Master System and Network Engineering

University of Amsterdam

Cybercrime and Forensics Project

Forensic analysis of Chromecast and Miracast devices

Peter van Bolhuis
peter.vanbolhuis@os3.nl

Cedric Van Bockhaven
cedric.vanbockhaven@os3.nl

March, 2014

Abstract

Google's Chromecast and Miracast dongles are gadgets that allow people to stream movies and other media content to an HDMI-capable device. This paper atempts to find out what forensically interesting information is stored on both devices and how this information can be retrieved.

The Chromecast makes it difficult to obtain the NAND memory by encrypting the contents with a unique per device key. Attempts to retrieve this memory failed. However, contents of crash logs show that information with absolute timestamps is logged on the device. It suggests that information, including wireless access points and MAC addresses, is logged.

The Miracast dongle researched during this project, the Measy A2W, runs the EZCast firmware. EZCast contains multiple software vulnerabilities that may lead to shell access on similar devices. However, the method used to access the memory and flash on the Measy A2W was via the UART interface. A memory dump could then be retrieved over the network using `netcat` or via UART using `hexdump`. The memory of the Miracast contained information including partial images, links of visited websites, and MAC addresses of nearby and connected devices.

Contents

1 Introduction

HDMI dongles enable everyone to watch streaming content on their television. The Google Chromecast and the Measy Miracast are two of these devices. From a forensic point of view, these devices are interesting, given the fact that they are likely to contain information about the watched videos (e.g. cached fragments of media).

At the moment of writing, there is no public information about the forensic properties of these devices. This is why this research project sets out to find out what information is stored on the Google Chromecast and the Measy A2W Miracast. The research question is defined as:

What data can be extracted from Chromecast/Miracast devices that can be used in forensic cases?

To answer the research question, the following subquestions have been defined:

- How do both devices protect against access to the filesystem? In which cases is access to the filesystem possible?
- What data can be gathered in a live environment without modifying the device in any way?
- What data can be acquired with access to the NAND memory of the device?
- Is access to the RAM possible?

The report is divided in the explanation of the setup we used, and our findings for both the Chromecast and Miracast.

2 Research

The research comprises setting up the environment and testing both the Chomecast and Miracast devices. This section will describe the process of research on both of them, including technical details of the devices.

2.1 Infrastructure

Before starting any tests on either device, a man-in-the-middle (MITM) infrastructure was set up. Using this infrastructure, all network communications between the HDMI-device and the internet could be captured.

Using Kali Linux, a wireless access point was set up. The access point distributed IP addresses to all connected devices using DHCP and logged the traffic.

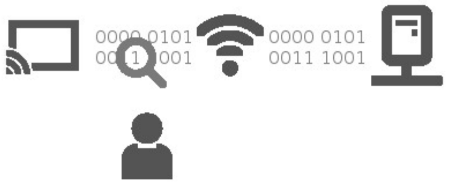

Figure 1: Schematic view of the network layout.

2.2 Chromecast

The Google Chromecast is a HDMI-dongle by Google. It enables users to stream web-content (e.g. YouTube, Netflix) to their television. During initial setup, the Chromecast sets up its own wireless network, `Chromecast????`, where the question marks are replaced with decimal numbers. The user can connect to this network with the Chromecast app. The Chromecast can then be given a name and will be configured to connect to a wireless network.

After this setup, all devices that are on the same network can *cast* to the Chromecast without any further authentication.

Table 1 Chromecast hardware specifications

Model	H2G2-42
CPU	Marvell 88DE3005 (ARMv7)
Flash	Micron MT29F16G08MAA 16 Gb (2 GB) NAND
RAM	Micron 3FE77D9PXV 512MB
Network	Azure Wave AW-NH387 802.11 b/g/n

2.2.1 Firmware binary

During the initial setup, the Chromecast was connected to the network as described in section 2.1. It proceeded to download an update[1], after which the device was ready to be used.

The downloaded update was retrieved and extracted. Closer inspection revealed that it was a squashfs filesystem. Mounting and inspecting this filesystem shows that most communications to the outside are done via a secured SSL/TLS connection. However, some binaries used HTTP for their communication, most notable the program `crash_uploader`.

2.2.2 Attack surfaces

When opened, the Chromecast shows shielded chips as can be seen in Figure 2. Underneath this shielding are pins that can be used for a UART connection to the device. This UART connection only shows output of the boot process [1].

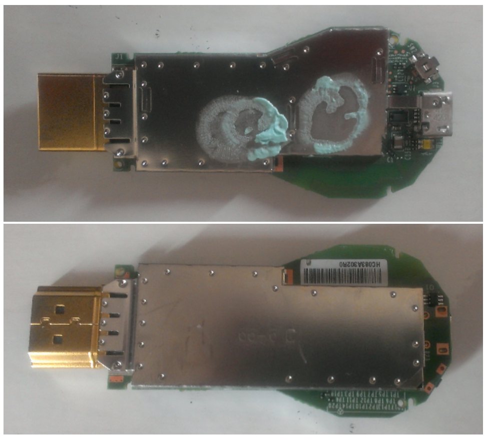

Figure 2: The inside of the Chromecast device, shielded

The device uses a 2GB flash chip for caching of the streaming videos. This chip will contain the most interesting information from a forensics point of view. However, this chip is encrypted with a unique per device key, as can be seen in the console output [2, 3].

[1]http://r9---sn-5hnezn7s.c.pack.google.com/edgedl/googletv-eureka/stable-channel/ota.16041.stable-channel.eureka-b3.4b2d484f74cc12a8ed827db43f7d76bd3a983d4a.zip

Scanning the device with `nmap` shows that it has two open ports. On port 8008 runs a webserver that controls the applications on the Chromecast. The apps can be controlled via /apps/<appname>. The webserver is also used to set up the device. Information about the device can be retrieved via /setup/.

Table 2 Scan results

Port	Service	Description
8008/tcp	http	Webserver that enables remote control of the apps on Chromecast. It can also give additional information that is used for setting up the device.
8009/tcp	ajp13	Used for the *casts* protocol [4].

2.2.3 Attacking the Chromecast

Because the hardware of the Chromecast was shielded and the memory chip encrypted, it was decided to attempt to crash the Chromecast via software attacks.

The Chromecast was connected to the network that sniffed all traffic. Next, the YouTube application was started, streaming a video. While the video was playing, a full service scan was performed with *nmap*. This service scan managed to crash the YouTube application, resulting in a crash report being sent. After the crash, the Chromecast returned to the default screen.

Analyzing the intercepted network traffic showed content being sent to a Google server over HTTP. This traffic contained a gzipped file which in turn contained the following:

- General information about the system:
  - Kernel version
  - Uptime
  - Build
- Memory info
- CPU info
- Process information (*top*)
- Running processes and threads
- Maps of all processes
- Log files
  - dmesg
  - system
  - main

The *dmesg* log file contained information about the booting of the device. All timestamps in this file were relative to the start of the boot. It also shows the layout of the NAND chip, which may be interesting if the NAND can be acquired and decrypted:

```
<5>[    0.280258] 0x000000000000-0x000000100000 : "block0"
<5>[    0.281386] 0x000000100000-0x000000900000 : "bootloader"
<5>[    0.282432] 0x000000900000-0x000001900000 : "kernel"
<5>[    0.283477] 0x000001900000-0x00001a900000 : "rootfs"
<5>[    0.284496] 0x00001a900000-0x00002d500000 : "cache"
<5>[    0.285522] 0x00002d500000-0x000075000000 : "userdata"
<5>[    0.286623] 0x000075000000-0x000078000000 : "recovery"
<5>[    0.287650] 0x000078000000-0x00007e000000 : "backupsys"
<5>[    0.288665] 0x00007e000000-0x00007e800000 : "fts"
<5>[    0.289688] 0x00007e800000-0x00007f800000 : "factory_store"
<5>[    0.291108] 0x00007f800000-0x000080000000 : "bbt"
```

Both the *system* and *main* logs contained absolute timestamps. However, the *main* log only contained information about the last few minutes. Information in this log is about starting and stopping videos and skipping to certain offsets in videos.

The *system* log contained information about the system being enabled, dating back two weeks. However, information about wireless access points, and other identifying information was not included in the sent reports. The commands used to generate the logs were included in the logs. These indicate that information about wireless networks and MAC addresses is on the flash chip (appendix B).

2.2.4 Forensically processing the Chromecast

The following steps are advised to obtain logs from a Chromecast device. Note that this method will alter some of the memory and is therefore not forensically sound.

1. Leave the Chromecast powered on. Also leave the television that the Chromecast is plugged into on.
2. Perform a man-in-the-middle on the Chromecast, for instance with ARP poisoning or a network tap. Write all captured data to a file. This will make sure that when a report is sent, it will be captured.
3. Crash the Chromecast. If YouTube is running, this may be done with an nmap service scan. At the moment of writing, no other methods are known.
4. The log file can be extracted from the captured network traffic.

2.3 Measy A2W Miracast

The Measy A2W Miracast allows to stream media content like music, video, and photos. People can stream these from their mobile devices or desktop computer, or choose to stream their screen live on TV. It's also possible to stream YouTube videos directly to the Measy A2W. There's support for the EZCast, DLNA, EZMirror, and EZAir protocols.

2.3.1 Miracast technology

Miracast is a peer-to-peer wireless screencasting standard formed via Wi-Fi Direct connections in a manner similar to Bluetooth. It enables wireless delivery of audio and video to or from desktops, tablets, mobile phones, and other devices. [5]

2.3.2 Specifications

The investigated Miracast device in this research, is the Measy A2W Miracast. Many of the techniques we will explain in this section are applicable to other Miracast devices as well, i.e. those with similar hardware and those that are running the same firmware.

Table 3 Measy A2W hardware specifications

Model	Measy A2W Miracast
Chipset	Actions-Micro AM8251
Flash	Zentel A5U1GA31ATS-BC 1Gb SLC NAND Flash
RAM	128M DDR3
Network	RTL8188EUS IEEE 802.11 b/g/n
Protocols	DLNA/Airplay/EZcast/Miracast

When using the Measy A2W dongle for the first time, the user connects to a wireless network that looks like `EZCast-????????`, where the question marks represent hexadecimal characters taken from the MAC address. The network is protected with WPA2 and the default password 12345678. The current password is displayed at all times on the TV screen. Furthermore, the Measy A2W contains two wireless interfaces. One of the interfaces is used to connect to the wireless home network, and the other acts a host access point which users can connect to. The access point subnet, 192.168.203.0/24 is routed to the other network, so it can provide internet access.

2.3.3 Firmware binary

The Miracast was approached in the same way as the Chromecast: a man-in-the-middle node was constructed that logged all the traffic going to and from the device. When the device was first turned on, it downloaded an update from the iezvu.com domain[2]. The binary was then investigated to find possible weaknesses that would grant us shell access on the device.

The binary contains a header *ActionsFirmware*, a version number *v1.17*, a checksum, and the offsets of the different firmware parts. An overview of these offsets is given in table 4. These offsets are very likely to change in future firmware versions, but may give an idea of the contents that are to be expected on the flash. The firmware details and format can be extracted using a provided utility, see appendix A.

Table 4 Firmware section offsets

Section	Start address	Length
ADECadfus	0x00000400	0x00002000
ADFUadfus	0x00002400	0x00002608
HWSChwsc	0x00004c00	0x000027a0
F648fwsc	0x00007400	0x00017570
F648mbrec	0x0001ea00	0x00001800
F648brec	0x00020200	0x00020000
RECOVER BIN	0x00042200	0x00007000
WELCOME.BIN	0x00049200	0x000ea800
LCM.BIN	0x00133a00	0x00000200
BACKLIGHT.BIN	0x00133c00	0x00000200
GAMMA.BIN	0x00133e00	0x00000600
GPIO.BIN	0x00134400	0x00000200
SYSCFG.SYS	0x00134600	0x00331200
INITRD.DAT	0x00465800	0x00200000
BOOTARG.TXT	0x00665800	0x00000200
rootfs	0x00665a00	0x02d00000
user1	0x03365a00	0x0027d000
vram	0x035e2a00	0x00005a00

Apart from the iezvu.com domain, *ActionsFirmware* was a second clue that the actual firmware was being developed by a different company than Measy. The iezvu.com domain is hosting the EZCast firmware, which is used on a variety of devices. Their WHOIS reveals that the domain is in control of

[2]http://www.iezvu.com/upgrade/ezcast/ezcast.bin

Actions Microelectronics Co., Ltd.. This is the same company that developed the on-board AM8251 chip.

The binary contains several file systems:

- INITRD.DAT: an initial RAM disk (ext2, recovery) which then loads the rootfs.
- rootfs: the root file system (ext2) on the NAND flash, which the EZCast binaries run from.
- user1: an ext2 file system that holds user settings and can persist between upgrades.
- vram: a vfat file system that holds user and device specific settings.

INITRD.DAT and rootfs both contain a Linux kernel of which the sources have not been published. This is in clear violation of the GPL.

2.3.3.1 Bundled software

The EZCast firmware runs BusyBox 1.15.1, and comes bundled with BIND 9.9.2-P1, thttpd/2.25b, and hostapd v0.6.9. Applications that take care of the screen display, and the thttpd CGI web server binaries are proprietary.

2.3.3.1.1 Software with vulnerabilities

- **thttpd/2.25b**: a directory traversal vulnerability [6], which can only be reproduced when starting thttpd from the system root, like `thttpd -d /`. In this case, `thttpd` is started from /root/html, which makes this ”vulnerability” irrelevant.

- **udhcpc/1.15.1**: the DHCP client (`udhcpc`) in BusyBox before 1.20.0 allows remote DHCP servers to execute arbitrary commands via shell metacharacters in a set of DHCP options (CVE-2011-2716 [7]). This CVE is misleading, as it’s actually the `dhclient` script that is present on many Linux systems that carelessly handles data that is passed on from `udhcpc`. The `dhclient` script is not used in the EZCast firmware. Consequently, this vulnerability can not be used to gain shell access either.

2.3.3.1.2 thttpd CGI binaries

As found on the rootfs, in /root/html/cgi-bin/:

apply.cgi
A submit handler that contains functionality to change device settings such as the access point name and password. This functionality isn't actually hooked up for the Measy A2W. Disassembling the code leads to believe that it might work for different EZCast devices, since a few conditional statements were found that look at the currently loaded system modules in `/sys/module/8192cu` and `/sys/module/rtnet3070ap`.
Applying the configuration for these devices is done by calling a shell script with the form input data as arguments. No input sanitization is being done is done on these arguments. Injecting commands into the access point name or password using shell metacharacters could then lead to shell access for these devices.

conference_control.cgi
Unknown, but not working or unfinished. Contains a "user name" and "display location" field, possibly to broadcast the current screen to a specific IP in the future.

get_my_mac.cgi
Returns the MAC address of the visitor, by resolving the `REMOTE_ADDR` environment variable supplied to the CGI binary with `arp`.

info.cgi
Returns Wi-Fi settings queried from the configuration files
`/mnt/user1/softap/RT2870AP.dat` and
`/mnt/user1/softap/rtl_hostapd_01.conf`.

myall.cgi
Returns system information: current IP and gateway, populated from `/proc/net/route`.

upload.cgi
Allows to upload files with arbitrary names, which will be stored in the /tmp directory. Interestingly enough, uploading a file with the name *helpview.jpg* or *appInfo.json* will cause a command to be sent to the socket `/tmp/domainpath`, of which the use is currently unclear.

Looking into the software that is running on the EZCast already gave us some insights on the workings. However, no shell access to the device was obtained yet, which required looking at other angles.

2.3.4 ADFU mode

ADFU mode stands for Actions Device Firmware Update and is a USB mode in which Actions-Micro devices can be flashed with new firmware. The USB driver has been made publicly available by the Actions-Micro team[3].

The Tronsmart T1000, a similar Miracast device, has two pins which can be shorted to switch to ADFU mode [8]. The Measy A2W was successfully put into ADFU mode by shorting pins TP5 and TP6. These can be found at the height of the red mark in figure 3. The Measy A2W will identify using vendor ID `1DE1` and product ID `1205`.

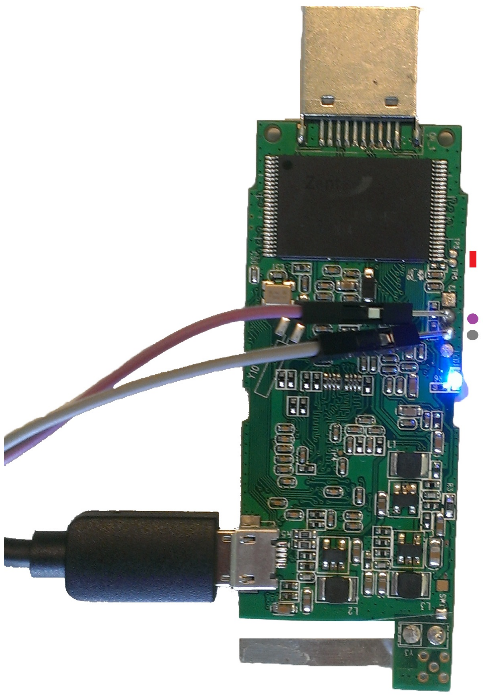

Figure 3: The Measy A2W with relevant pin markings.

Flashing new firmware would of course be disastrous for the purpose of performing a forensic analysis. However, it is not unthinkable that reading the flash would be possible using the same protocol just like writing the flash is made possible. This is supported by the schematics of the ADFU mode

[3]https://www.iezvu.com/init/ADFU_Driver.zip

as shown in figure 4, which show that the flash read driver is connected to the ADFU driver. Reverse engineering the USB protocol is an interesting possibility, but falls out of the scope of this research.

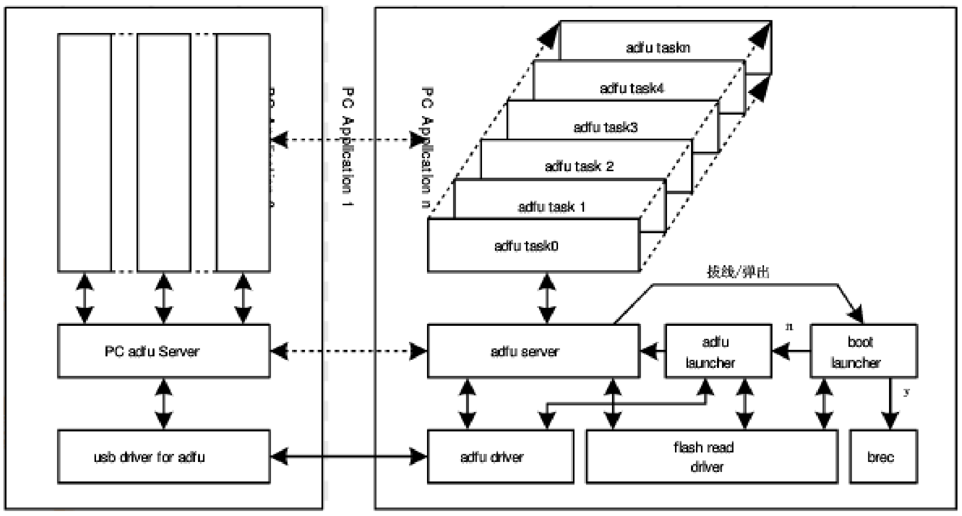

Figure 4: Schematics of the ADFU mode.[4]

2.3.5 UART

It was reported that other hardware by Actions-Micro (a digital picture frame) had a UART interface [9]. After scanning the pins on the Miracast device, this turned out to be the case as well for the Measy A2W. The Measy A2W prints debug output over UART, while also providing shell access. When the device is booted, it will print the startup output, which made it easier to detect the correct pins. In figure 3, the RX pin is marked in purple, while the TX pin is marked in gray. The other pins are VCC (square) and GND. The correct baud rate is 115200. This interface provides an unprotected root shell.

2.3.6 Reading out the memory

The EZCast firmware ships with a `curl` binary which allows to post files back to a server that is reachable over Wi-Fi. However, during tests, it loaded the file we wanted to obtain completely in memory. This is an unwanted effect as it overwrites the same memory that we would like to read out. Transferring the 128MB of memory fails, as it will try to allocate more memory than is actually available.

[4]Found at http://wendang.baidu.com/view/e5aa5b5cbe23482fb4da4c7d.html.

2.3.6.1 Using netcat over Wi-Fi

On the host side, a `netcat` listener can be set up that receives incoming files/dumps. A cross-compiled `netcat` binary can be downloaded using `curl` that is packed with the firmware. This binary can be acquired from a locally configured HTTP server.

Host side `# netcat -l 0.0.0.0 5353 > memory.dmp`

Miracast `# dd if=/dev/mem | ./netcat 192.168.203.66 5353`

Precompiled mipsel binaries for a few applications like `netcat` are provided, see appendix A.

2.3.6.2 Using hexdump over UART

A different solution is to use the `hd` (hexdump) utility which is already part of the firmware. It allows to dump the memory using the command `dd if=/dev/mem | hd`, which can then be logged over UART. While the memory is being dumped, debug output will also still be sent over the link. This means that after transferring the memory, the debug output will have to be filtered out from the dump. Converting the obtained hex output back to binary can be done using `xxd -r | dd conv=swab`. Since `hexdump` differs from `xxd` in how bytes are grouped, the endianness has to be swapped with `dd`. The impact of this technique on the memory is lower, since no external binaries have to be downloaded onto the device.

2.3.6.3 Finding MAC addresses

The Linux kernel leaves traces of MAC addresses in the memory, as documented by Minnaard [10]. A few regular expressions are proposed to carve these addresses out of memory dumps. Since user processes may generate the same structure in memory that conforms to the regular expressions below, it is desirable to only scan the relevant kernel address space. A tool is provided to carve these addresses directly from memory dumps, see appendix A.

Beacon frames

Beacon frames are sent out by access points on regular intervals to announce their presence. They contain the MAC address of the sender, and can be found in the memory using the following regex:

```
\x80\x00\x00\x00(?:\xff){6}(.{6})\1
```

Probe requests

Probe requests are sent out by Wi-Fi devices e.g. to determine which access points are in range. They contain the MAC address of the sender, and can be found in the memory using the following regex:

```
\x40(?:\x00|\x10)\x00\x00(?:\xff){6}(.{6})(?:\xff){6}
```

2.3.6.4 Carving links from the memory

The controller application (on Android or desktop) allows to browse the internet and stream the view to the projector. It's therefore interesting to see whether links can be recovered from the memory after these applications have been used. It's possible to carve links from the memory using a simple regular expression: `/https?://[^\s/$.?#].[^\s]*/iS`

During our tests we were able to find links and cookies back in the memory long after visiting them, even after browsing other websites or streaming videos. Finding specific URLs in memory doesn't necessarily mean that they were actually visited as well. They may have caused to appear in the memory in a different way (e.g. by appearing in the content of a web page). It's desirable to remove parts of the memory from the analysis in which the URLs should not appear, such as the kernel address space.

2.3.6.5 Carving images from the memory

The controller application allows to stream local photos and videos to the Measy A2W. After streaming photos to the Miracast device, closing the photo control application, and subsequently dumping the memory, we were only able to fetch (parts of) the last displayed image. The obtained images were in JPEG format or possibly M-JPEG.

Both the phone view (the images were casted from an Android phone), as the HDMI view (the projector view), were found in memory. In figure 5, an example is shown of what could be recuperated from a sample image. The phone view gives an idea of the content that was displayed, but the HDMI view only held a single stripe of yellow from the original image.

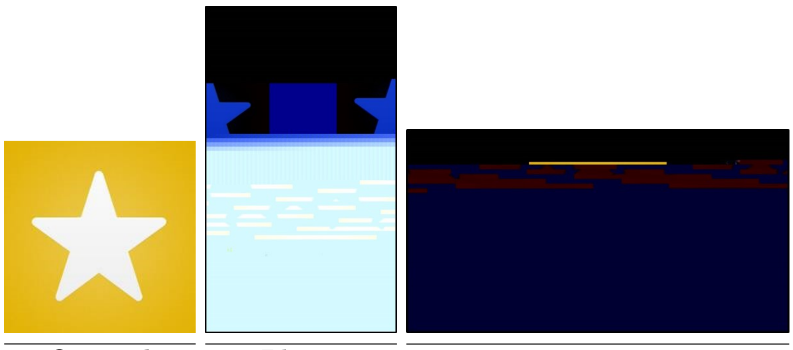

Figure 5: Memory-carved images as compared to their original counterpart.

The images were carved using the `scalpel` tool that looks for JPEG markers. Other tools may yield better results, or reveal multiple frames. At this time, we were only capable of recovering limited parts of the last shown image.

The same test was attempted for streaming videos, but led to no recoverable fragments (either video or picture frames).

2.3.7 Imaging the NAND

To image the NAND, a programmer or toolkit like NFI's memory toolkit [11] should be used with a write block. If these aren't available, a poor man's solution like the following can be used. It will not prevent writes, but it shouldn't touch any relevant files either. For the cross-compiled `netcat` and `zip` binaries, see appendix A.

Host side `# netcat -l 0.0.0.0 5353 > backup.zip`

Miracast
```
# find / \( -path /proc -o -path /sys -o -path /dev \)
    -prune -o -type f | /tmp/zip -@ -0 - |
    /tmp/netcat 192.168.203.20 5353
```

Timestamps of files that were extracted from the flash cannot be trusted: they are all either set to the compile time of the firmware binary, the Unix epoch (Jan 1, 1970), or the Microsoft DOS epoch (Jan 1, 1980).

2.3.7.1 Files of interest

The following files can be recovered from the NAND flash, and are likely to contain information that may be of interest:

/mnt/user1/dongleInfo.json
A JSON encoded file that contains the MAC address of the device, firmware version, SSID of the device and the home network, the language, the time that the device has been in use (subdivided into EZCast, DLNA, EZMirror, and EZAir sections), and if a password has been set.

/mnt/user1/softap/rtl_hostapd_01.conf
Contains the current WPA2 password of the Miracast device, which is also shown on the projector screen at all times.

/mnt/vram/edidinfo.bin
Information about the make and model of the projector device the Measy A2W is/was connected to.

/mnt/vram/wifi/latestAP.conf
Holds a reference to a file located in the same directory that contains the current SSID and password of the home network.

The system log is not saved to /var/log or /tmp, and can only be found in-memory for as long as the device is powered on.

2.3.8 Forensically processing the Measy A2W

1. If MAC addresses need to be collected from the Measy A2W, make sure to disable Wi-Fi functionality on personal devices before arriving on a scene. Leaving Wi-Fi turned on will send out probe requests, which will overwrite any MAC addresses that are in the circular array of probe requests.

2. Leave the Measy A2W powered on. If the Measy A2W uses a USB port of the television as a power source, make sure to leave the television on as well, as it might not supply power to the USB devices when turned off.

3. Carry out any grepping for data that is of most interest (e.g. finding MAC addresses or links). Any action take from this step on (including grep) will already overwrite memory.

4. Use an appropriate technique from section 2.3.6 to obtain the memory contents of the device.

5. The NAND can now be extracted, as explained in section 2.3.7. Other tools, like the NFI's memory toolkit [11] can be used as well, as long as the device isn't rebooted anymore, which will cause temporary files to be removed.

3 Conclusion

This research focused on finding out what information is stored in the flash and RAM memory of the Google Chromecast and the Measy A2W Miracast, and how these contents could be extracted in a forensically sound way.

3.1 Chromecast conclusions

When a Chromecast service crashes, crash reports will be sent over an unencrypted channel. These crash reports may contain valuable information such as the uptime of the system and scrambled log files. During our tests, these crashes could be triggered using a simple `nmap` scan. The logs contain information about the wireless networks that have been connected to and MAC addresses of devices that controlled the Chromecast. It also contains information about when the device was used with absolute timestamps. Some information was scrambled, but should be available in its original form on the NAND flash.

The Chromecast protects against access of the file system by encrypting the NAND flash chip. Access to the file system was not found possible in this research. If the flash chip is removed from the Chromecast, and the per device key retrieved, access to the chip should be possible. However, such methods were not available to the researchers. Provided the NAND has been decrypted, access to the Chromecasts NAND will contain log files and cache files. RAM access on the Chromecast was not possible during this research.

3.2 Miracast conclusions

The A2W Miracast has a debug interface that provides a root shell via UART. No changes other than removing the casing are required. This will enable full access to the device, including access to the flash and RAM memory.

The flash memory of the A2W Miracast contains the name and password of the wireless network it is connected to and the make and model of the projector the A2W is connected to. Log files are not saved persistently on the A2W Miracast and can therefore only be obtained from RAM. If the NAND chip from the A2W Miracast would be desoldered from the board, it should be easy to read the information stored on this chip using a memory toolkit.

The RAM of the A2W Miracast contained MAC addresses of connected devices and nearby devices. Furthermore, the memory contained information about visited links and (partial) images and videos. The images are saved in memory in two ways, one for the *phone view* of the image and one for the HDMI output.

Software attacks should also be possible on similar devices due to vulnerable software that is bundled with the EZCast firmware.

3.3 Future research

The flash memory of the Chromecast was encrypted with a per device key. Research should be done into retrieving and decrypting the flash. Given access to the device, it would be interesting to find out which other relevant data may be stored on the flash memory.

4 References

[1] GTVHacker. Google chromecast. Website, 2013. http://wiki.gtvhacker.com/index.php/Google_Chromecast.

[2] GTVHacker. Google chromecast console log. Website, 2013. http://wiki.gtvhacker.com/index.php/Google_Chromecast_Console_Out.

[3] DeadlyFoez. XDA - [Q] Read/Write the NAND of the Chromecast? http://forum.xda-developers.com/showthread.php?t=2602402.

[4] Justin Loutsenhizer. Google chromecast documentation. Website, 2014. https://github.com/jloutsenhizer/CR-Cast/wiki/Chromecast-Implementation-Documentation-WIP.

[5] The Free Encyclopedia Wikipedia. Miracast, 2014. http://en.wikipedia.org/w/index.phptitle=Miracast&oldid=608335427.

[6] Seclists.org. [Full-disclosure] Thttpd 2.25b Directory Traversal Vulnerablity. http://seclists.org/oss-sec/2013/q2/385.

[7] CVE Details. CVE-2011-2716 : The DHCP client (udhcpc) in BusyBox before 1.20.0 allows remote DHCP servers to execute arbitrary commands via shell metacharacters, 2011. http://www.cvedetails.com/cve/CVE-2011-2716/.

[8] China Gadgets Reviews. How to manually upgrade firmware for Tronsmart T1000, 2013. http://chinagadgetsreviews.blogspot.nl/2014/03/how-to-manually-upgrade-firmware-for.html.

[9] Matt Goring. Hack a BF801 Digital Photo Frame, 2012. http://matty.99k.org/BF801-photo-frame/.

[10] Wicher Minnaard. Out of sight, but not out of mind: Traces of nearby devices' wireless transmissions in volatile memory. *Digital Investigation*, 11:S104–S111, 2014.

[11] Nederlands Forensisch Instituut. Memory Toolkit. https://www.forensicinstitute.nl/products_and_services/forensic_products/memory_toolkit/.

Appendix

A GitHub

Some relevant tools for the Miracast were released on GitHub at http://github.com/c3c/miracast: a python script to carve MAC addresses from memory, a python script to receive files from a `curl` POST request, a C tool that extracts Actions-Micro firmware details, and cross-compiled mipsel binaries: `netcat`, `zip`, `gdb`.

B Chromecast logs

A Kernel/dmesg log

```
------ KERNEL LOG (sh -c dmesg | egrep -iv ’ ssid\|\([0-9a-f][0-9a-f]:
\)\{5\}[0-9a-f][0-9a-f]’) ------
<6>[    0.000000] Booting Linux on physical CPU 0x0
<6>[    0.000000] Initializing cgroup subsys cpu
<5>[    0.000000] Linux version 3.8.13 (mosaic-role@eurekabuild6.mtv.
corp.google.com) (gcc version 4.5.3 (gtv 20120928-afe6864) ) #3
PREEMPT Mon Mar 31 21:54:56 PDT 2014
<4>[    0.000000] CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7),
cr=10c5387d
<4>[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing
instruction cache
<6>[    0.000000] Machine: MV88DE3108, model: MARVELL BG2CD Dongle board
based on BERLIN2CD
<4>[    0.000000] mv88de3100_fixup
<4>[    0.000000] cmdline = c044bf8c
<6>[    0.000000] Add randomness with 128 bytes
<4>[    0.000000] Memory policy: ECC disabled, Data cache writeback
<7>[    0.000000] On node 0 totalpages: 78336
<7>[    0.000000] free_area_init_node: node 0, pgdat c0483ad4, node_mem_
map c06f6000
<7>[    0.000000]   Normal zone: 612 pages used for memmap
<7>[    0.000000]   Normal zone: 0 pages reserved
<7>[    0.000000]   Normal zone: 77724 pages, LIFO batch:15
<7>[    0.000000] pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768

(...)
```

B System log

```
------ SYSTEM LOG (sh -c logcat -v threadtime -d *:v | egrep -iv
'VERBOSE[123]:\|XMLHttpRequest\|ssid=\|friendlyname\|serial_num=\|\([0
-9a-f][0-9a-f]:\)\{5\}[0-9a-f][0-9a-f]') ------
--------- beginning of /dev/log/system
01-01 01:00:04.539   640   640 I boot_animation: set to 1080p
01-01 01:00:04.539   640   640 E boot_animation: MV_PE_VOutHDMIGet
SinkCaps failed: error 82000005
01-01 01:00:04.599   681   681 I Panel_main: Panel service start
01-01 01:00:04.599   681   681 I PWM     : file /dev/pwmch0
01-01 01:00:04.599   681   681 I PWM     : file /dev/pwmch1
01-01 01:00:04.599   681   681 I Panel   : start service
01-01 01:00:04.609   681   682 I Panel   : ThreadStart
01-01 01:00:04.619   681   681 I Panel   : start service
01-01 01:00:04.629   681   686 I Panel   : ThreadStart
01-01 01:00:04.629   681   686 I Button  : Run Service
01-01 01:00:04.979   679   679 I wpa_supplicant: Successfully
initialized wpa_supplicant
01-01 01:00:04.979   679   679 I wpa_supplicant: rfkill: Cannot open
RFKILL control device
01-01 01:00:05.919   691   691 I NetMgr  : Net_Mgr starting
01-01 01:00:05.959   691   691 I HotspotManager: success get Mac
address from cert
01-01 01:00:05.959   691   691 E WifiUtil: scan failed, response:
FAIL-BUSY
01-01 01:00:05.999   703   703 I extract_custk: /data/chrome/custk.bin
already exists
01-01 01:00:06.039   705   705 D dhcpcd  : version 5.2.10 starting
04-26 22:04:58.529   691   709 I WifiMonitor: Start monitor thread
04-26 22:04:58.539   691   709 E WifiUtil: scan failed, response:
FAIL-BUSY
04-26 22:05:00.399   705   705 D dhcpcd  : no interfaces have a carrier

(...)
```

C Main log

Note that this log is a part of the System log output, appendix B.

```
--------- beginning of /dev/log/main
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::currentTime
```

```
- seeking, returning 2850.000000
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::play()
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::playInternal
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::currentTime
- seeking, returning 2850.000000
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::invalidate
CachedTime
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::currentTime
- seeking, returning 2850.000000
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::updatePlay
State - shouldBePlaying = true, playerPaused = true
05-11 13:38:35.461  1697  1697 I eureka_shell: HTMLMediaElement::invalidate
CachedTime
05-11 13:38:35.461  1697  1697 I eureka_shell: [1699:1715:INFO:wifi_util
.cc(428)] Wifi metrics: playing=1, signal=60, noise=83, snr=23
05-11 13:38:35.461  1697  1697 I eureka_shell: [1699:1699:ERROR:power_
save_blocker_eureka.cc(20)] Not implemented reached in content::Power
SaveBlockerImpl::PowerSaveBlockerImpl(content::PowerSaveBlocker::Power
SaveBlockerType, const std::string&)
05-11 13:38:35.471  1697  1697 I eureka_shell: [12:17:INFO:chunk_demuxer
.cc(820)] Seek(2850)
05-11 13:38:35.471  1697  1697 I eureka_shell: [12:17:INFO:chunk_demuxer
.cc(583)] Seek
05-11 13:38:35.471  1697  1697 I eureka_shell: [12:17:INFO:chunk_demuxer
.cc(583)] Seek
05-11 13:38:35.531  1697  1697 I eureka_shell: [12]    16634 ms:
Scavenge 12.5 (29.9) -> 11.1 (30.9) MB, 1 / 19.8 ms (+ 156.4 ms in 21
steps since last GC) [Runtime::PerformGC].
05-11 13:38:35.581  1697  1697 I eureka_shell: [1699:1699:INFO:CONSOLE(0
)] "The page at 'https://XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX was
loaded over HTTPS, but ran insecure content from 'http://XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X this content should also be loaded over HTTPS.

(...)
```