Smart Contracts Section from Aptos Documentation

Overview: Smart Contracts

(From https://aptos.dev/build/smart-contracts)

Aptos contracts are written using Move, a next generation language for secure, sandboxed, and formally verified programming which is used for multiple chains. Move allows developers to write programs that flexibly manage and transfer assets while providing the security and protections against attacks on those assets.

Here is a `hello_blockchain` example of move

hello_blockchain.move

```
module hello_blockchain::message {    use std::error;    use std::signer;    use std::string;    use aptos_framework
    //:!:>resource    struct MessageHolder has key {        message: string::String,    }    //<:!:resource
    #[event]    struct MessageChange has drop, store {        account: address,        from_message: string::String,
    /// There is no message present    const ENO_MESSAGE: u64 = 0;
    #[view]    public fun get_message(addr: address): string::String acquires MessageHolder {        assert!(exists<
    public entry fun set_message(account: signer, message: string::String)    acquires MessageHolder {        let ac
    #[test(account = @0x1)]    public entry fun sender_can_set_message(account: signer) acquires MessageHolder {
        assert!(            get_message(addr) == string::utf8(b"Hello, Blockchain"),            ENO_MESSAGE        )
```

We have a new Move on Aptos compiler that supports Move 2. See this page for more information.

The Move Book

(From https://aptos.dev/build/smart-contracts/book)

The Move Book

Welcome to Move, a next generation language for secure, sandboxed, and formally verified programming. It has been used as the smart contract language for several blockchains including Aptos. Move allows developers to write programs that flexibly manage and transfer assets, while providing the security and protections against attacks on those assets. However, Move has been developed with use cases in mind outside a blockchain context as well.

Move takes its cue from Rust by using resource types with move (hence the name) semantics as an explicit representation of digital assets, such as currency.

Move was designed and created as a secure, verified, yet flexible programming language. The first use of Move is for the implementation of the Diem blockchain, and it is currently being used on Aptos.

This book is suitable for developers with some programming experience and who want to begin understanding the core programming language and see examples of its usage.

Begin with understanding modules and scripts and then work through the Move Tutorial.

Your First Move Module

(From https://aptos.dev/build/guides/first-move-module)

Your First Move Module

The Aptos blockchain allows developers to write Turing complete smart contracts (called "modules") with the secure-by-design Move language. Smart contracts enable users to send money with the blockchain, but also write arbitrary code, even games! It all starts with the Aptos CLI creating an account which will store the deployed ("published") Move module.

This tutorial will help you understand Move Modules by guiding you through setting up a minimal Aptos environment, then how to compile, test, publish and interact with Move modules on the Aptos Blockchain. You will learn how to:

0. Setup your environment, install the CLI
1. Create a devnet account and fund it
2. Compile and test a Move module
3. Publish (or "deploy") a Move module to the Aptos blockchain
4. Interact with the module
5. Keep building with Aptos (next steps)

Changes to the blockchain are called "transactions", and they require an account to pay the network fee ("gas fee"). We will need to create an account with some APT to pay that fee and own the published contract. In order to do that, we will need to use the Aptos CLI.

0. Install the Aptos CLI

Install the Aptos CLI (if you haven't already).

1. Open a new terminal

   Open a new terminal window or tab.

2. Verify the installation

   Run `aptos --version` to verify you have it installed.

   Terminal window

   ```
   aptos --version
   ```

   You should see a response like `aptos 4.6.1`.

3. Create a project folder

   Create a new folder for this tutorial by running:

   Terminal window

   ```
   mkdir my-first-module
   ```

4. Navigate to the project folder

   Run `cd my-first-module` to go into your new folder.

5. Initialize your account

   Run `aptos init` and press 'enter' for each step of setup to create a test account on `devnet`.

   You should see a success message like this:

   Terminal window

   ```
   ---Aptos CLI is now set up for account 0x9ec1cfa30b885a5c9d595f32f3381ec16d208734913b587be9e210f60be9f9ba as pro
   {  "Result": "Success"}
   ```

2. (Optional) Explore What You Just Did On-Chain

Section titled "2. (Optional) Explore What You Just Did On-Chain"

0. Copy your account address

   Copy the address from the command line for your new account.

   The address looks like this `0x9ec1cfa30b885a5c9d595f32f3381ec16d208734913b587be9e210f60be9f9ba` and you can find it in the line:

   Terminal window

   ```
   Aptos CLI is now set up for account 0x9ec1cfa30b885a5c9d595f32f3381ec16d208734913b587be9e210f60be9f9ba as profil
   ```

1. Open the Aptos Explorer

   Go to the Aptos Explorer.

   This is the primary way to quickly check what is happening on devnet, testnet, or mainnet. We will use it later on to view our deployed contracts.

2. Ensure you are on Devnet network.

   Look for "Devnet" in the top right corner, or switch networks by clicking the "Mainnet" dropdown and selecting Devnet

   Switching to Devnet network in Aptos Explorer

3. Search for your account

   Paste your newly created address into the search bar.

4. View the search results

   Wait for the results to appear, then click the top result.

5. Check the transaction

   You should see your newly created account and a transaction with the faucet function, funding it with devnet tokens.

   Viewing Account in Aptos Explorer

6. Verify your balance

   Click the "Coins" tab to see that you have 1 APT of the Aptos Coin. This will allow you to publish and interact with smart contracts on the aptos devnet.

3. Writing and Compiling Your First Module

Section titled "3. Writing and Compiling Your First Module"

Now that we have our environment set up and an account created, let's write and compile our first Move module. Unlike Ethereum where contracts exist independently, Move ties everything

(Note: The content for this page appears truncated in the extraction. For the full tutorial, visit the URL.)

Move On Aptos Compiler

(From https://aptos.dev/build/smart-contracts/compiler_v2)

Move On Aptos Compiler

The Move on Aptos compiler (codename 'compiler v2') translates Move source code into Move bytecode. It unifies the architectures of the Move compiler and the Move Prover, enabling faster innovation in the Move language. It also offers new tools for defining code optimizations which can be leveraged to generate more gas efficient code for Move programs.

The Move on Aptos compiler supports Move 2, the latest revision of the Move language. Head over to the release page in the book to learn more about the new features in Move 2. Starting at Aptos CLI v6.0.0, this language version and the Move on Aptos compiler are the default. Note that Move 2 is generally backwards compatible with Move 1.

Move on Aptos is in production and is now the default compiler, with Move 2 being the default language version.

If you run into issues, please use this link to create a github issue. If you are able to provide a small piece of Move code which reproduces the issue, debugging and fixing it will be easier for us.

Ensure to have installed the latest version of the Aptos CLI:

Terminal window

```
aptos update aptos # on supported OS
brew upgrade aptos # on MacOS
```

Move on Aptos compiler and Move 2 are now the default, requiring no changes to your usage. Examples:

Terminal window

```
aptos move compile
aptos move test
aptos move prove
```

Building with Objects

(From https://aptos.dev/build/smart-contracts/objects)

Building with Objects

In Move, Objects group resources together so they can be treated as a single entity on chain.

Objects have their own address and can own resources similar to an account. They are useful for representing more complicated data types on-chain as Objects can be used in entry functions directly, and can be transferred as complete packages instead of one resource at a time.

Here's an example of creating an Object and transferring it:

```
module my_addr::object_playground {
  use std::signer;
  use std::string::{Self, String};
  use aptos_framework::object::{Self, ObjectCore};
  struct MyStruct1 has key {
    message: String,
  }
  struct MyStruct2 has key {
    message: String,
  }
  entry fun create_and_transfer(caller: &signer, destination: address) {
    // Create object
    let caller_address = signer::address_of(caller);
    let constructor_ref = object::create_object(caller_address);
    let object_signer = object::generate_signer(&constructor_ref);

    // Set up the object by creating 2 resources in it
    move_to(&object_signer, MyStruct1 {
      message: string::utf8(b"hello")
    });
    move_to(&object_signer, MyStruct2 {
      message: string::utf8(b"world")
    });

    // Transfer to destination
    let object = object::object_from_constructor_ref<ObjectCore>(
      &constructor_ref
    );
    object::transfer(caller, object, destination);
  }
}
```

During construction, Objects can be configured to be transferrable and extensible.

For example, you could use an Object to represent a soulbound NFT by making it only transferrable once, and have it own resources for an image link and metadata. Objects can also own other Objects, so you could implement your own NFT collection Object by transferring several of the soulbound NFTs to it.

Randomness API

(From https://aptos.dev/build/smart-contracts/randomness)

Randomness API

How random numbers have been obtained, insecurely/awkwardly

Section titled "How random numbers have been obtained, insecurely/awkwardly"

Building a lottery system and pick a random winner from `n` participants is trivial, at least in the centralized world with a trusted server: the backend simply calls a random integer sampling function (`random.randint(0, n-1)` in python, or `Math.floor(Math.random() * n)` in JS).

Unfortunately, without an equivalent of `random.randint()` in Aptos Move, building a dApp version of it was actually much harder.

One may have written a contract where the random numbers are sampled insecurely (e.g., from the blockchain timestamp):

```
module module_owner::lottery {    // ...
    struct LotteryState {        players: vector<address>,        winner_idx: std::option::Option<u64>,    }
    fun load_lottery_state_mut(): &mut LotteryState {        // ...    }
    entry fun decide_winner() {        let lottery_state = load_lottery_state_mut();        let n = std::vector::ler
```

The implementation above is insecure in multiple ways:

- a malicious user may bias the result by picking the transaction submission time;
- a malicious validator can bias the result easily by selecting which block the `decide_winner` transaction goes to.

Other dApps may have chosen to use an external secure randomness source (e.g., drand), which is typically a complicated flow:

0. The participants agree on using a future randomness seed promised by the randomness source to determine the winner.
1. Once the randomness seed is revealed, the clients fetch it and derive the winner locally.
2. One of the participants submits the seed and the winner on chain.

```
module module_owner::lottery {    // ...
    struct LotteryState {        players: vector<address>,        /// public info about the "future randomness", tyi
    fun load_lottery_state_mut(): &mut LotteryState {        // ...    }
    fun is_valid_seed(seed_verifier: vector<u8>, seed: vector<u8>): bool {        // ...    }
    fun derive_winner(n: u64, seed: vector<u8>): u64 {        // ...    }
    entry fun update_winner(winner_idx: u64, seed: vector<u8>) {        let lottery_state = load_lottery_state_mut()
```

Achieve simplicity + security with Aptos randomness API

Section titled “Achieve simplicity + security with Aptos randomness API”

Using Aptos randomness API, the implementation will look like this:

```
module module_owner::lottery {    // ...
    struct LotteryState {        players: vector<address>,        winner_idx: std::option::Option<u64>,    }
    fun load_lottery_state_mut(): &mut Lottery {        // ...    }
    #[randomness]    entry fun decide_winner() {        let lottery_state = load_lottery_state_mut();        let n =
```

where:

- `let winner_idx = aptos_framework::randomness::u64_range(0, n);` is the randomness API call that returns an u64 integer in range `[0, n)` uniformly at random.
- `#[randomness]` is a required attribute to enable the API call at runtime.

Ensure you

(Note: The content for this page appears truncated in the extraction. For the full details, visit the URL.)

Aptos Standards

(From https://aptos.dev/build/smart-contracts/aptos-standards)

Aptos Standards

Standards define a common interoperable interface for all developers to build upon. They consist of rules to ensure compatibility across applications and wallets on the Aptos blockchain. See a list of known coin resource addresses on Aptos provided by hippospace.

The Object model allows Move to represent a complex type as a set of resources stored within a single address and offers a rich capability model that allows for fine-grained resource control and ownership management.

The new Aptos Digital Asset Standard allows:

- Rich, flexible assets and collectibles.

- Easy enhancement of base functionality to provide richer custom functionalities. An example of this is the aptos_token module

Digital Asset (DA) is recommended for any new collections or protocols that want to build NFT or semi-fungible tokens.

The new Aptos Fungible Asset Standard is a standard meant for simple, type-safe, and fungible assets based on object model intending to replace Aptos coin. Fungible Asset (FA) offers more features and flexibility to Aptos move developers on creating and managing fungible assets.

The old existing Token module, on the other hand:

- Encapsulates rich, flexible assets and collectibles. These assets are discrete (non-decimal) and can be fungible, semi-fungible, or non-fungible.
- The token standard is in its own `AptosToken` package at the Address `0x3` to allow for rapid iteration based on feedback from the community.

The Coin module is a lightweight standard meant for simple, type-safe, and fungible assets. The coin standard is separated out into its own Move module to ensure that:

- Applications and users can create and use simple tokens, with high performance and low gas overhead.
- The Coin standard is part of the Aptos core framework, so it can be used for currencies, including the gas currency.

Aptos Fungible Asset (FA) Standard

(From https://aptos.dev/build/smart-contracts/fungible-asset)

Aptos Fungible Asset (FA) Standard

The Aptos Fungible Asset Standard (also known as "Fungible Asset" or "FA") provides a standard, type-safe way to define fungible assets in the Move ecosystem. It is a modern replacement for the `coin` module that allows for seamless minting, transfer, and customization of fungible assets for any use case.

This standard is important because it allows fungible assets on Aptos (such as Currencies and Real World Assets (RWAs)) to represent and transfer ownership in a consistent way dApps can recognize. This standard also allows for more extensive customization than the `coin` module did by leveraging Move Objects to represent fungible asset data.

The FA standard provides all the functionality you need to create, mint, transfer, and burn fungible assets (as well as automatically allowing recipients of the fungible asset to store and manage any fungible assets they receive).

It does so by using two Move Objects:

1. `Object<Metadata>` - This represents details about the fungible asset itself, including information such as the `name`, `symbol`, and `decimals`.
2. `Object<FungibleStore>` - This stores a count of fungible asset units owned by this account. Fungible assets are interchangeable with any other fungible asset that has the same metadata. An account may own more than one `FungibleStore` for a single Fungible Asset, but that is only for advanced use cases.

The diagram below shows the relationship between these Objects. The `Metadata` Object is owned by the Fungible Asset creator, then referenced in FA holders' `FungibleStore`s to indicate which FA is being tracked:

FA Object Relationship

FA Object Relationship

This implementation is an improvement on the `coin` Standard because Move Objects are more customizable and extensible via smart contract. See the advanced guides on writing Move Objects for more details. The FA standard also automatically handles tracking how much of a fungible asset an account owns, as opposed to requiring the recipient to register a `CoinStore` resource separate from the transfer.

At a high level, this is done by:

1. Creating a non-deletable Object to own the newly created Fungible Asset `Metadata`.
2. Generating `Ref`s to enable any desired permissions.
3. Minting Fungible Assets and transferring them to any account you want to.

To start with, the Fungible Asset standard is implemented using Move Objects. Objects by default are transferable, can own multiple resources, and can be customized via smart contract. For full details on Objects and how they work, please read this guide.

To create an FA, first you need to create a non-deletable Object since destroying the metadata for a Fungible Asset while there are active balances would not make sense. You can do that by either calling `object::create_named_object(caller_address, NAME)` or `object::create_sticky_object(caller_address)` to create the Object on-chain.

When you call these functions, they will return a `ConstructorRef`. `Ref`s allow Objects to be customized immediately after they are created. You can use the `ConstructorRef` to generate other permissions that may be needed based on your use case.

One use for the `ConstructorRef` is to generate the FA `Metadata` Object. The standard provides a generator function called `primary_fungible_store::create_primary_store_enabled_fungible_asset` which will allow your fungible asset to be transferred to any account. This method makes it so the primary `FungibleStore` for recipients is automatically created or re-used so you don't need to create or index the store directly.

This is what `create_primary_store_enabled_fungible_asset` looks like:

```
public fun create_primary_store_enabled_fungible_asset(
    constructor_ref: &ConstructorRef,
    // This ensures total supply does not surpass this limit - however,
    // Setting this will prevent any parallel execution of mint and burn.
    maximum_supply: Option<u128>,
    // The fields below here are purely metadata and have no impact on-chain.
    name: String,
    symbol: String,
    decimals: u8,
    icon_uri: String,
    project_uri: String,
)
```

Once you have created the Metadata, you can also use the `ConstructorRef` to generate additional `Ref`

(Note: The content for this page appears truncated in the extraction. For the full details, visit the URL.)

Move Security Guidelines

(From https://aptos.dev/build/smart-contracts/move-security-guidelines)

Move Security Guidelines

The Move language is designed with security and inherently offers several features including a type system and a linear logic. Despite this, its novelty and the intricacies of some business logic mean that developers might not always be familiar with Move's secure coding patterns, potentially leading to bugs.

This guide addresses this gap by detailing common anti-patterns and their secure alternatives. It provides practical examples to illustrate how security issues can arise and recommends best practices for secure coding. This guide aims to sharpen developers' understanding of Move's security mechanisms and ensure the robust development of smart contracts.

Object Ownership Verification

Every `Object<T>` can be accessed by anyone, which means any `Object<T>` can be passed to any function, even if the caller doesn't own it. It's important to verify that the `signer` is the rightful owner of the object.

In this module, a user must purchase a subscription before performing certain actions. The user invokes the registration function to acquire an `Object<Subscription>`, which they can later use to execute operations.

```
module 0x42::example {
    struct Subscription has key {
        end_subscription: u64
    }
    entry fun registration(user: &signer, end_subscription: u64) {
        let price = calculate_subscription_price(end_subscription);
        payment(user,price);
        let user_address = address_of(user);
        let constructor_ref = object::create_object(user_address);
        let subscription_signer = object::generate_signer(&constructor_ref);
        move_to(&subscription_signer, Subscription { end_subscription });
    }
    entry fun execute_action_with_valid_subscription(
```

```
        user: &signer, obj: Object<Subscription>
    ) acquires Subscription {
        let object_address = object::object_address(&obj);
        let subscription = borrow_global<Subscription>(object_address);
        assert!(subscription.end_subscription >= aptos_framework::timestamp::now_seconds(),1);
        // Use the subscription
        [...]
    }
}
```

In this insecure example, `execute_action_with_valid_subscription` does not verify if the user owns the `obj` passed to it. Consequently, anyone can use another person's subscription, bypassing the payment requirement.

Ensure that the signer owns the object.

```
module 0x42::example {
    struct Subscription has key {
        end_subscription: u64
    }
    entry fun registration(user: &signer, end_subscription: u64) {
        let price = calculate_subscription_price(end_subscription);
        payment(user,price);
        let user_address = address_of(user);
        let constructor_ref = object::create_object(user_address);
        let subscription_signer = object::generate_signer(&constructor_ref);
        move_to(&subscription_signer, Subscription { end_subscription });
    }
    entry fun execute_action_with_valid_subscription(
        user: &signer, obj: Object<Subscription>
    ) acquires Subscription {
        //ensure that the signer owns the object.
        assert!(object::owner(&obj)==address_of(user),ENOT_OWNWER);
        let object_address = object::object_address(&obj);
        let subscription = borrow_global<Subscription>(object_address);
        assert!(subscription.end_subscription >= aptos_framework::timestamp::now_seconds(),1);
        // Use the subscription
        [...]
    }
}
```

Signer Verification

Accepting a `&signer` is not always sufficient for access control purposes. Be sure to assert that the signer is the expected account, especially when performing sensitive operations.

Users without proper authorization can execute privileged actions.

This code snippet allows any user invoking the `delete` function to remove an `Object`, without verifying that the caller has the necessary permissions.

```
module 0x42::example {
    struct Object has key{
        data: vector<u8>
    }
    public fun delete(user: &signer, obj: Object) {
        let Object { data } = obj;
    }
}
```

A better alternative is to use the global storage provided by Move, by directly borrowing data off of `signer::address_of(signer)`. This approach ensures robust access control, as it exclusively accesses data contained within the address of the signer of the transaction. This method minimizes the risk of access control errors, ensuring that only the data owned by the `signer` can be manipulated.

```
module 0x42::example {
    struct Object has key{
```

```
        data: vector<u8>
    }
    public fun delete(user: &signer) {
        let Object { data } = move_from<Object>(signer::address_of(user));
    }
}
```

Principle of Least Privilege

Adhere to the principle of least privilege:

- Always start with private functions, change their visibility as it

(Note: The content for this page appears truncated in the extraction. For the full guidelines, visit the URL.)

Move Prover Overview

(From https://aptos.dev/build/smart-contracts/prover)

Move Prover Overview

The Move Prover supports formal specification and verification of Move code. The Move Prover can automatically validate logical properties of Move smart contracts while offering a user experience similar to a type checker or linter.

The Move Prover exists to make contracts more trustworthy; it:

- Protects massive assets managed by the Aptos blockchain from smart contract bugs
- Protects against well-resourced adversaries
- Anticipates justified regulator scrutiny and compliance requirements
- Allows domain experts with a mathematical background, but not necessarily a software engineering background, to understand what smart contracts do

For more information, refer to the documentation