



Solidity CTF — Part 2: “Safe Execution”



Alexander Wade

Jun 25, 2018 · 7 min read

Exploring methods to allow for safe, controlled execution of arbitrary code using `delegatecall`.

This article is part 2 of a series of Solidity wargames designed to demonstrate some of the low-level behavior of Solidity through the exploitation of vulnerable code. Each round will attempt to present some unique functionality or combination of functionalities which must be isolated, understood, and exploited in order to complete the challenge. Additionally, each round released will contain a thorough explanation of the previous round.

Part 2: “Safe Execution”

Part 2 has been deployed to Ropsten and explores the execution of arbitrary contracts through `delegatecall`. It also ties in some of the techniques used in Part 1 — so read on for an explanation!

The goal of Part 2 is to make yourself the owner of the contract.

There is a Reddit thread for this challenge as well. Feel free to participate and ask questions!

Good luck!

Explanation — Part 1: “Function Types”

Part 1 was released informally here and broken by address

0xef045a554cbb0016275E90e3002f4D21c6f263e1 within 5 hours. The challenge was to empty the contract of funds: 1 REth.

Let’s look at the code step-by-step (if you’d like to follow along, I’m using compiler version 0.4.24 and the optimizer is turned off!):

```
1  pragma solidity ^0.4.23;
2
3  contract FunctionTypes {
4
5      constructor() public payable { require(msg.value != 0); }
6
7      function withdraw() private {
8          require(msg.value == 0, 'dont send funds!');
9          address(msg.sender).transfer(address(this).balance);
10     }
11
12     function frwd() internal
13         { withdraw(); }
14
15     struct Func { function () internal f; }
16
17     function breakIt() public payable {
18         require(msg.value != 0, 'send funds!');
19         Func memory func;
20         func.f = frwd;
21         assembly { mstore(func, add(mload(func), callvalue)) }
22         func.f();
23     }
24 }
```

FunctionTypes.sol — Create the contract.

Our goal is to transfer the RETH out of the contract — which will only be possible via the `withdraw()` function — a private function that ensures the amount of wei sent to the contract is 0 — and then transfers the contract’s balance to the sender.

```
function withdraw() private {
    require(msg.value == 0, 'dont send funds!');
    address(msg.sender).transfer(address(this).balance);
}
```

In order to access the private function, we will need to call it from some other function. At first glance, `breakIt()` appears to call `withdraw()` through `frwd()` — but there is one caveat: `breakIt()` and `withdraw()` have conflicting requirements — one requires that `msg.value` is nonzero, and the other requires that it is zero. As-is, if `breakIt()` called `withdraw()` or `frwd()` the requirements would disallow any form of execution: a transaction cannot simultaneously have 0 wei sent, and not-0 wei sent. Luckily, `breakIt()` has some additional complexity:

```
function breakIt() public payable {
    require(msg.value != 0, 'send funds!');
    Func memory func;
    func.f = frwd;
    assembly { mstore(func, add(mload(func), callvalue)) }
    func.f();
}
```

To understand what’s going on here, we need to understand a little about structs and function types — as well as the way Solidity’s execution happens at runtime. Let’s start with the `Func` struct:

```
struct Func { function () internal f; }
```

The `Func` struct has a single member — `f`, which is an internal function that takes and returns no parameters. Function types in Solidity behave rather predictably — they are assigned the value of some function and when they are executed, call that function exactly the same way the function would be called if it were not stored as a variable. In the following example, function types are used to dynamically execute either the `add` or `sub` methods at runtime:

```

1  pragma solidity ^0.4.23;
2
3  contract AddSub {
4
5      function add(uint a, uint b) internal pure returns (uint) {
6          return a + b;
7      }
8
9      function sub(uint a, uint b) internal pure returns (uint) {
10         return a - b;
11     }
12
13     function math(uint _a, uint _b, bool _add) public pure returns (uint) {
14         function (uint, uint) internal pure returns (uint) func;
15         func = _add ? add : sub;
16         return func(_a, _b);
17     }
18 }

```

AddSub.sol hosted with ❤️ by GitHub

[view raw](#)





ADDSUB.SOI

For more information about function types, the Solidity documentation is very helpful.

On line 19, an empty `Func` struct is initialized. Structs in Solidity are essentially pointers — although Solidity does not provide a generic pointer type, a struct variable is stored on the stack as a pointer to a location in memory where the members of that struct are stored. To understand this better, let's have a quick look through Remix's step debugger, sending 1 wei to `breakIt()` to get past the initial check:

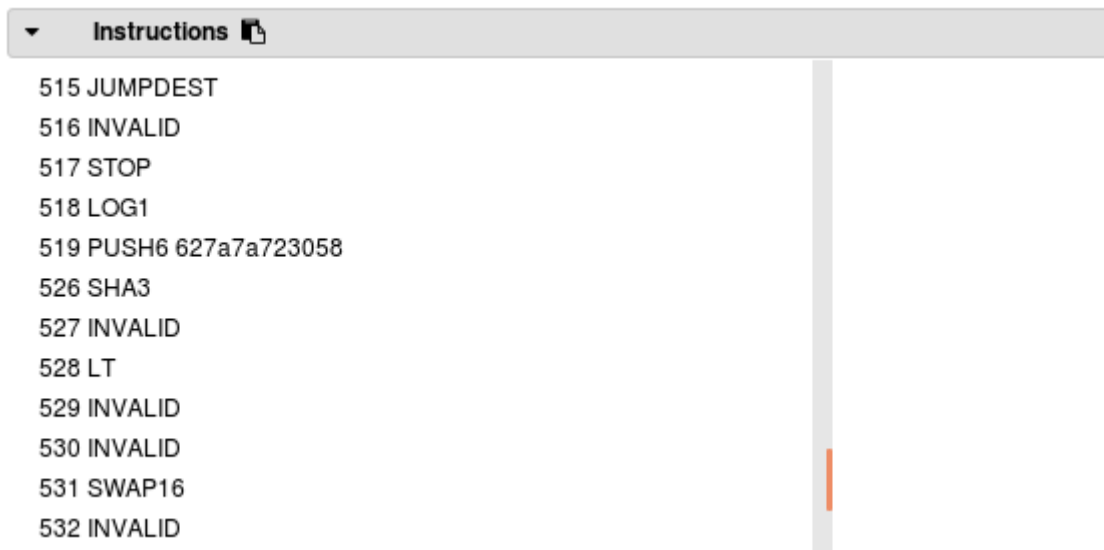
The screenshot shows the Remix IDE interface. On the left, the Solidity code is displayed, with line 19 highlighted. The code defines a contract `FunctionTypes` with a constructor, a `withdraw()` function, a `frwd()` function, and a `breakIt()` function. The `breakIt()` function initializes a `Func` struct and calls `assembly` to store the function pointer.

On the right, the step debugger is open. It shows the current transaction index and block number. The `Stack` panel displays the current stack state, including the `Func` struct. The `Memory` panel shows the memory layout, with the `Func` struct located at address `0x6a1f9e19`.

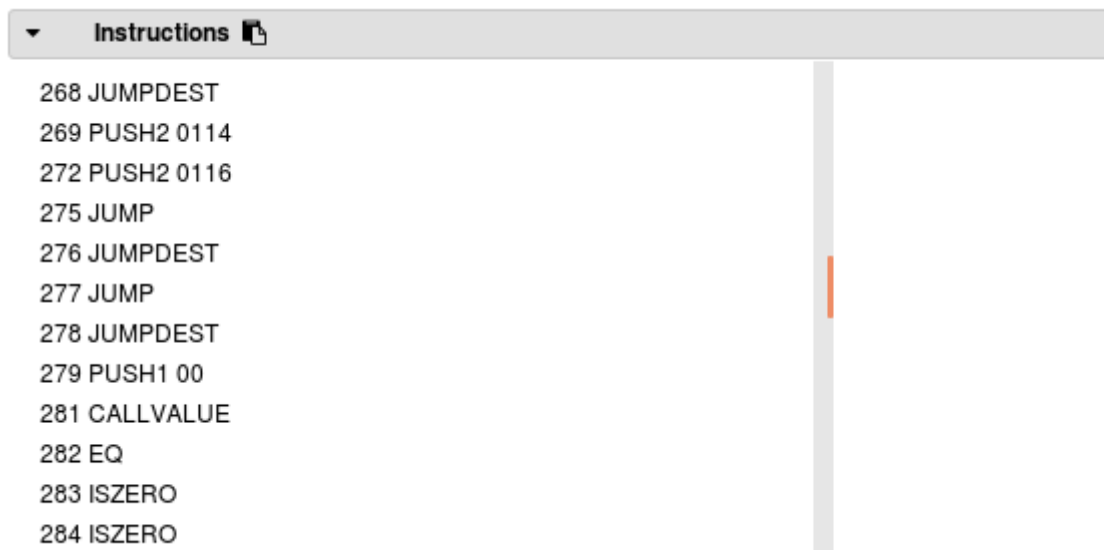
► Call Data 
► Call Stack 
► Return Value 
► Full Storages Changes 

Func struct — pointer to 0x80 in memory

Here we see that the `func` variable is stored on the stack as a simple hexadecimal number, 0x80. Expanding the memory tab, we can see that the corresponding memory location has a value of 0x203 at initialization, but that this number changes to 0x10C when `func.f` is assigned the value of `frwd()`. 0x203 and 0x10C correspond to locations in the contract’s bytecode. The easiest way to see these locations is to pop open the ‘Instructions’ tab in the debugger, and convert both values to base-10 (the ‘Instructions’ tab uses base-10 instead of hexadecimal):



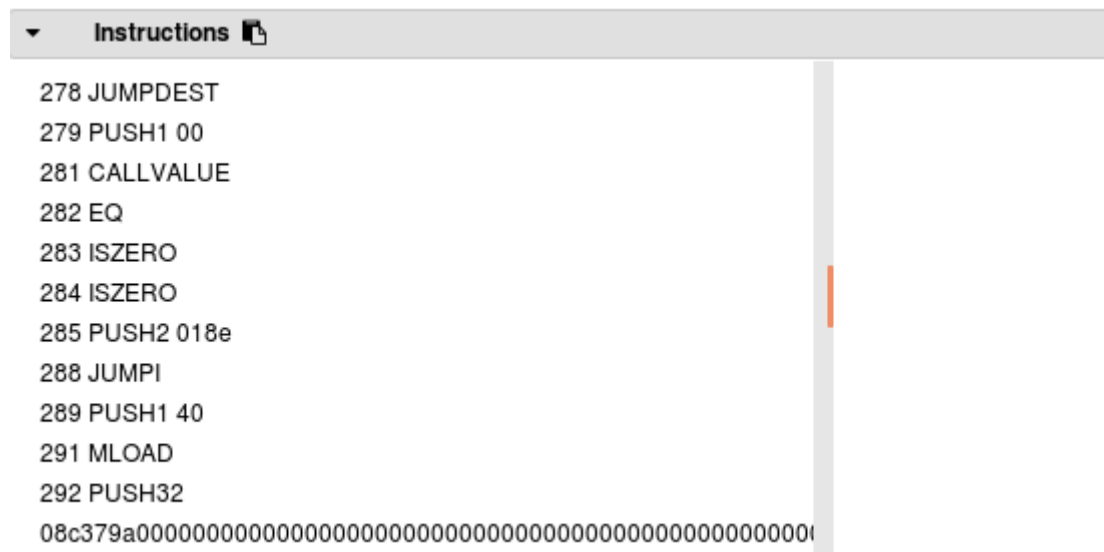
0x203 = 515 — ‘invalid’ jumpdest



0x10C = 268 — `frwd()` jumpdest

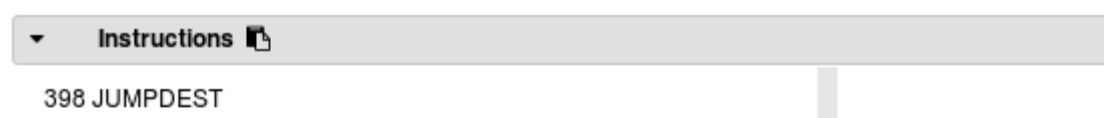
As expected, both values resolve to locations in the code that correspond to a `JUMPDEST` operation. The default `0x203` value moves to location 515, and as 516 contains an `INVALID` opcode, we know that jumping to this position in the code will throw on the next instruction. As `0x203` was the default value for the function type, this makes sense — you can't use a function type if it has not been assigned a function.

The 0x10C value corresponds to a much more sane position in the code. We see that 268 contains a `JUMPDEST`, after which 2 values are pushed, and another `JUMP` executed. Because `JUMP` uses the value on the top of the stack, it's fair to assume the destination it will jump to is 0x116 (278). The instructions at 268 correspond to what we should expect from the `frwd()` function — all it does is call `withdraw()`, so a simple `PUSH`, `PUSH`, `JUMP` means a function is immediately called. Let's take a look at the location it's jumping to:



0x116 = 278 — withdraw() jumpdest

Execution lands nicely at our expected `JUMPDEST` at 278! Briefly examining the next few instructions, we see a `CALLVALUE` — which pushes `msg.value` to the stack, followed by a check to see if `CALLVALUE` is 0. If it is, we push another location (0x18E) and jump there. If it isn't, the next few instructions in the screenshot reference the free memory pointer (0x40) and the function selector for `Error(string)`, which is used to revert with an error message (see the relevant docs for more info). For brevity's sake, we won't explore the next few instructions — but it's likely they will load 'dont send funds!' into memory and revert. Let's look instead at the location jumped to if `CALLVALUE` is 0, 0x18E:



```

399 CALLER
400 PUSH20 ffffffffffffffffffffffffff
421 AND
422 PUSH2 08fc
425 ADDRESS
426 PUSH20 ffffffffffffffffffffffffff
447 AND
448 BALANCE
449 SWAP1
450 DUP2
451 ISZERO

```

0x18E = 398 — funds transfer jumpdest

The first thing to notice are the opcodes `CALLER`, `ADDRESS`, and `BALANCE`, which correspond to `msg.sender`, `address(this)`, and `.balance`, respectively. Awesome — that’s where the fund transfer occurs!

Now that we have these values in mind, let’s examine at what happens when we run `breakIt()`. After `func.f` is set to `frwd()`, we have a block of assembly:

```
assembly { mstore(func, add(mload(func), callvalue)) }
```

This may look confusing initially, but it’s actually quite simple, especially when we take into consideration what we’ve discussed above. Looking at the Solidity assembly docs, we know that `mstore` takes two parameters: a location, and a value. `mstore` stores the 32 byte value at the location in memory. Great — so our location is `func`, and our value is `add(mload(func), callvalue)`. We could step-debug and see what happens when these values are used, but we already have all of the requisite information. `func` references the `Func` struct, which we know is the pointer `0x80`. `add(mload(func), callvalue)` simply pulls the value stored at the location `0x80` and adds `msg.value` to it. We showed earlier that `0x80` in memory was storing the value of the `frwd()` JUMPDEST, `0x10C`.

So after our assembly block executes, `func`, or `0x80`, will not be `0x10C` — it will be `0x10C + msg.value`. In our first screenshot, we sent `1 wei` — so we should expect `0x80` to store `0x10D`, instead. Let’s step through:

```

1 pragma solidity ^0.4.23;
2
3 contract FunctionTypes {
4
5     constructor() public payable { require(msg.value != 0); }
6
7     function withdraw() private {
8         require(msg.value == 0, 'dont send funds!');
9         address(msg.sender).transfer(address(this).balance);
10    }
11 }

```

Block number
Transaction index or hash

▶
■

Transaction

0x80 now contains 0x10D

To sum this all up — the challenge requires the user to send a ‘magic number’ amount of wei to the `breakIt()` function, which will be added to the original `JUMPDEST` to jump past the `require` statement in `withdraw()` and straight to the fund transfer. Checking Ropsten, our solver seems to have figured this out perfectly:

<https://medium.com/authio/solidity-ctf-part-2-safe-execution-ad6ded20e042>



Success! 130 wei sent, 1 REth received

Taking what we know now, we should expect that `130 + fwd()` is the destination we marked above as the start of the fund transfer block, 398. The proof of this is cheekily left as an exercise to the reader.

What have we learned?

1. Internal function types are simply references to locations in a contract's bytecode.
2. Functions are called using `JUMP`, which must have a corresponding `JUMPDEST`. `JUMP` is not limited to only functions, but any branching statement.
3. Structs are essentially pointers to memory.
4. By using assembly, it is possible to trick the compiler into allowing us to `JUMP` to unexpected locations, such as the middle of a function.

That's it! Best of luck in Round 2.

[Blockchain](#) [Security](#) [Ethereum](#) [Solidity](#)

[About](#) [Help](#) [Legal](#)