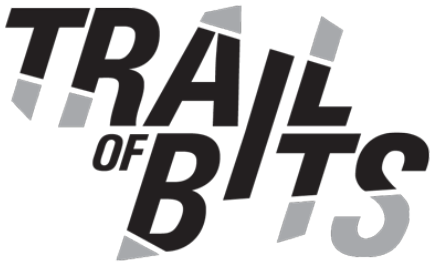

NEAR Protocol

Security Assessment

November 15, 2019

Prepared For:
Illia Polosukhin | *NEAR Protocol*
illia@nearprotocol.com

Prepared By:
Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Alan Cao | *Trail of Bits*
alan.cao@trailofbits.com

Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Executive Summary

From October 21 through November 15, 2019, NEAR Protocol engaged with Trail of Bits to review the security of the `nearcore` runtime. Trail of Bits conducted this assessment over the course of eight person-weeks with two engineers working from the master branch of the repository, syncing on a weekly basis.

During the first week, we focused on familiarizing ourselves with the codebase. We carefully read the three provided sources of documentation (The Beginner's Guide to the NEAR Blockchain, NightShade (was: Whale Fishing): A new NEAR sharding design, and On usability of blockchain applications). We verified that we could build the codebase, and that all unit tests pass. We ran `cargo-audit` over the codebase. We began manually reviewing the runtime components. Finally, we started work on one of the three fuzzers we developed.

During the second week, we read the additional provided documentation (The Nearnomicon). We completed work on the first of our three fuzzers. Based in part on results produced by that fuzzer, we started looking for sources of non-determinism within the codebase. Finally, we started work on the second of three fuzzers, the one that targeted the import functions in `near-vm-logic`.

During the third week, we completed work on the import fuzzer. We diagnosed two panics produced by the import fuzzer. Motivated by concerns over `usize` on a 32-bit platform, we tried to build NEAR on a 32-bit platform. Finally, we continued our search for sources of non-determinism.

During the fourth week, we identified what we believe was the main source of non-determinism observed earlier during the assessment. We made improvements to the import fuzzer, in particular, eliminating the use of the "mock" external implementation and replacing it with `RuntimeExt`. Finally, we diagnosed multiple (reproducible) hangs produced by the import fuzzer.

Many of the findings concern data validation (TOB-NEAR-004, TOB-NEAR-007, TOB-NEAR-008, TOB-NEAR-009, and TOB-NEAR-015), which is not surprising for any program that processes untrusted input. Several of the findings concern the possibility of undefined behavior (TOB-NEAR-005, TOB-NEAR-011, TOB-NEAR-012, and TOB-NEAR-013). All of the high-severity findings are related to (the possibility of) denial-of-service attacks (TOB-NEAR-014, TOB-NEAR-016, and TOB-NEAR-017).

The extensive documentation was helpful to us during our review. While NEAR protocol acknowledges less than 100% test coverage, many of the crucial parts appear to be tested. We found these tests useful both for understanding the code and for providing us a

scaffolding on which to build our fuzzers. Finally, as one would expect of a program written in a memory-safe language such as Rust, NEAR does not seem to feature the same "low-hanging fruit" that programs written in less safe languages might.

Most of the data validation and denial-of-service issues found appear to have simple fixes. As we explain in Appendix C, we believe we found the main source of non-determinism observed early in the assessment. However, given the particular concern that non-determinism presents to blockchain projects, there are additional steps that NEAR could take to mitigate this concern, e.g., adding a WASM runtime besides wasmer and incorporating differential fuzzing into the continuous integration process. This could help reveal any remaining sources of non-determinism, and also future data validation or denial-of-service related bugs that might arise.

Project Dashboard

Application Summary

Name	nearcore
Version	563598283393c23699a5ae62bad80ce921f95487 502d49fc318c01246f6fb84deed99ead996e526d 5802015e7de0cd6a7a46db9db5493dfe6141aafc 54b597cfc53de24aa7c9813eea87926ad9752aa1
Type	Rust, WASM
Platforms	POSIX

Engagement Summary

Dates	October 21 through November 15, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	8 person-weeks

Vulnerability Summary

Total High-Severity Issues	3	■ ■ ■
Total Medium-Severity Issues	4	■ ■ ■ ■
Total Low-Severity Issues	4	■ ■ ■ ■
Total Informational-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total	18	

Category Breakdown

Access Controls	1	■
Cryptography	1	■
Data Exposure	1	■
Data Validation	5	■ ■ ■ ■ ■
Denial of Service	2	■ ■
Patching	4	■ ■ ■ ■
Undefined Behavior	4	■ ■ ■ ■
Total	18	

Engagement Goals

The engagement was scoped to provide a security assessment of the `nearcore` runtime, and in particular, the following crates:

- `near-runtime-fees`
- `near-vm-errors`
- `near-vm-logic`
- `near-vm-runner`
- `near-vm-runner-standalone`
- `runtime`
- `runtime-params-estimator`

Specifically, we sought to answer the following questions:

- Are any incorrect assumptions made concerning wasmer?
- Are there any problems associated with crossing the host-to-guest machine boundary?
- Does the system error deterministically?
- Is it possible for two nodes to disagree on the outcome of a transaction?
- Are grinding/denial-of-service attacks possible?
- Are any NEAR rules abusable (e.g., unable to achieve their intent)?

Coverage

Each of the above crates was manually reviewed, and scanned for vulnerable dependencies using `cargo-audit`. Also, the crates' unit tests were verified to pass.

In addition, three fuzzers based on American Fuzzy Lop (AFL) were developed.

- One fuzzer was loosely based on the tests in `test_evil_contract.rs` and targeted the runtime as a whole. The fuzzer randomly generated a WASM binary, deployed it, and called one of its functions.
- The second fuzzer was focused on the import functions in `near-vm-logic`. The fuzzer would randomly choose between zero and five (inclusive) import functions and call them with random arguments.
- The last fuzzer was used to test WASM interactions with the codebase, and was loosely based on high-level functionality in the `near-vm-runner` subcrate, and lower-level WASM register and storage interactions with the VM in `near-vm-logic/logic.rs`, where fuzzing was used to guide property testing of the specification in the Nearnomicon.

Finally, other efforts in both manual and fuzz testing were dedicated to ensuring that critical external dependencies used by the runtime did not contain any known issues. Manual review was dedicated to cryptographic APIs and utility crates like `pwasm-utils` in order to find shortcomings in implementation that could lead to undefined behavior in NEAR.

Fuzzing was dedicated to libraries that interacted with WASM, in particular, `wasmparser` and `parity-wasm`. After identifying pre-existing test harnesses in their codebases that implement functionality found in NEAR, we fuzzed them with both AFL and Clang/LLVM's libFuzzer bindings through `cargo-fuzz` in order to identify any edge-case crashes. To validate that these could also be present in NEAR, the tests were modified in order to harness NEAR API calls that utilize the vulnerable functionality, and fuzzed again for the same crashes.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Upgrade spin from 0.5.0 to 0.5.2, and serde_cbor from 0.10.1 to 0.10.2.** (TOB-NEAR-001) This will eliminate a potential memory corruption vulnerability and a stack overflow vulnerability.

❑ **Have all tests bind to localhost instead of 0.0.0.0.** (TOB-NEAR-002) This will eliminate an unnecessary risk posed to NEAR developers who run those tests.

❑ **If tests in test_evil_contracts.rs need not be executed sequentially, then convert them to quickcheck tests.** (TOB-NEAR-003) This will make clear that the properties tested are meant to hold for arbitrary inputs, and will save NEAR developers from having to maintain code to generate inputs.

❑ **Compile and link against a locally patched version of pwasm-utils that fixes the bug described in TOB-NEAR-004.** This will eliminate an integer overflow vulnerability that can cause debug builds of NEAR to panic.

❑ **Consider incorporating code like that shown in Figure 5.2 of TOB-NEAR-005 into your codebase.** This code makes `HashMaps` behave deterministically in debug builds, but retains randomized hashing for release builds. Having repeatable debugging runs will make debugging easier.

❑ **Implement the Drop trait for SecretKey, with an iterator that zeroes out the stored key.** A possible implementation appears in Figure 6.2 of TOB-NEAR-006. This will prevent such keys from residing in RAM longer than they should, thereby reducing their potential exposure.

❑ **Use safe_add_gas for all gas additions, and create and use a similar function for multiplication.** (TOB-NEAR-007) In addition, incorporating sanity checks with `checked_add` and `checked_mul` can be done in instances where standard arithmetic operators are still viable (i.e., for large-width integral primitives like `u128`), but can still potentially cause an

abrupt panic in a debug build as a result of overflow. This will help to ensure that a bug is not introduced if gas costs change in the future.

❑ **Rewrite the code in Figure 8.1 of TOB-NEAR-008 to use checked_mul.** This will eliminate an integer overflow vulnerability that can cause debug builds of NEAR to panic.

❑ **Either change the "mock" storage implementation, or add checks to storage_iter_range to catch the case when both start_key and end_key are outside the range of the BTreeMap and start_key > end_key.** (TOB-NEAR-009) This will eliminate a source of unnecessary panics in debugging runs.

❑ **Document the fact that NEAR does not support 32-bit platforms.** (TOB-NEAR-010) Since NEAR cannot currently be tested on such platforms, its use should be discouraged on such platforms.

❑ **As new code is added to the NEAR runtime, avoid using features specific to wasmer.** (TOB-NEAR-011) This will make it easier to transition to another WASM runtime in the future.

❑ **Use idiomatic functions to confine mutation inside a limited scope, e.g., as shown in Figure 12.3.** (TOB-NEAR-012) For the mutable result described in the finding, shrink the domain for intermediate functions so they explicitly state which data is exactly required for computation and use immutable intermediate smaller results to build the final result. Adopting such an approach will reduce the likelihood of mutable state being modified accidentally.

❑ **Adjust the current overly general types so they describe the domain as closely as possible.** An example of such a fix is shown in Figure 13.3 of TOB-NEAR-013. This will reduce the likelihood of a developer introducing a panic by calling a function without knowing its preconditions.

❑ **Modify near-vm-runner's Cargo.toml file so that the feature default-backend-singlepass is enabled.** (TOB-NEAR-014) This will make the singlepass wasmer backend the default and eliminate certain potential denial-of-service attacks.

❑ **Compile and link against a locally patched version of parity-wasm that fixes the bug described in TOB-NEAR-015.** This will eliminate a deserialization bug in `parity-wasm` that can cause NEAR to panic.

❑ **Within promise_batch_action_add_key_with_function_call and storage_iter_range, move the gas deductions so they occur before the expensive allocations and reads.** (TOB-NEAR-016) This will eliminate a potential denial-of-service vector where an attacker calls one of those functions without sufficient gas to pay for the expensive operations.

❑ **Make the gas cost of promise_batch_then proportional to the product of the number of receipts and the size of the account ID.** (TOB-NEAR-017) Similar to TOB-NEAR-016, this will eliminate a potential denial-of-service vector where an attacker calls the function without sufficient gas to pay for the expensive operations.

❑ **Document the fact that FinalExecutionOutcome structures should be used only in contexts where they can be verified.** (TOB-NEAR-018) This will reduce the likelihood of a developer placing undue trust in the correctness of those structures.

Long Term

❑ **Regularly run cargo-audit over your codebase.** (TOB-NEAR-001) Doing so can help reveal vulnerable crates in the future.

❑ **Consider incorporating network monitoring into your development environment (something like Mac OS X's Firewall).** Doing so may uncover bugs similar to that described in TOB-NEAR-002.

❑ **As new tests are added to the codebase, consider whether they could benefit from being property-based tests.** (TOB-NEAR-003) Property-based tests have numerous advantages over conventional unit tests, including making clear that the properties tested are meant to hold for arbitrary inputs, and saving developers from having to maintain code to generate inputs.

❑ **Incorporate fuzzing into your continuous integration process.** Doing so may help to uncover bugs similar to those described in TOB-NEAR-004 and TOB-NEAR-015 in the future.

❑ **Avoid using any source of randomness that cannot be turned off in a debug build.** (TOB-NEAR-005) Use of randomness hinders repeatability, making debugging more difficult.

❑ **Implement a dedicated crate or submodule that ensures that sensitive allocations remain protected at runtime.** (TOB-NEAR-006) Examples of crates that are capable of this include secrets, which protect cryptographic secret allocations by enforcing guard pages, or clear_on_drop, which allows secure allocations on both stack and heap, enabling a developer to heap-allocate sensitive secrets on an `mlock`ed memory page. This ensures that sensitive allocations stored in-memory remain protected and are deallocated properly.

❑ **Add unit tests to verify that gas calculations do not overflow.** (TOB-NEAR-007) This will help to ensure that a bug is not introduced if gas costs change in the future.

❑ **Consider incorporating no-panic into critical code, such as logic.rs.** (TOB-NEAR-008 and TOB-NEAR-009) This crate provides a macro that requires the compiler to prove that a function cannot panic, even through functions that it calls. Use of the macro can help to reveal unknown panics. Judicious use of the macro can eliminate unnecessary panics entirely.

❑ **Investigate ways of preventing NEAR from being built on 32-bit platforms in code.** (TOB-NEAR-010) One possibility may be the use of `target_pointer_width`. Since NEAR cannot currently be tested on such platforms, its use should be discouraged on such platforms.

❑ **Add one or more WASM runtimes to NEAR besides wasmer.** (TOB-NEAR-011) This would facilitate differential fuzzing, thereby increasing confidence in each runtime.

❑ **Employ a critical attitude towards mutable state during code writing and reviewing.** (TOB-NEAR-012) Use mutable state only when absolutely necessary. This will reduce the likelihood of mutable state being modified accidentally.

❑ **Always choose the closest type representation for newly written code, especially when an overly general type would cause the code to panic or behave in an undefined manner.** (TOB-NEAR-013) This will reduce the likelihood of a developer introducing a panic by calling a function without knowing its preconditions.

❑ **Add continuous integration tests to verify that NEAR calls the singlepass backend when a WASM binary is compiled.** (TOB-NEAR-014) If something were to change causing wasmer to revert to the default cranelift compiler, this could help to catch such an error.

❑ **Review all gas deductions. Ensure that the deductions occur before the operations they are meant to pay for.** Doing so could help to reveal bugs similar to that described in TOB-NEAR-016.

❑ **Consider whether the size of an account ID should be limited.** (TOB-NEAR-017) Limiting the size will help reduce the damage if the gas cost of another operation fails to take into account the possibility of large account IDs.

❑ **Consider whether FinalExecutionOutcome structures could be useful in other contexts.** If so, require that they be signed by the node that generates them. This would reduce the incentive for a node to lie about the structure's contents, because a recipient of the structure would have proof of the lie.

Findings Summary

#	Title	Type	Severity
1	Codebase uses two crates with RUSTSEC advisories	Patching	Medium
2	Tests listen on all interfaces unnecessarily	Access Controls	Low
3	Some tests could benefit from becoming property-based tests	Patching	Informational
4	Integer overflow in pwasm-utils	Data Validation	Medium
5	Use of randomness hinders repeatability	Undefined Behavior	Informational
6	Private keys should be explicitly scrubbed before going out of scope	Data Exposure	Medium
7	Potential integer overflows in NEAR runtime	Data Validation	Informational
8	Integer overflow in import function “promise_and”	Data Validation	Low
9	Panic in import function “storage_iter_range”	Data Validation	Low
10	32-bit builds should be explicitly disallowed	Patching	Low
11	NEAR would benefit from additional WASM runtimes	Undefined Behavior	Informational
12	Unnecessary use of mutable state	Undefined Behavior	Informational
13	Imprecise types could cause unnecessary panics	Undefined Behavior	Informational

14	Singlepass wasmer backend is not used	Patching	High
15	Malformed input causes panic in WASM deserialization routine	Data Validation	Medium
16	Gas deductions should precede their associated operations	Denial of Service	High
17	Out-of-memory error caused by “promise_batch_then” and “create_receipt”	Denial of Service	High
18	Structure “FinalExecutionOutcome” could benefit from including a signature	Cryptography	Informational

1. Codebase uses two crates with RUSTSEC advisories

Severity: Medium
Type: Patching
Target: `spin 0.5.0, serde_cbor 0.10.1`

Difficulty: Undetermined
Finding ID: TOB-NEAR-001

Description
The nearcore repository makes use of two crates with Rust Security (RUSTSEC) advisories. Specifically, the crate `spin 0.5.0` contains a vulnerability that could lead to memory corruption, while the crate `serde_cbor 0.10.1` contains a stack overflow vulnerability.

Exploit Scenario
Eve discovers a code path leading to one of the vulnerable crates. Eve uses this code path to crash nodes, corrupt memory, etc.

Recommendation
Short term, upgrade `spin` from `0.5.0` to `0.5.2`, and `serde_cbor` from `0.10.1` to `0.10.2`.

Long term, regularly run `cargo-audit` over your codebase. Doing so can help reveal similar bugs in the future.

References
- RUSTSEC-2019-0013: spin: Wrong memory orderings in RwLock potentially violates mutual exclusion
- RUSTSEC-2019-0025: serde_cbor: Flaw in CBOR deserializer allows stack overflow

2. Tests listen on all interfaces unnecessarily

Severity: Low | Difficulty: High
Type: Access Controls | Finding ID: TOB-NEAR-002
Target: `chain/jsonrpc/src/lib.rs, chain/network/src/peer_manager.rs`

Description
Several of the tests in the runtime directory bind to an address of the form `0.0.0.0:...`, which allows the test to accept a connection on any interface. The specific locations where the bindings occur are given in Figures 2.1 and 2.2. Accepting a connection on any interface is overly broad and creates unnecessary risk.

```
pub fn start_http(
    config: RpcConfig,
    client_addr: Addr<ClientActor>,
    view_client_addr: Addr<ViewClientActor>,
) {
    let RpcConfig { addr, ... } = config;
    HttpServer::new(move || {
        ...
    })
    .bind(addr)
    ...
}
```

Figure 2.1: chain/jsonrpc/src/lib.rs#L322-L350.

```
    fn started(&mut self, ctx: &mut Self::Context) {
        // Start server if address provided.
        if let Some(server_addr) = self.config.addr {
            // TODO: for now crashes if server didn't start.
            let listener =
                TcpListener::bind(&server_addr).unwrap();
            ...
        }
        ...
    }
```

Figure 2.2: chain/network/src/peer_manager.rs#L541-L555.

Exploit Scenario
Alice is a NEAR developer working on the runtime code. Eve gains access to Alice's network and randomly tries to connect to open ports on Alice's machine. Eve's actions cause Alice's tests to fail sporadically. Alice wastes time trying to identify a nonexistent problem within her code.

Recommendation
Short term, have all tests bind to `localhost` instead of `0.0.0.0`.

Long term, consider incorporating network monitoring into your development environment (something like Mac OS X's Firewall). Doing so may uncover similar bugs in the future.

3. Some tests could benefit from becoming property-based tests

Severity: Informational
Type: Patching
Target: `test_evil_contracts.rs`

Difficulty: Not applicable
Finding ID: TOB-NEAR-003

Description
Tests within `test_evil_contracts.rs` loop over multiple values, and verify that a property holds for each of those values. An example appears in Figure 3.1. Such tests could be made into quickcheck (i.e., property-based) tests, like the one shown in Figure 3.2.

```
#[test]
fn test_evil_deep_trie() {
    let node =
        setup_test_contract(include_bytes!("..."));
    (0..50).for_each(|i| {
        println!("insertStrings #{}", i);
        ...
        assert_eq!(res.status, FinalExecutionStatus::
            SuccessValue(to_base64(&[])), "{:?}", res);
    });
    (0..50).rev().for_each(|i| {
        println!("deleteStrings #{}", i);
        ...
        assert_eq!(res.status, FinalExecutionStatus::
            SuccessValue(to_base64(&[])), "{:?}", res);
    });
}
```

Figure 3.1: runtime/runtime/tests/test_evil_contracts.rs#L36-L80.

The benefits of using quickcheck include:

- Use of the `quickcheck!` macro makes clear that the property is meant to hold for each value of the input.
- Developers do not have to maintain code to generate inputs to the properties that are tested.
- If a violation of a property is found, quickcheck's built-in "shrinkers" will try to find a minimal witness to the violation.

Exploit Scenario
Alice, a NEAR developer, makes a change to the codebase without realizing that she has violated an invariant by doing so. Time and effort are wasted debugging the errors that result from her change.

Recommendation

Short term, if tests in `test_evil_contracts.rs` need not be executed sequentially, then convert them to quickcheck tests. Review the tests in other files to see where else quickcheck might be applicable.

Long term, as new tests are added to the codebase, consider whether they could benefit from being property-based tests.

References

- BurntSushi/quickcheck
- Crate quickcheck

```
quickcheck! {
    fn test_evil_deep_trie_insert(i: u64) -> bool {
        let node =
            setup_test_contract(include_bytes!("..."));
        println!("insertStrings #{}", i);
        ...
        res.status == FinalExecutionStatus::
            SuccessValue(to_base64(&[]))
    }
}

quickcheck! {
    fn test_evil_deep_trie_delete(i: u64) -> bool {
        let node =
            setup_test_contract(include_bytes!("..."));
        println!("deleteStrings #{}", i);
        ...
        res.status == FinalExecutionStatus::
            SuccessValue(to_base64(&[]))
    }
}
```

Figure 3.2: The test from Figure 3.1 converted to a quickcheck test.

4. Integer overflow in pwasm-utils

Severity: Medium
Type: Data Validation
Target: `pwasm-utils-0.11.0`

Difficulty: Low
Finding ID: TOB-NEAR-004

Description
The nearcore repository uses `pwasm-utils` to inject calls to a gas counting function into a WASM binary. A function called `update_call_index` increments the index of each call instruction whose index is greater than or equal to the index of the gas counting function. (See Figure 4.1.) However, the function does not check for integer overflow. A WASM file (like the one in Figure 4.2) could be used to exploit this fact and cause a NEAR node built in debug mode to panic.

```
pub fn update_call_index(instructions: &mut elements::Instructions,
            inserted_index: u32) {
      use parity_wasm::elements::Instruction::*;
      for instruction in instructions.elements_mut().iter_mut() {
            if let &mut Call(ref mut call_index) = instruction {
                  if *call_index >= inserted_index { *call_index += 1}
            }
      }
}
```

Figure 4.1: paritytech/wasm-utils/src/gas/mod.rs#L17-L24.

```
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    call 4294967295)
  (export "abort_with_zero" (func 0)))
```

Figure 4.2: A WASM file that causes an integer overflow in `pwasm-utils`.

Exploit Scenario
Eve knows that Alice runs a NEAR node built in debug mode. Eve submits the WASM file in Figure 4.2 to the NEAR blockchain, causing Alice's node to crash.

Recommendation
Short term, compile and link against a locally patched version of `pwasm-utils`.

Long term, incorporate fuzzing into your continuous integration process. Doing so may help to uncover similar bugs in the future.

References
- Replacing dependencies with patch

5. Use of randomness hinders repeatability

Severity: Informational
Type: Undefined Behavior
Target: `nearcore`

Difficulty: Not applicable
Finding ID: TOB-NEAR-005

Description
`HashMap` and `HashSet` are used throughout the nearcore repository and its dependencies. `HashMap`/`HashSet`'s hashing function is normally randomly generated (see the warning in Figure 5.1). While this is useful in release builds, use of randomness can cause a program to behave differently across runs, making debugging more difficult.

> Warning: `hash_builder` is normally randomly generated, and is designed to allow `HashMaps` to be resistant to attacks that cause many collisions and very poor performance. Setting it manually using this function can expose a DoS attack vector.

Figure 5.1: A warning appearing in `HashMap`'s documentation.

Exploit Scenario
Alice, a NEAR developer, suspects that someone is trying to carry out grinding attacks against NEAR nodes. Alice's attempts to reproduce the attacks are frustrated by the use of `HashMap`/`HashSet` and the fact that NEAR nodes behave differently across runs.

Recommendation
Short term, consider incorporating code like that shown in Figure 5.2 into your codebase. This code makes `HashMaps` behave deterministically in debug builds, but retains randomized hashing for release builds.

```
#[cfg(not(debug_assertions))]
type HashMap<K, V> = std::collections::HashMap<K, V>;

#[cfg(debug_assertions)]
type HashMap<K, V> = std::collections::HashMap<
    K,
    V,
    std::hash::BuildHasherDefault<std::collections::hash_map::DefaultHasher>,
>;
```

Figure 5.2: Code to make `HashMaps` behave deterministically in debug builds.

Long term, avoid using any source of randomness that cannot be turned off in a debug build. Use of randomness hinders repeatability, making debugging more difficult.

6. Private keys should be explicitly scrubbed before going out of scope

Severity: Medium
Type: Data Exposure
Target: near-crypto

Difficulty: High
Finding ID: TOB-NEAR-006

Description

`near-crypto` implements `SecretKey`, an enum wrapper around either an ed25519 or secp256k1 private key. While Rust's ownership model guarantees that bindings go out of scope at the end of execution, allocations made may still persist in memory. Therefore, we recommend implementing code to explicitly clear the memory allocations that are made. This process should also ensure the code does not get optimized away, as compilers tend to remove such instructions due to dead store elimination optimizations.

```
#[derive(Clone, Eq, PartialEq, Debug)]
pub enum SecretKey {
    ED25519(sodiumoxide::crypto::sign::ed25519::SecretKey),
    SECP256K1(secp256k1::key::SecretKey),
}
```

Figure 6.1: core/crypto/src/signature.rs#L100-L104.

Exploit Scenario

Alice gains unauthorized remote access to Bob's machine running a NEAR validator node, and figures out its process ID. She can attach a debugger and read from memory the generated private key used to perform signing, and impersonate that node herself with the private key.

Recommendation

Short-term, implement the `Drop` trait for `SecretKey`, with an iterator that zeroes out the stored key. A possible implementation appears in Figure 6.2.

```
impl Drop for SecretKey {
    fn drop(&mut self) {
        match self {
            SecretKey::ED25519(key) => {
                for i in key.as_bytes().iter() {
                    *i = 0
                }
            },
            SecretKey::SECP256K1(key) => {
                // .. similar code here. Note that secp256k1 does its
own `Drop` deallocation.
            }
```

```
        }
    }
}
```

Figure 6.2: Implementation of `SecretKey` *enum wrapper around two curves for DSA.*

Long term, implement a dedicated crate or submodule that ensures that not only are sensitive allocations properly scrubbed and deallocated, but that they remain protected even during runtime. Examples of crates that are capable of this include secrets, which protect cryptographic secret allocations by enforcing guard pages, or clear_on_drop, which allows secure allocations on both stack and heap, enabling a developer to heap-allocate sensitive secrets on an `mlock`ed memory page.

References

- Erasing secrets from memory (zero on drop)

7. Potential integer overflows in NEAR runtime

Severity: Informational
Type: Data Validation
Target: `nearcore/runtime/runtime/src/config.rs`

Difficulty: Undetermined
Finding ID: TOB-NEAR-007

Description
There are several places where potential integer overflows could occur in regard to charge calculations in the NEAR runtime.

The first concerns the function `total_send_fees`, which is used to calculate the charge for a transaction. While the function uses `safe_add_gas` for checked addition, it omits checking in a few spots, allowing gas to overflow when large values are involved. An example appears in Figure 7.1.

```
DeployContract(DeployContractAction { code }) => {
    let num_bytes = code.len() as u64;
    cfg.deploy_contract_cost.send_fee(sender_is_receiver)
     + cfg.deploy_contract_cost_per_byte.send_fee(sender_is_receiver)
* num_bytes
}
```

Figure 7.1: runtime/runtime/src/config.rs#L69-L73.

Currently this attack is not possible due to the fact that configured costs per byte are low, which makes exploitable payload size greatly exceed available operating memory. However, should the configured costs ever rise to large values, the attack becomes possible and the error remains silent on a release build. An example payload of one gigabyte in size requires cost per byte to be greater than 17179869184.

In addition, transactional calculations for storage cost per block with `cost_per_block` are also susceptible to integer overflow, as seen in Figure 7.2.

```
let storage_cost_per_block =
u128::from(total_account_storage(account_id, account))
        * runtime_config.storage_cost_byte_per_block;
```

Figure 7.2: runtime/runtime/src/actions.rs#L48-L49.

Like the previous instance of potential overflow, this attack is not possible since configured costs per bytes are low, and a runtime configuration that utilizes higher costs per byte would not be used in a production mainnet.

Finally, other potential occurrences of integer overflows involving gas reward and refund calculation can be found in the `Runtime` interface, as seen in Figures 7.3 and 7.4, respectively.

```
let gas_reward = result.gas_burnt *
  self.config.transaction_costs.burnt_gas_reward.numerator /
  self.config.transaction_costs.burnt_gas_reward.denominator;

let mut validator_reward = Balance::from(result.gas_burnt) *
  action_receipt.gas_price;

if gas_reward > 0 {
    let mut account = get_account(state_update, account_id)?;
    if let Some(ref mut account) = account {
        let reward = Balance::from(gas_reward) *
          action_receipt.gas_price;
        validator_reward -= reward;
        account.amount += reward;
}
```

Figure 7.3: runtime/runtime/src/lib.rs#L707-L715.

```
let exec_gas = total_exec_fees(&self.config.transaction_costs,
  &action_receipt.actions).expect(OVERFLOW_CHECKED_ERR) +
self.config.transaction_costs.action_receipt_creation_config.exec_fee();

let mut deposit_refund = if result.result.is_err() { total_deposit }
else { 0 };

let gas_refund = if result.result.is_err() {
    prepaid_gas + exec_gas - result.gas_burnt
} else {
    prepaid_gas + exec_gas - result.gas_used
};
```

Figure 7.4: runtime/runtime/src/lib.rs#L600-L614.

Exploit Scenario

With the integer overflow in `total_send_fees`, an attacker sends `DeployContract` actions containing very large payloads. This causes the charge calculation to overflow, and as a result he is charged a negligible amount. The NEAR network is flooded with large payloads, causing a denial of service.

Recommendation

Short term, use `safe_add_gas` for all gas additions, and create and use a similar function for multiplication. In addition, incorporating sanity checks with `checked_add` and

`checked_mul` can be done in instances where standard arithmetic operators are still viable (i.e., for large-width integral primitives like `u128`), but can still potentially cause an abrupt panic in a debug build as a result of overflow.

Long term, add unit tests to verify that gas calculations do not overflow. This will help to ensure that a bug is not introduced if gas costs change in the future.

8. Integer overflow in import function “promise_and”

Severity: Low
Type: Data Validation
Target: `promise_and`

Difficulty: Low
Finding ID: TOB-NEAR-008

Description
The import function `promise_and` performs a multiplication using one of its arguments. (See Figure 8.1.) By setting the argument appropriately, a contract can cause the multiplication to overflow. This will result in a panic on a debug build of NEAR.

```
    pub fn promise_and(
        &mut self,
        promise_idx_ptr: u64,
        promise_idx_count: u64,
    ) -> Result<PromiseIndex> {
        ...
        self.gas_counter.pay_per_byte(
            self.config.ext_costs.promise_and_per_promise,
            promise_idx_count * size_of::<u64>() as u64,
        )?;
```

Figure 8.1: runtime/near-vm-logic/src/logic.rs#L699-L711.

Note: In a release build, a subsequent call to `memory_get_array_u64` should catch the overflow. For this reason, we consider this finding low-severity.

Exploit Scenario
Alice is developing a contract for the NEAR blockchain. Alice’s contract produces an error when run using a release build of NEAR. But when she runs the same contract on a debug build, NEAR panics. Alice wastes time and energy trying to understand why.

Recommendation
Short term, rewrite the code in Figure 8.1 to use `checked_mul`.

Long term, consider incorporating no-panic into critical code, such as `logic.rs`.

9. Panic in import function “storage_iter_range”

Severity: Low
Type: Data Validation
Target: `storage_iter_range`

Difficulty: Low
Finding ID: TOB-NEAR-009

Description
The import function `storage_iter_range` reads a `start_key` and `end_key` from memory. In the “mock” external implementation, storage is implemented using a `BTreeMap`. Consequently, `start_key` and `end_key` are passed to `BTreeMap`’s `range_search` function. (See Figures 9.1 and 9.2.) If both `start_key` and `end_key` are outside the range of the `BTreeMap` and `start_key` > `end_key`, then `range_search` panics.

```
    pub fn storage_iter_range(
        &mut self,
        start_len: u64,
        start_ptr: u64,
        end_len: u64,
        end_ptr: u64,
    ) -> Result<u64> {
        ...
        let iterator_index =
            self.ext.storage_iter_range(&start_key, &end_key)?;
```

Figure 9.1: runtime/near-vm-logic/src/logic.rs#L1645-L1665.

```
fn range_search<BorrowType, K, V, Q: ?Sized, R: RangeBounds<Q>>(
    root1: NodeRef<BorrowType, K, V, marker::LeafOrInternal>,
    root2: NodeRef<BorrowType, K, V, marker::LeafOrInternal>,
    range: R
)-> ...
{
    match (range.start_bound(), range.end_bound()) {
        ...
        (Excluded(s), Excluded(e)) if s>e =>
            panic!("range start is greater than range end in BTreeMap"),
        _ => {},
    };
```

Figure 9.2: rust/liballoc/collections/btree/map.rs#L1864-L1881.

As mentioned above, the panic occurs in the “mock” external implementation, not `RuntimeExt`. For this reason, we consider this finding low-severity. Nonetheless, `BTreeMap` is used throughout `nearcore`. These other uses of `BTreeMap` should be reviewed to ensure that the same bug is not present elsewhere.

Exploit Scenario
Alice is a NEAR developer testing NEAR's import functions. Panics in the "mock" external implementation cause some of those tests to fail. Critical functionality is not tested because Alice spends her time trying to diagnose the panics.

Recommendation
Short term, either change the "mock" storage implementation, or add checks to `storage_iter_range` to catch the case when both `start_key` and `end_key` are outside the range of the `BTreeMap` and `start_key` > `end_key`. Review uses of `BTreeMap` outside of the "mock" storage implementation to ensure that the same bug is not present elsewhere.

Long term, as mentioned in TOB-NEAR-008, consider incorporating no-panic into critical code, such as `logic.rs`.

10. 32-bit builds should be explicitly disallowed

Severity: Low
Type: Patching
Target: `nearcore`

Difficulty: High
Finding ID: TOB-NEAR-010

Description
NEAR does not currently build on 32-bit platforms because wasmer does not build on 32-bit platforms. NEAR should not rely on wasmer in this way. If wasmer were to change and become buildable on 32-bit platforms, it could enable the use of an essentially untested version of NEAR.

One reason for wasmer's failure to build on 32-bit platforms is the line appearing in Figure 10.1. On a 32-bit architecture, `usize` is 32 bits. Thus, this type cannot hold the constant `1 << 32`.

```
pub const SAFE_STATIC_HEAP_SIZE: usize = 1 << 32; // 4 GiB
```

Figure 10.1: `wasmer/lib/runtime-core/src/memory/static_.rs#L5`.

Because NEAR does not currently build on 32-bit platforms, it cannot be tested on such platforms.

Rather than rely on wasmer to prevent building on 32-bit platforms, NEAR should explicitly disallow such a practice at least in documentation, and preferably in code.

Exploit Scenario
Wasmer is updated to become buildable on 32-bit platforms. Alice builds NEAR on a 32-bit platform using this updated version of wasmer. Subtle differences in how Alice's node operates cause it fork from the NEAR mainnet. Alice suffers financial losses as a result.

Recommendation
Short term, document the fact that NEAR does not support 32-bit platforms.

Long term, investigate ways of preventing NEAR from being built on 32-bit platforms in code. One possibility may be the use of `target_pointer_width`.

References
- How to check in Rust if architecture is 32 or 64 bit?

11. NEAR would benefit from additional WASM runtimes

Severity: Informational
Type: Undefined Behavior
Target: `runtime`

Difficulty: Not applicable
Finding ID: TOB-NEAR-011

Description
NEAR currently has one WASM runtime, wasmer. Adding WASM runtimes would facilitate differential fuzzing, thereby increasing confidence in each runtime.

Wikipedia describes differential fuzzing as "providing the same input to a series of similar applications (or to different implementations of the same application), and observing differences in their execution." Differential fuzzing is effective in situations where defining correct behavior precisely would be difficult. Such is the case, for example, for a WASM compiler or interpreter.

Besides wasmer, awesome-wasm-runtimes lists the following WASM runtimes for Rust:

- Lucet
- Motor
- Serverless wasm
- Wasmo
- wasmvm

We have not attempted to judge the quality of any of these implementations. However, even if an implementation is of lesser quality than wasmer, comparing its behavior to wasmer could help identify bugs in wasmer.

Exploit Scenario
Wasmer contains a bug that might be found through differential fuzzing. Eventually, the bug surfaces by causing the blockchain to fork.

Recommendation
Short term, as new code is added to the NEAR runtime, avoid using features specific to wasmer. This will make it easier to transition to another WASM runtime in the future.

Long term, add one or more WASM runtimes to NEAR besides wasmer.

References
- Destroying x86_64 instruction decoders with differential fuzzing

12. Unnecessary use of mutable state

Severity: Informational
Type: Undefined Behavior
Target: `runtime`

Difficulty: Not applicable
Finding ID: TOB-NEAR-012

Description
The NEAR codebase contains places with unnecessary use of mutable state. It is a good security practice to limit the use of mutable state. We've identified two places with a repeating pattern.

The first one is the use of a mutable accumulator as shown in Figure 12.1.

```
let mut total_gas: Gas = 0;
for action in actions {
    total_gas = safe_add_gas(total_gas,
action.get_prepaid_gas())?;
}
Ok(total_gas)
```

Figure 12.1: runtime/runtime/src/config.rs#L166-L170.

The second one is the use of a mutable result variable spanning a number of different function calls as shown in Figure 12.2.

```
let mut result = ActionResult::default();
```

Figure 12.2: runtime/runtime/src/lib.rs#L523.

Exploit Scenario
A NEAR developer writes a piece of code in which an unrelated mutable variable is modified. This opens the possibility of a number of subtle logic errors and security issues. Time is lost on debugging, and reasoning about state is made more difficult.

Recommendation
Short term, for the accumulator case, use idiomatic functions to confine the mutation inside a limited scope as shown in Figure 12.3.

```
actions.iter().try_fold(0, |sum, action| {
    safe_add_gas(sum, action.get_prepaid_gas())
}
```

Figure 12.3: Code from Figure 12.1 rewritten without explicit mutable state.

For the mutable result, shrink the domain for intermediate functions so they explicitly state which data is exactly required for computation and use immutable intermediate smaller results to build the final result.

Long term, employ a critical attitude towards mutable state during code writing and reviewing. Use mutable state only when absolutely necessary.

13. Imprecise types could cause unnecessary panics

Severity: Informational
Type: Undefined Behavior
Target: `runtime`

Difficulty: Not applicable
Finding ID: TOB-NEAR-013

Description
Some functions have imprecise argument types allowing values for which the function is essentially undefined. Such undefined values are handled with panics. Rust's type system should be used to describe more precisely the actual function domain and deliver the required proof at a call site. An example of a function with an argument type that could be made more precise is shown in Figure 13.1.

```
pub(crate) fn action_function_call(
        ...
        account: &mut Option<Account>,
        ...
    ) -> Result<(), StorageError> {
        let account = account.as_mut().unwrap();
        ...
}
```

Figure 13.1: runtime/runtime/src/actions.rs#L103-L198.

A more complex example with an explicit panic is presented in Figure 13.2. It is not related to a function argument, but to a local variable whose type allows values for which the rest of the code is undefined.

```
let mut postponed_receipts: Vec<Receipt> = vec![];
...
for receipt in postponed_receipts {
    let account_id = &receipt.receiver_id;
    let action_receipt = match &receipt.receipt {
        ReceiptEnum::Action(a) => a,
        _ => panic!("Expected action receipt"),
    };
    ...
}
```

Figure 13.2: runtime/runtime/src/lib.rs#L1034-L1083.

Exploit Scenario
A NEAR developer uses a function that panics for a specific value. The developer doesn't know about the preconditions needed to call the function in a safe way, which opens up a possibility for an attacker to force a node to crash.

Recommendation
Short term, adjust the current overly general types so they describe the domain as closely as possible. An example of such a fix is shown in Figure 13.3.

```
pub(crate) fn action_function_call(
    ...
    account: &mut Account,
    ...
) -> Result<(), StorageError> {
    ...
}
```

Figure 13.3: The function from figure 13.1 rewritten without panic.

Long term, always choose the closest type representation for newly written code, especially when an overly general type would cause the code to panic or behave in an undefined manner.

14. Singlepass wasmer backend is not used

Severity: High
Type: Patching
Target: `near-vm-runner`

Difficulty: Low
Finding ID: TOB-NEAR-014

Description
The crate `near-vm-runner` enables the singlepass wasmer backend, but does not make it the default. Furthermore, only the default backend is used. Consequently, the singlepass backend is not used.

The relevant line from `near-vm-runner`'s `Cargo.toml` file appears in Figure 14.1. Note that the feature `singlepass` is enabled. However, as per the wasmer source code (Figure 14.2), the feature `default-backend-singlepass` must be enabled in order for the singlepass backend to be the default.

```
wasmer-runtime = { version = "0.9.0", features = ["singlepass"] }
```

Figure 14.1: runtime/near-vm-runner/Cargo.toml#L17.

```
#[cfg(feature = "default-backend-singlepass")]
use wasmer_singlepass_backend::SinglePassCompiler as DefaultCompiler;
```

Figure 14.2: wasmer/lib/runtime/src/lib.rs#L218-L219.

As already noted by NEAR Protocol, it is important that the singlepass backend be used in order to avoid denial-of-service attacks.

Exploit Scenario
Eve discovers a denial-of-service attack against wasmer's default cranelift backend. Eve writes a WASM file that exploits the vulnerability, and submits it to the NEAR blockchain. Eve's WASM file causes NEAR nodes to crash, go offline, etc.

Recommendation
Short term, modify `near-vm-runner`'s `Cargo.toml` file so that the feature `default-backend-singlepass` is enabled.

Long term, add continuous integration tests to verify that NEAR calls the singlepass backend when a WASM binary is compiled.

References
- Making WebAssembly even faster: Firefox's new streaming and tiering compiler

15. Malformed input causes panic in WASM deserialization routine

Severity: Medium
Type: Data Validation
Target: `near-vm-runner/src/prepare.rs`

Difficulty: Medium
Finding ID: TOB-NEAR-015

Description
NEAR's runtime creates an internal `ContractModule` in order to consume a deserialized WASM module, which is parsed using `parity-wasm`. Running a fuzzer against the deserialization routine, `deserialize_buffer`, results in a crash when malformed/zeroed input is passed in, despite checks being enforced to determine if the WASM module is valid (see Figure 15.1).

```
impl<'a> ContractModule<'a> {
    fn init(original_code: &[u8], config: &'a VMConfig) ->
Result<Self, PrepareError> {
        wasmparser::validate(original_code, None).map_err(|_|
PrepareError::Deserialization)?;
        let module = elements::deserialize_buffer(original_code)
            .map_err(|_| PrepareError::Deserialization)?;
        Ok(ContractModule { module, config })
    }
    ...
}
```

Figure 15.1: runtime/near-vm-runner/src/prepare.rs.

This finding was discovered using `libFuzzer` with the test harness seen in Figure 15.2, which is similar to `parity-wasm`'s own fuzzing harness.

```
fn fuzz_target(data: &[u8]) {

    // we use binaryen in order to convert random seed bytes into
valid Wasm. If the assertion below passes but `deserialize_buffer`
fails, we may have found a bug
    let module = binaryen::tools::translate_to_fuzz(data);

    // binaryen 0.3.0 validation check
    assert!(module.is_valid());

    let wasm = module.write();
    let config = Config::default();

    // validate with wasmparser as well
    wasmparser::validate(wasm, None)
        .expect("could not validate as valid Module");
```

```
    // helper for preparing a WASM module as contract, calls
`deserialize_buffer` at a lower-level
    prepare::prepare_contract(&wasm, &config)
        .expect("could not deserialize, might be bug");
}
```

Figure 15.2: Fuzzing harness function for testing input WASM modules against `parity_wasm::deserialize_buffer`.

Note that in order to adapt the harness so that a high probability of arbitrary input test cases do not fail, inputs are instead passed to the `binaryen` compiler's auxiliary helper function, `translate_to_fuzz`, for fuzz testing. This enables fuzzer-generated inputs to act as *seeds* for generating random but valid WASM modules for fuzzing.

The result output log from using `libFuzzer` with `cargo-fuzz` can be seen in Figure 15.3.

```
NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better
crash reports.
SUMMARY: libFuzzer: deadly signal
MS: 5 ChangeBinInt-ChangeBinInt-ShuffleBytes-CrossOver-EraseBytes-;
base unit: b258dbb6b908fd279be170b0b64e39e06937e4cb
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x9b,0x9b,0x0,0x0,0x0,0x0,0x0,0x1e,0x0,0x0,0x0,
\x00\x00\x00\x00\x00\x00\x9b\x9b\x00\x00\x00\x00\x1e\x00\x00\x00
artifact_prefix='/home/ex0dus/fuzz/parity-wasm/fuzz/artifacts/deseria
lize/'; Test unit written to
/home/audit/fuzz/artifacts/deserialize/crash-06e2a39035053f1c2ae2db5f
df81a515b01cb1c6
Base64: AAAAAAAAm5sAAAAAHgAAAA==
```

Figure 15.3: `libFuzzer` *output from running the test harness.*

Because the generated fuzzer input was deemed valid by both `binaryen` and `wasmparser` sanity checks but still returns as a crashing edge case, `parity-wasm` is considered the culprit, and the bug represents its inability to reason with the specific module.

Exploit Scenario
Alice is a malicious actor who can interact with Bob, a node operator that runs an instance of NEAR, by using the `near-vm-runner` standalone package. She crafts a malicious WASM contract for the runner to consume, resulting in an abrupt panic and downtime in Bob's node instance.

Recommendation
Short term, compile and link against a locally patched version of `parity-wasm`.

Long term, continue to use patched upstream versions of `parity-wasm` when available, and consider incorporating fuzzing as part of standard and regression testing, as well as the continuous integration pipeline.

References

- `parity-wasm`'s deserialization fuzzer
- Fuzzing with `binaryen`
- Replacing dependencies with patch

16. Gas deductions should precede their associated operations

Severity: High
Type: Denial of Service
Target: `near-vm-logic/src/logic.rs`

Difficulty: Low
Finding ID: TOB-NEAR-016

Description
Some import functions perform expensive operations and then charge for gas afterward. A contract can abuse this fact to cause a node to perform expensive operations for which it does not have the gas to pay.

The function `promise_batch_action_add_key_with_function_call` provides just such an example (see Figure 16.1). The function performs a series of large allocations and reads, but does not charge for gas until after those operations have been completed. The function `storager_iter_range` follows a similar pattern (see Figure 16.2).

```
let public_key = Self::get_from_memory_or_register(
    self.memory,
    &self.registers,
    public_key_ptr,
    public_key_len,
)?;
let allowance = Self::memory_get_u128(self.memory, allowance_ptr)?;
let allowance = if allowance > 0 { Some(allowance) } else { None };
let receiver_id = self.read_and_parse_account_id(receiver_id_ptr,
receiver_id_len)?;
let method_names = Self::get_from_memory_or_register(
    self.memory,
    &self.registers,
    method_names_ptr,
    method_names_len,
)?;
...

self.gas_counter.pay_action_base(...)?;
self.gas_counter.pay_action_per_byte(...)?;
```

Figure 16.1: near-vm-logic/src/logic.rs#L1123-L1163.

```
let start_key =
    Self::get_from_memory_or_register(self.memory,
        &self.registers, start_ptr, start_len)?;
let end_key =
    Self::get_from_memory_or_register(self.memory,
```

```
        &self.registers, end_ptr, end_len)?;
self.gas_counter.pay_per_byte(
    self.config.ext_costs.storage_iter_create_key_byte,
    start_key.len() as u64,
)?;
self.gas_counter.pay_per_byte(
    self.config.ext_costs.storage_iter_create_key_byte,
    end_key.len() as u64,
)?;
```

Figure 16.2: near-vm-logic/src/logic.rs#L1653-L1664.

Exploit Scenario

Eve writes a contract that makes an expensive call to `promise_batch_action_add_key_with_function_call`. Eve repeatedly calls the contract with insufficient gas, slowing down all nodes that try to execute her transaction.

Recommendation

Short term, within `promise_batch_action_add_key_with_function_call` and `storage_iter_range`, move the gas deductions so they occur before the expensive allocations and reads.

Long term, review all gas deductions. Ensure that the deductions occur before the operations they are meant to pay for.

17. Out-of-memory error caused by “promise_batch_then” and “create_receipt”

Severity: High
Type: Denial of Service
Target: `near-vm-logic/src/logic.rs, runtime/src/ext.rs`

Difficulty: Low
Finding ID: TOB-NEAR-017

Description
The function `create_receipt`, called by `promise_batch_then`, can be made to perform large allocations, causing a NEAR code to crash.

The function `create_receipt` takes a vector of receipt indices (`receipt_indices`) and an account ID (`receiver_id`) as arguments (see Figure 17.2). The vector can be long and the account ID can be large—thousands of elements and several megabytes (respectively) within our fuzzing runs. Thus, a NEAR node can be asked to allocate several gigabytes of RAM.

```
fn create_receipt(&mut self, receipt_indices: Vec<u64>,
        receiver_id: String) -> ExtResult<u64> {
    let mut input_data_ids = vec![];
    for receipt_index in receipt_indices {
        let data_id = self.new_data_id();
        self.action_receipts
            .get_mut(receipt_index as usize)
            .expect("receipt index should be present")
            .1
            .output_data_receivers
            .push(DataReceiver { data_id,
                receiver_id: receiver_id.clone() });
        input_data_ids.push(data_id);
    }
```

Figure 17.1: runtime/runtime/src/ext.rs#L178-L189.

Compounding the problem, the gas charged does not take into account the account ID length (see Figure 17.2). Thus, the gas charged is not proportional to the true cost of the work performed.

```
        let sir = account_id == self.context.current_account_id;
        let deps: Vec<_> = receipt_dependencies
            .iter()
            .map(|receipt_idx| {
                self.receipt_to_account
```

```
                    .get(receipt_idx)
                    .expect(...)
                    == &account_id
            })
            .collect();
        self.pay_gas_for_new_receipt(sir, &deps)?;
```

Figure 17.2: near-vm-logic/src/logic.rs#L800-L810.

Exploit Scenario

Eve writes a contract that makes an expensive call to `promise_batch_then`. Eve repeatedly calls the contract with insufficient gas, slowing down all nodes that try to execute her transaction.

Recommendation

Short term, make the gas cost of `promise_batch_then` proportional to the product of the number of receipts and the size of the account ID.

Long term, consider whether the size of an account ID should be limited. Limiting the size will help reduce the damage if the gas cost of another operation fails to take into account the possibility of large account IDs.

18. Structure “FinalExecutionOutcome” could benefit from including a signature

Severity: Informational
Type: Cryptography
Target: `runtime`

Difficulty: Undetermined
Finding ID: TOB-NEAR-018

Description
Code in a repository neighboring `nearcore` (`nearlib`) suggests that `FinalExecutionOutcome` structures will be used outside of NEAR nodes (see Figure 18.1). If these structures are to be used in a context where they cannot be verified, they would benefit from including a signature produced by the node that generated them.

```
export interface FinalExecutionOutcome {
    status: FinalExecutionStatus | FinalExecutionStatusBasic;
    transaction: ExecutionOutcomeWithId;
    receipts: ExecutionOutcomeWithId[];
}
```

Figure 18.1: nearlib/src.ts/providers/provider.ts#L65-L69.

Consider the example of light wallets. Suppose that a light wallet sends a transaction to a node and receives the transaction’s `FinalExecutionOutcome` in return. If the `FinalExecutionOutcome` contained a signature, the node would be less inclined to lie to the light wallet, because the holder of the light wallet would have proof of the lie.

Exploit Scenario
Alice asks Eve, a NEAR node operator, to send her rent payment to Bob. Eve sends Alice a fraudulent `FinalExecutionOutcome` indicating that the transfer was successful. Alice learns that the structure was fraudulent by way of an eviction notice.

Recommendation
Short term, document the fact that `FinalExecutionOutcome` structures should be used only in contexts where they can be verified.

Long term, consider whether `FinalExecutionOutcome` structures could be useful in other contexts. If so, require that they be signed by the node that generates them.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Non–Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- **Testing and non-testing code appear together in some files.** Keep testing and non-testing code in separate files, and give files containing testing code predictable names, e.g., "`test_….rs`." Suppose you become concerned about the use of a feature, but you're willing to continue using it in testing code. Following our recommended strategy can make it easier to identify all of the non-testing code that uses that feature.

- **Typo in "invlid" should be reported to wasmparser maintainers.** The existence of this typo forces `nearcore` to check for an error message featuring that typo. Reporting this typo to the `wasmparser` maintainers would save NEAR developers from having to worry that the typo lies in `nearcore` alone.

C. Non-Determinism Concerns

Non-determinism was observed in the NEAR runtime during the course of our assessment. This appendix explains why there may be sources of non-determinism in the NEAR runtime beyond those that we've identified, why this is a concern, and what NEAR Protocol can do to address this concern.

The NEAR runtime was fuzzed using American Fuzzy Lop (AFL). One of the statistics that AFL reports for a program is "stability." Roughly speaking, stability is the fraction of instructions that are executed the same number of times across different runs of the program with the same input. So, for example, if, for some input, an instruction is executed once across all runs of a program, that instruction will count *toward* stability. On the other hand, if for some input an instruction is executed once in one run but twice in another, that instruction will count *against* stability.[1]

If a program is completely deterministic, it should have 100% stability or close thereto. While fuzzing the NEAR runtime, we observed stability of less than 20%. Again, roughly speaking, this means that fewer than 20% of instructions were executed the same number of times across different runs of the program with the same input.

It's worth mentioning that while we found AFL's stability metric to be quite reliable in practice, it is not perfect. For example, the AFL source code contains the following comment:

```
Note that it's an explicit non-goal to instrument hand-written assembly,
be it in separate .s files or in __asm__ blocks. The only aspiration this
utility has right now is to be able to skip them gracefully and allow the
compilation process to continue.
```

Thus, low stability is not a *guarantee* of the presence of non-determinism. Put another way, it is possible that AFL's stability reports for the NEAR runtime are lower than they should be.

However, non-determinism is of particular concern for blockchain projects. If non-determinism causes two different nodes to decide that two different blocks should be at the head of the blockchain, the network may fork. Thus, the possibility that non-determinism might reside within the NEAR runtime motivated us to investigate further.

[1] A more precise definition would involve instrumentation points, not instructions. Unfortunately, the best documentation for stability is the AFL code itself.

One source of non-determinism identified was the use of randomness by `HashMap` and `HashSet` (TOB-NEAR-005). In order to deter DoS attacks, `HashMap` and `HashSet` use randomness to construct a hashing function that an attacker can't predict. In our opinion, this use of randomness is appropriate and should be continued *in release builds*. However, care must be taken to ensure `HashMap` and `HashSet` are not used in places where they could cause disagreement amongst NEAR nodes, as described above.

Another source of non-determinism identified was wasmer's use of the rayon library's `ParallelIterator`. Wasmer uses this facility to compile WASM functions across multiple threads. Since this use of multithreading occurs outside of contract execution, it appears to be innocuous.

Of course, WebAssembly itself presents a source of nondeterminism, but such non-determinism is believed to be *limited* and *local*.

Even after disabling `HashMap` and `HashSet`'s use of randomness and wasmer's use of `ParallelIterator`, AFL still reported low stability. At present, we believe the low stability reports to be the result of using AFL in "persistent mode" on Rust programs that use `lazy_static`.

- In persistent mode, AFL does not launch a new process for each fuzzing run. Rather, the process loops, repeatedly executing the function under test with new fuzzing inputs. Persistent mode is used by `afl.rs`'s `fuzz!` macro.[2]
- Rust's `lazy_static!` macro allows one to declare a static variable that is initialized on its first use. For instance, `lazy_static!` is used in `core/primitives`.

It seems that in the first iteration of the fuzzing loop, one or more `lazy_static` variables is initialized; in subsequent iterations of the loop, the variables' initializers are not run. The instructions in those initializers are counted against the stability, causing it to be low.

As mentioned above, non-determinism is of particular concern for blockchain projects. Crucially, If any non-determinism is still present, it must not cause two different nodes to disagree on the state of the blockchain. So, although we are reasonably confident in the above theory, we recommend adding one more WASM runtimes besides wasmer (TOB-NEAR-011). This will facilitate differential fuzzing. Moreover, it would seem unreasonable to expect two WASM runtimes to exhibit non-determinism in the same way. Thus, if wasmer were to be non-deterministic, differential fuzzing could help expose that fact and flush out the sources of that non-determinism.

Another way to address non-determinism under a fuzzing environment is to extend the currently supported "mock" submodules and incorporate a helper function to act as a

[2] https://github.com/rust-fuzz/afl.rs/blob/f3d6159/src/lib.rs#L150-L172

deterministic RNG. This helper function would swap-in for calls to `rand::thread_rng`, not only enabling higher stability, but also allowing for coverage guidance under a fuzzer to explore program states more effectively, without the obstruction of randomness.