Pentest-Report Nitrokey Storage Firmware 05.2015

Cure53, Dr.-Ing. Mario Heiderich, Jann Horn, Nikolai Krein

Index

Introduction

“Nitrokey is an USB key to enable highly secure encryption and signing of emails and data, as well as login to the Web, networks and computers. Other than ordinary software solutions, the secret keys are always stored securely inside the Nitrokey. Their extraction is impossible which makes Nitrokey immune to computer viruses and Trojan horses. The user-chosen PIN and the tamper-proof smart card protect in case of loss and theft. Hardware and software are both available as Open Source to allow verifying the security and integration with other applications.”

From https://www.nitrokey.com/introduction

This penetration test against the Nitrokey Storage firmware, as well as the Nitrokey desktop app, was performed by a team of three penetration-testers and took eleven days in total to complete. The test is part of a larger series of security assessments. In later phases, security-focused assignments will include tests against the hardware itself, alongside detailed look into other models of the Nitrokey and its accompanying applications and tools. The main scope of a particular assessment covered by this report

is the Nitrokey Storage firmware openly available on GitHub.[1] The majority of code is composed in the C and C++ languages. The test was performed as a pure source code audit, hence the software was neither compiled nor run on any of the Nitrokey hardware. As already mentioned, the actual hardware testing and integration tests are scheduled to be performed at a later stage of the auditing process.

The test was carried out in reference with a threat model delivered by the Nitrokey Team. Said threat model essentially specifies the attack scenarios which the tested soft- and hardware seeks to protect against. It reads as follows:

- Malware trying to steal/export private keys while the device is being used on the compromised computer.
- An attacker stealing or finding the (switched off) device and trying to tamper the device in order to get access to the private keys or encrypted mass data. This includes sophisticated physical attacks with laboratory equipment and (digital) brute force attacks. ("offline")
- Nitrokey protects against Malware attempting to program malicious firmware onto the stick by instrumenting an alleged tamper-proof smart card.

The following security assumptions have been added to complete the threat model:

- The Admin PIN and Firmware Password are only used on secure computers for administrative purposes.
- The User PIN can be used on the potentially compromised computers.

Keep in mind the specificity of this basic threat model and the set of security assumptions tied to it. Those are subject to this particular test only and might therefore be changed, altered or extended for the upcoming hardware audit and/or any additional firmware and application security checks.

In this document we report on the test findings which scope ten vulnerabilities and four general weaknesses and recommendations overall. None of the identified issues were classified to be of critical severity. At the same time four issues were marked with severity “High“, as the underlying vulnerabilities would allow for a compromise of the Nitrokey threat model used for this source code audit.

[1] https://github.com/Nitrokey/nitrokey-storage-firmware

Scope

- **Nitrokey Storage Firmware**
  - https://github.com/Nitrokey/nitrokey-storage-firmware
- **Nitrokey App**
  - https://github.com/Nitrokey/nitrokey-app

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact, which is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier *(*e.g. *NK-01-00X)* for the purpose of facilitating any future follow-up correspondence.

NK-01-004 Block number is used as IV in CBC mode *(Medium)*

The method `STICK20_mci_aes()` encrypts in CBC mode[2], with `BlockNr_u32` repeated four times as IV.[3] This allows an attacker to recognize a certain plaintext pattern in the ciphertext: If `(plaintext of first 128 bytes of block A XOR plaintext of first 128 bytes of block B) = repeat(chunkNumberA XOR chunkNumberB)`, the first ciphertext blocks of the blocks will arrive at an equal result value.

Depending on the plaintext's structure, this problem may not only aid the reconstruction of its shape, but could also assist an adversary who has gained access to the encrypted flash storage and then manages to recognize watermarks that intentionally contain the pattern.

It is recommended to encrypt or hash the chunk number. An encrypted or hashed chunk number as IV should be used instead.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

[2] http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_.28CBC.29
[3] http://en.wikipedia.org/wiki/Initialization_vector

NK-01-005 Out-of-Bounds Read in ConvertMatrixDataToPassword *(Low)*

In the file *nitrokey-storage-firmware-master/src/USER_INTERFACE/html_io.c*[4], the method `HID_ExcuteCmd` uses the `HID_String_au8` buffer with a size of 50 to determine the current action. If the first character is "M", the user password gets converted from a matrix form. However, the method `ConvertMatrixDataToPassword()` uses a statically allocated `MatrixColumsUserPW_au8` stack buffer with size 20, while the conversion loop is bounded by the string length of `HID_String_au8`, thus resulting in an out-of-bounds memory access when the values of `MatrixColumsUserPW_au8` and `PWM_Password-Matrix_as8` are read.

It is recommended to ensure that the conversion loop is bounded by the size of `MatrixColumsUserPW_au8` instead of setting `n` to `strlen(MatrixData_au8)`.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-006 cmd_user_authenticate does not clear Password from RAM *(Low)*

Whenever the Smart Card is queried to verify a password with, for example, the method `LA_OpenPGP_V20_Verify()`, the caller has to make sure that the password is cleanly wiped from the RAM.[5] In the file *report_protocol.c* when a `CMD_USER_AUTHENTICATE-Command` is invoked, `cmd_user_authenticate()` uses the `user_password` buffer for authentication. However after the method `LA_OpenPGP_V20_Test_SendUserPW2()` returns, this critical memory area is left untouched even if the password was correct, thus letting it remain inside the RAM.

It is recommended to memset this buffer to 0, similar to `cmd_first_authenticate()`.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-007 OTP commands can be used without authorization *(High)*

There are certain intended steps for the process of plugging-in the device in order to be able to generate OTP codes when OTP[6] access is password-protected. They seem to include the following: (1) providing the `user_password` and a new `temp_user_password` to `CMD_USER_AUTHENTICATE`, (2) providing the `temp_user_password` and `authorized_user_crc` (the CRC32 checksum of the following request) to `CMD_USER_AUTHORIZE`, thereby (3) allowing the following request to be carried out. Finally, (4) a request to generate an OTP code with `CMD_GET_CODE` is to be sent and authorized based on `authorized_user_crc`. Consequently, the OTP code from the device is received..

[4] https://github.com/Nitrokey/nitrokey-storage-firmware/blob/master/src/USER_INTERFACE/html_io.c
[5] http://en.wikipedia.org/wiki/Random-access_memory
[6] http://en.wikipedia.org/wiki/One-time_password

However, the code path for `CMD_GET_CODE` does not verify that `authorized_user_crc` has already been set, meaning that if an attacker skips all steps before `CMD_GET_CODE`, he can pass the authorization check as long as he can construct 64 bytes with a CRC32 checksum of *0xFFFFFFFF*. The latter is obviously trivial.

It is recommended to send `temp_user_password` with every report that requires it and remove the CRC32 code.[7]

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-008 OTP can be unlocked by replacing Smart Card *(High)*

In order to be granted access to the OTP functions, the user has to authenticate to the Nitrokey via `CMD_USER_AUTHENTICATE`. This command merely asks the Smart Card to verify the correctness of the password, subsequently granting access. By replacing the Smart Card with a device that reports a successful verification for all requests, it is therefore possible to bypass the authentication for OTP commands.

Since a possibility of replacing the Smart Card is even advertised as a feature,[8] this attack would require an adversary to perform no complex hardware attacks.

Note: It was agreed with the maintainers of the project that this issue isn't covered by the currently communicated threat model and will therefore be closed as 'wontfix' for now.

NK-01-009 Passwords are encrypted in ECB mode *(Low)*

The method `AES_StorageKeyEncryption()`, which encrypts and decrypts data in ECB mode, is used in `PWS_ReadSlot()` to decrypt a `typePasswordSafeSlot_st`. This means that if two stored passwords are, for instance, identical from the fourth character onwards (which could also point to passwords that are only four characters long), or, alternatively, if two login names have a common part, an attacker could easily detect that.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-010 GetRandomNumber_u32 mixes randomness improperly *(Medium)*

The method `GetRandomNumber_u32()` contains code that is designed to provide a small amount of security even if the Smart Card does not return proper random numbers. However, this code actually weakens the generated randomness.

It is true that random data cannot be made less random by means of XOR-ing[9] it with independent data. Conversely, in this case the first invocation of

[7] http://en.wikipedia.org/wiki/Cyclic_redundancy_check
[8] https://www.nitrokey.com/shop/de/home/12-crypto-stick-storage-beta.html
[9] http://en.wikipedia.org/wiki/Exclusive_or

`GetRandomNumber_u32()`, `Data_pu8[0]` is fed into `srand()` together with a time stamp. Afterwards, the output of `rand()` is XOR-ed together with `Data_pu8[0]`. This weakens the randomness of the first `Data_pu8[0]` slightly, meaning that a quick test indicates that if the attacker knows the time at which `GetRandomNumber_u32()` was invoked, he can reduce the entropy of the result by about three bits. However, this depends on the knowledge of the precise time of the key generation.

It is recommended to let the Smart Card generate a separate byte that is only used for the `srand()` call.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-013 Encryption of uninitialized memory in HV_WriteSlot_u8 *(Medium)*

The method `HV_WriteSlot_u8()` takes a `HiddenVolumeKeySlot_tst` (size 44 bytes), copies it into a buffer with size `HV_SLOT_SIZE (64 bytes)`, encrypts the whole buffer in ECB mode, and writes the result to flash storage. The remaining 20 bytes of the temporary buffer are not initialized, meaning that the buffer will contain uninitialized stack data. If the last 16 bytes (AES block size) of the uninitialized data are the same for two runs of the method `HV_WriteSlot_u8()`, this will be visible in the ciphertext. This is due to the usage of the ECB since the corresponding ciphertext blocks will be the same. Therefore, an adversary with access to the internal Flash storage could exploit this issue to reveal the existence of hidden volumes.

It is recommended to fill the unused area with random data and consider using AES in a different mode, such as CBC or CTR.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-014 Security Bit is bound to Firmware Updates *(High)*

The Nitrokey has the Security Bit set to 0 by default. The user can set it to 1 by consulting the “Lock stick hardware” menu entry. However, introducing this alternation simultaneously disables firmware updates, which is why a large majority of users will likely not set it.

When the Security Bit is 0 anyone who can talk to the bootloader can selectively overwrite program code without clearing secrets from flash memory or extracting secrets from flash memory directly with the use of the `READ` and `SAVEBUFFER` BatchISP commands.[10] To gain access to the bootloader, an attacker who can talk to the Nitrokey additionally needs one of the following:

- An admin password which gives him a capacity to enable a stick update through `STICK20_CMD_ENABLE_FIRMWARE_UPDATE`

[10] http://www.avrfreaks.net/forum/batchisp-uc3-chips-demystified

- An ability to replace the Smart Card. Since the admin password check is performed on the Smart Card, it also allows an attacker to pass the password check. Although this requires physical access, it does not require a sophisticated attack because the Nitrokey intentionally allows the user to replace the Smart Card.

By carrying out the above described attack an adversary would gain access to all OTP secrets stored on the Nitrokey. In addition, it would signify gaining an ability to analyze other parts of the flash memory, which would be required for hidden volume detection attacks. Moreover, he could replace the firmware of the Nitrokey without alerting the user due to the fact that the secrets have been cleared.

It is recommended to always set the Security Bit to 1. A firmware update would still be possible afterwards as the Security Bit merely enforces that all non-bootloader flash memory is cleared using `ERASE F` prior to allowing all normal bootloader commands[11] (which `StartApp.bat` seems to be doing already). If it should remain possible to disable firmware updates, it is recommended to use a separate variable in flash memory to store this particular setting.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-015 Admin Check can be bypassed by resetting Smart Card *(High)*

The Nitrokey allows an unauthenticated software-only attacker (e.g. a machine into which the user only plugged his Nitrokey to use the OTP function without entering any PIN) to send arbitrary commands to the Smart Card via CCID.[12] This happens through the following call path:

```
1. USB_CCID_GetDataFromUSB()
2. USB_to_CRD_DispatchUSBMessage_v()
3. PC_to_RDR_XfrBlock_u8()
4. CCID_XfrBlock_u8()
5. ISO7816_T1_DirectXfr()
6. ISO7816_SendString()
```

The finally called method ends up sending a fully attacker-controlled string to the Smart Card. This lets the attacker factory-reset the Smart Card without any authentication. After resetting the card, the attacker can use the default admin PIN to authenticate to the Nitrokey, activate firmware update mode, dump all flash memory of the Nitrokey as described in NK-01-014 and upload a rogue firmware. Upon insertion of the Nitrokey into the victim's main computer, the malicious firmware can then expose a FAT filesystem with a malicious executable to the PC and send a keystroke sequence like the following:

```
[Win]+R, "E:\backdoor.exe", [Enter]
```

[11] http://www.atmel.com/images/doc7745.pdf
[12] http://en.wikipedia.org/wiki/CCID_%28protocol%29

This would compromise the victim's machine. Note that if the drive letter is not known, an attacker can brute-force the drive letter. The victim's operating system can be detected by observing disk access patterns or may simply be bruteforced.

It is recommended to either prevent factory resets through CCID (by whitelisting known-safe ISO 7816 commands[13]) or to verify that the Smart Card has not been reset. This would require asking it to decrypt a challenge ciphertext stored on the Nitrokey directly after the Smart Card verifies a PIN.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-016 Out-of-Bounds Read in CCID Handling *(Medium)*

CCID commands are processed in the specific manner. First, the method `USB_CCID_GetDataFromUSB()` reads data from the PC into `USB_data`, a buffer with size 270 bytes (`CCID_MAX_XFER_LENGTH`). `USB_to_CRD_DispatchUSBMessage_v()`. Secondly, it extracts the 16-bit value `CCID_datalen` (exact maximum due to "signed-ness" of the chars: $(2^7\text{-}1)*256 + (2^7\text{-}1) = 0x7f7f$) and the message type from the buffer. If the message type is `PC_TO_RDR_XFRBLOCK`, the buffer and the extracted length are then passed to `PC_to_RDR_XfrBlock_u8()`, which checks that `CCID_datalen` isn't zero (but performs no further checks on the length) and passes control to `CCID_XfrBlock_u8()`.

This method extracts the field's length again as `XfrLenght_s32`, then passes the unvalidated length to `ISO7816_T1_DirectXfr()`. The latter uses it as the buffer length for `ISO7816_SendString()`, which in turn sends the buffer out to the Smart Card.

All in all, this means that an attacker with an ability to replace the Smart Card with a malicious device can read large parts of the Nitrokey's RAM. In the current build, `USB_CCID_data_st.USB_data` is at the address 0xaf8, meaning that memory up to (exclusive) 0x8a77 can be disclosed. This includes some statically allocated buffers, as well as 0x29df bytes of heap memory, which is mostly occupied by task stacks started by `xTaskCreate()`. As all four tasks (i.e. the internal work task, the USB task, the mass storage task and the CCID task) started in `main()`, they all use stacks of 1024 bytes or less, which makes them fit into 0x1000 bytes of heap memory.

This allows an attacker with physical access to the Nitrokey to, for example, dump the OTP secrets from the device if authentication for OTP access is disabled or if the attacker can somehow circumvent the authentication. In order to do this, the attacker could e.g. dump the OTP secret from the USB thread's stack, specifically from the buffer `buffer` in the stack frame of `hmac_sha1()`. Directly accessing the static buffer containing the secret remains impossible because `USB_CCID_data_st` is behind it.

[13] http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4.aspx

It is recommended to perform bounding checks on all length fields that have been received from the outside. Further, as a non-security fix, it is recommended to cast the characters from `USB_data` to `unsigned char` prior to combining them into a length value employing addition.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid attackers in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

NK-01-001 One-Byte Buffer Overflow in HTML_CheckInput() *(Low)*

In the file *nitrokey-storage-firmware/src/USER_INTERFACE/html_io.c*[14], the function `HTML_CheckRamDisk()` allocates a buffer `Text_u8` with size `HTML_INPUTBUFFER_SIZE` on the stack, then passes a pointer to that buffer to `HTML_CheckInput()`.

This function then reads up to `HTML_INPUTBUFFER_SIZE` bytes into the buffer and appends a null-byte as a terminator. This means that if the call to `read()` already reads `HTML_INPUTBUFFER_SIZE` bytes, the terminating nullbyte is written beyond the end of the allocated buffer.

It is recommended to either increase the size of the buffer by one byte or reduce the number of bytes read by one byte.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-002 Read access to uninitialized stack memory *(Medium)*

The method `HTML_CheckRamDisk()` calls `HTML_CheckInput()` to fill the buffer `Text_u8` with at least one byte of user-input. It then extracts `HTML_Command_s32` from it thanks to `atoi(Text_u8)`, and runs a command selected by `HTML_Command_s32` next, sometimes with part of the buffer attached as an argument. For example, when `HTML_Command_s32` is equal to `HTML_CMD_ENABLE_AES_LUN(1)`, `&Text_u8[HTML_CMD_START-BYTE_OF_PAYLOAD]`, it is passed to `GetStorageKey_u32()`.

However, this pointer might be directing to uninitialized memory: if the user merely sends “1”, this code path will be reached and an uninitialized stack memory will be used as password.

[14] https://github.com/Nitrokey/nitrokey-storage-firmware/blob/master/src/USER_INTERFACE/html_io.c

It is recommended to either verify that the amount of received data is sufficient or fill `Text_u8` with zeroes before calling `HTML_CheckInput()`.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-011 HV_InitSlot_u8 zeroes encrypted slot data *(Medium)*

The audited code contains a method labelled `HV_InitSlot_u8()` that is never called. This function zeroes the encrypted slot data of a Hidden Volume slot, which could be seen by an attacker with access to flash storage. It could also be used to determine which hidden volume slots have been initialized.

A complete removal of this method is recommended.

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

NK-01-017 Sightings of outdated, deprecated or unused code *(Info)*

This ticket is meant to collect sightings of code that would require a refactoring, modification or removal. While the list does not translate into direct security impact, it is recommended to approach these findings as a step towards a more comprehensible and maintainable code base. Further, handling these issues could help avoid future weaknesses and confusion.

Unused function that zeroes a hidden volume slot reveals whether slots are used or not:

- https://github.com/Nitrokey/nitrokey-storage-firmware/blob/b0e41fb0bf6c9895887e8503c5284cdd4c90e6b7/src/HighLevelFunctions/HiddenVolume.c#L391

Statements like `if` without conditionally executed code:

- https://github.com/Nitrokey/nitrokey-storage-firmware/blob/b0e41fb0bf6c9895887e8503c5284cdd4c90e6b7/src/OTP/report_protocol.c#L1679

Bad or ambiguous file naming:

- `src/mass_storage_example.c` has “example” in its name but actually contains the `main()` method of the firmware;
- `SOFTWARE_FRAMEWORK/DRIVERS/aes/ram_aes_ram_example.c` again has “example” in its name yet contains the code used in production;
- `src/OTP/report_protocol.c` contains code that is not OTP-specific.

Commented-out, outdated code:

- https://github.com/Nitrokey/nitrokey-storage-firmware/blob/b0e41fb0bf6c9895887e8503c5284cdd4c90e6b7/src/OTP/report_protocol.c#L1827
- https://github.com/Nitrokey/nitrokey-storage-firmware/blob/b0e41fb0bf6c9895887e8503c5284cdd4c90e6b7/src/OTP/report_protocol.c#L2107

Below some of the methods with a return value that is never checked are shown. The reasoning behind listing them here (apart from superfluous return statements) is that if code that might fail is added to one of these methods, the person editing the code might believe that returning an error code is sufficient to inform the caller of a failure that the caller needs to handle.

- `XorAesKey_u32()` (always returns `TRUE`)
- `cmd_getFactoryReset()` (either returns 1 or returns by reaching the end of the function, which results in undefined behavior)
- `CheckForNewSdCard()` (always returns `TRUE`)
- `cmd_get_status()` (always returns 0)
- `cmd_get_password_retry_count()` (always returns 0)
- `parse_report()` (always returns 0)
- `cmd_write_to_slot()` (returns 0 on success, 1 on some failures)
- `cmd_read_slot_name()` (always returns 0)

Note: This issue was fixed by the Nitrokey maintainers, the fix was verified by Cure53.

Conclusion

Nitrokey marks an interesting idea among the approaches thriving towards enhanced user-privacy, data security and tamper-safety. Contrary to the software-only products, Nitrokey is capable of functioning properly and securely even if the machine that the hardware is being used on is infected by Malware or suffers from comparable problems.

The code audit was slightly hindered by how the sources were authored and structured. For the next testing stages it is recommended to have a refactoring happening first. The test identified several parts of the code that are unused, several duplicate implementations of the same feature, as well as interfaces that were planned to be removed entirely yet still reside in the inspected sources, with full or partial presence. Aside from discussing and fixing the listed vulnerabilities and weaknesses, a refactoring sprint is highly recommended before the project moves on with feature implementations and additional tests.

Given the complexity of this project, the overall amount of ten vulnerabilities and four general weaknesses is a rather positive result. It speaks to the strengths of the concept and its implementation. Nevertheless, it must be underlined that this penetration test and report is only a first assessment of the series, with several others to follow. Thus, it should by no means be read as an ultimate verdict on the Nitrokey's capabilities to keep the security promises it gives.

Please note: At the time of writing this final report, all issues spotted in the actual Nitrokey Storage firmware have been reported to the project maintainers and those that were considered actionable have been fixed promptly and quickly. All fixes have been reviewed and verified as working by the Cure53 team.

Cure53 would like to thank Jan Suhr and the entire Nitrokey Team for this interesting project as well as their continuously good support and assistance during this assignment.

We would like to further express our gratitude to the Open Technology Fund in Washington D.C., USA, for generously funding this and other penetration test projects and enabling us to publish the results.