Towards a Solider Solidity

João Reis, Mário Pereira and António Ravara
NOVA LINCS and NOVA SCT

17 July 2023

1 Introduction

Blockchain technology has emerged as a groundbreaking innovation in recent years. Ethereum, a prominent blockchain platform, has gained widespread adoption due to its ability to execute smart contracts [3]. These self-executing agreements enable transparent and autonomous interactions between parties. Solidity, Ethereum's primary programming language for smart contracts, empowers developers to create complex decentralized applications. However, Solidity's safety has been a concern, leading to potential vulnerabilities in smart contract execution, being one of those the weak typing of address type [1].

Inspired by the need for a safer and more robust variant of Solidity, we re-visit Featherweight Solidity (FS), a formalization and implementation of a type-safe subset of the Solidity programming language. This formalization is inspired on the previous work of Silvia Crafa and Matteo Di Pirro [4, 5], with some modifications. Our implementation supports Solidity's multi inheritance, using C3 linearization [2], which the original formalization does not. We keep their most proeminent modification, implementing a new type system, introducing an extension to the type address, being able to be more restrictive and identifying addresses referencing different contracts or externally owned accounts. We also leveraged the expressiveness power of OCaml, a functional programming language known for its strong type inference and formal verification capabilities, to implement Featherweight Solidity as a proof-of-concept compiler.[1]

In this paper, we will present this vulnerability we are addressing. Our presentation will be based on illustrative examples, showing code with errors and the correction that we propose. We will also show how our tools works with those examples.

In the next section we will show in detail how type casts exists in Solidity and how dangerous they may be. In the third section we will introduce a new contract, using our new subtype for addresses, running with our tools. In the end we present a brief conclusion.

[1] https://github.com/jcrreis/featherweight-solidity

2 Type Casts in Solidity

The Solidity compiler can detect some typical data type errors errors (e.g., assigning an integer value to a variable of type string), but is too permissive when it comes to the `address` type. A contract written in the Solidity language can call another contract by directly referencing the callee contract's instance. However, when a contracts calls another contracts function, it only checks if the interface matches, not checking if the contract passed as an argument is from the same type as it is declared in a function. As a result, a developer should be careful whenever a public function in a contract calls another contract interface. The solidity code in Figure 1 shows an example of this vulnerability.

```
1  interface Counter {
2      function add(uint num) external returns (uint);
3  }
4  contract FakeCounter is Counter{
5      uint public counter;
6      function add(uint num) public returns (uint){
7          counter += 0;
8          return counter;
9      }
10 }
11 contract CounterLibrary is Counter{
12     uint public counter;
13     function add(uint num) public returns (uint){
14         counter += num;
15         return counter;
16     }
17 }
18 contract Game {
19     function play(CounterLibrary c) public returns (uint){
20         return c.add(1);
21     }
22     function getCounter(CounterLibrary c) public view returns (uint
           ){
23         return c.counter();
24     }
25 }
```

Figure 1: An example of Type Casts vulnerability.

After deploying these contracts into the blockchain, one can call the `play` function from the `Game` contract, which receives a `CounterLibrary` type contract. Nevertheless, one can either call this function with `CounterLibrary` contract address or `FakeCounter` contract address, even if only the first contract match the type declared, both match the interface required by play function. They will have different outputs: `CounterLibrary` will return a true counter (lines 16 and 17), but `FakeCounter` will always return 0 (lines 8 and 9).

In the current state of the art of Solidity, there are still no ways to prevent this vulnerability.

3 Featherweight Solidity in detail

To explain our solution, we present a smart contract that allows multiple NFT stores, where each NFT follows the ERC-721 specification. Figure 2 shows the contract and its function signatures.

```
import "./NFTStorage.sol";
contract Game is Context{
    mapping(address => NFTStorage) stores;
    constructor() Context(){

    }
    function fb() {

    }
    function createStore() returns (address) {
        ...
    }
    function addExternalStore(address<NFTStorage> store) {
        ...
    }
    function setNFTPrice(address<NFTStorage> store, uint price) {
        ...
    }
    function createNFT(address<NFTStorage> store, address to) {
        ...
    }
    function buyNFT(address<NFTStorage> store) {
        ...
    }
    function transferNFT(address<NFTStorage> store, uint tokenId,
        address from, address to) {
        ...
    }
    function destroyNFT(address<NFTStorage> store, uint tokenId) {
        ...
    }
}
```

Figure 2: A smart contract written in our Solidity refined grammar.

We can see that our syntax is very similar with the one Solidity has, except we allow declaring a subset of type address, as we can see in function signatures. With this refined syntax, we can detect before executing each function, if the address passed is indeed pointing to the `NFTStorage` contract specification. Thus passing an address belonging to an externally-owned account, will result in an error, before executing the function.

To illustrate the operational semantics, we show a possible interaction with these contracts. First, we deploy the `NFTStorage` contract into our mock blockchain, where we have two addresses, say **a1** (0xe1abd8...) and **a2** (0x229005...). Let both interact with the contract deployed and the whole blockchain through function calling (i.e., transactions).

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract
    Name: Game, State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract
    Name: NFTStorage, State Variables:
balances ----> [
 ]
lastTokenId ----> 0
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
 ]
price ----> 0
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance:
    100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance:
    100000
```

Figure 3: Output of our tool after executing first two transactions.

First **a1** deploys the `Game` contract, creating an instance of it on our blockchain and we can referencing this instance either by their unique instance identifier or address. Then the same user creates a new store, invoking function `createStore`. After the transaction we have another contract on our blockchain, an instance of `NFTStorage` and it is stored in the state variable `stores`. We also return the address where this instance is located and store it because it is useful to pass on the next functions. It is important to say that this address is stored in the type environment as a subtype of the address type and therefore its refined type is `address`⟨`NFTStorage`⟩. Figure 3 shows the state of the blockchain after these first two transactions (for each contract shows the content of state variables and its balance). Next **a1** sets each NFT price to 322. In Figure 4 you can see that **a2** earned an NFT, by invoking the `buyNFT` function: balances and owners are two state variables of `NFTStorage` contract, both being maps. The former mapping each externally owned account to the amount of tokens this user is owner of and the latter is a mapping between each token and its owner. We repeat this transaction in Figure 5, making now **a2** holding two NFTs. Lastly **a2** transfers the first NFT bought to **a1**, using the `transferNFT` function, as you can see in Figure 6: key 0 of state variable `owners`, it changed the value and we can also note that **a1** now has a balance of 1 in state variable `balances`.

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract
    Name: Game, State Variables:
stores ----> [
  0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract
    Name: NFTStorage, State Variables:
balances ----> [
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 1
 ]
lastTokenId ----> 1
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance:
    100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance:
    100000
```

Figure 4: Output after the fourth transaction.

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract
    Name: Game, State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract
    Name: NFTStorage, State Variables:
balances ----> [
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 2
 ]
lastTokenId ----> 2
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0x2290052e72090ecde135bde4c864732239f32c19
  1 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance:
    100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance:
    100000
```

Figure 5: Output after the fifth transaction.

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d , Contract
    Name: Game , State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751 , Contract
    Name: NFTStorage , State Variables:
balances ----> [
  0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097 ---> 1
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 1
 ]
lastTokenId ----> 2
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097
  1 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance:
    100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance:
    100000
```

Figure 6: Output after all transactions.

4 Conclusion

In short our technical contributions are the following: 1) a parser for our extended version of Featherweight Solidity to allow experimenting our proof-of-concept without much effort; 2) the formalization and implementation of a type system offering a new address type with support for multiple inheritance via the C3 linearization algorithm; 3) the formalization and implementation of an interpreter to execute contracts.

By addressing the weak typing concerns of the original language and leveraging the strengths of OCaml, we show how to offer developers a safer and more reliable programming paradigm for smart contracts. As briefly presented herein, with our demonstration of the tool's behavior, exemplified by a collection of NFT stores contract, we will showcase the benefits that our implementation brings to the table and the feasibility of it. By leveraging this new subtype, developers can now enjoy increased confidence in their code, reduced risk of errors, and improved overall efficiency. We also expect our prototype to be easily extended to cope with further Solidity features, as well as yet to be discovered vulnerabilities.

In this paper we showed our enhancement of Featherweight Solidity, a novel formalization and implementation of a type-safe subset of Solidity.

Acknowledgement This work is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[2] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In Lougie Anderson and James Coplien, editors, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, pages 69–82. ACM, 1996.

[3] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf, 2014.

[4] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11599 of *Lecture Notes in Computer Science*, pages 138–153. Springer, 2019.

[5] Matteo Di Pirro. How solid is solidity? an in-dept study of solidity's type safety. Msc Thesis. Università degli Studi di Padova, 2018.