Evasion Techniques

State Of The Art

Web3

Apehex

04/08/2023

Known Techniques

1. Faking

1.1. Fake Standard Implementation

1.1.1. Technical Details

This technique takes on the function & class names from the ERC standards, but the code inside is actually different.

The malicious contracts generally pretend to be:

proxies but the implementation is either not used or different from the ERC-1967 proxy

tokens but the transfer and / or approve functions behave differently than ERC-20 / 721 / 1155

1.1.2. Evasion Targets

Etherscan the interpretation of proxy is fixed, it can easily be fooled

users few users actually check the code, having a valid front is enough

1.1.3. Samples

Fake EIP-1967 Proxy

Standard EIP-1967 [**eip-1967**] has pointers located in specific storage slots:

0x360894a13ba... location of the logic contract address

0xa3f0ad74e54... location of the beacon contract address

These can be kept null or point to any contract, while the proxy actually uses another address.

A minimal example was given at DEFI summit 2023 [**video-masquerading-code**]:

```
function _getImplementation() internal view returns (address) {
    return
        StorageSlot
            .getAddressSlot(bytes32(uint256(keccak256("eip1967.fake")) - 1)).
            .value;
}
```

Etherscan will show some irrelevant contract, giving the impression it is legit.

1.1.4. Detection & Countermeasures

Several sources can be monitored:

Storage comparing the target of `delegateCall` to the address in the storage slots of the standards

Events changes to the address of the logic contract should come with an `Upgraded` event

Bytecode the implementation of known selectors can be checked agains the standard's reference bytecode

1.2. Overriding Standards Implementation

1.2.1. Technical Details

Like the previous technique 1.1, the goal is to have a malicious contract confused with legitimate code.

It is achieved by inheriting from standardized code like `Ownable`, `Upgradeable`, etc. Then, the child class overwrites key elements with:

redefinition an existing keyword is defined a second time for the references in the child class only

polymorphism an existing method can be redined with a slightly different signature

From the perspective of the source code, a single keyword like `owner` can refer to different storage slot depending on its context. It is only in the bytecode that a clear difference is made.

1.2.2. Evasion Targets

This technique is refinment of the previous one: it will work on more targets.

Etherscan blockchain explorers lack even more flexibility to detect these exploits

Users the source code is even closer to a legitimate contract

Reviewers the interpretation of the source code is subtle, and reviewing the bytecode is very time consuming

1.2.3. Samples

Attribute Overwriting

In section 3.2.2, the paper [**paper-art-of-the-scam**] shows an example of inheritance overriding with `KingOfTheHill` :

```
contract KingOfTheHill is Ownable {
    address public owner; // different from the owner in Ownable

    function () public payable {
        if(msg.value > jackpot) owner = msg.sender; // local owner
        jackpot += msg.value;
    }
    function takeAll () public onlyOwner { // contract creator
        msg.sender.transfer(this.balance);
        jackpot = 0;
    }
}
```

In the modifier on `takeAll`, the `owner` points to the contract creator. It is at storage slot 1, while the fallback function overwrites the storage slot 2.

In short, sending funds to this contract will never make you the actual owner.

Method Overwriting

1.2.4. Detection & Countermeasures

While subtle for the human reader, it is rather easy to detect when expected:

source code the sources can be checked for duplicate definitions & polymorphism

bytecode

Since the whole point is to advertize for a functionality with the sources, they will be available.

2. Morphing

2.1. Red-Pill

The red-pill technique detects simulation environment to disable its exploits upon scrutiny.

2.1.1. Evades

Live tests in transaction simulations: often performed by wallets before sending a transaction.

2.1.2. How

The contract detects simulation environments by:

- comparing the global variables with settings found in simulated environments:
  - block.basefee with
  - block.coinbase with 0x0000000000000000000000000000000000000000
  - tx.gasprice with

Then it triggers legitimate code in simulation contexts and malicious code on the mainnet.

2.1.3. Samples

The contract [FakeWethGiveaway](red-pill/FakeWethGiveaway.sol) checks the current block miner's address:

```
function checkCoinbase() private view returns (bool result) {
    assembly {
        result := eq(coinbase(), 0x0000000000000000000000000000000000000000)
    }
}
```

When null (test env), it actually sends a reward and otherwise it just accepts transfers without doing anything.s

2.1.4. Detection & Countermeasures

- Looking for unusual opcodes: typically ‘block.coinbase‘.
- Replaying transactions and fuzzing the global variables.

2.1.5. Resources

- [**article-red-pill**]

3. Obfuscation

3.1. Hiding In Plain Sight

3.1.1. Evades

Here, the goal is to overwhelm source code reviewers with the sheer volume of code.

It also lowers the efficiency of ML algorithms.

3.1.2. How

By stacking dependencies, the scammer grows the volume of the source code to thousands of lines.

99% of the code is classic, legitimate implementation of standards.

And the remaining percent is malicious code, hidden inside one of the numerous dependencies for example.

3.1.3. Samples

Hidden among 7k+ lines of code:

```
// no authorization modifier `onlyOwner`
function transferOwnership(address newOwner) public virtual {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}
```

3.1.4. Detection & Countermeasures

1. The proportion of unused code can be leveraged from the transaction history.

3.2. Hiding Behind Proxies

3.2.1. Evades

- Etherscan code verification - source code reviews

3.2.2. How

Keeping the sources closed by only exposing a proxy contract.

3.3. Hidden State

3.3.1. Evades

Totally bypasses source & bytecode analysis by humans & tools.

3.3.2. How

At the construction / initialization, data can be put in storage at arbitrary slots.

3.3.3. Detection & Countermeasures

Detecting access to:

- arbitray storage locations
- locations given as input

4. Poisoning

4.1. Event Poisoning

5. Redirection

5.1. Hidden Proxy

5.1.1. Evades

This technique allows scammers to verify their contracts will dodging source code reviews.

5.1.2. How

The contract performs delegateCalls on any unknown selector.

The target address can be hardcoded, making it

In the end, the exposed functionalities are not meaningful, the logic is located at a seemingly unrelated address.

5.1.3. Samples

```
fallback () external {
  if (msg.sender == owner()) {
    (bool success, bytes memory data) = address(0
        x25B072502FB398eb4f428D60D01f18e8Ffa01448).delegateCall(
      msg.data
    );
  }
}
```

5.2. Selector Collisions

5.2.1. Evades

This subtle

5.2.2. How

Because the function selectors are only 4 bytes long, it is easy to find collisions.

When a selector in the proxy contract collides with another on the implementation side, the proxy takes precedence.

This can be used to override key elements of the implementation.

5.2.3. Samples

As shown in [the talk by Yoav Weiss at DSS 2023][video-masquerading-code]:

```
function IMGURL() public pure returns (bool) {
  return true;
}
```

This function has the same selector as keccak("vaultManagers(address)")[0:4].

Foreseen Techniques

6. Obfuscation

6.1. Payload Packing

6.1.1. Evades

Pattern matching on the bytecode.

6.1.2. How

Encryption / encoding / compression can be leveraged to make malicious code unreadable.

6.1.3. Detection & Countermeasures

1. Scanning for high entropy data

Appendices

G. Samples

H. Red Pill