Web3 Unleashed: Upgrading Smart Contracts - Should You Do it and How?¶

Written byEmily Lin

Last updated 8/19/2022

Overview¶

In this episode of Web3 Unleashed, we'll be going over smart contract upgrades: what they are, the security implications of doing so, and how to do it!

Watch our livestream recording with security solutions architectMichael Lewellen fromOpenZeppelin onYouTube for a more in-depth explanation and interview! There, we'll dive further in detail about the different types of upgrade patterns as well as tips and tricks should you decide to write an upgradeable smart contract!

What is a smart contract upgrade?¶

By default, smart contracts are immutable, which is necessary to the trustlessness, decentralization, and security of Ethereum. However, what happens when you discover a smart contract vulnerability? Or what if you want to add in new features and capabilities? Smart contract upgrades are essentially different strategies one might take to change the functionality of a contract. Note that the initial contract must be deployed in an upgradeable way if you want to change contract code. This is also NOT the same as being able to change the internal code. Instead, upgradeability means you are changing the code that gets executed. Extensive research has been done to discover various patterns for writing upgradeable smart contracts while trying to minimize centralization and the inherent security risks. OpenZeppelin has a great articlehere that goes over them.

Should you upgrade smart contracts?¶

What are the tradeoffs?¶

Before diving into even how we upgrade, we should first consider whether or not we should do it in the first place. The pros for upgradeable smart contracts largely falls into two categories:

1. Discovered vulnerabilities post-deployment are easier and faster to fix.
2. Developers can improve their dapps by experimenting with and adding new features over time.

While this sounds great, violating immutability affects trustlessness, security, and decentralization in the following ways:

1. Because developers can change the code, users must trust the developers to not do so maliciously or arbitrarily.
2. Writing upgradeable smart contracts is inherently difficult and complex. As a result, developers may introduce more flaws than would have existed otherwise.
3. If the ability to upgrade the contract is insecure or centralized, it is easy for attackers to make malicious upgrades.

Lastly, depending on how you decide to upgrade your contracts, you could potentially incur high gas costs.

How do you decide to upgrade?¶

After taking into consideration the implications of smart contract upgrades, the next step is to actually make the decision of whether or not to go through it. It is critical this decision does not fall in the hands of a single account. A single account not only overturns decentralization, but compromised keys will have disastrous consequences for the security of the contract. There are a few popular ways of enacting the upgrade:

1. Multi-sig
2. contracts allow for there to be multiple owners, with decisions being made once a certain threshold of stakeholders agree
3. Timelock
4. refers to a time delay for when the change actually goes into effect. This gives users time to exit if they disagree with the change. However there are two issues that arise from timelocks:* The delay can be a major blocker to a quick response to a critical bug.
5.
    - This can be mitigated bypausing
6.
    - andescape hatches
7.
    - . In this case, trusted developers are allowed to pause operations as soon as an issue is detected, such as stopping token transfers, to prevent more harm. Meanwhile, users can exit the system, such as extracting out their funds, using an escape hatch that was coded into the smart contract.
8.
    - Publishing a timelocked upgrade to be added later allows attackers to reverse-engineer the change and potentially exploit the bug before the change goes into effect. In this case,commit-reveal
9.
    - strategies are used by announcing an upgrade, but not revealing it until the delay expires.
10. Voting
11. decentralizes the decision making further by granting your community the right to vote on changes, usually done through some governance token. Note that this is often used in conjunction with the other strategies listed above.

How do you upgrade a smart contract?¶

As mentioned before, there are a number of technically complex ugprade patterns laid outhere .

At the core of it, upgrade patterns rely on aproxy contract and animplementation contract (akalogic contract). The proxy contract knows the contract address of the implementation contract and delegates all calls it receives to it. This means that:

1. Execution of the implementation contract code is happening within the context of the proxy contract.
2. Reads or writes to storage only affect the storage of the proxy contract, not the implementation contract.
3. msg.sender
4. is the address of whoever called the proxy contract

This is all possible because of the opcodeDELEGATECALL , which basically allows a contract to execute code from another contract as if it were an internal function. As a result, upgrading is actually relatively straightforward - you just change out the implementation address. The real complexity comes in when considering the actual upgrade logic.

We won't dive into it all the variations, but theOpenZeppelin Upgrades plugin uses thetransparent proxy pattern pattern, which you can read more abouthere andhere .

Now, let's actually go walk through an example! The completed code ishere .

Download System Requirements¶

You'll need to install:

- Node.js
- , v12 or higher
- truffle
- ganache UI
- organache CLI

Create an Infura account and project¶

To connect your DApp to Ethereum mainnet and testnets, you'll need an Infura account. Sign up for an accounthere .

Once you're signed in, create a project! Let's call itupgrade-contract , and select Web3 API from the dropdown

Register for a MetaMask wallet¶

To interact with your DApp in the browser, you'll need a MetaMask wallet. Sign up for an accounthere .

Download VS Code¶

Feel free to use whatever IDE you want, but we highly recommend using VS Code! You can run through most of this tutorial using the Truffle extension to create, build, and deploy your smart contracts, all without using the CLI! You can read more about ithere .

Get Some Test Eth¶

In order to deploy to the public testnets, you'll need some test Eth to cover your gas fees has a great MultiFaucet that deposits funds across 8 different networks all at once. If you're looking specifically for goerli eth, trythis one orthis one .

Set Up Your Project¶

Truffle has some nifty functions to scaffold your truffle project and add example contracts and tests. We'll be building our project in a folder calledupgrade-contract .

truffle init upgrade-contractcd

upgrade-contract truffle create contract UpgradeablePet truffle create test

UpgradeablePet Afterwards, your project structure should look something like this:

upgrade-contract ├── contracts | └── UpgradeablePet.sol ├── migrations | └── 1_deploy_contracts.js ├── test | └── upgradeable_pet.js └── truffle-config.js

Write an Upgradeable Contract V1¶

Let's start by writing our base contract that we'll be progressively upgrading!

First off, our contract needs to be upgrade safe. This means that the contract:

1. cannot have a constructor
2. should not use theselfdestruct
3. ordelegatecall
4. operations

You can read more about whyhere .

Our first iteration ofUpgradeablePet is gonna be super simple - all it will do is store a value and get that value. It should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; contract

UpgradeablePet

{

uint256

private

_value ;

// Emitted when the stored value changes

event

ValueChanged (uint256

value);

// Stores a new value in the contract

function

store (uint256

value)

public

{

_value

=

value ;

emit

ValueChanged (value);

}

// Reads the last stored value

function

retrieve ()

public

view

returns

(uint256)

{

return

_value ;

} } Let's say we actually only want the pet owner to be able to change the contents ofUpgradeablePet . How do we pass in the appropriate address if we can't have a constructor? OpenZeppelin has provided a base contract calledInitializer , which will help us run the necessary initalization code. First, we will need to download it as follows:

npm i @openzeppelin/contracts-upgradeable And then, we can modifyUpgradeablePet :

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; import

"@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol" ; contract

UpgradeablePet

is

Initializable

{

uint256

private

_value ;

address

private

_petOwner ;

// Emitted when the stored value changes

event

ValueChanged (uint256

value);

function

initialize (address

petOwner)

public

initializer

{

_petOwner

=

petOwner ;

}

/// @custom:oz-upgrades-unsafe-allow constructor

constructor ()

{

_disableInitializers ();

}

// Stores a new value in the contract

function

store (uint256

value)

public

{

require (msg . sender

==

_petOwner ,

"UpgradeablePet: not owner");

_value

=

value ;

emit

ValueChanged (value);

}

// Reads the last stored value

function

retrieve ()

public

view

returns

(uint256)

{

return

_value ;

} } Two things to note:

1. If there are any parent contracts,initialize
2. will have to manually call theinitalize
3. functions of those parent contracts well.
4. You'll notice we actually did leave in a constructor in addition toinitialize
5. . This ensures the contract is in an initialized state. Otherwise, an uninitialized implementation contract can be taken over by an attacker.

Now, we need to modify1_deploy_contracts.js to tell Truffle how to deploy this file. We'll first need to download the plugin:

npm i --save-dev @openzeppelin/truffle-upgrades Then, modify your migration file as follows:

const

{

deployProxy

}

=

require ('@openzeppelin/truffle-upgrades'); const

UpgradeablePet

=

artifacts . require ('UpgradeablePet'); module . exports

=

async

function

(deployer ,

network ,

accounts)

{

await

deployProxy (UpgradeablePet ,

[accounts [0]],

{

deployer ,

initializer :

'initialize'

}); }; In order to test this, we'll just do this on the fly. You can either calltruffle develop , which will bring up a ganache instance on 9545, or open up your own ganache instance, modifydevelopment intruffle-config.js , and runtruffle console . For this guide, we recommend opening up a separate ganache instance so that the contract addresses are preserved.

truffle(develop)

migrate

Compiling your contracts...==========

Everything is up to date, there is nothing to compile.

Starting migrations...==========

Network name: 'development'

Network id: 1660859525632

Block gas limit: 30000000

(0x1c9c380) 1_deploy_contracts.js==========

Replacing 'UpgradeablePet'

---

transaction hash: 0xb42a280a989a089efb526930b1da5f80cd41a487f4b0facd9d6ccc376e273f56

Blocks: 0

Seconds: 0

contract address: 0xc094d30a290db2C0781fF97874D35A6dF8c0F225

block number: 11

block timestamp: 1660863108

account: 0xA8469E3bF6474abb1290a4c03F43021667df130e

balance: 999 .985882023380156735

gas used: 410834

(0x644d2)

gas price: 2 .739722993 gwei

value sent: 0

ETH

total cost: 0 .001125571356106162 ETH Deploying 'ProxyAdmin'

---

transaction hash: 0xac44c24fa0ca5e3118e1c027474e55584c8ec13e48e797e315d6812d5ccc94b6

Blocks: 0

Seconds: 0

contract address: 0x749D40F055727817e9E9D56e5247722407ccae17

block number: 12

block timestamp: 1660863108

account: 0xA8469E3bF6474abb1290a4c03F43021667df130e

balance: 999 .984570049252513955

gas used: 484020

(0x762b4)

gas price: 2 .710578339 gwei

value sent: 0

ETH

total cost: 0 .00131197412764278 ETH Deploying 'TransparentUpgradeableProxy'

---

transaction hash: 0xd82a7bcec734f0e69d614016d3408a64ef564082d328215e4f79e8669934f9c7

Blocks: 0

Seconds: 0

contract address: 0xb85a509102B82f02281b0451C43FA37e00d625ad

block number: 13

block timestamp: 1660863108

account: 0xA8469E3bF6474abb1290a4c03F43021667df130e

balance: 999 .982844095713016935

gas used: 642788

(0x9cee4)

gas price: 2 .685105415 gwei

value sent: 0

ETH

total cost: 0 .00172595353949702 ETH

Saving artifacts

Total cost: 0 .004163499023245962 ETHSummary =======

Total deployments: 3

Final cost: 0 .004163499023245962 ETH As you can see,deployProxy does three things:

1. Deploy the implementation contract (our Box contract)
2. Deploy the ProxyAdmin contract (the admin for our proxy).
3. Deploy the proxy contract and run any initializer function.

Now, we can just call contract functions directly from the console to quickly see if our contract is working.

truffle(development)

let

contract

=

await UpgradeablePet.deployed() ; undefined truffle(development)

await contract.store(5) {

tx: '0xeb7971ae96003a2be24ed38e7d62ab8741f5a8f772d5155679f41929d2808a6f' , receipt: {

transactionHash: '0xeb7971ae96003a2be24ed38e7d62ab8741f5a8f772d5155679f41929d2808a6f' , transactionIndex: 0 , blockNumber: 14 , blockHash: '0x5ebca4e4c9d5bed1300e6fe9399144f47f8f751ce2bd4a655b160242cb540e44' , from: '0xa8469e3bf6474abb1290a4c03f43021667df130e' , to: '0xb85a509102b82f02281b0451c43fa37e00d625ad' , cumulativeGasUsed: 54413 , gasUsed: 54413 , contractAddress: null, logs: [

[Object]

] , logsBloom:
'0x4000000000000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
, status: true, effectiveGasPrice: 2662958768 , type: '0x2' , rawLogs: [

[Object]

]

} , logs: [

{

address: '0xb85a509102B82f02281b0451C43FA37e00d625ad' , blockHash: '0x5ebca4e4c9d5bed1300e6fe9399144f47f8f751ce2bd4a655b160242cb540e44' , blockNumber: 14 , logIndex: 0 , removed: false, transactionHash: '0xeb7971ae96003a2be24ed38e7d62ab8741f5a8f772d5155679f41929d2808a6f' , transactionIndex: 0 , id: 'log_c488ac08' , event: 'ValueChanged' , args: [Result]

}

] } truffle(development)

contract.retrieve() BN {

negative: 0 , words: [

5 , <1

empty item>] , length: 1 , red: null } Nice! Before moving forward, let's write a test! When writing tests for upgrades we'll need to write tests for both the implementation contract AND the proxy contract. Luckily, we can use OpenZeppelin'sdeployProxy in our tests.

Let's first install OpenZeppelin's test helpers to make testing a little easier.

npm i --save-dev @openzeppelin/test-helpers Writing the test for the implementation contract is the same as usual. Let's create a file calledupgradeable_pets.js under thetest folder and add this code:

const

{

expectRevert ,

expectEvent

}

=

```
require ( '@openzeppelin/test-helpers' ); const
UpgradeablePet
=
artifacts . require ( "UpgradeablePet" ); contract ( "UpgradeablePet" ,
function
( accounts )
{
it ( "should retrieve correctly stored value" ,
async
function
()
{
const
upgradeablePetInstance
=
await
UpgradeablePet . deployed ();
let
tx
=
await
upgradeablePetInstance . store ( 5 );
expectEvent ( tx ,
"ValueChanged" ,
{
value :
"5"
});
let
value
=
await
upgradeablePetInstance . retrieve ();
assert . equal ( value ,
5 ,
"UpgradeablePet did not store correct value" );
});
it ( "should not set the stored value if not owner" ,
async
function
()
{
const
upgradeablePetInstance
=
await
UpgradeablePet . deployed ();
// Failed require in function
await
expectRevert ( upgradeablePetInstance . store ( 10 ,
{ from :
accounts [ 1 ]}),
"UpgradeablePet: not owner" );
let
value
=
await
upgradeablePetInstance . retrieve ();
assert . equal ( value ,
5 ,
```

"UpgradeablePet value should not have changed");

}); }); Then, create a test file calledupgradeable_pets.proxy.js , and add the following:

```
const
{
expectRevert ,
expectEvent
}
=
require ( '@openzeppelin/test-helpers' ); const
{
deployProxy
}
=
require ( '@openzeppelin/truffle-upgrades' ); const
UpgradeablePet
=
artifacts . require ( "UpgradeablePet" ); contract ( "UpgradeablePet (Proxy)" ,
function
( accounts )
{
it ( "should retrieve correctly stored value" ,
async
function
()
{
const
upgradeablePetInstance
=
await
deployProxy ( UpgradeablePet ,
[ accounts [ 0 ]],
{
initializer :
'initialize'
});
let
tx
=
await
upgradeablePetInstance . store ( 5 );
expectEvent ( tx ,
"ValueChanged" ,
{
value :
"5"
});
let
value
=
await
upgradeablePetInstance . retrieve ();
assert . equal ( value ,
5 ,
"UpgradeablePet did not store correct value" );
});
it ( "should not set the stored value if not owner" ,
async
function
()
{
```

const

upgradeablePetInstance

=

await

deployProxy (UpgradeablePet ,

[accounts [0]],

{

initializer :

'initialize'

});

// Failed require in function

await

expectRevert (upgradeablePetInstance . store (10 ,

{ from :

accounts [1]}),

"UpgradeablePet: not owner");

let

value

=

await

upgradeablePetInstance . retrieve ();

assert . equal (value ,

0 ,

"UpgradeablePet value should not have changed");

}); }); You'll notice instead of usingUpgradeablePet.deployed() , we usedeployProxy to get our contract instance.

To test, simply calltruffle test to make sure everything is working properly.

Write the Upgradeable Contract V2¶

Now, let's get to the exciting part: actually adding a change! We will first create a new duplicate contract and then add anincrement function to increment the stored value.

Create a new contractUpgradeablePetV2 with anincrement function. It should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; import

"@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol" ; contract

UpgradeablePetV2

is

Initializable

{

uint256

private

_value ;

address

private

_petOwner ;

// Emitted when the stored value changes

event

ValueChanged (uint256

value);

function

initialize (address

petOwner)

public

initializer

{

_petOwner

=

petOwner ;

}

/// @custom:oz-upgrades-unsafe-allow constructor

constructor ()

initializer

{}

// Stores a new value in the contract

function

store (uint256

value)

public

{

require (msg . sender

==

_petOwner ,

"UpgradeablePet: not owner");

_value

=

value ;

emit

ValueChanged (value);

}

// Reads the last stored value

function

retrieve ()

public

view

returns

(uint256)

{

return

_value ;

}

// Increments the stored value by 1

function

increment ()

public

{

_value

=

_value

+

1 ;

emit

ValueChanged (_value);

} } Now, we'll be using OpenZeppelin'supgradeProxy function, which will:

1. Deploy the implementation contract (UpgradeablePetV2
2.)
3. Call theProxyAdmin
4. to update the proxy contract to use the new implementation.

We will use this in our new deployment script:

const

{

upgradeProxy

}

=

require ('@openzeppelin/truffle-upgrades'); const

UpgradeablePet

=

artifacts . require ('UpgradeablePet'); const

UpgradeablePetV2

=

artifacts . require ('UpgradeablePetV2'); module . exports

=

async

function

(deployer)

{

const

alreadyDeployed

=

await

UpgradeablePet . deployed ();

await

upgradeProxy (alreadyDeployed . address ,

UpgradeablePetV2 ,

{

deployer

}); }; Then, callmigrate from the console, and test out increment:

truffle(development)

    contract

=

await UpgradeablePet.deployed() undefined truffle(development)

    contract.address'0xAe02BB114AAD3Edf8b87827Cf001F3D49165b426' truffle(development)

    let

contractv2

=

await UpgradeablePetV2.at(contract.address) undefined truffle(development)

    contractv2.retrieve() BN {

negative: 0 , words: [

5 , <1

empty item>] , length: 1 , red: null } truffle(development)

    contractv2.increment() {

tx: '0x76820ff204a1ba364ee3deed7e62371f0803eb0b661fd5d88d845abb3f972fbc' , receipt: {

transactionHash: '0x76820ff204a1ba364ee3deed7e62371f0803eb0b661fd5d88d845abb3f972fbc' , transactionIndex: 0 , blockNumber: 36 , blockHash: '0xe75e2d55a8d310f6686c61f5f8ecf75e05b8be96b985ac31f515a90eafe058d2' , from: '0xa8469e3bf6474abb1290a4c03f43021667df130e' , to: '0xae02bb114aad3edf8b87827cf001f3d49165b426' , cumulativeGasUsed: 35076 , gasUsed: 35076 , contractAddress: null, logs: [

[Object]

] , logsBloom:
'0x400000000000000000000000000000000000000000000000000000000000000000000000000040000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
, status: true, effectiveGasPrice: 2509015503 , type: '0x2' , rawLogs: [

[Object]

]

} , logs: [

{

address: '0xAe02BB114AAD3Edf8b87827Cf001F3D49165b426' , blockHash: '0xe75e2d55a8d310f6686c61f5f8ecf75e05b8be96b985ac31f515a90eafe058d2' , blockNumber: 36 , logIndex: 0 , removed: false, transactionHash: '0x76820ff204a1ba364ee3deed7e62371f0803eb0b661fd5d88d845abb3f972fbc' , transactionIndex: 0 , id: 'log_b88893d4' , event: 'ValueChanged' , args: [Result]

}

] } truffle(development)

    contractv2.retrieve() BN {

negative: 0 , words: [

6 , <1

empty item>] , length: 1 , red: null } Notice that we usedlet contractv2 = await UpgradeablePetV2.at(contract.address) ..at is a special Truffle function that allows you to create a new abstraction at the same address.

Finally, let's write the tests. Again, we'll need to test both the implementation contract and the proxy contract.

Create a fileupgradeable_pet_V2.js and add:

const

{

expectEvent

}

=

require ('@openzeppelin/test-helpers'); const

UpgradeablePetV2

=

artifacts . require ("UpgradeablePetV2"); contract ("UpgradeablePetV2" ,

function

(accounts)

{

```
it ( "should increment the stored value" ,
async
function
()
{
const
upgradeablePetV2Instance
=
await
UpgradeablePetV2 . deployed ();
let
tx
=
await
upgradeablePetV2Instance . store ( 5 );
expectEvent ( tx ,
"ValueChanged" ,
{
value :
"5"
});
let
value
=
await
upgradeablePetV2Instance . retrieve ();
assert . equal ( value ,
5 ,
"UpgradeablePetV2 did not store correct value" );
await
upgradeablePetV2Instance . increment ();
value
=
await
upgradeablePetV2Instance . retrieve ();
assert . equal ( value ,
6 ,
"UpgradeablePetV2 did not increment" );
}); });
```

LikedeployProxy , we can also useupgradeProxy in our tests. Create a new test calledupgradeable_pet_V2.proxy.js :

```
const
{
deployProxy ,
upgradeProxy
}
=
require ( '@openzeppelin/truffle-upgrades' ); const
UpgradeablePet
=
artifacts . require ( "UpgradeablePet" ); const
UpgradeablePetV2
=
artifacts . require ( "UpgradeablePetV2" ); contract ( "UpgradeablePetV2 (Proxy)" ,
function
( accounts )
{
it ( "should increment the stored value" ,
async
function
()
{
```

const

upgradeablePetInstance

=

await

deployProxy (UpgradeablePet ,

[accounts [0]],

{

initializer :

'initialize'

});

await

upgradeablePetInstance . store (5);

let

value

=

await

upgradeablePetInstance . retrieve ();

assert . equal (value ,

5 ,

"UpgradeablePet did not store correct value");

const

upgradeablePetV2Instance

=

await

upgradeProxy (upgradeablePetInstance . address ,

UpgradeablePetV2);

value

=

await

upgradeablePetV2Instance . retrieve ();

assert . equal (value ,

5 ,

"UpgradeablePetV2 did not store correct value");

await

upgradeablePetV2Instance . increment ();

value

=

await

upgradeablePetV2Instance . retrieve ();

assert . equal (value ,

6 ,

"UpgradeablePetV2 did not increment");

}); }); The point we want to test here is that state was preserved between V1 and V2 of the smart contract.

Future Extensions¶

And there you have it! You've upgraded a smart contract! Again, be sure to watch the livestream onYouTube , and see what's upcoming on ourGitHub page . OpenZeppelin has also written their own blog post that goes much more in depth and includes real-world exampleshere . If you're interested in making the previous episode's contracts upgradable, you'll need to use the ugpradable versions of their base contracts, which can be installed vianpm i @open-zeppelin/upgradeable-contract , which will useinitialize instead ofconstructor .

If you want to talk about this content, make suggestions for what you'd like to see or ask questions about the series, start a discussionhere . If you want to show off what you built or just hang with the Unleashed community in general, join ourDiscord ! Lastly, don't forget to follow us onTwitter for the latest updates on all things Truffle.