# Galactic Dead Ends: Retrofitting Encrypted Firmware

Peter Geissler
HAXXIN (https://haxx.in)

# Good afternoon, SASCon!

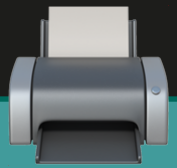

**Who still has one of these things in their (home)office? 🖨️**

**I’m Peter (aka blasty), I hack them for fun and sometimes profit. 🤓💰**

# Introduction

## what is this talk all about?

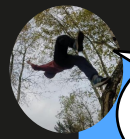
This talk is divided into two sections



**Dissecting a new layer of encryption in firmware for Lexmark printers**

**Learning more about a security coprocessor present in some Marvell SoC's**

# Story time(line)



lexmark added new crypto a day before pwn2own plz install this backdoor and dump rootfs

okidoki

20 October, 2023

SAS24 cfp?

OK, maybe

15 March, 2024

new lexmark crypto will be haxed soon™

NULLCON

7 July, 2024

cfp eta wen?

almost done!

10 July, 2024

fsck, now I **actually** need to work on this..

18 July, 2024

SAS24

😊

23 October, 2024

# Oldschool lexmark firmware updates

- Lexmark printers run Linux and have been studied quite well.
- Firmware update is a PJL (Printer Job Language) stream containing two encrypted blobs.
- RSA encrypted header
- RSA public keys can be obtained using static analysis from existing firmware (🐔, 🥚 problem)
  - https://github.com/blasty/lexmark/blob/main/tools/keys.py
  - https://github.com/blasty/lexmark/blob/main/tools/fw_decrypt.py
- Decrypting the blobs gives us access to a squashFS image containing the root filesystem.
  - Sufficient for doing analysis of the network-facing codebase and writing exploits

```
$ head -n3 CXLBL.076.308.fls
-12345X@PJL
@PJL COMMENT NETFLASH ID="CXLBL,CXLBN,CSLBN" RIP="076.308" ENG="BL.076.E020" IOT="1.1.10+git0+73d8293210" DATE="20220301"
@PJL LPROGRAMRIP SOCKET=1 KERNELCOUNT=5698336 TYPECOUNT=135148064 KERNELENCR=3 FKSIGNSZ=5696779
FLASHOPTS="CV=076.308;SV=2.0;TV=1;AF=1;MD=1;NUH=1;LJF=1;SMA=1;" RIPNAME="granite2-color-lite"
```

```
$ python3 fw_decrypt.py CXLBL.076.308.fls CXLBL.076.308
> section header:
  - decrypted signature size : 0x00000014
  - decrypted aes key size   : 0x00000010
  - aes key byte size        : 0x00000010
  - aes mode                 : 0x00000001
  - decrypted data size      : 0x0056ef20

> signature : e2151d33148447cf89a4bda5cf92b07c980a8be3
> AES key   : f3719be787c735eb1085608c49dc393f
> unpacking section 'flasher': . done!

> section header:
  - decrypted signature size : 0x00000014
  - decrypted aes key size   : 0x00000010
  - aes key byte size        : 0x00000010
  - aes mode                 : 0x00000001
  - decrypted data size      : 0x080e2e20

> signature : 995390d6c271163dc4ee67e3f85659e67e7dd110
> AES key   : 97791ddcdd4574c418712d71d87fb28f
> unpacking section 'main': ................. done!

> saved content 'uboot' to 'CXLBL.076.308/main/content_uboot.bin'
> saved content 'initramfs' to 'CXLBL.076.308/main/content_initramfs.bin'
> saved content 'rootfs' to 'CXLBL.076.308/main/content_rootfs.bin'
> saved content 'license' to 'CXLBL.076.308/main/content_license.bin'
> saved content 'scannerfs' to 'CXLBL.076.308/main/content_scannerfs.bin'
```

```
$ unsquashfs CXLBL.076.308/main/content_rootfs.bin

$ ls -la squashfs-root
total 8
drwx------   22 user  staff   704 Oct  3 12:45 .
drwxr-xr-x   30 user  staff   960 Oct  3 12:45 ..
drwxr-xr-x    2 user  staff    64 Nov 16  2021 .devtool
-rw-r--r--    1 user  staff   980 Mar  1  2022 Build.Info
drwxr-xr-x   97 user  staff  3104 Mar  1  2022 bin
drwxr-xr-x    2 user  staff    64 Mar  1  2022 boot
drwxr-xr-x    2 user  staff    64 Mar 11  2021 dev
drwxr-xr-x  148 user  staff  4736 Mar  1  2022 etc
drwxr-xr-x    7 user  staff   224 Mar  1  2022 home
drwxr-xr-x   79 user  staff  2528 Mar  1  2022 lib
drwxr-xr-x    2 user  staff    64 Mar 11  2021 media
drwxr-xr-x    2 user  staff    64 Mar 11  2021 mnt
drwxr-xr-x    5 user  staff   160 Mar  1  2022 opt
drwxr-xr-x    4 user  staff   128 Mar  1  2022 pkg-netapps
dr-xr-xr-x    2 user  staff    64 Mar 11  2021 proc
drwx------    4 user  staff   128 Mar  1  2022 root
drwxr-xr-x    2 user  staff    64 Mar 11  2021 run
drwxr-xr-x   97 user  staff  3104 Mar  1  2022 sbin
drwxr-xr-x    2 user  staff    64 Mar 11  2021 srv
dr-xr-xr-x    2 user  staff    64 Mar 11  2021 sys
drwxrwxrwt    2 user  staff    64 Mar 11  2021 tmp
drwx------    6 user  staff   192 Oct  3 12:45 usr
```

```
$ python3 fw_decrypt.py CXLBL.230.037.fls CXLBL.230.037
> section header:
  - decrypted signature size : 0x00000014
  - decrypted aes key size   : 0x00000010
  - aes key byte size        : 0x00000010
  - aes mode                 : 0x00000001
  - decrypted data size      : 0x008b51b0

> signature : 8628db0a74e88d66c2d3d0cc5ae7f42abe5fd99f
> AES key   : fd9844b269f9de4620b46bc19f2f2786
> unpacking section 'flasher': .. done!

> section header:
  - decrypted signature size : 0x00000014
  - decrypted aes key size   : 0x00000010
  - aes key byte size        : 0x00000010
  - aes mode                 : 0x00000001
  - decrypted data size      : 0x0910f150

> signature : 1129b982bf0b3f7e3a7a4caceac7d2cfad896b3c
> AES key   : a559bb2efd4f5dd2dfa104ee9ad3d05c
> unpacking section 'main': ................... done!

> saved content 'uboot' to 'CXLBL.230.037/main/content_uboot.bin'
> saved content 'initramfs' to 'CXLBL.230.037/main/content_initramfs.bin'
> saved content 'rootfs' to 'CXLBL.230.037/main/content_rootfs.bin'
> saved content 'license' to 'CXLBL.230.037/main/content_license.bin'
```

```
$ unsquashfs \
      CXLBL.230.037/main/content_rootfs.bin
FATAL ERROR: Can't find a valid
SQUASHFS superblock on
CXLBL.230.037/main/content_rootfs.bin

$ file CXLBL.230.037/main/content_rootfs.bin
CXLBL.230.037/main/content_rootfs.bin: data
```

# U-boot environment

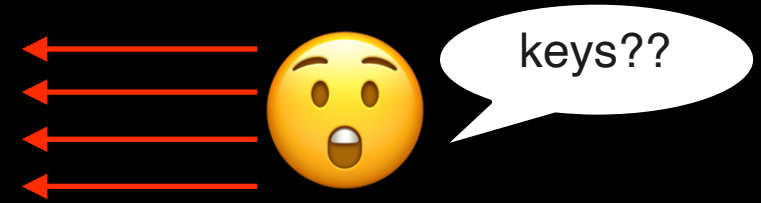
```
setenv bootargs "$bootargs dm-mod.create=\”[...]

crypt_root,,1,ro,0 ${sectors} crypt aes-cbc-plain64be:plain
:32:user:wtm:rootfs 0 /dev/dm-0 0

\" root=/dev/dm-1"
```

# U-boot environment

```
setenv bootargs "$bootargs dm-mod.create=\"[...]

crypt_root,,1,ro,0 ${sectors} crypt aes-cbc-plain64be:plain
:32:user:wtm:rootfs 0 /dev/dm-0 0

\" root=/dev/dm-1"
```

# initramfs

```
$ (cd uInitramfs_unpacked ; find . | grep wtm)
/lib/modules/5.4.254-yocto-standard/extra/drivers/wtm-client
/lib/modules/5.4.254-yocto-standard/extra/drivers/wtm-client/wtm-client.ko
/lib/modules/5.4.254-yocto-standard/extra/drivers/wtm-controller
/lib/modules/5.4.254-yocto-standard/extra/drivers/wtm-controller/wtm-controller.ko
/usr/share/wtm-crypt
/usr/share/wtm-crypt/wkey2.bin
/usr/share/wtm-crypt/wkey4.bin
/usr/share/wtm-crypt/wkey3.bin
/usr/share/wtm-crypt/wkey1.bin
```



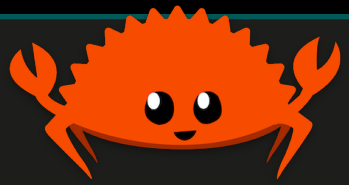
# More observations

```
$ modinfo wtm-client/wtm-client.ko
filename:       wtm-client.ko
alias:          net-pf-16-proto-16-family-wtm-mailbox
description:    WTM Mailbox Client
license:        Proprietary
alias:          of:N*T*Cmarvell,wtm-mailbox-clientC*
alias:          of:N*T*Cmarvell,wtm-mailbox-client
depends:
name:           wtm_client
vermagic:       5.4.254-yocto-standard SMP preempt mod_unload ARMv7 p2v8
```

```
root@ET788C77F816DD:~# grep '^Hardware' /proc/cpuinfo
Hardware        : Marvell Pegmatite (Device Tree)
```

MARVEL

```
$ strings uInitramfs_unpacked/sbin/init \
       | grep wtm

/usr/src/debug/libstd-rs/1.66.0-r0/rustc-
1.66.0-src/library/core/src/str/pattern.rs
src/main.rs/lib/modulespath to strextradri
vers.komodule paramsinit_module/dev/ubiblo
ck0_2no dm-mod.create/dev/mmcblkp/mnt/bin/
etc/lib/usr/varproc/sys/devrootinitBasecst
rcramfsmount boot/mnt/rootfs.keyno rootfs.
keyread keyno indexno flagsno tabledevtmpf
ssquashfsumount bootwtm-controllerload wtm
-controller
```



# More observations

```
$ modinfo wtm-client/wtm-client.ko
filename:       wtm-client.ko
alias:          net-pf-16-proto-16-family-wtm-mailbox
description:    WTM Mailbox Client
license:        Proprietary
alias:          of:N*T*Cmarvell,wtm-mailbox-clientC*
alias:          of:N*T*Cmarvell,wtm-mailbox-client
depends:
name:           wtm_client
vermagic:       5.4.254-yocto-standard SMP preempt mod_unload ARMv7 p2v8
```

```
root@ET788C77F816DD:~# grep '^Hardware' /proc/cpuinfo
Hardware        : Marvell Pegmatite (Device Tree)
```

```
$ strings uInitramfs_unpacked/sbin/init \
       | grep wtm

/usr/src/debug/libstd-rs/1.66.0-r0/rustc-
1.66.0-src/library/core/src/str/pattern.rs
src/main.rs/lib/modulespath to strextradri
vers.komodule paramsinit_module/dev/ubiblo
ck0_2no dm-mod.create/dev/mmcblkp/mnt/bin/
etc/lib/usr/varproc/sys/devrootinitBasecst
rcramfsmount boot/mnt/rootfs.keyno rootfs.
keyread keyno indexno flagsno tabledevtmpf
ssquashfsumount bootwtm-controllerload wtm
-controller
```


# OSINT Time

The PXA-2128 and PXA-610 SoCs are equipped with a dedicated security hardware module known as **WTM** (**Wireless Trusted Module**) that offers the trusted computing services required for user authentication, identity management, secure storage as well as secure communication. Within WTM, there is a pool of the hardware cryptographic engines that performs at high throughput of the cryptographic operations over a set of FIPS-Approved algorithms, such as AES, TDES, SHA, HMAC, RSA, and EC-DSA. In addition, the on-chip hardware entropy-bit-generator under WTM is a reliable source of the entropy seeding to the FIPS-Approved DRBG schemes. The dedicated **WTM secure firmware is responsible for device trusted boot, access control, authentication, and key management**.

source: FIPS 140-2 Security Policy for Marvell, Inc. Armada Mobile Processor

# OSINT Time


✅ 👍

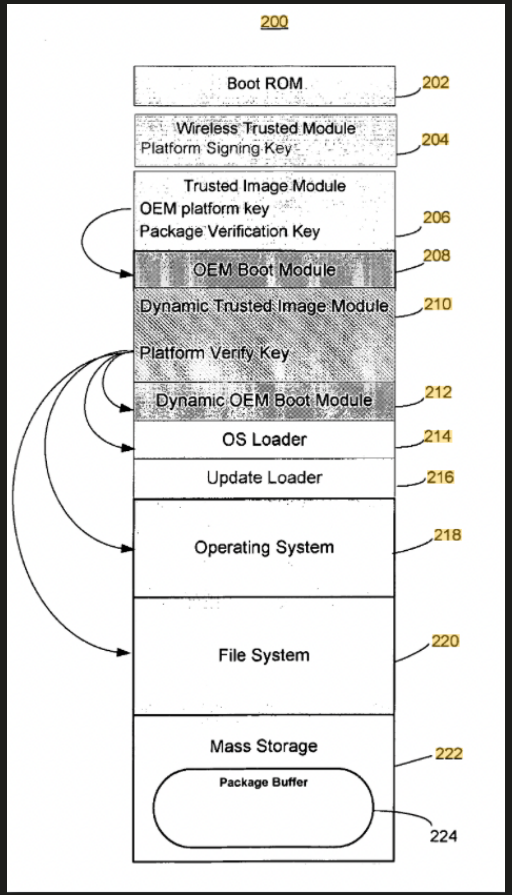
# OSINT Time

“Trusted modular firmware update using digital certificate”

**Application US12/108,910 events**

| | |
|---|---|
| **2008-04-24** | Application filed by Marvell International Ltd |
| **2008-04-24** | Priority to US12/108,910 |
| **2013-10-15** | Application granted |
| **2013-10-15** | Publication of US8560823B1 |
| **Status** | Active |
| **2031-11-12** | Adjusted expiration |

source: patents.google.com



# OSINT Time

*The OLPC XO (formerly known as $100 Laptop, Children's Machine, 2B1) is a low cost laptop computer intended to be distributed to children in developing countries around the world, to provide them with access to knowledge, and opportunities to "explore, experiment and express themselves" (constructionist learning).*

XO-1.75 is designed around the **Marvell Armada** 610 system on a chip

# source/drivers/input/serio/olpc_apsp.c

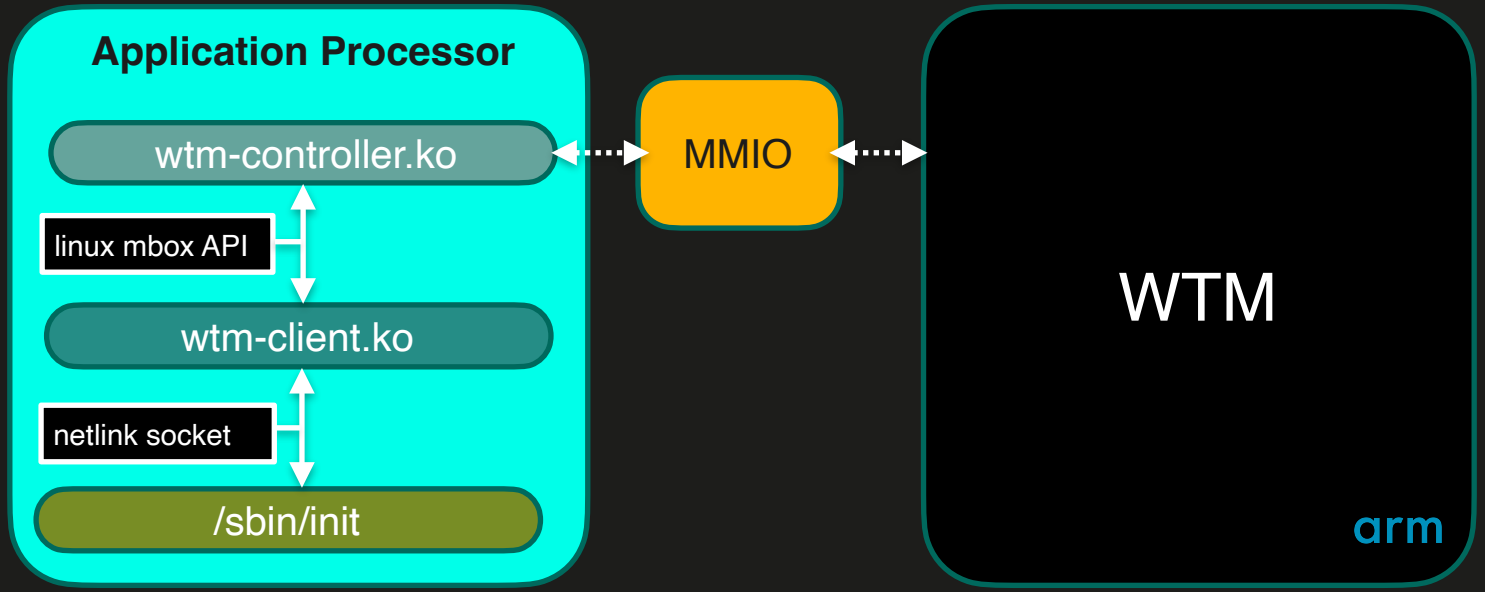
```
/*
 * The OLPC XO-1.75 and XO-4 laptops do not have a hardware PS/2 controller.
 * Instead, the OLPC firmware runs a bit-banging PS/2 implementation on an
 * otherwise-unused slow processor which is included in the Marvell MMP2/MMP3
 * SoC, known as the “Security Processor” (SP) or "Wireless Trusted Module"
 * (WTM). This firmware then reports its results via the WTM registers,
 * which we read from the Application Processor (AP, i.e. main CPU) in this
 * driver.
 *
 * On the hardware side we have a PS/2 mouse and an AT keyboard, the data
 * is multiplexed through this system. We create a serio port for each one,
 * and demultiplex the data accordingly.
 */
```

# OLPC WTM firmware

- Is actually a small forth interpreter 🤓
- All code available on dev.laptop.org git repositories.
- Interesting reference since we don't actually have a manual for the WTM
- From the Makefiles in this code we can tell the WTM is actually an ARM based core

```
$ cat src/app/arm-xo-1.75/ps2.fth
h# 282000 value ic-base  \ Interrupt controller

: ic@  ( offset -- l )  ic-base + io@  ;
: ic!  ( l offset -- )  ic-base + io!  ;

: block-irqs  ( -- )  1 h# 10c ic!  ;
: unblock-irqs  ( -- )  0 h# 10c ic!  ;

: irq-enabled?  ( level -- flag )  /l* ic@ h# 10 and 0<>  ;
: enable-irq  ( level -- )  h# 11 swap /l* ic!  ;  \ Enable for IRQ0
: disable-irq  ( level -- )  0 swap /l* ic!  ;

: all-interrupts-off  ( -- )
   d# 64 0  do  i disable-irq  loop
;
```

# WTM on Lexmark printers



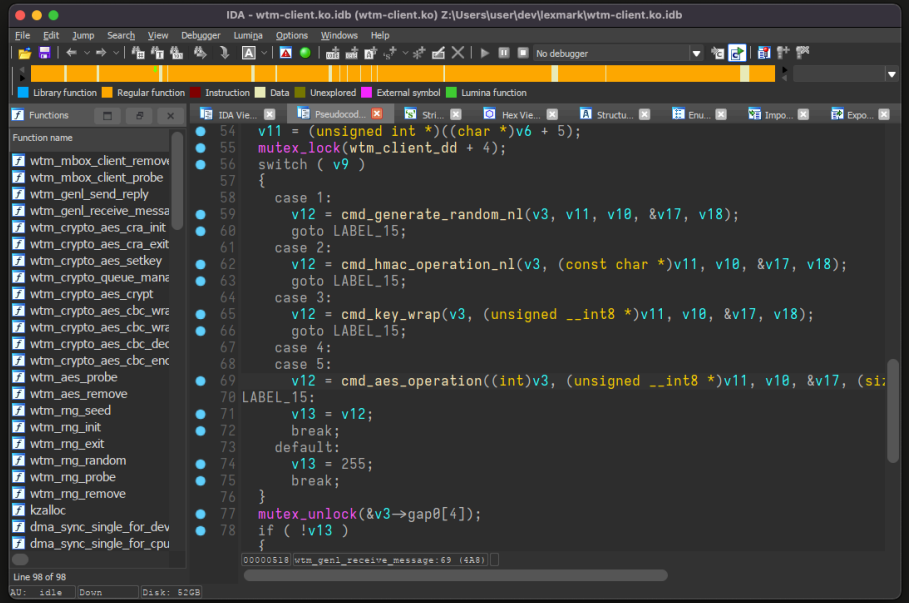
# wtm-client.ko static analysis

## Highlevel Netlink Messages

```
1: cmd_generate_random
2: cmd_hmac_operation
3: cmd_key_wrap
4: cmd_aes_operation
5: cmd_aes_operation
```

↓ WTM Commands

```
0x3007: key_wrap
0x3008: key_unwrap_load
0x4000: generate_random
0x5000: aes_init
0x5001: aes_zeroize
0x7004: hmac_init
```



# let’s emulate /sbin/init

## in a qemu-user chroot

```
mount@.plt      ──▶  mov r0, #0    ; bx lr
umount@.plt     ──▶  mov r0, #0    ; bx lr
socket@.plt     ──▶  mov r0, #3    ; bx lr
sub_33e7c       ──▶  mov r0, #0x86 ; bx lr
“/proc/cmdline” ──▶  “/prac/cmdline”
```

```
rootfs $ cat preinit.c
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>

int main() {
    struct sockaddr_in servaddr;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(0x1337);
    connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    char *argv[]={ "init", NULL };
    execve("/init", argv, NULL);
}
```

```
rootfs $ mount -t proc none ./proc
rootfs $ mount -t sysfs none ./sys
rootfs $ cat prac/cmdline
[...] dm-mod.create=“ [...] ;crypt_root,,1,ro,0
268584 crypt aes-cbc-plain64be:plain
:32:user:wtm:rootfs 0 /dev/dm-0 0" root=/dev/
dm-1
rootfs $ chroot $(pwd) /preinit
```

# netlink

init → wtm-client

```
000000: 2400 0000 1000 0500 0000 0000 0000 0000  $...............
000010: 0302 0000 1000 0200 7774 6d2d 6d61 696c  ........wtm-mail
000020: 626f 7800                                box.
```

wtm-client → init

```
000000: 5c00 0000 1000 0000 0000 0000 a403 0000  \...............
000010: 0102 0000 1000 0200 7774 6d2d 6d61 696c  ........wtm-mail
000020: 626f 7800 0600 0100 1700 0000 0800 0300  box.............
000030: 0100 0000 0800 0400 0000 0000 0800 0500  ................
000040: 0200 0000 1800 0600 1400 0100 0800 0100  ................
000050: 0100 0000 0800 0200 0a00 0000            ............

000000: 2400 0000 0200 0001 0000 0000 a504 0000  $...............
000010: 0000 0000 2400 0000 1000 0500 0000 0000  ....$...........
000020: 0000 0000                                ....
```

init → wtm-client

```
000000: 8c02 0000 1700 0100 0000 0000 0000 0000  ................
000010: 0101 0000 7502 0200 0506 0101 2000 0000  ....u.......  ...
000020: 0105 8000 0008 0000 0069 4e38 d798 91f0  .........iN8....
[...]
000260: 7987 bb8d cef1 0228 d88b aa6b 9d88 5d8b  y......(...k..].
000270: ad5f 4d89 3f3c ed0b bc05 661d 55ba 7549  ._M.?<....f.U.uI
000280: 8071 ae5e 1f55 fc9f 7b00 0000            .q.^.U..{...
```

wtm-client → init

```
000000: 4000 0000 1700 0000 0100 0000 0000 0000  @...............
000010: 0101 0000 0600 0100 0000 0000 2400 0200  ............$...
000020: 7930 6820 6430 3064 2c20 6730 2064 756d  =..iS!...m.$...v
000030: 7020 6a30 3072 2030 776e 206b 3379 7a21  #..]c.16..;.5...
```

# netlink

init → wtm-client

```
000000: 2400 0000 1000 0500 0000 0000 0000 0000  $...............
000010: 0302 0000 1000 0200 7774 6d2d 6d61 696c  ........wtm-mail
000020: 626f 7800                                box.
```

wtm-client → init

```
000000: 5c00 0000 1000 0000 0000 0000 a403 0000  \...............
000010: 0102 0000 1000 0200 7774 6d2d 6d61 696c  ........wtm-mail
000020: 626f 7800 0600 0100 1700 0000 0800 0300  box.............
000030: 0100 0000 0800 0400 0000 0000 0800 0500  ................
000040: 0200 0000 1800 0600 1400 0100 0800 0100  ................
000050: 0100 0000 0800 0200 0a00 0000            ............

000000: 2400 0000 0200 0001 0000 0000 a504 0000  $...............
000010: 0000 0000 2400 0000 1000 0500 0000 0000  ....$...........
000020: 0000 0000                                ....
```

handshake/init (boring)

init → wtm-client

```
000000: 8c02 0000 1700 0100 0000 0000 0000 0000  ................
000010: 0101 0000 7502 0200 0506 0101 2000 0000  ....u.......  ...
000020: 0105 8000 0008 0000 0069 4e38 d798 91f0  .........iN8....
[...]
000260: 7987 bb8d cef1 0228 d88b aa6b 9d88 5d8b  y......(...k..].
000270: ad5f 4d89 3f3c ed0b bc05 661d 55ba 7549  ._M.?<....f.U.uI
000280: 8071 ae5e 1f55 fc9f 7b00 0000            .q.^.U..{...
```

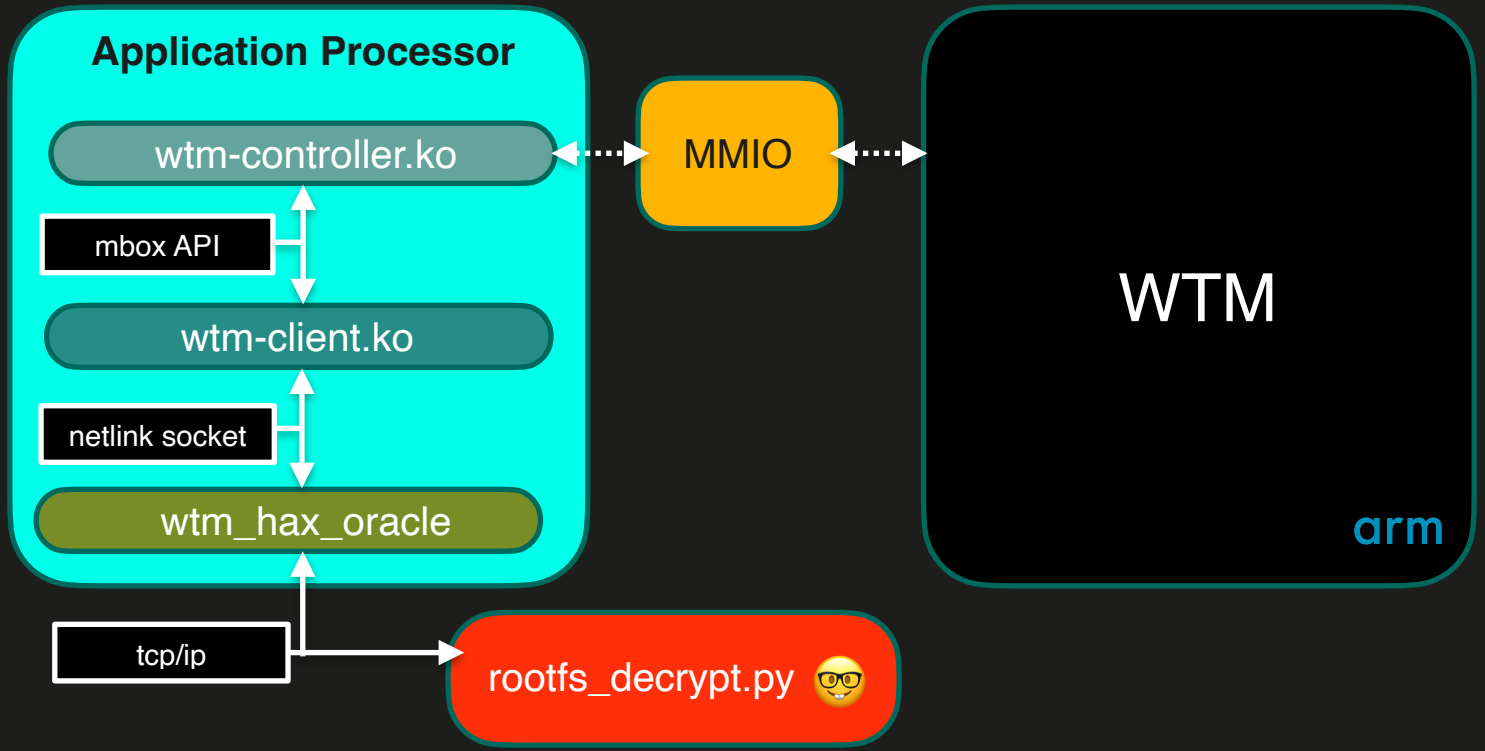
contents of wkey4.bin
last 0x20 bytes of rootfs.key
first 0x10 bytes of rootfs.key

wtm-client → init

```
000000: 4000 0000 1700 0000 0100 0000 0000 0000  @...............
000010: 0101 0000 0600 0100 0000 0000 2400 0200  ............$...
000020: 7930 6820 6430 3064 2c20 6730 2064 756d  =..iS!...m.$...v
000030: 7020 6a30 3072 2030 776e 206b 3379 7a21  #..]c.16..;.5...
```

rootfs AES-256 key!

# WTM on Lexmark printers



# Decrypting root filesystem

## assisted by wtm_hax_oracle

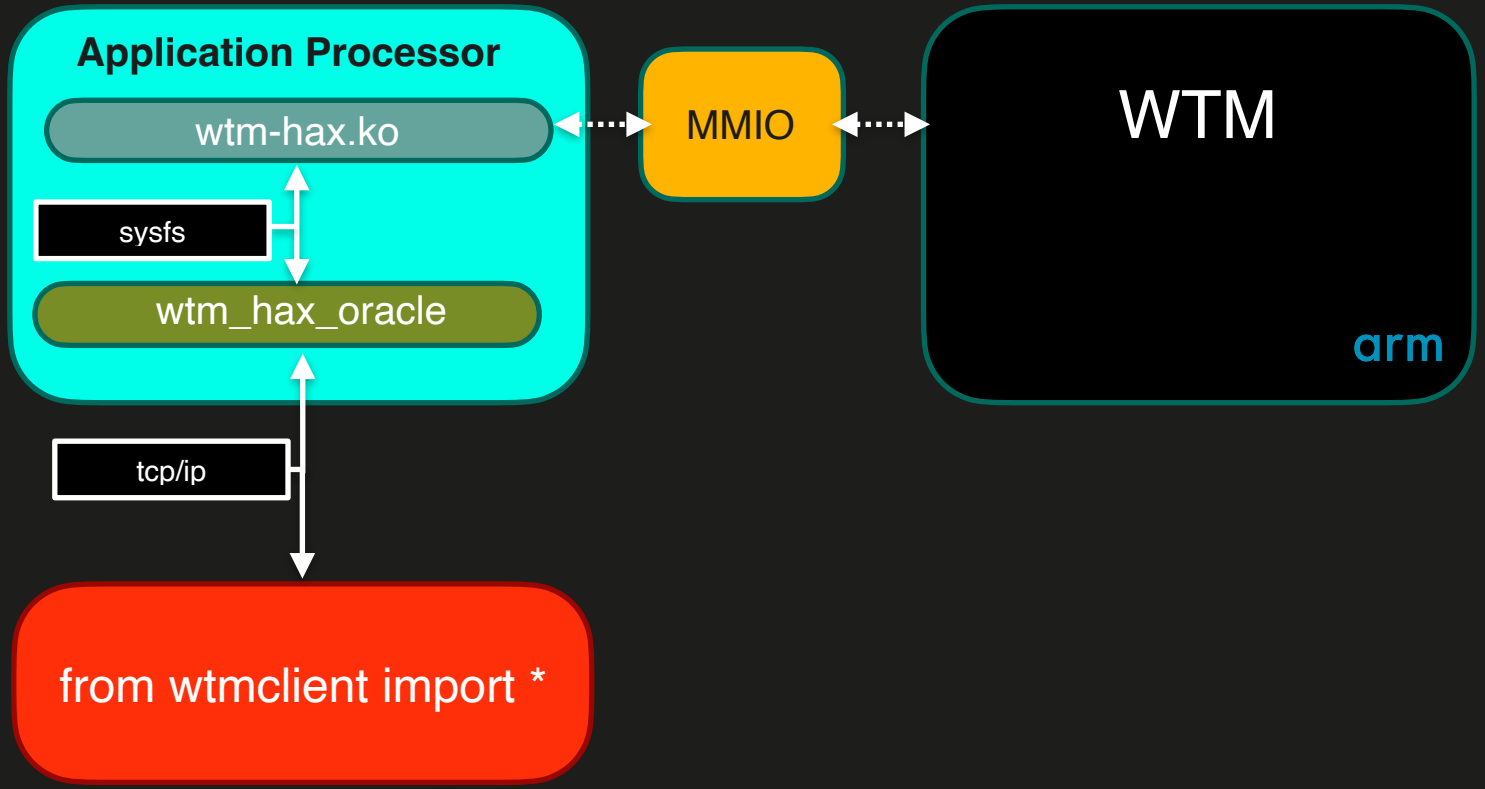
```
$ python3 rootfs_decrypt.py ./CXLBL.230.037 192.168.0.13
> wrapped rootfs key   : 05661d55ba [..] 93f3ced0bbc
> unwrapped rootfs key : -- 8< snip snip 8< --
100%|████████████████████| 270736/270736 [00:02<00:00, 132357.71it/s]
> decrypted rootfs written to ./CXLBL.230.037/main/content_rootfs_dec.bin

$ file CXLBL.230.037/main/content_rootfs_dec.bin
CXLBL.230.037/main/content_rootfs_dec.bin: Squashfs filesystem, little
endian, version 4.0, xz compressed, 137513245 bytes, 14547 inodes,
blocksize: 131072 bytes, created: Wed Oct 11 18:15:44 2023
```

# Talking directly to WTM

- Build a kernel module that can talk to WTM over MMIO directly
- Kernel module exposes an interface through sysfs for reading/writing WTM MMIO space from userland
- Extended wtm_hax_oracle daemon with commands that talk to this kernel module

| | |
|---|---|
| CMD_READ8<br>CMD_READ16<br>CMD_READ32 | read from WTM MMIO region |
| CMD_WRITE8<br>CMD_WRITE16<br>CMD_WRITE32 | write to WTM MMIO region |
| CMD_GET_SCRATCH | get pointer to scratch buffer |
| CMD_EXEC_CMD | send a command to the WTM |

# Talking directly to WTM



# Where is the Lexmark WTM firmware?

- Early analysis of decrypted data section was always done using tools like binwalk/unblob.
- Later revised the decrypter tool to properly unpack the subsections of the data section
- Browsing through the various (unpacked) blobs and nested formats I found something odd..
- A CPIO archive containing a tiny ramdisk filesystem..
- Some wtm related kernel driver I had not seen before
- And a kernel version string that doesn’t match the one I had seen before..

```
$ find . -type f
./etc/version
./etc/timestamp
./lib/modules/5.4.90/modules.alias
./lib/modules/5.4.90/modules.devname
./lib/modules/5.4.90/modules.dep
./lib/modules/5.4.90/modules.order
./lib/modules/5.4.90/modules.dep.bin
./lib/modules/5.4.90/modules.symbols.bin
./lib/modules/5.4.90/extra/wtm-linux.ko
./lib/modules/5.4.90/modules.builtin.bin
./lib/modules/5.4.90/modules.builtin
./lib/modules/5.4.90/modules.symbols
./lib/modules/5.4.90/modules.alias.bin
./lib/modules/5.4.90/modules.softdep
./lib/modules/5.4.90/modules.builtin.modinfo
./init
./bin/sh
./dev/console
```

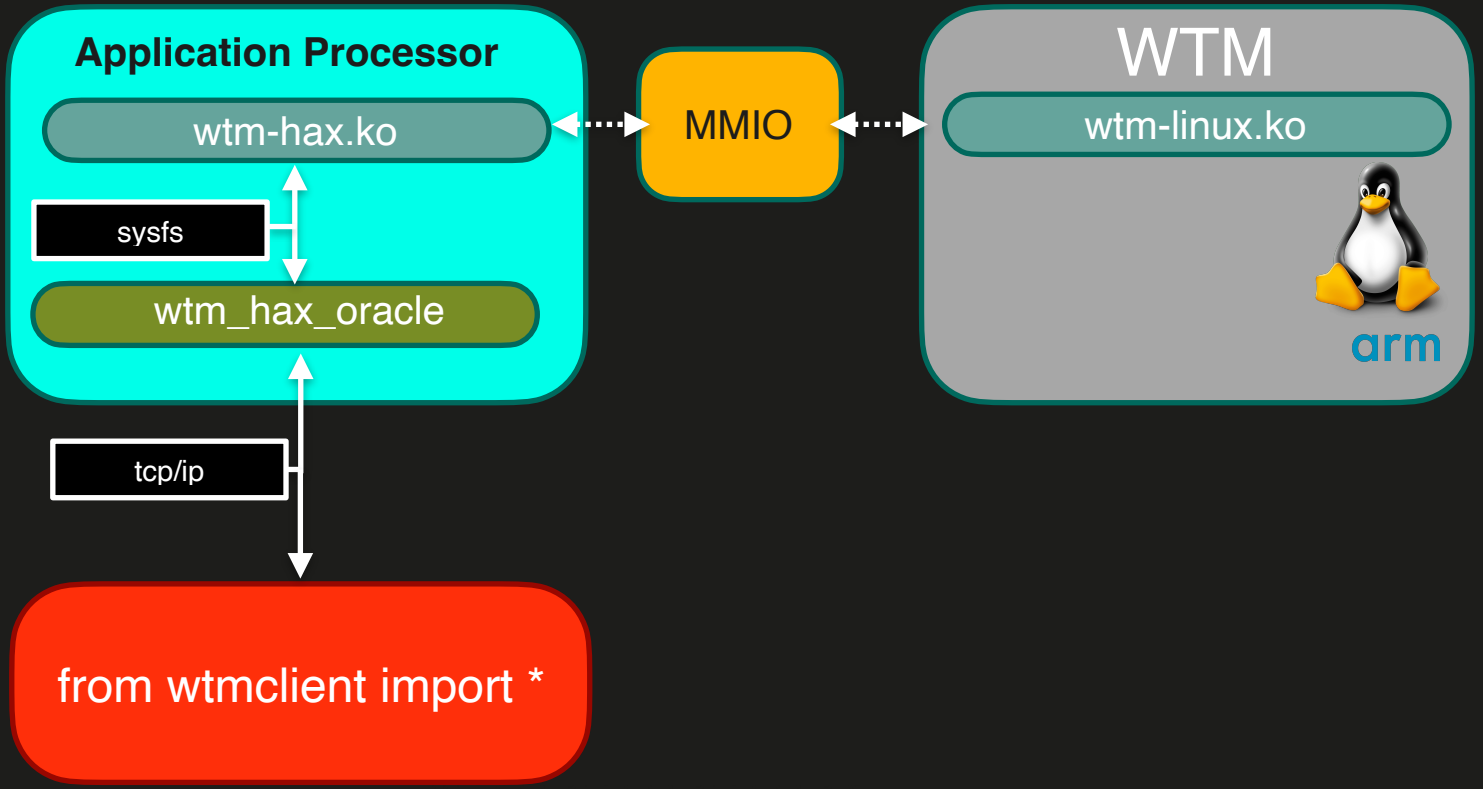
OMG, the WTM itself runs Linux! 🚀

**wtm-linux.ko** implements all WTM command handling, we can now reverse all of this!

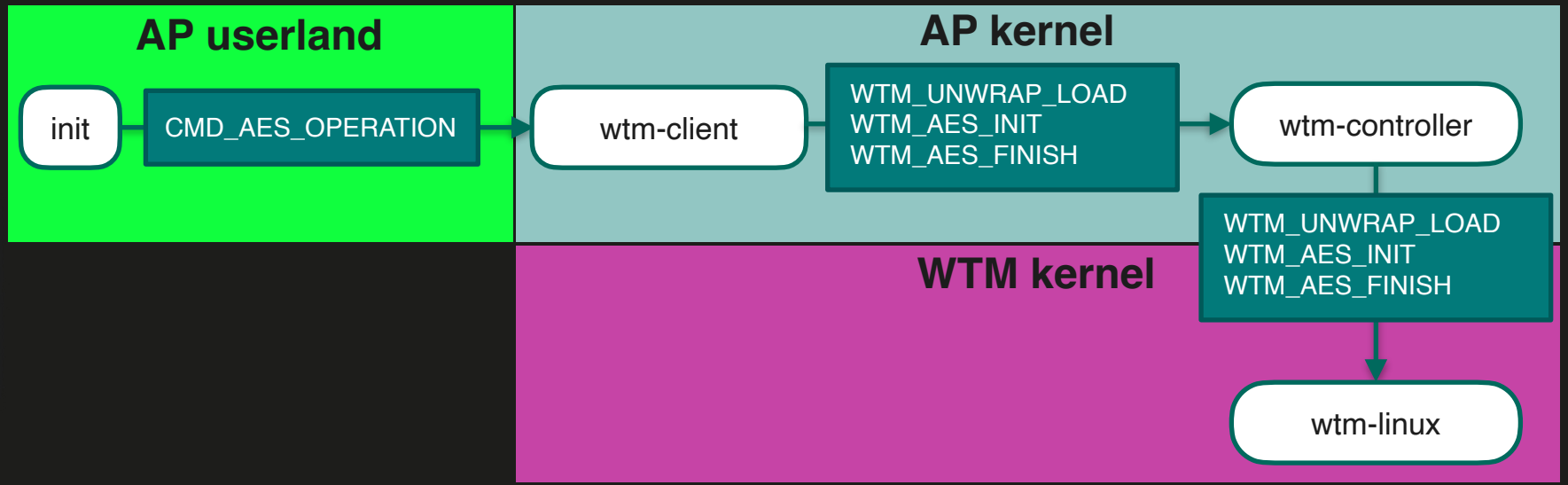
# so many commands!

```
0x0000: wtm_reset
0x0001: wtm_init
0x0002: wtm_self_test
0x0003: wtm_configure_dma
0x0004: wtm_set_batch_count
0x0005: wtm_ack_batch_error
0x0006: wtm_freq_change
0x1000: wtm_get_trust_status_register
0x1001: wtm_kernel_version_read
0x1002: wtm_software_version_advance
0x1003: wtm_software_version_read
0x1004: wtm_lifecycle_advance
0x1005: wtm_lifecycle_read
0x2000: wtm_oem_platform_bind
0x2001: wtm_oem_platform_verify
0x2002: wtm_oem_jtag_key_bind
0x2003: wtm_oem_jtag_key_verify
0x2004: wtm_otp_write_platform_config
0x2005: wtm_otp_read_platform_config
0x2006: wtm_rkek_provision
0x2007: wtm_set_aes_use_rkek
0x2008: wtm_otp_block_write
0x2009: wtm_otp_block_read
0x200a: wtm_oem_usbid_provision
0x200b: wtm_oem_usbid_read
0x200c: wtm_set_jtag_permanent_disable
0x200d: wtm_set_temporary_fa_disable
0x200e: wtm_device_pin_provision
0x3000: wtm_get_context_info
0x3001: wtm_load_engine_context
0x3002: wtm_store_engine_context
0x3003: wtm_load_engine_context_external
0x3004: wtm_store_engine_context_external
0x3005: wtm_purge_context
0x3006: wtm_get_next_cache_slot_id
0x3007: wtm_key_wrap
0x3008: wtm_key_unwrap_load
0x4000: wtm_drbg_gen_ran_bits
0x4001: wtm_x931_drbg
0x5000: wtm_aes_init
0x5001: wtm_aes_zeroize
0x5002: wtm_aes_process
0x5003: wtm_aes_finish
0x6000: wtm_des_init
0x6001: wtm_des_zeroize
0x6002: wtm_des_process
0x6003: wtm_des_finish
0x7000: wtm_hash_init
0x7001: wtm_hash_zeroize
0x7002: wtm_hash_update
0x7003: wtm_hash_final
0x7004: wtm_hmac_init
0x7005: wtm_hmac_zeroize
0x7006: wtm_hmac_update
0x7007: wtm_hmac_final
0x8000: wtm_emsa_pkcs1_v15_verify
0x8001: wtm_emsa_pkcs1_v15_verify_init
0x8002: wtm_emsa_pkcs1_v15_verify_update
0x8003: wtm_emsa_pkcs1_v15_verify_final
0x8004: wtm_emsa_pkcs1_v15_zeroize
0x8005: wtm_emsa_pkcs1_v15_sign
0x8006: wtm_emsa_pkcs1_v15_sign_init
0x8007: wtm_emsa_pkcs1_v15_sign_update
0x8008: wtm_emsa_pkcs1_v15_sign_final
0x9000: wtm_dh_public_key_derive
0x9001: wtm_dh_shared_key_gen
```

# Talking directly to WTM



# how does lexmark rootfs key derivation *actually* work?



# wtm_unwrap_load

- Takes two arguments: u16 cipher_id and u8 *blob (0x268 bytes in size)
- cipher_id specifies what kind of key is being unwrapped

```
aes_select_hardware_key()
iv_decrypted = aes_ecb_decrypt(blob->iv, 0x10)
aes_select_hardware_key()
aes_set_iv(iv_decrypted)
body_decrypted = aes_cbc_decrypt(blob->body, 0x200)

if sha256(body_decrypted) == blob->digest:
  aes_state_zeroize()
  aes_load_key(body_decrypted[0x20:0x20+aes_keywidth])
  return 0
else:
  return 0x154
```

blob is decrypted using secret invisible key

decrypted blob integrity is checked

slice of decrypted blob is loaded into AES engine

# wtm_store_engine_context_external

- WTM firmware command list has some comands that mention “engine context” in their name..
- Quickly reversing them reveals they are used to load/restore the register state of the HW crypto peripherals!
- And the `_external` variant will load/save the data from/to a user-supplied DRAM buffer.

```
int aes_store_context(uint8_t *out)
{
  mem_move32(out + 0x10, AES_MMIO, 7);
  mem_move32(out + 0x68, AES_MMIO + 0x58, 21);
  return 0;
}
```

# dumping unwrapped keys

```
from wtmclient import *
from Crypto.Cipher import AES

c = WTMClient()

wkey = open("wkey.bin", "rb").read()
c.scratch_write(0, wkey)

c.wtm_cmd(WTM_CMD_KEY_UNWRAP_LOAD, [MODE_AES_256_CBC, c.scratch])
c.wtm_cmd(WTM_CMD_STORE_ENGINE_CONTEXT_EXTERNAL, [MODE_AES_256_CBC, c.scratch])

blob = c.scratch_read(0, 0x100)
unwrapped_aes_key = blob[0x88 : 0x88 + 0x20]

print("Unwrapped AES key: %s" % unwrapped_aes_key.hex())
```

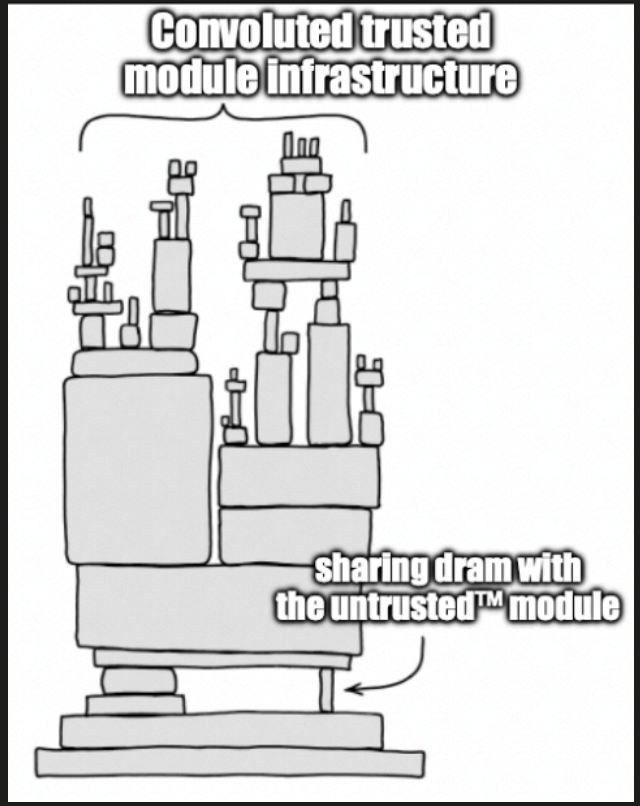
- dumping the unwrapped keys means we no longer need the printer as an oracle.
- lexmark has been changing the encrypted rootfs encryption keys, **but not the wrapped keys.**
- of course if they do change the wrapped keys we can always unwrap them again with this trick. 😊

# Getting arbitrary code execution on the WTM

- Would be nice to get arbitrary code execution on the WTM itself..
- There’s a myriad of commands we can audit to find some good bugs
- Started dreaming up some exploit strategies and weird machinery..
- But is it actually required? 🤔

# Getting arbitrary code execution on the WTM

- Since WTM on Lexmark runs linux.. surely that can’t all fit into SRAM
- So this means WTM has some dedicated DRAM?
- No. **WTM shares the DRAM with the Application Processor**!
- If we write to the right location in /dev/mem from the AP we can completely take over the WTM. 😂
- If we overwrite the handler for an existing WTM command we can get a convenient primitive that can be triggered from the AP.



# world’s dumbest implant

```
_start:
cmp     r0, #1
beq     cmd_peek32
cmp     r0, #2
beq     cmd_poke32

mov     r0, #0
bx      lr

cmd_peek32:
ldr     r1, [r1]
str     r1, [r2]
bx      lr

cmd_poke32:
str     r2, [r1]
mov     r0, #0
bx      lr
```

```
def wtm_read32(self, addr):
    self.wtm_cmd(WTM_CMD_HAX, [WTM_HAX_CMD_READ32, addr, self.scratch])
    return struct.unpack("<L", self.scratch_read(0, 4))[0]

def wtm_write32(self, addr, value):
    self.wtm_cmd(WTM_CMD_HAX, [WTM_HAX_CMD_WRITE32, addr, value])

def wtm_clear32(self, addr, value):
    self.wtm_write32(addr, self.wtm_read32(addr) & (~value & 0xFFFFFFFF))

def wtm_set32(self, addr, value):
    self.wtm_write32(addr, self.wtm_read32(addr) | value)
```

# Grabbing some WTM loot

```python
from wtmclient import *

c = WTMClient()

ROM_BASE = 0xFFE00000
ROM_SIZE = 0x00020000

rom = b""
for i in range(ROM_SIZE // 4):
    rom += c.wtm_read32(ROM_BASE + i * 4).to_bytes(4, "little")

with open("rom.bin", "wb") as f:
    f.write(rom)
```

dump_bootrom.py

```python
from wtmclient import *

c = WTMClient()

OTP_BASE = 0xD1D22800
OTP_SIZE = 0x800

o = b""
for i in range(OTP_BASE, OTP_BASE + OTP_SIZE, 4):
    o += struct.pack("<I", c.wtm_read32(i))

with open("otp.bin", "wb") as f:
    f.write(o)
```

dump_otp.py

# Closing notes

- We broke the new Lexmark firmware encryption
- We learnt quite a bit about the Marvell WTM in the process
  - Some secrets remain: how does the “hardware key” work?
  - Is it derived from OTP data maybe?
- Tacking on additional layers of obfuscation/encryption/security **retroactively** is typically futile if adversaries already have code execution capabilities on your device.
- A detailed two part blogpost will be up on https://haxx.in/
- All code will be published on https://github.com/blasty/lexmark
- No paper or ink cartridges were harmed (or utilised) in the creation of this content. 🌳🖨️


Thanks for your attention.
Questions?
Let’s chat!🍻🥃
Peter Geissler <peter@haxx.in>
HAXXIN (https://haxx.in/)