A replay attack is a type of exploit where the attacker intercepts and manipulates data transmission over a network

. In the realm of blockchain, replay attacks pose a significant threat to the security of transactions. This article explores the different types of replay attacks, explaining their mechanics and effective strategies to protect against them.

Prerequisites

- Understanding of how signatures and authentication work in Ethereum
- High-level understanding of ECDSA key generation, signing, and verification
- Knowledge of EIP-191 and EIP-712 Signature Standards

To follow this article, the following articles are recommended as prerequisites:

- Understanding Ethereum Signature Standards EIP-191 & EIP-712
- ECDSA Signatures

— Note that the code in this article is for demonstrative purposes and has not undergone a thorough security review, do not use it as production code.

What is a Replay Attack?

In the context of blockchain technology, attackers exploit this vulnerability by manipulating and reusing valid transactions for their gain. This can be achieved by copying the transaction to another chain or copying or modifying the signature, thus passing any validation checks.

Let's now go through the following types of blockchain replay attacks and how to protect against them:

1. Missing Application Nonce
2. Hard Fork
3. Cross-chain
4. Signature Malleability
5. Cryptographic nonce reuse

1. Missing Application Nonce

Signatures provide a means of cryptographic authentication in blockchain technology, serving as a unique "fingerprint", forming the backbone of blockchain transactions.

They are used to validate computation performed off-chain and authorize transactions on behalf of a signer. Signatures for previous or pending transactions can be replayed by attackers, if not handled correctly, meaning that attackers pass the validation checks and the malicious transactions are executed.

A nonce is a unique identifier or a "number used once".

Its absence can allow attackers to repeatedly exploit captured signatures for fraudulent transactions. To avoid confusion, we will refer to this specific nonce as an "application nonce

".

An application nonce is implemented into a smart contract to ensure that a signature can only be used once. It should be included as a field to the message

that is signed by the signer.

How missing application nonce replay attacks work:

1. The signer signs a message that does not include a nonce.
2. The signature is intercepted and copied by an attacker from the previous transaction.
3. The attacker is then able to re-use this signature to re-send the transaction.

The addition of a nonce would prevent this behavior as the contract would know that the signature is outdated.

Mitigating against Missing Nonce Replay

To implement this, include a mapping to keep track of each signer’s last used nonce:

This can then be used as part of the signed message. For a complete example of implementing nonces plus other replay prevention data, see the “Preventing Replay Attacks” section of this article.

Example of Nonce Replay

In the following *KYCRegistry* contract, signatures are used to grant KYC status to the users via the addKYCAddressViaSignature()

function. However, if KYC status is revoked for a user before the deadline, signature replay is not prevented. If the attacker calls addKYCAddressViaSignature()

again, using the original signature, KYC status will be forcefully granted to the attacker again.

Implementing a nonce mapping for each signer would remediate this vulnerability, as explained earlier. More information can be found here.

This article by Dacian provides a comprehensive walkthrough for mitigating missing nonce replay attacks.

2. Hard Fork Replay Attacks

A hard fork involves a significant and non-backward-compatible change to theprotocol of the blockchain, resulting in the creation of two separate ledgers: the original ledger and the newly forked ledger.

The potential for a replay attack arises when the two ledgers share the same transaction and address formats.

During a hard fork, the original and the new ledger may have similar or identical transaction and address formats, making them vulnerable to replay attacks. In a replay attack following a hard fork, a malicious actor can intercept and duplicate a legitimate transaction from one ledger onto the other. An attacker can, for example, mint tokens on the legacy chain and then copy the transaction onto the canonical chain.

Image: A Hard Fork Replay Attack

Mitigating Against Hard Fork Replay

- Strong replay protection

: involves attaching a unique marker or mechanism to the transactions on the new ledger, making them incompatible with the original ledger. This ensures that transactions in the updated system are distinct and cannot be validly replayed on the legacy ledger, thereby preventing the malicious duplication of transactions and associated security risks.

- Opt-in Replay Protection:

relevant in a hard fork that follows a cryptocurrency's ledger update rather than a split into two distinct ledgers. In the presence of opt-in replay protection, users are required to proactively make manual adjustments to their transactions to prevent the possibility of unintended replays.

Examples of Hard Fork Replay

Ethereum Classic Hard Fork:

Ethereum underwent a hard fork in 2016, focussing on improving efficiency and scalability. The old chain was renamed to Ethereum Classic (ETC) while the new chain retained the original name Ethereum (ETH). Since the chains had a similar specification, transactions were valid on both chains. People were able to take advantage of this replay vulnerability, for example by withdrawing ETH from exchanges and gaining ETC in the same amount.

Ethereum Merge

: In September 2022, Ethereum underwent a hard fork called The Merge.

“The Merge was the joining of the original execution layer of Ethereum

(the Mainnet that has existed since genesis) with its new proof-of-stake consensus layer, the Beacon Chain

. It eliminated the need for energy-intensive mining and instead enabled the network to be secured using staked ETH."

Since Ethereum PoW (the hard fork) had the same chain ID as Ethereum (post-merge Mainnet), replay attacks were possible. If Bob sends 100ETH on the PoW chain to Alice, Alice could replay this transaction on Ethereum and steal funds. This is an example of why it is important to not interact with a hard fork to prevent being vulnerable to replay attacks.

3. Cross-Chain Replay Attacks

It is also worth noting that replay attacks can extend beyond individual blockchain networks, as with hard-fork replay, to cross-chain scenarios. In cross-chain replay attacks, the threat arises when transactions from one blockchain network are replicated on a different blockchain network, often leveraging similar or interoperable protocols where the same or similar smart contracts are deployed to multiple chains.

Rollups and Cross-Chain Replay

For EVM-compatible rollups, it is also worth noting that when transactions are sent from L1 to L2, the address of the sender of the transaction on L2 will be set to the address of the sender of the transaction on L1. However, the address of the sender of a transaction on L2 will be different if the transaction was triggered by a smart contract on L1.

It is possible to have smart contracts on both the L1 and L2 with the same address but different bytecode (different implementations) due to the behavior of the CREATE

opcode. If the sender of the L2 transaction is an L1 smart contract with the same address as a contract on the L2, the transaction can be replayed on the L2 but using the L2 contract implementation. To mitigate this, some contracts on rollups are aliased to avoid them being called maliciously.

Mitigating Against Cross-Chain Replay

To prevent cross-chain replay attacks, use a chain-specific signature scheme such as [EIP-155

](https://eips.ethereum.org/EIPS/eip-155/), which includes the chain ID in the signed message. The signature should also be verified using the chain ID. This will prevent transactions signed on one chain from being replayed on another chain.

4. Signature Malleability

The elliptic curve (SECP256k1) used in ECDSA, as with all elliptic curves, is symmetrical about the x-axis. Therefore, for every (r,s,v)

, there exists another coordinate that returns the same valid result. This means that two signatures exist for one signer at any one point on the curve, allowing attackers to compute the second, valid signature without requiring the signer's private key.

SECP256k1 is the set of points on the curve $y^2 = x^3 + 7$

, used alongside ECDSA to generate keys.

This curve can be visualized as:

[Source

](https://medium.com/draftkings-engineering/signature-malleability-7a804429b14a): SECP256k1 curve

ECDSA signatures are represented as (r,s,v)

, where v

is used to determine the public key from the value of r

. Since the curve is reflected over the x-axis, there are two valid public keys for each value of r

.

Due to the x-axis symmetry, if (r,s)

is a valid signature, so is (r, -s mod n)

(where n

is the number of points defined in SECP256k1).

Key takeaway T he x-axis symmetry in the curve means that there are two valid signatures for each value of r .

To mitigate this issue, Ethereum underwent a hard fork: EIP-2. EIP-2 limited the s

values to prevent signature malleability by only allowing lower levels of s

.

By restricting the valid range in half, EIP-2 effectively removes half the points from the group, ensuring there is at most one valid point at each **x-**coordinate (value of r

) therefore one valid signature. However, these changes were not reflected in the precompiled contract implementing ecrecover

; therefore, using ecrecover

directly still presents an issue.

Example of Signature Malleability

The following [code

](https://github.com/code-423n4/2021-04-meebits/blob/2ec4ce8e98374be2048126485ad8ddacc2d36d2f/Beebots.sol#L556-L576) from Larva Labs' code4rena audit shows ecrecover

being used directly without checks on s

to restrict its value to only the lower levels:

Since s

is not restricted, two valid signatures exist and therefore the contract is vulnerable to signature malleability attacks.

In this Sherlock audit, the contract uses OpenZeppelin in a version lower than 4.7.3, in which the signature malleability bug is present. An attacker can use signature malleability to re-submit the request, as long as the old request has not yet expired. The issue is with the verification function ecrecover

allowing both the upper and lower values for s

.

In OpenZeppelin versions ≥ 4.7.3, tryRecover

checks that s

is restricted:

Mitigating Against Signature Malleability

When implementing ecrecover

directly, the value of s

should be restricted to only allow the lower levels.

This issue usually arises when ecrecover

is used directly; therefore, OpenZeppelin's ECDSA library (version 4.7.3 or greater) should be used.

To read more about signature malleability and how to prevent it, refer to this article by ImmuneFi.

5. Cryptographic Nonce Reuse

As mentioned earlier, the s

value of a signature is computed using a random number k

which represents the nonce. If two messages are signed with the same nonce, the ECDSA private key can be extracted. This means that attackers can compromise the signer's account and use the private key to sign malicious messages.

For a full derivation of how and why it is possible to recover a private key if the nonce is used more than once, refer to this comprehensive walkthrough.

Mitigating Against Nonce Reuse

Ensure that k

is unique for every signature

and never reused. It is also important that each k

value cannot be computed from a preceding k

value, e.g. k = k + 1

, otherwise if k

is known and therefore the private key can still be extracted. Using OpenZeppelin's ECDSA library (version 4.7.3 or greater) also prevents nonce reuse replay so is also advised.

Preventing Replay Attacks

To prevent signature replay attacks, smart contracts must:

1. Include a unique nonce that is validated for each signature
2. Set and check an expiration date
3. Restrict the s

value of the signature to only one-half

1. Include a chain ID
2. Include unique identifiers (e.g., if there are multiple messages to sign in the same contract/chain etc.)

Optional, but recommended:

1. Check signature length (check for EIP-2098)
2. Check if the claimant is a contract (ERC-1271: contract compatibility)
3. Check ecrecover

's return result

The following code is an example of this in practice:

Summary

There are many vectors in which smart contracts can be vulnerable to replay attacks. To ensure that attackers cannot replay transactions or reuse signatures, it is important to ensure that smart contracts are thoroughly tested and audited.

Care should be taken when signatures are being used, especially as an argument to a function.