©2013 科学技術振興機構

D-Visor86 + D-System Monitor
環境構築手順書

科学技術振興機構
DEOS 研究開発センター

第 2.0 版
2013 年 5 月 1 日

目 次

1 D-Visor86 の概要 4

2 前提要件 5
- 2.1 ターゲットハードウェア 5
- 2.2 ターゲットソフトウェア 5

3 開発機での構築手順 6
- 3.1 OS インストール 6
- 3.2 ソースコードの展開とパッチ 6
- 3.3 コンパイル 7
  - 3.3.1 生成されるファイル 8
  - 3.3.2 コンパイル時間の短縮 8
- 3.4 ユーザランドの準備 9
  - 3.4.1 最小構成 initrd イメージ 9
    - 3.4.1.1 最小構成 initrd イメージの再構成 9
    - 3.4.1.2 BusyBox イメージの再構成 9
  - 3.4.2 Ubuntu initrd イメージ 11
    - 3.4.2.1 カーネルモジュールの作成 11
    - 3.4.2.2 initrd イメージの作成 11
- 3.5 バイナリアーカイブの作成 12
  - 3.5.1 カーネル実行イメージおよび initrd イメージ 12
  - 3.5.2 モジュール 13

4 ターゲット実機セットアップ手順 14
- 4.1 OS インストール 14
- 4.2 OS パッケージの追加と削除 15
  - 4.2.1 ユーザ OS (/dev/sda) のパッケージ調整 15
  - 4.2.2 監視 OS (/dev/sdb) のパッケージ調整 15
- 4.3 OS 設定 17
  - 4.3.1 ユーザ OS (/dev/sda) の設定 17
  - 4.3.2 監視 OS (/dev/sdb) の設定 18
- 4.4 起動に必要なファイルのコピー 18
- 4.5 Grub メニューの追加 19
- 4.6 アプリケーションのインストール 21
- 4.7 rootkit の自動起動設定 21
- 4.8 実行例 21

5 D-System Monitor 23
- 5.1 D-System Monitor 概要 23
  - 5.1.1 VIRTIO 機能 23
  - 5.1.2 ユーザ OS カーネル観測機能 23
  - 5.1.3 アプリケーション機能 24
- 5.2 dsysmon オプション一覧 24

5.2.1 共通オプション 25
5.2.2 VIRTIO オプション 25
5.2.3 FoxyKBD オプション 26
5.2.4 RootkitLibra オプション 26
5.2.5 LMS オプション 27
5.3 VIRTIO 機能の利用方法 27
5.3.1 virtio_console の利用方法 27
5.3.2 virtio_net の利用方法 28
5.3.2.1 host-to-host モードの利用方法 28
5.3.2.2 bridge モードの利用方法 29
5.3.3 virtio_blk の利用方法 29
5.4 FoxyKBD アプリケーション 30
5.5 RootkitLibra アプリケーション 31
5.6 LMS アプリケーション 32

6 GUI ツール — dsm-gui.py 33
6.1 dsm-gui.py オプション一覧 33
6.2 dsm-gui.py の操作方法 34
6.2.1 メニュー領域 35
6.2.2 FoxyKBD 用表示領域 35
6.2.3 RootkitLibra 用表示領域 36
6.2.4 LMS 用表示領域 36

7 デモ用サンプル rootkit 37
7.1 file_rootkit.ko 37
7.2 process_rootkit.ko 38
7.3 ttyrpld 38
7.3.1 rpld_receiver 39

8 デモ実行手順 40
8.1 デモシナリオ概要 40
8.2 デモシナリオ A 41
8.3 デモシナリオ B 41
8.3.1 FoxyKBD の操作 43
8.3.2 RootkitLibra の操作 43
8.3.3 LMS の操作 44

1　D-Visor86 の概要

D-Visor86 は、前身である Veidt VMM の成果を引き継ぎ、名称を変更して開発を継続しているマルチコアプロセッサ用の仮想マシンモニタ (VMM) システムである。マルチコア版プロセッサ（Hyper Threading Technology を含む）を搭載する i386 互換アーキテクチャ上で、各プロセッサコア毎に修正を施した Linux カーネルを動作させることができる。D-Visor86 の動作概念を図 1 に示す。

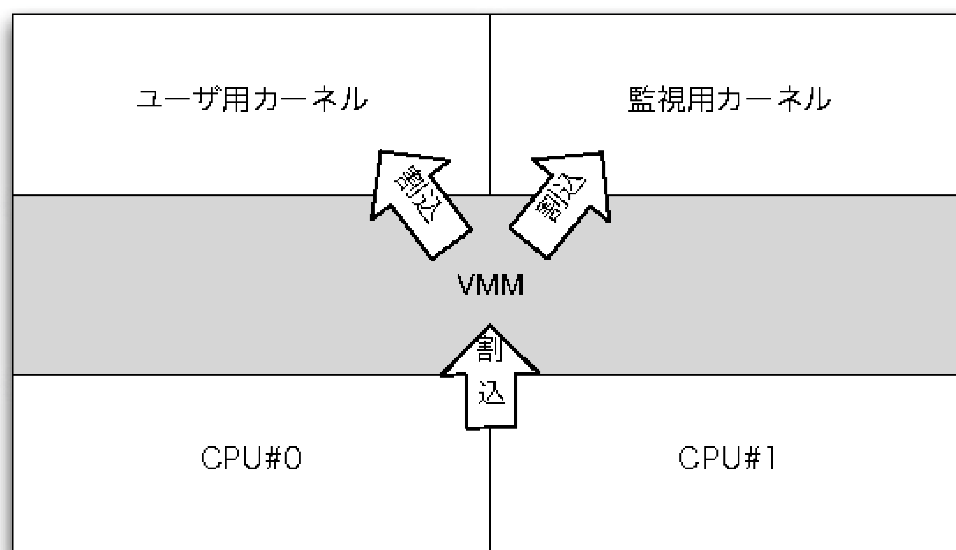

図 1: D-Visor86 の動作概念

- ブートローダ (Grub) が、カーネルとして VMM をロードして実行する。
- VMM が監視用カーネルと、ユーザ用カーネルを、異なる物理メモリ領域にロードする。同時にそれぞれのカーネル用に initrd もロードする。
- VMM は、監視用カーネルの実行を開始する。ユーザ用カーネルは、同時に実行することも、後に監視用カーネルからの指示に従って実行させることもできる。
- VMM は、メモリの割り当て制御と、割り込みを適当な側のカーネルにディスパッチする制御を主に行う。
- 各カーネルにおいて、アプリケーションプログラムは、sysfs を通して VMM とやりとりすることができる。

なお、D-Visor86 のソースコードの中には “veidt” の文字列が残されている部分があるため、本書の記述においても “veidt” の名称がそのまま使用されることがある。

2　前提要件

2.1　ターゲットハードウェア

現在、D-Visor86 は株式会社コンテック製 “BX-100n-DC5000-C01” をターゲットとして開発作業を行っている。表 1 にターゲット機の主要諸元を示す。

表 1: CONTEC BX-100n-DC5000-C01 の主要諸元

CPU	Intel Atom Processor N270 1.6GHz
チップセット	Intel 945GSE + ICH7M-DH
BIOS	Award 社製
メモリ	PC2-4300 DDR2 SDRAM（1GB 実装済み）
ビデオ	Intel 945GSE 内蔵
オーディオ	AC97 準拠
内蔵ディスク	2.5 インチ SATA HDD × 2 機（スロットイン） CF カードスロット Type 1 × 2 機（IDE としてブート可能）
シリアル IF	RS-232C × 5ch（9 ピン D-sub）
ネットワーク	1000BASE-T × 2ch (Intel 82573L)

2.2　ターゲットソフトウェア

現在、D-Visor86 は Ubuntu 10.04 をベースに開発作業を行なっている。ゲスト OS として動作させる Linux カーネルおよび initrd に使用する BusyBox として、それぞれ以下のバージョンをサポートしている。

- Linux 2.6.32.28
- BusyBox 1.18.5

3　開発機での構築手順

D-Visor86 システムは、Ubuntu 10.04 にて構築作業を行なうように調整されている。以下、一般的な i386 互換機を開発機として D-Visor86 システムを構築する手順を説明する。

3.1　OS インストール

開発機に Ubuntu 10.04 32 bit 版を標準的な手順でインストールする。インストールするパッケージの選択はデフォルトの状態とする。Ubuntu 10.04 のインストール手順詳細については、本書では割愛する。参考 URL を以下に挙げておく。

- https://help.ubuntu.com/10.04/installation-guide/i386/index.html
- https://wiki.ubuntulinux.jp/UbuntuTips

Ubuntu 10.04 のインストール後、以下に示すパッケージを追加する。

- patch
- zlib1g-dev

3.2　ソースコードの展開とパッチ

D-Visor86 のソースコードは、“dvisor86-<date>.tar.gz” という名称のアーカイブファイルとして配布されている。本書では、2011 年 10 月にリリースされた “dvisor86-20111031.tar.gz” を対象とする。アーカイブファイルを展開すると、表 2 に示すディレクトリとファイルが作成される。アーカイブファイルを展開したディレクトリを以降 “$HOME” と表記する。

別途、Linux カーネル 2.6.32.28 のソースアーカイブ “linux-2.6.32.28.tar.gz” が必要となるので、Linux カーネルの公式サイト http://kernel.org/ などから入手しておく。入手したソースアーカイブを次の手順で展開し、必要なパッチを適用する。

1) $HOME をワーキングディレクトリとして、Linux カーネルのソースアーカイブを展開する。ディレクトリ “linux-2.6.32.28” が生成される。

```
$ tar xvzf linux-2.6.32.28.tar.gz
```

2) Linux カーネルソースを展開したディレクトリ名を “linux-2.6.32.28” から “linux-2.6.32.28-dv86” に変更する。

```
$ mv linux-2.6.32.28 linux-2.6.32.28-dv86
```

3) 展開した Linux カーネルソースに対して、D-Visor86 用のパッチを適用する

```
$ make linux-apply-patch
```

表 2: D-Visor86 配布ファイルの内容

名前	内容
Makefile	構築用のトップレベル Makefile
mk/	構築用の Makefile 補助設定ファイルなど
files/	実行環境の構築に必要となるファイル群
files/initrd/	initrd に組み込む設定ファイルやスクリプトなど
tools/	D-System Monitor 監視プログラム (dsysmon) およびテスト用ユーザランドプログラムなど
host-tools/	エミュレータ実行用のスクリプトなど
rootkits/	D-System Monitor 監視プログラム動作試験用 rootkit など
vmm/	D-Visor86 VMM のソースコード
configs/	Linux カーネルおよび BusyBox 用コンフィグレーションファイル
linux-2.6.32.28-native-config	D-Visor86 組み込み無し
linux-2.6.32.28-watcher-config	監視カーネル用
linux-2.6.32.28-user-config	ユーザカーネル用
busybox-1.18.5-config	initrd busybox 用
patches/	パッチファイル群
linux-2.6.32.28-dv86-patches	Linux 2.6.32.28 カーネルソースツリーに適用するパッチファイル
ubuntu-10.04-init-getty.patch	監視 OS、ユーザ OS 両方に適用する ttyS1, hvc0 で getty を起動するためのパッチファイル
ubuntu-10.04-nfs-kernel-server.patch	監視 OS 上で動作する NFS サーバの起動スクリプトに適用するパッチファイル

3.3 コンパイル

D-Visor86 のコンパイルは、$HOME ディレクトリで “make all” コマンドを実行することで行われる。

```
$ cd $HOME
$ make all
```

VMM と 3 種類の Linux カーネルを生成するため、相応の時間がかかる。make コマンドによって実行される主な処理内容は以下の通りである。

1) $HOME/vmm/lib ディレクトリで、VMM 用ライブラリをコンパイルする。カーネル構築時に使用する “veidtdefs.h” ファイルも生成する。

2) $HOME/vmm/kernel ディレクトリで、VMM 本体オブジェクトをコンパイルする。

3) $HOME/vmm ディレクトリで VMM 用ライブラリと VMM 本体オブジェクトをリンクし、VMM の実行バイナリを構築する。

4) $HOME/build/linux-watcher ディレクトリで、監視用カーネルをコンパイルする。

5) $HOME/build/linux-user ディレクトリで、ユーザ用カーネルをコンパイルする。

6) $HOME/build/linux-native ディレクトリで、native 用カーネルをコンパイルする。

7) $HOME/tools ディレクトリで D-System Monitor 監視プログラム (dsysmon) およびテスト用のプログラム群をコンパイルする。

8) $HOME/rootkits ディレクトリで D-System Monitor 監視プログラム (dsysmon) 動作試験用 rootkit 群をコンパイルする。

9) $HOME/build/initrd ディレクトリに initrd に含めるファイル群を展開し、initrd のバイナリイメージを作成する。

3.3.1 生成されるファイル

"make all" コマンドの実行によって生成される主要なファイルを表 3 に示す。

表 3: D-Visor86 の構築により生成されるファイル

ファイル名	機能・役割
vmm/dv86-vmm	VMM 本体の実行イメージ
build/linux-user/vmlinux.bin	ユーザ用カーネルの実行イメージ
build/linux-user/arch/x86/boot/bzImage	ユーザ用カーネルの bzImage
build/linux-watcher/vmlinux.bin	監視用カーネルの実行イメージ
build/linux-watcher/arch/x86/boot/bzImage	監視用カーネルの bzImage
build/linux-native/vmlinux.bin	native カーネルの実行イメージ
build/linux-native/arch/x86/boot/bzImage	native カーネルの bzImage
tools/dsysmon/dsysmon	D-System Monitor 監視プログラム
tools/dsm-gui/dsm-gui.py	D-System Monitor GUI ツール
tools/rtkl-collect/rtkl-collect	RootkitLibra 監視機能のユーザ OS 用サポートプログラム
tools/dxfeed/dxfeed	DXFEED 機能テスト用プログラム（デバッグ用）
tools/gp_fault/gp_fault	GP 例外発生ツール（デバッグ用）
tools/profile/profile	VMM プロファイル表示ツール（デバッグ用）
rootkits/file-rootkit/file_rootkit.ko	ファイル隠蔽およびファイルサイズ改竄 rootkit
rootkits/process-rootkit/process_rootkit.ko	プロセス隠蔽 rootkit
rootkits/rpld-receiver/rpld_receiver	TTY 入出力受け取り用 TCP サーバ
rootkits/ttyrpld-2.6.0/k_linux2.6/rpldev.ko	rpld 用カーネルモジュール
rootkits/ttyrpld-2.6.0/user/rpld	TTY 入出力ロギング rootkit
build/initrd/user-kernel.map	監視機能で使用するユーザ用カーネルのシンボル情報
build/initrd/dv86-scripts/*	D-Visor86 用各種スクリプト
build/initrd-dv86.img	BusyBox の実行イメージと設定ファイル、D-System Monitor 用の各種ツールなどを含む initrd イメージ

3.3.2 コンパイル時間の短縮

D-Visor86 の構築システムは GNU make コマンドが提供する parallel-make 機能をサポートしている。make コマンドの引数として "-j *n*" を指定することで、同時に *n* 個の構築プロセスが走行する。開発機がマルチ

プロセッサ構成であり、十分なメモリを搭載している場合、parallel-make 機能を利用して構築時間を短縮することができる。

3.4 ユーザランドの準備

D-Visor86 を動作させるためには、各 Linux カーネルに応じたユーザランドが必要になる。特に、initrd はカーネルのバージョンや構成に依存する部分があるので、それぞれの作成方法を説明する。

3.4.1 最小構成 initrd イメージ

通常、監視用 OS では多数のアプリケーションを実行することはなく、すべてのユーザランド機能を initrd のみで実現することも可能である。D-Visor86 の構築時に、必要最小限の機能を含む initrd イメージが "$HOME/build/initrd-dv86.img" として作成されるため、これをそのまま利用することができる。

3.4.1.1 最小構成 initrd イメージの再構成

D-Visor86 にて作成される $HOME/build/initrd-dv86.img には、以下に示す 3 種類のファイルが含まれる。

1) あらかじめコンパイルされている BusyBox の実行イメージとシンボリックリンク群
2) $HOME/files/initrd 以下にあらかじめ用意されている設定ファイルとスクリプト群
3) コンパイルされたコマンド・ツールの実行イメージおよび動的に生成される情報ファイル群

これらのすべてのファイルは、一旦 $HOME/build/initrd ディレクトリ以下に集められた後、cpio にてアーカイブされ、gzip にて圧縮されて initrd イメージが構成される [1]。表 4 に initrd-dv86.img に含まれるファイルのうち、busybox のシンボリックリンクを除くファイルの一覧を示す。

initrd-dv86.img にさらにスクリプトやファイルを追加したい場合は、$HOME/files/initrd 以下に追加したいファイルを置き、$HOME/build/initrd-dv86.cpio を削除してから "make initrd" コマンドを実行すれば良い。

```
$ cd $HOME
$ cp <追加ファイル名> files/initrd/
$ rm build/initrd-dv86.cpio
$ make initrd
```

3.4.1.2 BusyBox イメージの再構成

D-Visor86 の配布アーカイブファイルでは、構築処理の簡略化のために initrd で利用される BusyBox をコンパイル済みのバイナリとして用意している。あらかじめコンパイルされた BusyBox の実行バイナリとシンボリックリンク群は、"$HOME/files/busybox-installed.img" に initrd イメージと同様の cpio + gzip 形式で格納されている。また、BusyBox をコンパイルする際に使用されたコンフィグレーションファイルが "$HOME/configs/busybox-1.18.5-config" として含まれている。

BusyBox のコンフィグレーションを変更したい場合、あるいは BusyBox のバージョンを更新したい場合は、以下の手順にしたがい BusyBox を再コンパイルし、$HOME/files/busybox-installed.img を置き換える。

1) 開発機の Ubuntu 10.04 環境に必要なパッケージを追加する。

```
# apt-get build-dep busybox
# apt-get install libncurses5-dev
```

[1] D-Visor86 で作成される initrd イメージは、cpio + gzip で処理されているため、厳密には "initramfs" 形式である。本書では、慣例に従って "initrd" という表現を用いている。

表 4: initrd-dv86.img に含まれるファイルの一覧

ファイル名	説明
/init	初期実行スクリプト
/user-kernel.map	ユーザ用カーネルのシンボル情報ファイル
/etc/inittab	/bin/init 動作指定設定ファイル
/etc/fstab	ファイルシステムマウント指定ファイル
/etc/{passwd,group}	パスワード・グループ設定ファイル
/etc/init.d/rcS	起動スクリプト
/etc/rcS.d/	起動スクリプト用ディレクトリ（現状は空）
/bin/busybox	busybox 実行バイナリ
/dv86-scripts/start_virtio	VIRTIO 機能開始用スクリプト
/dv86-scripts/setup_bridge	VIRTIO ネットワークブリッジ設定スクリプト
/dsm-tools/dsysmon	D-System Monitor 監視プログラム
/dsm-tools/dxfeed	DXFEED 機能テスト用プログラム
/dsm-tools/rtkl-collect	RootkitLibra 監視機能サポートプログラム
/dsm-tools/gp_fault	GP 例外発生ツール
/dsm-tools/profile	VMM プロファイル表示ツール
/dsm-tools/DEMO-A.sh	デモシナリオ A 用 dsysmon 起動スクリプト
/dsm-tools/DEMO-B-dsm.sh	デモシナリオ B 用 dsysmon 起動スクリプト
/dsm-tools/DEMO-B-gui.sh	デモシナリオ B 用 dsm-gui.py 起動スクリプト
/dsm-tools/dsm-gui.py	D-System Monitor GUI ツールプログラム
/rootkits/dsm-rtkt.sh	rootkit 自動インストールおよび起動用スクリプト
/rootkits/file_rootkit.ko	ファイル隠蔽、ファイルサイズ改竄 rootkit
/rootkits/process_rootkit.ko	プロセス隠蔽 rootkit
/rootkits/rpld	TTY 入出力ロギングツール
/rootkits/rpld.conf	rpld 設定ファイル
/rootkits/rpld_receiver	TTY 入出力受信用 TCP サーバプログラム
/rootkits/rpldev.ko	rpld 用カーネルモジュール

2) $HOME をワーキングディレクトリとし、BusyBox のソースアーカイブをダウンロードして展開する。

```
$ cd $HOME
$ wget http://busybox.net/downloads/busybox-1.18.5.tar.bz2
$ tar xvzf busybox-1.18.5.tar.bz2
$ cd busybox-1.18.5
```

3) 設定ファイルをコピーして構築する。必要であれば設定を変更する。

```
$ cp $HOME/configs/busybos-1.18.5-config .config
$ make oldconfig
$ make
```

4) オプションを与えずに BusyBox をインストールする。ここでは、_install サブディレクトリにバイナリとシンボリックリンク群がインストールされる。

```
$ make install
```

5) インストールされたファイル群をアーカイブし、$HOME/files/busybox-installed.img を置き換える。

```
$ cd _install
$ find . | cpio --quiet -H newc -o | gzip -c > $HOME/files/busybox-installed.img
```

3.4.2 Ubuntu initrd イメージ

ユーザ OS や、より詳細な検証などを行なう場合の監視 OS では、さまざまなプログラムを実行する可能性があるため、開発マシンと同一のユーザランドを使用したい。initrd は、使用するカーネルとユーザランドの両方に合致するものを用意する必要があるので、「開発機上で」initrd イメージを作成する方法を説明する。なお、実験目的で、ユーザ OS で多くのプログラムを実行しない場合などは、3.4.1 節で示したように最小構成 initrd をそのまま使用しても構わない。

3.4.2.1 カーネルモジュールの作成

まず、initrd に格納するカーネルモジュールを作成し、開発機の適当なディレクトリにインストールする。これにより、開発機で動作する Ubuntu 10.04 に付属するツールを使って、簡単に initrd イメージを作成することができる。なお、配布ファイルに含まれるカーネルコンフィグレーションでは、scsi_wait_scan.ko モジュールのみが作成される。

1) $HOME をワーキングディレクトリとする。

2) ユーザ側カーネルのモジュールを構築する [2]。

```
$ make -C linux-2.6.32.28-dv86 O=$HOME/build/linux-user modules
```

3) root 権限にて、構築したカーネルモジュールを所定のパスにインストールする。

```
# make -C linux-2.6.32.28-dv86 O=$HOME/build/linux-user modules_install
```

4) 同様の手順で、監視側のモジュールを構築する。

```
$ make -C linux-2.6.32.28-dv86 O=$HOME/build/linux-watcher modules
```

5) root 権限にて、監視側のカーネルモジュールを所定のパスにインストールする。

```
# make -C linux-2.6.32.28-dv86 O=$HOME/build/linux-watcher modules_install
```

以上の手順により、ユーザ用、監視用それぞれのカーネルモジュールとモジュール依存ファイルなどが開発機の /lib/modules/2.6.32.28+dv86-user および /lib/modules/2.6.32.28+dv86-watcher ディレクトリにインストールされる。

3.4.2.2 initrd イメージの作成

カーネルモジュールを所定のパスにインストールしておけば、Ubuntu 10.04 に附属する標準の mkinitramfs コマンドで Ubuntu 10.04 用の initrd イメージを作成することができる。$HOME を作業ディレクトリとし、以下のコマンドを実行する。

```
$ /usr/sbin/mkinitramfs -v -o ./ubuntu-initrd-dv86-user.img 2.6.32.28+dv86-user
$ /usr/sbin/mkinitramfs -v -o ./ubuntu-initrd-dv86-watcher.img 2.6.32.28+dv86-watcher
```

生成された ubuntu-initrd-dv86-user.img および ubuntu-initrd-dv86-watcher.img が、それぞれがユーザ用カーネルと監視用カーネルで Ubuntu 10.04 のユーザランドを使用するときの initrd イメージとなる。

[2] ‘O=’ で指定するディレクトリは絶対パスで指定する必要がある。

3.5 バイナリアーカイブの作成

D-Visor86 をターゲット実機で動作させるためには、VMM の実行バイナリやカーネル実行イメージ、initrd イメージなどをターゲット環境にコピーする必要がある。以下、本節ではターゲット実機にコピーするファイルをアーカイブにまとめる手順を説明する。

3.5.1 カーネル実行イメージおよび initrd イメージ

ターゲット実機の /boot ディレクトリには、表 5 に示すファイルをコピーする必要がある。

表 5: /boot にコピーするファイル一覧

ファイル名	コピー元	説明
dv86-vmm	vmm/dv86-vmm	D-Visor86 VMM 実行イメージ
initrd-dv86.img	build/initrd-dv86.img	最小構成 initrd イメージ
vmlinux.bin-dv86-watcher	build/linux-watcher/vmlinux.bin	監視用カーネル
vmlinux.bin-dv86-user	build/linux-user/vmlinux.bin	ユーザ用カーネル
vmlinux.bin-dv86-native	build/linux-native/vmlinux.bin	native 用カーネル
vmlinuz-dv86-watcher	build/linux-watcher/arch/x86/boot/bzImage	監視用カーネル (bzImage)
vmlinuz-dv86-user	build/linux-user/arch/x86/boot/bzImage	ユーザ用カーネル (bzImage)
vmlinuz-dv86-native	build/linux-native/arch/x86/boot/bzImage	native 用カーネル (bzImage)
ubuntu-initrd-dv86-user.img	ユーザ OS 用 Ubuntu initrd イメージ	3.4.2 節参照
ubuntu-initrd-dv86-watcher.img	監視 OS 用 Ubuntu initrd イメージ	3.4.2 節参照
menu.lst.BX100n-sample	files/menu.lst.BX100n-sample	Grub メニューのサンプル

以下の手順にしたがい、表 5 に示すファイル群と監視 OS、ユーザ OS の基本設定を変更するパッチファイルを実機にコピーするためのアーカイブファイルを作成する。

1) $HOME をワーキングディレクトリとする。

2) “make runtime” を実行し、生成されたバイナリファイルと initrd イメージを $HOME/runtime ディレクトリ以下にまとめる。

```
$ make runtime
```

3) 3.4.2 節で生成したユーザ OS 用および監視 OS 用の initrd イメージを $HOME/runtime/boot ディレクトリにコピーする。

```
$ cp $HOME/ubuntu-initrd-dv86-user.img runtime/boot
$ cp $HOME/ubuntu-initrd-dv86-watcher.img runtime/boot
```

4) ユーザ OS および監視 OS のユーザランドに適用するパッチファイルを $HOME/runtime ディレクトリにコピーする。

```
$ cp $HOME/patches/ubuntu-10.04-init-getty.patch runtime/
$ cp $HOME/patches/ubuntu-10.04-nfs-kernel-server.patch runtime/
```

5) $HOME/runtime 以下のファイルをアーカイブにまとめる。

```
$ tar cvzf dv86-kernels.tar.gz -C runtime .
```

3.5.2 モジュール

以下の手順にしたがい、3.4.2 節で構築したカーネルモジュールをアーカイブファイルにまとめる。

1) $HOME をワーキングディレクトリとする。

2) /lib/modules/2.36.28+dv86-user および /lib/modules/2.36.28+dv86-watcher の両ディレクトリ以下のカーネルモジュールをアーカイブファイルにまとめる。

```
$ tar cvzf 2.6.32.28+dv86-user.tar.gz -C /lib/modules 2.6.32.28+dv86-user
$ tar cvzf 2.6.32.28+dv86-watcher.tar.gz -C /lib/modules 2.6.32.28+dv86-watcher
```

4 ターゲット実機セットアップ手順

本節では、ターゲット実機に D-Visor86 を組み込むための方法を説明する。ターゲット実機は 2.1 節に示したようにコンテック製 “BX-100n-DC5000-C01” を対象とする。

ターゲット実機上におけるユーザ OS、監視 OS それぞれ設定上のポイントを以下に示す。

1) ターゲット実機の BIOS 設定は “Optimized Default” の状態とし、起動 HDD として Master 側を選択し、USB キーボード機能を有効にする [3]。

2) ターゲット実機に内蔵されている二台の HDD には、ユーザ用と監視用の Ubuntu 10.04 をそれぞれ個別にインストールする。ユーザ用を Master 側 (sda) とし、監視用を Slave 側 (sdb) とする。

3) COM1 は VMM のコンソールメッセージ専用とする。デモ動作時に VMM のメッセージを参照する必要はなく、また入力を行なうこともないので、COM1 に端末を接続しておく必要はない。

4) COM2 はユーザ OS または監視 OS のコンソール出力として使用されるので、端末を接続しておく。シリアル回線の速度は 115200 bps とする [4]。

5) 後述のデモシナリオにおいては、ターゲット実機のイーサネットは使用しない。ただし、ターゲット実機への OS インストール時などにはパッケージ取得のために外部との接続が必要となる。必要時以外はケーブルを接続しないようにする。

6) ユーザ OS は後述のデモシナリオ A/B で rootfs をマウントする経路（デバイス）が異なるため、/etc/fstab に固定的な値を記述できない。rootfs と swap の記述は削除する [5]。

7) ユーザ OS、監視 OS ともに、後述のデモシナリオによって VGA/GUI 以外のデバイスをコンソールとすることがある。コンソールで getty を起動するための設定ファイルを追加しておく必要がある。

8) 後述のシナリオ B にける監視 OS では、GUI 表示プログラムなどが動作するので、相当量のメモリを消費する。監視 OS では swap を有効にしておく。

9) 後述のシナリオ B におけるユーザ OS では、nfsroot が供給されるネットワークデバイス (eth0) が起動時の処理にて再初期化されてはいけない。ユーザ OS では、ネットワークの自動設定はすべて無効としておく必要がある。

4.1 OS インストール

Ubuntu 10.04 32 bit 版をインストールする。インストールするパッケージの選択はデフォルトの状態とする。ただし、インストール先の HDD として、ユーザ OS と監視 OS でそれぞれ異なるディスクを選択する必要がある。ユーザ OS、監視 OS のインストール先は以下のようにする。

ユーザ OS Master 側 HDD (/dev/sda) にインストールする。ユーザ OS インストール時には、Slave 側 HDD (/dev/sdb) には触らないようにする。

監視 OS Slave 側 HDD (/dev/sdb) にインストールする。既にユーザ OS がインストールされている場合、インストーラは自動的に /dev/sda を選択した状態となるので、選択を切り替えて /dev/sdb にインストールするように指定する。監視 OS インストール時には Master 側 HDD (/dev/sda) には触らないようにする。

[3] IDE コントローラのモードを AHCI としても動作する。AHCI とした方が性能的に有利と思われる。
[4] 後述のデモシナリオにおいては、監視 OS のみが COM2 を使用する。
[5] rootfs の記述を削除しても、カーネル起動オプションで指定されたデバイスが継続使用されるので問題はない。

また、ターゲット実機である BX-100n-DC5000-C01 には光学ドライブが内蔵されていないため、外付け USB CD-ROM ドライブなどを用いてインストールを行なう必要がある。

4.2 OS パッケージの追加と削除

ユーザ OS、監視 OS それぞれでパッケージの追加と削除を行う。パッケージの追加をする前に、パッケージインデックスファイルの更新を行っておく必要がある。

```
# apt-get update
```

4.2.1 ユーザ OS (/dev/sda) のパッケージ調整

- Network Manager パッケージを削除する。

```
# apt-get purge network-manager
```

- ureadahead パッケージを削除する。

```
# apt-get purge ureadahead
```

- grub-pc (GRUB2) パッケージを削除する。念のため、/boot/grub ディレクトリをバックアップしてから grub-pc をアンインストールする。

```
# mv /boot/grub /boot/grub2
# apt-get purge grub-pc
```

- grub (GRUB Legacy) パッケージを追加する。update-grub 実行時の質問には「Y」と答える。grub-install に指定するデバイスはユーザ OS をインストールした /dev/sda とする。

```
# mkdir /boot/grub
# apt-get install grub
# update-grub
# grub-install /dev/sda
```

- patch パッケージを追加する。

```
# apt-get install patch
```

- libhx22 パッケージを追加する。これは、TTY 入出力ロギングプログラム rpld で必要となる。

```
# apt-get install libhx22 libhx-dev
```

4.2.2 監視 OS (/dev/sdb) のパッケージ調整

- Network Manager パッケージを削除する。

```
# apt-get purge network-manager
```

- ureadahead パッケージを削除する。

```
# apt-get purge ureadahead
```

- grub-pc (GRUB2) パッケージを削除する。念のため、/boot/grub ディレクトリをバックアップしてから grub-pc をアンインストールする。

```
# mv /boot/grub /boot/grub2
# apt-get purge grub-pc
```

- grub (GRUB Legacy) パッケージを追加する。update-grub 実行時の質問には「Y」と答える。grub-install に指定するデバイスは監視 OS をインストールした /dev/sdb とする。

```
# mkdir /boot/grub
# apt-get install grub
# update-grub
# grub-install /dev/sdb
```

- patch パッケージを追加する。

```
# apt-get install patch
```

- nfs-kernel-server パッケージを追加する。これは、監視 OS で NFS サーバを稼働させるために必要となる。

```
# apt-get install nfs-kernel-server
```

- python-wxgtk2.8 および python-matplotlib パッケージを追加する。これらは、D-System Monitor GUI ツール "dsm-gui.py" で必要となる。

```
# apt-get install python-wxgtk2.8 python-matplotlib
```

- bridge-utils パッケージを追加する。このパッケージには、virtio_net で bridge モードを設定する際に使用する "brctl" コマンドが含まれる。後述のデモシナリオでは bridge モードは使用しないので、必ずしもインストールする必要はない。

```
# apt-get install brige-utils
```

- uml-utilities パッケージを追加する。このパッケージには、virtio_net で host-to-host モードを設定する際に tun/tap インタフェースを操作・削除する "tunctl" コマンドが含まれる。後述するデモシナリオでは tun/tap インタフェースを手動で制御する必要はないため、必ずしもインストールする必要はない。

```
# apt-get install uml-utilities
```

なお、もし Grub の画面に /dev/sdb の Ubuntu を起動するための選択項目が無く、/dev/sdb の Ubuntu を起動できない場合、Grub 内の /dev/sda の Ubuntu 起動用のエントリを編集して /dev/sdb で起動できるようにできる。手順は以下のようになる。

1) Grub の OS 選択画面を起動する。
2) 編集したいエントリ上で "e" キーを押す。今回はユーザ OS 起動用のエントリ上で "e" を押す。
3) 「uuid ...」となっている項目上で "d" キーを押し、削除する。
4) 「kernel ...」となっている項目上で "e" キーを押し、編集を開始する。
5) 「root=UUID=...」と記載されている部分を「root=/dev/sdb1」に置き換える
6) エンターキーを押し、編集を完了する。
7) "b" キーを押し、編集したエントリで起動する。

4.3 OS 設定

後述するデモシナリオを動作させるために、OS の基本設定をいくつか変更・修正しておく必要がある。OS の設定をするためには一部パッチを適用する必要があるため、3.5 節で作成した dv86-kernels.tar.gz アーカイブファイルをユーザ OS、監視 OS 上でそれぞれ展開しておく。

1) dv86-kernels.tar.gz アーカイブファイルを展開する。ここでは、ユーザのホームディレクトリに展開するものと想定する。

```
# tar xvzf dv86-kernels.tar.gz ~/
```

2) 展開すると、表 6 に示すファイル群が得られる。

表 6: dv86-kernels.tar.gz に含まれるファイル

名前	説明
ubuntu-10.04-init-getty.patch	ttyS1, hvc0 で getty を起動するためのパッチファイル
ubuntu-10.04-nfs-kernel-server.patch	NFS サーバの起動スクリプトに適用するパッチファイル
boot/	VMM/Kernel 一式
dsm-tools/	D-System Monitor プログラム一式
rootkits/	デモ用 rootkit 一式

4.3.1 ユーザ OS (/dev/sda) の設定

ユーザ OS において、以下に示す設定を行なう。

ネットワーク設定

Network Manager パッケージが削除されているため、ネットワークインタフェースの設定は /etc/network/interfaces の設定内容に応じて自動的に初期化される。後述のデモシナリオでは、ネットワークインタフェースが自動的に初期化されることは望ましくないため、/etc/network/interfaces において lo 以外のインタフェースを auto 指定から削除しておく。

ファイルシステムマウント設定

ユーザ OS の /etc/fstab に含まれる rootfs と swap の行を削除する。ユーザ OS では、シナリオにより rootfs として物理的な HDD を直接マウントする場合と、NFS (virtio_net) 経由でマウントする場合が生じるが、この際に混乱しないようにしておく。どちらの場合も、カーネル起動オプションで指定されたデバイスが継続的に利用されるので、指定を削除しても問題はない。

getty 起動スクリプト設定

ttyS1 または hvc0 (virtio_console) に対して、後述のデモシナリオに応じて選択的に getty を起動するため、/etc/init 以下に ttyS1.conf と hvc0.conf を追加する。4.3 節冒頭で展開したアーカイブより得られた ubuntu-10.04-init-getty.patch を適用すると、これらの設定ファイルが作成される。

```
# cd /etc/init
# patch -p2 < ubuntu-10.04-init-getty.patch
```

4.3.2　監視 OS (/dev/sdb) の設定

監視 OS において、以下に示す設定を行なう。

ネットワーク設定

Network Manager パッケージが削除されているため、ネットワークインタフェースの設定は /etc/network/interfaces の設定内容に応じて自動的に初期化される。後述のデモシナリオでは、ネットワークインタフェースが自動的に初期化されることは望ましくないため、/etc/network/interfaces において lo 以外のインタフェースを auto 指定から削除しておく。

getty 起動スクリプト設定

ttyS1 または hvc0 (virtio_console) に対して、後述のデモシナリオに応じて選択的に getty を起動するため、/etc/init 以下に ttyS1.conf と hvc0.conf を追加する。4.3 節冒頭で展開したアーカイブより得られた ubuntu-10.04-init-getty.patch を適用すると、これらの設定ファイルが作成される。

```
# cd /etc/init
# patch -p2 < ubuntu-10.04-init-getty.patch
```

NFS サーバ起動スクリプト修正

Ubuntu 10.04 オリジナルの nfs-kernel-server パッケージで追加される起動スクリプト /etc/init.d/nfs-kernel-server にはロジック上のミスがあり、後述のデモシナリオの構成では NFS サーバ機能が正しく起動できないため、修正しておく必要がある。4.3 節冒頭で展開したアーカイブより得られた ubuntu-10.04-nfs-kernel-server.patch を適用すると、起動スクリプトが修正される。

```
# cd /etc/init.d
# patch -p2 < ubuntu-10.04-nfs-kernel-server.patch
```

NFS サーバ export 設定

後述のデモシナリオ B においてユーザ OS の rootfs を NFS で供給する際、ユーザ OS の rootfs である /dev/sda1 をマウントするディレクトリ /export/user-root を作成し、当該ディレクトリがエクスポートされるように /etc/exports ファイルに設定を追加する。エクスポートするディレクトリは Grub の menu.lst 内で指定されるため、menu.lst の指定と一致していなければならない。

```
# mkdir -p /export/user-root
```

/etc/exports の末尾に以下の行を追加する。

```
/export/user-root      192.168.1.0/24(rw,sync,no_root_squash,no_subtree_check)
```

4.4　起動に必要なファイルのコピー

4.3 節の冒頭で展開したディレクトリ内にある boot ディレクトリを、ユーザ OS と監視 OS の /boot ディレクトリにコピーする。

1) ユーザ OS、監視 OS それぞれで~/boot を /boot にコピーする。

```
# cp ~/boot/* /boot
```

3.5 節で作成した 2.6.32.28+dv86-user.tar.gz および 2.6.32.28+dv86-watcher.tar.gz ファイルを、それぞれユーザ OS および監視 OS の /lib/modules ディレクトリに展開する。

1) ユーザ OS 上で 2.6.32.28+dv86-user.tar.gz ファイルを展開する。

```
# tar xvzf 2.6.32.28+dv86-user.tar.gz -C /lib/modules
```

2) 監視 OS 上で 2.6.32.28+dv86-watcher.tar.gz ファイルを展開する。

```
# tar xvzf 2.6.32.28+dv86-watcher.tar.gz -C /lib/modules
```

dv86-kernels.tar.gz, 2.6.32.28+dv86-user.tar.gz および 2.6.32.28+dv86-watcher.tar.gz については、バイナリ配布パッケージに含まれているものを使用することも可能である [6]。

4.5 Grub メニューの追加

D-Visor86 VMM、Linux カーネル実行バイナリ、および initrd イメージは、Grub によりメモリ上に読み込まれる。すなわち、Grub エントリの kernel 行を用いて VMM を、続く module 行を用いてカーネル実行バイナリと initrd イメージを指定する。

/boot/grub/menu.lst ファイルを編集して、D-Visor86 VMM をロードするエントリを追加する。追加する内容の一例を以下に示す。

```
01: title    D-Visor86 (Without VMM, User PV-Kernel sda1)
02: root     (hd0,0)
03: kernel   /boot/vmlinuz.dv86-user root=/dev/sda1 ro
04: initrd   /boot/ubuntu-dv86-initrd.img
05:
06: title    D-Visor86 (COM1->VMM COM2->Watcher VGA->User sda1)
07: root     (hd0,0)
08: kernel   /boot/dv86-vmm loglevel=info console=serial,0,115200 \
09:          cpu0_lower_mem=0x1000,16K cpu0_upper_mem=0x03000000,384M \
10:          cpu0_use=com2 \
11:          cpu1_lower_mem=0x11000,256K cpu1_upper_mem=0x1b000000,512M \
12:          cpu1_use=vga,kbd,pci,ata cpu1_no_auto_start
13: module   /boot/vmlinux.bin-dv86-watcher console=ttyS1,115200 \
14:          earlyprintk=ttyS1,115200
15: module   /boot/initrd-dv86.img
16: module   /boot/vmlinux.bin-dv86-user root=/dev/sda1 ro
17: module   /boot/ubuntu-initrd-dv86-user.img
```

上記例には表示していないが、メニュー表示（hiddenmenu をコメントアウト）や、タイムアウト時間 (timeout) も調整しておいた方がよいだろう。また、必要に応じて /etc/fstab ファイルを変更し、マウントするファイルシステムを調整しておく必要がある。

1~4 行目で、VMM なしの状態でユーザ用カーネルを起動するエントリを追加している。13~15 行目で監視用のカーネルとその initrd イメージを、16~17 行目でユーザ用のカーネルとその initrd イメージを指定している。最初にロードされた VMM が、これらのパラメータを解析してカーネル実行バイナリと initrd イメージをロードする。3 行目と 16 行目の 'root=' の設定は環境に合わせて変更しておく。

ここで、最も重要なのは 8~12 行目で、kernel 行に指定する VMM のコマンドラインである。指定可能なパラメータを以下に示す。

console= コンソールを "serial,<ポート>,[<スピード>]" の形式で指定する。<ポート> は 0 から始まる数値で、<スピード> を省略した場合は '115200' となる。

loglevel= ログレベルとして、'info' 'error' 'detail1' 'detail2' 'detail3' のいずれかを指定する。

[6] これらのバイナリアーカイブファイルは、D-Visor86 の配布ソースアーカイブには含まれていない。

cpu#_uses= ‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS が使用するデバイスの一覧を指定する。デバイスには ‘com1’ ‘com2’ ‘com3’ ‘com4’ ‘vga’ ‘kdb’ ‘pci’ ‘ata’ から必要なものをカンマ文字 ‘,’ で区切って指定する。

cpu#_os= ‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS を指定する。現在指定できるのは ‘linux’ のみ。

cpu#_load_addr=

‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS をロードするアドレスを指定する。省略した場合は、プロテクトメモリの最下位部にロードされる。

cpu#_lower_mem=

‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS が使用するコンベンショナルメモリ領域を “<開始アドレス>,<サイズ>” の形式で指定する。

cpu#_upper_mem=

‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS が使用するプロテクトメモリ領域を “<開始アドレス>,<サイズ>” の形式で指定する。< 開始アドレス > は、カーネルコンフィグレーションの CONFIG_PHYSICAL_START オプションと一致している必要がある。配布ファイルでは、監視用カーネルが 0x03000000、ユーザ用カーネルが 0x1b000000 である。

cpu#_no_auto_start

‘#’ には CPU 番号（‘0’ か ‘1’）を指定し、その CPU で実行するゲスト OS を自動的には実行開始しないことを指定する。但し、‘cpu1_no_auto_start’ は、指定しても無視される。

3.5 節で作成した dv86-kernels.tar.gz ファイルに含まれる menu.lst.BX100n-sample は、実際に開発段階で使用されていた menu.lst の内容をそのまま記録したものである。既存の menu.lst と置き換えた上で、一部修正して使用することを想定している。

```
# cp /boot/menu.lst.BX100n-sample /boot/grub/menu.lst
```

menu.lst.BX100n-sample に含まれる各エントリの ‘root=’ オプションについては、パーティション構成等に合わせて変更する必要がある。これはターゲット実機上にて各パーティションの UUID を確認して書き換えると良い。各パーティションの UUID は blkid コマンドで確認できる。たとえば、以下のような表示となる。

```
# blkid
/dev/sda1: UUID="cd9ec974-7b90-4810-865e-9bfc8e53f9e5" TYPE="ext4"
/dev/sda5: UUID="1add48bb-eab5-4f8b-9889-46a76b96742b" TYPE="swap"
/dev/sdb1: UUID="27222c67-b401-49c9-a6d5-63ec0b30deb2" TYPE="ext4"
/dev/sdb5: UUID="e0857da8-155d-41fc-a517-a36cd1044962" TYPE="swap"
```

なお、本書ではターゲット実機の BIOS 設定にて起動 HDD を Primary としているため、通常はユーザ OS 側に組み込まれた menu.lst が読み込まれ、ユーザ OS 側の /boot 以下に置かれている VMM, kernel, initrd 等が使用される。BIOS 設定にて起動 HDD を Secondary とすると、監視 OS 側の設定やバイナリが使用されることになる。

4.6　アプリケーションのインストール

後述のデモを実行するためには、ユーザ OS、監視 OS 上それぞれについて rootfs 上にアプリケーションファイルなどをインストールしておく必要がある。4.3 節冒頭で展開した dsm-tools ディレクトリと rootkits ディレクトリをユーザ OS、監視 OS 上にそれぞれコピーする。

- ~/dsm-tools を / 以下にコピーする。

```
# cp -r ~/dsm-tools /
```

- ~/rootkits を / 以下にコピーする。

```
# cp -r ~/rootkits /
```

4.7　rootkit の自動起動設定

ユーザ OS では、起動時にデモ用 rootkit を組み込み、ファイルの隠蔽およびプロセスの隠蔽が自動的に行なわれるように、起動スクリプトを設定する。次のように、/rootkits/dsm-rtkt.sh を /etc/rc3.d/S95dsm-rtkt にシンボリックリンクする。

```
# cd /etc/rc3.d
# ln -s ../../rootkits/dsm-rtkt.sh ./S95-dsm-rtkt
```

rootkit 群の自動起動スクリプトである S95-dsm-rtkt は /etc/rc3.d に配置されるため、Linux 起動後に run-level が 3 となった場合に有効となる。Grub の menu.lst に含まれるオリジナルの起動項目では run-level が 2 となるため、自動的に rootkit 群が組み込まれることはない。D-System Monitor デモ用の起動項目では run-level を 3 としているため、自動的に rootkit が組み込まれる。

自動起動スクリプト dsm-rtkt.sh で実行される処理概要は以下の通りである。

- eth1 が存在する場合、IP アドレス 192.168.2.86 として初期化を行う。
- ttyrpld を起動する。TTY 入出力のログは /var/log/rpl/rpld.log に書き込まれる。加えて、ネットワークを経由して 192.168.2.87 の 10080 番ポートにログを送信する。
- ファイル隠蔽用 rootkit を組み込み、/rootkits 以下にある以下の 4 つのファイルを隠す。
  - rpldev.ko
  - file_rootkit.ko
  - process_rootkit.ko
  - rpld
- プロセス隠蔽 rootkit を組み込み、ttyrpld のデーモンである rpld プロセスを隠す。なお、rpld プロセスの PID は /var/run/rpld.pid に記録されている。

4.8　実行例

1) ターゲット実機を再起動し、Grub メニューから 4.5 節で例示したエントリ

   D-Visor86 (COM1->VMM COM2->Watcher VGA->User sda1)

   を選択し、D-Visor86 VMM を起動する。

2) シリアルコンソール 0 (COM1) に D-Visor86 VMM の起動メッセージが表示される。

```
[init] starting vmm.
[init] vmm using 0xfc000000 ... 0xfd7fffff
[cpu0] [Startup memory map]
[cpu0] guest           0x00000000 ... 0x00004fff -> 0x00000000- A,D,RW
[cpu0] guest           0x00005000 ... 0x02ffffff -> 0x3f700000  A,D,RW
[cpu0] guest           0x03000000 ... 0x24ffffff -> 0x03000000- A,D,RW
[cpu0] guest           0x25000000 ... 0xfbffffff -> 0x3f700000  A,D,RW
[cpu0] vmmkernel       0xfc100000 ... 0xfc142fff -> 0x00100000- G,A,D,RW
[cpu0] percpu          0xfc143000 ... 0xfc55bfff -> 0x0170d000- G,A,D,RW
[cpu0] iomem(apic)     0xfd55c000 ... 0xfd55cfff -> 0xfee00000  G,A,D,PCD,RW
[cpu0] Linux kernel loaded in 0x03000000 ... 0x033c9cde
[cpu0] Linux initrd loaded in 0x03bbe000 ... 0x03fff3ff
[cpu0] Command line is 'console=ttyS1,115200    earlyprintk=ttyS1,115200 \
nosep idle=halt reservetop=67108864'
[cpu0] starting guest.
```

3) シリアルコンソール 1 (COM2) には、監視用カーネルの起動メッセージに続いて、コンソール有効化を促すメッセージが表示される。リターンを入力することで BusyBox のシェルプロンプトが表示される。

```
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 2.6.32.28+dv86-watcher(tom@endgame.soum.co.jp) \
(gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5) ) #3 Mon Aug 22 05:52:25 JST 2011
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
……（途中省略）……
Mounting filesystems.
Hello, this is Watcher OS.

Please press Enter to activate this console.
~ # /dsm-tools/dsysmon -s
```

4) シリアルコンソール 1 (COM2) のプロンプトに対して、“dsysmon -s” コマンドを実行すると、CPU1 を使用してユーザ用カーネルが起動する。シリアルコンソール 0 (COM1) には VMM の動作メッセージが表示され、VGA コンソールにはユーザ用カーネルの起動メッセージに続いて、ログイン画面が表示される。

```
[cpu1] [Startup memory map]
[cpu1] guest           0x00000000 ... 0x00000fff -> 0x00005000  A,D,RW
[cpu1] guest           0x00001000 ... 0x00010fff -> 0x3f700000  A,D,RW
[cpu1] guest           0x00011000 ... 0x00050fff -> 0x00011000- A,D,RW
[cpu1] guest           0x00051000 ... 0x0009efff -> 0x3f700000  A,D,RW
[cpu1] guest           0x0009f000 ... 0x000c7fff -> 0x0009f000- A,D,RW
[cpu1] guest           0x000c8000 ... 0x000effff -> 0x3f700000  A,D,RW
[cpu1] guest           0x000f0000 ... 0x000fffff -> 0x000f0000- A,D,RW
[cpu1] guest           0x00100000 ... 0x03ffffff -> 0x3f700000  A,D,RW
[cpu1] guest           0x04000000 ... 0x24ffffff -> 0x04000000- A,D,RW
[cpu1] guest           0x25000000 ... 0x3f6dffff -> 0x3f700000  A,D,RW
[cpu1] guest           0x3f6e0000 ... 0xfbffffff -> 0x3f6e0000- A,D,RW
[cpu1] vmmkernel       0xfc100000 ... 0xfc142fff -> 0x00100000- G,A,D,RW
[cpu1] percpu          0xfc143000 ... 0xfc55bfff -> 0x01b26000- G,A,D,RW
[cpu1] iomem(apic)     0xfd55c000 ... 0xfd55cfff -> 0xfee00000  G,A,D,PCD,RW
[cpu1] Linux kernel loaded in 0x04000000 ... 0x0451087a
[cpu1] Linux initrd loaded in 0x23b70000 ... 0x23fff1ff
[cpu1] Command line is 'root=/dev/sda1 ro nosep idle=halt reservetop=67108864'
[cpu1] starting guest.
```

5　D-System Monitor

D-System Monitor (DSM) は、ユーザ OS カーネルに対する意図的な改竄を動的に検出する監視システムで、D-Visor86 VMM と密接に連携して動作する。D-Visor86 の配布ソースアーカイブには、D-System Monitor の実装一式も含まれている。

D-System Monitor の改竄検出アルゴリズムは、監視 OS カーネルの下で動作する監視モニタ "dsysmon" プログラムによって実装されている。本節では、D-System Monitor の概要を示すとともに、dsysmon プログラムの使用方法について説明する。

5.1　D-System Monitor 概要

ユーザ OS カーネルに対する意図的な改竄を検出するには、監視システムに以下の機能が必要となる。

- ユーザ OS のデバイス入出力を観測する機能
- ユーザ OS のカーネル動作を観測する機能
- 改竄検出のアルゴリズム (アプリケーション) 実装

D-System Monitor では、これらの機能を D-Visor86 VMM、監視 OS カーネルに組み込まれる専用デバイスドライバ (dde)、および監視 OS で動作する監視モニタ dsysmon が協調することで実現している。

5.1.1　VIRTIO 機能

D-System Monitor では、ユーザ OS のデバイス入出力を Linux カーネルに標準で用意されている **VIRTIO 機能**により行なうことを前提としている。VIRTIO 機能のバックエンド部分を監視 OS 側に実装することにより、監視 OS においてユーザ OS のデバイス入出力を観測できるようにしている。最終的に、VIRTIO 機能のバックエンドとして実デバイスへのアクセスを行なうのは、監視 OS で動作する監視モニタである dsysmon プログラムの役割となる。dsysmon は単なるユーザプロセスとして実装されており、アプリケーション機能の組み込みを容易にしている。

D-System Monitor の VIRTIO 機能バックエンドは、以下の VIRTIO 仮想デバイスをサポートしている。

- virtio_console — キャラクタコンソールデバイス
- virtio_net — ネットワークインタフェース
- virtio_bkl — ブロック (HDD) デバイス

VIRTIO 機能を利用するためには、dsysmon の起動時に仮想デバイスと実デバイスとの結び付きをオプションにて指定する必要がある。オプションの詳細は 5.2 節を参照のこと。

5.1.2　ユーザ OS カーネル観測機能

D-System Monitor は、ユーザ OS カーネルの状態および動作を観測する以下の機能を実装している。

- ユーザ OS のカーネル仮想メモリを参照する機能
- ユーザ OS がカーネルモードに移行したことを検出する機能
- ユーザ OS で実行されるシステムコール情報を取得する機能

ユーザ OS カーネルは D-Visor86 VMM の制御下で動作しており、原理的には VMM 層にて上記に示したすべての情報を収集することができる。収集された情報は VMM 層によるプロセッサ間の通信を経由して監視 OS カーネルに渡され、専用デバイスドライバにより dsysmon に通知される。

しかしながら、ユーザ OS カーネルで処理されるシステムコールの情報を VMM 層のみで収集するのは技術的に — 特に性能面において — 容易ではないため、D-System Monitor は代替策としてユーザ OS で動作するサポートプログラムを利用する方法を実装している。当該サポートプログラムは、自らシステムコールを発行し、その引数および戻り値の情報を **DXFEED 機能**を用いて直接的に VMM 層に通知する。このとき、監視対象であるユーザ OS カーネルは完全にバイパスされる。VMM 層は DXFEED 機能で受け取った情報を監視 OS で動作する dsysmon に通知する。

5.1.3 アプリケーション機能

D-System Monitor には、FoxyKBD, RootkitLibra, Lightweight Monitoring Service (LMS) の 3 つのアプリケーション機能が実装されている。

FoxyKBD

FoxyKBD は、ユーザ OS に組み込まれた “キーロガー” を検知するアプリケーションである。D-System Monitor では、監視対象 OS として組み込み Linux を想定していることから、TTY ベースのコンソールデバイスを観測対象としている。

RootkitLibra

RootkitLibra は、ユーザ OS に組み込まれた “ルートキット” をファイルシステムの観点から検知するアプリケーションである。ルートキットが自身の存在を隠蔽する場合、特定のファイルの存在を隠蔽、または特定のファイルのサイズ情報を改竄することで、ユーザランドからの検出を逃れる。RootkitLibra は、ユーザ OS のファイルシステムを NFS マウントすることで、ネットワークを経由する NFS パケットの内容を取得解析し、またユーザ OS で実行されるシステムコールの情報を取得解析して両者を比較することで、ファイルの隠蔽・改竄状態を検知する。

Lightweight Monitoring Service (LMS)

Lightweight Monitoring Service（以下、LMS と略称する）は、ユーザ OS に組み込まれた “ルートキット” をプロセスの観点から検知するアプリケーションである。ルートキットが自身の存在を隠蔽する場合、カーネル内部で維持されるプロセスリストから特定のプロセスを除外するよう改竄することで、ユーザランドからの検出を逃れる。LMS は、ユーザ OS カーネルが維持するプロセスリストと実行キューを動的に解析することで、プロセスの隠蔽状態を検出する。

これらのアプリケーション機能は dsysmon に組み込まれており、起動時のオプションにて有効化する機能を選択する。

5.2 dsysmon オプション一覧

D-System Monitor の監視モニタである dsysmon には、共通オプション、VIRTIO オプション、アプリケーション固有オプションが指定できる。有効化するアプリケーション機能を指定するオプションを**アプリケーションセレクタ**と呼ぶ。アプリケーションセレクタを指定しない場合、dsysmon は単に VIRTIO バックエンドとして動作することになる。以下に、dsysmon に指定できるオプションの一覧を示す。

5.2.1 共通オプション

−−verbose (−v)

冗長メッセージの表示を有効にする。−−verbose オプションが指定されると、dsysmon の各モジュール（有効化されているアプリケーションを含む）において、動作状況を報告する冗長なメッセージが表示されるようになる。−−verbose オプションは –v と略記することもできる。

−−debug (−d)

デバッグメッセージの表示を有効にする。−−debug オプションが指定されると、dsysmon の各モジュール（有効化されているアプリケーションを含む）において、デバッグメッセージが表示されるようになる。デバッグメッセージは大量に表示されるため、dsysmon の動作を阻害する要因となる。利用には注意を要する。−−debug オプションは –d と略記することもできる。

−−coop (−c)

協調 (cooperation) モードで dsysmon を起動する。協調モードでは /tmp/dsysmon<PID> 以下に外部制御プログラムとの通信を行なう FIFO を作成する。FIFO に対して特定の操作をすることで外部プログラムから dsysmon のアプリケーションと協調動作する事ができる。GUI ツールを利用する場合は指定する必要がある。−−coop オプションは –c と略記することもできる。

−−start (−s)

VIRTIO 機能を初期化した後、自動的にゲスト OS カーネルを起動する。−−start オプションは –s と略記することもできる。

−−help

dsysmon で指定可能なオプションの一覧を表示する。

5.2.2 VIRTIO オプション

−−console

virtio_console デバイスのバックエンドを有効にする。実デバイスとして、dsysmon プロセスの標準入出力が使用される。ユーザ OS 側では、/dev/hvc0 がフロントエンドのデバイスノードとなる。

−−tunnet=<ipaddr>[:<macaddr>]

virtio_net デバイスのバックエンドを、host-to-host モードで有効にする。監視 OS カーネルの /dev/tun デバイスを利用し、監視 OS 側に作成される TAP インタフェース “tap*N*” と、ユーザ OS 側に新たにフロントエンドとして作成されるネットワークインタフェース “eth*N*” を一対一で直接接続する。監視 OS 側の TAP デバイスには、<ipaddr> で指定された IP アドレスが付与される。

省略可能オプションの <macaddr> を用いて、フロントエンドのインタフェースに設定する MAC アドレスを指定することができる。

−−tunnet=bridge:<bridgename>[:<macaddr>]

virtio_net デバイスのバックエンドを、bridge モードで有効にする。監視 OS カーネルの /dev/tun デバイスを利用し、監視 OS 側に別途用意したブリッジデバイス <bridgename> を、ユーザ OS 側に新たにフロントエンドとして作成されるネットワークインタフェース “eth*N*” に接続する。ユーザ OS 側のフロントエンドから、ブリッジ機能を経由して外部と通信することができる。

省略可能オプションの <macaddr> を用いて、フロントエンドのインタフェースに設定する MAC アドレスを指定することができる。

−−block=<devname>

virtio_blk デバイスのバックエンドを有効にする。<devname> には、実デバイスとしてアクセスするデバイスノードまたはファイルを指定する。ユーザ OS 側では、/dev/vda0 がフロントエンドのデバイスノードとなる。<devname> には /dev/sda など、ディスク装置全体を表すブロックデバイスノードを指定するのが一般的だが、通常ファイルを指定することも可能である。

5.2.3　FoxyKBD オプション

−−fkbd

FoxyKBD アプリケーションを有効化するアプリケーションセレクタ。以下に示す FoxyKBD 固有オプションは、−−fkbd オプションが指定された場合のみ指定可能となる。

−−injection=<interval>[:<start-delay>:<runtime>]

virtio_console デバイスに対する文字列注入の実行間隔、開始までの待ち時間、継続時間をそれぞれ <interval>, <start-delay>, <runtime> として、ミリ秒単位で指定する。<start-delay>, <runtime> については省略することができる。省略した場合、外部から文字列注入の指示が無い限り、文字列注入は行われない。−−injection オプションを指定する場合、同時に −−console オプションを指定してコンソールデバイスの VIRTIO バックエンドを有効にしておく必要がある。

−−netlog<index>[=<filename>]

virtio_net デバイスの <index> で指定するインスタンスに対して、ユーザ OS から出力されるデータの転送量をタイムスタンプと共に <filename> に記録する。<filename> が指定されなかった場合、受け取ったデータは FIFO に書かれた後すぐさま破棄される。<index> を省略することも可能であり、その場合は 0 が指定されたとみなされる。−−netlog オプションを指定する場合、同時に −−tunnet オプションを指定してネットワークデバイスの VIRTIO バックエンドを有効にしておく必要がある。

−−hddlog<index>[=<filename>]

virtio_blk デバイスの <index> で指定するインスタンスに対して、ユーザ OS から出力されるデータの転送量をタイムスタンプと共に <filename> に記録する。<filename> が指定されなかった場合、受け取ったデータは外部制御用 FIFO に書き込まれた後すぐさま破棄される。<index> を省略することも可能であり、その場合は 0 が指定されたものとみなされる。−−hddlog オプションを指定する場合、同時に −−block オプションを指定してブロックデバイスの VIRTIO バックエンドを有効にしておく必要がある。

5.2.4　RootkitLibra オプション

−−rtkl

RootkitLibra アプリケーションを有効化するアプリケーションセレクタ。以下に示す RootkitLibra 固有オプションは、−−rtkl オプションが指定された場合のみ指定可能となる。

RootkitLibra アプリケーションは、−−rtkl−interval オプションで指定された時間間隔でユーザ OS のファイル隠蔽・改竄状態を確認し、隠蔽・改竄を検知した場合に標準出力にメッセージを表示する。

なお、RootkitLibra アプリケーションを正しく動作させるためには、−−tunnet オプションを指定してネットワークデバイスの VIRTIO バックエンドを有効化し、ユーザ OS カーネルが参照するルートファイルシステムが NFS マウントされるように設定しておく必要がある。

−−rtkl-interval=<interval>

ユーザ OS によるファイルの隠蔽・改竄状態を、定期的に確認するための時間間隔をミリ秒単位で指定する。

5.2.5 LMS オプション

−−lms

LMS アプリケーションを有効化するアプリケーションセレクタ。以下に示す LMS 固有オプションは、−−lms オプションが指定された場合のみ指定可能となる。

LMS アプリケーションは、−−lms−interval オプションで指定された時間間隔でユーザ OS のプロセス隠蔽状態を確認し、隠蔽を検知した場合に標準出力にメッセージを表示する。

−−lms-interval=<interval>

ユーザ OS によるプロセスの隠蔽状態を、定期的に確認するためのインターバル時間をミリ秒単位で指定する。

−−lms-loose=<count>

隠蔽プロセス検出時にユーザ OS のカーネルモード移行を許容する閾値を指定する。デフォルトは 5 となっており、5 回までの移行を許容する。

−−init-task=<addr>

ユーザ OS カーネルにおける "init_task" シンボルのカーネル仮想アドレスを指定する。−−init−task オプションが指定されない場合、dsysmon のコンパイル時に取得した値をデフォルトとして使用する。同一の構築ディレクトリで同時にコンパイルされたユーザ OS カーネルと dsysmon を組み合わせて使用する限り、−−init−task オプションを指定する必要はない。

−−per-cpu-rq=<addr>

ユーザ OS カーネルにおける "per_cpu__runqueues" シンボルのカーネル仮想アドレスを指定する。−−per−cpu−rq オプションが指定されない場合、dsysmon のコンパイル時に取得した値をデフォルトとして使用する。同一の構築ディレクトリで同時にコンパイルされたユーザ OS カーネルと dsysmon を組み合わせて使用する限り、−−per−cpu−rq オプションを指定する必要はない。

5.3 VIRTIO 機能の利用方法

本節では、D-System Monitor における VIRTIO 機能の具体的な利用方法を説明する。

5.3.1 virtio_console の利用方法

virtio_console デバイスのバックエンドは、dsysmon に "−−console" オプションを指定することで有効となる。バックエンドの実デバイスは dsysmon を起動した際の標準入出力が利用されるため、監視 OS 側での特別な設定などは不要である。ユーザ OS 側のフロントエンドとしては、/dev/hvc0 デバイスが使用される。

ユーザ OS を自動起動し、virtio_console デバイスを有効にするには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# dsysmon -s --console
```

ユーザ OS 起動後、virtio_console デバイスを用いてログインセッションを開始するには、ユーザ OS 側のシェルにて以下のコマンドを実行する。

```
# /sbin/getty -8 38400 hvc0
```

ユーザ OS のログインセッションが開始されると、dsysmon を起動した監視 OS の TTY 端末に、以下のようにログインプロンプトが表示される。

```
Ubuntu 10.04.3 LTS dv86-1.soum.co.jp hvc0

dv86-1.soum.co.jp login: tom
Password:
Last login: Fri Aug 19 19:31:13 JST 2011 on tty1
Linux dv86-1.soum.co.jp 2.6.32.28+dv86-user #3 Mon Aug 22 05:52:41 JST 2011 \
i686 GNU/Linux Ubuntu 10.04.3 LTS

Welcome to Ubuntu!
 * Documentation:  https://help.ubuntu.com/

Warning: no access to tty (Inappropriate ioctl for device).
Thus no job control in this shell.
%
```

5.3.2 virtio_net の利用方法

virtio_net デバイスのバックエンドは、dsysmon に "−−tunnet" オプションを指定することで有効となる。バックエンドの実デバイスは動作モードにより異なる。ユーザ OS 側のフロントエンドとしては、自動的に "eth*N*" インタフェースが割り当てられる。

- host-to-host モードでは、ユーザ OS の仮想ネットワークインタフェース eth*N* は、監視 OS 側に自動的に生成される tap*N* インタフェースと直接接続される。
- bridge モードでは、監視 OS 側に別途作成するブリッジデバイス br*N* を用いて、ユーザ OS の仮想ネットワークインタフェース eth*N* を、監視 OS の物理的なネットワークにブリッジする。

5.3.2.1 host-to-host モードの利用方法

host-to-host モードでは、監視 OS 側の実デバイスは使用されない。ユーザ OS のネットワークインタフェース eth*N* は、監視 OS のネットワークインタフェース tap*N* と直接一対一で接続される。監視 OS の tap*N* は、dsysmon 起動時に自動的に有効にされ、−−tunnet オプションで指定された IP アドレスが設定される。ユーザ OS の eth*N* は、別途設定されていない限り自動的には有効にならないため、必要であれば手動にて IP アドレスの付与などを行なう必要がある。

ユーザ OS を自動起動し、virtio_net デバイスを host-to-host モードで有効にするには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# dsysmon -s --tunnet=192.168.1.100
```

監視 OS において別途 /dev/tun デバイスを利用していなければ、dsysmon により作成されるネットワークインタフェースは "tun0" となる。tun0 には IP アドレス 192.168.1.100 が付与される。

ユーザ OS では、ネットワークインタフェース eth*N* が生成される。ターゲット実機にはネットワークインタフェースが 2 つ装備されているため、新たに作成されるネットワークインタフェースは "eth2" となる [7]。

[7] ユーザ OS 側にネットワークインタフェースを 2 つとも割り当てた場合。D-Visor86 VMM のオプションにて、割り当てを変更することもできる。

ユーザ OS において eth2 に IP アドレス 192.168.1.99 を付与して初期化することで、監視 OS の tun0 と相互通信が可能となる。

```
# ifconfig eth2 inet 192.168.1.99 up
# ping 192.168.1.100
PING 192.168.1.100 (192.168.1.100): 56(84) bytes of data.
64 bytes from 192.168.1.100: icmp_seq=0 ttl=64 time=5.196ms
64 bytes from 192.168.1.100: icmp_seq=1 ttl=64 time=0.556ms
64 bytes from 192.168.1.100: icmp_seq=2 ttl=64 time=0.528ms
......
```

5.3.2.2 bridge モードの利用方法

bridge モードでは、dsysmon を起動する前にブリッジデバイスの設定を行なう必要がある。監視 OS にて brctl コマンドを用いてブリッジデバイスを作成し、物理ネットワークインタフェースに割り当てておく。ここで、監視 OS において物理ネットワークインタフェースが存在することが必要となる。D-Visor86 VMM の起動オプションを調整し、物理ネットワークインタフェースが監視 OS に割り当てられるようにしておく。

```
# brctl addbr br0
# brctl addif br0 eth0
# ifconfig br0 up
# ifconfig eth0 0.0.0.0 up
[   10.818027] device eth0 entered promiscuous mode
[   13.571169] e1000e: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX/TX
[   13.575493] br0: port 1(eth0) entering learning state
[   28.578992] br0: port 1(eth0) entering forwarding state
```

上記実行例では、監視 OS にてブリッジデバイス br0 を作成し、物理ネットワークインタフェース eth0 とブリッジを構成している。ブリッジとして使用される物理ネットワークインタフェースには、有効な IP アドレスを付与してはいけないことに注意する。なお、上記例と同様の処理を行なうスクリプトが、最小構成 initrd イメージの /dv86-scrips/setup_bridge として用意されている。

ブリッジデバイスを作成した後、ユーザ OS を自動起動し、virtio_net デバイスを bridge モードで有効にするには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# dsysmon -s --tunnet=bridge:br0
[  111.645491] device tap0 entered promiscuous mode
[  111.650986] br0: port 2(tap0) entering learning state
[  126.652597] br0: port 2(tap0) entering forwarding state
```

ユーザ OS では、ネットワークインタフェース eth*N* が生成される。ターゲット実機にネットワークインタフェースが割り当てられていないため、新たに作成されるネットワークインタフェースは “eth0” となる。ユーザ OS において eth0 に IP アドレス 172.19.86.1 を付与して初期化することで、監視 OS の br0 および eth0 を経由してターゲット実機外部との通信が可能となる。

```
# ifconfig eth0 inet 172.19.86.1 up
# ping 172.19.2.61
PING 172.19.2.61 (172.19.2.61): 56(84) bytes of data.
64 bytes from 172.19.2.61: icmp_seq=0 ttl=64 time=2.87ms
64 bytes from 172.19.2.61: icmp_seq=1 ttl=64 time=0.710ms
64 bytes from 172.19.2.61: icmp_seq=2 ttl=64 time=0.682ms
......
```

5.3.3 virtio_blk の利用方法

virtio_blk デバイスのバックエンドは、dsysmon に “−−block” オプションを指定することで有効となる。バックエンドの実デバイスは、−−block オプションで指定したデバイスノードまたはファイルとなる。ユー

ザ OS 側のフロントエンドとしては、/dev/vda*N* デバイスが使用される。ここで、*N* は実デバイスに設定されているパーティションを表す。

ユーザ OS を自動起動し、virtio_block デバイスを有効にするには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# dsysmon -s --block=/dev/sda
```

上記例では、バックエンドの実デバイスとして、監視 OS からアクセス可能なディスクデバイス /dev/sda を指定している。/dev/sda にパーティションが設定されている場合 — つまり /dev/sda1, /dev/sda2, /dev/sda5 などが設定されている場合は、フロントエンド側でも同様のパーティション情報が検出され、/dev/vda1, /dev/vda2, /dev/vda5 などが利用可能となる。

ユーザ OS 用に Ubuntu 10.04 をインストールしたディスク (/dev/sda) を、監視 OS 側から virtio_blk の実デバイスとして供給する場合、ユーザ OS のカーネル起動オプションとして "root=/dev/vda1" を指定する。ユーザ OS カーネル起動時のメッセージの抜粋を以下に示す。

```
......
[    0.000000] Kernel command line: root=/dev/vda1 ro pci=veidt_busset=1 \
nosep idle=halt reservetop=67108864
......（途中省略）......
[    0.454171] virtio: device 0 virtqueue 0 at 0x24001000 IRQ 7
[    0.454808]  vda: vda1 vda2 < vda5 >
......
[    9.454741] EXT4-fs (vda1): mounted filesystem with ordered data mode
```

5.4 FoxyKBD アプリケーション

FoxyKBD アプリケーションでは、ユーザ OS のコンソールに対して文字列を強制的に注入すると同時に、ネットワークおよびディスクに対する書き込み転送量の記録を取得する。これらの動作はキーロガーの存在を検知するために行なわれるが、FoxyKBD には実際に検知処理を行なうアルゴリズムは実装されていない。得られたログを別途統計処理することで判断することとなっている。したがって、FoxyKBD は動作状態においてメッセージなどを表示することはない。

FoxyKBD の文字列注入機能は、−−injection オプションにより指定する。パラメータとして、文字列注入の時間間隔と、注入開始までの待ち時間、および注入動作の合計継続時間をそれぞれ':' 文字で区切って与える。たとえば、注入開始まで 5 秒間待った後、100 ms 間隔ごとに 10 秒間注入を行なう場合、オプションパラメータは "100:5000:10000" となる。

ネットワークおよびディスクの書き込み転送量ログは、−−netlog および −−hddlog オプションを指定することで有効となる。いずれも、パラメータとしてログファイル名を指定する。ログファイルはテキスト形式で、転送が発生するごとに次のような行が追記される。

<timestamp> <length>

<timestamp> は、dsysmon の内部で計測した時刻をミリ秒の単位で表した数値で、処理する際は前後の行からの差分として扱うことを想定している。<length> は当該時刻に発生した書き込み転送データ量をバイト単位で表したものである。

FoxyKBD の具体的な起動例として、以下のケースを想定する。

- dsysmon からユーザ OS を自動起動する。
- virtio_console バックエンドを有効にする。
- virtio_net バックエンドを bridge モードで有効にする。

- virtio_blk バックエンドを /dev/sda を実デバイスとして有効にする。
- 待ち時間 5 秒、注入間隔 100 ms、継続時間 10 秒としてコンソールへの文字列注入を行なう。
- ネットワーク転送量ログを “net.log” ファイルに記録する。
- ディスク転送量ログを “hdd.log” ファイルに記録する。

上記のケースで FoxyKBD を起動する具体的なコマンド指定は以下のようになる。

```
# brctl addbr br0
# brctl addif br0 eth0
# ifconfig br0 up
# ifconfig eth0 0.0.0.0 up
# dsysmon -s --console --tunnet=bridge:br0 --block=/dev/sda \
          --injection=5000:100:10000 --netlog=net.log --hddlog=hdd.log
```

5.5 RootkitLibra アプリケーション

RootkitLibra アプリケーションは、ユーザ OS がアクセスするファイルシステムを NFS マウントし、仮想ネットワークデバイスを経由させることで NFS パケットの取得・解析を行ない、同時にファイルアクセスに関するシステムコールの情報を取得・解析することで、ファイルの改竄・隠蔽を検知するものである。したがって、RootkitLibra を動作させるためには、virtio_net デバイスのバックエンドを有効にする必要がある。また、システムコール情報を取得するために、DXFEED 機能を用いたサポートプログラム “rtkl-collect” をユーザ OS 側で動作させる必要がある。

“rtkl-collect” プログラムのソースコードは、D-Visor86 ソースツリーの $HOME/tools/rtkl-collect 以下にあり、D-Visor86 の構築時に同時にコンパイルされ、最小構成 initrd イメージに “/dsm-tools/rtkl-collect” として含まれる。“rtkl-collect” をユーザ OS の下で動作させるように、ユーザ OS のファイルシステムなどにあらかじめコピーしておく必要がある。

RootkitLibra では、−−rtkl−interval オプションにより、一定時間ごとに収集したファイル・ディレクトリ情報を確認する。ここで、ファイルサイズの不一致が検知された場合、以下のようなメッセージが表示される。

[*<timestamp>*] RTKL: file size mismatch, i-node I (nfs:N != sys:S)

ここで、*<timestamp>* は dsysmon 起動からの経過秒数を、I は不一致が検知されたファイルの i-node 番号を、N は NFS 由来の trusted view によるファイルサイズを、S はシステムコール由来の untrusted view によるファイルサイズ数を表す。

また、ディレクトリエントリ数の不一致が検知された場合は、以下のようなメッセージが表示される。

[*<timestamp*] RTKL: number of directory entry mismatch i-node I (nfs:N != sys:S)

ここで、*<tmiestamp>* は dsysmon 起動からの経過秒数を、I は不一致が検知されたディレクトリの i-node 番号を、N は NFS 由来の trusted view によるエントリ数を、S はシステムコール由来の untrusted view によるエントリ数を表す。

ユーザ OS を自動起動し、virtio_net デバイスを bridge モードで有効にし、また RootkitLibra アプリケーションの検知間隔を 5 秒 (5000 ms) として動作させるには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# brctl addbr br0
# brctl addif br0 eth0
# ifconfig br0 up
# ifconfig eth0 0.0.0.0 up
# dsysmon -s --tunnet=bridge:br0 --rtkl --rtkl-interval=5000
```

ユーザ OS 起動後、検査対象とするパーティションを NFS マウントし、サポートプログラムである “rtkl-collect” を動作させるには、ユーザ OS のシェルより以下のように操作する。

```
# ifconfig eth0 inet 192.168.86.1 up
# mount 192.168.86.1:/export /mnt
# rtkl-collect -r /mnt
```

5.6 LMS アプリケーション

LMS アプリケーションは、D-System Monitor が提供するユーザ OS カーネルメモリ参照機能、およびカーネルモード移行検出機能のみを利用して動作する。LMS の動作はユーザ OS の仮想デバイスには依存していないため、dsysmon にて VIRTIO バックエンドを有効にする必要はない。また、LMS を動作させるために、何らかの事前設定なども不要である。

−−lms−interval オプションで指定された時間間隔ごとに、ユーザ OS のカーネルメモリを参照し、プロセスの実行キューとプロセスリストを探索し、プロセスの隠蔽を検出する。隠蔽を検知した場合、標準出力に以下のメッセージが表示される。

[<*timestamp*>] LMS: found hidden process, PID *P* [<*proc-name*>]

ここで <*timestamp*> は dsysmon 起動からの経過秒数を、*P* は隠蔽されているプロセスのプロセス ID を、<*proc-name*> は PID*P* を持つプロセス名を表している。

ユーザ OS を自動起動し、LMS アプリケーションを検知間隔を 5 秒 (5000 ms) として有効にするには、監視 OS のシェルより dsysmon を以下のように起動する。

```
# dsysmon -s --lms --lms-interval=5000
```

6　GUI ツール — dsm-gui.py

D-System Monitor では、dsysmon の 3 つのアプリケーションをより簡便に利用するための GUI ツール "dsm-gui.py" を用意している。dsm-gui.py は Python 言語で記述されているため、動作させるためには Python インタプリタを必要とする。

dsm-gui.py は、dsysmon の協調モードで作成される外部制御用 FIFO を通じて、以下に示す制御および表示を行なう。

- FoxyKBD のネットワーク、ディスク書き込みログのグラフ表示
- FoxyKBD の文字列インジェクションの開始と停止
- FoxyKBD の文字列インジェクションのタイミング制御
- RootkitLibra の rootkit 警告の表示
- RootkitLibra の rootkit チェック開始と停止
- RootkitLibra のファイルシステム整合性情報を保持している内部テーブルのクリア
- LMS の rootkit 警告の表示
- LMS のチェックの開始と停止
- LMS のチェックのタイミング制御

dsm-gui.py の GUI ウィンドウ表示例を図 2 に示す。

6.1　dsm-gui.py オプション一覧

dsm-gui.py に指定できるオプションの一覧を以下に示す。

−d <coop-dir>
: dsysmon プロセスが協調モードで作成する外部制御用 FIFO が含まれるディレクトリを指定する。このオプションは必須であり、指定されない場合はエラーとなる。

−g <graph-style>
: FoxyKBD 用の転送量グラフの表示スタイルを指定する。"bar" が指定された場合は棒グラフを、"plot" が指定された場合は折れ線グラフを用いる。デフォルトでは折れ線グラフとなる。

−i <interval>
: FoxyKBD 用の転送量グラフの横軸目盛の更新間隔を秒単位で指定する。デフォルトでは横軸目盛は 2 秒間隔となる。

−l <device>
: FoxyKBD 用の転送量グラフのうち、左側の領域に表示するデバイスを指定する。"netlog<index>" が指定された場合は virtio_net デバイスの転送ログが、"hddlog<index>" が指定された場合は virtio_blk デバイスの転送ログが表示される。<index> は、それぞれのデバイス種類におけるインスタンス番号を表す。<index> は省略可能であり、省略された場合は 0 が指定されたものとみなされる。–l オプションが指定されなかった場合は、デフォルトとして "netlog" が選択される。

−r <device>
: FoxyKBD 用の転送量グラフのうち、右側の領域に表示するデバイスを指定する。"netlog<index>" が指定された場合は virtio_net デバイスの転送ログが、"hddlog<index>" が指定された場合は virtio_blk デバイスの転送ログが表示される。<index> は、それぞれのデバイス種類におけるインスタンス番号を表す。<index> は省略可能であり、省略された

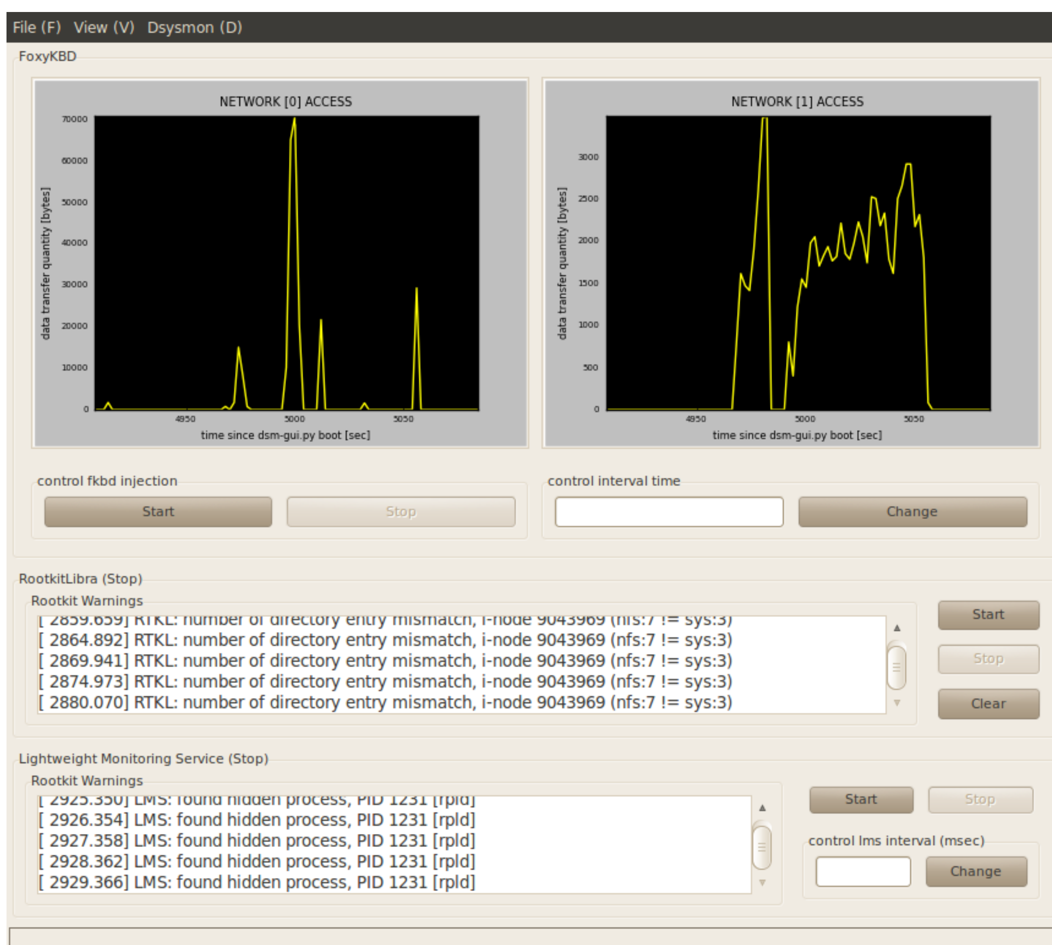

図 2: dsm-gui.py のウィンドウ表示例

場合は 0 が指定されたものとみなされる。–r オプションが指定されなかった場合は、デフォルトとして “hddlog” が選択される。

–s <grid> FoxyKBD 用の転送量グラフに表示する横軸のグリッド数を指定する。デフォルトは 90 である。

–v dsm-gui.py のバージョンを表示する。

–h ヘルプメッセージを表示する。

6.2 dsm-gui.py の操作方法

dsm-gui.py のウィンドウは、大きく以下の 4 つの領域より構成されている。

1) メニュー領域
2) FoxyKBD 用表示領域
3) RootkitLibra 用表示領域
4) LMS 用表示領域

以下、それぞれの領域について操作方法などを説明する。

6.2.1　メニュー領域

メニュー領域には、dsm-gui.py を操作するためのメニュー項目が含まれている。メニュー項目の構成とその機能を以下に示す。

File　GUI ツール全般について制御を行うメニューである。Alt+F キーがショートカットキーとして登録されている。

- **Quit**　GUI ツールを終了する。Ctrl+Q キーがショートカットキーとして登録されている。

View　GUI の表示に関する制御を行うメニューである。Alt+V キーがショートカットキーとして登録されている。

- **Fullscreen**　GUI ツールをフルスクリーン表示状態にする。既にフルスクリーン表示状態であれば、フルスクリーン表示状態を解除する。F11 キーがショートカットキーとして登録されている。

Dsysmon　D-System Monitor の動作に関する制御を行うメニュー項目である。Alt+D キーがショートカットキーとして登録されている。

- **Start User OS**
  ユーザ OS がまだ起動していない場合は、ユーザ OS を起動する。ユーザ OS が既に起動されている場合、この項目はグレイアウトされて無効化される。Ctrl+S キーがショートカットキーとして登録されている。
- **Clear RTKL warning**
  RootkitLibra の警告表示領域 “Rootkit Warnings” の内容をクリアする。
- **Clear LMS warning**
  LMS の警告表示領域 “Rootkit Warnings” の内容をクリアする。
- **Clear ALL warning**
  RootkitLibra と LMS 両者の警告表示領域 “Rootkit Warnings” の内容を同時にクリアする。
- **Clear ALL graphs**
  FoxyKBD の全てのグラフ表示をクリアする。

6.2.2　FoxyKBD 用表示領域

ウィンドウの上方に位置する、“FoxyKBD” と表題が付けられた領域である。2 つのグラフと 3 つのボタン、1 つのテキスト入力領域を持つ。

グラフは、横軸が dsm-gui.py を起動してからの秒数、縦軸がデータ転送量を示す。左側のグラフには、dsm-gui.py の “–l” オプションで指定されたデバイスの転送量を表示する。デフォルトでは netlog（ネットワークへの書き込みアクセス）が選択される。右側のグラフには、dsm-gui.py の “–r” オプションで指定されたデバイスの転送量を表示する。デフォルトでは hddlog（ディスクへの書き込みアクセス）が選択される。

3 つのボタンと 1 つのテキスト入力領域は、FoxyKBD の文字列インジェクション機能を制御するために使用する。

Start FoxyKBD の文字列インジェクションを開始する。dsm-gui.py 起動時には文字列インジェクションは停止状態となっている。

Stop FoxyKBD の文字列インジェクションを停止する。

Change テキスト領域に入力された数値（ミリ秒単位）を文字列インジェクションの間隔として設定する。

6.2.3 RootkitLibra 用表示領域

ウィンドウの中央に位置する、“RootkitLibra” と表題が付けられた領域である。1 つの表示領域と 3 つのボタンを持つ。

左側に配置され、“Rootkit Warnings” と表題が付けられた表示領域には、RootkitLibra がファイルの隠蔽やファイルサイズの改竄を検出した際に、警告メッセージが表示される。

右側にある 3 つのボタンは、RootkitLibra のファイルシステム整合性情報に関する制御を行うために使用する。

Start RootkitLibra によるファイルシステム整合性情報のチェックを開始する。dsm-gui.py 起動時にはチェック機能は停止状態となっている。

Stop RootkitLibra によるファイルシステム整合性情報のチェックを停止する。

Clear RootkitLibra が内部に持つ、ファイルシステム整合性情報のテーブルをクリアする。

6.2.4 LMS 用表示領域

ウィンドウの下方に位置する、“Lightweight Monitoring Service” と表題が付けられた領域である。1 つの表示領域と 3 つのボタン、1 つのテキスト入力領域を持つ。

左側に配置され、“Rootkit Warnings” と表題が付けられた表示領域には、LMS がプロセスの隠蔽を検出した際に、警告メッセージが表示される。

右側にある 3 つのボタンと 1 つのテキスト入力領域は、LMS のプロセス検出動作に関する制御を行なうために使用する。

Start LMS によるプロセス検出処理を開始する。dsm-gui.py 起動時には検出処理は停止状態となっている。

Stop LMS によるプロセス検出処理を停止する。

Change テキスト領域に入力された数値（ミリ秒単位）をプロセス検出処理の動作間隔として設定する。デフォルトでは、一秒毎に検出処理が起動されるようになっている。

7　デモ用サンプル rootkit

D-System Monitor では、監視 OS 上で実行される 3 つアプリケーションでユーザ OS カーネルに対する意図的な改竄を動的に検出する。本節では、その各アプリケーショの動作確認用に用意した三つの rootkit の使用法について説明する。

7.1　file_rootkit.ko

file_rootkit.ko は、ファイルの隠蔽、ファイルサイズの改竄を行うカーネルモジュールである。file_rootkit.ko にはファイルの隠蔽、ファイルサイズの改竄をしたいファイルの i-node 番号を登録する。file_rootkit.ko は、カーネルモジュールの登録時に以下のシステムコールを置き換える。置き換えたシステムコールでは登録された i-node 番号を持つファイルを隠蔽したり、ファイルサイズの改竄したりするように動作する。

open(), close(), oldfstat(), fstat(), fstat64(), oldlstat(), lstat(), lstat64(), getdents(), getdents64()

file_rootkit.ko は、以下のコマンドでインストールする事ができる。

```
# insmod file_rootkit.ko
```

ファイルを隠すためには、以下のように /proc/fhide に隠したいファイルの i-node 番号を書き込む。

```
$ echo <i-node number> > /proc/fhide
```

また、隠したファイルを復活するためには、以下のように /proc/fhide に既に隠したファイルの i-node 番号を再度書き込む事で復活できる。

```
$ echo <i-node number> > /proc/fhide
```

ファイルサイズを改竄するためには、/proc/fmodify にファイルサイズを改竄したいファイルの i-node 番号と改竄する値を書き込む。

以下の例では、指定した inode 番号を持つファイルのファイルサイズに 500 を加算する。

```
$ echo <i-node number> +500 > /proc/fmodify
```

以下の例では、指定した inode 番号を持つファイルのファイルサイズに 500 を減算する。

```
$ echo <i-node number> -500 > /proc/fmodify
```

以下の例では、指定した inode 番号を持つファイルのファイルサイズを 500 に設定する。

```
$ echo <i-node number> 500 > /proc/fmodify
```

また、ファイルサイズを改竄したファイルのファイルサイズを元に戻すためには、改竄したファイルの i-node 番号を再度 /proc/fmodify に書き込む事で元に戻す事ができる。

```
$ echo <i-node number> > /proc/fmodify
```

なお、/proc/fhide、/proc/fmodify 共に、cat 等で読み込む事で現在隠している、もしくは、ファイルサイズを改竄しているファイルの i-node 番号一覧を標準出力に出力できる。

7.2 process_rootkit.ko

process_rootkit.ko は、プロセスの隠蔽を行うカーネルモジュールである。process_rootkit.ko は、登録された PID を持つプロセスを隠すように振る舞う。process_rootkit.ko は、読み込み時に /proc 以下のエントリを読む際に使われる関数を置き換える。置き換えられた関数では、登録された PID の /proc 以下のエントリを隠すように動作する。また、隠したい PID を登録する際に、プロセスの一覧を管理しているカーネル内の構造体のリスト task_struct から該当する PID を持つエントリを削除する。

process_rootkit.ko は、以下のコマンドでインストールする事ができる。

```
# insmod process_rootkit.ko
```

プロセスを隠すためには、以下のように /proc/phide に隠したいプロセスの PID を書き込む。

```
$ echo <PID> > /proc/phide
```

process_rootkit.ko の制限事項として、隠したプロセスの復活をすることはできない。プロセスを隠す際に、リスト task_struct から隠したいプロセスのエントリを削除してしまっており、プロセスを可視状態に戻すためにはそのリストに削除したプロセスのエントリを再び入れなければならない。ただし、元に戻す際にプロセスがまだ動作しているか否かを判別する簡易な方法が無いため、制約として隠したプロセスを復活できないようになっている。

7.3 ttyrpld

ttyrpld は、以下の URL で開発されているオープンソースの tty logging daemon である。全ての TTY の入出力を保存する事ができる。詳細な使用方法については本書では割愛する。

http://ttyrpld.sourceforge.net/

ttyrpld では、以下のように "rpld" コマンドを動作させておく事で、TTY に対する入出力を全て保存する事ができる。

```
# rpld
```

入出力先等は全て "rpld" コマンドの設定ファイル、rpld.conf により行う事ができる。rpld.conf には以下の設定を記述する事ができる。

OFMT=<PATH>
TTY の入出力を保存するファイル名のパスを指定する。

また、上記の設定の他に今回のための拡張として、ネットワーク越しにログを送信して保存するために以下の設定を追加してある。これは後述する "rpld_receiver" プログラムに向けて送信される。

NET_ADDR=<IP_ADDR>
TTY 入出力のログを送信する宛先のアドレスを指定する。

NET_PORT=<PORT>
TTY 入出力のログを送信する宛先のポート番号を指定する。

NET_RETRY=<1|0>
TTY 入出力のログを保存する際に、TTY ログの送り先との接続が確立されていなければ、再接続をしに行くかどうかを制御する。この値が 1 ならば再接続を行い、0 ならば再接続を行わない。デフォルトでは 0 となっている。

7.3.1 rpld_receiver

rpld_receiver は、“rpld” より送信される TTY 入出力のログを受け取る TCP サーバプログラムである。
以下のように起動する事で動作する。

```
$ rpld_receiver
```

rpld_receiver に指定できるオプションを示す。

-o <ファイル名>
: 受け取った TTY 入出力のログを指定されたファイルに書き出す。何も指定されないと、受け取ったログはそのまま捨てられる。

-p <ポート番号>
: rpld_receiver が待ち受けるポート番号を指定する。このオプションが指定されないと、10080 番で TTY 入出力のログを待ち受ける。

-v 冗長メッセージを有効とする。

-h ヘルプメッセージを表示する。

8　デモ実行手順

ターゲット実機にてデモシナリオを実行する際の手順は、概ね以下のようになる。

1) シナリオに応じて VMM と監視 OS を起動
2) 監視 OS において、dsysmon を所定のオプションにて起動
3) ユーザ OS が自動的に起動
4) デモシナリオに応じてユーザ OS、または監視 OS にて操作

8.1　デモシナリオ概要

想定しているデモの概要について説明する。デモは二種類の動作状態を想定し、それぞれにてデモを実行する。

シナリオ A: 通常動作状態

ユーザ OS の定常的な運用状態を想定したデモモード。ユーザ OS を VGA/GUI で通常動作させ、監視 OS では initrd を用いて軽微な検査ツール (LMS) のみを動作させる。

シナリオ B: 緊急検査状態

ユーザ OS に何らかの問題が発見された際に、緊急検査を行なうデモモード。ユーザ OS はテキストコンソール状態で動作させ、監視 OS を VGA/GUI で動作させて FoxyKBD、RootkitLibra、LMS のすべての検査ツールを動作させる。

両デモシナリオにおける OS 動作状態と VIRTIO デバイス利用状態の一覧を表 7 および表 8 に示す。

表 7: デモシナリオ A における OS 動作状態と VIRTIO デバイス利用状態

シナリオ A:	CPU#0	CPU#1
カーネル	DV86-User	DV86-Watcher
ルートファイルシステム	HDD (sda1)	initrd
コンソールデバイス	VGA/GUI	ttyS1 (COM2)
VIRTIO デバイス	–	–

表 8: デモシナリオ B における OS 動作状態と VIRTIO デバイス利用状態

シナリオ B:	CPU#0	CPU#1
カーネル	DV86-User	DV86-Watcher
ルートファイルシステム	NFS (virtio_net)	HDD (sdb1)
コンソールデバイス	hvc0 (virtio_console)	VGA/GUI
VIRTIO デバイス	NET0/NET1/CONSOLE	(backend)

8.2 デモシナリオ A

1) ターゲット実機の電源を投入し、Grub のメニューから

   D-System Monitor DEMO scenario A [W:initrd / U:sda1+GUI]

   の項目を選択して起動させる。監視 OS が ttyS1 (COM2) をコンソールとして起動する。ここで監視 OS は initrd で動作しており、COM2 では BusyBox のシェルが動作している。

2) LMS アプリケーションを有効として dsysmon を動作させ、ユーザ OS を起動する。

```
# dsysmon --start --lms
```

3) VGA にてユーザ OS が GUI 状態で起動する。起動後に自動的にプロセス隠蔽 rootkit が動作する。

4) 監視 OS で動作する LMS がユーザ OS の "rpld" プロセスの隠蔽を発見し、コンソール (COM2) に以下のようなメッセージを表示する。

   [12345.678] LMS: found hidden process, PID 1515 [rpld]

なお、dsysmon を上記のオプションで起動するための簡易スクリプト "DEMO-A.sh" が別途用意されている。以下のように起動することにより、デモシナリオ A を容易に開始できる。

```
# /dsm-tools/DEMO-A.sh
```

また、LMS は走行状態にあるプロセスを検出対象としているため、隠蔽されている rpld プロセスがある程度アクティブに動作する状態でないと、検出できない恐れがある。この問題の対策として、rpld プロセスの動作時ににプロセッサを余計に消費させる機能が実装されている。

```
$ echo 1000 > /proc/rpl_loop
```

/proc/rpl_loop に書き込む値は、rpld プロセスがログ書き込み後に余分なループを回る回数を 1000 で割った数値を表す。したがって、上記例のように 1000 を書き込むと、$1000 \times 1000 = 1,000,000$ 回ループを回ることとなる。

8.3 デモシナリオ B

1) ターゲット実機の電源を投入し、Grub のメニューから

   D-System Monitor DEMO scenario B [W:sdb1+GUI / U:NFS+hvc0]

   の項目を選択して起動させる。監視 OS が VGA をコンソールとして GUI モードで起動する。デモ用ユーザでログインし、端末エミュレータを三つ起動する。

2) 端末エミュレータから以下のコマンドを実行し、ユーザ OS 用の rootfs を /export/user-root にマウントする。

```
# mount /dev/sda1 /export/user-root
```

3) 監視 OS の端末エミュレータから、すべてのアプリケーションを有効にして dsysmon を起動する。

```
# /dsm-tools/dsysmon --coop --console \
             --tunnet=192.168.1.87 --tunnet=192.168.2.87 \
             --fkbd --netlog0 --netlog1 --rtkl --lms
```

上記では virtio-net を二つ準備している。ネットワークデバイスや IP アドレスの設定はそれぞれ以下の表 9 のようになる。

表 9: デモシナリオ B の IP アドレス設定

デバイス	監視 OS	ユーザ OS
virtio-net-0	tap0 192.168.1.87	eth0 192.168.1.86
virtio-net-1	tap1 192.168.2.87	eth1 192.168.2.86

nfsroot は virtio-net-0 側のインタフェースで処理される。

4) ユーザ OS で動作する ttyrpld がネットワーク経由で出力するログを受け取るため、監視 OS の別の端末エミュレータから "rpld_receiver" を起動する。

```
# /rootkits/rpld_receiver [-o <log-file>]
```

オプション "-o" にて、受信したログを書き込むファイルを指定できる。ファイルにはコンソールに対する全ての入出力が文字列として記録される。

5) GUI ツールを起動する。まず、動作している dsysmon が作成した外部制御用 FIFO ディレクトリを確認する。

```
# ls /tmp/dsysmon*
dsysmon1855
```

確認したディレクトリを "-d" オプションのパラメータに指定し、GUI ツールを以下のように起動する。

```
# /dsm-tools/dsm-gui.py -l netlog0 -r netlog1 \
          -d /tmp/dsysmon1855
```

上記では、デバイス転送量グラフとして左側に virtio-net-0 を、右側に virtio-net-1 を表示するように指示している。

6) GUI ツールのウィンドウが表示された後、"Dsysmon" メニューより、"Start USER OS" を選択するとユーザ OS が起動し、dsysmon を起動させている端末エミュレータにユーザ OS の起動メッセージが表示される。起動完了後、ログインプロンプトが表示される。

7) GUI ツールより適宜操作を行う。操作に応じて、マルウェアによる不審な挙動が発見された場合は、GUI ツールの表示領域に警告メッセージが表示される。

上記に示した各手順を実行するための簡易スクリプト "DEMO-B-dsm.sh" および "DEMO-B-gui.sh" が別途用意されている。DEMO-B-dsm.sh は上記手順の 2) と 3) を、DEMO-B-gui.sh は 4) と 5) をそれぞれ実行する。端末エミュレータを二つ用意し、それぞれで以下のようにスクリプトを実行することにより、デモシナリオ B を容易に開始できる。

一つ目の端末エミュレータで DEMO-B-dsm.sh スクリプトを実行する。この端末はユーザ OS のコンソールとなる。

```
# /dsm-tools/DEMO-B-dsm.sh
```

二つ目の端末エミュレータで DEMO-B-gui.sh スクリプトを実行する。実行後、GUI ツールのウィンドウが表示される。

```
# /dsm-tools/DEMO-B-gui.sh
```

8.3.1 FoxyKBD の操作

FoxyKBD アプリケーションでは、上部に virtio-net-0（nfsroot を供給）と virtio-net-1（rpld のログを送信）の書き込み転送量が折れ線グラフとして表示される。

"Start" および "Stop" ボタンにて、コンソールに対するインジェクションの開始、停止を指示する。インジェクションは特定の文字列を一定間隔でコンソールデバイスに送り込むことで行われる。インジェクションを行う時間間隔はデフォルトで 300ms に設定されているが、入力フィールドに希望の値 (ms) を入力し、"Change" ボタンを押す事で変更できる [8]。

インジェクションを開始すると、virtio-net-1 の転送量グラフに一定の変化が生じることで、キーロガーの存在を検知できる。

なお、rpld はローカルディスクにもログを書き込んでいるため、結果として nfsroot を供給している virtio-net-0 側の転送量グラフにも変化が生じるが、ユーザ OS カーネルによるキャッシュの影響で、容易に目視確認できるグラフ形状にはならないので注意すること。

8.3.2 RootkitLibra の操作

RootkitLibra アプリケーションは、NFS トラフィックを解析して得られたファイルシステム情報と、特定のシステムコールを解析して得られたファイルシステム情報を比較し、ファイルの隠蔽を検知する。

NFS トラフィックの解析は、デモシナリオ B の起動直後から自動的に実行されているが、システムコールの解析はユーザ OS にて "rtkl-collect" プログラムを実行することで行われる。rtkl-collect に "-r <dir>" オプションを与えて起動することで、ディレクトリ <dir> 以下のファイルシステムツリーに対するシステムコール情報が収集される。

デモシナリオ B では、以下のファイルがユーザ OS 起動時に自動的に隠蔽される状態となっている。

```
/rootkit/rpldev.ko
/rootkit/file_rootkit.ko
/rootkit/process_rootkit.ko
/rootkit/rpld
```

これらのファイルの情報を取得するために、rtkl-collect プログラムをユーザ OS のプロンプトから以下のように実行する。

```
# /dsm-tools/rtkl-collect -r /rootkits
```

"Start" および "Stop" ボタンにて、ファイルシステム不整合の検出処理の開始、停止を指示することができる。開始後、5 秒間隔で検出処理が動作し、不整合が検出されるとその都度メッセージ領域に警告が表示される。なお、検出処理の動作間隔を設定する機能は用意されていない。

ファイルサイズの不整合を検出した場合、以下のように表示される。

[12345.678] RTKL: file size mismatch, i-node XXX (nfs:XXX != sys:XXX)

ディレクトリエントリ数の不整合を検出した場合、以下のように表示される。

[12345.678] RTKL: number of directory entry mismatch, i-node XXX (nfs:XXX != sys:XXX)

[8] ターゲット実機である BX-100n-DC5000-C01 では、インターバルをデフォルトの 300ms のままインジェクションを開始する、高負荷により dsm-gui.py が反応しなくなるため "Stop" ボタンの押下が認識されず、インジェクションを停止させることができなくなる。この場合、"# echo stop > /tmp/dsysmon<PID>/fkbd-ctl" のように dsysmon の FIFO に直接指示を書き込むことで停止させることができる。インターバルを 500ms とした場合は dsm-gui.py も動作し続けることができる

行頭の [12345.678] の部分は、dsysmon が起動してからの経過時間を秒単位で表した数値となる。メッセージが出力されたタイミングを確認することができる。

"Clear" ボタンにて、ファイルシステムの整合性情報を保持している RootkitLibra の内部テーブルを初期化できる。ただし、RootkitLibra の内部テーブルをクリアした場合、それ以降は隠蔽されたファイルが一定期間以上検出されなくなる。RootkitLibra では、隠蔽されたファイルの検出に NFS の READDIR または、READDIRPLUS プロシージャから得られるディレクトリ内エントリ数と rtkl-collect により収集されたシスムコール経由のディレクトリ内エントリ数を利用している。一旦、READDIR または、READDIRPLUS プロシージャが呼び出されると、その情報はユーザ OS のカーネルにキャッシュされ、それ以降はそのキャッシュが参照されるため、これらのプロシージャは呼び出されなくなる。

RootkitLibra の内部テーブルをクリアすると、NFS 経由のディレクトリ内エントリ数も消去されるが、上述のようにユーザ OS のキャッシュが有効の間は READDIR または、READDIRPLUS プロシージャが呼び出されないため、NFS 経由のディレクトリ内エントリ数が不明のままとなり、隠蔽されたファイルが検出できなくなる。

ただし、手元の調査では、ユーザ OS 上で以下のコマンドを実行することで、カーネル内のキャッシュをクリアできることが確認されている。カーネル内のキャッシュをクリアした後は、再び READDIR、または READDIRPLUS プロシージャが発行され、隠蔽されたファイルの検出が可能となる。

```
# echo 3 > /proc/sys/vm/drop_caches
```

8.3.3 LMS の操作

"Start" および "Stop" ボタンにて、LMS の検出処理の開始、停止を指示することができる。開始後、一定間隔でユーザ OS のプロセス状態を監視する。監視処理の実行間隔はデフォルトで 1 秒に設定されているが、入力フィールドに希望の値 (ms) を入力し、"Change" ボタンを押すことで変更できる。

監視処理においてプロセスの隠蔽が検出された場合には、その都度メッセージ領域に以下のメッセージが表示される。

    [12345.678] LMS: found hidden process, PID 1515 [rpld]

行頭の [12345.678] の部分は、dsysmon が起動してからの経過時間を秒単位で表した数値となる。メッセージが出力されたタイミングを確認することができる。

LMS では、隠蔽されているプロセスが走行状態にある場合のみに検出可能となる。デモシナリオ B では、キーロガーのデーモンプロセスである rpld が自動的に隠蔽されるため、ユーザ OS にてコンソールに対する何らかの入出力が行われ、ログの記録が行われる場合に検出されることとなる。

なお、LMS では加えて監視処理の実行中にユーザ OS がカーネルモードに移行しなかった場合にのみ検出成功とするように制限を与えている。この制限は、デモを実行する実機の性能に影響されるため、状況によっては検出できないこともある。

LMS では、検出条件を微調整するための仕組みを二つ実装している。必要に応じてこれらを調整することが望ましい。

1) 前述のように、/proc/rpl_loop に値を書き込むことで rpld の動作時に余分なループを実行させることができる。プロセッサが高速な場合、ユーザ OS において rpld が動作する時間が極めて短くなると、監視 OS による一定間隔の監視動作とタイミングが合わずに検出できなくなると考えられる。この場合、ユーザ OS にて /proc/rpl_loop に適当な値を設定し、余計なループを回らせることで検出される可能性を高めることができる [9]。

[9] ターゲット実機である BX-100n-DC5000-C01 でも、標準の状態ではなかなか隠蔽が検出されない。ユーザ OS にて /proc/rpl_loop に 1000 を設定することで検出頻度が高くなることを確認している

2) 前述のように、LMS は監視処理中にユーザ OS が一度もカーネルモードに移行しない場合を正常検出としている。プロセッサが低速で監視 OS の動作が遅い場合、監視動作の最中にユーザ OS 側でタイマ割り込みなどが発生してしまうと、この制限により正常検出とならない。この制約を緩め、一定の回数以内であれば正常検出として扱うように LMS の動作を変更するオプション “–lms-loose=<count>” が dsysmon に用意されている。<count> には、正常検出として許容するカーネルモード移行の回数を指定する。なお、許容回数はデフォルトでは 5 となっている。