Ambire Contest
Findings & Analysis Report

2021-11-11

Overview

ABOUT C4

Code 423n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the Ambire smart contract system written in Solidity. The code contest took place between October 15—October 18 2021.

WARDENS

11 Wardens contributed reports to the Ambire code contest:

1. cmichel
2. gpersoon
3. WatchPug
    - jtp
    - ming
4. pauliax
5. pmerkleplant
6. ye0lde
7. loop
8. JMukesh
9. cryptojedi88
10. defsec

This contest was judged by AlexTheEntreprenerd.

Final report assembled by moneylegobatman .

Summary

The C4 analysis yielded an aggregated total of 41 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 0 received a risk rating in the category of MEDIUM severity, and 6 received a risk rating in the category of LOW severity.

C4 analysis also identified 10 non-critical recommendations and 21 gas optimizations.

Scope

The code under review can be found within the C4 Ambire code contest repository is composed of 8 smart contracts written in the Solidity programming language and includes 755 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] PREVENT EXECUTION WITH INVALID SIGNATURES

Submitted by gpersoon

Impact

Suppose one of the supplied `addrs\[i]` to the constructor of `Identity.sol` happens to be 0 (by accident).

In that case: `privileges\[0] = 1`

Now suppose you call `execute()` with an invalid signature, then `recoverAddrImpl` will return a value of 0 and thus signer=0. If you then check "`privileges\[signer] !=0`" this will be true and anyone can perform any transaction.

This is clearly an unwanted situation.

Proof of Concept

- `Identity.sol#L23` L30
- `Identity.sol#L97` L98

Recommended Mitigation Steps

In the constructor of `Identity.sol`, add in the for loop the following:

```
require (addrs\[i] !=0,"Zero not allowed");
```

Ivshti (Ambire) confirmed:

Ivshti (Ambire) patched:

> resolved in https://github.com/AmbireTech/adex-protocol-eth/commit/08d050676773fcdf7ec1c4eb53d51820b7e42534

GalloDaSballo (judge) commented:

> This seems to be the risk of having `erecover` returning zero, any invalid signature ends up being usable from any address to execute arbitrary logic.
>
> Mitigation can be achieved by either reverting when about to return address(0), which the sponsor has used for mitigation
>
> The other mitigation is to ensure that an account with address(0) cannot have privileges set to 1
>
> I believe mitigation from sponsor to be sufficient, however I'd recommend adding a check against having address(0) in the constructor for Identity.sol just to be sure

Ivshti (Ambire) commented:

> @GalloDeSballo an extra check is superfluous IMO, not only cause the revert on 0 in SIgnatureValidatorV2 guarantees that this is fixed, but also because it has to be in three places: constructor, setAddrPrivilege and the account creation system in `js/IdentityProxyDeploy` which rolls out bytecode that `sstores` privileges directly

[H-02] QuickAccManager.sol#cancel() Wrong hashTx makes it impossible to cancel a scheduled transaction

Submitted by WatchPug, also found by gpersoon

In `QuickAccManager.sol#cancel()`, the `hashTx` to identify the transaction to be canceled is wrong. The last parameter is missing.

As a result, users will be unable to cancel a scheduled transaction.

`QuickAccManager.sol#L91` L91

```
function cancel(Identity identity, QuickAccount calldata acc, uint nonce, bytes calldata s
    bytes32 accHash = keccak256(abi.encode(acc));
    require(identity.privileges(address(this)) == accHash, 'WRONG_ACC_OR_NO_PRIV');

    bytes32 hash = keccak256(abi.encode(CANCEL_PREFIX, address(this), block.chainid, accHa
    address signer = SignatureValidator.recoverAddr(hash, sig);
    require(signer == acc.one || signer == acc.two, 'INVALID_SIGNATURE');

    // @NOTE: should we allow cancelling even when it's matured? probably not, otherwise t
    // opportunity: someone wants to cancel post-maturity, and you front them with execSch
    bytes32 hashTx = keccak256(abi.encode(address(this), block.chainid, accHash, nonce, tx
    require(scheduled[hashTx] != 0 && block.timestamp < scheduled[hashTx], 'TOO_LATE');
    delete scheduled[hashTx];

    emit LogCancelled(hashTx, accHash, signer, block.timestamp);
}
```

Recommendation

Change to:

```
bytes32 hashTx = keccak256(abi.encode(address(this), block.chainid, accHash, nonce, txns,
```

Ivshti (Ambire) confirmed and resolved:

> Great find, resolved in https://github.com/AmbireTech/adex-protocol-eth/commit/5c5e6f0cb47e83793dafc08630577b93500c86ab

GalloDaSballo (judge) commented:

> The warden has found that the method `cancel` was calculating the wrong `hashTx`, this hash, used to verify which transaction to cancel, making it impossible to cancel a transaction.
>
> The sponsor has mitigated in a subsequent pr

[H-03] Signature replay attacks for different identities (nonce on wrong party)

Submitted by cmichel, also found by WatchPug

A single `QuickAccount` can serve as the "privilege" for multiple identities, see the comment in `QuickAccManager.sol`:

> NOTE: a single accHash can control multiple identities, as long as those identities set it's hash in privileges[address(this)]. this is by design

If there exist two different identities that *both share the same QuickAccount* (`identity1.privileges(address(this)) == identity2.privileges(address(this)) == accHash`) the following attack is possible in `QuickAccManager.send`:

Upon observing a valid `send` on the first identity, the same transactions can be replayed on the second identity by an attacker calling `send` with the same arguments and just changing the `identity` to the second identity.

This is because the `identity` is not part of the `hash`. Including the nonce of the identity in the hash is not enough.

Two fresh identities will both take on nonces on zero and lead to the same hash.

Impact

Transactions on one identity can be replayed on another one if it uses the same `QuickAccount`. For example, a transaction paying a contractor can be replayed by the contract on the second identity earning the payment twice.

Recommended Mitigation Steps

1. Nonces should not be indexed by the identity but by the `accHash`. This is because nonces are used to stop replay attacks and thus need to be on the *signer* (`QuickAccount` in this case), not on the target contract to call.
2. The `identity` *address* itself needs to be part of `hash` as otherwise the `send` can be frontrun and executed by anyone on the other identity by switching out the `identity` parameter.

Other occurrences

This issue of using the wrong nonce (on the `identity` which means the nonces repeat per identity) and not including `identity` address leads to other attacks throughout the `QuickAccManager`:

- `cancel`: attacker can use the same signature to cancel the same transactions on the second identity
- `execScheduled`: can frontrun this call and execute it on the second identity instead. This will make the original transaction fail as `scheduled[hash]` is deleted.
- `sendTransfer`: same transfers can be replayed on second identity
- `sendTxns`: same transactions can be replayed on second identity

Ivshti (Ambire) confirmed:

> duplicate of #24, but it's better documented

Ivshti (Ambire) patched:

> mitigation step 1 is not going to be done, since there's already plenty of upper level code relying on indexing by identity, and it doesn't really hurt if the replay attack is mitigated
>
> plus, it makes it harder to look up the nonce value, as we have to compute the accHash in the client-side code

> the replay attack has been fixed here https://github.com/AmbireTech/adex-protocol-eth/commit/f70ca38f368da30c9881d1ee5554fd0161c94486

GalloDaSballo (judge) commented:

> The warden identified a Signature Replay attack, allowing to re-use a signature throughout the system.
>
> Requiring the identity to be part of the signatures mitigates the vulnerability
>
> The sponsor has mitigated in a subsequent PR

[H-04] QUICKACCMANAGER SMART CONTRACT SIGNATURE VERIFICATION CAN BE EXPLOITED

Submitted by cmichel

Several different signature modes can be used and `Identity.execute` forwards the `signature` parameter to the `SignatureValidator` library. The returned `signer` is then used for the `privileges` check:

```
address signer = SignatureValidator.recoverAddrImpl(hash, signature, true);
// signer will be QuickAccountContract
require(privileges[signer] != bytes32(0), 'INSUFFICIENT_PRIVILEGE');
```

It's possible to create a smart contract mode signature (`SignatureMode.SmartWallet`) for arbitrary transactions as the `QuickAccManager.isValidSignature` uses an attacker-controlled `id` identity contract for the privileges check. An attacker can just create an attacker contract returning the desired values and the smart-wallet signature appears to be valid:

```
// @audit id is attacker-controlled
(address payable id, uint timelock, bytes memory sig1, bytes memory sig2) = abi.decode(sig
// @audit this may not be used for authorization, attacker can return desired value
if (Identity(id).privileges(address(this)) == accHash) {
  // bytes4(keccak256("isValidSignature(bytes32,bytes)")
  return 0x1626ba7e;
} else {
  return 0xffffffff;
}
```

POC

Assume an `Identity` contract is set up with a `QuickAccManager` as the `privileges` account, i.e. `privileges[accHash] != 0`.

We can construct a `SignatureMode.SmartWallet` signature for an *arbitrary* hash:

1. Call `Identity.execute(txns, spoofedSignature)` where `spoofedSignature = abi.encode(attackerContract, timelock=0, sig1=0, sig2=0, address(quickAccountManager), SignatureMode.SmartWallet)`
2. This will call `recoverAddrImpl(txnsHash, spoofedSignature, true)`, decode the bytes at the end of `spoofedSignature` and determine `mode = SignatureMode.SmartWallet` and `wallet = quickAccountManager`. It will cut off these arguments and call `quickAccountManager.isValidSignature(txnsHash, (attackerContract, 0, 0, 0))`

3. The `QuickAccManager` will decode the signature, construct `accHash` which is the hash of all zeroes (due to failed signatures returning 0). It will then call `attacker.privileges(address(this))` and the attacker contract can return the `accHash` that matches an account hash of failed signatures, i.e., `keccak256(abi.encode(QuickAccount(0,0,0)))`. The comparison is satisfied and it returns the success value.
4. The checks in `Identity.execute` pass and the transactions `txns` are executed.

Impact

Any `Identity` contract using `QuickAccManager` can be exploited. Funds can then be stolen from the wallet.

Recommendation

The issue is that `QuickAccManager` blindly trusts the values in `signature`. It might be enough to remove the `id` from the `signature` and use `msg.sender` as the identity instead: `Identity(msg.sender).privileges(address(this)) == accHash`. This seems to work with the current `Identity` implementation but might not work if this is extended and the `isValidSignature` is called from another contract and wants to verify a signature on a different identity. In that case, the `Identity/SignatureValidator` may not blindly forward the attacker-supplied signature and instead needs to re-encode the parameters with trusted values before calling `QuickAccManager`.

Ivshti (Ambire) confirmed and patched:

> great find! Mitigated in https://github.com/AmbireTech/adex-protocol-eth/commit/17c073d037ded76d56d6145faa92c1959fd47226 but still figuring out whether this is the best way to do it

GalloDaSballo (judge) commented:

> May need to sit on this one for another day before I can fully comment
>
> Fundamentally by calling Identity.execute with mostly 0 data, you are able to call back to `QuickAccManager. isValidSignature` which, due to the implementation of `ecrecover` at the time, will return valid checks for address(0), allowing to bypass all the logic and returning true for the signature, allowing for the execution of arbitrary code.
>
> Again, need to sit on this one
>
> But wouldn't you also be able to set a malicious smartContractWallet as the `IERC1271Wallet`, hence you can sidestep the entire logic, as your malicious contract wallet can be programmed to always return true on any input value?

Ivshti (Ambire) commented:

> @GalloDaSballo (judge) this doesn't have to do with `address(0)`
>
> Using smart wallets for signatures by itself is not a problem - since they authorize as themselves.
>
> The fundamental root of this issue is that ERC 1271 was designed with the assumption that 1 contract = 1 wallet. And as such, `isValidSignature` only returns `true`/`false`. This makes sense, as essentially we're asking the wallet "is this a valid signature from you", and then the wallet decides how to actually validate this it depending on it's own behavior and permissions.

> However, the QuickAccManager is a singleton contract - one single QuickAccManager represents multiple users. As such, combining it with ERC 1271 is a logical misunderstanding, as we can't really ask it "is this a valid sig for X identity" through the ERC 1271 interface. So instead, we encode the identity that we're signing as in the sig itself, but then a malicious user could call a top-level identity with a sig that validates in the singleton `QuickAccManager`, but meant to validate with a differerent identity.
>
> Because what we pass to `isValidSignature` is opaque data (the smart wallet may be any contract with any logic, not just our `QuickAccManager`) we can't just peak into the sig and see if it's meant to validate with the caller identity.
>
> Excellent finding IMO
>
> The current mitigation is hacky, and essentially leads to an `isValidSignature` implementation that is unusable (and doesn't make sense) off-chain, but we prefer it to introducing a new sig type especially for QuickAccManager.

GalloDaSballo (judge) commented:

> @Ivshti (Ambire) To clarify: Would adding `privileges[QuickAccountManager] = bytes32(uint(1))` enable the exploit?

Ivshti (Ambire) commented:

> @GalloDaSballo (judge) yes, it would. Any authorized quickAcc would enable the exploit

GalloDaSballo (judge) commented:

> I'm starting to get this
>
> The `id` sent to `isValidSignature` is an untrusted, unverified address The contract at that address can be programmed to have a function `privileges` which would return any `bytes32` value to match `accHash` This effectively allows to run arbitrary transactions.
>
> A way to mitigate would be to have a way to ensure the called `id` is trusted A registry of trusted ids may be effective
>
> The mitigation the sponsor has chosen does solve for only using trusted Identities as in the case of a malicious Identity, the Identity would just validate it's own transaction, not putting other Identities funds at risk.
>
> An alternative solution would be to change the `IdentityFactory` to use the OpenZeppelin Clones Library (or similar) to ensure that the correct Logic is deployed (by deploying a minimal-proxy pointing to the trusted implementation). This would require a fair tech-lift and would limit the type of deployments that the IdentityFactory can perform.
>
> The exploit was severe and the sponsor has mitigated by checking the `msg.sender` against the `id` provided in the signature

---

Low Risk Findings (6)

- [L-01] `ecrecover` may return empty address *Submitted by pauliax*
- [L-02] `block.chainid` may change in case of a hardfork *Submitted by pauliax*
- [L-03] Hardcoded WETH *Submitted by pauliax*
- [L-04] `Zapper` should `safeApprove(0)` first *Submitted by cmichel*

- [L-05] If zero address is added as privilege anyone can execute arbitrary transactions *Submitted by cmichel, also found by pmerkleplant*
- [L-06] Address with privilege for `QuickAccount` with `address(0)`'s can execute arbitrary transactions *Submitted by pmerkleplant*

Non-Critical Findings (10)

- [N-01] create2 assembly *Submitted by pauliax*
- [N-02] Hex selector *Submitted by pauliax*
- [N-03] Some code is commented out *Submitted by loop*
- [N-04] Inconsistent code style of for loops *Submitted by WatchPug*
- [N-05] lack of require message *Submitted by JMukesh*
- [N-06] use of floating pragma *Submitted by JMukesh*
- [N-07] No check for signature malleability *Submitted by cmichel*
- [N-08] `Identity` fallback returns too many bytes *Submitted by cmichel*
- [N-09] No ERC20 safe* versions called & no return values checked *Submitted by cmichel, also found by JMukesh, loop, cryptojedi88, defsec, and loop*
- [N-10] `Zapper` only works for whitelisted tokens *Submitted by cmichel*

Gas Optimizations (21)

- [G-01] `QuickAccManager.sol` Constants should be marked as `constant` *Submitted by WatchPug, also found by JMukesh*
- [G-02] `QuickAccManager.sol#send()` Avoid unnecessary read from storage can save gas *Submitted by WatchPug*
- [G-03] Cache array length in for loops can save gas *Submitted by WatchPug*
- [G-04] `Zapper.sol#wrapETH()` Use `WETH.deposit` can save some gas *Submitted by WatchPug, also found by cryptojedi88*
- [G-05] `Zapper.sol#tradeV3Single()` Remove unnecessary variable can make the code simpler and save gas *Submitted by WatchPug*
- [G-06] Unnecessary storage variables *Submitted by WatchPug, also found by pauliax and pmerkleplant*
- [G-07] Cache storage variables in the stack can save gas *Submitted by WatchPug, also found by pauliax*
- [G-08] Adding unchecked directive can save gas *Submitted by WatchPug, also found by pauliax and ye0lde*
- [G-09] Gas: `BytesLib` addition can be unchecked *Submitted by cmichel*
- [G-10] Gas: `SignatureValidatorV2.recoverAddrImpl` should use `else if` *Submitted by cmichel*
- [G-11] Safe some gas on the nonce increment *Submitted by gpersoon*
- [G-12] Compare with 0 and 1 in a more efficient way *Submitted by gpersoon*

- [G-13] IdentityFactory.withdraw can be external *Submitted by loop*
- [G-14] Duplicate math operations *Submitted by pauliax*
- [G-15] LibBytes uses itself *Submitted by pauliax*
- [G-16] Only prepare tx when the fee is present *Submitted by pauliax*
- [G-17] Set `QuickAccManager::DOMAIN_SEPARATOR` as immutable *Submitted by pmerkleplant*
- [G-18] Set `IdentityFactory::creator` as immutable *Submitted by pmerkleplant*
- [G-19] Set `QuickAccManager::CANCEL_PREFIX` as constant *Submitted by pmerkleplant*
- [G-20] Long Revert Strings *Submitted by ye0lde*
- [G-21] Assignment Of Variable To Default (`Identity.sol`) *Submitted by ye0lde*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.