Security Best Practices

By Venkkatesh Sekar and Roel Storms

January 16th, 2023

WHOAMI?

Security Best Practices: Canisters and Web Apps

Security Best Practices > Rust Canister Development Security Best Practices

Rust Canister Development Security Best Practices

Smart Contracts Canister Control

Use a decentralized governance system like SNS to make a canister have a decentralized controller

Security Concern

The controller of a canister can change / update the canister whenever they like. If a canister e.g. stores assets such as ICP, this effectively means that the controller can steal these by updating the canister and transfer the cycles to their account.

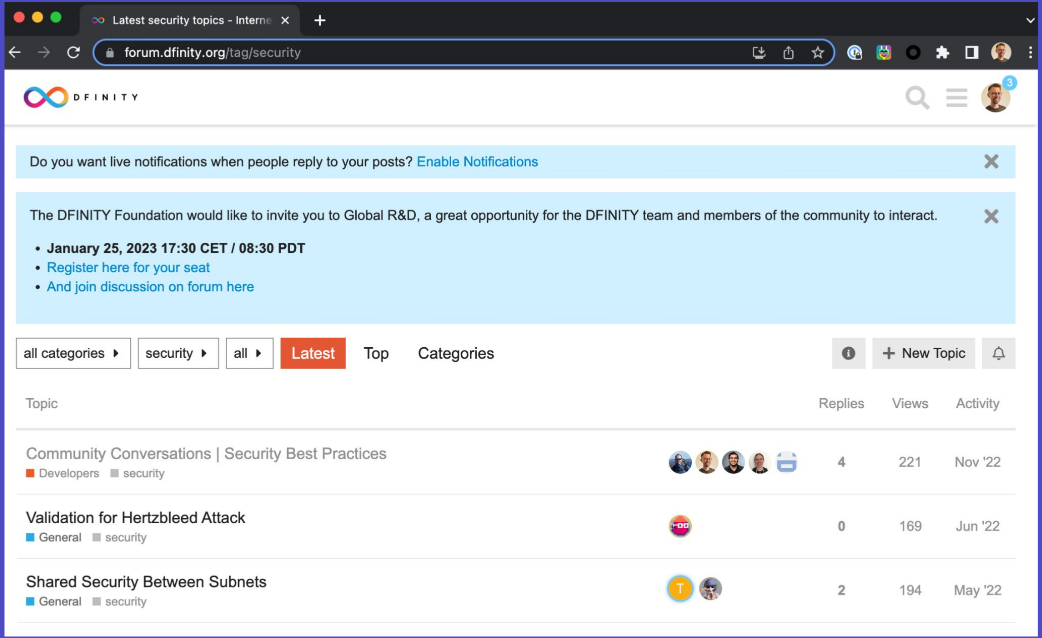
Recommendation

- Consider passing canister control to a decentralized governance system such as the Internet Computer's Service Nervous System (SNS), so that changes to the canister are only executed if the SNS community approves them collectively through voting. If an SNS is used, use an SNS on the SNS subnet as this guarantees that the SNS is running an NNS-blessed version and maintained as part of the IC. These SNSs will be available soon. See the roadmap here and the design proposal here
- Another option would be to create an immutable canister smart contract by removing the canister controller completely. However, note that this implies that the canister cannot be upgraded, which may have severe

- Inspired by actual security bugs
- Raise awareness
- Address issues early in the dev lifecycle
- Third party audits still recommended

https://internetcomputer.org/docs/current/references/security/

Security Best Practices: Forum Discussion



Questions and feedback welcome!

https://forum.dfinity.org/tag/security

Outline

Inter-canister calls and state changes	**30min**
Randomness	**7min**
Time API	**6min**
Certified Variable	**7min**
QA	**10min**

Inter-Canister Calls and State Changes

Example: Canister Refund ICP for Users

SimpleLedger Canister

User (Principal)	ICP balance
Roel	0
Venkkatesh	0
Robin	0
DEX Account	21’007

refund

DEX Canister

HashMap

`local_balances`	
User (Principal)	**ICP balance**
Roel	20’000
Venkkatesh	1’000
Robin	7

Example: Canister Refund ICP for Users

SimpleLedger Canister

User (Principal)	ICP balance
Roel	0
Venkkatesh	0
Robin	7
DEX Account	21’000

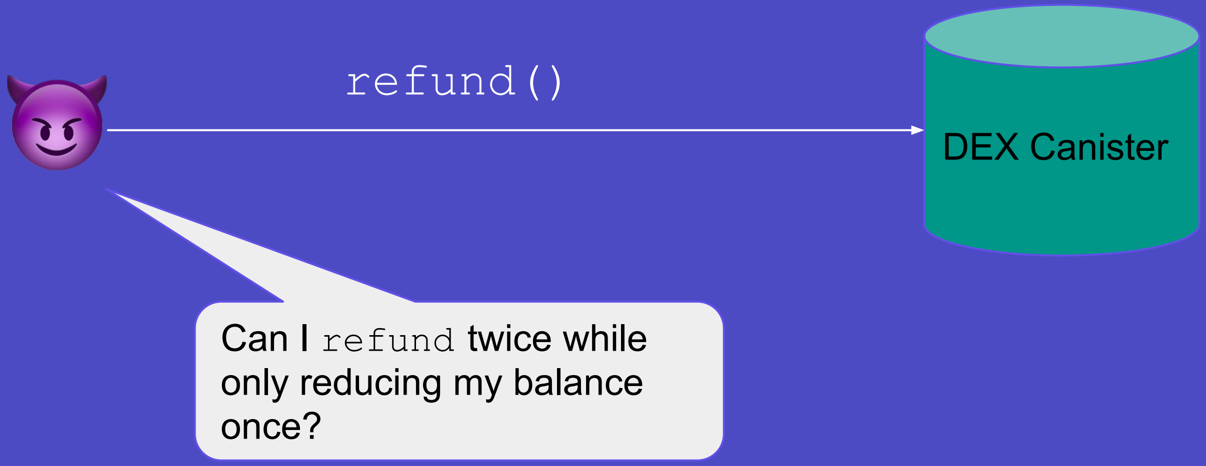
refund

DEX Canister

HashMap

`local_balances`	
User (Principal)	**ICP balance**
Roel	20’000
Venkkatesh	1’000
Robin	0

Motivation: Double-Spending Issues

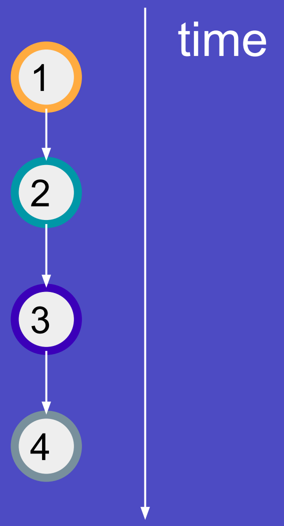


- Double-spending issues related to inter-canister calls / messaging.
- How to avoid these?

Message Execution Basics

A *message* is a set of consecutive instructions.

Only a single message is processed at a time.



Message Execution Basics

Each call (query / update) triggers a message. When an inter-canister call is made, the code after the call (the *callback*) is executed as a separate message.

call

```
public shared func example () : async Result {
  // some code
  let result = await SomeCanister.some_function();
  // some code
  #ok {}
};
```

First message: the code until the inter-canister call is made 1

Second message: when the inter-canister call returns, the *callback* is invoked. 2

Message scheduling:

1

2

Messaging Model

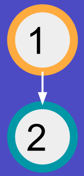
Each call (query / update) triggers a message. When an inter-canister call is made, the code after the call (the *callback*) is executed as a separate message.

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
  let caller_balance = Option.get(local_balances.get(caller),0);
  let result = await SimpleLedger.deposit(caller_balance, caller?);
  if (Result.isOk(result)) {
    local_balances.put(caller, 0);
  };
  update_statistics_after_refund(caller);
  return result;
};
```

First message: the code until the inter-canister call is made 1

Second message: when the inter-canister call returns, the *callback* is invoked. 2

Message scheduling: Call to `refund()`



Trap after `await`

On a trap / panic, modifications to the state for the current message are not applied.

```
public shared func example () : async Result {
  // some code   Trap
  let result = await some_inter_canister_call();
  // some code   Trap
  #ok {}
};
```

Trap after `await`

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
  let caller_balance = Option.get(local_balances.get(caller),0);
    let result = await SimpleLedger.deposit(caller_balance, caller?);
  if (Result.isOk(result)) {
    local_balances.put(caller, 0);
  };
  update_statistics_after_refund(caller);
  return result;
;
```

Trap

Assume after some number of calls, this method always traps due to a full data structure.

Security bug: due to the trap, `local_balances` are never reduced. An attacker can refund the balance multiple times, stealing other user’s ICP!

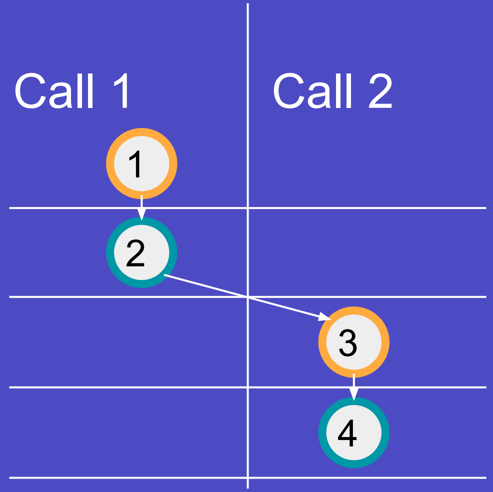
Recommendation: Avoid traps / panics after await

Message Scheduling

Let’s call `refund` twice in parallel

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
 let caller_balance = Option.get(local_balances.get(caller),0);
   let result = await SimpleLedger.deposit(caller_balance, caller?);
 if (Result.isOk(result)) {
   local_balances.put(caller, 0);
 };
 #ok {}
};
```

Possible ordering:



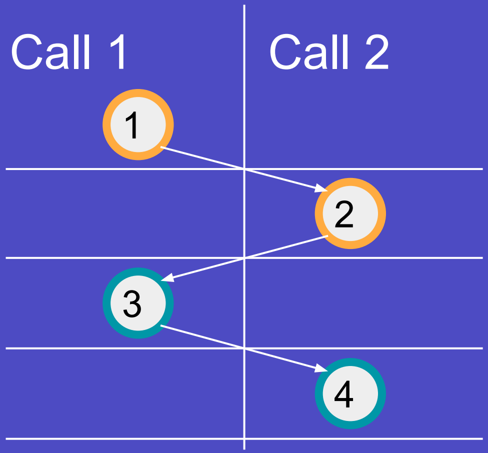
Works as intended: on message 3, `local_balances == 0` and no refund is issued in Call 2.

Message Scheduling

Let’s call `refund` twice in parallel

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
  let caller_balance = Option.get(local_balances.get(caller),0);
    let result = await SimpleLedger.deposit(caller_balance, caller?);
  if (Result.isOk(result)) {
    local_balances.put(caller, 0);
  };
  #ok {}
};
```

Possible ordering:



Security Bug: when message 3 is executed, the `balance` is not (yet) `0` and thus the refund is issued again.

This is a “double spending” bug

Could get refund many times by issuing many calls

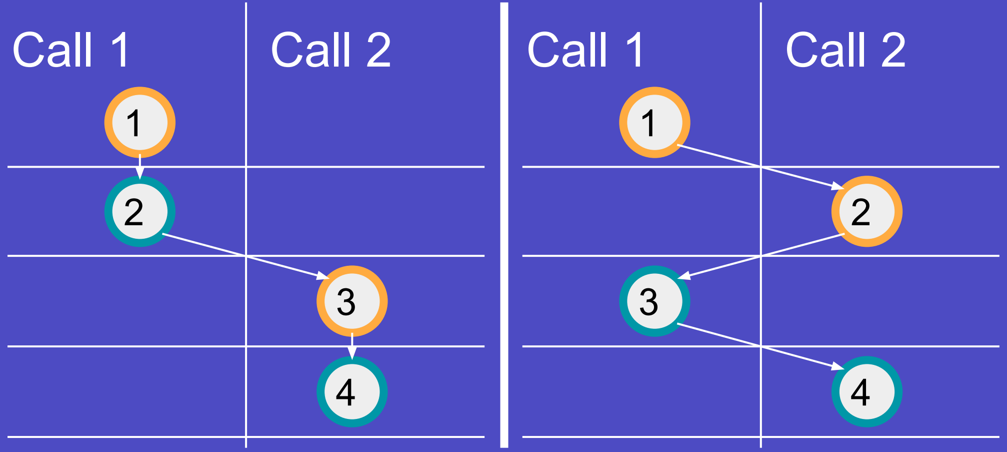
Message Execution Basics

Messages from interleaving calls have no reliable execution ordering.

Let's issue two calls in parallel

```
public shared func example () : async Result {
  // some code
  let result = await some_inter_canister_call();
  // some code
  #ok {}
};
```

Possible orderings (examples):



Example: Paying Refunds

Any user can call `refund()` to transfer their balance back to their ICP account:

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
 let caller_balance = Option.get(local_balances.get(caller),0);
 let result = await SimpleLedger.deposit(caller_balance, caller?);
 if (Result.isOk(result)) {
   local_balances.put(caller, 0);
 };
 update_statistics_after_refund(caller);
 return result;
};
```

The calling principal, read from the system API

Read the caller's balance

Issue an inter-canister call to the ledger to refund the ICP

Set the user's balance to zero

Inter-Canister Calls and State Changes

—

“Avoid Panics after `await`”

Inter-Canister Calls and State Changes

—

“Be aware that there is no reliable message ordering”

Message Scheduling

Callbacks are executed as separate “messages” on the IC:

```
public shared ({caller}) func refund () : async Result.Result<Text, Text> {
  let caller_balance = Option.get(local_balances.get(caller),0);
    let result = await SimpleLedger.deposit(caller_balance, caller?);
  if (Result.isOk(result)) {
    local_balances.put(caller, 0);
  };
  #ok {}
};
```

First message: the code until the inter-canister call is made 1

Second message: when the inter-canister call returns, the “callback” is invoked. 2

Message execution: Call to `refund()`

1

2

Recall: If there are several messages to be executed, there is no reliable ordering of their execution.

How to Solve this Issue? - Locking Pattern

Such issues can be addressed using locks. In our example, we can make sure that there is at most one refund happening per caller:

```
func some_call () {
    check_lock(caller);
    lock(caller);
    // read/modify state, make inter-canister calls
    unlock(caller);
    // return result
};
```

Return immediately if the caller is locked.

Lock the caller

Release the lock

How to Solve this Issue? - Locking Pattern

```
public shared ({caller}) func refund () : async Result.Re     e, Text
 if (Option.isSome(ongoing_transactions.get(caller))) {
   return #err "transaction already in progress";
 };
 ongoing_transactions.put(caller);
 let caller_balance = Option.get(local_balances.get(caller),0);
let result = await SimpleLedger.deposit(caller_balance, caller?);
 if (Result.isOk(result)) {
   local_balances.put(caller, 0);
 };
 let _ = ongoing_transactions.remove(caller);
 #ok {}
};
```

Use `ongoing_transactions` to *lock* callers that have a refund in progress

Return immediately if the caller is locked.

If the caller is not locked, lock it!

Release the lock even if the ledger returned an error

Recommendations

- Review all inter-canister calls (`await`) carefully:
  - If two messages access (read / write) the same state, is it possible to find a scheduling of these messages that leads to illegal transactions or inconsistent state?
- Employ locking mechanisms to avoid such bugs
- For Rust developers: Release locks in your `Drop` implementation

Randomness

Why do we need Secure Randomness?

- Minting a randomized NFT from a collection.
- Airdropping tokens to a randomized set of participants among a pool.
- Procedural generation / casino based games.

Requirements for secure randomness

- Unbiased : The value shouldn't be influenced by anyone
- Unpredictable : The value is unknown to anyone before its released.

Source of Randomness

- In Linux, the seed is consumed from different peripherals and exposed via `/dev/random` and `/dev/urandom` through a CSPRNG
- In Windows, the seed is collected from a variety of sources and store at *`HKEY_LOCAL_MACHINE\SYSTEM\RNG\Seed`*
- Cloudflare uses LavaRand for Randomness, which utilizes an picture of an array of lava lamps as seed for its CSPRNG.

Randomness in IC

- The IC exposes a System API `ic0.raw_rand` which accepts no input and returns 32 bytes of cryptographically secure randomness.
- The return value is unknown to any part of the IC at time of the submission of this call.
- Every new call to this method generates a new return value.

In Motoko:

```
import Random “mo:base/Random”;
let entropy = Random.blob(); // 32 bytes
```

Using the Randomness

There are two variants of methods present in Motoko to consume the secure randomness and return discrete outputs.

1. Provide the secure randomness directly into the methods.
   - Developers need to take care of randomness management
   - Always provide fresh randomness to your methods.

```
import Random “mo:base/Random”;
let entropy = Random.blob();
var random_byte1 = Random.byteFrom(entropy); // consumes the first byte of randomness
var random_byte2 = Random.byteFrom(entropy); // consumes the first byte of randomness again.
assert (random_byte1 == random_byte2);              // always TRUE
```

The `Finite` Class

2. Seed the `Finite` class with randomness and use its methods. (Recommended!)

- The class will handle the randomness management.
- Once the randomness is exhausted, further method invocations will return `null`.
- *At this point, it needs to be reseeded with fresh randomness*.

```
import Random “mo:base/Random”;
let entropy = Random.blob();
var f = Random.Finite(entropy);
var random_byte1 = f.byte(); // consumes first byte of the randomness
var random_byte2 = f.byte(); // consumes second byte of the randomness
assert (random_byte1 == random_byte2); // can be TRUE or FALSE
```

Now let’s see an live example of using the `Finite` class!

Time

Time is not strictly increasing in the IC

- Canisters can obtain current time by querying the system API `ic0.time()` which gives the nanoseconds since 1970-01-01.
- In Motoko,

```
import Time “mo:base/Time”;
let time1 : Int = await Time.now();
Let time2 : Int = await Time.now();
// IC guarantees that time2 >= time1
```

- From the view of a canister, the API can return same time on multiple invocations as long as the messages are executed in the same block.

Security Concerns

Let us consider an example : Financial transactions

- In traditional systems, timestamps are sufficient to know when a transaction happened and achieve an order among other transactions.
- In IC, for a single canister, the timestamps alone are not sufficient.
- One must employ logical counters or locks to safely guarantee the order of the transactions.

Now let’s see a live example of the Time API.

Bonus : Security Concerns (Contd.)

- The times observed by different canisters are unrelated, and calls from one canister to another may appear to travel "backwards in time".
- Hence, there is no reliable way to order transactions across multiple canisters by only depending on current time.
- But can you? Yes, by using logical clocks. Check out https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Certified Variables

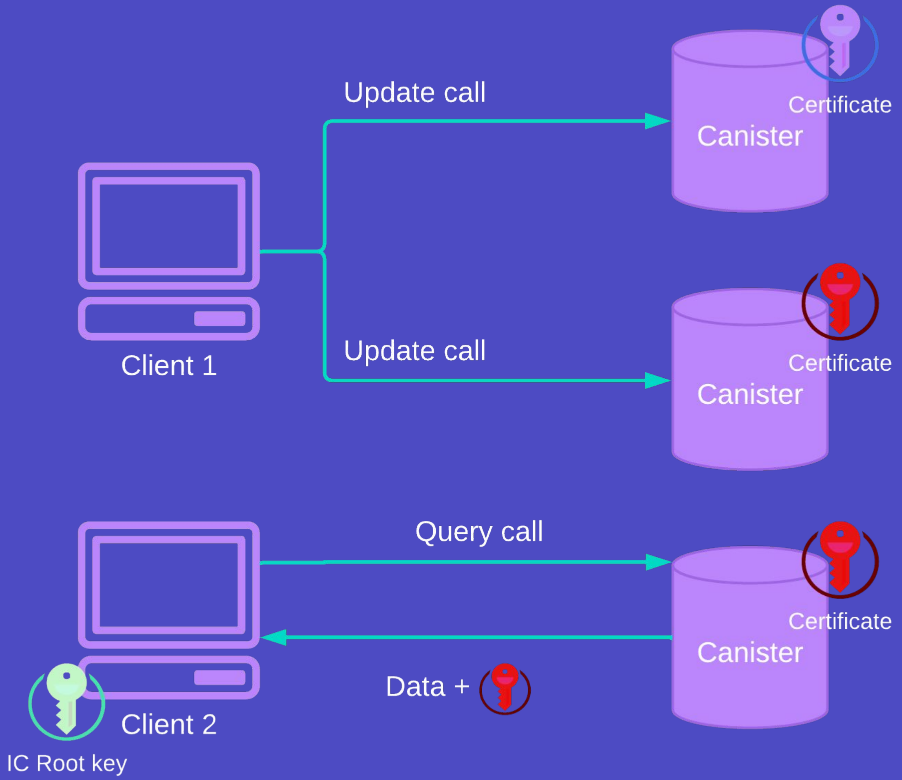
Query call

- Query call is analogous to READ
- The query call is executed on a single replica node chosen on random in a subnet.
- Very fast response time.
- Relatively less secure since there is no integrity protection. The responding node maybe honest or malicious.

Update call

- Update call is analogous to READ and WRITE
- On an update call, the request is replicated among multiple nodes, goes through consensus and the response is threshold signed by the subnet.
- Relatively slow and more expensive. (depending on the size of the subnet).
- Secure response, since it’s certified by the subnet and guarantees integrity.

Best of both worlds : `CertifiedData`



Certificate for the data is set.

Since the data is updated, the certificate is also updated.

On the query call, the data and certificate is sent in the response.

Best of both worlds : `CertifiedData (Contd)`

- To obtain query like response times and update like certified response, developers could use `CertifiedData`.
- Now, a canister can store a small amount of data (32 bytes) during an update call.
- During query call, the canister can obtain a certificate about that data, which can be validated by the IC root public key.

- Structure of the Certificate :
    - HashTree - A data structure which allows storing hashes of values as leaves to build a single root hash which can be certified. (Root hash is 32 bytes).
    - Signature - A threshold signature on the root hash of the tree by the subnet public key.
    - Delegation - Link the subnet public key to the the IC root key.

Let’s see a live example of using `CertifiedData` with a simple Counter Canister.

Multiple Variables & Verifying Certification

In practice however,

- Since a canister can store only 32 bytes of CertifiedData, the solution doesn't scale for multiple variables or large amounts of data.
  - Rust developers currently use ic-certified-map to store hashes of multiple variables in a map which can be consumed into a single hash to fit into 32 bytes.
  - For Motoko, this is currently under construction. https://github.com/dfinity/motoko-base/issues/409
- Verifying the certification is a lot more involved and it is recommended to use known clients like `agent-js` to perform validation.
- For more information on certification : https://internetcomputer.org/docs/current/references/ic-interface-spec#certification

To see this in practice, try executing

```
dfx canister --network ic call rkp4c-7iaaa-aaaaa-aaaca-cai get_average_icp_xdr_conversion_rate
```

(This is a query to the Cycles minting canister to obtain the average ICP<>XDR conversion rate, which is exposed along with the certificate using certified data)

Q&A