Selected Cryptography Vulnerabilities of IoT Implementations

ICMC 2022

Paul Bottinelli [paul.bottinelli@nccgroup.com] - Cryptography Services

Internet of Things (IoT) – *from buzzword to ubiquity*

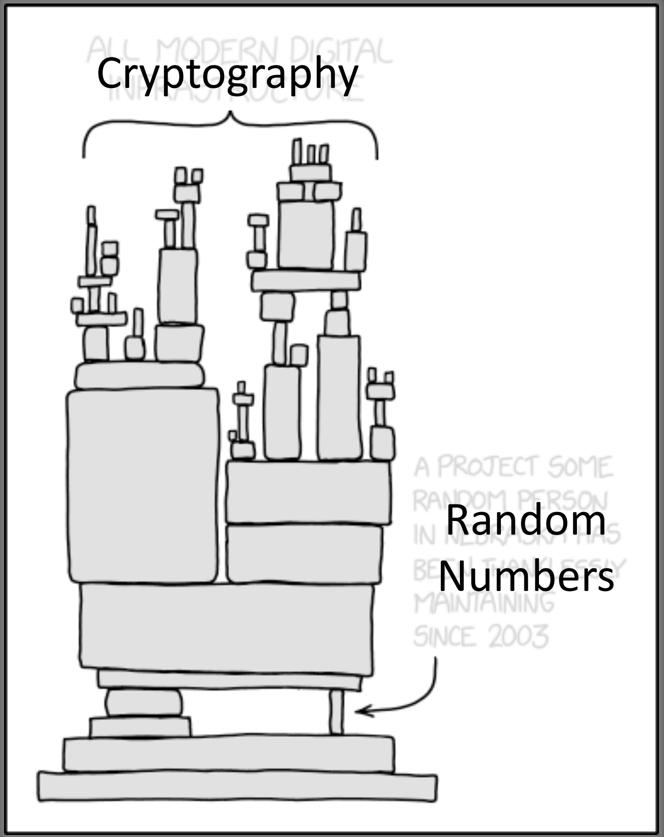
- Large surface, many protocols
- IoT encompasses the wide variety of security reviews we perform
- This presentation discusses commonly encountered issues and dives into a few selected ones
- Examples come either from NCC public reports, CVEs or vulnerabilities seen repeatedly
- https://research.nccgroup.com/category/public-report/

Where do we find crypto vulnerabilities?

- Randomness
- Algorithm Confusion
- Key Management
- Cryptographic Primitives
- Protocols
- Side-channel Attacks

Randomness

- Usage and correct seeding of Cryptographically Secure PRNG
- Unbiased, uniformly random values in correct range
- Consider range endpoints
- Usage of random numbers for other purposes than private keys



Adapted from https://xkcd.com/2347/

Zcash NU5
Random EC point generation does not result in point on given curve.

View Report

Key Management

- Hardcoded and default keys (e.g. finding private firmware signing key in the device firmware)
- Key reuse (e.g. Same key for entire device fleet)
- Inappropriate key sizes
- Poor updating / rolling / revocation mechanisms

WhatsApp E2E Encrypted Backups
Weak 512-bit RSA signing key can be factored, used to forge signatures.

View Report

Bluetooth Low Energy Advisory in FIDO Security Keys

- Bluetooth example pairing keys are published in the spec
- Found hardcoded into firmware and used in production deployments
- Allows attacker to impersonate device

CVE-2019-2102
Bluetooth Low Energy Advisory

View Advisory

When in Secure Simple Pairing debug mode, the Link Manager shall use the following Diffie Hellman private / public key pairs:

For P-192:

Private key:	07915f86918ddc27005df1d6cf0c142b625ed2eff4a518ff
Public key (X):	15207009984421a6586f9fc3fe7e4329d2809ea51125f8ed
Public key (Y):	b09d42b81bc5bd009f79e4b59dbbaa857fca856fb9f7ea25

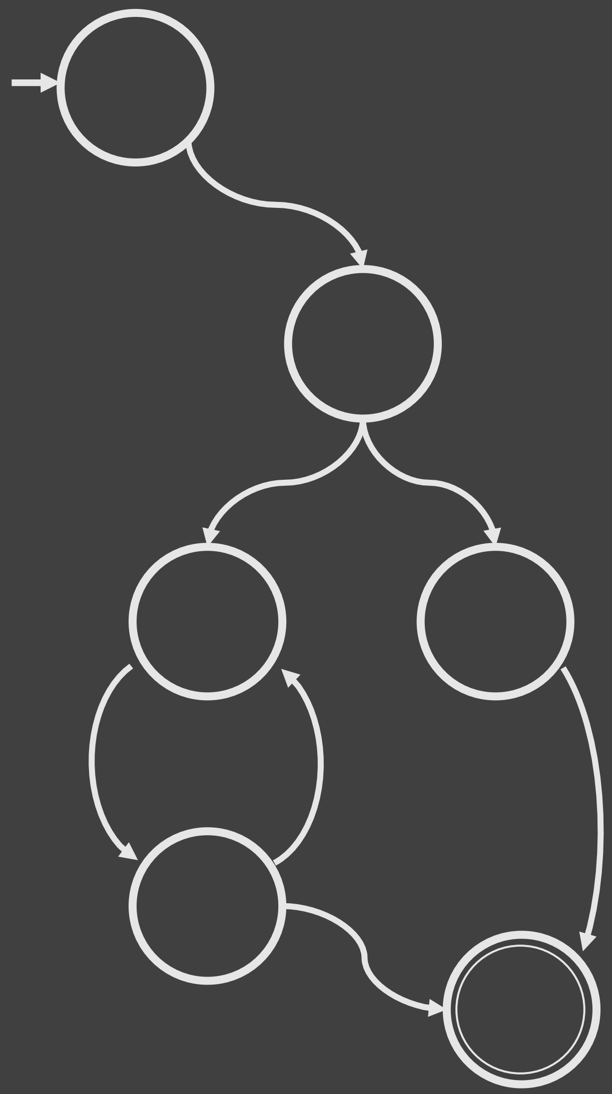
For P-256:

Private key:	3f49f6d4 a3c55f38 74c9b3e3 d2103f50 4aff607b eb40b799 5899b8a6 cd3c1abd
Public key (X):	20b003d2 f297be2c 5e2c83a7 e9f9a5b9 eff49111 acf4fddb cc030148 0e359de6
Public key (Y):	dc809c49 652aeb6d 63329abf 5a52155c 766345c2 8fed3024 741c8ed0 1589d28b

Sec. 7.6.4 - Bluetooth Core Specification v5.3:
https://www.bluetooth.com/specifications/specs/core-specification-5-3/

Protocols

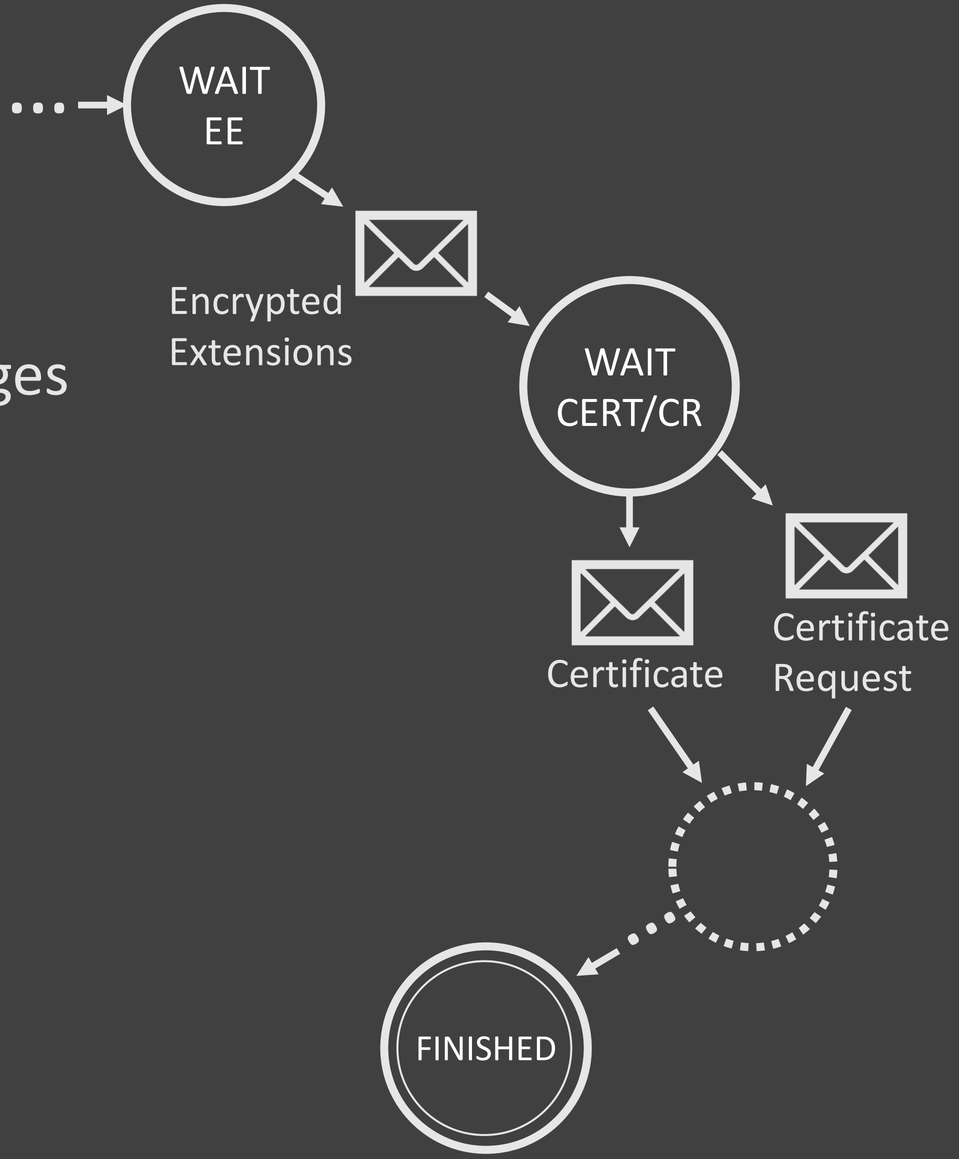
- Protocols are complex state machines
- Message flow tampering (dropped, injected, re-ordered, repeated, or modified messages)
- Backwards compatibility and downgrade attacks
- Interoperability with other implementations
- Missing input validation!



wolfSSL TLS 1.3 Client MitM

- RFC 8446 defines following handshake messages

```
select (Handshake.msg_type) {
    case client_hello:          ClientHello;
    case server_hello:          ServerHello;
    case end_of_early_data:     EndOfEarlyData;
    case encrypted_extensions:  EncryptedExtensions;
    case certificate_request:   CertificateRequest;
    case certificate:           Certificate;
    case certificate_verify:    CertificateVerify;
    case finished:              Finished;
    case new_session_ticket:    NewSessionTicket;
    case key_update:            KeyUpdate;
```



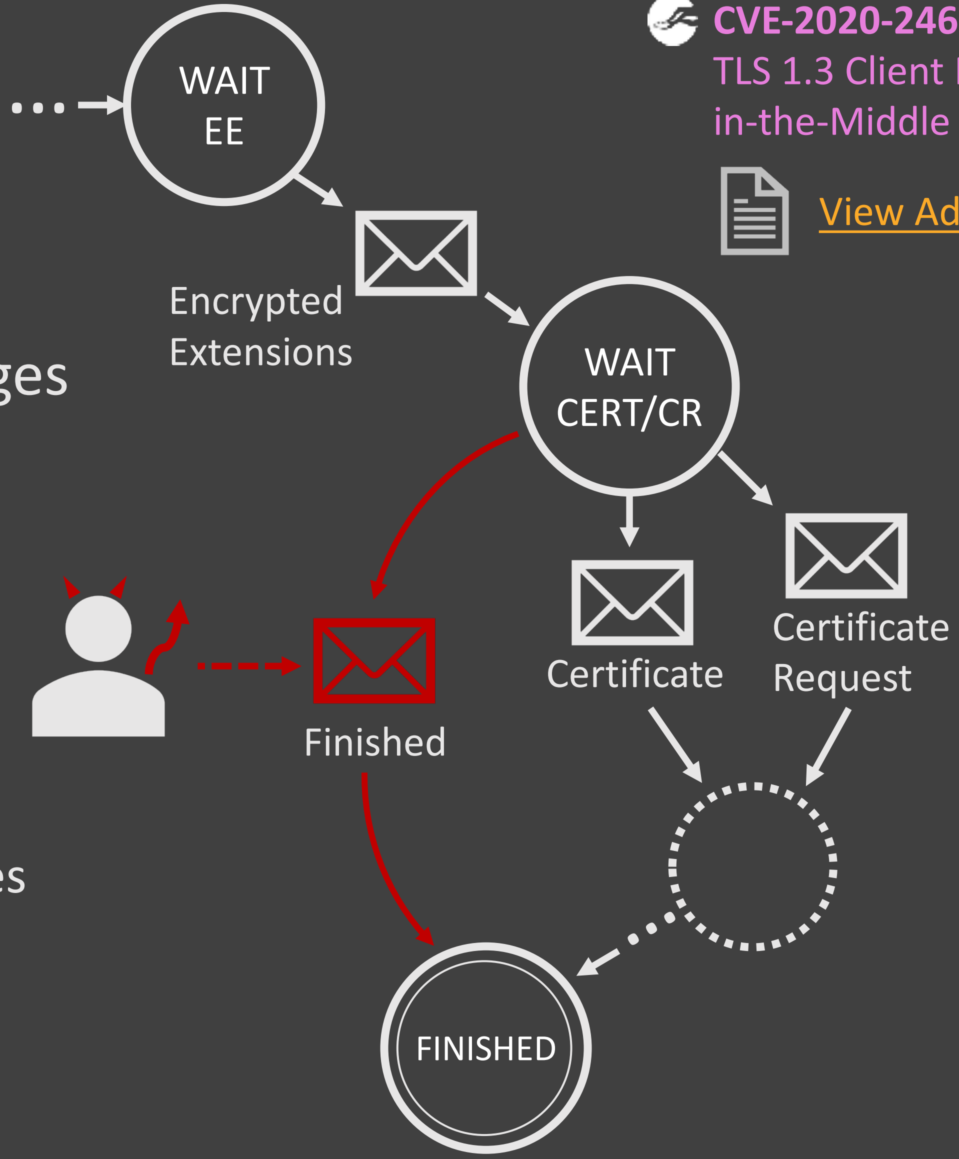
wolfSSL TLS 1.3 Client MitM

CVE-2020-24613
TLS 1.3 Client Man-in-the-Middle Attack
[View Advisory]

- RFC 8446 defines following handshake messages

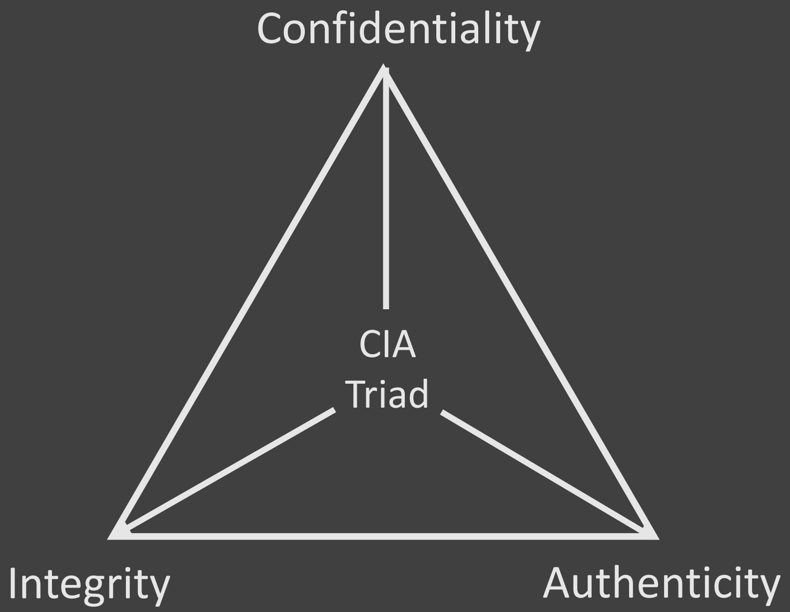
```
select (Handshake.msg_type) {
    case client_hello:          ClientHello;
    case server_hello:          ServerHello;
    case end_of_early_data:     EndOfEarlyData;
    case encrypted_extensions:  EncryptedExtensions;
    case certificate_request:   CertificateRequest;
    case certificate:           Certificate;
    case certificate_verify:    CertificateVerify;
    case finished:              Finished;
```

- Bug in TLS 1.3 client state machine
  - Only “CertificateRequest” or “Certificate” messages are valid from WAIT_CERT_CR state
  - Implementation accepts “Finished” message
- Attacker can impersonate any TLS 1.3 Server



Algorithm Confusion

- Lack of cryptography
- Outdated and obsolete algorithms (DES, RC4)
- Using algorithms for the wrong purposes
  - “encrypting” with a hash function
  - “signing” using AES
  - assuming encrypted data is protected against tampering
- Use of cryptographic hash functions that do not provide collision resistance (MD5, SHA1)



CVE-2022-32509
Lack of Certificate Validation on TLS Communications in Nuki smart locks

View Advisory

Cryptographic Primitives – Symmetric Cryptography

- Use of non-authenticated modes and ciphers
  - Don’t use ECB!
- Use of custom “encrypt + MAC” constructions instead of AEAD modes (e.g., GCM)
- IV/Nonce considerations
  - Should be authenticated
  - Should not be treated as secret
  - Random or unique IV requirements vary by mode
- Key usage limits and IV wrap-around

CTR → Nonce: must be used only once but can be predictable

CBC → IV: must be unpredictable

Repeating CTR Nonces in CCM

```
5.  Nonce Suggestions

   The main requirement is that, within the scope of a single key, the
   nonce values are unique for each message.  A common technique is to
   number messages sequentially, and to use this number as the nonce.
   Sequential message numbers are also used to detect replay attacks and
   to detect message reordering, so in many situations (such as IPsec
   ESP [ESP]) the sequence numbers are already available.
```

Counter with CBC-MAC (CCM): https://www.ietf.org/rfc/rfc3610.txt

How to prevent nonce reuse with two parties?

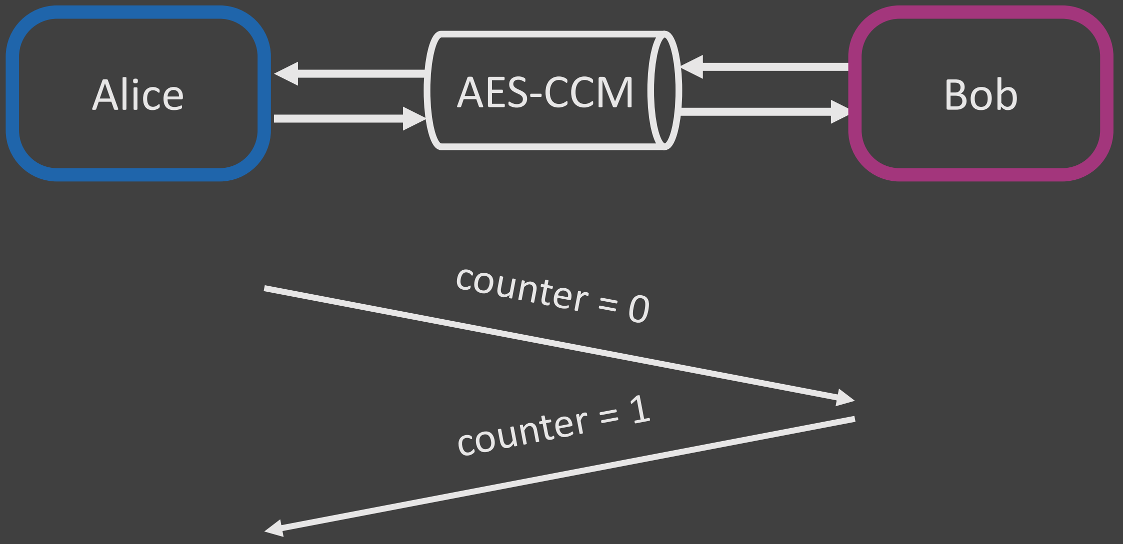
0, 2, 4, 6, …

1, 3, 5, 7, …

Repeating CTR Nonces

0, 2, 4, 6, …

1, 3, 5, 7, …



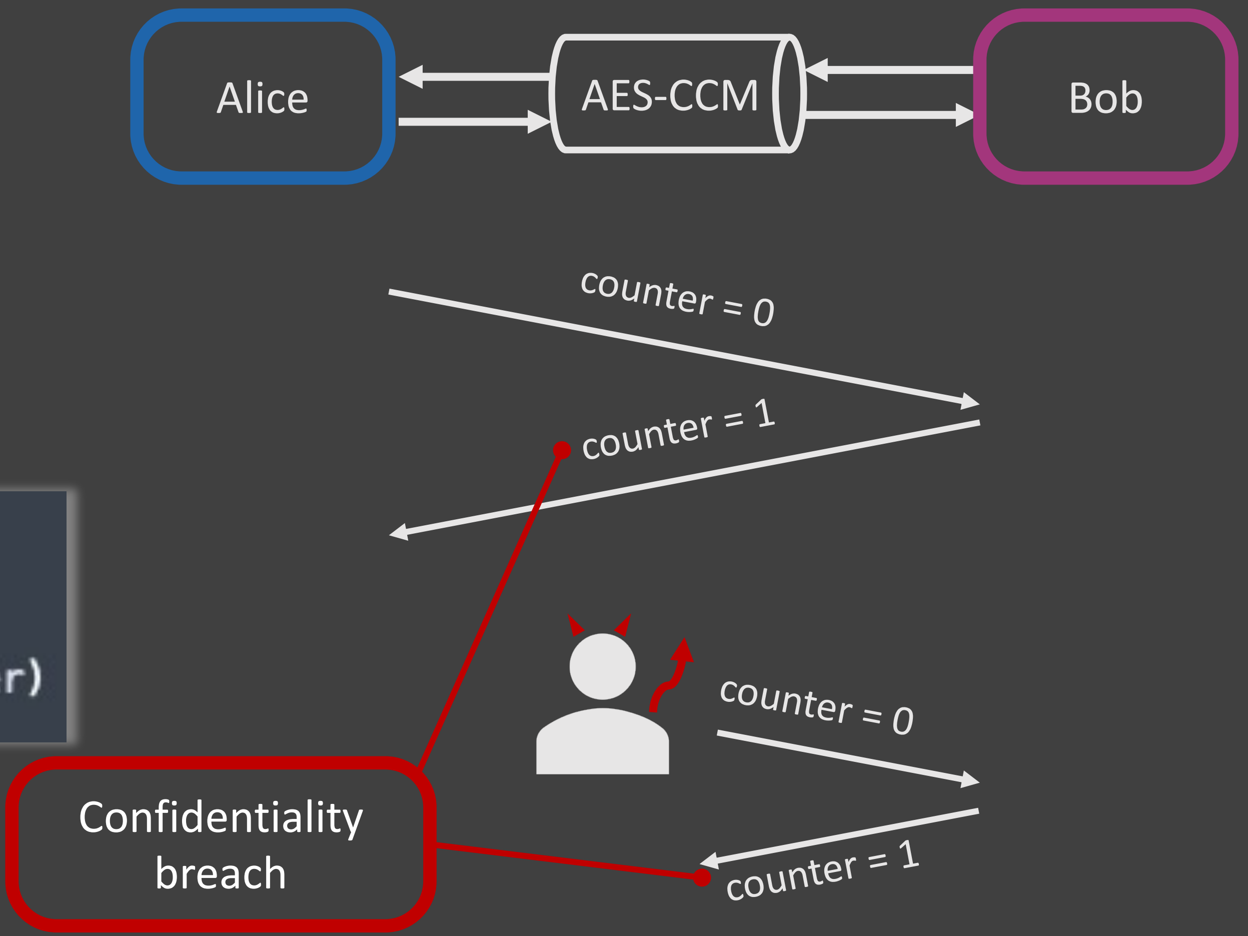

```python
def send_message(received_msg, ptxt):
    counter = received_msg.counter
    counter += 1
    ctxt = AES_CCM(self.key, ptxt, counter)
```

Repeating CTR Nonces

0, 2, 4, 6, …

1, 3, 5, 7, …

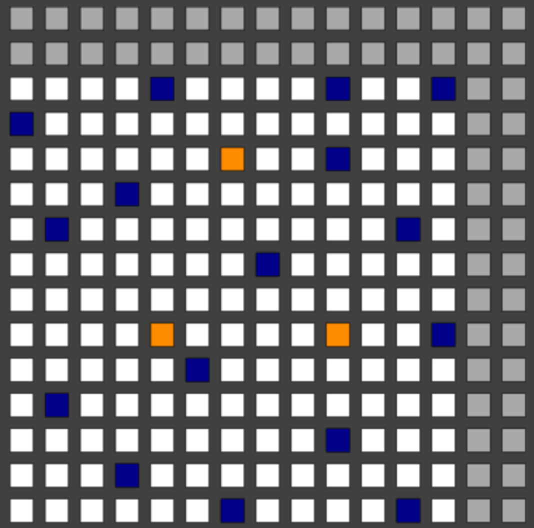
```python
def send_message(received_msg, ptxt):
    counter = received_msg.counter
    counter += 1
    ctxt = AES_CCM(self.key, ptxt, counter)
```



Cryptographic Primitives – Asymmetric Cryptography

- Vulnerable use of RSA
  - Should use RSA-PSS for signatures, RSA-OAEP for encryption
- Elliptic Curve Public Key Validation
  - Point coordinates lower than field modulus
  - The coordinates correspond to a valid curve point.
  - The point is not the point at infinity.
  - Point in expected subgroup
- Elliptic curve point corner cases
  - Handling point at infinity and its representation
  - Canonical representations of points
- ECDSA-specific issues
  - Fragile wrt nonce biases/reuse, malleability of signatures

$y^2 = x^3 + ax + b$



https://research.nccgroup.com/2021/11/18/an-illustrated-guide-to-elliptic-curve-cryptography-validation/

ECDSA Signature Verification

- Embedded systems may have cryptographic coprocessor
  - Sometimes doesn't support ECC primitives
  - Custom implementation in firmware
- May be vulnerable to basic ECDSA signature verification bypass if standard not implemented properly

ECDSA SIGNATURE VERIFICATION. To verify A's signature (r, s) on m, B obtains an authentic copy of A's domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated public key Q. It is recommended that B also validates D and Q (see §5.4 and §6.2). B then does the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute SHA-1(m) and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1 G + u_2 Q$.
6. If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x-coordinate x_1 of X to an integer $\overline{x}_1$, and compute $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

ANSI X9.62 ECDSA Signature Verification

ECDSA Signature Verification

```python
def ecdsa_verify(self, hash, signature):
    G = self.generator
    n = G.order()
    r = signature.r
    s = signature.s
    c = inverse_mod(s, n)
    u1 = (hash * c) % n
    u2 = (r * c) % n
    xy = u1 * G + u2 * self.point

    v = xy.x() % n
    return v == r
```

ECDSA SIGNATURE VERIFICATION. To verify A's signature (r, s) on m, B obtains an authentic copy of A's domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated public key Q. It is recommended that B also validates D and Q (see §5.4 and §6.2). B then does the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute SHA-1(m) and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1G + u_2Q$.
6. If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x-coordinate x_1 of X to an integer $\overline{x}_1$, and compute $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

ECDSA Signature Verification Bypass

```
def ecdsa_verify(self, hash, signature):     # signature = (0,0)
    G = self.generator
    n = G.order()
    r = signature.r                          # r  = 0
    s = signature.s                          # s  = 0
    c = inverse_mod(s, n)                    # c  = s^(-1)    = 0
    u1 = (hash * c) % n                      # u1 = hash * 0 = 0
    u2 = (r * c) % n                         # u2 = 0 * 0     = 0
    xy = u1 * G + u2 * self.point            # xy = 0G + 0P   = 0

    v = xy.x() % n                           # v = 0.x        = 0
    return v == r                            # 0 == 0 is always true
```

Easy signature verification bypass*:* *σ = (0, 0)* is a valid signature for *any* message and *any* public key!

CVE-2021-{43568—43572}
Arbitrary ECDSA signature forgery due to missing check r ≠ 0, s ≠ 0.

View Advisory

CVE-2022-21449
ECDSA signature forgery in Java 15 and above.

View Advisory

Side-channel Attacks

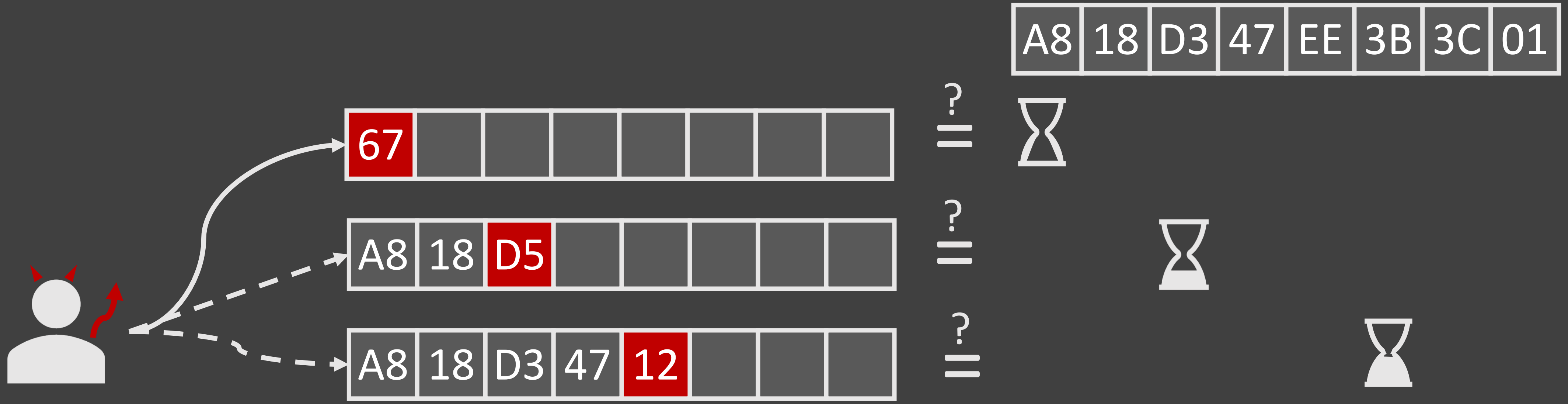
- Leaks from timing, power consumption, cache/memory usage
- Padding oracles may result from attempted decryption
  - Vaudenay's CBC padding oracle attack
  - Bleichenbacher's attack on RSA-PKCS #1 v1.5 encryption
- Constant-time implementations
  - Special algorithms for modular exponentiation, EC point multiplication
  - Must avoid conditionals, early returns, data-dependent accesses, etc.
  - Issues introduced by compiler optimizations (e.g., "bool")

DFinity Threshold ECDSA Cryptography Review
Conditional Assignment Is Not Constant-Time

View Report

nccgroup

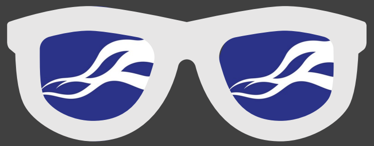
HMAC Timing Side-Channel Attack



- Non constant-time implementations return early in case of failure
- E.g., using libc's `memcmp()` function will return false as soon as it encounters a different byte

Conclusion

- Use vetted, industry-standard crypto libraries
- Don’t reinvent the wheel, follow the standards
- Validate all inputs!
- Test! Test! Test! (unit, integration/end-to-end, fuzzing)
- Hire an additional pair of eyes



nccgroup