Pitfalls of High-Level Cryptography in .NET

Stan Drapkin

April 2018

Who am I?

- Stan Drapkin – sdrapkin@sdprime.com
- CTO of IT firm (cybersecurity & regulatory compliance)
- OSS library author (github.com/sdrapkin)
  - **Inferno** – *.NET crypto done right*
  - **TinyORM** – *.NET micro ORM done right*
- Book author
  - “SecurityDriven .NET” (2014)
  - “Application Security in .NET, Succinctly” (2017)

I will talk about

Stages of crypto enlightenment

Symmetric crypto

LL vs HL crypto

Streaming crypto

Asymmetric & Hybrid crypto

RSA “fun”

Modern EC crypto

4 stages of crypto enlightenment

“If you think cryptography is the answer to your problem, then you don’t know what your problem is.”

Dr. Peter G. Neumann

A better solution might not need crypto at all

Symmetric crypto

Low-level (LL) crypto API dangers

- "Pitfalls of `System.Security.Cryptography`" talk
  Vladimir Kochetkov, 2015 (on YouTube)
- Every step of LL crypto is filled with decisions that
  You are not aware you need to make
  You are not qualified to make
- One of key takeaways:
  Avoid LL crypto. Use HL crypto instead.
- But do **you** know what HL crypto API is, or should be?

Symptoms of non-HL crypto library

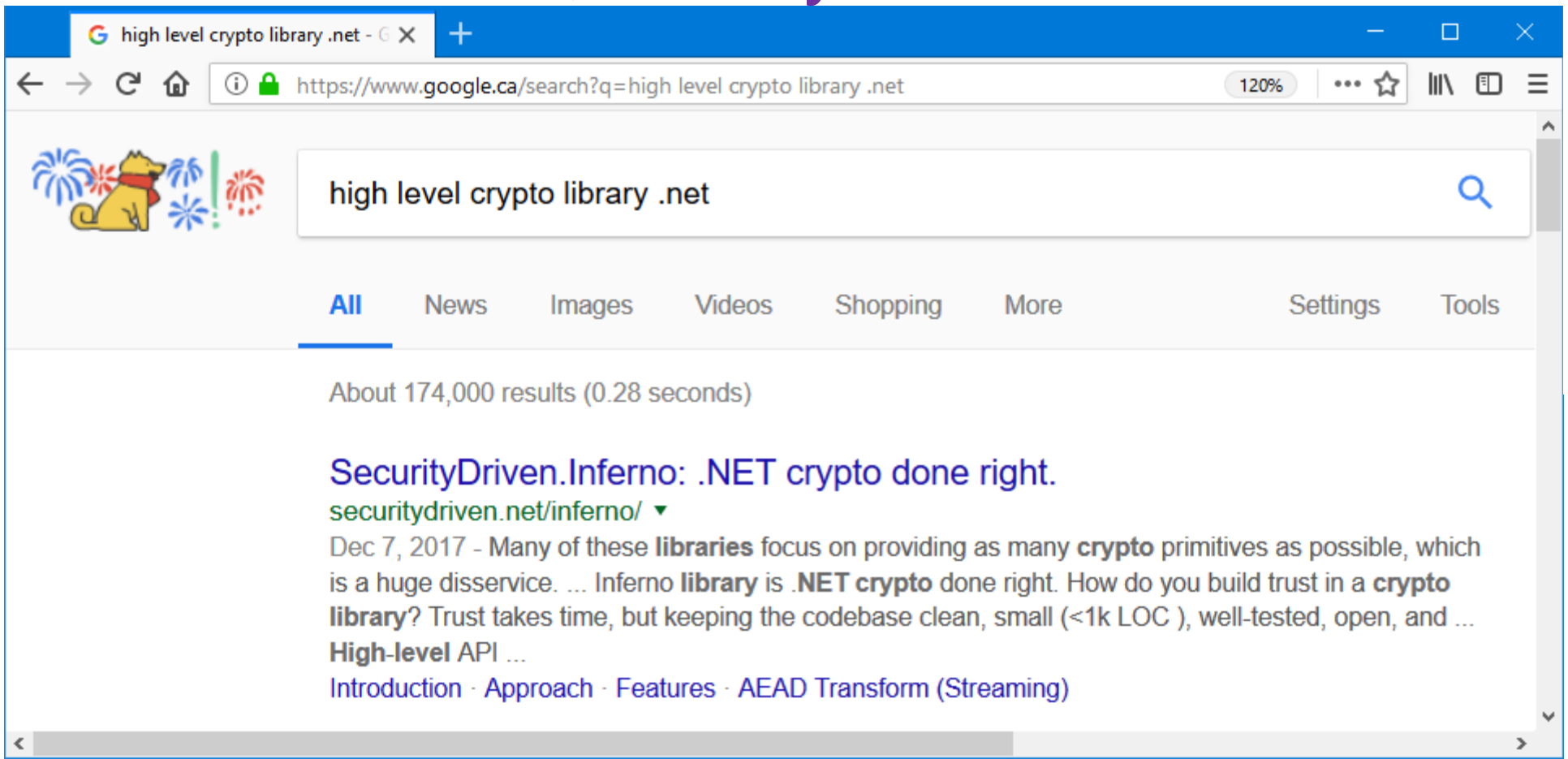
- API doesn't feel .NET-native (feels like a LL wrapper)
- API is easy to misuse
- Forces you to generate weird LL things (Nonces, IVs)
- Forces you to make uncomfortable decisions
  Algorithms, padding modes, key/nonce/IV/tag sizes, etc
- Lacks good streaming API

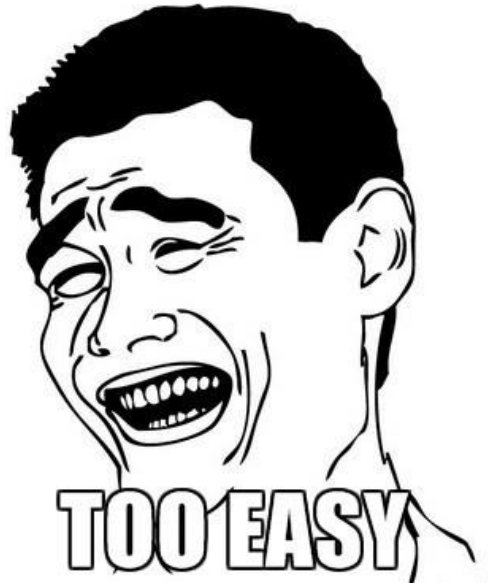
What is HL crypto API?

- Intuitive and eloquent to read and write
- Easy to learn
- Easy to use
- Hard to misuse
- Powerful (achieves the objective with little effort)
- Low-friction (just works – no caveats/constraints)

How do we find a HL crypto library for .NET?

- We could Google.. But that's too easy.
- Let's "research", and try alternatives..





Authenticated Encryption: basic concepts

Plaintext P
(variable length)

$\overline{\text{N}}$	Reuse of nonce **N** under same key **K** - Compromises confidentiality of plaintext (a bad thing™)
AD	**A**ssociated **D**ata - Optional data which is authenticated, but not encrypted

Libsodium.NET – the glorious choices

AES-GCM with 96-bit nonce
550 GB/key; 64 GB/msg; 2^{32} msg limit

ChaCha20/Poly1305 with 64-bit nonce
64-bit nonce is too small
SHOULD NOT BE USED AT ALL

xSalsa20/Poly1305 with 192-bit nonce
Missing AD (Associated Data)

Libsodium.NET – more questions

What to do with Nonce?

Manually append/prepend to ciphertext. Somehow.

AD can have any length, right?

16 bytes max.

What happens on decryption failure?

Exception is raised.

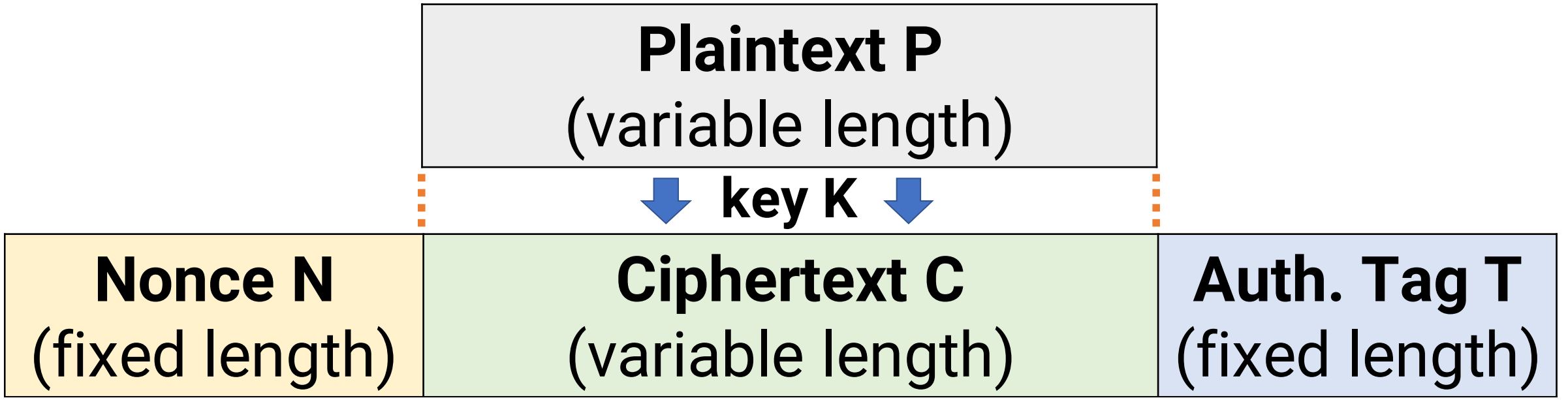
What if my key is not exactly 32 bytes?

Your problem. Libsodium keys must be exactly 32 bytes.

Can I reuse byte arrays to relieve GC pressure?

No.

Authenticated Encryption: comparison



AES-GCM	**96**	**128** (<128)	$\overline{\text{N}}$→forgery of all **C** under same **K**
Chacha/Poly	**64**	**128** (<106)	$\overline{\text{N}}$→forgery of all **C** under $\overline{\text{N}}$
xSalsa/Poly	**192**	**128** (<106)	$\overline{\text{N}}$ not probable
Inferno	**320**	**128** (128)	$\overline{\text{N}}$ not probable; no forgeries ✓

Libsodium.NET purpose – follow the docs

- "Libsodium.NET is a c# wrapper around **libsodium**"
- "Libsodium is a fork of **NaCl** with compatible API"
- "NaCl's goal is to provide all of the core operations needed to **build higher-level cryptographic tools**"

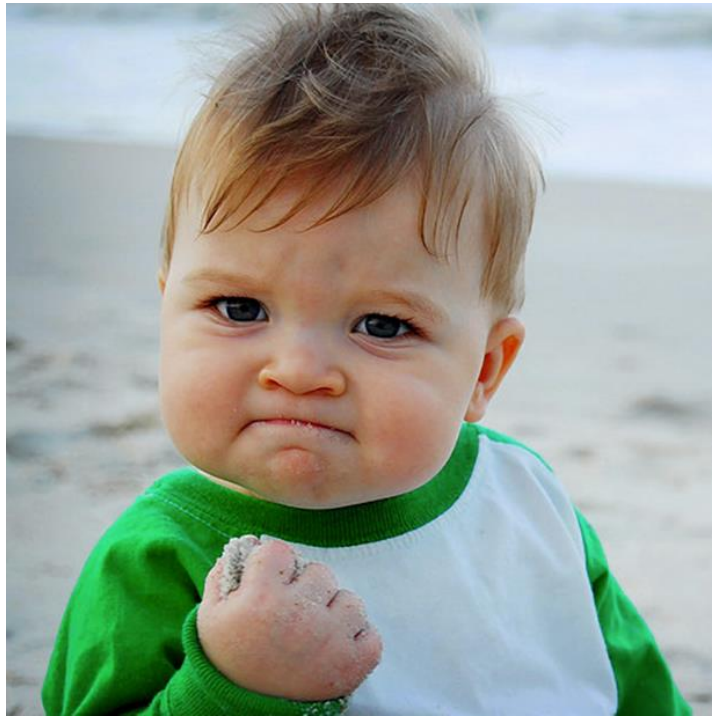
- If you need a HL crypto → pick a good HL crypto lib

  Don't take a LL lib wrapper, and pretend it is HL

Inferno

```
c = SuiteB.Encrypt(key, p, ad);        // ad is optional, ofc
d = SuiteB.Decrypt(key, c, ad);
```

- No nonces
- No decisions
- Decrypt error → d is null
  nothing is thrown



Let’s encrypt some strings – should be easy

- Only 2 possible values: “LEFT” and “RIGHT”

```
c1 = SuiteB.Encrypt(key, “LEFT”);
c2 = SuiteB.Encrypt(key, “RIGHT”);
```

- This is production-ready. Right?

- What is the problem? How can we fix it?

Let’s encrypt some strings – should be easy

- Length leaking – that’s not a “real” problem.. Right?

JANUARY 23, 2018:

“TINDER’S LACK OF ENCRYPTION LETS STRANGERS SPY ON YOUR SWIPES”

Swipe-left = 278 bytes

Swipe-right = 374 bytes

http://images.gotinder.com

CHECKMARX

Let’s encrypt a file – how hard can it be?

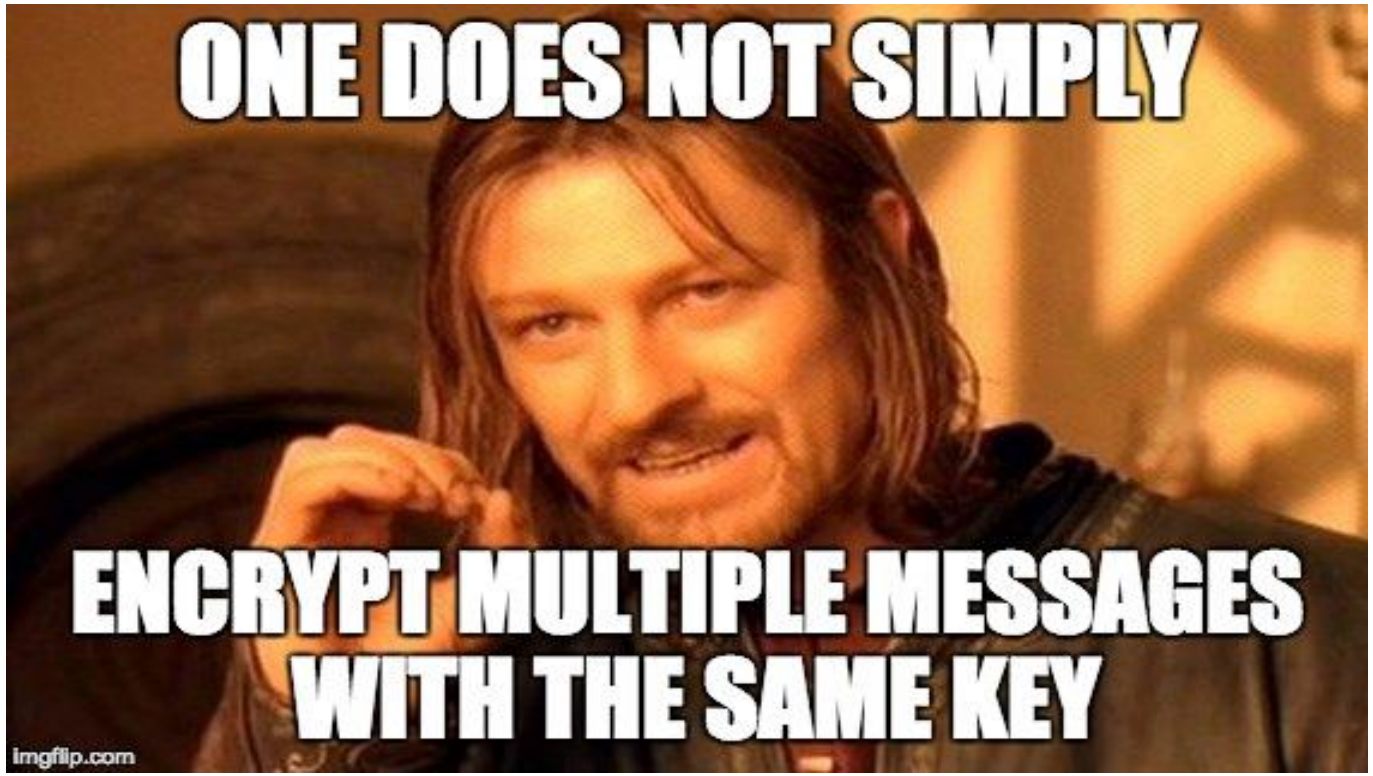
- Libsodium.NET: not supported
- Inferno:

```
using (var fsource = new FileStream(“fname.txt”, FileMode.Open))
using (var ftarget = new FileStream(“fname.enc”, FileMode.Create))

using (var t = new EtM_EncryptTransform(key)) //  ← Inferno is used
using (var cryptoStream =
        new CryptoStream(ftarget, t, CryptoStreamMode.Write))
        await fsource.CopyToAsync(cryptoStream);
```

HL crypto – message limits with fixed key

- Inferno: 2^{112} messages of 2^{64} blocks (ie. **no limit**)
- Libsodium.NET: depends. 2^{38} or 2^{64} bytes



https://blog.cloudflare.com/tls-nonce-nse/

Associated Data (AD) – different notions

- Weak: AD is not participating in enc/dec.
- Strong: AD is required for (ie. alters) enc/dec.

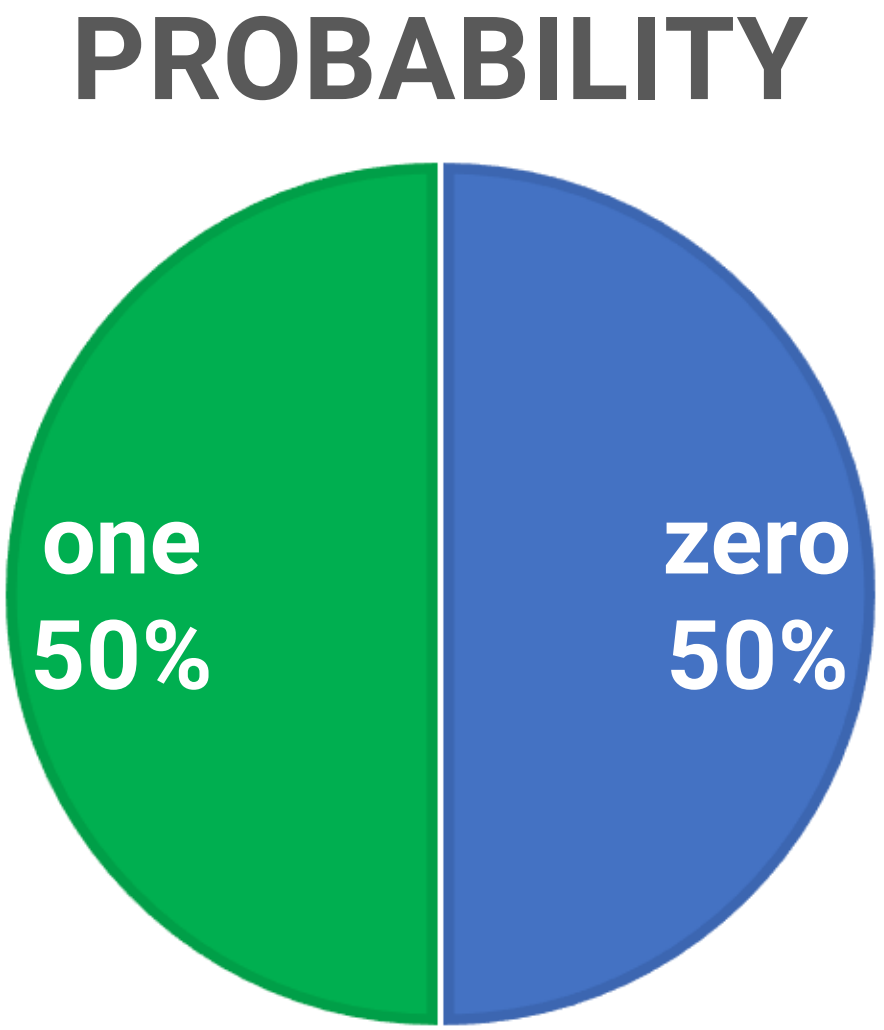
- Inferno uses “strong” AD (AD → encryption tweak)
- Most other libraries use “weak” AD

Which security level should HL crypto target?

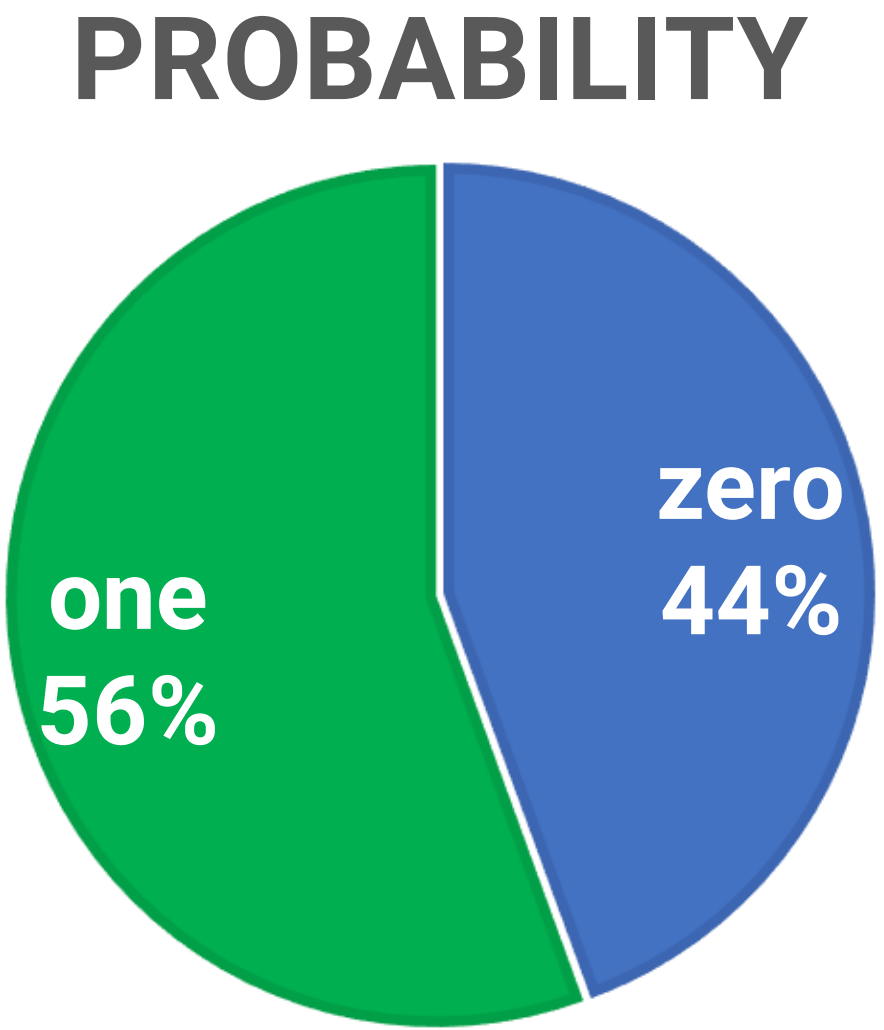
- 256-bit encryption, with 128-bit authentication tag.
- Why do we need 256-bit keys?
- To allow for potential **biases** in CSRBG key creation.

- What is bias?

No bias (good Random Bit Generator)



25% bias (biased Random Bit Generator)



Which security level should HL crypto target?

- 256-bit encryption, with 128-bit authentication tag.
- Why do we need 256-bit keys?
- To allow for potential biases in CSRBG key creation.
- **N**-bit RBG entropy = $-\mathbf{LOG}_2(\frac{1}{2} + |\mathbf{bias}|) * \mathbf{N}$
- **25%** bias over 128-bit key → 53 bits of entropy

  Broken
- **25%** bias over 256-bit key → 106 bits of entropy

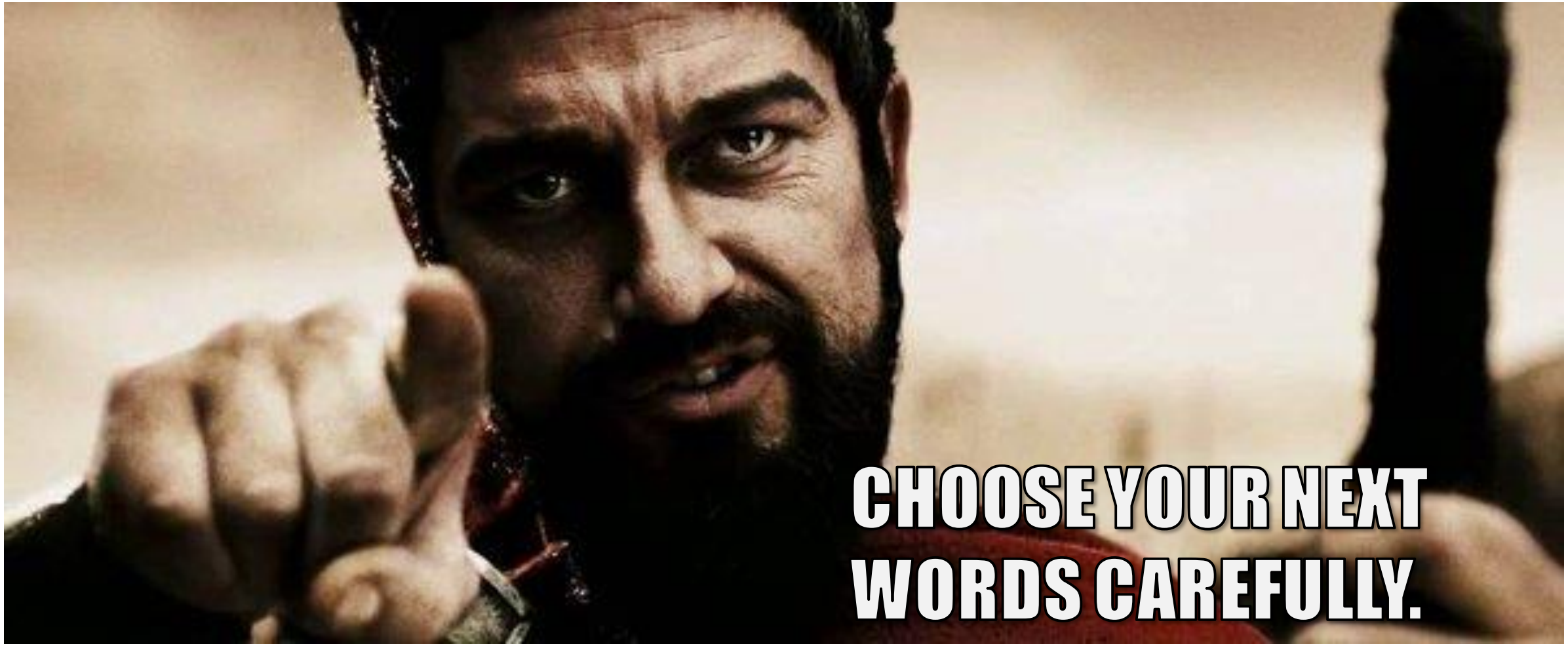
  Practically unbreakable

Symmetric crypto – summary

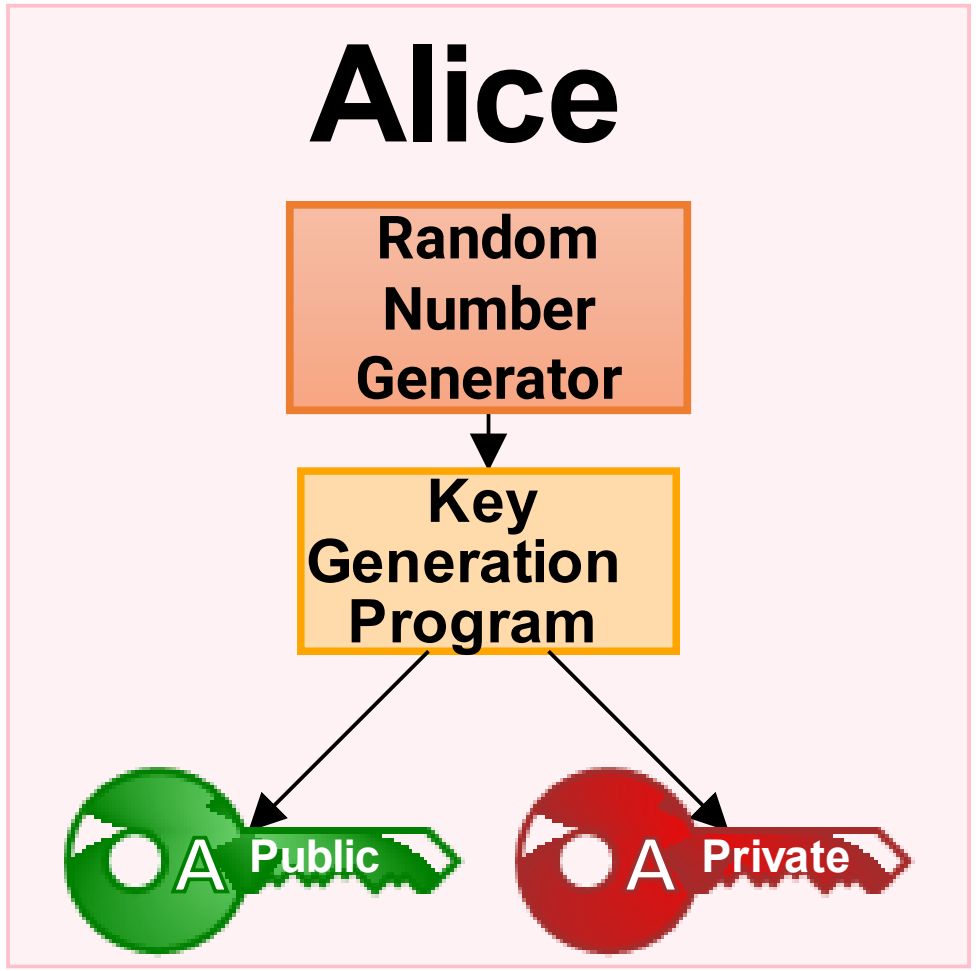
- Use a well-designed HL crypto library
- Encrypt streaming data with streaming crypto API
- Use 256-bit random keys (minimum length)
- HL crypto can leak confidentiality

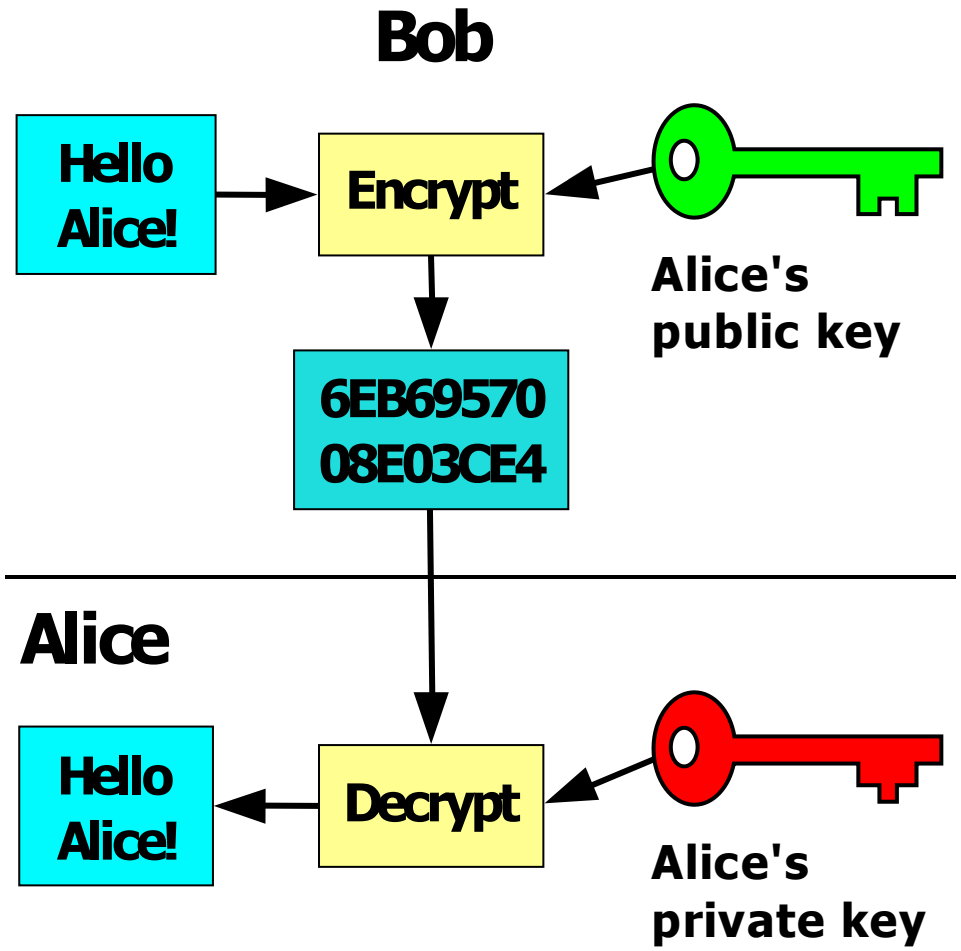
Asymmetric & Hybrid crypto

RSA. You **do know** how to use it. Right?



RSA encryption: Quick refresher (Wikipedia)





RSA. Can you use it? Yes? Let’s test that.

```
var rsa = RSA.Create();
var c = rsa.Encrypt(p, paddingMode);  //select mode
```

#1: Pkcs1
#2: OaepSHA1
#3: OaepSHA256
#4: OaepSHA384
#5: OaepSHA512

RSA. Can you use it? Yes? Let's test that.

```
var rsa = RSA.Create();
var c = rsa.Encrypt(p, paddingMode);  //select mode
```

#1: Pkcs1 → does not throw

#2: OaepSHA1 → does not throw

#3: OaepSHA256 → throws "padding not valid" ex.

#4: OaepSHA384 → throws "padding not valid" ex.

#5: OaepSHA512 → throws "padding not valid" ex.

RSA. Can you use it? Yes? Let's test that.

```
var rsa = RSA.Create();
var c = rsa.Encrypt(p, OaepSHA1);
```

- What RSA key size did we just use?

```
WriteLine(rsa.KeySize); // care to guess?
```

- We're going to set the key size explicitly..

RSA. Can you use it? Yes? Let's test that.

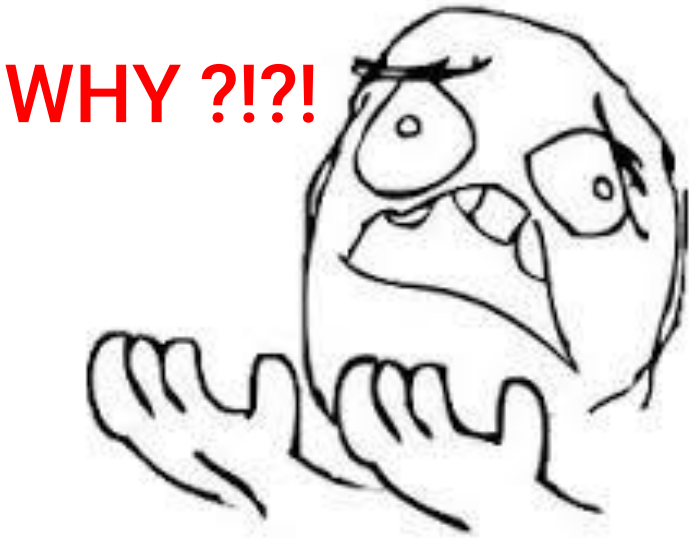
MS docs for ".KeySize":
"Gets or sets the size, in bits, of the key modulus, used by the asymmetric algorithm." Perfect, let's use it.

```
var rsa = RSA.Create();
rsa.KeySize = 3072; // proceed to encrypt secrets

WriteLine(rsa.KeySize); // care to guess?
rsa.ExportParameters(false).Modulus.Length * 8
```

RSA. Can you use it? Yes? Let's test that.

- RSACng class – brand new in .NET 4.6
- Cng = Cryptography Next Generation!




```
var rsa = RSACng.Create();
rsa.KeySize = 3072;

WriteLine(rsa.KeySize); // 1024
WriteLine(rsa.GetType()); // RSACryptoServiceProvider
```

RSA. Can you use it? Yes? Let's test that.

```
var rsa = new RSACng(); // must use ctor directly
WriteLine(rsa.KeySize); // 2048, a better default
rsa.KeySize = 3072;
WriteLine(rsa.KeySize); // 3072 !!!
```

Achievement unlocked!

RSA. Default key sizes. Or are they?

- RSA.Create() return type can be set in machine.config
  - → RSA implementation could be changed on you
  - → Default keysize could be changed on you

- Never trust RSA defaults!

  Set explicit keysize. Always.

RSA. Default key sizes. How good are they?

```
var rsa1 = new RSACryptoServiceProvider(); // 1024
var rsa2 = new RSACng(); // 2048
```

Basic operation?

BitCoin Network (BCN) hashrate ≈ 2^{64} hashes/**second**

≈ 2^{90} hashes/**year** (as of February 2018)

≈ 2^{70} “basic ops” can break RSA-1024 (1 BCN minute)

≈ 2^{90} “basic ops” can break RSA-2048 (1 BCN year)

- Use explicit RSA keysize! (3072 or 4096 bits)

RSA. How to export the public/private keys?

```
var rsa1 = new RSACryptoServiceProvider(4096);
var rsa2 = new RSACng(4096);

var kPub1 = rsa1.ExportCspBlob(includePrivateParameters: false); //532 bytes
var kPub2 = rsa2.Key.Export(CngKeyBlobFormat.GenericPublicBlob); //539 bytes

var kPrv1 = rsa1.ExportCspBlob(includePrivateParameters: true); // 2324 bytes
var kPrv2 = rsa2.Key.Export(CngKeyBlobFormat.GenericPrivateBlob); // 1051 bytes
```

- 2 incompatible import/export APIs; be consistent

RSA. Can you use it? Let's try to encrypt..

```
var data = new byte[640];
rsa.Encrypt(data, OaepSHA1);
```

CryptographicException: The parameter is incorrect.

- Trying all padding types… The same exception for all.
- Data it is. What's wrong with the data? Let's half it:

```
var data = new byte[320];
rsa.Encrypt(data, OaepSHA1); // seems to work..
```

RSA. Can you use it? Let's try to encrypt..

- We are told that "SHA1-anything" is bad
- Let's switch padding from OaepSHA1 to OaepSHA256

```
var data = new byte[320];
rsa.Encrypt(data, OaepSHA256);
```

CryptographicException: The parameter is incorrect.

- Data size limit is a function of **padding** and **keysize**!

RSA. Can you use it? Yes? Let’s test that.

- Is there a magic formula for max data size? YES!
- You should use

int GetMaxDataSizeForEnc(RSAEncryptionPadding pad)

…which does not exist.

Basic information to use RSA correctly is not available.

Reasons to avoid RSA, even for signatures.

- Poor .NET API
- Forces you to make decisions (padding, data length)
- RSA-4096 is needed for 128-bit security level
  - (priv / pub / sig) = (1051 / 539 / 512) bytes
- RSA-15360 is needed for 256-bit security level
  - Unusable (keygen alone takes 1.25 minutes on my laptop)
- Slow key generation, and slow signing
  - TLS: SIGN is on the Server (slow); VERIFY is on the Client


SAY “RSA”
ONE MORE TIME...

Modern Elliptic-Curve (EC) crypto primitives.

- **ECDSA** – Digital Signature Algorithm
  - replaces RSA signatures
  - code: securitydriven.net/inferno/#DSA Signatures
- **ECIES** – Integrated Encryption Scheme
  - replaces RSA hybrid encryption
  - code: securitydriven.net/inferno/#ECIES example
- **ECDH** – Diffie-Hellman key exchange
  - creates symmetric-encryption keys; forward secrecy
  - code: securitydriven.net/inferno/#DHM Key Exchange

Summary

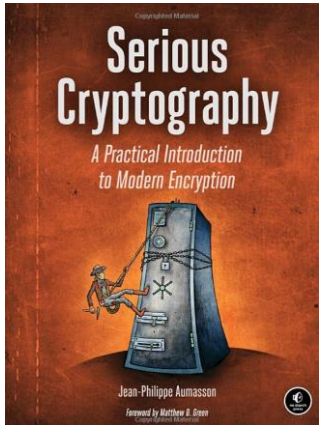
- Use HL crypto API that does not require decisions.
- Abandon RSA. If you can't – learn to use it correctly.
- Get comfy with ECDSA/ECDH/ECIES (future talk?).
- Think about your goals – HL crypto doesn't cure all.

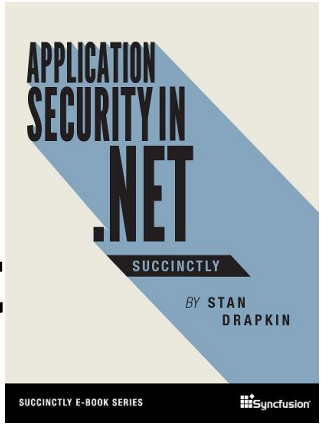
"Cryptography doesn't solve problems by itself. Symmetric encryption merely turns your data confidentiality problem into key management problem."

CodesInChaos (StackOverflow)

Recommended resources

- **SecurityDriven.Inferno** (documentation)
  decent HL crypto lib for .NET
- **Serious Cryptography**
  great overview of modern crypto
- **Application Security in .NET, Succinctly**
  free ebook covering more .NET security pitfalls
- **slideshare.net/kochetkov.vladimir/appsec-net**
  simplified AppSec theory
  explains causes of “insecurity” vs “lack of safety”





Thank you for your attention!

Questions?

sdrapkin@sdprime.com
twitter.com/sdrapkin
github.com/sdrapkin

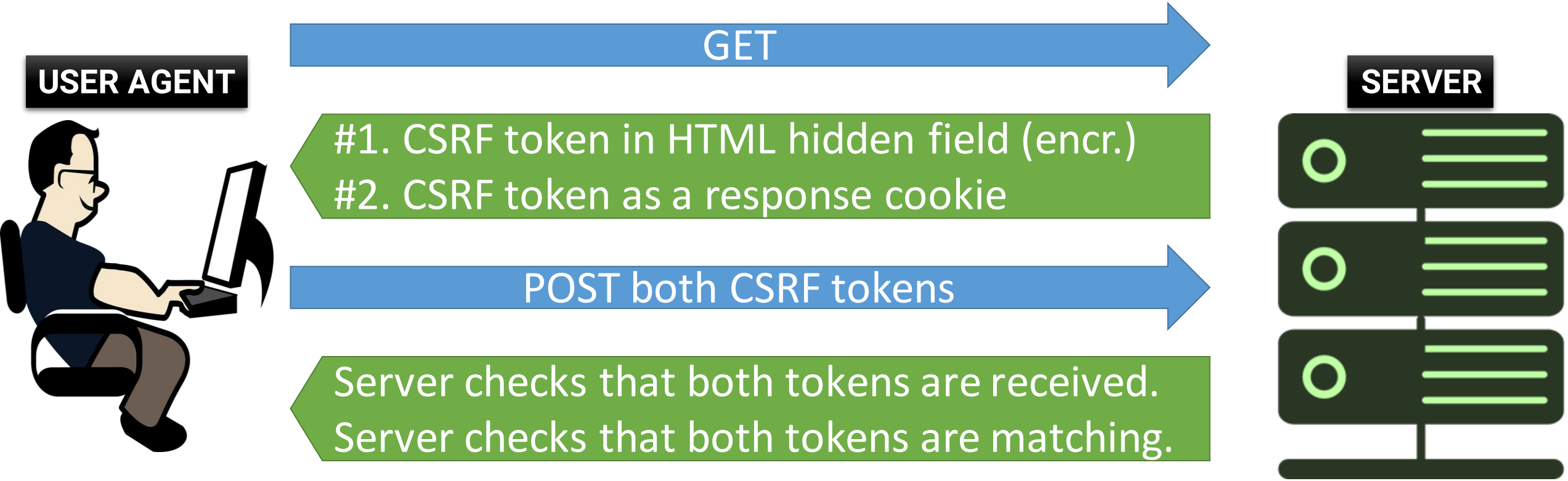
Bonus slides

You said…

"…a better solution might not need crypto at all"

What does that mean? An example, perhaps?

CSRF – classic .NET protection



CSRF token generation/validation uses encryption

→ complex, expensive (cpu, memory, latency, etc.)

→ HTML token injection is complicated, messy, inconvenient

Preventing CSRF **without** crypto

#1 set-cookie:

S=7TWFDB5YR7MX3Z1AK4FB2D7ZJXX3DCWEGQG4S4PHMQ91BE5Y; **HttpOnly**

#2 set-cookie:

T=7TWFDB5YR7MX3Z1AK4FB2D;

- 30 random bytes → Base32 → 48 chars. 22-char prefix = CSRF token **T**.
- CSRF token + 26-char secret = 48-char Session **S**. Each char = 5 bits of entropy.
- CSRF token = 110 bits (22*5). Secret Session part = 130 bits (26*5). NIST ✓
- No crypto at all
- HTML is untouched
- Developers don't need to do anything