Ledger Filecoin App

Security Assessment

July 24, 2020

Prepared For:
Jonathan Victor | *Protocol Labs*
jonathan@protocol.ai

Juan Leni | *Zondax GmbH*
juan.leni@zondax.ch

Jon Schwartz | *Open Work Labs*
jonathan@openworklabs.com

Prepared By:
Brad Larsen | *Trail of Bits*
brad.larsen@trailofbits.com

Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Executive Summary

From July 13 through July 24, 2020, Protocol Labs engaged Trail of Bits to review the security of the Ledger Filecoin app. Trail of Bits conducted this assessment over the course of two person-weeks with two engineers working from commit hash `328fb293` from the `Zondax/ledger-filecoin` repository.

In the first week, we familiarized ourselves with the codebase, compiled all tests to run both natively and in the Ledger emulator, and compiled the code with address and undefined behavior instrumentation for fuzz testing (see Appendix D). In week two, we performed manual review and triaged the dynamic analysis findings.

Five issues were found, ranging from informational to low severity, with one finding of undetermined severity. The low-severity issues were related to data validation and undefined behavior resulting from a maliciously crafted transaction; at worst, they could theoretically cause the device to crash. We were unable to find any attack vectors in this engagement that would allow an attacker to compromise secrets or sign arbitrary transactions. The undetermined-severity finding is related to apparent latent bugs in a third-party library that was not written by the Ledger Filecoin team but is used by the app.

The code is written with clear forethought about the hardware limitations of the Ledger device, and employs many security best practices such as read-only memory and stack canaries. However, *all* Ledger app development is made precarious by the device's inherent hardware constraints:

- Severe memory restrictions.
- Stack depth limitations.
- Sensitivity to integer over- and underflow.
- Data truncation on the small screen.
- Etc.

We suggest that the Filecoin app integrate fuzz testing into its development lifecycle, simplify its dependency management, and further vet all third-party libraries employed in production. Other issues that do not have direct security implications are included in Appendix B.

Project Dashboard

Application Summary

Name	Ledger Filecoin app
Version	commit hash 328fb293
Type	C and C++
Platforms	Ledger Nano S and X

Engagement Summary

Dates	July 13–July 24, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	0	
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	1	■
Total Undetermined-Severity Issues	1	■
Total	5	

Category Breakdown

Configuration	1	■
Data Validation	2	■■
Undefined Behavior	2	■■
Total	5	

Engagement Goals

The engagement was scoped to provide a security assessment of the Filecoin app for Ledger devices. The web wallet and signing libraries were out of scope and will be assessed in a subsequent engagement.

Specifically, we sought to answer the following questions:

- Is it possible to retrieve a private key from a device?
- Could a maliciously crafted transaction appear as a legitimate one on the device, tricking the user into signing it?
- Are the hardware constraints of the devices properly mitigated? Is there potential for stack or memory exhaustion?
- Are there latent logical errors in the application?
- Does any of the code exhibit undefined behavior?

Coverage

Fuzz-tested the codebase. We wrote fuzzers using `libFuzzer` for six different APIs that are involved with processing user-controllable data. See Appendix C.

Tested for memory errors with a combination of the LLVM Address Sanitizer (ASan) and manual analysis.

Tested for undefined behavior with a combination of the LLVM Undefined Behavior Sanitizer (UBSan) and manual analysis.

UI state machine correctness. Manual review.

Cryptographic correctness. Manually checked for common cryptographic errors and patterns.

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.

Note: Some of these recommendations have already been implemented between the initial delivery of this report and its final publication, but we have retained all of the original recommendations in this section and listed specific remediations in the individual finding summaries below.

Short Term

☐ **Validate inputs to the `base32` functions** to ensure undefined behavior cannot occur.

☐ **Add a check to `formatProtocol` before calling `decompressLEB128` to ensure the input buffer is large enough.** This will require a single conditional, and will cohere with the existing data validation patterns of the surrounding code. It will improve defense in depth, and prevent this from becoming a real issue when future code changes are made.

☐ **Fix the data validation issue that can cause a read overrun in `base32` encoding.** This may simply require an additional check after the call to `base32_encode` to see if the output buffer's full length was used.

☐ **Document the testing Docker container.** For the two images used to build, add an overview and metadata on Docker Hub describing what the images do and where they come from. Describe in the project's README how the Docker image could be built from scratch or swapped out. Either employ Docker's "content trust" digital signatures in the container or pin against a hash of the expected container.

☐ **On a regular basis: 1. Run your test suites with Address Sanitizer and Undefined Behavior Sanitizers enabled, and 2. Run the fuzz testing targets we added during this assessment.** Seek clarification on these issues from the TinyCBOR developers, potentially by opening GitHub issues on their project.

Long Term

☐ **Improve unit test coverage.** Test the behavior of the `base32` functions and consider switching to a different `base32` implementation.

☐ **Regularly test the codebase and perform fuzz testing with Address Sanitizer.** This will detect many issues that would not otherwise be found through traditional unit testing. Run fuzz testing routinely against APIs that deal with user-controllable data.

☐ **Prefer safer alternatives to C-string functions, e.g., `strnlen` instead of `strlen`.** This is particularly important when dealing with user-controllable binary data, such as Filecoin transaction data.

☐ **Use automated builds to produce the images pushed to Docker Hub through Docker's own mechanism or through GitHub Actions.** Ensure that it is possible to perform a build of the Ledger Filecoin app without network access and build the images from scratch.

☐ **Consider a focused security assessment and/or security hardening engagement for the `tinycbor` dependency.** There appear to be latent issues in the codebase that could become problematic in the future.

Findings Summary

#	Title	Type	Severity
1	Undefined behavior in `base32` codec	Low	Undefined Behavior
2	Missing data validation in `formatProtocol()`	Informational	Data Validation
3	A malicious transaction can cause a read overrun	Low	Data Validation
4	Possible supply-chain attack in Docker images used for testing	Low	Configuration
5	A malicious transaction can cause `tinycbor` to crash	Undetermined	Undefined Behavior

1. Undefined behavior in base32 codec

Severity: Low
Type: Undefined Behavior
Target: app/src/base32.c

Difficulty: High
Finding ID: TOB-FILECOIN-001

Description
The base32_encode and base32_decode functions invoke undefined behavior when processing certain inputs. Specifically, an int variable in each function can be left-shifted too many places, or left-shifted when its value is negative. Both instances are undefined behavior:

> *The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 × 2E2, reduced modulo one more than the maximum value representable in the result type. If E1 has a signed type and nonnegative value, and E1 × 2E2 is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.*
>
> ISO C99 (6.5.7/4)

A standards-conforming C compiler is under no obligations when a program invokes undefined behavior, and the resulting behavior cannot be safely reasoned about. In other projects, gcc 4.8 and newer have been observed to silently elide code that relies on undefined behavior, even when all compiler warnings are enabled.

The vulnerable base32_encode function can be transitively called on untrusted transaction input from tx_getItem. The base32_decode function does not appear to be used.

This issue was found through fuzz testing with Undefined Behavior Sanitizer enabled, which is discussed in more detail in Appendix C. The issue can reproduced with a 4-byte input using -fsanitize=undefined that uses either GCC 9.3 or Clang 10 on Ubuntu 20.04 in the following test program:

```
#include <cstdint>
#include <cstdio>
#include "base32.h"

static uint8_t INPUT[] = {100, 255, 255, 0};
static uint8_t ENCODED[128];

int main(int argc, const char **argv) {
    int encoded_size = base32_encode(INPUT, (int)sizeof(INPUT),
                                     ENCODED, (int)sizeof(ENCODED));
    if (encoded_size == -1)
        return 1;
    return 0;
}
```

Figure 1.1: Test program to reproduce the finding.

Then, to observe the undefined behavior:

```
$ UBSAN_OPTIONS=print_stacktrace=1 ./build/base32_problems
app/src/base32.c:84:28: runtime error: left shift of 1694498560 by 3 places cannot be
represented in type 'int'
    #0 0x4c8714 in base32_encode app/src/base32.c:84:28
    #1 0x4c6b4a in main base32_problems.cpp:13:24
    #2 0x7f90216740b2 in __libc_start_main
/build/glibc-YYA7BZ/glibc-2.31/csu/../csu/libc-start.c:308:16
    #3 0x41c2dd in _start (build/base32_problems+0x41c2dd)
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior app/src/base32.c:84:28 in
```

Figure 2.1: Running the test program to observe the undefined behavior.

Exploit Scenario
Alice constructs a malicious transaction that exploits the undefined behavior and allows an incorrect address to be used in the transaction.

Recommendation
Short term, validate inputs to the `base32` functions to ensure that undefined behavior cannot occur.

Long term, add unit test coverage to check this behavior, and consider switching to a different `base32` implementation.

Remediation

- All counters were converted to unsigned integers in `base32` encoding in commit `e087a24a`.
- Decoding was modified to accept the encoded size in commit `888201a4`.

2. Missing data validation in `formatProtocol()`

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-FILECOIN-002
Target: app/src/crypto.c

Description

The `formatProtocol()` function has incomplete data validation logic: Before `decompressLEB128` is called, there is no check that the input buffer argument (`addressBytes + 1`) is big enough, and a read overrun could occur.

```
uint16_t formatProtocol(const uint8_t *addressBytes,
                        uint16_t addressSize,
                        uint8_t *formattedAddress,
                        uint16_t formattedAddressSize) {
    if (formattedAddress == NULL) {
        return 0;
    }
    MEMZERO(formattedAddress, formattedAddressSize);

    const uint8_t protocol = addressBytes[0];

    formattedAddress[0] = isTestnet() ? 't' : 'f';
    formattedAddress[1] = (char) (protocol + '0');

    uint16_t payloadSize = 0;
    switch (protocol) {
        case ADDRESS_PROTOCOL_ID: {
            uint64_t val = 0;

            if (!decompressLEB128(addressBytes + 1, &val)) {
                return 0;
            }

            if (uint64_to_str((char *) formattedAddress + 2,
                              (uint32_t) (formattedAddressSize - 2), val) != NULL) {
                return 0;
            }

            return strlen((const char *) formattedAddress);
        }
 ⋮
```

Figure 2.1: Missing data validation on lines 251–280 of `crypto.c`.

Note: Presently, this is not an exploitable issue due to the exact details of the code, and because the input buffer is sufficiently large along every execution path that could call `formatProtocol`.

Exploit Scenario
A vulnerable execution path to this function is added in future codebase modifications. An attacker crafts a malicious transaction that calls into `decompressLEB128` with a buffer that is too small, causing a Ledger device to crash or data to be exfiltrated.

Recommendation
Short term, for defense in depth and to prevent this from becoming a real issue when future code changes are made, add a check to `formatProtocol` before calling `decompressLEB128` to ensure the input buffer is large enough. This will require a single conditional, and will cohere with the existing data validation patterns of the surrounding code.

Long term, regularly test the codebase and perform fuzz testing with Address Sanitizer to detect similar issues.

Remediation
- Input size was added as an argument to `decompressLEB128` in commit `9d6036e4`.
- Output buffer size was increased to ensure ample padding in commit `dd29f1db`.
- Fuzzing target was added for the `decompressLEB128` function.

3. A malicious transaction can cause a read overrun

Severity: Low
Type: Data Validation
Target: app/src/crypto.c

Difficulty: Medium
Finding ID: TOB-FILECOIN-003

Description

The base32_encode function is not guaranteed to null-terminate its output. If an encoded value is just the right length, formattedAddress will not be null-terminated, and strlen will read past its end.

```
    // Now prepare the address output
    if (base32_encode(payload_crc,
                      (int) (payloadSize + CHECKSUM_LENGTH),
                      (formattedAddress + 2),
                      formattedAddressSize - 2) < 0) {
        return 0;
    }

    return strlen((char *) formattedAddress);
```

Figure 3.1: Formatted address encoding on lines 312–320 of crypto.c.

The formattedAddress function is called from parser_getItem, which is called upon transaction receipt to parse the CBOR payload.

The following base64-encoded CBOR payload triggers this read overrun:

iQBYMQMKAAD//////////wAAAAAAAAAAEIAiQAAQAAAQMD5RAD////////8AAAAAAAAAAABCAIkAQEAAAEA=

And the read overrun corresponds to this valid CBOR input structure:

```
[0,
b'\x03\n\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00
\x00B\x00\x89\x00\x00@\x00\x00@\xc0\xf9D\x00\xff\xff\xff\xff\xff\xff\x00\x00\x00
\x00\x00\x00\x00\x00\x00', b'\x00\x89', 0, b'', b'', 0, 0, b'']
```

This finding has low severity because although it can cause the app to crash, there seems to be no other way to exploit the bug.

Exploit Scenario

A maliciously crafted transaction causes the app to crash.

Recommendation

Short term, fix this data validation issue. This may simply require an additional check after the call to base32_encode to see if the output buffer's full length was used.

Long term, prefer safer alternatives to C-string functions, e.g., `strnlen` instead of `strlen`, particularly when dealing with user-controllable binary data such as Filecoin transaction data.

Also, regularly run fuzz testing against APIs that deal with user-controllable data. As part of this assessment, we have provided several fuzz testing targets, which are discussed in more detail in Appendix C.

Remediation

- Length-bounded string comparison was added in commit `9d6036e4`.

4. Possible supply-chain attack in Docker images used for testing

Severity: Low
Difficulty: High
Type: Configuration
Finding ID: TOB-FILECOIN-004
Target: deps/ledger-zxlib/dockerized_build.mk

Description
The Dockerized build Makefile in the `ledger-zxlib` dependency uses either the `zondax/builder-bolos` or the `zondax/builder-bolos-1001` Docker image to build. During the build, the image is pulled from Docker Hub but the code to build these images is not included in the `ledger-filecoin` repository, there's no container signing, and there's no hash of the container pinned.

Note that the Docker image is used only in development and testing, and that an official build of the Ledger Filecoin app is built from source and cryptographically signed by Ledger.

Exploit Scenario
An attacker interposes a malicious Docker `zondax/builder-bolos` image into the supply chain and uses this to steal secrets from development machines.

Recommendation
Short term, for the two images used to build, add an overview and metadata on Docker Hub describing what the images do and where they come from. Describe in the project's README how the Docker image could be built from scratch or swapped out. Either employ docker's "content trust" digital signatures in the container or pin against a hash of the expected container.

Long term, use automated builds to produce the images pushed to Docker Hub through Docker's own mechanism or through GitHub Actions. Ensure that it is possible to perform a build of the Ledger Filecoin app without network access and build the images from scratch.

References
- https://docs.docker.com/engine/security/trust/content_trust/
- https://docs.docker.com/docker-hub/builds/

Remediation
- README was updated with a link to the source Dockerfile in commit `b5ad6513`.
- `sha256` hash of the installer script was pinned in the Dockerfile in commit `404f3e51` and in the specific git commit hash in `f0002c8d`.
- Specific Docker images were pinned in commit `41f1be63`.

5. Multiple uses of undefined behavior in tinycbor

Severity: Undetermined
Difficulty: High
Type: Undefined Behavior
Finding ID: TOB-FILECOIN-005
Target: `deps/tinycbor`

Description

The `tinycbor` library invokes undefined behavior in several places when the Filecoin app processes user-controllable transaction data. These appear to be previously undiscovered bugs in the third-party `tinycbor` library that is a dependency of the Ledger Filecoin app, and is not due to flaws inherent in the Ledger Filecoin code itself. These issues were discovered toward the end of the engagement and there was insufficient time to determine whether they are exploitable.

A standards-conforming C compiler is under no obligations when a program invokes undefined behavior, and the resulting behavior cannot be safely reasoned about. Changes to anything, including compiler toolchain version, build flags, target hardware, operating system, system header files, application source code, etc., could result in different observable behavior. This includes applying a non-zero offset to a null pointer, as well as numeric over- and underflows.

The former behavior is due to pointer arithmetic that in certain code paths will allow a null pointer to be offset by `SIZE_MAX`. The addition occurs in the `iterate_string_chunks` function and is called with the erroneous arguments in several instances, including string copying and string length calculation. Despite invoking undefined behavior, these functions do currently appear to compile to semantically correct code for the Ledger devices. However, compiler optimization changes are not guaranteed to preserve this behavior. The behavior can be reproduced with virtually any valid CBOR input that contains strings.

Note that the "offsetting a `NULL` pointer" undefined behavior does not pose immediate risk to the Ledger Filecoin app, but could become a problem in the future. Starting with LLVM 10, the optimizer in Clang takes advantage of this, and could unexpectedly eliminate code. It is undetermined whether newer versions of GCC take advantage of this.

The latter undefined behavior related to numeric over- and underflow is of more concern on Ledger devices. It appears this behavior can be triggered by passing a large value as the version parameter in the Filecoin transaction's CBOR payload, and may occur because there's no check for the `CborIteratorFlag_IntegerValueTooLarge` flag in `tinycbor`'s `cbor_value_get_int64` function.

When built with Clang 10 and Undefined Behavior Sanitizer, the `fuzz-parser_parse` fuzz target (see Appendix C) reveals the following undefined behaviors:

```
deps/tinycbor/src/cborparser.c:1126:37: runtime error: applying zero offset to null pointer
    #0 0x57f801 in iterate_string_chunks deps/tinycbor/src/cborparser.c:1126:37
    #1 0x57ba84 in _cbor_value_copy_string deps/tinycbor/src/cborparser.c:1214:21
    #2 0x575b24 in advance_recursive deps/tinycbor/src/cborparser.c:508:16
    #3 0x574246 in cbor_value_advance deps/tinycbor/src/cborparser.c:546:12
    #4 0x55fae4 in _read app/src/parser_impl.c:271:5
    #5 0x552e77 in parser_parse app/src/parser.c:34:12
    ...

deps/tinycbor/src/cborparser.c:1136:33: runtime error: applying non-zero offset 2 to null
pointer
    #0 0x57fe63 in iterate_string_chunks deps/tinycbor/src/cborparser.c:1136:33
    #1 0x57ba84 in _cbor_value_copy_string deps/tinycbor/src/cborparser.c:1214:21
    #2 0x575b24 in advance_recursive deps/tinycbor/src/cborparser.c:508:16
    #3 0x574246 in cbor_value_advance deps/tinycbor/src/cborparser.c:546:12
    #4 0x55fae4 in _read app/src/parser_impl.c:271:5
    #5 0x552e77 in parser_parse app/src/parser.c:34:12
    …

deps/tinycbor/src/cbor.h:375:19: runtime error: negation of -9223372036854775808 cannot be
represented in type 'int64_t' (aka 'long'); cast to an unsigned type to negate this value to
itself
    #0 0x5668d1 in cbor_value_get_int64 deps/tinycbor/src/cbor.h:375:19
    #1 0x55e2db in _read app/src/parser_impl.c:260:5
    #2 0x552e77 in parser_parse app/src/parser.c:34:12
    ...

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior deps/tinycbor/src/cbor.h:375:19 in
deps/tinycbor/src/cbor.h:375:28: runtime error: signed integer overflow:
-9223372036854775808 - 1 cannot be represented in type 'long'
    #0 0x56693a in cbor_value_get_int64 deps/tinycbor/src/cbor.h:375:28
    #1 0x55e2db in _read app/src/parser_impl.c:260:5
    #2 0x552e77 in parser_parse app/src/parser.c:34:12
    ...
```

Figure 5.1: Fuzz testing to reveal the undefined behavior.

Exploit Scenario
An attacker crafts a malicious Filecoin transaction that makes the `tinycbor` code invoke undefined behavior, which causes the Ledger device to crash or the transaction to be mishandled.

Recommendations
Short term, on a regular basis: 1. Run your test suites with Address Sanitizer and Undefined Behavior Sanitizers enabled, and 2. Run the fuzz testing targets we added during this assessment. Seek clarification on these issues from the TinyCBOR developers, potentially by opening GitHub issues on their project.

Long term, consider a focused security assessment and/or security hardening engagement for the `tinycbor` dependency.

References
- https://ledger.readthedocs.io/en/latest/additional/security_guidelines.html?highlight=overflow#integer-overflows-underflows
- https://reviews.llvm.org/D67122
- https://reviews.llvm.org/rL369789

Remediation
- `tinycbor` library was locally patched to remove the undefined behavior caused by offsetting a null pointer in commit 860b25af.

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. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Keep all dependencies up to date. Both the `fmt` and `jsoncpp` dependencies that are installed through the Conan package manager are pinned to outdated versions from obsolete Conan packages. The `fmt` library uses the obsolete Conan package from `https://github.com/bincrafters/conan-fmt`. Likewise, the jsoncpp library uses the obsolete Conan package from `https://github.com/theirix/conan-jsoncpp`, is over a year and three revisions out of date, and is missing several bug fixes. These libraries are only used for testing, so there is no immediate security risk for the production app. The Conan dependencies were updated and version-pinned in commit `feab95d8`. JavaScript dependencies were also updated in commit `a5ec864d`.

Use a single mechanism for dealing with third-party dependencies, even in testing code. Currently, the Ledger Filecoin app uses three different mechanisms for managing third-party dependencies:

- "Vendoring," where the source of the dependency is added directly to the repository (used for `tinycbor`, `tincbor-ledger`, and `ledger-zxlib`).
- Git submodules, where an external repository is referenced as a subdirectory (used for BLAKE2 and `nons-secure-sdk`).
- The Conan package manager, which imports "recipes" specified via Python scripts from GitHub (used for `jsoncpp` and `fmt`).

Having three mechanisms significantly complicates the build process, and makes it difficult to keep an inventory of the third-party dependencies of the project.

Add source code comments regarding credentials stored in git. The Dockerized build Makefile in the `ledger-zxlib` dependency contains hard-coded “SCP” credentials. These credentials are used solely for testing and are not used in production. However, a future developer reading the code may not know their purpose and may be alarmed at the presence of hard-coded credentials. A comment should be added to the Makefile to describe their innocuous purpose. Also, consider integrating a tool like truffleHog into your continuous integration pipeline.

C. Fuzz Testing

At a very high level, fuzz testing is a type of systems testing where random inputs are used to elicit errors and unexpected behaviors from the system. Fuzz testing in software can be particularly effective at identifying memory errors and data validation issues.

As part of this assessment, we wrote fuzz testing targets using `libFuzzer` for six different functions used by the Filecoin app to handle user-controllable transaction data. We ran these fuzz testing targets for approximately three CPU days and found several informational- and low-severity issues related to undefined behavior, memory errors, and data validation (TOB-FILECOIN-001, TOB-FILECOIN-002, TOB-FILECOIN-003, TOB-FILECOIN-005).

We have provided these fuzz testing targets and they have been integrated into the codebase in pull request #53.

D. Compiling the Code with Clang 10

The Filecoin Ledger app fails to compile when using Clang 10:

```
[ 66%] Building CXX object CMakeFiles/unittests.dir/tests/crypto.cpp.o
/usr/bin/clang++  -DAPP_CONSUMER -Ibuild/googletest-src/googlemock/include
-Ideps/tinycbor/src -Ideps/BLAKE2/ref -Ideps/ledger-zxlib/include -Iapp/src
-Iapp/src/lib -Iapp/src/common -isystem build/googletest-src/googletest/include
-isystem
/home/test/.conan/data/fmt/6.0.0/bincrafters/stable/package/803d1cd04e225b70657161961aa
f49e6f4674680/include -isystem
/home/test/.conan/data/jsoncpp/1.9.0/theirix/stable/package/d3f06493bbe2be57151a94f1078
08e265edeb733/include -isystem build/googletest-src/googletest  -fsanitize=address
-fno-omit-frame-pointer -g   -std=gnu++11 -o
CMakeFiles/unittests.dir/tests/crypto.cpp.o -c tests/crypto.cpp
In file included from tests/crypto.cpp:21:
In file included from app/src/crypto.h:24:
app/src/coin.h:53:29: error: invalid suffix on literal; C++11 requires a space between
literal and identifier [-Wreserved-user-defined-literal]
#define APPVERSION_LINE2 "v"APPVERSION
```

This is not an issue for the production build since Ledger will build the app itself using a specific version of `gcc`. However, for development and testing, it is useful to employ the latest version of Clang to take advantage of its various security features and sanitizers.

The Clang 10 incompatibility occurs because a C preprocessor symbol is defined in a way that could be confused with a [C++11 user-defined literal]() at `app/src/coin.h:53`:

```
#define APPVERSION_LINE2 "v"APPVERSION
```

This can be resolved by adding a space between "`v`" and `APPVERSION`.

These recommendations were subsequently implemented in the codebase in commit [19adde85]().