Rolling your own Crypto

Why you shouldn't

Jules Poon

whoami

- **2** Years *security* experience:
  - **Centre for Strategic Infocomm Technologies:** Did RE and Tool dev
  - **STAR Labs:** Died reversing XNU and kernel extensions

- **1** Years playing cyber competitions (**Social Engineering Experts**)
  - Did Crypto and RE (and a little exploitation)

- **0** Years being a professional cryptographer

Contents

Contents

WarpConduit Computing's *Amazing* Protocol

WarpConduit Computing

QUICK TIPS WEB DESI

HOME WORDPRESS PLUGINS PASSWORD GENERATOR ABOUT CONTACT

Highly Secure Data Encryption & Decryption Made Easy with PHP, MCrypt, Rijndael-256, and CBC

April 14, 2013 by Josh Hartman

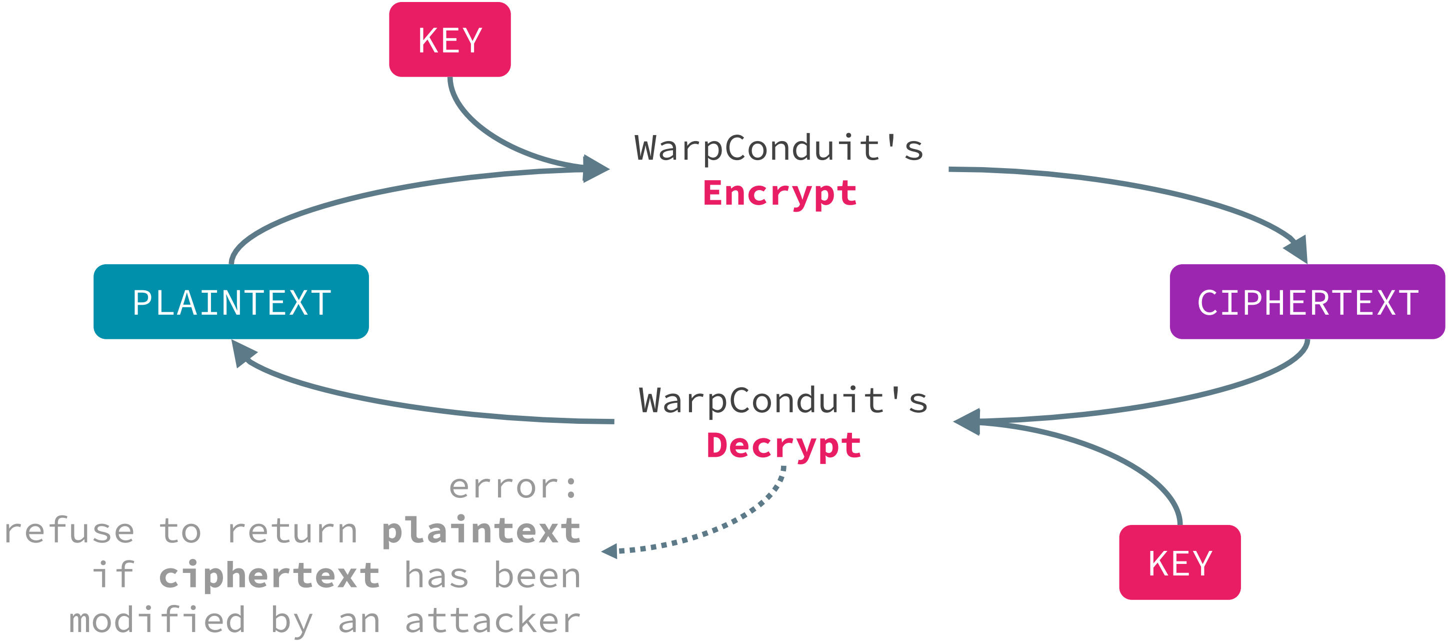
In various projects in the past I've had to revisit the topic of data encryption and decryption and the best way to accomplish it. In the interest of developing in the simplest, most efficient, and most secure way I have chosen the MCrypt PHP library (built-in to PHP since v4.0.2), Rijndael-256 cipher, and the Cipher Block Chaining (CBC) mode.

Published: 14/04/2013
Retrieved: 02/10/2022

Encapsulation that does ***integrity-authenticated*** **symmetric encryption**

Attackers cannot change the ciphertext to decrypt into a different plaintext

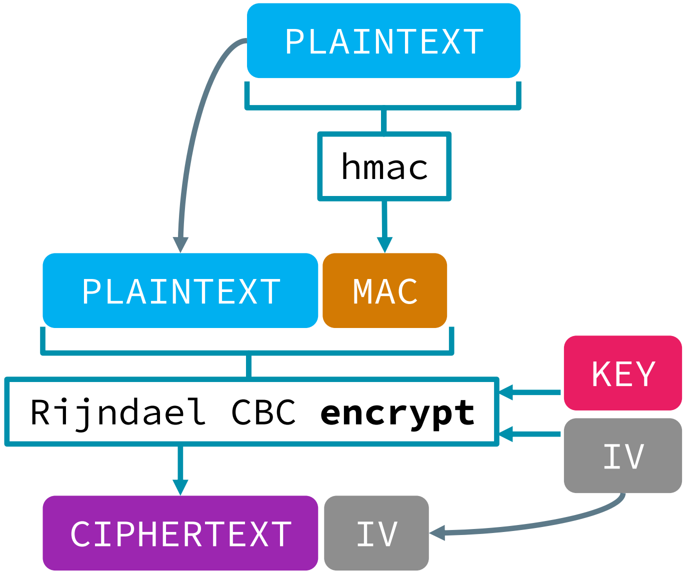
WarpConduit Computing's *Amazing* Protocol



WarpConduit Computing's *Amazing* Protocol

```
function mc_encrypt($encrypt, $key){
    $encrypt = serialize($encrypt);
    $iv = mcrypt_create_iv(
        mcrypt_get_iv_size(
             MCRYPT_RIJNDAEL_256,
             MCRYPT_MODE_CBC),
        MCRYPT_DEV_URANDOM);
    $key = pack('H*', $key);
    $mac = hash_hmac(
             'sha256', $encrypt,
             substr(bin2hex($key), -32));
    $passcrypt = mcrypt_encrypt(
        MCRYPT_RIJNDAEL_256,
        $key, $encrypt.$mac,
        MCRYPT_MODE_CBC, $iv);
    $encoded = base64_encode($passcrypt)
             .'|'
             .base64_encode($iv);
    return $encoded;
}
```

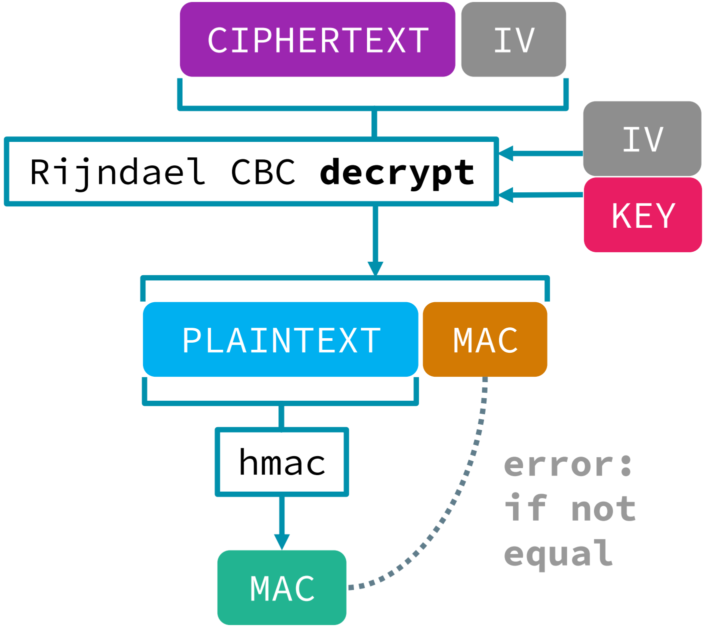
Encrypting



WarpConduit Computing's *Amazing* Protocol

```
function mc_decrypt($decrypt, $key){
    $decrypt = explode('|', $decrypt.'|');
    $decoded = base64_decode($decrypt[0]);
    $iv = base64_decode($decrypt[1]);
    if (strlen($iv)
        !== mcrypt_get_iv_size(
            MCRYPT_RIJNDAEL_256,
            MCRYPT_MODE_CBC)){return false;}
    $key = pack('H*', $key);
    $decrypted = trim(mcrypt_decrypt(
        MCRYPT_RIJNDAEL_256, $key,
        $decoded, MCRYPT_MODE_CBC, $iv));
    $mac = substr($decrypted, -64);
    $decrypted = substr($decrypted, 0, -64);
    $calcmac = hash_hmac(
        'sha256', $decrypted, s
        ubstr(bin2hex($key), -32));
    if($calcmac!==$mac){return false;}
    $decrypted = unserialize($decrypted);
    return $decrypted;
}
```

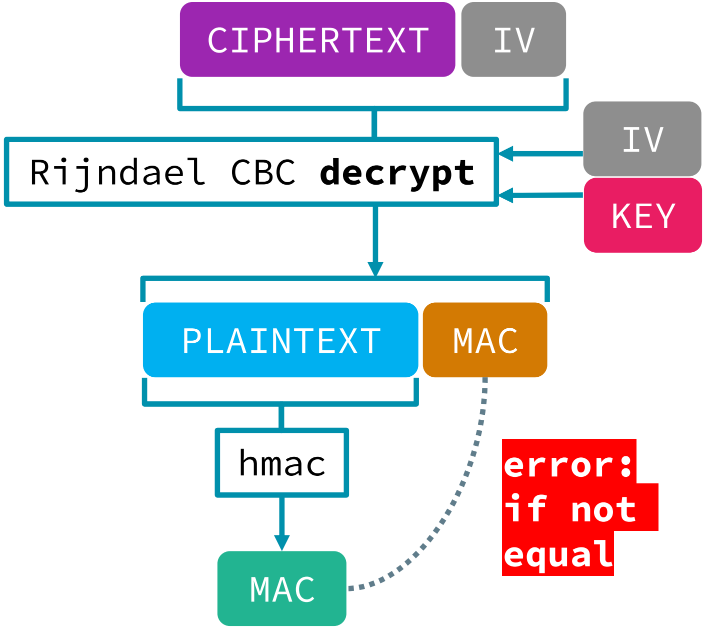
Decrypting



WarpConduit Computing's *Amazing* Protocol

MAC check prevents modifying of ciphertext

An attacker that modifies the ciphertext cannot compute a new **MAC** without knowing the **KEY**, so the check fails.



Reasoning by LEGO

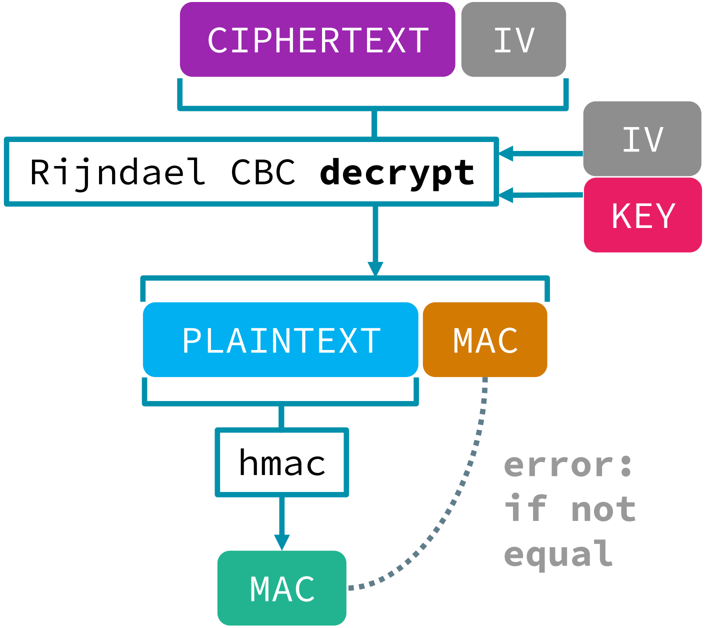
Coined by *Taylor Hornby* @DefuseSec

Reasoning by LEGO

Primitives used:
- Rijndael-256
- SHA-256

Both these **primitives are secure**

- protocol is just a bunch of **secure primitives wired together**
- **protocol is secure**



Reasoning by LEGO

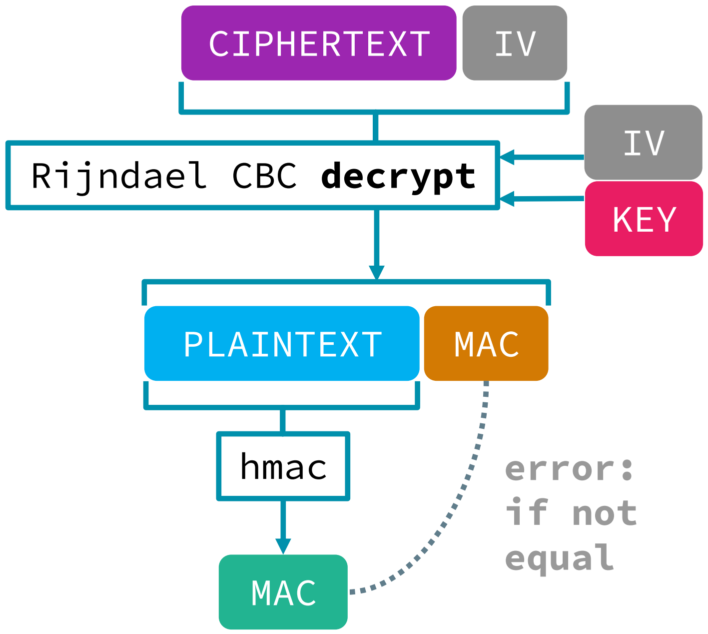
What this protocol *should* provide:

Privacy:

- **Rijndael-256** is secure so no problem

Integrity:

- **SHA-256** is secure so no problem



Reasoning by LEGO

- Reasoning by LEGO is good for implementation
    - Concept of abstraction in software development

Reasoning by LEGO

- Reasoning by LEGO is good for implementation
  - Concept of abstraction in software development
- Reasoning by LEGO is **wrong** for cryptanalysis

Reasoning by LEGO

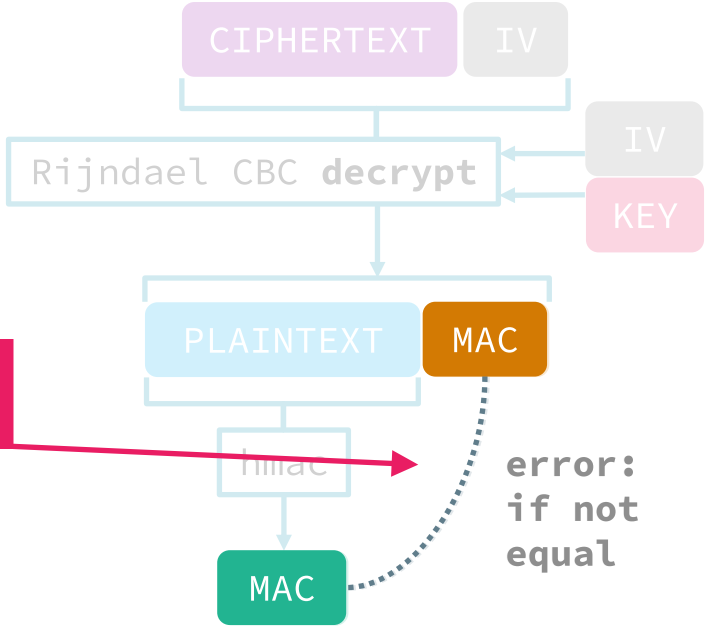
- Reasoning by LEGO is good for implementation
    - Concept of abstraction in software development
- Reasoning by LEGO is **wrong** for cryptanalysis
- One **must consider the whole system together**

WarpConduit Computing's *Amazing* Protocol: The Vuln

```
function mc_decrypt($decrypt, $key){
    $decrypt = explode('|', $decrypt.'|');
    $decoded = base64_decode($decrypt[0]);
    $iv = base64_decode($decrypt[1]);
    if (strlen($iv)
        !== mcrypt_get_iv_size(
            MCRYPT_RIJNDAEL_256,
            MCRYPT_MODE_CBC)){return false;}
    $key = pack('H*', $key);
    $decrypted = trim(mcrypt_decrypt(
        MCRYPT_RIJNDAEL_256, $key,
        $decoded, MCRYPT_MODE_CBC, $iv));
    $mac = substr($decrypted, -64);
    $decrypted = substr($decrypted, 0, -64);
    $calcmac = hash_hmac(
        'sha256', $decrypted, s
        ubstr(bin2hex($key), -32));
    if($calcmac!==$mac){return false;}
    $decrypted = unserialize($decrypted);
    return $decrypted;
}
```

Not *constant time* compare

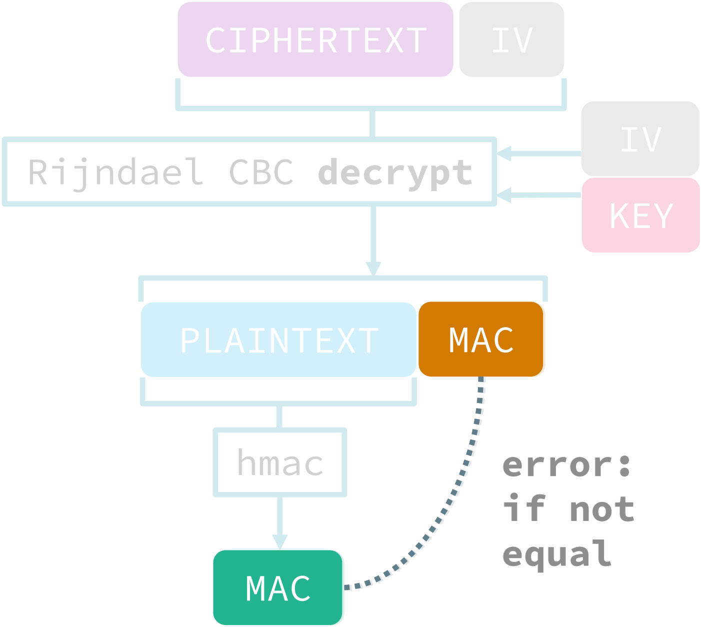
We can leak info about the **plaintext**!



WarpConduit Computing's *Amazing* Protocol: The Vuln

Reasoning by LEGO:

- A vuln here shouldn't have *anything* to do with the **plaintext** (only the **MAC**).
- Mayyybe **integrity** is compromised but not **privacy**??



WarpConduit Computing's *Amazing* Protocol: The Vuln

Not constant-time string compare:

Your Code

```
if (!strcmp(secret, attacker_input))
    ERROR("...");
/* ... */
```

Glibc source

```
int
strcmp(const char *p1, const char *p2)
{
    const unsigned char *s1 = (const unsigned char *)p1;
    const unsigned char *s2 = (const unsigned char *)p2;
    unsigned char c1, c2;
    do
    {
        c1 = (unsigned char)*s1++;
        c2 = (unsigned char)*s2++;
        if (c1 == '\0')
            return c1 - c2;
    } while (c1 == c2);
    return c1 - c2;
}
```

Early stopping:
First char wrong
-> function returns
faster

WarpConduit Computing's *Amazing* Protocol: The Vuln

Web Timing Attacks Made Practical*

Timothy D. Morgan† Jason W. Morgan‡

August 3, 2015

Abstract

This paper addresses the problem of exploiting timing side channels in web applications. To date, differences in execution time have been difficult to detect and to exploit. Very small differences in execution time induced by different security logics, coupled with the fact that these small differences are often lost to significant network noise, make their detection difficult. Additionally, testing for and taking advantage of timing

WarpConduit Computing's *Amazing* Protocol: The Vuln

		Delta (ns)				
Scenario	Classifier	25000	5000	1000	200	40
lnx						
	midsummary	29 obs	**894 obs**	17147 obs	**16.60% err**	**38.60% err**
	quadsummary	26 obs	**894 obs**	**16289 obs**	20.55% err	47.30% err
	septasummary	**15 obs**	**894 obs**	17147 obs	22.35% err	45.20% err
	boxtest	146 obs	20.80% err	36.30% err	47.55% err	49.85% err

WarpConduit Computing's *Amazing* Protocol: The Vuln

Not constant-time string compare: [Aside]

Glibc source [x8664 arch optimised impl]

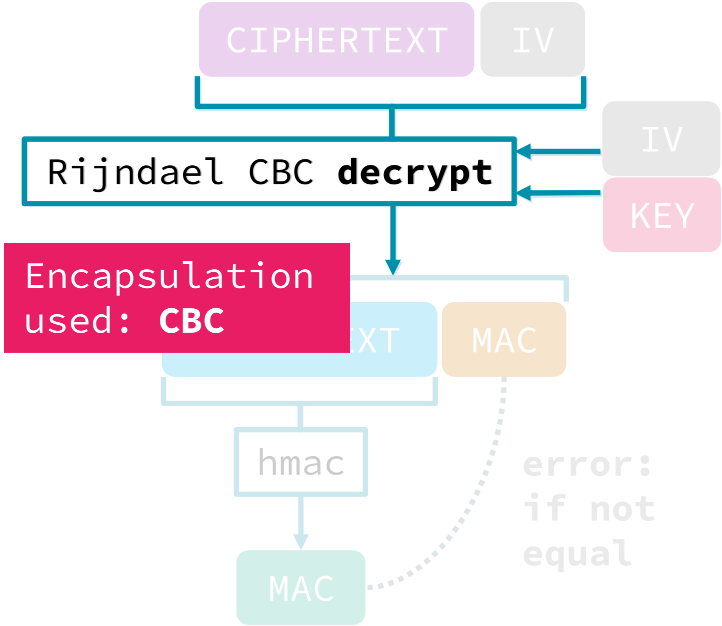
```
L(no_page_cross):
    /* Safe to compare 4x vectors.  */
    VMOVU   (%rdi), %ymm0
    /* 1s where s1 and s2 equal. Just VPCMPEQ if its not strcasecmp.
       Otherwise converts ymm0 and load from rsi to lower. ymm2 is
       scratch and ymm1 is the return.  */
    CMP_R1_S2_ymm (%ymm0, (%rsi), %ymm2, %ymm1)
    /* 1s at null CHAR.  */
    VPCMPEQ %ymm0, %ymmZERO, %ymm2
    /* 1s where s1 and s2 equal AND not null CHAR.  */
    vpandn  %ymm1, %ymm2, %ymm1
    /* All 1s -> keep going, any 0s -> return.  */
    vpmovmskb %ymm1, %ecx
```

Compares 32 bytes at once:
Timing attack is way harder here

WarpConduit Computing's *Amazing* Protocol: The Attack

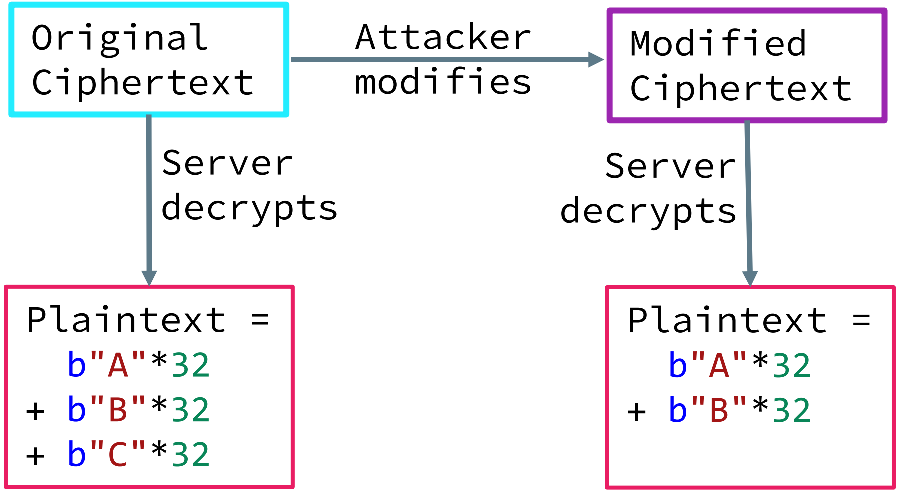
[Aside] Properties of the *Encapsulation* used:

- Rijndael-256 encrypts in blocks of 32 bytes
- To encrypt longer data it uses **CBC Encapsulation**



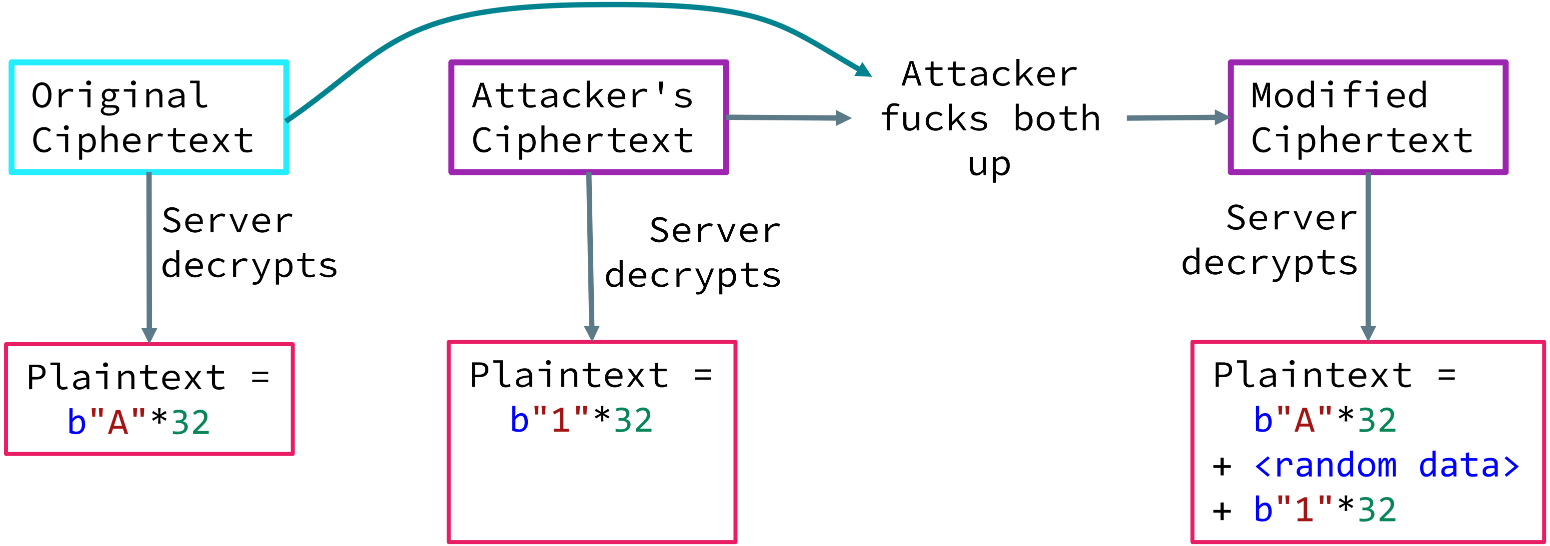
WarpConduit Computing's *Amazing* Protocol: The Attack

CBC: Can truncate plaintext by 32 bytes by modifying ciphertext **without the key.**



WarpConduit Computing's *Amazing* Protocol: The Attack

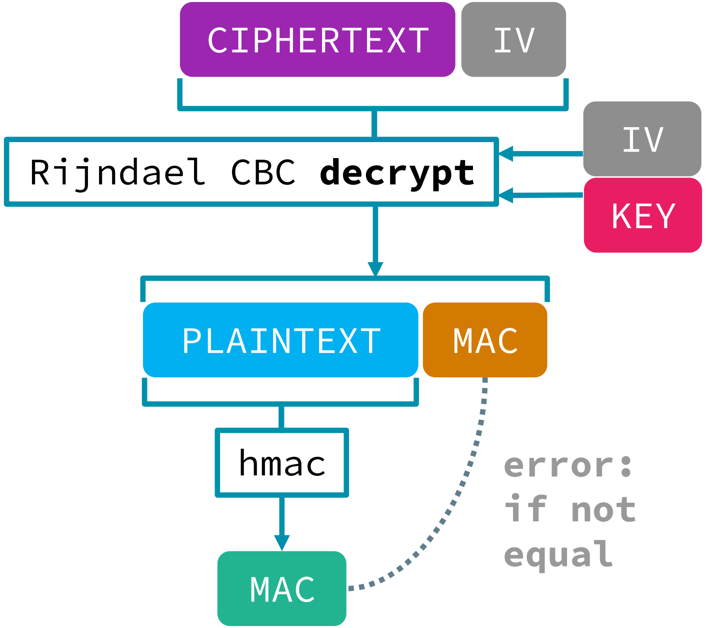
CBC: Can combine two ciphertexts such that the plaintext concats **without the key.**



WarpConduit Computing's *Amazing* Protocol: The Attack

Attack Scenario:

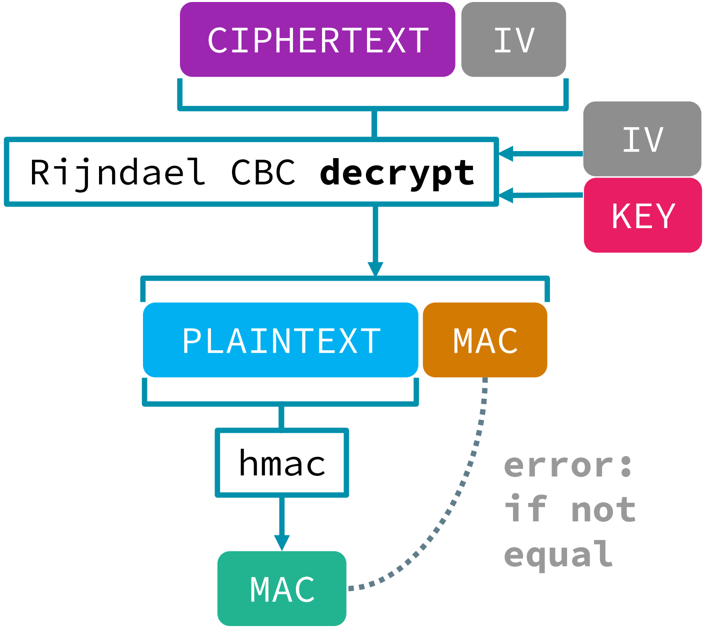
- Server encrypts whatever attacker sends
- Server decrypts whatever attacker sends **but doesn't return**
- Goal: recover secret given its ciphertext



WarpConduit Computing's *Amazing* Protocol: The Attack

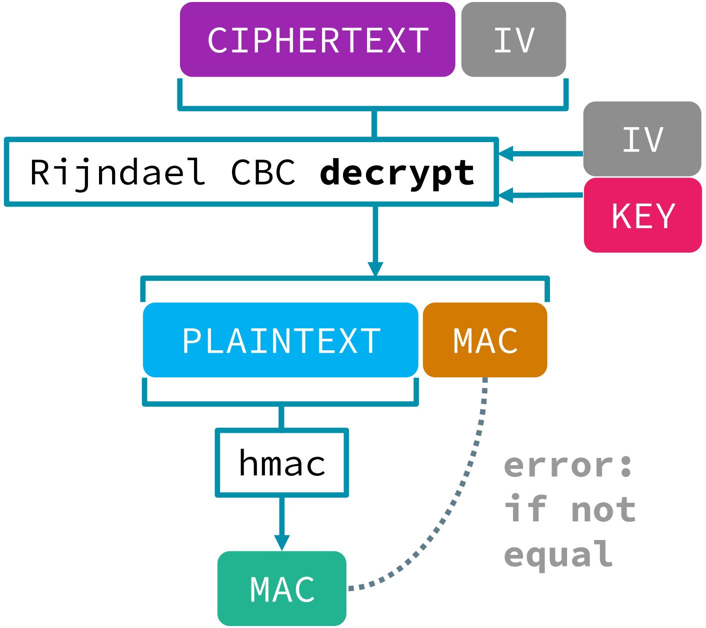
Assumptions on attacker:

1. Attacker can measure time taken by server to compare **MAC** to tell how many bytes were matched



WarpConduit Computing's *Amazing* Protocol: The Attack

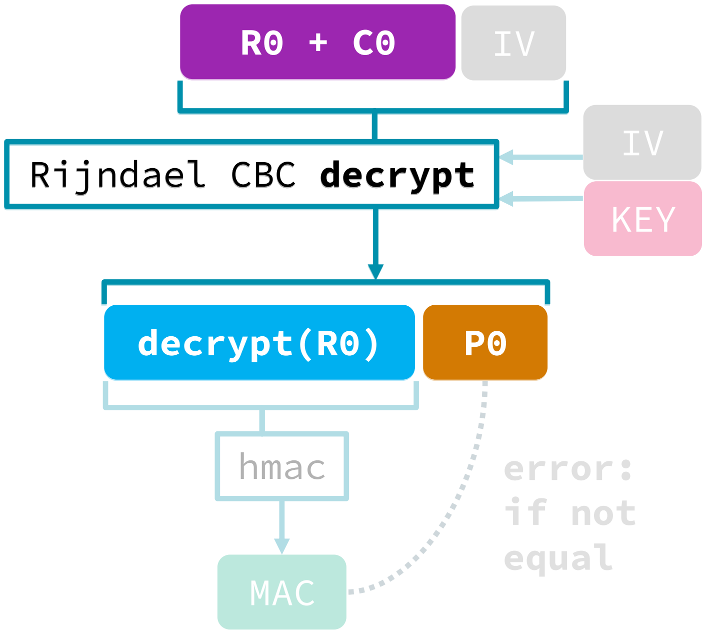
- Not an unrealistic scenario
  - E.g. Secret hidden in session cookie



WarpConduit Computing's *Amazing* Protocol: The Attack

ATTACK:

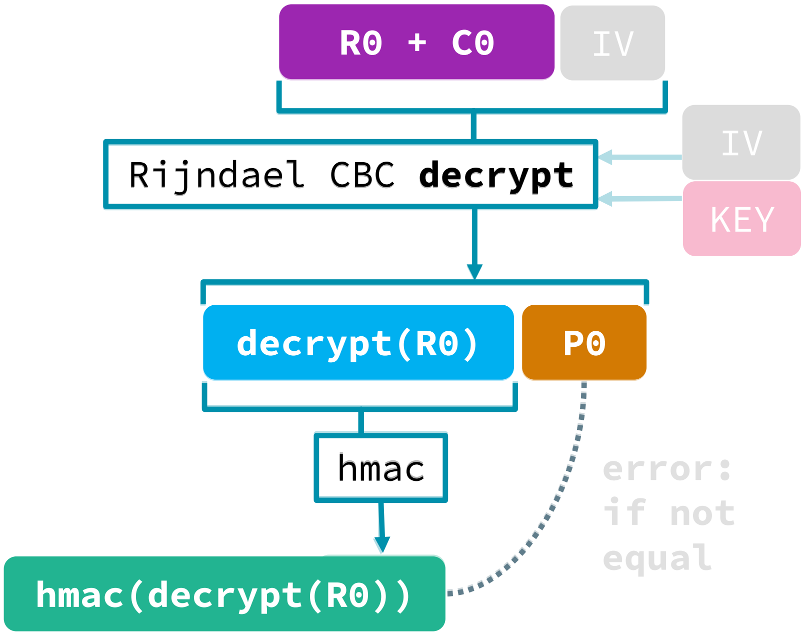
- Get 256 ciphertexts whose plaintext starts with b"\x00" to b"\xff" (**A0, A1, ... A255**) by requesting the server. **decrypt(Ax)[0] = x**
- Take first 32 bytes of encrypted secret **C0**, prepend with randomly generated block **R0**
  - Tricks server to take first 32 bytes of secret (**P0**) as **MAC**



WarpConduit Computing's *Amazing* Protocol: The Attack

ATTACK:

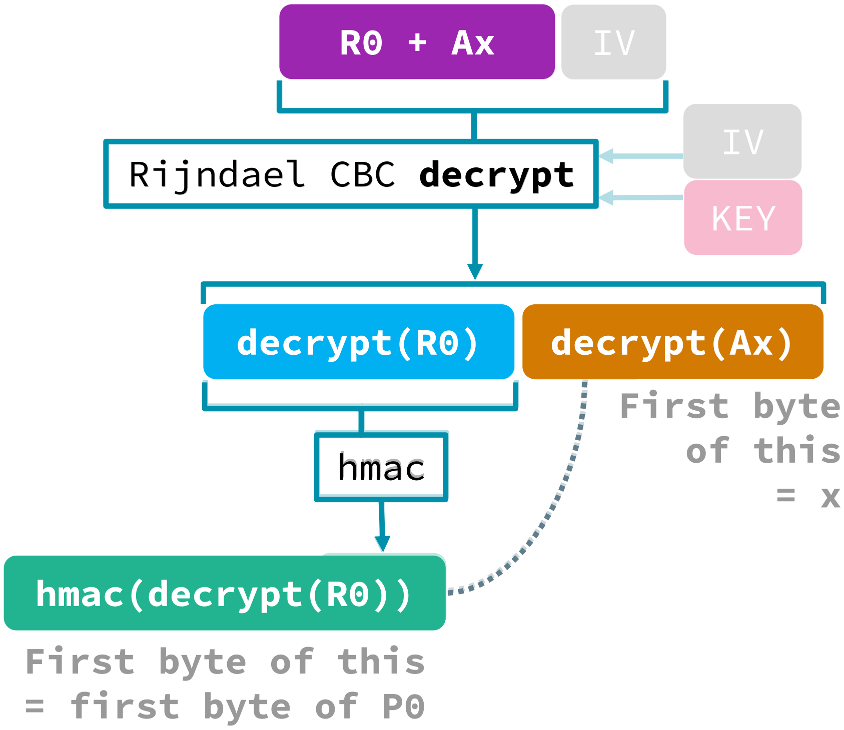
- Try different **R0** until **hmac(decrypt(R0))[0]** = **P0[0]**
  - Attacker can tell via timing attack



WarpConduit Computing's *Amazing* Protocol: The Attack

ATTACK:

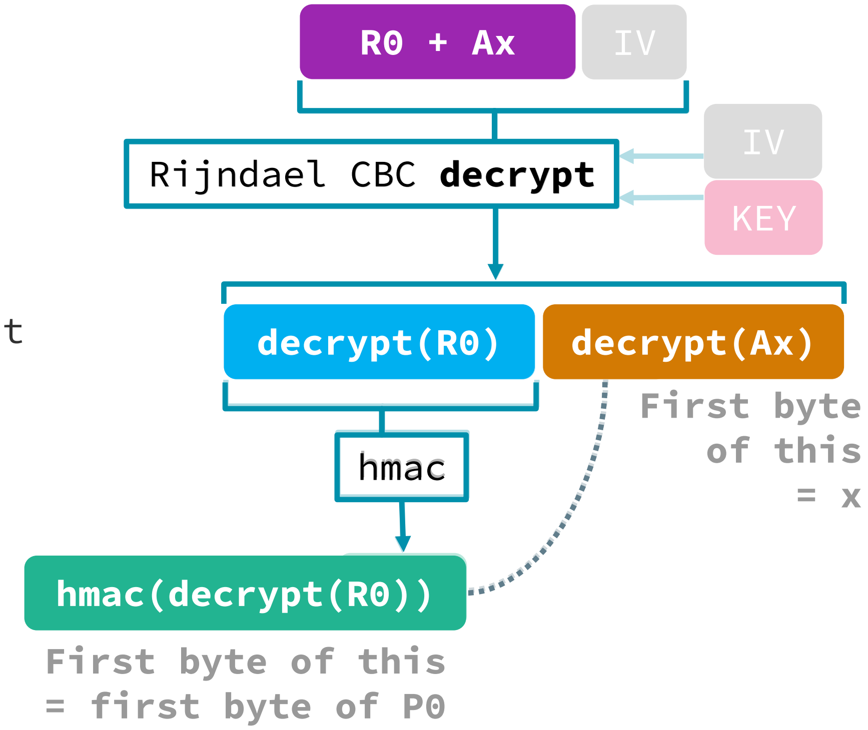
- Swap **C0** with **A0,...,A255** until **decrypt(Ax)[0] = x = decrypt(R0)[0] = P0[0]**
  - E.g. If **P0**'s first byte is 42, using **A42** will make **hmac** comparison take longer
- **P0[0]** recovered!



WarpConduit Computing's *Amazing* Protocol: The Attack

ATTACK:

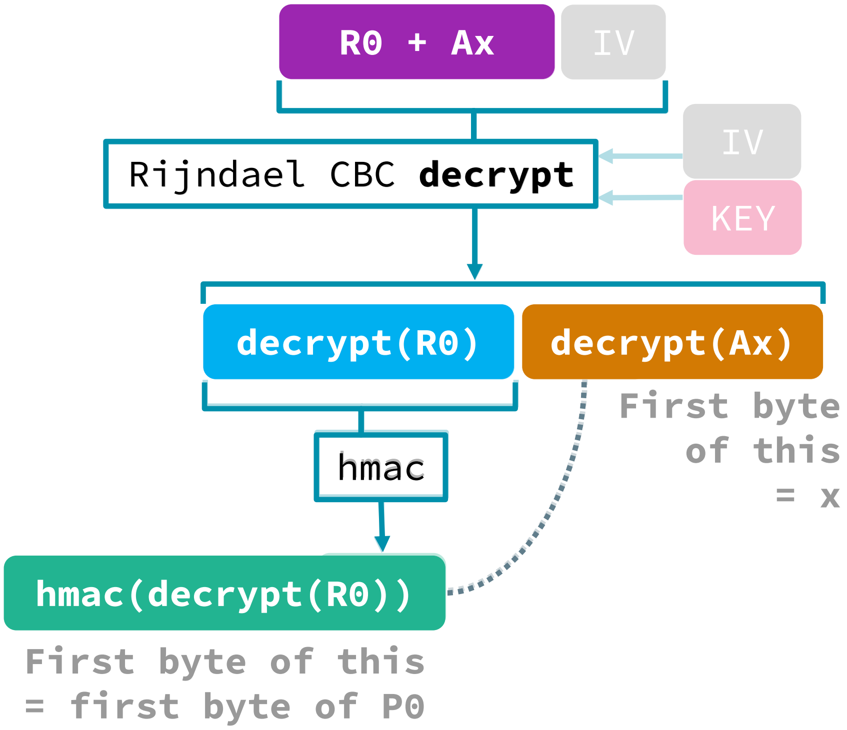
- Can be extended to leak first few bytes of every 32-byte block of plaintext
  - Potentially all bytes of plaintext if one can control block alignment.



WarpConduit Computing's *Amazing* Protocol: The Attack

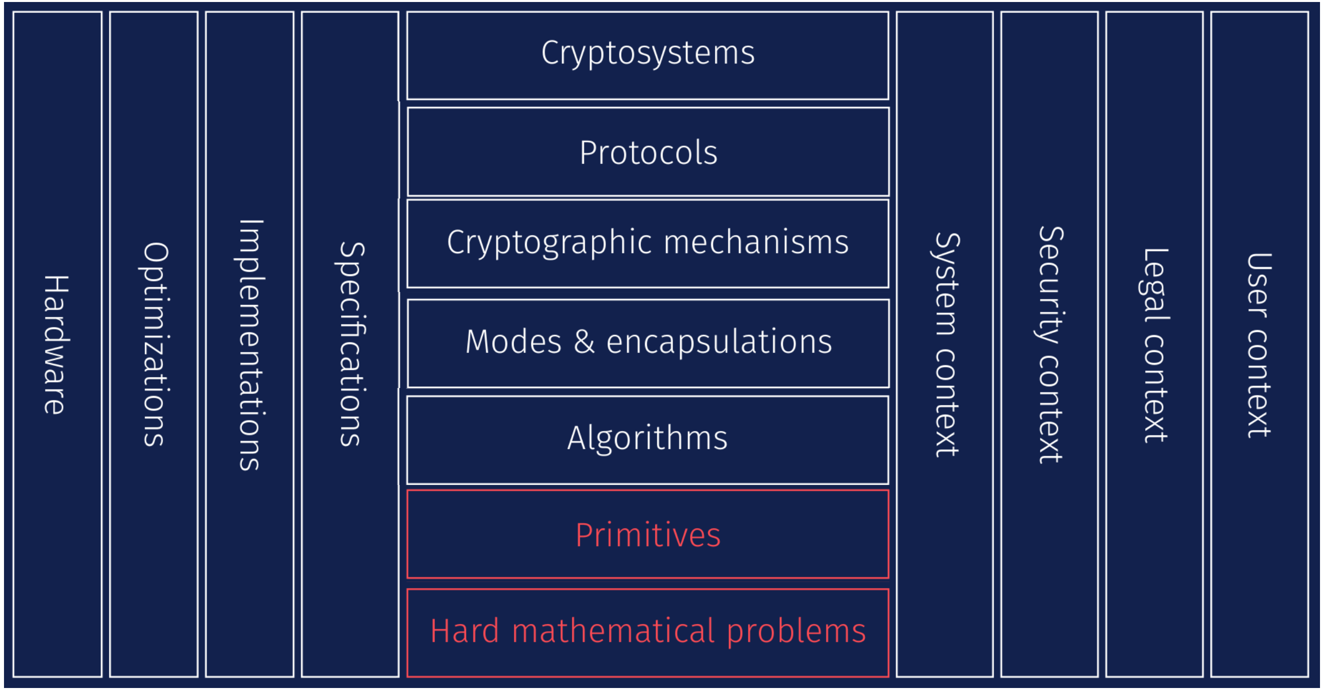
Take Away:

- Parts of the system that seem totally disconnected **can fuck each other up**
- There's **so much things to consider beyond the primitives**
  - Primitives are not a good abstraction of security guarantees



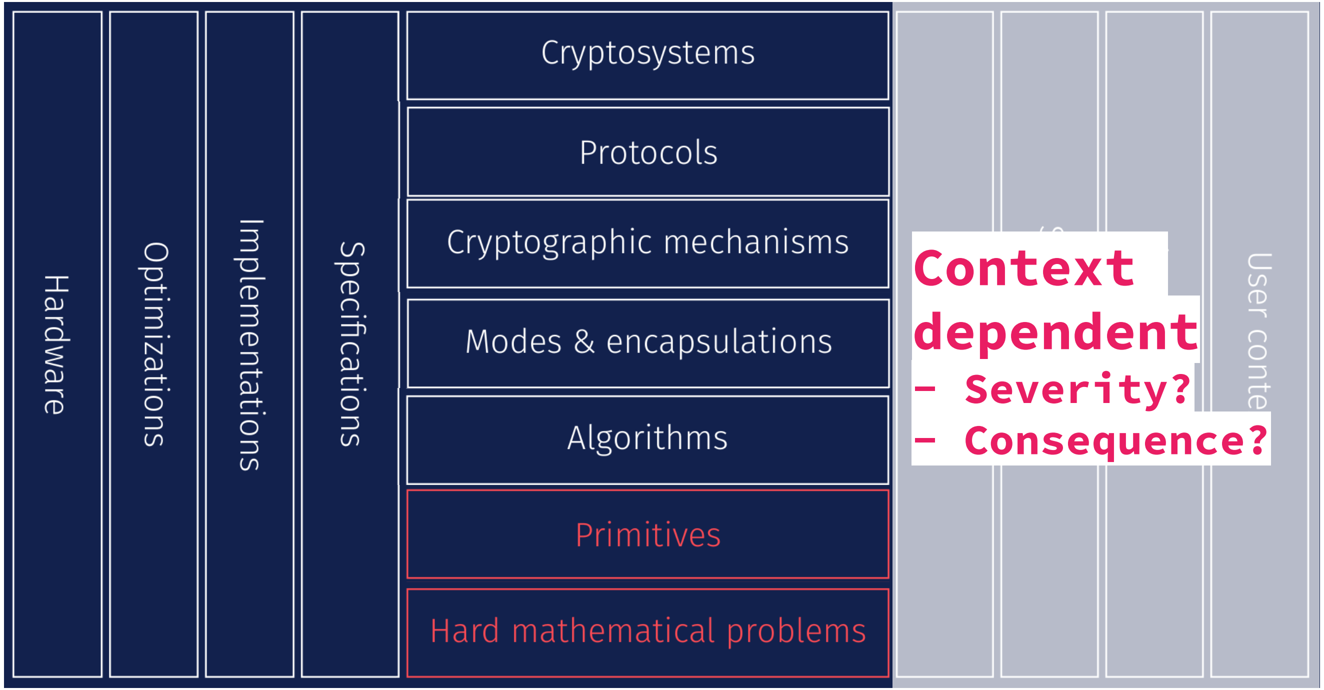
Contents

The Stack



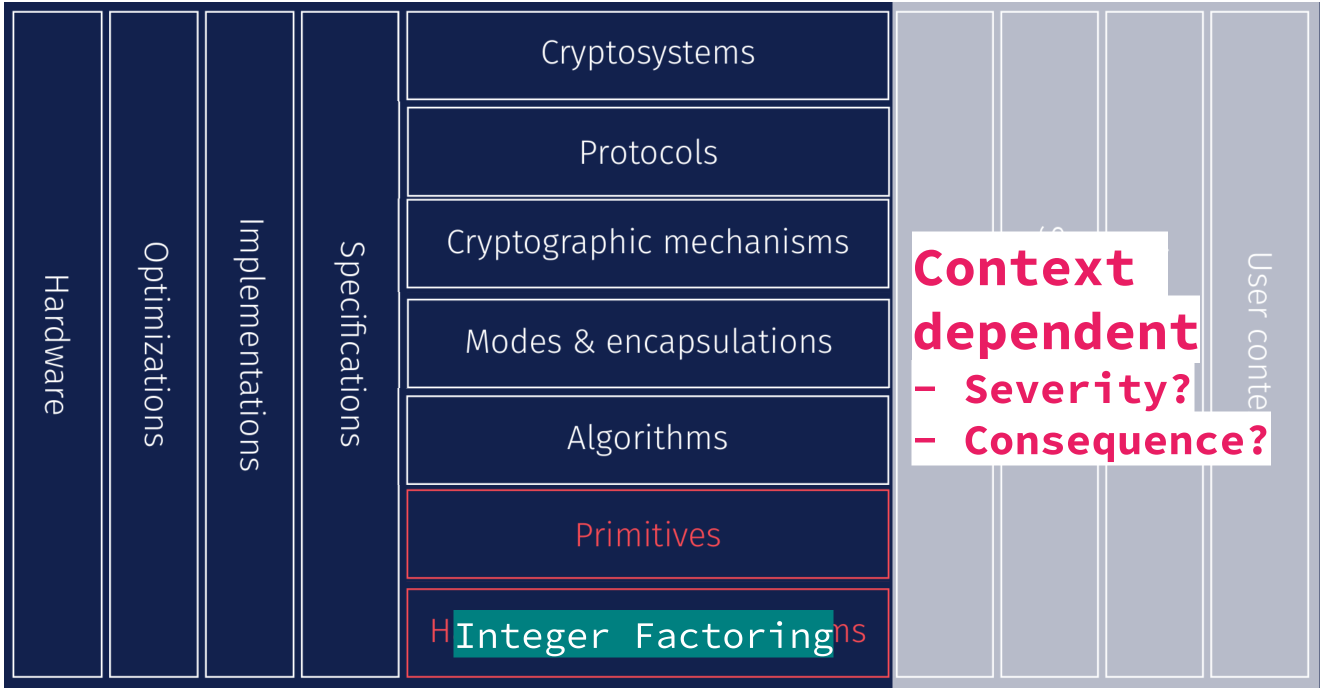
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



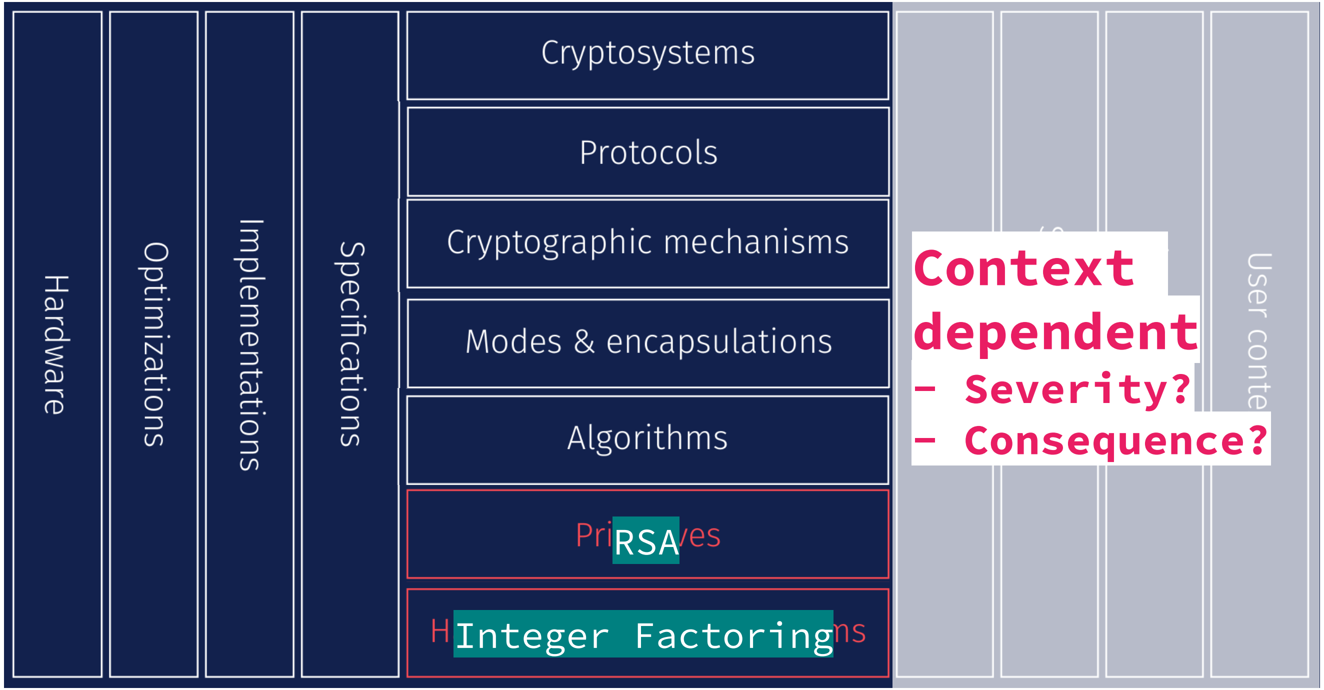
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



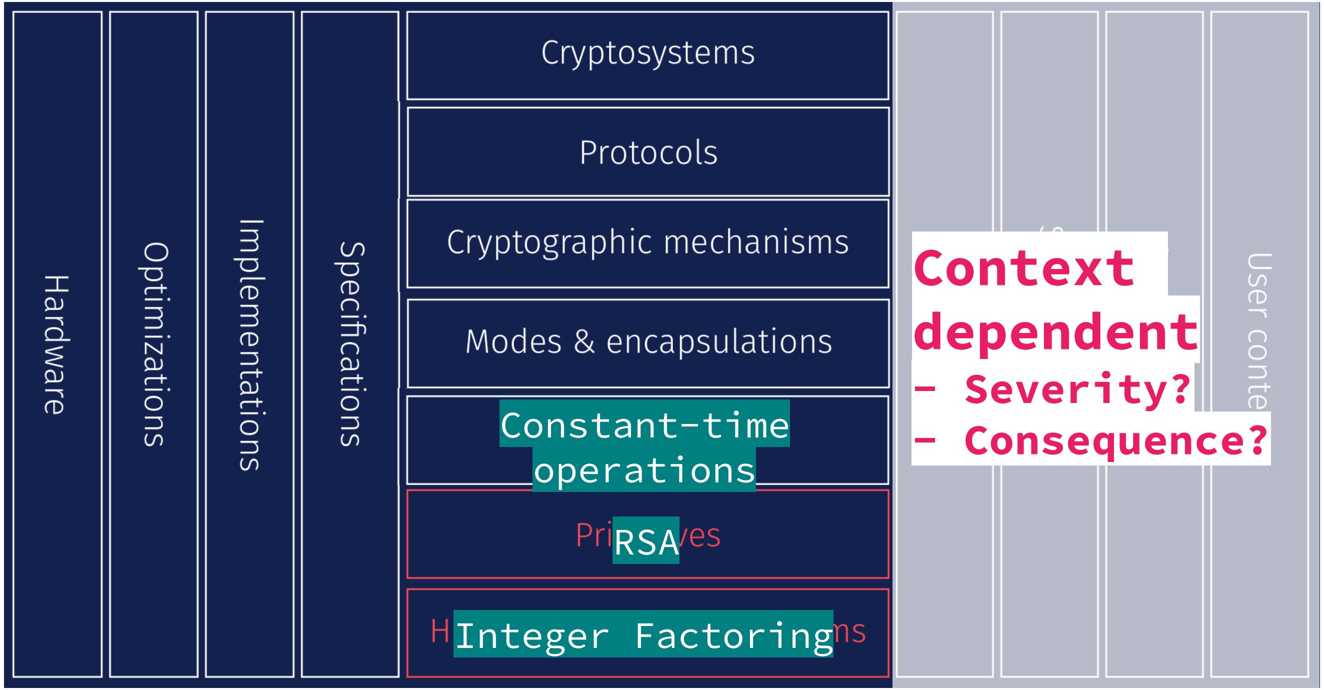
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



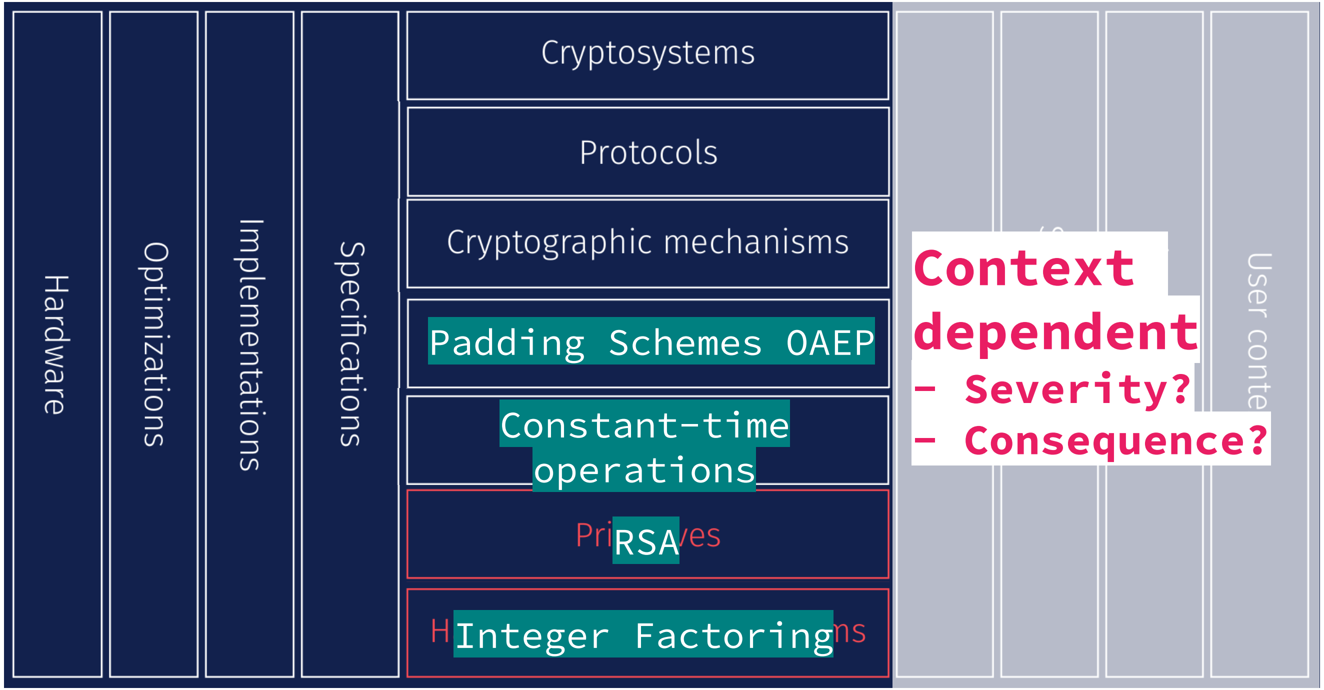
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



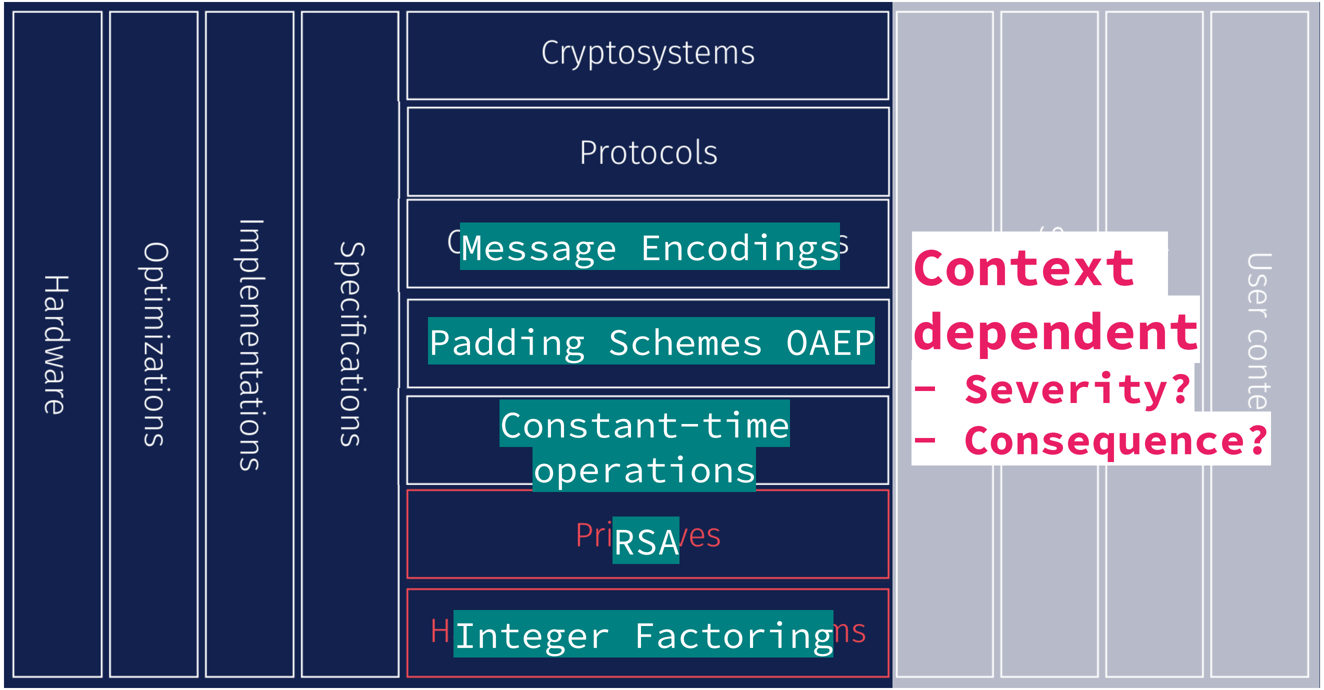
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



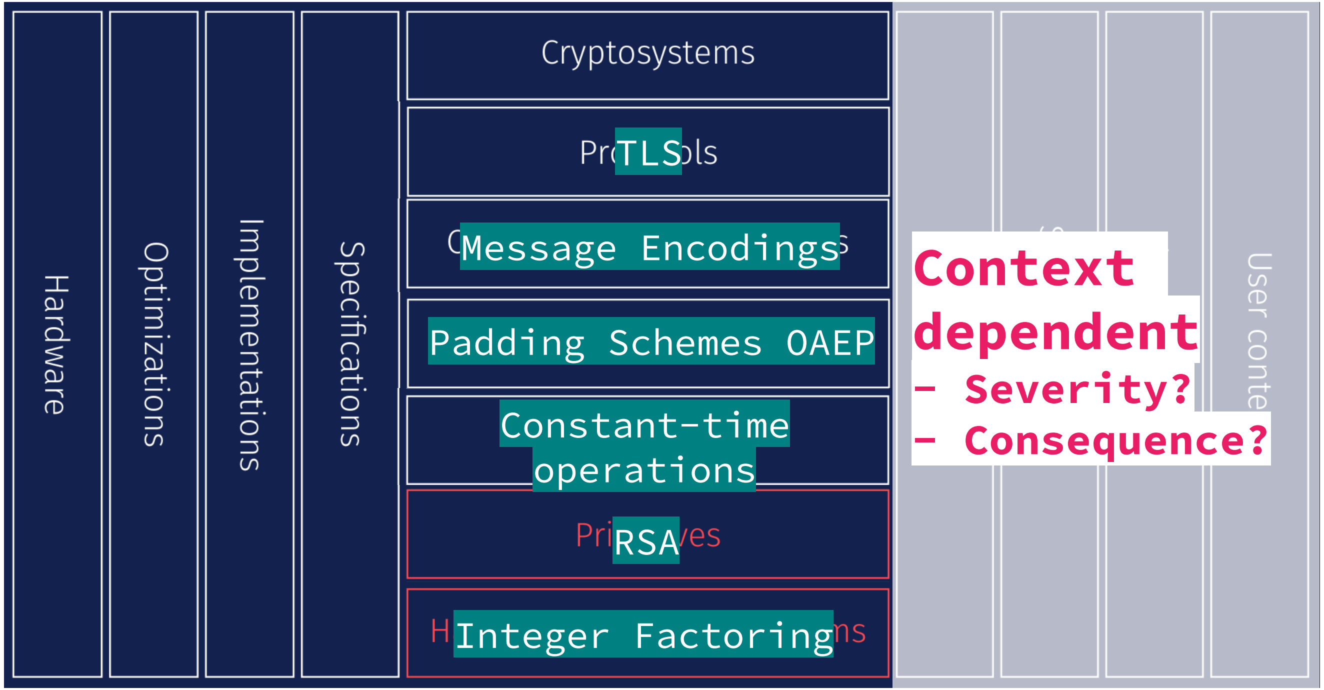
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



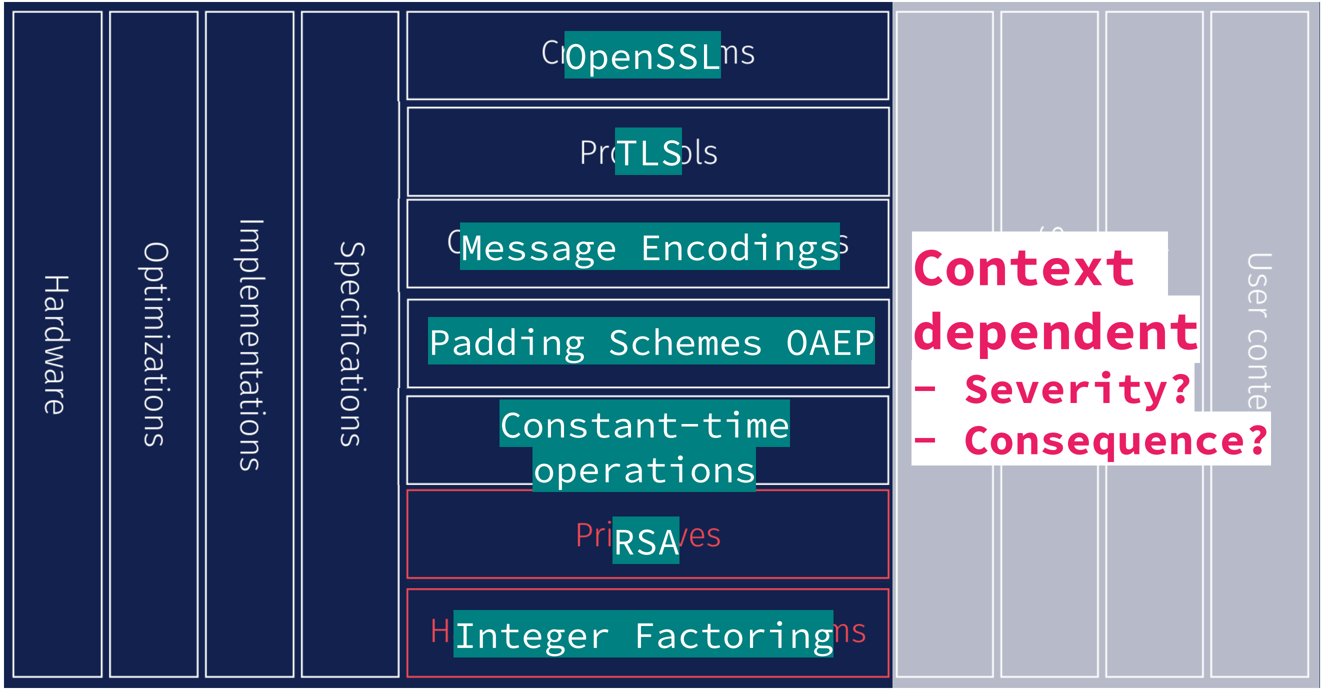
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



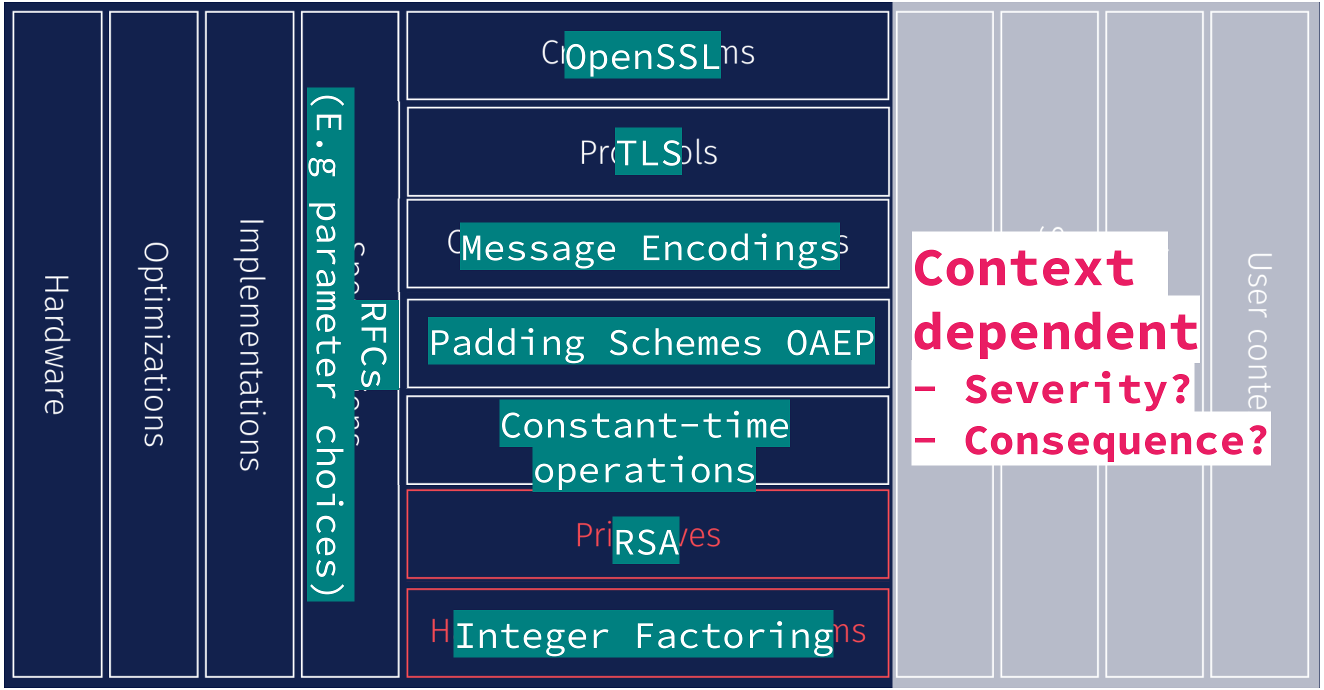
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



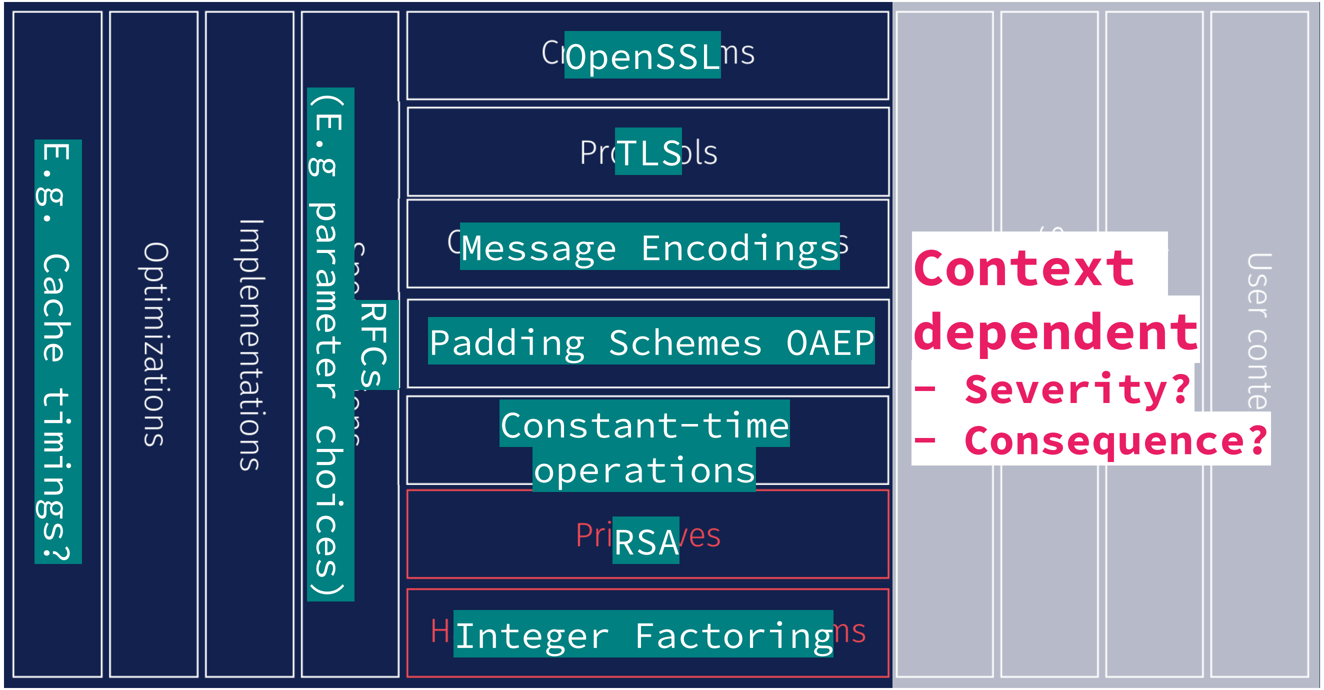
Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack



Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

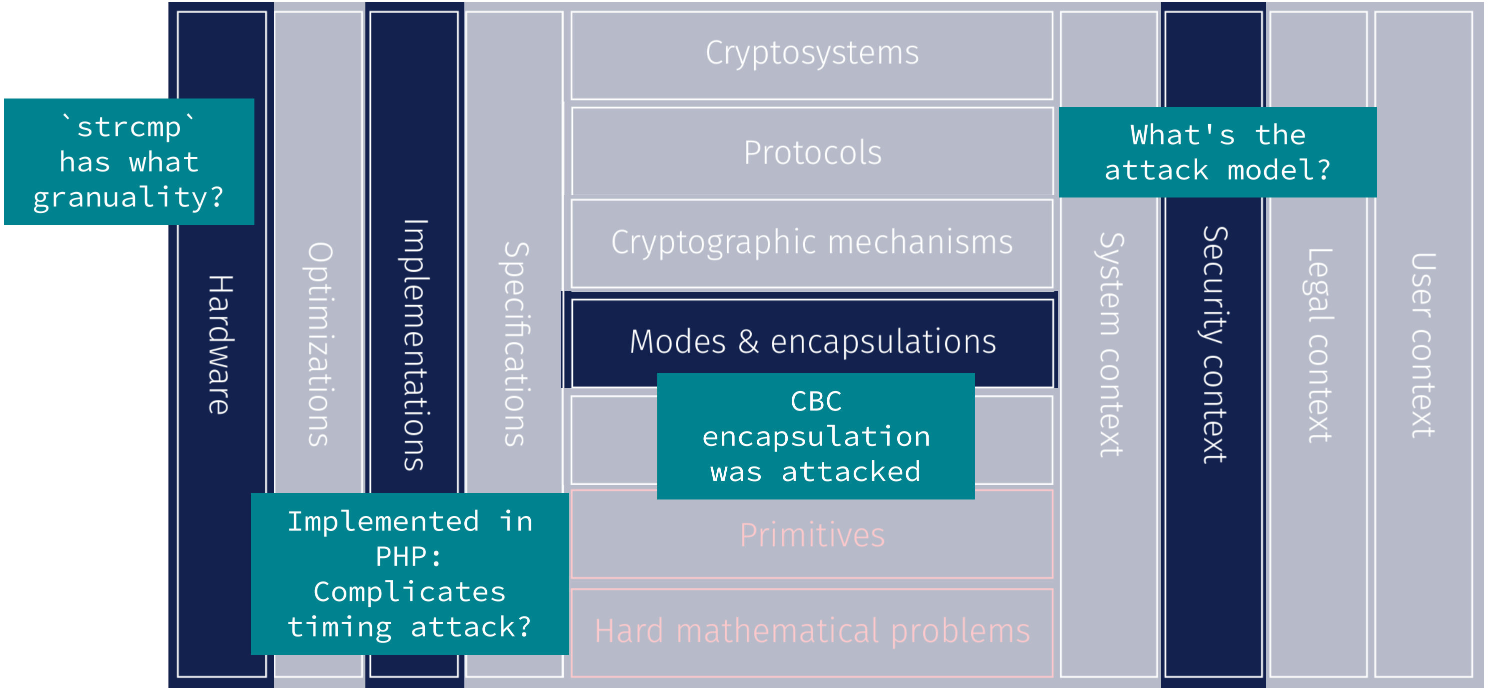
The Stack



Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack

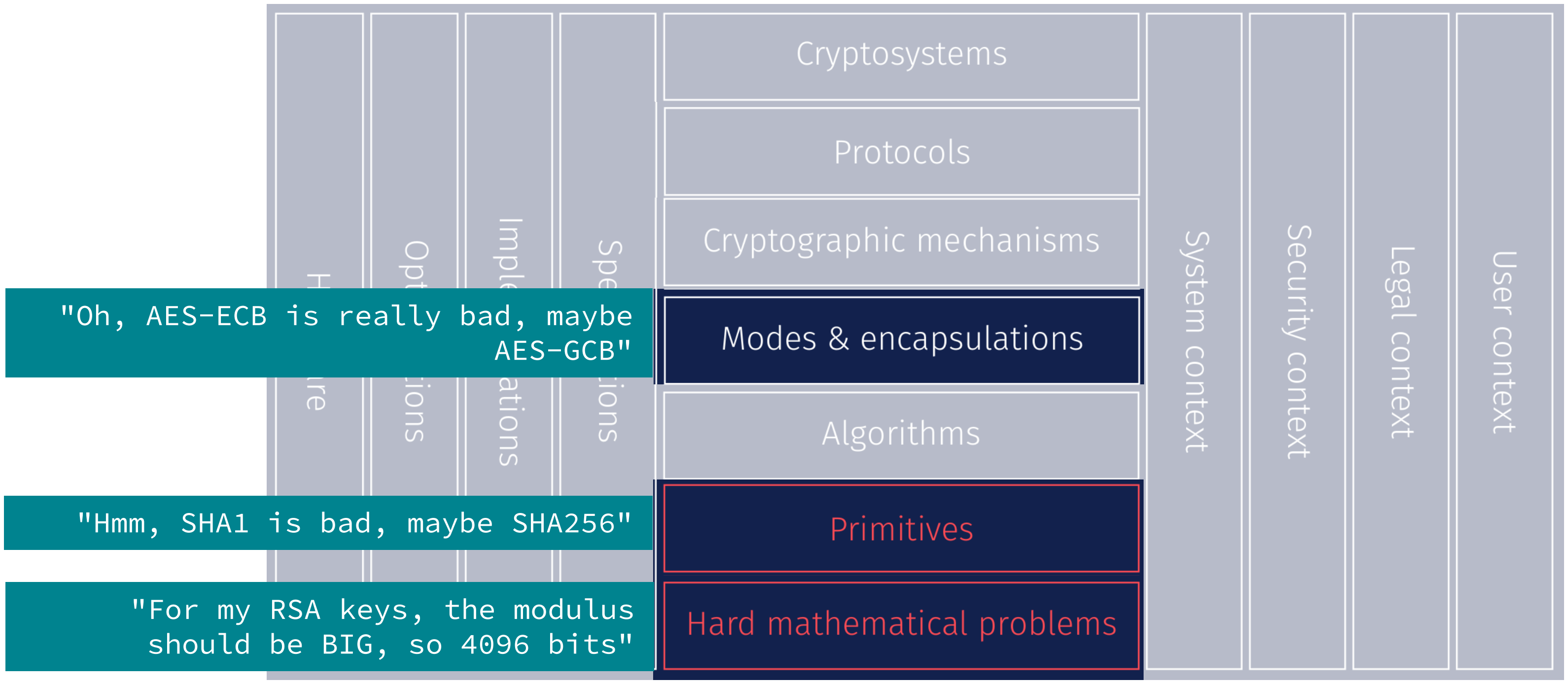
What was considered in the attack covered



Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

The Stack

Where most devs stop at



Stolen from: https://blog.quarkslab.com/status-of-post-quantum-cryptography-implementation.html

Unknown Unknowns

---

*"There are things that **we don’t know we don’t know**."*

~ Donald Rumsfeld

- Cryptography is all about **unknown unknowns**.
  - MACs have to be compared constant time?
  - Nonce shouldn't be used more than once?
  - You need to properly pad RSA messages?

Contents

Hyundai's D-Audio2V Firmware Update

- Firmware updates are often **signed**
  - Ensure updates are from a trusted source (not an attacker)

- Firmware is also often **encrypted**
  - Prevents reverse engineering

Hyundai's D-Audio2V Firmware Update

D-Audio2V Firmware Update: A Disaster
Research done by programmingwithstyle.com/posts/howihackedmycar

- Firmware signed with RSA
    - **Public Key is an example key** -> Private key is online

Hyundai's D-Audio2V Firmware Update

D-Audio2V Firmware Update: A Disaster
Research done by programmingwithstyle.com/posts/howihackedmycar

- Firmware signed with RSA
  - **Public Key is an example key** -> Private key is online
- Firmware update comes in encrypted ZIP
  - PKZIP vulnerable to a **Known Plaintext Attack**

Hyundai's D-Audio2V Firmware Update

D-Audio2V Firmware Update: A Disaster
Research done by programmingwithstyle.com/posts/howihackedmycar

- Firmware signed with RSA
    - **Public Key is an example key** -> Private key is online
- Firmware update comes in encrypted ZIP
    - PKZIP vulnerable to a **Known Plaintext Attack**
- System image in update encrypted with AES-CBC
    - **Key (same for enc and dec) publicly available** in open-sourced code

Contents

The Bad API Problem: Current State

- Developers not aware of what cryptographic operations to do
  - Sign? Encrypt??
- Developers no idea *how* to implement operations

The Bad API Problem: Current State

Javascript Object Signing and Encryption (JOSE)

A set of standards to *sign* and *encrypt* json (essentially)

JWS: Anybody can see the data, and there's a signature to verify the legitimacy of data

JWE: Data is encrypted

The Bad API Problem: Current State

JWT Web Tokens:

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Alg used
(not encrypted)

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

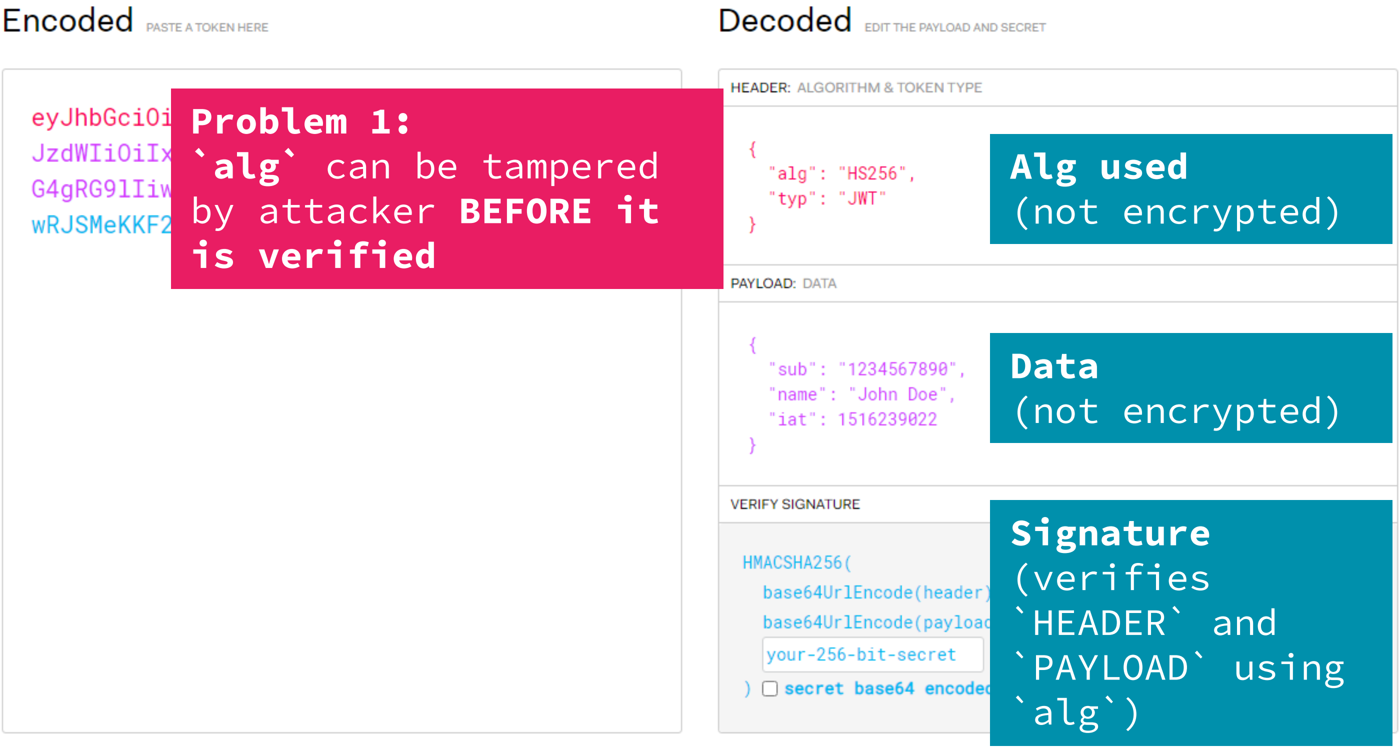
Data
(not encrypted)

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header)
  base64UrlEncode(payloac
  your-256-bit-secret
) ☐ secret base64 encodec
```

Signature
(verifies `HEADER` and `PAYLOAD` using `alg`)

The Bad API Problem: Current State



The Bad API Problem: Current State

Possible `**alg**` values:

- **HS256 <-- Symmetric signing**
- **RS256 <-- Asymmetric signing**
- **None** <-- **Lmao**

Trivial Attack 1:
Attacker specifies `**alg: None**`. Server doesn't verify signature

Trivial Attack 2:
Attacker changes `**alg: RS256**` to `**alg: HS256**` and signs token with **PUBLIC key**.

The Bad API Problem: Current State

Possible `alg` values:

- **HS256 <-- Symmetric signing**
- **RS256 <-- Asymmetric signing**
- **None** <-- **Lmao**

*The "alg" value [...] **MUST be present and MUST be understood and processed by implementations.***

~ RFC7515 Section 4.1.1

Server **HAVE** to use the **`alg`** given in JWT to be RFC7515 compliant

The Bad API Problem: Current State

Signing: JWS

```
import jose

claims = {
    'iss': 'http://www.example.com',
    'exp': int(time()) + 3600,
    'sub': 42,
}

jwk = {'k': 'password'}

jws = jose.sign(claims, jwk, alg='HS256')
```

```
jwt = jose.serialize_compact(jws)
'eyJhbGciOiAiSFMyNTYifQ.eyJpc3MiOiAiaHR0cDov
L3d3dy5leGFtcGxlLmNvbSIsICJzdWIiOiA0MiwgImV4
cCI6IDEzOTU2NzQ0Mjd9.WYApAiwiKd-
eDClA1fg7XFrnfHzUTgrmdRQY4M19Vr8'

jws = jose.deserialize_compact(jwt)
jose.verify(jws, jwk, jsw.alg)
JWT(header={u'alg': u'HS256'},
claims={u'iss': u'http://www.example.com',
u'sub': 42, u'exp': 1395674427})
```

The Bad API Problem: Current State

Signing: JWS

```
import jose

claims = {
    'iss': 'http://www.example.com',
    'exp': int(time()) + 3600,
    'sub': 42,
}

jwk = {'k': 'password'}

jws = jose.sign(claims, jwk, alg='HS256')
```

```
jwt = jose.serialize_compact(jws)
'eyJhbGciOiAiSFMyNTYifQ.eyJpc3MiOiAiaHR0cDov
L3d3dy5leGFtcGxlLmNvbSIsICJzdWIiOiA0MiwgImV4
cCI6IDEzOTU2NzQ0Mjd9.WYApAiwiKd-
eDClA1fg7XFrnfHzUTgrmdRQY4M19Vr8'

jws = jose.deserialize_compact(jwt)
y(jws, jwk, jsw.alg)
={u'alg': u'HS256'},
iss': u'http://www.example.com',
u'sub': 42, u'exp': 1395674427})
```

Problem 2:
Potential for **key reuse**

The Bad API Problem: Current State

*A JSON Web Key (JWK) [...] represents a cryptographic key. [...] **allows for a generalized key as input that can be applied to a number of different algorithms that may expect a different number of inputs.***

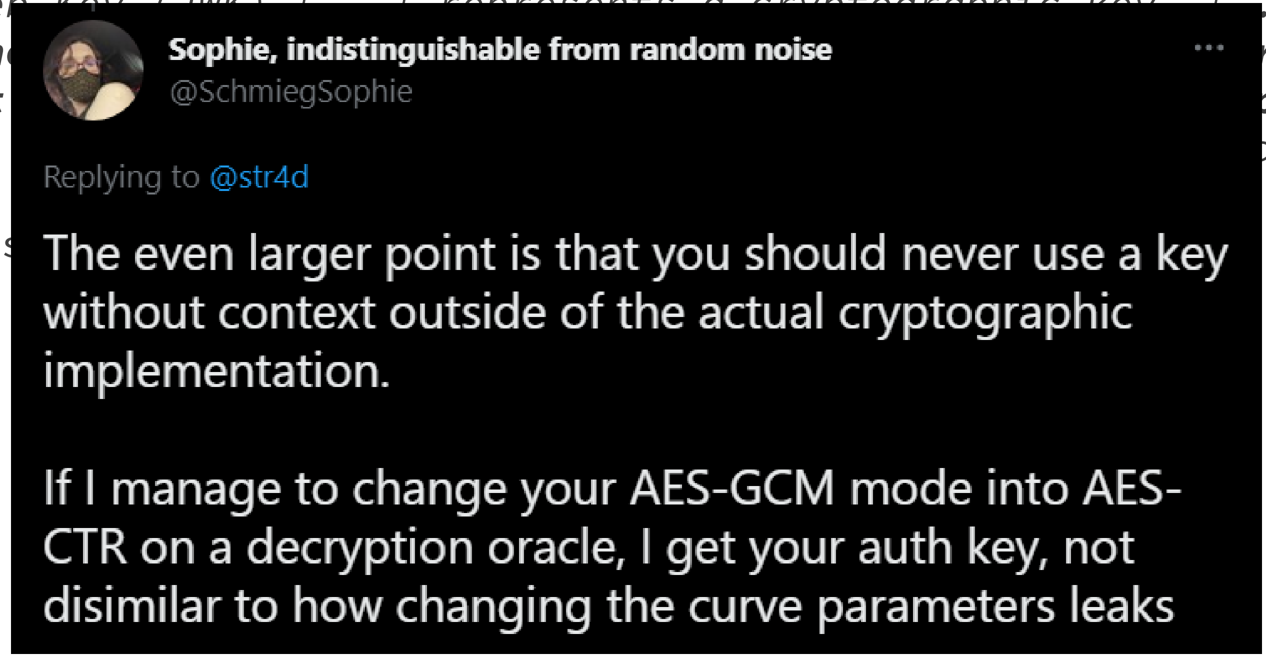
~ JOSE Official Documentation

- **JWK** designed with reuse in different algos in mind

The Bad API Problem: Current State

A JSON We ... *..] **allows***
for a gen ... ***r of***
different ... ***puts.***
cumentation

- **JWK** des



The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, `**alg**` param)
    1. **RSA with PKCS #1v1.5 padding**
    2. **RSA with OAEP padding**
    3. **ECDH**
    4. **AES-GCM**

2. Algo to encrypt the data (Claims Encryption, `**enc**` param)
    1. **AES-CBC + HMAC**
    2. **AES-GCM**

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, **`alg`** param)
    1. **RSA with PKCS #1v1.5 padding**
    2. **RSA with OAEP padding**
    3. **ECDH**
    4. **AES-GCM**

2. Algo to encrypt the data (Claims Encryption, **`enc`** param)
    1. **AES-CBC + HMAC**
    2. **AES-GCM**

Problem 3:
INSECURE CONFIGURATIONS

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, **`alg`** param)
    1. **RSA with PKCS #1v1.5 padding**
    2. **RSA with OAEP padding**
    3. **ECDH**
    4. **AES-GCM**
2. Algo to encrypt the data (Claims Encryption, **`enc`** param)
    1. **AES-CBC + HMAC**
    2. **AES-GCM**

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, `alg` param)
    1. **RSA with PKCS #1v1.5 padding** **Very famous padding oracle attack**
    2. **RSA with OAEP padding**
    3. **ECDH**
    4. **AES-GCM**
2. Algo to encrypt the data (Claims Encryption, `enc` param)
    1. **AES-CBC + HMAC**
    2. **AES-GCM**

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, `alg` param)
    1. **RSA with PKCS #1v1.5 padding** **Very famous padding oracle attack**
    2. **RSA with OAEP padding**
    3. **ECDH** **Weierstrass form: Prone to invalid curve attacks**
    4. **AES-GCM**
2. Algo to encrypt the data (Claims Encryption, `enc` param)
    1. **AES-CBC + HMAC**
    2. **AES-GCM**

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, `alg` param)
   1. **RSA with PKCS #1v1.5 padding** **Very famous padding oracle attack**
   2. **RSA with OAEP padding**
   3. **ECDH** **Weierstrass form: Prone to invalid curve attacks**
   4. **AES-GCM** **Shared key?????**
2. Algo to encrypt the data (Claims Encryption, `enc` param)
   1. **AES-CBC + HMAC**
   2. **AES-GCM**

The Bad API Problem: Current State

JWE: Developers specify two algos

1. Algo to encrypt the key (CEK Encryption, `alg` param)
    1. **RSA with PKCS #1v1.5 padding** **Very famous padding oracle attack**
    2. **RSA with OAEP padding**
    3. **ECDH** **Weierstrass form: Prone to invalid curve attacks**
    4. **AES-GCM** **Shared key?????**
2. Algo to encrypt the data (Claims Encryption, `enc` param)
    1. **AES-CBC + HMAC** **CBC: Malleable (as we have seen)**
    2. **AES-GCM**

The Bad API Problem: Current State

JOSE: Takeaway

1. Insecure standard
    - Secure usage breaks standard
2. "Sign or encrypt? Or both?? Wtf is **none**??"
3. "What **alg** for key/msg enc?"
    - More insecure configs than secure configs
4. API leaves too much room for error (e.g. usage of key)

The Bad API Problem

Devs trying to implement crypto properly **go down deep rabbitholes**

- *"developers lacked the confidence to choose the best algorithm or parameter"* (Hazhirpasand et al, 2021)

Solution?

- ~~Developers should git gud~~

- Crypto libraries should be proper abstractions
    - **easy to use**, **hard to misuse**, and **secure by default**

The Bad API Problem: Solutions

Libsodium: Password hashing

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE, crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

The Bad API Problem: Solutions

Libsodium: Password hashing

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE, crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

Generates salt for you

The Bad API Problem: Solutions

Libsodium: Password hashing

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE, crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

Generates salt for you

Dev-relevant parameters

The Bad API Problem: Solutions

Libsodium: Password hashing

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE, crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

Generates salt for you

Dev-relevant parameters

Never touch the primitives

The Bad API Problem: Solutions

Libsodium: Authenticated Encryption Auxillary Data (AEAD)

AEAD constructions

Documentation tells you exactly what AEAD is

This operation:

- Encrypts a message with a key and a nonce to keep it confidential
- Computes an authentication tag. This tag is used to make sure that the message, as well as optional, non-confidential (non-encrypted) data, haven't been tampered with.

A typical use case for additional data is to authenticate protocol-specific metadata about the message, such as its length and encoding.

The Bad API Problem: Solutions

Libsodium: Authenticated Encryption Auxillary Data (AEAD)

Availability and interoperability

Construction	Key size	Nonce size	Block size
AES256-GCM	256 bits	96 bits	128 bits
ChaCha20-Poly1305	256 bits	64 bits	512 bits
ChaCha20-Poly1305-IETF	256 bits	96 bits	512 bits
XChaCha20-Poly1305-IETF	256 bits	192 bits	512 bits

**Multiple choices?
All of them are secure**

The Bad API Problem: Solutions

Libsodium: Authenticated Encryption Auxillary Data (AEAD)

Limitations

Construction	Max bytes for a single (key,nonce)	Max bytes for a single key
AES256-GCM	~ 64 GB	~ 350 GB (for ~16 KB long messages)
ChaCha20-Poly1305	No practical limits (~ 2^64 bytes)	Up to 2^64* messages, no practical total size limits
ChaCha20-Poly1305-IETF	256 GB	Up to 2^64* messages, no practical total size limits
XChaCha20-Poly1305-IETF	No practical limits (~ 2^64 bytes)	Up to 2^64* messages, no practical total size limits

Developers are given dev-relevant info about each choice

The Bad API Problem: Solutions

Libsodium: Authenticated Encryption Auxillary Data (AEAD)

Still unsure?
There's recommendations if you don't have any special considerations

TL;DR: which one should I use?

`XChaCha20-Poly1305-IETF` is the safest choice.

Other choices are only present for interoperability with other libraries that don't implement `XChaCha20-Poly1305-IETF` yet.

QnA