Capitolo 4

Analisi, progettazione e realizzazione della piattaforma di emulazione di dispositivi USB

Il sistema USBCheckIn è stato realizzato in via puramente teorica, senza aver mai provato concretamente le difese realizzate. Si è reso, quindi, opportuno creare una piattaforma che permettesse di emulare dispositivi USB arbitrari, consentendo così la simulazione di attacchi reali e certificando l'effettiva efficacia delle difese offerte.

4.1 Analisi dei requisiti

Nella sezione 1.2 è stata presentata l'architettura ad alto livello dello standard USB. Quanto detto è stato utilizzato come linea guida per l'emulazione di un qualsiasi dispositivo USB. Sebbene le strutture utilizzate dallo standard siano note, per poter emulare correttamente un dispositivo si è reso necessario comprendere nei dettagli come host e periferica effettuino, alla connessione, la configurazione sul control endpoint.

4.1.1 Analisi del traffico USB durante la configurazione del dispositivo

Per poter analizzare il traffico USB di una determinata porta durante la connessione di un dispositivo, è stato utilizzato il software **Wireshark**[Wir]: *packet sniffer* generalmente noto come analizzatore del traffico di rete. Tramite il modulo **usbmon**[Zai] di Linux, però, è possibile espandere le funzionalità di Wireshark e collezionare l'intero I/O sul bus USB. Combinando i due strumenti è stato possibile raccogliere le informazioni di qualsiasi pacchetto in transito, individuandone i campi ed interpretandone il significato.

```
55 10.618212    host          3.14.0        USB      64 GET DESCRIPTOR Request DEVICE
56 10.619010    3.14.0        host          USB      82 GET DESCRIPTOR Response DEVICE
57 10.619019    host          3.14.0        USB      64 GET DESCRIPTOR Request CONFIGURATION
58 10.619596    3.14.0        host          USB      73 GET DESCRIPTOR Response CONFIGURATION
59 10.619600    host          3.14.0        USB      64 GET DESCRIPTOR Request CONFIGURATION
60 10.621235    3.14.0        host          USB     123 GET DESCRIPTOR Response CONFIGURATION
61 10.621242    host          3.14.0        USB      64 GET DESCRIPTOR Request STRING
62 10.621583    3.14.0        host          USB      68 GET DESCRIPTOR Response STRING
63 10.621588    host          3.14.0        USB      64 GET DESCRIPTOR Request STRING
64 10.622609    3.14.0        host          USB      90 GET DESCRIPTOR Response STRING
65 10.622615    host          3.14.0        USB      64 GET DESCRIPTOR Request STRING
66 10.623392    3.14.0        host          USB      84 GET DESCRIPTOR Response STRING
67 10.623717    host          3.14.0        USB      64 SET CONFIGURATION Request
68 10.623887    3.14.0        host          USB      64 SET CONFIGURATION Response
```

Figura 4.1: Dump del setup di un dispositivo catturato da Wireshark.

In sostanza, quello che avviene è l'invio di una serie di richieste da parte dell'host mirate a ricevere i descriptor del dispositivo per procedere alla sua identificazione. In ordine, le transazioni analizzate sono:

1. GET DESCRIPTOR (DEVICE)
2. GET DESCRIPTOR (CONFIGURATION)
3. GET DESCRIPTOR (CONFIGURATION + INTERFACE + ENDPOINT)
4. GET STRING (CONFIGURATION) (lista con ID dei linguaggi supportati)
5. GET STRING (MANUFACTURER) (se presente nel device descriptor)
6. GET STRING (PRODUCT) (se presente nel device descriptor)
7. GET STRING (SERIAL NUMBER) (se presente nel device descriptor)

Una volta ricevute correttamente le risposte, le request successive variano in base al tipo di dispositivo connesso. Nel caso di interesse, l'analisi con Wireshark si è focalizzata su due classi di device, HID e mass storage, poichè le difese realizzate si concentrano, per ora, su attacchi con queste periferiche.

Per gli HID, le ulteriori request servono ad ottenere l'HID Report di ogni interfaccia dichiarata dal dispositivo e sono identificate come GET DESCRIPTOR (HID REPORT).

```
71 10.624287    host          3.14.0        USBHID     64 GET DESCRIPTOR Request HID Report
72 10.626234    3.14.0        host          USBHID    118 GET DESCRIPTOR Response HID Report
```

Figura 4.2: Dump della configurazione di un dispositivo HID.

Per le periferiche mass storage, viene richiesto il numero di unità logiche (LUN) presenti, identificato come GET MAX LUN.

```
57 4.802615     host          3.15.0        USBMS      64 GET MAX LUN Request
58 4.803533     3.15.0        host          USBMS      65 GET MAX LUN Response
```

Figura 4.3: Dump della configurazione di un dispositivo mass storage.

Se anche questa fase si conclude correttamente, inizia la comunicazione sugli endpoints standard. Ogni classe di dispositivi codifica il pacchetto diversamente.
I byte di un pacchetto proveniente dal mouse sono i seguenti:

Byte	Descrizione
00h	Tasto premuto
01h	Spostamento X
02h	Spostamento Y
03h	Spostamento Z
04h	Bottoni aggiuntivi

Tabella 4.1: Byte di un pacchetto del mouse.

Mentre la tastiera invia i seguenti byte:

Byte	Descrizione
00h	Modificatore (es. CTRL)
01h	Riservato
02h	Tasto 1
03h	Tasto 2
04h	Tasto 3
05h	Tasto 4
06h	Tasto 5
07h	Tasto 6

Tabella 4.2: Byte di un pacchetto della tastiera.

Per i dispositivi mass storage, i dati scambiati rispecchiano quanto discusso ampiamente nella sezione 3.2, ovvero strutture CBW, dati bulk e CSW.

```
167 4.814360    host      3.15.2    USBMS     95 SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 8)
168 4.814369    3.15.2    host      USB       64 URB_BULK out
169 4.814372    host      3.15.1    USB       64 URB_BULK in
170 4.815553    3.15.1    host      USBMS   4160 SCSI: Data In LUN: 0x00 (Read(10) Response Data)
171 4.815558    host      3.15.1    USB       64 URB_BULK in
172 4.815597    3.15.1    host      USBMS     77 SCSI: Response LUN: 0x00 (Read(10)) (Good)
173 4.816436    host      3.15.2    USBMS     95 SCSI: Test Unit Ready LUN: 0x00
174 4.816451    3.15.2    host      USB       64 URB_BULK out
175 4.816456    host      3.15.1    USB       64 URB_BULK in
176 4.816516    3.15.1    host      USBMS     77 SCSI: Response LUN: 0x00 (Test Unit Ready) (Good)
177 4.816602    host      3.15.2    USBMS     95 SCSI: Read(10) LUN: 0x00 (LBA: 0x0039d780, Len: 8)
```

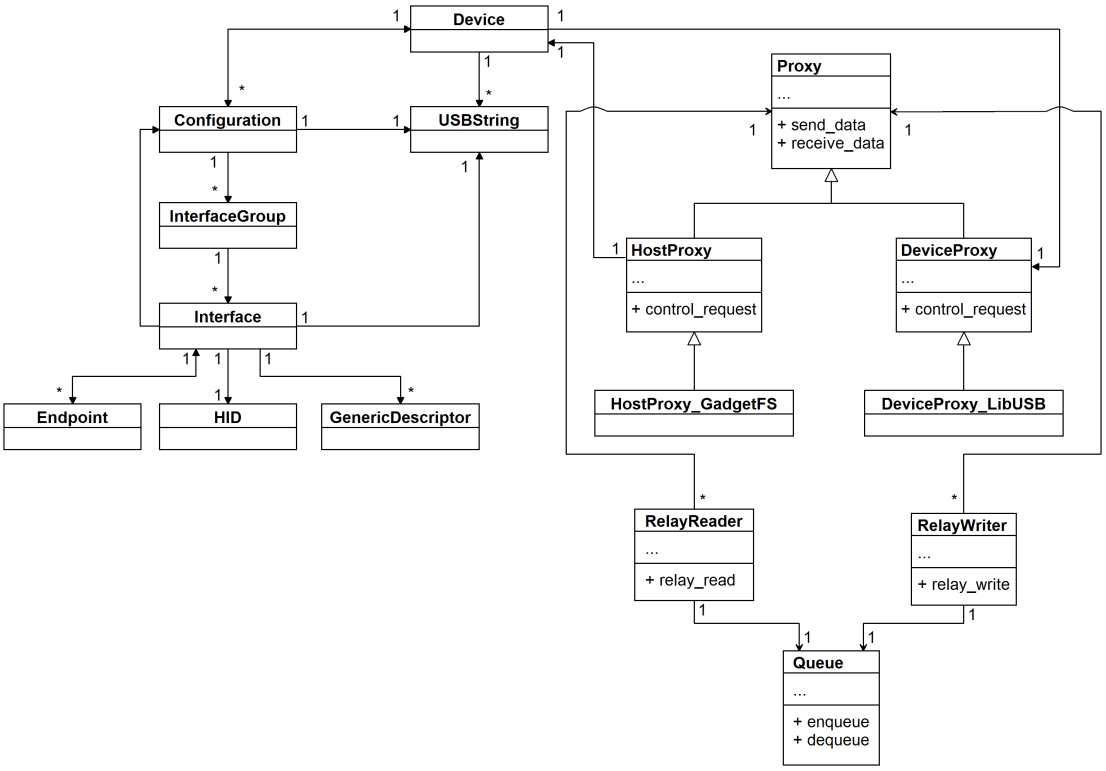
Figura 4.4: Protocollo Bulk-Only Transport catturato da Wireshark.

Uno dei compiti del software da realizzare è quello di emulare il comportamento rilevato analizzando il traffico tra host e periferica, permettendo di simulare correttamente il setup, e i pacchetti standard, di una determinata classe di dispositivi senza possederli fisicamente.
Un altro requisito fondamentale è la modularità delle periferiche da emulare: deve essere infatti possibile aggiungere ed utilizzare un nuovo "virtual device" senza modificare il codice e soprattutto in maniera semplice e seguendo la logica dei descriptor. Inoltre, per i dispositivi di input, si necessita di un meccanismo che permetta di inviare all'host comandi provenienti dall'interazione umana, come la pressione di un tasto del mouse o

la battitura di un testo con la tastiera, tramite il finto dispositivo (anche in questo caso senza modificare il codice).

4.1.2 Analisi dell'architettura di USBProxy

Si è scelto nuovamente di utilizzare, come base per la piattaforma, **USBProxy**. Il framework, però, è concepito per il logging del traffico di una periferica USB realmente presente e non prevede la gestione di dispositivi non fisicamente connessi all'hardware. Tuttavia, la sua architettura rappresenta un ottimo punto di partenza per soddisfare i requisiti desiderati, sebbene è stato necessario uno studio approfondito di alcune delle classi presenti.



Le classi `Device`, `Configuration`, `Interface`, `Endpoint`, `HID`, `GenericDescriptor` e `USBString` descrivono, nel mondo ad oggetti, le strutture dati dei descriptor. La classe `Device` dialoga fisicamente con il dispositivo USB connesso, richiedendo i descriptor e creando istanze delle classi illustrate precedentemente.

Nel caso in esame, sebbene non sia presente una periferica reale, tutti i metodi e gli attributi di `Device` sono richiesti per il corretto funzionamento del framework, quindi si necessita di leggere i descriptor e creare le istanze delle varie classi in una via alternativa, che permetta la selezione arbitraria del dispositivo da emulare.

4.1.3 Analisi della comunicazione tra host e dispositivo in USBProxy

La comunicazione dal lato host viene effettuata con la classe `HostProxy_GadgetFS`. Durante l'inizializzazione dell'istanza di questa classe, viene scritto all'interno del file referenziato da GadgetFS l'intero descriptor del dispositivo connesso, letto da `Device` e relative classi associate. Il funzionamento di `HostProxy_GadgetFS` non varia se la periferica non è presente, è necessario soltanto passare correttamente le strutture per la scrittura dei descriptor sul file.

Per quanto riguarda la comunicazione con il dispositivo, la responsabilità è delegata alla classe `DeviceProxy_LibUSB`. Nel caso dell'emulazione, non è presente una periferica e di conseguenza libUSB non risulta utilizzabile: è necessario progettare un nuovo `DeviceProxy`.

Per ogni endpoint dichiarato vengono creati due thread che eseguono, fino alla fine dell'esecuzione, rispettivamente il metodo `relay_read` di `RelayReader` e `relay_write` di `RelayWriter`. Internamente, questi metodi richiamano ciclicamente `receive_data` e `send_data` dei relativi `Proxy`. Quindi, ogni volta che viene ricevuto un pacchetto (indifferentemente da host o device), l'interazione è la seguente:

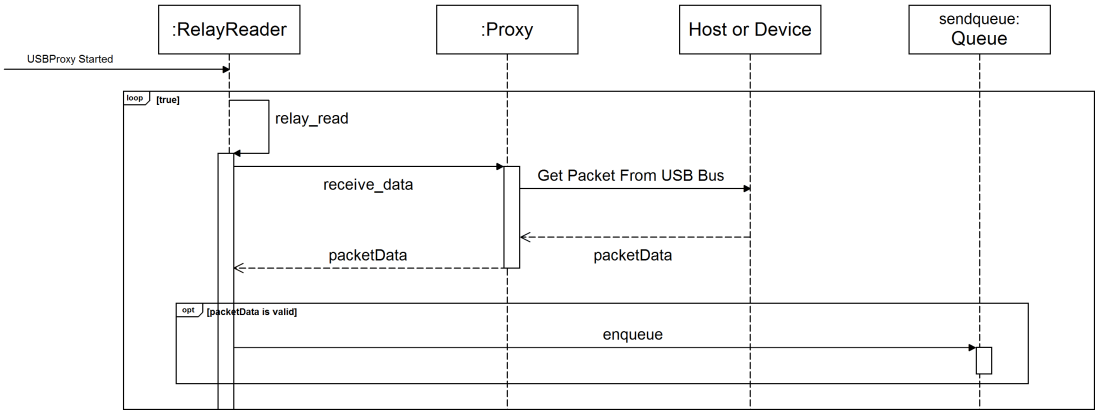


Figura 4.5: Interazione tra classi alla ricezione del pacchetto.

Il pacchetto viene ricevuto nel ciclo di `relay_read` dal metodo `receive_data` del relativo `Proxy` e, se è valido, inserito in una coda.
Una volta accodato, il pacchetto viene rigirato dalla porta USB di ricezione a quella di destinazione. L'interazione è la seguente:

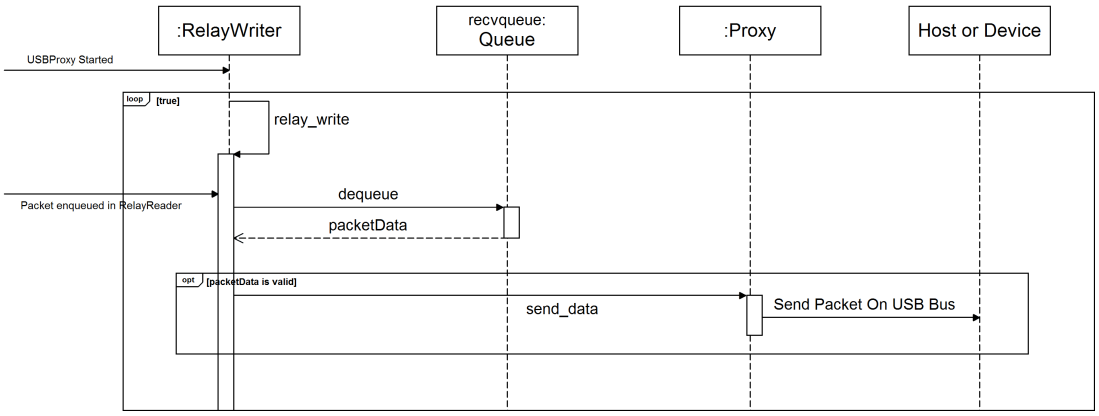


Figura 4.6: Interazione tra classi per l'instradamento del pacchetto a destinazione.

Per inviare pacchetti creati artificialmente, è necessario "iniettarli" all'interno di questa catena di eventi, nello specifico nel metodo `receive_data` di `DeviceProxy`. Infatti, quando è realmente connesso un dispositivo, i dati inviati dalla periferica vengono catturati da libUSB in `receive_data` e ritornati a `relay_read`, che li inserisce nella coda.

4.2 Progettazione della piattaforma

Alla fase di analisi discussa nella sezione 4.1, è seguita la progettazione della piattaforma, modificando l'architettura originale di USBProxy.

4.2.1 Definizione di un dispositivo da emulare nel software

Dato che l'obiettivo è quello di emulare dispositivi USB, il primo passo è stato quello di definire una metodologia per aggiungere o modificare i device virtuali da utilizzare. La struttura ideata è la seguente:

- ogni dispositivo da emulare è rappresentato da una cartella, nella quale deve essere presente il file binario `device` che contiene l'intera struttura dati del corrispondente Device Descriptor. La dimensione del file è esattamente di 18 bytes
- nella root del device virtuale sono presenti anche i file `stringConf`, `product`, `manufacturer` e `serialNumber` che contengono, se presenti, gli String Descriptor dichiarati dal dispositivo
- ogni configurazione è rappresentata da una cartella `configN`, dove N è un indice che varia in $[0, bNumConfigurations]$ espresso nel Device Descriptor. All'interno di ogni cartella deve essere presente il file `config`, corrispondente alla struttura Configuration Descriptor, della dimensione fissa di 9 bytes
- in ogni K-esima cartella `configK` è presente un altro file `epConfig`, che mappa gli endpoints di uscita con quelli di ingresso. Se, per esempio, all'endpoint 02h (*out*) corrisponde 81h (*in*), allora nel file saranno presenti i bytes: `02 81`. Se non c'è una corrispondenza tra due endpoints, al posto dell'*out* endpoint deve esserci il valore FFh. Per esempio se all'endpoint 81h non corrisponde nessuna uscita, allora i bytes presenti sono i seguenti: `FF 81`
- in ogni K-esima cartella `configK` sono presenti altrettante cartelle `ifaceN`, dove N è un indice che varia in $[0, bNumInterfaces]$ dichiarate nel Configuration Descriptor. All'interno di ogni cartella deve essere presente il file `iface`, corrispondente alla struttura Interface Descriptor, della dimensione di 9 bytes

- in ogni K-esima cartella `ifaceK` sono presenti i files `epN`, dove N è un indice che varia in $[0, bNumEndpoints]$ dichiarati nell'Interface Descriptor. Il loro contenuto è quello descritto dalla struttura Endpoint Descriptor, con dimensione di 7 bytes

- in ogni K-esima cartella `ifaceK` possono essere presenti N files `genDescN`, dove N è un indice che varia tra 0 ed il numero di Generic Descriptor dichiarati dall'interfaccia. Contengono le strutture dati dei Descriptor specifici per una classe di dispositivi

Ogni classe di dispositivi esaminata possiede, inoltre, alcuni file specifici.
Per la classe HID:

- in ogni K-esima cartella `ifaceK` è presente il file `hidDesc`, che contiene l'HID Descriptor di quell'interfaccia ed il file `hidReport`, che memorizza la struttura dati inviata come risposta alla richiesta GET DESCRIPTOR (HID REPORT)

Per la classe mass storage:

- nella root sono presenti i file `scsiDeviceInfo` contenente i dati ritornati dal dispositivo al comando INQUIRY e `scsiModePages` che memorizza le modalità di paging supportate dalla periferica

- è inoltre presente, nella root, un file `virtualDrive` che è il dump di un dispositivo mass storage reale, contenente la tabella delle partizioni e l'intero spazio di archiviazione

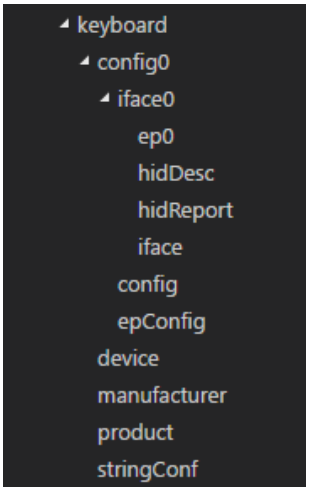


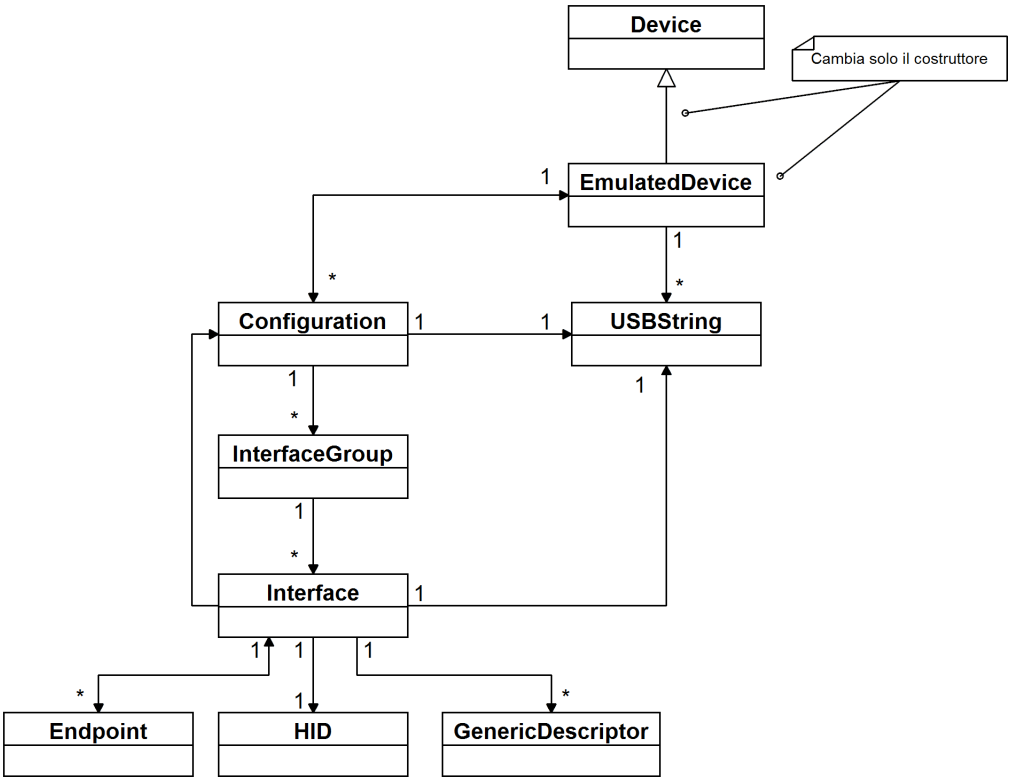
Figura 4.7: Un esempio di dispositivo emulabile nella piattaforma.

Utilizzando la struttura nidificata del file system è stato quindi possibile definire un metodo di emulazione dei dispositivi USB. La soluzione adottata consente una notevole facilità d'uso poiché è necessario soltanto creare cartelle e sottocartelle, e salvare i valori dei descriptor (catturandoli con Wireshark) in file binari con un qualsiasi editor esadecimale. Inoltre, un dispositivo è facilmente modificabile, e si può espandere il numero di interfacce o endpoints semplicemente inserendo files, e sottocartelle, a quelli già presenti.

4.2.2 Riconoscimento del dispositivo virtuale in USBProxy

Una volta definito il modo con cui creare nuovi dispositivi arbitrari, è necessario renderli utilizzabili all'interno di USBProxy. `Device` rappresenta un dispositivo realmente connesso all'hardware: è richiesta una classe che ci permetta di leggere i file creati dall'utente mantenendo però lo stesso comportamento di `Device`, indispensabile per continuare a garantire il corretto funzionamento dell'intero framework.
Viene introdotta la classe `EmulatedDevice`, sottoclasse di `Device` che differisce da quest'ultima per il solo costruttore: in questo caso, tutti i descriptor del dispositivo non vengono ricevuti interrogandolo, ma leggendoli dai file della periferica da emulare discussi nella sezione 4.2.1. Questo garantisce, grazie al *polimorfismo*, che tutte le altre classi non notino la differenza tra un'istanza di `Device` e `EmulatedDevice`, continuando

ad utilizzare i metodi senza comportamenti anomali.



4.2.3 Comunicazione tra host e dispositivo emulato

Grazie alla classe `EmulatedDevice`, il dispositivo viene correttamente riconosciuto dall'host, che lo enumera seguendo quanto dichiarato dai descriptor "artificiali". Il passo successivo è stato quello di progettare l'intera comunicazione tra host e device, ovvero garantire che tutte le richieste ricevano una risposta da parte del finto dispositivo. I diagrammi di interazione 4.5 e 4.6 hanno chiarito come USBProxy gestisce la cattura dei pacchetti da entrambe le porte ed il loro inoltro verso la destinazione. Non avendo un dispositivo connesso, `DeviceProxy_LibUSB` non era utilizzabile, quindi è stata creata una nuova sottoclasse di `DeviceProxy` chiamata `DeviceProxy_Emulation`.

A differenza di `DeviceProxy_LibUSB`, non viene usata nessuna libreria del sistema operativo che comunica con l'hardware fisicamente presente. I tre metodi principali `send_data`, `receive_data` e `control_request` fungono da semplici "forwarders" verso

un nuovo mini-framework progettato per interpretare le richieste in ingresso e generare flussi di pacchetti artificiali da inviare.

4.2.4 Gestione della risposta alle richieste dell'host

`Attack` è una classe che identifica un generico "attacco". In realtà si occupa anche di interpretare le richieste inviate dall'host e generare le relative risposte, che possono quindi essere adeguatamente alterate per simulare attacchi (come per esempio ARP Poisoning nel caso di una scheda Ethernet).
È noto che le richieste dell'host si dividono in due tipi: di **setup** (sul control endpoint) e **standard** (che variano in base alla tipologia della periferica).

Le richieste di setup sono identiche per ogni dispositivo, quindi è stato introdotto un metodo `parseSetupRequest` che si occupa di interpretarle e generare la risposta. Per fare ciò, il metodo utilizza una mappa `setupType2callback` che ha come chiave il codice della richiesta e come valore una funzione di callback da eseguire quando essa viene ricevuta. Le chiavi sono generate come la somma tra il valore di *wValue* e quello di *bRequest* del pacchetto di setup ricevuto. Per ogni contol request viene quindi definito un nuovo metodo (di callback) che è specifico per quest'ultima. In questo modo, è possibile aggiungere facilmente la risposta ad una nuova richiesta semplicemente dichiarando un ulteriore metodo e aggiungendo un'entry alla mappa. Nella classe `Attack` sono presenti soltanto callbacks a control request generiche. Nelle sue sottoclassi possono essere inserite risposte specifiche per il tipo di dispositivo rappresentato: scendendo nella gerarchia delle sottoclassi di `Attack` saranno dichiarate risposte sempre più specifiche per la tipologia di dispositivi emulata.

Le richieste standard invece variano; per questo il metodo `parseDeviceRequest` è astratto, e viene implementato diversamente da ogni sottoclasse di `Attack`. Tuttavia, ogni implementazione utilizza una mappa `deviceRequest2Callback` che individua, per ogni richiesta, un metodo callback da eseguire.

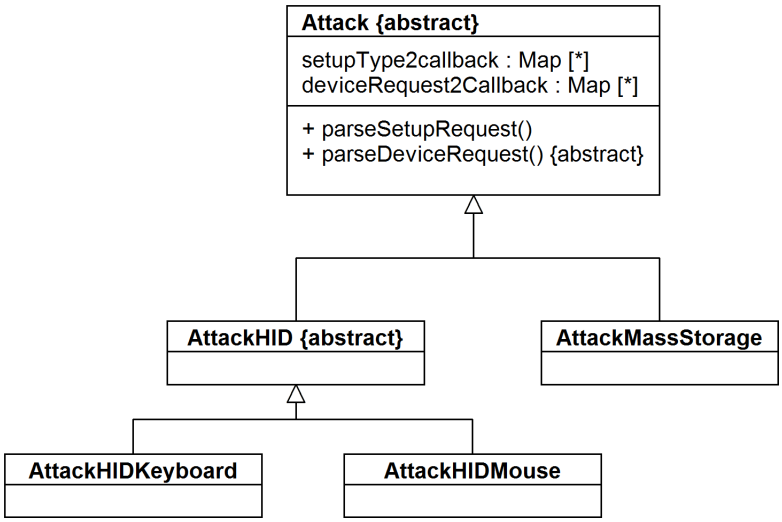
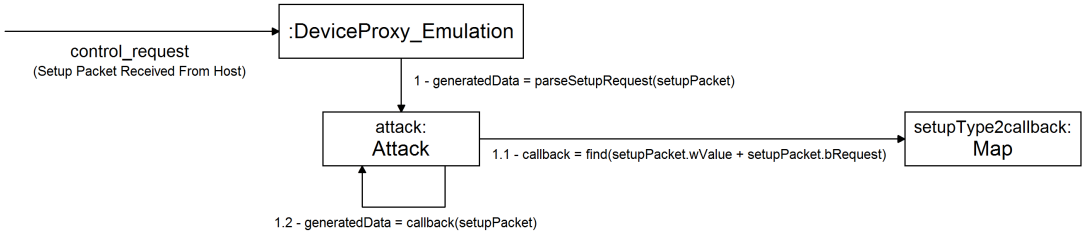


Figura 4.8: `Attack` e le sottoclassi definite nel caso di interesse.

I metodi `parseSetupRequest` e `parseDeviceRequest` vengono richiamati all'interno di `DeviceProxy_Emulation`, nello specifico:

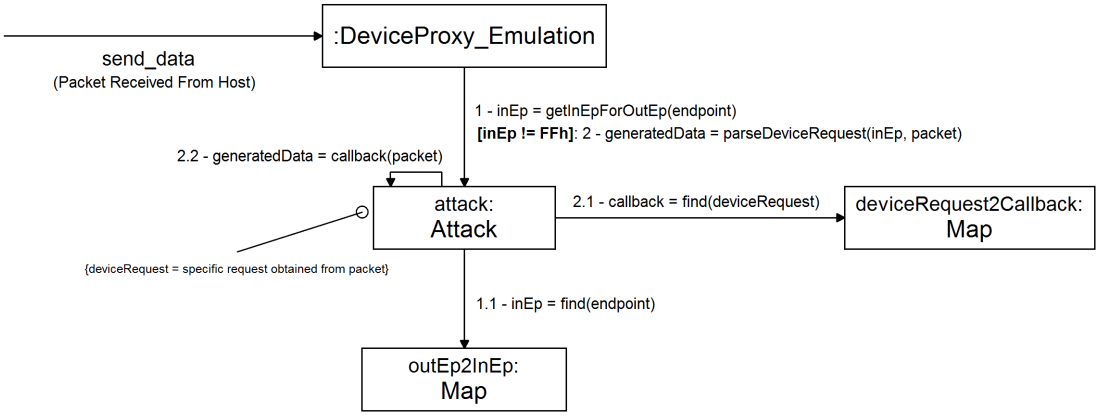
- `parseSetupRequest` in `control_request`
- `parseDeviceRequest` in `send_data`

`control_request` di `DeviceProxy_Emulation` è utilizzato soltanto durante la fase di setup: viene preso il pacchetto e rigirato verso `parseSetupRequest` di `Attack`.

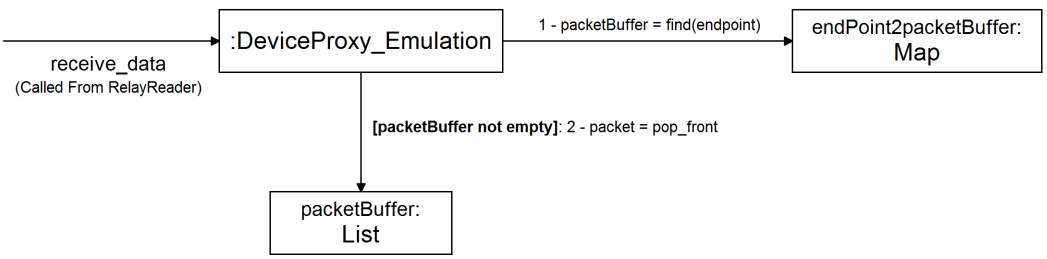


`send_data` è, come spiegato nella sezione 4.1.3, ciclicamente richiamato da `RelayWriter`. Nel caso di `DeviceProxy`, serve ad inviare al dispositivo un pacchetto proveniente dall'host. Gli endpoint sul quale transitano questi dati sono sempre *out* endpoint, quindi per inviare le risposte è necessario conoscere quale *in* endpoint utilizzare. In un dispositivo reale, questo viene deciso dal controller USB installato, nel caso del "virtual

device" il file `epConfig` serve ad eseguire il mapping. Quindi, alla creazione dell'oggetto istanza di `Attack`, il metodo `mapOutEpToInEp` memorizza all'interno della classe la corrispondenza tra *out/in* endpoints (in una mappa `outEp2InEp`), in modo da simulare il comportamento di un controller reale. Dopo aver ottenuto l'*in* endpoint, viene richiamata la funzione `parseDeviceRequest` che genera il blocco di pacchetti da trasmettere.



`receive_data` è l'opposto di `send_data`: invia i pacchetti dal dispositivo verso l'host. Per simulare la natura seriale dello standard USB, per ogni *in* endpoint è presente una coda che funge da buffer. Visto che `receive_data` è chiamato ciclicamente dal metodo `relay_read` di `RelayReader` in ogni *endpoint thread*, come illustrato nel diagramma 4.5, ogni volta che il metodo viene eseguito, grazie al valore del parametro `endpoint` si ottiene il buffer associato. Se quest'ultimo non è vuoto, un pacchetto viene "poppato" dalla testa e ritornato a `relay_read`, entrando così nella catena di invio verso l'host. Quindi, i pacchetti generati in `Attack` vengono accodati nel buffer, per essere estratti contemporaneamente dal thread `RelayWriter` che esegue `send_data`.

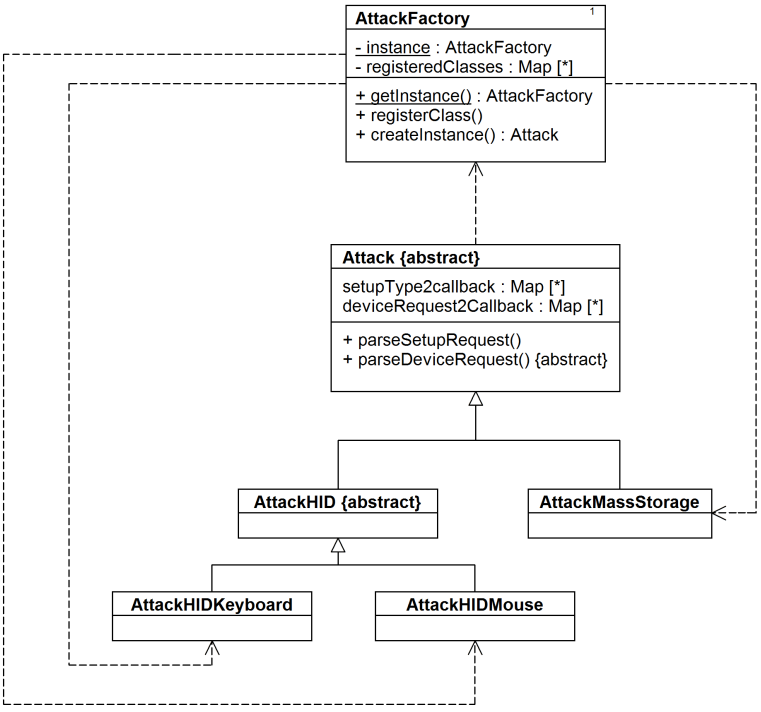


4.2.5 Selezione del dispositivo da emulare

A questo punto, si è in grado di enumerare il dispositivo emulato, ricevere e rispondere artificialmente alle richieste inviate dall'host. Rimangono ancora dei requisiti da soddisfare, uno di questi è la scelta del dispositivo virtuale senza modificare il codice.
La via più semplice è stata quella di aggiungere un argomento alla linea di comando utilizzata per eseguire il programma:

```
fakeusb -d <device_name>
```

`device_name` deve però trovare una corrispondenza (sottoforma di classe) all'interno del codice. Se per esempio `device_name` è "keyboard", il valore deve riferirsi ad una sottoclasse di `Attack` che identifica il dispositivo "tastiera". Siccome ogni tipologia di dispositivi utilizza un protocollo diverso, il matching deve essere 1 a 1, ovvero ad ogni `device_name` deve corrispondere una classe da utilizzare per processare le richieste ed inviare le specifiche risposte.
Per creare oggetti diversi in base alla stringa passata, è stata utilizzata una *Factory*: la classe `AttackFactory`, quindi, ritorna un oggetto istanza di una delle sottoclassi di `Attack` in base al parametro `device_name` indicato dall'utente.



Si aggiunge un ulteriore dettaglio alle specifiche: ogni cartella rappresentante un dispositivo deve possedere una corrispondente classe all'interno del framework, che si autoregistra nella *Factory* col valore da utilizzare per il parametro `device_name`. Se non esiste la corrispondenza tra `device_name` e classe, la risposta alle richieste è generica e non viene garantito il corretto funzionamento dell'emulazione.

Esempio la directory `mass-storage` individua i descriptor necessari per emulare un dispositivo mass storage. All'interno del framework è presente una classe `AttackMassStorage`, che si autoregistra nella *Factory* con la stringa "mass-storage" e permette di simulare ciò che si trova nella cartella `mass-storage` eseguendo il programma con il comando `fakeusb -d mass-storage`.

Viene così data la possibilità all'utente di scegliere il dispositivo da emulare, ma allo stesso tempo garantita la corrispondenza tra la directory del virtual device e la classe utilizzata per processare le richieste.

4.2.6 Emulazione dell'input di dispositivi HID

Un altro requisito è quello di simulare l'input dei dispositivi HID come se ci fosse un utente a comandarli. Gli HID non ricevono mai dati (al di fuori della fase di setup), bensì sono sempre loro ad inviare pacchetti all'host (che corrispondono a spostamenti del mouse o alla scrittura di un carattere, e così via). È richiesto che l'utente scelga, in maniera semplice e comoda, quali comandi inviare e soprattutto possa modificarli a proprio piacimento.
La soluzione è stata creare un mini-linguaggio di scripting per istruire il programma con le azioni da inviare all'host. Una serie di comandi scritti in un semplice file di testo specifica, quindi, cosa l'utente vorrebbe far compiere alla finta periferica.

I comandi definiti hanno una struttura molto semplice:

```
AZIONE(ENDPOINT) PARAMETRI
```

Quelli per la tastiera rispecchiano cosa è possibile fare concretamente con il dispositivo:

- `WRITE(EP) "stringa"`: scrive la stringa `stringa` come se fosse battuta sulla tastiera
- `PRESS_KEYS(EP) KEY1+KEY2+KEY3`: simula la pressione di una shortcut di massimo 3 tasti (es. ALT+F4 o CTRL+ALT+CANC)

Per il mouse, le azioni realmente possibili sono spostamento, click e la combinazione dei due, quindi i comandi ideati sono:

- `CLICK(EP) KEY`: simula il click del tasto `KEY` sulla periferica
- `MOVE(EP) X,Y`: sposta il cursore di `X` pixel in orizzontale e `Y` pixel in verticale
- `MOVE_AND_CLICK(EP) KEY X,Y`: sposta il cursore di `X` pixel in orizzontale e `Y` pixel in verticale mentre viene premuto il tasto `KEY`

Entrambi i dispositivi possono poi utilizzare:

- `DELAY(EP) TIME`: aspetta `TIME` millisecondi prima di inviare il comando successivo

Un esempio di script è il seguente:

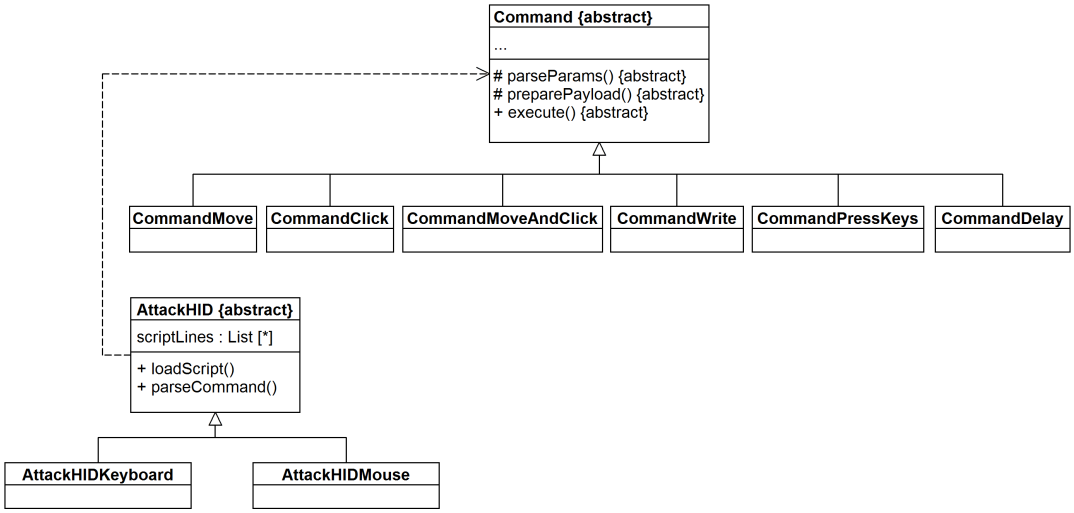
```
1 PRESS_KEYS(81)  CTRL+ALT+t
2 DELAY(81)       1500
3 WRITE(81)       "echo "Scrivo sulla console di Ubuntu!""
4 DELAY(81)       10
5 PRESS_KEYS(81)  ENTER
```

Il linguaggio di scripting realizzato permette di scegliere dinamicamente, ed in completa libertà, le azioni da eseguire con una periferica HID sull'host. Le stringhe dei comandi devono però tradursi in pacchetti da inviare sul bus USB del dispositivo emulato. La sottoclasse `AttackHID` possiede, quindi, un metodo `loadScript` che carica il file con lo script in memoria e `parseCommand` che interpreta ogni riga estraendo dal comando il valore di nome, endpoint e parametri.

L'esecuzione dei comandi viene delegata ad una nuova classe `Command`, che possiede tre metodi astratti:

- `parseParams`: analizza i valori e controlla che essi siano sintatticamente corretti rispetto alle specifiche del comando
- `preparePayload`: prende i valori dei parametri e genera N pacchetti in base ai dati passati. La dimensione e i dati presenti variano in base al dispositivo emulato e sono illustrati nella tabella 4.1 per mouse e 4.2 per tastiera
- `execute`: riceve la stringa contenente i parametri estratti da `parseCommand`, li analizza con `parseParams` e, se sono validi, genera il blocco di pacchetti da inviare all'host con `preparePayload`, ritornandolo al metodo chiamante

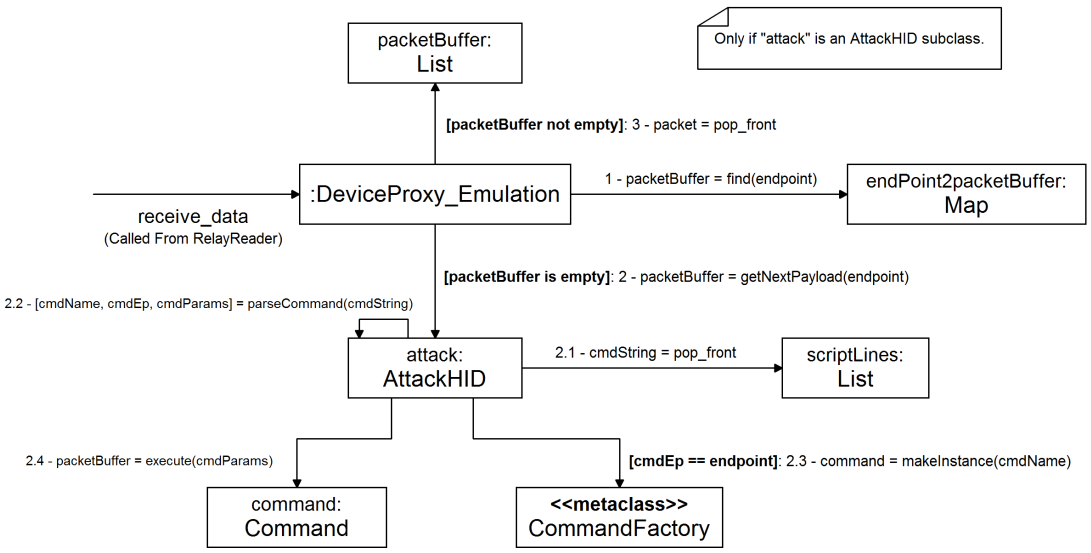
Ogni sottoclasse di `Command` individua un comando ed implementa i metodi astratti in base alle definizioni elencate precedentemente.



La conversione dalla stringa contenente il nome del comando all'istanza di una delle sottoclassi di `Command` è stata anche qui ottenuta con una *Factory*, chiamata `CommandFactory`. Siccome il comportamento di `AttackFactory` e `CommandFactory` è identico, cambia soltanto il tipo delle istanze da creare, è stata introdotta una *Abstract Factory* di nome `Factory` che ha come sottoclassi le due citate precedentemente.

Fino a questo punto, si è in grado di caricare lo script, interpretarne ogni riga, creare l'istanza della classe `Command` associata e generare i pacchetti artificiali da mandare all'host. Il tassello mancante è proprio la trasmissione dei dati.

Come già progettato in 4.2.4, con il buffer in `receive_data` è possibile inserire nella "catena di trasmissione" i pacchetti artificiali provenienti da `send_data` e creati da `Attack`.
In questo caso, però, non ci sono dati in ricezione, quindi `send_data` non viene mai eseguita. Tuttavia, `receive_data` è ciclicamente richiamata nel `RelayReader`, e astutamente si può utilizzare la sua invocazione per iniettare il blocco di pacchetti generato da `execute` della classe `Command`.
È stato quindi aggiunto un ulteriore controllo a quanto progettato precedentemente: se il buffer dell'*in* endpoint è vuoto, viene richiamato un nuovo metodo `getNextPayload` della classe `Attack`. `getNextPayload` non ha istruzioni nel suo corpo, e varia soltanto nella sottoclasse `AttackHID`: in questo caso viene presa la prossima istruzione da eseguire dello script, creata l'istanza del comando tramite `CommandFactory` ed invocato il metodo `execute`, che ritorna il blocco di pacchetti generato. I pacchetti vengono inseriti nel buffer dell'endpoint e sequenzialmente inviati verso l'host, consentendo l'esecuzione delle azioni indicate dall'utente.



Per scegliere quale script eseguire, è stato aggiunto un ulteriore argomento alla linea di comando utilizzata per lanciare il programma:

```
fakeusb -a <file_name>
```

`file_name` specifica il nome del file dello script da caricare.

4.2.7 Emulazione di periferiche mass storage

L'intero protocollo Bulk-Only Transport della classe di dispositivi mass storage è stato ampiamente discusso nella sezione 3.2. All'interno del framework progettato, la classe `AttackMassStorage` si occupa dell'emulazione di questo tipo di periferiche.
Come ben noto, il protocollo utilizza CBW per inviare comandi al dispositivo, al cui interno si trova il campo *CBWCB* che è un CDB dello standard SCSI. Il metodo `parseDeviceRequest` implementato si occupa, quindi, di controllare se il pacchetto ricevuto (tramite `send_data` di `DeviceProxy_Emulation`) è un CBW e, se la condizione è vera, estrarre l'OP Code del comando SCSI contenuto, corrispondente al byte 15. Il codice operativo viene poi usato come chiave per cercare la relativa funzione di callback nella mappa `deviceRequest2Callback`.
Ad ogni funzione di callback è delegata la responsabilità di generare il pacchetto CSW con lo stato dell'operazione, attraverso il metodo `buildCSWPacket`.

Particolare attenzione è necessaria per la progettazione della risposta alle richieste READ e WRITE, utilizzate per leggere e scrivere blocchi di dati dal dispositivo mass storage. Le richieste operano sui valori contenuti nei byte delle celle di memoria e specificano, entrambe, il *LBA*[1] di partenza ed il numero di blocchi da leggere o scrivere. I valori dei parametri vengono moltiplicati (internamente, dal dispositivo) per la dimensione del blocco logico (espressa in byte) specificata dalla periferica.

[1] Logical Block Address.

Esempio Il blocco logico dichiarato ha dimensione 512 bytes (200h).

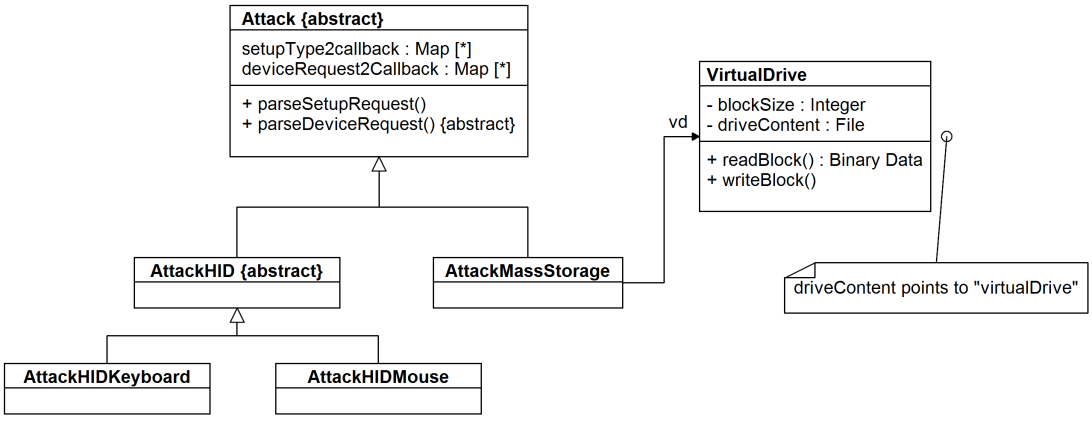
Il comando inviato è: READ, LBA: 00000001h, n. blocchi: 08h

I valori vengono convertiti come segue:

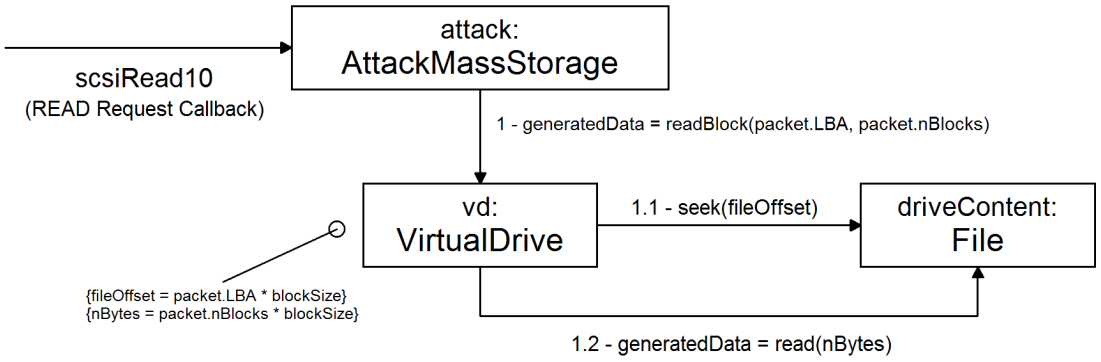
$$\text{LBA} \rightarrow 00000001h * 200h = 00000200h$$
$$\text{N. blocchi} \rightarrow 08h * 200h = 1000h$$

Quindi, l'indirizzo di partenza è 512 e verranno letti 4.096 byte.

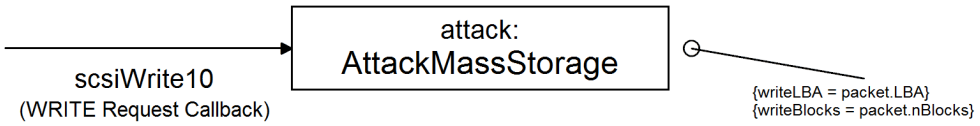
Nel caso dell'emulazione non è presente una memoria di massa, quindi è necessario introdurre un elemento che permetta di simularla in modo da poter rispondere correttamente alle richieste. La soluzione è stata aggiungere il file `virtualDrive`, descritto nella sezione 4.2 come il dump completo (byte per byte) di una periferica mass storage reale, ottenuto per esempio con il comando `dd` di Linux.
La memoria di massa è dunque rappresentata da un file binario: il LBA di partenza corrisponde all'offset dall'inizio del file, mentre il numero di blocchi è convertito in byte da leggere (o scrivere) partendo dall'offset.
Tutte le responsabilità relative all'emulazione di una memoria di massa non potevano essere assegnate ad `Attack`, che ha soltanto il compito di interpretare e rispondere alle richieste provenienti dall'host.
È stata quindi introdotta la classe `VirtualDrive`, che descrive uno "spazio di archiviazione virtuale" (rappresentato dal file `virtualDrive`), ne specifica la dimensione del blocco logico e permette le operazioni di lettura/scrittura con le opportune conversioni di parametri.

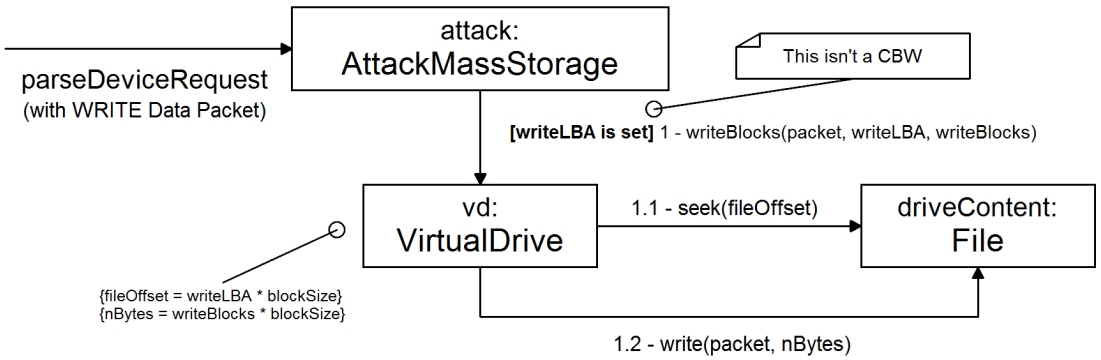


Ogni volta che una richiesta READ viene ricevuta in `AttackMassStorage`, il callback invoca il metodo `readBlock` di `VirtualDrive`, che converte il LBA in offset ed il numero di blocchi in bytes, legge i dati richiesti dal file binario e li ritorna all'host.



Per quanto riguarda il comando WRITE, nel CDB sono specificati soltanto i parametri LBA di partenza e numero di blocchi. I dati da scrivere sono inviati nel pacchetto successivo al CBW, che viene catturato da `send_data` e lo rigira a `parseDeviceRequest`. Qui è stato aggiunto un ulteriore controllo: se il pacchetto non è un CBW, allora si tratta di un blocco dati da scrivere sulla memoria di massa; viene richiamato il metodo `writeBlock` di `VirtualDrive` i cui parametri sono il pacchetto dati ricevuto, starting LBA e numero di blocchi da scrivere (entrambi memorizzati dal precedente CBW).





4.3 Realizzazione della piattaforma di emulazione

Quanto detto in 4.2 è stato poi realizzato in linguaggio C++ basandosi sul codice di USBProxy.

Nella cartella `config`, creata nella root della directory con il codice sorgente, devono essere inseriti tutti i dispositivi da emulare seguendo la logica descritta in 4.2.1.

La classe `EmulatedDevice`, descritta in 4.2.2, è stata ottenuta ereditando `Device` e sovraccaricando il costruttore:

```
class EmulatedDevice : public Device {
private:
    //[...]
public:
    EmulatedDevice(DeviceProxy *);
};
```

```
/* Porzione del costruttore, che carica i file contenenti i descriptor
   */
EmulatedDevice::EmulatedDevice(DeviceProxy * proxy) {
    this->proxy = proxy;
    this->proxy->setDevice(this);

    std::string deviceConfigurationDir = "/home/debian/FakeUSB/config/"
        + this->proxy->cfg->get("Device");
    std::string fileName = deviceConfigurationDir + "/device";

    FILE * fileHandler = fopen(fileName.c_str(), "rb");
    if(fileHandler) {
        fread(&descriptor, sizeof(descriptor), 1, fileHandler);
        configurations = (Configuration **) calloc(descriptor.
            bNumConfigurations, sizeof(*configurations));
        fclose(fileHandler);

        //[...]
    }

    //Continua...
}
```

La sostituzione dell'istanza avviene nella classe `Manager`, dove al posto di creare un'istanza `device` di `Device`, ne viene creata una di `EmulatedDevice`.

```
device = new EmulatedDevice(deviceProxy);
this->initAttack();
int rc = deviceProxy->connect();
//Continua...
```

La classe `Attack` è stata scritta seguendo quanto detto in 4.2.4.

```
class Attack {
protected:
    //[...]
    bool canAttack = false;
    __u32 DELAY_TIMER = 0;

    void mapOutEpToInEp();
    std::map<__u8, __u8> outEp2InEp;

    std::map<__u16, std::function<__u8(const usb_ctrlrequest, __u8 *)>>
        setupType2Callback;
    std::map<__u16, std::function<void(__u8 *, std::list<std::pair<__u8
        *, __u64>> **)>> deviceRequest2Callback;
    //Callbacks...
public:
    //[...]
    int parseSetupRequest(const usb_ctrlrequest, int *, __u8 *);
    virtual void parseDeviceRequest(__u16, __u8 *, __u64, std::list<std
        ::pair<__u8 *, __u64>> **) { }

    virtual void getNextPayload(std::list<std::pair<__u8 *, __u64>> **,
        __u8, __u16) { };

    __u8 getInEpForOutEp(__u8);

    void startAttack();
    bool canStartAttack() { if(DELAY_TIMER) DELAY_TIMER--; return this->
        canAttack && (DELAY_TIMER == 0); }
};
```

Sono stati aggiunti gli attributi `canAttack`, `DELAY_TIMER` e i metodi `startAttack` e `canStartAttack`. Questi nuovi elementi servono per attendere un determinato numero di millisecondi, specificati in `DELAY_TIMER`, prima di iniziare ad inviare i pacchetti artificiali all'host. Si è reso necessario inserirli perché, durante i test eseguiti, i pacchetti venivano trasmessi nella fase di setup ed erano ignorati dall'host. La flag `canAttack`, che permette ai pacchetti di essere trasmessi, viene settata a `true` alla fine dell'inizializzazione degli *endpoint threads* nel `Manager`, tramite il metodo `startAttack`.

```
void Attack::startAttack() {
    //[...]
    this->mapOutEpToInEp();
    this->canAttack = true;
}
```

```
/* Esempio di coppia chiave-valore che specifica l'identificativo di una
     control request ed il callback da eseguire */
this->setupType2Callback.insert(
    std::pair<__u16, std::function<__u8(const usb_ctrlrequest, __u8 *)
        >>(
        0x0306, std::bind(&Attack::getStringDescriptor, this, std::
            placeholders::_1, std::placeholders::_2)
    )
);
```

```
/* Il metodo parseSetupRequest progettato precedentemente */
int Attack::parseSetupRequest(const usb_ctrlrequest setupPacket, int *
    nBytes, __u8 * dataPtr) {
    *nBytes = 0;

    __u16 searchValue = setupPacket.wValue + setupPacket.bRequest;
    std::map<__u16, std::function<__u8(const usb_ctrlrequest, __u8 *)
        >>::iterator it = this->setupType2Callback.find(searchValue);

    if(it != this->setupType2Callback.end())
        *nBytes = (*it).second(setupPacket, dataPtr);

    return 0;
}
```

```
/* Un esempio di callback ad una control request */
__u8 Attack::getStringDescriptor(const usb_ctrlrequest packet, __u8 *
    dataPtr) {
    __u8 descIndex = packet.wValue & 0xFF;

    const usb_string_descriptor * stringDescriptor = this->device->
        get_string(descIndex, packet.wIndex)->get_descriptor();
    __le16 packetSize = (packet.wLength < stringDescriptor->bLength) ?
        packet.wLength : stringDescriptor->bLength;

    memcpy(dataPtr, stringDescriptor, packetSize);

    return packetSize;
}
```

I metodi `control_request`, `send_data` e `receive_data` di `DeviceProxy_Emulation` sono invece i seguenti, progettati come descritto nelle sezioni 4.2.3 e 4.2.4:

```
int DeviceProxy_Emulation::control_request(const usb_ctrlrequest *
    setup_packet, int * nbytes, __u8 * dataptr, int timeout) {
    return this->attack->parseSetupRequest(*setup_packet, nbytes,
        dataptr);
}

void DeviceProxy_Emulation::send_data(__u8 endpoint, __u8 attributes,
    __u16 maxPacketSize, __u8 * dataptr, __u64 length) {
    __u8 inEp = this->attack->getInEpForOutEp(endpoint);
    if(inEp != 0xFF) {
        std::list<std::pair<__u8 *, __u64>> * packetBuffer = this->
            getPacketBufferForEndpoint(inEp);
        this->attack->parseDeviceRequest(maxPacketSize, dataptr, length,
             &packetBuffer);
        this->setPacketBufferForEndpoint(packetBuffer, inEp);
    }
}

void DeviceProxy_Emulation::receive_data(__u8 endpoint, __u8 attributes,
     __u16 maxPacketSize, __u8 ** dataptr, __u64 * length, int timeout) {
    *length = 0;
```

```
16
17      if(!this->attack->canStartAttack())
18          return;
19
20      std::list<std::pair<__u8 *, __u64>> * packetBuffer = this->
            getPacketBufferForEndpoint(endpoint);
21
22      if(packetBuffer->empty())
23          this->attack->getNextPayload(&packetBuffer, endpoint,
                maxPacketSize);
24
25      if(!packetBuffer->empty()) {
26          std::pair<__u8 *, __u64> dataAndLength = packetBuffer->front();
27
28          (*dataptr) = (__u8 *) calloc(dataAndLength.second, sizeof(__u8))
                ;
29          memcpy((*dataptr), dataAndLength.first, dataAndLength.second);
30
31          *length = dataAndLength.second;
32          packetBuffer->pop_front();
33      }
34
35      this->setPacketBufferForEndpoint(packetBuffer, endpoint);
36  }
```

La selezione del dispositivo da emulare è ottenuta dal valore dell'argomento `-d`, la stringa viene poi passata ad `AttackFactory` che crea un'istanza di una delle sottoclassi di `Attack` (sezione 4.2.5). Il tutto avviene in `Manager`, nel metodo `initAttack`.

```
1  /* Parsing del valore dell'argomento -d nel main */
2  while ((opt = getopt(argc, argv, "a:d:")) != EOF) {
3      switch (opt) {
4          case 'd':
5              cfg->set("Device", std::string(optarg));
6          break;
7          //[...]
8      }
9  }
```

```
void Manager::initAttack() {
    this->attack = AttackFactory::getInstance()->createInstance(this->
        cfg->get("Device"));
    //[...]
}
```

La *Abstract Factory*, descritta durante la progettazione, è la seguente:

```
template<typename T>
    class Factory : public Singleton<Factory<T>> {
    protected:
        std::map<std::string, T * (*)()> * registeredClasses = NULL;

    public:
        Factory() {
            this->registeredClasses = new std::map<std::string, T * (*)
                ()>;
        }

        bool registerClass(const std::string &name, T * (*createMethod)
            ()) {
            this->registeredClasses->insert(std::pair<std::string, T *
                (*)()>(name, createMethod));

            return true;
        }

        T * createInstance(const std::string &name) {
            auto it = this->registeredClasses->find(name);

            if(it != this->registeredClasses->end())
                return (*it).second();

            return NULL;
        }
    };
```

La classe `AttackFactory` la eredita, specificando il tipo da passare come template.

```
class AttackFactory : public Factory<Attack> {
    public:
```

```
3         AttackFactory() : Factory() {}
4         ~AttackFactory() {}
5 };
```

Per gestire le numerose istanze *Singleton* presenti all'interno del codice, è stata creata una classe parametrizzata `Singleton` (che viene ereditata anche da `Factory`):

```
1 template<typename T>
2     class Singleton {
3     protected:
4         static T * instance;
5
6     public:
7         Singleton() {
8             instance = static_cast<T *>(this);
9         }
10
11        ~Singleton() {}
12
13        static T * getInstance() {
14            return instance;
15        }
16    };
17
18 template<typename T>
19    T * Singleton<T>::instance = new T();
```

Per la creazione di un'istanza di una classe a partire da una stringa, normalmente si utilizza la tecnica chiamata *riflessione*. In C++, essa non è disponibile di default nel linguaggio, come per esempio accade in Java.

Per realizzare la riflessione, nella classe che intende autoregistrarsi in una *Factory*, viene definito un metodo statico `makeInstance` che permette di istanziare oggetti della specifica classe. Durante la definizione, viene richiamato il metodo `registerClass` di `AttackFactory`, passandogli come parametro la stringa da utilizzare (che deve essere uguale al valore dell'argomento `-d` associato) ed il puntatore al metodo `makeInstance`.

```
1 /* Esempio di classe che si autoregistra in AttackFactory */
2 class AttackHIDKeyboard : public AttackHID {
3 public:
```

```
4       AttackHIDKeyboard();
5       ~AttackHIDKeyboard();
6       /* Metodo statico "makeInstance" da usare in AttackFactory */
7       static Attack * makeInstance() { return new AttackHIDKeyboard(); }
8   };
9
10  /* Autoregistrazione in AttackFactory */
11  const bool hasRegistered = AttackFactory::getInstance()->registerClass("
        keyboard", &AttackHIDKeyboard::makeInstance);
```

Per la gestione dei comandi, descritti in 4.2.6, la classe `Command` implementata è la seguente:

```
1   class Command {
2   protected:
3       Command();
4       virtual std::vector<std::string> * parseParams(const std::string &)
            = 0;
5       virtual std::list<std::pair<__u8 *, __u64>> * preparePayLoad(std::
            string, __u16) = 0;
6
7   public:
8       virtual ~Command();
9       virtual std::list<std::pair<__u8 *, __u64>> * execute(const std::
            string &, __u16) = 0;
10  };
```

`CommandFactory` è identica alla *Factory* di `Attack`, differisce solo il tipo passato nel template:

```
1   class CommandFactory : public Factory<Command> {
2       public:
3           CommandFactory() : Factory() {}
4           ~CommandFactory() {}
5   };
```

Ogni comando eredita poi `Command` ed implementa i metodi astratti, un esempio è `CommandWrite`.

```
1   /* Metodi astratti implementati */
```

```
2  std::list<std::pair<__u8 *, __u64>> * CommandWrite::preparePayLoad(std::
       string stringToWrite, __u16 maxPacketSize) {
3      std::list<std::pair<__u8 *, __u64>> * payLoad = new std::list<std::
           pair<__u8 *, __u64>>;
4
5      for(unsigned int i = 0; i < stringToWrite.length(); ++i) {
6          __u8 * packetPressed = (__u8 *) calloc(maxPacketSize, sizeof(
               __u8));
7          std::pair<__u8, __u8> firstAndThirdByte = findCharacter(
               stringToWrite[i]);
8
9          packetPressed[0x00] = firstAndThirdByte.first;
10         packetPressed[0x02] = firstAndThirdByte.second;
11
12         payLoad->push_back(std::pair<__u8 *, __u64>(packetPressed,
               maxPacketSize));
13     }
14
15     for(std::list<std::pair<__u8 *, __u64>>::iterator it = payLoad->
           begin(); it != payLoad->end(); ++it) {
16         std::list<std::pair<__u8 *, __u64>>::iterator nextPacket = std::
               next(it, 1);
17
18         if(nextPacket != payLoad->end())
19             if((*it).first[0x02] == (*nextPacket).first[0x02]) {
20                 __u8 * packetReleased = (__u8 *) calloc(maxPacketSize,
                       sizeof(__u8));
21                 payLoad->insert(nextPacket, std::pair<__u8 *, __u64>(
                       packetReleased, maxPacketSize));
22             }
23
24
25         std::prev(it, 1);
26     }
27
28     __u8 * packetReleased = (__u8 *) calloc(maxPacketSize, sizeof(__u8))
           ;
29     payLoad->push_back(std::pair<__u8 *, __u64>(packetReleased,
```

```
        maxPacketSize));
30
31      return payLoad;
32  }
33
34
35  std::vector<std::string> * CommandWrite::parseParams(const std::string &
        paramString) {
36      std::regex paramRegex("^\"(.*)\"$", std::regex_constants::icase);
37      std::smatch matches; std::regex_search(paramString, matches,
            paramRegex);
38
39      if(!matches[1].str().empty()) {
40          std::vector<std::string> * paramVector = new std::vector<std::
                string>;
41          paramVector->push_back(matches[1].str());
42
43          return paramVector;
44      }
45
46      return NULL;
47  }
48
49  std::list<std::pair<__u8 *, __u64>> * CommandWrite::execute(const std::
        string &paramString, __u16 maxPacketSize) {
50      std::vector<std::string> * paramList = this->parseParams(paramString
            );
51
52      if(paramList) {
53          std::list<std::pair<__u8 *, __u64>> * payLoad = this->
                preparePayLoad(paramList->at(0), maxPacketSize);
54          delete(paramList);
55
56          return payLoad;
57      }
58
59      return new std::list<std::pair<__u8 *, __u64>>;
60  }
```

Per mappare i caratteri in formato *ASCII* (usati dalle stringhe in C++) ai relativi valori dei byte USB, è stata introdotta una serie di utilities, formata da mappe e funzioni, che permettono la conversione, racchiuse nell'header `KeyMap.h`.

`getNextPayload` e `parseCommand` di `AttackHID`, che permettono l'esecuzione dei comandi, sono implementati come segue:

```
1  std::tuple<std::string, __u8, std::string> AttackHID::parseCommand(const
       std::string &command) {
2      std::regex commandRegex("^([A-Za-z_]+)\\(([0-9a-fA-F]{2})\\)(\\s)
           +(.*)$", std::regex_constants::icase);
3      std::smatch matches; std::regex_search(command, matches,
           commandRegex);
4
5      if(!matches[1].str().empty() && !matches[2].str().empty() && !
           matches[4].str().empty())
6          return std::make_tuple(matches[1].str(), std::stoi(matches[2].
               str(), 0, 16), matches[4].str());
7
8      return std::make_tuple("", 0x00, "");
9  }
10
11 void AttackHID::getNextPayload(std::list<std::pair<__u8 *, __u64>> **
       payload, __u8 endpoint, __u16 maxPacketSize) {
12     if(!this->attackCommands->empty()) {
13         std::string commandString = this->scriptLines->front();
14         std::string commandName, commandParams; __u8 ep;
15
16         std::tie(commandName, ep, commandParams) = this->parseCommand(
               commandString);
17
18         if(!commandName.empty()) {
19             if(endpoint == ep) {
20                 this->scriptLines->pop_front();
21                 Command * command = CommandFactory::getInstance()->
                       createInstance(commandName);
22
23                 if(command) {
```

```
24                      std::list<std::pair<__u8 *, __u64>> * newPayload =
                            command->execute(commandParams, maxPacketSize);
25                      std::copy(newPayload->begin(), newPayload->end(),
                            std::back_insert_iterator<std::list<std::pair<
                            __u8 *, __u64>>>(**payload));
26
27                      delete(command);
28                      delete(newPayload);
29                  }
30              }
31          }
32      }
33  }
```

Il metodo `parseDeviceRequest` di `AttackMassStorage` e i callback alle richieste READ e WRITE sono realizzati seguendo quanto detto nella sezione 4.2.7.

```
1  /* Esempio di coppia chiave-valore che specifica l'OP Code ed il
       callback da eseguire */
2  this->deviceRequest2Callback.insert(
3      std::pair<__u16, std::function<void(__u8 *, std::list<std::pair<__u8
           *, __u64>> **)>>(
4          SCSI_CMD_READ_10, std::bind(&AttackMassStorage::scsiRead10, this
               , std::placeholders::_1, std::placeholders::_2)
5      )
6  );
```

```
1  void AttackMassStorage::parseDeviceRequest(__u16 maxPacketSize, __u8 *
       dataPtr, __u64 length, std::list<std::pair<__u8 *, __u64>> **
       packetBuffer) {
2      if ((length == 31) &&
3          (dataPtr[0x00] == 0x55) &&
4          (dataPtr[0x01] == 0x53) &&
5          (dataPtr[0x02] == 0x42) &&
6          (dataPtr[0x03] == 0x43)) {
7          /* E' un CBW */
8          __u16 opCode = dataPtr[0x0f];
9
10         std::map<__u16, std::function<void(__u8 *, std::list<std::pair<
               __u8 *, __u64>> **)>>::iterator it = this->
```

```
            deviceRequest2Callback.find(opCode);
11
12          if(it != this->deviceRequest2Callback.end())
13              (*it).second(dataPtr, packetBuffer);
14
15          return;
16      }
17
18      /* E' un pacchetto dati da scrivere sul file virtualDrive */
19      if(this->writeLBA != 0xffffffffffffffff) {
20          this->virtualDrive->writeBlock(dataPtr, this->writeLBA, length,
                this->writtenBlocks);
21          this->writtenBlocks++;
22
23          return;
24      }
25  }
```

```
1   /* Callback per il comando READ */
2   void AttackMassStorage::scsiRead10(__u8 * dataPtr, std::list<std::pair<
        __u8 *, __u64>> ** packetBuffer) {
3       MS_CommandBlockWrapper_t * cbw = new MS_CommandBlockWrapper_t;
4       memcpy(cbw, dataPtr, MS_CBW_SIZE);
5
6       /* Conversione da little endian a big endian dei due campi */
7       __u64 startingReadLBA = cbw->SCSICommandData[0x02] << 24 | cbw->
            SCSICommandData[0x03] << 16 | cbw->SCSICommandData[0x04] << 8 |
            cbw->SCSICommandData[0x05];
8       __u64 blocksToRead = (cbw->SCSICommandData[0x07] << 8) | cbw->
            SCSICommandData[0x08];
9
10      __u32 readBlocks = 0x00;
11      while(readBlocks < blocksToRead) {
12          __u64 blockSize = 0x00;
13          __u8 * dataFromDrive = this->virtualDrive->readBlock(
                startingReadLBA, &blockSize, readBlocks);
14          (*packetBuffer)->push_back(std::pair<__u8 *, __u64>(
                dataFromDrive, blockSize));
15
```

```
16          readBlocks++;
17      }
18
19      (*packetBuffer)->push_back(std::pair<__u8 *, __u64>(this->
            buildCSWPacket(cbw->Tag, MS_SCSI_COMMAND_Pass), MS_CSW_SIZE));
20
21      delete(cbw);
22  }
```

```
1   /* Callback per il comando WRITE */
2   void AttackMassStorage::scsiWrite10(__u8 * dataPtr, std::list<std::pair<
        __u8 *, __u64>> ** packetBuffer) {
3       MS_CommandBlockWrapper_t * cbw = new MS_CommandBlockWrapper_t;
4       memcpy(cbw, dataPtr, MS_CBW_SIZE);
5
6       /* Salva LBA e numero di blocchi in attesa del pacchetto dati
            successivo (in parseDeviceRequest) */
7       this->writeLBA = cbw->SCSICommandData[0x02] << 24 | cbw->
            SCSICommandData[0x03] << 16 | cbw->SCSICommandData[0x04] << 8 |
            cbw->SCSICommandData[0x05];
8       this->writeBlocks = (cbw->SCSICommandData[0x07] << 8) | cbw->
            SCSICommandData[0x08];
9       this->writtenBlocks = 0x00;
10
11      (*packetBuffer)->push_back(std::pair<__u8 *, __u64>(this->
            buildCSWPacket(cbw->Tag, MS_SCSI_COMMAND_Pass), MS_CSW_SIZE));
12      delete(cbw);
13  }
```

L'implementazione della classe `VirtualDrive` è leggermente diversa da quanto progettato: siccome l'I/O su file è molto lento, spesso si verificavano letture di dati prima che una precedente scrittura avesse finito di modificarli, creando così inconsistenze. Per evitare ciò, l'intero file `virtualDrive` è stato copiato in memoria, dove le operazioni di lettura/scrittura sono molto più rapide. Periodicamente, un thread si occupa di sincronizzare il file con la versione dei dati aggiornata presente in memoria.

```
1   #define WRITE_THRESHOLD 60
2   class VirtualDrive {
3   private:
```

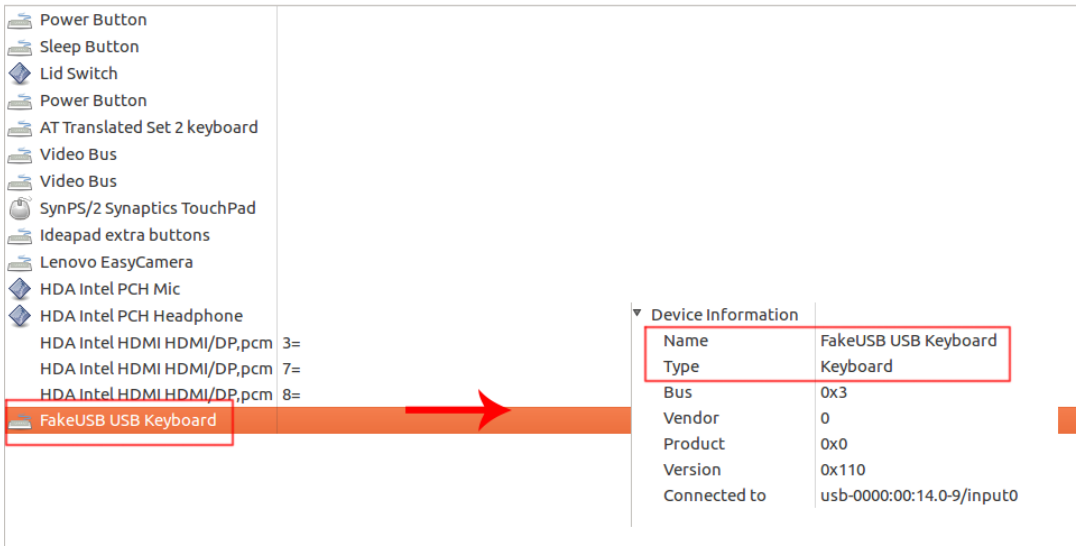
```
4       std::string driveLocation;
5       const __u32 blockSize = 512;
6       __u8 writeCount = 0;
7       bool writeLock = false;
8       void updateContent();
9       __u8 * driveContent = NULL;
10      void writeFile();
11      //[...]
12  public:
13      //[...]
14      __u8 * readBlock(__u64, __u64 *, __u32);
15      void writeBlock(__u8 *, __u64, __u64, __u32);
16  };
17
18  __u8 * VirtualDrive::readBlock(__u64 LBAFrom, __u64 * nBlocks, __u32
        offset) {
19      LBAFrom = (LBAFrom * blockSize) + (offset * blockSize);
20      (*nBlocks) = blockSize;
21
22      if(!this->driveContent) {
23          (*nBlocks) = 0;
24
25          return NULL;
26      }
27
28      __u8 * dataFromDrive = (__u8 *) calloc(blockSize, sizeof(__u8));
29
30      __u64 blocksCount = 0; __u64 LBACounter = LBAFrom;
31      /* Bounds checking per evitare overflows. */
32      while(blocksCount < blockSize && LBACounter <= this->realSize) {
33          dataFromDrive[blocksCount] = this->driveContent[LBACounter];
34          blocksCount++; LBACounter++;
35      }
36      return dataFromDrive;
37  }
38
39  void VirtualDrive::writeBlock(__u8 * data, __u64 LBAFrom, __u64 length,
        __u32 offset) {
```

```
40      while(this->writeLock); /* Non si puo' scrivere finche' il thread
            non ha finito di aggiornare i dati. */
41      LBAFrom = (LBAFrom * blockSize) + (offset * blockSize);
42
43      if(!this->driveContent)
44          return;
45
46      __u64 blocksCount = 0; __u64 LBACounter = LBAFrom;
47      while(blocksCount < length && LBACounter <= this->realSize) {
48          this->driveContent[LBACounter] = data[blocksCount];
49          blocksCount++; LBACounter++;
50      }
51      this->writeCount++;
52  }
53
54  /* Thread di aggiornamento */
55  void VirtualDrive::updateContent() {
56      while(!this->stopThread) {
57          if(this->writeCount > WRITE_THRESHOLD) {
58              this->writeLock = true;
59              this->writeFile();
60              this->writeLock = false;
61
62              this->writeCount = 0;
63          }
64      }
65      return;
66  }
67
68  /* Metodo che scrive i dati in memoria nel file virtualDrive */
69  void VirtualDrive::writeFile() {
70      FILE * virtualDrive = fopen(this->driveLocation.c_str(), "wb");
71      if(!virtualDrive)
72          return;
73      rewind(virtualDrive);
74      fwrite(this->driveContent, sizeof(__u8), this->realSize,
            virtualDrive);
75      fclose(virtualDrive);
```

```
76 }
77
78 //Continua...
```

4.4 Test della piattaforma tramite la simulazione di attacchi USB

Quanto realizzato è stato effettivamente testato. L'host enumera correttamente il dispositivo emulato:



Nel caso di periferiche HID, lo script dell'utente viene esattamente eseguito dalla piattaforma. La velocità di scrittura della finta tastiera è superiore a quella di qualsiasi umano, e consente di eseguire operazioni sull'host quasi istantaneamente. Anche il mouse mantiene le stesse prestazioni, sebbene gli spostamenti siano un pò imprecisi rispetto ai valori passati come parametro al comando.

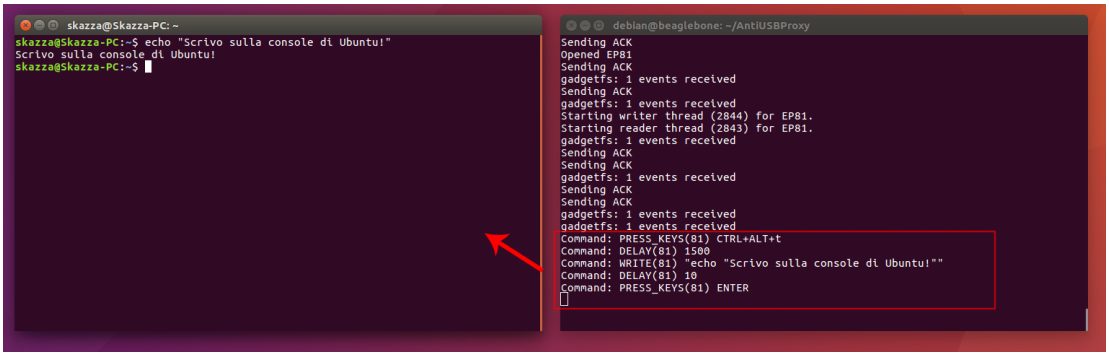
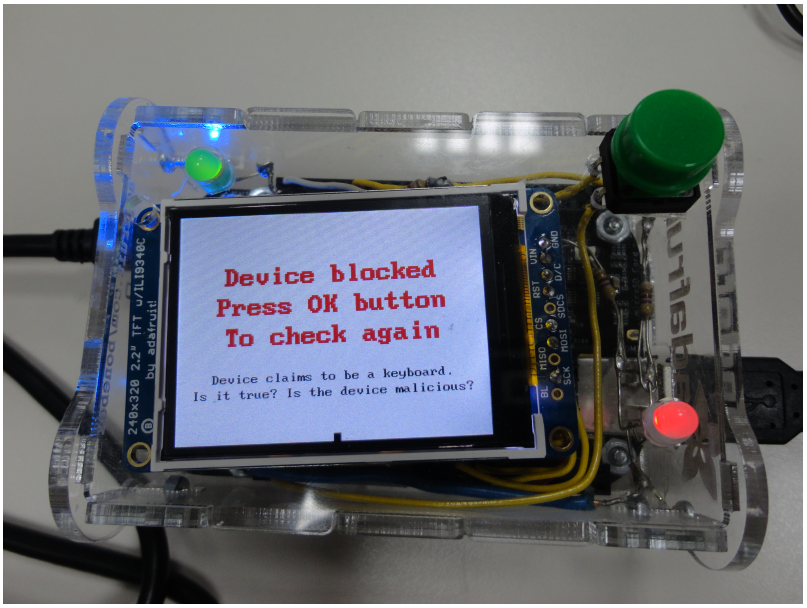
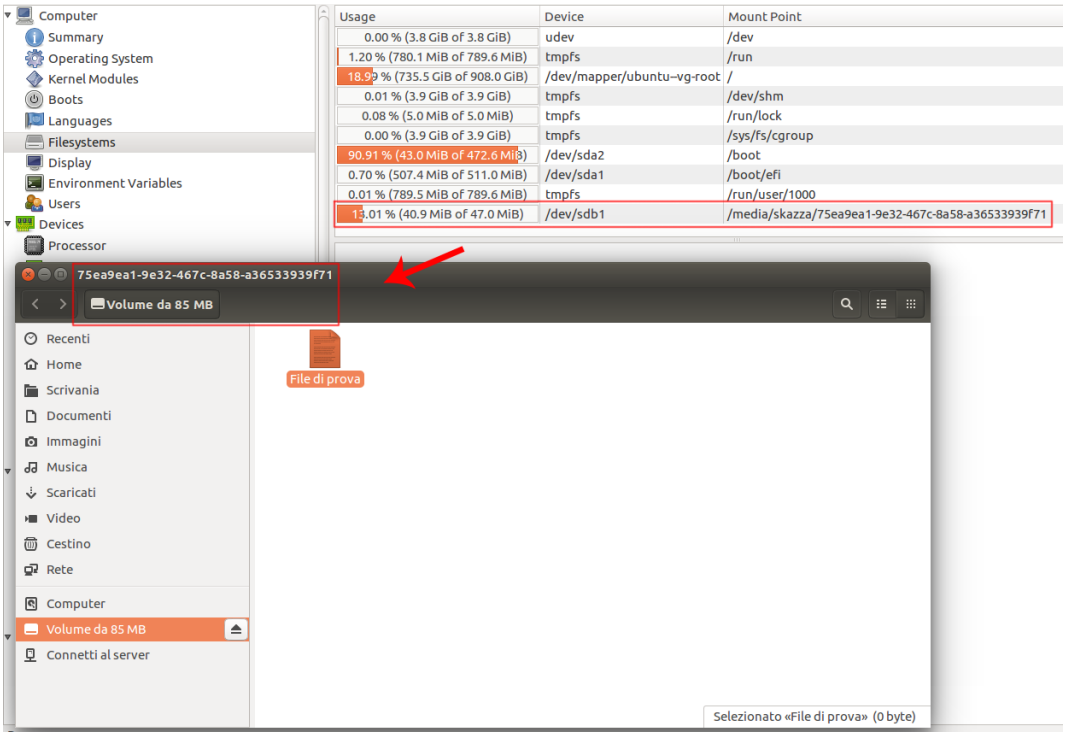


Figura 4.9: Esempio di script (scritto nella progettazione) eseguito dalla piattaforma.

USBCheckIn reagisce **correttamente** quando si simula il comportamento di un dispositivo HID che tenta di inviare comandi all'host prima di superare l'autenticazione, bloccando la periferica:



Nel caso dell'emulazione di mass storage, il dispositivo viene correttamente configurato e riconosciuto dal sistema operativo. La dimensione dello spazio di archiviazione e le partizioni corrispondono a quelle presenti nel file `virtualDrive` (di dimensione 85MB durante il test). La lettura e la scrittura di dati bulk procedono senza problemi.


Computer
Summary
Operating System
Kernel Modules
Boots
Languages
Filesystems
Display
Environment Variables
Users
Devices
Processor
Usage
Device
Mount Point
0.00 % (3.8 GiB of 3.8 GiB)
udev
/dev
1.20 % (780.1 MiB of 789.6 MiB)
tmpfs
/run
/dev/mapper/ubuntu--vg-root
/
0.01 % (3.9 GiB of 3.9 GiB)
tmpfs
/dev/shm
0.08 % (5.0 MiB of 5.0 MiB)
tmpfs
/run/lock
0.00 % (3.9 GiB of 3.9 GiB)
tmpfs
/sys/fs/cgroup
/dev/sda2
/boot
0.70 % (507.4 MiB of 511.0 MiB)
/dev/sda1
/boot/efi
0.01 % (789.5 MiB of 789.6 MiB)
tmpfs
/run/user/1000
/dev/sdb1
/media/skazza/75ea9ea1-9e32-467c-8a58-a36533939f71
75ea9ea1-9e32-467c-8a58-a36533939f71
Volume da 85 MB
Recenti
Home
Scrivania
Documenti
Immagini
Musica
Scaricati
Video
Cestino
Rete
Computer
Volume da 85 MB
Connetti al server
File di prova
Selezionato «File di prova» (0 byte)