is a quick, non-comprehensive comparison of these three password-based KDFs based on criteria that should be relevant from a .NET security engineering point of view. The last two

points make PBKDF2 the preferred choice, since you want neither to trust an unsupported implementation nor to roll out your own.

The .NET-provided implementation of PBKDF2 is in the **Rfc2898DeriveBytes** class, which implements the abstract **DeriveBytes** base. You might also come across **PasswordDeriveBytes**, which is an implementation of the older PBKDF1 algorithm—do not use it. The PBKDF1 algorithm is weak, and Microsoft's implementation of it is broken to boot.

One issue that immediately becomes apparent about **Rfc2898DeriveBytes** is that it implements the PBKDF2 algorithm with HMAC-SHA1 only (another example of Microsoft's tightly coupled design). It is not even possible to hack **Rfc2898DeriveBytes** into operating on any other HMAC hash primitive because SHA-1 is hard-coded into the **Rfc2898DeriveBytes** implementation. One reasonable approach is to simply accept what is given and use **Rfc2898DeriveBytes**. This is a reasonable choice given the limiting circumstances. Another approach is to build a new, flexible PBKDF2 implementation by reusing as much of Microsoft's implementation as possible, and only removing the hard-coded HMAC-SHA1 dependencies. You can find such **PBKDF2** class implementation in the [Inferno](#) crypto library. It has been validated against published PBKDF2-HMAC-SHA1, PBKDF2-HMAC-SHA256, and PBKDF-HMAC-SHA512 test vectors.

A side benefit of being forced to do a re-implementation of PBKDF2 is choosing better defaults. **Rfc2898DeriveBytes** defaults to 1,000 iterations, which was considered adequate in 1999, but is too low for current brute-forcing technology. PBKDF2 defaults to 10,000 iterations, but you are encouraged to override that and use the maximum possible value that does not affect usability in your scenario.

HKDF and SP800_108

There are many scenarios where your **source key material (SKM)** already has sufficient entropy (SKM is a CSP-generated 16-byte random key, which already has 128 bits of entropy). These high-entropy SKM scenarios do not require entropy extraction, but they might still benefit from entropy condensation, where a fixed-length or variable-length high-entropy SKM is condensed into a fixed-length uniformly distributed representation. This condensation is useful even when the SKM has a vast amount of entropy in it because the entropy may not be evenly distributed among all SKM bits (some SKM sections capture more entropy than others).

The salting step is still relevant, but for different reasons. Guessing is no longer a concern (due to high-entropy SKM), but potential SKM reuse still is, so a known distinguisher is often used for extra entropy in the expansion step. The expansion step is relevant when multiple derived keys need to be generated from a single master key, and the evenly-distributed, fixed-length condensation might not be long enough to simply slice it into multiple, derived key lengths. Why would you ever need multiple derived keys to begin with? Why would a single key derived from a high-entropy SKM (which you are fortunate enough to have) not be sufficient?

For over seven years (.NET 2.0 to .NET 4.0), Microsoft believed that it was perfectly sufficient, based on their ASP.NET **machineKey** implementation. Microsoft used the same strong **machineKey** master key for various components, each using the same master key for encryption.

Serious security [vulnerability](#) was uncovered in September 2010 in the implementation of one of these components, which allowed an attacker to uncover the encryption key for that component. The affected ASP.NET component was not a critical one, and did not itself have a high exploitability or damage potential. However, because the same key was used by all other components, an attacker essentially obtained the key to the kingdom. This ASP.NET vulnerability was a serious blow to the security design of ASP.NET 4.0 and all earlier versions—serious enough to warrant a major security overhaul in ASP.NET 4.5, and a new ASP.NET crypto stack (covered in the following chapters), which is now based on multiple keys derived from a single high-entropy **machineKey** master key.

The Microsoft ASP.NET security team working on ASP.NET 4.5 realized that .NET had no KDF implementation designed for high-entropy SKM, and had the [following](#) to say on the .NET Web Development and Tools Blog:

“At this point we resigned ourselves to the fact that we'd have to implement the KDF in our own layer rather than calling any existing APIs. ... We ended up choosing [NIST SP800-108](#) [PDF link] due to the fact that it is designed to derive a new key from an existing high-entropy key rather than from a low-entropy password, and this behavior more closely aligns with our intended usage.”

Microsoft's SP800-108 implementation is in a new **SP800_108** class, which along with about 18 other tightly-coupled new security classes, is marked **private internal** in their System.Web.dll. It seems that we also have to resign ourselves to the fact that we have to implement our own KDF because Microsoft does not share its new toys.

Faced with such resignation, we can give some thought to whether SP800-108 is the best general-purpose high-entropy KDF to implement. SP800-108 is a decent choice, and also comes NIST-recommended. Microsoft had to react quickly, and their choice of SP800-108 shifted the algorithm risk to NIST. SP800-108 actually defines three KDF algorithms: a counter mode algorithm, a feedback mode algorithm, and a double-pipeline iteration algorithm. Microsoft's **SP800_108** class implements the counter mode algorithm because it is simple, fast, and allows O(1) derivation of any key subsection. However, there is another KDF algorithm, **HMAC-based KDF (HKDF)**, which is arguably a better generic multi-purpose KDF than the SP800-108 algorithms. HKDF is proven to be secure, well-specified ([RFC-5869](#)), and includes distinct entropy-extraction and key-derivation steps. SP800_108 counter KDF is better suited for scenarios where performance is crucial and the entropy-extraction step is not necessary, while HKDF is a better and safer general-purpose KDF. The [Inferno](#) crypto library provides both [HKDF](#) and [SP 800_108_Ctr](#) implementations.

PBKDF2 combined with HKDF

It often makes sense to use both PBKDF2 and HKDF together to derive multiple keys. PBKDF2 excels at deriving a single master key from a low-entropy SKM, such as anything user provided. However, PBKDF2 is ill-suited for producing output beyond the length of its HMAC-wrapped hash function, because each additional PBKDF2 rotation is subject to the same slow computation process. For example, **PBKDF2-SHA256** has a single-rotation output length of 32 bytes (256 bits). If you ask PBKDF2-SHA256 for 64 bytes of output (for example, if you need two 256-bit derived keys), or even if you ask it for 32+1=33 bytes, PBKDF2-SHA256 will take twice as long to run, because it will do two rotations instead of one.

You have already picked the PBKDF2 iteration count to be as high as possible without affecting usability of your system. The maxed-out iteration count was for **one** rotation only—making more than one PBKDF2 rotation would surely affect usability. You could switch to a longer hash function for PBKDF2 (essentially maxing out at 512 bits with SHA-512), but you would be avoiding the problem instead of solving it (nor would it scale beyond 2–4 keys, depending on their purpose).

A better approach is to use PBKDF2 to derive the master key, and then use HKDF (or SP800_108_Ctr) keyed with the PBKDF2-derived master key to efficiently generate any number of additional master-key-derived keys.

Salt

Cryptographic **salt** is an additional non-secret source of entropy used to prevent vulnerabilities due to reuse of SKM, either by the same source, or across multiple sources that happen to use the same SKM. Salt should always be used when you cannot guarantee the non-reuse of SKM (either deterministically or statistically). Note that the quality or entropy of SKM does not matter, only its potential reuse does; you should use salt even if you have a high-quality SKM with a potential for reuse.

The only requirement for salt is to be unique. If salt is not unique, then it can no longer act as a distinguisher when SKM is reused, and you fail. The uniqueness requirement can be implemented deterministically, for example, by using an atomically incrementing counter which you can guarantee to never repeat for all SKMs. Uniqueness can also be implemented statistically or statelessly by using a 128-bit, CSP-generated random value.

GUIDs are perfect candidates for salt because they are designed to provide uniqueness, not unpredictability. Deterministic implementations have the undesirable requirement of having to keep state, which adds more complexity and more opportunities for mistakes. We recommend that you use a GUID when you need a salt for two reasons: (1) GUID meets all salt requirements and the new GUID generation in the .NET Framework is impossible to screw up (we dare you); (2) Using GUIDs for salt clearly communicates the intent: uniqueness, not unpredictability. When we say GUID, we mean the 16-byte array representing a GUID, not GUID string form.

Salt is often confused with initialization vectors and misused; we'll discuss the differences in the [“Symmetric Encryption”](#) chapter.

Client-side KDF

One obvious consequence of employing PBKDF2 and similar password-based KDF schemes to derive keys from low-entropy passwords is their high computational cost, measured in CPU or memory utilization. This cost can quickly become a limiting factor for server-side throughput, measured in requests per second.

One tempting idea is moving PBKDF2 computation from the server side to the client side. Let's cover why this idea falls apart on closer inspection.

Typical client-side devices tend to have inferior CPU and memory resources compared to server-side infrastructure. Client-side software is often not even able to fully utilize the client-side resources that are available. Many modern browsers are stuck in 32-bit mode even when they are running in 64-bit environments and are unable to fully utilize available memory. Even when browsers are running in 64-bit mode, their computational capabilities are typically limited by their single-threaded JavaScript engines and are unable to fully utilize available CPUs. Mobile client devices are often primarily battery-powered, and are even further constrained in their resource capabilities. There is also client-side CPU throttling to preserve and extend battery life.

The crucial implication of client-side resource inferiority and unpredictability is that the number of client-side PBKDF2 rounds that can be tolerated with acceptable usability is typically a lot lower compared to server-side execution. Any substantial reduction in the number of PBKDF2 rounds would weaken PBKDF2 security and circumvent its intended purpose.

Another problem is that PBKDF2 is salted (if it's not, then you're not using it right). Salt values are available on the server side (which makes server-side PBKDF2 straightforward), but not on the client side. Sending salt values to the client side would not only introduce additional complexity, but would weaken security as well. While salt values do not have to be secret, they should not be blissfully disclosed to anyone who asks, either. An adversary could ask for a salt value of an administrator account and combine it with a list of common passwords to generate potential credentials and quickly mount an online attack. Blindly disclosing salt values is clearly not in our best interests.

Key separation

The **key separation principle** states that you should always use distinct keys for distinct algorithms and distinct modes of operation. Some violations of this principle can even lead to plaintext recovery without ever knowing the secret key. Fortunately, it is easy to adhere to the key separation principle by using a master-keyed HKDF or SP800_108_Ctr with a purpose-specific distinguisher to generate as many distinct keys as you require. In symmetric encryption scenarios, different encryption and decryption purposes as well as different operations (for example, encryption versus authentication) should use independent keys. In asymmetric encryption scenarios, encryption and signing should use independent key pairs.

Chapter 4 Comparing Byte Arrays

Comparing two byte arrays for equality is not only one of the most primitive reusable operations, but it is also one of the most commonly used in cryptography. The fundamental requirement for a **byte array comparison (BAC)** is to return **true** if two byte arrays are equal, and **false** otherwise. Two byte arrays are equal when their lengths are equal and all byte pairs in corresponding array positions are also equal.

BAC gets an important additional security requirement when used in security constructs: BAC implementation should not leak any information about the data within the two arrays being compared for equality.

Direct comparison (insecure)

Code Listing 6

```
static bool Direct(byte[] a, byte[] b)
{
    if (a.Length != b.Length) return false;
    for (int i = 0; i < a.Length; ++i)
    {
        if (a[i] != b[i]) return false;
    }
    return true;
}
```

Direct comparison implements the fundamental BAC requirement, but it also makes a variable number of comparisons based on the position of the first unequal byte pair, which leaks timing information that can help mount effective guessing attacks to uncover the data. These types of attacks are known as [side-channel attacks](#) (the side channel in this case is timing). You should never use direct comparison with sensitive or secret data.

AND comparison

Code Listing 7

```
static bool And(byte[] a, byte[] b)
```

```

{
    bool f = a.Length == b.Length;
    for (int i = 0; i < a.Length && i < b.Length; ++i)
    {
        f &= a[i] == b[i];
    }
    return f;
}

```

The **AND** comparison is similar to a direct comparison that keeps comparing byte pairs even after the result is determined to be **false**. It performs a fixed amount of work in every comparison, which protects against timing attacks. AND comparison is the BAC approach used by Microsoft within their internal **CryptoUtil** classes (**System.Web.Security.Cryptography** namespace—**System.Web** assembly and **System.Web.Helpers** namespace—**System.Web.WebPages** assembly).

XOR comparison

Code Listing 8

```

static bool Xor(byte[] a, byte[] b)
{
    int x = a.Length ^ b.Length;
    for (int i = 0; i < a.Length && i < b.Length; ++i)
    {
        x |= a[i] ^ b[i];
    }
    return x == 0;
}

```

XOR comparison exploits the fact that equal byte pairs XOR to zero and OR-ing all XOR-ed results together would only produce zero if all byte pairs are equal. The XOR approach might have a slight advantage over the AND approach because the XOR-OR combination is pure bit shifting, and is typically compiled into direct assembly equivalents, while Boolean AND combination can be compiled into conditional branching instructions, which might have different timings.

Double-HMAC comparison

The AND and XOR approaches force you to make certain assumptions about potential timing variabilities in the implementation of basic operations on a particular platform. Microsoft's implementation of .NET runs on Windows, where Microsoft has a good understanding and control of how compilation happens. Other .NET implementations, such as Mono, might also target specific platforms and have a good handle on what assembly-level code is produced.

There might be scenarios, however, where you are not comfortable making specific assumptions about the implementation of low-level primitives such as Boolean logic or bit-shifting operations. In these scenarios, you can do a random-key HMAC of both arrays first, and then do BAC on the HMAC results rather than on the original arrays. Once you replace the original arrays with their HMAC equivalents, the choice of BAC is no longer important: you can use either one.

Code Listing 9

```
static bool DoubleHMAC(byte[] a, byte[] b)
{
    byte[] aHMAC, bHMAC;
    using (var hmac = HMACFactories.HMACSHA1())
    {
        // key unpredictability not required
        hmac.Key = Guid.NewGuid().ToByteArray();
        aHMAC = hmac.ComputeHash(a);
        bHMAC = hmac.ComputeHash(b);
    }
    return Xor(aHMAC, bHMAC) && Xor(a, b);
}
```

Note that when **Xor(aHMAC, bHMAC)** returns **true**, we actually return the result of an additional **Xor(a, b)**. The reason is that fixed-length digests such as hash functions and HMAC are guaranteed to have domain collisions. There must exist two arbitrary-length unequal arrays **a** and **b**, which produce the same fixed-length hash/HMAC value. You cannot come up with such **a** and **b**, but they do exist. We perform an additional BAC on (**a**, **b**) directly in order to ensure that such collisions, when they happen, do not lead to a false positive result.

The double-HMAC approach is obviously slower and more complex than all other approaches discussed so far. We prefer the XOR approach for all Microsoft .NET-to-MSIL languages (C#, VB.NET, F#), and the double-HMAC approach for languages that ship source code to be compiled with an undetermined compiler (JavaScript).

Chapter 5 Binary Encodings

Binary encodings are commonly used to encode byte sequences into displayable, printable sequences (usually some subset of [ASCII](#)). Binary encodings are very useful when you need to store binary data in systems or protocols that do not offer native binary storage support, but do support text or ASCII-subset storage (for example, browser cookies). It is important to understand the most common binary encoding schemes and the various trade-offs involved when choosing which one to use and when. The Base64, Base32, and Base16 encodings we discuss next are covered in [RFC-4648](#).

Base64

Base64 encoding converts an 8-bit sequence into a 6-bit sequence, where each output character comes from a range of $2^6 = 64$ different values. The [least common multiple](#) (LCM) of 8 and 6 is 24, so 24 input bits represented by 3 bytes can be Base64-encoded into four 6-bit values. Therefore, Base64 encoding has a $4/3 = 1.33\times$ length bloat factor compared to the 1x length of the source. The 64 output ASCII values are comprised of 26 lowercase letters, 26 uppercase letters, 10 digits, and two additional characters, `/` and `+`. A distinct padding character, `=` (equal sign), is used at the end to indicate that the source sequence was one byte short of a multiple of 3 (a single `=` at the end), or two bytes short of a multiple of 3 (`==` at the end). Each encoded four-character block has a minimum of two non-pad characters, since a single 8-bit byte takes at least two 6-bit characters to encode.

Base64 encoding is extremely popular due to its small bloat factor and built-in availability in most libraries and frameworks. The .NET Framework has the [Convert.ToBase64String\(\)](#) and [Convert.FromBase64String\(\)](#) methods. Note that the `/`, `+`, and `=` characters in default Base64 often do not play nice when embedded into other text protocols in which these characters have special meaning (for example, in URLs and file paths). To address these scenarios, .NET also has a safe-alphabet Base64 implementation, [UrlTokenEncode\(\)](#) and [UrlTokenDecode\(\)](#) in `System.Web.HttpServerUtility`, which replaces `/` and `+` with `_` and `-`, and always appends a single-digit pad count (either 0, 1, or 2) instead of the `=` pad.

Code Listing 10

```
var a = new byte[] { 0 };
var b = new byte[] { 0, 0 };
var c = new byte[] { 0, 0, 0 };

Convert.ToBase64String(a).Dump();
System.Web.HttpServerUtility.UrlTokenEncode(a).Dump();
```

```

Convert.ToBase64String(b).Dump();
System.Web.HttpServerUtility.UrlTokenEncode(b).Dump();

Convert.ToBase64String(c).Dump();
System.Web.HttpServerUtility.UrlTokenEncode(c).Dump();

// Results:
AA==
AA2

AAA=
AAA1

AAAA
AAAA0

```

As you can see, safe Base64 can be longer, equal, or shorter than the equivalent default Base64 encoding. The .NET implementation of safe Base64—the **UrlTokenEncode/Decode** methods—calls its own default Base64 (**Convert** methods), and then does two more loops: one to replace the two special characters with their safe equivalents, and one to replace the variable-length = pad with a fixed-length digit pad. This “unoptimized” implementation makes **UrlTokenEncode/Decode** needlessly slower than the default Base64. This is something to be aware of for extreme edge cases where micro-optimizations are your last resort. It is definitely not a reason to roll out your own “optimized” version without ever measuring to obtain concrete evidence that this is your performance bottleneck. **UrlTokenEncode/Decode** Base64 methods should thus be preferred to equivalent **Convert** methods because they integrate better into other text protocols. If you prefer to avoid taking a dependency on **System.Web.dll**, we suggest using **ToB64Url()/FromB64Url()** methods from the [Inferno](#) library ([code sample](#)).

Base32

Base32 encoding converts an 8-bit sequence into a 5-bit sequence, where each output character comes from a range of $2^5 = 32$ different values. The LCM of 8 and 5 is 40, so 40 input bits represented by 5 bytes can be Base32-encoded into eight 5-bit values. Therefore, Base32 encoding has an $8/5 = 1.6\times$ bloat factor. The 32 output ASCII values are comprised of 26 same-

case letters (either lower or upper) and six digits between 2 and 7. Many other Base32 alphabets are possible, however (with different features). The same = pad is used as necessary (and repeated up to five times) to indicate source sequences that are not multiples of 5.

Base32 encoding seems very unattractive due to its 60 percent bloat factor (almost double the Base64 bloat), which raises the question of why anyone would use Base32 encoding instead of Base64. One Base32 advantage is that, unlike Base64, it's not case sensitive, and thus is easier for human beings to type. Another consequence of case insensitivity is that Base32 encoding can survive when embedded in case-insensitive protocols or other scenarios where case conversion might occur outside of your control.

One example is ASP.NET **SessionID**, which is a 15-byte sequence Base32-encoded by Microsoft into 24 characters (e.g., **fes21mw1yitkbbkqgkxek0zhp**). We speculate that Microsoft has chosen to use Base32 to encode **SessionID** because they had to support cookieless sessions, which required passing **SessionID** within the URL, which in turn required case insensitivity (various proxies and web servers can uppercase or lowercase a URL, which is outside of Microsoft's control). Microsoft probably wanted to avoid Base32 padding, which adds quite a lot of noise and bloat, so their **SessionID** sequence length choices were limited to multiples of 5. 5-byte (40-bit) and 10-byte (80-bit) sequences were too short, but 15 bytes (120-bit) gave adequate security against brute-force guessing. A longer, 20-byte (160-bit) or 25-byte (200-bit) Base32-encoded **SessionID** might be a slightly better choice today, but Microsoft is unlikely to change their defaults because a longer **SessionID** might break existing applications due to URL length limitations in various systems and other related incompatibilities triggered by longer **SessionIDs**.

Another example of Base32 relevancy is encoding short binary sequences that are intended to be typed or re-typed by human beings. Suppose that instead of relying on your users to come up with a sufficiently high-entropy password, you want to generate a high-entropy, 128-bit (16-byte) password for each user, which you have to binary-encode somehow for display. You want to avoid Base64 so that your users do not have to struggle with the Shift and Caps Lock keys when typing. You could use Base32, but there are some issues you need to address first. Base32 works best on multiple-of-5 sequences, so you could either settle for a 15-byte key (120-bits), or go for a longer, 20-byte key (160-bits). Alternatively, you could go for a 16-byte key like you wanted, and remove the five = pad characters from the end. This would create a 26-character, Base32-encoded key that your users can re-type, but it would also force you to assume a fixed key length (you have to add the missing pad characters yourself if you want to decode Base32 back into a byte sequence). This approach is similar to what [Mozilla Sync](#) does.

Another important Base32 consideration is what alphabet to use for encoding. We prefer the following:

Code Listing 11

```
static readonly char[] base32table =
{
    // 'i', 'l', and 'o' are omitted to avoid confusion.
    // Sort order is maintained.
```



```
'1','2','3','4','5','6','7','8','9','a','b','c','d','e','f','g',  
'h','j','k','m','n','p','q','r','s','t','u','v','w','x','y','z'  
};
```

This alphabet maintains the source sequence sort order within the output sort order, and it also removes the **0**, **o**, **i**, and **l** characters, which can be easily misread. **1** is usually distinct and clear enough in most fonts not to be mistaken for **l** (lowercase L) or **I** (capital i).

The .NET Framework has no public APIs for Base32 encoding or decoding, even though Microsoft uses Base32 encoding internally to encode **SessionID**. You can find all encode and decode implementations we discuss in the [Inferno](#) crypto library ([code sample](#)).

Base16

Base16 encoding converts 8-bit sequences into 4-bit sequences, where each output character comes from a range of $2^4 = 16$ different values. The LCM of 8 and 4 is 8, so 8 input bits represented by a single byte can be Base16-encoded into two 4-bit values. Therefore, Base16 encoding has a $2/1 = 2\times$ bloat factor (the worst so far).

A **hexadecimal (hex) encoding**, which is likely very familiar to you, is a particular alphabet of Base16 encoding. Hex encoding uses 10 digits (0–9) and six uppercase or lowercase letters (A–F), which makes it case insensitive.

There are other Base16 encodings, such as **modified hexadecimal (ModHex) encoding**, which uses a different 16-character alphabet (**CBDEFGHIJKLNRTUV** instead of **0123456789ABCDEF**) for keyboard layout independence.

Despite its 2x bloat factor, hex encoding is probably the most popular binary encoding used. You can find it anywhere, from databases (blob display) to CSS stylesheets (hex colors). Hex benefits include padless conversion, case insensitivity, and to/from conversion logic that is so simple it can easily be done mentally. Microsoft's usage of hex encoding within the .NET Framework includes storing arbitrary-length binary sequences such as encryption and validation keys within .NET configuration files (a good encoding choice) and encoding ASP.NET **FormsAuthentication** cookies (a not-so-good choice, as we will discuss later).

Chapter 6 Text Encodings

Text encodings are defined by an ISO [Universal Character Set](#) (UCS), which assigns every character in every language a unique name and integer number called its **code point**. The current [Unicode](#) 8.0 standard (an extension of UCS) has 1,114,112 code points, of which 260,319 are assigned, and 853,793 are unassigned. These code point integers need to be somehow encoded into byte sequences, and there is more than one way to do it. Standardized code-points-to-bytes encodings are called **UCS Transformation Formats (UTF)**—you likely know them as UTF-32, UTF-16, UTF-8, and UTF-7. The minimal bit combination that can represent a single character under a particular UTF encoding is called a **code unit**.

UTF-32

The total number of million-plus Unicode code points can be represented in about 21 bits, which means that every code point can be comfortably encoded in 32 bits—that is exactly what UTF-32 does. **UTF-32** is a fixed-width encoding that uses four bytes per character. For example, “hello” is encoded in 20 bytes under UTF-32; it uses 4-byte code units and a single code unit per character. The main advantage of UTF-32 is that it is easily indexable (random access) and searchable due to its fixed-width nature, but it is very space inefficient. The .NET Framework provides `Encoding.UTF32` in `System.Text`.

UTF-16

UTF-16 encoding uses 2-byte code units, which makes it a variable-width encoding (some characters require two code units) because 16 bits of a single code unit cannot represent every Unicode code point. .NET strings and JavaScript strings are UTF-16-encoded for in-memory representation. The Windows OS APIs mostly use UTF-16 for string representation as well, which allows a clean interop from .NET.

The original Unicode proposal (back in the 90s) was grounded in the assumption that every code point can be encoded in 16 bits (Unicode originally called for a 16-bit fixed-width encoding). UTF-16 was created to deal with over-16-bit code points, which also made it variable width. There are also two flavors of UTF-16 due to [endianness](#)—UTF-16-BE and UTF-16-LE—and an optional [byte order mark \(BOM\)](#) to indicate which UTF-16 flavor is being used. The same two-flavor endianness/BOM concept applies to UTF-32 as well.

As long as the complexity of UTF-16 representation and processing is taken care of for you by the underlying framework (OS/.NET Framework/JavaScript engine, etc.), you have little to worry about. However, the moment you need to take UTF-16 strings out of the safety of your framework and into some other storage (for example, to write them into a file), UTF-16 complexity will impose challenges you could avoid with a different UTF encoding. The .NET Framework confusingly calls its UTF-16 implementation “Unicode” and provides `Encoding.Unicode` (UTF-16-LE) and `Encoding.BigEndianUnicode` (UTF-16-BE) in `System.Text`.

UTF-8

UTF-8 encoding uses single-byte code units, which makes it a variable-width encoding. UTF-8 code unit values are backward-compatible with ASCII codes, but it can take up to four 8-bit UTF-8 code units to represent some Unicode code points. You might be wondering why three 8-bit code units will not suffice, since $3 \times 8 = 24$ bits can cover ~21-bit Unicode code point space. The reason is that not all 8 bits within a variable-width code unit carry code point data—some of the bits are used for meta information, such as whether a code unit is a continuation of a multi-unit code point. UTF-8 does not suffer from endian ambiguities, and is also more compact than other encodings when used for ASCII-dominant text, but less compact for Asian-character-dominant text. The .NET Framework provides **Encoding.UTF8** in **System.Text**.

UTF-7

UTF-7 is not a standard Unicode encoding, and is not part of the Unicode standard. It was proposed as a way to encode Unicode with 128 ASCII characters more efficiently than UTF-8 with Base64 or with [QP](#). UTF-7 allows multiple encodings of the same source string, which can lead to various security vulnerabilities and attacks—do not use it, even though the .NET Framework supports it.

UTF comparison

Table 4: UTF comparison

	UTF-8	UTF-16	UTF-32
Endianness/BOM/cross-platform problems	no	yes	yes
Fixed-width benefits	no	no	yes
Fewest bytes per char	1	2	4
Most bytes per char	4	4	4
ASCII compatibility	yes	no	no
Lexicographical order	yes	no	yes
Bytes per ASCII char	1	2	4

	UTF-8	UTF-16	UTF-32
Bytes per Asian char	3	2	4
Bytes per East-European & Middle-Eastern char	2	2	4
Framework in-memory representation	not common	common	not common
Web & Internet friendly (machine-to-machine)	yes	no	no




Unless you have specific requirements for UTF-16 or UTF-32, we recommend UTF-8 for storage as your default choice. Reduction of complexity is more important for security than potential storage savings for Asian alphabets. Text processing is probably already taken care of by the framework you use, but if you need to do your own byte-level processing, you are likely to make fewer mistakes by converting to UTF-8 first.

Safe construction

The .NET Framework exposes public static **Encoding.UTF-*** properties in **System.Text**, which are thread-safe and are wildly used to convert a byte sequence into a particular UTF text representation. Here is a typical example:

Code Listing 12

```
var encoding = Encoding.ASCII;
for (int i = 0; i < 1000; ++i)
{
    byte[] bytes = Guid.NewGuid().ToArray();
    encoding.GetString(bytes).Dump();
}
```

Not every byte combination can represent a well-formed code unit, and not every well-formed code unit can represent a valid code point. When valid mapping is not possible, the .NET UTF [implementations](#) can use either a “fallback” strategy (map to a question box character, ) , a “replacement” strategy (some custom mapping logic), or an “exception” strategy (throw). The public static **Encoding.*** UTF properties all use the fallback strategy, which means that they never throw and silently produce  on mapping failures. The code in Code Listing 12 never throws, which is precisely the problem, because silent  substitutions can lead to various

security vulnerabilities and attack vectors. Microsoft's own internal **CryptoUtil** helper class uses a custom UTF instance constructed as follows:

Code Listing 13

```
public static readonly UTF8Encoding SecureUTF8Encoding = new
UTF8Encoding(
    encoderShouldEmitUTF8Identifier: false,
    throwOnInvalidBytes: true);
```

You can create similar UTF-16 or UTF-32 instances configured to throw on invalid bytes. Do what Microsoft does: always use explicitly-throwing UTF instances instead of convenient-but-dangerous **Encoding.*** ones. Despite Microsoft's obvious understanding of the dangers of fallback-based encodings, they make a strange choice of using a fallback-based UTF-8 encoding in their **Rfc2898DeriveBytes** implementation:

Code Listing 14

```
string s1 = "\ud8ab";
string s2 = "\ud8cd";
var salt = Guid.Empty.ToByteArray(); // fixed salt
var skm1 = new Rfc2898DeriveBytes(password: s1, salt: salt).GetBytes(16);
var skm2 = new Rfc2898DeriveBytes(password: s2, salt: salt).GetBytes(16);
```

s1 and **s2** in Code Listing 14 are two valid but distinct .NET strings (distinct hash codes and failing equality), which can be passed around just like any other .NET string. Both **s1** and **s2** also happen to be invalid UTF-16 encoding—they do not have a valid Unicode code point. .NET does not protect you from storing invalid UTF-16 strings inside string type, since the .NET string is just a sequence of **char** instances, and **char** is a 2-byte container that can be losslessly cast into [short](#).

Since **s1** and **s2** represent different passwords, you would expect the **skm1** and **skm2** byte arrays to be different as well, because that is the whole purpose of password-based key derivation—yet **skm1** and **skm2** are BAC-equal. If you needed one more reason not to use **Rfc2898DeriveBytes**, this is it. Our PBKDF2 implementation does not have this problem due to proper string serialization.

Serialization

.NET strings do not enforce Unicode validity of their contents, as we discussed previously. When we need to convert **string** to **byte[]** and then back to **string**, we usually try to leverage one of the Unicode encoding schemes to do the job. Unicode conversion forces us to make a choice: either we use a fallback strategy and replace what we cannot code-point-map

with question marks, or we throw an exception. The first approach is bad for all the reasons we already discussed. The second approach is good when you are trying to convert from `byte[]` into a properly encoded `string`—i.e. you want to enforce a valid Unicode bytes-to-text conversion. However, the second approach is bad when you face scenarios such as entropy extraction, key derivation, string preservation, and round-tripping, where you do not wish to throw.

We need an alternative approach for `string-to-byte[]` conversion that does not involve Unicode. Since every string is just a sequence of `chars`, and every `char` is losslessly represented as two bytes, we can represent every .NET `string` of length n as a `byte[]` of length $2n$. We should also prepend the byte-encoded length n to avoid length-extension vulnerabilities. A quick way of byte-encoding a 32-bit integer n is simply as four bytes. There is also a so-called “compressed” way of byte-encoding a 32-bit integer, implemented in .NET by the [Write7bitEncodedInt](#) method on `BinaryWriter`. `Write7bitEncodedInt` makes more sense to use for byte-encoding n , since most .NET string lengths are from a tiny subset of 32-bit integer space.

The downside of string serialization is that it always takes two bytes per char (UTF-8 could be more compact), but the upside is that serialization and deserialization never fail and never lose data (assuming you are deserializing what was previously serialized). This is the approach Microsoft uses internally with forms authentication for storing user-provided strings inside the authentication ticket, but there are no public APIs. We speculate that Microsoft was driven by a desire to match published test vectors for `Rfc2898DeriveBytes` implementation, which do not use multiple-of-2 byte arrays for passwords, which caused them to use UTF-8 encoding instead of string serialization. We provide working implementations of string serialization and deserialization to binary in the [Inferno](#) crypto library.

Chapter 7 Symmetric Encryption

Symmetric encryption uses the same secret key to hide and later reveal (encrypt and decrypt) a secret message. Given such a simple concept, most frameworks—.NET included—fail to provide symmetric encryption APIs that are simple, foolproof (i.e. secure and resist misconfiguration and abuse), and useable by engineers (not cryptographers) who have a requirement to encrypt and decrypt data based on a shared secret key.

What most frameworks do provide, however, is a wide assortment of cryptographic primitives, which could be used to construct a secure symmetric encryption API, but are in reality used to create broken, insecure implementations that provide a false sense of security (until your company's data makes the news). Microsoft is reinforcing this false confidence by [claiming](#) the following: “You do not need to be an expert in cryptography to use [System.Security.Cryptography] classes.” This is like saying “You don't need to know the driving rules because we've given you a car.” Standardization bodies such as NIST are part of the problem because they produce standards for cryptographers—not engineers—which the aforementioned frameworks implement and expose to the world.

Imagine NIST telling you that they have a great car for you, called AES. It is secure, approved, time-tested, and safe to use. However, you get the engine, brakes, transmission, steering, and suspension separately. Once you figure out how to properly put everything together—and manage not to screw up the coding mechanics—the car should be safe to drive. How does that sound? AES is technically not even that car, but the engine of that car: a block cipher. [AES](#) is supposed to be an “Advanced Encryption Standard,” but it fails at being one. Instead, AES is really a specification to use a block cipher called Rijndael in three defined “strength” modes (10, 12, or 14 transformation cycles). AES should have been ABCS, “Advanced Block Cipher Standard.”

Actual symmetric encryption is a cryptographic zoo of block ciphers, chaining modes, padding modes, initialization vectors, nonces, and authenticated encryption modes—and that's just the basic stuff. Are you confident¹ in your ability to put everything together correctly? How about the rest of your team?

Never fear, however, because NIST is at your service again with a friendly [SP-800-38a](#) AES manual: 22+ pages of recommendations on how to use AES block cipher properly. That's like asking your car dealer where the “start car” button is, and being handed a technical manual on “wiring direct-injection twin-scroll-turbo engines with dual-clutch transmissions”...in German. [Doppelkupplungsgetriebe](#)?

You do not have to be an engine mechanic in order to safely drive a car. You do not have to be a dietitian to eat healthy. You should not have to be a cryptographer to encrypt and decrypt messages with a shared secret key.

The goal of this section is to provide simple rules that lead to secure implementations and prevent you from creating yet another broken permutation to litter the charred minefield of

¹ Confidence: The feeling you experience before you fully understand the situation.

symmetric encryption. The next chapter, [“Authenticated Encryption,”](#) is where symmetric encryption will actually get secure, but we need an encryption primitive first.

AES

AES block cipher is the only block cipher you should be comfortable using if you are not a cryptographer. That leads to a simple first rule: always use AES, and forget whatever else comes with your framework. AES has three different “strength” modes, which typically do not have a direct setting, but instead are set by using a particular AES key length. The AES algorithm is fundamentally the same, but higher “strength” modes simply do more rounds of that algorithm.

Table 5: Comparison of AES strength levels

	AES-128	AES-192	AES-256
Block cipher length (same for all modes)	128 bits	128 bits	128 bits
Key length	128 bits	192 bits	256 bits
Rounds	10	12	14

AES-256 is stronger than AES-128 because it has four more rounds—not because it happens to use a longer key, since a full-entropy 128-bit key is already strong enough for all plausible purposes. The .NET Framework provides two AES implementations in **AesManaged** and **AesCryptoServiceProvider** (FIPS-approved) classes.

The managed AES class instance is about 170 times faster to create than its FIPS equivalent, with a break-even point at ~2.8k of processed data. If you always encrypt less than ~2.8k of data, managed AES might be a little faster; otherwise, use **AES-CSP** implementation (which is what we use most of the time for FIPS compliance). Which AES should you use: AES-128, AES-192, or AES-256? Their encryption and decryption speed is usually proportional to the number of rounds—you can estimate AES-256 to be about 40 percent slower than AES-128. However, you would only feel the full extent of algorithmic performance difference when AES-processing sufficiently long messages. On most modern systems (including modern mobile platforms) the performance difference between AES flavors is unlikely to be an issue.

Available AES implementation defaults are another important decision factor. Both Microsoft implementations default to AES-256, which means that if you want to use AES-128 or AES-192, you will constantly have to fight the defaults. It makes more sense to just accept Microsoft’s default choice. NASA lost a \$125 million Mars orbiter due to metric-imperial mismatch—do not think that you can always remember to change the defaults.

If you’re still unconvinced that AES-256 is the way to go, here is a quote from [LibTomCrypt](#) (C-language crypto library):

*Ideally, your application should be making at least 256-bit keys. This is not because you are to be paranoid. It is because if your PRNG has a bias of any sort, the more bits the better. For example, if you have $\Pr[X = 1] = \frac{1}{2} \pm \text{bias}$, where $|\text{bias}| > 0$ then the total amount of entropy in N bits is $N * (-\log_2(\frac{1}{2} + |\text{bias}|))$. So if bias were 0.25 (a severe bias), a 256-bit key would have about [106](#) bits of entropy whereas a 128-bit key would have only 53 bits of entropy.*

Here is another quote by distinguished cryptography expert [Daniel J. Bernstein](#):

But NIST is foolishly continuing to recommend AES-128 keys, even though a 2^{80} attack will break somebody's AES-128 key out of a batch of 2^{48} keys.



Tip: Just use AES-256.

Key

AES implementations have a public `.Key` property, which can be used to get or set the `byte[]`-typed secret key. That public property is backed by a private container, which is initially null. The `.Key` getter will generate a new cryptographically strong key when the private key container is null, or will simply return an existing key when the key container is not null. This, effectively, is on-demand key generation, which speeds up AES instance construction. The generation or re-generation of the key can be explicitly triggered with a `.GenerateKey()` method. You can also generate the key externally with a cryptographically strong RNG and assign it to the `.Key` property directly. The only trick with externally generated AES keys is making sure they have the right size. If you adopt AES-256 for all your needs, external key generation becomes very simple—just get 32 bytes from a CSP-based RNG. We should also re-emphasize the obvious point that secret keys are *secret*. Do not include secret keys with the message. Do not hash, HMAC, XOR, salt, or deep-fry them to make any part of them non-secret, or you will fail. Keep secret keys and anything derived from them secret.

Cipher mode

Block ciphers such as AES operate on a single block, which, in the case of AES, is 16 bytes long. [Cipher modes](#) are employed to make block ciphers process arbitrary-length messages that may not fit into a single block. The only AES cipher mode you should be comfortable coding yourself is CBC. CBC is the only cipher mode supported by both managed and FIPS AES implementations in the .NET Framework, and is also the default mode. Accept that default.

Padding mode

Block ciphers such as AES operate on full blocks, yet not every message will be exactly of block-size-multiple length. The last block can often be incomplete, and various [padding modes](#) can be used to make the last incomplete block whole. Some cipher modes do not require padding, but many—CBC included—do. The important thing for you to remember is that it does

not matter which padding mode you choose with CBC, because they all function as reversible padding, and are all insecure (at least the ones available in the .NET Framework). We will address this insecurity of padding modes with authenticated encryption, but we are not there yet. Since it does not matter which padding mode is used, you might as well go with the default one, which in .NET AES implementations happens to be PKCS7.

Initialization vector (IV)

Initialization vector (IV) is an additional non-secret block of entropy used by block ciphers such as AES to prevent vulnerabilities (such as ciphertext patterns) due to secret key reuse and plaintext reuse, both of which are very common and expected. We typically expect to be able to use the same secret key to encrypt more than one message, and we typically expect to be able to encrypt the same message at different points in time without revealing the “sameness” of the messages. A block cipher IV is one block long (16 bytes for AES). Different cipher modes place different requirements on the contents of an IV. Fortunately, we only need to know the IV requirements for CBC, since that’s the cipher mode we are comfortable coding ourselves.



Tip: *CBC IV must be unpredictable for every execution of the encryption process.*

“Unpredictable” here means “generated by CSP-based RNG.” Also note the “every execution” part: if you deviate from this CBC IV requirement in any way, you fail. If you try to be “smart” by hashing, HMACing, HKDFing, or deriving it somehow—just to avoid sending the IV along with the encrypted message—you fail. If you forget the “every execution” requirement and reuse the IV—you fail. Given such a simple and easy-to-implement CBC IV requirement, it is amazing how many implementations get it wrong. A Microsoft MVP (Most Valuable Professional) for “Developer Security” got it wrong.

AES implementations have an **.IV** property, which works similarly to the **.Key** property, and also does “on-demand” CSP-based random IV generation. The generation or re-generation of the IV can be explicitly triggered with a **.GenerateIV()** method. You can also generate the IV externally with a CSP-based RNG and assign it to the **.IV** property directly. If you take the external approach, remember that the AES IV size is always 16 bytes. Once you have performed an AES encrypt operation with a freshly generated random IV, you need to include that IV (which is not a secret) with the ciphertext to make the decryption process possible later, and you should never reuse that IV for any other encryption operation.

AES in counter mode (CTR)

While the CBC mode is the safest, misuse-resistant, built-in cipher operation mode in .NET, it is not the most secure or convenient mode for advanced usage scenarios, such as designing more complex crypto constructs that use variable-length data encryption as a building block. Counter (**CTR**) mode is another hugely popular mode that is technically superior to CBC mode (but not as misuse resistant), and is better suited as a data encryption component in other, more complex crypto schemes. [Phillip Rogaway](#), a distinguished cryptography expert, said the following about CTR mode:

I am unable to think of any cryptographic design problem where, absent major legacy considerations, any of [CBC and other classic modes] would represent the mode of choice...I regard CTR as easily the best choice among the set of the confidentiality modes (meaning the set of modes aiming only for message privacy, as classically understood). It has unsurpassed performance characteristics and provable-security guarantees that are at least as good as any of the [other classic] modes with respect to classical notions of privacy. The simplicity, efficiency, and obvious correctness of CTR make it a mandatory member in any modern portfolio of [secure] schemes.

Since you should not be designing anything crypto-related, do not try to implement CTR yourself, and stick with the built-in CBC. However, a decent .NET implementation of CTR mode is available in the [Inferno](#) crypto library (not a valid reason to abandon CBC).

Chapter 8 Authenticated Encryption

It turns out that doing symmetric encryption (in any mode) without authentication leads to many practically exploitable vulnerabilities, which are serious enough to have the following rule:



Tip: Symmetric encryption must be authenticated encryption (AE).

There are some cases where symmetric encryption alone might be sufficient, but you have to be a cryptography expert to recognize and analyse them, which you are not. Always use AE for symmetric encryption. There is yet another cryptographic zoo of various AE [modes](#). All these modes are single-key—the same secret key that is used for encryption is also used to provide authentication, which is nice. Unfortunately, you should stay away from all these single-key AE modes, for two reasons.

One reason is that while two of the NIST-approved AE modes are implemented by the Microsoft CNG API, none of them are available in the .NET Framework (as of .NET 4.6.1). The other reason is that both of these CNG-implemented AE modes are counter-based, meaning you are responsible for ensuring that the counter never repeats. Microsoft's own ASP.NET Security Team contemplated which AE approach to take with ASP.NET 4.5, and decided not to use the single-key AE modes (documented on their [blog](#)).

The alternative to a single-key AE mode is a dual-key AE mode, which adds the extra complexity of using two separate keys (one for encryption and one for authentication) instead of a single key, and is also not as fast as some single-key AE modes. On the positive side, however, a dual-key AE mode is acceptably fast (fast enough), proven to be secure, and is easy to implement. This dual-key AE mode standard is **encrypt-then-MAC (EtM)**, [ISO-19772](#).

The EtM encryption process is fairly simple:

1. **AES**-encrypt plaintext **P** with a secret key **K_e** and a freshly-generated random **IV** to obtain ciphertext **C₁**.
2. Append the ciphertext **C₁** to the **IV** to obtain **C₂ = IV + C₁**.
3. Calculate **MAC = HMAC(K_m, C₂)** where **K_m** is a different secret key independent of **K_e**.
4. Append **MAC** to **C₂** and return **C₃ = C₂ + MAC**.

The EtM decryption process is also simple:

1. If input **C** length is less than (expected **MAC** length + expected **AES IV** length), abort.
2. Read **MAC_{expected}** = last-**MAC**-size bytes of **C**.
3. Calculate **MAC_{actual} = HMAC(K_m, C-without-last-MAC-size bytes)**.
4. If **BAC(MAC_{expected}, MAC_{actual})** is false, abort.

5. Set **IV** = (take-**IV**-size bytes from start of **C**). Set **C₂** = (bytes of **C** between **IV** and **MAC**).
6. **AES**-decrypt **C₂** with **IV** and **K_e** to obtain plaintext **P**. Return **P**.

The ISO-19772 specification of EtM only authenticates the ciphertext C_1 and does not authenticate the IV. In the ISO EtM encryption spec, C_2 is equal to C_1 and not to $IV + C_1$. We believe this to be a specification flaw. ISO-19772 only states the obvious: “...[IV] shall be distinct for every message to be protected during the lifetime of a key, and must be made available to the recipient of the message.” Just follow the EtM process shown previously.

EtM Key derivation

EtM uses two separate keys—encryption key K_e and MAC key K_m —which need to come from somewhere. However, we still want to avoid the complexity of dual-key management, and instead wish to use a single, secret “master key” from which we can securely derive K_e and K_m . This calls for HKDF or SP800_108_Ctr.

Since you often cannot guarantee the non-reuse of the master key (SKM), the EtM encrypt and decrypt methods should take an optional salt, which will be used by HKDF to generate salted flavors of K_e and K_m . If the master key already has non-reuse safeguards (e.g., randomly generated or salted PBKDF2-derived), then the HKDF salting is not necessary.

Primitive choices

Let’s summarize all primitive choices for a solid EtM implementation:

Table 6: Summary of good EtM choices

	Our Choice	Reasoning
AES flavor	AES-256	See “ AES ” section
MAC flavor	HMAC-SHA-512 or HMAC-SHA-384	See “ Cryptographic hashes ” and “ HMAC ” sections
KDF flavor	HKDF (using above MAC)	Fast max-entropy extraction from master key
MAC length	128 bits (truncated MAC)	Take 16 bytes of HMAC-SHA-384

Unlike hashes, n -bit HMAC construction is not susceptible to [birthday attacks](#) and provides n bits of security. Using more than 128 bits of HMAC-SHA-512 or HMAC-SHA-384 for a MAC digest is overkill because it increases the EtM ciphertext size with little security gain. You could

use more than 128 bits for a MAC if you have other requirements, as long as you understand that there is no security-motivated argument to do so. Complete EtM implementation is available in the [Inferno](#) crypto library.

Length leaks

Most encryption modes leak some information about the plaintext length, since longer ciphertext typically implies longer plaintext. Padding modes contribute to this problem by adding one extra padding block when the plaintext length is a multiple of the block size. A 16-byte plaintext would get padded to 32 bytes with an extra 16-byte padding block during AES-CBC encryption, while a 15-byte plaintext would get padded to 16 bytes only. If plaintext length confidentiality is required, you could (reversibly) pad all plaintext to the same length prior to encryption. This is often impractical for a variety of reasons (for example, knowing all possible plaintext lengths up front), so it is best to avoid dependency on plaintext length confidentiality.

Common mistakes

Here is a non-exhaustive list of common symmetric encryption mistakes you should learn to recognize:

- Not using an unpredictable IV in CBC mode (IV is calculated somehow, or reused)
- Using symmetric encryption without authenticated encryption (i.e. MAC is not included)
- Using authenticated encryption but forgetting to include IV as part of MAC
- Not using two independent keys for EtM (same key or dependent keys)
- Using a non-cryptographically-strong RNG (for example, using `System.Random()` instead of CSP-based RNG)
- Improperly deriving master key from SKM (for example, hashing a user-provided password)
- Not using salt (correctly, at all, or using too much—for example, more than 128 bits)
- Using silent fallback with UTF encodings or using UTF-based encodings instead of string serialization
- Not using constant-time byte array equality comparisons
- Using obsolete .NET crypto primitives still available in .NET (MD5, RC2, DES, etc.)

Chapter 9 Asymmetric Cryptography

[Asymmetric cryptography](#) relies on keys made of two parts: a private (secret) part and a public (non-secret) part. A common misconception is that asymmetric cryptography is the same thing as asymmetric encryption. In fact, asymmetric cryptography has several distinct applications:

- **Encryption:** encrypting with the public key, decrypting with the private key.
- Digital **signatures:** signing with the private key, verifying with the public key.
- Shared secret **key agreement** over insecure channel.

Encryption is for ensuring confidentiality. Signatures are for ensuring authenticity, integrity, and non-repudiation. Some asymmetric schemes offer signature generation only, while others can do both signatures and encryption. .NET 4.5 provides the following asymmetric implementations, all of which are FIPS-compliant:

Table 7: Asymmetric primitives available in .NET

Asymmetric class	API	Signature	Encryption	Key agreement	Hash
RSCryptoServiceProvider	CAPI	yes	yes	yes (via encrypt)	enc: SHA-1 sign: any
RSACng	CNG	yes	yes	yes (via encrypt)	enc: SHA-2 sign: any
DSACryptoServiceProvider	CAPI	yes	no	no	SHA-1 only
ECDsaCng	CNG	yes	no	no	SHA-1, SHA-2
ECDiffieHellmanCng	CNG	not directly	yes (via key)	yes	SHA-1, SHA-2

This quick comparison is by no means comprehensive—there are many other factors and features that could be considered when assessing suitability for a particular scenario. Our pragmatic engineering perspective, however, compels us to focus on a single asymmetric scheme that can be used to address most of the common asymmetric-crypto scenarios, and describe how to use it correctly. One such Swiss Army knife asymmetric scheme is RSA. RSA is easy to use, widely available, and can do signatures, encryption, and key agreement.

RSA key management

The default RSA-CSP constructor creates 1024-bit keys, which no longer offer sufficient security, and are in fact disallowed by NIST [SP800-131A](#). NIST allows ≥ 2048 -bit keys, and there is little security to be gained from going higher. NIST [SP800-57 Part1-Rev3](#) recommends 2048-bit RSA keys for ~ 112 bits of security, and 3072-bit RSA keys for ~ 128 bits of security. However, in August of 2015, the NSA made a poorly explained change to their [Suite B](#) recommendations and requirements. The NSA not only inexplicably added RSA to Suite B, but also required a minimum 3072-bit key size. Since the cryptographic community is [not buying](#) the NSA explanations for the change, it is probably prudent to follow the higher minimum (just in case).

We therefore recommend a 3072-bit RSA key size. [Another](#) RSA-CSP constructor explicitly takes the RSA key size as argument, which is the constructor you should use. RSA-CSP constructors do not trigger the expensive RSA key generation, and thus are fast. RSA key generation is triggered only during key-requiring operations (encryption, signing, key export) if no key was provided or imported prior.

RSA key-pair can be exported with `ExportParameters/ImportParameters` methods to get the `RSAPParameters` structure. `RSAPParameters` contains eight public `byte[]` fields, two of which describe the public key (`Exponent` and `Modulus`). Microsoft's RSA-CSP implementation uses a constant public exponent, [65,537](#), which translates into the `Exponent` field always being three bytes `{01, 00, 01}` (big-endian storage). This effectively means that the RSA-CSP public key can be represented by the `RSAPParameters.Modulus` byte array alone, which has the same length as the key length in bytes. For a 3072-bit RSA key-pair, the modulus is 384 bytes long.

Another way to export the key-pair is with `ExportCspBlob/ImportCspBlob` methods to get the serialized blob as `byte[]`. `ExportCspBlob(includePrivateParameters:false)` is a tempting alternative to using the `Modulus` field directly after `ExportParameters(includePrivateParameters:false)`. The public-key blob produced by `ExportCspBlob(false)` will be ~ 20 bytes longer than the `Modulus` byte array. Another reason for our slight preference toward public-key extraction from `Modulus` is that it is easier to make the mistake of accidentally blob-exporting the entire key instead of just its public components. Using `Modulus` directly makes it harder to make the mistake of using the entire key when only the public key is intended to be used.

`ExportCspBlob(true)` exports the entire key with a close-to-optimal blob size. You are unlikely to see any major savings from a custom serialization of all fields in `RSAPParameters` structure versus `ExportCspBlob(true)` output.

The default `RSACng` constructor creates 2048-bit keys (an improvement over RSA-CSP default). However, we reiterate our recommendation to use a minimum 3072-bit key size instead:

Code Listing 15

```
byte[] privKey;  
CngKey cngPrivKey;
```



```

byte[] pubKey;
CngKey cngPubKey;

byte[] signature;
byte[] data = new byte[] { 1, 2, 3 }; // some data

// generate RSA keys (private and public)
using (var rsa = new RSACng(keySize: 3072))
{
    privKey = rsa.Key.Export(CngKeyBlobFormat.GenericPrivateBlob);
    pubKey = rsa.Key.Export(CngKeyBlobFormat.GenericPublicBlob);
}

cngPrivKey = CngKey.Import(privKey, CngKeyBlobFormat.GenericPrivateBlob);
cngPubKey = CngKey.Import(pubKey, CngKeyBlobFormat.GenericPublicBlob);

// generate RSA signature with private key
using (var rsa = new RSACng(cngPrivKey))
{
    signature = rsa.SignData(
        data: data,
        hashAlgorithm: HashAlgorithmName.SHA384,
        padding: RSASignaturePadding.Pss);
}

// verify RSA signature with public key
using (var rsa = new RSACng(cngPubKey))

```

```
{
    rsa.VerifyData(
        data: data,
        signature: signature,
        hashAlgorithm: HashAlgorithmName.SHA384,
        padding: RSASignaturePadding.Pss).Dump();
}
```

RSA signatures

RSA-CSP provides **SignData/VerifyData** and **SignHash/VerifyHash** method pairs for signing. The ***Data** methods will internally hash the data prior to signing. The ***Hash** methods accept an externally calculated hash and an [OID](#) string matching the hash algorithm to be used.

Code Listing 16

```
string hashOID = CryptoConfig.MapNameToOID("SHA384");
// OID: 2.16.840.1.101.3.4.2.2

using (var rsa = new RSACryptoServiceProvider(3072))
using (var hashAlg = new SHA384Cng())
{
    var data = new byte[] { 1, 2, 3, 4 };
    var hash = hashAlg.ComputeHash(data);
    var signature1 = rsa.SignData(data, hashAlg);
    var signature2 = rsa.SignHash(hash, hashOID);
    Enumerable.SequenceEqual(signature1, signature2).Dump(); // True
}
```

The resulting RSA signature size is equal to the RSA key bit-length. The ***Data** methods call the ***Hash** methods internally, so the ***Hash** methods might be slightly faster. Another reason to prefer the ***Hash** methods is when the data hash is used in more than one place, and there is no need to calculate the data hash more than once. The ***Data** methods make no attempts to dispose or otherwise clean up the **IDisposable HashAlgorithm** object instance, which makes sense considering that it could be provided externally (as in our previous example). However, the hash instance can also be constructed internally if the passed-in object is not a **HashAlgorithm** instance but a string like "SHA384", an OID string, or a type, like

typeof(SHA384Cng). Even under these internal hash construction scenarios, no cleanup is done by the ***Data** methods. Since most fast hash implementations are native and use unmanaged memory, hoping for the garbage collector to eventually kick in and do its magic might not be sufficient, especially when classes like **SHA384Cng** with an internal **BCryptHashAlgorithm** nucleus have no [finalizers](#) for GC to trigger.

There might be nothing wrong with the ***Data** method memory cleanup in all scenarios. However, when unmanaged **IDisposable** objects are not disposed inside crypto code, it makes us very uneasy. We prefer to err on the side of caution and recommend that you do not use the ***Data** methods with a second parameter being anything other than an externally provided **HashAlgorithm** instance whose lifetime and cleanup you can control.

All RSA-CSP signature methods use an older [PKCS #1 v1.5](#) padding scheme, which has some weaknesses. A more modern RSA signature-padding scheme is **RSA-PSS**, which is implemented in newer (CNG) Windows OS crypto modules, but is not available in .NET 4.5. In 2013, we wrote:

Perhaps the next version of .NET would provide something like “RSACng” which would be more cryptographically up-to-date.

In July 2015, Microsoft announced [.NET 4.6](#), which added the **RSACng** class with RSA-PSS signing padding and **OAEP** encryption padding using the SHA-2 hash family. We will cover **RSACng** later, but if you are stuck with RSA-CSP for signatures, we recommend SHA-384 (**SHA384Cng**) on 64-bit .NET platforms.

RSA key exchange

RSA secret key exchange/agreement is often called “RSA encryption” because it technically is. RSA encryption can only process a short message of length equal to $[\text{RSA-key-size-in-bytes}] - N$, where N is a number of bytes consumed by and dependent on a particular RSA encryption padding scheme used. There are two RSA encryption padding schemes—PKCS #1 v1.5 and OAEP—both of which are available in RSA-CSP encryption APIs.

OAEP is the [more secure](#) padding scheme. OAEP padding is to RSA encryption what RSA-PSS padding is to RSA signatures. You should always use OAEP with RSA encryption.

Under OAEP padding, N is $2 \times [\text{hash-size-in-bytes}] + 2$. Example: Using SHA-384, $N = 2 \times 48 + 2 = 98$ bytes, which means that a to-be-encrypted message under a 384-byte (3072-bit) RSA key with SHA-384 OAEP padding can be at most $384 - 98 = 286$ bytes long. We could instead consider using a slower **SHA-256 OAEP** padding to increase maximum message size to 318 bytes ($384 - 2 \times 32 - 2$). This message length restriction makes classic RSA encryption mostly suitable for key exchange and agreement, since symmetric cryptographic keys are typically short enough.

RSA-CSP encryption implementation has no hash function agility and always uses SHA-1 (20 bytes) internally, which means that the maximum message length under RSA-CSP encryption with OAEP padding is $[\text{RSA-key-size-in-bytes}] - 2 \times 20 - 2$. For example, the RSA-CSP-encrypted message under a 384-byte key can be at most $384 - 42 = 342$ bytes long. If you need to RSA-encrypt more than 342 bytes, you can increase the RSA key size up to the

maximum RSA-CSP-supported 16,384 bytes (which can take minutes to generate). Note that RSA-OAEP encryption is non-deterministic and produces different ciphertext every time. The OAEP padding scheme ensures integrity; altered ciphertext will fail to decrypt. The hash function is only used within the OAEP padding scheme, and is not part of the raw RSA encryption. The use of SHA-1 in RSA-CSP OAEP implementation has no known weaknesses and is perfectly adequate, even by NIST standards (SHA-1 is not used for signatures here), although hash function agility would have been nice.

Code Listing 17

```
/* RSA-CSP */
int keyBits = 3072;

int maxDataBytes = (keyBits / 8) - (20 * 2) - 2;
//maxDataBytes += 1; // go beyond max size
maxDataBytes.Dump("maxDataBytes");

using (var rsa = new RSACryptoServiceProvider(keyBits))
{
    var data = new byte[maxDataBytes];
    new RNGCryptoServiceProvider().GetBytes(data);
    var cipher = rsa.Encrypt(data, fOAEP: true);
    "Encryption works.".Dump();
    var data2 = rsa.Decrypt(cipher, fOAEP: true);
    "Decryption works.".Dump();
    Enumerable.SequenceEqual(data, data2).Dump(); // should be true
}
```

```

/* RSACng */
int keyBits = 3072;

int maxDataBytes = (keyBits / 8) - (48 * 2) - 2; // SHA-384 is 48 bytes
//maxDataBytes += 1; // go beyond max size
maxDataBytes.Dump("maxDataBytes");

using (var rsa = new RSACng(keyBits))
{
    var data = new byte[maxDataBytes];
    new RNGCryptoServiceProvider().GetBytes(data);
    var padding = RSAEncryptionPadding.OaepSHA384;
    var cipher = rsa.Encrypt(data, padding);
    "Encryption works.".Dump();
    var data2 = rsa.Decrypt(cipher, padding);
    "Decryption works.".Dump();
    Enumerable.SequenceEqual(data, data2).Dump(); // should be true
}

```

One issue to be aware of with RSA-CSP **.Encrypt()** is that it will accept a message of length **maxDataBytes + 1** and produce a ciphertext without complaints, but decrypting such ciphertext with RSA-CSP **.Decrypt()** will throw an exception. We think this is just bad bounds-checking on Microsoft's part, so make sure you calculate your own valid message sizes, or at least verify message round-tripping. **RSACng**, on the other hand, will throw an unhelpful "The parameter is incorrect" **CryptographicException** on encrypting data larger than **maxDataBytes**. While **RSACng** will at least not produce flawed ciphertexts (unlike RSA-CSP), it does not produce a meaningful error message that explains what is going on. Neither RSA implementation provides a helper to calculate the maximum allowed message size for encryption. For some reason, Microsoft expects developers to be aware of internal details and do their own math. MSDN does not cover these important nuances, either.

RSA encryption

Classic RSA-OAEP encryption works on short-length messages only, so in many contexts, "RSA encryption" actually means "RSA key exchange," which we have just covered. A more useful "encryption" context is one that is similar to "AES encryption" context, where the message length is practically unlimited. While block ciphers such as AES have operation modes (CBC, CTR, etc.) to securely process variable-length plaintext, no such modes exist for asymmetric encryption, which is the main reason to consider a hybrid symmetric or asymmetric approach

(asymmetric encryption is often claimed to be much slower than symmetric encryption, but that is not the main reason for hybrid encryption popularity).

Let's assume that we have four parties, A, B, C, and D, with established RSA keys, and each party is in possession of everyone's authentic public keys. Consider a scenario of A wishing to send a confidential message to B and C, but not to D (the message for B and C is the same).

Hybrid RSA Encryption, Approach 1:

1. **A** AES-encrypts message **M** under random symmetric key **SymK**:
 - a. $E_1 = \text{AES}_{\text{SymK}}[\mathbf{M}]$
2. **A** RSA-encrypts **SymK** for **B** and for **C**:
 - a. $E_2 = \text{RSA-ENC}_{\text{PubK}_B}[\text{SymK}]$
 - b. $E_3 = \text{RSA-ENC}_{\text{PubK}_C}[\text{SymK}]$
3. **A** sends $E = E_1 + E_2 + E_3$ to **B** and **C**. **D** is able to get a copy as well.

Pros:

- Only **B** and **C** (but not **D**) are able to decrypt the message.

Cons:

- "Someone" sent the message—nothing identifies **A** as the message author.
- Message authenticity is inferred indirectly via **A**'s RSA encryption of a random symmetric key. **B** can decrypt **SymK** and **M**, change **M** to **M'**, re-encrypt under the same **SymK**, and send to **C**. **C** either gets both **M** and **M'** and has no basis for determining which one is authentic, or **C** gets only **M'** and trusts its authenticity, which is even worse.
- Symmetric encryption is used without authentication, which has all kinds of vulnerabilities.

Hybrid RSA Encryption, Approach 2:

1. **A** RSA-signs (hash of) message **M** with **A**'s private RSA key:
 - a. $E_1 = \text{RSA-SIGN}_{\text{PriK}_A}[\mathbf{M}]$
2. **A** EtM-encrypts message **M** and its signature under random symmetric master key **SymK**:
 - a. $E_2 = \text{EtM-AES}_{\text{SymK}}[\mathbf{M} + E_1]$
3. **A** RSA-encrypts **SymK** for **B** and for **C** (same as in Approach 1):
 - a. $E_3 = \text{RSA-ENC}_{\text{PubK}_B}[\text{SymK}]$

$$b. E_4 = \text{RSA-ENC}_{\text{PubK}_C}[\text{SymK}]$$

4. **A** sends $E = E_2 + E_3 + E_4$ to **B** and **C**. **D** is able to get a copy as well.

Pros:

- Only **B** and **C** (but not **D**) are able to decrypt the message. Message authenticity and author are inferred directly from **A**'s RSA signature of the actual message. Only authenticated symmetric encryption is used.

Cons:

- How should **B** and **C** infer that the RSA signature E_1 should be validated with **PubK_A**? We could modify E_2 to append either the full **PubK_A** (thus no longer requiring recipients to have it), or a hash of **PubK_A** to conserve space.
- If **PubK_A** or its hash is appended to E_2 , all observers will know that **A** sent message E .

Hybrid RSA Encryption, Approach 3:

1. **A** RSA-signs (hash of) message **M** with **A**'s private **RSA** key (same as Approach 2):

$$a. E_1 = \text{RSA-SIGN}_{\text{PriK}_A}[\text{M}]$$

2. **A** EtM-encrypts message **M**, its signature, and **PubK_A** under random symmetric master key **SymK**:

$$a. E_2 = \text{EtM-AES}_{\text{SymK}}[\text{M} + E_1 + \text{PubK}_A]$$

3. **A** RSA-encrypts **SymK** for **B** and for **C** (same as in Approach 1):

$$a. E_3 = \text{RSA-ENC}_{\text{PubK}_B}[\text{SymK}]$$

$$b. E_4 = \text{RSA-ENC}_{\text{PubK}_C}[\text{SymK}]$$

4. **A** sends $E = E_2 + E_3 + E_4$ to **B** and **C**. **D** is able to get a copy as well.

Pros:

- Only **B** and **C** (but not **D**) are able to decrypt the message. Message authenticity and author are inferred directly from **A**'s RSA signature of the actual message. Only authenticated symmetric encryption is used. **A**'s public key is now encrypted within the payload, so only intended recipients are able to infer and validate the message's author.

Cons:

- No major conceptual flaws (maybe), but practical implementation failures are likely.

The goal of this hybrid approach exercise is not to lead you to the "one correct algorithm" you should implement, but to emphasize that if you ever find yourself or someone on your team

playing these build-a-crypto-protocol games or, even worse, actually implementing something like that, you are doing it wrong.

There are at least three existing specifications with corresponding implementations that already leverage hybrid encryption concepts, and chances are they all do a better job than what you or your team is capable of:

- **SSL/TLS**
 - .NET Framework has decent [TLS support](#), including TLS 1.2.
- **Cryptographic Message Syntax (CMS)**
 - CMS is a superset of PKCS #7 standards, and has [good .NET support](#).
- **OpenPGP** standard
 - Not supported by .NET, but commercial .NET implementations exist (and require your trust).

TLS and **CMS** are based on a centralized trust model (centralized, allegedly trustworthy root CAs and trust chains), while **OpenPGP** is based on a decentralized trust model ([web of trust](#)). While neither model is perfect, the existing implementations of mechanics for container and payload cryptography are likely to be much better than a custom implementation.

Diffie-Hellman key agreement

Diffie-Hellman (DH) is a key agreement protocol that enables two parties **A** and **B** to establish a secret key over an open channel in such a way that both parties end up with the same key without divulging it to anyone listening in on the conversation.

RSA-encrypt protocol can be used for key agreement as well, where party A picks a key, encrypts it with B's public RSA key, and sends it to B. One interesting advantage of DH over RSA-encrypt for the purposes of key agreement is that DH key agreement implicitly binds both parties' public DH keys (sender and receiver), while RSA-encrypt key agreement binds only to the receiver's public RSA key.

Modern versions of the DH protocol are done over [elliptic curves](#), and are called **ECDH**. It does not matter what elliptic curves are, as long as you know that ECDH is a modern DH protocol. There are different types of elliptic curves that can be used with ECDH, which roughly correspond to different ECDH strengths (similar to AES 128/192/256 strengths). NIST has standardized three different curves: **P256**, **P384**, and **P521**, all of which are available in .NET, and roughly correspond to 128-bit, 192-bit, and 256-bit security levels. NIST [Suite B](#) recommends P384, and so do we—not because 128-bit security of P256 is an issue by itself, but because P384 allows for a healthy security margin in case PRNGs have a bias reducing ECDH key entropy. The shared ECDH key is often used as a master key to seed additional symmetric keys (for AES and MAC, for example), which would need to be at 128-bit security level, and using P384 rather than P256 to generate a master key at a higher (192-bit) security level is a conservative approach.

While NIST curves have withstood decades of public scrutiny, they no longer reflect the state of the art in elliptic curve design, which is an active cryptographic research area with lots of recent developments, newer elliptic curve designs, and standardization attempts.

Perfect forward secrecy

The RSA encryption and DH key agreement schemes we've discussed can provide security of the symmetric "session key" as long as recipients' private keys are not compromised. We might be reasonably assured that these private keys are not compromised today, but it is very difficult to extend that assurance into the future. An encrypted communication can be recorded by an adversary that plans on obtaining the private keys in the future—either through advances in technology, or via some other future [weakness](#) in private key safeguarding.

An important consequence of such private key compromise is that all past communication sessions that utilized these private keys would also become compromised—not just one specific communication session.

[Perfect forward secrecy \(PFS\)](#) is an additional property of asymmetric cryptography schemes that prevents session keys from being compromised when long-term private keys are compromised. Our previous RSA examples used the RSA private key for two purposes: signature and session key decryption. A compromise of this key would thus enable an adversary to falsify signatures and decrypt communications. Message authenticity and confidentiality are both important, but confidentiality is supposed to last for a very long time (ideally forever), while signatures are typically a secondary concern compared to the confidentiality of the data itself. PFS addresses this problem by using two sets of asymmetric key-pairs: one long-term key-pair is used for signatures, while a different short-term (session-term) key-pair is used for symmetric session key agreement. This additional short-term key-pair used for symmetric key agreement is called an [ephemeral key](#), with its public component signed by the sender's long-term private key.

New ephemeral keys are generated for each session, and both communication sides try to forget the (private portion of) ephemeral keys after each session. This makes it more difficult to compromise security of an individual session, and also prevents extending a single session compromise to other sessions—thus providing PFS.

Both RSA and Diffie-Hellman can be used as ephemeral keys. However, RSA keys are expensive to generate, while DH key generation is typically very fast. Most standards, including TLS, use DH keys for PFS. TLS [cipher suites](#) that provide PFS have "DHE" or "EDH" in them (Diffie-Hellman Ephemeral). The FIPS-approved TLS cipher suites can be found in [RFC 6460](#).

Elliptic Curve Integrated Encryption Scheme (ECIES)

Since PFS is a nice property to have, a common scenario is to use an ephemeral ECDH key-pair to agree on a new per-session master key, and then use that master key to securely send arbitrary-size messages. This approach is conceptually similar to the non-PFS, hybrid RSA encryption we have covered earlier. Such ECDH-based arbitrary-size message encryption has been standardized as [ECIES](#) (also ANSI X-9.63). ECIES uses the following components:

- Key agreement for generating a secret shared by two parties
- Key derivation function for deriving additional symmetric keys
- Encryption primitive (ENC) for symmetric encryption
- Hash primitive used within KDF and MAC/HMAC

Sender **A** follows this sequence of steps to encrypt some plaintext for receiver **B**:

1. **A** creates a new ephemeral ECDH key-pair **T**.
2. **A** uses **T**-private and **B**-public keys to calculate a shared ECDH secret **SK**.
3. **A** uses a KDF to derive ENC-key and MAC-key from **SK**, and EtM-encrypts the plaintext into ciphertext **C**.
4. **A** sends [**T**-public, **C**] to **B**, and immediately forgets **T**-private and **SK**.

Receiver **B** follows this sequence of steps to decrypt [**T**-public, **C**] received from **A**:

1. **B** uses **T**-public and **B**-private keys to calculate a shared ECDH secret **SK**.
2. **B** uses a KDF to derive ENC-key and MAC-key from **K**, and tries to EtM-decrypt **C**.



Note: *ECIES KDF can also take optional parameters that can act as “salt” or “associated data.”*

A simple example of ECIES implementation can be found in the [Inferno](#) library documentation.

Key separation

The **key separation principle** mentioned in the “[Key derivation](#)” chapter applies to asymmetric key pairs as well. We have covered how to use the same RSA key pair to do both encryption and signatures. However, it might be prudent to use one RSA key-pair for encryption, and a separate RSA key-pair for signatures. The reason is that encryption and signature keys often have different lifetimes and different escrow requirements. The encryption key is often destined for a long lifetime, and is often required to be escrowed to ensure that vital data can be decrypted. The signature key, on the other hand, should never be shared or escrowed to ensure non-repudiation, and can often have a short lifetime with an established expiration process.

Having separate asymmetric key pairs for encryption and signing thus enables a better key management process. One example is [GPG](#), which generates separate asymmetric subkeys for encryption and signature purposes.

Key wrap

Many cryptographic protocols and standards talk about “key wrap”, so it is important to understand what key wrap is, and more importantly, why you are unlikely to ever need it.

Key wrap (KW) is an encryption mechanism that aims to provide privacy and integrity of the plaintext **without the use of nonces** (such as IVs or random bits). In other words, KW is a specialized form of **authenticated encryption (AE)** that does not require counters or a cryptographically secure random number generator.

The lack of nonce or random-bit dependency makes KW schemes a form of deterministic authenticated encryption—the same key KW-encrypting the same plaintext will always produce the same ciphertext.

The following reasons might make KW schemes preferable over other AE schemes:

- Some environments either do not support high-quality CSRNG or make CSRNG bits very difficult and costly to obtain.
- Some environments implement only one crypto primitive, such as AES. KW schemes such as [AES-KW](#) use only a single crypto primitive (AES), while many AE schemes are more complicated.

Most real uses of KW are in legacy applications and protocols, or in low-level/hardware scenarios where CSRNG is unavailable or costly. Since KW schemes are trying to ensure that every single bit of ciphertext depends on every single bit of plaintext, they are doing a lot of additional permutation work, and are thus very slow—many times slower than AES. They also usually get progressively slower (per-bit) with longer plaintexts.

Fortunately, Windows OS and the .NET environment have a good, fast, and cheap CSRNG, which negates any potential benefits a KW scheme could have had over a good nonce-misuse-resistant AE scheme (for example, [AEAD in the Inferno library](#)). Some people do not understand KW, and mistakenly believe that KW is somehow superior to AE/AEAD because it is called “key wrap” and thus is somehow better suited for encrypting keys.

Do not fall for the “key wrap” hype. If you have a good AEAD scheme and a fast CSRNG, you can certainly forget about KW. Apple uses KW in iOS and for various hardware-accelerated key protection, but that does not make KW worth considering if you build .NET solutions. AE/AEAD always works, even when a plaintext is a key.

Chapter 10 Two-Factor Authentication (2FA)

It is often desirable to supplement single-factor “what-you-know” credentials with additional “what-you-have” credentials to avoid security breaches when “what-you-know” credentials are compromised. The “what-you-know” credentials often suffer from mass compromises (for example, thousands of account passwords are leaked in a single security breach), while the “what-you-have” credentials are typically less vulnerable to mass compromises (which can still [happen](#)).

One-time passwords (OTP) are commonly employed to mimic the “what-you-have” factor due to their simplicity. We say “mimic” and not “provide” because not all OTP schemes are true “what-you-have” factors, but they are often good enough as long as their limitations are understood.

The key distinguishing feature of all OTP schemes when compared to the “what-you-know” schemes is limitation on or prevention of credential replayability. Some OTP schemes use symmetric cryptography with shared secrets known to both communicating parties. Some OTP schemes use asymmetric (public-key) cryptography with private key on the client only. Some OTP schemes are challenge-response-based, with the server issuing an unpredictable challenge to the client, and the client returning a signed challenge to the server.

HOTP

The **HMAC-Based One-Time Password scheme (HOTP)** is a widely adopted open standard for OTP authentication. HOTP requires a shared symmetric secret and is event-based, where the “event” is an integer counter value incremented on each OTP generation and maintained by both client and server.

TOTP

The **Time-Based One-Time Password scheme (TOTP)** is an extension of HOTP that uses current time instead of an incrementing counter to limit replayability. The main advantage of TOTP over HOTP is that TOTP passwords are short-lived, while HOTP passwords can potentially be valid for a long time. TOTP is also easier to re-sync and allows multiple clients to authenticate against the server without any additional server-side complexity.

Both HOTP and TOTP schemes are based on a “long-term” symmetric secret key shared by client and server, which results in several important security implications:

- The secret key can be compromised on either the client or the server (two attack points).
- The *prover* and *verifier* roles are not mutually exclusive—the server can impersonate the client.

This last point also implies that single-client-multiple-server HOTP/TOTP deployments would not work if all servers do not have full mutual trust.

TOTP requires both the client and the server to know the current time, which might be a challenge for clients that lack time-tracking capability (such as externally or intermittently powered devices like USB tokens). In such cases the current time is typically provided to the client by the server as a “challenge,” which the client combines with its secret key to produce the TOTP response. This is how [YubiKey](#) supports TOTP. One downside of sending challenges to the client is that the client typically has no way of authenticating these challenges as legitimate.

Remember that one of the fundamental goals of leveraging OTP schemes is to mimic the “what-you-have” factor. If we can easily trick the TOTP client into generating a valid response for *any* value of “time,” then physically possessing the TOTP client will no longer be necessary to have a valid TOTP response for some “future” time, which will negate the very purpose of using an OTP scheme like TOTP. This implies that challenge-based TOTP clients should either never leave their owners, or be immediately re-keyed after returning to their owners.

Powered mobile devices capable of timekeeping are ubiquitous, and are ideally suited for TOTP. Intermittently powered mobile devices without timekeeping ability can still keep a counter, and are better suited for HOTP.

U2F

Universal 2nd Factor (U2F) is a relatively recent challenge-response authentication standard that uses specialized hardware devices (such as USB or NFC tokens). The key improvement of U2F over other two-factor methods is phishing and [MITM](#) protection. Modern web technologies coupled with a bit of social engineering make it very difficult for a casual user to be able to spot and comprehend the difference between an authentic destination (like Gmail.com) and an impostor site that looks identical. However, the user agent (“the browser”) does know the difference, and can distinguish different destinations even when the user cannot.

The biggest hurdle for U2F’s adoption is that not all browsers have U2F support yet. However, despite its young age, U2F appears to be open enough, secure enough, and convenient enough to become the dominant OTP alternative.

Chapter 11 Web Security

ASP.NET security

Security compromises of web applications are becoming [more and more ubiquitous these days](#). Web applications present an easily accessible battleground to combine various security primitives into more complex security protocols and test these protocols in action via wide exposure enabled by the web. This would be great if it wasn't for one little thing: security properties [cannot be tested](#) in a functional way (e.g., "Did we include the security feature? Yes? Checkmark! Deploy to production!"). What typically gets tested in the end is the insecurity, when it is already too late (i.e. the production environment is compromised).

The .NET web development tool of choice is ASP.NET, which is a collective term representing numerous frameworks such as [Web Forms](#), [AJAX](#), [Web Pages](#), [MVC](#), [Web API](#), [SignalR](#), [WCF](#), and SOAP Web Services ([ASMX](#)), each of which is going through its own evolution and maturity cycles. There is a great deal of complexity in the ASP.NET portfolio of technologies, and complexity is the eternal nemesis of security.

Microsoft has strived to address this complexity with varying degrees of success by providing "ready-to-use" security components for common web-application-related requirements, such as session management, user authentication (identity management), user authorization, and credential storage. Many of these ASP.NET-provided security components are quite dated because they were designed for ASP.NET 1.x (2002–2003), or ASP.NET 2.0 (2005). The web has moved on, however. There are practically-exploitable vulnerabilities in the modern web that well-intentioned ASP.NET-provided security components were never designed to address, or address insufficiently. The road to the pit of failure is paved with good intentions. Microsoft has made these provided security components as easy to use as possible (some of them are enabled by default), and it is very easy for developers to get instant functional gratification by using them, but it is also what makes these components dangerous.

Microsoft has made commendable effort with ASP.NET 4.0, and then again with ASP.NET 4.5 to improve these widely used security components and incorporate additional protection measures without compromising backwards compatibility (the user base is enormous). We feel, however, that many ASP.NET-provided security components are way past their expiration date, and should be replaced with new approaches which are actually designed for the threats and vulnerabilities of the modern web. On the other hand, we want to be able to reuse as much of the existing tried-and-tested security functionality as possible, and thus would only consider custom implementations as a last resort.

Microsoft is fully aware of the need to modernize ASP.NET and bring it up to speed with the evolution of the modern Web. ASP.NET 5 was a major redesign of ASP.NET framework to free the internal architecture from all things "legacy" that were holding ASP.NET back. This ambitious effort is apparently radical enough to [kill](#) "ASP.NET 5" and instead rebrand the project as "ASP.NET Core". While the ASP.NET 4.x "king" is officially not a king anymore, and the new king is not mature yet (ASP.NET Core 2.0 is still missing crucial cryptographic capabilities), there will remain countless ASP.NET 4.x (and 2.x) line-of-business applications. These "legacy" applications will be with us for a very long time, and it is important to understand and correctly apply the security concepts behind them.

Session state

The ASP.NET [session state](#) component enables server-side storage of any serializable web-session-specific data, and is enabled by default for all ASP.NET applications according to MSDN. The client side keeps track of associated server-side session state via a unique session state identifier called [SessionID](#). SessionID is supposed to be a Base32-encoded, case-insensitive, CSP-RNG-generated 120-bit value (see the “[Base32](#)” section for more info).

In December of 2008 (three years after ASP.NET 2.0), a security researcher investigating the claimed 120-bit entropy of SessionID wrote a [paper](#) about it. The paper showed that the internal algorithm indeed generates 120 bits of entropy, but Base32 encoding implementation has a flaw, which reduces the encoded entropy from 120 bits to 96 bits in the worst case, with the expected average entropy actually hovering around 108 bits. The actual SessionID key space was therefore $2^{(120-108)} = 4096$ times smaller than claimed. The reduced key space is still large enough to be out of reach of any practical exploits, but it goes to show that Microsoft has had its share of implementation mistakes. The bug was in bit-shifting right on random integers, and not realizing that for negative integers, the incoming-from-left bits will be ones and not zeroes.

Neither the flaw, nor the security paper was publicly acknowledged by Microsoft (at least we can't find a public record of it on the web). The latest .NET in 2008 was .NET 3.5, which was running on 2.0 runtime. .NET 4.0 was released in April 2010, which introduced a new 4.0 runtime. .NET [GAC](#) now has two **System.Web.dll** assemblies in it: version 2.0 and version 4.0. Here are the version 2.0 and version 4.0 algorithms for **Encode()** method:

Code Listing 18

```
// System.Web.SessionState.SessionId class in System.Web.dll version 2.0
private static string Encode(byte[] buffer) {
    char[] array = new char[24]; int num = 0;
    for (int i = 0; i < 15; i += 5) {
        int num2 = (int)buffer[i] | (int)buffer[i + 1] << 8 |
        (int)buffer[i + 2] << 16 | (int)buffer[i + 3] << 24;
        int num3 = num2 & 31;
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 5 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 10 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 15 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 20 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 25 & 31);
        array[num++] = SessionId.s_encoding[num3];
        num2 = (num2 >> 30 | (int)buffer[i + 4] << 2);
    }
}
```



```

    num3 = (num2 & 31);
    array[num++] = SessionId.s_encoding[num3];
    num3 = (num2 >> 5 & 31);
    array[num++] = SessionId.s_encoding[num3];
}
return new string(array);
}

```

Code Listing 19

```

// System.Web.SessionState.SessionId class in System.Web.dll version 4.0
private static string Encode(byte[] buffer) {
    char[] array = new char[24]; int num = 0;
    for (int i = 0; i < 15; i += 5) {
        int num2 = (int)buffer[i] | (int)buffer[i + 1] << 8 |
        (int)buffer[i + 2] << 16 | (int)buffer[i + 3] << 24;
        int num3 = num2 & 31;
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 5 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 10 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 15 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 20 & 31);
        array[num++] = SessionId.s_encoding[num3]; num3 = (num2 >> 25 & 31);
        array[num++] = SessionId.s_encoding[num3];
        num2 = (num2 >> 30 & 3 | (int)buffer[i + 4] << 2);
        num3 = (num2 & 31);
        array[num++] = SessionId.s_encoding[num3];
        num3 = (num2 >> 5 & 31);
        array[num++] = SessionId.s_encoding[num3];
    }
}

```



```
return new string(array);  
}
```

We highlighted the Microsoft fix in 4.0. The buggy line of code is supposed to use the remaining 2 random bits (the “>> 30” part) and OR them with the intermediate result. The developer assumed that the 30 bits incoming from the left will be zeroes, but `num2` is an `int` and can be negative. The “& 3” fix zeroes out these incoming 30 bits regardless of their value. This SessionID fix in .NET 4.0 was also not publicly mentioned by Microsoft, as far as we know. One easy way of telling that your SessionID-using ASP.NET application is still running on the 2.0 runtime is to observe the last characters of SessionIDs (2.0-based SessionIDs often end with “45” or “55”). The proper behavior of the right-shift operator is [documented](#) in MSDN.

SessionIDs are sent to the client side via a cookie (default behavior) or as part of the URL, with all the nasty URL rewriting handled by ASP.NET. The default cookie-based mode creates a cookie, which is marked [HttpOnly](#) (good), but not [Secure](#) (bad). There is a [setting](#) to enable secure session cookies, but it is not on by default. You might still feel at ease because your ASP.NET application uses [TLS](#) for all sensitive forms and pages anyway. Another aspect of SessionIDs is that if the client side fails to provide a matching SessionID value to the server side (either SessionID is not provided at all or the provided SessionID does not match), ASP.NET always generates a brand new server-side session state and sends its SessionID to the client side as a cookie, which will override any pre-existing client-side SessionID cookie.

To make it worse, if the client-provided SessionID is valid but matches to an expired or abandoned server-side session state, the newly generated server-side state will reuse the same client-provided SessionID (again, there is a [setting](#) to stop reuse, but it is not on by default). This SessionID reuse effectively enables the server side to switch to a different session state without the client side ever knowing about it, since the client keeps using the same SessionID value. To make this even worse, session state design will accept without question any client-side-provided SessionID string as long as it has a valid SessionID encoding. Effectively, the client gets to decide and fix the exact value of its own SessionID. Many automated vulnerability testing scans get tripped by ASP.NET SessionID cookie handling because it fails [session fixation](#) tests. Session fixation attacks use a different meaning of “session,” however. ASP.NET session state is a non-authenticated session, while session fixation attacks focus on authenticated sessions.

Imagine the following scenario:

Your bank runs TLS for all ASP.NET forms and pages, and uses a cookie-based session state. You feel reasonably secure doing online banking. One of the thousands or millions of URLs on your bank’s website (such as a static CSS file) happens to be HTTP instead of HTTPS. You might not even get a “mixed-content” browser warning if that HTTP resource is not loaded up front, but instead loads on-click or in a new browser window. The SessionID cookie traveled unsecured along with that HTTP request. Since you were banking over Wi-Fi in your local coffee shop, your bank account just got [hijacked](#) without you even knowing.

One easy way to defend against this might be to enforce TLS at the web-server level, no exceptions. However, most sites want to be accessible via HTTP (at least the main page), which then gets redirected to a safe HTTPS site. The only way to properly enforce TLS with session state in this scenario is to have the TLS-secured application on a separate sub-domain (such as `secure.myapp.com`) and restrict the SessionID cookies to that sub-domain. However, many web applications do not want to inconvenience their users with a separate secure sub-domain.

The root of the problem is that session state was never designed to be used for authentication, yet it is very commonly used to implement ASP.NET authentication. Developer ignorance, poor documentation, and ease of misuse all contribute to this dangerous abuse.



Tip: ASP.NET session state should never be used for authentication.

What **should** session state be used for? We are not sure. Someone suggested using it for sticky “non-sensitive” data, like users’ color preferences. This makes no sense to us because the “sensitive” part is not whether you prefer “blue” to “green,” but your implicit expectation of ownership over that decision, and your implicit assumption that nobody should be able to hijack that choice. It is the trust in sanctity of client-server interaction, and not merely the data itself, that is at stake. There is no such thing as “non-sensitive” data in a web session—every bit exchanged between client side and server side must be confidential and authentic.

Microsoft offers the following [guidance](#) on session state: *“The session-state feature is enabled by default. While the default configuration settings are set to the most secure values, you should disable session state if it is not required for your application.”*

We can only build on this by further recommending that you disable session state right away, because that is how you know or will find out that session state is not required for your application. Leaving session state enabled in your applications—even if you are fairly sure nobody is using it today—is asking for trouble. If you leave it enabled, somebody is likely to abuse it.

Just to kick the dead horse one more time, you should not feel safe using session state—even when you are 100% HTTPS and HTTP is not even in your vocabulary. A valid User-A can still set their own session state values and then fix User-B’s session to use the same valid User-A session, thus causing User-B to use User-A’s session without even knowing—HTTPS or not. If you try to fix it by making it obvious to User-B that they are actually using User-A’s session, you are using session state for authentication.

You can gather another bit of wisdom from Microsoft’s recommendation: If the default session state is already at its “most secure” configuration, what does that imply about the security of other session state modes, such as cookieless sessions? We leave this as an exercise to the reader.

Availability is technically also a security concern, and session state has issues here as well. Session state handlers are blocking by default because they require both read and write access to the session state storage, which is a mutually exclusive operation that requires locking. There is a [way](#) to configure ASP.NET handlers to only require read access, which makes them non-blocking and concurrent (at the expense of writability), but it is not the default. It is not uncommon to see an ASP.NET application with multiple AJAX-enabled independent web sections loading serially, one-by-one, even though they were clearly designed to be loaded concurrently. In some cases, the last-loading request will simply HTTP-time-out while waiting in this artificial session state handler queue.

If you have a user-specific, server-side state to maintain, use [in-memory cache](#) (single-server, non-persistent), a distributed cache (such as [Redis](#)), or simply leverage your main database with a simple optional caching layer. There are also many fast key-value stores (which is what

session state is) available if you use cloud hosting. You could also consider storing state on the client side, EtM-encrypted or not, in cookies or using [web storage](#).

We do live in the real world, full of legacy line-of-business ASP.NET applications that need to be supported and maintained. However, we trust that if you have any power over the design of new ASP.NET solutions, you will use it wisely.

CSRF

Cross-site request forgery (CSRF) attacks are number 5 and number 8 on the [OWASP “Top 10 Most Critical Web Application Security Risks”](#) list for 2010 and 2013, respectively. Most development frameworks, including .NET, implement CSRF defenses that follow OWASP recommendations (or, perhaps, OWASP CSRF defense recommendations follow what most development frameworks do currently).

OWASP says the following:

- Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.
- The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is subject to exposure.

Do you see a fundamental disconnect between these two statements? There is nothing wrong with the first statement—it clearly articulates a valid HTTP-level solution to CSRF prevention. The second statement, however, suddenly talks about tokens in “hidden fields”—an HTML feature. Where did HTML come from all of a sudden?

The [OSI](#) model conceptually represents many protocol layers that make the web work. You will find HTTP in that model as the topmost “application layer” protocol. HTML is not in that model because it is a “payload” protocol, just like JPEG, PNG, XML, JSON, CSS, or any other payload that can be transferred over HTTP.

Implementing CSRF defenses in HTML is like plugging a leak by building a dam in the middle of the ocean. CSRF is a variant of the [confused deputy](#) problem, which has two ingredients: a trust-based authority, and confusion that leads to misuse of that authority. Authority on the web is typically driven by authentication of a client to the server, which allows the server to infer authority from the client’s identity. ASP.NET [supports](#) two main authentication modes: Windows authentication and forms authentication. We have not discussed either of these yet, but it suffices to know that both of these authentication modes establish authentication at the HTTP level, meaning they work regardless of what the HTTP payload might be. In a CSRF attack, the confused deputy (your authenticated browser session) is tricked into performing a rogue HTTP request that the authenticated client (you) did not authorize. Did you notice any mention of HTML in the previous sentence?

Defending against CSRF in HTML is not wrong because it is somehow insecure, it is wrong because it is misguided, since it is a defense in the wrong place. It also coincidentally does not help with JSON-over-AJAX payloads or any other non-HTML HTTP requests, which require

additional workarounds to inject the CSRF token into some equivalent of HTML hidden fields for each payload protocol. You could add extra fields to XML or JSON, but good luck with binary payloads. CSRF is fundamentally an HTTP-level problem because that is the level at which authority is established on the web—it should ideally be mitigated at the same HTTP level. Practical solutions are not ideal, however.

Most CSRF defense approaches revolve around unpredictable per-session tokens (1st OWASP statement). These tokens need to be somehow verified for validity. This validation can be implemented statefully or statelessly. In a stateful approach, the server keeps state on all valid previously-issued/non-expired CSRF tokens (the same model is used by ASP.NET session state). The stateful approach does not scale well because there can be an unlimited number of HTTP interactions during a single session, and the amount of state the server side would have to keep can quickly grow beyond practical. The stateless approach keeps on the client side everything needed for the server side to validate CSRF tokens, which requires no server-side state and scales well. The downside of stateless approach is that there is more CSRF-related info that needs to be sent to the server side, since server side is stateless, but this extra CSRF-token-related bloat is usually reasonable. Let's explore the stateless approach.

The first thing we need is some kind of client-side storage for stateless CSRF tokens, and HTTP cookies are ideal. Other client-side storage mechanisms like [web storage](#) are HTML concepts—not HTTP—and thus are out. To be more precise, the HTTP mechanism we want to leverage for storage is HTTP headers, with cookies being a well-supported protocol for storing custom data in HTTP headers. HTTP cookies have the added benefit (for our purposes) of automatically sending what they store to the server side on each HTTP request.

Next, we need a way to ensure authenticity of client-side-stored tokens to prevent attackers from making their own valid tokens. A keyed HMAC (based on a server-side secret key) of the token (appended) will ensure token authenticity. However, HMAC will not help with replayed tokens or attackers being legitimate users and using their own authentic tokens to mount CSRF attacks. This uncovers two important problems that need to be resolved—replay prevention and token identity—which we ideally want to resolve at the HTTP level. There is a good, tested protocol we can leverage for replay prevention at HTTP level: [TLS](#). We could try to avoid tying CSRF tokens to identities by giving tokens a short time to live (for example, by including an HMACed absolute expiration date), which will make it more difficult to exploit legitimately obtained tokens for CSRF attacks. This, however, raises the difficulty bar but does not solve the problem, since a determined attacker would simply obtain a fresh token just in time. Tying CSRF tokens to client identity, on the other hand, would definitively resolve the problem of abusing legitimately issued tokens. Windows authentication does not allow attaching additional (dynamic) data to the identity, but forms authentication does.

The stateless CSRF token approach also requires a side-channel mechanism for token submission, which cannot be exploited by an attacker to trick the client (browser) to generate a valid HTTP request. The side-channel role is played by HTML hidden fields in the OWASP 2nd statement, while the primary channel is HTTP-cookie-based (stateless mode). However, we do not want to use HTML form fields unless we really have to. We need some HTTP-level “operation” that a legitimate browser could do, but an attacker could not.

One idea is to rely on the assumption that the CSRF attackers cannot inject or modify HTTP headers on HTTP requests of a confused browser, but legitimate self-originating requests can. The problem with this assumption is that some browser plugins do not play by the rules and are able to bypass this restriction.

Another idea is to leverage the [same-origin policy \(SOP\)](#) browser security mechanism, which is getting harder to bypass in modern web browsers. An unpredictable CSRF token could be stored client-side in a tamper-proof, encrypted authentication cookie (which is what forms authentication does), marked **HttpOnly** (inaccessible for reading). The same plaintext CSRF token value could also be stored in a second cookie **not** marked **HttpOnly**, which allows SOP-approved JavaScript to read it. A SOP-approved HTTP request in need of CSRF protection would first read the CSRF token value from the plaintext cookie and either inject that value into a custom HTTP header on the outgoing request, or, if header injection is not available, inject that value into the payload (a concession, since it is not a pure HTTP-only approach anymore). The server side would compare the CSRF token from the decrypted and validated authentication cookie container with the one provided on a side channel (HTTP header or payload-embedded), and only authorize the request upon a successful match.

The benefit of such an approach is that even if the targeted browser has a plugin that does not play by the rules and can inject or modify HTTP headers on outgoing requests, that plugin would not know what to use for the correct token value, since access to that value is additionally protected by the SOP. It is possible that a browser plugin could bypass the SOP, but then the browser is unsafe to begin with. SOP applies to [DOM](#), which includes cookies (via `document.cookie`) as well as any HTML element data, such as hidden fields (via `document.body`). SOP also applies to [XMLHttpRequest](#).

This approach can further be strengthened by adding an expiration to the mix. Forms authentication tokens, for example, often have a sliding expiration, which causes a new valid authentication token to be sent to the client when the old one is approaching expiration. If this authentication token re-issue mechanism could be plugged into, we could also issue a new CSRF token at the same time. This would decouple the web session, HTML page, and CSRF token lifecycles and enable shorter CSRF token lifespans.

The cookie-based, just-in-time CSRF token injection has several benefits over the more common approach of HTML-hidden-field token storage. One benefit is that the CSRF lifespan is no longer coupled to the HTML lifespan, which allows for shorter-lived, more secure tokens. Another benefit is that injection can be automated and made transparent to developers, who no longer have to remember to use special commands to generate a separate CSRF token for every form and XMLHttpRequest. Authentication and CSRF cookies automatically get issued, re-issued, rotated, expired, etc. with zero friction for developers—it just works. The payload manipulation is only used when custom HTTP headers cannot be set; for example, when submitting HTML forms.

This is a recipe for success—we just need the right ingredients to make that recipe work. Specifically, we need proper mechanics to access the non-**HttpOnly** CSRF cookie value and inject it into HTTP header or form submission just-in-time, right before an outgoing HTTP request is ready to go out. The just-in-time requirement is due to an expected asynchronous rotation of authentication and CSRF token cookies. This calls for some JavaScript code. HTTP calls must be triggered from somewhere, and browsers trigger them from either HTML or JavaScript (at least the server-state-altering calls). We need to intercept these HTTP calls right after they are triggered, but before they are made, so we employ some JavaScript to make it happen. We are using jQuery, but it can be easily ported to raw JS or to other DOM-manipulating JS libraries.

```
function getCsrftoken() {
    // use your favorite cookie-reading logic to return the CSRF cookie value
}

/*****
Set up a prefilter for $.ajax() call to append a csrf header with the csrf
token value, which is read from the csrf cookie prior to each $.ajax()
call.
The prefilter can further be overloaded via "beforeSend" function.
The call sequence is (1) $.prefilter; (2) beforeSend (3) $.ajax() call.
*****/
$.ajaxPrefilter(function (options, originalOptions, jqXHR)
{
    var csrfToken = getCsrftoken();
    if (csrfToken)
    {
        jqXHR.setRequestHeader("X-CSRF", csrfToken);
    }
});

/*****
There is no way to add custom HTTP headers to non-AJAX requests, such as
<form> POSTs. Instead, we auto-inject a hidden input element with CSRF
token into every form on submit.
*****/
var csrfName = "__csrf";
var submitHandler = function (e, form)
{
    form = form || this;
```

```

if (form.method === "post")
{
    var csrfToken = getCsrftoken(), $csrfInputElement;
    if (csrfToken)
    {
        $csrfInputElement = $("#" + csrfName, form);
        if (!$csrfInputElement.length)
        {
            var csrfHtml = '<input type="hidden" name="' + csrfName +
                '" id="' + csrfName + '" value="' + csrfToken + '" />';

            $(form).append(csrfHtml);
        }
        else $csrfInputElement["val"] (csrfToken);
    }
}
};

$(document).on('submit', 'form', submitHandler);

/*****
ASP.NET form submission is done via globally registered "__doPostBack"
function. We need to intercept it and add CSRF-handling logic.
*****/

var beforePostBack = function (beforeFunc)
{
    var old__doPostBack = this["__doPostBack"];
    if (old__doPostBack)
    {
        this["__doPostBack"] = function (target, argument)
        {
            beforeFunc(target, argument);
            return old__doPostBack(target, argument);
        }
    }
}

```



```

    };
}
};

$(function ()
{
    beforePostBack(function (target, argument)
    {
        submitHandler(null, $("[name='" + target + "']").closest("form")[0]);
        return true;
    });
});

```

We are not JavaScript masters, but this code is not too complicated, generic, and certainly automates all client-side CSRF-handling duties. We have not covered the mechanics of the server-side CSRF token handling yet. Since there are many benefits of tying CSRF tokens to authentication, we should first review forms authentication in depth to properly set the stage.

Forms authentication

ASP.NET [forms authentication](#) (FA) is Microsoft's stateless implementation of an [HTTP+HTML](#) authentication technique. While this technique is wildly popular and widespread, it is not standardized and is entirely implementation dependent. The client-side credentials in FA (for example, a username and password combination) are typically sent to the server side "in-band" as part of the HTML payload (for example, HTML form submission). The server-side generates an authentication ticket upon successful credential validation, and usually sends that ticket "out-of-band" to the client side, typically within an HTTP cookie header. Subsequent client-side requests provide this authentication ticket out-of-band as well. The server side is then able to verify the claimed identity, as well as the legitimacy of the identity claim without any additional client-side credential submission.

ASP.NET FA can operate in cookie-based or cookieless mode, with the [default](#) being a runtime choice made by ASP.NET based on the claimed HTTP capabilities of the client-side device. Cookieless FA is full of security vulnerabilities, so it is best to proactively turn it off by forcing a cookie-based FA mode at all times. FA requires TLS for security, so you must ensure that all FA-based solutions are 100 percent delivered over TLS. Since you are unlikely to actually do it just because you've read the previous sentence, you need proper motivation for yourself and your team to enforce TLS. That motivation is the [RequireSSL](#) attribute, which you should set to **true**, since it is **false** by default. **RequireSSL=true** causes FA cookies to be marked **Secure**, which prevents compliant client-side agents (such as modern browsers) from sending **Secure**-marked cookies over non-TLS HTTP connections. Thus, if you accidentally forget to deploy

TLS, FA will stop working. Deploying TLS properly means “TLS only, HTTP disabled,” not “TLS by default, HTTP works just as well.” However, if you manage to screw it up and HTTP ends up working just as well, **RequireSSL=true** might prevent an exploit. All FA cookies are marked as **HttpOnly**, which makes them inaccessible to scripts in compliant (modern) browser agents.

The default FA cookie [timeout](#) **T** is 30 minutes, and [sliding expiration](#) is enabled by default. Every cookie-wrapped FA ticket will expire in **T** minutes, while any valid client-side-submitted FA ticket with less than [half-life time to live \(TTL\)](#) will trigger the server-side ASP.NET FA module to automatically issue a brand new cookie-wrapped FA ticket. This renewed FA ticket is sent to the client side with the HTTP response, and buys **T** more minutes of authenticated access. If the ticket renewal is triggered right around the half-life TTL, the effective lifetime extension is only $\sim T/2$ because the old FA ticket would have lived for another $\sim T/2$ minutes, while the newly issued replacement ticket is good for **T** minutes from issue time. If the ticket renewal is triggered right before expiration, the effective lifetime extension is $\sim T$.

The FA sign-out mechanism is constrained by the stateless nature of the ASP.NET FA implementation. The server-side FA module implements sign-out by sending a void, already-expired cookie to the client side, which causes compliant browsers to “forget” the valid FA ticket they have been holding. A malicious client does not have to honor the “please-forget-your-valid-ticket” server-side request, however. Such a client can continue to use the previously obtained valid FA ticket, which the server will keep renewing forever, no questions asked. The only way for the server to stop accepting and automatically renewing valid FA tickets is to make them invalid, typically via changing the global server-side secret key ([machineKey](#)) used to EtM all tickets. This would invalidate *all* previously-issued FA tickets, meaning all currently authenticated users are kicked out.

FA tickets also do not differentiate between **issue date** and **creation date**, since only issue date is recorded within the ticket. The [IssueDate](#) tracked by the FA ticket is the time at which the ticket was created, regardless of whether this was the very first ticket creation event, or subsequent ticket renewal events. In other words, issue date is always equal to **expiration date** ([Expiration](#)) minus **T** minutes. The fact that a particular FA ticket is being renewed for the umpteenth time is not captured anywhere. This makes it impossible to distinguish legitimate FA tickets that were originally created minutes or hours ago from exploited, kept-alive tickets that were created days, months, or years ago. It would be nice for an improved FA implementation to also capture the creation date, which we define as the moment of the very first, original, non-renewed ticket creation (where “renewal count” = 0), but ASP.NET FA implementation does not do that.

An explicit renewal count integer would have been helpful as well, since it cannot be definitively inferred from the difference between the creation date and expiration date. Finally, a unique FA session ID would have been helpful to identify each unique FA session. The FA session ID could have been a GUID, since it is protected by the ticket container, or it could have been cryptographically generated (similar to session state SessionID) for unpredictability in unprotected-use scenarios.

Having a unique FA session ID would allow for an optional server-side state. For example, each server in a huge server farm could locally cache some information related to a specific FA session ID (just like the session state mechanism), and go to higher-latency data store only on cache misses. One useful scenario is going to a higher-latency data store to check whether the provided FA ticket corresponds to a valid identity (such as a non-disabled user account) and minimizing per-request data store trips with local caching. Another useful FA session ID

scenario is transparent (no user involvement) re-validation of client-side credentials against a lower-latency data store, but only on cache misses. We do not want to use any user-specific unique ID (such as user ID) for FA session ID because we should ideally support multiple parallel FA sessions per user account (for example, concurrent desktop and mobile device sessions, or simply multiple browsers signed-in in parallel).

The built-in ASP.NET 4.5 FA ticket stores the following data:

Table 8: ASP.NET 4.5 forms authentication ticket internals

FA ticket data	Comments
Serialized version (1 byte)	The version of internal ticket serialization, probably intended for future serialization protocol agility. Currently set to 1. Internal.
Ticket version (1 byte)	Developer-provided arbitrary ticket version, which defaults to 2. API exposes it as an <code>int</code> , which internally gets converted into a <code>byte</code> . If you try to set it to 256 and round-trip it, you will get 0 due to silent byte conversion. MSDN is silent about it, but MS reference source code comments say the following: <i>“Technically it should be stored as a 32-bit integer instead of just a byte, but we have historically been storing it as just a single byte forever and nobody has complained.”</i> Which does not make it right, of course.
Issue Date (8 bytes)	Date of ticket issue (original or renewal, as discussed previously).
Spacer (1 byte)	Used to break compatibility with pre-4.5 ASP.NET tickets. Internal.
Expiration Date (8 bytes)	
IsPersistent (1 byte)	
Name (variable-length)	User-provided .NET string, char-serialized.
User Data (variable-length)	User-provided .NET string, char-serialized. Defaults to empty string.

FA ticket data	Comments
Cookie Path (variable-length)	User-provided .NET string, char-serialized. Defaults to .config value.
Spacer (1 byte)	Another spacer to break compatibility with pre-4.5 ASP.NET tickets. Internal.

Pre-ASP.NET 4.5 FA ticket implementations had a different serialization format and a different authenticated symmetric encryption approach, which suffered in 2011 from a [critical security vulnerability](#) that compromised every version of .NET FA and required an out-of-band patch. This critical vulnerability has been in production for more than seven years. ASP.NET 4.5 uses a new serialization format (not really material from a security perspective) and a new EtM encryption approach, which is finally done properly (similar to the EtM approach we described). That alone is a compelling enough reason to run ASP.NET 4.5.

ASP.NET FA tickets have been historically encoded as Base16, which does not change with ASP.NET 4.5. Microsoft continues to support the cookieless FA mode, which is the likely reason for case-insensitive Base16 ticket encoding. Legacy compatibility is a bad thing to be stuck with. Unfortunately, all ASP.NET cookie-based FA deployments have to pay the price of Base16 bloat. It would have been nice if ASP.NET were intelligent enough to use Base16 ticket encoding for cookieless FA tickets, and Base64 encoding for cookie-based FA tickets. Switching from Base16 to Base64 would reduce encoded FA ticket size by 33 percent and increase the storage density by 50 percent. Given a maximum 4-kilobyte HTTP cookie size, Base16 can store at most 2 kilobytes, while Base64 can store at most 3 kilobytes—that's one extra kilobyte of storage (for example, a serialized 512-byte .NET string).

Membership

The identity-carrying part of the FA ticket is the mandatory **Name** string property, which should be set to a unique client identity. This unique identity should be set as an outcome of some black-box process of credential verification, which is out of scope for FA. Such credential verification process typically involves three phases:

- Registration (credential creation and storage)
- Submission (credential provision for the purposes of authentication)
- Verification (validation of submitted credentials)

ASP.NET implements these credential management phases in a separate “[membership](#)” API, which can have multiple “providers” (implementations). ASP.NET ships with SQL Server and Active Directory membership providers. There is also an abstract [ExtendedMembershipProvider](#) class with [SimpleMembershipProvider](#) implementation in `WebMatrix.WebData.dll`, and a separate NuGet-only Microsoft package offering “[universal](#)” providers, where the “universal” part seems to refer to the fact that tightly coupled SQL Server

persistence is exchanged for tightly coupled Entity Framework-based persistence, which supports a few additional storage backends.

From a security perspective, **SimpleMembershipProvider** is the only Microsoft-provided membership implementation that makes an attempt at doing the right thing, such as using computationally slow password-based key derivation. From a design and usability perspective, membership is an ancient API that might have been useful in 2005, but is no longer adequate when evaluated against modern, secure engineering design principles, because it suffers from insufficient security, multiple [SRP/SoC](#) violations, and tight persistence coupling. While Microsoft has tried to address some issues with **SimpleMembershipProvider**, it is a temporary fix at best. There is simply no good way to fix membership inadequacies other than to use a more modern credential management API. There is also no consistency of credential management among various ASP.NET technologies, such as Web Forms, MVC, Web Pages, and Web API.

Insider threats

An important security goal to aim for is defense against insider threats—attackers or current and former employees who might be able to access your precious data from within, and who should be assumed to have full insider knowledge and complete server-side read access (including full knowledge of all secret keys, complete DB read access, etc.). This is a tough security goal to reach, but it is necessary to have at least some defense mechanisms in place to thwart insider threats as part of a defense-in-depth approach to security. Membership APIs and—dare we say it—all other ASP.NET security mechanisms were never designed to protect against insider threats.

“There are only two types of companies: those that have been hacked, and those that will be. Even that is merging into one category: those that have been hacked and will be again.”

FBI Director Robert Mueller (March 1st, 2012)

Security experts tend to have an even gloomier perspective, that there are only two types of companies: those that have been hacked, and those that do not know they have been hacked. It is not rational to be concerned about an [advanced persistent threat \(APT\)](#) brute-forcing “weak” 1,000-iteration PBKDF2-SHA1 within **SimpleMembershipProvider** while ignoring that junior developer, fired last week, who has seen the <machineKey> secret passwords and thus is able to forge any desired FA cookie and identity because you never change <machineKey> passwords. Even if you had a policy to change secret passwords and bothered to actually roll out new passwords across your 100-server web farm (which, of course, you could painlessly do with a single click, since everything is so perfectly automated at your company that it almost runs itself), you would have to wait a week or two until the next scheduled production push anyway.

This hypothetical scenario is not that far-fetched, and could be a serious security threat. While insiders might not be particularly advanced, sophisticated, or malicious, their persistence and permanence, coupled with rampant (willful) ignorance, incompetence, poor training, and underfunding, more than make up for it (security rarely generates revenue). While APTs are more dangerous when they are well-funded, insiders are typically more dangerous when they are inadequately funded. Having the right mindset is crucial for insider threat mitigation, and the

best way to do it is to assume that the attacker is “you,” or, rather, your evil self from a parallel universe who knows everything you know, and has read access to everything you have read access to.

Credential storage

We have already covered PBKDF2 for deriving a master key (MK) from a low-entropy SKM, such as a user password. PBKDF2 forces you to provide a salt (for example, a GUID), so you are unlikely to forget to protect a MK against SKM reuse. You could then store the MK and a salt against a user record at this point, and in fact many systems do just that. We advise against it, however.

One issue is that at this point there is nothing connecting the MK to a user ID other than storage. Ideally, a MK should be correlated to a user ID via cryptography, and not just via storage. A hypothetical insider with read-only access to user IDs and read-write access to the MK and salt could replace one MK/salt record (for the “admin” user) with another MK/salt record (from an insider’s own account with a known password) to gain access, and possibly change it back to avoid discovery.

Another issue is that it is wise to have an explicit one-way transformation between key extraction/derivation and storage, which you skip if you store the PBKDF2-derived result directly. One reason to have this explicit one-way transformation is to cleanly obtain an image and a pre-image of MK. For example, a PBKDF2-derived secret can be the pre-image of MK (PMK), and a one-way image of PMK can be the actual MK saved to storage.

One obvious idea to generate MK might be $MK = \text{HASH}(\text{PMK})$, where HASH is a cryptographic hash like SHA-512. This does not address our desire to cryptographically connect MK and a user ID, so do not do that. Another idea might be $MK = \text{HASH}(\text{PMK} + \langle \text{user ID} \rangle)$ or $\text{HASH}(\langle \text{user ID} \rangle + \text{PMK})$, where + is concatenation. This idea is most likely susceptible to [length extension attacks](#), so definitely do not do that either (even with SHA-384). There is a proper cryptographic tool to authenticate data with a secret key: MAC, and our preferred MAC implementation, HMAC. We can do $MK = \text{HMAC}_{\text{PMK}}(\langle \text{user ID} \rangle)$. This is our preferred approach because it ensures a one-way PMK-to-MK transformation, and cryptographically ties MK to user ID at the same time.



Tip: As a general rule, never use a hash as a replacement for a MAC by creatively injecting a secret into a hashed message.

We store both MK and salt against the user ID record, but what should we do with PMK? Various ASP.NET membership providers have no separate concept of PMK because they skip the PMK-to-MK step. $\text{PMK} = \text{MK}$ for them. These membership providers also do MK verification against server-side storage only once: when the FA cookie is generated for the first time during (successful) login. Subsequent FA cookie submissions are validated based on server-side ability to EtM-decrypt the FA cookie with a fixed server-side secret key (or derived keys). As long as EtM-decryption was successful, the FA cookie is accepted as valid, and all claims made by such FA cookie contents are accepted. Effectively, the ASP.NET security model exchanges a secret, user-provided key for a bunch of identity claims signed with a server-side secret key that outsiders should not know, which prevents outsiders from forging identity claims. Note that

the server-side identity storage is not consulted as part of identity claim verification—all it takes to verify identity claims is EtM-decryption attempt. Insiders, however, are assumed to know all secret server-side keys, and are thus able to forge any identity claims because they know the secret key needed to make EtM-decryption succeed.

The obvious upside of the ASP.NET FA security model is that it scales well because it does not consult server-side storage for identity claim verification for every post-login request. The obvious downside of this approach is that insider threats are clearly out of scope for such a security model. ASP.NET FA's inability to invalidate tickets upon sign-out (which we discussed earlier) is a direct consequence of non-existent post-login, server-side credential validation. Wouldn't it be nice to have a more robust FA security model that offers some resistance against insider threats and allows for fine-tuning and striking the optimal balance between always checking storage for validation of post-login client requests, and never checking storage (ASP.NET FA)? This is where PMK can help.

We can store PMK within the EtM-encrypted FA ticket container (for example, within the **UserData** string) when a user logs in with credentials used to derive PMK. Subsequent client-server interactions will be authenticated via (1) EtM-decrypting the FA cookie with the server-side secret key; (2) extracting PMK and <user ID> claims from the FA cookie and checking $\text{HMAC}_{\text{PMK}}(\text{<user ID>})$ against the server-side storage record for that specific user ID. Insiders might be able to bypass (1) using knowledge of server-side secret keys, but it would be much harder for them to bypass (2) since that would either require knowledge of a user's PMK, or the ability to permanently or temporarily forge a <user ID> in storage.

Most knee-jerk reactions (known as “Internet advice”) to storing sensitive data in cookies typically look like this:

1. Use TLS with **Secure**, **HttpOnly** cookies.
2. Encrypt your cookies.
3. DO NOT DO IT.

The must-have requirement for TLS is sound. If someone can steal a valid (non-expired) FA cookie, it does not matter what is inside of it. The encryption requirement is technically not a must-have, since authenticity of the cookie (i.e. its MAC) is often more important than the privacy of its contents. However, we do want to encrypt our cookies, because EtM encryption not only gives us both privacy and authenticity, but also allows us to leverage that cookie container for data that truly requires privacy, such as PMK. The “do not do it” part is a result of [fear, uncertainty, and doubt](#), however.

PMK is a properly salted, PBKDF2-derived (and sufficiently iterated), fixed-length value. Assuming that both TLS protection and EtM-encryption of the cookie container are somehow bypassed (properly implemented cryptography is rarely broken and is typically bypassed), PMK could only be useful to an insider, since outsiders cannot forge FA cookies. PMK knowledge does not reveal the actual plaintext user password required to log in. If a user's plaintext password happens to be easily predictable, nothing would help thwart attackers anyway.

Improving forms authentication

We have already described the ASP.NET 4.5 FA deficiencies and its ticket container contents in a prior chapter. Let's summarize these deficiencies, which we will try to rectify:

- Space-inefficient Base16 container encoding.
- Not designed to be resistant to insider attacks:
 - No cryptographic connection to user credentials.
 - Credential validation is only done once (on login).
- No concept of separate creation date and (re-)issue date.
- No concept of (re-)issue count.
- No concept of unique session ID to identify each unique FA session, which survives ticket re-issues.
- Not designed to provide CSRF defense.

We will employ the following approaches to try to improve on these deficiencies:

- Use Base64 container encoding.
- Provide insider attack resistance by:
 - Adding user-credential-derived PMK to FA ticket.
 - Adding custom logic hooks to trigger credential validation as often or rarely as desired.
- Add creation date to FA ticket—(re-)issue date is already captured as **IssueDate**.
- Add re-issue count (unsigned integer) to FA ticket.
- Add a unique session ID to FA ticket.
- Provide CSRF defense by:
 - Adding an unpredictable CSRF token to FA ticket, regenerated anew on every ticket re-issue.
 - Adding a second cookie with a JavaScript-readable CSRF token value, lifetime-synced to FA cookie.

The additional data we want to add to the **FA** ticket container has to be stored in a **UserData** string in an API-transparent way so that the API-consuming user can continue to store user-provided data in the **UserData** string as its documentation specifies. Let's see what our storage requirements might look like:

Table 9: Alternative forms authentication ticket internals

Data	Size (bytes)	Comments
Creation Date	8	DateTime structure as binary.
Issue Count	4	UInt32 .
Session ID	16	128-bit random CSP-generated.
CSRF token	15	120-bit random CSP-generated.
User-credential-derived PMK	32	PBKDF2-HMAC-SHA-512/256 (i.e. leftmost 256 bits only).
Total	75	As byte array.
Base64(Total)	100	As UserData -stored string.
Text-serialized	200	Double the string size as a byte array.
Base-64-encoded cookie	~267	As added to FA cookie container.

As you can see, piggybacking on top of the **UserData** string field leads to a lot of storage overhead as a result of FA ticket APIs not supporting direct byte array storage. Had direct byte array storage been available to us, we would only need a single Base64 encoding for the final cookie container—a 1.33 times bloat factor, instead of 3.56 times.

On the positive side, all of the fields are fixed-length, which means that we can easily prepend and un-prepend their combined Base64-encoded value to or from the user-provided **UserData**. Algorithmic agility can be implemented via the existing mandatory **Version** byte field.

120-bit CSRF token security strength should be more than sufficient since CSRF tokens have a lifetime of a single FA ticket issue, which is 30 minutes, with default ASP.NET sliding expiration settings.

Using a full, 512-bit PMK would have been desirable from the security perspective. However, 64 bytes are a heavy payload to carry on every single FA cookie, which would get further bloated via the $1.33 \times 2 \times 1.33 = 3.54$ multiplier to 227 cookie bytes. Using a truncated, 256-bit PMK

reduces the extra cookie payload in half (to 113 bytes) while still providing adequate user password entropy extraction and security margin.

Given a 256-bit PMK, the server-side will store $MK = \text{HMAC-SHA-512/256}_{PMK}(<user\ ID>)$ —i.e. the PMK-keyed 512-bit HMAC of a user ID, truncated to 256 bits. It is crucial for the server side to have no record of users' PMKs. Each authenticated client-side request will provide a PMK claim and a user ID claim to the server side. The server side then has full discretion to either trust these claims because the EtM decryption was successful (weak claim authentication), or alternatively, do a bit more work by calculating MK and validating it against storage for the claimed user ID (strong claim authentication). An insider would be able to bypass weak authentication, but not strong authentication.

The inclusion of a unique, unpredictable session ID can also help reduce the costs or latencies of PMK-to-MK validation via a caching layer. The PMK-to-MK validation can only be triggered when the session ID is not in the cache; otherwise, the FA ticket is deemed to be strongly authenticated due to a prior PMK-to-MK validation. There are many distributed in-memory cache solutions available for .NET (free and commercial, from Microsoft and other vendors, cloud-based and on-server), with varying consistency features. Even if consistency is not perfect, the worst that can happen is an extra PMK-to-MK validation—there are no security failures when cache consistency fails. Simple scenarios can even use [ConcurrentDictionary](#) for session ID cache.

Our security model assumes that insiders have read-only access to all server-side secret keys and read-only access to all server-side storage, but have no access to application or web server memory. Even if insiders have read-write access to the in-memory session ID cache, the cached session IDs are anonymous. There is no way to determine that a particular session ID belongs to a particular logged-in user, and there is no way to predict or “fix” the Session ID for new logins. You should avoid using FA tickets as values in **ConcurrentDictionary** because that would de-anonymize session IDs.

You might consider using a custom FA ticket container that does not suffer from the bloat of ASP.NET FA ticket APIs, since you already know by now exactly what data you want to include, and how to serialize, store, encode, and encrypt it. We recommend not to do it, however, unless ASP.NET FA ticket storage and processing are serious bottlenecks for you that you have measured, and you have already exhausted all other optimization avenues. The key reason to piggyback on top of ASP.NET FA APIs is assurance of the following:

- Microsoft's implementation is mature (time-tested) and reasonably sane from a security perspective.
- Microsoft maintains it for you, and will fix any security issues faster and better than you can.
- Should any serious issues be uncovered in the future, ASP.NET FA will be broken for everybody, worldwide. The headlines will be about Microsoft and not about your company. You could stand behind “following Microsoft guidelines” instead of defending the merits of and reasons for your custom implementation. Microsoft could even be liable to your company in the extreme case.

The decision to trigger strong claim authentication is entirely flexible, and could depend on the following:

- Session ID cache miss.
- FA ticket renewal.
- FA ticket issue count getting above a certain, non-typical threshold.
- FA ticket issue count going through N increments (i.e. on every N^{th} ticket reissue).
- FA ticket lifetime (expiration date minus issue date) is above a certain threshold.
- CSRF token validation failure (which by itself should prevent a requested action, but perhaps you also want to check for and log any strong claim authentication failures in this case).
- IP-based, role/permission-based, action-based, or any other custom logic.

New ASP.NET crypto stack

ASP.NET 4.5 cryptographic code paths have been substantially revamped due to serious security vulnerabilities that have plagued prior ASP.NET versions. These changes are well-covered in [Part 1](#), [Part 2](#), and [Part 3](#) of Levi Broderick’s “Cryptographic Improvements in ASP.NET 4.5” MSDN blog posts. The old **MachineKey.Encode** and **MachineKey.Decode** methods have been obsoleted by new [MachineKey.Protect](#) and [MachineKey.Unprotect](#) methods, which do proper EtM with proper KDF and granular, context-driven master-key key derivation.

Code Listing 21

```
static byte[] Protect(byte[] userData, params string[] purposes)
static byte[] Unprotect(byte[] protectedData, params string[] purposes)
```

The secret encryption and verification master keys used by **Protect** and **Unprotect** are automatically sourced from `<machineKey>` configuration settings, just like with **Encode** and **Decode**. This implicit master-key configuration makes the **Protect** and **Unprotect** APIs easier to use, and more difficult to abuse or misuse. Unfortunately, this inability to use externally provided master keys also substantially limits the usefulness of the **Protect** and **Unprotect** APIs. One might be tempted to “hack” around this limitation by using fixed **MachineKey** keys and providing custom external master keys via the **purposes** array instead. While we cannot confirm or deny the security or insecurity of such an approach, we do know that the **purposes** array is designed to be a non-secret distinguisher, not a secret key. Abusing this design is no different than abusing HMAC for message confidentiality: it might work, but unless a professional cryptographer tells you that it does, you should avoid any misuse of cryptographic designs and APIs.

ASP.NET 4.5 allows for a complete replacement of the cryptographic black box used by the **Protect** and **Unprotect** methods (and thus virtually all ASP.NET crypto) to convert **userData** into **protectedData** and back. This replacement is enabled via any concrete implementation of

a new abstract [DataProtector](#) class coupled with .config file instructions to use such **DataProtector** implementation in place of the default ASP.NET 4.5 crypto. You could experiment with this feature by doing a simple **DataProtector** implementation that returns a plaintext byte array as a ciphertext byte array and vice versa—purely for learning purposes, and just to get a feel for it.

The **DataProtector** replacement capability is a great backup plan to have in case major security vulnerabilities are uncovered, either in ASP.NET’s implementation or in the cryptographic primitives used by ASP.NET. It still does not help us with custom, externally provided secret master keys since the **Protect** and **Unprotect** APIs are still the only available high-level APIs. Fortunately, we can leverage a proper EtM implementation from the [Inferno](#) library, which does not have implicit-key limitations. Inferno-based implementation of **DataProtector** also happens to be about 15 percent faster than Microsoft’s default **Protect/Unprotect** crypto, although that is not a good enough reason to prefer it, unless need for speed trumps all other concerns for you.

ASP.NET CSRF API

ASP.NET 4.5 comes with CSRF defense APIs located in the [AntiForgery](#) static helper, which provides two method pairs:

- **HtmlString GetHtml()**
- **void Validate()**

...and...

- **void GetTokens(string oldCookieToken,out string newCookieToken,
out string formToken)**
- **void Validate(string cookieToken,string formToken)**

MSDN has basic information on how to use the first pair of methods, but not the second pair. There is also very little information on how these APIs are designed to work, which makes it very difficult to use them properly. Let’s go over the **AntiForgery** mechanics first. There are two “tokens” at play: a cookie token and a form token.

Table 10: ASP.NET AntiForgery token internals

Cookie token	Form token
1-byte token version, set to 1.	1-byte token version, set to 1. 128-bit (16-byte) CSP-random value.

Cookie token	Form token
128-bit (16-byte) CSP-random value.	1-byte IsSession flag, set to 0.
1-byte IsSession flag, set to 1.	1-byte flag for whether claim-based (1) or not (0).
	ClaimUid bytes or Username string (UTF-8) based on above flag.
	AdditionalData string (UTF-8) or 0-byte if absent.
Total size (pre-encryption): 18 bytes	Total size (pre-encryption): 21 bytes minimum

The binding of cookie and form tokens to each other is done via their 128-bit CSP-random value, which must be identical for both tokens to infer a valid binding. Both cookie and form tokens are EtM-encrypted and Base64-encoded to become final string values. There is no real need to encrypt the cookie token since there is nothing secret in it, but Microsoft does it anyway, likely in order to authenticate it (the M part of EtM). This unnecessary encryption adds a lot of bloat to cookie tokens, which travel on every HTTP request (it adds up).

ASP.NET CSRF tokens are not integrated with ASP.NET authentication and authorization mechanisms. You might be forgiven for thinking that they are a new optional ASP.NET feature with no dependencies, because that is what their documentation implies. The reality is that ASP.NET CSRF tokens are fundamentally tied to context identity controlled by **HttpContext.Current.User.Identity**. If your application is setting identity on **HttpContext**, then the ASP.NET CSRF tokens will encode that identity into the form token and validate it later against the current **HttpContext**. However, a vast number of ASP.NET applications have custom authentication logic which is not driven by **HttpContext.Current.User.Identity**. ASP.NET CSRF form token would only capture and validate identity when **HttpContext.Current.User.Identity** exists and has **IsAuthenticated=true**.

The following code would set a generic “empty-string” identity, which has **IsAuthenticated=false**:

Code Listing 22

```
HttpContext.Current.User = new GenericPrincipal(
    new GenericIdentity(""), null);
```

If **HttpContext.Current.User.Identity** is null or has **IsAuthenticated=false**, the form token will encode an empty string as identity—it will not throw or otherwise alert you. This is dangerous because it makes CSRF tokens from all users interchangeable and allows existing users to mount CSRF attacks against each other. Thus ASP.NET CSRF token APIs do not work

out-of-the-box because they require specific ways of identity management. This dependency is not mentioned or covered by Microsoft's documentation, but even if it were documented, it would only emphasize that Microsoft's CSRF token implementation is an afterthought approach, which is inferior to alternative approaches that integrate CSRF defense mechanisms with identity and authentication by design.

For example, the 15-byte CSRF token we described in the [“Improving forms authentication”](#) section would Base64-encode to 20 bytes. Compare that with Microsoft's 18-byte CSRF token, which gets EtM-encrypted to 108 bytes:

18 bytes → 32 (AES padding) → 80 (16 bytes of IV; 32 bytes of MAC) → 108 (Base64)

Another issue is that ASP.NET CSRF tokens are identity-bound, not session-bound. Imagine that Twitter is running ASP.NET (try). You have just signed up for the @Ironman account. However, unbeknownst to you, someone already had the @Ironman account before and deleted it, which allowed you to grab it. That someone has valid CSRF tokens for the @Ironman identity, and can mount CSRF attacks against your account—forever.

Why forever? Yet another issue with ASP.NET CSRF tokens is that they have no expiration (sessions expire, identities do not). Form tokens are intended to live within HTML, and HTML does not expire either. The only way to invalidate already-issued ASP.NET CSRF tokens is to change the ASP.NET machine keys on the server.

You can work around that by making ASP.NET CSRF tokens session-bound. One approach is to use unique identities. Unique identities are not session-bound, but offer some defense against CSRF token replay. While the old and the new Twitter @Ironman handles are the same, you would internally identify them with different GUIDs, which become the real internal account identities. Another approach is to leverage form token's `AdditionalData` property. You would want to store some session-specific identifier in this property. You cannot set it directly, however, and can only do it through setting [AntiForgeryConfig.AdditionalDataProvider](#), which requires you to implement the [IAntiForgeryAdditionalDataProvider](#) interface.

Making CSRF tokens session-bound, and not merely identity-bound (even when that identity is unique), is also important for the insider threat model, which is rarely considered by built-in ASP.NET components. An insider with read access to machine keys can generate forever-valid CSRF tokens for any known identity, such as “Administrator.” Making CSRF tokens session-bound would require an insider attacker to guess a valid, short-lived session identifier for a specific identity, which is arguably a much harder task. Do not use ASP.NET SessionIDs as your session identifier because they can be client-side fixed by an attacker (do not use ASP.NET session at all). Client-side session fixation would enable an attacker to generate a specific known session and trick the victim into using that session instead of a random unguessable session, so make sure you have addressed that threat.

Since Microsoft's CSRF tokens use the ASP.NET crypto stack for EtM encryption, .NET 4.5-produced tokens will be very different and incompatible with tokens produced by earlier .NET Framework versions (see our discussion in the [“New ASP.NET crypto stack”](#) section). The .NET 4.5 crypto stack is only used by Microsoft's CSRF token logic when [HttpRequest.TargetFramework](#) returns a “4.5” version object—otherwise, pre-4.5 crypto APIs are used. In ASP.NET applications that property can be controlled via the .config file setting, as [this blog post](#) explains. Unfortunately, the `<httpRuntime targetFramework>` .config setting has no effect on non-ASP.NET applications, which will default to

HttpRuntime.TargetFramework=4.0, even when compiled for and running on .NET 4.5. One way to force non-ASP.NET applications running on .NET 4.5 into producing 4.5-based CSRF tokens is:

Code Listing 23

```
AppDomain.CurrentDomain.SetData("ASPNET_TARGETFRAMEWORK",  
    new FrameworkName(".NETFramework", new Version(4, 5)));  
HttpRuntime.TargetFramework.Dump(); // Version 4.5 now
```

Client-side PBKDF

One obvious consequence of employing PBKDF and similar password-based KDF schemes to derive keys from low-entropy passwords is their intentionally high computational cost, measured in CPU utilization, memory utilization, or both. This cost can quickly become a limiting factor for server-side throughput (for example, measured in requests per second).

One tempting idea is moving PBKDF computation from the server side to the client side. Let us cover why this idea falls apart on closer inspection. Typical client-side devices tend to have inferior CPU and memory resources compared to server-side infrastructure. Client-side software is often not even able to fully utilize the client-side resources that are available. Many modern browsers are stuck in 32-bit mode, even when they are running in 64-bit environments (that is, they are unable to fully utilize available memory). Even when browsers are running in 64-bit mode, their computational capabilities are typically limited by their single-threaded JavaScript engines (they are unable to fully utilize available CPUs). Mobile client devices are often primarily battery powered, and are even further constrained in their resource capabilities.

The crucial implication of client-side resource inferiority is that the number of client-side PBKDF rounds that can be tolerated with acceptable usability is typically a lot lower compared to server-side execution. Any substantial reduction in the number of PBKDF rounds would weaken PBKDF security and circumvent its intended purpose.

Another problem is that PBKDF is salted (if it is not, then you are not using it right). Salt values are available on the server side (which makes server-side PBKDF straightforward), but not on the client side. Sending salt values to the client side would not only introduce additional complexity, but would weaken security as well. While salt values do not have to be secret, they should not be blissfully disclosed to anyone who asks, either. An adversary could ask for a salt value of an “Administrator” account and combine it with a list of common passwords to generate potential credentials and quickly mount an online attack. Disclosing salt values is clearly not in our best interests.

Password reset

Password reset protocols try to provide users with an “acceptably secure” means of resetting or changing the what-you-know authenticator (password) *without* providing a valid existing

authenticator, which undermines the entire purpose for such an authenticator. It is therefore important to realize that most password reset protocols are an exercise in intentional weakening of security of the system you are trying to protect, and are often the weakest link in the chain of security measures.

Password authenticators tend to be the only “proof” of account ownership in many systems, due to practicality (ease of use, ease of management, and ease of delegation and revocation). The only secure password reset protocol for these systems is really “*password change/reset requires proof of knowledge of existing password.*” If the existing password is lost, there is no way to correctly establish account ownership when the existing password is the only proof of ownership. The various password “alternatives” used in the wild—such as email, SMS, and automated phone calls—merely avoid the establish-account-ownership problem by outsourcing it to another external system, or by splitting it among several external systems (email, SMS, and voice call codes are all required for a password change or reset).

Most side channels used for automated identity confirmation and password resets provide weak or no security. The only mitigating factor that makes the risk of using these insecure side channels marginally plausible is a short time-to-live property of identity confirmation tokens sent through those channels. Another serious problem of typical side channels is that their security is often interdependent. A stolen smartphone is likely to allow an adversary to receive phone calls, SMS, and read owner’s email due to a logged-in email client.

If you have no choice and must weaken your system’s security for the sake of usability (asking users to create new accounts when they are unable to provide valid credentials to existing or claimed accounts is bad for business), then an email side channel will likely be the first thing you will try to leverage, since you are likely already using this channel for user communication. We will now describe how you should do it in the least damaging way.

There are two distinct phases in the email-based password reset process: token request and token validation.

Token request:

1. Ask for account-linked email address and CAPTCHA solution.
2. If CAPTCHA is solved and email matches precisely one account:
 - a. Generate a CSP-random token **T** (120 bits are enough) and its creation UTC timestamp **TS**.
 - b. Use **T** as key to calculate signature **SIG** = $\text{HMAC}_T(\text{account ID} + \text{TS})$.
 - c. Record **SIG** and **TS** in the database (but not **TK**).
 - d. Email **Base64(T)** and forget **T** as soon as possible.

Token validation:

1. Obtain **T** from user—for example, user clicks on email-delivered URL with embedded **Base64(T)**.

2. Obtain account ID or username from user (do not disclose in the email) and CAPTCHA solution.
3. If CAPTCHA is solved and **TS** (looked up for claimed account ID) is valid:
 - a. Calculate **SIG'** = **HMAC**_τ(account ID + **TS**).
 - b. Compare **SIG'** and **SIG** (looked up for claimed account ID). If **SIG'** = **SIG**:
 - i. Ask user to choose a new password.
 - ii. Reset **SIG** and **TS** to null against that account ID.
 - iii. Send “your password has been changed” notification email.

Always keep detailed audit records on password reset requests (such as an IP address or browser fingerprint).

You should not trust identity or credential management and verification to a third-party provider (such as Google, Twitter, Facebook, or Microsoft), even if initial integration seems attractive (easy). Outsourcing such key pillars of your security infrastructure is likely to cause problems later.

Strict Transport Security (STS)

Secure web applications (or any of their parts or fragments) should never be loaded over HTTP (TLS should be mandatory). Unfortunately, there is a big usability problem with the “we speak HTTPS only” approach. The public has been conditioned to ignore URI schemas and avoid explicitly specifying the protocol. It is “facebook.com” and “google.com” rather than “https://www.facebook.com” and “https://www.google.com”. Since the schema or protocol is rarely specified, the “default” protocol choice is delegated to the user agents. Unfortunately, most (all?) user agents default to “HTTP” rather than “HTTPS” as their default schema. Not supporting HTTP will improve security, but it will also make the web application inaccessible to a lot of users.

Most web applications choose to respond via HTTP as well to minimize usability issues. The HTTP-based responses have no content, and instead return the HTTP-301 (permanent redirect) to the HTTPS-equivalent URL. Responding to HTTP requests improves usability, but exposes users to the risk of eventually stumbling into a hostile or untrusted networking environment that can maliciously intercept and modify HTTP responses to compromise user sessions.

Strict Transport Security (STS) is a mechanism to minimize the likelihood of a casual user being attacked when (eventually) accessing an HTTP-responding web application over a hostile or untrusted network. STS adds a **Strict-Transport-Security** header to HTTP and HTTPS responses, which specifies a period of time during which the user agent should access the server over HTTPS only. The STS-compliant user agent is expected to remember the STS instruction and the duration (**max-age**) to enforce it. The enforced STS causes user agents to automatically turn any HTTP requests into equivalent HTTPS requests before they are made. This automatic client-side HTTPS conversion will save users who accidentally stumble into a

rogue network environment from being served malicious HTTP content. However, STS will not offer any protection to those user agents that have never accessed an HTTP-responding web application before, and for which the user agent has no prior enforced STS entry recorded.

One complimentary STS mechanism is STS [preloading](#), which is a way to submit HTTPS-speaking domains for user agent inclusion as being HTTPS-only. You can inspect a full list of preloaded STS participants for the Chrome browser [here](#).

There is practically no downside to using STS, and it is a low-effort and high-impact preventative security mechanism. The IIS configuration (**web.config**) to add one-year STS enforcement is:

Code Listing 24

```
<system.webServer>
...
  <httpProtocol>
    <customHeaders>
      <add name="Strict-Transport-Security" value="max-age=31536000;
        includeSubDomains; preload "/>
    </customHeaders>
  </httpProtocol>
...
</system.webServer>
```

Not all user agents support STS (yet), so an explicit HTTP-301 redirect might still be useful.

X-Frame-Options (XFO)

[Clickjacking](#) is a common attack technique where a user is tricked into unwittingly performing an otherwise undesired or unapproved action the user never intended. Clickjack victims are usually tricked into clicking on some UI elements from another web application that is presented in a transparent layer over what the user believes to be clicking on. This trick is often achieved via exploiting `<frame>` or `<iframe>` HTML tags.

The **X-Frame-Options** ([XFO](#)) header instructs compliant browsers to impose certain defensive restrictions on loading XFO-marked content into `<frame>` and `<iframe>` tags. There are three settings for XFO value:

- **DENY**
- **SAMEORIGIN**
- **ALLOW-FROM uri**

DENY prevents XFO-marked resources from being framed. **SAMEORIGIN** allows XFO-marked content to be framed only from the same origin as the XFO-marked content. **ALLOW-FROM** allows XFO-marked content to be framed only from an origin specified by the `uri`.

Sample IIS configuration to set XFO is:

Code Listing 25

```
<system.webServer>
...
  <httpProtocol>
    <customHeaders>
      <add name="X-Frame-Options" value="SAMEORIGIN" />
    </customHeaders>
  </httpProtocol>
...
</system.webServer>
```

The **ALLOW-FROM** XFO setting is not supported by most browsers, but **DENY** and **SAMEORIGIN** are widely supported.

Content-Security-Policy (CSP)

The **Content-Security-Policy** header ([CSP](#), not to be confused with cryptographic service provider from previous sections) is a more robust replacement for XFO with a richer defensive functionality. While XFO has decent support in modern browsers, it is deprecated as a standard, and is obsoleted by the **frame-ancestors** directive of CSP.

CSP is a very powerful policy-setting mechanism, but requires careful policy configuration, testing, and tuning to be effective. Non-trivial CSP policies also tend to be quite verbose and waste a lot of bytes on the wire, adding response bloat. A simple CSP policy might be:

Code Listing 26

```
Content-Security-Policy: default-src 'self'
```

This policy forces all content to be loaded from the document's own origin only. Sample IIS configuration to set CSP is:

Code Listing 27

```
<system.webServer>
...
  <httpProtocol>
    <customHeaders>
      <add name="Content-Security-Policy" value="default-src 'self';" />
    </customHeaders>
  </httpProtocol>
...
</system.webServer>
```

Subresource Integrity (SRI)

Most non-trivial web applications are assembled from resources (images, scripts, stylesheets, fonts, etc.) that are loaded from multiple [origins](#). Therefore, comprehensive web application security requires authenticity guarantees from all these origins as well, most of which will be outside your domain of control. If authenticity guarantees are absent, not only are you exposed to the risk of these external origins getting hacked and legitimate resources being maliciously replaced, but you can also suffer from DNS poisoning attacks replacing the IP addresses of these origins, which might be an easier attack than hacking the servers directly.

TLS (HTTPS) connections, Strict-Transport-Security ([STS](#)), and Public Key Pinning ([HPKP](#)) mechanisms all play important roles in comprehensive web security. However, they are focused on authenticating the server, but not the content. Subresource Integrity ([SRI](#)) is a relatively new mechanism that allows web applications to pin the content as well (by validating its authenticity).

Code Listing 28

```
<script
  src="https://ajax.googleapis.com/ajax/libs/jquery/2.2.2/jquery.min.js"
  integrity="sha384-
mXQoED/lfIuocc//nss8aJOIrz7X7XruhR6b0+sGceiSyMELoVdZkN7F0oYwcFH+"
  crossorigin="anonymous">
</script>
```

The **integrity** attribute on the **<script>** tag contains a **SHA384** hash of the binary payload loaded from the **src** attribute. SRI-enabled user agents (e.g., browsers) will refuse to load any resources that fail a hash match. One way to calculate the SRI hash of a resource is as follows:

```

using (var http = new HttpClient())
using (var dataStream = await http.GetStreamAsync(
    "https://ajax.googleapis.com/ajax/libs/jquery/2.2.2/jquery.min.js"))
using (var hash = SecurityDriven.Inferno.Hash.HashFactories.SHA384())
{
    byte[] hashBytes = hash.ComputeHash(dataStream);
    string hash_b64 = Convert.ToBase64String(hashBytes);
    $"sha384-{hash_b64}".Dump();
}

// result:
// sha384-mXQoED/lFIuocc//nss8aJ0Irz7X7XruhR6b0+sGceiSyMELoVdZkN7F0oYwcFH+

```

Alternatively, you can use a convenient and untrusted online generator at srihash.org.

JSON Object Signing and Encryption (JOSE) framework

JOSE is a set of related specifications for encrypting and signing content, and making it portable and transferrable over the web. JOSE has many moving parts, namely:

- JSON Web Signature (JWS) – [RFC 7515](http://tools.ietf.org/html/rfc7515)
- JSON Web Encryption (JWE) – [RFC 7516](http://tools.ietf.org/html/rfc7516)
- JSON Web Key (JWK) – [RFC 7517](http://tools.ietf.org/html/rfc7517)
- JSON Web Algorithms (JWA) – [RFC 7518](http://tools.ietf.org/html/rfc7518)
- JSON Web Token (JWT) – [RFC 7519](http://tools.ietf.org/html/rfc7519)
- JOSE Cookbook containing examples – [RFC 7520](http://tools.ietf.org/html/rfc7520)

One of the JOSE coauthors (who is a Microsoft standards architect) described it as a “4.5-year journey to create a simple JSON-based security token format and underlying JSON-based cryptographic standards, with the goal to keep things simple.”

While JOSE specifications might indeed be simpler than some other Microsoft-backed standards such as SAML 2.0 (coauthored by the same Microsoft architect), they are far from simple (as 5+ RFCs suggest), and are more comparable to TLS RFCs in terms of specification complexity. The parallels between TLS and JOSE can also be seen in the sheer number of

possible combinations of various JOSE components, some of which might be secure, with the JOSE Cookbook stating that *“the full set of permutations is extremely large, and might be daunting to some.”* That is not how good security specifications that aim to “keep things simple” should start.

Continuing our TLS analogy, we can draw parallels between securing data at the transport layer (which is what TLS does), and similar attempts to secure data at the application layer (which is what JOSE tries to do via JSON).

There are very few high-quality TLS libraries out there, and unless you are a big browser-maker or browser-backer, or have Amazon or CloudFlare scale, writing a TLS library should not be on your agenda. Similarly, despite JOSE’s purported goal of keeping things simple, custom JOSE implementations are likely fraught with security vulnerabilities. JOSE was standardized in 2015, so it is still too new for any particular implementation to withstand the test of time (which is one of the key tests for security specifications).

If you are considering using some parts of JOSE, here are some things to keep in mind:

- JOSE makes heavy use of JSON serialization and deserialization, which (in .NET) is CPU-expensive, memory-expensive, and GC-expensive.
- JOSE does not efficiently encode binary data, or any other non-string data, and needlessly wastes a lot of storage, memory, and bytes-over-the-wire.
- JOSE (JSON) does not support lazy, just-in-time access to individual fields—the entire JSON structure must be de-serialized first.
- JOSE encryption (JWE) is optional (another shortcoming), and is often not used. JWT (tokens) are usually signed but not encrypted.
- JWE symmetric encryption modes are limited to AES-CBC (good conservative choice, but not as good as CTR, for example). The MAC tag is at least 32 bytes long (overkill that wastes storage).
- JOSE follows an “a la carte” approach in the name of algorithmic agility. We know how well that same “a la carte” approach worked for TLS, where it is a constant source of vulnerabilities. The TLS 1.3 working group is finally wising up and tightening the 1.3 spec to use fewer AEAD-only schemes without all broken “features” of prior versions. The algorithmic agility of JOSE is creating [security bugs](#) already.

Cookies

Some people make a false assumption that HTTP cookie-based authentication is an equivalent to server-side state, where the cookie only contains the session ID, and all session information is keyed on that ID and stored server-side. However, **Cookie** (request) and **Set-Cookie** (response) headers are just specific HTTP headers, and do not mandate any particular authentication mechanism or approach. Cookies are actually described in [RFC 6265](#) – “HTTP State Management Mechanism”, which states:

The Cookie header contains cookies the user agent received in previous Set-Cookie headers. The origin server is free to ignore the Cookie header or use its contents for an application-defined purpose.

It is that “application-defined purpose” that makes the cookie mechanism so useful, and the magic of cookies that makes them unlike any other HTTP header is the following capability:

Using the Set-Cookie header, a server can send the user agent a short string in an HTTP response that the user agent will return in future HTTP requests that are within the scope of the cookie.

This capability to send some data or state from the server to the client and have the client’s user agent (browser) automatically and transparently send that data or state back to the server on all subsequent requests is a great way of implementing client-side state (stateless from the server’s perspective).

Another excellent cookie capability is the **HttpOnly** attribute, which instructs the user agent to prohibit access to cookies via any non-HTTP API (specifically, JavaScript). We can use JavaScript APIs to read every HTTP header—except the **HttpOnly**-marked cookie header. Most other web defenses fall apart under cross-site scripting ([XSS](#)) or JavaScript-injection exploits, but not the **HttpOnly** cookies, which might be weakened (since many other attacks will become possible under XSS), but will not allow cookies (often containing user’s authenticated identity) to be hijacked.

XSS attacks are the bane of the modern web (lots of user-contributed content), and **HttpOnly** cookies are a great way of offloading state to the client without suffering the consequences. XSS attacks are usually not a matter of “if,” but a matter of “when”. If you are storing identity-bearing tokens client-side and are using some client-side storage other than **HttpOnly** cookies (for example, HTML5 local/session storage, or cookies without **HttpOnly**), you are effectively placing your “castle keys” outside the castle walls—are you really that confident?

HTTP/2

[HTTP/2](#) (RFC 7540) is the next version of the HTTP/1.1 protocol. HTTP/1.0 was released in 1996, followed by HTTP/1.1 in 1999, and HTTP/2 in 2015—16 years later. HTTP/2 is a major revision with significant differences from HTTP/1.1. Some of these differences are:

Table 11: HTTP/1.1 vs. HTTP/2

HTTP/1.1	HTTP/2
One connection per request (with browsers pooling 6-8 connections per origin)	One connection per origin
Textual, e.g., “GET /index.html HTTP/1.1”	Binary (i.e. much more byte-efficient)

HTTP/1.1	HTTP/2
Ordered and blocking (request-response)	Fully multiplexed (like TCP/IP)
Headers are repeated with every request	Headers are compressed (HPACK method)
Encryption is optional (HTTP vs. HTTPS)	Encryption is mandatory (sort of)
Client-pull only (explicit URI requests)	Server push (proactive data sending to client)

The HTTP/2 specification does not make encryption mandatory, but all current HTTP/2-supporting browsers will only establish HTTP/2 with TLS, which makes encryption practically mandatory with HTTP/2. Another important HTTP/2 security advance is that HTTP/2 implementations must use TLS 1.2 or higher. Therefore, using HTTP/2 automatically avoids broken or vulnerable old TLS versions. HTTP/2 is making the web faster and safer, so adopting HTTP/2 should be on your agenda. Microsoft supports HTTP/2 in IIS 10.0, which is part of [Windows Server 2016](#) OS. To see HTTP/2 in action, visit the [HTTP vs. HTTPS Test](#) and [Akamai demo](#) sites.