2019

Detecting Malicious Behavior in OpenWrt with QEMU Tracing

Jeremy Porter
Wright State University

Detecting Malicious Behavior in OpenWrt with QEMU Tracing

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

JEREMY PORTER
B.S., The Ohio State University, 1999

2019
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

July 19, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Jeremy Porter ENTITLED Detecting Malicious Behavior in OpenWrt with QEMU Tracing BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

_____
Junjie Zhang, Ph.D.
Thesis Director

_____
Mateen Rizki, Ph.D.
Chair, Department of Computer
Science and Engineering

Committee on
Final Examination

_____
Junjie Zhang, Ph.D.

_____
Krishnaprasad Thirunarayan, Ph.D.

_____
Meilin Liu, Ph.D.

_____
Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Porter, Jeremy. M.S.C.S., Department of Computer Science and Engineering, Wright University, 2019. *Detecting Malicious Behavior in OpenWrt with QEMU Tracing.*

In recent years embedded devices have become more ubiquitous than ever before and are expected to continue this trend. Embedded devices typically have a singular or more focused purpose, a smaller footprint, and often interact with the physical world. Some examples include routers, wearable heart rate monitors, and thermometers. These devices are excellent at providing real time data or completing a specific task quickly, but they lack many features that make security issues more obvious.

Generally, Embedded devices are not easily secured. Malware or rootkits in the firmware of an embedded system are difficult to detect because embedded devices do not have the usual human interfaces such as a keyboard, video, or a mouse. Traditional rootkits typically come in three variants: binary, library, and kernel. Binary rootkits aim to replace a binary file in the operating system such as `ls` (list files) or `cd` (change directory). Library rootkits replace system libraries with malicious code that can intercept system calls and provide incomplete or false information as it is relayed between user and kernel spaces. Kernel rootkits hook directly into the kernel and provide false or incomplete information to system calls. Kernel rootkits are often loadable kernel modules (LKM) that can be installed at run time. Typically, countermeasures and detection methods require specific security hardware tools or scanning the system in a traditional way with some interactive inputs/outputs provided to an end user or security researcher. These methods don't work well with embedded devices that lack additional security hardware and a keyboard, video, or mouse to display or interact. A more tailored and focused approach is required for embedded devices. This thesis takes a step toward building a framework for embedded device security auditing. The first component of this framework is a malicious router, the second component is QEMU used to trace the execution of the malicious router. An example

OpenWrt router with malicious behavior is demonstrated. The system consists of a client, a router, and a server. The router contains MITM Proxy software used to monitor and modify HTTP requests. The client uses `wget` and the server uses `uhttpd` to simulate an HTTP request/response scenario. The router is able to inject/modify HTTP requests and provide a response different than what the server would provide.

The second component, QEMU with tracing is explored and shown to be an effective measure to provide truthful data with respect to the operation of the malicious router. We believe this framework is a flexible paradigm for examining embedded device firmware. QEMU offers multiple tracing methods with more granular data as required.

In conclusion, we propose a two part detection method for detecting rootkits in embedded devices. The first part, a suspect system demonstrated by a router that performs HTTP injection and a second part that uses QEMU to trace the execution of the suspect system with some level of trust. We discuss some additional malicious systems that can be used with the Diamorphine rootkit.

Contents

List of Figures

List of Tables

Acknowledgement

Thank you Dr. Adam Bryant and Dr. Junjie Zhang for advising and consulting. Thanks to my thesis committee Dr. Zhang, Dr. T. K. Prasad, and Dr. Liu for taking the time to review this thesis and for it's defense.

For nearly five years, my wife has supported my efforts to continue my education. I would not have done this with out that support. During those five years, my family doubled in size with two boys. They both inspire me with contagious happiness and relentless laughter. A special thanks to my wife for giving me occasional quiet time to work on this thesis.

Dedicated to

My wife, Sanna and my two boys Elliott and Barrett

Introduction

In recent years there has been an explosion of Internet of Things (IoT) devices [19]. While these devices provide valuable real-time data, remote control capabilities and security features, the security of the IoT device is often challenging to verify. These devices often lack traditional input/output mechanisms of a normal computer, but they have many of the same networking interfaces.

To rectify this problem, we propose a possible solution to analyzing firmware for malicious activity. Specifically, we examine the case of HTTP injection through OpenWrt, an open source Linux-based router firmware. Our goal is to work through some of the challenges to emulating the Linux kernel directly and providing analysis. We develop context for examining maliciously modified firmware with a bias toward rootkits within embedded firmware.

1.1   Motivation

With the increasing number of IoT devices and attackers looking to monetize opportunities, there is no shortage of security breaches due to poor security in embedded devices. Before we explore the technical aspects of embedded device security, let's examine the sources of malicious firmware. [1]

There are at least three sources of malicious firmware, 1) traditional malware devel-

---

[1]This section is adapted from a previous publication [31].

1

opers, 2) manufacturer developed, 3) and state sponsored. With continued reliance on technology for basic services, such as health care, military protection, and electrical power, the threat of an attack utilizing malicious firmware is growing.

Malware developers are often looking for monetary gain. Controlling a users system can be one step in a larger scheme to steal large amounts of money, resources, or other goods. Malware developers are working on long term attacks, which will give hackers an ongoing and virtually undetectable access to the target system [22, p. 126].

If monetizing is the goal, hardware and device manufactures can do it readily. Manufacturer-sponsored rootkits are being developed. Lenovo created a piece of software called Lenovo Service Engine. This software installed Lenovo based update tools directly from the BIOS. The BIOS actually copies an autochk.exe that runs every time the Windows platform boots. Autochk.exe then creates the LenovoUpdate.exe and Lenovochk.exe binary files. Regardless of whether the drive is wiped or OS reinstalled, the behavior persists [12]. Manufacturers are not always trustworthy, and this is the case with Lenovo. Early in 2015, they were responsible for the SuperFish man-in-the-middle root certificate issue. This allowed Lenovo to monitor all user web traffic as well as inject advertisements of their choosing into the browser [24].

In 2013, Edward Snowden released thousands of U.S. intelligence documents. Analysis of the documents suggests that the National Security Agency (NSA) developed malware called EquationDrug and GrayFish that obtains a payload code from the Internet to flash the existing hard drive firmware. Kaspersky researchers stated that this particular malware surpasses anything else they have seen [42].

With malware developers looking for new targets and smaller footprints, manufacturers behaving less than trustworthy, and the advent of state sponsored malware an apparent reality, now more than ever there is a need for better detection and prevention of firmware based rootkits. With state sponsored surveillance, there is a threat to basic human rights worldwide.

1.2   Important Definitions

In this section, we identify several key concepts and definitions used throughout this paper. We define firmware, rootkits and IoT devices. We will build and explore on these concepts in throughout the paper.

Firmware is a piece of software that runs directly on computer component hardware. For example, the motherboard has a Basic Input Output System (BIOS) which manages features and functionality of the motherboard and interacts with attached hardware. The hard drive, graphics-processing unit (GPU), and the processor itself all have firmware that interact with and control hardware resources. This firmware is operating system for the embedded device.

A rootkit is a malicious piece of software that is able to hide itself from detection. It can perform malicious activities such as exfiltration of data, key logging, or allowing a backdoor to an operating system [33]. Firmware rootkits are a special type of rootkit that modify the firmware. This type of rootkit can overwrite the firmware and maintain persistence and thwart detection in similar ways that standard rootkits do.

Before further exploration of embedded or IoT devices, we will describe an existing definition of embedded systems. IoT is an easy term to use, but for the remainder of this paper we will use embedded system to describe this class of devices. Specifically, embedded systems [18, pp. 3-4]:

- are a component of a more complex system

- perform a specific task

- have a small footprint (they have smaller storage/memory requirements)

- consume less power

- have specially designed operating systems

3

- interact with the physical world

- are single board computers

While these characteristics are not conclusive, they are indicative of a specific group of devices. Embedded systems often have limited inputs and outputs and often interact in unique ways. For example, the OpenWrt router system used in this research project is not generally connected to a monitor, mouse, or keyboard. There is no obvious way to examine the firmware for security problems.

1.3 Problem Statement

Embedded systems are nearly ubiquitous. Internet of things (IoT) devices are showing up in consumer devices and industrial control systems. Intel claims that IoT devices accounted for 2 billion devices in 2006 and are projected to account for 200 billion in 2020 [19].

Embedded device security is challenging. With the number of IoT devices, it's practically impossible to secure them all. These devices are designed to connect easily and provide almost instant data with little to no user interaction. There is no easy detection when a security issue occurs.

There are many examples of exploited embedded devices in the news. One example, reported by Lee Mathews in 2017 involved a casino that was attacked via an Internet connected thermometer. The attackers exfiltrated 10 gigabytes of data to servers in Finland. The data exfiltration was the suspicious component. Traffic to and from the thermometer was normal [27].

These kind of incidents are all too common. Researchers have shown interest and provided some possible solutions. Firmadyne [3] is one such solution. Firmadyne is a framework for dynamic analysis of embedded Linux firmware. It scales well and provides

4

a detailed analysis of large numbers of firmware in an automated manner. Firmadyne uses a modified Linux kernel to complete the task. It provides analysis on the root file system of the firmware rather than the Linux kernel. Thus, rootkits are more challenging to detect with this tool. Another solution presented by Costin, et. al. [6] shows very similar behavior to Firmadyne. The problem with these two approaches is that the kernel is not emulated. Without kernel emulation it is not easy to understand the behavior of that kernel and the state of it's security.

Examining an embedded system for rootkits or general security is difficult. Some of the tools and methods include using file signatures or specialized security hardware. These methods are explored further in chapter 2.

Rootkits are more difficult to detect than typical malware because they hide their own malicious code from detection [7, p. 296]. Firmware rootkits have characteristics that make them nearly impossible to remove and even more difficult to detect than traditional rootkits. Firmware rootkits can be persistent across system reboots and operating system upgrades or reinstallation. Forensic analysis can easily overlook firmware rootkits [26].

Firmware rootkits can also go a step further and remain undetected by traditional techniques because most of these techniques do not validate the hardware state [7]. Firmware based rootkits are even more difficult to detect than standard rootkits that run on the operating system. Normal forensic investigations often overlook firmware rootkits. Techniques for detection are not widely available and mostly academic in nature [26].

There are two straight forward approaches to examining firmware for trustworthiness. First, the binary data can be analyzed with static analysis. While this method can uncover issues, it can be very challenging to examine large pieces of code without an understanding of the behavior. The second method, is dynamic analysis. This means the code is running and the behavior is examined. With this approach the embedded firmware must run inside a container. The firmware cannot be trusted to provide an accurate representation of it's own behavior. With the firmware running in a virtualization container, the executed instructions

can be examined.

In this thesis we aim to provide a specific method of examining the kernel of an embedded operating system. Instead of traditional methods, we propose the use of emulation tools to emulate the Linux kernel and examine it's behavior for malicious code. Other methodologies such as Firmadyne look at web interfaces or software installed on the device that resides in a file system. Our method uses QEMU as an emulation engine to run a specially crafted OpenWrt firmware. Further, we can trace execution of the firmware instruction by instruction. QEMU tracing is a pragmatic approach that provides a systematic approach to examining CPU instructions, network traffic, and various system calls without impacting these systems directly.

1.4   Organization

The rest of this paper is organized as follows: Chapter 2 provides a more thorough examination of rootkits and their variants. Chapter 3 introduces the build process of OpenWrt with MITM Proxy installed. This is an example piece of firmware that can be analyzed in a QEMU virtualization container. Chapter 4 demonstrates HTTP injection and modification. Methodologies for instruction tracing and actual rootkit injection are discussed. Finally Chapter 5, concludes and discusses future work.

A Background of Rootkits and Detection

2.1 Three Types of Rootkits

Binary Rootkits Binary rootkits replace critical system binary files. Binary rootkits provide an attacker remote access, local access, and are useful for evidence hiding. As an example of a binary rootkit, administrative tools or common Linux based daemons could be rewritten with malicious code and distributed to unsuspecting users or administrators. Installing these malicious tools would install a rootkit and replace system binary files such as grep, ls, or login. Binary rootkits use simple hiding techniques and can be detected via integrity checking tools [4, p. 10].

Kernel Rootkits Kernel level rootkits hook into the kernel and replace system calls. These rootkits operate at the kernel level and therefore can provide faulty information to user and OS thereby hiding themselves. This type of rootkit operates using a loadable kernel module (LKM). The LKM operates at the kernel level and modifies system calls with custom malicious code. The mitigation for kernel rootkits is to disable loading kernel modules [4, p. 13].

As an alternative to modifying the kernel, the system image in memory can be modified to change the system calls. For example, the rootkit called SucKit allows covert remote login to a target system and does not require any kernel modules [4, pp. 12-13].

Library Rootkits   Library rootkits replace a standard system library that relays information between kernel space and user space. The t0rn rootkit uses `libproc.a` to accomplish this task. The `libproc.a` library sanitizes results from the kernel. When a user requests a directory listing or process list, these can be falsified. To mitigate this an administrator can look directly at the `/proc` file system. Additionally, information can be relayed from a malicious library to the kernel. In other words, false information can be fed into the kernel. Any application linked to the library will report falsely. However, statically linked applications are not affected. Another method is to modify `/etc/ld.so.preload` placing the custom library before other system libraries that redirects standard calls to ensure malicious calls will execute in place of regular calls. Linux tools such as ltrace, strace, ltruss can trace library and system calls to detect this behavior [4, pp. 13-14].

The standard rootkits listed above have relatively well defined detection techniques. Firmware rootkits use similar techniques as standard rootkits. Firmware rooktits differ in where they are stored. Firmware rootkits infect and hide within firmware and therefore are persistent across reboots, system upgrades and reinstallation. Furthermore, removal of such a rootkit involves replacing the infected firmware, replacing the affected hardware or replacing the entire system hardware. Detecting a firmware rootkit can be particularly challenging because they do not reside within the operating system. The typical detection techniques do not apply here [41].

2.2   Rootkit Detection and Mitigation

2.2.1   Traditional Rootkit Detection

Existing rootkit detection techniques and their effectiveness are outlined by Campbell, et. al. [7, p. 307] These methods include file integrity, platform integrity and kernel integrity checking. Additionally, examining file signatures and physical memory can be

used to uncover rootkits.

2.2.2 File/Disk Integrity Checks

Integrity comes in three variations: file integrity, platform integrity, and kernel integrity. File integrity is very simple in that periodically checks the hashes of given files to ensure they have not changed. One tool that provides file integrity checks is Tripwire [7, p. 305].

2.2.3 Platform Integrity Checks (TPM)

Trusted Platform Module can provide platform integrity during boot using Platform Configuration Registers (PCRs) to store checksums of firmware. These checksum can be compared against current values to looking for modifications. [7, p. 306].

2.2.4 Runtime Kernel Integrity Checks

Runtime integrity check can be done using checksums of kernel code and immutable data. This is also possible with specialized hardware such as CoPilot, a PCI card that can read kernel memory via DMA. Performance counters on the processor can also be used to check the integrity of the kernel [7, p. 306].

2.2.5 File/Disk Signature Checks

Signature checking is simply looking for known malware based on signatures. This is heavily dependent on an up to date database of signatures. One example of software that uses signature check is chkrootkit [7, p. 306].

2.2.6 Physical Memory Signature Checks

Aside from file and disk, memory can be scanned for known signatures. Specialized hardware can be used to accomplish this to avoid reliance on the kernel. Hardware provides the advantage of being able to check the entire memory space [7, p. 306].

David, Chan et al. noted that their proof of concept rootkit cloaker was very effective at bypassing most existing detection approaches [7, p. 307]. These methods are insufficient to detect a firmware rootkit because embedded devices do not have mechanisms to provide the kind of data expected.

2.3 Firmware Rootkit Taxonomy

This section discusses different types of firmware rootkits. Each type discussed is listed below.

- BIOS

- Hard Drive Firmware

- GPU Firmware

- System Management Mode

- PCI Expansion ROM

- Internet of Things Device Firmware

2.3.1 BIOS

The BIOS is stored on an EEPROM chip on the motherboard of a system. The BIOS settings are stored in CMOS (Complementary Metal Oxide Semiconductor). BIOS based rootkits have been around for several years. Only in the last few years have they been

used to infect computers on a larger scale. John Heasman presented at Blackhat Europe in 2006 regarding BIOS based rootkits. He indicated some of the key features of firmware based rootkits, such as persistence over reboots, low footprint on disk, ability to reinfect installations of new/same OS, and the difficulty with detecting and removing BIOS based rootkits. Heasmans rootkit leverages the Advanced Configuration Power Interface (ACPI) to create a backdoor to both Windows NT and Linux based operating systems [17].

One of the earliest known BIOS attacks was called the CIH virus. CIH surfaced in June 1998 and simply erased the BIOS of the affected system. While this is not a rootkit, it was successful in disabling the system and notable in that the system BIOS can be modified [11]. The first BIOS based rootkit called Mebromi surfaced in 2011. Mebromi is much more advanced, though still lacks some functionality [13].

Giuliani argues that the complexity of BIOS rootkits means that they will not be much of a threat [13]. However, a modern BIOS rootkit developed by an Italian security group known as Hacking Team shows that it is still a viable attack. It appears from analysis that physical access to the target is required to infect the system. However, this rootkit actually behaves similar to the Lenovo rootkit described above. It simply ensures a specific service (Remote Control System) is installed [25].

2.3.2 Countermeasures

Heasman, at the time of his presentation in 2006, suggests that existing tools such as VICE, Blacklight, and RootkitRevealer can be used to detect BIOS based rootkits [16]. Implicitly, this must be based on the changes that are made to the operating system files, memory, etc. Secondly, Heasman suggests operating system event logs and auditing regarding ACPI to detect the proof of concept he presented [17]. These methods seem less than ideal, but at the time they were all that was available.

The more modern UEFI rootkit developed by Hacking Team has a more formal detection method provided by Intel. The Intel tool is called CHIPSEC [21]. According to

the GitHub website, CHIPSEC is a framework for analyzing the security of PC platforms including hardware, system firmware (BIOS/UEFI), and platform components [20]. In a CanSecWest 2014 presentation, Intel researchers presented CHIPSEC and detailed a number of its features, including the ability to verify SMRAM is locked [20].

2.3.3 Hard Drive Firmware

Before understanding how to detect and mitigate hard drive firmware, it is instructive to have a brief overview of the components of a hard drive with respect to this topic. The main focus will be on the electronic components on the PCB (Printed Circuit Board). For reference, there is a picture with callouts below. [8]
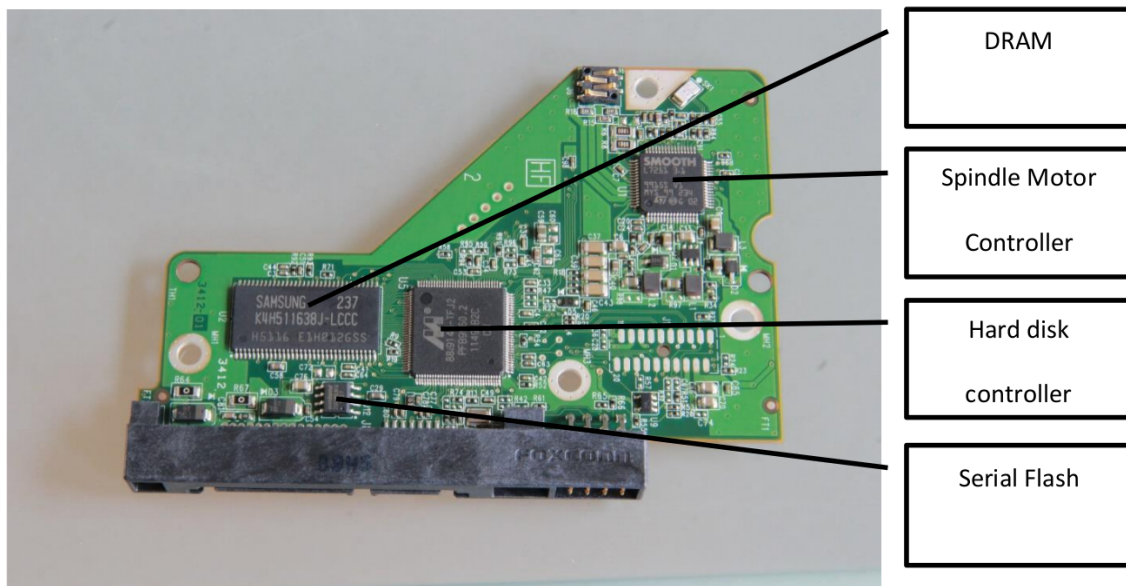

Figure 2.1: Hard drive controller

Some clever reverse engineers have done some work to document how the hard disk controller works. It seems that the hard disk controller supports JTAG (Joint Test Action Group). JTAG is an interface used for debugging and programming processors. With this information, it is possible to control a controller. It is possible to start, stop, modify

Table 2.1: Hard drive controller component descriptions

Part	Notes
DRAM	DRAM ranges from approximately 8 MB to 64 MB and is the cache for the hard drive.
Spindle motor	The spindle motor controller drives the motor on the hard drive.
Serial flash	Serial flash ranges from 64 KB to 256 KB and stores the program that disk controller boots from.
Hard disk controller	The hard disk controller is made by various manufacturers (Marvell, ST, LSI). Some HDD manufacturers make their own. These parts are not well documented.

memory, and set breakpoints. Furthermore, it is possible to dump out the boot ROM of the controller in using the serial port [8].

The attack as it stands is not very practical. An attacker would need physical access and a significant amount of time to reverse engineer a specific hard drive, modify the firmware, and reflash the firmware. However, after gaining some basic understanding of how the controller operates it is possible to skip the JTAG interface all together and directly use the serial flash on the PCB. Simply rewriting the code on the serial flash is sufficient. This too is not very practical, but it does provide the benefit of persistence. The rewritten code will run every time the hard drive boots up [8].

The step that makes this firmware attack practical is understanding the hard drive manufacturers have already made a software tool to reflash the firmware. It is also possible to hide data in service areas on the hard drive [1]. Another set of tools called idle3-tools[2] can be used to modify the hard drive behavior. The idle3-tools code can be modified to actually flash the firmware of the HD. At this point the attack is feasible [8]. All that is required is access to the hard drive. This may not be trivial to obtain, but a malicious supplier or vendor could readily modify hard drives this way. Or more likely, a malicious party with access to the manufacturer could inject their own code into the firmware.

Now that it is possible to modify this firmware in a feasible manner, the question is what can be done with this? One example is to wait until a machine reads the `/etc/shadow`

file where passwords are stored on Linux systems. The contents of the file could modified to include something preconfigured and thus an attacker could log on to the machine uninhibited [8].

2.3.4 Countermeasures

Some of the suggested mitigation techniques are applicable to most rootkit detection methods. Simply verifying the operating system integrity is the most obvious technique. But this kind of rootkit need not modify the operating system to be effective. It may also be productive to check network traffic for data exfiltration [40, pp. 48-51].

More specific methods to detect HDD firmware rootkits include reading data from the hard drive after a write occurs and verify the integrity. Another method is through remote attestation as indicated by the Trusted Computing Group [40, pp. 48-51].

2.3.5 GPU

There are at least three examples of GPU based malware at present: jellyfish, Demon Keylogger, and WIN_JELLY. A brief discussion of these three pieces of malware follows.

2.3.6 jellyfish

According to the GitHub page where the "jellyfish" code resides,

"Jellyfish is a Linux based userland gpu rootkit proof of concept project utilizing the LD_PRELOAD technique from Jynx (CPU), as well as the OpenCL API developed by Khronos group (GPU). [] some advantages of gpu stored memory:

- No GPU malware analysis tools available on web

- Can snoop on CPU host memory via DMA

- GPU can be used for fast/swift mathematical calculations like xor'ing or parsing

14

- Stub/signature generation

- Malicious memory may be retained across warm reboots. (Did more conductive research on the theory of malicious memory still being in GPU after shutdown)" [23]

At this point in time, no formal academic analysis of the proof of concept jellyfish rootkit is available. However, one article claims that jellyfish can be used to implement a keylogger with minimal GPU utilization–approximately 0.1%. This small usage is difficult to detect from normal behavior. Further, jellyfish can use direct memory access (DMA) to snoop on system memory. DMA allows direct access without involving the processor, further complicating detection of this kind of malware.

2.3.7 Demon Keylogger

Demon keylogger is a keylogger implemented in jellyfish. This keylogger is described by Ladakis, Koromilas, Vasiliadis, et. al. According to their paper, this is the first presentation of a GPU based keylogger as presented in April of 2013. They claim that their keylogger can capture all keystrokes with minimal overhead and no disruption to normal graphics processing [23, p. 1]

2.3.8 WIN_JELLY

WIN_JELLY is remote access toolkit (RAT) [14]. The source code is available on GitHub. As with the other mutations of jellyfish there no formal academic analysis of this code was found. As the name indicates, this software runs on the Microsoft Windows platform whereas the initial implementation of jellyfish was Linux based.

2.3.9 Countermeasures

Most of the standard methods apply to CPU architecture only and are not effective against GPU based malware [23, p. 4]. Further, there are few tools for forensics analysis of GPU malware [35].

Runtime detection. One method is to use DMA analysis. This type of analysis is looking for attacks that include bulk DMA transfers. However, not all GPU based malware does this kind of transfer. Another possible defense is to profile the GPU usage and access patterns. Presumably, under normal operation a baseline profile will be different than one that includes malware.

Aside from the countermeasures mentioned by Ladakis, Koromilas, et. al. no GPU specific countermeasures were found during the research for this paper. However, McAfee Labs rebuts the dangers of GPU based rootkits. According to McAfee Labs, there are four main points with respect to GPU based rootkits and malware:

- CPU host memory access from the GPU.

- Subsequent deletion of CPU host files.

- Persistence across warm reboots.

- Absence of GPU analysis tools.

The first claim regarding CPU host memory access from the GPU requires physical memory be mapped to the GPU. Unprivileged code is constrained "to memory mapped to a process's virtual address space, making ring 0 access a requirement" [28, p. 27]. If this is in fact the case then this kind of mapping is more difficult than it seems.

Further, once code is running on the GPU, the installation files for that code can be deleted. On Windows platforms, this will cause a Timeout Detection and Recover process to initiate. Although this value is configurable, any modification is considered suspicious and anti-malware could detect this behavior. Additionally, GPU code that runs for a long

period may cause an unresponsive GUI on Windows and other platforms as well. In the case of multiple GPUs or systems with not visual interaction (monitor) this may go unnoticed [28, p. 27].

As for persistence, McAfee Labs claims that code is stored on the GPU but it is not executing. Thus, in this case persistence refers to stored code retrieved from GPU memory and executed in user mode on the system. Therefore, the malicious code executing outside of the GPU can be detected using traditional malware techniques [28, p. 28].

McAfee confirms there is a lack of GPU analysis tools. However, some articles mention a tool called "JellyScan" as well as research being conducted by Dr. Golden G. Richard III.[1] At this time, there is no specific information about this tool or other forensic analysis tools of GPU hardware.

However, in a paper on GPU-Assisted Malware the authors contradict some of the claims that McAfee Labs makes. They write that code can be statically linked with the CUDA library to make self-contained malware. Additionally, GPU code execution does not require root privileges–it will run in user mode [38, pp. 290-291].

2.3.10 PCI Expansion ROM Rootkits

Another unique way to store a rootkit is in a peripheral PCI card within the system. These cards have some flashable ROM and therefore are candidates for storing malware [16, p. 4]. During a system boot, a system interrupt can be hooked allowing arbitrary code on a PCI ROM to be executed. At this point, an attacker has control of the system [16].

2.3.11 Countermeasures

Since the rootkit does not rely on memory that cannot be accessed in kernel mode and it uses the system processor (rather than ones on the PCI cards), the best detection

---

[1]See http://www.cs.uno.edu/ golden/gpu-malware-research.html

is looking at the modifications the rootkit has made to the system structures. Traditional methods may work well here. A second detection method is to audit the ROM. Heasman documents steps taken to obtain the contents of ROM and provides several characteristics of potentially malicious code in PCI ROM [16, p. 11].

Preventative measures for PCI ROM rootkits include a write protect switch available on some PCI cards and Trusted Platform Module. A traditional write protect switch simply prevents the ROM from being written. Trusted Platform Module (TPM) may also be a mitigation for PCI ROM rootkits. TPM actually works by building a Core Root of Trust Measurement (CRTM). This core is configured on a new system or a trusted system with its existing configuration. The TPM takes hash values of the ROM code in the PCI cards at this point and stores them. During a normal system boot with TPM these hash values are verified for change [16, p. 13]. While this will thwart a new piece of malware from infecting an existing PCI ROM, it does not verify an unknown PCI ROM. The other problem with TPM is that there is no validation of hardware from the manufacturer. If the manufacturer supplies hardware already infected with malware, TPM will not mitigate this.

2.3.12 System Management Mode (SMM) Rootkits

Intel architecture defines four processor modes: Real Mode, Virtual-8086 Mode, Protected Mode, and System Management Mode. Real mode and Virtual-8086 mode are legacy modes. Real mode has a 20 bit addressable memory space and provides no hardware memory protection. Modern operating systems run in 32-bit or 64-bit protected mode. Protected mode adds support for paging, memory protection, and multi-tasking. System management mode is a mode on Intel processors for low-level hardware control. Code at this level is not be preempted, does not have privilege levels, and does not honor memory protection. [10, p. 1593]. (This all needs verified with Intel architecture docs)

System Management Mode has a memory space to operate in called SMRAM. This holds the processor state, the SMI (System Management Interrupt) handler, and SMI han-

dler data. Intel identifies three locations for SMRAM: Compatible, High Memory, and Top of Memory Segment. The compatible region is the default location. Usually, SMRAM is only accessible by code, executing in SMM. The other two areas, High Memory and Top of Memory Segment extend SMM from a usual 128K to 256K of memory. When the processor receives a System Management Mode Interrupt, it enters System Management Mode [10, p. 1593].

To install a SMM Based Rootkit (SMBR) code must be written to the SMM handler area of SMRAM. The code must also be able to intercept system events and control execution of the system. There are three criteria to install a SMM handler: I/O port access privileges, access to map physical memory, and SMRAM must not be locked by the BIOS or the operating system software [10, p. 1593]. A SMBR proof of concept is able to provide an attacker key logging and data exfiltration [10, p. 1598]. SMBR provide stealth as they do not operate with caching or paging and therefore do not impact cache or translation lookaside buffer (TLB)[10, p. 1603].

2.3.13   Countermeasures

SMM rootkits are much more difficult to detect since they do not need to modify the operating system. This renders traditional heuristic methods useless. SMM rootkits also conceal memory footprints therefore signature based detection is not feasible [10, pp. 1603-1604].

To detect SMM rootkits there are two possible solutions suggested by Embleton, Sparks, and Zou: indirect detection and timing. Based on the SMM rootkit proof of concept produced from their work, the processor timestamp is updated while code executes in SMM. Therefore checking the timestamp before and after a SMI may provide a method of detection. However, SMM rootkits can modify the timestamp before passing control to the operating system [10, pp. 1603-1604].

Indirect detection encompasses cache and TLB discrepancies. SMM rootkits can re-

side in uncached memory and they do not rely on paging therefore, these indirect methods are not useful here. Another option is to use another device that has access to physical memory to perform detection. In existing architecture, SMRAM is not accessible to the system bus [10, p. 1603-1604]. This kind of method would require a hardware architecture modification. Lastly, the IOAPIC table contains a list of interrupts that have been rerouted. The SMM rootkit does use rerouting, but there are legitimate uses of rerouting as well such as legacy support for USB mouse and keyboard devices.

The authors Embleton, Sparks, and Zou suggest that focus be put on prevention rather than detection. If manufacturers secure the SMRAM register in the system BIOS this type of rootkit can be thwarted. Additionally, operating systems may lock the SMRAM register during the early boot phases. A third party anti-malware vendor could write a kernel driver to lock the SMRAM register during boot. It is not easy to guarantee that this driver will load before a malicious driver. As noted earlier, there is a tool to by Intel to check for SMRAM locking [20].

2.3.14 Internet of Things Rootkits

The Internet of Things (IoT) refers to devices that have Internet connectivity but are not necessarily traditional hosts. This category includes but is not limited to smart TVs, home automation devices, VoIP phones, and routers. In a recent study Costin, Zarras, and Francillon built a cloud based framework to validate the framework of these devices. They found approximately 10% of the firmware tested had some form of vulnerability [6, p. 1]. While the authors do not explicitly state these vulnerabilities can lead to firmware-based rootkits, it is likely that at least some of the vulnerabilities found lead to firmware compromise.

Costin, Zarras, and Francillon performed both static and dynamic analysis of embedded web interfaces within the firmware. From static analysis, they found cross-site scripting and file manipulation were the majority of the vulnerabilities. From dynamic analysis, they

found a variety of vulnerabilities including command injection, cross-site scripting, and cross-site request forgery [6, p. 8].

2.3.15 Countermeasures

The common countermeasure for embedded firmware is for the manufacturer to provide an update to the firmware. Costin, Zarras, and Francillon were not researching firmware rootkits, but rather firmware vulnerabilities in general. It is unlikely that a single countermeasure will apply to all firmware. However, there is a move to include TPM architecture into Internet of Things devices [37].

2.3.16 Rootkit Detection Evolution

Traditional detection techniques include file integrity or rootkit signatures. These techniques are not effective against rootkits that do not modify operating system data or code [7, p. 296]. New tools such as Malware Analysis System for Hidden Knotty Anomalies (MASHKA) described by Korkin and Nesterov can be used to detect kernel mode rootkits [22]. Another architecture, Rootkit Guard relies on TPM, TrustedGRUB, and SELinux to create a rootkit resilient environment [39]. While evolution of detection and mitigation techniques is taking place with these tools, they are not applied specifically to firmware rootkits. This is an area for further exploration and examination.

2.4   A Rootkit Detection Framework

One of the main techniques to detect a rootkit is not relying on the suspect system to provide answers. In another words, the system can't be expected to give a truthful answer about it's security state. To work around this, a virtualization tool such as QEMU can be used to emulate the system. In this way QEMU can be instrumented to provide data such

as executed instructions and register values.

Methods: Analyzing OpenWrt Running in QEMU

3.1   A Brief Overview

In our research, we built an OpenWrt system and installed MITM Proxy. We used QEMU to emulate the system, along with a client and server. Options within QEMU allow for logging executed instructions, i.e. `-d in_asm`. This is a key importance of the system we built. The primary importance is gathering information about the system from outside the system. In other words QEMU is a trusted system and I expect that QEMU reliably provides truthful data about the embedded firmware it emulates.

3.2   Motivation to use MITMProxy

Our initial research showed we could use `squid` proxy [34] along with `c-icap` [36] to perform HTTP injection. The ICAP protocol is documented in RFC 3507 [9]. After some trial and error, it is clear that this method is complex and requires additional devices. For this research project, we decided it is best to keep the architecture as simple as possible. For that reason, an alternative solution, MITM Proxy is used.

MITM Proxy has several benefits. First it is an all Python implementation. Although

it does have several prerequisites outside of Python, the code to run MITM Proxy is Python and it is installed using the PIP Python package manager. OpenWrt is capable of running Python 2 and Python 3. These options must be chosen during the build process.

3.3   System Architecture

The system consists of three primary devices: a router connecting a client and a server. Each of these three devices is an instance of OpenWrt that is built to run in QEMU. Below is a diagram of the network. The server contains a `uhttpd` service to host a simple web page. The client uses `wget` as a web client.

Figure 3.1: Simplified Architecture

client   router   server
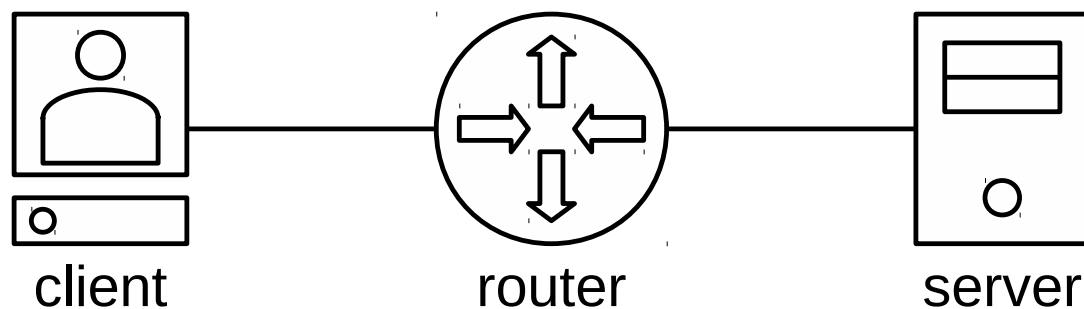
We created a virtual bridge called "lan-bridge" to connect the client and router. We created another bridge called "wan-bridge" to connect the router and server. Both of these bridges are created via `bridge-utils` in an Ubuntu 18.04 host. Each of the three device has an interface connected to one of the two virtual bridges. A more accurate diagram is below in 3.3.

In the following sections, each component and it's specifications are outlined. The components outlined are the OpenWrt router containing the MITMProxy software, the client containing the `wget` utility, the server containing the `uhttpd` server utility, the

Figure 3.2: Complete architecutre

client   lan switch   router   wan switch   server
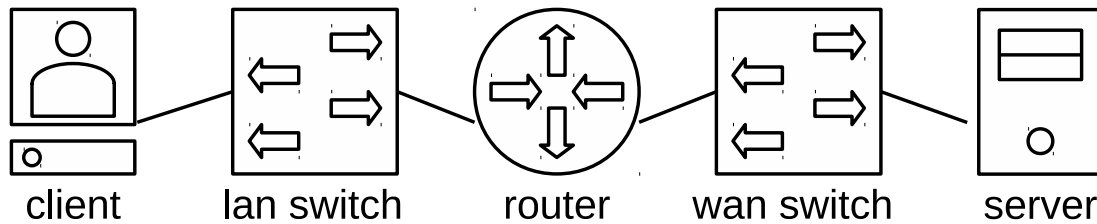
network bridges and how they are created, the network interfaces created for each of the devices, and finally, the MTIMProxy software and it's requirements.

3.3.1   OpenWrt Router

According to https://openwrt.org, OpenWrt is a "Linux operating system targeting embedded devices." We chose to use OpenWrt as an embedded system because it works readily as a router, it uses a standard Linux kernel, and allows for customization.

The OpenWrt router is built with very few configuration changes from the default. The default configuration is shared among the router, client, and server. The primary changes are made to the router to accommodate MITMProxy requirements. We made modifications to include various prerequisite Python 3 libraries, use the `glibc` library instead of `musl` library, storage considerations, and a network script (outlined in the networking section). In addition, the `uhttpd` and `wget` utilities are selected in the configuration for the client and server. These utilities are not used on the router. These specific requirements will be outlined in detail below. It is worth noting here that while the default configuration functions at a basic level, the configuration requirements for MITMProxy are challenging to configure properly. The network script for the router is built into the image in the appropriate location `/etc/config/network`.

The slightly abridged configuration used to build OpenWrt is shown in appendix D. The configuration shown includes only enabled options. Additionally, the network config-

uration used for the router, server and client are shown in appendix E.

3.3.2   Client

The client is an instance of an OpenWrt. It is only connected to one network therefore routing function is effectively nullified. The client contains the same default configuration as the router. The client network script is described below in the networking section. The client relies on the `wget` utility. The client doesn't require any additional configuration changes. The network script is built into the image in the appropriate location `/etc/config/network`.

3.3.3   Server

The server is also an instance of OpenWrt. It only connects directly to the router and has no need for routing outside of the single network connection. The server relies on the `uhttpd` web server component. It contains a simple web page. The network script for the server is built into the image in the appropriate location `/etc/config/network`.

3.4   Networking in OpenWrt

The first step to networking in QEMU is creating a virtual switch (bridge). Each of the three devices has a virtual interface connected to one of two virtual bridges. The first bridge is designated for network A. The second bridge is designated for network B.

3.4.1   How a Network Router Works

In order for devices on different networks to communicate a router must "route" traffic between the networks. The router does this via a routing table. The routing table contains

the network ID and the exit interface and/or the next hop interface. The exit interface is the network interface the traffic should go to if it is headed toward the corresponding network ID. The next hop interface is the IP address of the next "hop" on the path to the traffic's destination. Traffic can get to this information using a Layer 2 OSI protocol such as Ethernet. In this simple case the router will have only a few entries in the routing table.

3.4.2 Network Bridges, Interfaces, and Configuration

We created all of the network bridges and interfaces on an Ubuntu 18.04 host system with the appropriate QEMU emulation tools installed. We created two network bridges using `brctl`. One bridge is designated to connect the client and router and another bridge is designated to connect the server and router. We used `tunctl` to create four interfaces. The client and server each have one interface and the router has two interfaces – one for each network. All of these items are documented in a bash script in appendix F.

We added a third virtual interface to the router OpenWrt instance to communicate with the Internet. We only did this to install MITMProxy. It is possible to install MITMProxy without direct Internet access, but it is more cumbersome to do so. Once the proxy software is installed, the third interface may be removed.

Network Configuration

We created two networks connected by a router as shown below in figure 3.3

Neither the server nor client need to perform any additional routing functions. The router will contain a routing table and manage sending traffic from one network to another. This requires no additional configuration in OpenWrt.

The complete network configuration scripts are included in appendix E. Note in the router configuration, the Internet interface is used to install MITM Proxy and afterward it can be disabled during HTTP injection.
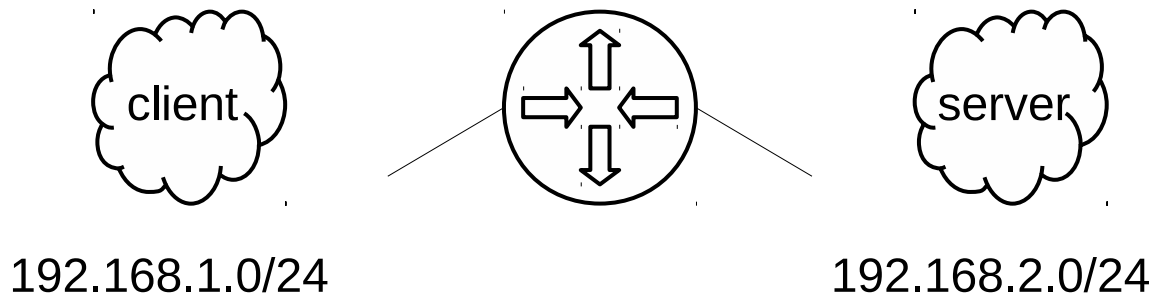
Figure 3.3: Client and server networks connected by a router

3.5 Building Openwrt

Openwrt relies on the Kbuild system. This is the same system the Linux kernel relies on. There are some customization to the build system for Openwrt but those are mostly not necessary for this work. Generally, building OpenWrt is a straightforward process that is mostly well documented. For this specific application there are a number of specific configuration pieces documented in the following sections.

3.5.1 Prerequsites for MITM Proxy

MITM Proxy is described as a "swiss-army knife for debugging, testing, privacy measurements, and penetration testing" [5]. As it stands, there is no simple way to install MITM Proxy into OpenWrt. MITM Proxy is implemented in pure Python and can be installed using package installer for Python (PIP). This greatly simplifies the process, but there are still a number of dependencies. A rough outline of the dependency tree is shown below in 3.4. In addition there are sub dependencies which are detailed below.

We identified two significant hurdles to installation: OpenWrt must be built with `glibc` as opposed to `musl` and `glibc` must be built within OpenWrt for a UTF-8 console environment. The UTF-8 console environment is required for the HTML text encoding. The `locale` binary allows the selection of UTF-8 in the console environment. To

28

Figure 3.4: Dependency Overview
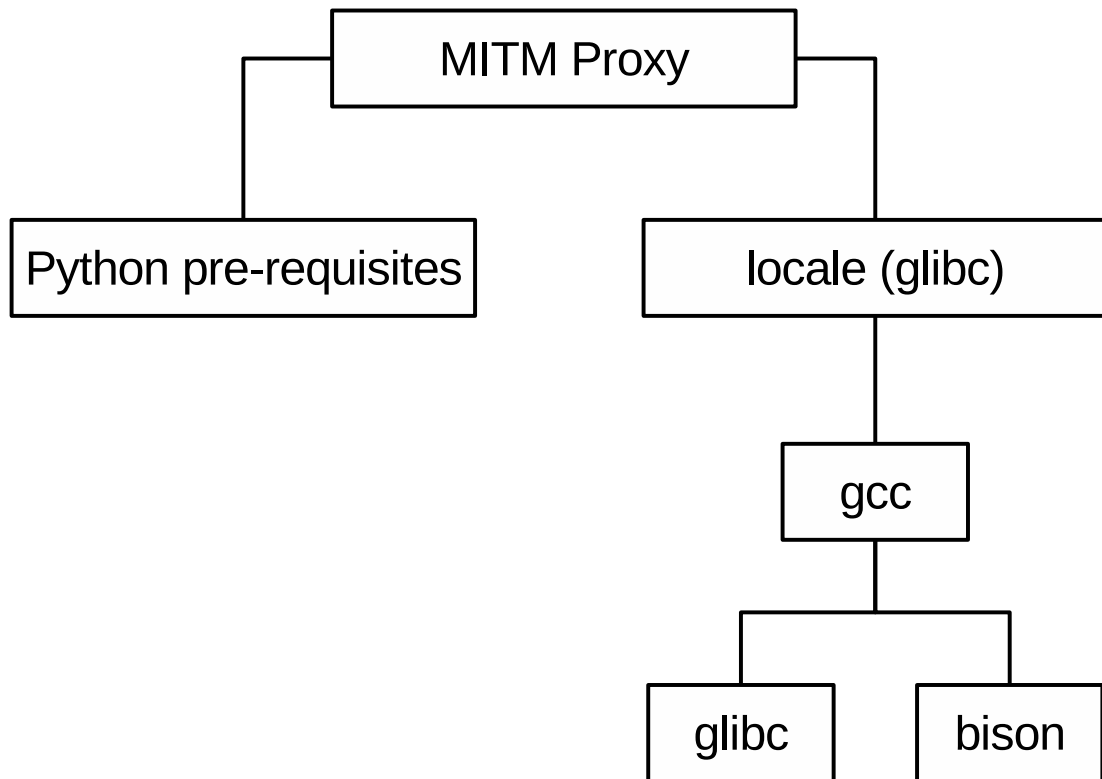
obtain the `locale` binary we need to compile `glibc`.

The build process is involved and requires prerequisites to MITM Proxy in addition to customization of the resultant OpenWrt system image. The first step to get MITM Proxy installed in OpenWrt is selecting the Python prerequisites. MITM Proxy requires little outside of the Python prerequisites and the `locale` binary to meet the UTF-8 console environment requirement. The `locale` binary is the real challenge. The `locale` binary is found in `glibc`. In turn, `glibc` requires `bison` to build within a running OpenWrt instance. In turn, building `bison` required the built-in `gcc` compiler to be modified to include the `gcc-ar` utility.

The first task is to build OpenWrt with appropriate Python prerequisites to support MITM Proxy. This process is moderately unstable and requires some effort. The best

approach is to build OpenWrt incrementally with small changes. However, this is also the most time consuming. Once prerequisites are identified it is a bit quicker, though a build takes several hours.

The following is a list of Python prerequisites for MITM Proxy. Many of these prerequisites have prerequsites. The full versioning and prerequisite tree is listed in appendix C.

- blinker

- brotlipy

- certifi

- click

- cryptography

- h2

- hyperframe

- kaitaistruct

- ldap3

- passlib

- pyOpenSSL

- pyasn1

- pyparsing

- pyperclip

- ruamel.yaml

- sortedcontainers

- tornado

- urwid

- wsproto

Building `glibc` requires a `gcc` compiler and MITM Proxy pre-requisite `crypt-opgraphy` requires modifications to the default `gcc` in OpenWrt. In a traditional Linux installation this is trivial. In fact, OpenWrt already has this option built-in. Unfortunately, the OpenWrt built-in `gcc` is artificially limited and does not include all of the important utilities of `gcc` necessary such as `gcc-ar`, an archival utility required to build `glibc`. Modifying the `gcc` compiler within OpenWrt is not overly challenging. It consists of modifying the existing `Makefile` for `gcc` and modifying to include `ar`. The modified `Makefile` is listed in appendix A.

After modifying the `gcc` `Makefile`, additional modifications to the `gcc` source code are necessary. `gcc` versions 2.26 and later do not include `struct ucontext` [32]. The offending code is located in `openwrt/build_dir/target-x86_64_glibc/gcc-5.4.0/x86_64-openwrt-linux-gnu/libgcc/md-unwind-support.h` at approximately lines 61 and 144. The full modified code for `md-unwind-support.h` is located in appendix B.

After building OpenWrt, build `glibc` on the resultant Linux based router. The fastest way is to include the source in the initial build so that building can take place right away. Building `glibc`, follows the traditional pattern of `configure`, `make`, and `make install` commands. However, to build `glibc`, the prerequisite of `bison` must be built and installed first. We also made a symlink to the appropriate location to `/usr/bin/bison`. Once `glibc` is built, MITM Proxy can be installed into OpenWrt through PIP.

Results

4.1 MITM Proxy Configuration

Using MITMProxy requires some configuration and a good understanding of how network traffic will flow. In this case, there is a client and server connected by a router. Traffic flows over the router where MITMProxy is installed and should be modified by the router directly. To make this work, the traffic must be redirected to the proxy prior to the routing function. In other words, before traffic is routed at OSI layer 3 it must be intercepted and sent to the proxy. This is done using iptables.

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j
REDIRECT --to-port 8080
```

The statement above means that traffic coming into the routers interface named eth0 using the TCP protocol port 80 will be redirected to port 8080. Port 8080 is the port that MITMProxy listens on. This causes the traffic to be captured (but not intercept)[1] by the proxy then sent on to the destination. Before traffic can be intercepted an intercept filter must be created. The filter text in MITM Proxy is q. Once traffic is intercepted by MITM Proxy, the request can be modified.

---

[1] MITMProxy distinguishes between captured traffic and intercepted traffic. Captured traffic cannot be modified and sent on to the destination, whereas intercepted traffic is modifiable.

4.2 Performing HTML Injection with MITMProxy on Open-Wrt

Once an OpenWrt router is built successfully and MITMProxy is installed. It is possible to perform a complete man-in-the-middle injection of HTML. Below is a diagram of the network. The client and server are actually instances of OpenWrt. The server contains a `uhttpd` service to host a simple web page. The client uses `wget` as a web client.

Figure 4.1: Simplified Architecture

client   router   server
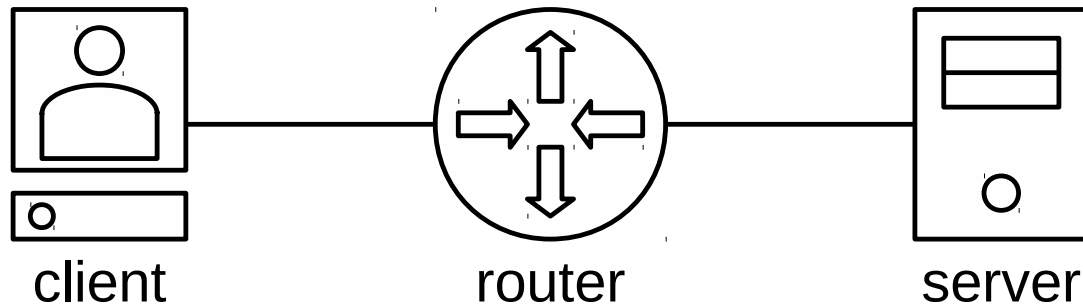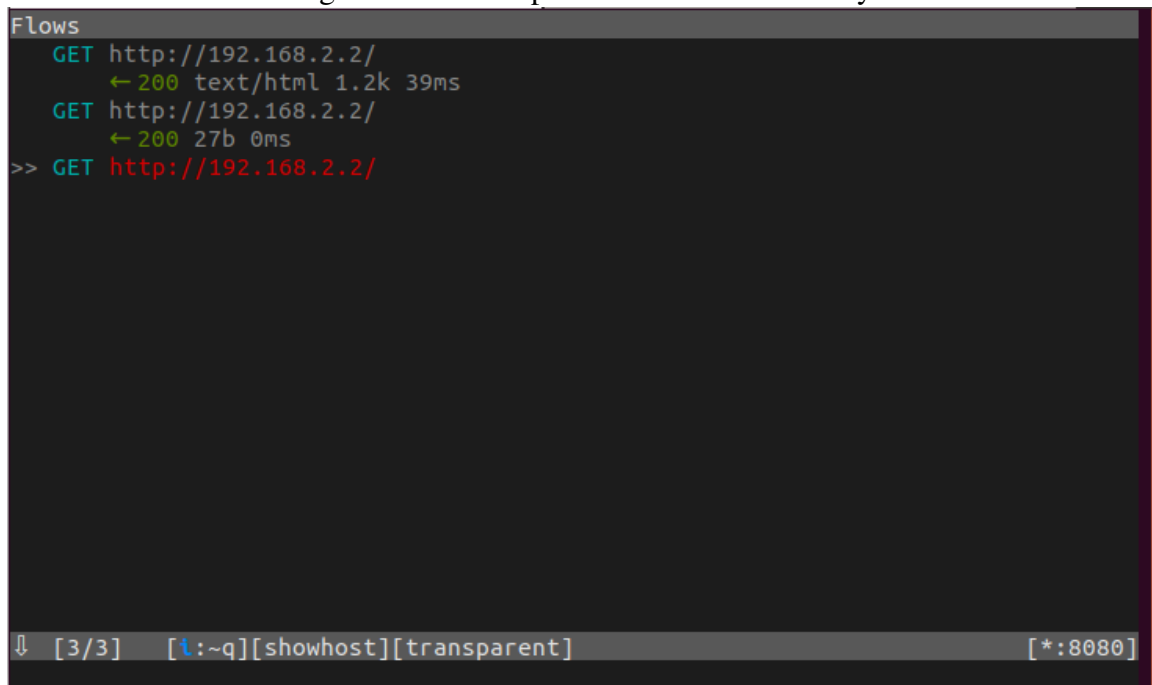
The scenario has the following steps:

1. The client requests the web page. At this point the `wget` application creates a request and sends the request down the network stack (OSI model). The traffic is sent out over the wire. In this case the wire is a virtual switch created with `bridge-utils`. The intercepted page is shown in 4.2

2. The router receives the traffic on the interfaces connected to the client network. Before the traffic is routed it is intercepted by the MITMProxy. MITMProxy reads the request and a response is crafted manually or dynamically using a script. MITMProxy crafts the response and provides the response directly to the router. The modified response is shown below in 4.2.

33

3. The router provides a response to the client's request. The router forwards the modified request to the client.

4. The client receives the request. The client has no way to know or detect that the router did all of the work of providing the HTTP response. The received page is shown in 4.2.

Figure 4.2: Intercepted traffic in MITMProxy

The scenario can be altered to allow MITMProxy to modify the original server page.

4.3 Generated Assembly QEMU

QEMU works by translating target instructions to host instructions. When a system like OpenWrt is emulated in QEMU it the instructions from the target OpenWrt system get translated to something the host can execute. This architecture has the ability to handle

Figure 4.3: Modified response in MITMProxy
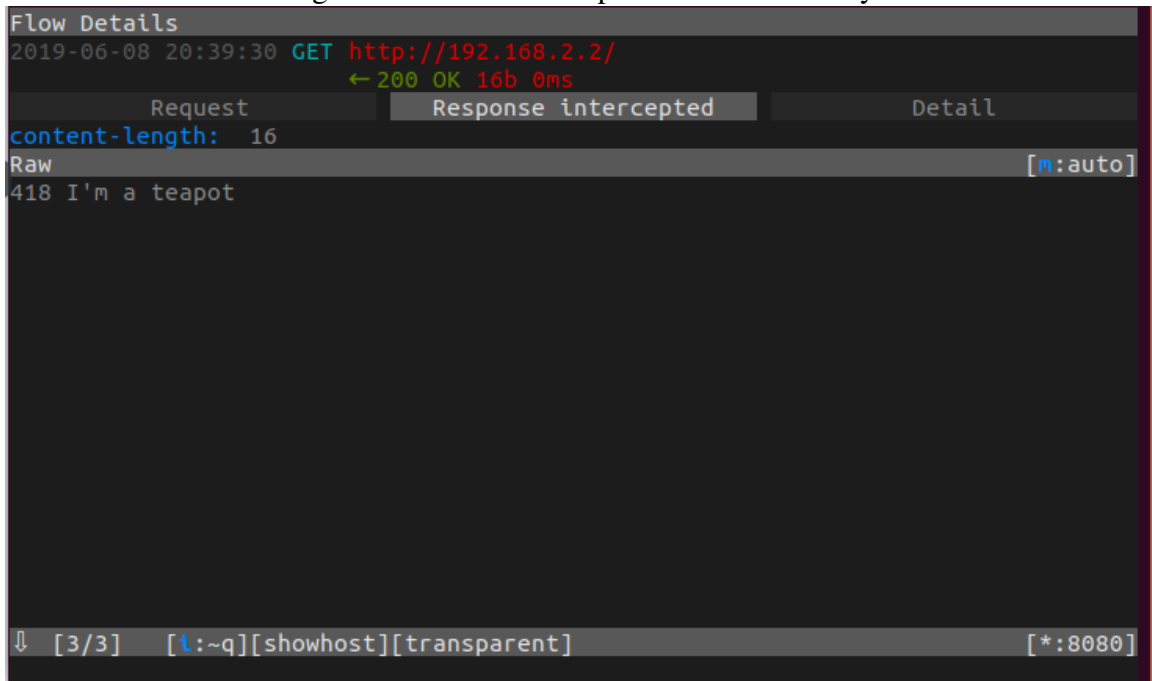
instruction sets other than the native host operating system i.e. the host can be x86_64 and the target can be MIPS. This architecture also provides an opportunity for QEMU to optimize instructions.

QEMU allows for these instructions to be logged both before and after the translation occurs along with additional options. The options in QEMU are `in_asm` and `out_asm`. With these options enabled the output of the OpenWrt boot process is logged. An example log is shown in figure 4.5. These instructions ultimately align to what is in the `bzImage` file. `bzImage` is a compressed Linux kernel with some additional boot information added. With just a little analysis of the QEMU log and the `objdump` of `bzImage`, the instruction alignment is confirmed.

The figure 4.6 shows the `objdump`. The first instruction in the `<.setup>` is `jmp 0x268`. Examining the QEMU log in figure 4.7 shows that the `jmp 0x268` instruction is at line 20614. The next instruction in the QEMU log is `movw %ds, %ax`. The corresponding instruction in the `objdump` log is at line 267. Note that the instructions in the `objdump` log are not identical to the instructions shown in the QEMU log. The easiest

Figure 4.4: The client shows the modified response
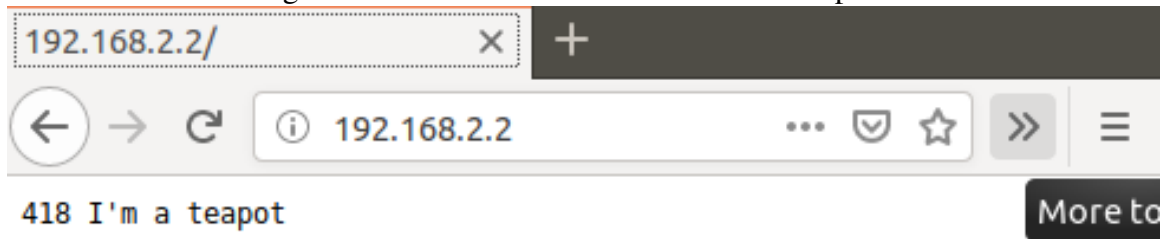
way to compare these logs is using the middle column which represents machine code. The line numbers of the `objdump` are incremented by the number of bytes in the machine code starting at line 200. On line 267 that shows the current instruction, skip the first byte which is 8c d8. From this point instruction alignment is straight forward as there are no `jmp` instructions that will skip a series of bytes.

4.4 QEMU Tracing

Another way to trace execution in QEMU is by a feature called QEMU Tracing. This function works at a slightly higher level than the lower level instruction level logging discussed previously. This method works with a trace-events file provided in the QEMU source doe. This is simply a list of all of the function headers available for tracing. Additional custom functions can be written. The second file involved is the custom events file

```
----------------
IN:
0xfffffff0:  ea 5b e0 00 f0                  ljmpw    $0xf000:$0xe05b


----------------
IN:
0x000fe05b:  2e 66 83 3e 48 5f 00            cmpl     $0, %cs:0x5f48
0x000fe062:  0f 85 66 f0                     jne      0xd0cc


----------------
IN:
0x000fe066:  31 d2                           xorw     %dx, %dx
0x000fe068:  8e d2                           movw     %dx, %ss


----------------
IN:
0x000fe06a:  66 bc 00 70 00 00               movl     $0x7000, %esp


----------------
IN:
0x000fe070:  66 ba 4d 19 0f 00               movl     $0xf194d, %edx
0x000fe076:  e9 d3 ee                        jmp      0xcf4c
```

Figure 4.5: OpenWrt boot instructions captured via QEMU with the in_asm option

that contains a matching function to trace. It works as a filter for the trace-events file but it only contains the name of the function [15].

This tracing methodology was an evolution of using other mechanisms such as GDB and LTTng that rely on system calls which may give unreliable information on a compromised system. It is more complex than the method above but it does provide a higher level logging which may be useful as data is moving through a network stack.

4.5   A Brief Exploration of Diamorphine

Diamorphine is an open source example of a kernel rootkit [29]. Diamorphine is a is a simple rootkit that hides processes in the system. The ps tool in Linux shows the currently running processes. Diamorphine can effectively remove process from that results

```
bzImage:        file format pei-x86-64


Disassembly of section .setup:

0000000000000200 <.setup>:
    200:        eb 66                   jmp     0x268
    202:        48                      rex.W
    203:        64 72 53                fs jb   0x259
    206:        0d 02 00 00 00          or      $0x2,%eax
    20b:        00 00                   add     %al,(%rax)
    20d:        10 40 30                adc     %al,0x30(%rax)
    210:        00 01                   add     %al,(%rcx)
    212:        00 80 00 00 10 00       add     %al,0x100000(%rax)
        ...
    224:        d0 51 00                rclb    0x0(%rcx)
    227:        00 00                   add     %al,(%rax)
    229:        00 00                   add     %al,(%rax)
    22b:        00 ff                   add     %bh,%bh
    22d:        ff                      (bad)
    22e:        ff                      (bad)
    22f:        7f 00                   jg      0x231
    231:        00 00                   add     %al,(%rax)
    233:        01 01                   add     %eax,(%rcx)
    235:        15 1b 00 ff 07          adc     $0x7ff001b,%eax
        ...
    246:        00 00                   add     %al,(%rax)
    248:        b4 03                   mov     $0x3,%ah
    24a:        00 00                   add     %al,(%rax)
    24c:        58                      pop     %rax
    24d:        88 3a                   mov     %bh,(%rdx)
        ...
    25b:        01 00                   add     %eax,(%rax)
    25d:        00 00                   add     %al,(%rax)
    25f:        00 00                   add     %al,(%rax)
    261:        20 5f 01                and     %bl,0x1(%rdi)
    264:        90                      nop
    265:        01 00                   add     %eax,(%rax)
    267:        00 8c d8 8e c0 fc 8c    add     %cl,-0x73033f72(%rax,%rbx,8)
    26e:        d2 39                   sarb    %cl,(%rcx)
    270:        c2 89 e2                retq    $0xe289
    273:        74 16                   je      0x28b
```

Figure 4.6: Objdump of bzImage

```
20612 ---------------
20613 IN:
20614 0x00010200:   eb 66                              jmp       0x268
20615
20616 ---------------
20617 IN:
20618 0x00010268:   8c d8                              movw      %ds, %ax
20619 0x0001026a:   8e c0                              movw      %ax, %es
20620 0x0001026c:   fc                                 cld
20621 0x0001026d:   8c d2                              movw      %ss, %dx
20622 0x0001026f:   39 c2                              cmpw      %ax, %dx
20623 0x00010271:   89 e2                              movw      %sp, %dx
20624 0x00010273:   74 16                              je        0x28b
20625
20626 ---------------
20627 IN:
20628 0x0001028b:   83 e2 fc                           andw      $0xfffc, %dx
20629 0x0001028e:   75 03                              jne       0x293
20630
```

Figure 4.7: QEMU log aligned with `objdump`

list. It works by sending a special signal numbered 31 to the process. Once that is done, the process is hidden from the `ps` results.

During this research project, we explored using Diamorphine and found it integrates readily with OpenWrt. As an example for discussion, the Diamorphine rootkit behavior should be detectable with QEMU tracing.

We predict, based on our existing observations that Diamorphine's behavior will appear in at least three places: 1) the request for `ps` results will demonstrate the hiding behavior, 2) the act of hiding the process with the `kill` command will reveal the behavior, and 3) the hidden process will execute additional instructions that may be detectable when compared to a system with a system without the rootkit.

Conclusions

5.1   Tracing HTTP Injection In OpenWrt

In this research we show how to perform instruction level tracing on an OpenWRT router running in QEMU. We built a framework for HTTP injection in OpenWrt. Through the steps identified in previous sections and outlined below, we show tracing HTTP injection in OpenWrt is possible though not trivial. We document the steps of building an instance of OpenWRT that includes the MITM Proxy software necessary to perform HTTP injection. We further discuss the possibility of using the Diamorphine rootkit within OpenWrt.

The first step to building an OpenWrt framework for HTTP injection is to identify and install Python prerequisites for MITM Proxy. The Python prerequisites are moderately involved and require multiple prerequisites themselves. Additionally, the Python cryptography library requires `gcc-ar`. To meet this requirement the `Makefile` for `gcc`.

Next, modifications to `gcc` source code are required due to code changes in more recent versions of `gcc`. The changes are made in the `md-unwind-support.h` file. Once these changes are made, OpenWrt compiles with the prerequisites necessary for MITM Proxy.

We traced the execution of OpenWrt in QEMU with the `in_asm` option. We were able to clearly identify instruction in the Linux kernel with the bzImage file produced by previous steps. We have demonstrated it is possible to trace instructions reliably in a com-

promised embedded system. We document system limitation and clear path forward for future research below.

5.2   Limitations

We believe the most challenging limitation to this system is that it's difficult to build. During the build process we found a number of build errors. Some were and easy to fix, while others took many days of effort to get past. What we learned through this experience is that the build process is fragile. It breaks easily with new versions of software and sometimes for reasons that were unclear to us. OpenWrt uses a standard Kconfig build system but there are many complex parts in OpenWrt and they can conflict with one another.

Another limitation to this system is that it's not easy to use. There is no trivial way to run the OpenWrt router and the clients. We wrote shell scripts included in the appendix to run this system. Once the system is running, there are still a number of steps to make MITM proxy operate. It relies on `ip-tables` along with creating filters within the proxy. None of these steps individually are difficult, but they need to be performed reliably and in order each time.

This system currently only works with OpenWrt firmware and requires a workstation with a significant amount of memory and storage for virtual machines. This could be reduced to more reasonable levels than what we used for this research. However, during experimentation with logging instructions in QEMU, we found writing the log file often stalled and the whole system moved slowly if at all. We suspect that a better logging mechanism to a remote machine may resolve this issue. Though it is not clear if this kind of mechanism is possible without significant modifications to QEMU source code.

The OpenWrt system we built provides a pragmatic approach to gathering instructions and behavioral data. We assumed that QEMU is a trusted piece of software. While we believe this assumption is reasonable, it is an assumption none the less. To verify this,

QEMU should be validated using formal methods.

5.3   Recommendations for Future Work

In our research we built a framework to analyze instruction level tracing of an embedded system using QEMU. We did not analyze all generated data as this is a significant undertaking in itself. We believe the first task in future work is to analyze the data output by the QEMU trace and identify the actions taken by MITM Proxy and/or Diamorphine. The existing research covering the exact behaviors of rootkits should be analyzed and distilled to a pattern that can be used in the framework that we have built. We believe creating a taxonomy of rootkit behavior and documenting that at the instruction level will prove useful in tracing and identifying those behaviors on suspect systems.

We believe there are several areas for interesting future work. The most significant area for future work is in integrating this system into another system such as Firmadyne. Firmadyne offers a number of framework pieces that could be leveraged to further analyze firmware. We believe that the Firmadyne can be modified uses an actual kernel to emulate the file system of firmware instead of emulating the file system as it does currently. One major technical hurdle with emulating the actual kernel is in device drivers. A possibility to overcome this is by building the hardware emulation into QEMU. That process is likely to be rather involved.

If Firmwadyne is successfully modified to emulate a kernel as described in this research, it could reveal malicious behavior in firmware provided by manufacturers. The existing Firmadyne framework focuses on finding vulnerabilities in the web interfaces and other network ingress/egress of device firmware. With the addition of kernel emulation this system could be used forensically to detect exploited firmware that may have rootkits or other malware installed.

The OpenWrt website provides a table of hardware that is supported. Further, the

details of the hardware (versions, various parameters, etc.) are available [30]. We believe using a variety of reverse engineering tools and techniques such as binwalk, objdump, dtc, etc. a more complete picture of device hardware can be built. QEMU is extensible and can emulate additional hardware. For example there is a QEMU PCI educational device in the QEMU source code. Creating a library of device drivers this way should allow for more complete emulation.

We believe there is a possibility of integrating hardware into QEMU by creating more general templates for each class of hardware. For example, wireless network cards are likely to share a number of similarities. These could be integrated into an extensible driver collection via a markup language like JSON or XML. The drivers could be read dynamically into QEMU allowing for a code base that doesn't need to be recompiled for each hardware addition. Thus a database of drivers can be generated readily.

We identified a discrepancy in the output of logged instructions between QEMU and the `objdump` of bzImage. We believe there is a good explanation for this, but we didn't investigate fully in this research. We relied on the hexadecimal machine code that was a perfect match. We also note that future work should include a better understanding of the boot sequence and associated boot instructions.

Lastly, we think refining the build process is a reasonable undertaking. Building MITM Proxy into OpenWrt with all of the prerequisites will allow for a more efficient build process. Kconfig allows for these modifications, but we did not refine the process.

Bibliography

[1] Ariel Berkman. Hiding Data in Hard-Drive's Service Areas. Technical report, Recover Information Technologies LTD, 2013.

[2] Christophe Bothamy. idle3-tools.

[3] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[4] Anton Chuvakin. An Overview of Unix Rootkits. 2003.

[5] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. mitmproxy - an interactive HTTPS proxy.

[6] Andrei Costin, Apostolis Zarras, and Aurlien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. *CoRR*, abs/1511.03609, 2015.

[7] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware supported rootkit concealment. In *Proceedings - IEEE Symposium on Security and Privacy*, 2008.

[8] J. Domburg. Sprites mods - Hard disk hacking - Intro.

[9] Jeremy Elson. Internet Content Adaptation Protocol (ICAP), 2003.

[10] Shawn Embleton, Sherri Sparks, and Cliff C. Zou. SMM rootkit: A new breed of OS independent malware. *Security and Communication Networks*, 2013.

[11] F-Secure. Virus:DOS/CIH Description — F-Secure Labs.

[12] Steve Gibson and Leo Laporte. Security Now! 521: Security is Difficult, 2015.

[13] M Giuliani. Mebromi: The First BIOS Rootkit in the Wild, 2011.

[14] GoldSparrow. WIN_JELLY GPU Malware Acts As Potentially Emerging Remote Access Tool or Trojan Threat.

[15] Stefan Hajnoczi. git.qemu.org Git - qemu.git/blob - docs/tracing.txt.

[16] John Heasman. Implementing and Detecting a PCI Rootkit. Technical report, Black Hat, 2007.

[17] John Heasman Black. Implementing and Detecting Implementing and Detecting an ACPI BIOS Rootkit an ACPI BIOS Rootkit. Technical report, Black Hat, 2006.

[18] Alan Holt and Chi-Yu Huang. *Embedded Operating Systems*. Springer-Verlag, London, 2014.

[19] Intel. A Guide to the Internet of Things Infographic.

[20] Intel. GitHub - chipsec/chipsec: Platform Security Assessment Framework, 2014.

[21] Intel Security. Advanced Threat Research — Intel Security, 2015.

[22] Igor Korkin and Ivan Nesterov. Applying Memory Forensics To Rootkit Detection. *Proceedings of the Conference on Digital Forensics, Security and Law*, pages 115–142, 2014.

[23] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *2013 European Workshop on Systems Security*, Prague, 2013.

[24] Lenovo. SuperFish Uninstall Instructions - us.

[25] Philippe Lin. Hacking Team Uses UEFI BIOS Rootkit to Keep RCS 9 Agent in Target Systems - TrendLabs Security Intelligence Blog, 2015.

[26] MalwareTech. MalwareTech SBK - A Bootkit Capable of Surviving Reformat - MalwareTech, 2015.

[27] Lee Mathews. Criminals Hacked A Fish Tank To Steal Data From A Casino, 2017.

[28] McAfee. McAfee Labs Threats Report. Technical report, McAfee, 2015.

[29] Victor Ramos Mello. GitHub - m0nad/Diamorphine: LKM rootkit for Linux Kernels 2.6.x/3.x/4.x (x86 and x86_64).

[30] OpenWrt. OpenWrt Project: Table of Hardware: Ideal for OpenWrt.

[31] J. Porter and A. Bryant. Detecting and mitigating rootkits in embedded systems. In *Proceedings of the 12th International Conference on Cyber Warfare and Security, ICCWS 2017*, 2017.

[32] Andreas Schwab. Release/2.26 - glibc wiki.

[33] Spam Laws. How Rootkits Work.

[34] squid-cache.org. squid : Optimising Web Delivery.

[35] Team Jellyfish. LucaBongiorni/jellyfish: GPU rootkit PoC by Team Jellyfish, 2014.

[36] The c-icap project. The c-icap project.

[37] Trusted Computing Group. Internet of Things.

[38] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. *International Journal of Information Security*, 2015.

[39] Teh Jia Yew, Khairulmizam Samsudin, Nur Izura Udzir, and Shaiful Jahari Hashim. Rootkit Guard (RG) - An architecture for rootkit resistant file-system implementation based on TPM. *Pertanika Journal of Science and Technology*, 21(2):507–520, 2013.

[40] Jonas Zaddach. Exploring the impact of a hard drive backdoor. Technical report, REcon, 2014.

[41] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurlien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference on - ACSAC '13*, 2013.

[42] Kim Zetter. How the NSA's Firmware Hacking Works and Why It's So Unsettling — WIRED, 2015.

gcc Makefile

```
1  #
2  # Copyright (C) 2008 OpenWrt.org
3  #
4  # This is free software, licensed under the GNU General Public License v2.
5  # See /LICENSE for more information.
6  #
7
8  include $(TOPDIR)/rules.mk
9
10 PKG_NAME:=gcc
11 PKG_VERSION:=5.4.0
12 PKG_RELEASE:=3
13 PKG_SOURCE_URL:=@GNU/gcc/gcc-$(PKG_VERSION)
14 PKG_HASH:=608df76dec2d34de6558249d8af4cbee21eceddbcb580d666f7a5a583ca3303a
15 PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
16 PKG_INSTALL:=1
17 PKG_FIXUP:=libtool
18 PKG_BUILD_PARALLEL:=1
19
20 include $(INCLUDE_DIR)/package.mk
21
22 TARGET_LANGUAGES:="c,c++"
23 BUGURL=https://dev.openwrt.org/
24 PKGVERSION=OpenWrt GCC $(PKG_VERSION)
25 TARGET_CPPFLAGS += -D_GLIBCXX_INCLUDE_NEXT_C_HEADERS
26
27 # not using sstrip here as this fucks up the .so's somehow
28 STRIP:=$(TOOLCHAIN_DIR)/bin/$(TARGET_CROSS)strip
29 RSTRIP:= \
30   NM="$(TOOLCHAIN_DIR)/bin/$(TARGET_CROSS)nm" \
31   STRIP="$(STRIP)" \
32   STRIP_KMOD="$(STRIP) --strip-debug" \
33   $(SCRIPT_DIR)/rstrip.sh
34
35 ifneq ($(CONFIG_SOFT_FLOAT),y)
36   ifeq ($(CONFIG_arm),y)
37     ARM_FLOAT_OPTION:= --with-float=hard
38   endif
39 endif
40
41 define Package/gcc
42   SECTION:=devel
43   CATEGORY:=Development
44   TITLE:=gcc
45   MAINTAINER:=Noble Pepper <gccmaintain@noblepepper.com>
46   DEPENDS:= +binutils +libstdcpp
47 endef
48
49 define Package/gcc/description
50   build a native toolchain for compiling on target
51 endef
52
53 GMPSRC=gmp-4.3.2
54
55 define Download/gmp
56   URL:=ftp://gcc.gnu.org/pub/gcc/infrastructure/
57   FILE:=$(GMPSRC).tar.bz2
58   HASH:=936162c0312886c21581002b79932829aa048cfaf9937c6265aeaa14f1cd1775
```

```
59 endef
60 $(eval $(call Download,gmp))
61
62 MPCSRC=mpc-0.8.1
63
64 define Download/mpc
65   URL:=ftp://gcc.gnu.org/pub/gcc/infrastructure/
66   FILE:=$(MPCSRC).tar.gz
67   HASH:=e664603757251fd8a352848276497a4c79b7f8b21fd8aedd5cc0598a38fee3e4
68 endef
69 $(eval $(call Download,mpc))
70
71 MPFRSRC=mpfr-2.4.2
72
73 define Download/mpfr
74   URL:=ftp://gcc.gnu.org/pub/gcc/infrastructure/
75   FILE:=$(MPFRSRC).tar.bz2
76   HASH:=c7e75a08a8d49d2082e4caee1591a05d11b9d5627514e678f02d66a124bcf2ba
77 endef
78 $(eval $(call Download,mpfr))
79
80 define Build/Prepare
81   $(PKG_UNPACK)
82 #   we have to download and unpack additional stuff before patching
83   tar -C $(PKG_BUILD_DIR) -xvjf $(DL_DIR)/$(GMPSRC).tar.bz2
84   ln -sf $(PKG_BUILD_DIR)/$(GMPSRC) $(PKG_BUILD_DIR)/gmp
85   tar -C $(PKG_BUILD_DIR) -xvzf $(DL_DIR)/$(MPCSRC).tar.gz
86   ln -sf $(PKG_BUILD_DIR)/$(MPCSRC) $(PKG_BUILD_DIR)/mpc
87   tar -C $(PKG_BUILD_DIR) -xvjf $(DL_DIR)/$(MPFRSRC).tar.bz2
88   ln -sf $(PKG_BUILD_DIR)/$(MPFRSRC) $(PKG_BUILD_DIR)/mpfr
89   $(Build/Patch)
90 # poor man's fix for 'none-openwrt-linux' not recognized when building with musl
91   cp $(PKG_BUILD_DIR)/config.sub $(PKG_BUILD_DIR)/mpfr/
92   cp $(PKG_BUILD_DIR)/config.sub $(PKG_BUILD_DIR)/gmp/
93   cp $(PKG_BUILD_DIR)/config.sub $(PKG_BUILD_DIR)/mpc/
94 endef
95
96 TARGET_CXX += -std=gnu++03
97 CONFIGURE_ARGS += CXX_FOR_TARGET="$(TARGET_CXX)" CXXFLAGS_FOR_TARGET="-g -O2 -
     D_GLIBCXX_INCLUDE_NEXT_C_HEADERS"
98
99 define Build/Configure
100   (cd $(PKG_BUILD_DIR); rm -f config.cache; \
101     SHELL="$(BASH)" \
102    $(TARGET_CONFIGURE_OPTS) \
103    $(PKG_BUILD_DIR)/configure \
104      $(CONFIGURE_ARGS) \
105      --build=$(GNU_HOST_NAME) \
106      --host=$(REAL_GNU_TARGET_NAME) \
107      --target=$(REAL_GNU_TARGET_NAME) \
108      --enable-languages=$(TARGET_LANGUAGES) \
109      --with-bugurl=$(BUGURL) \
110      --with-pkgversion="$(PKGVERSION)" \
111      --enable-shared \
112      $(if $(CONFIG_LIBC_USE_GLIBC),--enable,--disable)-__cxa_atexit \
113      --with-default-libstdcxx-abi=gcc4-compatible \
114      --enable-target-optspace \
115      --with-gnu-ld \
116      --disable-nls \
117      --disable-libsanitizer \
118      --disable-libvtv \
119      --disable-libcilkrts \
120      --disable-libmudflap \
121      --disable-multilib \
122      --disable-libgomp \
123      --disable-libquadmath \
124      --disable-libssp \
125      --disable-decimal-float \
126      --disable-libstdcxx-pch \
127      --with-host-libstdcxx=-lstdc++ \
128      --prefix=/usr \
129      --libexecdir=/usr/lib \
130      --with-local-prefix=/usr \
```

```
131        $(ARM_FLOAT_OPTION) \
132        $(SOFT_FLOAT_CONFIG_OPTION) \
133        $(call qstrip,$(CONFIG_EXTRA_GCC_CONFIG_OPTIONS)) \
134   );
135 endef
136
137 define Build/Compile
138   export SHELL="$(BASH)"; $(MAKE_VARS) $(MAKE) -C $(PKG_BUILD_DIR) \
139       DESTDIR="$(PKG_INSTALL_DIR)" $(MAKE_ARGS) all install
140 endef
141
142 ENVCFLAGS:="$(TARGET_OPTIMIZATION) $(EXTRA_OPTIMIZATION)
143 ifeq ($(CONFIG_SOFT_FLOAT),y)
144     ifeq ($(CONFIG_arm),y)
145   ENVCFLAGS+= -mfloat-abi=soft
146     else
147   ENVCFLAGS+= -msoft-float
148     endif
149 endif
150 ENVCFLAGS+="
151
152 ENVLDFLAGS:="-Wl,-rpath=/usr/lib -Wl,--dynamic-linker=/usr/lib/$(DYNLINKER) -L/
      usr/lib"
153
154 define Package/gcc/install
155   $(INSTALL_DIR) $(1)/usr/bin $(1)/usr/lib $(1)/usr/lib/$(PKG_NAME)/$(
          REAL_GNU_TARGET_NAME)/$(PKG_VERSION)
156   cp -ar $(PKG_INSTALL_DIR)/usr/include $(1)/usr
157   cp -a $(PKG_INSTALL_DIR)/usr/bin/{$(REAL_GNU_TARGET_NAME)-{g++,gcc,gcc-ar},cpp
          ,gcov} $(1)/usr/bin
158   ln -s $(REAL_GNU_TARGET_NAME)-g++ $(1)/usr/bin/c++
159   ln -s $(REAL_GNU_TARGET_NAME)-g++ $(1)/usr/bin/g++
160   ln -s $(REAL_GNU_TARGET_NAME)-g++ $(1)/usr/bin/$(REAL_GNU_TARGET_NAME)-c++
161   ln -s $(REAL_GNU_TARGET_NAME)-gcc $(1)/usr/bin/gcc
162   ln -s $(REAL_GNU_TARGET_NAME)-gcc $(1)/usr/bin/$(REAL_GNU_TARGET_NAME)-gcc-$(
          PKG_VERSION)
163   ln -s $(REAL_GNU_TARGET_NAME)-gcc-ar $(1)/usr/bin/gcc-ar
164   cp -ar $(PKG_INSTALL_DIR)/usr/lib/gcc $(1)/usr/lib
165   cp -ar $(TOOLCHAIN_DIR)/include $(1)/usr
166   cp -a $(TOOLCHAIN_DIR)/lib/*.{o,so*} $(1)/usr/lib/$(PKG_NAME)/$(
          REAL_GNU_TARGET_NAME)/$(PKG_VERSION)
167   cp -a $(TOOLCHAIN_DIR)/lib/*nonshared*.a  $(1)/usr/lib/$(PKG_NAME)/$(
          REAL_GNU_TARGET_NAME)/$(PKG_VERSION)
168   cp -a $(TOOLCHAIN_DIR)/lib/libm.a  $(1)/usr/lib/$(PKG_NAME)/$(
          REAL_GNU_TARGET_NAME)/$(PKG_VERSION)
169   rm -f $(1)/usr/lib/$(PKG_NAME)/$(REAL_GNU_TARGET_NAME)/$(PKG_VERSION)/libgo*
170   rm -f $(1)/usr/lib/$(PKG_NAME)/$(REAL_GNU_TARGET_NAME)/$(PKG_VERSION)/libcc1*
171   echo '#!/bin/sh' > $(1)/usr/bin/gcc_env.sh
172   echo 'export LDFLAGS=$(ENVLDFLAGS)' >> $(1)/usr/bin/gcc_env.sh
173   echo 'export CFLAGS=$(ENVCFLAGS)' >> $(1)/usr/bin/gcc_env.sh
174   chmod +x $(1)/usr/bin/gcc_env.sh
175 endef
176
177 $(eval $(call BuildPackage,gcc))
```

md-unwind-support.h

```
1  /* DWARF2 EH unwinding support for AMD x86-64 and x86.
2     Copyright (C) 2004-2015 Free Software Foundation, Inc.
3
4  This file is part of GCC.
5
6  GCC is free software; you can redistribute it and/or modify
7  it under the terms of the GNU General Public License as published by
8  the Free Software Foundation; either version 3, or (at your option)
9  any later version.
10
11  GCC is distributed in the hope that it will be useful,
12  but WITHOUT ANY WARRANTY; without even the implied warranty of
13  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  GNU General Public License for more details.
15
16  Under Section 7 of GPL version 3, you are granted additional
17  permissions described in the GCC Runtime Library Exception, version
18  3.1, as published by the Free Software Foundation.
19
20  You should have received a copy of the GNU General Public License and
21  a copy of the GCC Runtime Library Exception along with this program;
22  see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see
23  <http://www.gnu.org/licenses/>.  */
24
25  /* Do code reading to identify a signal frame, and set the frame
26     state data appropriately.  See unwind-dw2.c for the structs.
27     Don't use this at all if inhibit_libc is used.  */
28
29  #ifndef inhibit_libc
30
31  /* There's no sys/ucontext.h for glibc 2.0, so no
32     signal-turned-exceptions for them.  There's also no configure-run for
33     the target, so we can't check on (e.g.) HAVE_SYS_UCONTEXT_H.  Using the
34     target libc version macro should be enough.  */
35  #if defined __GLIBC__ && !(__GLIBC__ == 2 && __GLIBC_MINOR__ == 0)
36
37  #include <signal.h>
38  #include <sys/ucontext.h>
39
40  #ifdef __x86_64__
41
42  #define MD_FALLBACK_FRAME_STATE_FOR x86_64_fallback_frame_state
43
44  static _Unwind_Reason_Code
45  x86_64_fallback_frame_state (struct _Unwind_Context *context,
46                               _Unwind_FrameState *fs)
47  {
48    unsigned char *pc = context->ra;
49    struct sigcontext *sc;
50    long new_cfa;
51
52    /* movq $__NR_rt_sigreturn, %rax ; syscall.  */
53  #ifdef __LP64__
54  #define RT_SIGRETURN_SYSCALL   0x050f0000000fc0c7ULL
55  #else
56  #define RT_SIGRETURN_SYSCALL   0x050f40000201c0c7ULL
57  #endif
58    if (*(unsigned char *)(pc+0) == 0x48
```

52

```
 59          && *(unsigned long long *)(pc+1) == RT_SIGRETURN_SYSCALL)
 60      {
 61        //struct ucontext *uc_ = context->cfa;
 62        ucontext_t *uc_ = context->cfa;
 63        /* The void * cast is necessary to avoid an aliasing warning.
 64            The aliasing warning is correct, but should not be a problem
 65            because it does not alias anything.  */
 66        sc = (struct sigcontext *) (void *) &uc_->uc_mcontext;
 67      }
 68    else
 69      return _URC_END_OF_STACK;
 70
 71    new_cfa = sc->rsp;
 72    fs->regs.cfa_how = CFA_REG_OFFSET;
 73    /* Register 7 is rsp  */
 74    fs->regs.cfa_reg = 7;
 75    fs->regs.cfa_offset = new_cfa - (long) context->cfa;
 76
 77    /* The SVR4 register numbering macros aren't usable in libgcc.  */
 78    fs->regs.reg[0].how = REG_SAVED_OFFSET;
 79    fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;
 80    fs->regs.reg[1].how = REG_SAVED_OFFSET;
 81    fs->regs.reg[1].loc.offset = (long)&sc->rdx - new_cfa;
 82    fs->regs.reg[2].how = REG_SAVED_OFFSET;
 83    fs->regs.reg[2].loc.offset = (long)&sc->rcx - new_cfa;
 84    fs->regs.reg[3].how = REG_SAVED_OFFSET;
 85    fs->regs.reg[3].loc.offset = (long)&sc->rbx - new_cfa;
 86    fs->regs.reg[4].how = REG_SAVED_OFFSET;
 87    fs->regs.reg[4].loc.offset = (long)&sc->rsi - new_cfa;
 88    fs->regs.reg[5].how = REG_SAVED_OFFSET;
 89    fs->regs.reg[5].loc.offset = (long)&sc->rdi - new_cfa;
 90    fs->regs.reg[6].how = REG_SAVED_OFFSET;
 91    fs->regs.reg[6].loc.offset = (long)&sc->rbp - new_cfa;
 92    fs->regs.reg[8].how = REG_SAVED_OFFSET;
 93    fs->regs.reg[8].loc.offset = (long)&sc->r8 - new_cfa;
 94    fs->regs.reg[9].how = REG_SAVED_OFFSET;
 95    fs->regs.reg[9].loc.offset = (long)&sc->r9 - new_cfa;
 96    fs->regs.reg[10].how = REG_SAVED_OFFSET;
 97    fs->regs.reg[10].loc.offset = (long)&sc->r10 - new_cfa;
 98    fs->regs.reg[11].how = REG_SAVED_OFFSET;
 99    fs->regs.reg[11].loc.offset = (long)&sc->r11 - new_cfa;
100    fs->regs.reg[12].how = REG_SAVED_OFFSET;
101    fs->regs.reg[12].loc.offset = (long)&sc->r12 - new_cfa;
102    fs->regs.reg[13].how = REG_SAVED_OFFSET;
103    fs->regs.reg[13].loc.offset = (long)&sc->r13 - new_cfa;
104    fs->regs.reg[14].how = REG_SAVED_OFFSET;
105    fs->regs.reg[14].loc.offset = (long)&sc->r14 - new_cfa;
106    fs->regs.reg[15].how = REG_SAVED_OFFSET;
107    fs->regs.reg[15].loc.offset = (long)&sc->r15 - new_cfa;
108    fs->regs.reg[16].how = REG_SAVED_OFFSET;
109    fs->regs.reg[16].loc.offset = (long)&sc->rip - new_cfa;
110    fs->retaddr_column = 16;
111    fs->signal_frame = 1;
112    return _URC_NO_REASON;
113  }
114
115  #else /* ifdef __x86_64__  */
116
117  #define MD_FALLBACK_FRAME_STATE_FOR x86_fallback_frame_state
118
119  static _Unwind_Reason_Code
120  x86_fallback_frame_state (struct _Unwind_Context *context,
121            _Unwind_FrameState *fs)
122  {
123    unsigned char *pc = context->ra;
124    struct sigcontext *sc;
125    long new_cfa;
126
127    /* popl %eax ; movl $__NR_sigreturn,%eax ; int $0x80  */
128    if (*(unsigned short *)(pc+0) == 0xb858
129        && *(unsigned int *)(pc+2) == 119
130        && *(unsigned short *)(pc+6) == 0x80cd)
131      sc = context->cfa + 4;
```

```
132    /* movl $__NR_rt_sigreturn,%eax ; int $0x80  */
133    else if (*(unsigned char *)(pc+0) == 0xb8
134        && *(unsigned int *)(pc+1) == 173
135        && *(unsigned short *)(pc+5) == 0x80cd)
136      {
137        struct rt_sigframe {
138    int sig;
139    siginfo_t *pinfo;
140    void *puc;
141    siginfo_t info;
142    //struct ucontext uc;
143    ucontext_t uc;
144        } *rt_ = context->cfa;
145        /* The void * cast is necessary to avoid an aliasing warning.
146           The aliasing warning is correct, but should not be a problem
147           because it does not alias anything.  */
148        sc = (struct sigcontext *) (void *) &rt_->uc.uc_mcontext;
149      }
150    else
151      return _URC_END_OF_STACK;
152
153    new_cfa = sc->esp;
154    fs->regs.cfa_how = CFA_REG_OFFSET;
155    fs->regs.cfa_reg = 4;
156    fs->regs.cfa_offset = new_cfa - (long) context->cfa;
157
158    /* The SVR4 register numbering macros aren't usable in libgcc.  */
159    fs->regs.reg[0].how = REG_SAVED_OFFSET;
160    fs->regs.reg[0].loc.offset = (long)&sc->eax - new_cfa;
161    fs->regs.reg[3].how = REG_SAVED_OFFSET;
162    fs->regs.reg[3].loc.offset = (long)&sc->ebx - new_cfa;
163    fs->regs.reg[1].how = REG_SAVED_OFFSET;
164    fs->regs.reg[1].loc.offset = (long)&sc->ecx - new_cfa;
165    fs->regs.reg[2].how = REG_SAVED_OFFSET;
166    fs->regs.reg[2].loc.offset = (long)&sc->edx - new_cfa;
167    fs->regs.reg[6].how = REG_SAVED_OFFSET;
168    fs->regs.reg[6].loc.offset = (long)&sc->esi - new_cfa;
169    fs->regs.reg[7].how = REG_SAVED_OFFSET;
170    fs->regs.reg[7].loc.offset = (long)&sc->edi - new_cfa;
171    fs->regs.reg[5].how = REG_SAVED_OFFSET;
172    fs->regs.reg[5].loc.offset = (long)&sc->ebp - new_cfa;
173    fs->regs.reg[8].how = REG_SAVED_OFFSET;
174    fs->regs.reg[8].loc.offset = (long)&sc->eip - new_cfa;
175    fs->retaddr_column = 8;
176    fs->signal_frame = 1;
177    return _URC_NO_REASON;
178  }
179
180  #define MD_FROB_UPDATE_CONTEXT x86_frob_update_context
181
182  /* Fix up for kernels that have vDSO, but don't have S flag in it.  */
183
184  static void
185  x86_frob_update_context (struct _Unwind_Context *context,
186        _Unwind_FrameState *fs ATTRIBUTE_UNUSED)
187  {
188    unsigned char *pc = context->ra;
189
190    /* movl $__NR_rt_sigreturn,%eax ; {int $0x80 | syscall}  */
191    if (*(unsigned char *)(pc+0) == 0xb8
192        && *(unsigned int *)(pc+1) == 173
193        && (*(unsigned short *)(pc+5) == 0x80cd
194      || *(unsigned short *)(pc+5) == 0x050f))
195      _Unwind_SetSignalFrame (context, 1);
196  }
197
198  #endif /* ifdef __x86_64__  */
199  #endif /* not glibc 2.0 */
200  #endif /* ifdef inhibit_libc  */
```

MITM Proxy Pre-requisites

```
user@user-KVM:~$ johnnydep mitmproxy
2019-06-16 17:00:56 [info      ] init johnnydist                [johnnydep.lib] dist=mitmproxy parent=None
2019-06-16 17:01:05 [info      ] init johnnydist                [johnnydep.lib] dist=blinker<1.5,>=1.4 parent=mitmproxy
2019-06-16 17:01:08 [info      ] init johnnydist                [johnnydep.lib] dist=brotlipy<0.8,>=0.7.0 parent=mitmproxy
2019-06-16 17:01:11 [info      ] init johnnydist                [johnnydep.lib] dist=certifi>=2015.11.20.1 parent=mitmproxy
2019-06-16 17:01:13 [info      ] init johnnydist                [johnnydep.lib] dist=click<7,>=6.2 parent=mitmproxy
2019-06-16 17:01:15 [info      ] init johnnydist                [johnnydep.lib] dist=cryptography<2.4,>=2.1.4 parent=mitmproxy
2019-06-16 17:01:18 [info      ] init johnnydist                [johnnydep.lib] dist=h2<4,>=3.0.1 parent=mitmproxy
2019-06-16 17:01:21 [info      ] init johnnydist                [johnnydep.lib] dist=hyperframe<6,>=5.1.0 parent=mitmproxy
2019-06-16 17:01:23 [info      ] init johnnydist                [johnnydep.lib] dist=kaitaistruct<0.9,>=0.7 parent=mitmproxy
2019-06-16 17:01:26 [info      ] init johnnydist                [johnnydep.lib] dist=ldap3<2.6,>=2.5 parent=mitmproxy
2019-06-16 17:01:29 [info      ] init johnnydist                [johnnydep.lib] dist=passlib<1.8,>=1.6.5 parent=mitmproxy
2019-06-16 17:01:31 [info      ] init johnnydist                [johnnydep.lib] dist=pyOpenSSL<18.1,>=17.5 parent=mitmproxy
2019-06-16 17:01:33 [info      ] init johnnydist                [johnnydep.lib] dist=pyasn1<0.5,>=0.3.1 parent=mitmproxy
2019-06-16 17:01:42 [info      ] init johnnydist                [johnnydep.lib] dist=pyparsing<2.3,>=2.1.3 parent=mitmproxy
2019-06-16 17:01:44 [info      ] init johnnydist                [johnnydep.lib] dist=pyperclip<1.7,>=1.6.0 parent=mitmproxy
2019-06-16 17:01:48 [info      ] init johnnydist                [johnnydep.lib] dist=ruamel.yaml<0.16,>=0.13.2 parent=mitmproxy
2019-06-16 17:01:51 [info      ] init johnnydist                [johnnydep.lib] dist=sortedcontainers<2.1,>=1.5.4 parent=mitmproxy
2019-06-16 17:01:53 [info      ] init johnnydist                [johnnydep.lib] dist=tornado<5.2,>=4.3 parent=mitmproxy
2019-06-16 17:01:58 [info      ] init johnnydist                [johnnydep.lib] dist=urwid<2.1,>=2.0.1 parent=mitmproxy
2019-06-16 17:02:02 [info      ] init johnnydist                [johnnydep.lib] dist=wsproto<0.12.0,>=0.11.0 parent=mitmproxy
2019-06-16 17:02:06 [info      ] init johnnydist                [johnnydep.lib] dist=cffi>=1.0.0 parent=brotlipy<0.8,>=0.7.0
2019-06-16 17:02:09 [info      ] init johnnydist                [johnnydep.lib] dist=pycparser parent=cffi>=1.0.0
2019-06-16 17:02:13 [info      ] init johnnydist                [johnnydep.lib] dist=asn1crypto>=0.21.0 parent=cryptography<2.4,>=2.1.4
2019-06-16 17:02:15 [info      ] init johnnydist                [johnnydep.lib] dist=cffi!=1.11.3,>=1.7 parent=cryptography<2.4,>=2.1.4
2019-06-16 17:02:18 [info      ] init johnnydist                [johnnydep.lib] dist=idna>=2.1 parent=cryptography<2.4,>=2.1.4
2019-06-16 17:02:21 [info      ] init johnnydist                [johnnydep.lib] dist=six>=1.4.1 parent=cryptography<2.4,>=2.1.4
2019-06-16 17:02:23 [info      ] init johnnydist                [johnnydep.lib] dist=pycparser parent=cffi!=1.11.3,>=1.7
2019-06-16 17:02:23 [info      ] init johnnydist                [johnnydep.lib] dist=hpack<4,>=2.3 parent=h2<4,>=3.0.1
2019-06-16 17:02:25 [info      ] init johnnydist                [johnnydep.lib] dist=hyperframe<6,>=5.2.0 parent=h2<4,>=3.0.1
2019-06-16 17:02:27 [info      ] init johnnydist                [johnnydep.lib] dist=pyasn1>=0.1.8 parent=ldap3<2.6,>=2.5
2019-06-16 17:02:30 [info      ] init johnnydist                [johnnydep.lib] dist=cryptography>=2.2.1 parent=pyOpenSSL<18.1,>=17.5
2019-06-16 17:02:33 [info      ] init johnnydist                [johnnydep.lib] dist=six>=1.5.2 parent=pyOpenSSL<18.1,>=17.5
2019-06-16 17:02:35 [info      ] init johnnydist                [johnnydep.lib] dist=asn1crypto>=0.21.0 parent=cryptography>=2.2.1
2019-06-16 17:02:35 [info      ] init johnnydist                [johnnydep.lib] dist=cffi!=1.11.3,>=1.8 parent=cryptography>=2.2.1
2019-06-16 17:02:38 [info      ] init johnnydist                [johnnydep.lib] dist=six>=1.4.1 parent=cryptography>=2.2.1
2019-06-16 17:02:38 [info      ] init johnnydist                [johnnydep.lib] dist=pycparser parent=cffi!=1.11.3,>=1.8
2019-06-16 17:02:38 [info      ] init johnnydist                [johnnydep.lib] dist=h11~=0.7.0 parent=wsproto<0.12.0,>=0.11.0
```

Figure C.1: Dependencies listed from johnnydep Python tool

OpenWrt Config

The `.config` file is shown below. The unset symbols are omitted.

```
 1 #
 2 # Automatically generated file; DO NOT EDIT.
 3 # OpenWrt Configuration
 4 #
 5 CONFIG_MODULES=y
 6 CONFIG_HAVE_DOT_CONFIG=y
 7 CONFIG_TARGET_x86=y
 8 CONFIG_TARGET_x86_64=y
 9 CONFIG_TARGET_x86_64_Generic=y
10 CONFIG_HAS_SUBTARGETS=y
11 CONFIG_TARGET_BOARD="x86"
12 CONFIG_TARGET_SUBTARGET="64"
13 CONFIG_TARGET_PROFILE="Generic"
14 CONFIG_TARGET_ARCH_PACKAGES="x86_64"
15 CONFIG_DEFAULT_TARGET_OPTIMIZATION="-Os -pipe"
16 CONFIG_CPU_TYPE=" "
17 CONFIG_LINUX_4_14=y
18 CONFIG_DEFAULT_base-files=y
19 CONFIG_DEFAULT_busybox=y
20 CONFIG_DEFAULT_dnsmasq=y
21 CONFIG_DEFAULT_dropbear=y
22 CONFIG_DEFAULT_e2fsprogs=y
23 CONFIG_DEFAULT_firewall=y
24 CONFIG_DEFAULT_fstools=y
25 CONFIG_DEFAULT_ip6tables=y
26 CONFIG_DEFAULT_iptables=y
27 CONFIG_DEFAULT_kmod-bnx2=y
28 CONFIG_DEFAULT_kmod-button-hotplug=y
29 CONFIG_DEFAULT_kmod-e1000=y
30 CONFIG_DEFAULT_kmod-e1000e=y
31 CONFIG_DEFAULT_kmod-igb=y
32 CONFIG_DEFAULT_kmod-ipt-offload=y
33 CONFIG_DEFAULT_kmod-r8169=y
34 CONFIG_DEFAULT_libc=y
35 CONFIG_DEFAULT_libgcc=y
36 CONFIG_DEFAULT_logd=y
37 CONFIG_DEFAULT_mkf2fs=y
38 CONFIG_DEFAULT_mtd=y
39 CONFIG_DEFAULT_netifd=y
40 CONFIG_DEFAULT_odhcp6c=y
41 CONFIG_DEFAULT_odhcpd-ipv6only=y
42 CONFIG_DEFAULT_opkg=y
43 CONFIG_DEFAULT_partx-utils=y
44 CONFIG_DEFAULT_ppp=y
45 CONFIG_DEFAULT_ppp-mod-pppoe=y
46 CONFIG_DEFAULT_uci=y
47 CONFIG_DEFAULT_uclient-fetch=y
48 CONFIG_HAS_FPU=y
49 CONFIG_AUDIO_SUPPORT=y
50 CONFIG_GPIO_SUPPORT=y
51 CONFIG_PCI_SUPPORT=y
52 CONFIG_PCIE_SUPPORT=y
53 CONFIG_PCMCIA_SUPPORT=y
54 CONFIG_USB_SUPPORT=y
55 CONFIG_RTC_SUPPORT=y
```

```
56 CONFIG_USES_SQUASHFS=y
57 CONFIG_USES_EXT4=y
58 CONFIG_USES_TARGZ=y
59 CONFIG_ARCH_64BIT=y
60 CONFIG_VIRTIO_SUPPORT=y
61 CONFIG_x86_64=y
62 CONFIG_ARCH="x86_64"
63 #
64 # Target Images
65 #
66 CONFIG_EXTERNAL_CPIO=""
67 #
68 # Root filesystem archives
69 #
70 CONFIG_TARGET_ROOTFS_TARGZ=y
71 #
72 # Root filesystem images
73 #
74 CONFIG_TARGET_ROOTFS_EXT4FS=y
75 CONFIG_TARGET_EXT4_RESERVED_PCT=0
76 CONFIG_TARGET_EXT4_BLOCKSIZE_4K=y
77 CONFIG_TARGET_EXT4_BLOCKSIZE=4096
78 CONFIG_TARGET_ROOTFS_SQUASHFS=y
79 CONFIG_TARGET_SQUASHFS_BLOCK_SIZE=256
80 CONFIG_TARGET_UBIFS_FREE_SPACE_FIXUP=y
81 CONFIG_TARGET_UBIFS_JOURNAL_SIZE=""
82 CONFIG_GRUB_IMAGES=y
83 CONFIG_GRUB_CONSOLE=y
84 CONFIG_GRUB_SERIAL="ttyS0"
85 CONFIG_GRUB_BAUDRATE=115200
86 CONFIG_GRUB_BOOTOPTS=""
87 CONFIG_GRUB_TIMEOUT="5"
88 CONFIG_GRUB_TITLE="OpenWrt"
89 CONFIG_TARGET_IMAGES_GZIP=y
90 #
91 # Image Options
92 #
93 CONFIG_TARGET_KERNEL_PARTSIZE=16
94 CONFIG_TARGET_ROOTFS_PARTSIZE=4096
95 CONFIG_TARGET_ROOTFS_PARTNAME=""
96 #
97 # Global build settings
98 #
99 CONFIG_SIGNED_PACKAGES=y
100 #
101 # General build options
102 #
103 CONFIG_DISPLAY_SUPPORT=y
104 CONFIG_SHADOW_PASSWORDS=y
105 #
106 # Kernel build options
107 #
108 CONFIG_KERNEL_BUILD_USER=""
109 CONFIG_KERNEL_BUILD_DOMAIN=""
110 CONFIG_KERNEL_PRINTK=y
111 CONFIG_KERNEL_SWAP=y
112 CONFIG_KERNEL_DEBUG_FS=y
113 CONFIG_KERNEL_KALLSYMS=y
114 CONFIG_KERNEL_DEBUG_KERNEL=y
115 CONFIG_KERNEL_DEBUG_INFO=y
116 CONFIG_KERNEL_MAGIC_SYSRQ=y
117 CONFIG_KERNEL_COREDUMP=y
118 CONFIG_KERNEL_ELF_CORE=y
119 CONFIG_KERNEL_PRINTK_TIME=y
120 CONFIG_KERNEL_KEXEC=y
121 CONFIG_KERNEL_PROC_VMCORE=y
122 CONFIG_KERNEL_CRASH_DUMP=y
123 CONFIG_KERNEL_IP_MROUTE=y
124 CONFIG_KERNEL_IPV6=y
125 CONFIG_KERNEL_IPV6_MULTIPLE_TABLES=y
126 CONFIG_KERNEL_IPV6_SUBTREES=y
127 CONFIG_KERNEL_IPV6_MROUTE=y
128 #
```

```
129 # Filesystem ACL and attr support options
130 #
131 CONFIG_KERNEL_SQUASHFS_FRAGMENT_CACHE_SIZE=3
132 CONFIG_KERNEL_CC_OPTIMIZE_FOR_PERFORMANCE=y
133 #
134 # Package build options
135 #
136 CONFIG_IPV6=y
137 #
138 # Stripping options
139 #
140 CONFIG_USE_STRIP=y
141 CONFIG_STRIP_ARGS="--strip-all"
142 CONFIG_USE_LIBSTDCXX=y
143 #
144 # Hardening build options
145 #
146 CONFIG_PKG_CHECK_FORMAT_SECURITY=y
147 CONFIG_PKG_FORTIFY_SOURCE_1=y
148 CONFIG_PKG_RELRO_FULL=y
149 CONFIG_DEVEL=y
150 CONFIG_BINARY_FOLDER=""
151 CONFIG_DOWNLOAD_FOLDER=""
152 CONFIG_LOCALMIRROR=""
153 CONFIG_AUTOREBUILD=y
154 CONFIG_BUILD_SUFFIX=""
155 CONFIG_TARGET_ROOTFS_DIR=""
156 CONFIG_EXTERNAL_KERNEL_TREE=""
157 CONFIG_KERNEL_GIT_CLONE_URI=""
158 CONFIG_EXTRA_OPTIMIZATION="-fno-caller-saves -fno-plt"
159 CONFIG_TARGET_OPTIMIZATION="-Os -pipe"
160 CONFIG_NEED_TOOLCHAIN=y
161 CONFIG_TOOLCHAINOPTS=y
162 #
163 # Binary tools
164 #
165 CONFIG_BINUTILS_USE_VERSION_2_31_1=y
166 CONFIG_EXTRA_BINUTILS_CONFIG_OPTIONS=""
167 #
168 # Compiler
169 #
170 CONFIG_GCC_USE_VERSION_7=y
171 CONFIG_EXTRA_GCC_CONFIG_OPTIONS=""
172 CONFIG_GCC_LIBSSP=y
173 CONFIG_NASM=y
174 #
175 # C Library
176 #
177 CONFIG_LIBC_USE_GLIBC=y
178 #
179 # Debuggers
180 #
181 CONFIG_GDB=y
182 CONFIG_USE_GLIBC=y
183 CONFIG_SSP_SUPPORT=y
184 CONFIG_BINUTILS_VERSION_2_31_1=y
185 CONFIG_BINUTILS_VERSION="2.31.1"
186 CONFIG_GCC_VERSION="7.4.0"
187 CONFIG_LIBC="glibc"
188 CONFIG_TARGET_SUFFIX="gnu"
189 CONFIG_TARGET_PREINIT_SUPPRESS_STDERR=y
190 CONFIG_TARGET_PREINIT_TIMEOUT=2
191 CONFIG_TARGET_PREINIT_IFNAME=""
192 CONFIG_TARGET_PREINIT_IP="192.168.1.1"
193 CONFIG_TARGET_PREINIT_NETMASK="255.255.255.0"
194 CONFIG_TARGET_PREINIT_BROADCAST="192.168.1.255"
195 CONFIG_TARGET_INIT_PATH="/usr/sbin:/usr/bin:/sbin:/bin"
196 CONFIG_TARGET_INIT_ENV=""
197 CONFIG_TARGET_INIT_CMD="/sbin/init"
198 CONFIG_TARGET_INIT_SUPPRESS_STDERR=y
199 CONFIG_PER_FEED_REPO=y
200 CONFIG_FEED_packages=y
201 CONFIG_FEED_luci=y
```

```
202 CONFIG_FEED_routing=y
203 CONFIG_FEED_telephony=y
204 #
205 # Base system
206 #
207 CONFIG_PACKAGE_base-files=y
208 CONFIG_PACKAGE_busybox=y
209 CONFIG_BUSYBOX_DEFAULT_HAVE_DOT_CONFIG=y
210 CONFIG_BUSYBOX_DEFAULT_INCLUDE_SUSv2=y
211 CONFIG_BUSYBOX_DEFAULT_LONG_OPTS=y
212 CONFIG_BUSYBOX_DEFAULT_SHOW_USAGE=y
213 CONFIG_BUSYBOX_DEFAULT_FEATURE_VERBOSE_USAGE=y
214 CONFIG_BUSYBOX_DEFAULT_FEATURE_COMPRESS_USAGE=y
215 CONFIG_BUSYBOX_DEFAULT_LFS=y
216 CONFIG_BUSYBOX_DEFAULT_FEATURE_DEVPTS=y
217 CONFIG_BUSYBOX_DEFAULT_FEATURE_PIDFILE=y
218 CONFIG_BUSYBOX_DEFAULT_PID_FILE_PATH="/var/run"
219 CONFIG_BUSYBOX_DEFAULT_FEATURE_SUID=y
220 CONFIG_BUSYBOX_DEFAULT_FEATURE_PREFER_APPLETS=y
221 CONFIG_BUSYBOX_DEFAULT_BUSYBOX_EXEC_PATH="/proc/self/exe"
222 CONFIG_BUSYBOX_DEFAULT_FEATURE_SYSLOG=y
223 CONFIG_BUSYBOX_DEFAULT_PLATFORM_LINUX=y
224 CONFIG_BUSYBOX_DEFAULT_CROSS_COMPILER_PREFIX=""
225 CONFIG_BUSYBOX_DEFAULT_SYSROOT=""
226 CONFIG_BUSYBOX_DEFAULT_EXTRA_CFLAGS=""
227 CONFIG_BUSYBOX_DEFAULT_EXTRA_LDFLAGS=""
228 CONFIG_BUSYBOX_DEFAULT_EXTRA_LDLIBS=""
229 CONFIG_BUSYBOX_DEFAULT_INSTALL_APPLET_SYMLINKS=y
230 CONFIG_BUSYBOX_DEFAULT_PREFIX="./_install"
231 CONFIG_BUSYBOX_DEFAULT_NO_DEBUG_LIB=y
232 CONFIG_BUSYBOX_DEFAULT_FEATURE_BUFFERS_GO_ON_STACK=y
233 CONFIG_BUSYBOX_DEFAULT_PASSWORD_MINLEN=6
234 CONFIG_BUSYBOX_DEFAULT_MD5_SMALL=1
235 CONFIG_BUSYBOX_DEFAULT_SHA3_SMALL=1
236 CONFIG_BUSYBOX_DEFAULT_FEATURE_FAST_TOP=y
237 CONFIG_BUSYBOX_DEFAULT_FEATURE_EDITING=y
238 CONFIG_BUSYBOX_DEFAULT_FEATURE_EDITING_MAX_LEN=512
239 CONFIG_BUSYBOX_DEFAULT_FEATURE_EDITING_HISTORY=256
240 CONFIG_BUSYBOX_DEFAULT_FEATURE_TAB_COMPLETION=y
241 CONFIG_BUSYBOX_DEFAULT_FEATURE_EDITING_FANCY_PROMPT=y
242 CONFIG_BUSYBOX_DEFAULT_SUBST_WCHAR=0
243 CONFIG_BUSYBOX_DEFAULT_LAST_SUPPORTED_WCHAR=0
244 CONFIG_BUSYBOX_DEFAULT_FEATURE_NON_POSIX_CP=y
245 CONFIG_BUSYBOX_DEFAULT_FEATURE_USE_SENDFILE=y
246 CONFIG_BUSYBOX_DEFAULT_FEATURE_COPYBUF_KB=4
247 CONFIG_BUSYBOX_DEFAULT_IOCTL_HEX2STR_ERROR=y
248 CONFIG_BUSYBOX_DEFAULT_FEATURE_SEAMLESS_GZ=y
249 CONFIG_BUSYBOX_DEFAULT_GUNZIP=y
250 CONFIG_BUSYBOX_DEFAULT_ZCAT=y
251 CONFIG_BUSYBOX_DEFAULT_BUNZIP2=y
252 CONFIG_BUSYBOX_DEFAULT_BZCAT=y
253 CONFIG_BUSYBOX_DEFAULT_BZIP2_SMALL=0
254 CONFIG_BUSYBOX_DEFAULT_FEATURE_BZIP2_DECOMPRESS=y
255 CONFIG_BUSYBOX_DEFAULT_GZIP=y
256 CONFIG_BUSYBOX_DEFAULT_GZIP_FAST=0
257 CONFIG_BUSYBOX_DEFAULT_FEATURE_GZIP_DECOMPRESS=y
258 CONFIG_BUSYBOX_DEFAULT_TAR=y
259 CONFIG_BUSYBOX_DEFAULT_FEATURE_TAR_CREATE=y
260 CONFIG_BUSYBOX_DEFAULT_FEATURE_TAR_FROM=y
261 CONFIG_BUSYBOX_DEFAULT_FEATURE_TAR_GNU_EXTENSIONS=y
262 CONFIG_BUSYBOX_DEFAULT_BASENAME=y
263 CONFIG_BUSYBOX_DEFAULT_CAT=y
264 CONFIG_BUSYBOX_DEFAULT_CHGRP=y
265 CONFIG_BUSYBOX_DEFAULT_CHMOD=y
266 CONFIG_BUSYBOX_DEFAULT_CHOWN=y
267 CONFIG_BUSYBOX_DEFAULT_CHROOT=y
268 CONFIG_BUSYBOX_DEFAULT_CP=y
269 CONFIG_BUSYBOX_DEFAULT_CUT=y
270 CONFIG_BUSYBOX_DEFAULT_DATE=y
271 CONFIG_BUSYBOX_DEFAULT_FEATURE_DATE_ISOFMT=y
272 CONFIG_BUSYBOX_DEFAULT_DD=y
273 CONFIG_BUSYBOX_DEFAULT_FEATURE_DD_SIGNAL_HANDLING=y
274 CONFIG_BUSYBOX_DEFAULT_FEATURE_DD_IBS_OBS=y
```

```
275 CONFIG_BUSYBOX_DEFAULT_DF=y
276 CONFIG_BUSYBOX_DEFAULT_DIRNAME=y
277 CONFIG_BUSYBOX_DEFAULT_DU=y
278 CONFIG_BUSYBOX_DEFAULT_FEATURE_DU_DEFAULT_BLOCKSIZE_1K=y
279 CONFIG_BUSYBOX_DEFAULT_ECHO=y
280 CONFIG_BUSYBOX_DEFAULT_FEATURE_FANCY_ECHO=y
281 CONFIG_BUSYBOX_DEFAULT_ENV=y
282 CONFIG_BUSYBOX_DEFAULT_EXPR=y
283 CONFIG_BUSYBOX_DEFAULT_EXPR_MATH_SUPPORT_64=y
284 CONFIG_BUSYBOX_DEFAULT_FALSE=y
285 CONFIG_BUSYBOX_DEFAULT_FSYNC=y
286 CONFIG_BUSYBOX_DEFAULT_HEAD=y
287 CONFIG_BUSYBOX_DEFAULT_FEATURE_FANCY_HEAD=y
288 CONFIG_BUSYBOX_DEFAULT_ID=y
289 CONFIG_BUSYBOX_DEFAULT_LN=y
290 CONFIG_BUSYBOX_DEFAULT_LS=y
291 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_FILETYPES=y
292 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_FOLLOWLINKS=y
293 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_RECURSIVE=y
294 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_WIDTH=y
295 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_SORTFILES=y
296 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_TIMESTAMPS=y
297 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_USERNAME=y
298 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_COLOR=y
299 CONFIG_BUSYBOX_DEFAULT_FEATURE_LS_COLOR_IS_DEFAULT=y
300 CONFIG_BUSYBOX_DEFAULT_MD5SUM=y
301 CONFIG_BUSYBOX_DEFAULT_SHA256SUM=y
302 CONFIG_BUSYBOX_DEFAULT_FEATURE_MD5_SHA1_SUM_CHECK=y
303 CONFIG_BUSYBOX_DEFAULT_MKDIR=y
304 CONFIG_BUSYBOX_DEFAULT_MKFIFO=y
305 CONFIG_BUSYBOX_DEFAULT_MKNOD=y
306 CONFIG_BUSYBOX_DEFAULT_MKTEMP=y
307 CONFIG_BUSYBOX_DEFAULT_MV=y
308 CONFIG_BUSYBOX_DEFAULT_NICE=y
309 CONFIG_BUSYBOX_DEFAULT_PRINTF=y
310 CONFIG_BUSYBOX_DEFAULT_PWD=y
311 CONFIG_BUSYBOX_DEFAULT_READLINK=y
312 CONFIG_BUSYBOX_DEFAULT_FEATURE_READLINK_FOLLOW=y
313 CONFIG_BUSYBOX_DEFAULT_RM=y
314 CONFIG_BUSYBOX_DEFAULT_RMDIR=y
315 CONFIG_BUSYBOX_DEFAULT_SEQ=y
316 CONFIG_BUSYBOX_DEFAULT_SLEEP=y
317 CONFIG_BUSYBOX_DEFAULT_FEATURE_FANCY_SLEEP=y
318 CONFIG_BUSYBOX_DEFAULT_SORT=y
319 CONFIG_BUSYBOX_DEFAULT_SYNC=y
320 CONFIG_BUSYBOX_DEFAULT_TAIL=y
321 CONFIG_BUSYBOX_DEFAULT_FEATURE_FANCY_TAIL=y
322 CONFIG_BUSYBOX_DEFAULT_TEE=y
323 CONFIG_BUSYBOX_DEFAULT_FEATURE_TEE_USE_BLOCK_IO=y
324 CONFIG_BUSYBOX_DEFAULT_TEST=y
325 CONFIG_BUSYBOX_DEFAULT_TEST1=y
326 CONFIG_BUSYBOX_DEFAULT_TEST2=y
327 CONFIG_BUSYBOX_DEFAULT_FEATURE_TEST_64=y
328 CONFIG_BUSYBOX_DEFAULT_TOUCH=y
329 CONFIG_BUSYBOX_DEFAULT_FEATURE_TOUCH_SUSV3=y
330 CONFIG_BUSYBOX_DEFAULT_TR=y
331 CONFIG_BUSYBOX_DEFAULT_TRUE=y
332 CONFIG_BUSYBOX_DEFAULT_UNAME=y
333 CONFIG_BUSYBOX_DEFAULT_UNAME_OSNAME="GNU/Linux"
334 CONFIG_BUSYBOX_DEFAULT_UNIQ=y
335 CONFIG_BUSYBOX_DEFAULT_WC=y
336 CONFIG_BUSYBOX_DEFAULT_YES=y
337 CONFIG_BUSYBOX_DEFAULT_FEATURE_PRESERVE_HARDLINKS=y
338 CONFIG_BUSYBOX_DEFAULT_FEATURE_HUMAN_READABLE=y
339 CONFIG_BUSYBOX_DEFAULT_CLEAR=y
340 CONFIG_BUSYBOX_DEFAULT_DEFAULT_SETFONT_DIR=""
341 CONFIG_BUSYBOX_DEFAULT_RESET=y
342 CONFIG_BUSYBOX_DEFAULT_START_STOP_DAEMON=y
343 CONFIG_BUSYBOX_DEFAULT_WHICH=y
344 CONFIG_BUSYBOX_DEFAULT_AWK=y
345 CONFIG_BUSYBOX_DEFAULT_FEATURE_AWK_LIBM=y
346 CONFIG_BUSYBOX_DEFAULT_FEATURE_AWK_GNU_EXTENSIONS=y
347 CONFIG_BUSYBOX_DEFAULT_CMP=y
```

```
348 CONFIG_BUSYBOX_DEFAULT_SED=y
349 CONFIG_BUSYBOX_DEFAULT_VI=y
350 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_MAX_LEN=1024
351 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_COLON=y
352 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_YANKMARK=y
353 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_SEARCH=y
354 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_USE_SIGNALS=y
355 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_DOT_CMD=y
356 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_READONLY=y
357 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_SETOPTS=y
358 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_SET=y
359 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_WIN_RESIZE=y
360 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_ASK_TERMINAL=y
361 CONFIG_BUSYBOX_DEFAULT_FEATURE_VI_UNDO_QUEUE_MAX=0
362 CONFIG_BUSYBOX_DEFAULT_FEATURE_ALLOW_EXEC=y
363 CONFIG_BUSYBOX_DEFAULT_FIND=y
364 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_PRINT0=y
365 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_MTIME=y
366 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_PERM=y
367 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_TYPE=y
368 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_XDEV=y
369 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_MAXDEPTH=y
370 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_NEWER=y
371 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_EXEC=y
372 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_USER=y
373 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_GROUP=y
374 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_NOT=y
375 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_DEPTH=y
376 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_PAREN=y
377 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_SIZE=y
378 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_PRUNE=y
379 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_PATH=y
380 CONFIG_BUSYBOX_DEFAULT_FEATURE_FIND_REGEX=y
381 CONFIG_BUSYBOX_DEFAULT_GREP=y
382 CONFIG_BUSYBOX_DEFAULT_EGREP=y
383 CONFIG_BUSYBOX_DEFAULT_FGREP=y
384 CONFIG_BUSYBOX_DEFAULT_FEATURE_GREP_CONTEXT=y
385 CONFIG_BUSYBOX_DEFAULT_XARGS=y
386 CONFIG_BUSYBOX_DEFAULT_FEATURE_XARGS_SUPPORT_CONFIRMATION=y
387 CONFIG_BUSYBOX_DEFAULT_FEATURE_XARGS_SUPPORT_QUOTES=y
388 CONFIG_BUSYBOX_DEFAULT_FEATURE_XARGS_SUPPORT_TERMOPT=y
389 CONFIG_BUSYBOX_DEFAULT_FEATURE_XARGS_SUPPORT_ZERO_TERM=y
390 CONFIG_BUSYBOX_DEFAULT_HALT=y
391 CONFIG_BUSYBOX_DEFAULT_POWEROFF=y
392 CONFIG_BUSYBOX_DEFAULT_REBOOT=y
393 CONFIG_BUSYBOX_DEFAULT_TELINIT_PATH=""
394 CONFIG_BUSYBOX_DEFAULT_FEATURE_KILL_DELAY=0
395 CONFIG_BUSYBOX_DEFAULT_INIT_TERMINAL_TYPE=""
396 CONFIG_BUSYBOX_DEFAULT_FEATURE_SHADOWPASSWDS=y
397 CONFIG_BUSYBOX_DEFAULT_LAST_ID=0
398 CONFIG_BUSYBOX_DEFAULT_FIRST_SYSTEM_ID=0
399 CONFIG_BUSYBOX_DEFAULT_LAST_SYSTEM_ID=0
400 CONFIG_BUSYBOX_DEFAULT_FEATURE_DEFAULT_PASSWD_ALGO="md5"
401 CONFIG_BUSYBOX_DEFAULT_LOGIN=y
402 CONFIG_BUSYBOX_DEFAULT_LOGIN_SESSION_AS_CHILD=y
403 CONFIG_BUSYBOX_DEFAULT_PASSWD=y
404 CONFIG_BUSYBOX_DEFAULT_FEATURE_PASSWD_WEAK_CHECK=y
405 CONFIG_BUSYBOX_DEFAULT_DEFAULT_MODULES_DIR=""
406 CONFIG_BUSYBOX_DEFAULT_DEFAULT_DEPMOD_FILE=""
407 CONFIG_BUSYBOX_DEFAULT_DMESG=y
408 CONFIG_BUSYBOX_DEFAULT_FEATURE_DMESG_PRETTY=y
409 CONFIG_BUSYBOX_DEFAULT_FLOCK=y
410 CONFIG_BUSYBOX_DEFAULT_HEXDUMP=y
411 CONFIG_BUSYBOX_DEFAULT_HWCLOCK=y
412 CONFIG_BUSYBOX_DEFAULT_MKSWAP=y
413 CONFIG_BUSYBOX_DEFAULT_MOUNT=y
414 CONFIG_BUSYBOX_DEFAULT_FEATURE_MOUNT_HELPERS=y
415 CONFIG_BUSYBOX_DEFAULT_FEATURE_MOUNT_CIFS=y
416 CONFIG_BUSYBOX_DEFAULT_FEATURE_MOUNT_FLAGS=y
417 CONFIG_BUSYBOX_DEFAULT_FEATURE_MOUNT_FSTAB=y
418 CONFIG_BUSYBOX_DEFAULT_PIVOT_ROOT=y
419 CONFIG_BUSYBOX_DEFAULT_SWITCH_ROOT=y
420 CONFIG_BUSYBOX_DEFAULT_UMOUNT=y
```

```
421 CONFIG_BUSYBOX_DEFAULT_FEATURE_UMOUNT_ALL=y
422 CONFIG_BUSYBOX_DEFAULT_FEATURE_MOUNT_LOOP=y
423 CONFIG_BUSYBOX_DEFAULT_FEATURE_BEEP_FREQ=0
424 CONFIG_BUSYBOX_DEFAULT_FEATURE_BEEP_LENGTH_MS=0
425 CONFIG_BUSYBOX_DEFAULT_CROND=y
426 CONFIG_BUSYBOX_DEFAULT_FEATURE_CROND_DIR="/etc"
427 CONFIG_BUSYBOX_DEFAULT_CRONTAB=y
428 CONFIG_BUSYBOX_DEFAULT_LESS=y
429 CONFIG_BUSYBOX_DEFAULT_FEATURE_LESS_MAXLINES=9999999
430 CONFIG_BUSYBOX_DEFAULT_LOCK=y
431 CONFIG_BUSYBOX_DEFAULT_STRINGS=y
432 CONFIG_BUSYBOX_DEFAULT_TIME=y
433 CONFIG_BUSYBOX_DEFAULT_FEATURE_IPV6=y
434 CONFIG_BUSYBOX_DEFAULT_FEATURE_PREFER_IPV4_ADDRESS=y
435 CONFIG_BUSYBOX_DEFAULT_VERBOSE_RESOLUTION_ERRORS=y
436 CONFIG_BUSYBOX_DEFAULT_BRCTL=y
437 CONFIG_BUSYBOX_DEFAULT_FEATURE_BRCTL_FANCY=y
438 CONFIG_BUSYBOX_DEFAULT_FEATURE_BRCTL_SHOW=y
439 CONFIG_BUSYBOX_DEFAULT_IFCONFIG=y
440 CONFIG_BUSYBOX_DEFAULT_FEATURE_IFCONFIG_STATUS=y
441 CONFIG_BUSYBOX_DEFAULT_FEATURE_IFCONFIG_HW=y
442 CONFIG_BUSYBOX_DEFAULT_FEATURE_IFCONFIG_BROADCAST_PLUS=y
443 CONFIG_BUSYBOX_DEFAULT_IFUPDOWN_IFSTATE_PATH=""
444 CONFIG_BUSYBOX_DEFAULT_IP=y
445 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_ADDRESS=y
446 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_LINK=y
447 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_ROUTE=y
448 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_ROUTE_DIR="/etc/iproute2"
449 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_RULE=y
450 CONFIG_BUSYBOX_DEFAULT_FEATURE_IP_NEIGH=y
451 CONFIG_BUSYBOX_DEFAULT_NC=y
452 CONFIG_BUSYBOX_DEFAULT_NETMSG=y
453 CONFIG_BUSYBOX_DEFAULT_NETSTAT=y
454 CONFIG_BUSYBOX_DEFAULT_FEATURE_NETSTAT_WIDE=y
455 CONFIG_BUSYBOX_DEFAULT_FEATURE_NETSTAT_PRG=y
456 CONFIG_BUSYBOX_DEFAULT_NSLOOKUP_OPENWRT=y
457 CONFIG_BUSYBOX_DEFAULT_NTPD=y
458 CONFIG_BUSYBOX_DEFAULT_FEATURE_NTPD_SERVER=y
459 CONFIG_BUSYBOX_DEFAULT_PING=y
460 CONFIG_BUSYBOX_DEFAULT_PING6=y
461 CONFIG_BUSYBOX_DEFAULT_FEATURE_FANCY_PING=y
462 CONFIG_BUSYBOX_DEFAULT_ROUTE=y
463 CONFIG_BUSYBOX_DEFAULT_TRACEROUTE=y
464 CONFIG_BUSYBOX_DEFAULT_TRACEROUTE6=y
465 CONFIG_BUSYBOX_DEFAULT_FEATURE_TRACEROUTE_VERBOSE=y
466 CONFIG_BUSYBOX_DEFAULT_DHCPD_LEASES_FILE=""
467 CONFIG_BUSYBOX_DEFAULT_UDHCPC=y
468 CONFIG_BUSYBOX_DEFAULT_UDHCPC_DEFAULT_SCRIPT="/usr/share/udhcpc/default.script"
469 CONFIG_BUSYBOX_DEFAULT_UDHCP_DEBUG=0
470 CONFIG_BUSYBOX_DEFAULT_UDHCPC_SLACK_FOR_BUGGY_SERVERS=80
471 CONFIG_BUSYBOX_DEFAULT_FEATURE_UDHCP_RFC3397=y
472 CONFIG_BUSYBOX_DEFAULT_IFUPDOWN_UDHCPC_CMD_OPTIONS=""
473 CONFIG_BUSYBOX_DEFAULT_FEATURE_MIME_CHARSET=""
474 CONFIG_BUSYBOX_DEFAULT_FREE=y
475 CONFIG_BUSYBOX_DEFAULT_KILL=y
476 CONFIG_BUSYBOX_DEFAULT_KILLALL=y
477 CONFIG_BUSYBOX_DEFAULT_PGREP=y
478 CONFIG_BUSYBOX_DEFAULT_PIDOF=y
479 CONFIG_BUSYBOX_DEFAULT_PS=y
480 CONFIG_BUSYBOX_DEFAULT_FEATURE_PS_WIDE=y
481 CONFIG_BUSYBOX_DEFAULT_BB_SYSCTL=y
482 CONFIG_BUSYBOX_DEFAULT_TOP=y
483 CONFIG_BUSYBOX_DEFAULT_FEATURE_TOP_CPU_USAGE_PERCENTAGE=y
484 CONFIG_BUSYBOX_DEFAULT_FEATURE_TOP_CPU_GLOBAL_PERCENTS=y
485 CONFIG_BUSYBOX_DEFAULT_UPTIME=y
486 CONFIG_BUSYBOX_DEFAULT_SV_DEFAULT_SERVICE_DIR=""
487 CONFIG_BUSYBOX_DEFAULT_SH_IS_ASH=y
488 CONFIG_BUSYBOX_DEFAULT_BASH_IS_NONE=y
489 CONFIG_BUSYBOX_DEFAULT_ASH=y
490 CONFIG_BUSYBOX_DEFAULT_ASH_INTERNAL_GLOB=y
491 CONFIG_BUSYBOX_DEFAULT_ASH_BASH_COMPAT=y
492 CONFIG_BUSYBOX_DEFAULT_ASH_JOB_CONTROL=y
493 CONFIG_BUSYBOX_DEFAULT_ASH_ALIAS=y
```

```
494 CONFIG_BUSYBOX_DEFAULT_ASH_EXPAND_PRMT=y
495 CONFIG_BUSYBOX_DEFAULT_ASH_ECHO=y
496 CONFIG_BUSYBOX_DEFAULT_ASH_PRINTF=y
497 CONFIG_BUSYBOX_DEFAULT_ASH_TEST=y
498 CONFIG_BUSYBOX_DEFAULT_ASH_GETOPTS=y
499 CONFIG_BUSYBOX_DEFAULT_ASH_CMDCMD=y
500 CONFIG_BUSYBOX_DEFAULT_FEATURE_SH_MATH=y
501 CONFIG_BUSYBOX_DEFAULT_FEATURE_SH_MATH_64=y
502 CONFIG_BUSYBOX_DEFAULT_FEATURE_SH_NOFORK=y
503 CONFIG_BUSYBOX_DEFAULT_LOGGER=y
504 CONFIG_BUSYBOX_DEFAULT_FEATURE_SYSLOGD_READ_BUFFER_SIZE=0
505 CONFIG_BUSYBOX_DEFAULT_FEATURE_IPC_SYSLOG_BUFFER_SIZE=0
506 CONFIG_PACKAGE_dnsmasq=y
507 CONFIG_PACKAGE_dropbear=y
508 #
509 # Configuration
510 #
511 CONFIG_DROPBEAR_CURVE25519=y
512 CONFIG_DROPBEAR_DBCLIENT=y
513 CONFIG_PACKAGE_firewall=y
514 CONFIG_PACKAGE_fstools=y
515 CONFIG_PACKAGE_fwtool=y
516 CONFIG_PACKAGE_jsonfilter=y
517 CONFIG_PACKAGE_libc=y
518 CONFIG_PACKAGE_libgcc=y
519 CONFIG_PACKAGE_libpthread=y
520 CONFIG_PACKAGE_librt=y
521 CONFIG_PACKAGE_libssp=y
522 CONFIG_PACKAGE_libstdcpp=y
523 CONFIG_PACKAGE_logd=y
524 CONFIG_PACKAGE_mtd=y
525 CONFIG_PACKAGE_netifd=y
526 CONFIG_PACKAGE_openwrt-keyring=y
527 CONFIG_PACKAGE_opkg=y
528 CONFIG_PACKAGE_procd=y
529 #
530 # Configuration
531 #
532 CONFIG_PACKAGE_ubox=y
533 CONFIG_PACKAGE_ubus=y
534 CONFIG_PACKAGE_ubusd=y
535 CONFIG_PACKAGE_uci=y
536 CONFIG_PACKAGE_usign=y
537 #
538 # Administration
539 #
540 #
541 # SSL support
542 #
543 CONFIG_ZABBIX_NOSSL=y
544 #
545 # Database Software
546 #
547 CONFIG_ZABBIX_POSTGRESQL=y
548 #
549 # Boot Loaders
550 #
551 CONFIG_PACKAGE_grub2=y
552 #
553 # Development
554 #
555 #
556 # Libraries
557 #
558 CONFIG_PACKAGE_ar=y
559 CONFIG_PACKAGE_autoconf=y
560 CONFIG_PACKAGE_automake=y
561 CONFIG_PACKAGE_binutils=y
562 CONFIG_PACKAGE_gcc=y
563 CONFIG_PACKAGE_libtool-bin=y
564 CONFIG_PACKAGE_m4=y
565 CONFIG_PACKAGE_make=y
566 CONFIG_PACKAGE_objdump=y
```

```
567 #
568 # Extra packages
569 #
570 #
571 # Firmware
572 #
573 #
574 # ath10k IPQ4019 Boarddata
575 #
576 CONFIG_PACKAGE_bnx2-firmware=y
577 CONFIG_PACKAGE_r8169-firmware=y
578 #
579 # Kernel modules
580 #
581 #
582 # Hardware Monitoring Support
583 #
584 CONFIG_PACKAGE_kmod-hwmon-core=y
585 #
586 # I2C support
587 #
588 CONFIG_PACKAGE_kmod-i2c-core=y
589 CONFIG_PACKAGE_kmod-i2c-algo-bit=y
590 #
591 # Input modules
592 #
593 CONFIG_PACKAGE_kmod-input-core=y
594 #
595 # LED modules
596 #
597 #
598 # Libraries
599 #
600 CONFIG_PACKAGE_kmod-lib-crc-ccitt=y
601 #
602 # Native Language Support
603 #
604 CONFIG_PACKAGE_kmod-nls-base=y
605 CONFIG_PACKAGE_kmod-nls-utf8=y
606 #
607 # Netfilter Extensions
608 #
609 CONFIG_PACKAGE_kmod-ip6tables=y
610 CONFIG_PACKAGE_kmod-ipt-conntrack=y
611 CONFIG_PACKAGE_kmod-ipt-core=y
612 CONFIG_PACKAGE_kmod-ipt-nat=y
613 CONFIG_PACKAGE_kmod-ipt-offload=y
614 CONFIG_PACKAGE_kmod-nf-conntrack=y
615 CONFIG_PACKAGE_kmod-nf-conntrack6=y
616 CONFIG_PACKAGE_kmod-nf-flow=y
617 CONFIG_PACKAGE_kmod-nf-ipt=y
618 CONFIG_PACKAGE_kmod-nf-ipt6=y
619 CONFIG_PACKAGE_kmod-nf-nat=y
620 CONFIG_PACKAGE_kmod-nf-reject=y
621 CONFIG_PACKAGE_kmod-nf-reject6=y
622 #
623 # Network Devices
624 #
625 CONFIG_PACKAGE_kmod-bnx2=y
626 CONFIG_PACKAGE_kmod-e1000=y
627 CONFIG_PACKAGE_kmod-e1000e=y
628 CONFIG_PACKAGE_kmod-igb=y
629 CONFIG_PACKAGE_kmod-mii=y
630 CONFIG_PACKAGE_kmod-r8169=y
631 #
632 # Network Support
633 #
634 CONFIG_PACKAGE_kmod-ppp=y
635 CONFIG_PACKAGE_kmod-pppoe=y
636 CONFIG_PACKAGE_kmod-pppox=y
637 CONFIG_PACKAGE_kmod-slhc=y
638 #
639 # Other modules
```

```
640 #
641 CONFIG_PACKAGE_kmod-button-hotplug=y
642 CONFIG_PACKAGE_kmod-pps=y
643 CONFIG_PACKAGE_kmod-ptp=y
644 #
645 # Languages
646 #
647 #
648 # Perl
649 #
650 CONFIG_PACKAGE_perl=y
651 #
652 # Configuration
653 #
654 CONFIG_PERL_THREADS=y
655 CONFIG_PERL_NOCOMMENT=y
656 CONFIG_PACKAGE_perlbase-attributes=y
657 CONFIG_PACKAGE_perlbase-base=y
658 CONFIG_PACKAGE_perlbase-bytes=y
659 CONFIG_PACKAGE_perlbase-class=y
660 CONFIG_PACKAGE_perlbase-config=y
661 CONFIG_PACKAGE_perlbase-cwd=y
662 CONFIG_PACKAGE_perlbase-data=y
663 CONFIG_PACKAGE_perlbase-dynaloader=y
664 CONFIG_PACKAGE_perlbase-errno=y
665 CONFIG_PACKAGE_perlbase-essential=y
666 CONFIG_PACKAGE_perlbase-fcntl=y
667 CONFIG_PACKAGE_perlbase-file=y
668 CONFIG_PACKAGE_perlbase-filehandle=y
669 CONFIG_PACKAGE_perlbase-getopt=y
670 CONFIG_PACKAGE_perlbase-i18n=y
671 CONFIG_PACKAGE_perlbase-integer=y
672 CONFIG_PACKAGE_perlbase-io=y
673 CONFIG_PACKAGE_perlbase-list=y
674 CONFIG_PACKAGE_perlbase-locale=y
675 CONFIG_PACKAGE_perlbase-params=y
676 CONFIG_PACKAGE_perlbase-posix=y
677 CONFIG_PACKAGE_perlbase-re=y
678 CONFIG_PACKAGE_perlbase-scalar=y
679 CONFIG_PACKAGE_perlbase-selectsaver=y
680 CONFIG_PACKAGE_perlbase-selfloader=y
681 CONFIG_PACKAGE_perlbase-socket=y
682 CONFIG_PACKAGE_perlbase-symbol=y
683 CONFIG_PACKAGE_perlbase-text=y
684 CONFIG_PACKAGE_perlbase-thread=y
685 CONFIG_PACKAGE_perlbase-threads=y
686 CONFIG_PACKAGE_perlbase-tie=y
687 CONFIG_PACKAGE_perlbase-unicore=y
688 CONFIG_PACKAGE_perlbase-utf8=y
689 CONFIG_PACKAGE_perlbase-xsloader=y
690 #
691 # Python
692 #
693 CONFIG_PACKAGE_python=y
694 CONFIG_PACKAGE_python-asn1crypto=y
695 CONFIG_PACKAGE_python-base=y
696 CONFIG_PACKAGE_python-cffi=y
697 CONFIG_PACKAGE_python-codecs=y
698 CONFIG_PACKAGE_python-compiler=y
699 CONFIG_PACKAGE_python-cryptography=y
700 CONFIG_PACKAGE_python-ctypes=y
701 CONFIG_PACKAGE_python-db=y
702 CONFIG_PACKAGE_python-decimal=y
703 CONFIG_PACKAGE_python-dev=y
704 CONFIG_PACKAGE_python-distutils=y
705 CONFIG_PACKAGE_python-email=y
706 CONFIG_PACKAGE_python-enum34=y
707 CONFIG_PACKAGE_python-gdbm=y
708 CONFIG_PACKAGE_python-ipaddress=y
709 CONFIG_PACKAGE_python-lib2to3=y
710 CONFIG_PACKAGE_python-light=y
711 #
712 # Configuration
```

```
713 #
714 CONFIG_PACKAGE_python-logging=y
715 CONFIG_PACKAGE_python-multiprocessing=y
716 CONFIG_PACKAGE_python-ncurses=y
717 CONFIG_PACKAGE_python-openssl=y
718 CONFIG_PACKAGE_python-pip=y
719 CONFIG_PACKAGE_python-pip-conf=y
720 CONFIG_PACKAGE_python-ply=y
721 CONFIG_PACKAGE_python-pyasn1=y
722 CONFIG_PACKAGE_python-pycparser=y
723 CONFIG_PACKAGE_python-pydoc=y
724 CONFIG_PACKAGE_python-pyopenssl=y
725 CONFIG_PACKAGE_python-ruamel-yaml=y
726 CONFIG_PACKAGE_python-setuptools=y
727 CONFIG_PACKAGE_python-six=y
728 CONFIG_PACKAGE_python-sqlite3=y
729 CONFIG_PACKAGE_python-unittest=y
730 CONFIG_PACKAGE_python-xml=y
731 CONFIG_PACKAGE_python3=y
732 CONFIG_PACKAGE_python3-asn1crypto=y
733 CONFIG_PACKAGE_python3-asyncio=y
734 CONFIG_PACKAGE_python3-base=y
735 CONFIG_PACKAGE_python3-cffi=y
736 CONFIG_PACKAGE_python3-cgi=y
737 CONFIG_PACKAGE_python3-cgitb=y
738 CONFIG_PACKAGE_python3-codecs=y
739 CONFIG_PACKAGE_python3-cryptography=y
740 CONFIG_PACKAGE_python3-ctypes=y
741 CONFIG_PACKAGE_python3-dbm=y
742 CONFIG_PACKAGE_python3-decimal=y
743 CONFIG_PACKAGE_python3-dev=y
744 CONFIG_PACKAGE_python3-distutils=y
745 CONFIG_PACKAGE_python3-email=y
746 CONFIG_PACKAGE_python3-gdbm=y
747 CONFIG_PACKAGE_python3-lib2to3=y
748 CONFIG_PACKAGE_python3-light=y
749 #
750 #  Configuration
751 #
752 CONFIG_PACKAGE_python3-logging=y
753 CONFIG_PACKAGE_python3-lzma=y
754 CONFIG_PACKAGE_python3-multiprocessing=y
755 CONFIG_PACKAGE_python3-ncurses=y
756 CONFIG_PACKAGE_python3-openssl=y
757 CONFIG_PACKAGE_python3-pip=y
758 CONFIG_PACKAGE_python3-ply=y
759 CONFIG_PACKAGE_python3-pycparser=y
760 CONFIG_PACKAGE_python3-pydoc=y
761 CONFIG_PACKAGE_python3-pyopenssl=y
762 CONFIG_PACKAGE_python3-ruamel-yaml=y
763 CONFIG_PACKAGE_python3-setuptools=y
764 CONFIG_PACKAGE_python3-six=y
765 CONFIG_PACKAGE_python3-sqlite3=y
766 CONFIG_PACKAGE_python3-unittest=y
767 CONFIG_PACKAGE_python3-urllib=y
768 CONFIG_PACKAGE_python3-xml=y
769 #
770 #  Libraries
771 #
772 #
773 #  Compression
774 #
775 CONFIG_PACKAGE_libbz2=y
776 CONFIG_PACKAGE_liblzma=y
777 CONFIG_ZSTD_OPTIMIZE_O3=y
778 #
779 #  Firewall
780 #
781 CONFIG_PACKAGE_libip4tc=y
782 CONFIG_PACKAGE_libip6tc=y
783 CONFIG_PACKAGE_libxtables=y
784 #
785 #  SSL
```

```
786 #
787 CONFIG_PACKAGE_libopenssl=y
788 #
789 # Build Options
790 #
791 CONFIG_OPENSSL_OPTIMIZE_SPEED=y
792 CONFIG_OPENSSL_WITH_ASM=y
793 CONFIG_OPENSSL_WITH_DEPRECATED=y
794 CONFIG_OPENSSL_WITH_ERROR_MESSAGES=y
795 #
796 # Protocol Support
797 #
798 CONFIG_OPENSSL_WITH_TLS13=y
799 CONFIG_OPENSSL_WITH_SRP=y
800 CONFIG_OPENSSL_WITH_CMS=y
801 #
802 # Algorithm Selection
803 #
804 CONFIG_OPENSSL_WITH_EC=y
805 CONFIG_OPENSSL_WITH_CHACHA_POLY1305=y
806 CONFIG_OPENSSL_WITH_PSK=y
807 #
808 # Engine/Hardware Support
809 #
810 CONFIG_OPENSSL_ENGINE=y
811 #
812 # database
813 #
814 CONFIG_PACKAGE_libsqlite3=y
815 #
816 # Configuration
817 #
818 CONFIG_SQLITE3_DYNAMIC_EXTENSIONS=y
819 CONFIG_SQLITE3_FTS3=y
820 CONFIG_SQLITE3_FTS4=y
821 CONFIG_SQLITE3_FTS5=y
822 CONFIG_SQLITE3_JSON1=y
823 CONFIG_SQLITE3_RTREE=y
824 CONFIG_SQLITE3_THREADSAFE=y
825 #
826 # libelektra
827 #
828 CONFIG_PACKAGE_libbfd=y
829 CONFIG_PACKAGE_libblkid=y
830 CONFIG_PACKAGE_libblobmsg-json=y
831 CONFIG_PACKAGE_libbsd=y
832 CONFIG_PACKAGE_libcomerr=y
833 CONFIG_PACKAGE_libdb47=y
834 CONFIG_PACKAGE_libexpat=y
835 CONFIG_PACKAGE_libext2fs=y
836 CONFIG_PACKAGE_libf2fs=y
837 CONFIG_PACKAGE_libffi=y
838 CONFIG_PACKAGE_libgdbm=y
839 CONFIG_PACKAGE_libjson-c=y
840 CONFIG_PACKAGE_libncurses=y
841 CONFIG_PACKAGE_libnl-tiny=y
842 CONFIG_PACKAGE_libopcodes=y
843 CONFIG_PACKAGE_libpcre=y
844 CONFIG_PACKAGE_libreadline=y
845 CONFIG_PACKAGE_libsmartcols=y
846 CONFIG_PACKAGE_libss=y
847 CONFIG_PACKAGE_libubox=y
848 CONFIG_PACKAGE_libubus=y
849 CONFIG_PACKAGE_libuci=y
850 CONFIG_PACKAGE_libuclient=y
851 CONFIG_PACKAGE_libuuid=y
852 CONFIG_PACKAGE_libxml2=y
853 CONFIG_PACKAGE_terminfo=y
854 CONFIG_PACKAGE_zlib=y
855 #
856 # Configuration
857 #
858 #
```

```
859 # LuCI
860 #
861 #
862 # 1. Collections
863 #
864 #
865 # 2. Modules
866 #
867 CONFIG_LUCI_JSMIN=y
868 CONFIG_LUCI_CSSTIDY=y
869 #
870 # 3. Applications
871 #
872 #
873 # 4. Themes
874 #
875 #
876 # 5. Protocols
877 #
878 #
879 # 6. Libraries
880 #
881 CONFIG_PACKAGE_luci-lib-nixio_notls=y
882 #
883 # Select postfix build options
884 #
885 CONFIG_POSTFIX_TLS=y
886 CONFIG_POSTFIX_SASL=y
887 CONFIG_POSTFIX_LDAP=y
888 CONFIG_POSTFIX_CDB=y
889 CONFIG_POSTFIX_SQLITE=y
890 CONFIG_POSTFIX_PCRE=y
891 #
892 # Firewall
893 #
894 CONFIG_PACKAGE_ip6tables=y
895 CONFIG_PACKAGE_iptables=y
896 #
897 # OpenLDAP
898 #
899 CONFIG_OPENLDAP_DEBUG=y
900 #
901 # Web Servers/Proxies
902 #
903 CONFIG_PACKAGE_uhttpd=y
904 #
905 # wireless
906 #
907 CONFIG_PACKAGE_odhcp6c=y
908 CONFIG_PACKAGE_odhcp6c_ext_cer_id=0
909 CONFIG_PACKAGE_odhcpd-ipv6only=y
910 #
911 # Configuration
912 #
913 CONFIG_PACKAGE_odhcpd_ipv6only_ext_cer_id=0
914 CONFIG_PACKAGE_ppp=y
915 CONFIG_PACKAGE_ppp-mod-pppoe=y
916 #
917 # Disc
918 #
919 CONFIG_PACKAGE_partx-utils=y
920 #
921 # Filesystem
922 #
923 CONFIG_PACKAGE_e2fsprogs=y
924 CONFIG_PACKAGE_mkf2fs=y
925 #
926 # database
927 #
928 CONFIG_PACKAGE_gawk=y
929 CONFIG_PACKAGE_grep=y
930 CONFIG_PACKAGE_jshn=y
931 CONFIG_PACKAGE_libjson-script=y
```

932 CONFIG_STRACE_NONE=y

Network Scripts

E.1   Router

```
1  config interface 'loopback'
2      option ifname 'lo'
3      option proto 'static'
4      option ipaddr '127.0.0.1'
5      option netmask '255.0.0.0'
6
7  config interface 'eth0'
8      option ifname 'eth0'
9      option proto 'static'
10     option ipaddr '192.168.1.2'
11     option gateway '192.168.1.1'
12     option netmask '255.255.255.0'
13
14 config interface 'eth1'
15     option ifname 'eth1'
16     option proto 'dhcp'
```

E.2   Server

```
1  config interface 'loopback'
2      option ifname 'lo'
3      option proto 'static'
4      option ipaddr '127.0.0.1'
5      option netmask '255.0.0.0'
6
7  config interface 'lan'
8      option ifname 'eth0'
9      option proto 'static'
10     option ipaddr '192.168.2.2'
11     option gateway '192.168.2.1'
12     option netmask '255.255.255.0'
```

E.3   Client

```
1  config interface 'loopback'
2      option ifname 'lo'
3      option proto 'static'
4      option ipaddr '127.0.0.1'
5      option netmask '255.0.0.0'
6
7  config interface 'lan'
8      option ifname 'eth0'
9      option proto 'static'
10     option ipaddr '192.168.1.1'
```

```
11      option netmask '255.255.255.0'
12
13 config interface 'wan'
14      option ifname 'eth1'
15      option proto 'static'
16      option ipaddr '192.168.2.1'
17      option netmask '255.255.255.0'
18
19 config interface 'eth2'
20      option ifname 'eth2'
21      option proto 'dhcp'
```

Network Scripts

```
 1 sudo chgrp user /dev/net/tun
 2 #use 660 for root only
 3 #use 666 for non-root users
 4 sudo chmod 666 /dev/net/tun
 5
 6 #create a bridge
 7 sudo brctl addbr lan-bridge
 8 #turn off spanning tree
 9 sudo brctl stp lan-bridge off
10 #configure the IP address of the bridge
11 sudo ifconfig lan-bridge 192.168.1.254 netmask 255.255.255.0 up
12
13 #create a TAP interface for the router
14 sudo tunctl -u user -t lan-if
15 #bring up the interface
16 sudo ifconfig lan-if 0.0.0.0 promisc up
17 #add the interface the lan-bridge
18 sudo brctl addif lan-bridge lan-if
19
20 #create a TAP interface for the client
21 sudo tunctl -u user -t client-if
22 #bring up the interface
23 sudo ifconfig client-if 0.0.0.0 promisc up
24 #add the interface the lan-bridge
25 sudo brctl addif lan-bridge client-if
26
27 sudo brctl addbr wan-bridge
28 sudo brctl stp wan-bridge off
29 sudo ifconfig wan-bridge 192.168.2.254 netmask 255.255.255.0 up
30 sudo tunctl -u user -t wan-if
31 sudo ifconfig wan-if 0.0.0.0 -promisc up
32 sudo brctl addif wan-bridge wan-if
33
34 #create a TAP interface for the server
35 sudo tunctl -u user -t server-if
36 #bring up the interface
37 sudo ifconfig server-if 0.0.0.0 promisc up
38 #add the interface the lan-bridge
39 sudo brctl addif wan-bridge server-if
```