Return Data Length Validation: a Bug We Missed

Steve Marx
Jul 16, 2019 · 9 min read

Photo by patricia serna on Unsplash

A rather serious vulnerability was recently found in the 0x v2.0 Exchange, a smart contract system that our team audited. The bug went undiscovered for about a year and (thankfully) appears to never have been exploited. Kudos to samczsun, the security researcher who discovered the bug and reported it to 0x.

We know from our experience with them that the 0x team is highly competent and takes security extremely seriously. We've worked with 0x on a number of audits, including the one that covered this code, so our whole team was saddened to hear about this bug. You can find our audit report here, where we found a couple critical bugs but not this one: https://github.com/ConsenSys/0x_audit_report_2018-07-23.

When something goes wrong, it can be useful to conduct a *postmortem*, an after-the-fact analysis of what went wrong. The goal of a healthy postmortem is to identifying ways to improve.

In this blog post, I'd like to share some of what we learned from our postmortem on this particular bug. I'll explain the technical details of the vulnerability, demonstrate how smart contract developers can protect themselves from this class of bug, and share some lessons learned.

Validating Return Data Length

The Bug

I've distilled the vulnerable code down to a simple function. In the below code, assume that the `IFetcher` contract is supplied by a user. See if you can spot the bug:

```
pragma solidity 0.5.10;

interface IFetcher {
    function fetch() external view returns (uint256);
}

contract Vulnerable {
    function getNumber(IFetcher f) external view returns (uint256) {
        bytes32 selector = f.fetch.selector;

        uint256 ret;
        bool success;

        assembly {
            let ptr := mload(0x40)     // get free memory pointer
            mstore(0x40, add(ptr, 32)) // allocate 32 bytes
            mstore(ptr, selector)      // write the selector there
            success := staticcall(
                gas,                   // forward all gas
                f,                     // target
                ptr,                   // start of call data
                4,                     // call data length
                ptr,                   // where to write return data
                32                     // length of return data
            )
            ret := mload(ptr)          // copy return data into ret
        }

        require(success, "Call failed.");

        return ret;
    }
}
```

Under normal circumstances (a conformant `IFetcher` implementation), this code works fine, but something surprising happens if you pass in an externally owned account (EOA). An EOA address has no associated code, and this means **all calls to it succeed and return nothing**. The low-level `STATICCALL` opcode used here will return success but will only copy as much data into memory as was returned. So even though we asked for 32 bytes of return data, **nothing gets copied to memory because there is no data to copy**.

Because nothing gets copied, memory is just as it was before the call. That means `ptr` is still pointing to the `selector` data, so this function returns `0xa95c372d000000…` (the function selector for `fetch()`).

The same thing happens if you call a contract that implements `fetch()` without a return value or just implements a fallback function. The bug occurs any time the call succeeds but doesn't return as much data as the caller expects.

Why This Matters In the 0x Exchange

This bug manifested itself in the 0x v2.0 Exchange in a particularly nasty way due to where the bug was. The exchange allows trades to be constructed off chain and then validated and filled on chain. To know that a trade is authorized, the exchange must validate a trader's signature.

One option the exchange offers for validating a trader's signature is to delegate that checking to a contract. Essentially, the trader can be a smart wallet that implements a function with the signature `isValidSignature(bytes32,bytes)` and returns a boolean `true` if the signature is valid.

As you have no doubt anticipated, the problem is that `isValidSignature` is called via code that's very similar to the above vulnerable contract. If the target is *not* a smart contract (or otherwise returns no data), the data already in memory stays unchanged. Any non-zero value there is interpreted as `true` and authorizes the trade.

Even this would be okay if traders had to specifically opt in to this type of signature scheme, but remember that these trades are constructed off chain to avoid the expense and delay of making an on chain transaction. So the person *taking* the proposed trade is able to dictate which type of signature checking should happen via a flag.

All an attacker needs to do is trade with an externally owned account and specify that the signature should be checked by calling `isValidSignature`. This validation will succeed because no data is returned and the data already present in memory is non-zero.

How It Was Introduced

This bug was introduced in a fix to a critical reentrancy issue identified in the initial phase of our audit. The recently introduced `STATICCALL` opcode was an ideal solution, but it wasn't yet used by the current version of Solidity (0.4.22).

The 0x team considered two options that would allow them to use `STATICCALL`. The first was to add `pragma experimental “v0.5.0”;` to the file. The downside here was that many other new compiler features would be activated.

The second option was to use Solidity assembly, which gave access to the new opcode even in Solidity 0.4.22. The team opted for the latter approach, which required reimplementing the functionality (and protections) provided by Solidity's higher level function call syntax.

We reviewed their code but did not identify the issue in their implementation. A good method for reviewing this type of code in the future would be to compare it to the compiler output of analogous high-level Solidity code.

The Fix

There are a few ways to fix this class of vulnerability. The 0x team took a sensible defense-in-depth approach in their patch.

If you don't need to write hand-optimized assembly, by far the best approach is to take advantage of Solidity's high-level syntax for making contract calls:

```
contract SafeSolidity {
    function getNumber(IFetcher f) external view returns (uint256) {
        return f.fetch();
    }
}
```

Since Solidity 0.4.22, the compiler emits byte code that explicitly checks the size of the return data and reverts if not enough data is returned. If you pass in an EOA to the `SafeSolidity` contract, the call will revert.

If you *do* need to write assembly, a good fix is to write your own check for the return data size and revert if it's wrong:

```
success := staticcall(...)
if lt(returndatasize, 32) {
    revert(0, 0)
}
```

The 0x team implemented that fix but went beyond this in two ways:

1. Before even making the call, the new code uses `extcodesize` to check if the target address is a smart contract. If it isn't, the transaction is immediately reverted.
2. Instead of just checking for a boolean `true`, the new code requires a specific return value (a "magic salt" value). Anything else is treated as failure.

That second improvement also helps in the case that a contract implements `isValidSignature(bytes32,bytes)` for some other reason (e.g. to work with another contract that uses the same type of signature validation scheme). The contract author may not have intended to have their function used to validate 0x exchange signatures, and this improvement means they won't accidentally be used that way.

Lessons Learned

Aside from the technical lesson about a gotcha of the Ethereum Virtual Machine (EVM), there are some other lessons our team has taken from this experience. Some of these lessons may be relevant to others in the Ethereum community, so I'd like to share those here.

Audited Code Can Still Have Bugs

Perhaps the most important lesson is that an audit isn't a guarantee. All code is at risk of having bugs. A security audit can help to assess that risk and to find some of the worst bugs early, but it can't promise to identify all bugs. Even formal verification, which provides stricter proofs around correctness, can only find bugs to the extent that the code is formally specified. A bug like this one can easily slip through such a process.

Have a Contingency Plan

When a bug *is* discovered after deployment, it's important to have a plan for how to respond. Sometimes, it's simply too late by the time you learn about the bug because it's already been exploited in an irreparable way. But for cases where a security researcher discloses a bug privately or the bug has limited reach, you'll want to know what to do.

In "Upgradeability Is a Bug", I argued against the unrestricted ability to change contract code after deployment, but that doesn't mean you can't plan other mechanisms for fixing bugs. 0x has several safe upgrade mechanisms for various parts of the system, but in this case they opted for what is often the simplest and best plan. They redeployed the whole system and had users use the new contracts instead.

Having contingency plans is important, but you also need to make sure your processes are well documented. This includes things like how to rotate keys, how to redeploy the system, who can unlock a multisig wallet, how to update your DApp to use a new contract, etc. When something happens, you'll want to be able to act quickly and confidently.

Testing Isn't the Answer

In retrospect, it seems like there's a simple test that would have caught this issue: just try validating a signature with an externally owned account. This is often true **in retrospect**, but it's another matter to come up with all the important test cases before knowing exactly where a bug will be found.

In particular, please note that you can easily have 100% code coverage of this vulnerable code without catching the bug.

Inline Assembly Is Risky

Although we didn't call it out in this audit report (because the 0x team was well aware already), we tend to warn clients that inline assembly is riskier than sticking to Solidity.

Assembly code is harder to read/audit, and it's also just harder to get right. As gotchas like this one are discovered, Solidity gets smarter about the code it emits. Individual developers and auditors have to instead just try to remember everything.

Sharing and Encoding Knowledge

It's impossible for everyone to know about every gotcha, every technique, and every vulnerability. One aspect I really enjoy about working within the security community is that everyone seems committed to sharing their knowledge and expertise.

ConsenSys Diligence in particular aspires to the lofty mission of "creating a safe, trustworthy and healthy Ethereum ecosystem". That obviously won't be achieved by just auditing code and developing our in-house expertise.

Sharing knowledge like we're doing here is a great way to contribute, but we should all recognize that not everyone can know everything. For each piece of knowledge and expertise, our team likes to think about the following ways we can avoid just "everyone has to know this". This list is in order of preference:

1. **Can we make the problem go away altogether?** For example, uninitialized storage pointers used to be a common cause of vulnerabilities in Solidity code, but since Solidity 0.5.0, such vulnerable code simply won't compile. Now this is one less thing developers and auditors need to know about.
2. **Can we address the problem with tools?** Analysis tools in the Ethereum space have come a long way quite quickly. Our own team has MythX, a service for running a variety of analysis tools. When new classes of bugs appear, we can build knowledge of them into the tools. Reentrancy vulnerabilities are an example of an anti-pattern that tools can identify quite effectively.
3. **Can we address the problem with checklists?** Checklists are a well-known way of sharing knowledge and preventing mistakes. Every audit our team does starts with a list of "TODO" issues in GitHub that remind us of things to look for in the code we're auditing. After each audit, we perform a retrospective and look for things to add or remove from that list. To make sure the TODOs stay relevant, we plan to make more specific TODO lists for different types of code. This particular vulnerability, for example, might go in a list of things to look for in inline assembly. (This is, of course, only if the issue can't be adequately addressed with tooling.)
4. **Can we document the problem?** If nothing else, we can document and organize issues so others have an easier time learning about them. Our team maintains the Ethereum Smart Contract Security Best Practices and Smart Contract Weakness Classification Registry for this reason, but there's never enough time in the day to document everything we'd like to. Contributions are welcome to both, so please share your expertise!

Summary

- This particular bug has to do with a gotcha around the `CALL` and `STATICCALL` EVM opcodes. When using these low-level calls, it's up to you to explicitly check the length of the return data.

- If you want to avoid this class of bugs, avoid inline assembly and stick to Solidity. If you must use assembly, be sure to validate the return data length!
- Writing code carefully, testing thoroughly, and getting a security audit are all great ways to improve the security of your code, but none are a panacea. Make sure you have thought through contingency plans.
- We have to make it easier for *everyone* to write secure code, and this requires a holistic approach that takes into account improving the system itself, building tools, and sharing our expertise.

Further Reading

- samczsun, who discovered the vulnerability, has a writeup here: https://samczsun.com/the-0x-vulnerability-explained/.
- A related bug (but kind of the opposite) was discovered in code dealing with non-conforming ERC20 implementations last June: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca. There, it was a *fix* to Solidity that uncovered the issue.
- 0x's initial communication about the issue can be found here: https://blog.0xproject.com/shut-down-of-0x-exchange-v2-0-contract-and-migration-to-patched-version-6185097a1f39.

Thanks to Maurelian.

Ethereum Smart Contract Security Solidity Smart Contracts