Phat Contract Runtime Findings & Analysis Report

2024-04-11

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit 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 audit outlined in this document, C4 conducted an analysis of the Phat Contract Runtime smart contract system written in Rust. The audit took place between March 1 — March 22, 2024.

Wardens

19 Wardens contributed reports to Phat Contract Runtime:

1. DadeKuma
2. zhaojie
3. 0xTheC0der
4. Koolex
5. Cryptor
6. hunter_w3b
7. albahaca
8. popeye
9. ihtishamsudo
10. Bauchibred
11. XDZIBECX
12. Daniel526
13. fouzantanveer
14. 0xepley
15. aariiif
16. DarkTower (0xrex and haxatron)

17. rogueреggiant
18. kaveyjoe

This audit was judged by Lambda.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 7 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Phat Contract Runtime repository, and is composed of 13 smart contracts written in the Rust programming language and includes 2711 lines of Rust code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

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

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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (4)

[M-01] Limited availability of `balance_of(...)` method

Submitted by OxTheC0der

According to the documentation (online and in-line), the *availability* of the balance_of(...) method (see code below) should be **any contract** instead of **system only** which is caused by the present `ensure_system` check.

```
fn balance_of(
    &self,
    account: ext::AccountId,
) -> Result<(pink::Balance, pink::Balance), Self::Error> {
    self.ensure_system()?;    // @audit Availability should be 'any contract' instead of 'system only'
    let account: AccountId32 = account.convert_to();
    let total = crate::runtime::Balances::total_balance(&account);
    let free = crate::runtime::Balances::free_balance(&account);
    Ok((total, free))
}
```

The ensure_system(...) method returns a `BadOrigin` error in case the caller/origin is not the system contract.

```
fn ensure_system(&self) -> Result<(), DispatchError> {
    let contract: AccountId32 = self.address.convert_to();
    if Some(contract) != PalletPink::system_contract() {
        return Err(DispatchError::BadOrigin);
    }
    Ok(())
}
```

Consequence

The availability of the balance_of(...) method is limited to the system contract instead of being accessible to anyone. Therefore, user contracts relying on this method will inevitably fail.

For comparison:

The `import_latest_system_code(...)` method has consistent *system only* availability according to the implementation and documentation.

Proof of Concept

Please add the test case below to `phala-blockchain/crates/pink/runtime/tests/test_pink_contract.rs` and run it with `cargo test test_balance_of -- --nocapture`.

```
#[test]
fn test_balance_of() {
    const TEST_ADDRESS: AccountId32 = AccountId32::new([255u8; 32]);
```

```
        let (mut cluster, checker) = create_cluster();

        let balance = 114514;

        cluster.tx().deposit(TEST_ADDRESS.clone(), balance);

        let result = checker
            .call()
            .direct_balance_of(TEST_ADDRESS.convert_to())
            .query(&mut cluster);
        assert_eq!(result.unwrap(), (balance, balance));
    }
```

The test will fail with a **BadOrigin** error as discussed above:

```
called `Result::unwrap()` on an `Err` value: Failed to execute call: BadOrigin
```

Recommended Mitigation Steps

Remove the `ensure_system` check from the balance_of(...) method to ensure availability for **any contract**.

Assessed type

Invalid Validation

kvinwang (Phala) confirmed

[M-02] An attacker can bloat the Pink runtime storage with zero costs

Submitted by DadeKuma

An attacker can perform a bloat attack by creating a very high amount of dust accounts. This can occur with a minimal cost for the attacker, and results in a very high and increased cost in terms of storage and fees.

Proof of Concept

On substrate-based runtimes, the Existential Deposit (ED) is the minimum balance needed to have an active account, and it's useful to prevent dust accounts from bloating the storage state. In case the balance goes under the ED, the account will be reaped, (i.e. completely removed from storage) and the nonce reset.

Storage usage has a cost in terms of fees which impacts all the users, as there is a bigger overhead. The issue is that on the Pink runtime, the ED is dangerously low, which enables bloating attacks.

Let's see how feasible is to perform an attack from a price perspective. On Polkadot, the ED is `1 DOT`. At the moment of writing, `1 DOT` is worth about `$10 USD`, so if an attacker would want to split 10 million USD as dust between their own accounts, they would be able to create only 1 million accounts.

On the Pink runtime, this is not the case. At the moment of writing `1 PHA` is equal to `~$0.20 USD`; but here, the ED is only 1 unit:

```
        pub const ExistentialDeposit: Balance = 1;
```

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime.rs#L48

As balance is simply a type alias of `u128`:

```
        pub type Balance = u128;
```

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/types.rs#L11

And `1 PHA` is equal to `1_000_000_000_000`:

```
        pub const PHAS: Balance = 1_000_000_000_000;
        pub const DOLLARS: Balance = PHAS;
        pub const CENTS: Balance = DOLLARS / 100;
        pub const MILLICENTS: Balance = CENTS / 1_000;
```

This means that an account costs only `$0.20 / 1_000_000_000_000` for the attacker.

Following the previous `DOT` example, this means that an attacker could create **1 TRILLION accounts** by simply paying just `$0.20`, but they could create even more accounts (several order of magnitudes bigger) with a small "investment" to fill all the storage.

Recommended Mitigation Steps

Consider using a reasonable Existential Deposit. I recommend at least one `CENTS` (i.e. `1_000_000_000`):

```
-       pub const ExistentialDeposit: Balance = 1;
+       pub const ExistentialDeposit: Balance = 1 * CENTS;
```

Assessed type
Decimal

kvinwang (Phala) confirmed

[M-03] A cache that times out can be recovered

Submitted by zhaojie

Recover a cached value that has timed out, and a malicious user or contract can exploit this bug to fool other users or cause other unknown problems.

Proof of Concept
The `LocalCache#set_expire` function does not check whether the key has expired when setting a timeout:

```
    pub fn set_expire(&mut self, id: Cow<[u8]>, key: Cow<[u8]>, expire: u64) {
        //@audit key values that have timed out may not be deleted
        self.maybe_clear_expired();
        if expire == 0 {
            let _ = self.remove(id.as_ref(), key.as_ref());
        } else if let Some(v) = self
            .storages
            .get_mut(id.as_ref())
            .and_then(|storage| storage.kvs.get_mut(key.as_ref()))
        {
            //@audit You can increase the timeout period of a key value that has expired
            v.expire_at = now().saturating_add(expire)
        }
    }
    fn maybe_clear_expired(&mut self) {
        self.sets_since_last_gc += 1;
@>      if self.sets_since_last_gc == self.gc_interval {
            self.clear_expired();
        }
    }
```

As we can see from the above code, the `set_expire` function will first call `maybe_clear_expired`. The function `maybe_*` is called, so it won't necessarily delete keys that have expired. This function will not clean up expired keys until `gc count` reaches a certain value.

Therefore, if there are keys that have expired, they are still queried from `storages` and then reset the expiration time. In other words, the `set_expire` function can cause an expired key to be reactivated.

Let's look at another function, `LocalCache#get`:

```
    pub fn get(&self, id: &[u8], key: &[u8]) -> Option<Vec<u8>> {
        let entry = self.storages.get(id)?.kvs.get(key)?;
@>      if entry.expire_at <= now() {
            None
        } else {
            Some(entry.value.to_owned())
        }
    }
```

The `get` function returns `None` if it finds that key has expired; this results in inconsistency of cached data.

In this way, when a key value (such as balance signature or debt) has expired, the attacker declares that the value no longer exists. The user is then asked to take some action, so the victim queries `LocalCache#get` and then finds that the value indeed no longer exists. The problem is that an attacker can use `set_expire` to restore this value.

Another attack scenario is:

The key value (such as an nft) that the developer thinks has expired no longer exists. However, a malicious user can make this value expire indefinitely if `set_expire` can be called.

Tips:

`LocalCache#set` does not have this problem. `LocalCache#set` will call `Storage#set`, which will first call `self.remove` to remove the existing key.

Tools Used
VS Code

Recommended Mitigation Steps

```
    pub fn set_expire(&mut self, id: Cow<[u8]>, key: Cow<[u8]>, expire: u64) {
-       self.maybe_clear_expired();
+       self.clear_expired();
        if expire == 0 {
            let _ = self.remove(id.as_ref(), key.as_ref());
        } else if let Some(v) = self
            .storages
            .get_mut(id.as_ref())
            .and_then(|storage| storage.kvs.get_mut(key.as_ref()))
        {
            v.expire_at = now().saturating_add(expire)
        }
```

```
    }
```

kvinwang (Phala) confirmed, but disagreed with severity and commented:

> This is a good catch.
>
> I'm not sure if this should be classified as `High Risk` level. The purpose of the local cache is to store volatile data, such as information fetched from an HTTP server. This data is expected to be lost at any time and may vary between different workers for the same contract. If the data is lost, the contract will re-fetch it from the external system.
>
> Therefore, I don't think it is suitable for storing on-chain assets that users can query.

Lambda (judge) decreased severity to Medium and commented:

> Good finding, but agree that Medium is more appropriate, as the finding does not show any direct way how this can be used to steal funds, but only speculates about potential methods (which I am not sure if they can happen in practice and even if, they would have many assumptions).

DadeKuma (warden) commented:

> I'm a bit skeptical about this finding. First of all, the docs state that the cache will store off-chain computations and not on-chain data that users can query/call:
>
> //! The LocalCache provides a local KV cache for contracts to do some offchain computation. //! When we say local, it means that the data stored in the cache is different in different //! machines of the same contract. And the data might be lost when the runtime restart or caused //! by some kind of cache expiring mechanism.
>
> https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/chain-extension/src/local_cache.rs#L1-L4
>
> There is no proof that these off-chain computations can be leveraged to impact the protocol or leak value in any way, especially because:
>
> 1. This data is expected to be lost at any time and can vary between different workers.
> 2. These are off-chain computations which are not queriable by users.
>
> For these reasons, I believe this finding should be capped to QA/Low, not Medium risk.

zhaojie (warden) commented:

> Although the user cannot query the cached data directly, the user can query it indirectly through the contract. The cache stores off-chain data, which can also cause problems if it is recovered.
>
> For example, price data - the attacker lets the expired cache recover, there may be 2 different prices, which will lead to price manipulation. The report says that storing `xxxx` in the cache is just an assumption.
>
> Severity is determined by the judge, and I think it should at least remain Medium.

DadeKuma (warden) commented:

> ~~The `set_expire` function is never called inside the audit repository, nor in the main Phala repository, and~~ the only proof of how it will be used is inside the docs in the file itself, which points to off-chain computations.
>
> Using this cache to store on-chain data would be a misuse and a user error (and I don't think it would even make sense as the data is volatile/incongruent between the workers). This is also confirmed by the Sponsor in the comment above.
>
> Of course, the Judge will decide the final severity. I was just adding some details that might have been missed in the initial submission.
>
> **EDIT**: `set_expire` is actually called, GitHub search is broken; see comment below. My point on docs/normal usage stands.

zhaojie (warden) commented:

> @DadeKuma - You should search for `set_expiration`.

```
pub fn set_expiration(contract: &[u8], key: &[u8], expiration: u64) {
    with_global_cache(|cache| cache.set_expire(contract.into(), key.into(), expiration))
}
```

> https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/chain-extension/src/local_cache.rs#L273

Lambda (judge) commented:

> First of all, the docs state that the cache will store off-chain computations and not on-chain data that users can query/call:
>
> I agree with that. If it were used for on-chain data such as balances, High would be more appropriate. However, even for off-chain data, this behaviour could lead to problems. For instance, if the cache is used for caching some API/web responses (which seems to be one of the most common use cases for the cache), it is not unreasonable that a developer wants to

have a maximum age of the response (and with `cache_set_expire`, there is an exposed function for exactly doing that, which does not always correctly as the warden has shown).

> In these cases, it also does not matter that the data is volatile/different between workers, because you care about the maximum age, but fetching newer data is fine. One example I can think of is betting on some result of an API service that refreshes daily where you would set the expiration such that no new requests are made until the next refresh. Of course, this has some assumptions, but I think they are pretty reasonable for such a runtime and it is well possible that there could be contracts that trigger this issue.

[M-04] An attacker can crash the cluster system by sending an HTTP request with a huge timeout

Submitted by DadeKuma, also found by Koolex and zhaojie

Any user can intentionally crash a worker by sending a maliciously crafted request with a huge timeout. This attack has no costs for the attacker, and it can result in a DoS of the worker/cluster system.

Proof of Concept

A user can specify a timeout when doing a `batch_http_request`:

```
    pub fn batch_http_request(requests: Vec<HttpRequest>, timeout_ms: u64) -> ext::BatchHttpResult {
        const MAX_CONCURRENT_REQUESTS: usize = 5;
        if requests.len() > MAX_CONCURRENT_REQUESTS {
            return Err(ext::HttpRequestError::TooManyRequests);
        }
        block_on(async move {
            let futs = requests
                .into_iter()
                .map(|request| async_http_request(request, timeout_ms));
            tokio::time::timeout(
@>              Duration::from_millis(timeout_ms + 200),
                futures::future::join_all(futs),
            )
            .await
        })
        .or(Err(ext::HttpRequestError::Timeout))
    }
```

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/chain-extension/src/lib.rs#L56

The issue is that in Rust, the `+` operator can overflow when numerics bound are exceeded; this will result in a `panic` error.

When a malicious user sends a request with a timeout greater than `u64::MAX - 200`, they will crash the worker. As this action will cost nothing to the attacker, they can simply send multiple requests to crash all the workers, which will result in a DoS of the cluster system.

Coded PoC

Copy-paste the following test in `phala-blockchain/crates/pink/chain-extension/src/mock_ext.rs`:

```
    #[cfg(test)]
    mod tests {
        use crate::PinkRuntimeEnv;
        use pink::chain_extension::{HttpRequest, PinkExtBackend};

        use super::*;

        #[test]
        fn http_timeout_panics() {
            mock_all_ext();
            let ext = MockExtension;
            assert_eq!(ext.address(), &AccountId32::new([0; 32]));
            let responses = ext
                .batch_http_request(
                    vec![
                        HttpRequest {
                            method: "GET".into(),
                            url: "https://httpbin.org/get".into(),
                            body: Default::default(),
                            headers: Default::default(),
                        },
                        HttpRequest {
                            method: "GET".into(),
                            url: "https://httpbin.org/get".into(),
                            body: Default::default(),
                            headers: Default::default(),
                        },
                    ],
                    u64::MAX, //@audit this will cause an overflow
                )
                .unwrap()
                .unwrap();
            assert_eq!(responses.len(), 2);
            for response in responses {
                assert!(response.is_ok());
            }
        }
    }
```

Output:

```
    running 1 test
    thread 'mock_ext::tests::http_timeout_panics' panicked at crates/pink/chain-extension/src/lib.rs:66:35:
```

```
attempt to add with overflow
stack backtrace:
   0: std::panicking::begin_panic_handler
             at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/std/src/panicking.rs:645
   1: core::panicking::panic_fmt
             at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/panicking.rs:72
   2: core::panicking::panic
             at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/panicking.rs:127
   3: pink_chain_extension::batch_http_request::async_block$0
             at ./src/lib.rs:66
   4: tokio::runtime::park::impl$4::block_on::closure$0<enum2$<pink_chain_extension::batch_http_request::async_block_env$0> >
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/park.rs:282
   5: tokio::runtime::coop::with_budget
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/coop.rs:107
   6: tokio::runtime::coop::budget
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/coop.rs:73
   7: tokio::runtime::park::CachedParkThread::block_on<enum2$<pink_chain_extension::batch_http_request::async_block_env$0> >
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/park.rs:282
   8: tokio::runtime::context::blocking::BlockingRegionGuard::block_on<enum2$<pink_chain_extension::batch_http_request::asyn
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/context/blocking.rs:66
   9: tokio::runtime::scheduler::multi_thread::impl$0::block_on::closure$0<enum2$<pink_chain_extension::batch_http_request::
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/scheduler/multi_thread/mod.rs
  10: tokio::runtime::context::runtime::enter_runtime<tokio::runtime::scheduler::multi_thread::impl$0::block_on::closure_env
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/context/runtime.rs:65
  11: tokio::runtime::scheduler::multi_thread::MultiThread::block_on<enum2$<pink_chain_extension::batch_http_request::async_
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/scheduler/multi_thread/mod.rs
  12: tokio::runtime::runtime::Runtime::block_on<enum2$<pink_chain_extension::batch_http_request::async_block_env$0> >
             at /.cargo/registry/src/index.crates.io-6f17d22bba15001f/tokio-1.35.1/src/runtime/runtime.rs:350
  13: pink_chain_extension::block_on<enum2$<pink_chain_extension::batch_http_request::async_block_env$0> >
             at ./src/lib.rs:50
  14: pink_chain_extension::batch_http_request
             at ./src/lib.rs:61
  15: pink_chain_extension::impl$1::batch_http_request<pink_chain_extension::mock_ext::MockExtension,alloc::string::String>
             at ./src/lib.rs:192
  16: pink_chain_extension::mock_ext::impl$1::batch_http_request
             at ./src/mock_ext.rs:36
  17: pink_chain_extension::mock_ext::tests::http_timeout_panics
             at ./src/mock_ext.rs:198
  18: pink_chain_extension::mock_ext::tests::http_timeout_panics::closure$0
             at ./src/mock_ext.rs:194
  19: core::ops::function::FnOnce::call_once<pink_chain_extension::mock_ext::tests::http_timeout_panics::closure_env$0,tuple
             at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/ops/function.rs:250
  20: core::ops::function::FnOnce::call_once
             at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/ops/function.rs:250
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
test mock_ext::tests::http_timeout_panics ... FAILED
```

Recommended Mitigation Steps

Consider using `saturating_add` instead:

```
    tokio::time::timeout(
-       Duration::from_millis(timeout_ms + 200),
+       Duration::from_millis(timeout_ms.saturating_add(200)),
        futures::future::join_all(futs),
    )
```

Assessed type

Invalid Validation

kvinwang (Phala) confirmed, but disagreed with severity and commented:

> The runtime doesn't call the implementation directly. Instead, it calls into the worker, via `ocalls` here, and **the timeout is actually clamped in the worker side**. However, the suggested change is good to have. This might be a QA or Mid Risk level report.

Lambda (judge) commented:

> Not sure about the severity here. @kvinwang - Could you point out where the clamping happens? Because in the linked code it is a normal `u64` that could potentially be set to e.g. `u64::MAX - 1` to trigger the issue.

kvinwang (Phala) commented:

> This is the `OCalls` implementation in the worker, where the time remaining is less than the MAX*QUERY*TIME.

Lambda (judge) decreased severity to Medium and commented:

> Great, thanks for the link. In that case, I am downgrading this to a medium. It is not directly exploitable as an attacker, but the issue itself still exists within the codebase and if a future worker would integrate it differently/without limit, it could become exploitable.

Low Risk and Non-Critical Issues

For this audit, 7 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Cryptor** received the top score from the judge.

The following wardens also submitted reports: ihtishamsudo, XDZIBECX, 0xTheC0der, zhaojie, Bauchibred, *and* Daniel526.

[01] No point in using regular http request over `batchhttperequest`

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/capi/ecall_impl.rs#L299-L300

Since the len in `batchhttprequest` is not restricted to being `> 1`, a user can just use `batchhttperequest` over the regular `httprequest` as the former is less likely to revert due to the timeout bound being longer.

[02] `CallinCommand` is misleading and should be renamed `callinTransaction` for clarity

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/extension.rs#L336

[03] `is_running_in_command` function is missing or misnamed in the code

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/extension.rs#L439

The function `is_running_in_command` (from the documentation) is missing or misnamed in the code to be `is_it_in_transaction`.

[04] Functions that are not supported in transaction mode should revert not send empty arrays

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/extension.rs#L436

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/extension.rs#L473

Sending empty arrays could be misleading to users. It is recommended to revert and send a error that these functions are not support in transaction mode.

[05] Masking deposit calculation can lead to unexpected results in certain circumstances

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/contract.rs#L122-L135

The function coarse gas relies on complex bit math to mask deposits in order to mitigate side channel attacks. Therefore, we can consider that masked deposits must be at least higher or equal to the original deposit. However, this invariant can be broken.

Consider the fuzz test below:

```
#[test]
fn test_mask_deposit_properties(deposit in 1u128..=u128::MAX, deposit_per_byte in any::<u128>()) {
    let masked_deposit = mask_deposit(deposit, deposit_per_byte);

    // Property 1: Masked deposit should be greater than or equal to the original deposit.
    prop_assert!(masked_deposit >= deposit);
}
```

Here the invariant will be broken if `deposit_per_byte` is `= 664613997892457936451903530140172289`. This can be problematic if the cluster creator (the actor who sets the deposit per byte) is acting in a malicious way as this opens the door for the side channel attack that the function is supposed to prevent.

[06] `is_in_transaction` return value can be misleading

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/extension.rs#L439-L441

If a call is in estimating mode, then `is_in_transaction` will return true since estimating mode is treated the same as transaction, when in reality, it should be false. There should be a function to return whether the call is in estimating mode instead.

[07] No code size limit check in function `put_sidevm_code`

https://github.com/code-423n4/2024-03-phala-network/blob/a01ffbe992560d8d0f17deadfb9b9a2bed38377e/phala-blockchain/crates/pink/runtime/src/runtime/pallet_pink.rs#L134-L146

__ 141345 __ (lookout) commented:

> [01] - Non-Critical
> [02] - Non-Critical
> [03] - Non-Critical
> [04] - Non-Critical
> [05] - Low
> [06] - Non-Critical
> [07] - Non-Critical

kvinwang (Phala) confirmed and commented:

> [01] - Disagree, `len = 1` is a valid batch.
> [02] - Confirmed.
> [03] - Confirmed.

> [04] - Judgable. It did actually revert in the past, but the reverting can not be catched by the contract which leads bad dev experience.
> [05] If `deposit_per_byte is = 6646139978924579364519035301401722 89`, the cluster wouldn't be able to do anything as it is two expensive.
> [06] - Confirmed. This is actually on the plan.
> [07] - Confirmed. Might be good to have; however, there is already a on-chain limit check for that.

Lambda (judge) commented:

> Agree with the Lookout's assessment of L/NC and the sponsors assessment of the validity with the following differences:

> [01] - Valid design suggestion (not necessarily better than the current one, but just an alternative one).
> [04] - Also a design suggestion.

Audit Analysis

For this audit, 11 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **hunter_w3b** received the top score from the judge.

The following wardens also submitted reports: Cryptor, albahaca, popeye, fouzantanveer, 0xepley, aariiif, DarkTower, rogueregiant, kaveyjoe, *and* Bauchibred.

Description overview of Phala Network

Note: to view the provided image, please see the original submission here.

What is Phala Network?

Phala Network is revolutionizing Web3 by providing dApp developers with an off-chain compute infrastructure that is truly decentralized and trustless. By connecting Smart Contracts to our off-chain programs called Phat Contracts, developers can supercharge their dApps with seamless cross-chain integrations, connectivity to the internet, and heavy computation. Phat Contracts make your Smart Contracts even smarter, and can be integrated in minutes using our no-code developer experience Phat Bricks.

Web3 developers are constantly pushing up against the technical limitations of building on-chain. Modern dApps need more than just Smart Contracts to support rich feature sets, and as Web3 has evolved and matured, it has become clear that efficient off-chain computation will be vital for a number of standard dApp use cases. Phala gives dApp developers access to powerful off-chain services without compromising the principles of Web3. This is computation as it's meant to be.

The platform is built with multiple layers of security guarantees to ensure fully verifiable computation. Phala Network's compute providers, called Workers, execute computations faithfully and securely, backed by tokenomic incentives, hardware-based assurances, and cryptographic evidence of execution published and verified on the Phala blockchain.

Developers can deploy Phat Contracts using Phala Network's low-code experience, Phat Contract 2.0, which utilizes pre-written, audited Phat Contracts to create complex functions using TypeScript/JavaScript. Experienced developers can also use the Phat Contract Rust SDK to write custom programs for various use cases.

Some use cases for Phat Contracts include connecting Smart Contracts to APIs, interfacing with S3 storage platforms, computing over data, and automating Smart Contracts.

Pink Runtime is the ink! contract execution engine for Phala Network, built on Substrate's pallet-contracts with custom chain extensions. It executes Smart Contracts on the Phala Network and runs inside the off-chain TEE workers. The system architecture consists of two main components: the chain and the worker, with Pink Runtime operational inside the worker. Messages from end-users reach Pink Runtime via on-chain transactions or off-chain RPC queries.

System Overview

Scope

1. **runtime/src/runtime.rs:** Defines a custom runtime called `PinkRuntime` that includes several pallets such as `frame_system`, `pallet_timestamp`, `pallet_balances`, `pallet_insecure_randomness_collective_flip`, `pallet_contracts`, and `pallet_pink`.

    1. **Dependencies**: The code imports necessary modules and defines the types for some common Substrate components like `AccountId`, `Balance`, `BlockNumber`, `Hash`, `Nonce`, etc.
    2. **Runtime Construction**: The `construct_runtime!` macro is used to define the `PinkRuntime` structure, which includes several pallets. These pallets provide different functionalities to the runtime.
    3. **Parameter Types**: The `parameter_types!` macro is used to define constants for the runtime. These constants are used throughout the runtime and its pallets.
    4. **Config Implementations**: The code implements the `Config` trait for `PinkRuntime` for different pallets. This trait defines the configuration for each pallet, including the types for various components and some constant values.
    5. **Migrations**: The `pallet_contracts::Config` implementation includes a `Migrations` type that defines the migrations for the contracts pallet. These migrations are used to handle changes in the contracts pallet's storage between different versions.
    6. **Genesis and Runtime Upgrade Functions**: The `on_genesis` and `on_runtime_upgrade` functions are defined to handle the initialization and upgrade of the runtime, respectively.

7. **Metadata Check**: The `check_metadata` function is used to check if the runtime's metadata has changed. If the metadata has changed, the function writes the new metadata to a file and returns an error.
8. **The code defines various parameters for the runtime, including:**
9. `BlockHashCount`: The number of block hashes to keep in the chain.
10. `RuntimeBlockWeights`: The weight limits for different types of transactions.
11. `ExistentialDeposit`: The minimum balance required for an account to be considered active.
12. `MaxLocks`: The maximum number of locks that can be placed on an account.
13. `MaxReserves`: The maximum number of reserves that can be placed on an account.
14. `MaxHolds`: The maximum number of holds that can be placed on an account.
15. `DefaultDepositLimit`: The default deposit limit for smart contracts.
16. `MaxCodeLen`: The maximum length of smart contract code.
17. `MaxStorageKeyLen`: The maximum length of smart contract storage keys.

2. **runtime/src/contract.rs:** The contract provides functions for `instantiating` and executing contracts in a blockchain runtime. It also includes functions for masking certain values to make them `coarse-grained`, which can be useful for privacy or performance reasons. The code uses Substrate's `pallet_contracts` library to handle contract-related operations.

    1. **Functions**:
    2. `mask_low_bits64` **and** `mask_low_bits128`: These functions mask the lowest `bits` bits of a 64-bit or 128-bit integer, respectively. They are used to coarse-grain gas and storage deposit values to reduce the number of precision bits that need to be stored.
    3. `mask_deposit`: This function masks a storage deposit value to a specific granularity based on the `deposit_per_byte` parameter.
    4. `mask_gas`: This function masks a gas weight value to a specific granularity.
    5. `coarse_grained`: This function applies the `mask_gas` and `mask_deposit` functions to a `ContractResult` to coarse-grain its gas and storage deposit values.
    6. `check_instantiate_result`: This function checks the result of a contract instantiation and returns the account ID of the newly created contract if the instantiation was successful.
    7. `instantiate`: This function instantiates a contract with the given code hash, input data, salt, execution mode, and transaction arguments.
    8. `bare_call`: This function calls a method on a contract with the given address, input data, execution mode, and transaction arguments.
    9. `contract_tx`: This function executes a contract transaction and handles gas payment and refund.
    10. **Type Definitions**: The code defines several type aliases for the results of contract instantiation and execution.
    11. **Macro Definition**: The `define_mask_fn` macro is defined to generate functions for masking the lowest bits of a given number. The macro is used to define two functions: `mask_low_bits64` and `mask_low_bits128`.
    12. **Mask Functions**: The `mask_deposit` and `mask_gas` functions use the `mask_low_bits64` and `mask_low_bits128` functions to mask the deposit and gas values, respectively. The `mask_deposit` function masks the deposit value based on the deposit per byte, while the `mask_gas` function masks the gas value based on the weight's ref time.
    13. **Test Function**: The `mask_low_bits_works` function tests the correctness of the `mask_low_bits64` function for various inputs.
    14. **Coarse-Grained Function:** The `coarse_grained` function masks the gas consumed, gas required, and storage deposit values of a contract execution result.
    15. **Instantiation Function:** The `instantiate` function instantiates a new contract with the given code hash, input data, salt, and execution mode. It also logs the instantiation result.
    16. **Call Function**: The `bare_call` function calls a method of a contract with the given address, input data, execution mode, and transaction arguments.
    17. **Contract Transaction Function**: The `contract_tx` function is a helper function that pays for gas and executes a contract transaction. It also refunds any remaining gas after the transaction.

3. **runtime/src/storage/mod.rs:** The contract defines a `Storage` trait and its implementation for managing storage in the Pink runtime environment. It provides methods for executing runtime code using the `Storage` as a backend and committing changes to the underlying storage backend.

    1. **Key Components**:
    2. `Storage` **Trait:** Defines methods for committing transactions and executing runtime code.
    3. `Storage` **Implementation:** Implements the `Storage` trait using a provided storage backend.
    4. `execute_with` **Method:** Executes runtime code using the `Storage` as a backend and returns the result, side effects, and committed changes.
    5. `execute_mut` **Method:** Similar to `execute_with`, but commits the storage changes to the backend.
    6. `changes_transaction` **Method:** Converts overlay changes into a backend transaction.
    7. `commit_changes` **Method:** Commits storage changes to the backend.
    8. `get` **Method:** Retrieves the storage value for a specified key.
    9. **Usage:** The `Storage` trait and its implementation can be used to manage storage in the Pink runtime environment. Developers can use the `execute_with` or `execute_mut` methods to execute runtime code and make changes to the storage. The

`changes_transaction` and `commit_changes` methods can be used to commit the changes to the backend.

10. **Additional Notes:**
11. The contract includes a function called `maybe_emit_system_event_block` that allows Pink runtime to emit runtime events for external accessibility.
12. The `execute_with` and `execute_mut` methods take an `ExecContext` as an argument, which provides information about the current execution context.
13. The `Storage` implementation uses an overlayed changes mechanism to track changes made to the storage during runtime execution. allows changes to be committed to the storage backend if they are valid.

4. **runtime/src/storage/external_backend.rs:** The contract defines an `ExternalDB` struct that implements the `TrieBackendStorage` trait, which is a necessary requirement for `TrieBackend`. It also defines an `ExternalBackend` type that is a `TrieBackend` using `ExternalDB` as its storage backend, and an `ExternalStorage` type that is a `Storage` struct using `ExternalBackend` as its backend.

    1. **Key Components:**
    2. `ExternalDB` **Struct:** Implements the `TrieBackendStorage` trait, which provides methods for reading and writing key-value pairs from a storage backend.
    3. `ExternalBackend` **Type:** A `TrieBackend` that uses `ExternalDB` as its storage backend.
    4. `ExternalStorage` **Type:** A `Storage` struct that uses `ExternalBackend` as its backend.
    5. `code_exists` **Function:** Checks the existence of a particular ink code in the storage.
    6. **Usage:**
    7. The `ExternalDB` struct can be used to create a `TrieBackend` that can be used to manage storage in the Pink runtime environment.
    8. The `ExternalStorage` struct can be used to execute runtime code and make changes to the storage. The `code_exists` function can be used to check the existence of a particular ink code in the storage.
    9. **Additional Notes:**
    10. The `ExternalDB` struct does not manage any key-value backend by itself. Instead, it delegates the key-value reads and writes to the host via ocalls.
    11. The `ExternalBackend` type implements the `CommitTransaction` trait, which allows changes made to the storage to be committed to the backend.
    12. The `ExternalStorage` struct provides a number of methods for executing runtime code and managing storage.
    13. The `code_exists` function uses the `twox_128` hashing function to generate a key for the code info of a particular ink code.

5. **runtime/src/runtime/pallet_pink.rs:** The contract defines a pallet for managing the `cluster-wide` configuration and resources for the `Phala Network`. It includes storage items for `cluster ID`, `gas price`, `deposit per byte`, `deposit per item`, `treasury account`, `private key`, `sidevm codes`, `system contract address`, `next event block number`, and `last event block hash`. It also defines functions for setting these values and managing gas payments and refunds.

    1. **Key Components:**
    2. `ClusterId`: Stores the cluster ID.
    3. `GasPrice`: Stores the gas price.
    4. `DepositPerByte`: Stores the deposit per byte.
    5. `DepositPerItem`: Stores the deposit per item.
    6. `TreasuryAccount`: Stores the treasury account.
    7. `Key`: Stores the private key.
    8. `SidevmCodes`: Stores the uploaded sidevm codes.
    9. `SystemContract`: Stores the system contract address.
    10. `NextEventBlockNumber`: Stores the next event block number.
    11. `LastEventBlockHash`: Stores the last event block hash.
    12. **Usage:** The `pallet` can be used to manage the cluster-wide configuration and resources for the Phala Network. Developers can use the functions provided by the `pallet` to set the `cluster ID`, gas price, deposit per byte, deposit per item, treasury account, private key, sidevm codes, system contract address, next event block number, and last event block hash. They can also use the pallet to manage gas payments and refunds.
    13. **Additional Notes:**
    14. The pallet implements the `AddressGenerator` trait from the `pallet-contracts` crate, which allows it to generate contract addresses.
    15. The pallet uses the `convert` trait to convert between weight and balance.

6. **runtime/src/capi/mod.rs:** This contract is protected area in the memory that isolates sensitive data and code from Phala Network.

    1. **Imports**: The code starts with importing necessary modules and libraries. `pink_capi::v1` is the API used for interaction between the secure enclave and the untrusted part of the application. `phala_sanitized_logger` is used for logging.
    2. **Entry Point (** `__pink_runtime_init` **)**: This is the entry point of the runtime. It initializes the runtime and fills the ecalls table. It takes a configuration pointer and an ecalls pointer as arguments. It checks if the ocalls (Outside Calls) function can be set correctly and if the ecalls pointer is not null. If either check fails, it logs an error and returns -1. If the runtime is a dynamic library (`config.is_dylib != 0`), it initializes the logger.
    3. `get_version` **function**: This function retrieves the version of the runtime. It takes two pointers to u32 as arguments, which it fills with the major and minor version numbers.

4. `ecall` **function:** This function serves as the central hub for all 'ecall' functions (Enclave Calls). Upon invocation, it routes function calls by invoking the `ecall::dispatch` function. It takes a unique identifier for the function call (`call_id`), a pointer to the input data buffer (`data`), the size of the input data buffer (`len`), a mutable pointer to the context (`ctx`), and a function pointer to receive the SCALE encoded function return value (`output_fn`).
5. **Modules:** The code ends with two modules `ecall_impl` and `ocall_impl` which are likely to contain the implementation details of the ecalls and ocalls respectively.

7. **runtime/src/capi/ecall_impl.rs:** The `ecall_impl` contract is an implementation of the `ECalls` trait for the Pink runtime. It provides a set of functions that allow the Pink runtime to interact with the blockchain and execute contracts.

    1. **Functions:**
    2. `cluster_id`: Returns the cluster ID of the current runtime.
    3. `setup`: Sets up the cluster with the given configuration.
    4. `deposit`: Deposits the given amount of tokens into the account of the given user.
    5. `set_key`: Sets the secret key for the runtime.
    6. `get_key`: Returns the secret key for the runtime.
    7. `upload_code`: Uploads the given code to the runtime.
    8. `upload_sidevm_code`: Uploads the given sidevm code to the runtime.
    9. `get_sidevm_code`: Returns the sidevm code for the given hash.
    10. `system_contract`: Returns the address of the system contract.
    11. `free_balance`: Returns the free balance of the given account.
    12. `total_balance`: Returns the total balance of the given account.
    13. `code_hash`: Returns the code hash for the given account.
    14. `contract_instantiate`: Instantiates the given contract with the given input data.
    15. `contract_call`: Calls the given contract with the given input data.
    16. `git_revision`: Returns the git revision of the runtime.
    17. `on_genesis`: Called when the runtime is created.
    18. `on_runtime_upgrade`: Called when the runtime is upgraded.
    19. `on_idle`: Called when the runtime is idle.
    20. **Usage:** The contract can be used to interact with the Pink runtime in a variety of ways. For example, it can be used to:
    21. Create a new cluster.
    22. Deposit tokens into an account.
    23. Upload code to the runtime.
    24. Instantiate a contract.
    25. Call a contract.
    26. Get the git revision of the runtime.
    27. **Security Considerations:** The contract should be used with caution, as it can be used to perform malicious actions on the Phala Network. For example, it could be used to:

        - Create a new cluster with a malicious configuration.
        - Deposit tokens into an account that is controlled by an attacker.
        - Upload malicious code to the runtime.
        - Instantiate a contract that is designed to steal funds.
        - Call a contract in a way that causes it to revert.

8. **runtime/src/capi/ocall_impl.rs:** Defines a function `set_ocall_fn` that takes a set of ocall functions and sets them as the default ocall functions for the runtime. It also defines an `OCallImpl` struct that implements the `CrossCall` and `CrossCallMut` traits, which are used to make cross-calls from Rust to the host environment.

    - The `OCALL` static variable is used to store the default ocall function. The `_default_ocall` function is a placeholder ocall function that panics if it is called. The `set_ocall_fn` function sets the `OCALL` variable to the provided ocall function.
    - The `OCallImpl` struct implements the `CrossCall` and `CrossCallMut` traits. The `cross_call` method makes a cross-call to the host environment with the specified call ID and data. The `cross_call_mut` method does the same, but it allows the caller to modify the data before it is sent to the host environment.
    - The `allocator` module defines a custom memory allocator that delegates calls to the allocator in the host environment. This is done to ensure that all memory allocations and deallocations are accounted for by the host environment's allocator.
    - **Additional Notes:**
    - The contract uses the `pink_capi` crate, which provides a set of macros and functions for interfacing with the Pink runtime environment.
    - The contract assumes that the host environment provides an ocall function and optionally allocation and deallocation functions.
    - The `allocator` module is optional and can be used to delegate memory allocation and deallocation to the host's allocator.

9. **pink/runtime/src/runtime/extension.rs:** The `extension.rs` contract is an implementation of the `ChainExtension` trait for the Substrate framework, specifically designed for interacting with `pink contracts` within the Phala Network. It serves as a bridge between the Phala runtime and the pink contract execution environment, enabling communication and functionality exchange between the two.

    1. **Key Features:**
    2. **Chain extension for pink contracts:** The contract acts as an extension to the Substrate chain, providing an interface for pink contracts to access various functionalities and

interact with the runtime.

3. **Runtime environment:** It implements the `PinkRuntimeEnv` trait, defining the necessary methods for pink contracts to access information and perform actions within the Phala runtime environment.
4. **Extension backend:** The contract also implements the `PinkExtBackend` trait, providing a set of methods that pink contracts can call to access extended functionalities, such as HTTP requests, cryptography, and more.
5. **Side effects handling:** The contract tracks and manages side effects resulting from pink contract executions, including emitted events, instantiated contracts, and system events.
6. **Deterministic return values:** The contract ensures that the return values from pink contract executions are deterministic, preventing manipulation or exploitation.
7. **Functionality:**
8. **HTTP requests:** Pink contracts can make HTTP requests to external APIs or services.
9. **Batch HTTP requests:** Pink contracts can send multiple HTTP requests in a batch, optimizing network usage.
10. **Cryptography:** Pink contracts can perform various cryptographic operations, such as signing and verifying messages, deriving keys, and generating random numbers.
11. **Cache management:** Pink contracts can set, get, and remove data from the cache, providing temporary storage for frequently accessed data.
12. **Logging:** Pink contracts can log messages at different levels, allowing developers to debug and monitor contract behavior.
13. **Event emission:** Pink contracts can emit events that are captured and processed by the Phala runtime, enabling communication between contracts and the external world.
14. **Balance retrieval:** Pink contracts can query the balance of an account, providing access to financial information.
15. **Runtime version:** Pink contracts can obtain the version of the Phala runtime they are executing in.
16. **Current event chain head:** Pink contracts can access information about the current event chain head, allowing them to synchronize with the latest blockchain state.
17. **Implementation Details:**
18. **CallInQuery:** Represents the contract's runtime environment in a query mode, where side effects are not allowed.
19. **CallInCommand:** Represents the contract's runtime environment in a command mode, where side effects are permitted.
20. **PinkExtension:** The main chain extension implementation that handles calls from pink contracts and provides access to the extended functionalities.
21. **Usage:** Pink contracts can interact with the provided contract by calling the `chain_extension` intrinsic function with the extension ID set to 0. This will trigger the execution of the `PinkExtension` contract, allowing pink contracts to access the available `functionalities.sgx_quote()` - Gets the SGX quote of the worker.

10. **pink/capi/src/v1/mod.rs:** This contract defines the interfaces for `ECalls` and `OCalls` between the Pink runtime and the host environment.

    1. **ECalls (External Calls):**

       - `cluster_id`: Returns the ID of the cluster.
       - `setup`: Initializes the cluster with the provided configuration.
       - `deposit`: Mints a specified amount of balance for an account.
       - `set_key`: Sets the key of the cluster.
       - `get_key`: Returns the current cluster key.
       - `upload_code`: Uploads ink code WASM to the storage.
       - `upload_sidevm_code`: Uploads sidevm code to the account.
       - `get_sidevm_code`: Returns the sidevm code associated with a given hash.
       - `system_contract`: Returns the address of the system contract.
       - `free_balance`: Returns the free balance of the specified account.
       - `total_balance`: Returns the total balance of the specified account.
       - `code_hash`: Returns the hash of the code from the specified contract address.
       - `contract_instantiate`: Executes a contract instantiation with the provided arguments.
       - `contract_call`: Executes a contract call with the specified parameters.
       - `git_revision`: Returns the git revision that compiled the runtime.
       - `on_genesis`: Called when the cluster is created.
       - `on_runtime_upgrade`: Called right after the runtime is upgraded.
       - `on_idle`: Called once per block.

    2. **OCalls (Outward Calls):**

       - `storage_root`: Returns the storage root hash.
       - `storage_get`: Fetches a value from storage.
       - `storage_commit`: Commits changes to the storage and sets the new storage root hash.
       - `log_to_server`: Sends a log message from a contract to the log collection server.
       - `emit_side_effects`: Emits the side effects that occurred during contract execution.
       - `exec_context`: Returns the current execution context.
       - `worker_pubkey`: Returns the public key of the worker.

- `cache_get`: Fetches a cache value that is associated with the specified contract and key.
- `cache_set`: Sets a cache value associated with the specified contract and key.
- `cache_set_expiration`: Sets an expiration time for the cache value associated with the specified contract and key.
- `cache_remove`: Removes a cache value associated with a specified contract and key.
- `latest_system_code`: Returns the latest available system contract code.
- `http_request`: Performs a HTTP(S) request on behalf of the contract.
- `batch_http_request`: Performs a batch of HTTP(S) requests on behalf of the contract within a specified timeout period.
- `emit_system_event_block`: Emits a system event block that includes all events extracted from the current contract call.
- `contract_call_nonce`: Gets the nonce of the current contract call (if available).
- `entry_contract`: Returns the address of the entry contract if the execution was triggered by a contract call.
- `js_eval`: Evaluates a set of JavaScript code using the QuickJS engine running in SideVM.
- `origin`: Get the origin of the transaction (if available).
- `worker_sgx_quote`: Returns the SGX quote of the worker.

11. **pink/capi/src/types.rs:**

    1. **Types:**

       - `Hash`: A type representing a hash value.
       - `Hashing`: A type representing a hashing algorithm.
       - `AccountId`: A type representing an account ID.
       - `Balance`: A type representing a balance.
       - `BlockNumber`: A type representing a block number.
       - `Index`: A type representing an index.
       - `Address`: A type representing an address.
       - `Weight`: A type representing a weight.

    2. **Enums:**

       - `ExecutionMode`: An enum representing the mode in which the runtime is currently executing.
       - `ExecSideEffects`: An enum representing events emitted by contracts which can potentially lead to further actions by the runtime.

    3. **ExecutionMode:** The `ExecutionMode` enum has three variants:

       - `Query`: In this mode, the runtime is executing an RPC query. Any state changes are discarded after execution. Indeterministic operations like HTTP requests are allowed in this mode.
       - `Estimating`: In this mode, the runtime is simulating a transaction but state changes are discarded.
       - `Transaction`: In this mode, the runtime is executing a real transaction. State changes will be committed. Indeterministic operations like HTTP requests aren't allowed in this mode.

    4. **ExecSideEffects:** The `ExecSideEffects` enum has one variant:

       - `V1`: This variant contains three fields:
       - `pink_events`: A vector of tuples representing Pink events.
       - `ink_events`: A vector of tuples representing Ink events.
       - `instantiated`: A vector of tuples representing instantiated contracts.

       The `ExecSideEffects` enum is used to represent events emitted by contracts which can potentially lead to further actions by the runtime. For example, a contract may emit an event that triggers the creation of a new contract or the transfer of funds.

       The `into_query_only_effects` method filters and retains only those events which are permissible in a query context. The `is_empty` method returns true if there are no side effects inside.

12. **pink/chain-extension/src/lib.rs:** This contract defines several traits, structs, and functions that are used by the Pink runtime.

    1. **Traits:**

       - `PinkRuntimeEnv`: A trait that represents the runtime environment.
       - `PinkExtBackend`: A trait that represents the backend for the Pink chain extension.

    2. **Structs:**

       - `DefaultPinkExtension`: A struct that implements the `PinkExtBackend` trait.
       - `LimitedWriter`: A struct that implements the `std::io::Write` trait and limits the amount of data that can be written to it.

    3. **Functions:**

       - `batch_http_request`: A function that sends a batch of HTTP requests.
       - `http_request`: A function that sends an HTTP request.

- `async_http_request`: An asynchronous function that sends an HTTP request.
- `sign`: A function that signs a message.
- `verify`: A function that verifies a signature.
- `derive_sr25519_key`: A function that derives a Sr25519 key from a salt.
- `get_public_key`: A function that gets the public key from a private key.
- `cache_set`: A function that sets a value in the cache.
- `cache_set_expiration`: A function that sets the expiration time for a value in the cache.
- `cache_get`: A function that gets a value from the cache.
- `cache_remove`: A function that removes a value from the cache.
- `log`: A function that logs a message.
- `getrandom`: A function that generates random bytes.
- `is_in_transaction`: A function that checks if the runtime is in a transaction.
- `ecdsa_sign_prehashed`: A function that signs a prehashed message using ECDSA.
- `ecdsa_verify_prehashed`: A function that verifies a prehashed signature using ECDSA.
- `system_contract_id`: A function that returns the system contract ID.
- `balance_of`: A function that returns the balance of an account.
- `untrusted_millis_since_unix_epoch`: A function that returns the current time in milliseconds since the Unix epoch.
- `worker_pubkey`: A function that returns the worker's public key.
- `code_exists`: A function that checks if a code hash exists.
- `import_latest_system_code`: A function that imports the latest system code.
- `runtime_version`: A function that returns the runtime version.
- `current_event_chain_head`: A function that returns the current event chain head.
- `js_eval`: A function that evaluates JavaScript code.
- `worker_sgx_quote`: A function that returns the worker's SGX quote.

4. **Usage:** This contract is used by the Pink runtime to interact with the outside world. For example, the `http_request` function can be used to send HTTP requests to external APIs. The `sign` and `verify` functions can be used to sign and verify messages. The `cache_set` and `cache_get` functions can be used to store and retrieve data from the cache. The `log` function can be used to log messages.

13. **pink/chain-extension/src/local_cache.rs:** The contract is a local cache for contracts to perform `off-chain` computations. It provides methods for setting, getting, and removing key-value pairs, as well as setting expiration times for cached values.

    1. **Use of a global cache:** The cache is implemented as a global variable, which ensures that all contracts have access to the same data. This is important for ensuring that data is consistent across all contracts.
    2. **Use of a storage quota:** The cache has a maximum size, which is set by the contract's owner. This helps to prevent the cache from growing too large and consuming too much memory.
    3. **Use of a garbage collection mechanism:** The cache uses a garbage collection mechanism to remove expired values. This helps to keep the cache size under control and prevents old data from accumulating.
    4. **Use of a thread-safe implementation:** The cache is implemented using a thread-safe data structure, which ensures that multiple contracts can access the cache concurrently without causing data corruption.

Supported Chains

Phat Contract can technically connect to any blockchain for reading and writing operations, as it can effortlessly read from arbitrary blockchain nodes and trigger signed transactions via RPC calls. However, in practice, supporting a specific blockchain requires the corresponding Phat Contract RPC client, serialization library, and signing library to facilitate read and write operations. Presently, EVM and Substrate blockchains have more extensive library support.

1. **EVM blockchains:** At the Native Phat Contract level, it is possible to interact with any EVM-compatible blockchains through their RPC nodes, including:
    - **Ethereum**
    - **Polygon**
    - **Arbitrum**
    - **BSC**
    - **Optimism**
    - any other EVM-compatible blockchains
2. **Substrate blockchains:** Native Phat Contract fully supports Substrate-based blockchains, including:
    - **Polkadot**
    - **Kusama**
    - **Phala Network**
    - **Astar**

Features

Note: to view the provided image, please see the original submission here.

By combining on-chain verification with off-chain capabilities, Phat Contracts bring a plethora of features to decentralized applications.

1. **Connect your smart contract anywhere:** Universal compatibility across EVM and Substrate Blockchains means you can easily connect Phat Contracts to any blockchain without the need for a bridge, expanding your Smart Contract's capabilities.
2. **Gain access to the internet:** Send HTTP/HTTPS requests directly from your Smart Contracts enabling seamless integration with any Web2 APIs, unlocking a world of possibilities for your dApps.
3. **Run arbitrarily Complex Logic:** Execute intense off-chain computations in real-time while bypassing transaction fees and network latency, enhancing your dApps' functionality and user experience at minimum cost.
4. **Computation is always verifiable:** Complex computation on Phala Network is provided by a Decentralized Network: Secure, Robust, and Trustworthy Infrastructure.

Approach Taken in Evaluating Phat Contract

Accordingly, I analyzed and audited the subject in the following steps:

Core Protocol Contract Overview

I focused on thoroughly understanding the codebase and providing recommendations to improve its functionality. The main goal was to take a close look at the important contracts and how they work together in the Phat Contract.

I start with the following contracts, which play crucial roles in the Phat Contract:

```
runtime/src/runtime.rs
runtime/src/contract.rs
runtime/src/storage/mod.rs
runtime/src/capi/ecall_impl.rs
runtime/src/storage/external_backend.rs
pink/capi/src/types.rs
pink/chain-extension/src/lib.rs
pink/chain-extension/src/local_cache.rs
```

I started my analysis by examining the intricate structure and functionalities of the Phat Contract protocol, focusing on the custom runtime called `PinkRuntime` that includes several pallets such as `frame_system`, `pallet_timestamp`, `pallet_balances`, `pallet_insecure_randomness_collective_flip`, `pallet_contracts`, and `pallet_pink`.

The code imports necessary modules and defines types for common Substrate components like `AccountId`, `Balance`, `BlockNumber`, `Hash`, and `Nonce`. The `construct_runtime!` macro is used to define the `PinkRuntime` structure, which includes several pallets that provide different functionalities to the runtime. The `parameter_types!` macro is used to define constants for the runtime, which are used throughout the runtime and its pallets. The code implements the `Config` trait for `PinkRuntime` for different pallets, defining the configuration for each pallet, including types for various components and some constant values. The `pallet_contracts::Config` implementation includes a `Migrations` type that defines the migrations for the contracts pallet, which are used to handle changes in the contracts pallet's storage between different versions.

The `on_genesis` and `on_runtime_upgrade` functions are defined to handle the initialization and upgrade of the runtime, respectively. The `check_metadata` function is used to check if the runtime's metadata has changed, and if so, it writes the new metadata to a file and returns an error.

Additionally, the `PinkRuntime` defines various parameters for the runtime, including `BlockHashCount`, `RuntimeBlockWeights`, `ExistentialDeposit`, `MaxLocks`, `MaxReserves`, `MaxHolds`, `DefaultDepositLimit`, `MaxCodeLen`, and `MaxStorageKeyLen`. These parameters are used to configure the runtime and set limits on various aspects of the system.

The `runtime/src/contract.rs` file provides functions for instantiating and executing contracts in a blockchain runtime. It includes functions for masking certain values to make them "coarse-grained," which can be useful for privacy or performance reasons. The code uses Substrate's `pallet_contracts` library to handle contract-related operations.

The `runtime/src/storage/mod.rs` file defines a `Storage` trait and its implementation for managing storage in the Pink runtime environment. It provides methods for executing runtime code using the `Storage` as a backend and committing changes to the underlying storage backend. The `Storage` trait defines methods for committing transactions and executing runtime code, while the `Storage` implementation uses a provided storage backend to implement these methods.

The `execute_with` method executes runtime code using the `Storage` as a backend and returns the result, side effects, and committed changes. The `execute_mut` method is similar to `execute_with`, but it commits the storage changes to the backend. The `changes_transaction` method converts overlay changes into a backend transaction, while the `commit_changes` method commits storage changes to the backend. The `get` method retrieves the storage value for a specified key.

Documentation Review

Then reviewed the documentation, for a more detailed and technical explanation of Phat contract.

Compiling code and running provided tests

```
git clone https://github.com/code-423n4/2024-03-phala-network
cd 2024-03-phala-network/phala-blockchain/crates/pink/runtime # for test only
cargo test # if not already installed
cargo install cargo-llvm-cov # for coverage report
```

```
./cov.sh
```

Manual Code Review

In this phase, I initially conducted a line-by-line analysis, following that, I engaged in a comparison mode.

- **Line by Line Analysis**: Pay close attention to the contract's intended functionality and compare it with its actual behavior on a line-by-line basis.
- **Comparison Mode**: Compare the implementation of each function with established standards or existing implementations, focusing on the function names to identify any deviations.

Architecture

System Architecture

Note: to view the provided image, please see the original submission here.

The system architecture comprises two main components: the chain and the worker. Inside the worker, Pink Runtime is operational. Messages from end-users reach Pink Runtime via one of two routes: on-chain transactions or off-chain RPC queries.

The following diagram illustrates the typical flow of a query:

Note: to view the provided image, please see the original submission here.

Codebase Quality

Overall, I consider the quality of the Phala Network protocol codebase to be good. The code appears to be mature and well-developed. We have noticed the implementation of various standards adhere to appropriately. Details are explained below:

Codebase Quality Categories	Comments
Architecture & Design	The protocol features a modular design, segregating functionality into distinct contracts (e.g., `runtime`, `capi`, `chain-extension`) for clarity and ease of maintenance.
Error Handling & Input Validation	In the `Phat contract` codebase, error handling and input validation are implemented using different techniques. For example, error handling is implemented using the `Result` type, which represents either a successful result or an error. The `Result` type is used throughout the codebase to handle errors that may occur during the execution of a function. Input validation is implemented using various techniques such as pattern matching, conditional statements, and custom validation functions. For instance, in the `instantiate` function, input validation is performed by checking whether the provided `code_hash`, `input_data`, `salt`, and `mode` arguments meet certain criteria. If any of the arguments are invalid, an error is returned using the `Result` type. Similarly, in the `bare_call` function, input validation is performed by checking whether the provided `address`, `input_data`, `mode`, and `tx_args` arguments meet certain criteria. If any of the arguments are invalid, an error is returned using the `Result` type.
Code Maintainability and Reliability	The contracts are written with emphasis on sustainability and simplicity. The functions are single-purpose with little branching and low cyclomatic complexity.
Code Comments	There must be different types of comments used in the Phat contract protocol: `Single-line comments`, `Multi-line comments`, `Documentation comments` for documentation, explanations, TODOs, and structuring the code for readability. However, the codebase lacks many comments. `NatSpec` tags also allow for automatically generating documentation. The contracts must be accompanied by comprehensive comments to facilitate an understanding of the functional logic and critical operations within the code. Functions must be described purposefully, and complex sections should be elucidated with comments to guide readers through the logic. Despite this, certain areas, particularly those involving intricate mechanics, could benefit from even more detailed commentary to ensure clarity and ease of understanding for developers new to the project or those auditing the code.
Testing	The contracts exhibit a commendable level of test coverage `90%` but with aim to 100% for indicative of a robust testing regime. This coverage ensures that a wide array of `functionalities` and `edge cases` are tested, contributing to the reliability and security of the code. However, to further enhance the testing framework, the incorporation of fuzz testing and invariant testing is recommended. These testing methodologies can uncover deeper, systemic issues by simulating extreme conditions and verifying the invariants of the contract logic, thereby fortifying the codebase against unforeseen vulnerabilities.
Code Structure and Formatting	The codebase benefits from a consistent structure and formatting, adhering to the stylistic conventions and best practices of Rust programming. Logical grouping of functions and adherence to naming conventions contribute significantly to the readability and navigability of the code. While the current structure supports clarity, further modularization and separation of concerns could be achieved by breaking down complex contracts into smaller, more focused components. This approach would not only simplify individual contract logic but also facilitate easier updates and maintenance.
Strengths	The codebase is a well-structured, modular, and extensible implementation of a custom blockchain runtime using the Substrate framework, providing functionality for contract execution, storage management, chain extensions, and more, with a focus on security, performance, and universal compatibility across different blockchain networks.
Documentation	While the `Documentation` provides `comprehensive` details for all functions and the system. However, currently `no helpful inline comments` available for `auditors` or developers. It is crucial to develop inline comments to offer a comprehensive understanding of the contract's functionality, purpose, and interaction methods inside the codebase.

Systemic Risks, Centralization Risks, Technical Risks & Integration Risks

crates/pink/runtime/src/runtime.rs

1. Systemic Risks In `runtime.rs:

- `Weight and Fee Calculation`: The contract uses the Pink pallet for weight and fee calculation `type WeightPrice = Pink;`. If the weight and fee calculation is not

accurate, it could lead to issues such as denial of service attacks.

- `Custom Address Generation`: The contract uses the `Pink` pallet for address generation `type AddressGenerator = Pink;`. If there's an issue with the address generation process, it could lead to problems such as address collisions or incorrect address generation.
- `Migration Risks`: The contract includes several migrations (`NoopMigration<10>`, `v11::Migration<Self>`, `v12::Migration<Self, Balances>`, `v13::Migration<Self>`, `v14::Migration<Self, Balances>`, `v15::Migration<Self>`). If these migrations are not properly handled or tested, they could lead to data loss or other issues during contract upgrades or runtime updates.

```
    type Migrations = (
    // Our on-chain runtime was started from polkadot-v0.9.41 but it already contains
    // the changes handled by the v10::Migration. So we just use a NoopMigration here.
    NoopMigration<10>,
    v11::Migration<Self>,
    v12::Migration<Self, Balances>,
    v13::Migration<Self>,
    v14::Migration<Self, Balances>,
    v15::Migration<Self>,
);
```

- `Limited Call Stack`: The contract has a limited call stack of 5 frames (`type CallStack = [Frame<Self>; 5];`). If the call stack is not sufficient for the contract's execution, it could lead to issues such as transaction failures or contract freezing.
- `Unbounded Vector in System Events`: The contract uses a vector to store system events (`pub type SystemEvents = Vec<frame_system::EventRecord<RuntimeEvent, Hash>>;`). If this vector is not properly managed, it could lead to issues such as memory exhaustion or DoS attacks.

2. **Centralization Risks In `runtime.rs`:**

- The `pallet_insecure_randomness_collective_flip` is used as the randomness source. This pallet provides a simple way to generate randomness, but it's not secure and can be manipulated by the block author. This could lead to centralization risks if the block author has malicious intentions.

3. **Technical Risks In `runtime.rs`:**

- `Insecure Randomness`: Usage of `pallet_insecure_randomness_collective_flip` for randomness generation may introduce technical risks related to `predictability` and security of random values, especially in contexts where strong randomness is crucial (e.g., for cryptographic operations).
- `Hardcoded values`: The contract code includes several hardcoded values, such as `MAX_CODE_LEN`, `MaxStorageKeyLen`, and others. These hardcoded values may need to be reviewed and adjusted based on the specific requirements and security considerations of the system.
- The contract has a maximum code length of `2 * 1024 * 1024` bytes. If the contract code becomes too complex, it could lead to issues with code maintainability.

4. **Integration Risks In `runtime.rs`:**

- The contract uses several migrations (`v11`, `v12`, `v13`, `v14`, `v15`, `NoopMigration<10>`). If these migrations are not properly handled, it could lead to data loss or other issues.
- `Metadata Consistency`: Tests checking metadata consistency (`check_metadata` functions) are present, indicating potential integration risks related to metadata handling or compatibility. Changes in metadata representation or `storage mechanisms` could disrupt interoperability with other components.

crates/pink/runtime/src/contract.rs

1. **Systemic Risks In `contract.rs`:**

- `Gas Calculation`: The contract uses the `mask_gas` function to mask the gas consumption of a contract call. This custom gas calculation could lead to unexpected behavior if not implemented correctly. For example, it could lead to an underestimation of the actual gas consumption, which could be exploited by an attacker to perform denial of service attacks.
- `Deposit Calculation`: The contract uses the `mask_deposit` function to mask the storage deposit of a contract call. This custom deposit calculation could lead to incorrect storage deposit calculations if not implemented correctly. For example, it could lead to an underestimation of the actual `storage deposit`, which could result in insufficient storage deposit errors.
- `Instantiation and Call Functions`: The contract uses the `instantiate` and `bare_call` functions to instantiate and call contracts. These custom functions could lead to issues such as `contract instantiation failures`, `incorrect contract calls`, or unexpected contract behavior if not implemented correctly.
- `Gas Payment and Refund Handling`: The contract uses the `contract_tx` function to handle gas payments and refunds for a contract transaction. This handling could lead to risks such as `loss of funds`, incorrect `gas calculations`, or `transaction failures`.

2. **Centralization Risks In `contract.rs`:**

- The contract uses the `AccountId` type for representing accounts. If the account ID generation process is not secure or decentralized, it could lead to centralization risks.

3. **Technical Risks In `contract.rs`:**

- The contract uses the `mask_low_bits64` and `mask_low_bits128` functions to mask the lowest bits of a given number. If these functions are not implemented correctly, it could lead to unexpected behavior.

- The contract uses the `coarse_grained` function to mask the gas consumption and storage deposit of a contract call. If this function is not implemented correctly, it could lead to `incorrect gas calculations` or `storage deposit issues`.
- The contract uses the `instantiate` and `bare_call` functions to instantiate and call contracts. If these functions are not implemented correctly, it could lead to issues such as `contract instantiation failures` or `incorrect contract calls`.

4. Integration Risks In `contract.rs`:

- The contract uses the `contract_tx` function to execute a contract transaction. If this function is not implemented correctly, it could lead to issues such as incorrect gas calculations or transaction failures.
- `Transaction Arguments`: The contract uses the `TransactionArguments` struct to pass transaction arguments to the `instantiate` and `bare_call` functions. This custom struct could lead to integration issues with other `pallets` or the overall Substrate framework if not properly defined or handled. For example, it could lead to `incorrect transaction execution`.
- `Contract Call Handling`: The contract uses the `contract_tx` function to execute a contract transaction. For example, it could lead to `incorrect gas calculations` or `transaction failures`.

crates/pink/runtime/src/storage/mod.rs

1. Systemic Risks In `mod.rs`:

- `Storage Implementation`: The contract uses a custom `Storage` struct to manage storage the implementation could lead to issues such as `incorrect storage management`, `storage corruption`, or `storage key collisions` if not implemented correctly.

2. Centralization Risks In `mod.rs`:

- The contract uses the `ExecContext` struct to provide execution context to the `execute_with` and `execute_mut` functions. If the execution context is not properly defined or handled, it could lead to centralization risks.

3. Technical Risks In `mod.rs`:

- `Context Management`: The contract uses the `ExecContext` struct to provide execution context to the `execute_with` and `execute_mut` functions the issues such as `incorrect execution` context or inconsistent state if not implemented correctly.

4. Integration Risks In `mod.rs`:

- `Not obvious integration risk`

crates/pink/runtime/src/capi/ecall_impl.rs

1. Systemic Risks In `ecall_impl.rs`:

- `Unbounded Memory Cost`: In the `upload_code` function, the contract estimates the `max_wasmi_cost` of the uploaded code and checks it against a maximum limit. However, the estimation is based on the decompressed code size, and the actual memory usage during execution could be higher due to dynamic memory allocation. This could lead to unbounded memory cost and potential `denial-of-service` attacks.

```
fn upload_code(
  &mut self,
  account: AccountId,
  code: Vec<u8>,
  deterministic: bool,
) -> Result<Hash, String> {
  /*
  According to the cost estimation in ink tests: https://github.com/paritytech/substrate/pull/12993/files#diff-70e9723e9db62816e35f6f885b6
  If we set max code len to 2MB, the max memory cost for a single call stack would be calculated as:
  cost = (MaxCodeLen * 4 + MAX_STACK_SIZE + max_heap_size) * max_call_depth
       = (2MB * 4 + 1MB + 4MB) * 6
       = 78MB
  If we allow 8 concurrent calls, the total memory cost would be 78MB * 8 = 624MB.
  */
  let info =
      phala_wasm_checker::wasm_info(&code).map_err(|err| format!("Invalid wasm: {err:?}"))?;
  let max_wasmi_cost = crate::runtime::MaxCodeLen::get() as usize * 4;
  if info.estimate_wasmi_memory_cost() > max_wasmi_cost {
      return Err("DecompressedCodeTooLarge".into());
  }
  crate::runtime::Contracts::bare_upload_code(
      account,
      code,
      None,
      if deterministic {
          Determinism::Enforced
      } else {
          Determinism::Relaxed
      },
  )
  .map(|v| v.code_hash)
  .map_err(|err| format!("{err:?}"))
}
```

- `Key Leakage`: The `get_key` function allows anyone to retrieve the current key used by the contract. If the key is not properly protected or if it is used for other purposes, this could lead to `key leakage`.

```
fn get_key(&self) -> Option<Sr25519SecretKey> {
    PalletPink::key()
}
```

- `Unchecked Code Execution`: The `contract_instantiate` and `contract_call` functions allow users to instantiate and call contracts with `arbitrary input data`, but

there is no check on the code hash or the origin account. This could allow `unchecked code execution` and potential exploits.

- `Unvalidated Input Data`: The `contract_instantiate` and `contract_call` functions take input data as a byte array, but there is no validation or sanitization of the input data. This could allow malformed or malicious input to be passed to the contract, potentially causing unexpected behavior.
- `Unprotected System Contract`: The `system_contract` function allows anyone to retrieve the address of the system contract, but there is no protection or access control for the system contract.

```
    fn system_contract(&self) -> Option<AccountId> {
    PalletPink::system_contract()
}
```

- `Unrestricted SideVM Code Upload`: The `upload_sidevm_code` function allows users to upload arbitrary `sidevm` code, but there is no restriction or validation of the code. This could allow malicious or vulnerable code to be uploaded and executed in the SideVM environment.

```
    fn upload_sidevm_code(&mut self, account: AccountId, code: Vec<u8>) -> Result<Hash, String> {
    PalletPink::put_sidevm_code(account, code).map_err(|err| format!("{err:?}"))
}
```

- `Unvalidated Cluster Setup Config`: The `setup` function takes a `ClusterSetupConfig` struct as input, but there is no validation of the input data.

```
    fn setup(&mut self, config: ClusterSetupConfig) -> Result<(), String> {
    on_genesis();
    let ClusterSetupConfig {
          cluster_id,
          owner,
          deposit,
          gas_price,
          deposit_per_item,
          deposit_per_byte,
          treasury_account,
          system_code,
    } = config;
```

2. Centralization Risks In `ecall_impl.rs`:

- The `setup` function sets an owner account which has control over certain aspects of the contract. This could lead to centralization of control if the owner account is controlled by a single entity.
- The `set_key` function allows the owner to set a new key, potentially giving them full control over the contract.

```
  fn set_key(&mut self, key: Sr25519SecretKey) {
   PalletPink::set_key(key);
}
```

3. Technical Risks In `ecall_impl.rs`:

- The `upload_code` function allows users to upload arbitrary WASM code, which could contain vulnerabilities or malicious logic.

```
fn upload_code(
    &mut self,
    account: AccountId,
    code: Vec<u8>,
    deterministic: bool,
) -> Result<Hash, String> {
    /*
    According to the cost estimation in ink tests: https://github.com/paritytech/substrate/pull/12993/files#diff-70e9723e9db62816e35f6f885b
    If we set max code len to 2MB, the max memory cost for a single call stack would be calculated as:
    cost = (MaxCodeLen * 4 + MAX_STACK_SIZE + max_heap_size) * max_call_depth
          = (2MB * 4 + 1MB + 4MB) * 6
          = 78MB
    If we allow 8 concurrent calls, the total memory cost would be 78MB * 8 = 624MB.
    */
    let info =
          phala_wasm_checker::wasm_info(&code).map_err(|err| format!("Invalid wasm: {err:?}"))?;
    let max_wasmi_cost = crate::runtime::MaxCodeLen::get() as usize * 4;
    if info.estimate_wasmi_memory_cost() > max_wasmi_cost {
          return Err("DecompressedCodeTooLarge".into());
    }
    crate::runtime::Contracts::bare_upload_code(
          account,
          code,
          None,
          if deterministic {
             Determinism::Enforced
          } else {
             Determinism::Relaxed
          },
    )
    .map(|v| v.code_hash)
    .map_err(|err| format!("{err:?}"))
}
```

- The `contract_instantiate` and `contract_call` functions allow users to instantiate and call contracts with arbitrary input data, which could lead to unexpected behavior or vulnerabilities.
- The `deposit` function allows users to deposit balance to any account, which could be exploited if there are vulnerabilities in the balance handling logic.

```
    fn deposit(&mut self, who: AccountId, value: Balance) {
    let _ = PalletBalances::deposit_creating(&who, value);
```

```
    }
```

4. Integration Risks In `ecall_impl.rs`:

    - The contract uses the `log` and `tracing` crates for logging, but the exact logging setup and configuration is not provided. If the logging is not set up correctly, it could lead to issues with debugging or monitoring the contract.
    - The contract uses the `phala_wasm_checker` crate to check the uploaded WASM code, but the exact version and configuration of this crate is not provided. If the wasm checker is not set up correctly, it could allow malicious or vulnerable code to be uploaded.

crates/pink/runtime/src/storage/external_backend.rs

1. Systemic Risks In `external_backend.rs`:

    - Dependency on external OCalls (`OCallImpl`) for storage operations (`get`, `commit`) which could lead to potential system-wide issues if the host system has any failures.

2. Centralization Risks In `external_backend.rs`:

    - The use of a single `OCallImpl` instance for all storage operations could lead to a central point of failure or `bottleneck`.

3. Technical Risks In `external_backend.rs`:

    - The `get` function in `TrieBackendStorage` does not use the provided prefix, which could potentially lead to incorrect data retrieval.
    - The `code_exists` function instantiates a new storage instance every time it's called, which could be potentially lead to issues with data consistency.

4. Integration Risks In `external_backend.rs`:

    - The contract heavily relies on the correct `implementation` and integration of the `OCallImpl` any issues with the OCallImpl could lead to integration problems.
    - The contract uses specific hashing functions (`twox_128`) and key generation logic in the helper module, which could lead to integration issues if not correctly aligned.

crates/pink/runtime/src/runtime/pallet_pink.rs

1. Systemic Risks In `pallet_pink.rs`:

    - The contract relies on the `frame_system::Config` trait for some of its types (e.g., `AccountId`, `Hash`), which may lead to unintended behavior if the underlying system's configuration changes.
    - The contract has a centralized `TreasuryAccount` that handles all the `fee payments`, which could lead to a systemic risk if this account is compromised.

2. Centralization Risks In `pallet_pink.rs`:

    - The contract has several centralized components, such as the `ClusterId`, `Key`, `TreasuryAccount`, and `SystemContract`, which could lead to centralization risks.
    - The `put_sidevm_code` function allows only the contract owner to upload new `WasmCode`, which could limit the decentralization of the system.
    - The contract has functions to set the `cluster ID`, `key`, `system contract`, `GasPrice`, `DepositPerItem`, `DepositPerByte`, and `treasury account`. If these functions are not properly protected, they could be misused by malicious actors to gain control over the contract or manipulate its behavior.

3. Technical Risks In `pallet_pink.rs`:

    - The contract uses the `Twox64Concat` hashing algorithm for the `SidevmCodes` storage, which may not be the most secure or efficient hashing algorithm available.
    - The contract does not have any error handling or fallback mechanisms for the case when the `TreasuryAccount` is not set, which could lead to runtime errors.
    - Storage leaks - Storing raw code blobs and secrets like private keys in plain storage could leak sensitive data.
    - Missing input validation - Functions like `set_cluster_id` don't validate inputs, allowing malicious values to be set.

```
        pub fn set_cluster_id(cluster_id: Hash) {
            <ClusterId<T>>::put(cluster_id);
    }
```

4. Integration Risks In `pallet_pink.rs`:

    - The contract is tightly coupled with the `pallet-contracts` module, which could make it difficult to integrate with other systems or frameworks.
    - The contract uses several custom types, such as `WasmCode`, `Sr25519SecretKey`, and `Hash`, which could make it challenging to integrate with other systems that may not use the same types.
    - The contract uses the `phala_types::contract::contract_id_preimage` function to generate a contract address. If this function is not properly implemented or has vulnerabilities, it could lead to issues with contract deployment or address generation.

```
            let buf = phala_types::contract::contract_id_preimage(
                deploying_address.as_ref(),
                code_hash.as_ref(),
                cluster_id.as_ref(),
                salt,
            );
```

crates/pink/runtime/src/capi/mod.rs

1. Systemic Risks:

   - The `__pink_runtime_init` function takes a `config` pointer as an argument and initializes the runtime based on this configuration. If the configuration is incorrect, incomplete, or maliciously crafted, it could lead to unpredictable behavior. Additionally, the contract does not perform any validation or error handling on the `config` argument, which could exacerbate this risk.
   - The contract does not seem to have a consistent error handling mechanism. In the `__pink_runtime_init` function, when an error occurs during the `initialization` of the `OCalls function pointer`, the contract logs an error message and `returns -1`. However, it is unclear how this error is propagated or handled by the calling code.

2. Centralization Risks:

   - The contract relies heavily on a centralized component (`pink_capi::v1`) for its core functionality. If this component experiences downtime or malfunctions, it could result in a single point of failure, impacting the entire contract's operation.
   - The `__pink_runtime_init` function initializes the runtime and fills the ecalls table. If there is a single point of failure in the initialization process, it could lead to centralization risks.

3. Technical Risks:

   - The contract relies on external `FFI`, which could introduce risks such as `incorrect memory management` or `ABI` mismatches.
   - The contract uses raw pointers and `manual memory management`, which could lead to memory leaks, `use-after-free`, or double-free vulnerabilities.
   - `Unsafe Function Usage`: Several functions in the contract, such as `__pink_runtime_init` and `ecall`, are marked as unsafe.Improper usage of these functions or incorrect manipulation of raw pointers could introduce technical risks such as memory safety issues.
   - `Raw Pointer Interaction`: The `ecall` function involves raw pointer interaction, which poses technical risks related to memory safety and proper handling of pointers. Improper dereferencing or manipulation of pointers could lead to memory corruption.

4. Integration Risks:

   - The contract assumes that the input data is `SCALE` encoded, but it does not perform any validation or error handling for incorrectly encoded data, which could lead to integration issues.
   - The contract does not perform any `input validation` for the `call_id`, `data`, `len`, `ctx`, and `output_fn` arguments, which could lead to integration issues if these arguments are incorrect.
   - `Dynamic Library Initialization`: The initialization process includes logic for initializing a dynamic logger (`logger::init_subscriber`) based on certain conditions, if the dynamic library initialization process encounters errors or inconsistencies.

crates/pink/runtime/src/capi/ocall_impl.rs

1. Systemic Risks:

   - `Uninitialized Function Pointer`: The `OCALL` function pointer is initially set to the `_default_ocall` function, which panics if called. If the `set_ocall_fn` function is not called or if it fails to set the `OCALL` function pointer, any subsequent cross-contract calls could panic and cause undefined behavior.

```
static mut OCALL: InnerType<cross_call_fn_t> = _default_ocall;
```

   - `Unchecked Function Call`: The `cross_call` function calls the `OCALL` function pointer without any checks or validation. If the `OCALL` function pointer is set to an invalid or malicious function, it could lead to arbitrary code execution or security breaches.

```
    fn cross_call(&self, id: u32, data: &[u8]) -> Vec<u8> {
        unsafe extern "C" fn output_fn(ctx: *mut ::core::ffi::c_void, data: *const u8, len: usize) {
            let output = &mut *(ctx as *mut Vec<u8>);
            output.extend_from_slice(std::slice::from_raw_parts(data, len));
        }
        unsafe {
            let mut output = Vec::new();
            let ctx = &mut output as *mut _ as *mut ::core::ffi::c_void;
            OCALL(id, data.as_ptr(), data.len(), ctx, Some(output_fn));
            output
        }
    }
}
```

   - `Memory Allocation Failures`: The `allocator` module uses the global allocator provided by the runtime environment, but if the allocator functions are not set correctly or if the memory allocation fails, it could lead to allocation failures and memory leaks.
   - `Unvalidated Layout`: The `allocator` module uses the `Layout` struct provided by the `Rust standard library` to `allocate` and `deallocate` memory, but it does not validate the `size` and `align` fields of the `Layout` struct. If the `size` and `align` fields are invalid or malicious, it could lead to memory allocation failures and memory corruption.
   - The purpose and usage of the `call_id` parameter in the `cross_call` function is not clear.

2. Centralization Risks:

- `The ocall_impl contract does not have any inherent centralization risks as it does not have any owner or control mechanisms.`

3. Technical Risks:

- `Unsafe Code`: The contract uses unsafe code to set the `OCALL` function pointer and to call the output function. If the unsafe code is not handled correctly, it could lead to undefined behavior or memory safety issues.

4. Integration Risks:

- Compatibility Issues: The contract uses the `pink_capi` library to handle `cross-contract` calls and `memory allocation`, but if the library version or configuration is not compatible with the contract or the runtime environment, it could lead to compatibility issues and errors.

crates/pink/runtime/src/runtime/extension.rs

1. Systemic Risks:

- `js_eval`: The contract allows `js_eval` function there is a risk of allowing arbitrary code execution.

```
Warning: Executing JavaScript from a string is an enormous security risk. It is far too easy for a bad actor to run arbitrary code whe
```

MDN Web Docs

```
fn js_eval(&self, codes: Vec<JsCode>, args: Vec<String>) -> Result<JsValue, Self::Error> {
   Ok(OCallImpl.js_eval(self.address.clone(), codes, args))
}

fn js_eval(&self, _code: Vec<JsCode>, _args: Vec<String>) -> Result<JsValue, Self::Error> {
   Ok(JsValue::Exception(
         "Js evaluation is not supported in transaction".into(),
   ))
}
```

- `Mode Risks`: The contract uses a `mode` for operations there is a risk of allowing unauthorized operations in the wrong mode.

```
      let mode = OCallImpl.exec_context().mode;
```

- `Instantiation Risks`: The contract handles contract instantiation (`instantiated` variable), which could lead to instantiated allowing unauthorized contract creation.

```
            ContractEvent::Instantiated {
               deployer,
               contract: address,
            } => instantiated.push((deployer.clone(), address.clone())),
            ContractEvent::ContractEmitted {
               contract: address,
```

2. Centralization Risks:

- The contract has a system contract which has special privileges, such as the ability to `sign` and `verify` signatures. This could lead to centralization of control.

```
fn verify(
&self,
sigtype: SigType,
pubkey: Cow<[u8]>,
message: Cow<[u8]>,
signature: Cow<[u8]>,
) -> Result<bool, Self::Error> {
DefaultPinkExtension::new(self).verify(sigtype, pubkey, message, signature)
}
```

- The contract uses a single source for randomness `getrandom` function, which could be manipulated if the source is centralized or predictable.

3. Technical Risks:

- The contract uses a custom caching mechanism (`cache_set`, `cache_get`, `cache_remove` functions), which could lead to data inconsistencies if not implemented correctly.

4. Integration Risks:

- `code_exists` Function Risk: The contract relies on the existence of certain code hashes (`code_exists` function), which could lead to issues if the code is not available or the hash is incorrect.

```
fn code_exists(&self, code_hash: Hash, sidevm: bool) -> Result<bool, Self::Error> {
   if sidevm {
         Ok(PalletPink::sidevm_code_exists(&code_hash.into()))
   } else {
         Ok(crate::storage::external_backend::code_exists(
            &code_hash.into(),
         ))
   }
}
```

- The contract interacts with other pallets (`Balances`, `Contracts`, `System`) and external services `HttpRequest`, `HttpResponse` which could lead to integration issues if these components are not compatible or if their interfaces change.

- The contract uses a custom event handling mechanism `deposit_pink_event` function, which could lead to integration issues if not compatible with the event dispatching mechanism of the runtime.

```
    fn deposit_pink_event(contract: AccountId, event: PinkEvent) {
     let topics = [pink::PinkEvent::event_topic().into()];
     let event = super::RuntimeEvent::Contracts(pallet_contracts::Event::ContractEmitted {
        contract,
        data: event.encode(),
     });
     super::System::deposit_event_indexed(&topics[..], event);
    }
```

crates/pink/capi/src/v1/mod.rs

1. Systemic Risks:

- `Dependence on external services`: The contract relies on external services such as HTTP requests and JavaScript evaluation, which could lead to unpredictable behavior if these services are unavailable, altered, or compromised.
- `Unvalidated HTTP Requests`: The `http_request` and `batch_http_request` functions in the `OCalls` trait allow for making HTTP requests on behalf of a contract. However, there is no validation performed on the request parameters, which could potentially lead to unintended or malicious HTTP requests being made.

```
    fn http_request(
       &self,
       contract: AccountId,
       request: HttpRequest,
    ) -> Result<HttpResponse, HttpRequestError>;

    /// Performs a batch of HTTP(S) requests on behalf of the contract within a specified timeout period.
    /// Returns the collective results of all HTTP requests.
    #[xcall(id = 15)]
    fn batch_http_request(
       &self,
       contract: AccountId,
       requests: Vec<HttpRequest>,
       timeout_ms: u64,
    ) -> BatchHttpResult;
```

2. Centralization Risks:

- `Centralization of control`: The contract has a system contract and a `cluster owner`, which could lead to centralization of control.
- `Single point of failure`: The `cluster setup` is controlled by a single owner, which could lead to centralization of control and a single point of failure. `Cluster Setup`: The `ECalls` trait defines a method setup for initializing the cluster with a provided configuration. Depending on how this setup process is designed and executed, there could be centralization risks, particularly if certain nodes or entities have disproportionate control or influence over the cluster.

3. Technical Risks:

- `Code execution`: The contract allows for code execution, such as contract calls and instantiation, which could lead to security vulnerabilities if not properly sandboxed.
- `Cross-Call Interface`: The entire contract relies on `cross-call` interfaces defined by traits like `CrossCall`, `CrossCallMut`, `ECall`, and `OCall`. Technical risks may arise if these interfaces are not implemented correctly or if there are vulnerabilities in the `cross-call` mechanism, leading to potential exploits or contract failures.
- `Storage Operations`: Methods like `storage_get`, `storage_commit`, and others defined in the `OCalls` trait involve interactions with storage. Technical risks may arise from improper handling of storage operations, such as data corruption, unauthorized access, or inefficient storage usage.

4. Integration Risks:

- Compatibility issues: The contract relies on the `Pink runtime` and interacts with various components, such as the `host`, `storage`, `cache`, and `system` contract, which could lead to compatibility issues and potential bugs.

crates/pink/capi/src/types.rs

1. Systemic Risks:

- `Unhandled Execution Modes`: The contract defines an `ExecutionMode` enum with three modes - `Query`, `Estimating`, and `Transaction`. If the contract is executed in the wrong mode, it could lead to unexpected behavior. For example, if a transaction is executed in `Query` mode, any state changes made by the transaction will be discarded.

```
    pub enum ExecutionMode {
    /// In this mode, the runtime is executing an RPC query. Any state changes are discarded
    /// after execution. Indeterministic operations like HTTP requests are allowed in this mode.
    Query,
    /// In this mode, the runtime is simulating a transaction but state changes are discarded.
    #[default]
    Estimating,
    /// In this mode, the runtime is executing a real transaction. State changes will be committed.
    /// Indeterministic operations like HTTP requests aren't allowed in this mode.
    Transaction,
    }
```

- `Unfiltered Side Effects`: The `ExecSideEffects` enum defines various side effects that can be emitted by contracts. If these side effects are not properly filtered or handled, they could lead to unintended consequences. For example, the `ink_events` and `instantiated` fields are not allowed in a query context, but the contract does not seem to have any checks to prevent them from being included.

```
pub enum ExecSideEffects {
    V1 {
        pink_events: Vec<(AccountId, PinkEvent)>,
        ink_events: Vec<(AccountId, Vec<Hash>, Vec<u8>)>,
        instantiated: Vec<(AccountId, AccountId)>,
    },
}
```

- `Misuse of Deterministic Execution`: The contract has a `deterministic_required` method that returns whether the execution mode requires deterministic execution. If a `non-deterministic` operation is executed in a mode that requires deterministic execution, it could lead to different results each time the contract is executed.

```
    pub fn deterministic_required(&self) -> bool {
        match self {
            ExecutionMode::Query => false,
            ExecutionMode::Estimating => true,
            ExecutionMode::Transaction => true,
        }
    }
```

2. Centralization Risks:

- No obvious Centralization risks.

3. Technical Risks:

- The contract uses the `Blake2b256` hashing algorithm, which is generally secure but may be vulnerable to certain attacks if not implemented correctly.

4. Integration Risks

- No obvious Integration risks.

crates/pink/chain-extension/src/lib.rs

1. Systemic Risks:

- The contract uses the `LimitedWriter` struct to limit the size of `HttpResponse` bodies. If the response body is larger than the limit, an error is returned. This could potentially lead to denial-of-service attacks if an attacker can send large responses to the contract.

```
struct LimitedWriter<W> {
    writer: W,
    written: usize,
    limit: usize,
}
```

- The contract uses the `HeaderMap`, `HeaderName`, and `HeaderValue` types from the `request` library to handle HTTP headers. This could potentially lead to header injection or header manipulation attacks if the headers are not properly validated.
- The contract uses the `ecdsa_sign_prehashed` and `ecdsa_verify_prehashed` functions to sign and verify prehashed messages using ECDSA. If the hashing function used to prehash the messages is not properly validated, it could be vulnerable to hash collision attacks, where an attacker creates two different messages with the same hash.

```
    fn ecdsa_sign_prehashed(
        &self,
        key: Cow<[u8]>,
        message_hash: Hash,
    ) -> Result<EcdsaSignature, Self::Error> {
        let pair = sp_core::ecdsa::Pair::from_seed_slice(&key).or(Err("Invalid key"))?;
        let signature = pair.sign_prehashed(&message_hash);
        Ok(signature.0)
    }
```

2. Centralization Risks:

- The contract does not seem to have any mechanisms to prevent centralization of control. For example, the `cache_set` and `cache_set_expiration` methods allow arbitrary key-value pairs to be stored and retrieved. If these methods are used to store sensitive information, it could be accessed or manipulated by a malicious actor with access to the cache.

crates/pink/chain-extension/src/local_cache.rs

1. Systemic Risks:

- The contract relies on a global cache, which could lead to `system-wide` issues if the cache fails or becomes unavailable.
- The contract uses a `BTreeMap` for storage, which may not be the most efficient data structure for all use cases, potentially leading to performance issues.

```
struct Storage {
// Sum of the size of all the keys and values.
size: usize,
max_size: usize,
kvs: BTreeMap<Vec<u8>, StorageValue>,
}
```

2. Centralization Risks:

- The use of a single global cache could lead to centralization risks, as it represents a single point of failure. If the cache is compromised or becomes unavailable, the entire system could be affected.

Suggestions

What ideas can be incorporated?

- **Code isolation:** Run smart contracts in isolated environments like dockers/enclaves to contain exploits. Limit syscalls and block sensitive APIs.
- **Consensus integration:** Runtime upgrades and feature flags could integrate with BFT or PoS protocols for coordinated defense.
- **Runtime snapshots:** Periodic checkpoints allow reverting compromised state. Combined with state validation on node joins.
- **Side effect logging:** Track and audit all events/state changes for forensics and attribution in case of exploits.
- **Secure enclaves:** Hardware security modules, Intel SGX can isolate critical components like key management and extensions execution.
- **Fuzz testing:** Automated fuzzing of inputs and edge cases to detect vulnerabilities during development and upgrades.
- **Inactivity purging:** Automatically purge unclaimed assets, expired locks and stale data as per configured retention policy.
- **Memory safety:** Link-time optimization, dynamic checks ensure bounded memory access preventing overflows/underflows.

What's unique?

The unique aspect of this code is that it implements a custom runtime called `PinkRuntime` that includes several pallets such as `frame_system`, `pallet_timestamp`, `pallet_balances`, `pallet_insecure_randomness_collective_flip`, `pallet_contracts`, and `pallet_pink`.

The `pallet_pink` pallet is particularly interesting as it provides functionality specific to the Pink Network, such as managing cluster-wide configuration and resources, including gas price, deposit per byte, deposit per item, treasury account, private key, sidevm codes, system contract address, next event block number, and last event block hash.

The code also implements the `AddressGenerator` trait from the `pallet-contracts` crate, which allows it to generate contract addresses. Additionally, the code uses the `convert` trait to convert between weight and balance. The `pallet_pink` pallet also provides functions for setting and retrieving these values, as well as managing gas payments and refunds. Overall, this code demonstrates how Substrate can be used to build custom runtimes with unique functionality and features.

Issues surfaced from attack ideas

1. **Insecure randomness generation**: The `pallet_insecure_randomness_collective_flip` pallet is used in the `PinkRuntime`, which generates randomness based on block hashes. This could potentially be exploited by attackers who control a significant portion of the network's mining power or can predict block hashes. They could manipulate the randomness generation process to their advantage, leading to issues such as biased randomness or predictable outcomes.
2. **Insufficient access controls**: The `pallet_pink` pallet provides functionality specific to the Pink Network, including managing cluster-wide configuration and resources. If proper access controls are not implemented, attackers could potentially manipulate these resources to their advantage. For example, they could modify the gas price, deposit per byte, or deposit per item to disrupt the network's economic model.
3. **Denial of service attacks**: The `pallet_contracts` pallet is used to manage smart contracts on the network. If an attacker can create a contract that consumes a large amount of resources (e.g., gas, storage), they could potentially launch a denial of service attack against the network, preventing legitimate users from using the system.
4. **Malicious code upload**: An attacker could upload smart contracts containing malicious logic like stealing funds, freezing accounts, or spamming network resources. Proper validation of code is needed.
5. **Injection attacks**: Malformed inputs could enable code injection, privilege escalation etc if not sanitized properly before processing. Strong input sanitization is a must.

Time spent

32 hours

Disclosures

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

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and rust 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.