Technology Consulting Company
Research, Development &
Global Standard

ケーススタディ:
BitVisor におけるバイナリBLOB USB ドライバの利用

北村　朋宏／株式会社イーゲル

2013/12/6

BitVisor powered by IGEL

- 導入サポート
  - プラットフォーム選定、動作検証
  - インストール、セキュリティポリシー設定
- 受託カスタマイズ
  - 新たなデバイス対応、新機能実装
- ソリューション開拓
  - USBデバイス隠ぺい（偽装）
  - ICカード連携画面ロック
- オープンソース・コミュニティ活動
  - インストールマニュアル作成
  - Realtek RTL8169ドライバ、ADvisorのマージ

イーゲルの BitVisor との関わり

2007	USB対応 (USBメモリ暗号化、libusb互換API for ICカード)	ATAPI対応 (光学メディア暗号化)
2008		
2009	保護ドメイン (VMM内Ring-3で動作)	
2010	ATA piggyback (VMMからディスクI/O)	サスペンド対応
2011	TPM対応 (Trusted Boot、乱数生成器)	バックグラウンドFDE (OS動作中のFull Disk Encryption)
2012	USBドングル利用 (バイナリライブラリのリンク)	最新VT機能の対応 (性能向上)
2013	UEFI 対応	

□:私が関わったもの

背景

USB ドングル（Rockey6）を BitVisor 内部で利用したい。しかし、ライブラリはソースコード未公開。何か方法はないか？

ライブラリをバイナリ BLOB として BitVisor に取り込めれば利用できる可能性がある。

Rockey6 とは？

Rockey6 は、プロセッサを
内蔵したUSBドングル製品

対象ソフトウェアの重要なロジックをドングル内に移すことでプロテクトを行う。

PCとのインターフェースは、USB1.1、HID デバイスとして認識され、インタラプト転送で通信

Rockey6 アクセス API

Rockey6 デバイスへのアクセスは、以下 API が使われる。

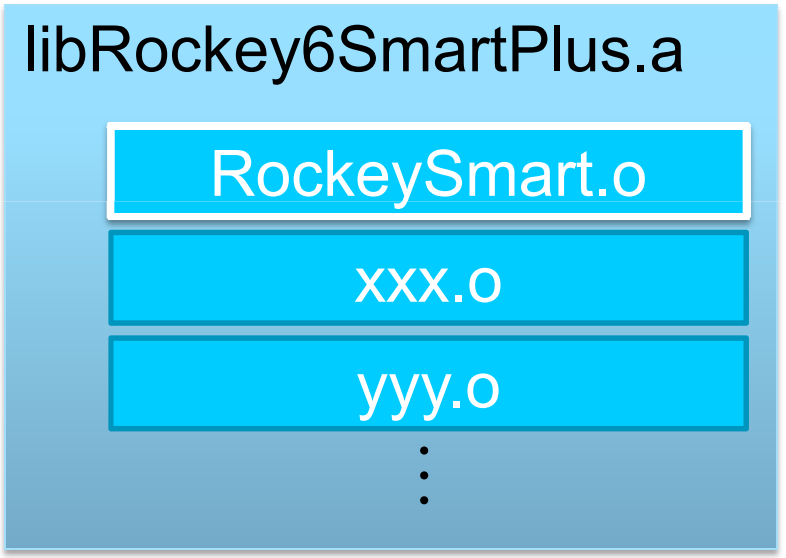
•Rockey6 デバイスアクセスのAPI シーケンス

1. DIC_Find で Rockey6 デバイス有無の検査
2. DIC_Open でデバイスを Open
3. DIC_Command, DIC_Set でデバイスを操作
4. DIC_Close でデバイスを Close

Rockey6 API を含むオブジェクト

Rockey6 API は、Rockey6 SDK 付属ライブラリアーカイブファイル（libRockey6SmartPlus.a）にて提供

アーカイブから `ar -x` でオブジェクトを取り出し、API の含まれるオブジェクトを探した結果、RockeySmart.o に含まれる。



この RockeySmart.o を BitVisor にリンクできれば、BitVisor 内から Rockey6 API を使えるのでは？

Rockey6 ライブラリが依存するシンボル（1）

RockeySmart.o を `objdump -x` し、”UND”で grep
オブジェクト内の未定義シンボルを検索

```
0000000000000000      *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000      *UND*  0000000000000000 memcpy
0000000000000000      *UND*  0000000000000000 memset
0000000000000000      *UND*  0000000000000000 time
0000000000000000      *UND*  0000000000000000 memcmp
0000000000000000      *UND*  0000000000000000 __stack_chk_fail
0000000000000000      *UND*  0000000000000000 pthread_mutex_init
0000000000000000      *UND*  0000000000000000 usb_bulk_write
0000000000000000      *UND*  0000000000000000 usb_bulk_read
0000000000000000      *UND*  0000000000000000 lock
0000000000000000      *UND*  0000000000000000 pthread_mutex_lock
0000000000000000      *UND*  0000000000000000 pthread_mutex_unlock
0000000000000000      *UND*  0000000000000000 unlock
```

（以下略）

Rockey6 ライブラリが依存するシンボル（2）

Rockey6 ライブラリの未定義シンボルを整理し、それらを分類

- C ライブラリ関数　(atol, atoi, memcpy, strlen …)
- pthread 関数 (pthread_mutex_lock …)
- libusb 関数 (usb_init, usb_find_bussees …)
- stack guard (__stack_chk_fail)
- Global offset table (__GLOBAL_OFFSET_TABLE)

これらの未定義シンボルを解決すれば、Rockey6 ライブラリを BitVisor に取り込めるはず。

実装方針（1）

未定義シンボルの解決方法について考慮

Cライブラリ関数、pthread 関数

- BitVisor 内に同等の実装、代用品があるものは、それを利用（sprintf -> snprintf、pthread_mutex_lock -> spinlock_lock）
- 同等品が無い場合は、スタブとして追加実装（atoi、atol）

libusb 関数

- BitVisor は libusb 互換実装を既に持っているため、それを利用

  libusb互換実装は、本来の libusb との差異により問題が生じる可能性があるが。

BitVisor libusb 互換実装

libusb は、USB を利用するためのCライブラリ

BitVisor は、BitVisor の IC カード実装から利用するため、
libusb 互換実装を備えている。

- drivers/usb/usb.h（一部抜粋）

```
/* usb.c */
  usb_dev_handle *usb_open(struct usb_device *dev);
  int usb_close(usb_dev_handle *dev);
  int usb_get_string(usb_dev_handle *dev, int index,
                     int langid, char *buf, size_t buflen);
```

usb_init、usb_find_busses、usb_find_devices、
usb_interrupt_write(read)などがRockey6 API から使われていた。

実装方針(2)

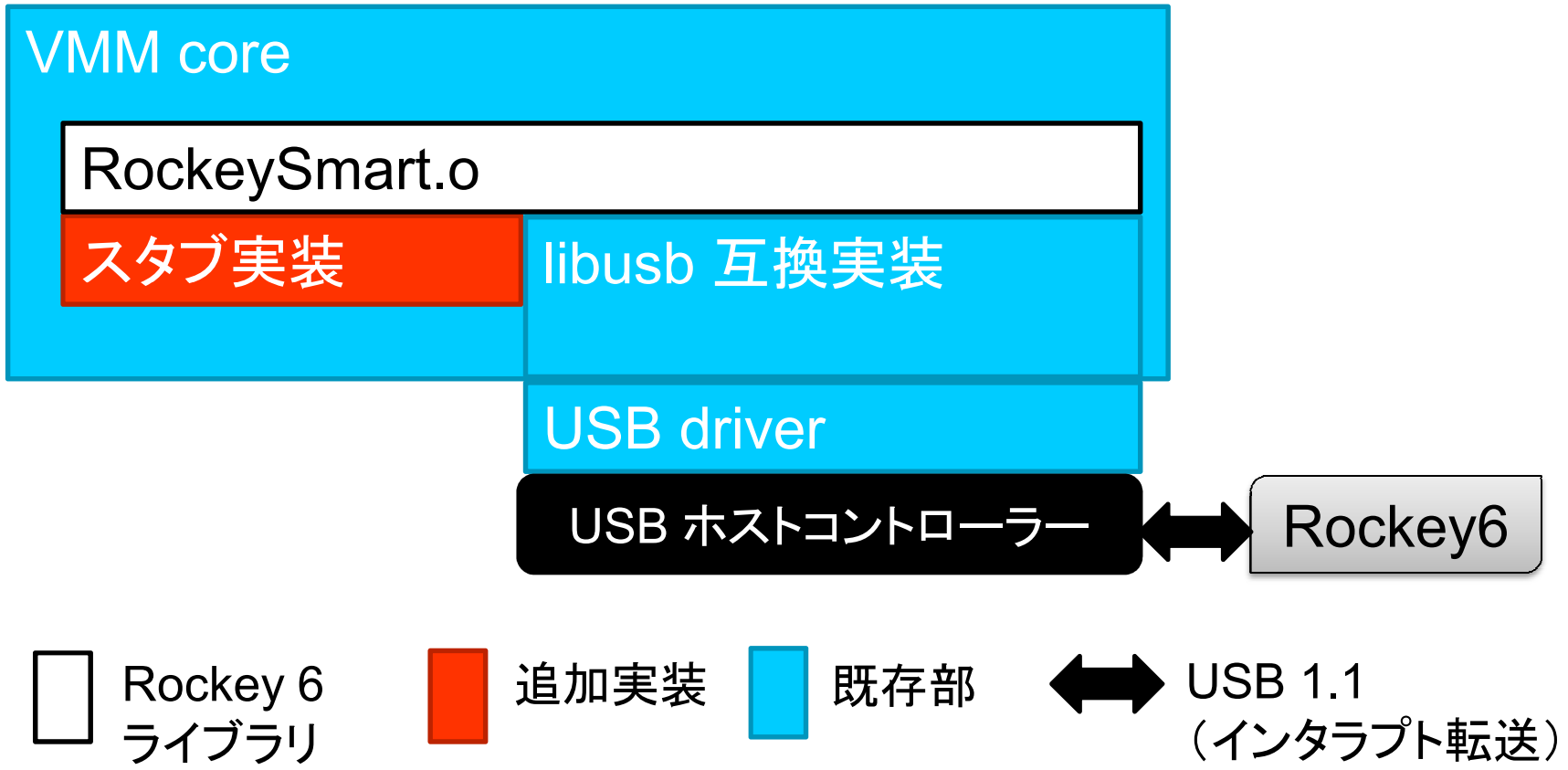
残りの未定義シンボル(赤カッコ)への対応

- C ライブラリ関数　＞　スタブを実装
- pthread 関数　＞　同上
- libusb 関数　＞　既存 libusb 実装を利用
- stack guard (__stack_chk_fail)
- Global offset table (__GLOBAL_OFFSET_TABLE)

stack guard(__stack_chk_fail)はスタブを実装
Global offset table は、リンクエラー要因にはならないため、特に対応策を意識せず進めた。

実装方針(3)

実装構成図


VMM core
RockeySmart.o
スタブ実装
libusb 互換実装
USB driver
USB ホストコントローラー
Rockey6
Rockey 6
ライブラリ
追加実装
既存部
USB 1.1
(インタラプト転送)

実装方針(4)

追加ファイルの構成

```
./bitvisor
  |-- rockey6/
  |     |-- Makefile
  |     |-- RockeySmart.o    (Rockey6 API オブジェクト)
  |     |-- dic32.h          (Rockey6 API ヘッダファイル)
  |     `-- stub.c           (スタブ実装)
  |-- core
        |-- rockey6_samplecodes.c (Rockey6 サンプルプログラム)
        |-- rockey_test.c          (BitVisor デバッグシェルに
                                    rockey コマンドを追加)
```

Rockey6 のサンプルプログラムを core の中に追加し、BitVisor デバッグシェルからサンプルプログラムを起動できるよう実装

Rockey6 サンプル動作イメージ

BitVisor デバッグシェルから Rockey6 プログラムを実行（イメージ）

```
> rockey
rockey>   5
execute test 5
=== rockey6_testcase5 ===
Find Rockey6Smart: 1
Open Rockey6Smart:
FactoryTime  4BF6XXXX
HardSerial   7502XXXX
ShipTime     4BF6XXXX
COSVersion   02.02
```

（サンプル5 は、Rockey6 デバイスのシリアル等のハード情報を取得するもの）

実装終了

一通りスタブ部の実装を終えて、Rockey6 サンプルプログラムを VMM コアにリンクし、動作テスト

DIC_Find してRockey6デバイスの検出を試みると、Find で早速、問題発生

•Rockey6 デバイスアクセスのAPI シーケンス

1. **DIC_Find で Rockey6 デバイス有無の検査**
2. DIC_Open でデバイスを Open
3. DIC_Command, DIC_Set でデバイスを操作
4. DIC_Close でデバイスを Close

DIC_Find 問題調査

DIC_Find の問題

- PageFault で panic

  原因

  1. libusb 互換実装の usb_bus 構造体の差異
  2. Rockey6 オブジェクトの stack guard
  3. Rockey6 オブジェクト GOT セクションの不適切な位置への配置

- Rockey6 デバイスが見つからない

  原因

  - libusb 互換実装へ追加した usb_busses の実装

usb_bus 構造体の差異

DIC_Find で Page fault した原因は、libusb の usb_bus 構造体のメンバ devices の位置が異なっていたため。
libusb の usb_bus 構造体の内容が BitVisor では必要最小限のみ実装されていたので、本家の libusb の構造体と全く配置及びサイズになるようにメンバを追加（太字部分）

```
struct  usb_bus {
-       LIST_DEFINE(usb_busses);
+       LIST_DBL_DEFINE(usb_busses);
+       char filename[4097];                    /* unused */
+       struct usb_device *devices;
+       unsigned int location;                  /* unused */
+       struct usb_device *root_dev;            /* unused */
         struct usb_host *host;
-       struct usb_device *device;
        u8 busnum;
 };
```

stack guard（1）

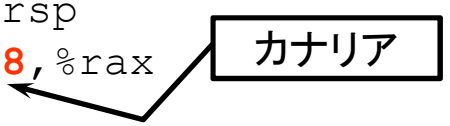
igel

gcc により付加されるスタック破壊検出の仕組み

stack guard は fs レジスタが指すアドレスにスタック破壊検出用のデータ（カナリア）を用意する。BitVisor 本体は stack guard を使っておらず、fs レジスタも未使用だったため、fs レジスタにカナリアを指す値が設定されておらず、page fault した。

- 関数入口でのカナリアの読み出し

```
4288:   55                          push   %rbp
4289:   48 89 e5                    mov    %rsp,%rbp
428c:   53                          push   %rbx
428d:   48 81 ec 88 00 00 00        sub    $0x88,%rsp
4294:   64 48 8b 04 25 28 00        mov    %fs:0x28,%rax
429b:   00 00
429d:   48 89 45 e8         mov    %rax,0xffffffffffffffe8(%rbp)
```



stack guard（2）

- 関数出口でのカナリアのチェック

```
771f: 48 8b 55 e8                mov    0xfffffffffffffe8(%rbp),%rdx
7723: 64 48 33 14 25 28 00       xor    %fs:0x28,%rdx
772a: 00 00
772c: 74 05                      je     7733 <DIC_Command+0x2d75>
772e: e8 00 00 00 00             callq  7733 <DIC_Command+0x2d75>
       772f: R_X86_64_PLT32        __stack_chk_fail+0xfffffffffffffffc
7733: 48 81 c4 a8 04 00 00       add    $0x4a8,%rsp
773a: 5b                         pop    %rbx
773b: c9                         leaveq
```

また、32 bit ビルドだと、stack guard は、gs レジスタを使用する。gs レジスタは BitVisor も利用しており、お互いに干渉しない配慮が必要になる。

gs レジスタ干渉、カナリアの算出を考えると stack guard サポートは面倒なので、Rockey 6 ライブラリから stack guard オプションを除いてビルドして貰うことで対応

GOT セクションの問題（1）

GOT(.got、.got.plt)セクションは、ライブラリがPIC（Position Independent Code）でビルドされている場合に追加される。

Rockey6 ライブラリが PIC であったため、bitvisor.elf のイメージに .got、.got.plt セクションが追加されたが、BitVisor のリンカスクリプトには、これらの配置位置が記述されていなかった。

- bitvisor.elf の objdump –x（.data以降抜粋）

```
 7 .data     00952000 40265000 00265000  00165000  2**12
             CONTENTS, ALLOC, LOAD, DATA
 8 .got      00000030 40bb7000 00bb7000  00ab7000  2**3
             CONTENTS, ALLOC, LOAD, DATA
 9 .got.plt 00000018  40bb7030  00bb7030  00ab7030  2**3
             CONTENTS, ALLOC, LOAD, DATA
```

GOT セクションの問題（2）

結果、ヒープ領域にこれらのセクションが割り当てられ、PageFault を起こしていた。

これは、リンカスクリプトで明示的に .got、.got.plt セクション配置箇所を指定することで解決できたが、最終的には、-fPIC オプションを除いてビルドして貰うことで解決

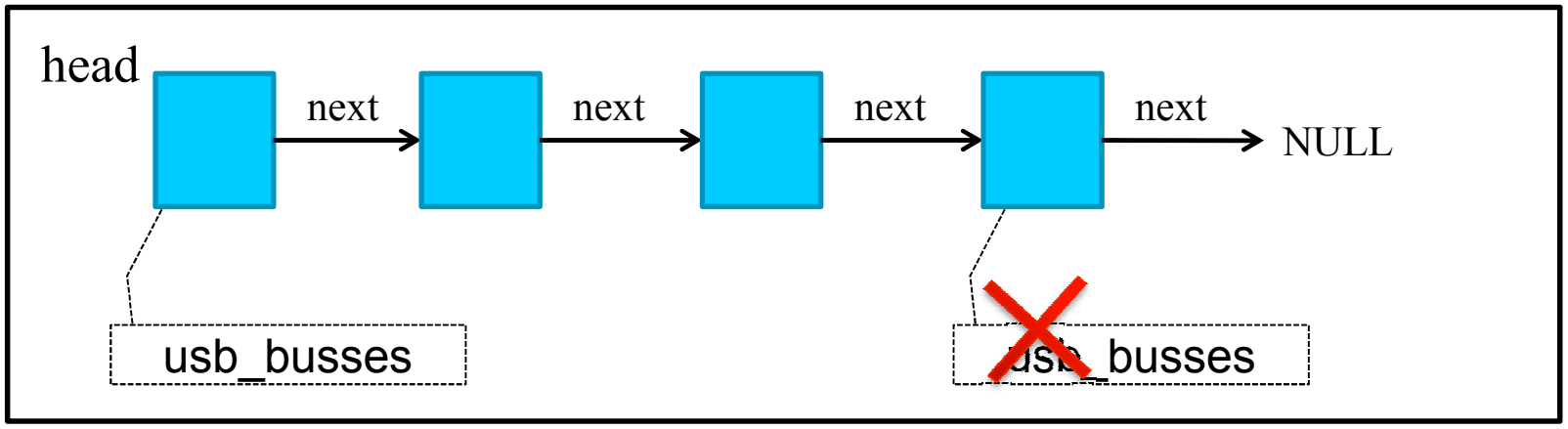
- bitvisor.lds（.data 以降を一部抜粋）

```
.data : AT (phys + (data - head)) {
        data = .;
        *(.data)
        . = ALIGN (8);
        __initfunc_start = .;
        *(.initfunc)
        __initfunc_end = .;
/* }
.bss : AT (phys + (bss - head)) { */
        bss = .;
        *(.bss)
        *(COMMON)
        . = ALIGN (4096);
}
end = .;
```

usb_busses の実装

DIC_Find は、libusb の global 変数 usb_busses を直接参照している。libusb 互換実装には、usb_busses は未実装であるため、追加したが、usb_busses がusb_bus のリスト末尾を指していた。（先頭を指しているのが正しい）

※usb_busses は直接アクセスしないで、libusb の usb_get_busses API を使うべきだが、Rockey6 のライブラリは usb_get_busses を使っていなかった。



DIC_Find 成功

DIC_Find で Rockey6 デバイスが見つかるようになった。DIC_Open は？

•Rockey6 デバイスアクセスのAPI シーケンス

1. DIC_Find で Rockey6 デバイス有無の検査
2. DIC_Open でデバイスを Open
3. DIC_Command, DIC_Set でデバイスを操作
4. DIC_Close でデバイスを Close

DIC_Open 問題発生

DIC_Open でも問題が発生

•Rockey6 デバイスアクセスのAPI シーケンス

1. DIC_Find で Rockey6 デバイス有無の検査
2. **DIC_Open でデバイスを Open**
3. DIC_Command, DIC_Set でデバイスを操作
4. DIC_Close でデバイスを Close

DIC_Open 問題調査

DIC_Open の問題

- PageFault で panic
  原因
    libusb 互換実装の usb_device 構造体の差異
- DIC_Open が終了しない
  原因
    スタブ関数での実装ミスでのループ
    ＞バイナリ BLOB 内での問題を疑い、DIC_Open の動作をデバッグ

DIC_Open デバッグ（1）

レジスタ内容をスタックに積みダンプするコードを作成

バイナリエディットでバイナリ BLOB 中の関数入口で スタックを進める “sub $0x260, %rsp” をデバッグコードを呼ぶ “call hoge”（e8 xx xx xx xx）に置き換える。

```
.global hoge
hoge:
    push %rax
（中略）
    push %r15
    mov  %rsp,%rdi
    call dump_regs
    pop  %r15
（中略）
    pop  %rax
    sub  $0x260-8,%rsp
    jmp *0x260-8(%rsp)
```

```
void
dump_regs(struct stack *stack)
{
  printf ("ret = %lx¥n", stack->ret);
  printf ("rdi = %lx¥n", stack->rdi);
（中略）
}
```

これらのコードで、DIC_Open 内のあるルーチンが呼び出された回数及びレジスタ状況をダンプして把握

← ：呼び出しの流れ

DIC_Open デバッグ（2）

ジャンプ命令 or “nop “ を “int 3” （0xcc: software breakpoint）に置き換え、panic を起こし、スタックの中身を確認

call 命令を nop に置き換え

色々とデバッグした結果、DIC_Open 中の strcpy で実行が止まっていた。strcpy の call を nop にバイナリでエディットし、結果を確認

＞ nop に置き換えると、DIC_Open が正常終了
オープンしたデバイスで、Rockey6 のデバイス情報取得成功

解決

DIC_Open が動作した後、DIC_Open、DIC_Command、DIC_Close まで動作

•Rockey6 デバイスアクセスのAPI シーケンス

1. DIC_Find で Rockey6 デバイス有無の検査
2. DIC_Open でデバイスを Open
3. DIC_Command, DIC_Set でデバイスを操作
4. DIC_Close でデバイスを Close

まとめ

DIC_Find、DIC_Open API が動作するまで、
予想外で不慣れな対応項目もあり、大変だった。

- 構造体のオフセット
- stack guard
- GOT
- トラブル時のバイナリ BLOB デバッグ