Siklu EtherHaul 8010 – Teardown and Firmware Decryption

Published on 30 . 04 . 2025 • 24 min read •

Overview

In this post I wanted to dig into the methodology I used to understand how a vendor (Siklu) encrypts their firmware images, and from that hopefully extract the keys and reverse the process to enable easier analysis of firmware files.

Having physical access to a device you are attempting to analyze makes the process much easier, in the current situation without a physical device we would be stuck with an encrypted firmware file. Luckily in this instance I was able to physically obtain a piece of old hardware, and crack it open to have look at what we are dealing with.

Physical Teardown

Removing the casing reveals a main PCB with 2x smaller PCBs housing the 70/80Ghz radios for vertical and horizontal, to protect my self whilst working with the device on the desk I carefully disconnected the ribbon cables to prevent any potentially dangerous RF being generated.

Reviewing the PCB we can see many header pins pre-soldered ready to go, some labelled JTAG which may be useful, however before we look at those we should look to see if there is an UART pins.

UART pins are generally clustered in groups of 4x pins (TX/RX/VCC/GND) or sometimes 3x pins (TX/RX/GND) using this logic we can identify two potential headers `J7` and `JM3`

Using a multimeter and logic analyzer I was able to confirm `J7` is the UART header running on baud 115200, with the below pin out, now that we have UART lets dig into that next.

J7 – UART (3.3V)

```
GND | VCC
---------
RX  | TX
```


UART

Attaching a UART/TTL to USB adapter to the identified header, we can power the device up and monitor the boot process. (full boot log available **here**)

Observing the boot process we are able to confirm this device utilizes U-Boot as its bootloader, with a `Freescale i.MX6ULL` ARMv6 CPU, and 512MiB of RAM as well as 128MiB of NAND storage.

```
U-Boot 2017.11-svn25732 (Feb 11 2019 - 19:40:50 +0200)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 44C
Reset cause: POR
Model: Siklu TBD
Board: Siklu PCB19X
I2C:   ready
DRAM:  512 MiB
NAND:  128 MiB
```

```
In:    serial
Out:   serial
Err:   serial
Init SYSEEPROM Data...
SF: Detected gd25q16 with page size 256 Bytes, erase size 64 KiB, total 
Erasing NAND...
Erasing at 0x0 -- 100% complete.
Writing to NAND... OK
Net:   PHY reset timed out
FEC0
Hit any key to stop autoboot:  3
```

The NAND storage appears to be partitioned into 7 partitions, with two containing the firmware image (one likely being a backup/secondary firmware slot)

```
Creating 7 MTD partitions on "gpmi-nand":
0x000000000000-0x000000020000 : "env"
0x000000020000-0x000000040000 : "env2"
0x000000040000-0x000000060000 : "env_t"
0x000000060000-0x000002860000 : "uimage0"
0x000002860000-0x000005060000 : "uimage1"
0x000005060000-0x000006060000 : "conf"
0x000006060000-0x000008000000 : "log"
```

We can also identify the device is using a Linux based firmware, and once booted provides a login prompt

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.9.11+ (edwardk@rene.siklu.local) (gcc version 7.2.1 2017
...
sw login:
```

Shell access

Normally in this situation we would look at utilizing the unlocked U-Boot to adjust the boot process into single user mode and bypass the login prompt, however luckily (or unfortunately) it appears the device uses the same `root` user credentials that were discovered on the Siklu TG series devices.

Update: The static root password has been captured under CVE-2025-57175

After logging in as root we are able to dump some information regarding running processes and get a lay of the land.

running processes

```
# ps
PID   USER     COMMAND
    1 root     init
...
  440 root     /usr/sbin/dropbear -k -j -K 10 -I 0 -b /tmp/ssh_banner
  464 root     [kworker/u2:2]
  506 root     /usr/sbin/crond -c /etc/cron.d -L /dev/null
  514 root     {cli_prio_cntrl.} /bin/sh /home/sw/bin/cli_prio_cntrl.sh
  525 root     /home/sw/bin/watchdogd
  550 root     /home/sw/bin/bspd -i
  558 root     /home/sw/bin/npud
  566 root     /home/sw/bin/swupgrd
  574 root     /home/sw/bin/modemd
  588 root     /home/sw/bin/ctrl_txd
  596 root     /home/sw/bin/oamd
  604 root     /home/sw/bin/rfpiped
  711 root     /home/sw/bin/cad
  728 root     /home/sw/bin/mini_httpd -C /etc/httpd.conf
  742 root     /usr/sbin/snmpd -c /tmp/snmpd.conf
  757 root     [kworker/u2:3]
  762 root     [kworker/0:2]
  767 root     -sh
  784 root     /usr/sbin/rsyslogd
  797 root     ps
```

listening network sockets

```
# netstat -tunl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTE
tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTE
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTE
tcp        0      0 127.0.0.1:8081          0.0.0.0:*               LISTE
tcp        0      0 127.0.0.1:8082          0.0.0.0:*               LISTE
tcp        0      0 :::22                   :::*                    LISTE
tcp        0      0 :::443                  :::*                    LISTE
tcp        0      0 :::555                  :::*                    LISTE
tcp        0      0 :::80                   :::*                    LISTE
```

```
udp        0      0 0.0.0.0:161             0.0.0.0:*
udp        0      0 0.0.0.0:68              0.0.0.0:*
udp        0      0 :::161                  :::*
```

Upgrade process

With shell access we can also monitor what processes run during the upgrade process and hopefully find what might handle the decryption stage.

After "downloading" (uploading) a new firmware to the device via the web interface, we observe a process called `swupgrd_decrypt` start and spawn `openssl` to decrypt the image, of note is the `-pass stdin` argument.

As the key is being passed through to `openssl` via `stdin` we could substitute `openssl` with another binary/script which dumps stdin into a file.

```
 1100 root     swupgrd_decrypt /tmp/siklu-uimage-nxp-enc-10.7.3-18993-bal
 1223 root     sh -c openssl enc -in /tmp/siklu-uimage-nxp-enc-10.7.3-189
 1233 root     openssl enc -in /tmp/siklu-uimage-nxp-enc-10.7.3-18993-bal
 1240 root     /home/sw/bin/mini_httpd -C /etc/httpd.conf
```

Reviewing the PATH order we drop a fake `openssl` executable script into `/home/sw/bin` to intercept the stdin and save to a file

```
#!/bin/sh
cp /dev/stdin decrypt.bin
```

Whilst this did capture what appeared to be a 32 byte key, it would only work on the specific image we used to capture, capturing another image decryption key we observed the first 16 bytes were static and the second 16 bytes changed.

Digging into `swupgrd_decrypt`

Loading the binary into a de-compilation tool and searching for `openssl` we quickly identify the function that appears to be used for decryption.

In this function we can observe two `fwrite` calls to the `openssl` process handle, both being 16 bytes long.

However we still are unsure where those last 16 bytes come from (ignoring my comments) so lets dig into the calling function `main`

```
00011cc0  int32_t decrypt_file(int32_t arg1, char* enc_file)
00011cde      void openssl_cmd
00011cde      snprintf(&openssl_cmd, 0x100, "openssl enc -in %s -out %s
00011ce8      FILE* openssl_stdin = popen(&openssl_cmd, &data_129e4)
...
00011cf0      if (openssl_stdin == 0) // Check stdin is ready
...
00011cfc      else // Write 16bytes from master_key to Openssl stdin
00011cfc          fwrite(&master_key, 16, 1, openssl_stdin)
00011d08          // Write image key to OpenSSL stdin, last 48 bytes of
00011d08          // passed to arg1, with key starting at 16 bytes, and
00011d08          // being 16 bytes long
00011d08          fwrite(arg1 + 16, 16, 1, openssl_stdin)
```

In the `main` function we can observe the process reading the last 48 bytes of the image, using this to check a signature and validate the CRC32 of the image.

After this is done is removes these 48 bytes from the file, before passing them into the `decrypt_file` function, which we can see extracts 16 bytes from 16 bytes into the 48 bytes.

```
0001140c  int32_t main(int32_t argc, char** argv, char** envp)
00011424      void encFile
00011424      memset(&encFile, 0, 0xc0)
0001142a      int32_t i_11
0001142a      char* var_318
0001142a      int32_t var_300
0001142a      int32_t var_2f4
0001142a      void* var_20c
0001142a      int32_t var_204
0001142a      if (argc == 2)
00011434          strncpy(&encFile, argv[1], 0xbf)
0001143c          int32_t fileHandle = open(&encFile, argc)
00011442          char const* const var_328
00011442          if (fileHandle s< 0)
00011534              char const* const* var_2f8_1 = &var_328
00011536              var_318 = "cannot open {}"
00011536              int32_t var_314_1 = 0xe
0001153e              var_300 = 12
0001154c              int32_t var_358_2 = var_300
0001154c              int32_t var_354_3 = 0
```

```
00011558                var_328 = &encFile
00011558                int32_t var_324_2 = 0
0001155c                fmt::v8::vformat()
00011562                int32_t i_9 = 0
00011564                char const* const r1_9 = "/home/jenkins/agent/works
00011570                int32_t i
00011570                do
00011566                    r1_9 = &r1_9[1]
0001156a                    i = i_9
0001156c                    i_9 = i_9 + 1
0001156c                while (zx.d(*r1_9) != 0)
00011574                if (i u> 1)
0001157a                    void* r3_5 = &(*"/home/jenkins/agent/workspace/
00011582                    do
00011584                        uint32_t r1_10 = zx.d(*r3_5)
00011584                        r3_5 = r3_5 - 1
0001158a                        if (r1_10 == 0x2f)
0001158a                            break
0001157e                        i = i - 1
0001157e                    while (i != 1)
000115a6                syslog(0xb, "[%s:%d %s] %s", &(*"home/jenkins/agent
000115a6                goto label_115aa
0001144e            lseek(fileHandle, -48)
00011458            // Read footer which contains encryption key
00011458            ssize_t r0_4 = read(fileHandle, &var_300, 48)
0001145e            int32_t var_2d4
0001145e            int32_t r0_15
0001145e            char* var_330
0001145e            int32_t* var_310
0001145e            int32_t var_30c
0001145e            if (r0_4 == 0x30)
0001146c                if (var_300 == 0x6e47d950)
000115c8                    r0_15 = crc32(0, &var_300, 44)
000115d2                    if (r0_15 != (var_2d4 u>> 0x18 | (var_2d4 u>> 0
000115d4                        unimplemented  {vmov.I32 d16, #0}
000115e2                    var_310 = &var_330
000115ec                    var_328 = "invalid enc footer"
000115f4                        unimplemented  {vstr d16, [r7, #0x30]}
000115f8                    int32_t* r2_10 = var_310
000115fa                    char* var_358_3 = var_318
000115fa                    int32_t var_314
000115fa                    int32_t var_354_5 = var_314
00011604                        unimplemented  {vstr d16, [r7, #0x18]}
00011608                    fmt::v8::vformat()
0001160e                    int32_t i_6 = 0
00011610                    char const* const r1_16 = "/home/jenkins/ag
0001161c                    int32_t i_1
0001161c                    do
00011612                        r1_16 = &r1_16[1]
```

```
00011616                        i_1 = i_6
00011618                        i_6 = i_6 + 1
00011618                    while (zx.d(*r1_16) != 0)
0001162e                    for (; i_1 u> 1; i_1 = i_1 - 1)
00011628                        if (zx.d((*"/home/jenkins/agent/workspa
00011628                            break
0001164a                    syslog(0xb, "[%s:%d %s] %s", &(*"home/jenk
00011656                else  // Trim last 48 bytes, and extract key
00011656                    int32_t var_2e0
00011656                    ftruncate(fileHandle, var_2e0 u>> 24 | (var
00011664                    if (decrypt_file(&var_300, &encFile) != 1)
```

Wrapping it up

A golden rule in IT security is once someone has physical access, all bets are off, and this is a clear case of that.

With physical access we were able to obtain a shell to the underlying OS of the device, and from there monitor the decryption process and obtain the binaries responsible.

Whilst the static decryption key cannot be disclosed, below is an example python script to perform a decryption of the firmware.

1. Extract image key from file (seek to last 32 bytes of image, read 16 bytes)
2. Duplicate image file, and trim last 48 bytes of image
3. Start `openssl` process and pass in master key + image key

Usage

Simple run the python3 script passing the encrypted image file as the first argument

```
python3 decrypt_firmware.py <image>
```

```
python3 decrypt_firmware.py siklu-uimage-nxp-enc-10_6_2-18707-ea552dc00b
Processing file: siklu-uimage-nxp-enc-10_6_2-18707-ea552dc00b.zip

*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

```
Cleaning up truncated file siklu-uimage-nxp-enc-10_6_2-18707-ea552dc00b.z
Decrypted file saved to siklu-uimage-nxp-enc-10_6_2-18707-ea552dc00b.zip_
```

Code

```
import subprocess
import sys, os
import binascii

# Check if there are enough arguments
if len(sys.argv) > 1:
    input_file_path = sys.argv[1]
    print("Processing file:", input_file_path)
else:
    print("No file specified.")
    sys.exit(1)


# Define file paths for truncated encrypted file, and where to save decr
truncated_file_path = input_file_path + '_truncated'
output_file_path = input_file_path + '_decrypted'

# Common base encryption key used by Siklu
base_key = "<REDACTED>"

# Function to remove the last 48 bytes from file
def remove_last_48_bytes(in_file, out_file):
    with open(in_file, 'rb') as f:
        content = f.read()

    new_content = content[:-48]  # Slice the content, excluding the last

    with open(out_file, 'wb') as f:
        f.write(new_content)

# Function to exctract image encryption key which is stored in the last
def extract_key(in_file):
    with open(in_file, 'rb') as f:
        # Seek to the position 16 bytes before the end of the file
        f.seek(-32, 2)  # 2 means to seek relative to the end of the fil

        # Read the last 16 bytes
        key_bytes = f.read(16)

    return binascii.hexlify(key_bytes).decode('utf-8')

# Build full decryption key from base key and key from input file
```

```
decrypt_key = bytearray.fromhex( base_key + extract_key(input_file_path))

# Strip extra data from input file, produce valid OpenSSL encrypted file
remove_last_48_bytes(input_file_path, truncated_file_path)

# Execute OpenSSL and setup pipes to input key
openssl = subprocess.Popen(
    ['openssl', 'enc', '-d', '-aes256', '-pass', 'stdin', '-md', 'md5',
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE
)

# Pass the byte data (password) to the openssl process via stdin
stdout, stderr = openssl.communicate(input=decrypt_key)

# Optionally, print the stdout and stderr
print(stdout.decode())  # Decrypted output
print(stderr.decode())  # Any error messages

print(f'Cleaning up truncated file {truncated_file_path}')
os.remove(truncated_file_path)

print(f'Decrypted file saved to {output_file_path}')
```