Zellic

Cosmos SDK Sign Mode Textual

Application Security Assessment

May 23, 2023

Prepared for:

Marko Baricevic

Cosmos Network

Prepared by:

William Bowling, Aaron Esau, and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Cosmos Network from May 2nd to May 23rd, 2023. During this engagement, Zellic reviewed Cosmos SDK Sign Mode Textual's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a rogue app show a different textual representation of a transaction than expected?
- Can the same signature be used to validate two transactions that can result in different chain states?
- Since C strings are null terminated, what impact will CBOR strings containing null bytes have?
- Can we craft a textual/JSON polyglot transaction since the signing mode is not a part of the transaction object that is signed?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Out-of-scope core Ledger device firmware
- Physical hardware attacks
- Cosmos network validators' CBOR verification (comparison) code, CBOR encoding libraries, or use of in-scope code.

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

Due to time constraints during this engagement, though we assessed all code in scope to some extent, we prioritised the C code as it appeared to present the most risk. We recommend further assessment of the Golang code to ensure its security.

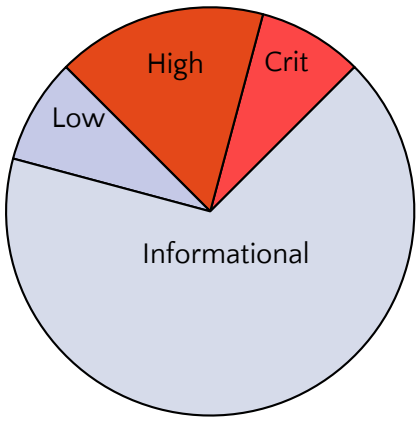
1.3 Results

During our assessment on the scoped Cosmos SDK Sign Mode Textual code, we discovered 12 findings. One critical issue was found. Of the other 11 findings, two were of high impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Cosmos Network's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	2
Medium	0
Low	1
Informational	8



2 Introduction

2.1 About Cosmos SDK Sign Mode Textual

Cosmos SDK Sign Mode Textual's `SIGN_MODE_TEXTUAL` is a new string-based sign mode that targets signing with hardware devices. It works by creating a representation of a transaction as a sequence of "screens", which are then encoded using CBOR (RFC 8949).

These screens have a title, content, and indentation level to represent data that has multiple levels and an "expert" flag, which can be used to conceal information that may not be relevant to nontechnical users, such as the hash of the raw bytes or the public key being used.

2.2 Methodology

During an application security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope. We aim to identify vulnerabilities related to memory corruption, input validation, and other security risks specific to low-level programming languages like C.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and code quality issues:

Memory management issues. Firmware applications often deal with limited resources and require efficient memory management. Zellic identifies potential vulnerabilities such as buffer overflows or other forms of memory corruption, memory leaks, and uninitialized variables that can lead to crashes, instability, remote code execution, or enable undefined behavior.

Input validation flaws. Ensuring the security of such applications involves validating and sanitizing all user inputs to prevent exploitation. Zellic examines the application's input validation mechanisms to detect vulnerabilities such as input injection, command injection, and format string vulnerabilities.

Cryptographic vulnerabilities. Such applications often employ cryptographic algorithms for secure communication and data protection. Zellic assesses the implementation of cryptographic functions to identify weaknesses such as insecure key generation, weak encryption algorithms, or improper usage of cryptographic primitives.

Interoperability and integration risks. Firmware applications often interact with other components or systems, and vulnerabilities can arise from these interactions. Zellic reviews the application's integration points and examines potential risks such as insecure data exchange, protocol vulnerabilities, or unauthorized access to external resources.

Code quality and maintainability. Zellic analyzes the overall codebase to identify areas for improvement in terms of code quality, adherence to coding standards, and maintainability. This includes identifying code smells, antipatterns, and potential performance optimizations.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Cosmos SDK Sign Mode Textual Programs

Repositories	https://github.com/cosmos/cosmos-sdk/
	https://github.com/cosmos/ledger-cosmos-go/
	https://github.com/cosmos/ledger-cosmos/
Versions	cosmos-sdk: `a8dceddad91cf0b00b20c4f7d4260583719a6403`
	ledger-cosmos-go: `d25ab48ebe5237a7553e9068b25eac88a7860f1a`
	ledger-cosmos: `e24a884915e2a275cc2b271096194fc2a0eab678`

Scopes	cosmos-sdk: `x/tx/signing/textual/** and x/auth/tx/textual.go`
	ledger-cosmos-go: `/**`
	ledger-cosmos: `App/src/**`
Types	C, Go

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of eight person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

William Bowling, Engineer
vakzz@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 2, 2023	Start of primary review period
May 4, 2023	Kick-off call
May 23, 2023	End of primary review period

3 Detailed Findings

3.1 Buffer overflow in Unicode expansion

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

The `tx_display_translation` function in tx_display.c performs a translation of characters from `src` into characters written to `dst`.

The purpose is to substitute ASCII control characters with their escape sequence equivalents and transform non-ASCII characters into their Unicode escape sequence equivalents. Any trailing whitespace or '@' characters in `src` are included in the result if the `dst` buffer is long enough. Once the translation is complete, the `dst` buffer is terminated with a null character.

The length of the `src` buffer is denoted by `srcLen`, and the length of the `dst` buffer is denoted by `dstLen`. The function constantly checks the bounds of the `dst` buffer by running the `ASSERT_PTR_BOUNDS` macro, which increments a `count` variable and compares it against `dstLen`:

```
#define ASSERT_PTR_BOUNDS(count, dstLen) \
    count++; \
    if(count > dstLen) { \
        return parser_transaction_too_big; \
    } \

// [...]

parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen) {
    // [...]
    uint8_t count = 0;
    // [...]
}
```

However, `count` is a uint8_t while `dstLen` is a uint16_t, meaning the pointer bounds

check will always pass if the lower eight bits of `dstLen` is 0 regardless of the value of the upper eight bits.

A consequence of this potential integer overflow is that Unicode expansions may overflow the `dst` buffer.

Note that there is no need to bypass any stack canary check; there is no reference to the `CHECK_APP_CANARY()` macro in the `tx_display_translation` or `parser_screenPrint` (which calls `tx_display_translation`) functions.

Impact

An attacker could potentially exploit the `dst` buffer overflow to execute arbitrary code and thereby manipulate the text displayed on the screen or sign arbitrary transactions.

In an exploit payload, for example, the single `'\xff'` byte will be expanded into the six-byte string “`\u00FF`”, which allows the attacker to quickly consume `count`.

The following message triggers a crash:

```
{1: [{2:
    ”ZZZZ\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ff
    \u00ff\u00ff\u00ff\u00ff\u00ff\u00ff\u00ffABCD”
}]}
```

It defines a single screen without any title, and the contents include a lot of bytes that each expand into longer sequences of bytes, until the buffer overflows. The `ABCD` string at the end ends up overwriting the `PC` register on the Ledger, making it try to execute code at address `0x44434240` (the last bit on ARM is used to signal ARM or Thumb mode, otherwise it would be ...`0x41`).

Do note that after our payload, the application will also write a single null byte. We are also limited to writing with ASCII values in the range `0x20` to `0x7F` as other bytes will be escaped to hex characters. However, it is possible to do a partial overwrite of the original PC value provided that the last byte is a null byte.

During initial testing, we demonstrated this bug in the Speculos emulator, which runs the target application in QEMU, which has significantly worse security than the Ledger device. Namely, it has an executable stack by default and no address randomization (PIE/PIC).

For example, consider the following APDU, which inserts the previously mentioned JSON message:

```
55020201f7a10181a10278f05a5a5a5ac3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3
bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bf
c3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3
bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bf
c3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3
bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bf
c3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bfc3bf41424344
```

Entering the APDU would cause the crash in Figure 1 (where the screenshot is from debugging the "Nano S" target).

Figure 2 shows the backtrace (keep in mind that many functions are inlined).

This demonstrates we have control of multiple register values, not just `PC`.

Note again that we do not control all bytes in the payload because non-ASCII bytes get expanded. The easiest exploit path here would be to create some valid transaction that contains, for example, a MEMO element at the end that triggers the overflow, making the app jump straight to the verification routine and immediately sign the transaction without any user input.

As long as the validator comes up with the same CBOR data as the Ledger app signed, starting from a given TX, this signature will be accepted. However, jumping to this area is not necessarily easy to do.

Some devices may support PIE/PIC, which complicates exploitation. When testing on real Ledger devices, we found that PIC address layout is static for a single application and even persists across reboots. Fortunately, the address depends on some unknowns such as the number of apps previously installed on the device, their sizes, and so forth.

Installing the same app over and over seemed to increase the PIC address in a deterministic way, but without any means of leaking this address, exploitation seems difficult. But an attacker only has to be lucky once.

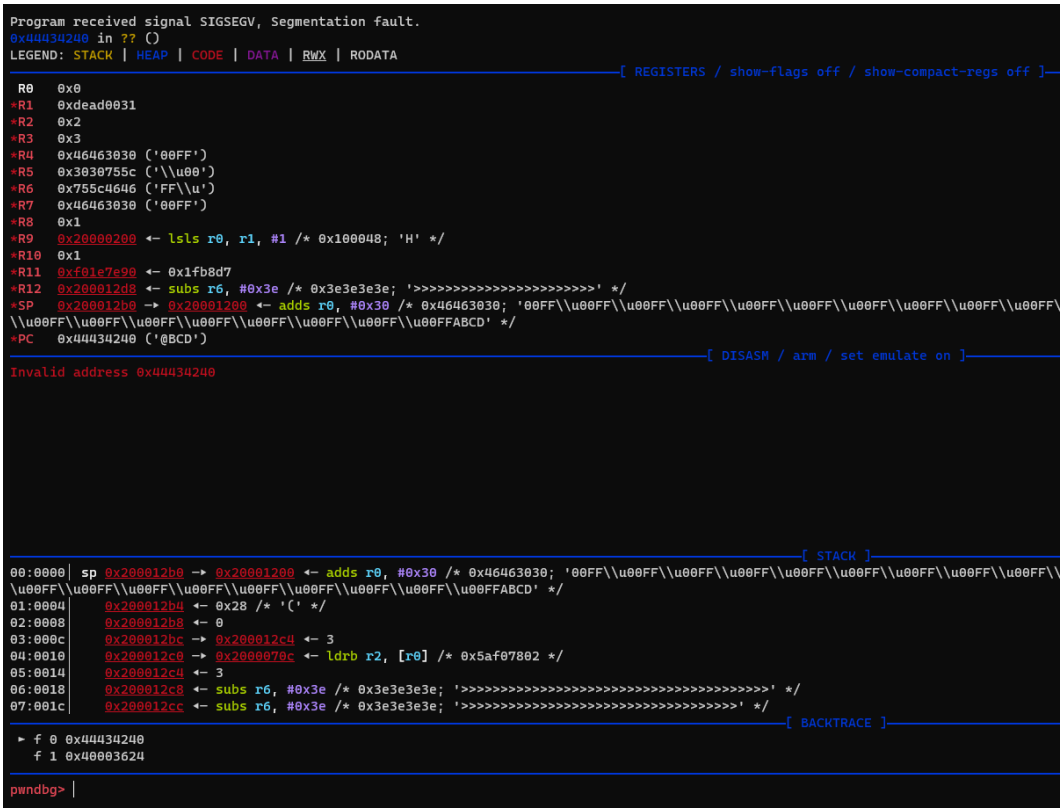


Figure 1: A crafted APDU causes a buffer overflow on a Nano S.

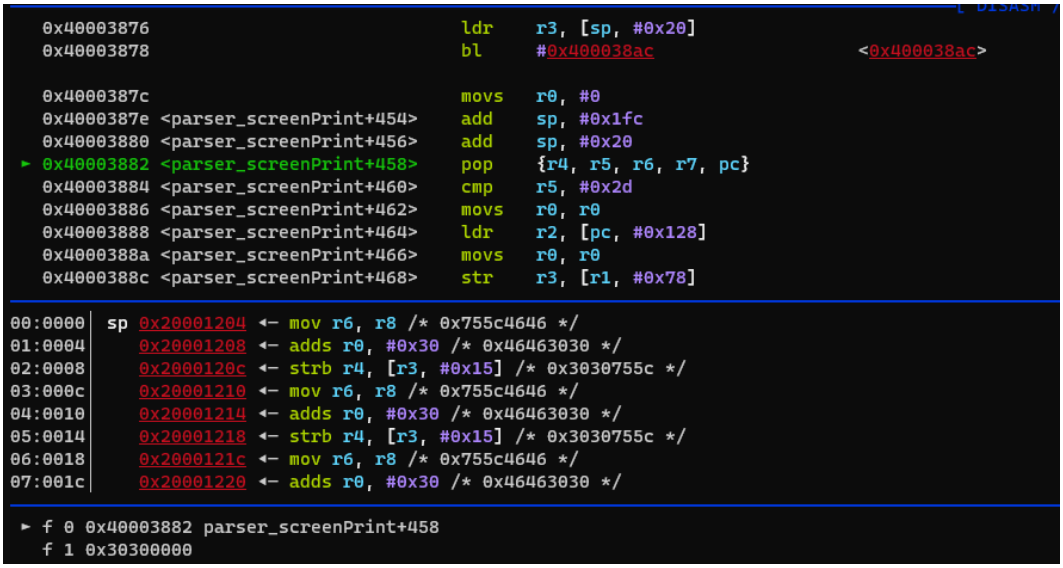


Figure 2: We obtain a backtrace from the buffer overflow shown in Figure 1.

Recommendations

Change the declaration of `count` to a uint16_t:

```
parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen) {
    // [...]
    uint8_t count = 0;
    uint16_t count = 0;
    // [...]
}
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `17d26659`.

3.2 Null-terminated strings enable trickery attacks

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `tx_display_translation` function copies the `src` buffer to `dst`, incrementing the pointer `p` each iteration, formatting bytes as desired.

However, since the `src` buffer can contain a null byte, the while loop may be stopped early:

```
parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen) {
    MEMZERO(dst, dstLen);
    char *p = src;
    uint8_t count = 0;
    uint8_t verified_bytes = 0;

    while (*p) {
        utf8_int32_t tmp_codepoint = 0;
        p = utf8codepoint(p, &tmp_codepoint);
        // [...]
    }

    /// [...]
}
```

Impact

Placing a null byte in a string that gets displayed may hide information.

For example, consider the following transaction:

```
data = {1:
    [{1: 'Chain id', 2: 'my-chain'},
     {1: 'Account number', 2: '1'},
     {1: 'Sequence', 2: '2'},
     {1: 'Address', 2: 'cosmos1ulav3hsenupswqfkw2y3sup5kgtqwnvqa8eyhs',
    4: True},
```

```
     {1: 'Public key', 2: '/cosmos.crypto.secp256k1.PubKey', 4: True},
     {2: 'PubKey object', 3: 1, 4: True},
     {1: 'Key', 2: '02EB DD7F E4FD EB76 DC8A 205E F65D 790C D30E 8A37
    5A5C 2528 EB3A 923A F1FB 4D79 4D', 3: 2, 4: True},
     {2: 'This transaction has 1 Message'},
     {1: 'Message (1/1)', 2: '/cosmos.bank.v1beta1.MsgSend', 3: 1}, {2:
    'MsgSend object', 3: 2},
     {1: 'From address', 2:
    'cosmos1ulav3hsenupswqfkw2y3sup5kgtqwnvqa8eyhs', 3: 3},
     {1: 'To address', 2:
    'cosmos1ejrf4cur2wy6kfurg9f2jppp2h3afe5h6pkh5t', 3: 3},
     {1: 'Amount', 2: '10 ATOM', 3: 3}, {2: 'End of Message'},
     {1: 'Memo', 2: 'GG\0I hereby declare war on Arstotzka!'},
     {1: 'Fees', 2: '0.002 ATOM'},
     {1: 'Gas limit', 2: "100'000", 4: True},
     {1: 'Hash of raw bytes', 2:
    '9c043290109c270b2ffa9f3c0fa55a090c0125ebef881f7da53978dbf93f7385',
    4: True}
    ]
}
```

The null byte in `Memo` would conceal the declaration of war from the country signing the transaction on the Ledger device as shown in Figure 3.

Figure 3: The declaration of war against Arstotzka is hidden on the Ledger device.

In general, important information may be concealed from the signer by inserting a null byte in a field that is displayed.

Recommendations

Instead of checking if `*p` is null, check that we have not consumed the entire `src` buffer:

```
parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen) {
    MEMZERO(dst, dstLen);
    char *p = src;
    uint8_t count = 0;
    uint8_t verified_bytes = 0;

    while (*p) {
    while (p < src + srcLen) {
        // [...]
    }

    // [...]
}
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `fb90358d`.

3.3 Some bytes are not displayed

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: Medium
- **Impact**: High

Description

When `tx_display_translation` is encoding the `src` buffer to `dst`, any bytes less than `0x0F` will be caught in the following branch:

```
if (tmp_codepoint < 0x0F) {
    for (size_t i = 0; i < array_length(ascii_substitutions); i++) {
        if ((char)tmp_codepoint == ascii_substitutions[i].ascii_code) {
            *dst++ = '\\';
            ASSERT_PTR_BOUNDS(count, dstLen);
            *dst++ = ascii_substitutions[i].str;
            ASSERT_PTR_BOUNDS(count, dstLen);
            break;
        }
    }
} // [...]
```

However, if the byte is not found in the following `ascii_substitutions` array, nothing will be written to the buffer:

```
static const ascii_subst_t ascii_substitutions[] = {
    {0x07, 'a'}, {0x08, 'b'}, {0x0C, 'f'},
    {0x0A, 'n'}, {0x0D, 'r'}, {0x09, 't'},
    {0x0B, 'v'}, {0x5C, '\\'},
};
```

Impact

Any `0x00`, `0x01`, `0x02`, `0x03`, `0x04`, `0x05`, `0x06`, or `0x0E` bytes in the `src` buffer will not be represented in the `dst` buffer, potentially misleading users into signing a transaction they do not expect.

Recommendations

If the for loop does not find the ASCII substitution character, output it in decimal in `\xNN` format:

```
if (tmp_codepoint < 0x0F) {
    uint8_t found = 1;
    for (size_t i = 0; i < array_length(ascii_substitutions); i++) {
    for (size_t i = 0; i < array_length(ascii_substitutions)
            || (found = false); i++) {
        if ((char)tmp_codepoint == ascii_substitutions[i].ascii_code) {
            *dst++ = '\\';
            ASSERT_PTR_BOUNDS(count, dstLen);
            *dst++ = ascii_substitutions[i].str;
            ASSERT_PTR_BOUNDS(count, dstLen);
            break;
        }
    }
    if (!found) {
        // Write out the value as a hex escape, \xNN
        count += 4;
        if (count > dstLen) {
            return parser_unexpected_value;
        }
        snprintf(dst, 4, "\\x%.02X", tmp_codepoint);
        dst += 4;
    }
} // [...]
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `fb90358d`.

3.4 Backslash characters are not escaped

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The `tx_display_translation` function is responsible for escaping bytes such as newlines. The following is a mapping of a byte to the suffix, which is appended after a '\' character:

```
static const ascii_subst_t ascii_substitutions[] = {
    {0x07, 'a'}, {0x08, 'b'}, {0x0C, 'f'},
    {0x0A, 'n'}, {0x0D, 'r'}, {0x09, 't'},
    {0x0B, 'v'}, {0x5C, '\\'},
};
```

The following code performs the escaping that is done using the `ascii_substitutions` array:

```
if (tmp_codepoint < 0x0F) {
    for (size_t i = 0; i < array_length(ascii_substitutions); i++) {
        if ((char)tmp_codepoint == ascii_substitutions[i].ascii_code) {
            *dst++ = '\\';
            ASSERT_PTR_BOUNDS(count, dstLen);
            *dst++ = ascii_substitutions[i].str;
            ASSERT_PTR_BOUNDS(count, dstLen);
            break;
        }
    }
// [...]
```

Because of the `if (tmp_codepoint < 0x0F)` condition, the `0x5C` byte is never substituted with "\\".

Impact

The backslash character ('\', ASCII `0x5C`) will never be escaped, meaning two different inputs can have the same canonical, textual representation.

For example, consider the following `data`:

```
{1: [ {1:"Chain id", 2: "lol\\u00FF\xff"} ]}
```

The display would look as shown in Figure 4.

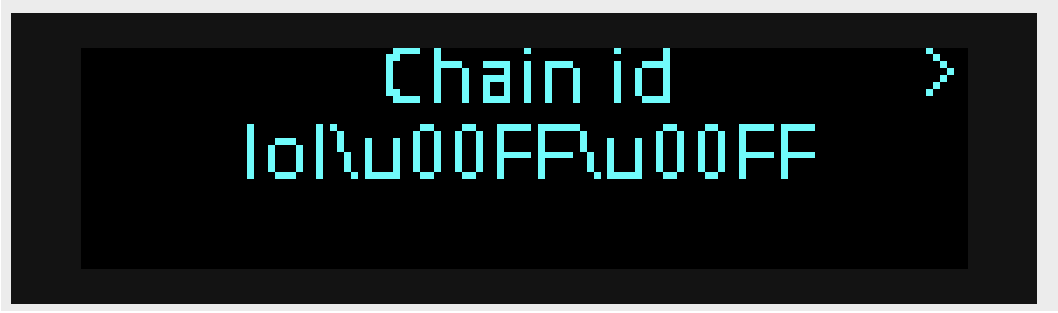


Figure 4: The backslash character is not properly escaped.

The "fake" and legitimately escaped strings are indistinguishable on the device.

Recommendations

Update the branch logic such that the `0x5C` byte is considered for substitution.

```
if (tmp_codepoint < 0x0F) {
if (tmp_codepoint < 0x0F || tmp_codepoint == 0x5C) {
    for (size_t i = 0; i < array_length(ascii_substitutions); i++) {
        // [...]
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit 17d26659.

3.5 Buffer overflows in ledger-zxlib

- **Target**: zxformat.h (ledger-cosmos dependency, ledger-zxlib)
- **Category**: Coding Mistakes
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: Informational

Description

Though not specifically in scope, while browsing the ledger-zxlib dependency's source tree, we observed the following potential bugs.

In the `pageStringHex` function, the `outValueLen` and `lastChunkLen` variables are uint16_t, meaning their maximum values are 65535 (`0xffff`):

```
__Z_INLINE void pageStringHex(char *outValue, uint16_t outValueLen,
                              const char *inValue, uint16_t inValueLen,
                              uint8_t pageIdx, uint8_t *pageCount) {
    // [...]
    const uint16_t lastChunkLen = (msgHexLen % outValueLen);
    // [...]

    if (pageIdx < *pageCount) {
        if (lastChunkLen > 0 && pageIdx == *pageCount - 1) {
            array_to_hexstr(outValue, outValueLen,
    (const uint8_t*)inValue+(pageIdx * (outValueLen/2)), lastChunkLen/2);
        } else {
            array_to_hexstr(outValue, outValueLen,
    (const uint8_t*)inValue+(pageIdx * (outValueLen/2)), outValueLen/2);
        }
    }
}
```

The last parameter of the `array_to_hexstr` function is `count`, which is a uint8_t, meaning the `lastChunkLen/2` and `outValueLen/2` arguments — both of which have potential maximum values of 32767 (`0xffff/2`) — will be cast to uint8_t, which can store a maximum of 255 (`0xff`).

Though cast truncation is possible here, it would likely not be exploitable since the `count` controls the number of bytes written to the `dst` buffer.

However, in `array_to_hexstr`, the following size check also contains an integer overflow bug:

```
__Z_INLINE uint32_t array_to_hexstr(char *dst, uint16_t dstLen,
    const uint8_t *src, uint8_t count) {
    MEMZERO(dst, dstLen);
    if (dstLen < (count * 2 + 1)) {
        return 0;
    }

    const char hexchars[] = "0123456789abcdef";
    for (uint8_t i = 0; i < count; i++, src++) {
        *dst++ = hexchars[*src >> 4u];
        *dst++ = hexchars[*src & 0x0Fu];
    }
    *dst = 0; // terminate string

    return (uint32_t) (count * 2);
}
```

Any `count` value greater than 127 ($(0xff-1)/2$) will result in the $count*2+1$ calculation overflowing, allowing the `dst` buffer length check to be bypassed and potentially enabling a `dst` buffer overflow.

Impact

The `pageStringHex` function does not appear to be used by ledger-cosmos, so it does not present any immediate risk.

Note that the `array_to_hexstr` function is used once, but has a hardcoded `count` argument that is not high enough to overflow, so the bugs will not be triggered in this case:

```
char buf[18] = {0};
array_to_hexstr(buf, sizeof(buf), (uint8_t *) &swapped, 8);
```

Recommendations

Add checks to prevent the cast truncation and integer overflow.

Remediation

A fix was added to the zxlib dependency in commit 72bed6ab.

3.6 Stack canary is chosen at compile time

- **Target**: ledger-cosmos
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The stack canary (checked using the `CHECK_APP_CANARY()` macro) is simply a hardcoded value `0xDEAD0031`:

```
#define APP_STACK_CANARY_MAGIC 0xDEAD0031
```

```
#pragma clang diagnostic push
#pragma ide diagnostic ignored "EndlessLoop"

void handle_stack_overflow() {
    zemu_log("!!!!!!!!!!!!!!!!!!!! CANARY TRIGGERED!!! STACK OVERFLOW
    DETECTED\n");
#if defined (TARGET_NANOS) || defined(TARGET_NANOX)
    || defined(TARGET_NANOS2)
    io_seproxyhal_se_reset();
#else
    while (1);
#endif
}

#pragma clang diagnostic pop

__Z_UNUSED void check_app_canary() {
#if defined (TARGET_NANOS) || defined(TARGET_NANOX)
    || defined(TARGET_NANOS2)
    if (app_stack_canary ≠ APP_STACK_CANARY_MAGIC)
    handle_stack_overflow();
#endif
}

// [...]
```

An attacker can predict the value and potentially exploit buffer overflow vulnerabilities, bypassing this check.

Impact

While the canary may help detect accidental buffer overflows, it provides little mitigation against intentional buffer overflow exploits.

Recommendations

Consider choosing a random stack canary at runtime for additional safety.

Remediation

This issue has been acknowledged by Cosmos Network.

3.7 Pointer bounds assertion after write leads to buffer overflow

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `ASSERT_PTR_BOUNDS` macro increments `count` then checks that the count is within the bounds of the buffer:

```
#define ASSERT_PTR_BOUNDS(count, dstLen) \
    count++; \
    if(count > dstLen) { \
        return parser_transaction_too_big; \
    } \
```

However, the assertion is always placed after writing a byte (i.e., the code writes before checking the bounds), potentially causing a buffer overflow.

Impact

A one-byte buffer overflow would likely be unexploitable. Ideally the `tx_display_translation` function would return the `parser_transaction_too_big` error and cause the destination buffer to be unused, but the return value is unused per Finding 3.10.

Recommendations

Check that the buffer is large enough (that the pointer is in bounds) before writing each byte.

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `17d26659`.

3.8 Exception handler variables missing `volatile` keyword

- **Target**: crypto.c, apdu_handler.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Per Ledger, one of the common pitfalls is in the exception handling. Their recommendation is

> When modifying variables within a `try` / `catch` / `finally` context, always declare those variables `volatile`. This will prevent the compiler from making invalid assumptions when optimizing your code because it doesn't understand how our exception model works.

Ledger has implemented exception handling through OS support, using `os_longjmp` to jump to magic addresses that the OS intercepts and translates. This is not very transparent to an optimizing compiler and could turn into a compile-time mistake if the code is not handling it.

The examples mentioned in the troubleshooting guide are for changing the status of an error object during the `catch` context, then emitting it during `finally` or using it later.

The same code pattern emerges in a few places in the ledger-cosmos repository. For example,

- `crypto.c` variable `err`:

```
zxerr_t crypto_extractPublicKey(const uint32_t
    path[HDPATH_LEN_DEFAULT], uint8_t *pubKey, uint16_t pubKeyLen)
    {
    cx_ecfp_public_key_t cx_publicKey;
    cx_ecfp_private_key_t cx_privateKey;
    uint8_t privateKeyData[32];

    if (pubKeyLen < PK_LEN_SECP256K1) {
        return zxerr_invalid_crypto_settings;
    }

    zxerr_t err = zxerr_ok;
```

```
    BEGIN_TRY
    {
        TRY {
            os_perso_derive_node_bip32(CX_CURVE_256K1,
                                       path,
                                       HDPATH_LEN_DEFAULT,
                                       privateKeyData, NULL);

            cx_ecfp_init_private_key(CX_CURVE_256K1, privateKeyData,
    32, &cx_privateKey);
            cx_ecfp_init_public_key(CX_CURVE_256K1, NULL, 0,
    &cx_publicKey);
            cx_ecfp_generate_pair(CX_CURVE_256K1, &cx_publicKey,
    &cx_privateKey, 1);
        }
        CATCH_OTHER(e) {
            err = zxerr_ledger_api_error;
        }
        FINALLY {
            MEMZERO(&cx_privateKey, sizeof(cx_privateKey));
            MEMZERO(privateKeyData, 32);
        }
    }
    END_TRY;

    if (err ≠ zxerr_ok) {
        return err;
    }
    // More code here
```

- `apdu_handler.c` variable `sw`:

```
void handleApdu(volatile uint32_t *flags, volatile uint32_t *tx,
    uint32_t rx) {
    uint16_t sw = 0;

    BEGIN_TRY
    {
        TRY
        {
            // ...
```

```
        }
        CATCH(EXCEPTION_IO_RESET)
        {
            THROW(EXCEPTION_IO_RESET);
        }
        CATCH_OTHER(e)
        {
            switch (e & 0xF000) {
                case 0x6000:
                case APDU_CODE_OK:
                    sw = e;
                    break;
                default:
                    sw = 0x6800 | (e & 0x7FF);
                    break;
            }
            G_io_apdu_buffer[*tx] = sw >> 8;
            G_io_apdu_buffer[*tx + 1] = sw;
            *tx += 2;
        }
        FINALLY
        {
        }
    }
    END_TRY;
}
```

Impact

With just minor optimizations enabled, the compiler can be confused and optimize away variable modifications that do not seem to have any clear side effects. These bugs usually surface up near the end of the development cycle, when compiler optimizations are enabled to save on memory/flash footprint. The result could be that an actual error status is masked, and the application continues on like if it was successful. In the case of the `crypto.c` example, this would lead to the wrong public key being calculated in `crypto_fillAddress()`.

Do also note that the entire APDU handler runs everything inside a big exception handler loop, which means it can return there at any point, and great care has to be taken when accessing globals there. An example where it could return early is in `crypto.c` where the function `cx_hash_sha256()` is called outside of a try catch. It is recom-

mended to use a function like `cx_hash_no_throw` instead there to avoid a very deep return back to the APDU handler.

Recommendations

Mark variables that can be changed inside exception handlers with the `volatile` keyword. Use functions like `cx_hash_no_throw()`, then return gracefully on error, or wrap error-throwing functions like `cx_hash_sha256()` in TRY/EXCEPT blocks where used.

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `fb90358d`.

3.9 Incorrect size check when encoding Unicode

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following code is part of the `tx_display_translation` function and converts a sequence of Unicode codepoints into a string of escaped UTF-8 characters:

```
*dst++ = '\\';
ASSERT_PTR_BOUNDS(count, dstLen);

uint8_t bytes_to_print = 8;
int32_t swapped = ZX_SWAP(tmp_codepoint);
if (tmp_codepoint > 0xFFFF) {
    *dst++ = 'U';
    ASSERT_PTR_BOUNDS(count, dstLen);
} else {
    *dst++ = 'u';
    ASSERT_PTR_BOUNDS(count, dstLen);
    bytes_to_print = 4;
    swapped = (swapped >> 16) & 0xFFFF;
}

if (dstLen < bytes_to_print) {
    return parser_unexpected_value;
}

char buf[18] = {0};
array_to_hexstr(buf, sizeof(buf), (uint8_t *) &swapped, 8);
for (int i = 0; i < bytes_to_print; i++) {
    *dst++ = (buf[i] ≥ 'a' && buf[i] ≤ 'z') ? (buf[i] - 32) : buf[i];
    ASSERT_PTR_BOUNDS(count, dstLen);
}
```

The following size check does not take into account the number of bytes already written to the `dst` buffer:

```
if (dstLen < bytes_to_print) {
    return parser_unexpected_value;
}
```

Impact

The following line in the for loop when copying the `buf` buffer to the `dst` buffer would catch a buffer overflow:

```
ASSERT_PTR_BOUNDS(count, dstLen);
```

So, the buffer overflow would be unexploitable.

Recommendations

We recommend changing the size check to account for the number of bytes already written to the buffer:

```
if (dstLen < bytes_to_print) {
if (dstLen < bytes_to_print + count) {
    return parser_unexpected_value;
}
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `5a7c3cfe`.

3.10 Return value of `tx_display_translation` is ignored

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The return value of `tx_display_translation` is ignored. The function returns a `parser_error_t` if any abnormal behavior occurs:

```
parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen);
```

Impact

Errors may not be reported.

Recommendations

Catch the return errors, if any, and handle them as desired.

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit 17d26659.

3.11 Missing pointer bounds checks in `tx_display_translation`

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following code is missing pointer bounds checks when writing the last two bytes:

```
if (src[srcLen - 1] == ' ' || src[srcLen - 1] == '@') {
    if (src[dstLen - 1] + 1 > dstLen) {
        return parser_unexpected_value;
    }
    *dst++ = '@';
}

// Terminate string
*dst = 0;
```

Also, the check inside the if statement seems to not do anything useful. It checks if the ASCII value of the last byte is at least 1 larger than the length of the destination buffer, and errors if it is. This is likely a coding mistake.

Impact

There is a potential for a two-byte buffer overflow. However, the bytes are not fully controlled, and it is likely unexploitable.

Recommendations

Add pointer bounds assertions:

```
if (src[srcLen - 1] == ' ' || src[srcLen - 1] == '@') {
    if (src[dstLen - 1] + 1 > dstLen) {
        return parser_unexpected_value;
    }
    ASSERT_PTR_BOUNDS(count, dstLen);
    *dst++ = '@';
}
```

```
// Terminate string
ASSERT_PTR_BOUNDS(count, dstLen);
*dst = 0;
```

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `5a7c3cfe`.

3.12 Out-of-bounds read in `tx_display_translation`

- **Target**: tx_display.c (ledger-cosmos)
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `swapped` variable is defined as a 32-bit signed integer:

```
int32_t swapped = ZX_SWAP(tmp_codepoint);
```

However, the following line of code reads 8 bytes starting at `swapped` because the last argument (`count`) is 8:

```
array_to_hexstr(buf, sizeof(buf), (uint8_t *) &swapped, 8);
```

Impact

The four bytes after `swapped` may be leaked in the `dst` buffer.

Recommendations

Change `count` to 4, or change `swapped` to an int64_t.

Remediation

This issue has been acknowledged by Cosmos Network, and a fix was implemented in commit `17d26659`.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Documentation bugs

We observed the following errors in the in-code documentation.

Documentation for `GET_VERSION` response incorrect

The documentation for the `GET_VERSION` response in docs/APDUSPEC.md is as follows:

Field	Type	Content	Note
CLA	byte (1)	Test mode	0xFF means test mode is enabled.
MAJOR	byte (1)	Version major	
MINOR	byte (1)	Version minor	
PATCH	byte (1)	Version patch	
LOCKED	byte (1)	Device is locked	
SW1-SW2	byte (2)	Return code	See list of return codes.

The table does not mention that there are four bytes copied from `TARGET_ID` per the code:

```
__Z_INLINE void handle_getversion(volatile uint32_t *flags,
    volatile uint32_t *tx, uint32_t rx) {
#ifdef DEBUG
    G_io_apdu_buffer[0] = 0xFF;
#else
    G_io_apdu_buffer[0] = 0;
#endif
    G_io_apdu_buffer[1] = LEDGER_MAJOR_VERSION;
    G_io_apdu_buffer[2] = LEDGER_MINOR_VERSION;
    G_io_apdu_buffer[3] = LEDGER_PATCH_VERSION;
```

```
    G_io_apdu_buffer[4] = !IS_UX_ALLOWED;

    G_io_apdu_buffer[5] = (TARGET_ID >> 24) & 0xFF;
    G_io_apdu_buffer[6] = (TARGET_ID >> 16) & 0xFF;
    G_io_apdu_buffer[7] = (TARGET_ID >> 8) & 0xFF;
    G_io_apdu_buffer[8] = (TARGET_ID >> 0) & 0xFF;

    *tx += 9;
    THROW(APDU_CODE_OK);
}
```

4.2 Missing `addr_to_textual` definition

Note that the `addr_to_textual` function below — although declared in addr.h — is never defined:

```
zxerr_t addr_to_textual(char *s, uint16_t max, const char *text,
    uint16_t textLen);
```

4.3 Unused `verified_bytes` variable

In tx_display.c's `tx_display_translation` function, there is a `verified_bytes` variable that increments each iteration of the while loop:

```
parser_error_t tx_display_translation(char *dst, uint16_t dstLen,
    char *src, uint16_t srcLen) {
    // [...]
    uint8_t verified_bytes = 0;

    while (*p) {
        // [...]
        verified_bytes ++;
    }

    // [...]
}
```

The `verified_bytes` variable is not used after the loop. We assume this code is incomplete, but the count for the number of bytes is incorrect anyway; in some of the

if branches in the code, multiple bytes may be written per iteration.

5 Audit Results

During our assessment on the scoped Cosmos SDK Sign Mode Textual programs, we discovered 12 findings. One critical issue was found. Two were of high impact, one was of low impact, and the remaining findings were informational in nature.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.