TwizSec Library Crate

Surendra Jammishetti

Twizzler

1 Early Goals

The TwizSec crate aims to provide an external library for the Twizzler kernel that has the following goals (summarized from me and Daniels meeting on 11/26/24).

1. storing and receiving capabilities
2. signing and verifying capabilities
3. programming the mmu / io to reflect security policy data

2 Planning

2.1 Needs

The plan is to work on the second item first, as its the path of least resistance. Ideally expose two functions.

1. Takens in capability and signature, returns if they are correct or not
2. Given a capability, construct a signature

2.2 Deps

The kernel has crypto libraries already integrated, use those to build these features currently this is all we got

p256 : https://crates.io/crates/p256
sha2 : https://crates.io/crates/sha2
Which, atleast right now, should have everything we need.

2.3 Capabilities

Currently we dont have a capability struct, so Im going to use what was in the security paper as an example.

Additionally I'm considering making the two functions impl'd onto the struct, so that way they can be called on any capability struct, as I think it would be nice and ergonomic but not sure what others would think.

This is the spec inside the paper

```
CAP := {
    target, accessor : ObjectId,
    permissions, flags : BitField,
    gates: Gates,
    revocation : Revoc,
    siglen: Length,
    sig: u8[],
}
```

3 Explaining the Implementation

3.1 Cap struct

The Cap struct encompasses the functionality of the Capability, having everything but the gates and revocation fields present in the paper, only because I wasnt planning to finish that this time around, maybe next time. I'll list some key decision details below.

- UnsignedCapability => Capability VS Capability::new()

  My first iteration had a UnsignedCap struct that would only have two methods, new(...) -> Self , which would initialize it, and a sign(self) -> Cap, which would consume the unsignedcap and return a signed capability. The second implementation would only have the capability struct, which gets signed on initialization. While I liked how "satisfying" the sign function was in the first implementation, the conciseness of the second implementation is cleaner since its one struct, one function, one capability, so I went ahead with that.

- The Serialize Function

  Since its a no_std environment, the most pressing concern is that there is no std library, nor any memory allocator. Which means that when im trying to "serialize" the contents of the capability struct to hash (where all of the hashing functions take in a [u8]). The easiest solution, in my mind atleast, was to just count how many bytes the array is and just create a new array of that size, copy the bytes of each field over one by one, and boom, serialized capability struct as fast as possible. Just note that once the revoc and gates fields are added, this hashing function will need to be changed to account for them.

- Verification function

  I tried to follow the spec in the security paper as much as I could, the basic steps I take are

  - verify_sig(&self, verifying_key) -¿ Result¡(), CapError¿; // function signature
  - parse hashing_algo and signing scheme from capability flags
  - serialize self and store in buffer
  - use the hashing_algo on the buffer to get the capabilities hash
  - ensure that signing_scheme matches the signing_scheme on verifying_key
  - verify the signature according to the signing scheme using the verifying key

  originally this function took in the private key of the target object to generate the verification key but now, due to Daniel's advice, It takes in a verification key, meaning we only have to generate the verification keys once and can reuse them to verify capabilities

- Errors

  Currently the errors are generalized to the entire library and are very simplistic, which is good but might need to consider expanding them once theres more parts of the sec model implemented.

- testing Theres only one pretty simple test that creates a capability for one object to access another, and then generates a verification key and tests if the capability can be verified. It works!

- Benchmarking While yes its fairly early to have benchmarks, fast code => happiness so I imported a reliable benchmarking crate as a dev-dependency and tested some core capability ops (can be found in /benches/capability_ops.rs). Good news, its pretty good!

3.2 Literally anything else

check out some of the code comments or message me and I'd gladly answer.