Least Authority

PRIVACY MATTERS

Ledger Application

Final Security Audit Report

Tezos

Report Version: 24 July 2018

Table of Contents

Overview

The Tezos Foundation has requested that Least Authority perform a security audit of the two applications for the Ledger Nano S Hardware Wallet developed by Obsidian Systems, in preparation for the upcoming betanet and mainnet launches. The two applications include:

1. **Tezos Ledger Baking Application.** Functionality includes:
   - Passively sign blocks and endorsements for a given baker with a given key
   - At start up, the user would authorize signing with a given key on the device, which would enable 'passive' signing as the baker is asked to sign
   - The application enforces a 'High Water Mark' (HWM) of the highest block level it has signed. If asked to sign a block/endorsements below this level, the device automatically rejects signing. The HWM can be reset
2. **Tezos Ledger Wallet Application.** Functionality includes:
   - Sending and receiving tokens from the Ledger
   - Delegating baking/voting rights to another key

The audit was performed from June 18-22, 2018 by Ramakrishnan Muthukrishnan and Meejah. The initial report was issued on June 22, 2018. The final report was issued on July 24, 2018, following a discussion and verification phase.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Ledger Applications followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

- https://github.com/obsidiansystems/ledger-app-tezos.git

Specifically, we examined the Git revision:

```
F416975f94eee27b7c7b5179da38dd89bae0bd41
```

All file references in this document use Unix-style paths relative to the project’s root directory.

The verification was done on the audit-cleanup branch at the git revision:

```
292431e042182d164bf526328871c20568cc68d6
```

Manual Code Review

In manually reviewing all of the contract code, we looked for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also kept an eye out for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus was on the application code, we examined dependency code and behavior when it was relevant to a particular line of investigation.

Our investigation focused on the following areas:

- Potential compromise of secrets
- C programming pitfalls

- Input validation
- Other methods that attackers can render the program useless

The files we manually reviewed included:

- Everything in the repo under src/*.{c, h}, Makefile, install scripts

Findings

Code Quality

The code is well organized with good interfaces between modules and is easy to read, understand and, as a result, easy to maintain. We did not find any serious issues such as leakage of secrets. We did find a few programming pitfalls which are not too serious in the current context and are outlined below.

Issues

We list the issues we found in the code in the order we reported them.

ISSUE / SUGGESTION	STATUS
Issue A: Dereferencing Pointers Without NULL Check	*Resolved*
Issue B: Issue B: Incrementing the Void Pointer	*Resolved*
Issue C: Signed Int Versus Unsigned Int Comparison	*Resolved*
Issue D: Incorrect Types for Loop Variables	*Resolved*
Issue E: Functions with No Input Incorrectly Defined	*Resolved*
Issue F: Use of Magic Numbers in the Code	*Resolved*
Issue G: Support for Nano-S Running Older Firmware Versions	*Confirmed Won't Fix*
Suggestion 1: Build Documentation Incomplete	*Resolved*
Suggestion 2: Makefile: Warnings are Turned Off	*Resolved*

Issue A: Dereferencing Pointers Without NULL Check

Synopsis

In a few functions, the pointer passed to the function is dereferenced without performing a NULL check.

Impact

It can lead to random crashes in certain cases (depending on the code coverage and various paths taken by the code).

Preconditions

The affected functions would need to be called with NULL pointers.

Feasibility

Low probability, as in most cases in this application, the data is coming from the `G_io_apdu_buffer`, but still worth addressing for the overall robustness of the program.

Technical Details

In `apdu_pubkey.c:prompt_address_prepro()`, the input parameter "`element`" is dereferenced without NULL checks. In `protocol.c:is_block_valid()` and in `get_block_level()`, pointers are referenced without NULL check. In certain cases, there are `NULL` checks done outside the function. However, we feel it is much better to make it self contained in the function definition itself.

Remediation

In some cases these checks are done but outside the function, before calling the function. The trap with that approach is that the function may get called in multiple places and the programmer has to take care of doing the null check everywhere. Instead, doing it inside the function is a lot more safe.

Status

A new `check_null` inline function is introduced and if the input pointer is a NULL, an error is thrown. This function is used in various other functions before dereferencing the pointers.

Verification

Resolved. A review of the newly added function and the various places it has been used has been conducted.

Issue B: Incrementing the Void Pointer

Synopsis

A void pointer does not know how much is the size of the data it is pointing to by definition. Incrementing it is illegal in standard C.

Impact

Low, because common C compilers like gcc and clang increment by one byte and in this case, we happen to be pointing to a byte stream.

Technical Details

In `display.c convert_address()` function is defined as follows:

```
int convert_address(char *buff, uint32_t buff_size, void *raw_bytes, uint32_t
size);
```

The `raw_bytes` pointer is later used as follows:

```
…
      case 0x02: // Ed25519
            {
            // Already compressed
            blake2b(data.hash, sizeof(data.hash), raw_bytes + 1, size - 1,
NULL, 0);
```

```
            break;

            }
```

…

The blake2b internally typecasts the parameter into (uint8_t *). However, we are doing pointer arithmetic on void pointer at the time of calling the blake2b() function.

Remediation

The function can be called as follows:

```
blake2b(data.hash, sizeof(data.hash), (void *)((uint8_t *)raw_bytes + 1), size
- 1, NULL, 0);
```

Status

Much of the function in question has been rewritten into prompt_pubkey.c and the reported issue has been fixed.

Verification

Resolved. No further use of an increment of a void pointer observed.

Issue C: Signed Int Versus Unsigned Int Comparison

Synopsis

Comparing signed integers and unsigned integers has subtle portability issues.

Impact

Low.

Technical Details

In base58.c, certain variables are declared as type ssize_t and then compared with other variables of type size_t. While it is not a problem in the current context, it has subtle issues and has inconsistent behaviour because of C's integer promotion strategy as well as other platform specific behaviour, so it is best to avoid them as far as possible.

```
ssize_t i, j, high, zcount = 0;
```

Remediation

In the case of base58.c, declaring the offending variables as size_t seem to solve the problem.

```
size_t i, j, high, zcount = 0;
```

It may also be better to declare the loop variables inside the loop so that they are invalid outside the block, which helps in accidental changes to these variables.

Status

The suggestions noted in order to remediate the issue have been implemented in the audit-cleanup. branch.

Verification

Resolved. No further use of signed int vs unsigned int comparison observed.

Issue D: Incorrect Types for Loop Variables

Synopsis

C standard has specified a `type size_t` for loop variables (instead of using signed integers).

Impact

Low.

Technical Details

In almost every for loop used in the project, `size_t` is not used as the type of the loop variable. Use of `size_t` helps in maintainability, portability and readability of the code.

An example is in `paths.c:path_to_string()`.

```
for (uint32_t i = 0; i < path_length; i++) {
        …
        ...
}
```

Remediation

Change all loop variables to use `size_t`.

Status

The suggestions noted in order to remediate the issue have been implemented in the `audit-cleanup.` branch.

Verification

Resolved. The loop variables have all been changed to `size_t` and no further incorrect loop variable types observed.

Issue E: Functions with No Input Incorrectly Defined

Synopsis

C standard says that functions defined as "`XXX foo()`" (ignore the return type XXX for this discussion) can take "any" number of argument.

Impact

Low because almost all the places where it is used are in private functions. However if it is used in an exposed function, one can write arbitrary bytes into the stack and can try various exploits.

Feasibility

Hard.

Technical Details

An example is in `apdu_sign.c`

```
void sign_ok() {

      int tx = perform_signature(true);

      delay_send(tx);

}
```

`sign_ok` was not meant to take any inputs, but because of the way it is defined, it can take any number of inputs. There are a few instances in other files as well.

Remediation

Change functions that take no input arguments as in the following example:

```
void sign_ok(void) {

      int tx = perform_signature(true);

      delay_send(tx);

}
```

Status

The suggestions noted in order to remediate the issue have been implemented in the `audit-cleanup.` branch.

Verification

Resolved. All existing functions in the source code with an arity of zero changed to the suggested form.

Issue F: Use of Magic Numbers in the Code

Synopsis

Magic numbers make code hard to read and comprehend. There are many instances of the use of magic numbers in the code.

Impact

Low.

Technical Details

All the use of `THROW` macro are using various 16-bit codes. These code seem to come from the `APDU` response codes, however it is nice to have macros for these codes and use them instead. We understand that it is the responsibility of OS vendor (in this case Ledger) to provide these macros to application writers. However, until they provide them, defining and using these codes make the code much more readable and maintainable and incur zero runtime cost, e.g.:

`apdu_sign.c` has many instances of THROW(0x6B00). Similarly in other source files.

Mitigation

Define descriptive names for each of the codes used in the app in a header file and use those names.

Status

The audit-fixes branch has implemented the above suggestions. Each possible error code is defined as a C macro and used as arguments to the THROW macro. It is much easier to read the code now.

Verification

Resolved. All the magic numbers in the code have been changed to use the macros (`EXC_XXX_YYY`) defined in `apdu.h`.

Issue G: Support for Nano-S Running Older Firmware Versions

Synopsis

The app uses Level 8 APIs. Lower levels are not supported.

Impact

People running older firmware for their Nano S devices won't be able to get the Tezos application to work.

Technical Details

The Nano S SDK defines a macro in the file `include/os_apilevel.h` called `CX_APILEVEL` which is set to 8 in the latest release (for firmware 1.4.2). Certain OS system calls are different for API Level less than 8. However the tezos ledger app does not make these older API calls depending on the `CX_APILEVEL`, which would mean, people running older firmware on their hardware wallets won't be able to run the tezos app reliably.

Remediation

Make the right system call API via a compile time switch

```
#if CX_APILEVEL >= 8

 tx = cx_eddssa_sign(....);

#else

tx=cx_eddsa_sign(....);

#endif
```

Status

Older hardware versions are not going to be supported by the Tezos Ledger App as they are less secure as per Obsidian. A compile-time error is given if the app is compiled with the old API.

Verification

Confirmed Won't Fix . Following discussion with Obsidian, it has been agreed that backward compatibility will not be supported due to older versions of the hardware being less secure.

Suggestions

Suggestion 1: Better Build Documentation

Synopsis

The BUILDING.md file talks about the various build steps. However the cross compiler binaries have a number of other runtime dependencies without which, the code cannot be compiled.

Impact

Code cannot be compiled without complete installation steps.

Technical Details

On a typical GNU/Linux system (we used Debian GNU/Linux on amd64 platform as our development machine), a number of runtime dependencies for clang had to be installed to make the compiler work.

Mitigation

Perhaps adding a note on these runtime dependencies (We had to install libc6:armhf, zlib1g:armhf and a few more libraries to get clang to run).

Status

The readme file has been updated in the `audit-cleanup` branch is now much useful than the previous version.

Verification

Resolved. The document has been updated to include all the steps in the development, installation and upgrade of the firmware on the hardware device and use with the `tezos-client`.

Suggestion 2: Makefile: Turn on all Warnings

Synopsis

It is a good practice to compile with all the compiler warnings turned on and to make the code warning free.

Impact

Turning on the warnings almost always throws up some bugs in the code.

Technical Details

The Makefile CFLAGS does not turn on any warnings. But turning them on (`-Wall -Wextra`) throws up a number of warnings. It also shows a number of warnings in the SDK which we didn't investigate as it is outside the scope of this audit.

Mitigation

Add `-Wall` and `-Wextra` to the CFLAGS variable and recompile the code and fix the reported warnings as much as possible.

Status

CFLAGS is now enhanced with `-Wall` and `-Wextra` build flags.

Verification

Resolved. The Makefile now compiles code with all the warnings turned on.

Recommendations

Per our recommendation, the Issues and Suggestion stated in the initial report were addressed and followed up with a verification by the auditing team. We recommend that future audits be conducted on future development releases to ensure that any potential issues and vulnerabilities are identified, addressed and verified.

Appendix 1: Activity Log

These are notes from the reviewers about their activities during the code audit. They detail the approach and investigative activities undertaken. All issues found are listed in the report. This is just for the purposes of transparency and could be helpful for another auditor to understand the evaluation activities.

June 18, 2018:

Cloned the repo. Started reading the available documentation for BOLOS. Read a bit about the hardware (ST31 secure microcontroller + ARM SC000 series core). There is an sdk from Ledger for Nano-S and Blue, two different hardware from Ledger.

Started reading the ledger.readthedocs.io/en/2 documentation about the OS.

So, there is a master binary seed, from which everything is derived via the HD wallet scheme. 256 bits of random bits from True RNG. We take SHA256 of this random number, take last 8 bits of it and append it to the random number (let us call it r) to get n = r || (lsByte(sha256(r)). This 264 bit number is divided into 24 words of 11 bits each. Each of this 11-bit word is used as an index into a table of english words, so we get 24 english words, which can be written down somewhere. All private keys are generated deterministically from these words.

Followed the compilation steps, but getting an error “/lib/ld-linux-armhf.so.3: no such file or directory”.

June 19, 2018:

Managed to compile the code with the following steps:

1. Setup a shell script that sets up BOLOS_ENV and BOLOS_SDK
2. Download the clang and gcc referenced in the readme file.
3. dpkg --add-architecture armhf
4. apt-get update
5. Install libc6:armhf, zlib1g:armhf, libstdc++6:armhf and so on.. (install and then run ‘make’ and see which one clang is complaining about and then install that package.. Until make no longer complains)

Learning about the smartcard standards (APDU - request/response loop, this is the entry point of interaction with the app). Also all user actions go via apdu.

Looking at apdu_sign.c, apdu.c, main.c and boot.c. boot.c has the main, which calls app_main (defined in main.c). app_main() calls main_loop defined in apdu.c. app_main() installs handler functions for the supported user actions. Main_loop is doing the request/response loop for apdu commands.

There seem to be CX_APILEVEL (currently defined as 8 in the SDK). Certain APIs seem different before and after version 8. The app code seem to be written for API level 8. Should we support older APIs too for those running older version of firmware on their wallet devices?

THROW(XXXX). XXXX is a magic number. Where does it come from? Grepped through the SDK, didn’t find any definitions. Looks like these are standard codes for APDU response (documented here: https://www.eftlab.co.uk/index.php/site-map/knowledge-base/118-apdu-response-list). Would be nice if there are C macro definitions for these codes. They have zero runtime cost.

Checks various “==” vs “=” errors inside the conditions of “if” statements etc. Didn’t find any.

Found a few function definitions that use "()" to indicate "no input". This is wrong. This means "any number of inputs". It should be foo(void) if function foo takes no input.

Digs into APDU more. Found APDU command and response structure here: https://www.blackhat.com/presentations/bh-usa-08/Buetler/BH_US_08_Buetler_SmartCard_APDU_Analysis_V1_0_2.pdf Pretty useful.

So, we have only 7 different operations (or rather instructions in the APDU lingo).

1 -> authorize baking

2 -> get pubkey

3 -> prompt pub key

4 -> sign

5 -> sign unsafe

6 -> reset

7 -> exit

0 -> undefined

Do we rightly index the handler function array within array boundaries? Looks like yes. We "AND" instruction coming from APDU with INS_MASK (0x07). Good.

June 20, 2018:

Looks at apdu_pubkey.c. Looks okay. All for loops should use size_t for indices, that's the modern style..

Pokes around with Makefile. So, warnings are not turned on by default. Turns on -Wall and -Wextra. That finds a bunch of warnings, which are most likely bugs. Yet to look at them.

Looks at the debug/app.map file to see how buffers are laid out in memory, whether some "interesting" buffers are next to each other (for example - any buffers with private keys sitting next to display buffers..).

Looks at base58.c. Looks good so far. Loop variables can be initiated at the point of use, so that it is not valid beyond the block. Just a good practice that can reduce bugs.

apdu_pubkey.c:prompt_address_prepro() seem to be dereferencing the element pointer without checking for NULL.

June 21:

apdu_pubkey.c:prompt_address_prepro()

- The variable "display" should be a boolean ?
- Didn't fully grok the return values. OK, so this function is supposed to be used as a callback, if it returns null, then the display is not supposed to be redrawn.

Looks at ui.c

+1 to "#pragma once" in header files. Much less error prone and widely supported.

Base58.h, reset_screens.h, sign_screens.h are missing the "pragma once" directive or an equivalent include guard.

baking_auth.c:

blake2b-ref.c:

display.c

In convert_address() function: first call to blake2b function, raw_bytes is incremented. However raw_bytes is a pointer to void. So, this should be first cast to a concrete type and then incremented? Gcc (and clang) allows pointer arithmetic on void pointers. But it should be avoided as it is illegal in standard C.

paths.c

Loop variables should be size_t.

protocol.c

is_block_valid(), get_block_level():: blk dereferenced without null check.

base58.c:b58enc():

ssize_t i, j, high, zcount = 0; -> should be size_t. Otherwise, we will be doing comparisons between signed and unsigned numbers.

In display.c:convert_address(): The line's first parameter:

cx_hash_sha256(&data, sizeof(data) - sizeof(data.checksum), checksum, sizeof(checksum));

may need a type cast as "data" is an anonymous array, but cx_hash_sha256() takes a pointer to const unsigned char.

June 22, 2018:

Finishing up the report.

July 24, 2018:

Verification of the reported fixes in the `audit-cleanup` branch.