PUBLIC

Code Assessment

of the Permit2 Smart Contracts

November 18, 2022

Produced for

by

Contents

1 Executive Summary

Dear Uniswap,

Thank you for trusting us to help Uniswap with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Permit2 according to Scope to support you in forming an opinion on their security risks.

Uniswap implements `Permit2` and `Permit2Lib` which are smart contracts that enable permit-style approvals and transfers using signatures for ERC20 tokens that do not support such functionality.

The most critical subjects covered in our audit are functional correctness, signature handling and front-running. Security regarding front-running is improvable due to a possible attack vector on permit approvals, see Race Condition on Approvals. Security regarding functional correctness and signature handling is high.

The general subjects covered are specification correctness and uncommon language features. Security regarding all the aforementioned subjects is high.

In summary, we find that the level of security of the codebase is high. Discovered issues do not render the contracts immediately unsafe, but enable potential human errors.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	1
• Risk Accepted	1
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the following source code files inside the `src` folder of Permit2 repository:

- AllowanceTransfer.sol
- EIP712.sol
- interfaces/IAllowanceTransfer.sol
- interfaces/IDAIPermit.sol
- interfaces/IERC1271.sol
- interfaces/ISignatureTransfer.sol
- libraries/Allowance.sol
- libraries/Permit2Lib.sol
- libraries/PermitHash.sol
- libraries/SafeCast160.sol - was added in **Version 2**
- libraries/SignatureVerification.sol
- Permit2.sol
- PermitErrors.sol
- SignatureTransfer.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 October 2022	e3e496f532792fb764eb61c6a95851fd873e5ae0	Initial Version
2	11 November 2022	8e981ae18fb29bbcfb539424c7f098e2559e83d6	Version with fixes
3	15 November 2022	12757bf42a030df007f3bd1d38404d86c3d29b44	Version with fixes
4	18 November 2022	9681052496b12ddc3cfb312ba10839a5c8090eba	Updated Permit2 address

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

2.1.1 Excluded from scope

Any imported libraries and contracts that are not mentioned in the Scope. Since we do not fully know how `Permit2Lib` will be used in other codebases, there is the possibility of misuse and, hence, its usage is out of scope.

2.1.2 Assumptions

For this assessment, it is assumed that reviewed contracts will be deployed and run on the Ethereum mainnet.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of the system overview, we list changes to the system introduced in later versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Uniswap implemented `Permit2` - an approval management system for ERC20 tokens. The existing EIP-2612 is an EIP-20 extension that allows EOA addresses to sign approval data off-chain to use the signature later in a `permit` function call to set the allowance. The `Permit2` contract can be seen as an intermediate contract that enables signature-based approval functionality on tokens that do not implement `EIP-2612` directly.

2.2.1 Permit2

The functionality of `Permit2` is derived from 2 contracts:

- `AllowanceTransfer`
- `SignatureTransfer`

2.2.1.1 `AllowanceTransfer`

This contract tracks allowance using the `PackedAllowance(uint160 amount, uint64 expiration, uint32 nonce)` struct for any owner, spender and token. The contract defines the following state-modifying functions:

- `approve` - sets allowance amount and expiration for given token and spender with `msg.sender` as an owner.
- `permit` - using a signature, sets allowance amount and expiration for a given token and spender. The provided owner should be the origin of the signature. The nonce of the allowance is increased by 1.
- `permitBatch` - batched version of `permit`. Only the nonce of the first token allowance is increased by 1.
- `transferFrom` - calls `token.transferFrom` after validation of approval expiration, and decreases the allowance by the transferred amount.
- `batchTransferFrom` - batched version of `transferFrom`.
- `lockdown` - for `msg.sender` as owner sets the allowance amount to 0, given an array of tokens and spenders.
- `invalidateNonces` - for `msg.sender` as the owner increases the allowance nonce by the specified amount, given a spender and token. The maximum increase is `uint16.max`.

2.2.1.2 SignatureTransfer

This contract allows `msg.sender` to transfer a specified amount of a specified token from the owner if the signature of the owner is provided. The transfer parameters must be hashed and signed according to the EIP-712 standard. Before the transfer, the deadline of each signature is checked to ensure that they are not expired yet. The `SignatureTransfer` contract tracks a `uint256 nonce` value for any given owner. Nonces are not incremental and can be used in random order. The contract defines the following state-modifying functions:

- `permitTransferFrom` - transfer a given amount of given tokens from the signer of the transfer params.
- `permitBatchTransferFrom` - batched version of `permitTransferFrom`.
- `permitWitnessTransferFrom` - version of `permitTransferFrom` with a `bytes32 witness` parameter. This is an EIP-712 hash of a witness struct that users can define themselves. Users need to provide `witnessTypeName` and `witnessType` string parameters that are needed for a proper definition of the EIP-712 `encodeType` function of this witness struct.
- `permitBatchWitnessTransferFrom` - batched version of `permitWitnessTransferFrom`.
- `invalidateUnorderedNonces` - allows `msg.sender` to invalidate own nonces.

The spender signed by the owner must be `msg.sender`. The functions that use `witness` receive string parameter `witnessTypeString`. The sender must provide the correct string for the hashing functions to be compliant with the `EIP-712`.

2.2.2 Permit2Lib

In addition to `Permit2`, Uniswap implements the `Permit2Lib` contract. This is a smart contract library that other smart contracts can use to `transferFrom` and `permit` ERC20 tokens with a call to `Permit2` as a fallback option. `Permit2Lib` contains two functions:

- `transferFrom2`. The execution logic of this function follows this flow:
  1. When this function is called, first a call to the `ERC20.transferFrom` function is performed.
  2. If the call from 1. fails or returns "false", a call to `Permit2.transferFrom` is performed.
  3. If the second call fails, `transferFrom2` reverts.
- `permit2`. The execution logic of this function follows this flow:
  1. When this function is called, an attempt to read an `ERC20.DOMAIN_SEPARATOR` is made.
  2. If the returned `DOMAIN_SEPARATOR` matches the one for the mainnet DAI token, the `DAI.permit` function is called. Otherwise, the `EIP-2612.permit` function is called.
  3. If any of the calls performed in 1. or 2. fail, a call to `Permit2.permit` is done. If it fails, `permit2` reverts.

2.2.3 Changes in Version 2

The following changes that affect the previous statements in this section were introduced in Version 2:

- `AllowanceTransfer.permitBatch` is renamed to `AllowanceTransfer.permit`. The arguments of this function are different compared to the non-batch version of the same named function.
- `AllowanceTransfer.batchTransferFrom` is renamed to `AllowanceTransfer.transferFrom`. The arguments of this function are different compared to the non-batch version of the same named function.
- `AllowanceTransfer.invalidateNonces` - for `msg.sender` as the owner specifies new nonce instead of delta. The maximum increase is `uint16.max`.

- `AllowanceTransfer` tracks allowance using the updated `PackedAllowance(uint160 amount, uint48 expiration, uint48 nonce)` struct. The bit sizes of the `expiration` and `nonce` fields were changed.
- `SignatureTransfer.permitBatchTransferFrom` is renamed to `SignatureTransfer.permitTransferFrom`. The arguments of this function are different compared to the non-batch version of the same named function.
- `SignatureTransfer.permitBatchWitnessTransferFrom` is renamed to `SignatureTransfer.permitWitnessTransferFrom`. The arguments of this function are different compared to the non-batch version of the same named function.
- `SignatureTransfer` functions with witness only need a single `witnessTypeString` parameter from user, instead of `witnessTypeName` and `witnessType` string parameters.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Race Condition on Approvals Risk Accepted

Low-Severity Findings	0

5.1 Race Condition on Approvals

Design Medium Version 1 Risk Accepted

Since there is no direct way to increase and decrease allowance relative to its current value, the function `AllowanceTransfer.approve()` has a race condition similar to one of ERC-20 approvals. Further details regarding the race condition can be found here.

Risk accepted:

Uniswap responded:

```
We opted not to address this issue. If users really care about this attack vector it
means they are likely signing a spender they don't fully trust, and they can always approve(x),
approve(0), approve(y). We also expose a lockdown function that can batch remove approvals for users,
before setting new approvals.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Permit2Lib Argument Casting Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	1

- CALL to DOMAIN_SEPARATOR() Code Corrected

6.1 Permit2Lib Argument Casting

Security High Version 1 Code Corrected

The functions `permit2` and `transferFrom2` of `Permit2Lib` both take `uint256 amount` as an argument. The lib will first attempt to call the `token` directly and falls back to the call to `Permit2` if it fails. However, the `Permit2.permit` and `Permit2.transferFrom` take `uint160 amount` as an argument. The initial `uint256 amount` will be cast to `uint160` for that call. Assuming some contract A relies on `transferFrom2` for token transfers, the following can happen:

1. The user calls a function on A that attempts to pull funds from the user using `transferFrom2`. For `amount`, the user specifies `2**170`.
2. A direct call to `token.transferFrom` fails.
3. Permit2Lib falls back to `Permit2.transferFrom` with `uint160(2**170) == 0` as an amount.
4. The call is successful. No value is actually transferred.
5. Contract A now thinks that `2**170` tokens were actually transferred.

Similar casting happens in the `permit2` function.

Code corrected:

The `SafeCast` library is now used for casting to a uint160 before the `Permit2` contract is called. The casting of a value that is greater than `type(uint160).max` would revert now.

6.2 CALL to DOMAIN_SEPARATOR()

Design Low Version 1 Code Corrected

EIP-712 defines the function `DOMAIN_SEPARATOR()` as a `view` function. Hence, it is expected to always work properly with `STATICCALL`. However, `Permit2Lib.permit2()` queries the domain separator with `CALL`, allowing the state to change in sub-calls as well as reentrancy. The contracts that will use the `Permit2Lib` could break unexpectedly.

Code corrected:

The `STATICCALL` is used to query the `DOMAIN_SEPARATOR` in Version 2 of the code.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Overflow Theoretically Possible for `AllowanceTransfer.nonces`

Note Version 1

Nonces are incremented with unchecked arithmetic. This means that incrementing them may lead to overflows, allowing for replay attacks. This is unlikely to happen solely through `permit`, which increases the nonce by one since the nonce is of type `uint32`. However, with `AllowanceTransfer.invalidateNonces()` overflows could happen after `65537` calls since it uses type `uint16`. Thus, signers can potentially endanger themselves by misusing the `invalidateNonces` function.

Version 2 changes: nonce is of type `uint48` in updated code. Thus, while the overflow is theoretically still possible, practically it is highly unlikely to happen.

7.2 Signature Malleability if Misused

Note Version 2

In the Version 2 of the code the `SignatureVerification.verify` function accepts EIP-2098 compact 64 byte signature in addition to the traditional 65 byte signature format. If the replay protection mechanism is implemented using the signature itself, an attack can be performed. The contracts of Permit2 use nonces for replay protection and thus are safe. But any reuse of the `SignatureVerification` library must be done with this attack in mind. OpenZeppelin library had such an incident before.

Also, the `SignatureVerification` does not perform checks described in Appendix F of the Ethereum Yellow paper e.g. `0 < s < secp256k1n ÷ 2 + 1`. Thus, for any given signature a signature with s-values in the upper range can be calculated. If the replay protection mechanism is implemented using the signature itself, an attack can be performed.

7.3 `invalidateUnorderedNonces` Possible Arguments

Note Version 1

`SignatureTransfer.invalidateUnorderedNonces` can invalidate nonces with `wordPos` values up to `uint256.max`. However `_useUnorderedNonce` can only invalidate up to `uint248.max`. This allows the invalidation of nonces that can never be used.