Abstract

Our research focused on creating a methodology which uses open source tools to make an automatic forensic imager. A choice was made to use generic Linux system tools as much as possible. This way our model can be implemented on different types of hard- and software. In our method we’ve focused on doing an acquisition to an external USB device or transferring the forensic data to a remote server using SSH. In order to implement our method we’ve built a proof of concept using a Raspberry Pi 2 model B development board. Building a proof of concept resulted in a working device which is able to automate the acquisition process.

Introduction

In order to do a forensic investigation, an investigator has to be well trained and prepared. In most cases he has to take expensive equipment with him. To make a forensic image one could use a laptop with a write blocker and a specific forensic operating system. Setting up the acquisition tools takes time and requires specific knowledge. A cheap automatic forensic imager can make the acquisition phase within hand’s reach for less trained people.

Research Questions

The goal of the research is to answer the following questions:

- What are generic solutions to automate interface activity?
- What are ways to implement software based forensic acquisition?
- Is it possible to use a Raspberry Pi 2 model B as an portable automated forensic imager?

Related Work

A write blocker prevents evidence data being altered during the acquisition process. There are hardware and software based write blockers[1, pp. 47-48]. Standards are defined to test the forensic reliability of software based write blockers [2] [3]. Both proprietary and open source tools are available [4].

Polstra in his article [5] provides a cheap solution to prevent write operations by constructing a hardware write blocker using a development board. Also, Polstra created a USB Deck (UDeck)[6] which is a set of scripts for the BeagleBone Black[7] to allow write blocking. Digital Forensics Investigation Research Laboratory (DFIRE labs) provide their own solution called FIREBrick[8]. FIREBrick is an open source solution, which can be assembled from mass-produced components for around $199.

To conclude, none of the above described solutions are fully automated and most of them focus on write blocking instead of the acquisition process.

Method

First, we decided to implement two ways of acquisition: USB(input) to USB(output) and USB(input) to network(output). The Raspberry Pi 2 model B has four USB ports and one Ethernet port. We specified the two right USB ports as input. The two left USB ports and the Ethernet port are specified as output. An illustrative image is shown in Figure 1 [9].
Figure 1: Usage of USB ports in Raspberry Pi 2 model B [9].

Figure 2 gives an abstract view of the separate parts that need to be build in order to reach an automated acquisition process. There are three events: "Network detection", "USB output detection" and "USB input detection". All of them start the execution of a script. This script checks if the prerequisites are met to do either an "USB to network" or an "USB to USB" image acquisition.

choices

Since the proof of concept needed to be mobile it was important to use a lightweight operating system which uses less resources. Our choice was Arch Linux [10]. It is known for its low resource footprint and is actively supported on the Raspberry Pi 2 model B [11]. By default, Arch does not auto mount any block device.

We wanted our solution to be modular and supported on multiple platforms. Aside from the hardware, we wanted it to be transferable to other Linux operating systems. Therefore, a choice was made to use Bash as the main scripting language.

An acquisition of a disk needs to be forensically correct. One of the main goals was to use open source software. Available software such as dfldd[12] and dc3dd[13] will calculate the hash during the image acquisition. Both of those programs are based on the dd program. In order to make our solution more future proof, we decided to use dc3dd. Dc3dd is a patch of the current version of dd and for that reason, it will be updated when a new version of dd is available [14].

Figure 2: Abstract program model.
In order to transfer the data to a remote location different techniques can be used. The main requirement for the chosen solution was that it needed to be forensically correct. We decided to use SSH to transfer the data to a remote server.

**Workflow**

As shown in Figure 2, there are three possible program workflows:

- USB input plug in
- USB output plug in
- Network plug in

Figure 3 shows the program flow when an USB input device is plugged in. In this case symlink ”evidence” is going to be created to specify this device. Next, the system will check for the existence of a USB output device. If it is connected, then an ”USB to USB” acquisition is started. If not, then check for a network connection. If a network connection is present, then start an ”USB to network” acquisition. And finally, if there is no output present, end the program flow.

![Workflow Diagram](image)

Figure 3: Scenario of USB input plug in.
Figure 4 illustrates the scenario when a USB output device is plugged in. In this case symlink "output" is going to be created to specify this device. After that, the system will check for the existence of the "evidence" symlink. If it exists, then start an "USB to USB" acquisition. If not, end the program flow. Figure 5 demonstrates the program flow when a network connection is initiated. It creates a file "net_output" as an identification of a working network connection. The next step is to check the existence of the "evidence" symlink. If it exists, start an "USB to network" acquisition. If not, end the program flow.

Implementation

In order to implement the abstract model from Figure 2, some research was done on the possibilities of generic Linux system tools. According to the model, a method of event handling was needed. For example, after plugging in a USB device a script needs to be executed. This section describes how the first part of the abstract model was implemented.

udev

As mentioned, we divided USB ports into two groups. In order to make a distinction between the USB ports, a low level solution is applied to separate the USB interfaces. In Linux, udev is used to do generic kernel device management [15]. The paths for the USB interfaces, handled by udev, are displayed in Listing 1.

```
/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.2
/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.3
/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.4
/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.5
```

Listing 1: Device paths used for determine the four different USB interfaces
The paths stay consistent after a reboot. Those paths can describe different block devices which are eventually listed in the `/dev` directory.

A solution we propose is the use of defined symlinks in the `/dev` directory. Depending on the device path, we can assure that certain devices are connected on either one of the left or right USB interfaces [16]. The udev rules, which are implemented in our project, are documented in Appendix A.

A second usage of udev rules is the execution of a script. When a certain device is plugged in, it creates an event. According to our model, we need to check if multiple peripherals are connected. During research we found that executing a script using udev rules can become unstable. Listing 2 shows that scripts which run more than 2 minutes will be killed by a watchdog. Another issue is that block devices, mounted by those scripts, don’t appear in the user space environment.

As a solution, systemd was used to execute a script. This solution makes the execution of the script reliable. The process is also better to debug using the general systemd tools [17] [18].

dhcpcd

By default, Arch uses the systemd-network daemon to get an IP address using DHCP. However, systemd-networkd is not modular and is hard to customize. We made an approach to disable systemd-networkd and use the `dhcpcd` program, which allows the usage of so called ‘hooks’. Those hooks act as triggers and allow to execute external scripts. In order to be uniform with the program flow, systemd is used to start the script which checks the available peripherals. The dhcpcd-hooks file can be found in Appendix B.

systemd

All of the device events end in executing `systemctl` to start a service. This service is a placeholder to execute a script. The executed script checks the available peripherals in order to decide which type of acquisition needs to be done. The unit file of this service is shown in Listing 3.

```
[Unit]
Description=Execute the /root/scripts/symlinks.sh

[Service]
Type=simple
ExecStart=/root/scripts/symlinks.sh /dev/tty1
```

Listing 3: Placeholder systemd service file which executes a simple script

summary

The abstract model from Figure 2 is implemented by using generic system tools, available in most Linux distributions. Figure 6 illustrates the different parts of the abstract model. Depending on the outcome of the last script, called “symlinks.sh”, an image acquisition script will be executed. The “symlinks.sh” script is documented in Appendix C.
Image acquisition

The following section describes the acquisition process. Depending on the first part of the model (see Figure 2 and 6), an acquisition script needs to be started. Two scripts were needed: one that transfers the image to a server and one that transfers the image to an external USB storage device.

USB to USB acquisition

The implementation of an "USB to USB" acquisition is shown in Appendix D. The program will check the free space on the USB output device. The USB output device is mounted before the acquisition and unmounted afterwards.

The first step of the acquisition process is calculating a SHA-256 sum of the USB input device. Before acquisition is started, the directory where the data will be stored is created. The name of the directory is the start date and time of the acquisition process together with the serial number of the USB input device. Then \texttt{dc3dd} is started. Information about the USB input device and the log of \texttt{dc3dd} are stored in the same directory as the image. Device information contains manufacturer name, serial number, capacity, etc. The log contains start and end time of the acquisition and hash of the image. The final step is a hash comparison of the USB input device and the image, taken by \texttt{dc3dd}, to make sure that the image is transferred correctly without changing the data.

USB to network acquisition

The script that does a "USB to network" acquisition can be found in Appendix E. To communicate with the remote machine we chose SSH. At first, check the free disk space on the remote location. If we have enough free space, we can execute the acquisition process.

The flow of the acquisition process will be similar with the "USB to USB acquisition" until the execution of \texttt{dc3dd}. The difference is that as an output device we use a SSH connection. On the other side of the SSH connection we also start \texttt{dc3dd} and use the SSH data as input of the \texttt{dc3dd} process. Listing 4 shows part of the script, where \texttt{dc3dd} is used with SSH. The code is adjusted to make it easier to read.

```
\texttt{dc3dd if=/dev/evidence hash=sha256 | ssh ${username}@${host} ‘dc3dd of=images/${SERIAL}.img log=images/${SERIAL}.log hash=sha256’}
```

Listing 4: Acquisition and transferring data over network using SSH

The output is stored in the directory which was made before. As discussed in "USB to USB acquisition", we also store the basic information of the USB input device in the output directory. This also counts for
the log information from the \texttt{dc3dd} process.

After the acquisition process three hashes need to be compared. The first hash is the one which was calculated before the acquisition process. Both \texttt{dc3dd} processes, local and remote, also have a hash which was calculated during the acquisition process. If all of the three hashes are the same, a conclusion can be drawn that the data is not altered during the acquisition and transfer. The hash and the outcome of the comparison is shown on the TFT screen.

\textbf{dc3dd}

\textit{Dc3dd} displays the progress of an ongoing acquisition job. The screen we’ve used to display is too small to fit the output of \textit{dc3dd}. In order to fit the output on the 1.8 inch screen we adjusted some print statements in the source code of \textit{dc3dd}. We described each change in detail in order to be open in the changes we made. Line numbers are included in the code below. The first change is the amount of characters we have available on our screen. Changes are shown in Listing 5.

\begin{lstlisting}[language=C]
1189   pthread_mutex_lock(&reporting_lock);
1190   fprintf(stderr, "\r");
1189   pthread_mutex_lock(&reporting_lock);
1190   fprintf(stderr, "\r");
\end{lstlisting}

Listing 5: Changed the amount of characters on the screen

The first part of the printing statement is the amount of bytes it transferred and the human readable presentation of it. Since the percentage is also available and takes less characters we use that instead of the amount of bytes. Listing 6 shows the lines which were commented.

\begin{lstlisting}[language=C]
1192   sprintf(stats, "%12"PRIuMAX" bytes ( %3s ) copied"),
1193          task->input->bytes_input,
1194          human_readable(task->input->bytes_input, hbuf, human_opts, 1, 1));
\end{lstlisting}

Listing 6: Commented the first part of the progress output

Listing 7 and 8 show minor changes which colors the output to make reading easier.

\begin{lstlisting}[language=C]
1203   fprintf(stderr, " ( %2.0f\% )", percent_complete);
1203   fprintf(stderr, "\033[38;5;226m%2.0f\%\033[39m", percent_complete);
\end{lstlisting}

Listing 7: Adjusted the code to print colored output

\begin{lstlisting}[language=C]
1243   fprintf (stderr, _(" %4.0fs \033[38;5;226m%s/s\033[39m"),
1243          bytes_per_second, final ? "\n" : "\r");
\end{lstlisting}

Listing 8: Adjusted the code to print colored output

\textbf{Screen}

The Raspberry Pi has a possibility to attach different modules. A decision was made to attach a screen in order to see the status of the acquisition process and some additional information. We attached a screen,
as shown in Figure 7, with a resolution of 128x164 pixels (1.8 inch). To enable the screen we downloaded TFT drivers from Github[19] and updated the kernel with the rpi-update[20] script.

![Figure 7: Proof of concept with screen.](image)

**Conclusions**

We’ve researched the possibilities of using existing open source tools as a way of acquiring forensic images. Acquisition data can be written to a local connected USB device or to a remote location using a network connection.

In order to automate the acquisition process we created an event based model. We used **udev**, **dhcpcd** and **systemd** as tools to automate the process. Those tools can be found in different Linux distributions and therefore our solution can be implemented on multiple Linux based operating systems.

We implemented our model on a Raspberry Pi 2 model B as a proof of concept. Our proof of concept shows that it is possible to use low-cost off-the-shelf hardware as an open source portable automated forensic imager.

**Future work**

The current bottleneck on using the Raspberry Pi 2 model B is the USB controller[21] or its implementation. At most a speed of 4.7 MB/s is reached to use **dc3dd** with. We’ve tested this on a 8 GB USB thumbdrive which took around 27 minutes to image. A 32 GB USB thumbdrive took 106 minutes. We can conclude that a 512 GB hard disk will take around:

\[
\frac{512}{4.7} \times 106 \approx 28.3 \text{ hours}
\]

Using different hardware could speed up the process.

One has to know for sure that the used software is downloaded from an authentic source. During the development only the image of the operating system was PGP signed. As for dc3dd, rpi-update and fbTFT they still need to be verified. The dc3dd package is worth mentioning. The authentic source code seems only available via SourceForge [13]. We expected differently from a forensic tool. A different solution would be doing a code review on the used open source tools. As for the TFT drivers, they are in the "staging" process to be accepted in the Linux kernel.
References


Appendix A

```bash
# Making symlinks for USB output devices (for writing the acquired disk images)
ACTION=="add", DEVPATH="/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.[23]/*",
  -> SUBSYSTEMS="block", ATTR{partition}="1", SYMLINK="usb_output",
  -> RUN="/usr/bin/systemctl --no-block start check_links"

# Making symlinks for USB input devices, also called evidence device.
ACTION=="add", DEVPATH="/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.[45]/*",
  -> SUBSYSTEMS="block", ATTR{removable}="0|1", SYMLINK="evidence",
  -> RUN="/usr/bin/systemctl --no-block start check_links"
```

Listing 9: Udev rules, for detecting separate USB devices on left and right interfaces

Appendix B

```bash
# Small hook script that makes an empty file in /dev/ called 'net_output'
# if interface goes down the file is removed.
#
OUT=/dev/tty1

case "${reason}" in
  CARRIER)
    if [ ${ifcarrier} == "up" ]; then
      echo -e "\033[48;5;87m\033[38;5;0mnet interface up\033[39m\033[49m" > $OUT
    fi
  ;;
  BOUND)
    if [ ${ifcarrier} == "up" ]; then
      echo -e "\033[48;5;87m\033[38;5;0m>$(ip addr show eth0 | egrep 'inet' | awk '{print $2}')\033[39m\033[49m" > $OUT
      if [ ! -e "/dev/net_output" ]; then
        touch /dev/net_output
        # execute script here
        /usr/bin/systemctl --no-block start check_links
      fi
    ;;
  NOCARRIER)
    if [ ${ifcarrier} == "down" ]; then
      echo -e "\033[48;5;87m\033[38;5;0mnet interface down\033[39m\033[49m" > $OUT
      if [ ! -e "/dev/net_output" ]; then
        rm /dev/net_output
      fi
    ;;
esac
```

Listing 10: dhcpcd hooks implementation
Appendix C

Listing 11: Check the symlinks to decide which acquisition script needs to be executed
#!/bin/bash

# £1 --> output device (/dev/ttyX or /dev/pts/X)
OUT=$1
USB_OUT=/dev/usb_output
MOUNT_DIR=/mnt
HASH_MET="sha256"

MSG_CLR_ok='\033[32m' #Green fg
MSG_CLR_er='\033[31m' #Red fg
MSG_CLR_im='\033[44m' #Blue bg
RST_CLR_fg='\033[39m' #Reset fg
RST_CLR_bg='\033[49m' #Reset bg

function checkDir() {
    if [ ! -d "*/mnt/images" ]; then
        mkdir /mnt/images
        echo -e "${MSG_CLR_ok}>Images dir created${RST_CLR_fg}" > $OUT
    else
        echo -e "${MSG_CLR_ok}>Images dir exists${RST_CLR_fg}" > $OUT
    fi
}

function localTransfer() {
    SERIAL=$(udevadm info /dev/evidence | egrep 'ID_SERIAL_SHORT' | cut -d '=' -f2)
    DATE=$(date +%Y_%m_%d_%H_%M_%S_)
    HSH_bef=$(sha256sum /dev/evidence | cut -d ' ' -f1)
    mkdir /mnt/images/$DATE
    echo -e "${MSG_CLR_ok}Start dc3dd${RST_CLR_fg}" > $OUT
    dc3dd if=/dev/evidence hash=${HASH_MET} of=/mnt/images/$DATE
    echo -e "${MSG_CLR_ok}Gather info of${RST_CLR_fg}" > $OUT
    udevadm info --name=/dev/evidence > /mnt/images/$DATE
    echo -e "${MSG_CLR_ok}Imaging finished${RST_CLR_fg}" > $OUT
    sync
}

function checkFreeSpace() {
    EVIDENCE_bytes=$(lsblk -bd /dev/evidence | awk '{print $4}')
    USB_OUT_bytes=$(df -B 512 $(df -B 512 USB_OUT) | awk '{print $2}')
    MB=$((1024*1024))
    NEEDED=$((($EVIDENCE_bytes+$MB)))
    echo -e "${MSG_CLR_ok}Check output space${RST_CLR_fg}" > $OUT
Listing 12: Acquire forensic image and store it on a external USB device
#!/bin/bash
# £1 --> output device (/dev/ttyX or /dev/pts/X)
OUT=$1
USER="<USER>", "<SERVER_IP>"
SERVER_DIR="images"
HASH_MET="sha256"
MSG_CLR_ok='\033[32m'
#Green fg
MSG_CLR_er='\033[31m'
#Red fg
RST_CLR_fg='\033[39m'
#Reset fg
MSG_CLR_im='\033[44m'
#Blue bg
RST_CLR_bg='\033[49m'
#Reset bg

function checkFreeSpace() {
  EVIDENCE_bytes=$(lsblk -bd /dev/evidence | awk '{print $4}' | egrep '^\[0-9\]*$')
  SERVER_OUT_bytes=$(($(eval "ssh ${USER}@${SERVER} 'df -B 512 ${SERVER_DIR}'" | awk '{print $2}' | egrep '^[0-9]*$')*512))
  SERVER_OUT_free=$(($(eval "ssh ${USER}@${SERVER} 'df -B 512 ${SERVER_DIR}'" | awk '{print $4}' | egrep '^[0-9]*$')*512))
  MB=$((1024*1024))
  NEEDED=$((${EVIDENCE_bytes}+${MB}))
  echo -e "${MSG_CLR_ok}Check output space${RST_CLR_fg}" > $OUT
  echo -e "${MSG_CLR_ok} Check size${RST_CLR_fg}" > $OUT
  #echo "needed £{NEEDED}"
  #echo "size £{SERVER_OUT_bytes}"
  if [ "${NEEDED}" -lt "${SERVER_OUT_bytes}" ]; then
    echo -e "${MSG_CLR_ok} big enough${RST_CLR_fg}" > $OUT
    return 1
  else
    echo -e "${MSG_CLR_er} error size${RST_CLR_fg}" > $OUT
    return 1
  fi
  echo -e "${MSG_CLR_ok} Check free space${RST_CLR_fg}" > $OUT
  #echo "needed £{NEEDED}"
  #echo "available £{SERVER_OUT_free}"
  if [ "${NEEDED}" -lt "${SERVER_OUT_free}" ]; then
    echo -e "${MSG_CLR_ok} enough free${RST_CLR_fg}" > $OUT
    return 1
  else
    echo -e "${MSG_CLR_er} error free${RST_CLR_fg}" > $OUT
    return 1
  fi
  return 0
}

function doProgram() {
  SERIAL=$(udevadm info /dev/evidence | egrep 'ID_SERIAL_SHORT' | cut -d '=' -f2)
  DATE=$(date +%Y%m%d%H%M%S)
  ping -c1 ${SERVER} &> /dev/null
  if [ $? -eq 0 ]
  then
  #other code
  fi
}

Appendix E
Listing 13: Acquire forensic image and store it on remote location using SSH