Lessons from 3 years of **crypto and blockchain** security audits

JP Aumasson

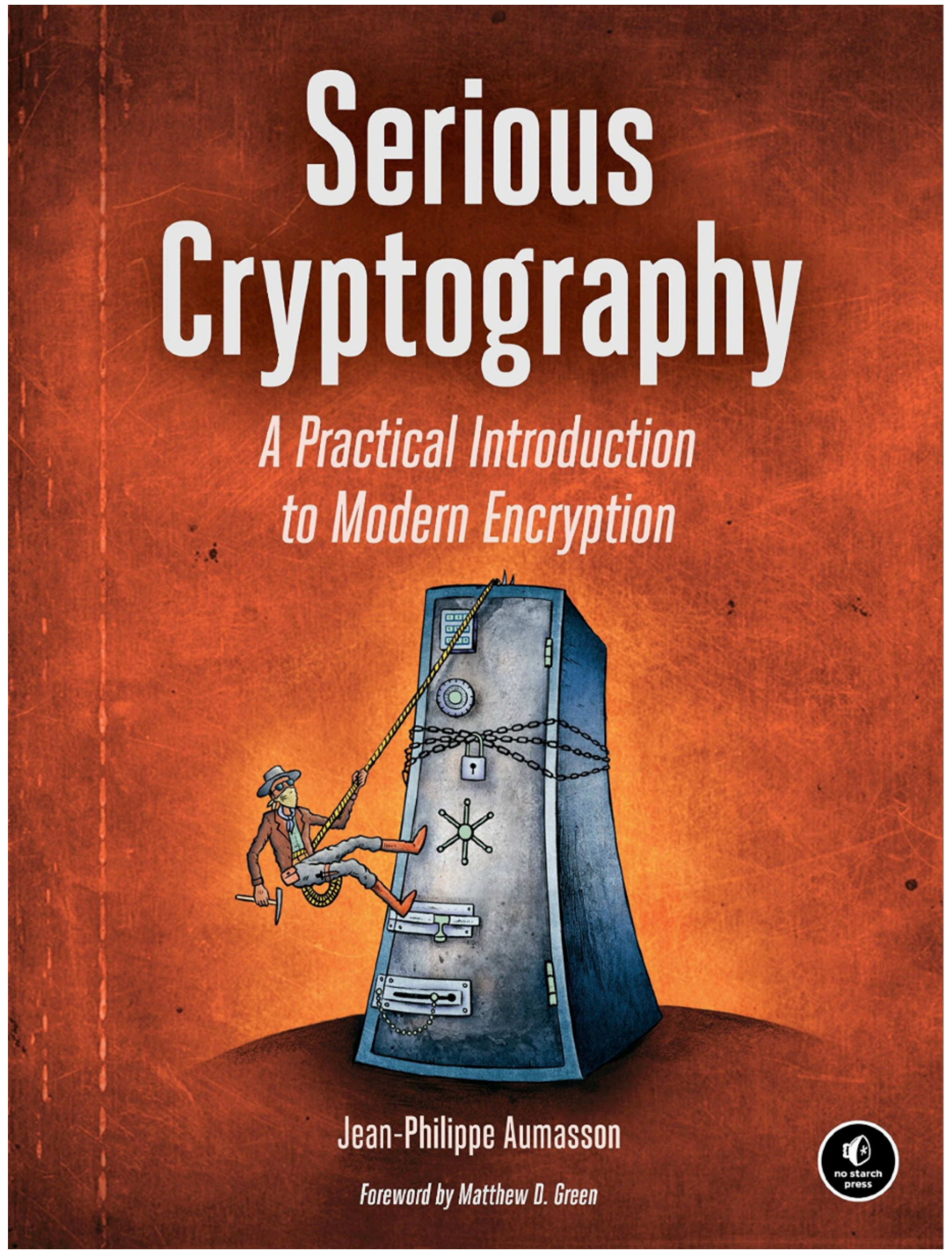
KUDELSKI SECURITY

Crypto audits lead @ KUDELSKI SECURITY

I do crypto, live in Switzerland

https://**aumasson.jp** **@veorq**

Teserakt TAURUS



People also ask

What does it mean to audit something?

an official examination and verification of financial accounts and records. 2. a final report detailing an **audit**. 3. the inspection or examination of **something**, as a building, to determine its safety, efficiency, or the like.

We look for security issues and help fix them

In **source code**, mainly C(++), JS, Rust, Java, Go

Sometimes documentation is available

We **get paid** for it (unless we do it for fun)

Reports are sometimes published

Include findings, recommendations, status


KUDELSKI SECURITY

RandomX Security Audit

Final Report, 2019-07-02

FOR PUBLIC RELEASE

2.4 BEAM-F-004: Weak password key derivation

Severity: Medium

Description

The keystore encryption key is directly taken as the SHA-256 of the password, allowing efficient bruteforce search of the password and possibly offline attacks if one of the blocks is predictable:

```
void init_aes_enc(AES::Encoder& enc, const void* password, size_t passwordLen) {
    ECC::NoLeak<ECC::Hash::Processor> hp;
    ECC::NoLeak<ECC::Hash::Value> key;
    hp.V.Write(password, passwordLen);
    hp.V >> key.V;
```



```
    enc.Init(key.V.m_pData);
}
```

Recommendation

We recommend to use a password hashing function that mitigates bruteforce attacks by being slow, such as PBKDF2 (with at least 50000 iterations) or Argon2.

Status

Beam fixed this by removing the weak password derivation.

Agenda

1. Common **crypto bugs** from real audits
2. The case of **Rust**: typical bugs and recommendations
3. What we’ve **learnt**; tips for auditors and customers

Agenda

1. Common **crypto bugs** from real audits

2. The case of **Rust**: typical bugs and recommendations

3. What we’ve **learnt**; tips for auditors and customers

Agenda

1. Common **crypto bugs** from real audits

2. The case of **Rust**: typical bugs and recommendations

3. What we’ve **learnt**; tips for auditors and customers

Finding bugs is easy
Writing bug-free code is hard
Writing bug-free crypto is harder

Bug#1
Strong cipher yet weak encryption

```
addrAttrNonce :: ByteString
addrAttrNonce = "serokellfore"

-- | Serialize tree path and encrypt it using HDPassphrase via ChaChaPoly1305.
packHDAddressAttr :: HDPassphrase -> [Word32] -> HDAddressPayload
packHDAddressAttr (HDPassphrase passphrase) path = do
  let !pathSer = serialize' path
  let !packCF = encryptChaChaPoly addrAttrNonce passphrase "" pathSer
  case packCF of
    CryptoFailed er -> panic $ "Error in packHDAddressAttr: " <> show er
    CryptoPassed p  -> HDAddressPayload p
```

Found in a major cryptocurrency wallet, totally defeats encryption

Bug#2
Weak key derivation from a password

```
encryption_key = SHA-256(password)
```

Encryption key then easy to break

Need to use a password hash with salt and cost

Found in several audits (with various hash functions)

Bug#3
Hijacking accounts in a $3B cryptocurrency

```
(publicKey, privateKey) = deriveKey(seed)

address = hash(publicKey)
```

With **64-bit** `address`, what can go wrong?

Bug#3
Hijacking accounts in a $3B cryptocurrency

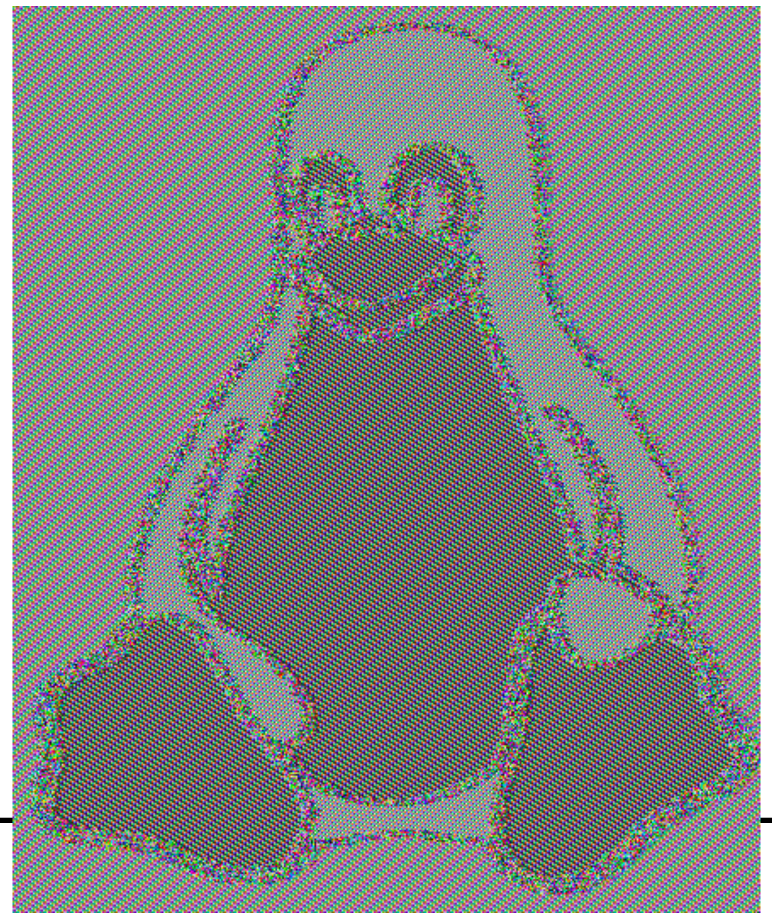
```
(publicKey, privateKey) = deriveKey(seed)

address = hash(publicKey)
```

With **64-bit** **`address`**, what can go wrong?

Find another key pair with the same address in 2^{64} elliptic curve operations, exploitable to **hijack accounts**, unfixable

Bug#4
Weak encryption in credentials store

```
void aes_encrypt(void* buffer, size_t bufferLen, const void* password, size_t
 ↪  passwordLen) {
    AES::Encoder enc;
    init_aes_enc(enc, password, passwordLen);
    uint8_t* p = (uint8_t*)buffer;
    uint8_t* end = p + bufferLen;
    for (; p<end; p+=AES::s_BlockSize) {
        enc.Proceed(p, p);
    }
}
```



Found in an anonymous cryptocurrency wallet

Bug#5
Flaws in NFC cryptocurrency wallet

Symmetric key sent in clear

Hash(PIN) sent to unauthenticated receivers

Default PIN length of 3 digits

Control commands sent without authentication (spoofable)

Bug#6
Entropy data ignored in key generation

In a BIP32 hierarchical key derivation software

Generating an address from a 64-byte seed:

```
$ echo bc0ef283f57fd5e4f36657053228eae8d2d5b0e4d87c6ee069a9cade39411d63 |
bip32gen -x -i entropy  -o addr m
1Jzuo5xm62i8gFQLQb58f2F5a7nTK3o8bD
```

Bug#6
Entropy data ignored in key generation

In a BIP32 hierarchical key derivation software

Generating an address from a 64-byte seed:

```
$ echo bc0ef283f57fd5e4f36657053228eae8d2d5b0e4d87c6ee069a9cade39411d63 |
bip32gen -x -i entropy  -o addr m
1Jzuo5xm62i8gFQLQb58f2F5a7nTK3o8bD
```

When truncating the seed to 32 bytes, same result. 🤔

```
$ echo bc0ef283f57fd5e4f36657053228eae8 |
bip32gen -x -i entropy -o addr m
1Jzuo5xm62i8gFQLQb58f2F5a7nTK3o8bD
```

Agenda

1. Common **crypto bugs** from real audits
2. The case of **Rust**: typical bugs and recommendations
3. What we've **learnt**; tips for auditors and customers

Memory-safe system language, using reference counting (no GC)

Used more and more for crypto, for its **safety and performance**

Example: a large part of **Zcash**'s reference code is in Rust

Pre-auditing

```
cargo test

cargo clippy

cargo audit

rg unsafe
```

`unsafe` can be unsafe

`unsafe` blocks of code can **break memory safety** — typically used when using raw pointers in FFI calls

Review all unsafe blocks for e.g. out-of-bound read/write

```
#[no_mangle]
pub extern "C" fn wallet_from_seed(seed_ptr: *const c_uchar, out: *mut c_uchar) {
    let seed = unsafe { read_seed(seed_ptr) };
    let xprv = hdwallet::XPrv::generate_from_seed(&seed);
    unsafe { write_xprv(&xprv, out) }
}
```

```
unsafe fn read_seed(seed_ptr: *const c_uchar) -> hdwallet::Seed {
        let seed_slice = std::slice::from_raw_parts(seed_ptr, hdwallet::SEED_SIZE);
        hdwallet::Seed::from_slice(seed_slice).unwrap()
}
```

Careful with `unwrap()`

`unwrap()` will **panic** if the `Option`/`Result` processed is `None`/`Err`

To avoid DoS, panic should be reserved for unrecoverable errors

Example from an audit, where `deserialize()` can return `Err`

```
impl RawBlock {
    pub fn from_dat(dat: Vec<u8>) -> Self { RawBlock(dat) }
    pub fn decode(&self) -> cbor_event::Result<Block> {
 ↪   RawCbor::from(&self.0).deserialize() }
    pub fn to_header(&self) -> RawBlockHeader {

 ↪   // TODO optimise if possible with the CBOR structure by skipping some prefix
        let blk = self.decode().unwrap();
        blk.get_header().to_raw()
    }
}
```

Zeroize or not zeroize?

Sensitive values can be reliably erased/zeroized in C(++)

Usually not in garbage-collected languages (e.g. Go, Java, JS)

What about Rust?

Zeroize or not zeroize?

More reliable for heap than stack (no control on stack allocator)

Caveats: moves, copies, heap reallocations, etc.

Consider using the crate `zeroize`

Crypto and Rust

Rust programmers tend to be good programmers – fewer bugs per LoC

Fewer tools available than for C, but these are mostly useless anyway :)

Potential **timing leaks** usually easy to notice…

2.7 KZENC-F-007: Possible Timing Leak in Mpz::Modulo::mod_sub

Severity: Low

Description

In `big_gmp.rs`, the `Mpz::Modulo::mod_sub()` function is implemented as follows:

```
fn mod_sub(a: &Self, b: &Self, modulus: &Self) -> Self {
    let a_m = a.mod_floor(modulus);
    let b_m = b.mod_floor(modulus);
    if a_m >= b_m {
        (a_m - b_m).mod_floor(modulus)
    } else {
        (a + (-b + modulus)).mod_floor(modulus)
    }
}
```

2.8 KZENC-F-008: Possible Timing Attack in ECScalar::from()

Severity: Low

Description

In `ed25519.rs`, the `ECScalar::from()` function is implemented as follows:

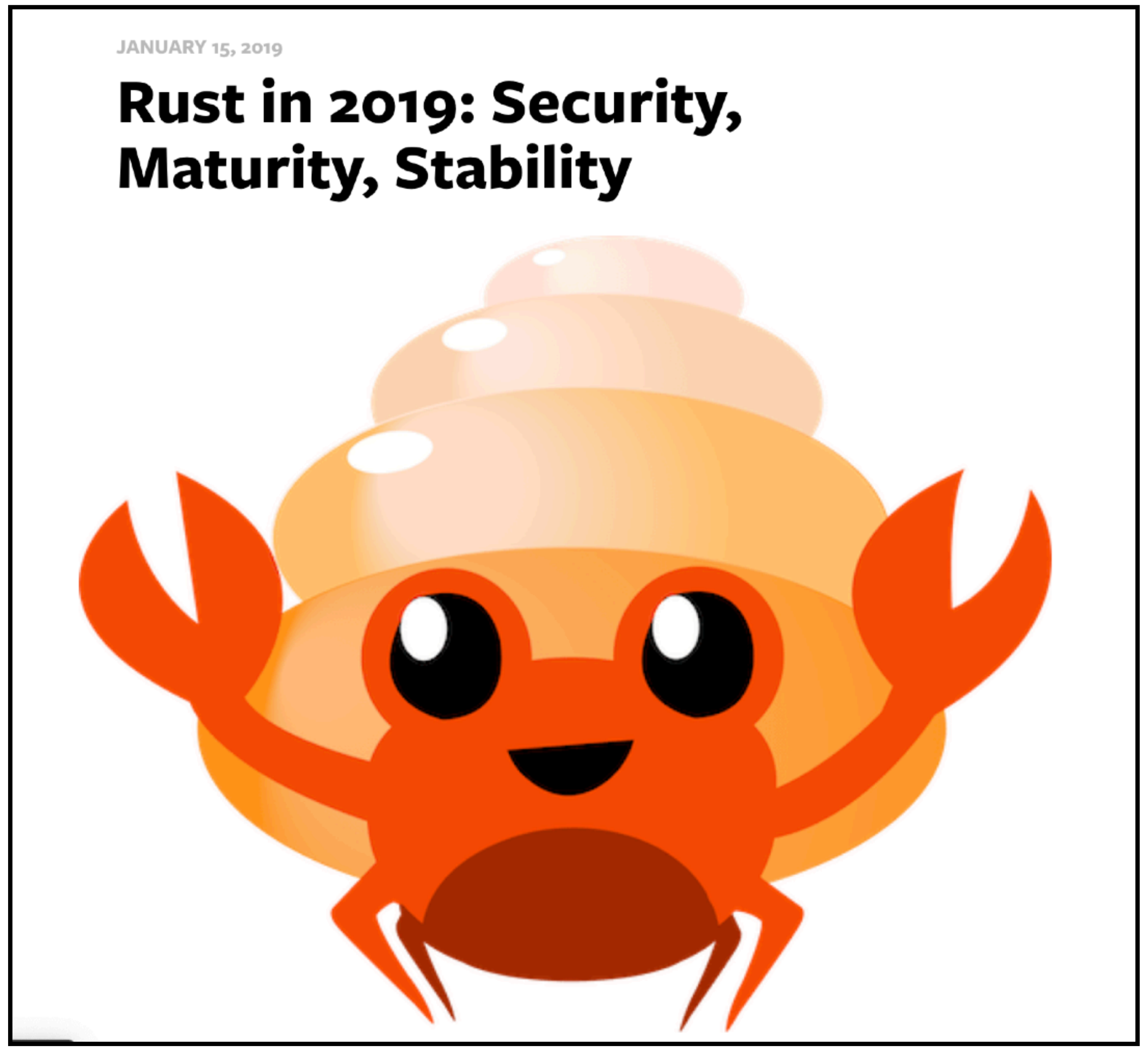
```
    fn from(n: &BigInt) -> Ed25519Scalar {
        let mut v = BigInt::to_vec(&n);
        let mut bytes_array_32: [u8; 32];
        if v.len() < SECRET_KEY_SIZE {
            let mut template = vec![0; SECRET_KEY_SIZE - v.len()];
            template.extend_from_slice(&v);
            v = template;
        }
        bytes_array_32 = [0; SECRET_KEY_SIZE];
        let bytes = &v[..SECRET_KEY_SIZE];
        bytes_array_32.copy_from_slice(&bytes);
        bytes_array_32.reverse();
        Ed25519Scalar {
            purpose: "from_big_int",
        fe: SK::from_bytes(&bytes_array_32),
        }
    }
```

The conditional `if` statement before padding introduces a possible timing leak in case the secret key has a lot of leading zeroes.

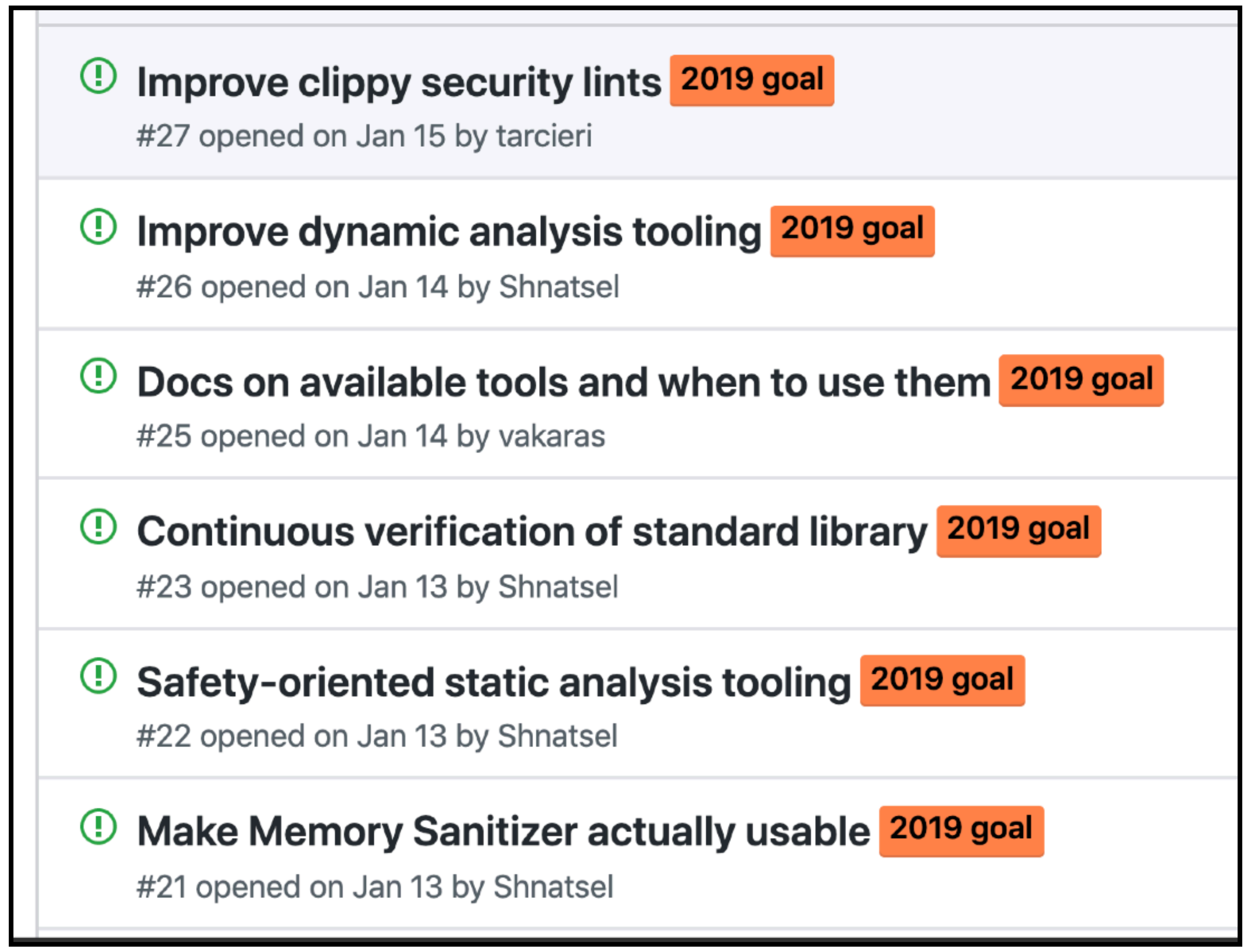
References

There's much more to say about Rust and its security

https://tonyarcieri.com/rust-in-2019-security-maturity-stability



https://github.com/rust-secure-code/


Improve clippy security lints 2019 goal
#27 opened on Jan 15 by tarcieri

Improve dynamic analysis tooling 2019 goal
#26 opened on Jan 14 by Shnatsel

Docs on available tools and when to use them 2019 goal
#25 opened on Jan 14 by vakaras

Continuous verification of standard library 2019 goal
#23 opened on Jan 13 by Shnatsel

Safety-oriented static analysis tooling 2019 goal
#22 opened on Jan 13 by Shnatsel

Make Memory Sanitizer actually usable 2019 goal
#21 opened on Jan 13 by Shnatsel

Agenda

1. Common **crypto bugs** from real audits

2. The case of **Rust**: typical bugs and recommendations

3. What we’ve **learnt**; tips for auditors and customers

The situation is much better than 10 years ago

Cryptography is easier to use, the average developers understands more crypto, more resources and software

Many crypto audits are **not much about crypto**

Language knowledge and familiarity with all classes of bugs at least as important as pure crypto knowledge

Both sides must be prepared

Auditor: Be familiar with the kind of system/protocol audited, its components, security notions, language/frameworks

Customer: Provide a description of critical assets and functionalities, intended behavior, documentation, security model

Scoping and effort estimate is hard

There are trade-offs between completeness, flexibility, and cost

`cloc` results are useful but not sufficient

Distribution of the time of findings' varies

Sometimes most issues found at the **beginning** of the audit; low-hanging fruits then diminishing returns

Sometimes later, because of the **learning curve**

(Depends on the functionality, code and system complexity)

Severity ratings is not always easy

Should be risk-based (impact × exploitability), as in CVSS

Overestimation is more common than underestimation

A cryptographer may cringe if they see MD5 or AES-ECB used, but these may not be actual security issues

Understand the security model

For example, when reviewing a proof-of-work, consider attacks by both block authors and miners

Empathize with developers

After writing the report, read it and imagine that you’re the developer who wrote the code, and revise the tone accordingly

Provide a clear description, mitigation suggestions, links to relevant documentation/articles, review the patch

Communicate, report findings

Establish a group chat with developers, ask questions, report findings to 1) know if relevant or FP/incorrect, 2) help developers mitigate earlier

Audits are no security guarantee

Security audits tend to be broader than they’re deep

Different teams/persons have different fields of expertise

Audit limited in time/scope/budget

Vulnerabilities can be in dependencies/runtime/platform

Thank you!

jpa@pm.me @veorq

kudelskisecurity.com

KUDELSKI SECURITY