OSWAR

Open Standard Web3 Attack Reference

Security Framework for Decentralized Technologies

www.OSWAR.org

Table of Content

Table of Content

Contributors

Deddy Lavid
CEO, Co-Founder
Cyvers

Meir Dolev
CTO, Co-Founder
Cyvers

Jonatan Blum
Blockchain Researcher
Cyvers

Evangelos Deirmentzoglou
CISO
Paydock

Swetha Balla
Security Engineering Manager
Google

Robert Kugler
Head of Security & Compliance
Cresta

Pedro Barata
Sr. Cloud Security Engineer
BigID

Dom S
Security Operations Lead
Komainu

Chirag Agrawal
CEO, Founder
Web3Sec

Souhail Mssassi
CEO, Founder
ShellBoxes

Orhan Demis
CTO, Co-Founder
SolidProof

Nir Duan
CEO, Co-Founder
Sayfer

OSWAR (Open Standard Web3 Attack Reference) is an open framework that offers a systematic and actionable approach to understanding attacker behaviors, techniques, and vulnerabilities unique to Web3 technologies. It provides a clear and structured view of adversaries' tactics, techniques, and procedures in decentralized systems, such as blockchain protocols and dApps.

Objectives

1. Improve awareness and understanding of security risks in the decentralized technology landscape among developers and security professionals.
2. Create a common language for discussing and sharing information about security threats and vulnerabilities, thus enabling more effective communication and collaboration among different parties.
3. Establish best practices and guidelines for securing decentralized applications (dApps), protocols, and infrastructures, helping to build more robust and resilient systems.
4. Foster a proactive approach to security by encouraging regular assessments, monitoring, and threat modeling.
5. Facilitate the development and adoption of security tools and solutions tailored to the unique needs of the Web3 ecosystem.
6. Enhance the overall trust and confidence in decentralized technologies, which is crucial for their broader adoption and success.

Open Standard Web3 Attack Reference: Security Framework for Decentralized Technologies

FAQ

Who is OSWAR for?

OSWAR is designed for a wide range of audiences, including web3 security enthusiasts, security experts, Web3 developers, researchers, and organizations working with decentralized technologies. The framework aims to enhance the understanding of Web3-related attacks and promote secure development practices across the ecosystem.

What is the purpose of OSWAR?

The primary purpose of OSWAR is to provide a comprehensive and structured reference for Web3-related attacks and vulnerabilities. By offering detailed information about potential threats, the framework helps users adopt effective security measures, develop secure applications, and maintain a robust decentralized ecosystem.

How does OSWAR differ from the MITRE ATT&CK framework?

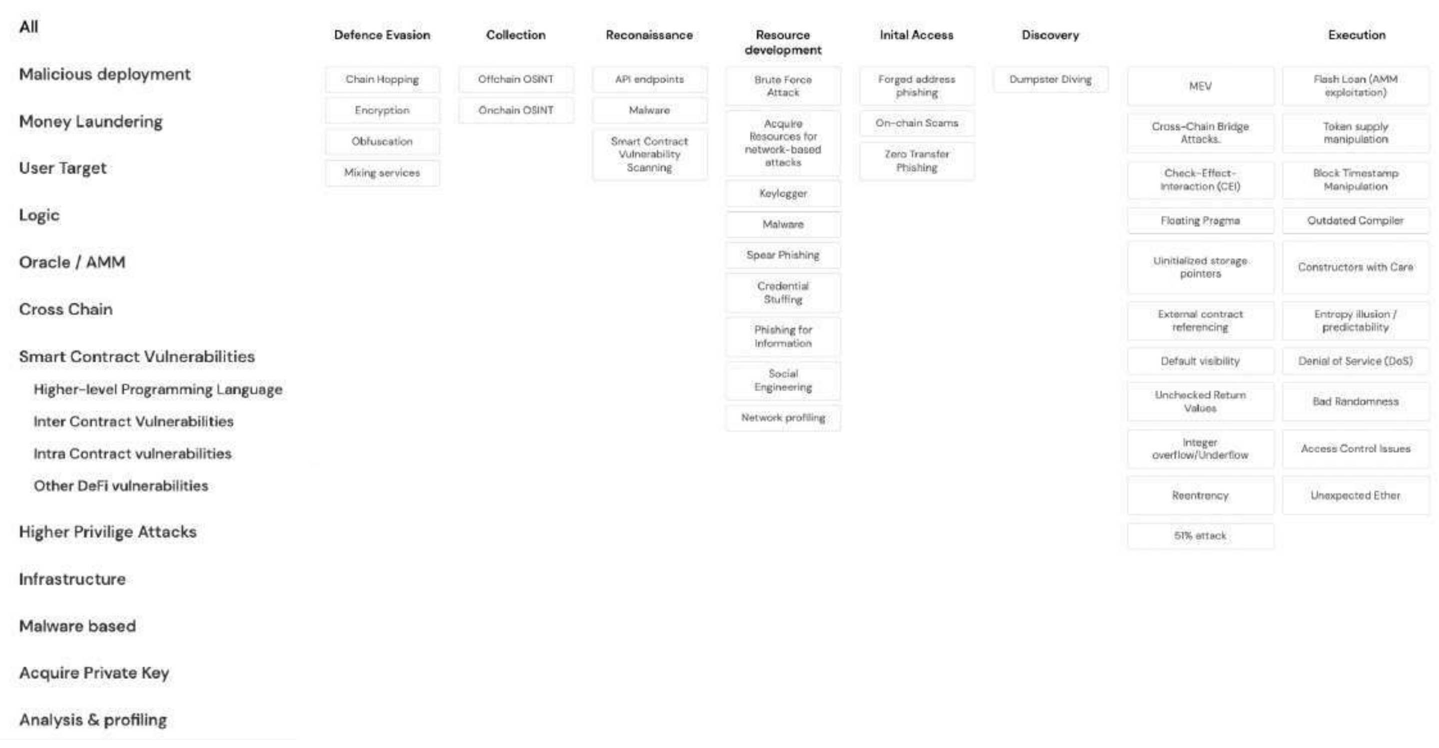
While the MITRE ATT&CK framework offers a broad perspective on cybersecurity threats and covers a wide range of technologies related to "Web2", OSWAR is specifically tailored to address the unique security challenges and attack vectors associated with Web3 technologies. OSWAR is its own unique framework and provides in-depth insights into decentralized systems, such as blockchain platforms and dApps.

How can OSWAR help Web3 developers?

OSWAR provides Web3 developers with a valuable resource to understand the various attack vectors, techniques, and vulnerabilities that can impact decentralized systems. By using OSWAR as a reference with its actionable real-world examples, developers can learn about best practices for secure development, identify potential weaknesses in their applications, and implement effective countermeasures to protect against Web3-specific threats.

How can I contribute to OSWAR?

OSWAR is an open standard, and contributions from the community are essential for its growth and development. Security experts, researchers, and developers can contribute by sharing their knowledge, reporting new attack vectors or vulnerabilities, and providing feedback on existing entries. Collaboration helps ensure that OSWAR remains up-to-date and relevant to the ever-evolving Web3 landscape.



All

Malicious deployment

Money Laundering

User Target

Logic

Oracle / AMM

Cross Chain

Smart Contract Vulnerabilities

- Higher-level Programming Language
- Inter Contract Vulnerabilities
- Intra Contract vulnerabilities
- Other DeFi vulnerabilities

Higher Privilige Attacks

Infrastructure

Malware based

Acquire Private Key

Analysis & profiling

Defence Evasion
- Chain Hopping
- Encryption
- Obfuscation
- Mixing services

Collection
- Offchain OSINT
- Onchain OSINT

Reconaissance
- API endpoints
- Malware
- Smart Contract Vulnerability Scanning

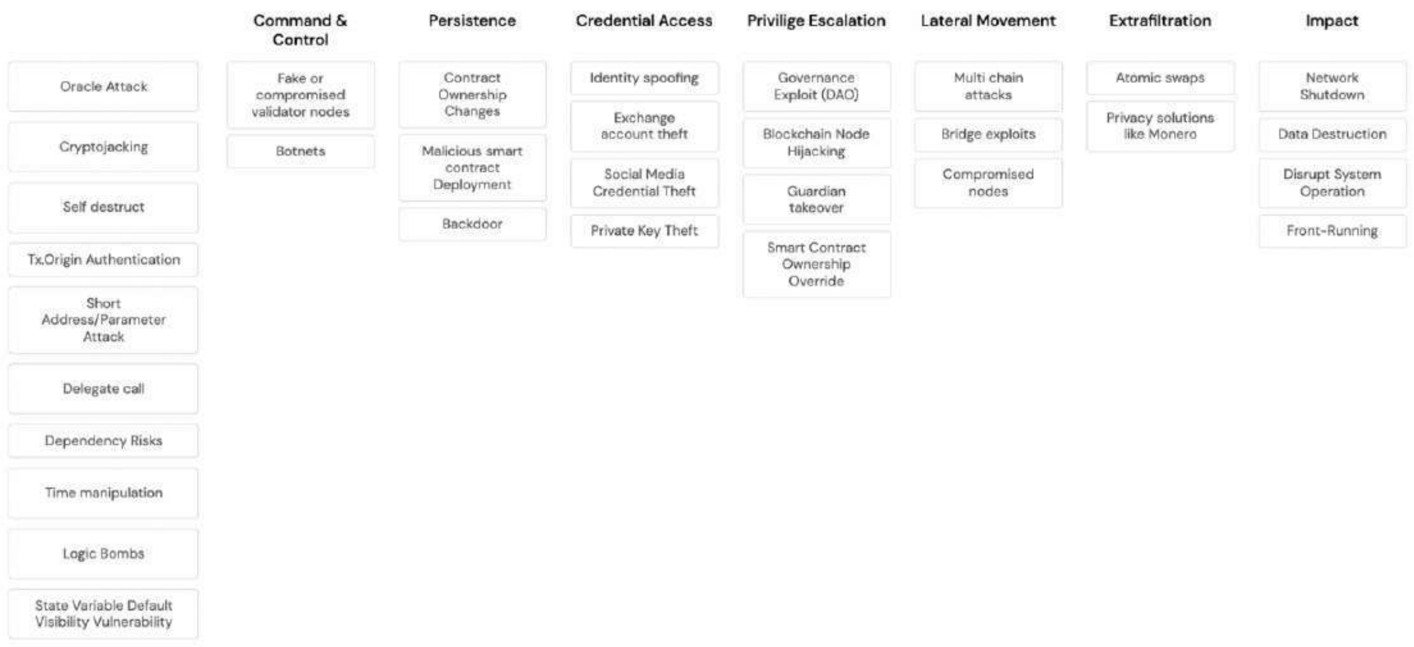
Resource development
- Brute Force Attack
- Acquire Resources for network-based attacks
- Keylogger
- Malware
- Spear Phishing
- Credential Stuffing
- Phishing for Information
- Social Engineering
- Network profiling

Inital Access
- Forged address phishing
- On-chain Scams
- Zero Transfer Phishing

Discovery
- Dumpster Diving

Execution

MEV	Flash Loan (AMM exploitation)
Cross-Chain Bridge Attacks.	Token supply manipulation
Check-Effect-Interaction (CEI)	Block Timestamp Manipulation
Floating Pragma	Outdated Compiler
Unitialized storage pointers	Constructors with Care
External contract referencing	Entropy illusion / predictability
Default visibility	Denial of Service (DoS)
Unchecked Return Values	Bad Randomness
Integer overflow/Underflow	Access Control Issues
Reentrency	Unexpected Ether
51% attack	



- Oracle Attack
- Cryptojacking
- Self destruct
- Tx.Origin Authentication
- Short Address/Parameter Attack
- Delegate call
- Dependency Risks
- Time manipulation
- Logic Bombs
- State Variable Default Visibility Vulnerability

Command & Control
- Fake or compromised validator nodes
- Botnets

Persistence
- Contract Ownership Changes
- Malicious smart contract Deployment
- Backdoor

Credential Access
- Identity spoofing
- Exchange account theft
- Social Media Credential Theft
- Private Key Theft

Privilige Escalation
- Governance Exploit (DAO)
- Blockchain Node Hijacking
- Guardian takeover
- Smart Contract Ownership Override

Lateral Movement
- Multi chain attacks
- Bridge exploits
- Compromised nodes

Extrafiltration
- Atomic swaps
- Privacy solutions like Monero

Impact
- Network Shutdown
- Data Destruction
- Disrupt System Operation
- Front-Running

Defense Evasion

What is “Defense Evasion”?

Defense evasion refers to strategies malicious parties use to stay undetected while compromising or hacking. Defense evasion methods commonly include removing or disabling security software and obscuring/encrypting data and scripts. In the Web3 world, it comprises hiding your traces by using mixing services such as Tornado Cash. To conceal and disguise themselves, the malicious parties also take advantage of trusted processes and abuse them.
Defense Evasion does NOT belong to any stage of the Web3 hack. It is a part of almost all Web3 hacks and involves:

- Hiding the connection to other wallets
- Hiding one's own identity
- Hiding the trace of funds
- Hiding where the funds will go after a hack.

Essentially, the hacker's goal is to take the profits and be able to use them while having his identity un-compromised and having the funds laundered.
A hacker will use mixer services before and after a hack is initiated.
The money laundering process takes part at the start and end of the lifecycle of an attack, but, in the end, after a successful attack, it is the most crucial step of defense evasion & money laundering.

Defense Evasion

Chain Hopping

Tag: Defense Evasion **Category: Money Laundering**

What is chain hopping?

Chain hopping is a technique used by crypto money launderers to conceal the origin and destination of illicit funds. It involves moving funds from one cryptocurrency to another, often through multiple exchanges or wallets, to obscure the transaction trail.
The basic idea behind chain hopping is to make it more difficult for investigators to trace the flow of funds. By moving funds through multiple cryptocurrencies, it becomes much harder to establish a clear line of ownership and track the final destination of the funds. One of the primary advantages of chain hopping is that it allows money launderers to take advantage of the relatively low transaction fees and high liquidity of certain assets.

Example

For example, hackers may use a low-fee cryptocurrency like Litecoin to move funds quickly and cheaply from wallet to wallet and then convert them into a more stable and widely accepted cryptocurrency like Bitcoin or Ethereum, moving funds across chains and wallets multiple times to try to hide their tracks. Additionally, it can be used to obscure the source of illicit funds. By moving funds through multiple exchanges or wallets, money launderers can make it appear that the funds come from multiple sources rather than a single source. The essence here is to move the funds as much as possible across chains so that it becomes difficult to trace.

Mitigation

Despite the apparent advantages, chain hopping is not foolproof. Investigators can still use various techniques to trace the flow of funds, including analyzing blockchain data, monitoring exchanges, and conducting traditional financial investigations. Additionally, some cryptocurrencies make chain-hopping more difficult.
While chain hopping can be an effective tool for crypto money launderers, it is not without risks. As regulators and law enforcement agencies become more sophisticated in tracking cryptocurrency transactions, money launderers must continue evolving tactics to stay ahead of the curve.

Strategies to prevent or detect it:

- Monitoring network traffic: Monitoring network traffic for connections to known cryptocurrency exchanges and mixers can help to identify when a user is switching between different blockchain networks or cryptocurrencies.

- Identifying transaction patterns: Analyzing transaction patterns on different blockchain networks can help to identify when a user is hopping between different networks or currencies. This can include monitoring for changes in transaction volume or frequency, as well as identifying transactions that use similar addresses or follow similar patterns.
- Analysis of blockchain data: Analyzing blockchain data can help identify activity patterns indicative of chain hopping. This can include monitoring for large transfers of funds between different blockchain networks, as well as analyzing the addresses and transaction histories of known chain hoppers.
- Collaboration with law enforcement: Working with law enforcement agencies can help to identify and track down chain hoppers. This can include sharing intelligence on the latest chain-hopping techniques and collaborating on investigations.

Detecting chain hopping involves combining technical controls, analysis of blockchain data, and collaboration with law enforcement. By implementing these strategies, organizations can reduce the risk of fraud and other security incidents caused by chain hopping.

Encryption

Tag: Defense Evasion **Category: Money Laundering**

What is "Encryption"?

Encryption is the process of converting information into code to make it unreadable to unauthorized users. For example, it can help protect sensitive data during transmission or storage using a cryptographic key to transform plain text into ciphertext.
Attackers can use encryption techniques to encrypt their communications and data. This can make it difficult for defenders to intercept and understand the attacker's communications. For example, attackers may use encryption to hide their IP addresses or the location of their command-and-control servers.

Example

Encryption is typically associated with mixer services and other privacy protocols and networks. For instance, if a hacker has stolen assets on the Ethereum network, their every transaction can and will be traced. They may use protocols like atomic swaps or non-KYC exchanges to swap the stolen assets into Monero. By using Monero, one of the most well-known privacy and encryption-focused cryptocurrencies, they can send transactions to other wallets and thereby cover their own tracks.

Mitigation:

Using Monero or other privacy-focused cryptocurrencies and encryption services can make it difficult to trace the flow of funds or communications during a hack. However, several mitigation strategies can be used to reduce the risk of these techniques being used to hide tracks:

- Monitor for unusual network traffic: It's important to monitor network traffic for any unusual or suspicious activity, such as large amounts of encrypted traffic or traffic to known cryptocurrency exchanges or mixers. This can help identify potential hacks or data exfiltration attempts.
- Collaboration with law enforcement: Working with law enforcement agencies can help track down and apprehend attackers who use Monero or other encryption services to hide their tracks. This can include sharing intelligence on the latest hacking techniques and collaborating on investigations. Overall, mitigation strategies for using Monero or other encryption services in a hack involve a combination of technical controls, user education, and collaboration with law enforcement. By implementing these strategies, organizations can reduce the risk of data breaches and other security incidents caused by these techniques.

Obfuscation

Tag: Defense Evasion **Category: Money Laundering**

What is "Obfuscation"?

Obfuscation is a technique attackers use to conceal their malicious code or actions. The goal is to make it difficult for defenders to detect and block the attack. The term "obfuscation" comes from the Latin word "obfuscare," which means "to darken" or "to make obscure".

Attackers can use obfuscation techniques to hide their malicious code or actions, making it difficult for defenders to detect and block the attack. For example, code obfuscation techniques can hide malware or malicious smart contracts code in the smart contracts. However, it is not "foolproof" and can be exposed.

Usually, obfuscation is meant to make the code or data difficult to understand or decipher, thereby making it harder for someone to identify and remove malicious code or contracts. Code compression, renaming variables and functions, and adding dummy code. However, these techniques can be reverse-engineered, and advanced malware detection tools can still identify malicious code even if it is obfuscated. In summary, while obfuscation can make detecting and removing malicious code or contracts harder, it is not a foolproof technique. As such, it is still important to have strong security measures and practices to protect against malware and other malicious activity on blockchains and contracts.

Example

- The attacker can use code obfuscation techniques to make the malicious code harder to understand and detect. This can involve renaming variables, using different encoding techniques, and inserting extraneous code to make it more difficult for an analyst to identify the malicious code. In general, it is done to make the code harder to interpret.
- Storing malicious code off-chain: The attacker can store the malicious code off-chain and only include a small piece of code in the smart contract that interacts with the off-chain code. This can make it harder to detect malicious code because it is not all contained in the smart contract.
- Using a multi-contract architecture: The attacker can use a multi-contract architecture to hide the malicious code in a separate contract that is not easily accessible or visible to outsiders. This can make it harder to detect the malicious code because it is not all contained in one place.

Mitigation:

To mitigate obfuscation, defenders can use various techniques, including:

- Code analysis tools: Defenders can use tools that analyze code and detect obfuscation techniques. These tools can help identify hidden or obfuscated code and enable defenders to remove it.
- Whitelisting: Defenders can use whitelisting only to allow approved programs to run on a system. This can prevent attackers from running obfuscated code on a system.
- Regular updates: Defenders should regularly update their software and systems to ensure that they have the latest security patches. This can help prevent attackers from exploiting vulnerabilities that may be present in older versions of software or systems.

Mixing services

Tag: Defense Evasion

Category: Money Laundering

What are mixing services?

Mixer services are tools used by attackers, such as Tornado Cash, to conceal their transactions and make it difficult to trace their actions on the blockchain. This can pose a challenge for defenders in tracking and blocking the attack. Mixer services are a common example of defense evasion techniques used in the Web3 world to hide the flow of cryptocurrency transactions. They are also known as tumblers or coin mixers, designed to help users obscure the origins and destinations of their transactions.
Mixer services work by receiving cryptocurrency from a user, mixing it with other coins in their pool, and returning it to the user in a way that makes it difficult to trace the original transaction.

Example

One popular mixer service in the Web3 world is Tornado Cash. It provides high anonymity to users who want to protect their transactions. Tornado Cash is an Ethereum-based mixing service that uses smart contracts to break the transaction link between the original and new addresses. The smart contracts hold a pool of ETH, which users can deposit into using their wallets. Once the funds are deposited, the smart contracts mix them with other deposits and return them to the user's new address in the form of a new ETH amount with a different history. Before an attack, Tornado Cash is used by Web3 users who want to hide their cryptocurrency transactions from being tracked or traced by other users or even government authorities. Using Tornado Cash, these users can protect their privacy and hide their financial activities from others. After an attack, hackers or malicious actors can use Tornado Cash to obscure their financial transactions and prevent investigators from tracking their movements.

Attackers can use Tornado Cash or similar mixing services to mix stolen funds with other coins, making them difficult or impossible to trace. This makes it harder for law enforcement or investigators to identify and recover the stolen funds. However, it should be noted that the use of mixer services is not illegal, and many legitimate users may use them for privacy reasons. Only when the services are used for illegal activities do they become problematic.

Mitigation:

As of 2022, some nodes have started to block processing transactions of wallets blacklisted by OFAC. Wallets that have interacted with the mixer can be blacklisted, preventing them from using the services in the future. However, currently, it is almost impossible to prevent the use of mixing services as they are open-source software.

Collection

What is “Collection”?

The "Collection" phase involves gathering information about the targeted smart contract or decentralized application (dApp), such as the contract logic, dependencies, and user behaviors. This information is then used to refine the attack plan and increase the chances of success. It is important to note that the Collection phase is not intended to detect vulnerabilities, as that falls under the "Reconnaissance" phase.

In addition to legitimate information gathering, "Collection" also refers to the malicious collection and aggregation of sensitive data from various sources, including blockchain transactions, smart contract interactions, and user activity on dApps.

During the collection stage, attackers seek to gather the information that is already public or otherwise accessible without unauthorized access. For example, data can be collected by analyzing public blockchain transactions or scraping information from websites that publicly display Web3 application data. "Collection" and "Reconnaissance" refer to different stages in the cyberattack lifecycle within the Web3 framework.

"Collection" tactics employed by malicious actors can include the aggregation of data obtained from multiple sources, and the sources themselves can include individuals, organizations, and decentralized applications (dApps). At its core, the "collection" stage is about obtaining data and utilizing various techniques to gather information, which can then be used for a variety of purposes, such as planning attacks, targeted phishing, or gaining unauthorized access to systems.

Collection

Off-chain OSINT

Tag: Collection

Category: Analysis & profiling

What is “Off-chain OSINT”?

Off-chain OSINT (Open-Source Intelligence) is a category in the OSWAR framework that refers to the process of gathering publicly available information from off-chain sources to analyze and identify potential security threats or vulnerabilities in the Web3 ecosystem. This involves collecting and analyzing information from various sources, such as social media, forums, blogs, news articles, and developer repositories, to gain insight into potential attack vectors, security issues, or vulnerabilities related to Web3 applications and infrastructure. This information can be valuable for attackers to identify weak points.

Example:

An attacker might use off-chain OSINT to monitor discussions on developer forums or social media channels to identify potential vulnerabilities in a popular DeFi protocol. They could come across a developer mentioning a possible exploit in the smart contract code that has not been patched yet. The researcher could then use this information to analyze the vulnerability and recommend appropriate mitigations to the protocol's team or the wider community.

Mitigation

To mitigate the risks associated with off-chain OSINT, several steps can be taken:

1. Be cautious about sharing sensitive information: Developers, team members, and users should be mindful of the information they share on public platforms. Revealing too much information about a project's security mechanisms, known vulnerabilities, or internal processes can expose the project to potential attacks.
2. Monitor public discussions: Actively monitor public forums, social media channels, and other online platforms where your project or technology is being discussed. This can help you identify potential security issues, vulnerabilities, or attack vectors before exploitation.
3. Implement secure coding practices: Ensure that your smart contracts and other code are developed using secure coding practices, such as adhering to established security guidelines, performing regular code reviews, and using automated testing tools to identify and fix vulnerabilities.
4. Establish a vulnerability disclosure program: Encourage responsible disclosure of security vulnerabilities by setting up a clear process for reporting and addressing potential issues. This can help ensure that vulnerabilities are addressed in a timely manner before they can be exploited.

5. Educate your team and community: Provide security awareness training to your team and educate your user community on best practices for protecting their assets and interactions with your project. This can help reduce the risk of social engineering attacks and other security issues arising from off-chain OSINT.

On-chain OSINT

Tag: Collection

Category: Analysis & profiling

What is On-chain OSINT?

On-chain OSINT stands for "Open-Source Intelligence," which involves collecting and analyzing data from open sources, both covert and publicly available.
Since blockchain data, such as transactions, is publicly available, attackers can use blockchain analysis tools to trace transactions, identify wallet addresses, and uncover other sensitive data related to blockchain users. This technique is often used in the collection phase of a web3 hack. By using publicly available blockchain analysis tools, hackers can go through a lot of data to identify a protocol, DApp, or target in general.

Example

Attackers can use blockchain analysis to track transactions and identify the parties involved, including wallet addresses and other sensitive data. This information can be used to exploit vulnerabilities in the system, launch phishing attacks, or steal cryptocurrency. However, it can also be as simple as examining certain protocols' Total Value Locked (TVL) to identify a target further.
An attacker might use blockchain analysis to identify the owners of substantial amounts of cryptocurrency and then use social engineering techniques to trick them into revealing their private keys or other sensitive information. Additionally, attackers can collect information on dApps, such as their smart contract code, user data, transaction history, volume, and more.

Mitigation

Unfortunately, it is not possible to prevent on-chain OSINT entirely because blockchain technology is designed to be transparent and decentralized. However, some measures can be taken to reduce the risk of attacks, such as using privacy-enhancing technologies like mixers or tumblers to obfuscate transactions and prevent tracking. Additionally, users can take steps to protect their private keys and other sensitive information, such as using hardware wallets or secure storage solutions.

Reconaissance

What is “Reconnaissance"?

Reconnaissance involves gathering information about the smart contracts or decentralized applications (dApps) being targeted, including the contract address, ABI, and API endpoints. This information is then used to identify potential vulnerabilities that can be exploited during an attack.
During this stage, the attacker actively probes and scans the target's systems or network to identify vulnerabilities and gather information for a potential attack. Port scanning tools may be used to identify open ports on the target's system or network. DNS reconnaissance may be performed to gather information on the target's domain names and associated IP addresses.
The techniques used by malicious parties to gather data that can be utilized to help target an organization are referred to as "reconnaissance." Surveillance and gathering details on the target organization's infrastructure, personnel, or staff may be included. The adversary can then use this knowledge to its advantage in various stages of the adversary lifecycle, such as understanding how the organization maintains its operations and its current security procedures.
The malicious party would be planning and carrying out Initial Access to gain access to the internal network and defining and prioritizing post-compromise goals to determine what objectives it wants to achieve.

Reconaissance

API endpoints

Tag: Reconnaissance

Category: Infrastructure

What are API endpoints?

API endpoints are a key target for reconnaissance, as they can provide valuable information about the dApp's functionality and underlying blockchain network. An API endpoint is a URL that can be accessed to interact with a specific component of the dApp, such as a smart contract or a node on the blockchain network.

Example

An attacker may use an API endpoint to gather information about a smart contract's functions and input parameters, which could reveal vulnerabilities that can be exploited to manipulate or steal assets from the contract.

Mitigation

To mitigate the risk of reconnaissance attacks, dApp developers should take several steps.

- First, they should ensure that sensitive information is not exposed through API endpoints, such as private keys or other authentication credentials. Developers should also ensure the API keys are secure.
- Developers can also implement rate limiting and IP blocking to prevent automated reconnaissance attacks.
- Additionally, developers can use obfuscation techniques to make it more difficult for attackers to extract information from API endpoints, such as using random identifiers for function names or input parameters.
- Finally, developers should regularly audit their dApp and blockchain networks for potential vulnerabilities and implement patches to stay ahead of attackers.

Malware

Tag: Reconnaissance **Category: Malware based**

What is Malware?

“Malware” refers to any software or code designed to gain unauthorized access to an organization's systems or steal sensitive information. In the world of Web3, blockchains, and crypto, malware can be particularly dangerous because it can be used to steal private keys, wallet addresses, and other resources that can be used to support an attack. It is worth noting that when a hacker successfully installs malware on a target's computer, the Web3 hack has not yet begun. Instead, it is still considered a traditional "Web2" hack aimed at achieving goals that will facilitate a Web3 hack, such as acquiring the private key of a wallet.

Example

One common example of malware in the Web3 context is a keylogger. This type of malware records every keystroke on a device, including passwords, private keys, and other sensitive information. Once captured, the attacker can use this information to access a victim's accounts or wallets. Another type of malware commonly used in Web3 attacks is ransomware. This malware encrypts a victim's files or systems, making them inaccessible. The attacker then demands a ransom payment in exchange for the decryption key. In the context of Web3, ransomware can be used to encrypt a victim's wallets or blockchain nodes, making them inaccessible and forcing the victim to pay the ransom to regain access.

It is worth noting that in many cases, malware is deployed on a target's computer through phishing. A prime example is the Lazarus Group, which ran a fraudulent job advertisement scheme. They posted job openings on sites like LinkedIn and told people who were interested in the job to download a PDF file that contained an executable file inside. This malware enabled Lazarus operatives to exploit vulnerabilities in the victim's system, stealing sensitive data from employees at existing crypto companies.

Mitigation

To mitigate the risk of malware attacks, Web3 users should use antivirus and anti-malware software to detect and remove any malicious software from their devices. It is also important to keep software and systems up-to-date with the latest security patches to prevent attackers from exploiting known vulnerabilities.

Web3 users should also be cautious when downloading and installing software or apps and should only use trusted sources. It is important to verify the authenticity and security of any software, link, or app before downloading it. Lastly, Web3 users should implement strong security measures, such as using two-factor authentication, encrypting sensitive information, and limiting access to resources, to reduce the impact of malware attacks.

Smart Contract Scanning

Tag: Reconnaissance **Category: Analysis & profiling**

What is “smart contract scanning”?

Smart contract scanning involves analyzing the open-source code of smart contracts, which are self-executing contracts with the terms of the agreement directly written into lines of code on a blockchain, to identify any potential security vulnerabilities.

Example

A real-world example of why smart contract scanning is important is the DAO (Decentralized Autonomous Organization) hack in 2016. The DAO was a decentralized venture capital fund that raised over $150 million worth of ether, a cryptocurrency used on the Ethereum blockchain. However, a DAO's smart contract code flaw was exploited, allowing an attacker to drain approximately $50 million worth of ether. The hack resulted in a hard fork of the Ethereum blockchain, where a new version was created to reverse the transactions and return the stolen funds.

This is not the only example. All hacks occur due to a vulnerability of the smart contracts that have, at some point, been scanned and identified by the hacker.

Mitigation

It is not possible to prevent the “scanning” of blockchain smart contracts which are open source.

However, one should perform smart contract audits and bug bounties to mitigate the risk of being hacked. This involves using tools and techniques to analyze the smart contract code for any weaknesses or vulnerabilities that attackers could exploit. The goal is to identify potential issues before malicious actors can exploit them.

In addition to audits, dApps should implement proactive real-time monitoring to scan for malicious activity. This involves AI-based & Machine learning monitoring solutions that scan smart contracts and entire blockchain networks to identify malicious smart contracts being deployed.

Resource development

What is “Resource Development”?

Resource development involves creating or acquiring the necessary tools and resources for an attack, such as exploit code (malicious smart contracts), phishing scams, or social engineering tactics to move to the next attack phase.

The term "Resource Development" refers to strategies in which malicious actors create, acquire, or steal resources that can be used to aid in the hacking process. These resources can support various stages of the attack life cycle. Resource development involves identifying, gathering, or creating tools, techniques, and infrastructure necessary for an attack.

Some specific tools that can be used to carry out these tactics include:

- Metasploit: an open-source framework that can create and test exploits on a target's computer or network.
- Maltego: a tool used for data mining and information gathering. It can collect information on a target's Web3 wallets or other cryptocurrency-related accounts.
- Keylogger Pro: commercial keylogging software that can record keystrokes on a target's computer.
- Burp Suite: a web application security testing tool to identify vulnerabilities in Web3 platforms and applications.
- Social-Engineer Toolkit (SET): an open-source tool for creating and executing social engineering attacks.

Resource development

Brute Force Attack

Tag: Resource Development **Category: Infrastructure**

What is a "Brute Force attack"?

A Brute Force attack is a hacking method where attackers use automated software or code to guess passwords, private keys, or other sensitive information. This involves systematically checking every possible combination of characters until the correct one is found. In the context of Web3, blockchains, and crypto, Brute Force attacks can be used to gain access to wallets, accounts, and other resources that can be used to support an attack. Concerning resource development, refers to creating a tool or method to carry out a Brute Force attack later. Brute Force attacks involve using automated software or code to guess passwords, private keys, or other sensitive information. In the context of Web3, blockchains, and crypto, Brute Force attacks can be used to gain access to wallets, accounts, and other resources that can be used to support an attack.

Example

A common example of Brute Force attacks in the Web3 context is a dictionary attack on a wallet or account. Dictionary attacks involve an attacker using a list of common words or phrases as passwords and then systematically checking each one until the correct password is found. In the case of Web3, an attacker may use a list of commonly used passwords or private keys to try and gain access to a wallet or account. Another example of Brute Force attacks in the Web3 context is a rainbow table attack on a hashed password. Rainbow table attacks involve precomputing the hashes of all possible character combinations and then comparing them to the hash of a target password. The attacker can use the pre-computed password to access the account or wallet if a match is found.

Mitigation

To mitigate Brute Force attacks, Web3 users should use strong passwords or passphrases that are difficult to guess. Using unique passwords for each account or wallet is also important to prevent attackers from accessing multiple resources if one password is compromised. Additionally, two-factor authentication (2FA) can add an extra layer of security to accounts and wallets, making it more difficult for attackers to gain access.

Web3 users should also keep their software and systems up-to-date with the latest security patches and use antivirus and firewall software to protect their devices from malware. Monitoring accounts and wallets regularly for any unauthorized access or suspicious activity is also essential. Lastly, blockchain and crypto projects should implement strong security measures, such as password strength requirements, rate limiting, and IP blocking, to prevent Brute Force attacks on their platforms.

Resources for network-based attacks

Tag: Resource Development **Category: Infrastructure**

What are "Resources for network-based attacks?"

"Resources for network-based attacks" refer to the tools, techniques, and strategies that attackers can use to compromise the security of a network, such as a blockchain network. These resources may include software vulnerabilities, malware, social engineering tactics, brute force attacks, denial-of-service attacks, and other similar methods.
The most commonly used consensus mechanisms within blockchain networks are Proof of Work (PoW) and Proof of Stake (PoS). For an attacker to take control of these distributed consensus networks, they must acquire enough computing power in the form of hash-rate or enough tokens within a token's circulating supply.

Example

An example of an attack that exploits network vulnerabilities is the 51% attack on a blockchain network. In this type of attack, an attacker needs to gain control of the majority of the network's computing power or tokens, enabling them to manipulate the blockchain's ledger and transactions. While any PoS or PoW-based network is theoretically vulnerable to such an attack, executing on well-established blockchains like Ethereum or Bitcoin is extremely difficult.

Mitigation

Blockchain networks need to be as decentralized as possible to prevent such attacks. The issue arises with the introduction of decentralization and decentralized consensus. Various networks have different degrees of percentage when it comes to being able to take over the network. Some have it as high as 2/3, meaning 66%. If a network has been 51% attacked, there is not much to do to prevent it. It will often just result in a network fork. There will be two chains, and then the community needs to decide which "correct" chain is.

Keylogger

Tag: Resource Development **Category: Malware based**

What is "Keylogger"?

A keylogger is malware designed to capture keystrokes on a target's computer. This can be used to steal private keys by recording the keys used to unlock wallets or access other Web3 platforms.

Keyloggers, also known as keystroke loggers or keystroke recorders, are types of malware that record every keystroke a user makes on a computer or mobile device. This can include sensitive information such as usernames, passwords, private keys, and other data that can be used to carry out cyber-attacks or steal cryptocurrency. Keyloggers can be either hardware or software-based, with software keyloggers being more common in modern times. They can be installed on a device via phishing attacks, malicious downloads, or other means and can run in the background without the user's knowledge.

Example

Sources have come forth alleging that the 2022 Lastpass hack event, whereby thousands if not millions of sensitive emails and passwords were "leaked" or compromised, occurred due to a keylogger hack targeting an employee. The company lost encrypted password vault data for all customers to a hacker secretly poking around LastPass' systems for weeks.

"In Monday's update(Opens in a new window), LastPass added that only four DevOps engineers at the company possessed the necessary decryption keys through a "highly restricted set of shared folders." However, the hacker circumvented the company's security safeguards by serving malware to one of the DevOps engineers at their home.

"This was accomplished by targeting the DevOps engineer's home computer and exploiting a vulnerable third-party media software package, which enabled remote code execution capability and allowed the threat actor to implant keylogger malware," LastPass said."

Mitigation

To mitigate the risk of keyloggers, users should follow basic cybersecurity practices and implement malware detection software, avoiding suspicious downloads and links and using two-factor authentication for all accounts. Users can use anti-malware software that includes keylogger detection and removal capabilities to protect their devices from this type of malware.

Source:

https://www.pcmag.com/news/hacker-breached-lastpass-by-installing-keylogger-on-employees-home-computer

Acquiring/creating Malware

Tag: Resource Development **Category: X**

What is “Malware”?

Malware within resource development involves acquiring malware to target the infrastructure in resource development. Various forms of malware exist, and some are purchased on the darkweb. The forms of malware in this section are traditional “Web2” malware & software. Here, it is malware which is dedicated to acquire resources like the private key.

Malware is short for malicious software, which is designed to infiltrate and damage computer systems without the owner's consent or knowledge. Malware can take many forms, including viruses, worms, Trojans, ransomware, spyware, and adware. It can be spread through various means, including email attachments, infected software, compromised websites, or social engineering.

Example

One example of malware is ransomware. Ransomware is malware that encrypts the victim's data, rendering it inaccessible, and demands payment in exchange for the decryption key. This type of malware has been responsible for numerous high-profile attacks in recent years, including the WannaCry and Petya/NotPetya outbreaks.

Mitigation

Preventing malware attacks is critical for maintaining the security and integrity of computer systems. Here are some best practices for mitigating the risks associated with malware:

1. Keep software and operating systems up to date with the latest security patches and updates.
2. Install reputable antivirus and antimalware software and keep it updated.
3. Use strong and unique passwords for all accounts and enable two-factor authentication wherever possible.
4. Educate employees on recognizing and avoiding phishing scams and other social engineering tactics.
5. Regularly back up important data to an external source.
6. Monitor network traffic and system logs for signs of unusual activity.
7. Implement a least privilege policy to limit access to sensitive data and systems.
8. Conduct regular vulnerability assessments and penetration testing to identify and address security weaknesses.

By following these best practices, individuals and organizations can significantly reduce their risk of falling victim to malware attacks.

Spear Phishing

Tag: Resource Development **Category: Acquire Private Key**

What is "Spear Phishing"?

Spear phishing is a type of phishing attack that is targeted at a specific individual or group of individuals rather than a broader audience. In the context of Web3, blockchains, and crypto, spear phishing can be used to steal private keys, wallet addresses, and other resources that can be used to support an attack.

Spear phishing attacks typically involve crafting convincing emails or messages that appear to come from a trusted source, such as a colleague, friend, or family member. The emails may contain requests for sensitive information, or they may include links to malicious websites or downloads that can be used to steal information or gain access to a target's computer or network.

Example

A common example of a spear phishing attack in the web3 context is when an attacker sends a personalized email to a target claiming to be a member of a blockchain project or an investor in a cryptocurrency. The email may contain information specific to the target, such as their name or recent activity on the blockchain. The email may also include a request for the target to click on a link or download a file that appears to be legitimate but is malicious. Once the target clicks on the link or downloads the file, the attacker can use it to steal private keys, wallet addresses, and other sensitive information.

Another example of spear phishing in the web3 context is when an attacker creates a fake social media account and contacts a target with a message that appears to come from a friend or colleague. The message may contain a link to a malicious website or download that can be used to steal sensitive information.

Mitigation

To mitigate spear phishing attacks, web3 users should be cautious when receiving emails or messages from unknown or untrusted sources. Users should verify the authenticity of any requests for sensitive information before responding or providing any information. It is also important to use strong passwords and two-factor authentication to protect accounts and wallets from unauthorized access.

Web3 users should also be aware of the latest phishing techniques. They should keep their software and systems up-to-date with the latest security patches to prevent attackers from exploiting known vulnerabilities.

Credential Stuffing

Tag: Resource Development

Category: Acquire Private Key

What is "Credential Stuffing"?

Credential stuffing is a technique cybercriminals use to gain unauthorized access to a target's account by using stolen login credentials. This can be used to steal private keys by accessing the target's Web3 wallet or other cryptocurrency-related accounts. Cybercriminals exploit the vulnerability of reused or weak passwords across different accounts to execute this type of cyberattack. Credential stuffing attacks involve automated attempts to log into a target's account using combinations of usernames and passwords obtained from data breaches and other sources. Resource development involves acquiring the credential stuffing tool to carry out the attack.

Credential stuffing is a popular technique among cybercriminals because it requires minimal effort to execute and can lead to significant financial gain.

In the context of Web3 security, credential stuffing can be used to access a target's Web3 wallet or other cryptocurrency-related accounts, enabling the attacker to steal private keys and access funds.

Example

In 2019, a hacker group compromised 4.9 million Capital One credit card applications by exploiting a vulnerability in the company's firewall. The attackers then used a credential-stuffing attack to access the AWS server containing the stolen data. The attack resulted in the theft of personal information, including names, addresses, credit scores, and Social Security numbers of the affected individuals.

Mitigation

Organizations can implement several measures to mitigate credential-stuffing attacks, such as strong password policies that require users to create complex passwords and enable two-factor authentication (2FA). Other best practices include monitoring for unusual login attempts and implementing rate-limiting mechanisms to prevent brute-force attacks. Additionally, organizations can use third-party services to monitor and alert them of compromised credentials, allowing them to prompt affected users to change their passwords. Finally, educating users on the dangers of password reuse and encouraging them to use unique passwords across different accounts is essential.

Source: https://www.cnbc.com/2019/07/30/capital-one-data-breach-suspect-paige-thompson-had-access-to-servers.html

Phishing for Information

Tag: Resource Development **Category: Acquire Private Key**

What is "Phishing for Information"?

Phishing for information is the practice of tricking targets into revealing sensitive information through deceptive emails, social media messages, or other communications. In the context of Web3, phishing attacks can be used to steal private keys, wallet addresses, and other valuable resources that can be used to support an attack. Phishing is a common attack vector and can be used in different stages of an attack.

In the context of resource development, phishing is primarily used to obtain more information rather than the actual private key, as the hacker may not be aware of which validator or person holds the private key.

Example

A real-world example of phishing for information in Web3 is the event where Coinbase employees received a phishing SMS on their phones. The phishing link sought to gather access to sensitive information from the Coinbase staff.

It all started on Sunday, February 5, 2023, when several Coinbase employees received text messages asking them to use the link sent by the attacker for an urgent login. While all recipients ignored the text, one employee logged in with their username and password.

With the help of the employee's login credentials, the attacker attempted to access Coinbase's internal network. However, since the company had enabled multi-factor authentication (MFA) for employees, the attacker could not bypass the security feature and could not proceed further even after several attempts.

While the attacker was unsuccessful in accessing Coinbase's system, a limited amount of data from the company's directory was exposed, including names, email addresses, and phone numbers of a limited number of employees.

The Call

The second phase of the attack began with a phone call to the employee's mobile phone, with the attacker claiming to be a member of Coinbase's corporate Information Technology (IT) team.

Trusting that the caller was a legitimate Coinbase IT staff member, the employee logged into their workstation and followed the attacker's instructions. However, as the conversation progressed, the employee grew increasingly suspicious of the requests.

Thankfully, the employee's suspicions were enough to prevent damage. No funds were taken, and no customer information was accessed or viewed during the incident.

Example

Based on the attacker’s modus operandi, Coinbase believes the incident was not an isolated one and is linked to a series of cyberattacks that have taken place recently, including Twilio, DoorDash, Zendesk, Namecheap, and others.

Source: https://www.hackread.com/coinbase-employees-sms-phishing-attack/

Mitigation

To protect against phishing attacks, Web3 users can take several measures, including:

- Using anti-phishing browser extensions: Browser extensions, such as those for MetaMask and MyEtherWallet, can detect and block phishing websites and messages.
- Verifying URLs: Users should verify the URL of the website they are visiting, especially when dealing with sensitive information.
- Avoiding clicking on suspicious links: Users should avoid clicking on links in emails or messages from unknown senders or messages that seem too good to be true.
- Enabling two-factor authentication: Two-factor authentication can add an extra layer of security to users' accounts and prevent attackers from gaining access even if they have the user's password.
- Educating users: Educating users about the risks of phishing attacks and how to identify and avoid them can help prevent successful attacks.

Social Engineering

Tag: Resource Development **Category: Acquire Private Key**

What is "Social Engineering"?

Social engineering is a technique cybercriminals use to trick people into revealing sensitive information or performing actions that compromise their security. It involves manipulating individuals into giving away confidential information or performing actions that may put their security and privacy at risk. Social engineering uses psychological manipulation to trick victims into revealing sensitive information. In the context of Web3, blockchains, and crypto, social engineering can be used to gain access to social media accounts, email accounts, and other resources that can be used to support an attack.

Example

A phishing attack is a common example of social engineering in the web3 context. Phishing attacks usually involve an attacker pretending to be a trustworthy entity, such as a cryptocurrency exchange, and tricking the victim into providing sensitive information, such as their login credentials or private keys. In a web3 phishing attack, the attacker may also request or fetch the victim's wallet address, private key, or other sensitive data.

Pretexting is another technique used in social engineering. It involves creating a false scenario or pretext to gain the victim's trust and extract information. For instance, an attacker may pose as a bank employee, call a customer and request sensitive information, such as their social security number or credit card details, claiming that their account has been compromised.

This happened in the second stage of the Coinbase phishing event:

"The Call - The second phase of the attack began with a phone call to the employee's mobile phone, with the attacker claiming to be a Coinbase's corporate Information Technology (IT) team member.

Trusting that the caller was a legitimate Coinbase IT staff member, the employee logged into their workstation and followed the attacker's instructions. However, the employee grew increasingly suspicious of the requests as the conversation progressed.

Thankfully, the employee's suspicions were enough to prevent damage. No funds were taken, and no customer information was accessed or viewed during the incident.

Based on the attacker's modus operandi, Coinbase believes the incident was not an isolated one and is linked to a series of cyberattacks that have taken place recently, including Twilio, DoorDash, Zendesk, Namecheap and others."

Source: https://www.hackread.com/coinbase-employees-sms-phishing-attack/

Another example of pretexting is when an attacker poses as an IT support representative and calls an employee, claiming that their computer has been infected with a virus and they need to install a remote access tool to fix the issue. The remote access tool is actually malware that allows the attacker to gain access to the victim's system and steal sensitive data.

Overall, social engineering attacks can take various forms, and it is essential to be cautious and vigilant about any unexpected or suspicious requests for information or actions.

The last example of social engineering is swapping. In this attack, the attacker convinces the victim's mobile carrier to transfer the victim's phone number to a new SIM card, which the attacker controls. Once the attacker has control over the victim's phone number, they can reset passwords, receive 2FA codes, and gain access to the victim's accounts.

Mitigation

To mitigate social engineering attacks, web3 users should be cautious of unsolicited messages, links, or requests, especially those requesting sensitive information. It is essential to verify the authenticity of any request before providing sensitive information or performing any action that may put their security and privacy at risk. Users should also use two-factor authentication (2FA) wherever possible, which adds an extra layer of security to their accounts (sim cards are less secure, as mentioned). Additionally, users should keep their software and systems up-to-date with the latest security patches and use antivirus and firewall software to protect their devices from malware and other threats.

Lastly, education and awareness are crucial to prevent social engineering attacks, and users should be educated on the latest tactics used by attackers to trick them into giving away sensitive information or performing harmful actions.

Network profiling

Tag: Resource Development **Category: Analysis & profiling**

What is Network profiling?

Attackers can gather information on the topology of Web3 networks, identifying nodes, miners, and other network participants to identify potential targets for further attacks. In the context of web3 hacking, gathering information on the topology of Web3 networks is a reconnaissance technique that attackers use to map out the network infrastructure and identify potential targets for further attacks. By understanding the network structure, attackers can identify vulnerable nodes, miners, and other network participants to exploit.

Example

For example, an attacker might use network scanning tools to map out a blockchain network, identifying nodes that are publicly accessible and have weak security controls. They could then target those nodes with various attacks, such as denial-of-service attacks or exploits that exploit vulnerabilities in the node's software.

Mitigation

t is not possible to completely prevent attackers from gathering information on the topology of Web3 networks, as the information is publicly available. However, “network administrators” can take steps to make it more difficult for attackers to identify vulnerable nodes, such as implementing stronger security controls and limiting the amount of information that is publicly available about network participants.

Inital Access

What is Initial Access?

Techniques that use different entry vectors to establish a foothold inside a network are considered initial access. The first step in exploiting a smart contract or dApp is to gain initial access, which can be achieved through various methods such as exploiting vulnerabilities, stealing private keys or deploying malware. This phase is known as "initial access," marking the beginning of the Exploitation phase. Targeted phishing strategies and taking advantage of vulnerabilities on servers, smart contracts, wallets, and credentials are some methods used to establish a foothold. Footholds acquired during the preparation phase are then used to initiate access to the network or organization.

Example of Initial Access

1. Credential Theft: Hackers may attempt to steal login credentials or private keys to gain initial access to a Web3 network or platform. This can be done through phishing attacks, social engineering, or exploiting vulnerabilities in wallets or other cryptocurrency-related accounts.
2. API Vulnerabilities: Application Programming Interfaces (APIs) connect different Web3 platforms and applications. However, if these APIs are not properly secured, they may contain vulnerabilities that hackers can exploit to gain initial access to a Web3 network or platform.

Here are some specific examples of vulnerabilities or tactics that could be employed to gain initial access to a Web3 network or platform:

- Taking advantage of unpatched server vulnerabilities, such as those related to Apache Struts or Microsoft Exchange Server.
- Using brute-force attacks to crack weak login credentials or private keys.
- Exploiting weaknesses in APIs used to connect different Web3 platforms or applications, such as those related to API keys or authentication tokens.
- Using social engineering tactics to deceive users into revealing their login credentials or private keys.
- Using malware to infect targeted computers or servers.

Inital Access

Forged address phishing

Tag: Initial Access

Category: User Target

What is "Forged address phishing"?

Forged address phishing is a type of scam where an attacker creates a fake address that looks similar to a legitimate one and sends a small amount of cryptocurrency to the target's account from the fake address. This scam is similar to Zero Transfer Phishing, but instead of zero transfers, attackers use actual amounts. The attacker hopes that the target will mistake the fake address for the real one and copy it for a larger transaction, leading to the target mistakenly sending their cryptocurrency to the attacker's address instead of the intended recipient.

Example

For example, let's say a user wants to transfer $50,000 worth of Ether to their friend's address, 0x123456789abcdef. The attacker monitors this transaction and creates a fake address that looks very similar, such as 0x123456789abcdee, and sends a small amount of Ether, say $0.1, from the fake address to the user's account. The attacker hopes that the user will copy the fake address instead of the real one when making the large transfer, leading to the user mistakenly sending the $50,000 worth of Ether to the attacker's address.

Mitigation

To avoid falling for this type of phishing scam, users should always double-check the authenticity of the address they are sending funds, especially when dealing with large amounts of cryptocurrency. One way to do this is by comparing the first and last few characters of the address to ensure they match the intended recipient's address. Additionally, users can use secure communication channels, such as encrypted messaging or phone calls, to confirm the legitimacy of the recipient's address before sending any funds. Finally, it is essential to be aware of common cryptocurrency scams and stay vigilant against suspicious activity.

On-chain Scams

Tag: Initial Access **Category: User Target**

What are “On-chain Scams?”

On-chain Scams are a type of cyberattack that targets users of Web3 technology. In this case, attackers create a fake project or platform, such as a decentralized application (dApp) or a non-fungible token (NFT) mint, to lure victims into connecting their wallets. Once the victim connects their wallet, the malicious smart contract automatically drains the user's funds without their knowledge or consent.

Example

An attacker sets up a fraudulent NFT marketplace that imitates a popular and legitimate platform. They promote the fake marketplace through social media, forums, and email campaigns. Unsuspecting users, believing the marketplace to be legitimate, connect their Web3 wallets to the platform. Upon connecting, the malicious smart contract embedded in the fake marketplace executes, withdrawing funds from the connected wallets and transferring them to the attacker's wallet.

Mitigation

Verify platform legitimacy: Before connecting your wallet to a platform, ensure that it is a legitimate and trusted platform by checking its URL, reading reviews, checking etherscan, using smart contract reviewing tools, and seeking explanations of the smart contract from trusted sources.

Be cautious with links: Avoid clicking on links from unknown sources or that appear suspicious, and always double-check URLs before entering sensitive information.

Educate yourself: Stay informed about the latest security threats and best practices for safeguarding your digital assets. The more you know, the better you can protect yourself from phishing attacks and other types of cyber threats.

Zero Transfer Phishing

Tag: Initial Access

Category: User Target

What is Zero Transfer Phishing?

Illicit smart contracts generate "transfers" of zero-value tokens from the addresses of victims to fake addresses that resemble those with which the victims had previously interacted. The "transfers" have zero value because they don't actually represent the transfer of any tokens. As a result, they can be processed without the usual consent from the source or the victim's wallet.

The goal is to deceive the victim into mistakenly being sent to the attacker's fake address rather than the legitimate one they had previously communicated with. How does that function? Because many users frequently examine their transaction history to determine which addresses they have once sent to and copy and paste this address from the most recent transaction the victim submitted to it while setting up a new transaction. And how do the majority of users verify that an address is accurate? To ensure that the wallet address is constant throughout their previous transactions, they will swiftly scan the first and final few characters. They frequently need to evaluate and compare every character. Scan, Copy, Paste, Theft!

This type of hack is targeted at individuals and EOA wallets.

Example

A real-world example and explanation can be found on the Coinbase blog:
https://www.coinbase.com/blog/zero-transfer-phishing-part-1-attack-analysis

Mitigation

To prevent falling for Zero Transfer Phishing, triple-check that the wallet or contract address you are interacting with is correct. Do not only check the last numerals/letters in the address.

As Coinbase mentions in their article, there are other mitigation techniques:

- Verify the entirety of the address before sending. Attackers may have generated a vanity address to resemble a legitimate one closely.
- Be mindful about copying addresses from transactions that you did not originate or that look suspicious. Existing ERC-20 tokens will continue allowing zero transactions to and from arbitrary transactions.
- Use blockchain explorers (e.g., Etherscan) and wallets (e.g., Coinbase Wallet) which flag or filter malicious transactions and addresses.

Blockchain explorers and wallets can implement the following approaches to help shield consumers from this and similar threats:

Flag or filter transfer events with the value set to 0. Consider derivative exploitation vectors for non-ERC-20 transfer events (e.g. NFTs, staking, etc.).

Implement address mask collision detection to identify similar addresses unlikely to have been generated randomly (e.g., same N first and last characters).

If shortening addresses, consider including 3+ bytes on each side to make mass vanity generation harder (e.g. 0x123456...abcdef).

Alert users on new/unknown addresses when initiating transfers.

Discovery

What is “Discovery”?

Discovery involves mapping out the structure of a network. Attackers may use "discovery" tactics to gain insight into the system and internal network. Malicious parties can use these strategies to evaluate their surroundings and position themselves before an attack, in order to decide how to respond. They also investigate what they can control and the area around their access point to discover how certain strategies might help them achieve their current goal. Summing up discovery involves getting a list of all necessary information, such as wallets and smart contracts. For example, an attacker might fork the codebase and test several strategies before executing it in reality.

While the "Discovery" category does involve gathering information about a target system (which could be part of the preparation phase of an attack), it usually occurs after the initial access has been established. Once an attacker has gained a foothold in a target network, they may use various techniques to map out the network, identify assets of interest, and determine how to move laterally through the network. Therefore, the "Discovery" category can also be considered part of the active exploitation rather than the preparation phase.

Discovery

API Discovery

Tag: Discovery

Category: X

What is "API Discovery"?

After gaining initial access to a dApp, an attacker may attempt to discover its underlying infrastructure, such as connected backend services, databases, or APIs. API Discovery is a technique adversaries use to identify and enumerate Application Programming Interfaces (APIs) exposed by blockchain nodes or decentralized applications (dApps). These APIs interact with the blockchain network and perform various tasks, such as submitting transactions, querying data, or monitoring events. Adversaries can use multiple techniques to discover and enumerate these APIs, such as scanning the network, analyzing the source code of smart contracts or dApps, or using specialized tools designed for API discovery. Once vulnerable or misconfigured APIs are identified, adversaries can exploit them to gain unauthorized access to sensitive data or disrupt the network.

Example

An example of this is the FTX collapse. External sources have cited that unauthorized access to API keys was one of the reasons for the hack and subsequent collapse.

Mitigation

To reduce the risks associated with API discovery, blockchain developers and organizations can take the following measures:

- Implement Access Controls: Ensure that APIs are protected with authentication and access controls, like API keys or OAuth tokens. This will help prevent unauthorized access to sensitive data or functions.
- Monitor API Activity: Keep a close eye on API activity, and log all requests and responses to detect suspicious or unauthorized behavior. This will help identify potential attacks and provide forensic evidence in case of a breach.
- Regularly Update and Patch: Keep APIs up-to-date, and patch them to address known vulnerabilities and misconfigurations. This will help reduce the attack surface and prevent the exploitation of known weaknesses.
- Use Security Tools: Use specialized security tools designed for API discovery and vulnerability scanning, like Nmap, Burp Suite, or ZAP, to identify and fix vulnerabilities in blockchain APIs. These tools can also help validate access controls' effectiveness and identify potential API implementation weaknesses.

Source: https://beincrypto.com/ftx-users-lose-millions-to-api-exploit/

Dumpster Diving

Tag: Discovery

Category: Acquire Private Key

What is Dumpster Diving?

Dumpster diving is a technique used in social engineering that involves searching through an organization's trash or recycling for sensitive information. In the context of Web3, dumpster diving can be used to find information such as notes containing private keys, wallet addresses, and other resources that can be used to facilitate an attack. Attackers can use this method to gather information about their target and plan an attack.

Example

For instance, a hacker might have accessed a DApp developer's computer. In this situation, they may use other discovery strategies, like dumpster diving, to uncover more information to aid them in their attack. This scenario is part of the discovery phase since the hacker is still seeking to gather more information, even though they have already hacked into the victim's computer.

Mitigation

Organizations should dispose of sensitive information appropriately to reduce the risk of dumpster diving. This can include shredding documents that contain sensitive information and destroying hard drives and other storage devices that are no longer needed. It is also essential to educate employees about the risks of dumpster diving and implement security protocols that limit the amount of sensitive information available physically. Finally, organizations should consider using encryption and other security measures to protect sensitive data, even if it is accidentally disposed of.

Execution

What is “Execution”?

Execution refers to the methods attackers use to run malware or malicious code and carry out active exploitation techniques on local or remote systems to achieve broader objectives such as network exploration, data theft, or monetary gain. Within the context of Web3, execution can encompass a wide range of attack vectors and exploit techniques specifically tailored to decentralized environments, including blockchain networks and smart contracts.

Attackers may utilize private keys obtained during the initial access phase to deploy malicious smart contracts, interact with existing contracts, or manipulate user wallets and resources. They may also use known vulnerabilities in smart contracts or decentralized applications (dApps) to execute unauthorized transactions, create backdoors, or compromise user data.

Moreover, execution in the Web3 landscape can involve abusing decentralized finance (DeFi) protocols, tokenization platforms, and other Web3 services. Exploits may include flash loan attacks, reentrancy attacks, or oracle manipulation, which allow attackers to profit from poorly designed or insecure smart contracts and protocols.

By incorporating Web3-specific execution techniques alongside traditional approaches, attackers can effectively adapt their tactics to the unique characteristics of decentralized systems, making the execution phase a crucial component in the exploitation process. This enables attackers to conduct a wide range of attacks targeting Web3 ecosystems, ultimately impacting the security and integrity of these systems and their users.

Execution

MEV

Tag: Execution **Category: User Target**

What is MEV?

MEV (Maximal Extractable Value) refers to identifying and taking advantage of opportunities created by the order in which transactions are processed within a block.
For example, consider a decentralized exchange (DEX) that uses an automated market maker (AMM) algorithm to determine the price of a cryptocurrency. When a trader wants to swap one token for another on this DEX, they send a transaction processed by the AMM and the blockchain. However, several other traders may also attempt to take advantage of the same price movements by submitting transactions simultaneously while carefully monitoring other traders. In this scenario, the order in which the transactions are included in the block is crucial in determining which trader's transaction is executed first and who ultimately profits from the transaction.

Example

Imagine a DeFi lending protocol where users can borrow funds by collateralizing their cryptocurrency holdings. If the value of this collateral drops below a certain threshold, the protocol can liquidate the collateral to recover the borrowed funds. A trader observing the blockchain for these liquidation events can submit a transaction that buys up the liquidated assets at a discount before other traders can react and then sell them for a profit.

In this scenario, the MEV opportunity arises from the order of transactions in the block rather than by manipulating the transaction pool through front-running.

Mitigation

MEV (Miner Extractable Value) exploits refer to a type of attack on a blockchain that allows miners to manipulate transaction orders and potentially profit at the expense of other users. Here are some ways to prevent MEV exploits:

- Use an MEV protection tool: Tools such as Flashbots can help protect against MEV exploits by allowing users to bundle their transactions and communicate directly with miners. This reduces the incentive for miners to engage in MEV exploits.
- Implement transaction fee caps: Users can limit the profit miners make from MEV exploits by setting caps on transaction fees. This can be done by implementing fee market protocols

Use privacy-preserving technologies: MEV exploits often rely on the ability to track transactions and manipulate their order. By implementing privacy-preserving technologies such as zk-SNARKs, transactions can be made more private and less susceptible to manipulation.

Implement transaction finality: MEV exploits often rely on the ability to manipulate transaction order. By implementing transaction finality, transactions become irreversible and less susceptible to manipulation. Use decentralized exchanges: Decentralized exchanges (DEXs) can help prevent MEV exploits by eliminating the need for transaction ordering. By using a DEX, transactions are settled in a trustless manner, reducing the risk of MEV exploits.

It is important to note that preventing MEV exploits is an ongoing challenge in the blockchain space, and new solutions may emerge over time. Users need to stay informed and vigilant against potential threats to their transactions.

Sources: https://flashbots.net/

https://flashbots.net/

https://www.coindesk.com/what-is-mev-crypto

Flash Loan (AMM Exploitation)

Tag: Execution **Category: Oracle / AMM**

What is an AMM Exploit (Flash Loan)?

An Automated Market Maker (AMM) is a decentralized exchange (DEX) model that enables users to trade cryptocurrencies without relying on order books. Instead, AMMs use smart contracts to pool liquidity and determine the price of assets based on a mathematical algorithm, which allows for instant settlement.

The standard AMM algorithm is X*Y = K.

Flash loans are uncollateralized loans that enable users to borrow funds without providing any collateral as long as the funds are repaid within the same transaction. In this attack, the attacker exploits the liquidity pool of an AMM using a flash loan. The attacker borrows a significant amount of cryptocurrency to buy or sell a token on the AMM, temporarily causing the price to shift in their favor. They then repay the loan, pocket the profits, and withdraw their original funds from the pool.

This attack is possible because AMMs have no external price feeds and rely solely on the internal price determined by the smart contract, making them vulnerable to manipulation by large trades that can temporarily shift the price in the attacker's favor. However, some AMMs have implemented measures to prevent this attack, such as implementing price oracles to provide external price feeds and limiting the amount of liquidity that can be traded in a single transaction.

Example

In February 2021, Platypus Finance suffered a flash loan reentrancy attack resulting in $8.5 million in lost funds.

In May 2021, a flash loan attack on PancakeBunny, a decentralized finance protocol, resulted in losing $200 million worth of assets.

Value DeFi was hit by a flash loan attack in November 2020, causing a loss of $6 million in funds. The attacker exploited a vulnerability in the system and drained the liquidity pool of Value DeFi's MultiStables vault. Cream Finance, a decentralized finance lending platform, was attacked by a flash loan in February 2021, resulting in a loss of $37.5 million. The attacker exploited a vulnerability in Cream's Iron Bank protocol and escaped with many funds.

In August 2020, bZx, a decentralized finance platform, experienced a flash loan attack where the attacker manipulated the price of two tokens and caused a loss of $8 million.

Mitigation

Preventing flash loan exploits in automated market maker (AMM) pools can be challenging due to their decentralized and permissionless nature. AMM utilise liquidity pools and automatically adjust the price of an asset based on supply and demand. This creates opportunities for arbitrage and other trading strategies. However, this also means that anyone with an internet connection can access DeFi protocols and execute flash loans, making it difficult to prevent or restrict their use. Additionally, these loans can be challenging to detect, as they often involve complex trading patterns and multiple transactions across different protocols. Preventing flash loan attacks is challenging as the price of assets in AMM pools is determined by supply and demand, and flash loans just enable someone to manipulate the pools which huge supply. Flash loans are used to manipulate the price of an asset in the pool, creating opportunities for traders to profit at the expense of other liquidity providers, especially in illiquid or low-volume markets.

To mitigate the risks associated with flash loans, DeFi developers and liquidity providers can implement several measures, including implementing circuit breakers or other mechanisms to temporarily halt trading in the event of sudden price changes or liquidity imbalances, transaction fees, or other restrictions on flash loan usage to deter or limit the use of flash loans in trading strategies. Improving liquidity in AMM pools to reduce the impact of flash loan arbitrage and developing more sophisticated monitoring and analysis tools to detect and prevent flash loan attacks can also be helpful.

Overall, preventing flash loans in DeFi is challenging. Still, by implementing best practices and developing more sophisticated tools and protocols, it is possible to mitigate the risks associated with flash loans and protect the interests of liquidity providers and other DeFi users.

Source https://cointelegraph.com/news/7-defi-protocol-hacks-in-feb-sees-21-million-in-funds-pilfered-defillama

https://www.coindesk.com/pancakebunny-defi-attack https://cointelegraph.com/news/value-defi-hack-takes-6m-in-yet-another-flash-loan-attack

https://cointelegraph.com/news/cream-finance-suffers-37-5m-flash-loan-attack https://www.coindesk.com/bzx-hacked-again-possible-losses-in-2-millions

Oracle attack

Tag: Execution **Category: Oracle / AMM**

What is an Oracle Attack?

An Oracle is a trustworthy third-party data source that a smart contract can use to obtain external information. Oracle attacks involve manipulating the Oracle to provide false or malicious data to the smart contract or any party depending on the data. This can result in unauthorized access, theft of cryptocurrency, or even liquidation events.

One type of Oracle attack is incorrect or insecure validation, where malicious actors can manually change the price of an Oracle by exploiting vulnerabilities in the validation process. This allows them to provide inaccurate data to the smart contract, which can cause financial losses or unauthorized access.

Example

A similar hack occurred with BonqDAO. According to reports, the hacker gained access to the Tellor price feed for (wrapped) WALBT collateral by staking 10 TRB tokens, which were valued at approximately $175. source Another instance is the compound oracle liquidation event, in which Coinbase's DAI stablecoin oracles were susceptible to spoofing and oracle manipulation. source

Oracle attacks can vary between decentralized and centralized exchanges. Oracles are frequently centralized entities, making them a single point of failure. In such cases, an Oracle attack can result in market data manipulation, leading to market distortions and financial losses. One example is spoofing, which involves executing false orders to manipulate prices and the perception of price action.

In decentralized exchanges, the most common types of "Oracles" are Automated Market Makers (AMMs) and Order Books. With the AMM model, attackers can manipulate the price of tokens by exploiting liquidity imbalances using flash loans. This can lead to significant market distortions, causing financial losses for traders. On the other hand, with the Order Book model, attackers can manipulate the order books to falsely represent supply and demand, resulting in significant market distortions and financial losses. Although quite distinct, it has its section in the execution list.

Mitigation

To prevent oracle attacks, it is best to follow secure coding practices such as using multiple independent oracles and drawing a median of the reported price. It is important to thoroughly verify the data and reported price received from the oracle. Choose a trustworthy oracle: The first step in preventing oracle attacks is to choose a reputable oracle. It is essential to research the oracle thoroughly and verify its reputation.

Use a decentralized oracle network: A decentralized oracle network can add an extra layer of security to your dApp. Decentralized oracle networks ensure data integrity by using multiple oracles to verify the same data. Chainlink oracles are an excellent example of this.

Cross-Chain Bridge Attacks

Tag: Execution **Category: Cross Chain**

What is a Cross-Chain Bridge attack?

A cross-chain smart contract attack is an attack that exploits vulnerabilities in smart contracts that interact with multiple blockchains or networks. Cross-chain smart contracts enable users to perform transactions or execute code on different blockchains, allowing for interoperability and functionality. However, this also opens up new attack vectors for hackers to exploit.

A cross-chain smart contract attack typically involves the exploitation of a vulnerability in one smart contract to gain unauthorized access to another smart contract on a different blockchain or network.

Example

A real-world example of a cross-chain attack is the Nomad hack.

In August, a security flaw was found in the cross-chain bridge Nomad, and almost all of its funds (more than $190 billion) were drained from its platform. It was when Nomad first altered their code that the assault began. The Nomad Bridge incident was not perpetrated by one entity or organization but involved hundreds of addresses. Many people "jumped on the train," noticing that Nomad had a vulnerability that could be exploited. Precisely at 9:32 p.. UTC on August 1, 100 Wrapped $BTC ($WBTC) got stolen from the platform, creating the beginnings of what we now recognize as a significant security exploit.

The attackers exploited a flaw in the smart contract's initialize method to send messages that tricked Noad Bridge into sending stored tokens without proper authorization. With this vulnerability, the malicious actors withdrew more money than they had originally deposited. The attackers continued exploiting the bridge until an estimated $190 billion worth of cryptocurrency was stolen.

Mitigation

To prevent cross-chain smart contract attacks, developers should implement best practices such as:

- Auditing smart contracts for vulnerabilities and testing them under various scenarios.
- Cross-chain Real-time monitoring.
- Implementing secure communication channels between blockchains to prevent unauthorized access.
- Using secure key management and encryption techniques to protect private keys and other sensitive information.
- Implementing robust access control mechanisms and limiting the exposure of sensitive information.
- Monitoring and analyzing blockchain activities and transactions to detect and prevent suspicious activities.
- Leveraging third-party security experts to identify and address potential vulnerabilities in cross-chain smart contracts.

Token supply manipulation

Tag: Execution

Category: Smart Contract Vulnerabilities

What is “token supply manipulation”?

Token supply manipulation, also known as "minting" or "inflation", is a vulnerability that can occur in smart contracts that allow for the creation of new tokens beyond the initial supply. This vulnerability can arise if the contract owner or an authorized user can mint new tokens without proper oversight or limitations.

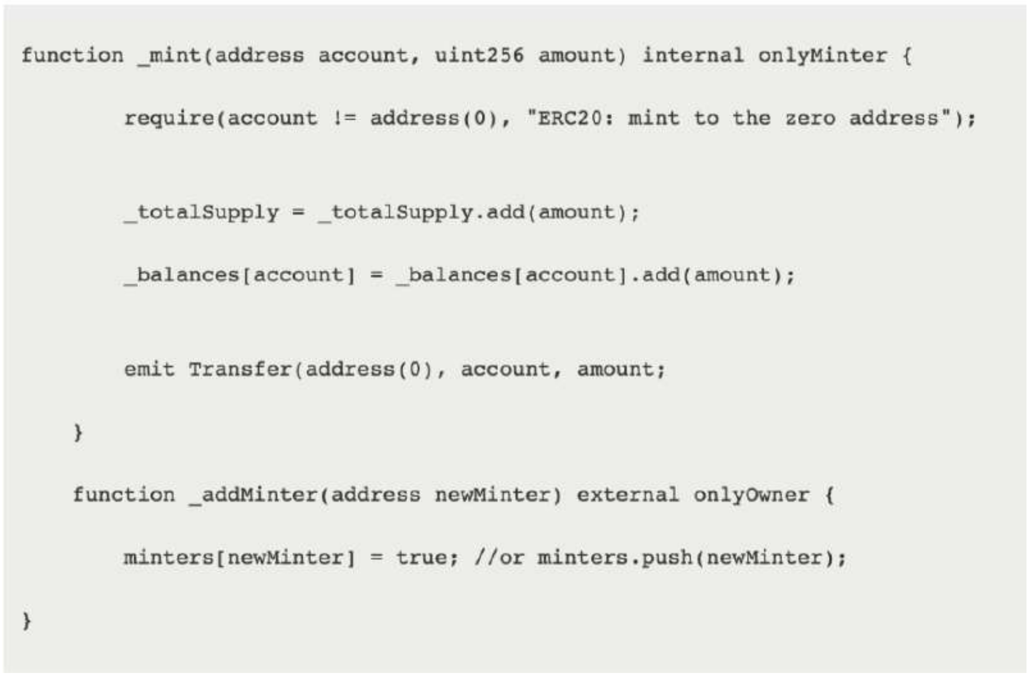
An example of token supply manipulation is if a contract owner can mint new tokens at will, without any restrictions or oversight. This can lead to the dilution of existing token holders' shares and potentially impact the token's value.

An endless mint attack happens when a malicious party or hacker creates excessive tokens within a protocol, raising the supply to an unhealthy level and eroding the token's value. Attackers frequently complete the operation quickly and leave with tokens valued at millions of dollars. Attackers frequently go on to flood the market with all the newly created tokens, driving the price down. Smart contracts are susceptible to this kind of attack mostly due to code flaws that let hackers take advantage of bugs and weak code areas.

Example

Cover Protocol.

Hackers used shield mining contracts in the Cover Protocol attack to obtain unauthorized crypto rewards from the system. The Cover staking pool's token price fell by 97% due to the hacker's successful use of 40 quintillion tokens on the network. In this instance, the attacker used 1inch to liquidate over 11,700 coins and steal tokens valued at almost $5 million.



```
function _mint(address account, uint256 amount) internal onlyMinter {

        require(account != address(0), "ERC20: mint to the zero address");


        _totalSupply = _totalSupply.add(amount);

        _balances[account] = _balances[account].add(amount);


        emit Transfer(address(0), account, amount;

    }

    function _addMinter(address newMinter) external onlyOwner {

        minters[newMinter] = true; //or minters.push(newMinter);

}
```

A cybersecurity attack that took advantage of a flaw in a Cover Protocol smart contract was known as the Cover Protocol exploited in 2020. Because of the vulnerability, attackers could create COVER tokens indefinitely. A security company rectified the flaw in the Cover Protocol smart contract.

Source: x

Mitigation

To prevent token supply manipulation, it is important to implement proper limitations and oversight mechanisms for the minting function. This can include setting a maximum supply limit, requiring multiple approvals or signatures for minting, or implementing a community-driven governance mechanism to oversee the minting process.

Additionally, it is important to conduct regular audits and security checks of the contract to ensure no vulnerabilities that could allow unauthorized parties to mint new tokens. Any potential vulnerabilities or weaknesses should be identified and addressed promptly to ensure the security and integrity of the contract and its tokens.

Crypto-jacking

Tag: Execution **Category: x**

What is crypto-jacking?

Crypto-jacking is a cyber-attack where malicious actors use a victim's computing resources to mine cryptocurrency without their knowledge or consent. This attack is usually achieved by injecting a script into a website or a software program, which runs in the background without the user's knowledge or consent. The attacker benefits by receiving the mined cryptocurrency, while the victim suffers from the degraded performance of their device and increased energy consumption.

Example

An example of crypto-jacking is the Coinhive script, which was widely used by cybercriminals to mine Monero cryptocurrency by exploiting the computing power of users who visited compromised websites. Another example is the XMRig malware, which infects computers and mobile devices to mine Monero.

Mitigation

To prevent crypto-jacking, users can take the following measures:

1. Install ad-blocking and anti-malware software: These tools can detect and block crypto-jacking scripts before they can infect your device.
2. Use a browser extension: Some browser extensions, like NoCoin can prevent crypto-jacking scripts from running on your device.

Keep your software up-to-date: Cryptojackers often exploit vulnerabilities in outdated software. Regularly updating your software can help prevent these attacks.

Be cautious of suspicious links and downloads: Crypto-jacking malware can be hidden in phishing emails, malicious websites, and software downloads. Be wary of any suspicious links or downloads, and only download software from trusted sources.

Monitor your device's performance: If it is running slower than usual or consuming more energy, it may be a sign of crypto-jacking. Monitor your device's performance and investigate any suspicious activity.

Check-Effect- Interaction (CEI)

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is "Check Effect Interaction"?

"Check-Effect-Interaction" (CEI) is a common pattern used in smart contract development to prevent race conditions and ensure that transactions execute as intended. The CEI pattern involves three steps:

1. Check: The contract checks whether the transaction is valid or not.
2. Effect: If the transaction is valid, the contract executes the intended changes to the contract state.
3. Interaction: The contract interacts with other contracts or external entities, such as sending or receiving funds.

Example

An example of CEI can be seen in a contract that allows users to withdraw funds. The contract would check that the user has sufficient funds to withdraw, then effect the withdrawal by updating the user's balance, and finally interact with the external entity to send the requested funds to the user's account.

Without CEI, there is a risk of a race condition where two or more transactions attempt to modify the contract state simultaneously, leading to unexpected behavior and potential vulnerabilities.

Mitigation

To prevent issues related to race conditions, it is essential to follow the CEI pattern when designing and implementing smart contracts. Additionally, contracts should be tested thoroughly to ensure they behave as expected and resist any potential vulnerabilities that may arise from race conditions.

Block Timestamp Manipulation

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is “Block timestamp manipulation”?

Block Timestamp Manipulation vulnerability is a type of vulnerability in smart contracts where an attacker can manipulate the timestamp of a block. The timestamp can be used in smart contracts to determine if a certain action can be executed, such as releasing funds after a certain period. If the timestamp can be manipulated, an attacker can trick the smart contract into executing an action prematurely or delaying it indefinitely.

Example

One real-world example of Block Timestamp Manipulation is the batchOverflow attack on the BEC token smart contract. In this attack, the attacker manipulated the block timestamp to cause an integer overflow when calculating the number of tokens to be transferred, transferring excessive tokens to the attacker's account.

Source: ☒

Mitigation

The mitigation for Block Timestamp Manipulation involves using a secure time source that attackers cannot manipulate. One solution is to use the block's median timestamp instead of its timestamp as a measure of time. Another solution is to use an external time source, such as an oracle, to provide the time for the smart contract. Additionally, developers should perform proper input validation and limit the number of funds transferred in a single transaction.

Self-destruct

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is the “self-destruct” function?

The "self-destruct" function in Solidity is a feature that allows a smart contract to be destroyed and its funds to be sent to a designated address. While this can be useful for cleaning up unused contracts and returning funds to investors, it can also be a potential vulnerability if not used properly.

Example

One example of a vulnerability that can arise from the self-destruct function is when a contract's address is publicly available, and an attacker can call the self-destruct function on the contract, causing it to be destroyed. Its funds are to be sent to the attacker's address.

Another example is when a contract's self-destruct function is combined with a vulnerable function, such as a function that allows an attacker to set the self-destruct address. In this case, the attacker can set the self-destruct address to their own address and then call the vulnerable function, causing the contract to be destroyed and its funds to be sent to the attacker.

Mitigation

To prevent self-destruct vulnerability, it is important to carefully consider the usage of the self-destruct function in a contract and to use it only when necessary. If the self-destruct function is used, it should only be called by an authorized user or function. The designated address should be carefully chosen to ensure funds are sent to the intended recipient. Additionally, contracts should be tested and audited regularly to ensure they remain secure as changes are made to the code.

Floating Pragma

Tag: Execution

Category: Smart Contract Vulnerabilities

What is “floating pragma”?

The "floating pragma" is a vulnerability in smart contracts written in the Solidity programming language. Using a floating pragma statement can result in unexpected behavior due to changes in the compiler version.

In Solidity, a pragma statement is used to specify the compiler version that should be used to compile the contract. A floating pragma statement is a pragma statement that uses a caret (^) symbol to allow for automatic updates to the compiler version. For example, the statement "^0.8.0" would allow automatic updates to any version greater than or equal to 0.8.0, but less than 0.9.0.

Compile the contract. A floating pragma statement is a pragma statement that uses a caret (^) symbol to allow for automatic updates to the compiler version. For example, the statement "^0.8.0" would allow automatic updates to any version greater than or equal to 0.8.0, but less than 0.9.0.

An example of a floating pragma vulnerability is if a contract uses a floating pragma statement that allows for updates to any version greater than or equal to 0.8.0. If a new compiler version is released that introduces breaking changes to the Solidity language, the contract may be compiled with the new version, resulting in unexpected behavior or even vulnerabilities.

Example Unknown

Mitigation

To prevent the floating pragma vulnerability, it is recommended to use a fixed pragma statement that specifies a specific compiler version that is known to work with the contract. This can be done by using a pragma statement such as "pragma solidity 0.8.0;" instead of a floating pragma statement. Additionally, contracts should be tested and audited regularly to ensure they remain secure and functional as changes are made to the Solidity language and compiler.

If you leave a floating pragma in your code (pragma solidity ≥ 0.7.0 < 0.9.0.), you will not be sure which version has been used to compile your code which means that you might encounter unexpected behaviors.

You should lock the solidity pragma to a specific solidity version so you can be sure of how the contract will behave once deployed.

Source: https://medium.com/coinmonks/smart-contracts-common-vulnerabilities-solidity-e64c5506b7f4

Outdated Compiler

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is an outdated compiler?

An outdated compiler in the database for Web3 refers to an older version of the Solidity programming language compiler, which is used to write smart contracts on the Ethereum blockchain. Solidity is a programming language that enables the development of smart contracts on the Ethereum blockchain. The compiler is responsible for translating Solidity code into bytecode that can be executed on the Ethereum Virtual Machine (EVM).

As with any software, newer versions of the Solidity compiler are regularly released to fix bugs, improve performance, and introduce new features. An outdated compiler may have security vulnerabilities that attackers could exploit, leading to potential loss of funds or other unintended consequences.

Example

The issue with an outdated compiler would be that it does not include security fixes for known vulnerabilities or may need certain security features added in more recent versions. This could make smart contracts written with an outdated compiler more susceptible to attacks.

- No real-world example was found.

Mitigation

The mitigation for an outdated compiler is to update to a recent version of the Solidity compiler. This can be done by downloading the latest compiler version from the official Solidity website or using a package manager like npm or yarn. It is recommended to regularly update the Solidity compiler to ensure the security and reliability of smart

contracts running on the Ethereum blockchain.

Tx.Origin Authentication

Tag: Execution

Category: Smart Contract Vulnerabilities

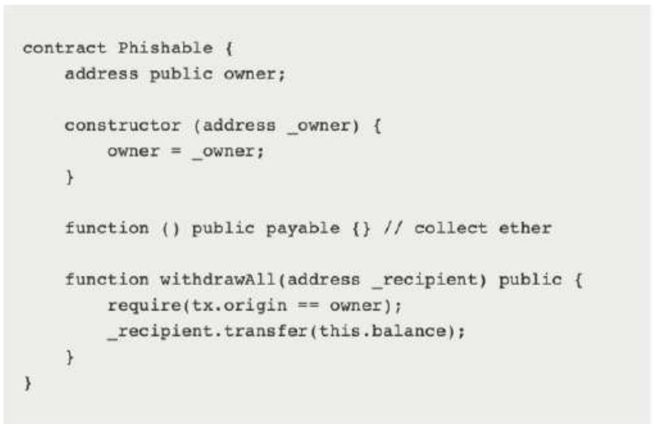
What is "Tx. Origin Authentication"?

Solidity has a global variable, tx. origin, which traverses the entire call stack and returns the account address that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.

Contracts that authorize users to use the tx.origin variable are typically vulnerable to phishing attacks which can trick users into performing authenticated actions on the vulnerable contract.

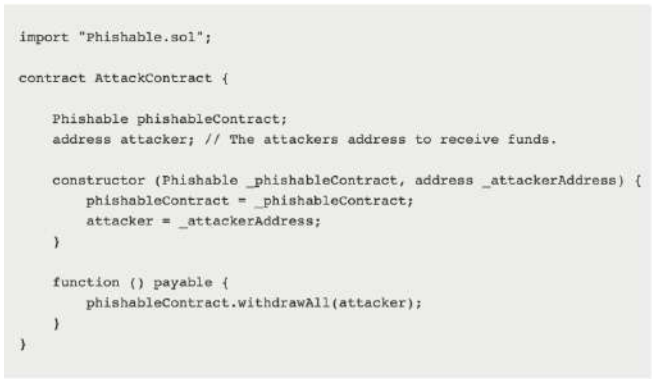
Example

Consider the simple contract,



```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

This contract authorises the withdrawAll() function using tx.origin. This contract allows for an attacker to create an attacking contract of the form,



```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attackers address to receive funds.

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

To utilize this contract, an attacker would deploy it and then convince the owner of the Phishable contract to send this contract some amount of ether. The attacker may disguise this contract as their own private address and social engineer the victim to send some form of transaction to the address. The victim, unless careful, may not notice that there is code at the attacker's address, or the attacker may pass it off as being a multi-signature wallet or some advanced storage wallet (remember, the source code of public contracts is not available by default).

In any case, if the victim sends a transaction (with enough gas) to the AttackContract address, it will invoke the fallback function, which in turn calls the withdrawAll() function of the Phishable contract, with the parameter attacker. This will result in the withdrawing all funds from the Phishable contract to the attacker address. This is because the address that first initialized the call was the victim (i.e. the owner of the Phishable contract). Therefore, tx.origin will be equal to owner and the require online [11] of the Phishable contract will pass.

Mitigation

tx.origin should not be used for authorization in smart contracts. This isn't to say that the tx.origin variable should never be used. It does have some legitimate use cases in smart contracts. For example, if one wanted to deny external contracts from calling the current contract, they could implement a require of the from require(tx.origin == msg.sender). This prevents intermediate contracts from being used to call the current contract, limiting the contract to regular code-less addresses.

Uninitialized storage pointers

Tag: Execution

Category: Smart Contract Vulnerabilities

What are “Uninitialized storage pointers”?

Uninitialized storage pointers vulnerability occurs when a smart contract uses uninitialized storage pointers that can be modified by a potential attacker, allowing them to write malicious code or steal funds. Storage pointers are variables used with smart contracts to store information on the blockchain. Uninitialized storage pointers occur when a developer fails to assign an initial value to a storage pointer.

The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it can produce vulnerable contracts by inappropriately initializing variables.

To read more about storage and memory in the EVM, see the Solidity Docs: Data Location, Solidity Docs: Layout of State Variables in Storage, Solidity Docs: Layout in Memory.

This section is based off the excellent post by Stefan Beyer. Further reading on this topic can be found in Sefan's inspiration, which is this Reddit thread.

Local variables within functions default to storage or memory depending on their type. Uninitialized local storage variables can point to other unexpected storage variables in the contract, leading to intentional (i.e., the developer intentionally puts them there to attack later) or unintentional vulnerabilities.

Example

A honey pot named OpenAddressLottery (contract code) was deployed that used this uninitialized storage variable query to collect ether from some would-be hackers. The contract is in-depth, so I will leave the discussion to this Reddit thread, where the attack is clearly explained. Another honey pot, CryptoRoulette (contract code), also uses this trick to collect some ether. If you need help figuring out how the attack works, see An analysis of a couple of Ethereum honeypot contracts for an overview of this contract and others.

Mitigation

The Solidity compiler raises uninitialized storage variables as warnings. Thus developers should pay careful attention to these warnings when building smart contracts. The current version of mist (0.10) doesn't allow these contracts to be compiled. It is good practice to explicitly use the memory or storage keywords when dealing with complex types to ensure they behave as expected as of Solidity version 0.5.0use of memory and storageare mandatory. To mitigate this vulnerability, developers should ensure that all storage pointers are initialized with a default value, such as zero or null, before being used in the smart contract. Developers should also perform thorough testing and auditing of their smart contracts to identify and address potential vulnerabilities before deploying them on the blockchain. Additionally, developers should follow best practices for secure codings, such as using secure development frameworks and the principle of least privilege.

Source: https://github.com/sigp/solidity-security-blog#storage

Constructors with Care

Tag: Execution

Category: Smart Contract Vulnerabilities

What is “Constructors with care”?

Constructors with Care is a vulnerability in Solidity smart contracts where the constructor function is not designed properly, leading to unexpected results and potential vulnerabilities. The constructor function is executed only once during the deployment of the contract, and it initializes the state variables of the contract. The issue arises when the constructor function does not take proper care of the variables and can lead to unintended behavior in the contract.

Constructors are special functions that often perform critical, privileged tasks when initializing contracts. Before solidity v0.4.22, constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, it becomes a normal, callable function if the constructor name isn't changed. As you can imagine, this has led to some interesting contract hacks.

For further reading, the reader should attempt the Ethernaught Challenges (in particular, the Fallout level).

Vulnerability: If the contract name gets modified or there is a typo in the constructor's name such that it no longer matches the name of the contract, the constructor will behave like a normal function. This can lead to dire consequences, especially if the constructor performs privileged operations. Consider the following contract

```
contract OwnerWallet {
    address public owner;

    //constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

This contract collects ether and only allows the owner to withdraw all the ether by calling the withdraw() function. The issue arises because the constructor is not exactly named after the contract. Specifically, ownerWallet is not the same as OwnerWallet. Thus, any user can call the ownerWallet() function, set themselves as the owner, and then take all the ether in the contract by calling withdraw().

Example

Rubixi (contract code) was another pyramid scheme exhibiting this vulnerability. It was originally called DynamicPyramid, but the contract name was changed before deployment to Rubixi. The constructor's name wasn't changed, allowing any user to become the creator. Some interesting discussions related to this bug can be found on this Bitcoin Thread. Ultimately, it allowed users to fight for creator status to claim the fees from the pyramid scheme. More detail on this particular bug can be found here.

Mitigation

This issue has been primarily addressed in the Solidity compiler in version 0.4.22. This version introduced a constructor keyword that specifies the constructor rather than requiring the function's name to match the contract name. As highlighted above, using this keyword to specify constructors is recommended to prevent naming issues. To mitigate the Constructors with Care vulnerability, developers should properly initialize the variables in the constructor function and ensure it is designed to be executed only once. Additionally, contracts should be thoroughly tested and audited to ensure no unexpected behaviors. Best practices should be followed, and external libraries or contracts should be used when possible instead of creating custom code.

Source: https://github.com/sigp/solidity-security-blog#constructors

Short Address/Parameter Attack

Tag: Execution

Category: Smart Contract Vulnerabilities

What is a Short Address/Parameter attack?

The Short Address/Parameter Attack vulnerability occurs when a contract or function doesn't validate the length of the input data. It allows an attacker to send a transaction with a shortened input, which can lead to unexpected behavior, including transferring funds to an unintended address or bypassing the intended logic of the contract. The attack is possible because Ethereum's virtual machine (EVM) pads the input data to a specific length, but it doesn't check if the input is that length.

Example

n 2018, the smart contract of a blockchain-based game called Fomo3D was found to be vulnerable to a Short Address Attack. The contract was designed to allow players to buy keys and compete for a pot of Ether. However, the function that handled the purchase of keys didn't check the length of the input data, which allowed attackers to exploit the contract and drain the pot of Ether. By sending a transaction with a shortened input, the attacker could bypass the intended logic of the contract and transfer the Ether to their address.

Source: https://www.apriorit.com/dev-blog/556-fomo3d-vulnerability

Mitigation

Developers can mitigate the Short Address/Parameter Attack vulnerability by implementing input validation in their smart contracts. They should check the length of the input data and reject any transactions that don't meet the expected length. Additionally, contracts can use a checksum to verify the integrity of the input data. Using standardized interfaces, like ERC-20 and ERC-721, can also help mitigate the risk of this vulnerability, as these interfaces include standardized functions that validate input parameters. Finally, users can protect themselves by checking the address they are sending funds, as some wallets automatically pad addresses to prevent this attack.

External contract referencing

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is “External contract referencing”?

External Contract Referencing (ECR) is a vulnerability that arises when a smart contract relies on an external contract whose address can be changed by an attacker. This can occur when a smart contract references another contract to perform a specific function. Still, the address of the external contract is not fixed or hard coded in the smart contract. An attacker can exploit this vulnerability by changing the address of the external contract, causing the smart contract to interact with a malicious contract and potentially leading to unauthorized access or data theft.

Example

An example of ECR vulnerability is the King of the Ether smart contract game developed in 2016. The game was designed to be played by depositing Ether into a smart contract, with the winner being the player who deposits the most Ether within a specific time frame. However, the smart contract relied on an external contract for some of its functionality, and the address of this external contract was not hard coded. This allowed an attacker to exploit the vulnerability by deploying a malicious contract with the same name as the external contract and changing its address. The attacker then called the functions in the malicious contract instead of the intended external contract, allowing them to steal the deposited Ether and win the game.

Source: https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8

Mitigation

To mitigate the ECR vulnerability, developers should ensure that the addresses of all external contracts that a smart contract relies on are hardcoded within the smart contract. This makes it more difficult for an attacker to change the address of the external contract and exploit the vulnerability. Additionally, developers should perform extensive testing and auditing to identify and address potential vulnerabilities in their smart contracts.

Entropy illusion/predictability

Tag: Execution

Category: Smart Contract Vulnerabilities

What is “Entropu Illusion”?

The Entropy Illusion vulnerability occurs when a blockchain application generates random numbers with insufficient entropy, which can lead to predictable or easily guessable numbers. This can compromise the security of cryptographic operations, such as private key generation or cryptographic signatures, which rely on unpredictable random numbers for their strength.

All transactions on the Ethereum blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the Ethereum ecosystem, and it does so in a calculable way with no uncertainty. This ultimately means no source of entropy or randomness inside the blockchain ecosystem.

Example

A real-world example of the Entropy Illusion vulnerability is the case of the Android Bitcoin Wallet, which was found to use a predictable source of entropy for generating Bitcoin addresses. This made it possible for attackers to predict the addresses generated by the wallet and steal the bitcoins associated with those addresses.

Source: https://arstechnica.com/information-technology/2015/05/crypto-flaws-in-blockchain-android-app-sent-bitcoins-to-the-wrong-address/

Mitigation

To mitigate the Entropy Illusion vulnerability, developers should ensure that their blockchain applications use enough random number generators or trusted external sources of randomness. Additionally, developers should use well-established cryptographic libraries to generate secure random numbers. It is also recommended to periodically review the source code of blockchain applications to identify potential vulnerabilities and to apply security patches promptly.

Delegate call

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is a “Delegate call” vulnerability?

The delegate call vulnerability is a vulnerability in smart contracts on the Ethereum blockchain that allows attackers to call a function in another contract with all of the calling contract's context, including the contract's storage, balance, and code. This vulnerability can allow attackers to take control of a contract or steal funds from it by exploiting the trust relationship between contracts.

The vulnerability arises because of the delegatecall() function, which can be used to call a function in another contract and is commonly used to implement libraries in Solidity. However, if the input data is not properly validated, an attacker can execute malicious code and take control of the calling contract.

Example

Parity multi-sig (wallet hack)

The second Parity multi-sig wallet hack is an example of how the context of well-written library code can be exploited if run in its non-intended context. Several good explanations of this hack exist, such as this overview: Parity MultiSig Hacked. Again by Anthony Akentiev, this stack exchange question and An In-Depth Look at the Parity Multisig Bug.

Mitigation

“Solidity provides the library keyword for implementing library contracts (see the Solidity Docs for further details). This ensures the library contract is stateless and non-self-destructable. Forcing libraries to be stateless mitigates the complexities of storage context demonstrated in this section. Stateless libraries also prevent attacks whereby attackers modify the state of the library directly to affect the contracts that depend on the library's code. As a general rule of thumb, when using DELEGATECALL pay careful attention to the possible calling context of both the library contract and the calling contract, and whenever possible, build state-less libraries.”

Source: https://blog.sigmaprime.io/solidity-security.html#dc-example

Default Visibility

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is Default visibility?

Default visibilities vulnerability in Web3 refers to an exposure caused by the lack of access modifiers in Solidity contracts, which can lead to unexpected behavior and potentially malicious actions.

By default, Solidity contract functions have a public visibility level, meaning anyone can call them. This can lead to unintentional actions, such as transferring funds or modifying data, by anyone interacting with the contract.

For example, if a contract has a function that transfers funds to a specified address and it is set to public visibility, anyone can call this function and transfer funds to any address they choose. This can result in losing funds for the contract owner or users.

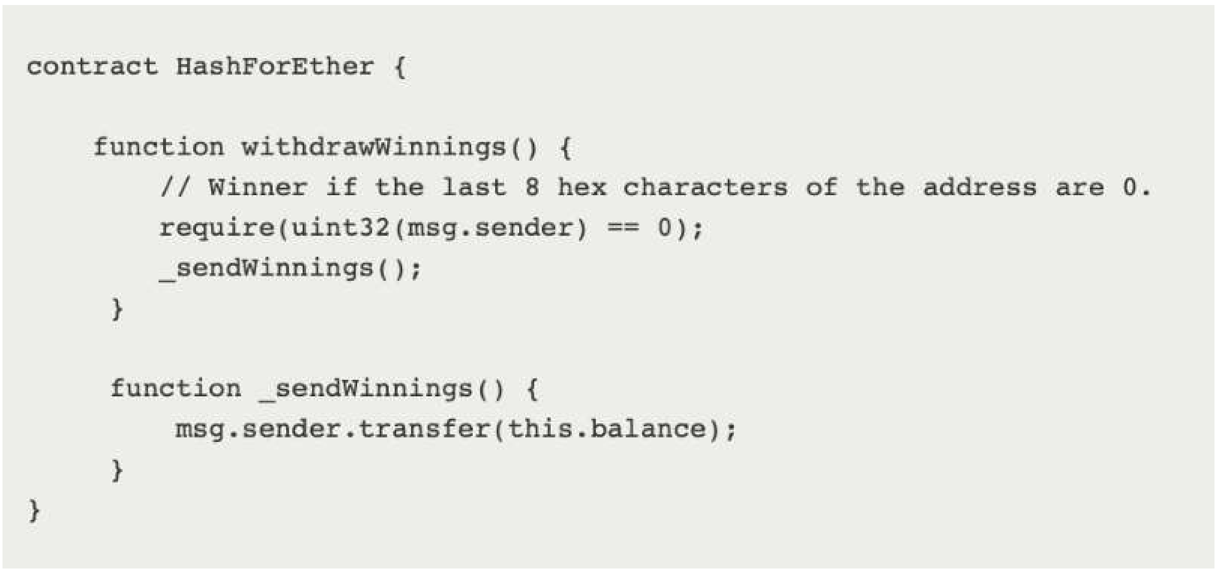
Example

The first Parity multi-sig hack

In the first Parity multi-sig hack, about $31M worth of Ether was stolen from primarily three wallets. A good recap of exactly how this was done is given by Haseeb Qureshi in this post.

Mitigation

To mitigate this vulnerability, it is recommended to use access modifiers such as private, internal, and external to control the visibility and accessibility of functions and variables within a contract. Setting appropriate access levels can greatly reduce the potential for unexpected behavior and malicious actions.



```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0.
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

Denial of Service (DoS)

Tag: Execution

Category: X

What is Denial of Service attacks?

A denial of Service (DoS) attack is a type of cyber attack that aims to disrupt the normal functioning of a website or network by overwhelming it with a flood of traffic or requests, rendering it inaccessible to legitimate users. In the context of web3, DoS attacks can take several forms, including DoS with (Unexpected) revert and DoS with Block Gas Limit.

DoS with (Unexpected) revert occurs when an attacker intentionally triggers a function to fail with a revert message, which causes the transaction to consume all the gas allocated to it without achieving its intended purpose. As a result, the remaining transactions in the block fail to execute, leading to a denial of service. This attack can also be launched by exploiting vulnerabilities in the contract code, which allows the attacker to consume all the gas in the block without providing any value to the network.

DoS with Block Gas Limit is a type of DoS attack where an attacker exploits the block gas limit to consume more resources than required, thereby preventing other transactions from being processed. The attacker can achieve this by either submitting transactions with high gas prices or creating many transactions that consume more gas than the block limit.

Distributed Denial of Service (DDoS) is another DoS attack involving the attacker controlling multiple devices to launch an attack on the target node. The attacker observes the target node and channels the multiple devices under his control to send a large amount of information, flooding the target node. This makes the target crash and unable to fulfill the specified task.

Example

Ethereum experienced a DDoS attack, where transactions were "spamming" the network. "Ethereum developers are hard at work on a patch, and the attack already costs the hacker about $4.50 per minute. The attack was successful insofar as it slowed down transactions and made the price of ether drop, but other than that, the network is proving resilient."

Source: https://www.inverse.com/article/21310-ethereum-ddos-cryptocurrency-hackers

Mitigation

There are several ways to mitigate DoS attacks in web3. First of all, it depends on what is being attacked. It can be a blockchain network in itself, like the example. It can also be a simple website experiencing a DoS attack due to bots spamming the website and causing a server overload. This is usually prevented with Captcha-like limitations.

Developers can implement rate-limiting techniques, such as limiting the number of requests per second, to prevent attackers from flooding any network with requests. Network-level solutions, such as load balancers and firewalls, can also be implemented to filter out malicious traffic and prevent DoS attacks.

It is also essential to monitor traffic to spot irregularities.

Dependency Risks

Tag: Execution **Category: Smart Contract Vulnerabilities**

What are “Dependency Risks”?

In the context of smart contracts, dependency risk refers to the potential vulnerabilities that can be introduced into the smart contract code due to external dependencies such as libraries or APIs.

Smart contracts often rely on external dependencies to perform certain functions or access external resources such as external data feeds or other smart contracts. However, if these external dependencies are not properly secured or validated, they can introduce vulnerabilities in the smart contract code. For example, an attacker could exploit a vulnerability in an external library used by a smart contract to gain unauthorized access to the smart contract's funds or execute malicious code.

Example

1. Malicious Dependencies: This refers to using a malicious dependency by a smart contract. It can happen when a developer unknowingly uses a third-party library that contains malicious code, which can then be used to exploit the smart contract.
2. Versioning Issues: Versioning issues arise when a smart contract relies on a specific dependency version, which becomes deprecated or is no longer supported. If the developer doesn't update the dependency, it can lead to potential security vulnerabilities.
3. Conflicting Dependencies: Sometimes, different dependencies can have conflicting versions of the same library, which can cause issues in the smart contract. If the smart contract relies on these dependencies, it can lead to unexpected behavior or security vulnerabilities.
4. Package Management Issues: Smart contracts can have package management issues if they use a package manager that is not secure or is susceptible to attacks. Attackers can then inject malicious code into the package manager, which can then be used to exploit the smart contract.
5. Abandoned Dependencies: Sometimes, dependencies can become abandoned by the developer, meaning they are no longer maintained or updated. If the smart contract relies on these dependencies, it can lead to potential security vulnerabilities, as any issues or bugs in the dependency will not be addressed.

Mitigation

To mitigate dependency risk, smart contract developers should carefully vet and validate any external dependencies used in their code. They should also consider using secure coding practices such as input validation and defensive programming techniques to prevent potential attacks. Additionally, developers should regularly monitor and update their dependencies to promptly address any vulnerabilities or security issues.

Unchecked Return Values

Tag: Execution

Category: Smart Contract Vulnerabilities

What is Unchecked Return Values?

Unchecked return values are a vulnerability category within the "Exploitation" stage of the attack lifecycle. This vulnerability occurs when a smart contract function call returns a value, but the calling contract fails to verify or use the returned value, leaving it unchecked.

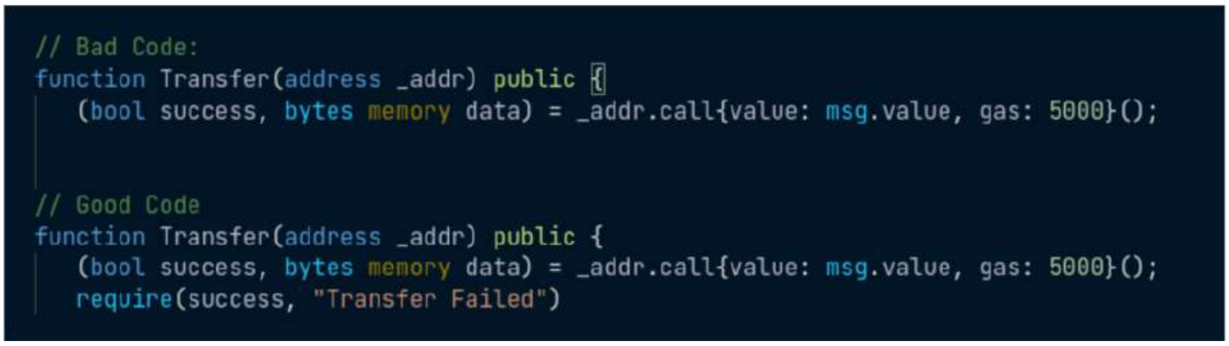
There are several ways of performing external calls in Solidity. Sending ether to external accounts is commonly performed via the transfer method. However, the send function can also be used, and for more versatile external calls, the CALL opcode can be directly employed in Solidity.

You can send Ether to other contracts by

- transfer (2300 gas, throws an error)
- send (2300 gas, returns bool)
- call (forward all gas or set gas, returns bool)

Here we can see that when we use to send or call to send ether or perform any transactions, it returns a boolean value i.e. true or false.

The call and send functions to return a Boolean indicating whether the call succeeded or failed. As a result, if the call return value is not checked, execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behavior in the subsequent program logic.



```
// Bad Code:
function Transfer(address _addr) public {
    (bool success, bytes memory data) = _addr.call{value: msg.value, gas: 5000}();


// Good Code
function Transfer(address _addr) public {
    (bool success, bytes memory data) = _addr.call{value: msg.value, gas: 5000}();
    require(success, "Transfer Failed")
```

In the above code, you can see that there is a Transfer function that uses a call method to transfer the amount. In the first case, it doesn't check for the return value, where there is no error handling if the transfer fails.

In the second one, there is a check for the call's return value. If the call fails it will revert with a "transfer failed" message.

Real-World Example

When sending ETH from one contract to another, like from the King of the Ether contract to an Ethereum Mist "contract-based wallet" contract, it's possible for the transfer to fail if implemented in the "obvious" way in the Solidity contract language due to insufficient gas.

This resulted in failed transfers from the Kings of Ether contract to users. Without any checks for the call return value, a failed transaction was recorded as a completed transaction in the contract.

Source 1: KotET - Post-Mortem Investigation During the 'Turbulent Age' (06 Feb 2016 to 08 Feb 2016) of the King of the Ether Throne, a serious issue caused some...

www.kingoftheether.com

King of the Ether

Etherpot

Source 2: https://sm4rty.medium.com/unchecked-call-return-value-solidity-security-1-fe794a7cdb6f

Mitigation

If send or call is used, Always make sure to handle the possibility that the call will fail, by checking the return value.

To mitigate this vulnerability, developers should ensure that their smart contracts properly handle and verify all return values. This includes checking for errors and verifying that the expected value was returned before proceeding with further actions. Additionally, developers should use tools such as static analysis and code reviews to identify and address potential unchecked return value vulnerabilities before deploying smart contracts

Bad Randomness

Tag: Execution

Category: Logic

What is bad randomness?

In Web3, "bad randomness" refers to the lack of or weakness in random number generation in smart contracts, making them vulnerable to attacks. A smart contract's functionality may depend on generating random numbers, for example, in gambling or other games that rely on chance.

If a smart contract's random number generation algorithm is not implemented correctly, an attacker can predict or manipulate it. For example, an attacker could identify patterns in generating random numbers and use this information to manipulate the outcome of a game or other transaction in their favor.

This vulnerability is categorized under the "Execution" phase because it can be exploited during the actual execution of the smart contract. To mitigate this vulnerability, it is important to use secure and unpredictable random number generation methods, such as using multiple sources of randomness or relying on trusted external sources for randomness.

Example

In the 2023 Cyvers Web3 security report, the Wintermute hack was analyzed. One alleged reason for the hack reason of Wintermute was due to the profanity vanity address (private key) generator.

Its design flaw enabled hackers to predict the outcome through enough computing force. This could be an example of “bad randomness” where hackers could return to the generator and re-compute the answer. It wasn't random enough and followed a pattern that could be “decrypted” through enough computing power.

Mitigation

Preventing "bad randomness" smart contract vulnerability can be challenging, as generating truly random numbers in a deterministic and transparent blockchain environment is difficult. However, there are several techniques and best practices that developers can follow to mitigate this vulnerability:

Use External Randomness Sources: Smart contracts can use external randomness sources to generate random numbers, such as the Oraclize service or a trusted decentralized random number generator like Chainlink VRF. These sources provide an additional layer of randomness that is difficult for attackers to predict or manipulate.

Avoid Using Block Information: Block information such as the block timestamp or block hash should not be used to generate random numbers, as miners can manipulate them. An attacker who knows the exact block information can generate a predictable outcome and manipulate the contract to their advantage.

Pseudorandom Number Generation: If external randomness sources are not available or practical, developers can use pseudorandom number generation techniques. Pseudorandom number generation uses a deterministic algorithm to generate a sequence of numbers that appears random but is repeatable. However, it is important to use a high-quality algorithm and a large enough seed to generate a truly unpredictable sequence.

Publicly Verifiable Randomness: Smart contracts should use publicly verifiable randomness techniques that allow anyone to verify the randomness of the generated number. This ensures that the generated number is not biased or manipulated and that the contract operates as intended.

Third-Party Auditing: Smart contracts should be audited by third-party security experts to identify and address any vulnerabilities, including bad randomness. This helps ensure that the contract is secure and operates as intended, and can prevent potential loss of funds due to vulnerabilities.

Time manipulation

Tag: Execution

Category: Smart Contract Vulnerabilities

What is Time Manipulation?

Time manipulation is a smart contract vulnerability that allows attackers to exploit a contract by manipulating the timestamps or block numbers. In a blockchain environment, timestamps and block numbers are crucial components of the consensus algorithm that ensures the integrity and immutability of the blockchain. In a smart contract, timestamps and block numbers determine when certain functions should be executed or funds should be unlocked.

An attacker can exploit this vulnerability by manipulating the timestamps or block numbers to trick the contract into unlocking funds before they are supposed to be available or accessing a specific function in the contract at a reasonable time. This can be especially dangerous in time-sensitive contracts, such as those that involve auctions or token sales.

Real-World Example

An old Ponzi scheme called GovernMental amassed a considerable quantity of ether. Moreover, it was open to timestamp-based attacks. The last player to join a round (for at least one minute) received payment per the contract terms. A miner who was a player might change the timestamp (to a future time to make it seem like a minute had passed), making it seem like they were the last to join for more than a minute (even though this is not true in reality).

More detail on this can be found in the History of Ethereum Security Vulnerabilities Post by Tanya Bahrynovska.

Mitigation

The following are some mitigation strategies that can be used to address the time manipulation vulnerability in smart contracts:

1. Use Relative Time: Instead of using absolute timestamps, smart contracts can use relative time to determine when certain functions should be executed or funds should be unlocked. This can prevent attackers from manipulating the timestamps to their advantage.
2. Block Verification: Smart contracts can verify the current block number and timestamp before executing certain functions or unlocking funds. This can prevent attackers from exploiting the contract using outdated or manipulated block numbers and timestamps.
3. Third-Party Libraries: Developers can use third-party libraries with secure timestamps and block number verification mechanisms. These libraries can help ensure the integrity and immutability of the blockchain and prevent attackers from exploiting vulnerabilities in smart contracts.

Integer overflow/Underflow

Tag: Execution

Category: Smart Contract Vulnerabilities

What is Integer underflow/overflow?

Underflow/overflow issues can occur in smart contracts when performing mathematical operations on integers without proper bounds checking. For example, if a smart contract subtracts a larger number from a smaller one, it can result in an underflow and unexpected results. Similarly, if a smart contract adds a number to a value already at the maximum limit of the variable, it can result in an overflow.

Attackers can exploit these types of vulnerabilities to manipulate the behavior of the smart contract and steal funds. Therefore, it is essential for smart contract developers to implement proper bounds checking and testing to prevent these types of issues.

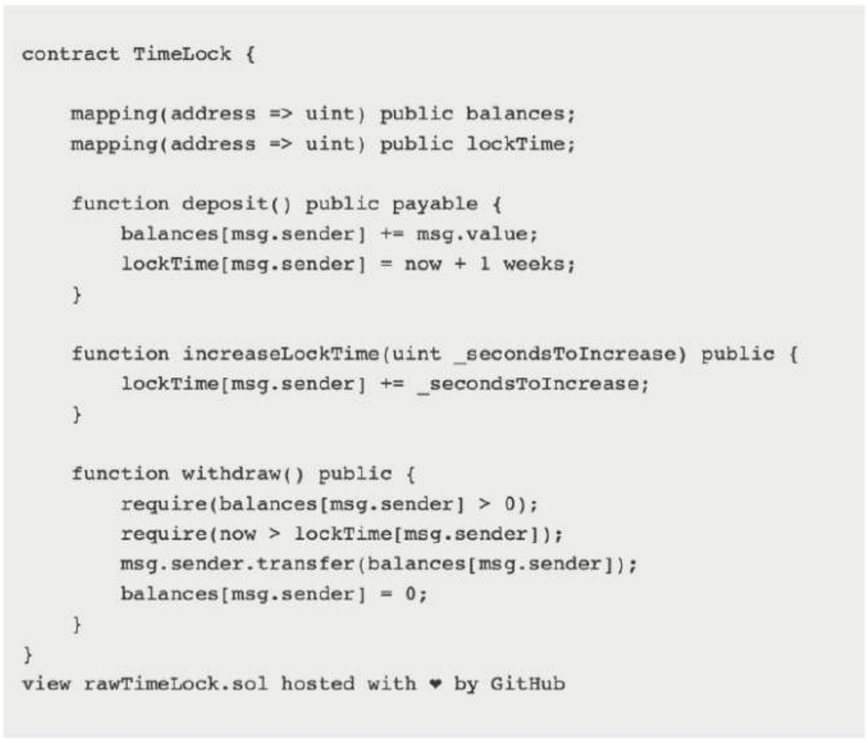
These are issues that usually get patched during audits.

In computer programming, an integer overflow/underflow occurs when an arithmetic operation on an integer exceeds the maximum or minimum value that the data type can represent. An overflow/underflow can cause unexpected behavior in a program, including incorrect results or program crashes. In the context of Web3, integer overflow/underflow can occur in smart contracts when a mathematical operation on a variable exceeds the maximum or minimum value that can be represented by its data type, potentially leading to incorrect results or even financial losses.

Example

For example, in a smart contract that manages a token, an integer overflow can occur when a user attempts to transfer more tokens than they have, causing the contract to interpret the integer value as a negative number and resulting in an unintended transfer of tokens.

TimeLock.sol



```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        msg.sender.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }
}
view rawTimeLock.sol hosted with ❤ by GitHub
```

This contract is designed to act like a time vault, where users can deposit ether into the contract and it will be locked there for at least a week. The user may extend the time longer than 1 week if they choose, but once deposited, the user can be sure their ether is locked in safely for at least a week. Or can they?...

In the event a user is forced to hand over their private key (think hostage situation) a contract such as this may be handy to ensure ether is unobtainable in short periods of time. If a user had locked in 100 ether in this contract and handed their keys over to an attacker, an attacker could use an overflow to receive the ether, regardless of the lockTime.

The attacker could determine the current lockTime for the address they now hold the key for (its a public variable). Let's call this userLockTime. They could then call the increaseLockTime function and pass as an argument the number 2^256 - userLockTime. This number would be added to the current userLockTime and cause an overflow, resetting lockTime[msg.sender] to 0. The attacker could then call the withdraw function to obtain their reward."

Mitigation

The (currently) conventional technique to guard against under/overflow vulnerabilities is to use or build mathematical libraries which replace the standard math operators; addition, subtraction, and multiplication (division is excluded as it doesn't cause over/underflows, and the EVM throws on division by 0).

OppenZepplin has done a great job building and auditing secure libraries, which the Ethereum community can leverage. In particular, their Safe Math Library is a reference or library to use to avoid under/overflow vulnerabilities.

To demonstrate how these libraries are used in Solidity, let us correct the TimeLock contract using Open Zepplin's SafeMathlibrary. The overflow-free contract would become:

Source: https://medium.com/hackernoon/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148

To mitigate the risk of integer overflow/underflow vulnerabilities in smart contracts, developers can follow best practices such as:

- Careful selection of variable data types: Developers should choose variable data types representing the maximum and minimum values required by the smart contract's operations.
- Use of SafeMath libraries: SafeMath is a library that provides safe arithmetic operations for uint variables in Solidity, the programming language used to write Ethereum smart contracts. This library ensures that arithmetic operations do not result in integer overflow or underflow.
- Code review / Audits: Smart contracts should be thoroughly audited by experienced developers and security experts to identify and mitigate potential vulnerabilities, including integer overflow/underflow issues.
- Testing: Smart contracts should be tested extensively to ensure they function as intended and do not contain any vulnerabilities, including integer overflow/underflow vulnerabilities.

Access Control Issues

Tag: Execution **Category: Smart Contract Vulnerabilities**

What is Access control?

Access control issues refer to a type of security vulnerability that occurs when inadequate controls or restrictions exist on who can access and modify certain resources or data within a system. This vulnerability can occur in various system areas, including user accounts, databases, APIs, and smart contracts.

Example

In the context of smart contracts, access control issues can arise when there need to be more restrictions on who can execute certain functions or modify the state of the contract. For example, suppose a smart contract needs proper access controls. In that case, a malicious actor may be able to manipulate the contract's data or execute unauthorized functions, leading to various types of attacks, such as theft of funds or unauthorized data access.

Access control issues can also arise in decentralized applications (dApps) that rely on smart contracts. In these cases, the issue may be related to the dApp's user interface, which could allow malicious actors to bypass certain access controls or execute unauthorized functions within the smart contract.

Mitigation

To prevent access control issues, it is essential to implement proper authentication and authorization mechanisms that limit access to sensitive resources and ensure that only authorized users can execute certain functions or modify certain data. This includes implementing multi-factor authentication, role-based access control, and secure coding practices when developing smart contracts and dApps.

Similar to “15.1 validator privileges,” - Inadequate access control smart contracts give hackers access through the lack of restrictions in updating the smart contract state.

Logic Bombs

Tag: Execution

Category: Smart Contract Vulnerabilities

What is "Logic Bombs" in Web3?

Deployment of malicious smart contract contract

Logic bombs" refer to malicious code (smart contracts) or programs intentionally inserted into software or system by a hacker to execute a harmful action when a specific trigger condition is met.

In the context of Web3, a logic bomb can be a type of smart contract vulnerability where a certain piece of code is designed to execute an attack or steal funds when a particular condition is met, such as when a specific date or time is reached, or when a particular transaction occurs. For example, a malicious actor could create a smart contract with a logic bomb triggered when a specific user or address interacts with the contract, allowing the attacker to steal funds from that user.

Example

Logic bombs" can be deployed in various ways, including through a hacker's own created smart contract. It involves inserting malicious code into a program or smart contract that will execute when certain conditions are met, such as a specific date, time, or event. The code can then carry out malicious actions, such as stealing funds or causing the smart contract to behave unexpectedly.

A malicious smart contract that interacts with a dApp smart contract can be labeled as a logic bomb if designed to carry out a destructive action at a specific time or under specific conditions.

For example, a malicious smart contract that appears to provide a legitimate service but is designed to trigger a destructive action when a certain condition is met, such as when a specific address interacts with the contract, could be considered a logic bomb.

Mitigation

To mitigate the risk of logic bombs in Web3, it is most important to monitor smart contracts proactively with real-time monitoring. This will enable developers and protocol founders to detect malicious deployments and interactions, thereby giving them a chance to detect to prevent devastating damage.

Several on-chain & real-time monitoring solutions exist today, like Cyvers, Forta & Lossless.

Reentrancy

Tag: Execution

Category: Smart Contract Vulnerabilities

What is a reentrancy attack?

A reentrancy attack is a vulnerability that can occur in a smart contract running on a blockchain platform. It happens when an attacker exploits a flaw in the smart contract's code to repeatedly call back into the contract before the previous invocation has been completed. This allows the attacker to drain the contract's funds or manipulate its state.

Example

In 2016, the DAO attack was an example of this hack. An attacker exploited a vulnerability in the DAO smart contract by calling a function that had a recursive call. This recursive call led to an overflow of the attacker's account balance, resulting in the attacker being able to withdraw Ether from the DAO's funds. One example of the Unchecked Call Return Value hack is the infamous DAO attack on the Ethereum blockchain in 2016.

The DAO was a decentralized autonomous organization that aimed to operate as a venture capital fund for the blockchain industry. It raised over $150 million in Ether (ETH) through an initial coin offering (ICO). However, a vulnerability in the smart contract allowed an attacker to drain one-third of the funds, amounting to about $50 million in ETH.

The attacker used a combination of a reentrancy attack and the Unchecked Call Return Value vulnerability to exploit the smart contract. They used the DAO's function that allowed users to split their tokens and withdraw their share of the funds. However, the attacker created a recursive call loop by reentering the same function multiple times.

Additionally, the function call used to withdraw the funds did not check the return value of the recursive call. This allowed the attacker to repeatedly drain the funds until they could steal significant ETH from the DAO.

The DAO attack was a significant event in the history of blockchain technology, and it led to the Ethereum community hard-forking the blockchain to recover the stolen funds. The incident also highlighted the importance of conducting thorough security audits and testing smart contracts to identify and mitigate vulnerabilities like the Unchecked Call Return Value hack.

To prevent this type of attack, it is important to ensure that smart contracts validate the return value of every function call and that they implement proper exception-handling mechanisms to handle unexpected return values. Additionally, developers should follow best practices for smart contract development to minimize the risk of vulnerabilities and attacks on the blockchain.

Mitigation

To mitigate the risk of reentrancy attacks, developers must carefully design and test their smart contracts. Some specific measures that can be taken to prevent these attacks include:

Implementing checks on the state of the contract before and after each call to prevent reentry

Using mutex locks to prevent concurrent calls to the same function

Limiting the amount of Ether that can be withdrawn from the contract at any one time

Avoiding calling external contracts or functions within a smart contract, if possible

Implementing fail-safes and emergency stop mechanisms to prevent significant losses in the event of an attack

By taking these steps and staying informed about the latest security best practices, developers can help protect their smart contracts and the users who rely on them.

Sources

https://consensys.github.io/smart-contract-best-practices/known_attacks/#unchecked-call-return

http://www.web3isgoinggreat.com/

Unexpected Ether

Tag: Execution

Category: Smart Contract Vulnerabilities

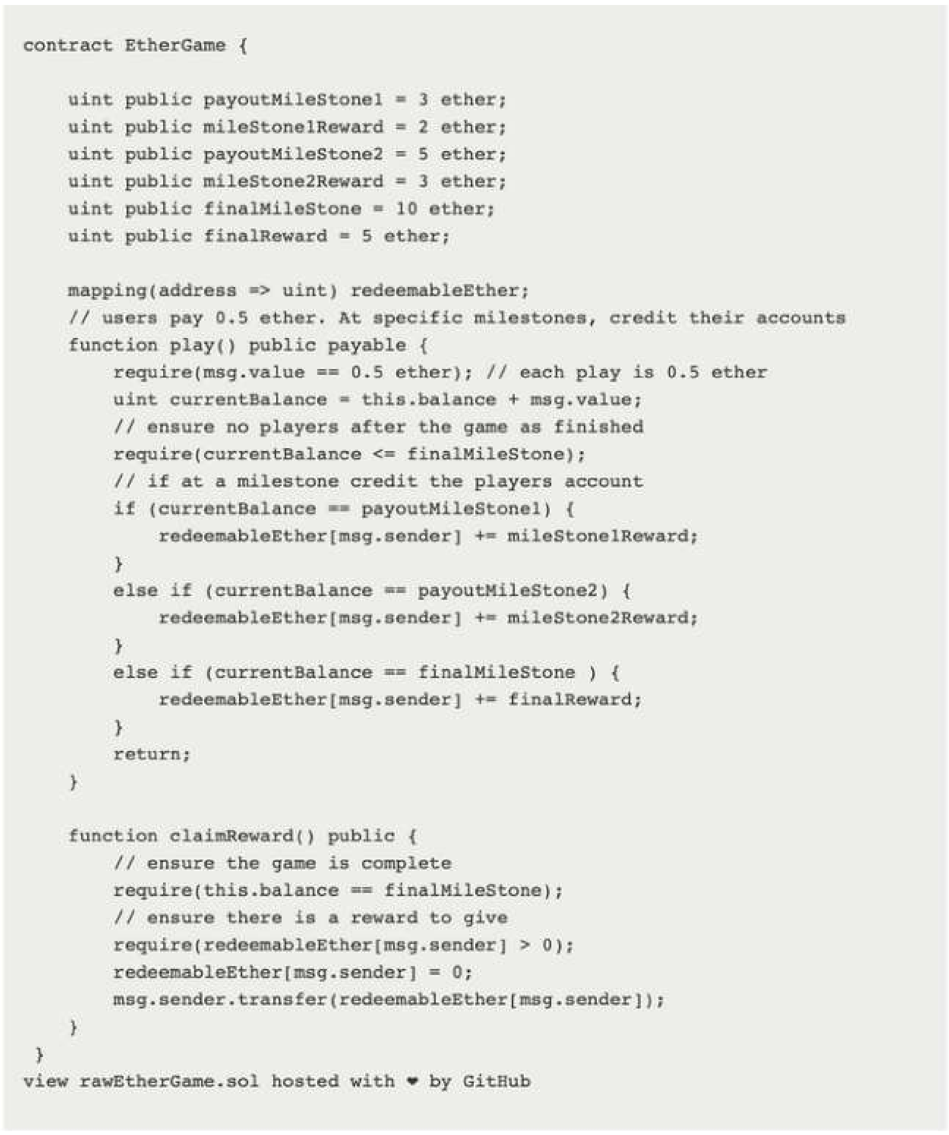
What is "Unexpected Ether"?

The vulnerability known as "Unexpected Ether" is a reentrancy attack that can occur in smart contracts. It happens when a smart contract receives Ether as payment and calls an external contract in the same transaction without updating its state beforehand. An attacker can exploit this vulnerability by calling a malicious contract that triggers a reentrancy attack, causing the original contract to send unexpected amounts of Ether to the attacker's address.

The "unexpected ether" vulnerability is actually a type of re-entrancy attack. A malicious contract takes advantage of a vulnerable contract with a recursive call pattern. The malicious contract calls the vulnerable contract and recursively calls itself before the vulnerable contract can complete its execution. This allows the malicious contract to repeatedly withdraw ether from the vulnerable contract, leading to unexpected ether balance reductions. The malicious contract "re-enters" the vulnerable contract multiple times, exploiting its recursive call pattern.

Real-World Example:

EtherGame.sol



```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
 }
view rawEtherGame.sol hosted with ❤ by GitHub
```

“This contract represents a simple game (which would naturally invoke race-conditions) whereby players send 0.5 etherquanta to the contract in hope to be the player that reaches one of three milestones first. Milestone's are denominated in ether. The first to reach the milestone may claim a portion of the ether when the game has ended. The game ends when the final milestone (10 ether) is reached and users can claim their rewards.

The issues with the EtherGame contract come from the poor use of this.balance in both lines [14] (and by association [16]) and [32]. A mischievous attacker could forcibly send a small amount of ether, let's say 0.1 ether via the selfdestruct()function (discussed above) to prevent any future players from reaching a milestone. As all legitimate players can only send 0.5 ether increments, this.balance would no longer be half integer numbers, as it would also have the 0.1 ethercontribution. This prevents all the if conditions on lines [18], [21] and [24] from being true.

Even worse, a vengeful attacker who missed a milestone, could forcibly send 10 ether (or an equivalent amount of ether that pushes the contract's balance above the finalMileStone) which would lock all rewards in the contract forever. This is because the claimReward() function will always revert, due to the require on line [32] (i.e. this.balance is greater than finalMileStone).”

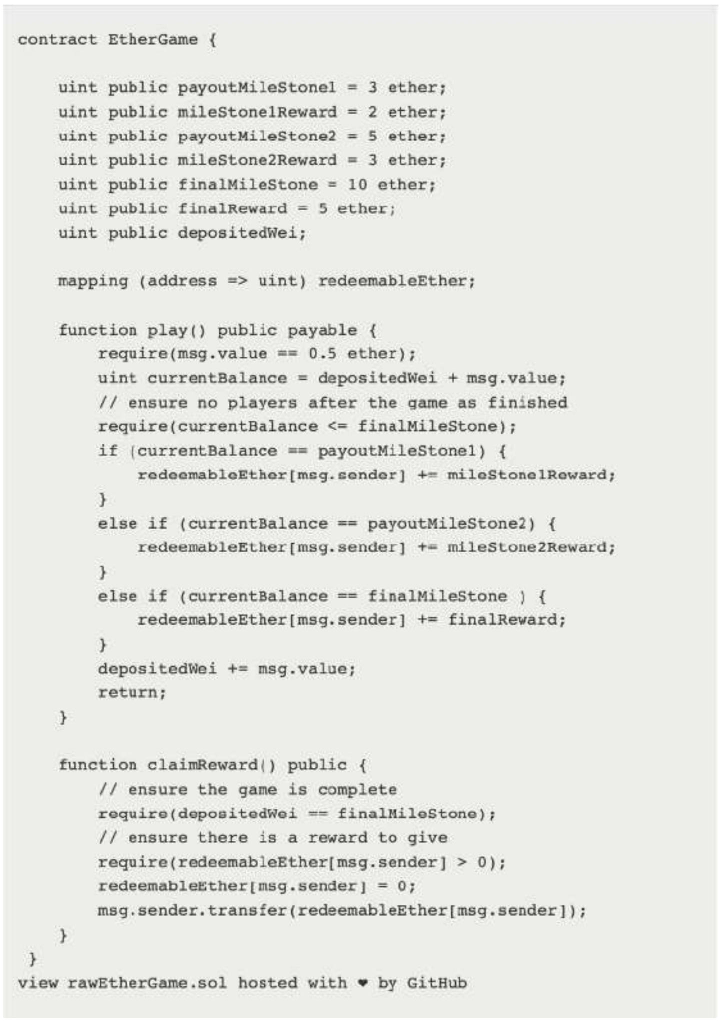
Source: https://medium.com/hackernoon/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148

Mitigation

This vulnerability typically arises from the misuse of this.balance. Contract logic, when possible, should avoid being dependent on the exact values of the balance of the contract because it can be artificially manipulated. If applying logic based on this.balance, ensure to account for unexpected balances.

If exact values of deposited ether are required, a self-defined variable that gets incremented in payable functions should be used to track the deposited ether safely. This variable will not be influenced by the forced ether sent via a selfdestruct() call.

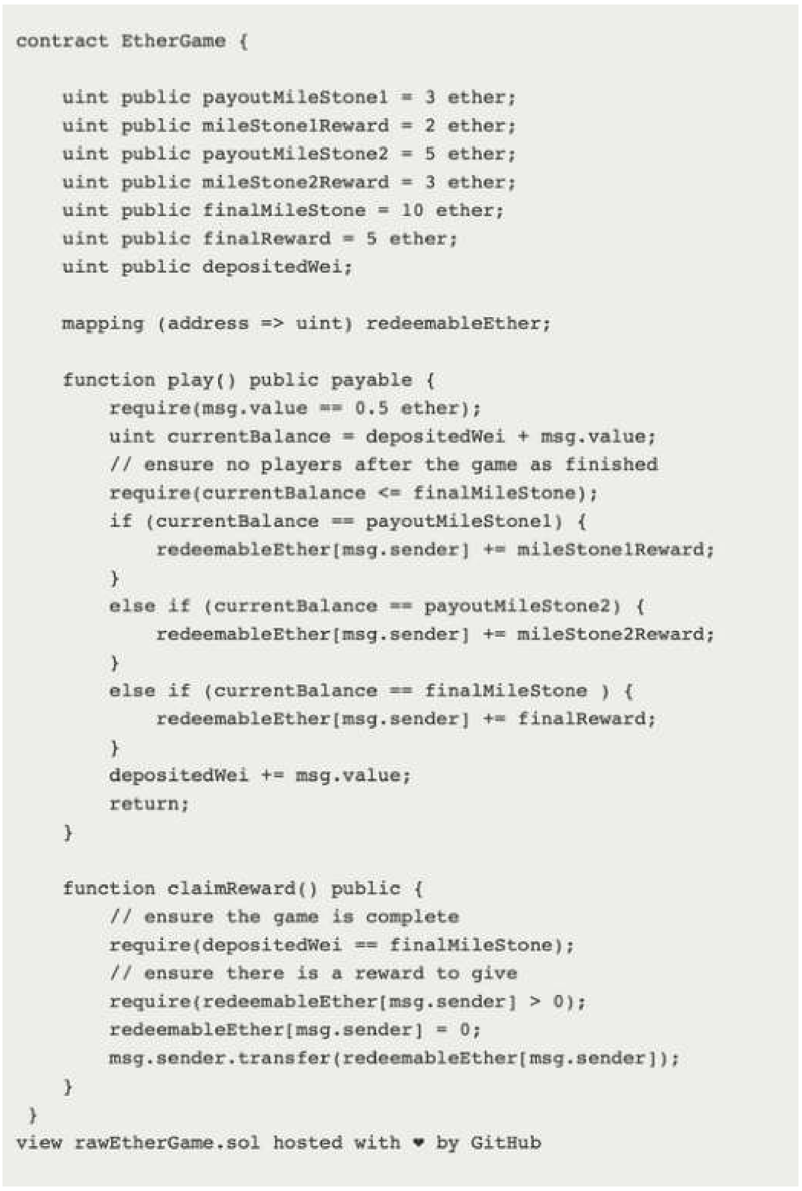
With this in mind, a corrected version of the EtherGame contract could look like this:



```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemableEther;

    function play() public payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(depositedWei == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
view rawEtherGame.sol hosted with ❤ by GitHub
```

Here, we have just created a new variable, depositedEther which keeps track of the known ether deposited, and it is this variable to which we perform our requirements and tests. Notice, that we no longer have any reference to this.balance."

Mitigating the "Unexpected Ether" vulnerability involves ensuring that smart contracts are designed to update their own state before calling any external contracts in the same transaction. This can be done by using the "checks-effects-interactions" pattern, which involves first checking that all the conditions for the transaction are met, then updating the contract's state, and finally interacting with external contracts. Additionally, developers should ensure that their contracts have proper access control mechanisms in place to prevent unauthorized access to the contract's funds. By following these best practices, developers can significantly reduce the risk of unexpected ether attacks in their smart contracts.



```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemableEther;

    function play() public payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(depositedWei == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
 }
view rawEtherGame.sol hosted with ❤ by GitHub
```

State Variable Default Visibility Vulnerability

Tag: Execution

Category: Smart Contract Vulnerabilities

What is state Variable Default Visibility Vulnerability?

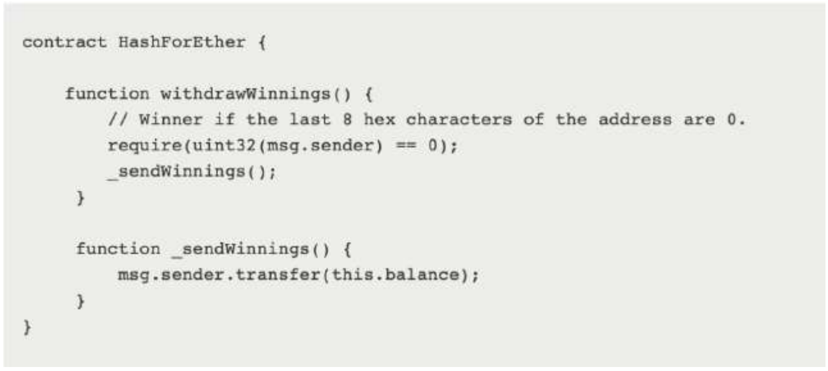
State Variable Default Visibility Vulnerability is a type of vulnerability in smart contracts that occurs due to the default visibility of state variables. In Solidity, state variables have internal visibility by default, meaning that they can be accessed by other functions within the same contract but not by functions in other contracts. However, if a developer forgets to explicitly specify the visibility of a state variable, it can become publicly visible, which could lead to unintended consequences.

In Solidity, functions have visibility specifiers that limit how they can be called. A function's visibility specifies whether it can be called only internally, only externally, by users, by other derived contracts, or only internally. The Solidity Docs offer a detailed explanation of the four visibility specifiers. The default visibility setting for a function is "public," allowing external calls by other users. This section will explore various devastating vulnerabilities in smart contracts that may result from improper usage of visibility specifiers.

Example

Functions have a public visibility setting by default. The result is that external users will be allowed to invoke functions that do not indicate any visibility. The issue is that developers may overlook visibility specifiers on functions that should be private (or only callable within the contract itself).

Let's look at a simple example:



```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0.
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

view rawHashForEther.sol hosted with by GitHub

This simple contract is intended to function as a guess-the-address bounty game. A user must create an Ethereum address with the last 8 hex characters set to 0, in order to win the contract's balance. After they have it, they can use the WithdrawWinnings() function to get their reward. However, nothing has been said about how visible the functions will be. In particular, the function _sendWinnings() is public, allowing any address to use it to steal the payout.

Mitigation

Even if a function is intended to be publicly accessible, it is best practice to always declare the visibility of the function in a contract. To promote this practice, Solidity's most recent releases will now display warnings during compilation for functions that do not explicitly set their visibility.

51% attack

Tag: Execution **Category: Higher Privilige Attacks**

What is a "51% attack"?

A "51% attack" is an attack on a blockchain network. It occurs when an attacker gains control of more than 51% of the network's hash rate, which allows them to add new blocks to the chain faster than the rest of the network. This can result in the attacker being able to reverse transactions, double-spend coins, and potentially take control of the network. The attacker can effectively gain control over the network by creating a longer chain that invalidates previous transactions. A 51% attack also decreases the integrity of the blockchain and, therefore, can also be placed within “Impact”.

Validators or miners in a blockchain network compete to add new blocks to the chain by solving complex cryptographic puzzles. The first validator to solve the puzzle and add the block to the chain is rewarded with cryptocurrency.

Validators or miners are responsible for verifying and adding new blocks. If an attacker gains control over most of the network's computational power, they can reverse previous transactions and double-spend coins. This can lead to a loss of trust in the network and significant financial damage to users.

Example

Bitcoin underwent a 51% attack, which resulted in the creation of Bitcoin Cash. The attack occurred due to a disagreement within the early bitcoin community called the Block Wars.

Mitigation

There are several ways to reduce the risk of a 51% attack on a blockchain network:

1. Encourage decentralization: The community can make the network more decentralized by encouraging more participants to become validators or miners. This makes it more difficult for a single entity to gain control of the majority of the network's hash rate.
2. Implement consensus mechanisms: Consensus mechanisms like Proof-of-Stake (PoS) or Delegated Proof-of-Stake (DPoS) can help reduce the risk of a 51% attack. They require validators or miners to have a stake in the network.
3. Implement network monitoring: Network monitoring is essential to detect and respond to suspicious activity, including potential 51% attacks.

In summary, the best way to reduce the risk of a 51% attack is to encourage decentralization, implement consensus mechanisms, conduct regular audits and updates, implement network monitoring, promote diversity in mining hardware, and use checkpointing to protect past transactions. By taking a comprehensive approach to security, it is possible to reduce the risk of a 51% attack and preserve the integrity of the network.

Command & Control

What is “Command and Control”?

Malicious actors use various techniques to gain control of validators, smart contracts, or other factors in a network. These techniques are collectively known as command and control (C2). The attacker communicates with an already compromised system to take control of it.

To avoid detection, malicious actors often mimic typical, expected communication patterns. Depending on the network architecture and security measures of the victim, an adversary can establish command and control in different ways and with varying levels of stealth.

C2 is distinct from other subcategories in the framework because it targets an attack's communication and control aspects. While attackers may use other subcategories, such as "Execution" or "Persistence," to achieve their objectives, C2 controls the attack remotely.

Once an attacker gains initial access to a system or network, the next step is establishing a connection between the attacker's command and control infrastructure and the compromised system. This allows the attacker to issue commands, exfiltrate data (which, in this case, is assets), and execute other malicious activities on the compromised system.

However, it's worth noting that the different phases of an attack can often overlap and occur simultaneously. For example, an attacker may use C2 to perform reconnaissance or escalate privileges, which could also be part of the attack's initial "Exploitation" phase.

Overall, the C2 subcategory focuses on an attack's communication and control aspects rather than the attacker's initial entry point or exploitation technique. Defending against C2 attacks requires strong security controls and monitoring for unusual network traffic or communication with suspicious domains or IP addresses.

The C2 subcategory comprises many techniques attackers utilize to gain and maintain control over compromised systems. These techniques may include command and control servers, domain fronting, and peer-to-peer (P2P) communication channels. By employing these methods, attackers can remain undetected by traditional security controls and maintain control over compromised systems.

Left out:

"Command and Control" refers to the technique attackers use to remotely manage and control compromised systems or networks. C2 can be used to issue commands, exfiltrate data, and execute other malicious activities.

Command & Control

Forged address phishing

Tag: Command and Control **Category: Higher Privilige Attacks**

What are “fake or compromised validator nodes”?

Fake or compromised validator nodes attack a blockchain network where malicious actors create fake nodes that appear legitimate validators. These nodes can then be used to gain control of the blockchain network and carry out various attacks, such as injecting fake transactions, censoring valid transactions, or manipulating the consensus mechanism.
Malicious actors can create fake validator nodes to gain control of a blockchain network. They can use these nodes to inject fake transactions, censor valid transactions, or manipulate the blockchain's consensus mechanism.
Creating fake validator nodes on a blockchain network can involve the attacker owning a significant amount of the cryptocurrency or asset associated with that network, which they can use to purchase the necessary equipment and set up the validator nodes. However, it is only sometimes needed for the attacker to own the asset or currency to set up the fake validator nodes.

The possibility of the attacker getting slashed for malicious activity depends on the specific blockchain network's consensus mechanism and governance model. Some blockchain networks have penalty mechanisms in place, where malicious behavior by a validator node can result in the node being removed from the network or having a portion of its stake or rewards slashed. However, it is possible that the attacker could evade such penalties by disguising their malicious activity or using a decentralized governance model where there is no central authority to enforce penalties.
It is important to note that creating fake validator nodes can cause significant harm to the blockchain network and its users.

Example

An attacker might create many fake validator nodes and use them to gain most of the votes in a proof-of-stake (PoS) consensus mechanism. This would allow the attacker to control the validation process and potentially carry out attacks such as double-spending or reorganizing the blockchain.

Mitigation

To mitigate this type of attack, blockchain networks can implement various security measures such as:
KYV (Know Your Validator): Validating the identity of all validators to ensure they are legitimate.

Multi-party computation: This involves breaking up sensitive data and computations into multiple parts, each processed by different validators, to prevent any single validator from having complete control.

Decentralized Governance: Implementing a governance model that allows for community decision-making and voting rights.

Security Audits: Conducting regular security audits to identify and address vulnerabilities in the blockchain network.

Consensus Mechanism Diversity: Using multiple consensus mechanisms that work together to provide stronger security and resilience against attacks.

Botnets

Tag: Command and Control

Category: Infrastructure

What are Botnets?

Botnets are networks of infected devices controlled by a single attacker. In Web3, botnets can launch Distributed Denial of Service (DDoS) attacks on blockchain-based networks, disrupting their operations and potentially causing financial losses to their users.

Botnets are a network of compromised devices, typically controlled by a single attacker or group of attackers, that can be used to conduct malicious activities such as spamming, distributed denial-of-service attacks, and data theft. In the context of the command and control section of the framework for Web3, botnets are often used to control the operation of malicious software on compromised devices.

Example

A common example of a botnet involves an attacker infecting many devices with malware, which allows the attacker to take control of the devices and use them to conduct malicious activities. The infected devices can be used to carry out distributed denial-of-service attacks, send spam emails, or steal sensitive data. The attacker can use a command and control (C2) server to communicate with the infected devices, issuing commands to carry out specific tasks or to receive information from the compromised devices.

Mitigation

Mitigating the threat botnets poses requires a combination of technical and non-technical measures. Technical measures include implementing network security controls such as firewalls and intrusion detection/prevention systems, using anti-malware software to detect and remove malware infections, and configuring systems to block traffic to known C2 servers.

Non-technical measures include educating users on identifying and avoiding attacks, ensuring that software and operating systems are kept up-to-date with security patches, and implementing strict access control policies to limit the damage caused by a compromised account or device.

DNS Firewall Threat Feeds can be used to choke botnets and automatically prevent users from accessing malware dropper and phishing sites. Additionally, implementing IP address restrictions using Classless Inter-Domain Routing (CIDR) notation can help to block traffic from known malicious IP addresses and ranges. Another possible mitigation strategy is to implement process mitigations such as Data Execution Prevention (DEP), which can help to prevent buffer overrun exploitation by marking certain regions of memory as non-executable.

In summary, botnets pose a significant threat in the context of the command and control section of the framework for Web3. Combating this threat requires a combination of technical and non-technical measures, including network security controls, anti-malware software, access control policies, and user education. Implementing process mitigations and IP address restrictions can also be effective strategies for blocking traffic from known malicious sources.

Persistence

What is a "Contract Ownership Change"?

Persistence consists of techniques malicious parties use to keep their access to networks and protocols across attempted actions to keep them out—changed credentials, network changes, and other interruptions that could cut off their access. Any access, action, or configuration modifications that enable them to keep a firm grip on systems, such as swapping out or hijacking programs or including startup code, are considered persistence techniques.

In the context of Web3 and blockchain, persistence can be a critical issue. These systems are designed to be decentralized and trustless, meaning that they rely on cryptographic protocols and smart contracts to ensure their integrity and security. However, if attackers gain persistent access to a Web3 or blockchain network, they could subvert these protocols and compromise the system's integrity.

Some of the specific techniques that attackers may use to establish persistence in a Web3 or blockchain environment include:

1. Exploiting logic vulnerabilities in smart contracts: Smart contracts are self-executing programs that run on a blockchain. They can be vulnerable to various attacks, including buffer overflow and integer overflow attacks. An attacker who successfully exploits a logic vulnerability in a DApp could gain persistent access to multiple parts or funds of the DApp.
2. Compromising blockchain nodes: Blockchain nodes are the distributed computer network that maintains the blockchain ledger. If an attacker can compromise one or more nodes, they may be able to gain persistent access to the blockchain network and potentially modify the entire ledger itself.
3. In the case of PoW cryptocurrencies, getting enough mining/hash power (51%) would give that entity control to subvert and control the entire blockchain, potentially leading to a 51% attack and a fork of the entire blockchain.
4. Installing malware on user devices: Users of Web3 and blockchain systems typically interact with the network using a web browser or specialized software. If an attacker can install malware on a user's device, they may be able to gain persistent access to the network by having compromised a key contributor to the network itself.

Persistence

Contract Ownership Changes

Tag: Persistence

Category: Higher Privilige Attacks

What is a “Contract Ownership Change”?

Contract Ownership Change is a type of attack in the context of decentralized applications (dApps) built on the Ethereum blockchain or other Web3 platforms. A smart contract is a self-executing program that runs on the blockchain and can manage assets and transactions without a centralized authority. In this type of attack, an attacker changes the ownership of a smart contract, granting them full control over it. This can allow them to modify or destroy the contract, steal funds or data, or execute other malicious actions.

In this type of attack, the attacker gains control over the ownership of a smart contract, either by exploiting a vulnerability in the contract code or by gaining access to the private keys of the contract owner. Once the attacker becomes the contract owner, they can execute any contract function, including modifying its code, stealing funds or data, or destroying the contract entirely. The change of contract ownership enables the attacker to establish a real foothold within the DApp/protocol.

Example

Imagine a DApp that manages a decentralized exchange where users can trade cryptocurrencies. The smart contract that powers the exchange has a function that allows the contract owner to withdraw all the funds held in the exchange. If an attacker gains control over the contract ownership, they can call this function and steal all the funds stored in the exchange.

Mitigation

To mitigate Contract Ownership Changes attacks, DApp developers should follow security best practices when coding their contracts, such as using established security frameworks, conducting thorough code audits, and implementing multi-signature mechanisms for critical functions. Additionally, DApp users should be cautious when interacting with smart contracts and only use trusted applications thoroughly audited and reviewed by the community.

The best tip, in this case, would be to implement real-time and proactive monitoring of the contract owner wallet, which essentially is the key central access to the entire dApp/protocol. Real-time monitoring can prevent the entire ownership by alerting the owner contract DApp/protocol in real time. It could even front-run the entire transaction by detecting it in the mempool. Real-time monitoring can also, in this case, be used to prevent further harm once the attacker has managed to establish his foothold and gain access to the wallet.

Malicious Smart Contract Deployment

Tag: Persistence

Category: Malicious deployment

What are “Malicious Smart Contracts”?

Malicious smart contracts are code deployed on a blockchain platform that contains harmful functions or vulnerabilities. These contracts are designed to exploit weaknesses in dApps or the blockchain, performing unauthorized actions or causing unintended consequences. Once deployed, malicious smart contracts persist on the blockchain, allowing attackers to maintain control over the affected dApps or extract value from unsuspecting users. In this attack, an attacker injects malicious code into a smart contract or decentralized application. The code can be designed to steal funds, modify or destroy the contract, or execute other malicious actions. Once injected, the code can persist and execute even if the contract is upgraded or migrated.

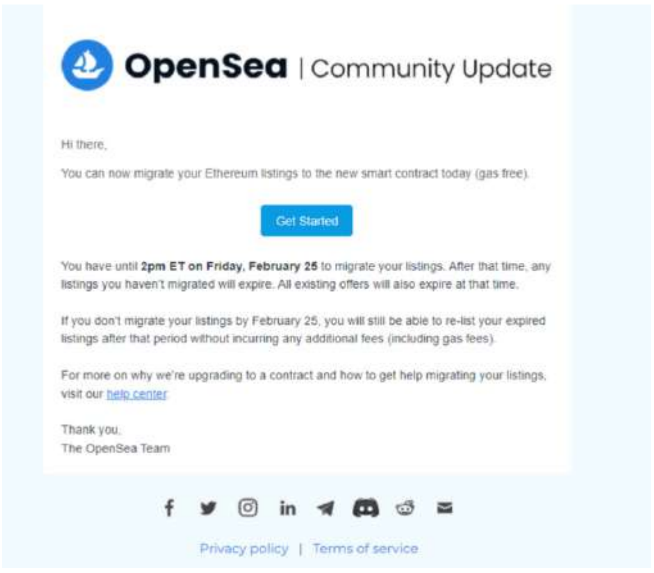
Malicious smart contracts are a growing concern in the Web3 ecosystem, as they can enable attackers to exploit vulnerabilities in decentralized applications (dApps) or blockchain platforms. These attacks can result in stolen funds, manipulated data, or disrupted operations. Due to their nature, malicious smart contracts are best placed under the "Persistence" tactic in the MITRE ATT&CK framework. They maintain their presence on the blockchain and can continuously execute malicious functions when triggered.

Example

Reentrancy attacks: The infamous DAO hack of 2016 is an example of a reentrancy attack where an attacker exploited a vulnerability in The DAO's smart contract, continuously withdrawing funds before the attack was detected and stopped.
Scams and phishing attacks: In the OpenSea phishing attack of February 2022, users were tricked into signing a malicious smart contract that transferred their NFTs to a hacker's address.

Flash loan attacks: In these attacks, hackers deploy malicious smart contracts that enable them to borrow and manipulate large amounts of cryptocurrency within a single transaction, exploiting vulnerabilities in decentralized finance (DeFi) platforms.

A famous example of one of these crypto smart contract scams was the $1.7m February 2022 OpenSea phishing attack.


OpenSea | Community Update

Hi there,

You can now migrate your Ethereum listings to the new smart contract today (gas free).

Get Started

You have until **2pm ET on Friday, February 25** to migrate your listings. After that time, any listings you haven't migrated will expire. All existing offers will also expire at that time.

If you don't migrate your listings by February 25, you will still be able to re-list your expired listings after that period without incurring any additional fees (including gas fees).

For more on why we're upgrading to a contract and how to get help migrating your listings, visit our help center

Thank you,
The OpenSea Team

Privacy policy | Terms of service

By opening up this phishing email, users were asked to sign a malicious smart contract that transferred all their NFTs to a hacker's address. This is a user-specific incident.
In many hacks, like flashloans, the hacker deploys malicious smart contracts, which enable the exploiter to execute transactions automatically.
Other malicious smart contract examples are when the hacker interacts with dApps and/ or protocol logic and exploits the vulnerability, tricking the smart contracts into thinking his malicious ones are real and original.
Another real-world example is the PAID Network, where the hack involved a malicious smart contract. Analysis to be found here: https://cryptoshine.medium.com/paid-contract-hack-deep-dive-4dd89e1414f5

Mitigation

To protect against malicious smart contracts, it is crucial to follow best practices for secure smart contract development and deployment. Some key strategies include:

Security reviews and testing: Perform thorough security audits, code reviews, and testing to identify vulnerabilities in smart contracts before deployment. This can help prevent the introduction of malicious code or exploitable weaknesses.

Implement access controls: Use access controls to restrict who can modify or interact with smart contracts, reducing the likelihood of unauthorized changes or exploitation.

Secure coding practices: Follow secure coding practices, such as input validation, sanitization, and proper error handling. Be aware of common smart contract vulnerabilities, like reentrancy attacks, and implement safeguards to mitigate them.

Monitoring and response: Implement real-time monitoring and automated alert systems to detect suspicious activity, such as unexpected changes to contract code or anomalous transaction patterns. Swiftly respond to identified threats to limit potential damage.

Education and awareness: Educate developers, users, and stakeholders about the risks associated with malicious smart contracts and the importance of following best practices for smart contract security.

By focusing on persistence and implementing these mitigation strategies, the Web3 ecosystem can better protect itself against the threat of malicious smart contracts and ensure the security of decentralized applications and blockchain platforms.

Proactive and real-time monitoring

Real-time monitoring can be a powerful tool in preventing the deployment of malicious contracts in web3. Monitoring smart contracts and dApps in real-time makes it possible to detect and respond to suspicious activity before it can cause harm. Here are some ways that real-time monitoring can help prevent the deployment of malicious contracts:

1. Detecting anomalous behavior: Real-time monitoring can help detect anomalous behavior in smart contracts and dApps. For example, sudden changes in transaction volume or activity patterns can indicate that a contract has been compromised or that an attacker is attempting to inject malicious code.

2. Identifying vulnerabilities: Real-time monitoring can help identify vulnerabilities in smart contracts and dApps. By identifying potential attack vectors and vulnerabilities, it is possible to address them before attackers can exploit them.

3. Alerting security teams: Real-time monitoring can provide real-time alerts when suspicious activity is detected. This can allow teams to respond quickly and prevent the deployment of malicious contracts or dApps.

3. Tracking changes: Real-time monitoring can track changes to smart contracts and dApps, allowing for a detailed audit trail of activity. This can help identify the source of a potential attack and provide valuable information for incident response and forensics.

4. Automated response: Real-time monitoring can also trigger automated responses when suspicious activity is detected. For example, an automated response might include disabling the contract or blocking certain types of transactions until the security team can investigate further.

In summary, real-time monitoring can help prevent the deployment of malicious contracts by providing early detection and alerting of suspicious activity, identifying vulnerabilities, tracking changes, and triggering automated responses. By combining real-time monitoring with other security best practices, such as secure coding and access controls, it is possible to create a comprehensive security strategy to help protect against a wide range of attacks in web3.

Backdoor

Tag: Persistence

Category: Higher Privilige Attacks

What is a “backdoor”?

One example of a persistence attack in Web3 where a hacker gains complete control over a network, or dApp is a "backdoor" attack. This attack involves inserting a hidden access point, or "backdoor," into a network or application that allows the attacker to bypass normal authentication and gain complete control over the system.

In a blockchain context, a backdoor could be inserted into the smart contract code, allowing the attacker to execute arbitrary code on the blockchain and manipulate its state. For example, the attacker could create new transactions, transfer funds, or change the ownership of assets without the knowledge or consent of the legitimate users of the blockchain.

Backdoors are secret entry points to a system or software that allow unauthorized access. In the context of Web3, backdoors can be used to gain persistent access to a smart contract, wallet, or other decentralized application, enabling attackers to steal funds or data.

Example

Not so common in the crypto world, especially DeFi; we may have a huge example in CeFi. Rumors had it that the FTX collapsed, not due to a private key, but due to a backdoor established inside FTX system by SBF himself.
Source: https://www.businessinsider.com/sam-bankman-fried-secret-backdoor-worth-65-billion-court-hears-2023-1

Mitigation

Web3 technology is still evolving, and while it offers many advantages over traditional web technologies, it also presents new security challenges. Backdoors are one such challenge that remains a challenge, and they can allow attackers to gain unauthorized access to web3 systems and exploit vulnerabilities for their own purposes. Here are some ways to mitigate backdoors in web3. It is also important to ensure all developers in a project are to be trusted. They can implement malicious code intentionally, without authority.

Backdoors can actually be introduced unintentionally during the development process. By following secure development practices, such as using secure coding techniques, conducting regular code reviews, and performing thorough testing, you can reduce the risk of introducing backdoors into your web3 applications. Follow best practices for smart contract development: Smart contracts are an integral part of many web3 applications and are through malicious smart contracts and/or loopholes that the backdoor access would be. It is, in this case, highly important to monitor and check the contracts for any backdoor access to unauthorized wallets.

Credential Access

What is "Credential Access"?

Credential Access refers to the methods used by attackers to obtain sensitive data, such as usernames, passwords, and private keys, which can be used to impersonate legitimate users and gain unauthorized access to their accounts or steal their funds. In the Web2 world, this usually involves passwords and identities, while in the Web3 world, Private Keys and social media or communication credentials are more commonly targeted. Discord serves as an example of a platform where these credentials are targeted.

Credential Access attacks can take various forms, including phishing, social engineering, and brute-force attacks. Phishing is a common tactic involving bogus emails or messages to deceive users into disclosing their login credentials. Social engineering involves manipulating users into divulging sensitive information through psychological manipulation or deception. Brute-force attacks utilize automated tools that try different combinations of characters to guess passwords.

After obtaining a target's credentials, attackers can use them to gain entry into accounts or obtain funds. Additionally, these credentials may conceal their identity while interacting with other entities within the Web3 ecosystem. This may involve the use of fraudulent identities or the impersonation of genuine users to remain undetected.

Credential Access attacks pose a serious threat to the security of Web3 systems and can result in financial loss, data breaches, and damage to the reputation of individuals and organizations. Therefore, Web3 users must take steps to protect their credentials and implement strong security practices.

To protect against Credential Access attacks, Web3 users should use strong, unique passwords for each account and enable two-factor authentication whenever possible. They should also exercise caution when clicking on links or downloading files from untrusted sources and avoid sharing sensitive information with unknown individuals or entities. Additionally, Web3 developers should follow secure coding practices and conduct regular security audits to ensure their systems are not vulnerable to Credential Access attacks.

In summary, Credential Access poses a significant threat to the security of Web3 systems. Malicious actors can use various techniques to access sensitive data, which can then be used to steal funds or impersonate legitimate users. To protect against Credential Access attacks, Web3 users and developers must implement strong security practices and remain vigilant against potential threats.

In Web3, the credentials needed for Credential Access are usually private keys and mnemonic phrases. These private keys and mnemonic phrases are used to access and manage cryptocurrency wallets, sign transactions, and verify ownership of digital assets on the blockchain. Private keys are a string of characters that give users access to their funds, and they should always be kept secret and secure.

Credentials for Web3 services, such as decentralized exchanges (DEXs) and other decentralized applications (dApps), may also be targeted. These credentials could include login credentials for dApps, API keys, and other forms of authentication used to interact with Web3 services. Protecting all Web3 credentials is essential, as they can be used to gain access to sensitive information and valuable assets.

Furthermore, social media and communication credentials, like Discord and Telegram, can also be targeted in Web3. These platforms often communicate with teams and communities in the crypto space. Attackers can use stolen credentials to impersonate team members or community leaders and gain access to sensitive information or assets. Therefore, it is crucial to protect all types of credentials in Web3 and use secure authentication practices to minimize the risk of credential access attacks.

Credential Access

Identity Spoofing

Tag: Credential Access

Category: User Target

What is Identity spoofing?

Identity spoofing is a cyber-attack where someone creates a false identity or impersonates a legitimate entity to gain access to sensitive information or accounts. It is important to implement identity verification processes and conduct regular audits to prevent such fraudulent activity. Attackers may use tactics like phishing emails, social engineering, or tools to create fake online identities. By being vigilant and taking appropriate precautions, such attacks can be mitigated.

Example

In Web3, identity spoofing can pose a significant threat as it may result in the loss of cryptocurrencies or other digital assets. For instance, an attacker could fabricate a false identity on a social media platform or messaging app and exploit it to establish trust with the victim. Once trust is established, the attacker can request sensitive information, such as private keys or login credentials, which can be used to gain unauthorized access to the victim's digital assets.

Mitigation

To avoid identity theft, it's essential to use robust authentication methods and educate users on how to recognize and report suspicious activity. This may involve implementing multi-factor authentication, designating trusted contacts, and exercising caution when divulging personal information online. Regular security audits are also necessary to pinpoint and resolve system vulnerabilities.

Exchange Account Theft

Tag: Credential Access **Category: Higher Privilige Attacks**

What us "Exchange Account Theft"?

Web3 exchanges often require users to create accounts and provide sensitive information, such as personal identification and bank account details. Malicious actors can use phishing or social engineering to steal exchange account credentials, which can be used to steal funds or make fraudulent trades. Exchange account theft is also related to the last phase of an attack: money laundering. In this phase, hackers use fake identities or exchange accounts to withdraw funds into a bank account or elsewhere.

Therefore, it is important to be cautious when providing personal information on Web3 exchanges and to take steps to protect your account. One way to do this is to avoid clicking on suspicious links or providing personal information to unsolicited sources. Additionally, you can enable two-factor authentication for added security.

To prevent Web3 exchange account theft, it is important to be cautious when providing personal information, avoid suspicious links, and enable two-factor authentication. Malicious actors can use phishing, social engineering, or malware to steal login credentials, which can be used for fraudulent trades or stealing funds.

Example

An instance of exchange account theft in 2019 was the Binance hack, in which hackers utilized various methods, such as phishing attacks and malware, to steal 7,000 BTC valued at approximately $40 million.

Source: https://www.bloomberg.com/news/articles/2019-05-08/crypto-exchange-giant-binance-reports-a-hack-of-7-000-bitcoin

Mitigation

To reduce the risk of exchange account theft, Web3 exchange platforms should implement robust security measures such as two-factor authentication, IP address whitelisting, and regular security audits. Exchange users should also take steps to safeguard their accounts, such as using strong passwords and avoiding sharing their login credentials with others. Additionally, users should be cautious of suspicious emails or messages and refrain from clicking on links or downloading files from unknown sources. Furthermore, Web3 exchanges need to comply with regulatory requirements, including Anti-Money Laundering (AML) and Know Your Customer (KYC) regulations. These regulations require that exchanges authenticate their users' identities and monitor their transactions to identify any suspicious activities that may indicate money laundering or other illicit activities.
All in all, exchange account theft poses a grave threat to the security of Web3 exchanges and can lead to significant financial losses for users. It is crucial for exchange platforms and users alike to proactively take measures to prevent such attacks from happening.

Social Media Credential Theft

Tag: Credential Access

Category: Higher Privilige Attacks

What is Social Media Credential Theft?

Social media platforms are often used by Web3 projects to engage communities and promote their projects. However, malicious actors may use phishing or social engineering techniques to steal social media credentials, including usernames and passwords. These stolen credentials can then be used to impersonate legitimate accounts, spread misinformation, conduct scams, or carry out other malicious activities.

Social media credential theft refers to stealing social media account login credentials through malicious means such as phishing, social engineering, or malware.

Social media credential theft is common in Web3 projects and NFT discords. Hackers may target community members of a particular Web3 project or NFT discord to steal their social media credentials and gain access to their accounts. Once the hacker gains access to these accounts, they can then impersonate legitimate users to spread misinformation or conduct scams related to the Web3 project or NFT discord.

Example

One instance of credential theft involves the Discord credential theft of an NFT project. In this scenario, hackers used a phishing scam to deceive users into divulging their login credentials for the NFT project's Discord server. Using these credentials, the hackers were able to infiltrate the Discord server and impersonate authorized users to disseminate misleading information and perpetrate scams related to the NFT project.

Mitigation

To reduce social media credential theft risk, Web3 project teams and NFT Discord moderators must educate their community members about the dangers of phishing and social engineering scams. Encouraging users to use strong passwords, enable two-factor authentication, and avoid clicking suspicious links or downloading unknown files is also essential. Furthermore, project teams and moderators should monitor their social media accounts and NFT Discords for suspicious activity and take swift action to address potential security breaches.

Overall, social media credential theft significantly threatens the security of Web3 projects and NFT Discords. All stakeholders must remain vigilant and take proactive measures to prevent such attacks.

Private Key Theft

Tag: Credential Access

Category: Acquire Private Key

What is “private key” theft?

This section does overlap with a lot of other sections.) Private keys are the main credentials used to access and manage Web3 assets, which include cryptocurrencies, NFTs, and smart contracts. Malicious actors may attempt to steal users' private keys through phishing, social engineering, or malware.

Example

The Ronin Network hack of 2022 is an example of a Guardian Takeover attack.

Axie Infinity, a popular blockchain gaming application, was developed on the Ronin Network. Regrettably, Ronin experienced one of its worst hacks in March 2022, when a malicious actor rapidly obtained 173,600 ether ($ETH) and 25.5 million USDC, which were later exchanged for $625 million. The hacker acquired the necessary private keys and consequently stole all the funds from the Ronin Bridge in just two transactions, making it one of the most significant DeFi breaches.

The Ronin Bridge had nine "validators" operating it, with a five out of nine thresholds. Sky Mavis, the company behind Axie Infinity, oversaw four validators, so the private keys needed to be distributed more. Additionally, Axie delegated their validator's signature to Sky Mavis in November 2021. While this delegation was meant to be temporary due to the heavy traffic Axie was experiencing, it was never revoked. Sky Mavis ended up with five validator signatures, enough to approve any message. Through a social-engineering attack, the attacker gained control of the keys. They could call withdrawERC from the bridge without a backing transaction on the other side once they had the keys.

Mitigation

Private key theft is a critical security issue in the blockchain and cryptocurrency world. It can result in the loss of funds and compromise the security of a blockchain network. Below are some practical ways to prevent private key theft while securing your crypto assets:

- Use a hardware wallet: A physical device stores your private keys offline, making it more challenging for hackers to access them. It is one of the most secure ways to store your private keys.
- Use a software wallet with two-factor authentication: If you use a software wallet to store your private keys, enable two-factor authentication (2FA) to add an extra layer of security. This will require a code generated by an app or text message in addition to your password to access your wallet.

- Use a strong password: Create a strong, unique password for your wallet and change it regularly. Avoid using easily guessable passwords, such as common words or phrases, birthdates, or pet names.
- Keep your private keys offline: Consider printing and storing them in a secure physical location, such as a safe or safety deposit box. This ensures that your private keys are not stored on a computer or device that can be hacked.
- Avoid phishing scams: Be wary of phishing scams that trick you into giving away your private keys. Only enter your private keys on trusted and secure websites.
- Regularly update your software: Keep your wallet software up-to-date with the latest security patches and updates to address any vulnerabilities.
- Use a multi-signature scheme: dApps can implement multi-signature schemes requiring multiple private keys to authorize a transaction. This adds an extra layer of security and reduces the risk of private key theft.

The best way to prevent private key theft is to stay vigilant and take proactive steps to secure your private keys. You can significantly reduce the risk of private key theft by using a combination of hardware and software wallets, two-factor authentication, strong passwords, offline storage, and regular updates.

Privilege Escalation?

What is a "Privilege Escalation"?

Privilege Escalation comprises of techniques malicious parties use to gain higher-level permissions on a protocol or network. Malicious parties can often enter and explore a network with unprivileged access but require elevated permissions to follow through on their objectives. Common approaches are to take advantage of system weaknesses and misconfiguration in logic and smart contract vulnerabilities.

Examples of elevated access include:

- Private Key access, full or semi SYSTEM/root/Admin level
- Multi-sig Private Key access, whole SYSTEM/root/Admin level
- a user account or wallet with admin-like access
- user accounts/wallets with access to specific systems or perform a specific function.

These techniques often overlap with Persistence.

Privilige Escalation

Governance exploit (DAO takeover)

Tag: Privilege Escalation

Category: Logic

What is a "Governance Exploit"?

In the case of DAOs, a governance exploit can occur when a hacker gains control of a governance contract or a malicious proposal is voted into effect. This can allow the hacker to gain administrative control over the DAO, allowing them to manipulate or steal assets held by the organization or implement overrides.

A DAO, or Decentralized Autonomous Organization, is a type of organization that operates through smart contracts on a blockchain. A DAO can be taken over when an attacker gains control of a sufficient number of voting rights/tokens or other governance exploit methods to influence the decision-making process of the organization. This essentially depends on the rules embedded into the governance structure itself.

Example

In a DAO takeover attack, the attacker seeks to gain control of the organization's governance process by either stealing tokens, performing flash loans, or gaining access to the private keys of a significant number of members. Once the attacker gains control, they can propose and vote on malicious proposals that could grant them additional privileges or access to the organization's assets. Flash loans, in this case, are the most popular method of gaining a sufficient amount of tokens to override a proposal.

A real-world example is Beanstalk DAO, which was exploited in 2022. Beanstalk is a DeFi network with its stablecoin $BEAN at the center of it. In April 2022, a malicious governance attack using a flash loan resulted in the theft of $182 million. In this case, PeckShield was the first to discover that the attacker used Beanstalk's majority rules governance system to steal the $182 million.

The attacker seized majority control of the protocol's governance with a flash loan of $1 billion from Aave, Uniswap, and SushiSwap. They gained enough voting power (majority rules) by swapping the funds and depositing them in the Beanstalk protocol liquidity pools, making it possible to call the emergencyCommit function and trigger an emergency governance execution. The attack leveraged the lack of delay between voting and execution to pass a malicious proposal that transferred deposited funds to the attacker's address. With these steps, the attacker made $80 million in profits.

Mitigation

The purpose of a DAO is to create decentralized governance practices and rules, and even though it is a noble intent, it can be exploited by malicious parties. Essentially, a DAO wants to enable the majority to implement changes in the future direction of a given protocol. However, this presents numerous vulnerabilities because one person can exploit and cheat themselves to power, even though they follow the rules of the DAO itself.

To prevent a DAO takeover, it is essential to ensure that the logic of the governance structure is set up correctly and implement other robust measures. Since the whole idea is to keep the organization decentralized, mitigation strategies like limiting access to trusted individuals are not an option.

In these scenarios, each DAO is architected differently, which presents new vulnerabilities. Some have added mechanisms, limits, or rules that may present a point of manipulation. It is crucial to regularly review and audit the government contracts of the organization to identify and mitigate potential vulnerabilities. In case of an attack, it is essential to have a response plan in place to quickly mitigate the damage and restore control over the organization. Real-time monitoring of the DAO smart contracts is also essential for detecting malicious activities.

Blockchain Node Hijacking

Tag: Privilege Escalation

Category: Higher Privilige Attacks

What is Blockchain Node Hijacking?

Blockchain nodes are critical components of the web3 infrastructure. In this attack, an attacker takes over a blockchain node to gain control of the network. Once control is gained, the attacker can manipulate transactions and potentially steal funds.

Blockchain Node Hijacking is a type of Privilege Escalation attack that aims to compromise and gain control of the blockchain network by hijacking a node responsible for validating transactions or mining blocks. In this attack, the attacker attempts to take over the blockchain node, which can lead to complete control over certain parts of the blockchain network.

Example

Blockchain nodes can potentially be compromised if the device or server on which they are stored is vulnerable to attacks. However, running a blockchain node on dedicated hardware can reduce the risk of compromise as it isolates the node from other processes running on the same device. Additionally, blockchain nodes are designed to be resistant to attacks and can detect and reject any invalid or fraudulent transactions. However, if an attacker gains control of a majority of the nodes in a blockchain network, they could potentially manipulate the transactions and undermine the security and integrity of the blockchain network.

Mitigation

To prevent Blockchain Node Hijacking, it is essential to implement strong security measures that prevent unauthorized access to the blockchain node. Some ways to prevent this attack include:

1. Limiting access to the blockchain node to authorized personnel only.
2. Implementing strong authentication mechanisms such as two-factor authentication (2FA) or multi-factor authentication (MFA).
3. Encrypting all data transmissions between nodes and network peers.
4. Regularly updating and patching the blockchain node's software to prevent known vulnerabilities from being exploited.
5. Monitoring the network for suspicious activities and implementing security controls to detect and prevent malicious activities.
6. Having sufficient decentralization and Node parameters in place.

By implementing these security measures, organizations can reduce the risk of Blockchain Node Hijacking and ensure the security of their blockchain network.

Guardian takeover

Tag: Privilege Escalation **Category: Higher Privilige Attacks**

What is a “Guardian takeover”?

A guardian takeover attack is a type of attack in which a hacker gains control of the guardian account for a decentralized application (dApp). This enables them to manipulate the smart contract that governs the dApp's operations and have complete control over the dApp. Such an attack can lead to theft of funds, modification of the contract's rules, or a complete shutdown of the dApp. In a dApp, a guardian is a designated party responsible for managing the smart contract that governs the dApp's operations. A guardian takeover attack occurs when a hacker gains control of the guardian account, allowing them to manipulate the smart contract and have complete control over the dApp. Once they have control, they can potentially steal funds, modify the contract's rules in their favor, or shut down the dApp entirely.

Example

The Ronin Network hack of 2022 serves as an example of a Guardian Takeover attack. The Axie Infinity blockchain gaming application gained a lot of popularity and was developed on the Ronin Network. Unfortunately, Ronin suffered one of its worst hacks in March 2022, when a malicious actor was able to quickly obtain 173,600 ether ($ETH) and 25.5 million USDC, which were later exchanged for $625 million. The hacker managed to get hold of the necessary private keys and consequently stole all the funds from the Ronin Bridge in just two transactions, making it one of the most significant DeFi breaches to date.

The Ronin Bridge was operated by nine "validators," with a five out of nine threshold. Sky Mavis (the company behind Axie Infinity) oversaw four validators, so the private keys weren't distributed enough. Furthermore, Axie delegated their validator's signature to Sky Mavis in November 2021. Although this delegation was supposed to be temporary because Axie was experiencing heavy traffic, it was never revoked. Sky Mavis ended up with five validator signatures, enough to approve any message. Through a social-engineering attack, the attacker obtained control of the keys. They could call withdrawERC from the bridge without a backing transaction on the other side once they had the keys.

Mitigation

Decentralization is a crucial feature of all dApps. This makes it more challenging for a single entity to gain control of the network. But the access points to get control of the dApp can still be exploited or compromised. To prevent guardian takeover attacks, strong security measures are required, including:

Proper access controls: Implementing proper access controls for guardians and validators can help prevent unauthorized access to sensitive areas of the dApp, reducing the risk of attacks.

Regular security audits: Conducting regular security audits can help identify vulnerabilities in the dApp's code and infrastructure, allowing for them to be addressed before they can be exploited.

Multi-signature authorization: Implementing multi-signature authorization can help prevent guardian takeover attacks by requiring multiple parties to authorize certain actions, such as fund transfers or changes to the smart contract.

Emergency protocols: Implementing emergency protocols can help prevent or mitigate the impact of attacks by allowing for quick action in the event of an attack.

Real-time monitoring: Implementing proactive security measures such as real-time monitroing is essential to be alerted int eh case of a potential attack.

In summary, preventing guardian takeover attacks requires strong security measures, including decentralization, robust consensus mechanisms, proper access controls, regular security audits, multi-signature authorization, and emergency protocols. By taking a comprehensive approach to security, it is possible to reduce the risk of attacks and protect the integrity of the dApp.

Smart Contract Ownership Override

Tag: Privilege Escalation

Category: Higher Privilige Attacks

What is the "smart contract ownership override"?

In this attack, an individual exploits a vulnerability in a smart contract to gain ownership. Once they have ownership, they can alter the contract to their preference, including providing greater access and control.

Smart contract override is a privilege escalation attack targeting smart contracts on a blockchain network. It is initiated when an attacker exploits a smart contract or network vulnerability that allows them to gain unauthorized access and control over the contract's operations. The attacker can then modify the contract's code, move funds, and execute malicious functions without the contract owner's awareness or permission. In the Web3 framework, smart contract override is classified as a privilege escalation attack. This is because the attacker gains elevated privileges over the smart contracts, enabling them to perform actions they would not typically have access to.

Here is an example: Let's consider a smart contract that sets the price of a commodity such as real estate.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.9;
contract RealEstatePrice {
 uint256 public apartmentprice;

 constructor(uint256 _price) {
   apartmentprice = _price; // default
 }

 function updateApartmentPrice(uint256 _price) external {
   apartmentprice = _price;
 }
}
```

"The contract defined above is a simple real estate price. The constructor sets the default price for the apartment. The updateApartmentPrice() function updates the apartment price with the new one. The contract appears innocent; however, if you observe closely, the function updateApartmentPrice() is an external function and can be called by anyone (attacker) apart from the deployer or the owner to update the apartment pricing. This is a simple and classic example of an ownership attack where an attacker can call a function to update the value and easily exploit it."

Mitigation

To prevent smart contract override attacks, there are several best practices that developers can follow:

Use secure coding practices: Developers should follow secure codes when creating smart contracts, such as input validation, error handling, and parameter checks.

Add a custom modifier that checks if you are the contract owner and only allows you to update the price in the function.

Use secure contracts. Some contracts are well-tested, proven, efficient, and widely adopted; we can reuse the owner smart contract, preventing us from rewriting the modifier like above.

Conduct thorough testing: Developers should conduct thorough testing of smart contracts to identify and address any potential vulnerabilities.

Implement access controls: Smart contracts should be designed with proper access controls in place to limit the actions that users can perform.

Use multi-signature wallets: Multi-signature wallets can be used to ensure that any changes to the smart contract require approval from multiple parties.

Monitor smart contracts: Regularly monitoring smart contracts can help identify any unauthorized access or modifications to the contract's code.

Source: https://blog.finxter.com/smart-contract-security-series-part-1-ownership-exploit/

Lateral Movement

What is “lateral movement”?

Malicious actors can access and control remote systems connected to a network using tactics known as lateral movement. In this scenario, the attacker is "moving" through the network, exploring and discovering their target to gain access to it later. This tactic is often necessary for malicious actors to carry out their primary aim, and they frequently switch between multiple systems and accounts to achieve their goal. To perform the lateral movement, adversaries may install their own remote access tools or use valid credentials with stealthier native networks and operating system capabilities.

In the context of Web3 systems, lateral movement is a relevant category of attack because many blockchain networks and decentralized applications are interconnected. This interconnectedness means that an attacker who gains access to one system may be able to pivot to other connected systems and potentially gain access to valuable assets.

For example, an attacker who gains access to a vulnerable smart contract on one blockchain network may be able to exploit that access to compromise a user's wallet on another connected blockchain network. This could result in cryptocurrency theft or other valuable assets, such as NFTs. Additionally, an attacker who gains access to a node on one blockchain network may leverage that access to compromise other nodes or smart contracts on the same or connected networks.

Lateral Movement

Multi-Chain Attacks

Tag: Lateral Movement **Category: Cross Chain**

What are "Multichain attacks"?

A "multichain attack" occurs when an attacker gains access to one blockchain network or dApp and then uses that access to pivot to other connected blockchain networks or dApps. This allows the attacker to move laterally through the environment and access valuable assets.

Within a lateral movement, multichain attacks refer to an attack tactic where an adversary gains access to one blockchain network or dApp, then moves laterally across multiple connected blockchain networks or dApps to reach their ultimate target. This tactic allows attackers to broaden their attack surface and access valuable assets across multiple blockchain networks.

Example

An example of a multichain attack within the lateral movement could involve an attacker gaining access to and exploiting a bug in a decentralized exchange (DEX) on one blockchain network and then using that access/bug exploit to pivot to other connected chains.

Mitigation

To mitigate multichain attacks within the lateral movement, it is important to implement strong access controls and monitoring tools to detect and prevent unauthorized access and movement within blockchain networks. This includes using multi-factor authentication, implementing network segmentation to restrict lateral movement, and conducting regular security audits and vulnerability assessments to identify and address potential weaknesses. Additionally, organizations should consider implementing blockchain-specific security solutions, such as smart contract audits and token whitelisting, to reduce the risk of multichain attacks within the lateral movement.

Bridge exploits

Tag: Lateral Movement **Category: Cross Chain**

Understanding "Bridge Exploits"

A "multichain attack" occurs when an attacker gains access to one blockchain network or dApp and then uses that access to pivot to other connected blockchain networks or dApps. This allows the attacker to move laterally through the environment and access valuable assets.

Within a lateral movement, multichain attacks refer to an attack tactic where an adversary gains access to one blockchain network or dApp, then moves laterally across multiple connected blockchain networks or dApps to reach their ultimate target. This tactic allows attackers to broaden their attack surface and access valuable assets across multiple blockchain networks.

Example

In 2021, the Poly Network, a decentralized finance platform, was hacked through a vulnerability in its smart contract. The attackers were able to steal over $600 million worth of cryptocurrencies across three different blockchains: Ethereum, Binance Smart Chain, and Polygon. The attackers used the stolen funds to create new smart contracts on each of the three blockchains to move the stolen assets around, making it more difficult to track and recover the funds.

Mitigation

A bridge is a tool that allows for communication between two different blockchain networks. Bridge hacks occur when attackers gain entry to one network and use it to access other connected networks through the bridge. This allows attackers to move laterally through the environment and access valuable assets on different blockchain networks.

Compromised nodes

Tag: Lateral Movement

Category: Higher Privilige Attacks

What are "compromised nodes"?

Compromised nodes are nodes within a blockchain network that an attacker has gained control of, often through a vulnerability or misconfiguration. Once an attacker has control of a node, they can use it to pivot to other nodes or systems within the same network, giving them access to valuable assets.

This access can allow attackers to move laterally through the environment and manipulate the network.

Example

For instance, an attacker who gains control of a node in a DeFi protocol can access the protocol's smart contracts and execute transactions on the protocol's behalf. This enables them to move laterally within the protocol's network and gain access to valuable assets such as user funds, governance tokens, or private data.

Mitigation

To mitigate the risk of compromised nodes, it is crucial to have a strong security posture in place. This includes regularly updating software and patches, using strong passwords, and limiting access to sensitive systems. Additionally, monitoring network traffic and system logs can help detect any suspicious activity and enable prompt response. Implementing a defense-in-depth approach, which involves layering multiple security mechanisms to prevent and detect attacks, is also recommended. This can include firewalls, intrusion detection systems, security information, and Rreal-time analysis tools.

Extrafiltration

What is "Exfiltration"?

Exfiltration is a term used to describe the methods that attackers use to steal and conceal data from a network. Once the attackers have gathered the data, they often take steps to package it, which can include encryption and compression, to hide or erase it.

Defence Evasion, a relevant concept throughout the lifecycle of a hack, is essentially the same thing as exfiltration in Web3.

In Web2, exfiltration usually involves data, while in Web3, it often involves assets.

Typically, methods for extracting data from a target network involve sending it across the command and control channel or another channel. Sometimes there are size restrictions on the transmission. In the case of Web3, this may involve stealing cryptocurrency or other digital assets from a compromised wallet or exchange. The attacker may use various techniques, such as encrypting the stolen data, disguising it as harmless traffic, or using covert channels to avoid detection while exfiltrating the data.

Extrafiltration

Multi-Chain Attacks

Tag: Exfiltration

Category: Money Laundering

What are "Atomic Swaps"?

Atomic swaps are a type of decentralized technology that enables the exchange of one cryptocurrency for another without the need for a centralized exchange. While this technology can be exploited to obscure the flow of funds and parties involved in a transaction during the money laundering phase of an attack, it can also be used for legitimate purposes.

In a money laundering attack, a hacker might use atomic swaps to convert stolen cryptocurrency into a more privacy-focused cryptocurrency like Monero or Zcash. Doing so makes it more difficult for investigators to trace the stolen funds back to the original source.

Atomic swaps utilize smart contracts to create a trustless exchange between two parties. For example, a hacker could set up a smart contract to exchange their stolen Bitcoin for an equivalent amount of Monero without needing a centralized exchange or intermediary.

Example

In a money laundering attack, a hacker might exploit atomic swaps to convert stolen cryptocurrency into a more privacy-focused cryptocurrency like Monero or Zcash, making it difficult for investigators to trace the stolen funds back to the original source.

Mitigation

To mitigate the risk of atomic swaps being used for money laundering, cryptocurrency exchanges and financial institutions can implement robust anti-money laundering (AML) and know-your-customer (KYC) policies. They can also use blockchain analytics tools to monitor transactions and detect suspicious activity. Additionally, regulators can impose stricter regulations on cryptocurrency exchanges and financial institutions to prevent using atomic swaps for illicit purposes.

Privacy solutions like Monero

Tag: Exfiltration

Category: Money Laundering

What is Monero?

One of the features that some blockchains offer is privacy through encryption and cryptography. Monero is a popular example of this.

Monero is a privacy-focused solution that uses advanced cryptography techniques to obscure transaction details, making it difficult to trace the source and destination of funds. It uses techniques like ring signatures, stealth addresses, and confidential transactions to make transactions untraceable and un-linkable, offering enhanced privacy and anonymity to its users. This makes it an attractive option for those prioritizing privacy in their transactions. In fact, stolen assets are often converted to Monero and sent to other wallets anonymously due to their privacy features.

Another use case for Monero is in the context of exfiltration. Hackers may use Monero to receive payment for stolen data, as it allows them to conceal their identity and makes it difficult for law enforcement to track the funds.

Example

Let's say a Web3 application on the Ethereum blockchain has been hacked, and funds have been stolen. To cover their tracks, a hacker might convert the assets to Monero and send them to anonymous wallet addresses. This is what's known as the "money laundering" phase of an attack.

Mitigation

To mitigate the risk of exfiltration, organizations can implement strong access controls, encryption, and monitoring systems. However, privacy solutions like Monero have not been compromised and are available on exchanges, making it challenging to prevent exfiltration except on a centralized exchange actively.

Impact

What is “Impact”?

Impact refers to methods malicious parties use to disrupt availability or compromise integrity by manipulating infrastructure operational processes. These techniques can include destroying or tampering with data. In some cases, processes may appear normal but could have been altered to benefit the adversaries’ goals. Malicious parties may use these methods to pursue their goals or to provide cover for a confidentiality breach.

The Impact category refers to the effects of an attack or vulnerability on an organization's systems, data, or operations. These categories include Integrity, Availability, Confidentiality, and Attribution.

In the context of Web3, Integrity is an important aspect of Impact. Smart contracts are self-executing code that runs on blockchain networks and is a central feature of many Web3 applications. A vulnerability in a smart contract could allow an attacker to modify or manipulate the state of the contract, potentially leading to financial loss or other negative consequences. Ensuring the integrity of Web3 applications and the smart contracts that underpin them is critical to maintaining trust in decentralized systems.

Availability is another aspect of Impact that is relevant to Web3. Decentralized applications and networks rely on a large number of nodes to maintain their operations. An attack that disrupts the availability of these nodes could render a Web3 application or network unusable. Ensuring the availability of Web3 systems is critical to maintaining their usefulness and ensuring their adoption.

Finally, the Attribution aspect of Impact is relevant to Web3 systems because of their decentralized and pseudonymous nature. Web3 applications and networks are designed to operate without central authorities or intermediaries, and transactions are often conducted pseudonymously. This can make it difficult to attribute the source of an attack or vulnerability. Ensuring that Web3 systems provide adequate mechanisms for verifying the identity of users and transactions is critical to maintaining security and accountability in decentralized systems.

Impact

Network shutdown

Tag: Impact

Category: Money Laundering

What is a "Network shutdown"?

Network shutdown is a type of cyber attack that can significantly affect the availability and integrity of Web3 systems. These attacks typically disrupt communication channels between nodes in a decentralized network, rendering the network unavailable or partially unavailable to legitimate users. Network shutdown attacks can take different forms, such as DDoS attacks, targeted attacks on specific nodes, or attacks on network infrastructure.

Example

In a decentralized cryptocurrency network like Bitcoin, a network shutdown attack could involve overwhelming the network with a high volume of malicious transactions or targeting key nodes in the network. This could lead to a slowdown or complete halt in the processing of legitimate transactions, resulting in financial losses for users and potentially harming the network's reputation. It's important to note that network shutdown is sometimes the reaction of protocol developers to halt an attack. Even if the attack is successful or not, such events highlight the issue of the integrity and availability of the blockchain.

Mitigation

To reduce the impact of network shutdown attacks on Web3 systems, organizations and developers can implement various measures. One approach is to deploy multiple nodes in different geographical locations, which can increase the resilience and redundancy of the network. Additionally, developers can design their applications to use alternative communication channels, such as off-chain channels or alternative consensus mechanisms, to reduce the impact of network shutdown attacks. Network monitoring and detection tools can also help organizations identify and respond to network shutdown attacks promptly. Finally, organizations can implement incident response plans and conduct regular security assessments to ensure the ongoing security and resilience of their Web3 systems.

Data Destruction

Tag: Impact

Category: Money Laundering

What is "data destruction"?

Data destruction refers to attackers' techniques to destroy, alter, or corrupt critical data stored on a system or network. In the context of Web3, this can include attacks on blockchain data, smart contract code, and other sensitive information used to facilitate transactions and user interactions. By destroying data, attackers can cause significant financial losses, disrupt business operations, and compromise the integrity and trust of the Web3 ecosystem.

This attack subcategory involves techniques that destroy or corrupt critical data stored on a Web3 network or application. Examples include wiping out transaction logs, altering or deleting smart contract code, or corrupting blockchain data.

Example

For example, an attacker may exploit a vulnerability in a smart contract to corrupt the code or alter the state of the blockchain, resulting in the loss or theft of funds. Another example is the use of ransomware to encrypt or delete critical data, demanding payment in exchange for the decryption key or restoration of the data. These types of attacks can have severe consequences, as they can result in the permanent loss of data, loss of customer trust, and legal or regulatory repercussions.

Mitigation

To mitigate the impact of data destruction attacks, developers and users of Web3 systems should implement strong security measures, such as using encryption to protect data at rest and in transit, implementing access controls and permissions to restrict unauthorized access, and regularly backing up critical data. It is important to note that most smart contracts are immutable and cannot be changed. Implementing disaster recovery plans and incident response procedures can also help to minimize the impact of data loss or corruption. Additionally, conducting regular security assessments and penetration testing can help to identify and address vulnerabilities before attackers exploit them.

Disrupt System Operation

Tag: Impact **Category: Money Laundering**

What is “Disrupt System Operation?

Disrupt System Operation refers to a set of techniques used by attackers to interfere with the normal functioning of a system or network. In the context of Web3, this can involve attacks on the blockchain, smart contracts, and decentralized applications that enable transactions and interactions between users. Disrupting system operations can cause service outages, disrupt business operations, or result in unauthorized access to sensitive information or assets. This category of attacks encompasses techniques that aim to disrupt the normal operation of a Web3 system or network.

Example

Examples of these techniques include launching DDoS attacks, manipulating smart contracts to cause unexpected behavior, or exploiting vulnerabilities to crash nodes or clients.

For instance, an attacker may flood a Web3 network with many requests to render it unable to process legitimate transactions. Another example is the exploitation of vulnerabilities in smart contracts, which can result in unexpected behavior or unauthorized access to funds. These types of attacks can lead to significant financial losses, damage the reputation of a business, and have legal or regulatory repercussions.

Mitigation

Developers and users of Web3 systems should implement best practices for security to mitigate the impact of Disrupt System Operation attacks. For example, regularly updating software, using multi-factor authentication, and conducting vulnerability assessments and penetration testing can help to ensure security. Additionally, implementing redundancy and backup measures, such as distributed data storage and failover mechanisms, can minimize the impact of system disruptions. Monitoring network traffic and system logs is also essential for detecting and responding to anomalous behavior and potential attacks promptly.

Front-Running

Tag: Impact

Category: User Target

What is Front-Running?

Transaction front-running is an attack where an attacker uses their knowledge of a pending transaction to execute a transaction before the original transaction completes, taking advantage of the price difference. This can occur in decentralized applications (dApps) built on a blockchain network like Ethereum.

It's a technique attackers use to identify and exploit vulnerabilities in the blockchain, such as transaction ordering, to gain a financial advantage over other network users. This involves placing a transaction in a block before another user's transaction to gain an unfair advantage.

Transactions aren't immediately added to the blockchain ledger. First, they're added to the mempool before being collected into blocks. Front-running attacks take advantage of adding transactions to blocks based on transaction fees. An attacker can ensure that their transaction is processed before any other transaction by including a higher fee. This is called a front-running attack.

Front running in Web3 and blockchain networks can be placed under "Impact" because it represents an action that directly affects the integrity and fairness of the system. By exploiting transaction ordering, front runners manipulate data to gain an unfair advantage, ultimately impacting the decision-making and operations of other participants in the network. This practice undermines the trust and transparency that are central to decentralized technologies, leading to potential financial losses for honest users and eroding confidence in the ecosystem.

Example

In decentralized exchanges, front-running can allow an attacker to buy many tokens before processing another user's transaction. This can drive up the token's price, enabling the attacker to sell it at a profit.

Mitigation

Here are some ways to prevent transaction front-running in the context of blockchain and web3:

- Increase Gas Fees: One way to prevent transaction front-running is to increase the gas fees, which are the transaction fees paid in Ethereum to miners to execute the transaction. If the gas fees are high, the cost of front-running a transaction will be too high for most attackers.

- Use Flashbots: Flashbots is a project that enables miners to coordinate and execute transactions off-chain using a private communication channel. This can help prevent front-running attacks, as the transactions are conducted off-chain and invisible to other miners.
- Use Private Transactions: Private transactions can be used to prevent front-running, as the transaction details are not visible to other participants. Private transactions can be achieved using various privacy protocols, such as zk-SNARKs, zk-STARKs, or Bulletproofs.
- Use Order Matching: Order matching can prevent front-running attacks in decentralized exchanges (DEXs). In an order-matching system, the buyer and seller's orders are matched by the exchange's smart contract rather than executed directly on the blockchain. This can help prevent front-running attacks, as the smart contract executes the order and is not visible to other participants.
- Use Time-Locks: Time-locks can delay the execution of a transaction, making it difficult for an attacker to front-run the transaction. A time-lock can be implemented using a smart contract, which only executes the transaction after a specified time has elapsed.