A System Emulation for Malware Detection in Routers

Tran Nghi Phu, Ngo Quoc Dung, Le Van Hoang, Nguyen Dai Tho, Nguyen Ngoc Binh

Abstract**: *Nowadays, there are many discussions on the fourth industrial revolution with a combination of real physical and virtual systems (Cyber Physical Systems), Internet of Things (IoT) and Internet of Services (IoS). Along with this revolution is the rapid development of malicious code on IoT devices, leading to not only the risk of personal privacy information leaking but also the risk of network security in general. In this paper, we propose C500-toolkit, a novel tool for malware detection in Commercial-off-the-shelf routers, based on dynamic analysis approach. The main contribution of C500-toolkit is to provide an environment for fully emulating router firmware image including both operating system and web-interface. To show the advantage of C500-toolkit, experiments of this tool with embedded malwares Linux/TheMoon and Linux/Mirai are presented.***

Keywords**: *IoT security, Router security, firmware, C500-toolkit.***

I. INTRODUCTION

The term Internet of Things (IoT) has become popular in recent years and it is the main core of the forth industrial revolution. Kevin Ashton defines “Internet of Things” as an integration of sensors and embedded controllers inside devices that are connected through the Internet, wire or wireless [1]. In this trend, object and human are given an identification and able to exchange related data, information each other without the intervention of human or computer. IoT devices such as smartTVs, router, ipCamera, lamps, microwave oven and so on are becoming more and more intelligent. These smart things can communicate with each other to exchange data to perform their intended functionality. According to Cisco’s report [2], 50 billions IoT devices will connect to Internet by 2020 around the world opening significant opportunities for a large number of new applications that promise to improve the quality of our lives. Because smart things can receive privacy-sensitive information from their sensors such as user’s location, user’s activities, with whom this user is talking, or carry out a safety critical function such as actuators that lock the front door, errors in the firmware of devices, whether present result in an accidental mistake or purposeful malice, can have serious and varying implications in both the digital and physical world [3]. Therefore, in parallel with the development of IoT technology, there are many security problems such as disruption to operation, information leaking or, in some scenarios, even loss of lives [4] when some smart thing can become a spying device to collect information and interact with us anytime.Recent recorded attacks show that these scenarios become more and more critic. In September 2016, an IoT botnet built from the Linux/Mirai malware was responsible for a 600 Gbps DDoS attack, perhaps the largest botnet on record, targeting Brian Krebs’s security blog [5]. Another case that could be mentioned is Weeping Angel [6], one program used Samsung smart-televisions as secret listening devices. According to the Wikileaks news, even when it appears to be turned off, the television “operates as a bug, recording conversations in the room and sending them over the Internet to a covert C.I.A. server”. There exist a lot of vulnerabilities that attackers can use for getting privileges of IoT devices and OWASP has identified top ten issues [7] in which insecure firmware, insecure web interface and insufficient authentication are mentioned. These issues have been attracting much attention from researchers who are dealing with IoT network devices. Inside IoT network, network devices such as router, switch, IoT gateway play a important role in transmitting data between devices. To secure these IoT devices, especially routers, works presented in [8, 9, 10, 11, 12, 13] allow identifying vulnerabilities/malware in commercial off-the-shelf (COST) network devices. There are two main classes in firmware analysis: static and dynamic approach. Static approach, consists of analyzing and evaluating without executing them, uses techniques such as Data Flow Graph (DFG), Control Flow Graph (CFG), Symbolic Execution (SE) to analyze every single file found in firmware and identify malware characteristics [13] such as Printable-Strings-Information (PSI), bytecode, headers, system-calls, API etc. For example, Drew *et al.* [9] perform static analysis on embedded systems with FIE based on symbolic execution engine to detect bugs in firmware for the popular micro-controllers MSP430 family. This method is precise for a small range of open source firmware programs of 16-bits RISC processors, but it can not be applied to large-scale analysis for IoT devices in general. To deal with this problem of scalability, Shoshitaishvili *et al.* [3] presented Firmalice, Costin *et al.* [14] have built an automated framework for analysing router’s firmware images. Firmalice is a static analysis framework built on top of a symbolic execution engine to identify authentication bypass flaw that attackers could determine the required inputs to execute privileged operations. This framework could analyze firmwares on the binary level, in a scalable manner and with no requirement to instrument code on the original device. While Costin’s framework scales to thousands of firmware images but it is

Revised Manuscript Received on September 03, 2019

***** Correspondence Author
Tran Nghi Phu*, VNU University of Engineering and TechnologyInstitute of Informatics, Hanoi, Vietnam.

Retrieval Number J99090881019/2019©BEIESP
DOI: 10.35940/ijitee.J9909.0981119

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

very generic and produces a large number of false positives. The main drawback of these methods is it cannot be used for malware detection purpose and firmwares containing complex obfuscation code. Therefore, using only the static analysis approach could not be sufficient to identify malware and should be combined with dynamic analysis [4].

Dynamic approach consists of monitoring the whole device during its run-time to detect abnormal behaviors. The principle of this method is to define a set of rules that are considered normal behavior to determine whether an observed device signals intentionally violates the these predefined rules. To perform this approach, the most important parts were to build a virtual emulator for running firmware. In such environment, malware can only affect the virtual PC and not the physical one. In general, researchers used QEMU [15], a very popular open-source system emulator, to deal with emulator challenges. Indeed, QEMU supports many types of architecture processor such as Intel ATOM, X86, ARM, MIPS, PowerPC that are widely used in IoT devices.

In this perspective, Jonas *et al*. have built the Avatar framework to analyze embedded devices by orchestrating the execution of a QEMU based emulator and hardware of that embedded device [10]. To analyze a firmware, Avatar connects to the target device through the communication channel or the debugging link such as Jtag and Uart to the analysis model, in which QEMU was used for emulating its CPU. Then, every CPU operations will be observed and analyzed to detect abnormal behaviors. The offloading execution of firmware to actual hardware reduces the gap between the simulation and the reality. This point gives Avatar the possibility to get real CPU operation signals during its run-time that a native QEMU based emulator cannot do. However, a lot of low-cost embedded devices do not have standard debugging or communication channels as required. Besides, the data transmission between the actual device and the analysis module through Jtag or Uart is slow. Therefore, using Avatar for real-time malware detection on IoT devices is impossible.

Sandboxes are used as a core of IoT malware analysis and detection frameworks such as [16, 17] aiming at extracting behaviors of a targeted file during its run-time. However, these frameworks exist flaws that limit the malware detection capability. IoTPot [17] has built a sandbox to capture and analyze Telnet behaviors of IoT malware that are used for DDoS attacks. This approach could be useful for detecting network abnormal behaviors, but can’t detect malware that behave mostly inside the operating system of the device such as Linux/TheMoon [18]. Rare [19] focused on how to activate malware on Router by discovering static and dynamic information to build a suitable environment for malware and Chang *et al*. [16] proposed IoT sandbox which can support 9 kinds of CPU architectures including ARM (EL, HF), MIPS, MIPSEL, POWERPC, SPARC, x86, x64 and sh4. But they only used OpenWrt to build emulated router, didn’t emulate NVRAM and didn’t mention to detect malware on router.

Current sandboxes have been mostly built on basic environments including common Linux-based operating system, additional tools without the IoT devices specificity. This point has a strong impact to capture behaviors of a target executable file and the whole detection process afterwards. But with the firmware, it is built and packaged for specialized devices, with specialized functions in many different environments that don’t have much in common. Therefore, many programs that cannot be executed on a standard environment, we also can’t install additional required environments but need to create an environment like real firmware, which means that the sandbox environment needs to be very diverse and like the firmware Real equipment. That means, with IoT sandbox we not only create an environment but have to create a lot of IoT sandbox environment, each sandbox has a close environment with real devices. The installation of these environments is not feasible due to the complexity, and they themselves are packaged in unpublished firmware. To our knowledge, there is not any sandbox currently that has the ability to build an environment based on the actual device firmware, able to simulate the device’s NVRam and gather enough syscall to serve the malware detection.

Another well-known method to deal with vulnerabilities detection purpose in large-scale is Firmadyne [12] proposed by Chen *et al.*. Firmadyne aims at emulating router firmware’s web-interface using QEMU. Firmadyne is able to auto-configure a suitable emulation environment for a wide range of routers that allows to performing dynamic analysis of 23,035 firmware images gathered from 42 device vendors. This method does not rely on physical hardware to perform the analysis like Avatar but Firmadyne emulate perfectly the firmware non-volatile memory to execute the firmware web-interface. Once the router web-interface is emulated, the popular scanning framework Metasploit is used for exploring vulnerabilities and its corresponding exploits. Firmadyne is the best tool available currently that can emulate the actual device firmware with simulating NVRam. However, Firmadyne analyzes only the web interfaces by scanning from outside and ignore behaviors of the firmware operating system. Hence, these methods can not collect abnormal behaviors to detect malware and shows in the analysis of Linux/TheMoon and Linux/Mirai malware experimentations. In some cases, Chen said that their NVRam emulation implementation didn’t work for all firmware images, and our test experiment shows that some of whom will run with NVRam information extracted from the real device, not emulated NVRam. In addition, COTS embedded devices, which directly extract from router devices, are usually controlled by vendors and firmware images could be different with those showcased on their websites.

In general, the considered methods have two drawbacks: Firstly, they focus on analyzing only standard Linux image or router firmware images that are downloaded from vendor websites and not dealing with those that are embedded the actual routers, so many firmwares still couldn’t operate. Secondly, the emulated firmware environment don’t enough a suitable environment for malware activate it’s behavior and don’t have tools to allow monitoring system behavior of malware. To overcome the shortcomings of previous works, a new tool is proposed in this paper to emulate actual router firmware images to perform the vulnerabilities and/or malware detection at the same time. This method is relied on a toolkit, named C500-Toolkit, with the capabilities of:

- The C500-toolkit is able to extract firmware image from actual router’s FLASH chip.

- The C500-toolkit is able to crawl firmware images from router vendor’s websites.

• The C500-toolkit automatically standardize router's firmware image as required to fully emulate it including the operating system and the web interface.

• The C500-toolkit can insert monitoring tools such as Strace, TcpDump to collect data during the emulation of router.

• The C500-toolkit is able to perform dynamic analysis to detect malware and vulnerabilities in router's firmware image.

II. ROUTER AND FIRMWARE STRUCTURE

This section presents the router, the firmware image structure and the challenges that we faced when building C500-toolkit.

The router is a network device in layer 3 of the Network Layer model OSI with main components are presented in Fig. 1:

<table>
<tr><td colspan="5">Firmware</td></tr>
<tr><td>CPU</td><td>RAM</td><td>NVRAM</td><td>FLASH</td><td>ROM</td></tr>
<tr><td colspan="5">Interfaces</td></tr>
</table>

Fig 1: Router structure

• CPU: is the heart of router that executes every instructions given by the operating system.

• ROM: is the Read-Only-Memory contains boot strap programming and basis software for starting up the router.

• RAM: is Random-Access-Memory and similar to the RAM used in traditional computer or PC, this volatile memory is lost all stored data when the router is shutdown or restart.

• FLASH: is a None-Volatile Memory where the firmware image is held, this component can be erasable/writable and keep data when the power is lost.

• NVRAM (None-Volatile RAM): has the same functionality as the FLASH but with less storage capacity. NVRAM contains configuration files of the device to ensure that when booting up, the default configurations of the router, including network and system configurations, will be automatically loaded correctly into the storing states. This component could be included in the FLASH.

• Interfaces: are physical hardware that ensure the transmission of packets come in and come out the router.

The composition of router is similar to PC except the architecture of CPU, about 86% of the routers use MIPS, ARM as in-house architecture because it was originally simpler and cheaper. Thanks to QEMU, a full-system emulation for any supported architecture including MIPS, ARM, X86, SPARC etc that allows creating an emulator for router. A router has the same components as a computer, except NVRam and the component is also not emulated in QEMU, it only updated by Firmadyne. It means origin QEMU couldn't emulate router like PC. But Firmadyne only use emulated NVRam information, it can't use information from actual NVRam of router devices.

The structure of firmware image may vary, depending on the function and the design of each manufacturer. Firmware image can be divided into the following types:

• Full-blown (full OS/kernel + boot-loader + libs + app): this is a complete firmware that contains a minimalist operating system. In this type, user-mode and kernel-mode are separated; applications run in user-mode allowing inserting custom tools for vulnerabilities and malware detection purpose.

• Integrated (apps + OS-as-a-lib): this is an incomplete firmware. Functions, operating system are built as a library and there is only kernel-mode in this type.

• Partial updates (apps or libs or resources or support): this type of firmware contains only files that used to update the firmware.

In this paper, firmware images of the first type (full-blown) are taken into account for the vulnerabilities and malware detection purpose. This choice is made because full-blown firmware images contain required components such as *bootloader*, *linux kernel*, *file-system image*, *user-land binaries*, *web-interface* and support files for the router emulation. These firmware images can be extracted from two sources, the first one is from vendor websites and the second one is to extract them from actual router devices. The first source is widely used by researchers such as Costin [14] or Chen [12]. The second source is rarely used in previous works even it has an important impact to determine if the embedded firmware image on actual routers contain malicious code or/and vulnerabilities.

To deal with the discussed challenges in the previous section, C500-Toolkit was built with the objectives of extracting, emulating and analyzing actual router firmware images, especially Linux-based firmwares. To showcase the capabilities of C500-toolkit, a real router (Tp-Link Wr842nd with the version of the FLASH chip is Winbond W25Q64FV) and malware (Linux/TheMoon and Linux/Mirai) were used for our experiments. To summarize, the contributions of this proposed toolkit are:

• The C500-toolkit has a module, named C500-Extractor, for extracting firmware image from an actual router device.

• The C500-toolkit has a module, named C500-Standardization, for completing the missing parts into a firmware image to get a pre-emulated image that can be performed by QEMU.

• The C500-toolkit has C500-Detector module, for detecting malware and vulnerabilities in a target firmware image.

III. C500-TOOLKIT

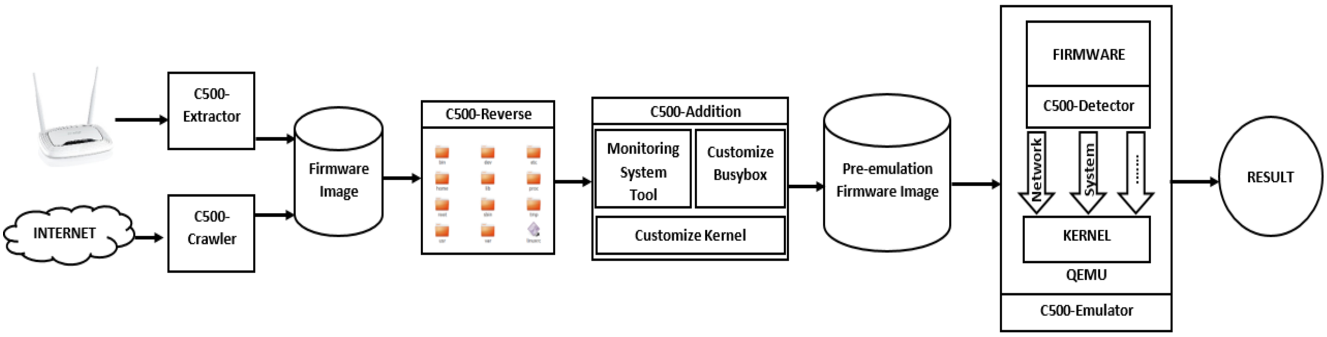


Fig 2: C500-Toolkit Composition

This section describs the C500-toolkit main components and their functions. The overview of this toolkit is shown in Fig. 2.

A. C500-Extractor

The extraction of firmware images on SOHO routers (Small Office - Home Office) can be performed through several methods such as using the deviceâ€™s backup function, using the serial console port or the Jtag debug channel. However, in some cases these extraction methods may not be as effective as desired because the manufacturers remove these ports on the printed circuit board, therefore do not allow users to interfere with the device. In this case, extracting the firmware image directly from the FLASH chip is necessary, which is also the goal of the C500-Extractor.

The C500-Extractor is a hardware module and it is designed to read/write data from/to the FLASH chip containing the firmware image. The first prototype version was built to deal with popular FLASH chips used on Tp-Link and Linksys routers that are:

- The Winbond FLASH chips W25Q32FV, W25Q64FV, W25Q128FV
- The Fidelix FLASH chip FM25Q64
- The Macronix FLASH chip MX25L1606E
- The Eon FLASH chip EN29LV320B-70TC

To read/write data from these FLASH chips, two main interfaces were used are the SPI (Serial Peripheral Interface) [20] and the FSMC (Flexible Static Memory Controller) [21]. The FSMC port is employed for the Eon FLASH chip extraction, while the SPI port is applied to the Winbond, Fidelix, Macronix FLASH chips. This hardware module is controlled by a micro-controller STM32F10 family and support both communication channel SPI and FSMC. This module is shown in Fig. 3. To start the firmware image extraction process, the target FLASH chip is put into a corresponding compartment. The extractor would read the ID code of that FLASH chip to detect which one is put into, then it adjusts the appropriate configurations to extract data to get a binary file. The extracted binary file is then transferred to a computer through the micro USB connection. This binary file usually contains NVRAM, Art, Kernel vmlinux, Bootloader, Rootfs in which three parts kernel vmlinux, Rootfs and NVRAM have an important role. The first part indicates the Linux kernel version used by the operating system that is found in the Rootfs. And the NVRAM indicates the required configurations for executing the router correctly, especially in user-mode. Once a firmware image is extracted, C500-toolkit standardizes this one by completing missing data, in using the extracted parts, to emulate it by QEMU.

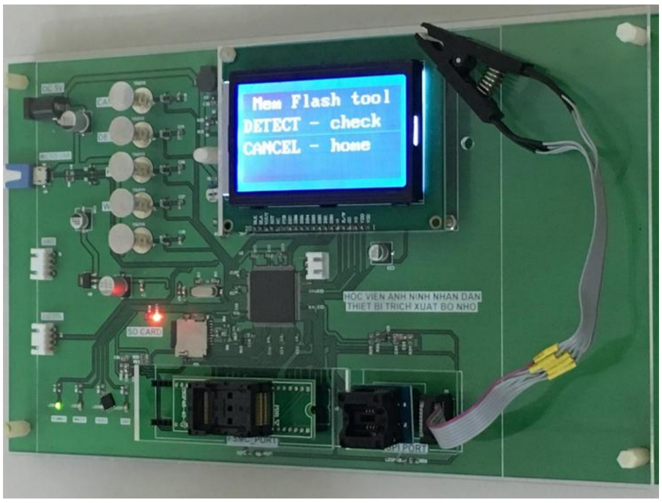


Fig 3: C500-Extractor Component

B. C500-Standardization

The standardization of the firmware image is performed by two steps:

- Extraction and identification of hardware architecture to replace Busybox, named C500-Reverse.
- Addition of missing data including NVRAM, Linux kernel and Busybox, to get a pre-emulated image for QEMU.

First, C500-Reverse is called to identify the CPU architecture, its endianness, the kernel version and then, unzip the extracted firmware image to get file-system directories. There were several tools that are similar to C500-Reverse such as Firmware-mod-kit (FMK) [22], Binwalk [23] or the Extractor module of Firmadyne [12]. However, these modules have non-negligeable drawbacks that were presented in [24]. As a result, C500-Reverse was built in using the most popular extraction tools for file-system formats such as Jefferson, Sasquatch, CRAMfs, Yaffs. These formats represent 97% of 13,275 tested firmware images that were crawled from router vendor websites.

The most popular file-system formats, based on

13,275 tested firmware images, are shown in Fig. 4.

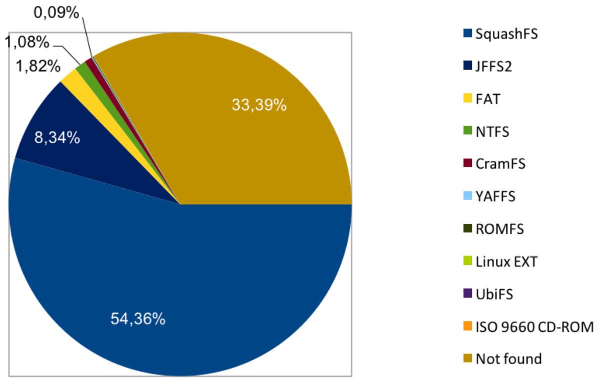


Fig 4: C500-Extractor Component

C500-Reverse module was built in modular manner allowing to be updated frequently. The details of this component is presented in [24] and the file-system directories of Tp-Link Wr842nd are shown in Fig. 5.

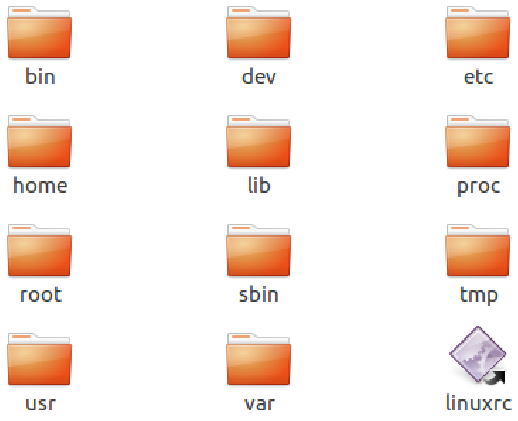


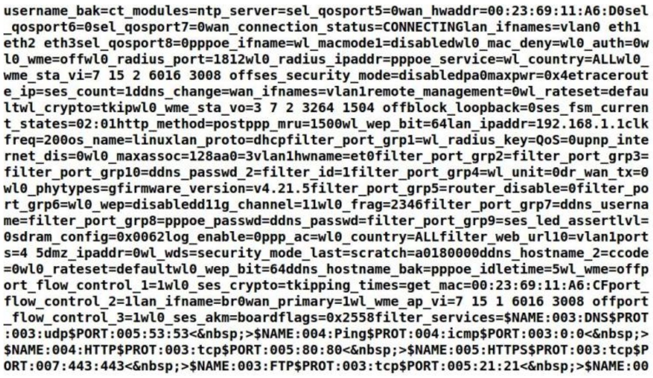
Fig 5: Root file-system directories

Along with the familiar directories found in linux root file system such as *etc*, *bin*, *lib*, we can find the router web-interface in *www*, *cli* directory. At this stage, the module C500-Addition starts to build an empty image format *raw*, then copy required data progressively into it to make a pre-emulated firmware. This process is similar to Firmadyne in order to emulate the web-interface [12] except two points:

- The initialization of the NVRAM
- The addition of tools for monitoring the network and the operating system behavior

For the first point, Chen *et al.* [12] have used a shared library, named $libNvram.so$ to persist device-specific configuration parameters, for tested firmware images. Because of using a generic library for all of router firmware images, the firmware image emulation successful rate is quite low (4,992 firmware images out of 9,486). To overcome this shortcoming, C500-Addition module uses the extracted NVRAM as an input for customizing $libNvram.so$ to fit the required configurations of corresponding router device. After being extracted from an actual device, NVRAM was stored in a database to be reused afterwards. One part of the extracted NVRAM of Tp-Link Wr842nd is shown in Fig. 6 in which we can retrieve information such as:

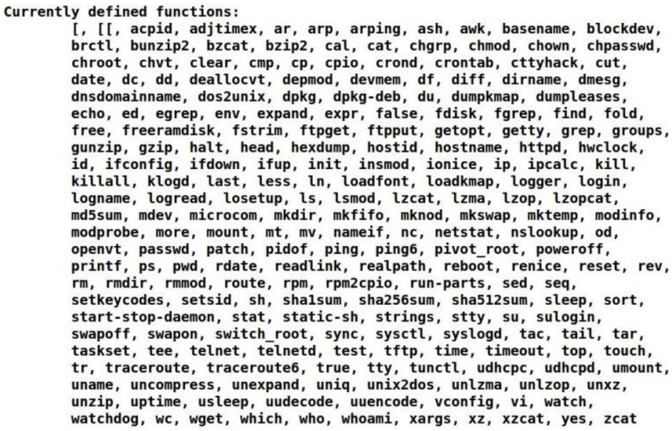
- The device Mac address is: 00:23;69:11:A6:D0
- The LAN interfaces are: vlan0, eth1, eth2, eth3
- The defaut Ip address is: 192.168.1.1
- The firmware version is: v4.21.5



```
username_bak=ct_modules=ntp_server=sel_qosport5=0wan_hwaddr=00:23:69:11:A6:D0sel
_qosport6=0sel_qosport7=0wan_connection_status=CONNECTINGlan_ifnames=vlan0 eth1
eth2 eth3sel_qosport8=0pppoe_ifname=wl_macmode1=disabledwl0_mac_deny=wl0_auth=0w
l0_wme=offwl0_radius_port=1812wl0_radius_ipaddr=pppoe_service=wl_country=ALLwl0_
wme_sta_vi=7 15 2 6016 3008 offses_security_mode=disabledpa0maxpwr=0x4etracerout
e_ip=ses_count=1ddns_change=wan_ifnames=vlan1remote_management=0wl_rateset=defau
ltwl_crypto=tkipwl0_wme_sta_vo=3 7 2 3264 1504 offblock_loopback=0ses_fsm_curren
t_states=02:01http_method=postppp_mru=1500wl_wep_bit=64lan_ipaddr=192.168.1.1clk
freq=200os_name=linuxlan_proto=dhcpfilter_port_grp1=wl_radius_key=QoS=0upnp_inte
rnet_dis=0wl0_maxassoc=128aa0=3vlan1hwname=et0filter_port_grp2=filter_port_grp3=
filter_port_grp10=ddns_passwd_2=filter_id=1filter_port_grp4=wl_unit=0dr_wan_tx=0
wl0_phytypes=gfirmware_version=v4.21.5filter_port_grp5=router_disable=0filter_po
rt_grp6=wl0_wep=disabledd11g_channel=11wl0_frag=2346filter_port_grp7=ddns_userna
me=filter_port_grp8=pppoe_passwd=ddns_passwd=filter_port_grp9=ses_led_assertlvl=
0sdram_config=0x0062log_enable=0ppp_ac=wl0_country=ALLfilter_web_url10=vlan1port
s=4 5dmz_ipaddr=0wl_wds=security_mode_last=scratch=a0180000ddns_hostname_2=ccode
=0wl0_rateset=defaultwl0_wep_bit=64ddns_hostname_bak=pppoe_idletime=5wl_wme=offp
ort_flow_control_1=1wl0_ses_crypto=tkipping_times=get_mac=00:23:69:11:A6:CFport_
flow_control_2=1lan_ifname=br0wan_primary=1wl_wme_ap_vi=7 15 1 6016 3008 offport
_flow_control_3=1wl0_ses_akm=boardflags=0x2558filter_services=$NAME:003:DNS$PROT
:003:udp$PORT:005:53:53<&nbsp;>$NAME:004:Ping$PROT:004:icmp$PORT:003:0:0<&nbsp;>
$NAME:004:HTTP$PROT:003:tcp$PORT:005:80:80<&nbsp;>$NAME:005:HTTPS$PROT:003:tcp$P
ORT:007:443:443<&nbsp;>$NAME:003:FTP$PROT:003:tcp$PORT:005:21:21<&nbsp;>$NAME:00
```

Fig 6: Extracted Tp-Link Wr842nd NVRAM

The second point is not trivial to resolve. First, manufacturers always search to optimize the storage, therefore they have customized their firmware image by removing unnecessary packages. Hence, Busybox, a popular tool in embedded systems, combines tiny versions of many common UNIX utilities into a single small-size program [25], thought it is necessary to determine if the employed Busybox version has enough utilities for desired monitoring tools. In this paper, a customized *Strace* version and a customized *Tcpdump* are used for the vulnerabilities and/or malware detection purpose. *Strace* is a diagnostic, debugging and instructional user-space utility for a specific program in Linux, while *Tcpdump* is used for analyzing network traffic data. These tools are popularly used in malware analysis, based on abnomal behavior analysis. Thanks to Landley [25] with different pre-built Busybox versions, we can customize them to get more utilities. For each Linux kernel version, a suitable Busybox version is taken to replace the one found in *bin* directory, a full utilities system is then achieved as showing in Fig. 7. Second, about 93% of COTS router firmware images operate with Linux kernel versions 2.6.x. The difference between Linux kernel versions 2.6.x is not significant, therefore make a generic pre-built Linux kernel version 2.6.34 is the adopted solution for C500-Addition module.



```
Currently defined functions:
        [, [[, acpid, adjtimex, ar, arp, arping, ash, awk, basename, blockdev,
        brctl, bunzip2, bzcat, bzip2, cal, cat, chgrp, chmod, chown, chpasswd,
        chroot, chvt, clear, cmp, cp, cpio, crond, crontab, cttyhack, cut,
        date, dc, dd, deallocvt, depmod, devmem, df, diff, dirname, dmesg,
        dnsdomainname, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dumpleases,
        echo, ed, egrep, env, expand, expr, false, fdisk, fgrep, find, fold,
        free, freeramdisk, fstrim, ftpget, ftpput, getopt, getty, grep, groups,
        gunzip, gzip, halt, head, hexdump, hostid, hostname, httpd, hwclock,
        id, ifconfig, ifdown, ifup, init, insmod, ionice, ip, ipcalc, kill,
        killall, klogd, last, less, ln, loadfont, loadkmap, logger, login,
        logname, logread, losetup, ls, lsmod, lzcat, lzma, lzop, lzopcat,
        md5sum, mdev, microcom, mkdir, mkfifo, mknod, mkswap, mktemp, modinfo,
        modprobe, more, mount, mt, mv, nameif, nc, netstat, nslookup, od,
        openvt, passwd, patch, pidof, ping, ping6, pivot_root, poweroff,
        printf, ps, pwd, rdate, readlink, realpath, reboot, renice, reset, rev,
        rm, rmdir, rmmod, route, rpm, rpm2cpio, run-parts, sed, seq,
        setkeycodes, setsid, sh, sha1sum, sha256sum, sha512sum, sleep, sort,
        start-stop-daemon, stat, static-sh, strings, stty, su, sulogin,
        swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar,
        taskset, tee, telnet, telnetd, test, tftp, time, timeout, top, touch,
        tr, traceroute, traceroute6, true, tty, tunctl, udhcpc, udhcpd, umount,
        uname, uncompress, unexpand, uniq, unix2dos, unlzma, unlzop, unxz,
        unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, watch,
        watchdog, wc, wget, which, who, whoami, xargs, xz, xzcat, yes, zcat
```

Fig 7: Full utilities busybox

After customizing NVRAM, Busybox and Linux Kernel with the C500-Addition module, the *raw* image became a pre-emulated firmware image and could be performed by QEMU. Finally, the C500-Detector is used for detecting vulnerabilities and/or malware during the run-time of the firmware.

C. C500-Detector

This module is built for examining the malware's behavior from the operating system and the network during its run-time. In this paper, C500-Detector analyzes logs provided by *Strace* and *Tcpdump* to identify abnormal behaviors of creating/deleting files without user permission; listening, opening and scanning unauthorized ports; connecting to static black-list IP; creating infinite loops. The experimentation results is presented in the next section.

IV. EXPERIMENTAL RESULTS

In this section, we show how 500-Toolkit deals with two well-known malware Linux/TheMoon and Linux/Mirai. The experiement scenarios are as follows:

1. **Linux/TheMoon** is downloaded from [26], this very first uncovered router malware was observed by a researcher at the SANS Internet Storm Center spreading itself to a large number of Linksys router models.

(MD5: 88a5c5f9c5de5ba612ec96682d61c7bb)

2. **Linux/Mirai** is downloaded from [27], this malware turned million of web-camera, printers and baby monitor hijacked by a botnet made worldwide news after it had taken down some very high-profile websites and this incident was described as the largest denial-of-service attack to date [28].

(MD5: c86082bc4a75c2bbb62d8aef52a57168)

3. Two couple copies of respectively firmware images of Linksys E2500 and Tp-Link Wr842nd are taken into account for Linux/TheMoon and Linux/Mirai. These firmware images were extracted from actual devices, the extracted NVRAM is used by C500-Addition module; then we only infect the downloaded mawlares into the first one (by copying it into the *raw* image) and keep clean for the second.

4. We perform the three presented step to get two pre-emulated *raw* images.

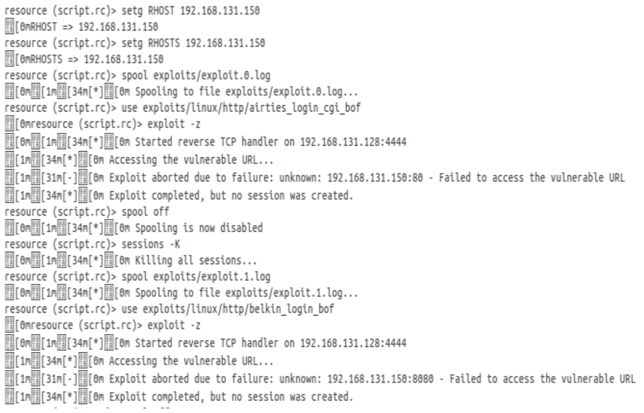
5. We emulate these two couple images and two analysis are then performed:

- Using *Metasploit* to scan the emulated images to check if there are more vulnerabilities during Linux/TheMoon and Linux/Mirai execution.

- Using *Strace* and *Tcpdump* to scan two couple images and log every system-calls and network behaviors.

6. Based on obtained results from *Strace* and *Tcpdump*, *C500-Detector* module perform the analysis to warn abnormal behaviors.

These two malware samples are slightly different in terms of behavior. Linux/Mirai has to connect to a C&C server and infected firmware images are set into infinite loop waiting for commands from the C&C Server but it's not necessary for Linux/TheMoon. Linux/TheMoon interacts a lot with operating system by creating and deleting files before executing shell commands on a vulnerable router that has Remote Management Access enabled, and downloads a copy of itself. Therefore, we present in details the results of Linux/TheMoon with *Strace* and Linux/Mirai with *Strace* and *Tcpdump*.First, we perform the vulnerabilities detection for *raw* images containing Linux/TheMoon and Linux/ Mirai. They were emulated by QEMU to start up the web-interface, which could be scanned by *Metasploit*. At this stage, no CVE was found even though two *raw* had been operating with active Linux/TheMoon and Linux/Mirai malware, this result is shown in Fig. 8. This means that Linux/TheMoon and Linux/Mirai did not create any vulnerability during its execution, therefore only Firmadyne is not enough to detect abnormal behaviors.



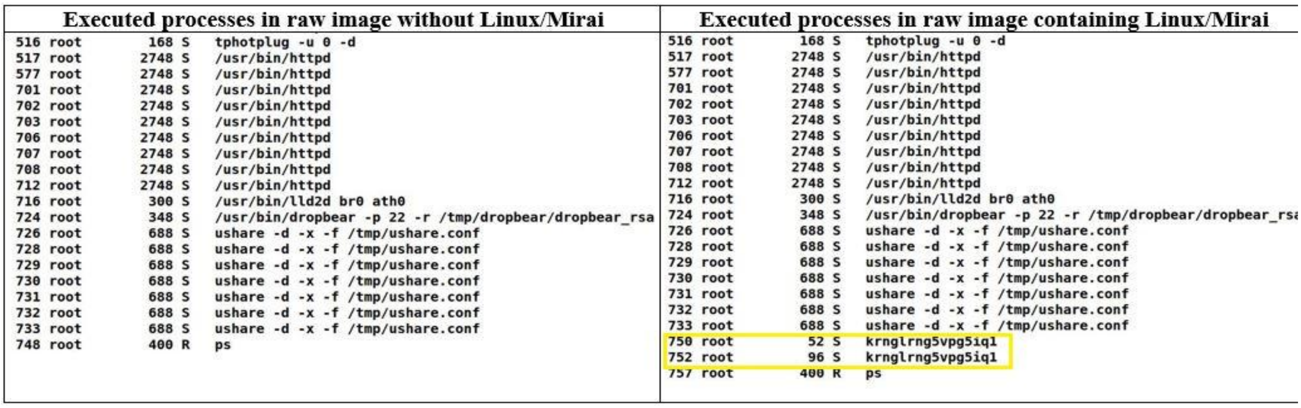
```
resource (script.rc)> setg RHOST 192.168.131.150
␛[0mRHOST => 192.168.131.150
resource (script.rc)> setg RHOSTS 192.168.131.150
␛[0mRHOSTS => 192.168.131.150
resource (script.rc)> spool exploits/exploit.0.log
␛[0m␛[1m␛[34m[*]␛[0m Spooling to file exploits/exploit.0.log...
resource (script.rc)> use exploits/linux/http/airties_login_cgi_bof
␛[0mresource (script.rc)> exploit -z
␛[0m␛[1m␛[34m[*]␛[0m Started reverse TCP handler on 192.168.131.128:4444
␛[1m␛[34m[*]␛[0m Accessing the vulnerable URL...
␛[1m␛[31m[-]␛[0m Exploit aborted due to failure: unknown: 192.168.131.150:80 - Failed to access the vulnerable URL
␛[1m␛[34m[*]␛[0m Exploit completed, but no session was created.
resource (script.rc)> spool off
␛[0m␛[1m␛[34m[*]␛[0m Spooling is now disabled
resource (script.rc)> sessions -K
␛[0m␛[1m␛[34m[*]␛[0m Killing all sessions...
resource (script.rc)> spool exploits/exploit.1.log
␛[0m␛[1m␛[34m[*]␛[0m Spooling to file exploits/exploit.1.log...
resource (script.rc)> use exploits/linux/http/belkin_login_bof
␛[0mresource (script.rc)> exploit -z
␛[0m␛[1m␛[34m[*]␛[0m Started reverse TCP handler on 192.168.131.128:4444
␛[1m␛[34m[*]␛[0m Accessing the vulnerable URL...
␛[1m␛[31m[-]␛[0m Exploit aborted due to failure: unknown: 192.168.131.150:8080 - Failed to access the vulnerable URL
␛[1m␛[34m[*]␛[0m Exploit completed, but no session was created.
```

Fig 8: Obtained result with Metasploit

Results with *Strace*

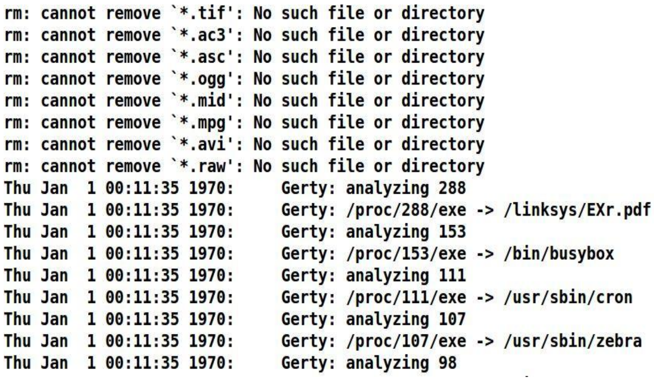
Linux/TheMoon

Linux/TheMoon (EXr.pdf) was launched within the Linksys E2500 firmware image and some facts appeared as Fig. 9.



Executed processes in raw image without Linux/Mirai	Executed processes in raw image containing Linux/Mirai
`516 root 168 S tphotplug -u 0 -d`	`516 root 168 S tphotplug -u 0 -d`
`517 root 2748 S /usr/bin/httpd`	`517 root 2748 S /usr/bin/httpd`
`577 root 2748 S /usr/bin/httpd`	`577 root 2748 S /usr/bin/httpd`
`701 root 2748 S /usr/bin/httpd`	`701 root 2748 S /usr/bin/httpd`
`702 root 2748 S /usr/bin/httpd`	`702 root 2748 S /usr/bin/httpd`
`703 root 2748 S /usr/bin/httpd`	`703 root 2748 S /usr/bin/httpd`
`706 root 2748 S /usr/bin/httpd`	`706 root 2748 S /usr/bin/httpd`
`707 root 2748 S /usr/bin/httpd`	`707 root 2748 S /usr/bin/httpd`
`708 root 2748 S /usr/bin/httpd`	`708 root 2748 S /usr/bin/httpd`
`712 root 2748 S /usr/bin/httpd`	`712 root 2748 S /usr/bin/httpd`
`716 root 300 S /usr/bin/lld2d br0 ath0`	`716 root 300 S /usr/bin/lld2d br0 ath0`
`724 root 348 S /usr/bin/dropbear -p 22 -r /tmp/dropbear/dropbear_rsa`	`724 root 348 S /usr/bin/dropbear -p 22 -r /tmp/dropbear/dropbear_rsa`
`726 root 688 S ushare -d -x -f /tmp/ushare.conf`	`726 root 688 S ushare -d -x -f /tmp/ushare.conf`
`728 root 688 S ushare -d -x -f /tmp/ushare.conf`	`728 root 688 S ushare -d -x -f /tmp/ushare.conf`
`729 root 688 S ushare -d -x -f /tmp/ushare.conf`	`729 root 688 S ushare -d -x -f /tmp/ushare.conf`
`730 root 688 S ushare -d -x -f /tmp/ushare.conf`	`730 root 688 S ushare -d -x -f /tmp/ushare.conf`
`731 root 688 S ushare -d -x -f /tmp/ushare.conf`	`731 root 688 S ushare -d -x -f /tmp/ushare.conf`
`732 root 688 S ushare -d -x -f /tmp/ushare.conf`	`732 root 688 S ushare -d -x -f /tmp/ushare.conf`
`733 root 688 S ushare -d -x -f /tmp/ushare.conf`	`733 root 688 S ushare -d -x -f /tmp/ushare.conf`
`748 root 400 R ps`	`750 root 52 S krnglrng5vpg5iq1`
	`752 root 96 S krnglrng5vpg5iq1`
	`757 root 400 R ps`

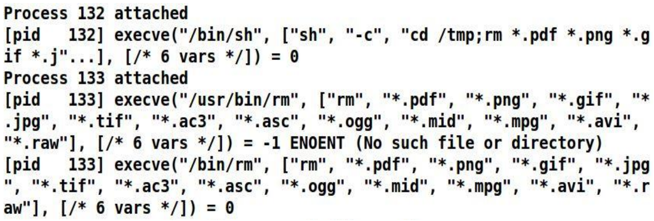
Fig 13: Executed processes in both raw images



```
rm: cannot remove `*.tif': No such file or directory
rm: cannot remove `*.ac3': No such file or directory
rm: cannot remove `*.asc': No such file or directory
rm: cannot remove `*.ogg': No such file or directory
rm: cannot remove `*.mid': No such file or directory
rm: cannot remove `*.mpg': No such file or directory
rm: cannot remove `*.avi': No such file or directory
rm: cannot remove `*.raw': No such file or directory
Thu Jan  1 00:11:35 1970:     Gerty: analyzing 288
Thu Jan  1 00:11:35 1970:     Gerty: /proc/288/exe -> /linksys/EXr.pdf
Thu Jan  1 00:11:35 1970:     Gerty: analyzing 153
Thu Jan  1 00:11:35 1970:     Gerty: /proc/153/exe -> /bin/busybox
Thu Jan  1 00:11:35 1970:     Gerty: analyzing 111
Thu Jan  1 00:11:35 1970:     Gerty: /proc/111/exe -> /usr/sbin/cron
Thu Jan  1 00:11:35 1970:     Gerty: analyzing 107
Thu Jan  1 00:11:35 1970:     Gerty: /proc/107/exe -> /usr/sbin/zebra
Thu Jan  1 00:11:35 1970:     Gerty: analyzing 98
```

Fig 9: Captured results during Linux/TheMoon execution

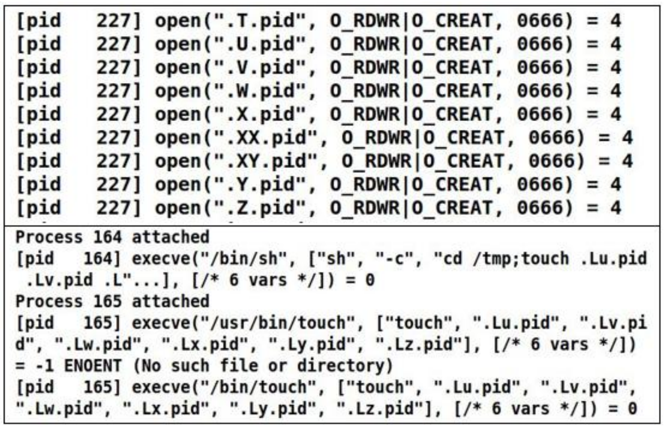
At the first sight, we can conclude that this firmware image does not behave in normal way because there are deleting file behaviors. To be more precise, *Strace* is used for getting more analysis in detail on system-calls made by Linux/TheMoon. After launching *Strace*, C500-Detector detects deleting files with extensions *.pdf*, *.png*, *.gif*, *.jpg*, *.tif* etc. in */tmp* directory as presented by Fig. 10. Actually, these extensions are used by Linux/TheMoon for its several variants, therefore this step tried to remove all traces it may have left in the system.



```
Process 132 attached
[pid   132] execve("/bin/sh", ["sh", "-c", "cd /tmp;rm *.pdf *.png *.g
if *.j"...], [/* 6 vars */]) = 0
Process 133 attached
[pid   133] execve("/usr/bin/rm", ["rm", "*.pdf", "*.png", "*.gif", "*
.jpg", "*.tif", "*.ac3", "*.asc", "*.ogg", "*.mid", "*.mpg", "*.avi",
"*.raw"], [/* 6 vars */]) = -1 ENOENT (No such file or directory)
[pid   133] execve("/bin/rm", ["rm", "*.pdf", "*.png", "*.gif", "*.jpg
", "*.tif", "*.ac3", "*.asc", "*.ogg", "*.mid", "*.mpg", "*.avi", "*.r
aw"], [/* 6 vars */]) = 0
```

Fig 10: Deleting file behaviors

Then, Linux/TheMoon created an important amount of new and empty files, about 110 files, with two commands: *touch* and *open* behaviors as showning by Fig. 11.



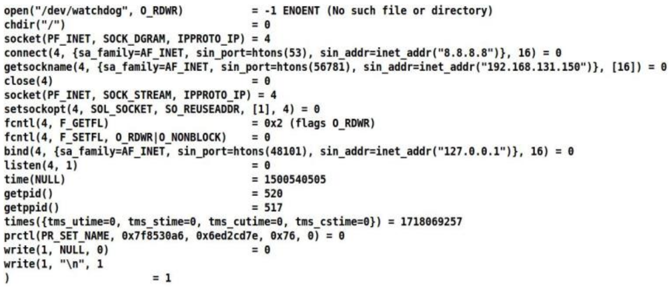
```
[pid   227] open(".T.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".U.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".V.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".W.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".X.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".XX.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".XY.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".Y.pid", O_RDWR|O_CREAT, 0666) = 4
[pid   227] open(".Z.pid", O_RDWR|O_CREAT, 0666) = 4
```

```
Process 164 attached
[pid   164] execve("/bin/sh", ["sh", "-c", "cd /tmp;touch .Lu.pid
 .Lv.pid .L"...], [/* 6 vars */]) = 0
Process 165 attached
[pid   165] execve("/usr/bin/touch", ["touch", ".Lu.pid", ".Lv.pi
d", ".Lw.pid", ".Lx.pid", ".Ly.pid", ".Lz.pid"], [/* 6 vars */])
= -1 ENOENT (No such file or directory)
[pid   165] execve("/bin/touch", ["touch", ".Lu.pid", ".Lv.pid",
".Lw.pid", ".Lx.pid", ".Ly.pid", ".Lz.pid"], [/* 6 vars */]) = 0
```

Fig 11: Creating file behaviors

As a result, we can conclude that this firmware image has abnormal behaviors. As for the firmware image containing Linux/Mirai malwalre, *Strace* was launched for the first image and we can retrieve a lot of interesting information. One part of this result is shown in Fig. 12.

Linux/Mirai



```
open("/dev/watchdog", O_RDWR)           = -1 ENOENT (No such file or directory)
chdir("/")                              = 0
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("8.8.8.8")}, 16) = 0
getsockname(4, {sa_family=AF_INET, sin_port=htons(56781), sin_addr=inet_addr("192.168.131.150")}, [16]) = 0
close(4)                                = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 4
setsockopt(4, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
fcntl(4, F_GETFL)                       = 0x2 (flags O_RDWR)
fcntl(4, F_SETFL, O_RDWR|O_NONBLOCK)    = 0
bind(4, {sa_family=AF_INET, sin_port=htons(48101), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
listen(4, 1)                            = 0
time(NULL)                              = 1500540505
getpid()                                = 520
getppid()                               = 517
times({tms_utime=0, tms_stime=0, tms_cutime=0, tms_cstime=0}) = 1718069257
prctl(PR_SET_NAME, 0x7f8530a6, 0x6ed2cd7e, 0x76, 0) = 0
write(1, NULL, 0)                       = 0
write(1, "\n", 1
)                       = 1
```

Fig 12: Strace log for the first *raw* image

First, to control watchdog timer in embedded device, Linux/Mirai opens the */etc/watchdog* file in read-write state at the first line:

open("/dev/watchdog", O_RDWR)

Then, PF_INET socket for TCP protocol is opened through a specific port (53) to self connect to Google DNS Server (8.8.8.8) to ensure that the Internet connection was established.

connect(4, {sa_family=AF_INET, sin_port=hto ns(53), sin_addr=inet_addr("8.8.8.8")}, 16)

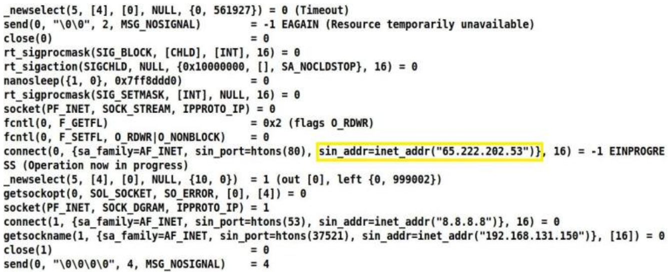
At this stage, everything seems fine but moving forward, we can see that there is an abnormal behavior consisting in opening a random TCP/port (48178), based on previous PF_INET socket, to the specific IP address (192.168.131.150) without user permission.

getsockname(4, {sa_family=AF_INET, sin_port=htons(56781), sin_addr=inet_addr("192.168.131.150")}, 16)

Then, Linux/Mirai opens another PF_INET socket at port 48101 from localhost IP address 127.0.0.1 and then starting to listen to the incoming connections and start its infinite loop.

bind(4, {sa_family=AF_INET, sin_port=htons(48101), sin_addr=inet_addr("127.0.0.1")}, 16) listen(4, 1)

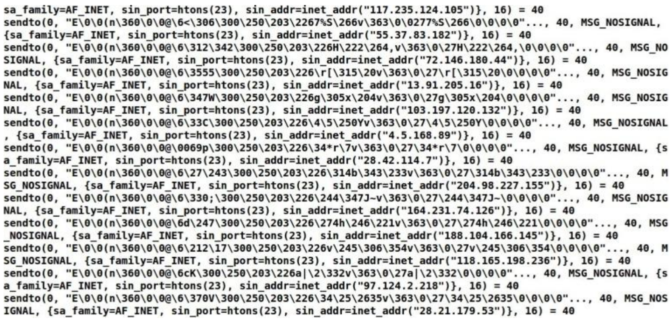
Although this log is not exhaustive to identify backdoor opening and changes in infected system files but it collects significant behaviors: modifying watchdog file, opening a TCP/port to a specific IP, listening from outside to connect to previous port and continuously looping. Accordingly, we can conclude that these actions is abnormal behaviors. To be more precised, we analyzed in detail the logs captured by *Strace* for both of *raw* images. Concerning system behaviors, there are two supplement processes created, while Linux/Mirai was launched, are PID 747 and 749 as showing is Fig. 13. The analysis of the process PID 747 with *Strace* shows that this process behaves as a Backdoor to connect via Http on IP 65.222.202.53 via the port 80. Details of this connection are shown in Fig. 14.



```
_newselect(5, [4], [0], NULL, {0, 561927}) = 0 (Timeout)
send(0, "\0\0", 2, MSG_NOSIGNAL)        = -1 EAGAIN (Resource temporarily unavailable)
close(0)                                = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [INT], 16) = 0
rt_sigaction(SIGCHLD, NULL, {0x10000000, [], SA_NOCLDSTOP}, 16) = 0
nanosleep({1, 0}, 0x7ff8ddd0)           = 0
rt_sigprocmask(SIG_SETMASK, [INT], NULL, 16) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 0
fcntl(0, F_GETFL)                       = 0x2 (flags O_RDWR)
fcntl(0, F_SETFL, O_RDWR|O_NONBLOCK)    = 0
connect(0, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("65.222.202.53")}, 16) = -1 EINPROGRE
SS (Operation now in progress)
_newselect(5, [4], [0], NULL, {10, 0})  = 1 (out [0], left {0, 999002})
getsockopt(0, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 1
connect(1, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("8.8.8.8")}, 16) = 0
getsockname(1, {sa_family=AF_INET, sin_port=htons(37521), sin_addr=inet_addr("192.168.131.150")}, [16]) = 0
close(1)                                = 0
send(0, "\0\0\0\0", 4, MSG_NOSIGNAL)    = 4
```

Fig 14: The connection of Linux/Mirai to 65.222.202.53 via the port 80

The process PID 749 scans the Telnet port (port 23) of random IP addresses, for example *117.235.124.105, 55.37.83.182...*, this process is aimed at infecting the Linux/Mirai malware to other devices on Internet. The scanning step is shown in Fig. 15

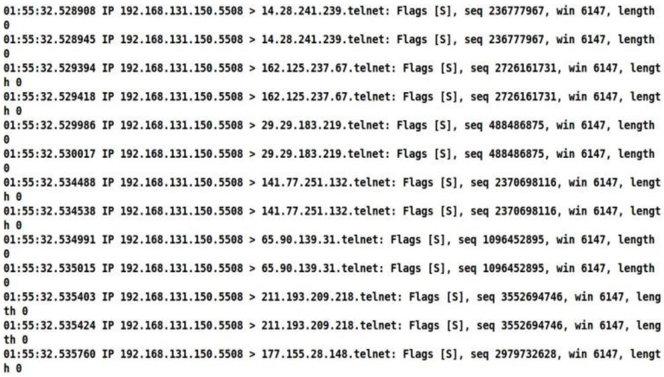


```
sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("117.235.124.105")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6<\306\300\250\203\2267%S\266v\363\0\0277%S\266\0\0\0\0"..., 40, MSG_NOSIGNAL,
{sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("55.37.83.182")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\312\342\300\250\203\226H\222\264,v\363\0\27H\222\264,\0\0\0\0"..., 40, MSG_NO
SIGNAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("72.146.180.44")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\3555\300\250\203\226\r[\315\20v\363\0\27\r[\315\20\0\0\0\0"..., 40, MSG_NOSIG
NAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("13.91.205.16")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\347W\300\250\203\226g\305x\204v\363\0\27g\305x\204\0\0\0\0"..., 40, MSG_NOSIG
NAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("103.197.120.132")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\33C\300\250\203\226\4\5\250Yv\363\0\27\4\5\250Y\0\0\0\0"..., 40, MSG_NOSIGNAL
, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("4.5.168.89")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\0069p\300\250\203\226\34*r\7v\363\0\27\34*r\7\0\0\0\0"..., 40, MSG_NOSIGNAL, {s
a_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("28.42.114.7")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\27\243\300\250\203\226\314b\343\233v\363\0\27\314b\343\233\0\0\0\0"..., 40, M
SG_NOSIGNAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("204.98.227.155")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\330;\300\250\203\226\244\347J~v\363\0\27\244\347J~\0\0\0\0"..., 40, MSG_NOSIG
NAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("164.231.74.126")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6d\247\300\250\203\226\274h\246\221v\363\0\27\274h\246\221\0\0\0\0"..., 40, MSG
_NOSIGNAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("188.104.166.145")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\212\17\300\250\203\226v\245\306\354v\363\0\27v\245\306\354\0\0\0\0"..., 40, M
SG_NOSIGNAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("118.165.198.236")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6cK\300\250\203\226a|\2\332v\363\0\27a|\2\332\0\0\0\0"..., 40, MSG_NOSIGNAL, {s
a_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("97.124.2.218")}, 16) = 40
sendto(0, "E\0\0(n\360\0\0@\6\370V\300\250\203\226\34\25\2635v\363\0\27\34\25\2635\0\0\0\0"..., 40, MSG_NOS
IGNAL, {sa_family=AF_INET, sin_port=htons(23), sin_addr=inet_addr("28.21.179.53")}, 16) = 40
```

Fig 15: Log of the Telnet scanning ports

To capture the network behaviors of Linux/Mirai, *Tcpdump* was launched and the obtained result is shown in Fig. 16

***Results with** Tcpdump*



```
01:55:32.528908 IP 192.168.131.150.5508 > 14.28.241.239.telnet: Flags [S], seq 236777967, win 6147, length
0
01:55:32.528945 IP 192.168.131.150.5508 > 14.28.241.239.telnet: Flags [S], seq 236777967, win 6147, length
0
01:55:32.529394 IP 192.168.131.150.5508 > 162.125.237.67.telnet: Flags [S], seq 2726161731, win 6147, lengt
h 0
01:55:32.529418 IP 192.168.131.150.5508 > 162.125.237.67.telnet: Flags [S], seq 2726161731, win 6147, lengt
h 0
01:55:32.529986 IP 192.168.131.150.5508 > 29.29.183.219.telnet: Flags [S], seq 488486875, win 6147, length
0
01:55:32.530017 IP 192.168.131.150.5508 > 29.29.183.219.telnet: Flags [S], seq 488486875, win 6147, length
0
01:55:32.534488 IP 192.168.131.150.5508 > 141.77.251.132.telnet: Flags [S], seq 2370698116, win 6147, lengt
h 0
01:55:32.534538 IP 192.168.131.150.5508 > 141.77.251.132.telnet: Flags [S], seq 2370698116, win 6147, lengt
h 0
01:55:32.534991 IP 192.168.131.150.5508 > 65.90.139.31.telnet: Flags [S], seq 1096452895, win 6147, length
0
01:55:32.535015 IP 192.168.131.150.5508 > 65.90.139.31.telnet: Flags [S], seq 1096452895, win 6147, length
0
01:55:32.535403 IP 192.168.131.150.5508 > 211.193.209.218.telnet: Flags [S], seq 3552694746, win 6147, leng
th 0
01:55:32.535424 IP 192.168.131.150.5508 > 211.193.209.218.telnet: Flags [S], seq 3552694746, win 6147, leng
th 0
01:55:32.535760 IP 192.168.131.150.5508 > 177.155.28.148.telnet: Flags [S], seq 2979732628, win 6147, lengt
h 0
```

Fig 16: Obtained result with *Tcpdump*

Tcpdump shows that Linux/Mirai try to reach a wide range random IP by sending ICMP packets. Because of this firmware image was running within our emulator and there is no Internet connection, therefore we cannot obserse futher the netword behaviors that Linux/Mirai perform. At this point, we can conclude that all Linux/Mirai behaviors could not be traced in details but two interesting comportments, without user permission, are captured and analyzed:

- The behavior of connecttion to an outsite IP and starting to listen to the incoming connections.

- The behavior of scanning a wide range of random IP to check if these devices are alive.

Based on these two behaviors, we can conclude that the examined firmware image could be infected by malware.

V. CONCLUSION

In this paper, we presented the C500-toolkit to deal with the detection of malware and vulnerabilities in SOHO routers. Three main contributions of this toolkit respectively are: extracting firmware from an actual device FLASH chip; auto-completing missing parts to get a pre-emulated firmware image by QEMU; capturing and analyzing abnormal behaviors during its run-time.

By extracting and using NVRAM for configuring firmware images, the C500-toolkit could get a higher successful emulation rate. Our proposed tool could use Firmadyne as a complement to perform a full device scanning in order to identify vulnerabilities. Although there remain some limitations in malware detection rules or FLASH chip types that can be extracted, but it showed that we could detect malware by using C500-toolkit based on abnormal behaviors captured. Dynamic analysis could be interested in many aspects such as the obfuscation issue avoiding and the scalability, but it could not explore all malware trigger conditions (time, system events or network inputs) as static analysis approach. Therefore, our next challenge is to use static analysis as a pre-treatment step to automatically identify malware trigger conditions to use as inputs to our simulator.

REFERENCES

1. Ashton Kevin, That Internet of things thing. RFiD Journal 22 (7) (2009) 97-114.
2. The internet of things: How the next evolution of the internet is changing everything. URL http://www.cisco.com/
3. Yan Shoshitaishvili, Wang Ruoyu, Hauser Christophe, Kruegel Christopher and Vigna Giovanni, Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware, NDSS, 2015.
4. Colin Tankard, The security issues of the internet of things, Computer Fraud&Security (9), pages: 11-14, 2015.
5. Elisa Bertino and Nayeem Islam. Botnets and Internet of Things Security. In IEEE Computer Society, 2017, p.76-79. IEEE, n.d. https://doi.org/10.1109/MC.2017.62.
6. P. Beckett. Gdpr compliance: your tech department's next big opportunity. Computer Fraud & Security 2017 (5) (2017) 9-13.
7. Internet of things top 10 project. URL www.owasp.org/
8. Pavel Celeda, Radek Krejci, and Vojtech Krmicek. Revealing and Analysing Modem Malware. In IEEE International Conference on Communications (ICC). Ottawa, ON, Canada, 2012, pp. 971-975. https://doi.org/10.1109/ICC.2012.6364598.
9. Drew Davidson, Benjamin Moench, Thomas Ristenpart, Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. USENIX Security Symposium (2013) 463-478.
10. Andrei Costin, Jonas Zaddach, Aure'lien Francillon and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In Proceedings of the 23rd USENIX Security Symposium, 2014, pp.95-110 Online]. Available: https://www.usenix.org/conference/usenixsecurity14/techincal-sessions/presentation/costin.
11. Yan Shoshitaishvili, Wang Ruoyu, Hauser Christophe, Kruegel Christopher and Vigna Giovanni. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. NDSS, 2015.
12. Daming Chen, Manuel Egele, Maverick Woo and David Brumley, Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. Carnegie Mellon University, 2015.
13. Christopher Kruegel, Yan Shoshitaishvili. Using static binary analysis to find vulnerabilities and backdoors in firmware. Black Hat USA, 2015.
14. Andrei Costin, Zarras Apostolis, and Aure'lien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ACM, 2016, pp. 437-448.
15. QEMU. URL http://www.qemu.org/
16. Kai-Chi Chang, Raylin Tso, Min-Chun Tsai, IoT sandbox: to analysis IoT malware Zollard, International Conference on Internet of things and Cloud Computing, pages: 4-12, 2017.
17. Pa Yin Minn Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: A Novel Honeypot for Revealing Current IoT Threats. Journal of Information Processing 24, no. 3 (2016): 522-533. https://doi.org/10.2197/ipsjjip.24.522.
18. Suspected Mass Exploit Against Linksys E1000 / E1200 Routers. Available at: https://isc.sans.edu/forums/diary/Suspected+Mass+Exploit+Against+Linksys+E1000+E1200+Routers/17621/.
19. Ahmad Darki, Chun-Yu Chuang, Michalis Faloutsos, Zhiyun Qian and Heng Yin. RARE: A Systematic Augmented Router Emulation for Malware Analysis. In Passive and Active Measurement, edited by Robert Beverly, Georgios Smaragdakis, and Anja Feldmann, 60-72. Lecture Notes in Computer Science. Springer International Publishing, 2018.
20. Frederic Leens. An Introduction to I2C and SPI Protocols. IEEE Instrumentation Measurement Magazine 12, no. 1 (February 2009): 8-13. https://doi.org/10.1109/MIM.2009.4762946.
21. E. Volpi, F. Sechi, T. Cecchini, F. Battini, L. Bacciarelli, L. Fanucci, M. Marinis. System Study for a Head-Up Display Based on a Flexible Sensor Interface. In Sensors and Microsystems, edited by Piero Malcovati, Andrea Baschirotto, Arnaldo Amico, and Corrado Natale, 413-417. Lecture Notes in Electrical Engineering. Springer Netherlands, 2010.
22. Firmware mod kit.
23. URL https://code.google.com/archive/p/firmware-mod-kit
24. Binwalk. URL http://binwalk.org/
25. Tran Nghi Phu, Nguyen Ngoc Binh, Ngo Quoc Dung, and Le Van Hoang. Towards Malware Detection in Routers with C500-Toolkit. In 2017 5th International Conference on Information and Communication Technology (ICoIC7), 1-5, 2017. https://doi.org/10.1109/ICoICT.2017.8074691.
26. Busybox. URL https://www.busybox.net/
27. The moon malware. URL https://www.sans.org/
28. Mirai malware. URL https://github.com/jgamblin/Mirai-Source-Code
29. T. Pultarova, Webcam hack shows vulnerability of connected devices, Engineering Technology 11 (11) (2016) 10-10.

AUTHORS PROFILE

Tran Nghi Phu is currently working as lecturer and network security specialist, People's Security Academy, Hanoi, Vietnam. He has completed his Master degree in Software Engineering at VNU University of Engineering and Technology (2014). He has been doing research, application and teaching since then in the fields of network security, artificial intelligence, IoT and more recently in malware analysis. Currently, he is a PhD student at the University of Engineering and Technology – VNU, Hanoi, Vietnam. He has owned some applications in network security field at http://firmware.vn.

Le Van Hoang currently a graduate Department of Information Technology and Information Security in People's Security Academy, Hanoi, Vietnam. His research area includes malware analysis, artificial intelligence, natural language processing.

Dr. Ngo Quoc Dung is currently working as Lecturer in the Department of Information Technology, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam. He has completed his doctoral degree in Informatics applied in automation and manufacturing at the Grenoble Institute of Technology, Grenoble, France. He has been actively participated in all the research activities. He has many books and has more than 10 research papers to his credit. He also has guest edited several edited books. His research area includes network security, malware analysis, artificial intelligence, optimal energy.

Dr. Dai Tho Nguyen received his PhD from University of Technology of Compiègne, France. He is currently the Head of Laboratory of Information Security, Faculty of Information Technology, University of Engineering and Technology, Vietnam National University, Hanoi. He teaches various courses on network security, information security management, digital forensics, and ethical hacking. His research areas include information security, distributed computing and networking, with interests ranging from theoretical concerns to concrete use cases and scenarios.

Nguyen Ngoc Binh is currently working as Visiting Professor, Faculty of Computer and Information Sciences (CIS), Hosei University, Tokyo, Japan. He is PhD in Information and Computer Sciences from Osaka University, Japan and is Honorary Doctor from Toyohashi University of Technology, Japan. His profile is at *http://uet.vnu.edu.vn/~nnbinh/*.