Universal Login
Smart Contracts Audit

INTRO

The Universal Login team is working on a decentralised protocol allowing user to easily access Ethereum dapps.. The team asked me to review and audit their smart contracts that implement the mechanism of user account management and transaction processing. .

The code which is audited is located in the open-source github repository UniversalLogin/UniversalLoginSDK.

The version of the code that is being reviewed was published as the commit `8b3701e99c03ab03b537014da96cd47fd62e152b`.

All of the issues are classified using the popular OWASP risk rating model. It estimates the severity taking into account both the likelihood of occurrence and the impact of consequences.

SUMMARY

The development team demonstrated great engineering skills implementing a complex meta-transaction protocol in a minimal and concise codebase. The architecture was carefully designed with every smart contract having a well-defined scope of responsibilities. The code is readable and popular Ethereum interfaces and standards are used widely.

There has been **no critical** error. **2 high** severity issues have been found that may strongly affect the network performance. Although the issues may have very significant consequences, the fixes are going to involve a minimal intervention in the code and no major updates to the smart contract architecture is required. There are also **3 medium and 5 low** severity errors with a lower significance that could be solved by adding extra validation and checks for edge case scenarios.

The team successfully implemented adequate fixes to all of the issues identified by introducing additional checks, validations and refactoring the code. For three of the issues affecting relayer gas costs, the team decided to implement safety measures on the relayer node directly to keep the smart-contract login concise, efficient and clear. This is a reasonable and pragmatic approach and should allow keeping the protocol usage costs for clients as low as possible.

ISSUES FOUND

Transaction censorship by a malicious relayer

Severity: **HIGH** (Impact: HIGH, Likelihood: MEDIUM)

PROBLEM

The executeSigned function increases the `nonce` of the `Wallet` even if the code execution by the `call` method at line 59 wasn't successful. A malicious relayer may cause any published transaction to fail, regardless of its code, by providing not enough gas for the `call` execution. At the same time, the relayer can pass just enough amount of gas to fully execute the rest of the method, increase the `nonce` and prevent any other relayer from processing a failed transaction again.

CONSEQUENCES

Because every message needs to be published to a blockchain network a malicious relayer may intercept a message and try to corrupt it by front-running and paying the amount of gas to cause it to fail. Although this behaviour incurs costs for the relayer to prevent every attempt of processing the submitted meta-transaction, the relayer may be economically motivated to block an important transaction such as sending a large amount of funds. In some circumstances, being able to increase the latency and delay the execution of a call.

RECOMMENDATIONS

I recommend validating the amount of gas provided for transaction submitted by a relayer. It will offer the network users the guarantee that certain methods are 100% sure to be fully executed and may fail only due to the errors on their side. It could be done by introducing a MINIMUM_GAS constant that defines the lower limit for relayer commitment and is checked at the beginning of the executeSigned function: `require(startingGas > MINIMUM_GAS);`

SOLUTION

The team fixed the problem by validating the gas amount provided by a relayer: `require(gasLimitExecution <= startingGas, "Relayer set gas limit too low");` *in this commit. This value is negotiated with the client in a transaction setup handshake.*

Avoiding relayer payments for a wallet creation

Severity: **HIGH** (Impact: HIGH, Likelihood: MEDIUM)

PROBLEM

The `createContract` method for the `WalletProxyFactory` contract accepts a signed message named `initializeWithENS` as one of the parameters. The message is supposed to trigger the invocation of the equally named `initializeWithENS` method from the proxied `Wallet` contract. However, no checks are done on the message content, so the caller is free to set up a proxy by calling an alternative method `initialize` which doesn't contain the refund logic and won't pay back the deployment costs to the relayer.

CONSEQUENCES

Lack of any validation of the passed initialisation message makes it possible to bypass the relayer refund method and create the wallet contract for free. Although the additional validation might be implemented by relayers, the protocol should not pass the responsibility for correctness of its payment logic to the client implementations.

RECOMMENDATIONS

I recommend validating the function signature that is included in the message to check if it's going to invoke the `initializeWithENS` method. It could be done by checking the first 4 bytes of the `initializeWithENS` parameter to be equal to `0x608e1818.`

SOLUTION

The team fixed the problem by adding a refund call to the basic initialise function: `refund(getDeploymentGasUsed(), gasPrice, gasToken, tx.origin);` *in this commit. Additionally, the relayer restricts functions that may be invoked only to these two methods.*

Griefing attack on relayers by spamming input data

Severity: **MEDIUM** (Impact: MEDIUM, Likelihood: MEDIUM)

PROBLEM

The executeSigned function from the `Executor` contract doesn't include the size of the `data` parameter in the calculation of relayer refund amount. Therefore a relayer is paying the cost of processing a large transaction input without being properly remunerated for the expenditure. The message signer doesn't bear any cost for the input data size so it's free to provide extremely large value if desired.

CONSEQUENCES

The lack of transaction data refund could be exploited as a griefing attack when an attacker doesn't get any direct benefit but causes the counterparty to incur much higher losses. Therefore, it could be used to destabilise the relayers network, draining their resources and discourage affected users from participating in the protocol.

Although this vulnerability may be avoided by having off-chain checks in the relayer node it passes the responsibility to the implementations and brings a risk of developing a malfunctioning client despite following the smart contracts protocol.

RECOMMENDATIONS

I recommend including the size of the `data` parameter that is being passed to the `executeSigned` function in the calculation of the `gasUsed` value that is used to trigger the refund of transaction costs to relayer.

SOLUTION

The team decided not to implement the on-chain verification due to the additional gas cost the goal to keep the smart-contracts as lean and efficient as possible. Instead, the safety measures are implemented on the relayer side, which bears the risk of the attack and is incentivised to accurately calculate the costs of gas. The final calculation must be accepted by the client in a transaction setup handshake eliminating the risk of a relayer overcharging for the data load.

Draining relayer gas with a malicious implementation of ENS

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `createContract` method for the `WalletProxyFactory` contract is supposed to register the wallet name on the Ethereum Name Service. It's done by the helper contract `ENSUtils` in the `registerENS` function. However, this function relies on the assumption that the caller passed references to the real ENS contracts as input parameters. Therefore, it's possible to abuse this mechanism by submitting references to a malicious contract that will drain the gas from relayer as there is no validation for the upper limit of gas consumed in the call.

CONSEQUENCES

Passing references to malicious contracts that will implement the `ENS` interface allows an attacker to provide an implementation which could consume a maximum allowed gas amount. The consumed gas could be used for the benefit of attacker either by loading and releasing the contract storage or by using a solution like the `gasToken`.

RECOMMENDATIONS

I recommend capping the amount of gas that could be spent on the ENS registration by a relayer. Such protection will cause providing a malicious implementation no longer beneficial for an attacker and remove the incentive for doing so.

SOLUTION

The team decided to implement the additional checks on the relayer side to keep the smart-contract logic minimal and efficient. The relayer could maintain a whitelist of secure ENS contracts which is a simple and practical solution. Additionally, the relayer is able to mitigate the risk of overpaying for a transaction by limiting the gas value attached to the transaction.

Accidently blocking access and freezing funds

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The removeKey function from the `KeyHolder` contract allows removing any key regardless if there are any more keys left to access the wallet. As a consequence, it's possible to irreversibly block the access to the wallet and freeze all of the funds that were left on the `Wallet` contract.

CONSEQUENCES

Removing a key could be a very dangerous operation leading to unintended consequences when executed by an inexperienced user. A mistake can lead to freezing funds or losing access to other valuable assets. Universal Login protocol offers a new quality of user experience of interacting with Ethereum, so it should minimize the possibility of simple human error to have very severe consequences.

RECOMMENDATIONS

I recommend minimizing the probability of making an irreversible mistake by requiring a user to provide a backup address when removing the last key by adding an optional parameter to the removeKey function. Alternatively, a user may just provide an address when all of the remaining funds should be sent while closing the account.

SOLUTION

The team fixed the problem by adding a modifier attached to the removeKey *function:* `require(keyCount > requiredSignatures, "More keys required");` *in this commit. Therefore, it is no longer possible to have fewer keys than required for authentication if a user remove one of the keys accidentally.*

It's possible to remove a non-existing key

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

PROBLEM

The removeKey(address _key) function from the `KeyHolder` contract doesn't check if the key exists before removing it. Therefore, it's possible to remove a single key a few times or to remove a random value by a mistake. The function always decrements the `keyCount` variable casing it to be inconsistent with the real number of keys stored by a contract.

CONSEQUENCES

Removing a non-existing key can block any further attempt to delete another key if a `keyCount` dropped to 0. It can also corrupt the execution of the `setRequiredSignatures` method which performs the validation of the required signatures number: `require(_requiredSignatures <= keyCount)`. Although the consequences could be mitigated by adding a random key-value, such an action could cause further problems if the mock keys are mistaken with the real ones.

RECOMMENDATIONS

I recommend adding an extra validation to check if a key exists before removing it in line 55 of the `KeyHolder` contract: `require(keyExist(_key), "Cannot remove a non-existing key");`

SOLUTION

The team fixed the problem by checking if a key exists before trying to remove it: `require(keyExist(_key), "Cannot remove a non-existing key");` *in this commit.*

Re-entrant multi-transaction execution

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

PROBLEM

The executeSigned function from the `Executor` contract is prone to a re-entrant access. An original message can call to a helper contract that will invoke the `Executor` back and pass another transaction. The content of this transaction is virtually impossible to verify as it could be dynamically substituted in the helper contract. Moreover, the hidden recurrent invocations will be executed with the context of the first meta-transaction parameters and could increase the nonce by any number if the pattern is executed multiple times.

CONSEQUENCES

Allowing uncontrolled re-entrant access to the `Executor` could break the relayer infrastructure if it relies on the assumption that every meta-transaction should lead to a single execution. It could lead to an inconsistency in nonce management and transaction ordering. To correctly handle all of the possible edge-cases relayers will need to implement much more complex logic.

RECOMMENDATIONS

I recommend preventing re-entrant meta-transactions by introducing a mutex in the implementation of the executeSigned function. I suggest using one of the available implementations of this pattern such as a simple lock described in this blog post.

SOLUTION

Implementing a reliable mechanism of detecting re-entrant calls requires maintaining additional data structures which will significantly increase the execution overhead for every transaction. Therefore, the team decided to add additional safety checks on the relayer side and reduce the risk of overspending by capping the relayer gas costs by setting up the `gasLimitExecution` parameter.

Inconsistent and minimal validation of signer keys

Severity: **LOW** (Impact: LOW, Likelihood: MEDIUM)

PROBLEM

The addKey function from the `KeyHolder` contract doesn't perform a full validation of the key candidates. First, it's inconsistent with the second method addKeys which additionally checks if an address isn't the contract itself. Moreover, both methods allow adding an address that points to an existing contract and therefore is not capable of signing a transaction.

CONSEQUENCES

Signing keys management is the most important part of the protocol from the security point of view, as it controls all of the access to the contracts behaviour. It's required that both interfaces for adding new controlling keys have the same restrictions.

Accepting all of the addresses including the ones occupied by a smart contract will allow an inexperienced user to delegate the control to the old wallet contract address and effectively block access to the wallet that is being updated.

RECOMMENDATIONS

I recommend moving the check if a candidate is not the contract from the addKeys method to the addKey and extend the check to prevent any contract from being added by checking the `extcodehash` or using one of the popular library methods like isContract from the OpenZeppelin.

SOLUTION

The team fixed the problem by validating the proposed keys: `require(!OpenZeppelinUpgradesAddress.isContract(_key), "Contract cannot be a key");` *in this commit* to check if given addresses aren't contracts. This is done by verification the `extcodehash.`

Using inconsistent libraries for signatures verification

Severity: **LOW** (Impact: LOW, Likelihood: MEDIUM)

PROBLEM

The Universal Login protocol validates transaction signatures in two places: in the `executeSigned` function of the `Executor` contract and the `createContract` function of the `WalletProxyFactory` . The first method utilised the in-house implementation of assembly code signature splitting implemented in by the `verifySignatures` function, while the second one utilises the OpenZeppelin's `recover` method.

CONSEQUENCES

Encoding the same feature in two different places using two different implementations is not considered a good coding practice as it increases the maintenance costs and may lead to a situation when a piece of code is updated in one place and not fixed in another.

More importantly, the `recover` approach has a slightly different semantics handling the edge cases of some external signing tools. Having an inconsistent signature verification process complicates the relayer implementation and can lead to errors if the difference is not managed correctly.

RECOMMENDATIONS

I recommend having a consistent implementation of signature verification and explicitly documenting if the protocol handles edge cases from external signing tools like those described by the ECDSA library.

SOLUTION

The team decided to work on an own version of signatures recovery procedure based on the popular OpenZeppelin ECDSA library implemented as a helper contract ECDSAUtils submitted in this commit.

Unused gasLimitExecution variable

Severity: **LOW** (Impact: LOW, Likelihood: MEDIUM)

PROBLEM

The `gasLimitExecution` variable is used for times in the `Executor` contract. It's treated as a required parameter for the message hash is required from the sender and further recalculated and verified on-chain. However, it's just a nonfunctional parameter that doesn't affect any execution logic at all.

CONSEQUENCES

Requiring a user to provide a `gasLimitExecution` may give a wrong impression that the execution will be capped up to that limit. Furthermore, any additional parameter adds to the protocol complexity and increases the execution costs being a waste of resources.

RECOMMENDATIONS

I recommend removing the unused parameter form both the `executeSigned` and `calculateMessageHash` functions.

SOLUTION

The team uses the `gasLimitExecution` *as a key mechanism to protect against malicious transactions draining relayers or causing unreasonable cost for users. The value is negotiated in a transaction setup handshake and verified on chain in two checks:*

```
require(gasLimitExecution <= startingGas, "Relayer set gas limit too low");
```

```
require(startingGas<=gasLimitExecution.add(gasLimitMargin()), "Relayer set gas limit too high");
```

in this commit .

Jakub Wojciechowski
Smart contract auditor