Towards a Solider Solidity: Taming Type Casts*

António Ravara
NOVA School of Science and Technology, Portugal

Abstract

We address an (as far as we know) untreated vulnerability in Solidity smart contracts, termed Type Casts, associated with the type address. The address type, functioning as an untyped pointer to either an externally-owned account (EOA) or a contract, poses a risk when casting an address to a contract instance. Building on Featherweight Solidity, a type-safe formal language capturing a significant subset of Solidity, we introduce a refined typing approach for addresses, effectively mitigating the identified vulnerability and contributing to safer smart contract development in Solidity. We validate our approach with illustrative examples.

1 Introduction and talk plan

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 [ABC17].

Inspired by the need for a safer and more robust variant of Solidity, we revisit Featherweight Solidity (FS), a type-safe formal (with a well-defined syntax, type system, and operational semantics) subset of the Solidity programming language [CPZ19, Pir18]. Our formalization extends the previous work of Silvia Crafa and Matteo Di Pirro in two crucial ways:

1. provides a formal account for Solidity's multi-inheritance, using C3 linearization [BCH+96];
2. refines the type address, allowing to restrict it so it can be possible to distinguish between addresses referencing different contracts and externally owned accounts.

We also leveraged the expressiveness power of OCaml, a functional programming language known for its strong type inference capabilities, to implement a type-checker and an interpreter for Featherweight Solidity.[1] These tools make thus available an environment to test with elaborate examples that would be too demanding to type and execute by hand. To exemplify the approach and the language expressiveness, we implemented real-life contracts that the interested reader can find in the `example` folder of the git project.

*Joint work with João Reis and Mário Pereira

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

In the proposed talk, we will present the vulnerability we are addressing, showing in detail with illustrative examples that type casts exists in Solidity and how dangerous they may be; then we will explain the correction that we propose and show how our version of FS avoids the problematic situations via its relevant features; finally we will demonstrate how our tools work with those illustrative examples.

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 the figure below 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).

Currently, there are still no ways in Solidity to prevent this vulnerability.

3 Featherweight Solidity 2.0 by Example

We will present a practical example to understand both the utility of our refined type system in mitigating unintended uses of the Solidity programming language, and how in practice the language features and mechanisms prevent the vulnerability. The full version of these contracts are in the GitHub repository. The NFTStorage generates non-fungible tokens (NFTs) following the ERC721 standard. We adapted its implementation from Open Zeppelin to fit Feathweight Solidity 2.0 syntax. The Game contract facilitates interaction with a set of NFTStorage contracts identified by their respective addresses.

In order to implement C3 Linearization [BCH+96] as an algorithm for resolving contract dependencies, we have formalised an existing generic implementation to align with the requirements of our project, as documented in [c3l].

4 Summary of contribution

There are other formalizations of Solidity, besides Featherweight Solidity. One notable paper [JKL+18] outlines the implementation of operational semantics for Solidity within the K-Framework [RS10], presenting an abstract model of semantics and elucidating various rules. Another proposal introduces TinySol as a core calculus language, providing a formalization of big-step semantics for Solidity [BGM19]. Additionally, there is also work that formalizes Solidity employing big-step semantics and approaching multiple inheritance through C3 linearization [Zak18].

We opted to improve Featherweight Solidity as it was the sole framework explicitly addressing the type cast vulnerability. To demonstrate the utility of our formalization, we developed a suit of examples. To make our formalization available as a tool to explore examples, we implemented a parser, a type-checker, and an interpreter. The code is open and public.

We acknowledge a limitation in our implementation: given that smart contracts operate on shared virtual machines, the risk of contract name collisions is substantial. To address this, we propose an enhancement: incorporating not only the contract name but also the hash of the contract code as part of the address type, denoted as $\texttt{address}\langle C, H\rangle$, where H represents the hash of the contract code. This modification aims to mitigate the likelihood of naming conflicts and enhance the robustness of our approach.

References

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Proceedings of the 6th International Conference on Principles of Security and Trust (POST'17), held as Part of*

the European Joint Conferences on Theory and Practice of Software (ETAPS'17), volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[BCH+96] 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'96)*, pages 69–82. ACM, 1996.

[BGM19] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquín García-Alfaro, editors, *Proceedings of the International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 11737 of *Lecture Notes in Computer Science*, pages 233–243. Springer, 2019.

[c3l] Implementing c3 linearization. `https://xivilization.net/~marek/blog/2014/12/08/implementing-c3-linearization/`.

[CPZ19] 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, *Revised Selected Papers of the International Workshop on Financial Cryptography and Data Security (FC'19)*, volume 11599 of *Lecture Notes in Computer Science*, pages 138–153. Springer, 2019.

[JKL+18] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Executable operational semantics of solidity. *CoRR*, abs/1804.01295, 2018.

[Pir18] Matteo Di Pirro. How Solid is Solidity? An In-dept Study of Solidity's Type Safety. Msc Thesis. Università degli Studi di Padova, 2018.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[Zak18] Jakub Zakrzewski. Towards verification of ethereum smart contracts: A formalization of core of solidity. In Ruzica Piskac and Philipp Rümmer, editors, *Revised Selected Papers of the 10th International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE'18)*, volume 11294 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2018.