UNIVERSITÀ DI TRENTO

ProSVED
Projection of Security Vulnerabilities caused by Exploits in Dependencies

Towards vulnerability impact quantification in Solidity smart contracts

Preparing for security risk prediction

Carlos E. Budde & Giacomo Checchini

Security group @ Dipartimento di Ingegneria e Scienza dell'Informazione

carlosesteban.budde@unitn.it

12.06.2024

OWASP top-10 (again? c'mon…)

OWASP

PROJECTS CHAPTERS EVENTS ABOUT

OWASP Smart Contract Top 10

Main | Acknowledgements | Join

About the Smart Contract Top 10

The OWASP Smart Contract Top 10 is a standard awareness document that intends to provide Web3 developers and security teams with insight into the top 10 vulnerabilities found in smart contracts.

It will serve as a reference to ensure that smart contracts are secured against the top 10 weaknesses exploited/discovered over the last couple of years.

Top 10

- SC01:2023 - Reentrancy Attacks
- SC02:2023 - Integer Overflow and Underflow
- SC03:2023 - Timestamp Dependence
- SC04:2023 - Access Control Vulnerabilities
- SC05:2023 - Front-running Attacks
- SC06:2023 - Denial of Service (DoS) Attacks
- SC07:2023 - Logic Errors
- SC08:2023 - Insecure Randomness
- SC09:2023 - Gas Limit Vulnerabilities
- SC10:2023 - Unchecked External Calls

OWASP top-10 (again? c'mon...)

OWASP PRO

OWASP Smart Contract Top 10

Main | Acknowledgements | Join

About the Smart Contract Top 10

The OWASP Smart Contract Top 10 is a standard awareness d
and security teams with insight into the top 10 vulnerabilities

It will serve as a reference to ensure that smart contracts are
discovered over the last couple of years.

Top 10

- SC01:2023 - Reentrancy Attacks
- SC02:2023 - Integer Overflow and Underflow
- SC03:2023 - Timestamp Dependence
- SC04:2023 - Access Control Vulnerabilities
- SC05:2023 - Front-running Attacks
- SC06:2023 - Denial of Service (DoS) Attacks
- SC07:2023 - Logic Errors
- SC08:2023 - Insecure Randomness
- SC09:2023 - Gas Limit Vulnerabilities
- SC10:2023 - Unchecked External Calls

Vulnerability: Reentrancy

Description:

A reentrancy attack exploits the vulnerability in smart contracts when a function makes an external call to another contract before updating its own state. This allows the external contract, possibly malicious, to reenter the original function and repeat certain actions, like withdrawals, using the same state. Through such attacks, an attacker can possibly drain all the funds from a contract.

Example (DAO Hack):

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

    uint fundsToBeMoved = (balances[msg.sender] * p.splitData[0].splitBalance) / p.splitData[0].totalSupply;
    //Since the balance is never updated the attacker can pass this modifier several times
    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false) throw;
```

OWASP top-10 (again? c'mon…)

OWASP

PRO

OWASP Smart Contract Top 10

Main | Acknowledgements | Join

About the Smart Contract Top 10

The OWASP Smart Contract Top 10 is a standard awareness d
and security teams with insight into the top 10 vulnerabilities

It will serve as a reference to ensure that smart contracts are
discovered over the last couple of years.

Top 10

- SC01:2023 - Reentrancy Attacks
- SC02:2023 - Integer Overflow and Underflow
- SC03:2023 - Timestamp Dependence
- SC04:2023 - Access Control Vulnerabilities
- SC05:2023 - Front-running Attacks
- SC06:2023 - Denial of Service (DoS) Attacks
- SC07:2023 - Logic Errors
- SC08:2023 - Insecure Randomness
- SC09:2023 - Gas Limit Vulnerabilities
- SC10:2023 - Unchecked External Calls

Vulnerability: Reentrancy

Description:

A reentrancy attack exploits the vulnerability in smart contracts when a function makes an external call to another contract before updating its own state. This allows the external contract, possibly malicious, … ls, using the same sta… contract.

Exampl…

```
function sp… _success) {
    ...

    uint fu… totalSupply;
    //Since
    if (p.s… ;
```

Timestamp Dependence

Description

Contracts that depend on block timestamps for critical operations are susceptible to manipulation, a miners can slightly adjust the timestamps.

Impact

This can lead to unfair advantages in games, easier puzzle solutions, and flawed randomness, all of which an attacker can exploit.

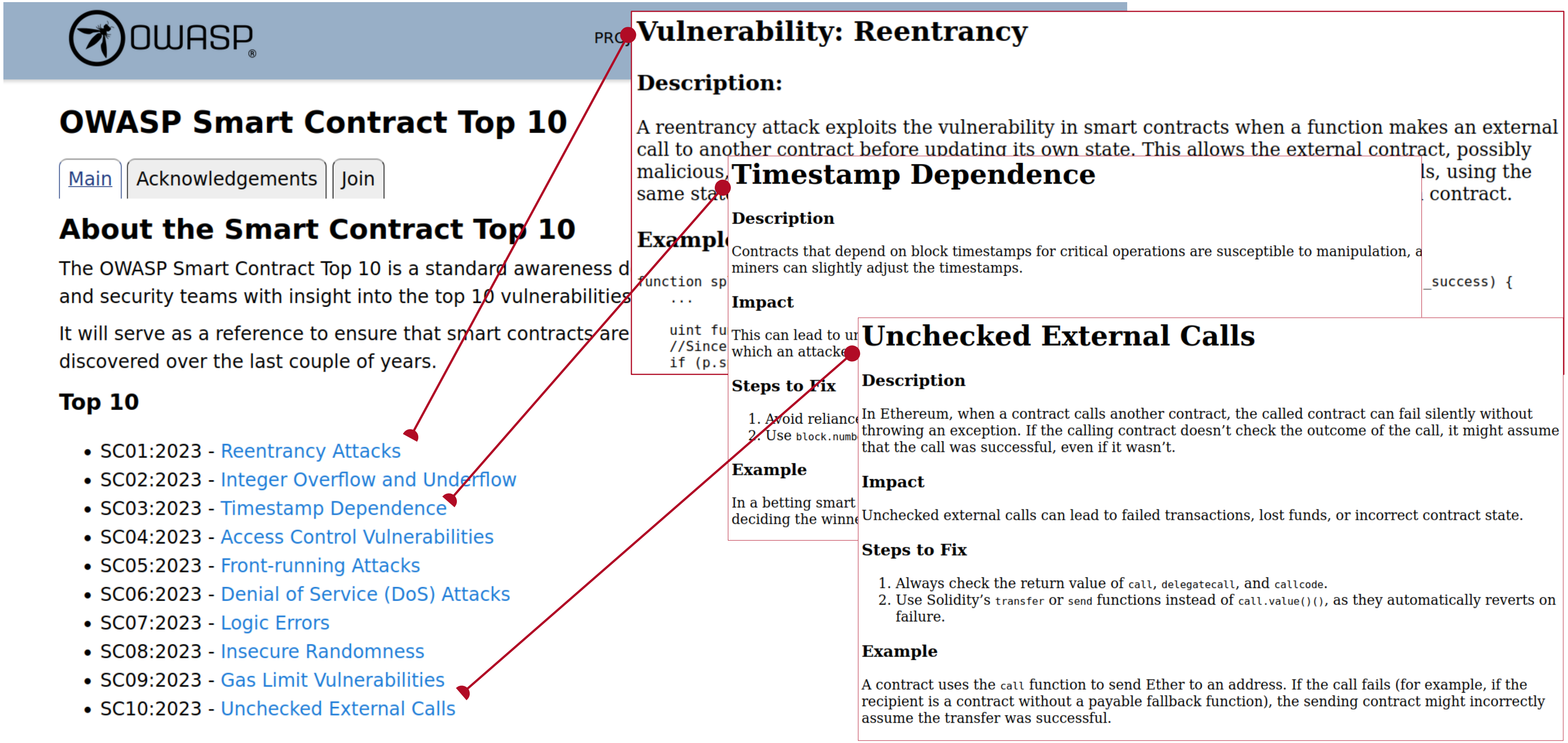
Steps to Fix

1. Avoid reliance on `block.timestamp` or `now` for crucial contract functionalities.
2. Use `block.number` for time-keeping if needed, as it is harder to manipulate.

Example

In a betting smart contract, if the outcome depends on a timestamp (like an even or odd timestamp deciding the winner), a miner could potentially manipulate the timestamp to affect the result.

OWASP top-10 (again? c'mon…)



OWASP

OWASP Smart Contract Top 10

Main | Acknowledgements | Join

About the Smart Contract Top 10

The OWASP Smart Contract Top 10 is a standard awareness d… and security teams with insight into the top 10 vulnerabilities…

It will serve as a reference to ensure that smart contracts are … discovered over the last couple of years.

Top 10

- SC01:2023 - Reentrancy Attacks
- SC02:2023 - Integer Overflow and Underflow
- SC03:2023 - Timestamp Dependence
- SC04:2023 - Access Control Vulnerabilities
- SC05:2023 - Front-running Attacks
- SC06:2023 - Denial of Service (DoS) Attacks
- SC07:2023 - Logic Errors
- SC08:2023 - Insecure Randomness
- SC09:2023 - Gas Limit Vulnerabilities
- SC10:2023 - Unchecked External Calls

Vulnerability: Reentrancy

Description:

A reentrancy attack exploits the vulnerability in smart contracts when a function makes an external call to another contract before updating its own state. This allows the external contract, possibly malicious, … ls, using the same sta… contract.

Exampl…

```
function sp… _success) {
    ...
    uint fu…
    //Since…
    if (p.s…
```

Timestamp Dependence

Description

Contracts that depend on block timestamps for critical operations are susceptible to manipulation, a… miners can slightly adjust the timestamps.

Impact

This can lead to u… which an attacke…

Steps to Fix

1. Avoid relianc…
2. Use `block.numb…`

Example

In a betting smart… deciding the winn…

Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the outcome of the call, it might assume that the call was successful, even if it wasn't.

Impact

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

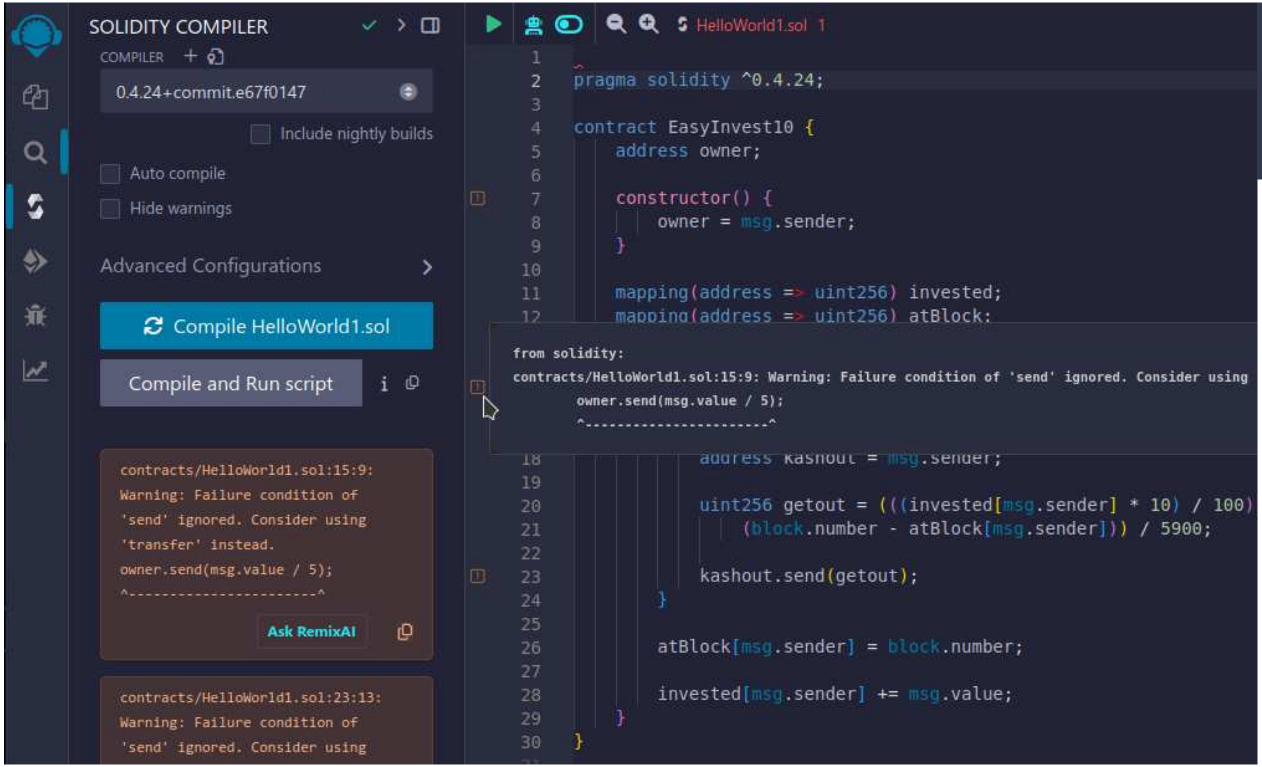
Steps to Fix

1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity's `transfer` or `send` functions instead of `call.value()()`, as they automatically reverts on failure.

Example

A contract uses the `call` function to send Ether to an address. If the call fails (for example, if the recipient is a contract without a payable fallback function), the sending contract might incorrectly assume the transfer was successful.

Navigating the web, coding…



Well, just fix'em!

Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the outcome of the call, it might assume that the call was successful, even if it wasn't.

Impact

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

Steps to Fix

1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity's `transfer` or `send` functions instead of `call.value()()`, as they automatically reverts on failure.

Example

A contract uses the `call` function to send Ether to an address. If the call fails (for example, if the recipient is a contract without a payable fallback function), the sending contract might incorrectly assume the transfer was successful.

Well, just fix’em!

Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn’t check the outcome of the call, it might assume that the call was successful, even if it wasn’t.

Impact

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

Was that so hard?

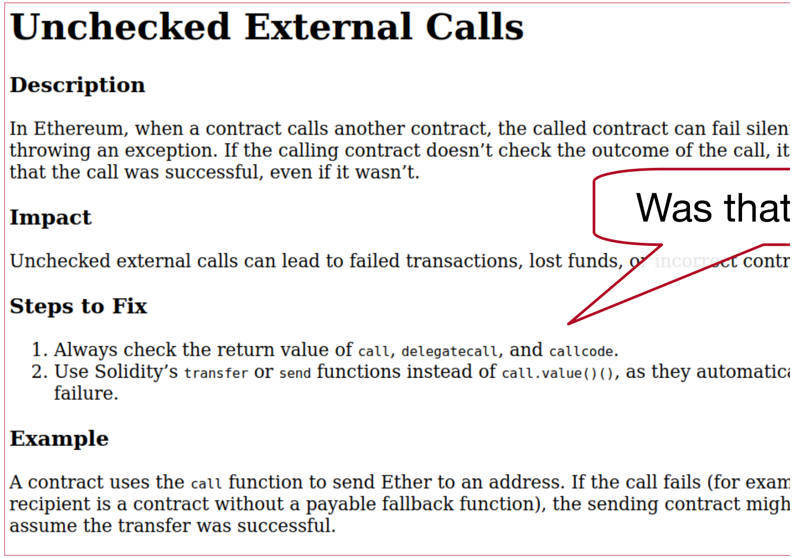
Steps to Fix

1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity’s `transfer` or `send` functions instead of `call.value()()`, as they automatically reverts on failure.

Example

A contract uses the `call` function to send Ether to an address. If the call fails (for example, if the recipient is a contract without a payable fallback function), the sending contract might incorrectly assume the transfer was successful.

Well, just fix'em!



Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silen
throwing an exception. If the calling contract doesn't check the outcome of the call, it
that the call was successful, even if it wasn't.

Impact

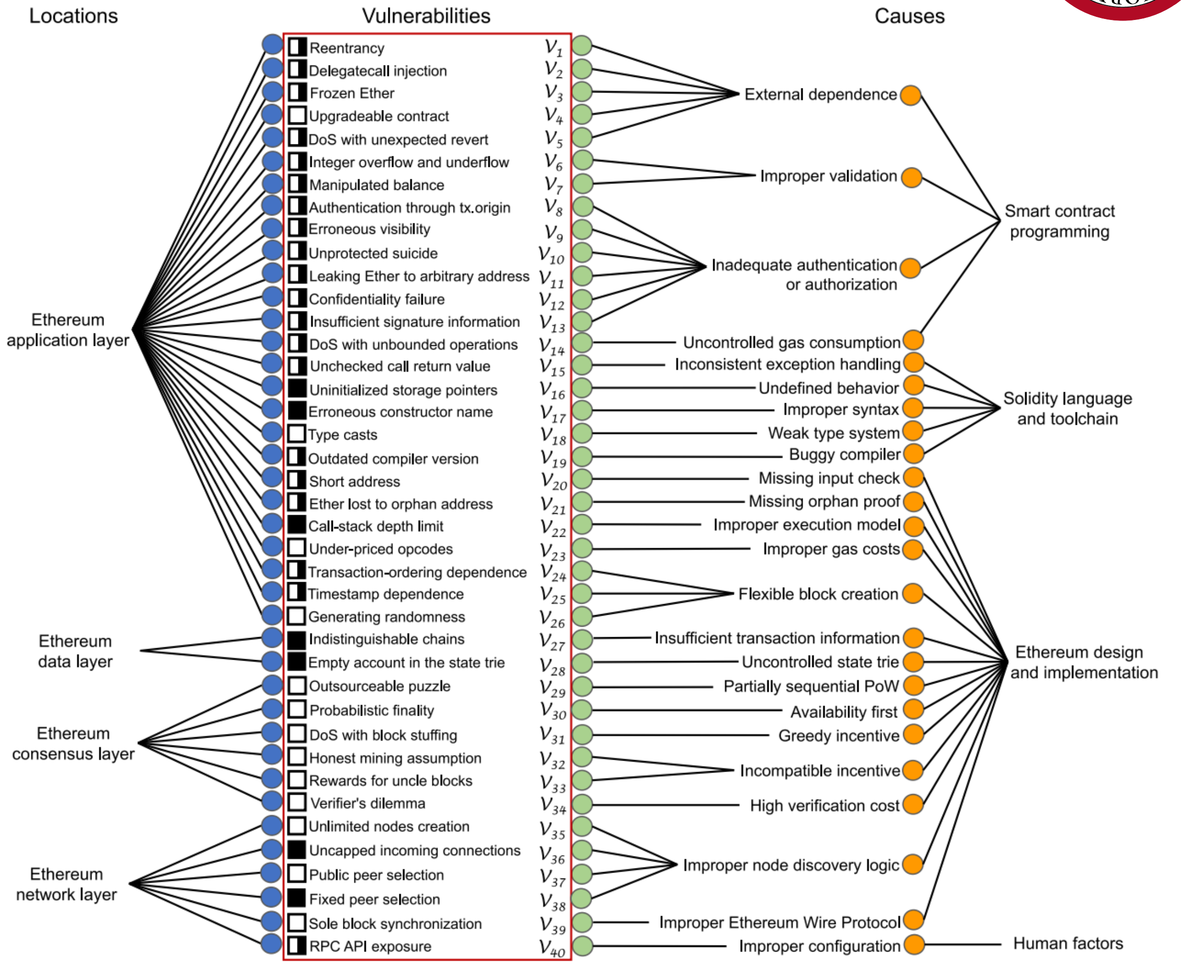
Unchecked external calls can lead to failed transactions, lost funds, or incorrect contr

Steps to Fix

1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity's `transfer` or `send` functions instead of `call.value()()`, as they automatic failure.

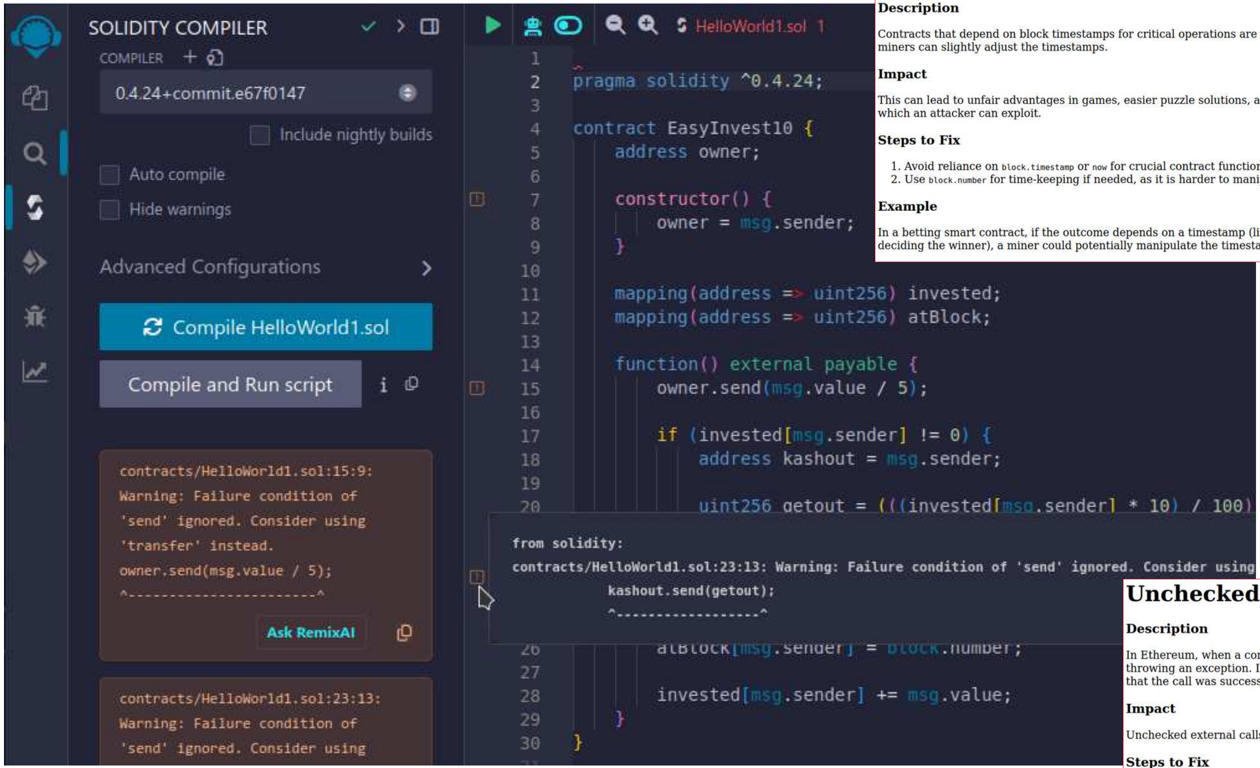
Example

A contract uses the `call` function to send Ether to an address. If the call fails (for exam
recipient is a contract without a payable fallback function), the sending contract migh
assume the transfer was successful.



Chen, Pendleton, Njilla, Xu: A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses (2020) ACM Comput. Surv.

Baby steps



Timestamp Dependence

Description

Contracts that depend on block timestamps for critical operations are susceptible to manipulation, a miners can slightly adjust the timestamps.

Impact

This can lead to unfair advantages in games, easier puzzle solutions, and flawed randomness, all of which an attacker can exploit.

Steps to Fix

1. Avoid reliance on `block.timestamp` or `now` for crucial contract functionalities.
2. Use `block.number` for time-keeping if needed, as it is harder to manipulate.

Example

In a betting smart contract, if the outcome depends on a timestamp (like an even or odd timestamp deciding the winner), a miner could potentially manipulate the timestamp to affect the result.

Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the outcome of the call, it might assume that the call was successful, even if it wasn't.

Impact

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

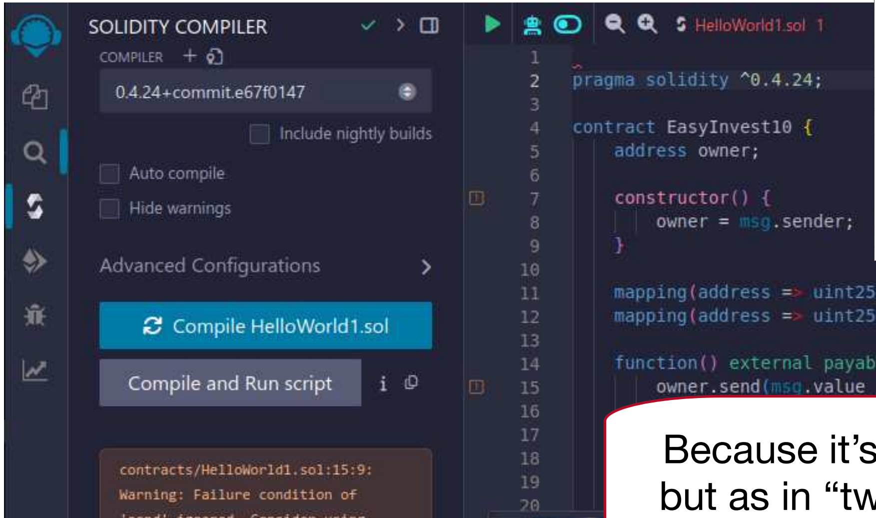
Steps to Fix

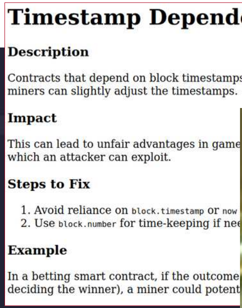
1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity's `transfer` or `send` functions instead of `call.value()()`, as they automatically reverts on failure.

Example

A contract uses the `call` function to send Ether to an address. If the call fails (for example, if the recipient is a contract without a payable fallback function), the sending contract might incorrectly assume the transfer was successful.

Baby steps





Timestamp Dependence

Description

Contracts that depend on block timestamps for critical operations are susceptible to manipulation, a miners can slightly adjust the timestamps.

Impact

This can lead to unfair advantages in game which an attacker can exploit.

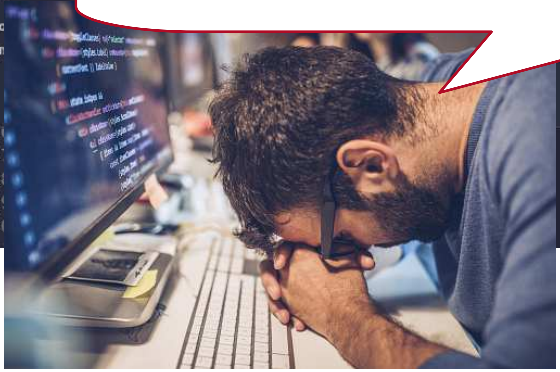
Steps to Fix

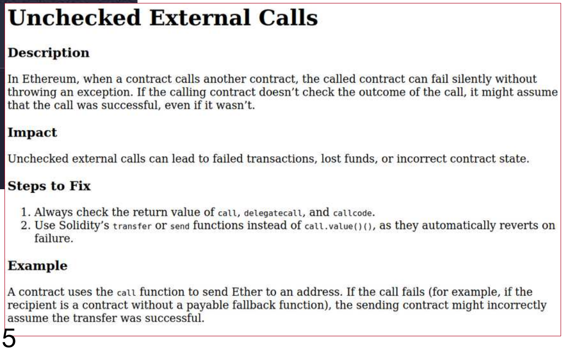
1. Avoid reliance on `block.timestamp` or `now`
2. Use `block.number` for time-keeping if nee

Example

In a betting smart contract, if the outcome deciding the winner), a miner could potent

Because it's not both as in "two", but as in "two hundred thousand"





Unchecked External Calls

Description

In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the outcome of the call, it might assume that the call was successful, even if it wasn't.

Impact

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

Steps to Fix

1. Always check the return value of `call`, `delegatecall`, and `callcode`.
2. Use Solidity's `transfer` or `send` functions instead of `call.value()()`, as they automatically reverts on failure.

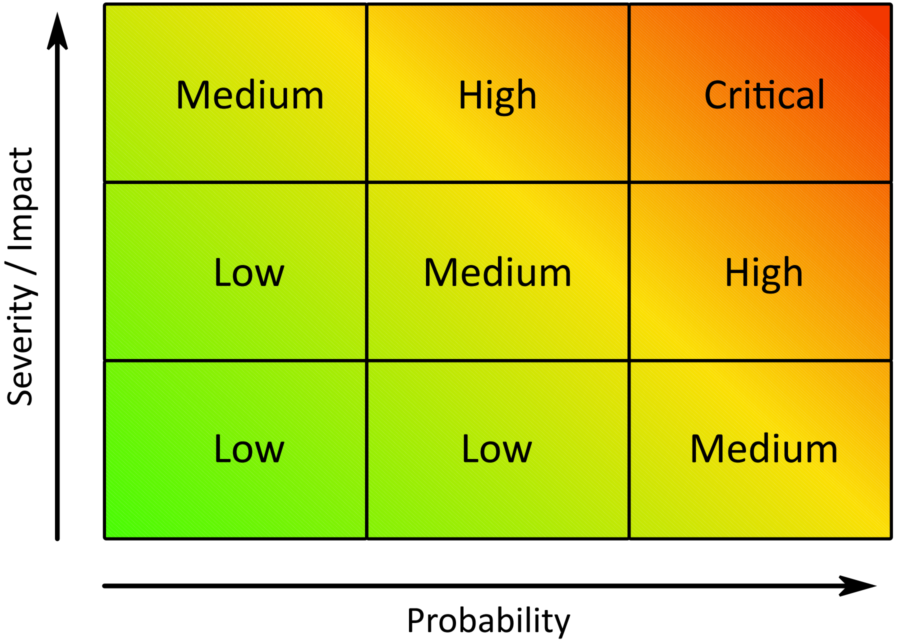
Example

A contract uses the `call` function to send Ether to an address. If the call fails (for example, if the recipient is a contract without a payable fallback function), the sending contract might incorrectly assume the transfer was successful.

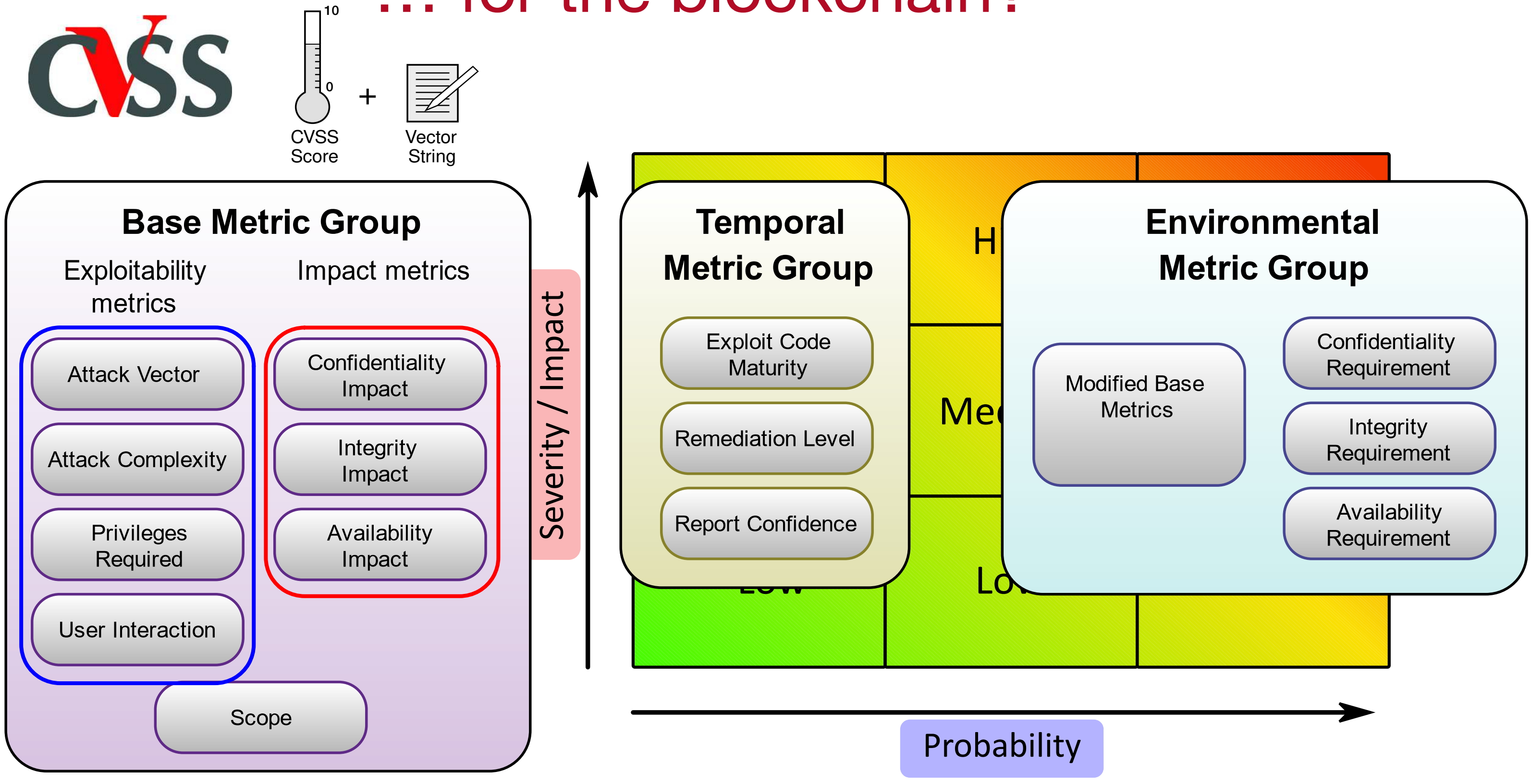
Risk quantification

$$\text{Risk} = \text{Probability} \times \text{Impact}$$

Risk assessment

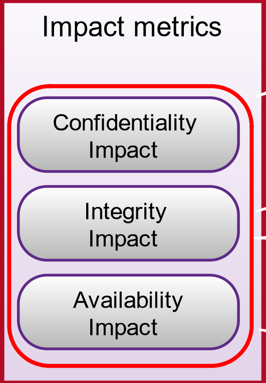


… for the blockchain?



Risk assessment: cybersecurity

… for the blockchain?



Everything's already in a public distributed ledger

(Byte) code deployed to the blockchain is unmodifiable

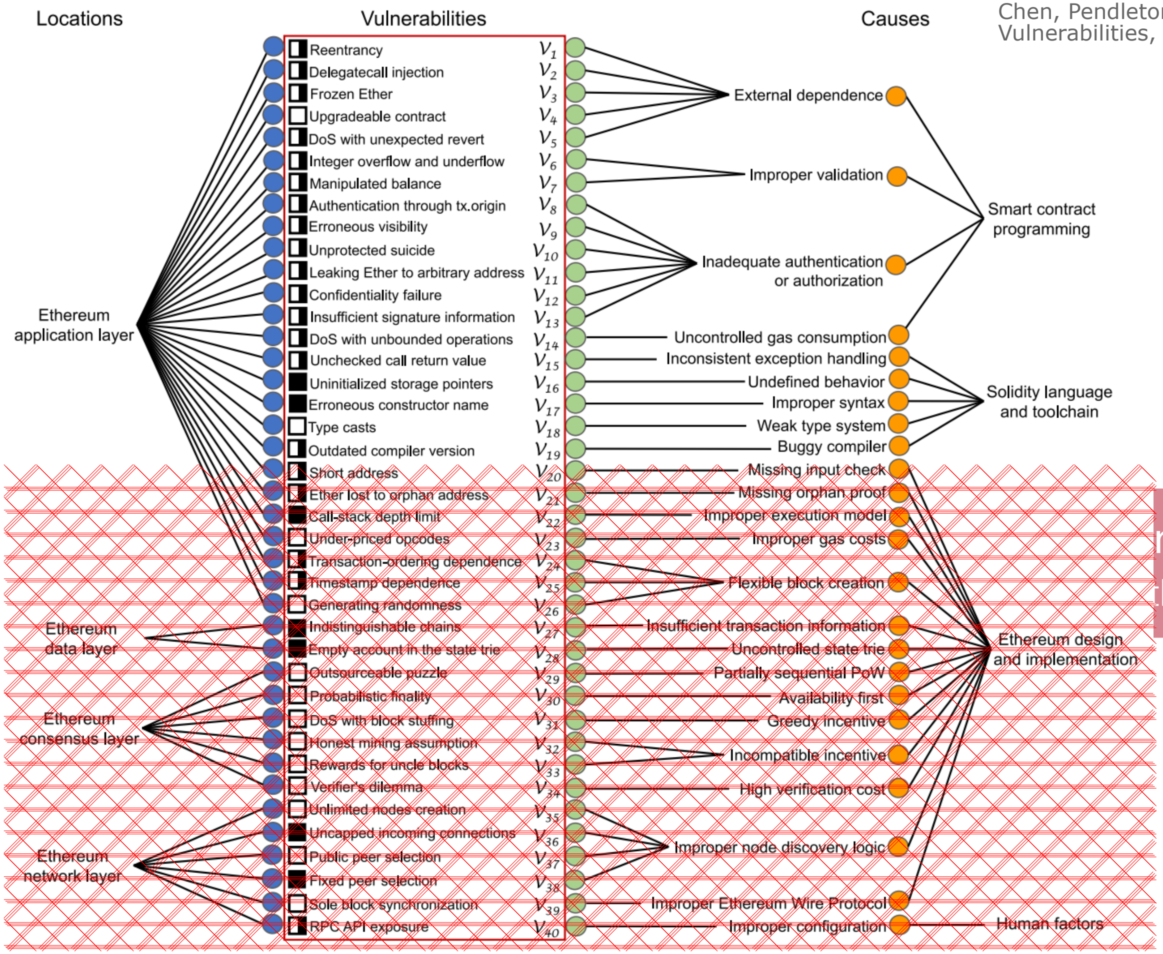
ETH/tokens in ledger are modifiable
(and modified, that's *the whole point* of crypto)

Moneyz also in ledger

… fine, this works: DoS kill smart contracts
E.g. infinite loops, blocking conditions like depleted accounts

The whole point of this presentation

Chen, Pendleton, Njilla, Xu: A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses (2020) ACM Comput. Surv.

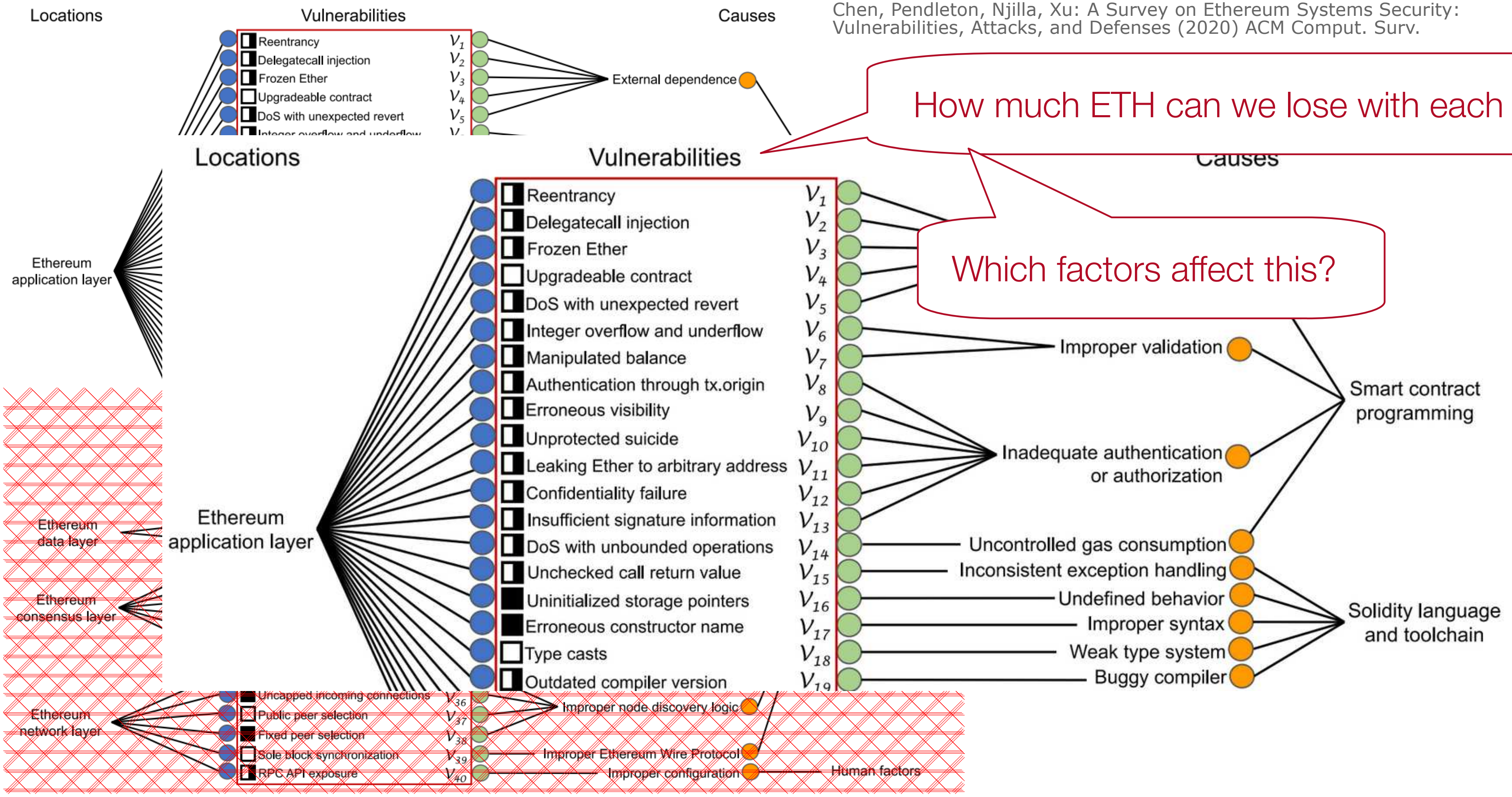


n ledger are modifiable

hat's *the whole point* of crypto)

The whole point of this presentation

Chen, Pendleton, Njilla, Xu: A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses (2020) ACM Comput. Surv.



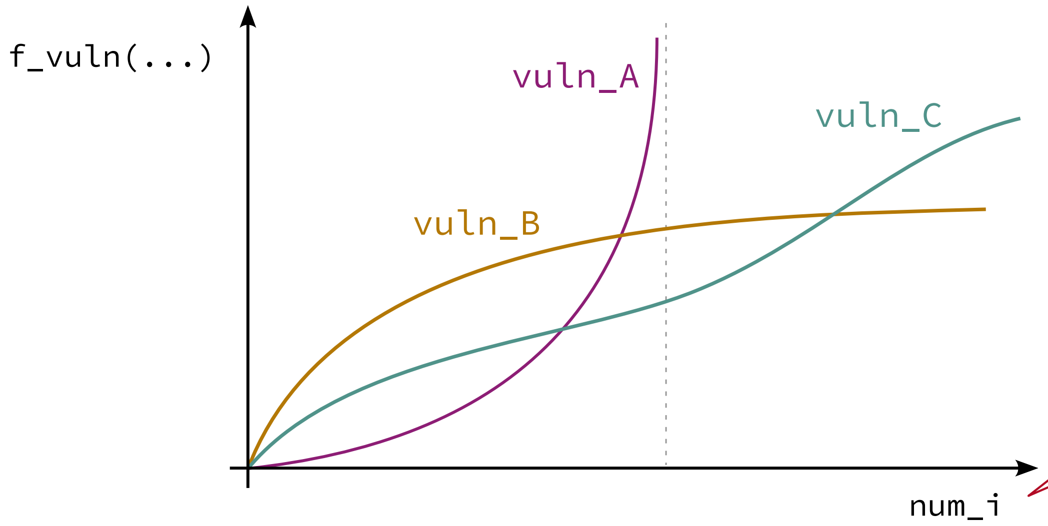
How much ETH can we lose with each vuln?

Which factors affect this?

Vulnerability impact quantification

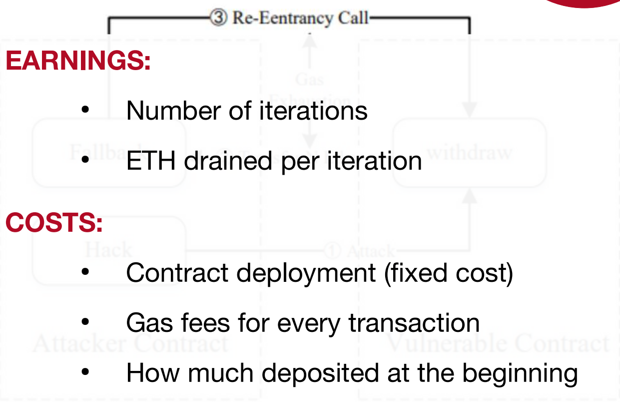
```
ETH stole = f_vuln(...)

f_vuln_A(num_i, param2, ... ) = num_i + param2 × param3 − param4
f_vuln_B(num_i, param2, ... ) = num_i² + param2
f_vuln_C(num_i, param2, ... ) = num_i ⋆ Unif(0,param2)
```



Number of iterations of the call, or transactions containing the exploit, or …

Vulnerability impact quantification: REEntrancy



EARNINGS:

- Number of iterations
- ETH drained per iteration

COSTS:

- Contract deployment (fixed cost)
- Gas fees for every transaction
- How much deposited at the beginning

Number of calls of reentrant function

ETH lost per call

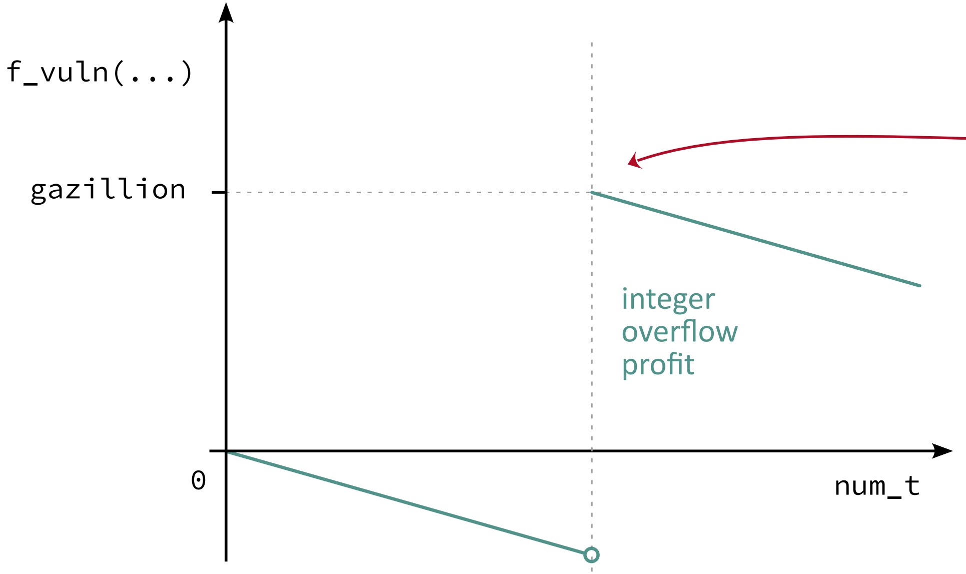
$$\text{f\_REE\_a}(\text{num\_i}, \text{ETH\_i}, \text{inital\_deposit}, \text{token\_cost}) = \text{num\_i} \times \text{ETH\_i} - \text{initial\_deposit} - \text{CONTR\_COST}$$

$$\text{f\_REE\_b}(\ldots) = \text{num\_i} \times \text{initial\_deposit} - \text{initial\_deposit} - \text{CONTR\_COST}$$

$$\text{f\_REE\_c}(\ldots) = \Sigma_{i=1}^{\text{num\_i}} \text{ETH\_i}^{\text{rate}} - \text{initial\_deposit} - \text{CONTR\_COST}$$

Rate of exchange of tokes for ETH

Vulnerability impact quantification: int over- or under-flow



EARNINGS:

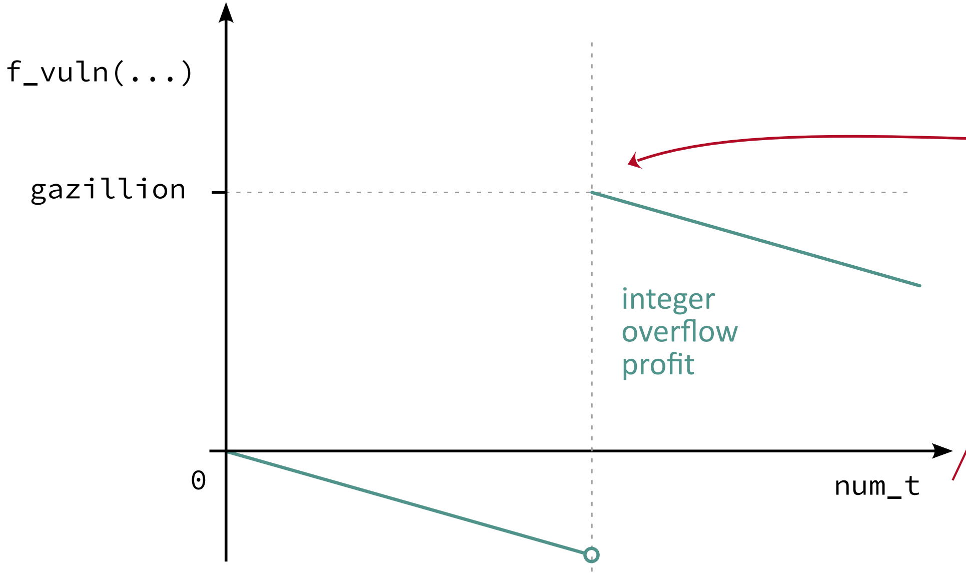
- Number of transactions in chain (not iterations of call)
- Nothing until everything: discontinuity in earnings function
- Cap by integer bit size

COSTS:

- Contract deployment (fixed cost)
- Pay (gas) transaction fees only
  vs.
  also deposit of ETH into account
- Gas fees for every transaction
- How much deposited at the beginning

```
f_IOF(…) = λ * #tokens - FIXED_COSTS
         + if (num_t < min_num_t) then -cost_t * num_t
                                  else gazillion
```

Vulnerability impact quantification: int over- or under-flow



Number of transactions

EARNINGS:

- Number of transactions in chain (not iterations of call)
- Nothing until everything: discontinuity in earnings function
- Cap by integer bit size

COSTS:

- Contract deployment (fixed cost)
- Pay (gas) transaction fees only vs. also deposit of ETH into account
- Gas fees for every transaction
- How much deposited at the beginning

These are visible in the chain (“non-atomic”): chance of detection increases

```
f_IOF(…) = λ * #tokens - FIXED_COSTS
         + if (num_t < min_num_t) then -cost_t * num_t
                                  else gazillion
```

Token – ETH conversion rate

Max integer representation (depends on int bit size)

Risk quantification

UNIVERSITÀ DI TRENTO

ProSVED
Projection of Security Vulnerabilities caused by Exploits in Dependencies

Towards vulnerability impact quantification in Solidity smart contracts

Preparing for security risk prediction

Carlos E. Budde & Giacomo Checchini

Security group @ Dipartimento di Ingegneria e Scienza dell'Informazione

carlosesteban.budde@unitn.it

12.06.2024