Fachhochschule
Münster University of
Applied Sciences

Fachbereich
Elektrotechnik und Informatik

Konzeption und Implementierung einer QEMU- und KVM-basierten USB-Fuzzing Infrastruktur

Sergej Schumilo

Erstprüfer: Prof. Dr.-Ing. Sebastian Schinzel, Fachhochschule Münster

Fachlicher Betreuer: Hendrik Schwartke, OpenSource Training Ralf Spenneberg

Eidesstattliche Erklärung

Ich versichere, dass ich diese schriftliche Arbeit selbständig angefertigt, alle Hilfen und Hilfsmittel angegeben und alle wörtlich oder im Sinne nach aus Veröffentlichungen oder anderen Quellen, insbesondere dem Internet entnommenen Inhalte, kenntlich gemacht habe. Steinfurt, 14. Oktober 2014

____

Unterschrift

Abstract

Die Gefahr durch Angriffe mittels USB-Hardware, ist in den letzten Jahren immer wieder aufgezeigt worden. Eine umfassende, systematische Suche dieser Schwachstelle war jedoch bisher sehr aufwendig. Das in dieser Bachelorarbeit entwickelte und behandelte Framework, ermöglicht ein systematisches und hochperformantes USB-Fuzzing der USB-Treiber verschiedenster Betriebssysteme. Realisiert wird dies durch den massiven, parallelen Einsatz von virtuellen Maschinen. Das Framework dokumentiert gefundene Schwachstellen in einem reproduzierbaren Format und erlaubt es somit Treiber-Entwicklern und Sicherheitsforschern diese zu reproduzieren und zu analysieren.

Das entwickelte Framework wird auf der IT-Sicherheits-Konferenz „Black Hat Europe 2014“ veröffentlicht und ist Teil der Präsentation „Don’t trust your USB! How to find bugs in USB device drivers“.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Verwandte Arbeiten zum Thema USB-Fuzzing . . . 2
1.3 Aufbau der Arbeit . . . 2

2 USB-Grundlagen 3
2.1 Abstraktion . . . 3
2.2 Standard-Requests . . . 4
2.3 USB-Enumeration . . . 5
2.4 Deskriptoren . . . 6
2.4.1 Device Descriptor . . . 7
2.4.2 Configuration Descriptor . . . 9
2.4.3 Interface Descriptor . . . 10
2.4.4 Endpoint Descriptor . . . 11
2.4.5 String Descriptor . . . 13
2.4.6 Weitere spezifische Deskriptoren . . . 13
2.5 Endpoints . . . 14
2.6 Transfertypen . . . 14
2.7 Host-Controller . . . 15

3 USB-Network-Redirection 16
3.1 usbredir . . . 16
3.2 Protokoll . . . 16
3.3 usbredirserver . . . 18
3.4 USB-Abstraktion . . . 18

4 QEMU / KVM 19
4.1 QEMU . . . 19
4.2 KVM . . . 19
4.3 QEMU und die Kommandozeile . . . 20
4.4 Image-Dateien . . . 20
4.4.1 RAW . . . 20
4.4.2 QCOW2 . . . 20
4.4.3 Snapshots . . . 21
4.5 qemu-img . . . 21
4.6 USB-Redirection-Interface . . . 21
4.6.1 Kompilierung . . . 22
4.7 QEMU Monitor . . . 23

5 Fuzzing 24
5.1 Grundlegender Aufbau . . . 24
5.2 Komplexität der Datengenerierung . . . 24
5.3 USB-Fuzzing . . . 24
5.3.1 Field-Fuzzing . . . 25
5.3.2 Device-Descriptor-Fuzzing . . . 25
5.4 Herangehensweisen . . . 26

5.4.1 Man-in-the-Middle 26
5.4.2 Emulation von USB-Geräten 26

6 vUSBf Framework **27**
6.1 Fuzzing-Methoden 27
6.1.1 Live-Fuzzing 27
6.1.2 Record- / Replay-Fuzzing 29
6.1.3 Fuzzing durch Emulation 30
6.2 vUSBf Architektur 31
6.3 USB-Emulation 32
6.3.1 USB-Emulation Architektur 32
6.3.2 Redir-Emulator 33
6.3.3 USB-Enumeration-Emulator 33
6.3.4 HID Emulator 33
6.3.5 Weitere Emulatoren 33
6.4 QEMU/KVM Abstraktion 34
6.4.1 QEMU-Controller 34
6.4.2 Überwachung und Kontrolle 34
6.4.3 Starten von QEMU-Prozessen 34
6.4.4 Monitoring 35
6.4.5 Image Management 36
6.5 Parallelisierung 38
6.5.1 Multiprocessing 38
6.5.2 Architektur 38
6.5.3 Prozesskommunikation 39
6.6 Clustering 41
6.6.1 Architektur 41
6.6.2 Clustering-Protokoll 41
6.7 Dynamische Testcase-Generierung 44
6.7.1 Verknüpfungen 44
6.7.2 Fuzzing-Instruktion 45
6.7.3 XML-Beschreibung 46

7 Ergebnisse **48**
7.1 Performance 48
7.2 Fehler in USB-Geräte-Treibern 49
7.2.1 Fehler reproduzieren 50
7.2.2 Auswertung 52
7.2.3 KGDB Debugging 53

8 Fazit **54**

9 Ausblick **55**
9.1 Facedancer-Interface 55
9.2 Arduino-Binary Generierung 55
9.3 Microsoft Windows Unterstützung 55
9.4 Bug Reports 56

Abbildungsverzeichnis

1 USB-Abstraktion 4
2 Aufbau eines Standard-Request 4
3 USB-Deskriptoren Hierarchie 7
4 Aufbau eines Device Descriptor 7
5 Aufbau eines Configuration Descriptor 9
6 bmAttribut-Bitfeld (Configuration Descriptor) 10
7 Aufbau eines Interface Descriptor 10
8 Aufbau eines Endpoint Descriptor 11
9 bmAttribut-Bitfeld (Endpoint Descriptor) 12
10 Aufbau eines String Descriptor 13
11 usbredir Header 17
12 Aufbau usbredirserver 18
13 Type 1 und Type 2 Hypervisor 19
14 Aufbau des USB-Redirection-Interface 21
15 Ungültiger String Descriptor 25
16 Device-Descriptor-Fuzzing beispielhafte Anwendung 25
17 Bildschirmfoto von vUSBf 27
18 Aufbau des usbredirserver-proxy 27
19 Ausschnitt aus der dekodierten USB-Mitschnitt-Datei 28
20 USB-Fuzzing der Daten in Richtung QEMU. 29
21 Aufbau des usb-redir-proxy 29
22 Hierarchischer Aufbau von USB-Emulatoren 30
23 Architektur des vUSBf-Framework 31
24 Klassendiagramm der Klasse „usb-emulator.py“ 32
25 Beispielhafte vUSBf-Konfiguration für Ubuntu 14.04 Desktop 35
26 Einfacher Aufbau von QEMU und einer Image-Datei 36
27 Verwendung von QCOW2-Overlay Dateien 36
28 Optimale Speichernutzung durch Overlays und einer RAM-Datei 37
29 Architektur des Multiprocessing im vUSBf-Framework 39
30 Kommunikation zwischen den Akteuren beim Clustering 41
31 Protokoll-Header des Clustering-Protokolls 42
32 Beispielhafter Message Flow zwischen einem Server und Client 43
33 Beispielhafte Verknüpfung von mehreren Testpools 44
34 Performance im Vergleich 48
35 Performance in Relation zur Laufzeit 49
36 Ausschnitt aus einer vUSBf Log-Datei 50
37 Payload-Archiv der aktuellsten Version 51
38 Selektierter Auszug aus den Log-Dateien (Ubuntu 14.04 LTS Desktop) 52

Tabellenverzeichnis

1 Auflistung aller Standard-Requests 5
2 Auflistung von bekannten USB-Deskriptoren 6
3 Auflistung der zulässigen Usage-Werte für das bmAttribut-Bitfeld (Endpoint Descriptor) 12
4 Auflistung der zulässigen Synchronization-Werte für das bmAttribut-Bitfeld (Endpoint Descriptor) 12
5 Auflistung der zulässigen Transfer-Werte für das bmAttribut-Bitfeld (Endpoint Descriptor) 12
6 Auflistung von relevanten USB-Redirection-Paketen 17
7 Auflistung aller von QEMU emulierbaren USB-Host-Controller 23
8 Auflistung aller Fuzzing-Aktionen 46

1 Einleitung

Der Universal Serial Bus, kurz USB, ist ein serielles Bussystem und ein Standard für die Kommunikation zwischen verschiedenster Peripheriegeräten und dem Host-Computer. Anschlüsse für USB-Geräte finden sich in allen Bereichen der heutigen Technik wieder. Ob nun am eigenen Computer, der Spielkonsole, dem Drucker oder in der automobilen Unterhaltungselektronik, überall findet der USB-Standard Anwendung. Umso wichtiger ist es, sich die Frage zu stellen, in wie fern die USB-Implementierungen in diesen Systemen den erwarteten Sicherheitsaspekten genügen.

Die Gefahren durch Angriffe mittels USB-Hardware sind in den letzten Jahren immer wieder aufgezeigt worden. Der USB stellt schon seit Jahren ein attraktives Ziel für Angreifer und ihre Exploits dar. Bekannte Angriffsvektoren waren in der Vergangenheit die Ausnutzung der Auto-Run Funktionalität älterer Windows-Versionen[vgl. 1] oder das automatisierte Absetzen von bösartigen Kommandos über die Emulation eines USB-HID-Gerätes (Human Interface Device)[vgl. 6]. Spätestens seit den aktuellsten Veröffentlichungen, wie zum Beispiel BadUSB[vgl. 14], fand eine Sensibilisierung für konzeptionelle Sicherheitsschwachstellen statt. Einige Meinungen gehen bereits soweit, die Sicherheitskonzepte des USB-Standard als komplett unzureichend zu bezeichnen[vgl. 9].

Darüber hinaus existiert ein weiterer Angriffsvektor. Dieser basiert auf der Ausnutzbarkeit von Implementierungsschwachstellen der, durch das Betriebssystem oder Dritthersteller bereitgestellten, Treiber. Da der USB Hot-Plug unterstützt, wählt das jeweilige Betriebssystem nach dem Einstecken und der anfänglichen Initialisierung des USB-Geräts, den jeweiligen generischen USB-Treiber anhand einer USB-Klasse (HID, Printer etc.) oder einen speziellen Geräte-Treiber. Das Betriebssystem übergibt nach der anfänglichen Initialisierung die Kontrolle an den ausgewählten USB-Treiber. Das Sicherheitsmodell impliziert jedoch, dass die Implementierung des geladenen Treibers über keine Sicherheitsschwachstellen verfügt. Das dieses Vertrauen bei etlichen USB-Treibern verschiedenster Betriebssysteme gebrochen wurde, zeigen verschiedenste Forschungen und Treiber-Untersuchungen dar. Problematisch ist hierbei, dass im Falle einer Kompromittierung des angegriffenen Systems, der Angreifer die Möglichkeit hat Schadcode im Kernel-Space auszuführen. Damit stehen ihm wesentlich mehr Privilegien, als bei den anderen oben aufgezeigten Vektoren, zur Verfügung.

Für die Untersuchung der Treiber von USB-Geräten auf Sicherheitslücken hat sich in der Vergangenheit die Fuzzing-Methode bewährt[vgl. 8]. Beim USB-Fuzzing wird entweder versucht, mit Hilfe eines Man-In-The-Middle-Ansatzes den USB-Traffic gezielt zu verändern oder durch die Implementierung eines USB-Emulators ein Gerät bereitzustellen, welches an ein Zielsystem unerwartete oder (teilweise) veränderte Daten schickt, um Abstürze oder Fehlverhalten des Zielsystems zu provozieren.

1.1 Motivation

In der Vergangenheit wurde keine Lösung vorgestellt, welche eine systematische Untersuchung aller verfügbaren USB-Treiber automatisiert auf Sicherheitsschwachstellen erlaubt. Problematisch ist hierbei die große Anzahl der zu untersuchenden USB-Treiber sowie die Kombination aller veränderbaren Felder der jeweiligen USB-Paketen. Durch die langsame Ausführgeschwindigkeit von bekannten Hardware-Lösungen[vgl. 18], schien eine systematische Untersuchung nicht praktikabel bzw. realisierbar.

Diese Bachelorarbeit beschreibt die Entwicklung und die hieraus resultierenden Ergebnisse eines

USB-Fuzzing-Frameworks namens vUSBf (Akronym für „**v**irtual **USB** **f**uzzer“). Das Framework versucht durch den Einsatz von massiver Parallelisierung, die Nachteile von älteren USB-Fuzzing-Lösungen zu umgehen. Außerdem ermöglicht das Framework, auf Grund der Verwendung von virtuellen Maschinen und speziellen Konzepten, die automatisierte Dokumentation der gefundenen Schwachstellen in einem reproduzierbaren Format. Somit lassen sich gefundene Fehler immer reproduzieren. Dies kann mit einem echten und physikalisch verfügbaren System nicht immer gewährleistet werden.

1.2 Verwandte Arbeiten zum Thema USB-Fuzzing

Bereits vor dieser Arbeit gab es verschiedene Ansätze und Herangehensweisen im Bezug auf das USB-Fuzzing. Frühere Arbeiten nutzten zum USB-Fuzzing programmierbare Mikrocontroller-Boards, welche über eine USB-Funktionalität verfügten[vgl. 10]. Hierbei ist besonders das Facedancer-Board[vgl. 18] zu nennen. Das Facedancer-Board, welches von Sergey Bratus und Travis Goodspeed entworfen wurde, ermöglicht das Vortäuschen eines beliebigen USB-Gerätes für ein Host-System und darüber hinaus auch das USB-Fuzzing. Basierend auf dem Facedancer-Board und dessen API, wurden automatisierte Software-Lösungen entwickelt. Eine bekannte USB-Fuzzing Software ist zum Beispiel „umap“, welche von Andy Davis 2013 vorgestellt wurde[vgl. 7]. Die Software ermöglicht das USB-Fuzzing mithilfe des Facedancer-Boards und der Emulation verschiedenster USB-Klassen.

Des Weiteren gab es in der Vergangenheit auch Veröffentlichungen, die das USB-Fuzzing in virtuellen Umgebungen beschreiben[vgl. 16]. Trotzdem wurde keine Lösung aufgezeigt, mit welcher es möglich ist, das USB-Fuzzing möglichst systematisch, umfassend und hoch reproduzierbar in einem praktischen Zeitrahmen zu gestalten.

1.3 Aufbau der Arbeit

Diese Arbeit ist unterteilt in vier Abschnitte:

(I) Der erste Teil der Arbeit vermittelt dem Leser die notwendigen Grundlagen und relevante Zusammenhänge aufgezeigt. Hierzu gehören die Kapitel 2, 3, 4 und 5.

(II) Der zweite Teil dieser Bachelorarbeit beschreibt den Entwicklungsprozess des vUSBf-Frameworks. Es werden Aspekte wie die verwendete Architektur und die Implementierung von wichtigen Funktionalitäten beschrieben. Dieser Teil entspricht dem gesammten Kapitel 6.

(III) Im Kapitel 7 werden die Ergebnisse dieser Bachelorarbeit präsentiert.

(IV) Zum Schluss wird noch ein Überblick über zukünftige geplante Funktionalitäten und Möglichkeiten in Form einer Aussicht gegeben. Außerdem werden zum Schluss noch einmal die wichtigsten Aspekte und Punkte dieser Arbeit in Form eines Fazits erörtert.

2 USB-Grundlagen

Dieser Teil der Arbeit dient dem Vermitteln der Grundlagen in der Funktionalität des Universal Serial Bus. Der Fokus liegt hierbei auf der Interaktion der USB-Peripherie mit den entsprechenden Treibern des verwendeten Systems. Dabei wird, bis auf wenige Ausnahmen, nicht auf die physikalischen Eigenschaften und das Low-Level USB-Protokoll eingegangen, da dieses keine Relevanz für die Entwicklung des USB-Fuzzing-Frameworks hatte.

Der interessierte Leser sei zusätzlich auf die offizielle USB 2.0 Dokumentation verwiesen. Alle nachfolgenden Informationen zum USB-Standard sind der offiziellen USB 2.0 Dokumentation entnommen [vgl. 20].

2.1 Abstraktion

Der Universal Serial Bus arbeitet auf verschiedenen Abstraktionsebenen. Diese lassen sich vereinfacht in drei verschiedene Layer unterteilen:

USB Bus Interface Layer:
Dieser Layer entspricht der physikalischen Datenübertragung. Auf der Host-Seite befindet sich hier der Host-Controller (siehe Kapitel 2.7.) und der physikalische USB-Anschluss wieder. Auf der Geräteseite findet sich der physikalische Anschluss, sowie spezielle Mikrocontroller wieder.

USB Device Layer:
Dieser Layer entspricht der logischen Datenübertragung in Form des USB-Protokolls. Das Host-System verfügt auf diesem Layer über den Host-Controller-Treiber, sowie dem USB-Core-Treiber für alle USB Basis-Funktionalitäten. Der Host-Controller-Treiber dient als Schnittstelle zum Host-Controller und der USB-Core-Treiber dient als Schnittstelle zum Function-Layer. Das USB-Gerät verfügt auf diesem Layer über eine Abstraktionsschicht zwischen dem Interface und Function Layer. Die USB-Enumeration (siehe Kapitel 2.3.) findet auf diesem Layer statt.

USB Function Layer:
Dieser Layer entspricht der logischen Datenübertragung über die USB-Endpoints (siehe Kapitel 2.5.). Die entsprechenden Treiber für klassenspezifische bzw. herstellerspezifische USB-Geräte finden sich hier wieder. Das USB-Gerät implementiert auf diesem Layer die eigentliche Funktionalität des Geräts.

Die genannten Layer lassen sich wie folgt auf das Host-System und ein USB-Peripheriegerät abstrahieren:

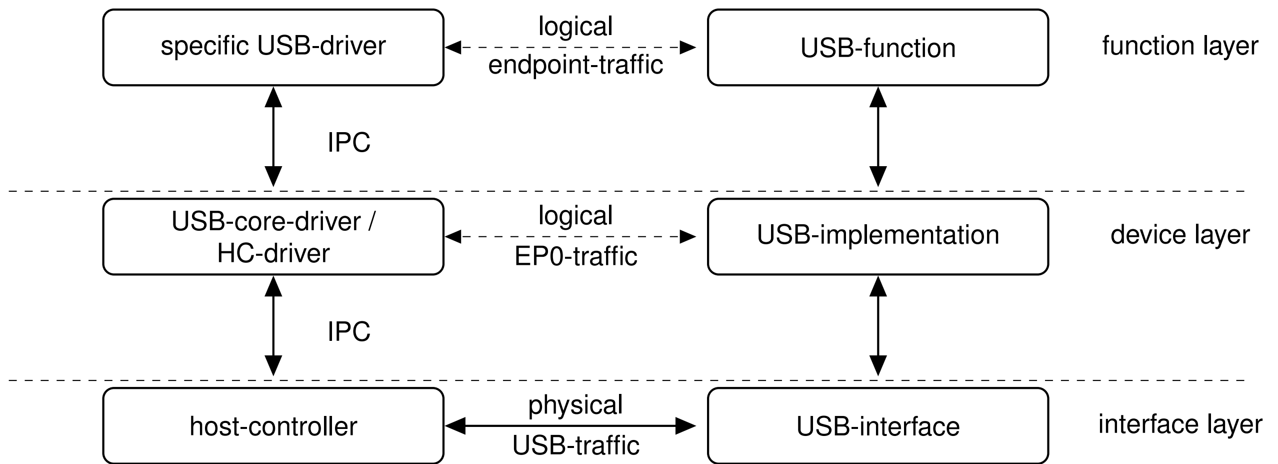

Abbildung 1: USB-Abstraktion[1]

Die Grafik verdeutlicht, dass die verschieden Layer untereinander immer logisch mit ihrem Pendant kommunizieren, wobei die Daten bis zur untersten Schicht durchgereicht und anschließend über das eigentliche USB-Kabel verschickt werden. Somit ist für den Entwickler eines USB-Treibers auf dem „Function-Layer“ irrelevant, wie die darunter liegenden Layer genau funktionieren.
Die Aufteilung der Layer ist vergleichbar mit dem OSI-Modell bei der Netzwerk-Übertragung.

2.2 Standard-Requests

Der Universal Serial Bus arbeitet auf der USB-Device-Layer mit sogenannten USB-Requests. USB-Standard-Requests stellen eine Teilmenge der USB-Requests dar und werden primär bei der USB-Enumeration verwendet. Jedes USB-Gerät muss mindestens auf die USB-Standard-Requests antworten. Anderenfalls wird dieses Gerät unter Umständen von Host-System nicht erkannt. Somit muss die Firmware des USB-Gerätes alle USB-Standard-Requests verarbeiten können.
USB-Requests werden in Form von sogenannten Setup-Paketen über den bidirektionalen Endpoint 0 verschickt. Dieser wird für die Enumeration verwendet. Setup-Pakete sind mindestens 8 Byte groß und haben den folgenden Aufbau:

Abbildung 2: Aufbau eines Standard-Request

bmRequestType:
Bitmap-Feld, welches den Request-Typen im nachfolgenden Teil des Pakets spezifiziert.

bRequest:
Dieses Feld enthält den eigentlichen Request, wobei die Bedeutung abhängig vom Request-Typen ist.

[1] Singgemäß entnommen aus [vgl. 20].

wValue:
Request-abhängige Daten.

wIndex:
Request-abhängige Daten.

wLength:
Anzahl der zusätzlich zu übertragenden Bytes.

Die USB-2.0 Spezifikation kennt die folgenden USB-Standard-Requests:

bRequest	Bezeichnung
0	GET_STATUS
1	CLEAR_FEATURE
2	Reserviert
3	SET_FEATURE
4	Reserviert
5	SET_ADDRESS
6	GET_DESCRIPTOR
7	SET_DESCRIPTOR
8	GET_CONFIGURATION
9	SET_CONFIGURATION
10	GET_INTERFACE
11	SET_INTERFACE
12	SYNC_FRAME

Tabelle 1: Auflistung aller Standard-Requests

Darüberhinaus existieren noch weitere klassenspezifische bzw. treiberspezifische USB-Requests.

2.3 USB-Enumeration

Die USB-Enumeration ist die anfängliche, generische Initialisierung eines USB-Gerätes am USB-Host. Diese folgt unmittelbar nach dem Einstecken des USB-Gerätes in den USB-Port. Die „Plug and Play"-Funktionalität des USB wird erst durch die Enumeration und den hiermit verbundenen generischen Datenaustausch ermöglicht.
Bei der Enumeration werden mit Hilfe sogenannter Deskriptoren, welche über den logischen Datenkanal Endpoint 0 übermittelt werden, dem Host-System alle benötigten Informationen übermittelt, um das direkte Betreiben des USB-Geräts am Host-System zu ermöglichen. Mit Hilfe der übermittelten Informationen kann das Host-System entsprechende Treiber laden und daraufhin mit dem eigentlichen gerätespezifischen Datentransfer beginnen.
Prinzipiell läuft die Enumeration eines USB-Geräts immer nach dem selben Muster ab. Hierzu werden nach dem Aufbau der Verbindung, mit Hilfe von Standard-Device-Requests, Anfragen über den Endpoint 0 an das USB-Gerät geschickt. Das USB-Gerät antwortet entsprechend auf diese Anfragen und übermittelt die angefragten Informationen in Form von „Control"-Pakten.

2.4 Deskriptoren

Deskriptoren dienen der Identifikation und Beschreibung der Eigenschaften von USB-Geräten. Hierzu existieren verschiedene Deskriptoren. Für die Enumeration von USB 1.0, 1.1 und 2.0[2] Geräten werden die folgenden Deskriptoren verwendet:

bDescriptorType	Deskriptor
0x01	Device Descriptor
0x02	Configuration Descriptor
0x03	String Descriptor
0x04	Interface Descriptor
0x05	Endpoint Descriptor
...	...
0x21	HID Descriptor
0x22	HID Report Descriptor
...	...

Tabelle 2: Auflistung von bekannten USB-Deskriptoren

Das USB Protokoll kennt darüber hinaus eine Hierarchie der Deskriptoren. Vereinfacht lässt sich sagen, dass zu jedem USB-Gerät genau ein *Device Descriptor* existiert und jeder *Device Descriptor* eine oder mehrere Konfigurationen kennt. Jeder *Configuration Descriptor*, welcher einer Konfiguration entspricht, hat außerdem noch mehrere *Interface Descriptors* sowie diese noch mehrere *Endpoint Descriptors*.[3] Die folgende Grafik dient der Veranschaulichung:

[2] USB2.0 verwendet für die Identifikation der Bus-Geschwindigkeit noch einen Weitere Deskriptor. Dieser wird *Device Qualifier Descriptor* genannt.

[3] Es werden in Abhängigkeit zu der verwendeten USB-Geräteklasse noch weitere Deskriptoren verwendet. Außerdem existieren für USB 2.0 und 3.0 Geräte noch weitere spezifische Deskriptoren.

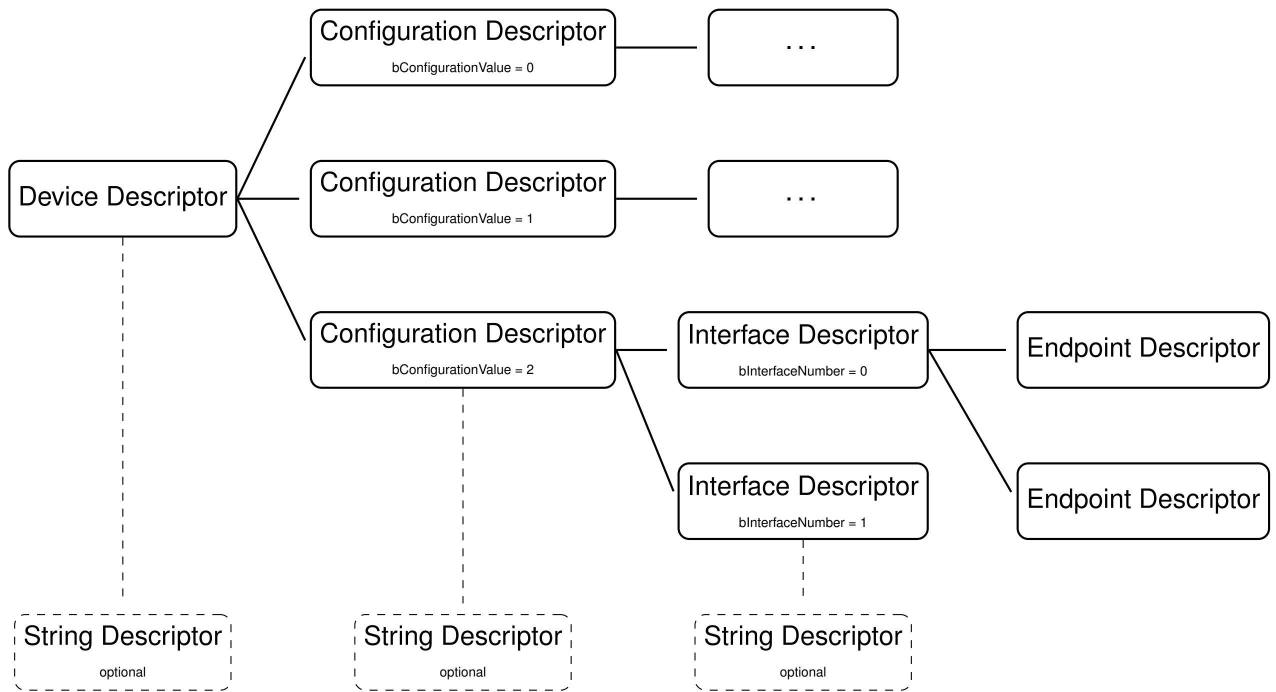

Abbildung 3: USB-Deskriptoren Hierarchie[4]

2.4.1 Device Descriptor

Jedes USB-Gerät hat genau einen *Device Descriptor*, welcher dazu dient alle benötigten physikalischen und logischen Attribute zu beschreiben. Dieser hat eine feste Größe von 18 Bytes und wird als erstes vom Host-System bei der Enumeration angefordert. Anhand der Informationen im *Device Descriptor* kann das Host-System weitere Deskriptoren für die Enumeration anfordern.

Die folgende Grafik repräsentiert einen *Device Descriptor*:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
bLength	bDescriptorType 0x1	bcdUSB	
bDeviceClass	bDeviceSubclass	bDeviceProtocol	bMaxPacketSize
idVendor		idProduct	
bcdDevice		iManufacturer	iProduct
iSerial	bNumConfigurations		

Abbildung 4: Aufbau eines *Device Descriptor*

bLength:

Größe des Deskriptors in Bytes (üblicherweise sind es bei einem *Device Descriptor* immer 18 Byte).

[4]Entnommen aus [vgl. 12].

bDescriptorType:

Konstanter Wert welcher den Deskriptor-Typen spezifiziert. Bei einem *Device Descriptor* ist dieser Wert 0x1.

bcdUSB:

Der Wert gibt die Version der USB-Spezifikation an. Zum Beispiel entspricht der Wert 0x2000 der Version 2.0.

bDeviceClass:

Diese Feld gibt die gewünschte USB-Klasse an. Dazu gehört zum Beispiel die generische Human Interface Device-Klasse oder die „Vendor-Specific“-Klasse für Geräte, die spezifische bzw. herstellerspezifische Treiber verlangen.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

bDeviceSubclass:

Dieses Feld spezifiziert die USB-Geräte-Klasse genauer (z.B. „Mass-Storage“-Klasse über die Subklasse SCSI).

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

bDeviceProtocol:

Spezifiziert das gewünschte Protokoll für die gewählte Geräte- und Subklasse (z.B. Mass-Storage-Klasse über die Subklasse SCSI und dem Protokoll „Bulk-Only“).

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

idVendor:

Dieses Feld gibt die eindeutige Hersteller-Identifikationsnummer an, welche vom USB-Implementers-Forum[5] vergeben wird.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

idProduct:

Dieses Feld gibt die Produkt-Identifikationsnummer an. In Kombination mit der Hersteller-Identifikationsnummer ist diese eindeutig und kann vom Hersteller frei gewählt werden.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

bcdDevice:

Binär-codierte Dezimalzahl für die Versionsbezeichnung des Gerätes.

iManufacturer:

Indexnummer für mögliche *String Descriptors*, die dazu dienen den Herstellernamne und weiter Information in Form von Zeichenketten zu übergeben.

iProduct:

Indexnummer für mögliche *String Descriptors*, die dazu dienen die Produktbezeichnung zu übergeben.

[5] http://www.usb.org

iSerial:
Indexnummer für mögliche *String Descriptors*, die dazu dienen die Geräte-Serien-Nummer zu übergeben.

bNumConfigurations:
Dieser Wert gibt die Anzahl aller für dieses Gerät verfügbaren *Configuration Descriptors* an.

2.4.2 Configuration Descriptor

Ein USB-Gerät kann über mehrere mögliche Konfigurationen verfügen. Diese Konfigurationen werden mit je einem *Configuration Descriptor* beschrieben und enthalten eine bestimmte Menge von *Interface Descriptors* sowie weitere hierarchisch unter dem *Configuration Descriptor* liegende generische und klassenspezifische Deskriptoren (siehe Abb. 3.).

Die folgende Grafik repräsentiert einen *Configuration Descriptor*:

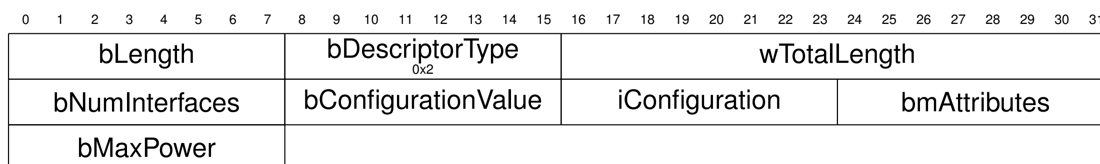

Abbildung 5: Aufbau eines *Configuration Descriptor*

bLength:
Größe des Deskriptors in Byte (üblicherweise sind es bei einem *Configuration Descriptor* 9 Byte).

bDescriptorType:
Konstanter Wert welcher den Deskriptor-Typen spezifiziert. Bei einem *Configuration Descriptor* ist dieser Wert immer 0x2.

wTotalLength:
Größe der zu dieser Konfiguration gehörenden Deskriptoren in Bytes (ausgenommen des *Configuration Descriptor* selbst).

bNumInterfaces:
Dieser Wert gibt die Anzahl aller für diese Konfiguration verfügbaren *Interface Descriptors* an.

bConfigurationValue:
Eindeutig nummerisch fortlaufende Indexnummer, welcher diese Konfiguration indiziert.

iConfiguration:
Indexnummer für mögliche *String Descriptors*, die dazu dienen die Konfigurationsbeschreibung zu übergeben.

bmAttributes:

Bitfeld, welches bestimmte Attribute für die Konfiguration enthält. In diesem Fall nur die Option „Remote-Wakeup“ und „Self-Powered“ :

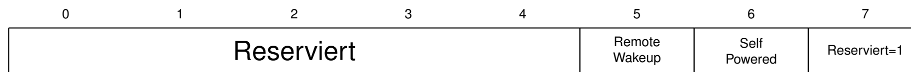

Abbildung 6: bmAttribut-Bitfeld (*Configuration Descriptor*)

bMaxPower:

Dieses Feld gibt die gewünschte Stromaufnahme in 2 mA-Einheiten an.

2.4.3 Interface Descriptor

Ein USB-Gerät kann ebenfalls über mehrere logische Funktionen verfügen, wobei jede Funktion über einen eigenen *Interface Descriptor* verfügt, der die Funktion beschreibt. Als Beispiel dient hierbei eine Webcam, welche über je ein Interface das Videobild und das Audiosignal überträgt.

Die folgende Grafik repräsentiert einen *Interface Descriptor*:

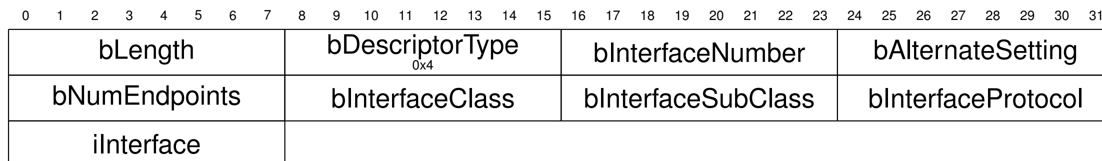

Abbildung 7: Aufbau eines *Interface Descriptor*

bLength:

Größe des Deskriptors in Byte (üblicherweise sind es bei einem *Interface Descriptor* 9 Byte).

bDescriptorType:

Konstanter Wert welcher den Deskriptor-Typen spezifiziert. Bei einem *Interface Descriptor* ist dieser Wert immer 0x4.

bInterfaceNumber:

Eindeutig nummerisch fortlaufende Indexnummer, welche diesem *Interface Descriptor* indiziert.

bAlternateSetting:

Ein Interface kann mehrere alternative Einstellungen haben. Es können daher mehrere *Interface Descriptors* mit der selben „bInterfaceNumber“-Nummer folgen, aber mit einer unterschiedlichen „bAlternateSetting“-Nummer. Der Wert in diesem Feld wird bei mehreren alternativen Einstellungen beginnend bei 0 inkrementiert.

bNumEndpoints:

Die Anzahl zu diesem *Interface Descriptor* gehörenden Endpoints, bzw. *Endpoint Descriptors*, wird hier angegeben.

bInterfaceClass:

Es lassen sich alternativ auch im *Interface Descriptor* die „DeviceClass“ spezifizieren. Falls die „DeviceClass“ im Interface spezifiziert wird, so muss das Feld im *Device Descriptor* auf den Wert 0 (entspricht somit „Defined at Interface level“) gesetzt werden.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

bInterfaceSubClass:

Es lassen sich alternativ auch im *Interface Descriptor* die „DeviceSubClass“ spezifizieren.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

bInterfaceProtocol:

Es lassen sich alternativ auch im *Interface Descriptor* das „DeviceProtocol“ spezifizieren.

Dieser Wert ist für das Betriebssystem relevant bei der Wahl des passenden Treibers.

iInterface:

Indexnummer für mögliche *String Descriptors*, die dazu dienen die Interface-Bezeichnung zu übergeben.

2.4.4 Endpoint Descriptor

Endpoint Descriptors beschreiben die eigentlichen logischen Datenkanäle zwischen den Treibern und dem USB-Gerät, den sogenannten Endpoints (siehe dazu Kapitel 2.5.).

Die folgende Grafik repräsentiert einen *Endpoint Descriptor*:

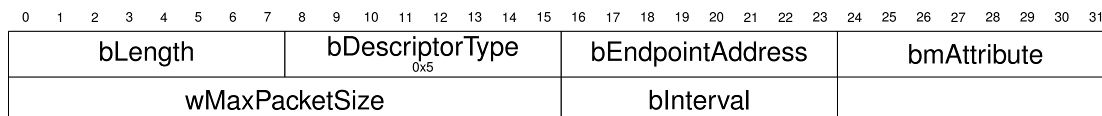

Abbildung 8: Aufbau eines *Endpoint Descriptor*

bLength:

Größe des Deskriptors in Byte (üblicherweise sind es bei einem Interface Endpoint 7 Byte).

bDescriptorType:

Konstanter Wert welcher den Deskriptor-Typen spezifiziert. Bei einem *Endpoint Descriptor* ist dieser Wert immer 0x5.

bEndpointAddress:

Eindeutige Adresse, welche den Endpoint bestimmt (siehe dazu Kapitel 2.5.)

bmAttribute:

Bitfeld, welches bestimmte Attribute für diesen Endpoint enthält:

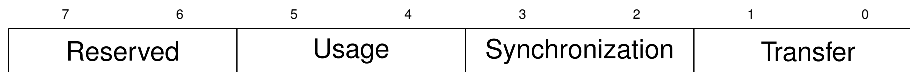

Abbildung 9: bmAttribut-Bitfeld (*Endpoint Descriptor*)

Reserved:

Reservierte Felder, welche mit 0 überschrieben werden müssen.

Usage:

00	Data Endpoint
01	Feedback Endpoint
10	Implicit feed
11	Reserved

Tabelle 3: Auflistung der zulässigen Usage-Werte für das bmAttribut-Bitfeld (*Endpoint Descriptor*)

Synchronization (nur für Isochronous-Transfer):

00	No Synchronization
01	Asynchronous
10	Adaptiv
11	Synchronous

Tabelle 4: Auflistung der zulässigen Synchronization-Werte für das bmAttribut-Bitfeld (*Endpoint Descriptor*)

Transfer:

00	Control-Transfer
01	Isochronous-Transfer
10	Bulk-Transfer
11	Interrupt-Transfer

Tabelle 5: Auflistung der zulässigen Transfer-Werte für das bmAttribut-Bitfeld (*Endpoint Descriptor*)

wMaxPacketSize:

Größe des FIFO-Buffers für den Endpoint.

bInterval:

Intervall-Angabe für das Polling bei Interrupt- und Isochronous-Datentransfer. Der Wert entspricht der Zeitangabe in Millisekunden.

2.4.5 String Descriptor

String Descriptors sind optional und können verwendet werden um den Gerätenamen, den Herstellernamen, die Seriennummer oder weitere Informationen zu übertragen. Hierzu werden die gewünschten Informationen Unicode-codiert in einem *String Descriptor* übertragen. Die Übertragung eines *String Descriptor* lässt sich im Linux-Kernel in den Kernel-Ausgaben mitverfolgen. Der folgende Ausschnitt zeigt eine beispielhafte Enumeration mit drei übertragenen *String Descriptors*:

```
[ 1913.372258] usb 1-1: new high-speed USB device number 43 using xhci_hcd
[ 1913.628632] usb 1-1: New USB device found, idVendor=07d0, idProduct=2001
[ 1913.631813] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 1913.635290] usb 1-1: Product: super_fast_usb_flash_drive
[ 1913.637844] usb 1-1: Manufacturer: a_usb_vendor
[ 1913.640606] usb 1-1: SerialNumber: a177728abvid22
[ 1914.159970] usb 1-1: USB disconnect, device number 43
```

Die folgende Grafik repräsentiert einen *String Descriptor*:

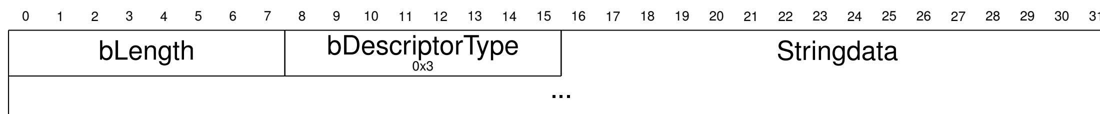

Abbildung 10: Aufbau eines *String Descriptor*

bLength:
Größe des Deskriptors in Byte (die Größe variiert und ist abhängig von der Größe des Feldes „Stringdata“).

bDescriptorType:
Konstanter Wert welcher den Deskriptor-Typen spezifiziert. Bei einem *String Descriptor* ist dieser Wert immer 0x5.

Stringdata:
Dieses Feld enthält eine Unicode-Zeichenkette, welche UTF-16 kodiert ist und somit pro Zeichen 2 Bytes groß sind. Daher beträgt auch die maximale Länge 126 Zeichen.

Außerdem lassen sich in *String Descriptors* spezielle „Language Identifiers“ übertragen. Diese ermöglichen es, sofern ein USB-Gerät über mehrere *String Descriptors* in verschiedenen Sprachen verfügt, dem Host-System eine Auswahl über die gewünschten *String Descriptors* treffen zu lassen. Diese sind ebenfalls zwei Byte groß und lassen sich ähnlich wie ein String aneinander reihen (es sind somit ebenfalls bis zu 126 „Language Identifiers“ pro Deskriptor möglich). Weitere Informationen sind in der offiziellen Dokumentation zu den „Language Identifiers“ zu finden[vgl. 21].

2.4.6 Weitere spezifische Deskriptoren

Es existieren darüber hinaus noch viele weitere Deskriptoren für verschiedene Anwendungen. Beispielsweise wird bei der USB-Enumeration von HID- oder USB-HUB[6]-Geräten spezifische USB-

[6]Ein USB-Hub erlaubt es den USB um weitere Anschlüsse zu erweitern.

Deskriptoren übertragen. Einige Hersteller nutzen auch die Möglichkeit, um eigene Deskriptoren zu definieren und diese bei der USB-Enumeration zu übertragen.

2.5 Endpoints

Endpoints stellen logische, unidirektionale Verbindungen dar und dienen als Kommunikationsschnittstelle in Form von Puffern für die Firmware eines USB-Gerätes und den USB-Treibern des Host-Systems. Eine Ausnahme stellt der Endpoint 0 dar. Für die Enumeration eines USB-Gerätes wird immer der Endpoint 0 mit der Adresse 0x00 verwendet. Dieser Endpoint ist der einzige, welcher bidirektional arbeitet. Außerdem ist sein Transfertyp immer „Control“.

Es existieren darüber hinaus bis zu 30 weitere Endpoints, mit frei wählbaren Transfertypen. Dazu gehören je 15 IN[7]- und 15 OUT-Endpoints. Die tatsächliche Anzahl der zur Verfügung stehenden Endpoints ist durch die technische Unterstützung des USB-Gerätes limitiert.
Die Endpoints werden mit Hilfe von Adressen und einer Richtungsangabe definiert. Dabei stellen die Adressen von 0x01 bis 0x0f die 15 möglichen Endpoints für den OUT-Transfer dar und die Adressen 0x81 bis 0x8f die möglichen Endpoints für den IN-Transfer.

Die Limitierung von 15 IN-Endpoints und 15 OUT-Endpoints ergibt sich aus der Tatsache, dass für Adressierung von Endpoints ein 4-Bit Feld verwendet wurde, was eine Anzahl von 32 möglichen Endpoints ergibt. Zählt man für den einzigen bidirektionalen Endpoint 0 noch zwei weitere Endpoints hinzu, da dieser technisch gesehen über zwei Adressen verfügt, kommt man auf die Anzahl von 32.

2.6 Transfertypen

Es sind 4 verschiedene Transfertypen in der USB 2.0 Spezifikation definiert:

Control:
Diese Transfertyp ist für kleine Pakete gedacht, welche für die Enumeration und der Steuerung verwendet werden. Der Endpoint 0 verschickt seine Daten immer über diesen Transfertypen. Control-Pakete haben eine Mindestgröße von 8 Byte, da dies der Größe eines USB-Request entspricht. Weitere möglichen Größen sind für Full-Speed[8] Geräte 8, 16, 32 und 64 Byte. Low-Speed[9] Geräte sind auf 8 Byte limitiert und High-Speed[10] Geräte müssen zwingend eine Größe von 64 Byte haben.

Interrupt:
Dieser Transfertyp zeichnet sich durch sehr kleine aber priorisierte Pakete aus. Diese werden üblicherweise für alle Anwendungen der HID-Klasse verwendet (z.B Tastatur). Dabei werden diese Pakete in definierten Zyklen, welche im *Endpoint Descriptor* angegeben sind, verschickt, um neue Zustände (z.B. ein Tastendruck) der USB-Geräte zu ermitteln. Der USB garantiert eine Bandbreite von 90% für Interrupt- und Isochronous-Pakete.

[7] Im Sinne von dem USB-Gerät in Richtung Host-System.
[8] USB1.1
[9] USB1.0
[10] USB2.0

Bulk:

Der Bulk-Transfertyp dient dem Transport von sehr großen Paketen. Diese müssen nicht priorisiert sein. Typischerweise verwenden Drucker oder USB-Massenspeicher diesen Transfertypen. Nur Full- und High-Speed Geräte unterstützen diesen Transfertypen.

Isochronous:

Der Isochronous-Transfertyp unterstützt den Transfer von größeren Datenpaketen. Dabei ist Paketverlust möglich. Relevant ist dieser Transfertyp vor allem bei Echtzeitanwendungen wie zum Beispiel USB-Audio- und USB-Video-Geräten. Gemeinsam mit den Interrupt-Transfertypen teilt sich dieser Transfertyp 90% der USB-Bandbreite. Dieser Transfertyp ist nur für High- und Full-Speed Geräte verfügbar.

2.7 Host-Controller

Host-Controller kümmern sich um die Low-Level-Übertragung zwischen einem USB-Gerät und dem Host-System. Die Kommunikation zwischen dem Host-Controller und dem Host-System geschieht über das Host-Controller-Interface. Da die Kommunikation zwischen dem Host-Controller und dem Host-System sowie die Low-Level-Übertragung, für diese Arbeit irrelevant sind, werden im Folgenden nur die verschiedenen Host-Controller-Typen aufgezählt und ihr Nutzen auf der höheren Abstraktion aufgezeigt. Für weitere Informationen, dient die offizielle USB-Dokumentation zu den verschiedenen Host-Controller-Typen.

OHCI:

Das **O**pen **H**ost **C**ontroller **I**nterface ermöglicht die Kommunikation zwischen einem kompatiblen Host-Controller und den Host-System für die Übertragung von USB1.x Daten.

UHCI:

Das **U**niversal **H**ost **C**ontroller **I**nterface unterstützt ebenfalls wie OHCI nur die USB1.x Kommunikation. Dieser Standard ist proprietär und erfordert eine Lizenzierung von Intel. Anders als OHCI, ist die Hardware einfacher aufgebaut, dafür hat jedoch der HCI-Treiber mehr Last zu bewältigen.

EHCI:

Das **E**nhanced **H**ost **C**ontroller **I**nterface ermöglicht die Kommunikation zwischen dem USB-Host-Controller und dem Host-System für die Übertragung von USB2.0 Kommunikation. Typischerweise kommuniziert ein Host-System über EHCI mit dem Host-Controller für USB2.0 Kommunikationen und per UHCI bzw. OHCI für USB1.x Kommunikationen. Ein einziger Host-Controller kann unter Umständen beide HCI-Standards unterstützen.

XHCI:

Das E**x**tensible **H**ost **C**ontroller **I**nterface kompatible Host-Controller unterstützen alle USB-Versionen, einschließlich USB3.0. XHCI ersetzt somit alle anderen HCI.

3 USB-Network-Redirection

Für die Entwicklung eines USB-Fuzzers in einer virtuellen Umgebung wird eine entsprechende Schnittstelle benötigt, um USB-Geräte in der virtuellen Umgebung verfügbar zu machen. Hierfür existieren in QEMU mehrere Möglichkeiten. Dazu zählt zum Beispiel der direkte USB-Passthrough von USB-Geräten des Host-Systems, der Nutzung von selbst implementierten, virtuellen USB-Geräten und die Verwendung des USB-Network-Redirection-Protokolls. Dieses Projekt nutzt auf Grund der Einfachheit und der Flexibilität das USB-Network-Redirection-Protokoll.

3.1 usbredir

Das USB-Redirection-Protokoll ermöglicht die Übertragung von USB-Traffic in Form von TCP, UDP oder Unix-Sockets und dient ursprünglich der Bereitstellung von entfernten USB-Geräte in virtuellen Umgebungen. Das Protokoll findet Verwendung in der gleichnamigen Software „usbredir“ und ist Teil von SPICE[11]. „usbredir“ funktioniert aber auch unabhängig von SPICE.

Die Software „usbredir“ ist in der Programmiersprache C geschrieben und besteht aus den folgenden Modulen:

- usbredirparser
  Dabei handelt es sich um den Parser für das USB-Network-Redirection-Protokoll. Dieses Modul wird von allen anderen Modulen verwendet.
- usbredirhost
  Eine C-Bibliothek für die Implementierung der Host-seitigen Verbindung.
- usbredirserver
  Ein TCP-Server, welcher an Hand der Vendor- und Product-ID ein USB-Gerät des Host-System im Netzwerk über das USB-Network-Redirection-Protokoll verfügbar macht.

Zum Zeitpunkt dieser Arbeit, verfügt nur QEMU über eine vollständige, nutzbare Client-Implementierung. Praktisch kann das Protokoll somit momentan nur für den Transport von USB-Daten über das Netzwerk zwischen einem Host-System und einem virtualisierten System eingesetzt werden.

3.2 Protokoll

Das USB-Network-Redirection-Protokoll beschreibt einen Header sowie 30 zusätzliche Subheader[vgl. 11]. Der Header spezifiziert jedes „usbredir“-Paket und hat den folgenden Aufbau:

[11] Simple Protocol for Independent Computing Environments - http://www.spice-space.org

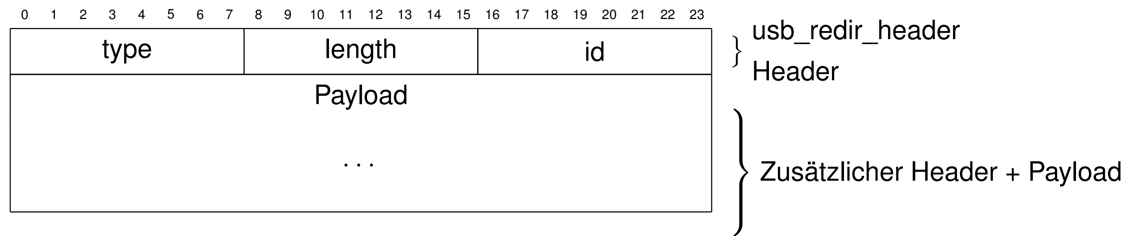

Abbildung 11: usbredir Header

Dabei spezifiziert das Feld „type“ den darunter liegenden Subheader und das Feld „length“ die Länge der Payload und somit die Größe des Pakets in Bytes abzüglich der Größe des Headers. Das Feld „id“, welches einer Identifikationsnummer entspricht, wird fortlaufend inkrementiert, sofern das Host-System Request-Pakete verschickt. Die Response-Paketen vom Gastsystem, also dem virtualisierten System, übernehmen die Identifikationsnummer des entsprechenden Request-Paketes. So kann bei asynchronem Paketversand immer eindeutig jede Anfrage vom Host-System jeder Antwort des Gastsystem zugeordnet werden.

Verwendet wurde für die Entwicklung des USB-Fuzzers nur eine Teilmenge der verfügbaren Subheaders. Dabei wird unterschieden zwischen den folgenden beiden Typen:

- „Control“-Pakettypen „Control“-Pakete sind USB-Network-Redirection-Protokoll spezifische Pakete.
- „Data“-Pakettypen „Data“-Pakete kapseln den Endpoint-Datentransfer.

Die folgende Auflistung ist ein Ausschnitt aus der offiziellen Dokumentation. Es werden in dieser Tabelle nur die für die Arbeit relevanten Pakettypen beschrieben. Ab 0x64 (dezimal 100) handelt es sich um die vier „Data“-Pakettypen. Der Verbindungsaufbau und Abbau geschieht mit „Control“-Paketen.

type_id	packet_name
0x0	usb_redir_device_connect
0x1	usb_redir_device_disconnect
0x2	usb_redir_reset
0x3	usb_redir_interface_info
0x4	usb_redir_ep_info
0x5	usb_redir_set_configuration
0x6	usb_redir_get_configuration
0x7	usb_redir_configuration_status
...	...
0x64	usb_redir_control_packet
0x65	usb_redir_bulk_packet
0x66	usb_redir_iso_packet
0x67	usb_redir_interrupt_packet

Tabelle 6: Auflistung von relevanten USB-Redirection-Paketen

3.3 usbredirserver

Für die Bereitstellung eines entfernten USB-Gerätes, dient der „usbredirserver“, welche als Teil der USB-Redirection-Suite ein eingestecktes USB-Gerät im Netzwerk bereitstellt. Hierfür wird das USB-Gerät per eigenem Treiber, welcher im User-Space läuft, gebunden und die Anfragen des Gastsystem an diesen durchgereicht. Dadurch entsteht eine logische Verbindung zwischen dem Gastsystem und dem USB-Gerät. Realisiert wird dies mit Hilfe von libusb[12]. libusb ist eine Bibliothek, welche in C geschrieben ist. Mit Hilfe von libusb ist es unter anderem möglich User-Space-Treiber zu schreiben. Verwendung findet libusb auch bei bekannten Projekten wie zum Beispiel SANE[13], einer API für standardisierte Zugriffe auf Scanner-Hardware.

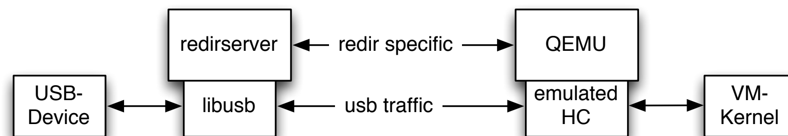

Abbildung 12: Aufbau usbredirserver

Der USB-Traffic wird dabei logisch zwischen dem libusb-Treiber und dem emulierten Host-Controller übertragen (Abb.12). Reell werden die Pakete jedoch in Form von USB-Redirection-Paketen gekapselt. Für den Verbindungsaufbau und die USB-Redirection-Kommunikation ist in diesem Fall die Software „usbredirserver“ und die QEMU-eigene Schnittstelle zuständig (siehe Kapitel 4.6.).

Vorteil bei der Verwendung des USB-Redirection-Protokoll ist die fehlende Limitierung der Anzahl der verfügbaren Endpoints. Es können somit USB-Geräte mit allen 31 Endpoints verwendet werden. Außerdem unterstützt das USB-Redirection-Protokoll auch das Kapseln des USB 3.0 Protokolls.

3.4 USB-Abstraktion

Da das USB-Redirection-Protokoll mit der QEMU / KVM Virtualisierung eingesetzt wird, wird der **USB-Interface-Layer**, welcher die physikalische Übertragungsschicht darstellt, komplett abstrahiert. Somit lässt sich nur das physikalische Ein- und Ausstecken des USB-Gerätes per entsprechendem Paket nachbilden. Andere Modifikationen auf physikalischer Ebene sind nicht möglich.
Mit Hilfe der „Control“-Paketen werden neben dem Verbindungsaufbau außerdem Teile der USB-Enumeration realisiert. Der **USB-Function-Layer**, lässt sich komplett über das USB-Redirection-Protokoll nachbilden, da mit Hilfe der „Data“-Pakete der Endpoint-Datenverkehr komplett gekapselt wird.

[12] http://www.libusb.org
[13] Scanner Access Now Easy - http://www.sane-project.org

4 QEMU / KVM

Dieses Kapitel beschreibt die Kombination von QEMU und der im Linux-Kernel integrierten Virtualisierungs-Infrastruktur KVM für den Einsatz im vUSBf-Framework.

4.1 QEMU

QEMU (engl. für Quick Emulator) ist ein Prozessor-Emulator, welcher unter der freien Softwarelizenz GPLv2 steht. Er emuliert verschiedene Prozessorarchitekturen per dynamischer Übersetzung, indem er die Prozessor-Instruktionen des Gastsystem in die Prozessor-Instruktionen für das Host-System zur Laufzeit übersetzt. QEMU erlaubt es somit, Software, die für eine andere Prozessorarchitektur kompiliert wurde, trotzdem auf einem gewünschten System lauffähig zu machen. Jedoch kostet die Emulation einen Mehraufwand an Zeit und Leistung. QEMU kann neben der Emulation der CPU, auch unzählige Peripheriegeräte und weitere Zusatzhardware emulieren. QEMU unterstützt unter anderem auch die Möglichkeit ein laufendes Gast-Betriebssystem, per Erstellung eines sogenannten Snapshot zu speichern und wiederherzustellen.

4.2 KVM

Bei KVM (Akronym für Kernel-based Virtual Machine) handelt es sich um ein, seit der Version 2.6.20, im Linux-Kernel integriertes Kernel-Modul. Dieses stellt eine Virtualisierungs-Infrastruktur im Kernel bereit. Hierfür nutzt KVM auf x86-Systemen die im Prozessor integrierten Virtualisierungstechniken Intel-VT oder AMD-V. Da KVM nur die Infrastruktur bereitstellt und keine Zusatzhardware emuliert, lässt sich in Kombination mit QEMU eine vollständige Virtualisierung nutzen.

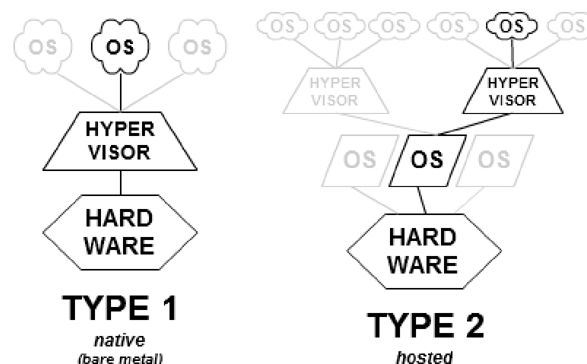

Abbildung 13: Type 1 und Type 2 Hypervisor[14]

Im speziellen stellt KVM hierbei den benötigten Hypervisor für die Virtualisierung im privilegierten Kernel-Space bereit. Es handelt sich um einen sogenannten Bare-Metal Hypervisor, welcher auch als Type 1 Hypervisor bezeichnet wird. Da KVM jedoch nicht ohne QEMU funktioniert, liegt KVM in Kombination mit QEMU somit in der Klassifizierung zwischen dem Type 1 und Type 2 Hypervisor. [15]

Der Vorteil bei der Prozessor-Virtualisierung ist, dass die Prozessor-Instruktionen einer virtuellen Maschine auch der verwendeten Prozessorarchitektur des Host-System entsprechen. Daher müssen die Instruktionen, anders als bei einer Emulation, nicht erst per dynamischer Übersetzung umgewandelt werden, was zu einer deutlich langsameren Ausführgeschwindigkeit führen würde. KVM

[14] Entnommen aus [vgl. 23]
[15] http://www.ijitee.org/attachments/File/v2i3/C0453022313.pdf

verhält sich somit unter Linux wie ein gewöhnlicher Prozess und lässt sich, sofern der Zugriff auf die Gerätedatei /dev/kvm gegeben ist, ohne Root-Rechte ausführen. Die verschiedene Teile von KVM stehen alle unter Variationen der freien Softwarelizenz GPL zur Verfügung.

4.3 QEMU und die Kommandozeile

QEMU lässt sich, ohne die Verwendung von weiteren Tools (insbesondere libvirt ist hier zu erwähnen), prinzipiell nur von der Kommandozeile aus starten und verwalten. Für die Verwaltung und Steuerung existiert keine grafische Oberfläche. Die Verwendung von QEMU erweist sich jedoch als überaus einfach und lässt sich mit dem Wissen über einige wenige Parameter fast vollständig an die eigenen Bedürfnisse anpassen.

Es ist zu empfehlen, sofern die selbe Prozessorarchitektur verwendet wird, auch die KVM-Unterstützung zu aktivieren. Hierfür wird der der Parameter `-enable-kvm` als Argument übergeben. Außerdem lässt sich über den Parameter `-m [SIZE]` die Größe des virtuellen Arbeitsspeichers definieren und per Argument `-hda [File]` oder `-hdb [File]` das zu verwendende Imagefile hinzufügen. `hda` steht hier für den IDE Master Kanal 1 und `hdb` für den IDE Slave Kanal1. Des Weiteren ermöglicht das Argument `-serial stdio` die Ausgabe von Daten, welche das emulierte bzw. virtualisierte System per (virtuellen) seriellen Port verschickt, an das Host-System zu übergeben. Auch einzugebende Daten lassen sich per seriellen Port an das Gastsystem schicken. Unter einigen Betriebssystemen, wie zum Beispiel Linux oder BSD, besteht die Möglichkeit eine TTY (Akronym für Teletype) per seriellen Port verfügbar zu machen. Somit lässt sich ein Gastsystem, über den selben Terminal, wie eine Kommandozeilen-Applikation bedienen. In Kombination mit dem Argument `-nographic`, wird auch die Ausgabe der virtuellen Grafikkarte deaktiviert und das Gastsystem ist nur noch über den seriellen Port erreichbar.
Ein beispielhaftes QEMU-Kommando könnte wie folgt aussehen:

```
qemu --enable-kvm -m 512 -hda ./debian6.qcow2 -hdb ./storage.qcow2
```

4.4 Image-Dateien

QEMU nutzt als virtuelle Festplatten sogenannte Image-Dateien. Image-Dateien entsprechen dem Inhalt einer virtuellen Festplatte in Form einer Datei. Jedoch wird hierbei zwischen den verschiedensten Image-Formate unterschieden.

4.4.1 RAW

Das einfachste und gleichzeitig schnellste Image-Format ist das sogenannten RAW-Format. Dazu gehören unter anderem Speicherdumps, welche sich zum Beispiel mit dem Unix-Tool „dd“ erstellen lassen. Dateien in diesem Format entsprechen immer der selben Größe, der virtuellen Festplatte.

4.4.2 QCOW2

Image-Formate, wie zum Beispiel das QEMU-eigene Image-Format, QCOW2 (QEMU Copy-on-Write Version 2) arbeiten weitaus intelligenter und bringen darüber hinaus noch unzählige Features mit sich. Anders als ein RAW-Image, alloziert eine QCOW2-Image-Datei nur soviel Speicherplatz wie

sie auch wirklich zum jeweiligen Zeitpunkt benötigt und entspricht somit im anfänglichen Initialzustand nur einer Größe von weniger als 1MB. Darüber hinaus ermöglichen QCOW2-Image-Dateien die Verschlüsselung und Komprimierung der Image-Datei. Außerdem ist die Erstellung von sogenannten Snapshots, welche in der Image-Datei gespeichert werden, zum Zeitpunkt dieser Arbeit, nur dem QCOW2-Format vorenthalten.

4.4.3 Snapshots

QCOW2-Snapshots lassen sich während des laufenden Betriebs erstellen und sichern den aktuellen Zustand der Festplatte, des Arbeitsspeichers und aller CPU-Registern. Der erstellte Snapshot lässt sich jederzeit wieder im laufenden Betrieb laden und das System wieder in einen vorherigen Zustand zurück versetzen. Außerdem besteht die Möglichkeit ein QCOW2-Snapshot direkt beim Starten von QEMU per „-loadvm [SNAPSHOT]" zu laden. Der Vorteil ist hierbei, dass eine virtuelle Maschine sich in wenigen Sekundenbruchteilen laden lässt und man somit den Boot-Vorgang überspringen kann.

4.5 qemu-img

Mit dem Tool qemu-img lassen sich Image-Dateien in den verschiedensten Formaten erstellen. Relevant ist für diese Arbeit nur das QEMU-eigenen QCOW2-Format, welches deshalb auch im Folgenden verwendet wird. Mit Hilfe von dem folgenden Parameter lässt sich eine 5GB große QCOW2-Datei erstellen:

```
qemu-img create -f qcow2 /tmp/image.img 5G
```

4.6 USB-Redirection-Interface

Die von QEMU implementierte USB-Redirection-Schnittstelle ist in der Lage entfernte USB-Geräte innerhalb der virtuellen Maschine bereit zu stellen. Das entfernte USB-Gerät kommuniziert über die USB-Redirection-Schnittstelle und einen emulierten USB-Host-Controller. Die Kommunikation sieht wie folgt aus:

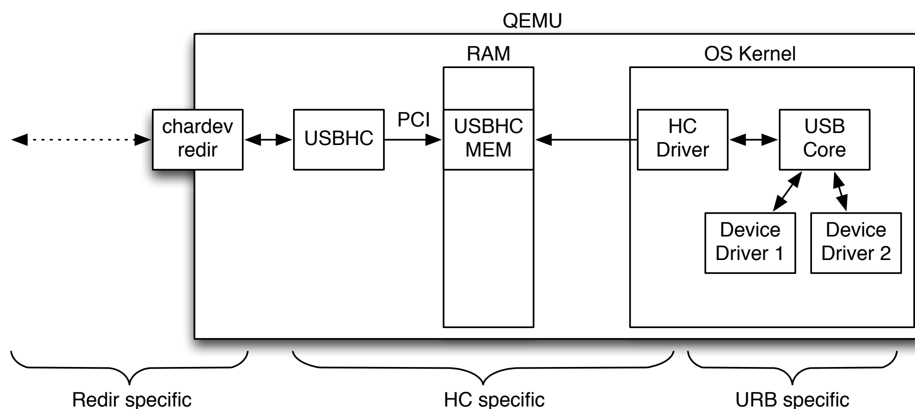

Abbildung 14: Aufbau des USB-Redirection-Interface

4.6.1 Kompilierung

Je nach verwendeter Binärdatei von QEMU, kann es sein, dass keine USB-Redirection Unterstützung hinzu kompiliert wurde oder die Unterstützung veraltet ist. Es ist daher ratsam, sich eine aktuelle Version von QEMU, mit der Option im configure-Skript „–enable-usb-redir“, und das zugehörige Paket „usb-redir“ selbst zu kompilieren.

Es wurden des Weiteren noch einige Änderungen am Quelltext von QEMU vollzogen, welche für das Fuzzing-Framework von Relevanz sind. Angepasst wurde hierbei die Größe des ID-Feldes des USB-Network-Redirection-Header. Dabei wurde das Feld auf eine maximale Größe von 32 Bit limitiert, um Kompatibilitätsprobleme mit älteren usbredir[16] Versionen zu vermeiden.

Die USB-Redirection-Schnittstelle wird unter QEMU als „character device“ (oder kurz chardev), also als zeichenorientiertes Gerät, bereitgestellt. Gemeint ist dabei die Eigenschaft, dass nur ein Byte (Zeichen) zur selben Zeit übertragen wird, was somit der seriellen Datenübertragung des USB entspricht.

Ein chardev lässt sich unter QEMU auf die verschiedensten Arten mit Daten versorgen. Von Bedeutung sind in dieser Arbeit jedoch nur die Optionen der Datenübertragung per TCP, UDP oder Unix-Sockets im Server- bzw. Client-Modus. Der Vollständigkeit halber sei jedoch auf die Man-Page von QEMU verwiesen. Durch geschickte Parametrisierung lässt sich somit die USB-Redirection-Schnittstelle, je nach Anwendung, entsprechend bereitstellen.

Ein beispielhafter Parameter würde wie folgt aussehen:

```
#Erstellung einer USB-Redirection-Schnittstelle als chardev ohne Debug-Ausgaben
-device usb-redir,chardev={chardevi_id}, debug=0
Erstellung einer Datenquelle fuer das chardev
-device {type},[server],id={chardev_id},[nowait],[path=FILE oder host=HOST,port=PORT
    ]
```

Zu ersetzen ist dabei „type“ mit tcp, udp oder socket. Die Option „server“ sollte hinzu geschrieben werden, falls man einen Server zur Verfügung stellen möchte, anderenfalls arbeitet dieses Device wie ein Client. Das Feld „id“ sollte mit dem definierten chardev, welches die USB-Redirection-Schnittstelle repräsentiert, verknüpft werden. Die Option „nowait“ definiert, ob auf eine Verbindung gewartet werden sollte, bevor die virtuelle Maschine gestartet wird oder nicht. Und der Parameter „path“ gibt den Pfad zu einem Unix-Socket an, wobei die Parameter „host“ und „port“ selbsterklärend sind und benötigt werden sofern der „type“ auf tcp bzw. udp gewählt wurde.

Des Weiteren ermöglicht QEMU, den gewünschten USB-Host-Controller für die virtuelle Maschine per Parameter selber zu wählen. Die Verwendung eines USB-Host-Controller ist zwingend erforderlich. Anderenfalls lässt sich keine Schnittstelle zwischen dem virtualisierten Betriebssystem und dem USB-Redirection-Interface herstellen. Dazu wird einfach ein USB-Host-Controller als weiteres Gerät wie folgt definiert:

```
-device [USB-Host-Controller-Name]
```

[16] http://cgit.freedesktop.org/spice/usbredir/tree/ChangeLog?id=usbredir-0.7

Folgende USB-Host-Controller[17]lassen in QEMU emulieren:

xHCI-Controller	USB-Version: 1.0, 2.0, 3.0
nec-usb-xhci	
EHCI-Controller	USB-Version: 2.0
usb-ehci	
ich9-usb-ehci1	
OHCI-Controller	USB-Version: 1.1
sysbus-ohci	
pci-ohci	
UHCI-Controller	USB-Version: 1.0
ich9-usb-uhci1	
ich9-usb-uhci2	
ich9-usb-uhci3	
piix3-usb-uhci	
piix4-usb-uhci	
vt82c686b-usb-uhci	

Tabelle 7: Auflistung aller von QEMU emulierbaren USB-Host-Controller

4.7 QEMU Monitor

Der QEMU Monitor, auch die QEMU Management Console genannt, ermöglicht die Überwachung und Steuerung von laufenden virtuellen Maschinen. Dazu gehört die Möglichkeit der Erstellung und dem Laden von Snapshots, der Überwachung der virtuellen Hardware, das Hinzufügen bzw. Durchreichen von emulierter bzw. realer Peripherie an die virtuelle Maschine und viele weitere Funktionen. Der QEMU Monitor lässt sich bereits zum Start von QEMU dauerhaft per Konsole bedienen, in dem QEMU mit dem „-monitor serial“ Parameter gestartet wird.

Leider ist diese Variante unvorteilhaft, sofern man den QEMU Monitor in Kombination mit der Ausgabe von Daten des virtuellen seriellen Ports nutzt. Alternativ lässt sich der QEMU Monitor auch parallel mit der Ein- und Ausgabe eines virtuellen seriellen Ports verwenden. Hierfür muss QEMU mit dem Parameter „-serial mon:stdio“ gestartet werden. Der Vorteil ist dabei, dass je nach Anwendung zwischen den beiden Datenkanälen umgeschaltet werden kann. Der QEMU Monitor lässt sich per Tastenkombination (`CTRL-A + c`) öffnen und wieder beenden.

Eine weitere Möglichkeit stellt die Nutzung des QEMU Monitor in Form eines chardev dar. Somit lässt sich der QEMU Monitor auch in einer alternativen TTY steuern. Die Konfiguration ist ähnlich wie bei der Bereitstellung der USB-Redirection-Schnittstelle:

```
-mon chardev={chardev_id}
```

[17] Die Bedeutung der einzelnen USB-Host-Controller ist im Kapitel 2.7 nachzulesen.

5 Fuzzing

Fuzzing, abgeleitet von Fuzz-Test („fuzz“ engl. unscharf), ist eine Softwaretest-Methodik um automatisiert Fehler und Schwachstellen in Software zu finden. Dies wird realisiert, indem die Eingabeschnittstelle der zu untersuchenden Applikation mit unerwarteten, (teil-)veränderten oder auch ungültigen Datenströmen versorgt wird. Das Ziel ist hierbei explizit die Provokation von Abstürzen oder Fehlverhalten der zu untersuchenden Applikation.

5.1 Grundlegender Aufbau

Das Fuzz-Testing bzw. der Aufbau einer Fuzzing-Umgebung lässt sich in sechs elementare Abläufe unterteilen[vgl. 15]:

(I) Die Festlegung auf das zu untersuchenden Programm.

(II) Die Identifizierung der zu verwendenden Eingabeschnittstelle der Applikation.

(III) Die Generierung der Daten für die zu untersuchende Applikation.

(IV) Die Übertragung der generierten Daten an die identifizierte Eingabeschnittstelle des zu untersuchenden Programms.

(V) Die Überwachung des zu untersuchenden Programms. Dieser Vorgang wird auch als Monitoring bezeichnet und dient der (automatisierten) Erkennung von Fehlverhalten und Abstürzen des zu untersuchenden Programms. Die generierten Daten, wie im Punkt 1 beschrieben, werden in Relation zur dokumentierten Auswirkung ebenfalls protokolliert, um somit eine gute Reproduzierbarkeit für spätere Untersuchungen zu gewährleisten. Idealerweise werden die beiden vorgestellten Punkte automatisiert.

(VI) Die Untersuchung der gefundenen Fehler auf ihre Auswirkung und die mögliche Ausnutzbarkeit.

Die Punkte 1 bis 3 beschreiben die notwendige Vorarbeit für das Fuzzing. Würde man Fuzzing als eine Zustandsmaschine beschreiben, so würden die Punkte 3, 4 und 5 in dieser Reihenfolge in einer Schleife ausgeführt werden. Der letzte Punkt wird erst nach dem erfolgreichen Fuzzing-Durchlauf eingeleitet.

5.2 Komplexität der Datengenerierung

Da beim Fuzzing der Code der zu untersuchenden Applikation ausgeführt wird, stellt somit die Ausführgeschwindigkeit den limitierenden Faktor in der Untersuchungsgeschwindigkeit dar. Aufgrund dessen sollte bereits im Vorfeld die Menge der möglichen Fuzzing-Operationen eingegrenzt werden. Man unterscheidet daher auch zwischen zwei verschiedenen Herangehensweisen[vgl. 22]:

(I) Die zufälligen Generierung von Daten, bzw. die zufällige Manipulation von validen Eingabedaten, zum Beispiel während des Abgreifens von Daten mit einer Man-in-the-Middle-Herangehensweise. Dieses Vorgehen impliziert (fast) keine Vorkenntnis über das verwendete Protokoll bzw. die Struktur der Daten und eignet sich somit gut, um in Abhängigkeit der zu untersuchenden Software und der Robustheit dieser, erste Implementierungsfehler zu finden.

(II) Die zweite Variante erfordert ein genaues Verständnis über die Eingabeschnittstelle und die Struktur dieser. Durch diese Kenntnis lässt sich die Menge der möglichen Fuzzing-Operationen beschränken auf eine Teilmenge „relevanter“ Tests.

5.3 USB-Fuzzing

Das in dieser Arbeit behandelte USB-Fuzzing lässt sich in je zwei verschiedene Methodiken unterteilen:

5.3.1 Field-Fuzzing

Bei diesem Vorgehen werden Felder der verschiedenen Deskriptoren, wie sie im Kapitel 2.4. vorgestellt wurden, gezielt verändert. Die eigentliche Übertragung, über den Endpoint 0 für die Enumeration, bleibt in diesem Fall unberührt, da das Ziel des USB-Fuzzing nicht der USB-Core-Driver ist. Die Herausforderung ist hierbei, das Fuzzing der Deskriptoren darauf zu beschränken, dass der USB-Core-Driver diese als valide akzeptiert und sie an die entsprechenden USB-Treiber übergibt, so dass dort Fehlverhalten oder gar Abstürze ausgelöst werden.

Das entwickelte Framework vUSBf beschränkt das Field-Fuzzing, in der aktuellen Version, auf die Enumeration, wobei in zukünftigen Versionen auch das Fuzzing der Function-Layer-Daten dazu gehört. Folgende Grafik verdeutlicht das Vorgehen:

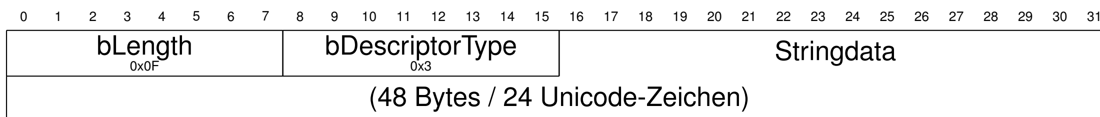

Abbildung 15: Ungültiger *String Descriptor*

Die Abbildung 15 repräsentiert einen ungültigen *String Descriptor*. Die Länge des Deskriptor entspricht jeweils 2 Bytes, für die Felder bLength und bDescriptorType, sowie die Länge der Payload in Form einer UTF-16 codierten Zeichenfolge, welcher in diesem Beispiel 48 Bytes lang ist. Da die Länge des *String Descriptor* somit 50 Byte entspricht, ist der Wert 0x0F im Feld bLength absichtlich falsch und könnte für einen Fuzzing-Test verwendet werden. Im Falle einer fehlerhaften Implementierung des jeweiligen USB-Gerätetreibers, ließe sich z.B. ein Pufferüberlauf provozieren.

5.3.2 Device-Descriptor-Fuzzing

Das Deskriptor-Fuzzing dient zum gezielten Verändern ganzer *Device Descriptors*. Die Idee ist hierbei ganze Zweige bestehender *Device Descriptors* hinzuzufügen oder zu löschen. Hierfür werden bei dem Hinzufügen oder dem Löschen auch entsprechende Felder wie zum Beispiel bNumInterfaces, bConfigurationValue und so weiter, angepasst. Das Device-Descriptor-Fuzzing wird idealerweise vor dem Field-Fuzzing realisiert, damit die angepassten Felder nachträglich manipuliert werden können.

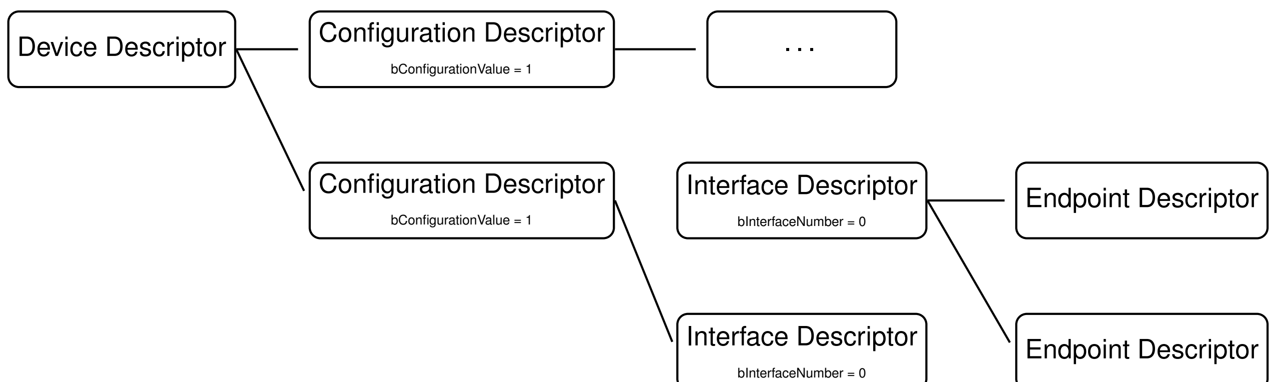

Abbildung 16: Device-Descriptor-Fuzzing beispielhafte Anwendung

Wie in der Abbildung 16 dargestellt, wird in dem Beispiel der Zweig zu einem *Interface Descriptor* entfernt und alle in Relation stehenden Felder anderer Deskriptoren werden angepasst (bInterfaceNumber und bConfigurationValue). In diesem Beispiel wurde die Verbindung nur entfernt. In der Praxis wird der entsprechende Zweig auch komplett gelöscht.

5.4 Herangehensweisen

Aufgrund der Tatsache das eine USB-Übertragung immer zustandsbasiert ist, stellt sich die Frage auf welcher Basis das USB-Fuzzing stattfinden soll. Es bieten sich unter Verwendung des USB-Redirection-Protokoll zwei mögliche Varianten an:

5.4.1 Man-in-the-Middle

Die Übertragung zwischen dem vorgestellten usbredirserver und QEMU geschieht standardmäßig per TCP/IP und erlaubt dahingehend die Nutzung einer Proxy-Applikation. Der USB-Traffic kann in dieser Variante gezielt verändert und abgehört werden. Es besteht somit keine Notwendigkeit, die USB-Redirection- bzw. USB-Übertragung nachzuahmen.

5.4.2 Emulation von USB-Geräten

Da das USB-Redirection-Protokoll vollständig dokumentiert ist, bietet sich die Möglichkeit einer eigenen Server-Applikation. Aufgrund der Gegebenheiten, kann ein USB-Emulator implementiert werden, welcher das USB-Redirection-Interface als Schnittstelle verwendet. Der Vorteil dieser Variante ist die Flexibilität, jedes erdenkliche USB-Gerät, beschränkt auf die Übertragung des Device Layers, vollständig emulieren zu können. Außerdem kann solch ein USB-Emulator, mit dem Ziel des USB-Fuzzings, um einen absichtlich fehlerhaften Function Layer erweitert werden. Als Beispiel sei hierbei ein USB-Mass-Storage-Emulator zu nennen, welcher um eine SCSI-Emulation erweitert wurde, wobei diese sich jedoch nicht Protokoll-konform verhält und absichtlich mit falschen Responsepaketen antwortet.

6 vUSBf Framework

Das im Laufe der Bachelorarbeit entwickelte USB-Fuzzing Framework vUSBf versucht die Nachteile anderer USB-Fuzzing Lösungen, insbesondere im Bezug auf die Ausführgeschwindigkeit, zu umgehen. Das Framework bietet eine erweiterbare API, verschiedene Ausführmodi, eine eigene Fuzzing-Engine und ein hohes Maß an Reproduzierbarkeit der gefundenen Fehler in USB-Gerätetreibern. Dieses Kapitel beschreibt die Entwicklung des Frameworks und erklärt wichtige Funktionalitäten.

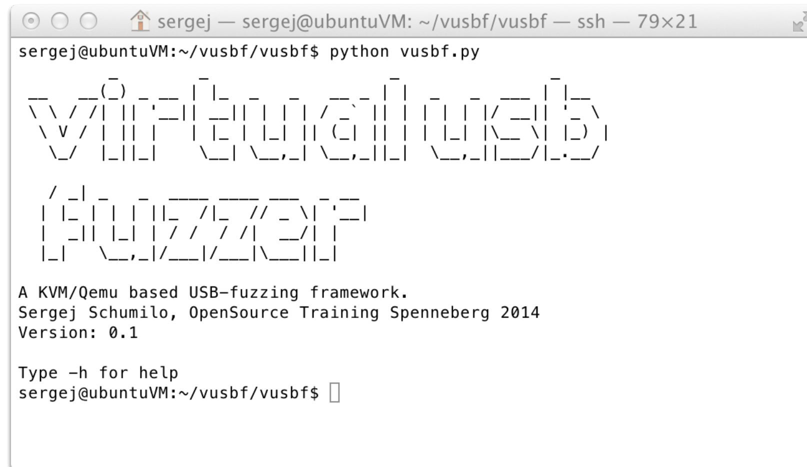

Abbildung 17: Bildschirmfoto von vUSBf

6.1 Fuzzing-Methoden

Das vUSBf ist das Ergebnis einer fortlaufenden, logischen Entwicklung verschiedener getestet USB-Fuzzing-Methoden. Während des Projektes wurden mehrere Herangehensweisen empirisch untersucht um ein effizientes und zugleich effektives USB-Fuzzing zu ermöglichen. Folgende Herangehensweisen repräsentieren in dieser Reihenfolge auch die Projektchronik:

6.1.1 Live-Fuzzing

Den Anfang des Projektes stellt der „usbredirserver"-Proxy dar. Dieser wurde mit dem Ziel entwickelt, das Analysieren und spätere Fuzzing der USB-Übertragungen zwischen einem Betriebssystem und einem USB-Gerät, ohne die Entwicklung bzw. Verwendung einer Hardware-Lösung, zu ermöglichen. Da es sich bei der Software usbredirserver um eine TCP/IP-Applikation handelt, kann ein simpler Proxy-Server ohne großen Aufwand betrieben werden. Die USB-Daten werden in diesem Aufbau wie folgt übertragen:

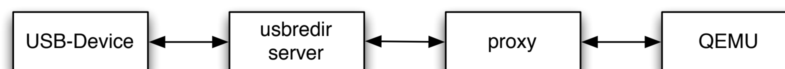

Abbildung 18: Aufbau des usbredirserver-proxy

Da der gesamte USB-Datenverkehr durch die Proxy-Applikation geht, lassen sich komplette Mitschnitte erstellen. Mit Hilfe von einem implementierten Decoder, für das USB-Redirection-Protokoll,

lassen sich die gekapselten USB-Daten extrahieren. Die USB-Redirection-Daten werden hierfür mithilfe des Scapy-Frameworks[vgl. 17] und erstellten Scapy-Dissektoren zerlegt. Die Daten liegen nach der Umwandlung in Form von einem Scapy-Objekt vor. Alternativ ließen sich auch für die Analyse Wireshark-Dissektoren schreiben, jedoch würde dann die Möglichkeit des USB-Fuzzings fehlen. Mitgeschnittene USB-Daten lassen sich mithilfe eines Parsers auch in lesbare Ausgaben umwandeln:

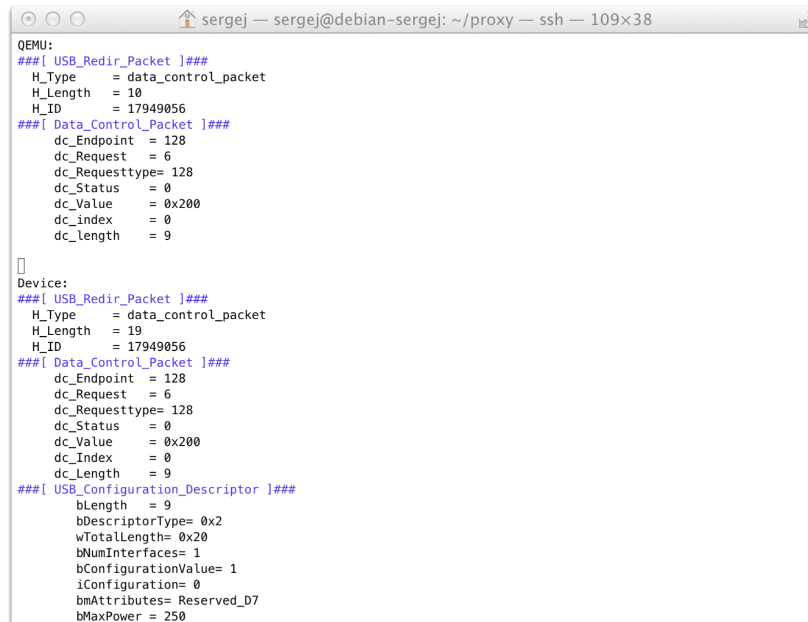

Abbildung 19: Ausschnitt aus der dekodierten USB-Mitschnitt-Datei

Alternative USB-Sniffer Lösungen bieten gegebenenfalls einen höheren Bedienkomfort dank einer grafischen Oberfläche. Trotzdem bietet diese simple Implementierung dieselbe Möglichkeit, um USB-Daten zu untersuchen, wie alternative USB-Sniffer Lösungen. Für verschiedene Betriebssysteme existieren fertige Software-Lösungen wie zum Beispiel usbsnoop, für Microsoft Windows-Systeme, und Wireshark mit aktivierter usbmon-Unterstützung[18] für Linux-Systeme.
Ein Vorteil im Vergleich zu anderen Software-Lösungen ist die Tatsache, dass nicht der eigene USB-Datenverkehr mitgeschnitten wird, sondern der zwischen einer virtuellen Maschine und einem echten USB-Gerät. Dies hat den Vorteil, dass bei Betriebssystem-Abstürzen keine Datensätze verloren gehen können. In Kombination mit dem USB-Fuzzing erweist sich dieser Vorteil als äußerst nützlich, da durch einen vollständigen Mitschnitt auch die Reproduzierbarkeit von gefundenen Fehlern in Betriebssystem- bzw. USB-Treibern im hohen Maße gegeben ist.
Alternativ lassen sich auch hardwarebasierte USB-Sniffer verwenden. Diese erlauben ebenfalls, wie der Redir-Proxy, die Erstellung von kompletter Datensätze. Jedoch sind die hohen Kosten ein nicht zu unterschätzender Nachteil. Des Weiteren bieten Hardware-basierte USB-Sniffer nicht immer die benötigte Flexibilität, welche für das USB-Fuzzing unter Umständen vorausgesetzt wird. Vorteile von Hardware-basierte USB-Sniffer[19], im Bezug auf die Geschwindigkeit und die Analysefähigkeit des Interface Layers, sind für die Entwicklung eines USB-Fuzzers, welcher auf der logischen und nicht der physikalischen Ebene arbeitet, nicht von Relevanz.

Der nächste logische Schritt ist das Field-Fuzzing von ausgewählten USB-Paketen, welche durch

[18]http://wiki.wireshark.org/CaptureSetup/USB
[19]http://www.totalphase.com/solutions/apps/usb-analyzer-benefits/

den Redir-Proxy gehen. Relevant sind nur die ausgehenden USB-Daten, da der USB-Host das Ziel der Untersuchung ist und nicht das USB-Gerät. Bereits das zufällige Verändern von einem Byte mit einer Wahrscheinlichkeit von 10% pro Redir-Paket, führte in den ersten Tests, nach einem mehrstündigen Fuzzing, zu Abstürzen von aktuellen Linux Kernel Versionen. Die Fuzzing-Erweiterung hat den folgenden Aufbau:

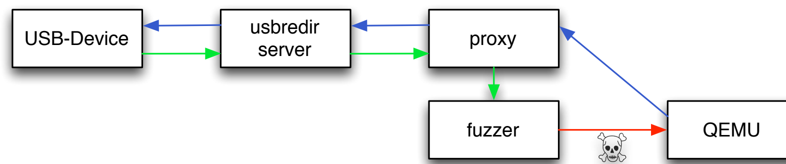

Abbildung 20: USB-Fuzzing der Daten in Richtung QEMU.

Der USB-Fuzzer verändert nur USB-Daten und lässt den darüberliegenden USB-Redirection-Layer unangetastet. Anderenfalls führt dies relativ schnell zu Abstürzen von QEMU, da die Implementierung des USB-Redirection-Interface, zum Zeitpunkt dieser Arbeit, relativ fragil zu seien scheint und keine ausreichenden Eingabevalidierung stattfindet.

6.1.2 Record- / Replay-Fuzzing

Da die Enumeration von USB-Geräten bis zu mehreren Sekunden dauern kann, lässt sich das Fuzzing mithilfe des Redir-Proxys nicht beschleunigen. Das USB-Gerät und das verwendete Betriebssystem stellt hierbei den limitierenden Faktor dar. Alternativ lässt sich das USB-Fuzzing auch parallelisieren. Jedoch muss hierfür auch die selbe Anzahl an physikalischen USB-Geräten zur Verfügung stehen. Auch lässt sich die Zahl der parallel arbeitenden Prozesse nicht beliebig skalieren, da die Bandbreite des USB bei mehreren eingesteckten USB-Geräten entsprechend verteilt werden muss und nur maximal 128 USB-Geräte, inklusive aller USB-Hubs, vom Host-Betriebssystem adressiert werden können.

Eine Möglichkeit, um diesen Flaschenhals zu umgehen, ist die Verwendung einer Replay-Funktion. Diese zeichnet eine USB-Übertragung im Voraus auf und spielt diese danach exakt wieder ab. Der Vorteil bei dieser Variante ist, dass nach einer einmaligen Aufzeichnung, diese beliebig parallel abgespielt werden kann. In diesem Fall lässt sich das USB-Fuzzing, je nach verwendeter Hardware, beliebig parallelisieren, was somit zu einer Maximierung der Fuzzing-Geschwindigkeit führt. Der Aufbau des Redir-Proxy-Recorder bzw. Replay-Fuzzers sieht wie folgt aus:

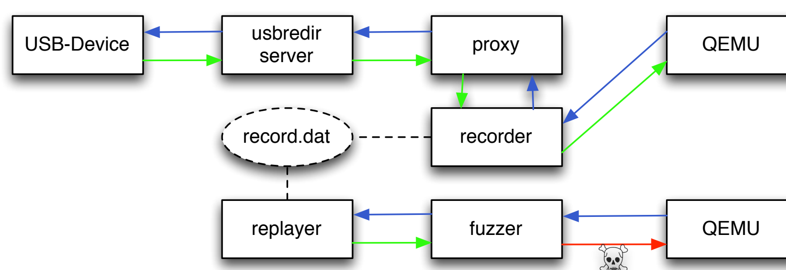

Abbildung 21: Aufbau des usb-redir-proxy

Hierfür erstellt der USB-Proxy-Recorder einen Mitschnitt, welcher im späteren Verlauf auch vom Replay-Fuzzer verwendet werden kann. Wichtig ist für dieses Vorgehen, dass QEMU-Snapshots verwendet werden. Dadurch wird bezweckt, dass ein Mitschnitt, welcher nach dem Laden eines Snapshots aufgezeichnet wurde, auch nach jedem weiteren Laden eines Snapshots funktioniert. Problematisch ist aber, dass in dieser Variante nicht auf „unvorhergesehenes“ Verhalten des Betriebssystems reagiert werden kann, da die benötigten Daten nicht Teil der Aufzeichnung sind. Es lassen sich somit unter Umständen nicht alle Fehler in USB-Gerätetreibern finden.

6.1.3 Fuzzing durch Emulation

Um das Problem der Fuzzing-Geschwindigkeit und der Reaktion auf unerwartete Anfragen des Betriebssystem zu lösen, werden in dem vUSBf-Framework USB-Geräte emuliert. Durch die Emulation wird eine vollständige Generierung des entsprechenden USB-Traffics ermöglicht und die Emulation lässt sich nach Belieben parallelisieren. USB-Emulatoren werden im Framework in Form von Modulen bereitgestellt und geladen. Jeder USB-Emulator, welcher den Function Layer emuliert, lädt erst den Enumeration-Emulator, welcher für die USB-Enumeration auf dem Device Layer zuständig ist, und erhält erst danach die Kontrolle von diesem.

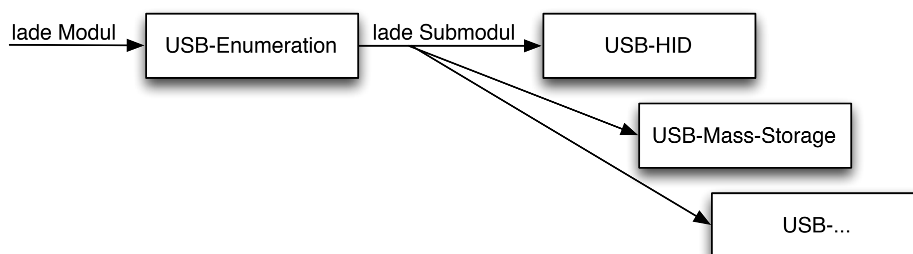

Abbildung 22: Hierarchischer Aufbau von USB-Emulatoren

6.2 vUSBf Architektur

Das vUSBf-Framework verwendet mehrere Teilmodule, welche im laufenden Prozess miteinander kommunizieren. Die Architektur des Frameworks entspricht der folgenden Grafik:

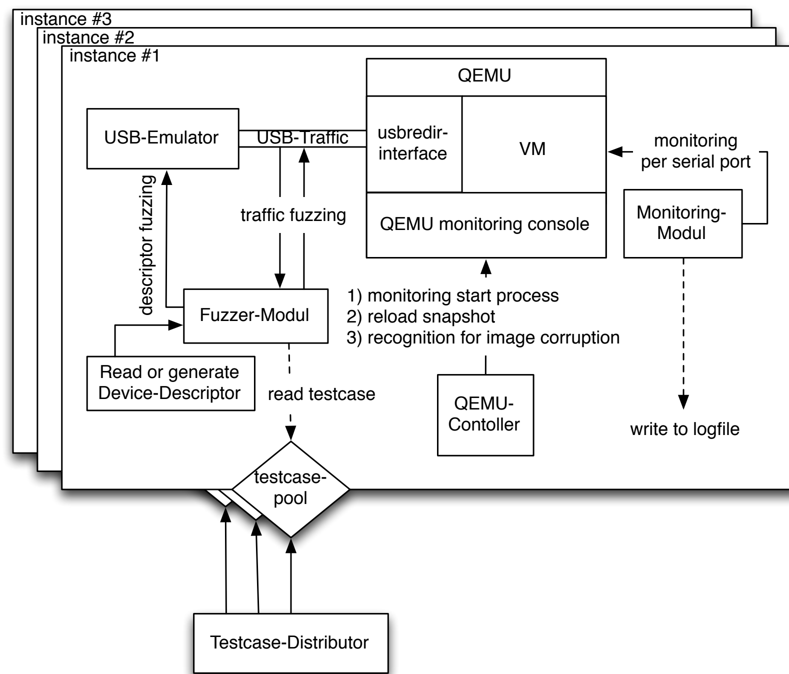

Abbildung 23: Architektur des vUSBf-Framework

USB-Emulator:

Virtuelle USB-Geräte werden mithilfe von USB-Emulatoren bereitgestellt. Das Framework bietet darüber hinaus eine erweiterbare API. Beschrieben ist die USB-Emulation im Kapitel 6.3.

QEMU-Controller:

Für die Realisierung einer QEMU- und KVM-basierten USB-Fuzzing Infrastruktur, wird der QEMU-Controller für die Erstellung, Überwachung und Steuerung der QEMU-Prozesse verwendet. Beschrieben ist die QEMU und KVM Abstraktion im Kapitel 6.4.

Monitoring-Modul:

Das Monitoring-Modul erlaubt die Überwachung des virtualisierten Betriebssystems auf das Verhalten nach einem USB-Fuzzing-Test. Das Monitoring ist im Kapitel 6.4.4. beschrieben.

Fuzzer-Modul und Testcases:

Testcases enthalten alle für einen einzigen Fuzzing-Test anzuwendenden Instruktionen für das Fuzzing-Modul. Wie Testcases aufgebaut sind und wie das Fuzzing realisiert wurde, ist im Kapitel 6.7. beschrieben.

Testcase-Distributor und Testcase-Pool:

Für die Realisierung einer Multiprocessing- bzw. Clustering-Infrastruktur, werden Methoden für die Verteilung von Testcases und die Kommunikation zwischen Prozessen und Computersystemen benötigt. Die Prozesskommunikation und das Clustering-Protokoll sind im Kapitel 6.5. und 6.6. beschrieben.

6.3 USB-Emulation

Die USB-Emulation wird mit einer erweiterbaren API und das USB-Redirection-Protokoll realisiert. Als Datenkanal, zwischen QEMU und einem USB-Emulator, wird ein exklusiver Unix-Socket pro virtueller Maschine verwendet.

6.3.1 USB-Emulation Architektur

USB-Emulatoren sind im vUSBf-Framework aus technischer Sicht als Klassen implementiert, welche sich von der ”abstrakten” Klasse USB-Emulator.py ableiten. Diese USB-Emulatoren implementieren eine Nachbildung der USB-Enumeration, sowie den jeweiligen Function Layer. Hierarchisch gesehen wird erst ein spezieller Emulator für den USB-Redirection-Protokoll Verbindungsaufbau gestartet, welcher danach die Kontrolle an den ausgewählten USB-Emulator übergibt, welcher sich um alle Anfragen des virtualisierten Systems kümmert.

Die abstrakte Klasse bietet als öffentliche Methode den klasseneigenen Konstruktor (__init__), welchem eine Instanz der Klasse „fuzzer.py“ übergeben wird. Diese Klasseninstanz entspricht einem vorkonfigurierten Fuzzing-Modul, der die ausgehenden USB-Daten des USB-Emulators, anhand von übergebenen Testcases (siehe Kapitel 6.7.), verändert.

Als weitere Schnittstelle existiert die Methode get_response(request), der ein USB-Request des virtualisierten Betriebssystems erhält und mit Hilfe dessen eine USB-Response generiert. Für die Realisierung des USB-Fuzzings wird das Scapy-Framework verwendet. Intern wird eine Response in Form eines Scapy-Objektes generiert und danach dem Fuzzing-Modul per API-Aufruf _fuzzdate() übergeben. Das Fuzzing-Modul kann mit Hilfe des Scapy-Objekt entsprechende Fuzzing-Operation hierauf anwenden. Nach dem die Fuzzing-Operation angewandt wurde, wird der generierte und gefuzzte Request in Form eines Byte-Arrays als Rückgabe-Parameter übergeben.

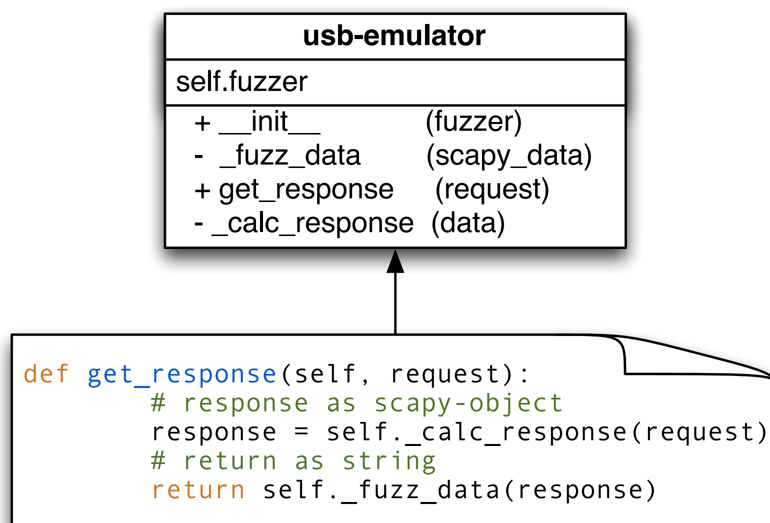

Abbildung 24: Klassendiagramm der Klasse „usb-emulator.py“

6.3.2 Redir-Emulator

Der Redir-Emulator kümmert sich um die Emulation des USB-Redirection spezifischen Verbindungsaufbau und die Kapselung von USB-Daten in USB-Redirection-Pakete. Der Redir-Emulator übergibt dekodierte USB-Pakete an den ausgewählten USB-Emulator und verschickt die vom USB-Emulator generierten USB-Response-Pakete.

6.3.3 USB-Enumeration-Emulator

Für die USB spezifische Emulation wird im ersten Schritt der sogenannte USB-Enumeration-Emulator verwendet. Dieser bildet die komplette USB-Enumeration nach und verschickt *Device Descriptors* an das virtualisierte System. Dazu lassen sich komplette *Device Descriptors* in Form von speziellen Datenstrukturen übergeben.

Üblicherweise erben weitere USB-Emulator-Implementierungen von der USB-Enumeration-Emulator-Klasse und übernehmen nach der erfolgreichen USB-Enumeration die Kontrolle über die eingehenden Daten. Da es sich bei der Implementierung des USB-Enumeration-Emulator um keine abstrakte Klasse handelt, lässt sich diese wie ein normaler Emulator verwenden.

6.3.4 HID Emulator

Zum Zeitpunkt dieser Arbeit, existiert bereits eine fertige Implementierung eines speziellen Function Layer USB-Emulators. Der USB-HID-Emulator ist in der Lage spezielle HID-Treiber zu laden, welche üblicherweise für Geräte wie Tastaturen, Computermäuse und Joysticks verwendet werden.

Für eine vollständige Emulation von HID-Geräten werden sogenannten HID-Reports benötigt. Diese werden bei der USB-Enumeration an das Host-System in Form von *Report Descriptors* geschickt und enthalten zusätzliche HID-spezifische Konfigurationen des HID-Gerätes. Diese HID-Reports[20] lassen sich an den USB-HID-Emulator per Parameter übergeben. In der Vergangenheit erwies sich das Fuzzing von *Report Descriptors* als attraktives Ziel für Sicherheitsforscher und Hacker[vgl. 5]. Die dort beschriebenen Schwachstellen ließen sich mit dem USB-HID-Emulator und präparierten HID-Reports reproduzieren.

6.3.5 Weitere Emulatoren

Weitere USB-Emulatoren lassen sich analog dem HID-Emulator implementieren. Die aktuellste Version des Framework enthält darüber hinaus noch einen USB-Abortion-Enumeration-Emulator. Dieser Emulator verhält sich ähnlich wie der USB-Enumeration-Emulator, bricht jedoch die USB-Enumeration nach einer vorher definierten Anzahl von Paketen, in Form eines simulierten „Abstecken“ des USB-Gerätes, ab. Das Ziel ist die Provokation von Fehlverhalten und Abstürzen, durch unvollständige Enumeration.

[20] http://www.usb.org/developers/hidpage/

6.4 QEMU/KVM Abstraktion

Das Framework basiert auf der Nutzung von QEMU in Kombination mit KVM. Der Schwerpunkt des Frameworks liegt in der Erstellung, Steuerung und der Kontrolle von QEMU Prozessen. Ein wichtiger Teil stellt hierbei das Modul QEMU-Controller da.

6.4.1 QEMU-Controller

Das Modul realisiert mit Hilfe der Python eigenen Klasse „subprocess“, die Erstellung und Kommunikation von externen Prozessen. Pro QEMU-Controller Instanz existiert ein QEMU-Prozess. Die Aufgaben des QEMU-Controller sind:

- das Starten und Beenden des eigenen QEMU-Prozesses
- die Überwachung des QEMU-Prozesses anhand der Standardausgabe auf relevante Keywords
- Das Absetzen von QEMU-eigenen Kommandos, mit Hilfe der QEMU-Monitoring-Console
- Das Disk-File Management
- Kommunikation mit einem Monitoring-Modul

6.4.2 Überwachung und Kontrolle

Der QEMU-Controller ist in der Lage selbständig QEMU-Prozesse zu starten. Der Start des externen Prozesses wird überwacht und der QEMU-Controller ist in der Lage Fehler, wie zum Beispiel ein korrumpiertes Image, zu erkennen. In diesem Fall wird das entsprechende Image automatisiert neu erstellt und der Start wird neu eingeleitet. Außerdem erlaubt der QEMU-Controller das Laden von Snapshots. Hierfür wird der entsprechende Befehl per Standardeingabe an die QEMU-Monitoring-Console übergeben und ausgeführt.

6.4.3 Starten von QEMU-Prozessen

Der QEMU-Controller ist in der Lage QEMU-Prozesse selbständig zu starten. Hierfür werden alle benötigten Informationen in speziellen Konfigurationsdateien übergeben. Abbildung 25 stellt eine beispielhafte Konfiguration dar.
Eine Konfiguration enthält folgende Information:

- der Pfad zu der QEMU Binärdatei, die gestartet werden soll
- aktivierte oder deaktivierte KVM-Unterstützung
- die Größe des konfigurierten Arbeitsspeichers für die virtuelle Maschine
- der Pfad zu der RAM-Datei, der im QCOW2-Format vorliegen muss
- der Pfad zu der Overlay-Datei, der im QCOW2-Format vorliegen muss
- der Pfad zum Ordner, in dem alle weiteren Overlay-Dateien kopiert werden sollen
- der präferierte und von QEMU unterstützte USB-Host-Controller
- weitere optionale QEMU Parameter
- der Name des Snapshots, welcher in der RAM-Datei gespeichert ist

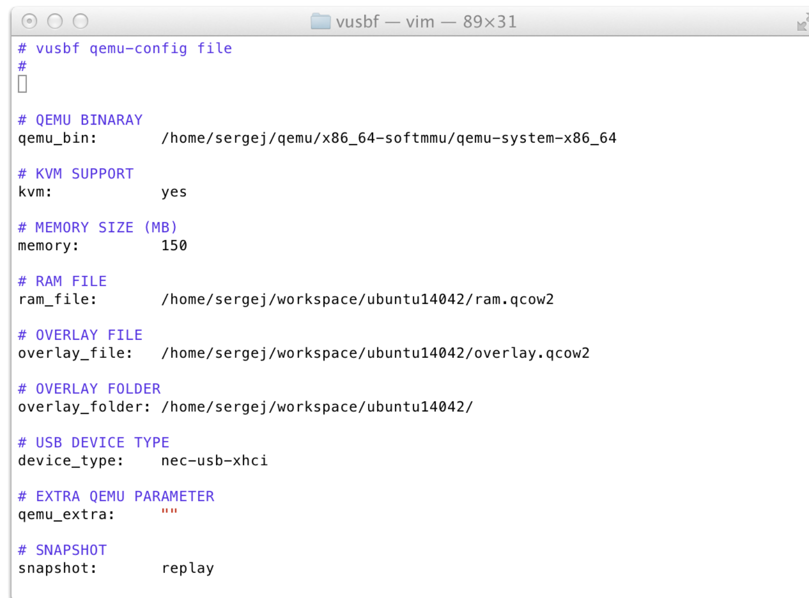

Abbildung 25: Beispielhafte vUSBf-Konfiguration für Ubuntu 14.04 Desktop

6.4.4 Monitoring

Für die Überwachung während des Fuzzings, wird ein Monitoring-Modul geladen. In der aktuellsten Version ist nur ein Monitoring-Modul für den Linux-Kernel fertiggestellt. Dieses interagiert mit dem virtualisierten Linux-System, in dem QEMU vorher eine TTY per virtuellen seriellen Port nach Außen zur Verfügung stellt. Außerdem wird das Verbosity-Level der Linux Kernel-Funktion `printk` auf das höchste Level (7) konfiguriert. Realisiert wird das mit dem Befehl:

```
echo '7' > /proc/sys/kernel/printk
```

Das Monitoring-Modul ist somit in der Lage, die kompletten Kernel-Ausgaben mitzuschneiden und zu protokollieren. Die Überwachung erfordert somit keine direkte Interaktion während des Fuzzings mit dem Linux-System. Der Vorteil an dieser Variante ist die detaillierte Ausgabe des Kernels, welcher im Fehlerfall ganze Stack-Traces und weitere Debug-Informationen ausgibt. Parallel zum Mitschneiden der Kernel-Ausgaben werden noch weitere Information wie der Zeitpunkt des letzten Snapshot-Ladevorgang protokolliert und die angewandte Testcase-ID. Diese Informationen werden alle pro Prozess in eine separate Log-Datei geschrieben und lassen sich später oder auch während des Fuzzings untersuchen. Falls während des Fuzzings keine Ausgabe des Kernels erfolgt, wird der QEMU-Controller benachrichtigt und dieser leitet ein Laden eines Snapshots bzw. den Neustart des QEMU-Prozesses ein.

Die Modularisierung der Überwachung erlaubt die Anpassung an andere Betriebssysteme für den Einsatz in vUSBf. Das Windows-Monitoring-Modul ist aktuell noch in der Entwicklung und erfordert mehr Aufwand, da ohne weiteres keine Möglichkeit besteht, die Kernel-Ausgaben auf den seriellen Port zu projizieren. Alternativ lässt sich auch eine VNC-Verbindung zum jeweiligen QEMU-Prozess kurzzeitig aufbauen und dann ein Bildschirmfoto erstellen. Anhand von diesem Bildschirm-Foto lässt sich mit Hilfe des Farbwertes auf einen typischen Bluescreen of Death und somit auf einen Absturz des Betriebssystems schliessen.

6.4.5 Image Management

Da das vUSBf-Framework auf die parallele Ausführung einer Vielzahl von QEMU-Prozessen setzt, muss hierfür pro laufendem Prozess auch eine virtuelle Maschine verfügbar sein.

Die Nutzung von QEMU impliziert für die virtuellen Maschinen die Verwendung von sogenannten Image-Dateien oder reellen Partitionen, welche für die Bereitstellung des Speicherplatzes der virtuellen Festplatten benötigt werden. Hierzu verwenden wir Image-Dateien im QCOW2-Format (QEMU Copy-On-Write). Um eine saubere Ausführung der virtuellen Maschinen zu gewährleisten, muss pro virtueller Maschine ein exklusives Image existieren. Anderenfalls kann es bei gleichzeitiger Nutzung von mehreren QEMU-Prozessen zu einer Korruption der Image-Datei kommen.

Der vom Aufwand her einfachste und gleichzeitig trivialste Weg, ist die Bereitstellung eines vollständigen VM-Image je QEMU-Prozess. Außerdem lässt sich im QCOW2-Image noch ein Snapshot hinterlegen. Das QCOW2-Image wächst in diesem Fall um ca. der Größe des Arbeitsspeichers, sowie den Festplatten- Änderungen. Die Größe der Image-Datei entspricht somit dem Snapshot, sowie dem allozierten Speicher im Image.

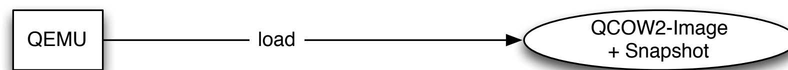

Abbildung 26: Einfacher Aufbau von QEMU und einer Image-Datei

Jeder virtuellen Maschine steht somit genau ein vollständiges Image zur Verfügung. Skaliert man die Anzahl der laufenden QEMU-Prozesse, so steigt parallel auch die Größe des Speicherverbrauches proportional an. Da dies jedoch zu einem riesigen Speicherverbrauch auf dem verwendeten Speichermedium führen kann, wurde ein anderes Vorgehen gewählt.

Das QCOW2-Format erlaubt die Erstellung sogenannter Overlay-Dateien. Diese werden mit dem ursprünglichen Image verknüpft. Ein Overlay enthält alle Änderungen seit der Erstellung, lässt das Original unberührt und verwendet dieses nur für Lesevorgänge. Würde man die Infrastruktur mit Hilfe von Overlays aufbauen, wäre das Problem des Speicherverbrauchs gelöst, da für jeden QEMU-Prozess ein exklusives Overlay erstellt werden könnte. Der Aufbau würde somit wie folgt aussehen:

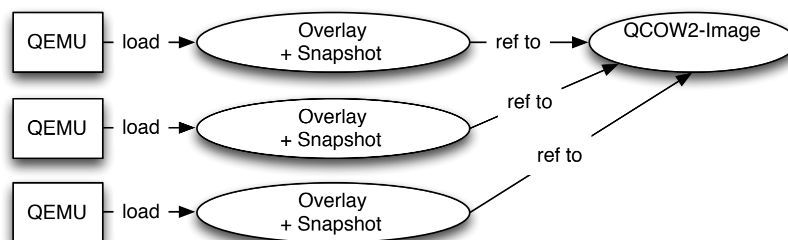

Abbildung 27: Verwendung von QCOW2-Overlay Dateien

Problematisch ist jedoch die Tatsache, dass Snapshots, welche im original Image erstellt wurden, nicht in Kombination mit einem Overlay verwendet werden können. Diese Limitierung lässt sich umgehen, in dem jedes Overlay einen eigenen Snapshot enthält. Dies würde aber wieder zu einem

großen Speicherverbrauch führen, da jedes Overlay um die Größe des Snapshots anwachsen würde, was circa der Größe des virtuellen Arbeitsspeichers entspricht.

QEMU speichert jedoch bei mehreren Image-Dateien den Snapshot immer in die zu erst geladene Image-Datei. Dieses undokumentierte Feature von QEMU lässt sich in Kombination mit einer weiteren Image-Datei elegant für die Lösung des Problems verwenden. Hierfür muss eine weitere Image-Datei erstellt werden, welche im späteren Verlauf als RAM-Image bezeichnet wird, und als erstes im Startbefehl parametrisiert werden:

```
qemu -hdb RAM.qcow2 -hda Overlay.qcow2
```

Die auf die Festplatte bezogenen Änderungen werden auf jeder Festplatte separat hinterlegt. Ein Snapshot lässt sich somit nur noch laden, wenn alle virtuellen Festplatten auch geladen werden. Der Aufbau würde wie folgt aussehen:

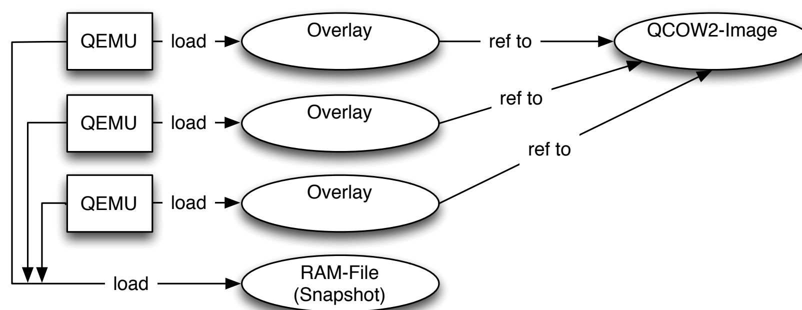

Abbildung 28: Optimale Speichernutzung durch Overlays und einer RAM-Datei

In dieser Konstellation werden nur sehr kleine individuelle Dateien pro virtueller Maschine benötigt. Das originale Image und die RAM-Datei werden von allen virtuellen Maschinen nur für Lesevorgänge verwendet und bleiben unberührt. Dies hat auch den Vorteil, dass in Abhängigkeit des verwendeten Kernels, die beiden Dateien sehr wahrscheinlich aggressiv gecachet werden, was somit einen nicht unerheblichen Geschwindigkeitesvorteil bietet. Alternativ lassen sich die beiden Dateien auch in eine RAM-Disk legen.

6.5 Parallelisierung

USB-Fuzzing war in der Vergangenheit mit einer langsamen Ausführgeschwindigkeit verbunden. Der limitierende Faktor stellt beim USB-Fuzzing die langsame USB-Enumeration dar, welche je nach Betriebssystem, bis zu mehrere Sekunden dauern kann. Trotzdem ist eine regelmäßiger Reconnect, welches die USB-Enumeration einleitet und nur so das erneute Laden eines Treibers ermöglicht, für eine umfassendes Fuzzing unabdingbar. Es gibt somit keine Möglichkeit das USB-Fuzzing mit nur einer laufenden Instanz zu beschleunigen.

Eine alternative Herangehensweise stellt das Parallelisieren des USB-Fuzzings dar. In Abhängigkeit der zur Verfügung stehenden Hardware bzw. der Ressourcen lässt sich das USB-Fuzzing beliebig parallelisieren. Die parallele Nutzung von USB-Fuzzing-Methoden, welche auf Hardware-Basis funktionieren, wie zum Beispiel das USB-Fuzzing mit Hilfe des Arduino-Boards oder dem Facedancer-Boards, setzen die selbe Anzahl an Hardware in Relation zu den parallel laufenden USB-Fuzzing-Instanzen voraus.

Da die Verwendung von USB-Emulatoren in diesem Framework, die Notwendigkeit der Verfügbarkeit von reellen physikalischen USB-Geräten de Facto ausschließt, ist eine Parallelisierung des USB-Fuzzings nur durch die zur Verfügung stehende Rechenleistung bzw. dem zu Verfügung stehenden Arbeitsspeichers limitiert. Außerdem kann das USB-Fuzzing, anders als das virtuelle USB-Fuzzing mit dem Facedancer-Board, auf einem einzigen Computersystem realisiert werden.

6.5.1 Multiprocessing

Für die Implementierung des parallelisierten USB-Fuzzings, wurde das Python-eigene Multiprocessing-Modul und nicht das Threading-Modul verwendet. Der Grund hierfür ist der sogenannte GIL (Akronym für Global Interpreter Lock) welcher Teil des CPython-Interpreter ist. CPython entspricht dem Standard-Python-Interpreter, welcher seinen Namen der Tatsache zu verdanken hat, dass dieser in C implementiert ist. Der GIL ist ein globaler Mutex, welcher die parallele Ausführung von Python-Threads in einigen kritischen Bereichen unterbindet, da einige Funktionalitäten des Interpreters nicht „Thread-Safe“ implementiert sind. Dies beruht auf der Integration von verschiedenen C-Bibliotheken, welche nicht „Thread-Safe“ sind. Somit erlaubt der CPython-Interpreter die Nebenläufigkeit von Python-Threads, jedoch nicht die vollständig parallele Ausführung. Mehr zu diesem Thema findet sich im offiziellen Wiki der Python-Website [21] [22].

Eine Möglichkeit ist der Wechsel auf einen anderen Python-Interpreter ohne GIL. Zur Auswahl stehen hier hauptsächlich Jython (der in Java implementierte Python Interpreter) und IronPython (C# basierter Python Interpreter). Problematisch ist aber hierbei das Fehlen von exklusiven CPython-Funktionen, wodurch sich zum Beispiel das benötigte Scapy-Framework nicht verwenden lässt.

6.5.2 Architektur

Die Architektur der Software entspricht im Prinzip drei Prozesstypen, welche sich untereinander über verschiedene IPC-Möglichkeit (Inter-process communication) austauschen. Die folgende Abbildung stellt die verwendete Architektur grafisch dar:

[21] https://wiki.python.org/moin/GlobalInterpreterLock
[22] http://www.dabeaz.com/python/UnderstandingGIL.pdf

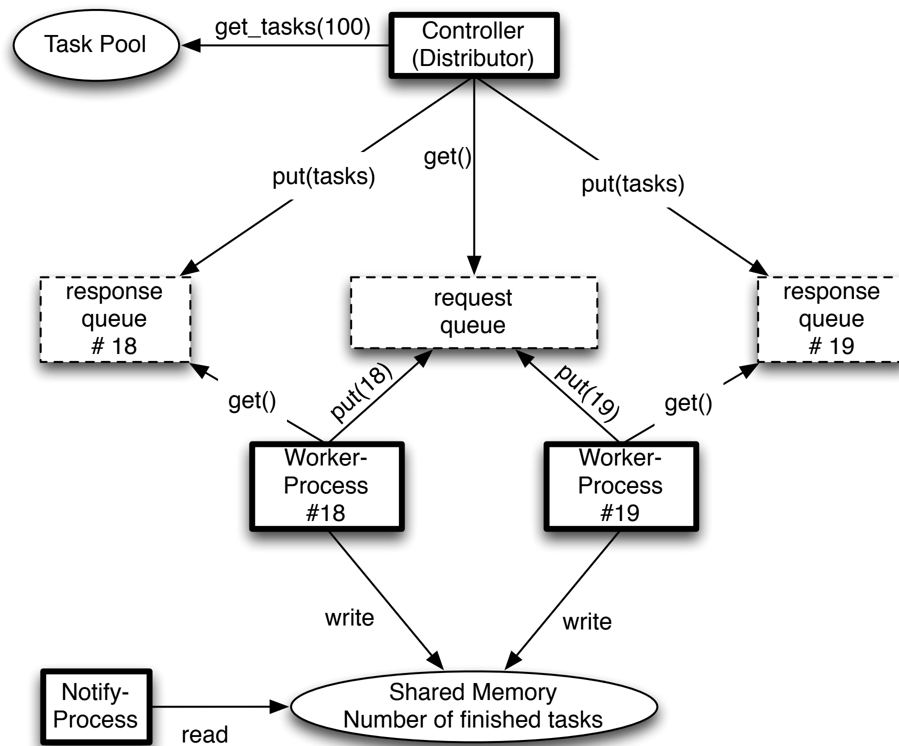

Abbildung 29: Architektur des Multiprocessing im vUSBf-Framework

Controller-Prozess:

Der Controller-Prozess wird nur einmal gestartet und hat einen exklusiven Zugriff auf einen vorher generierten „Task-Pool“, in welchem sich alle anzuwendenden Testcases befinden (siehe Kapitel 6.7).

Notify-Prozess:

Dieser Prozess dient zur Ausgabe des aktuellen Fortschritts. Die benötigten Daten liegen in einer Shared-Memory-Integer Variable.

Worker-Prozess:

Worker-Prozesse werden in Relation zu der Anzahl der parallel ablaufenden USB-Fuzzing-Instanzen gestartet. Jeder Prozess verfügt über eine eigene Instanz eines QEMU-Controllers, einem Monitoring-Modul und einem ausgewählten USB-Emulators. Die anzuwendenden Testcases werden über die Python Multiprocessing-Queues namens „request-queue“ und „response-queue“ bezogen.

6.5.3 Prozesskommunikation

Der Datenaustausch zwischen dem Controller-Prozess und den einzelnen Worker-Prozessen wird über die Queues namens „request-queue“ und „response-queue“ realisiert. Jeder einzelne Worker-Prozess verfügt über seine eigene exklusive „response-queue“. Alle Prozesse teilen sich die selbe „request-queue“. Diese wird für die Anfrage von neuen Testcases verwendet. Hierfür legt ein Worker-Prozess seine Prozess-ID in die Queue. Gleichzeitig wartet der Controller-Prozess, per get-Zugriff an der „request-queue“, auf neue Anfragen. Jede Anfrage wird mit einer Liste von Testcases an die jeweilige „response-queue“ beantwortet. Falls keine Testcases mehr verfügbar sind, antwortet der

Controller-Prozess mit einer leeren Liste von Testcases. Der jeweilige Worker-Prozess interpretiert eine leere Liste als Abbruchbedingung und beendet sich. Der Zugriff auf eine leere Queue per get()-Aufruf blockiert den entsprechenden Befehl, solange bis neue Daten in der Queue liegen.

Parallel zu dem Datenaustausch inkrementieren alle Worker-Prozesse, in regelmäßigen Abständen, den Wert der Shared-Memory-Variable. Der Notify-Prozess gibt den Wert per Standardausgabe aus und ermöglicht somit eine Darstellung des Programmfortschritts.

6.6 Clustering

Der nächste logische Schritt zu Steigerung der Ausführgeschwindigkeit, ist die Nutzung aller zur Verfügung stehenden Ressourcen. Da sich das USB-Fuzzing, wie im letzten Kapitel bereits erklärt, gut zur parallelen Ausführung eignet, war auch das Clustering bzw. verteilte Rechnen eine geeignete Möglichkeit zur weiteren Steigerung der Ausführgeschwindigkeit.

6.6.1 Architektur

Das vUSBf-eigene Clustering basiert auf einer Client-/Server-Architektur:

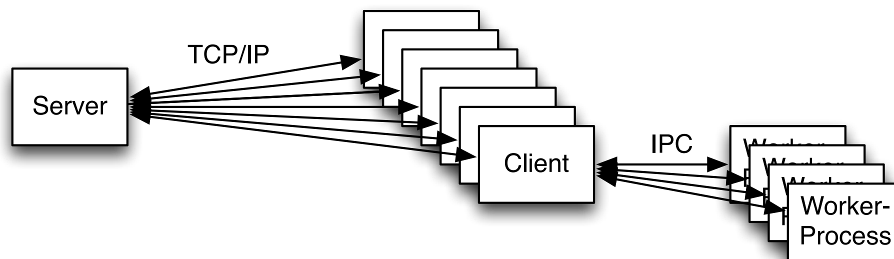

Abbildung 30: Kommunikation zwischen den Akteuren beim Clustering

Im Vergleich zur Architektur der parallelen Ausführung wurde eine weitere Rolle hinzugefügt, der Server. Dieser repräsentiert den Testcase-Verteiler, welcher sich um die Testcase-Generierung, sowie die Bereitstellung dieser kümmert. Die Clients entsprechen den Controller beim Multiprocessing, müssen jedoch für Zugriffe auf neue Testcases, Anfragen an den Server stellen. Hierfür wurde ein eigenes minimalistisches Protokoll entwickelt. Alle weiteren Prozesse verhalten sich absolut gleich dem Multiprocessing-Modus.

6.6.2 Clustering-Protokoll

Das Clustering wird über TCP/IP realisiert. Der Grund hierfür liegt in den Eigenschaften des Transmission Control Protocol. Zum einen ist die absolute Zuverlässigkeit der Übertragung, als auch die Möglichkeit der Segmentierung, fundamental für das Clustering. Das Clustering-Protokoll dient zur Übertragung von größeren serialisierten Python-Objekte, welche eine Liste von Testcases repräsentieren. Ohne TCP-Segmentierung wäre die Implementierung des Clustering-Protokolls mit einem höheren Aufwand verbunden. In dieser Version kümmert sich der TCP-Stack des Betriebssystems um die Segmentierung und nicht das Framework.

Der generische Protokoll-Header hat in der aktuellsten Version des vUSBf-Frameworks den folgenden Aufbau:

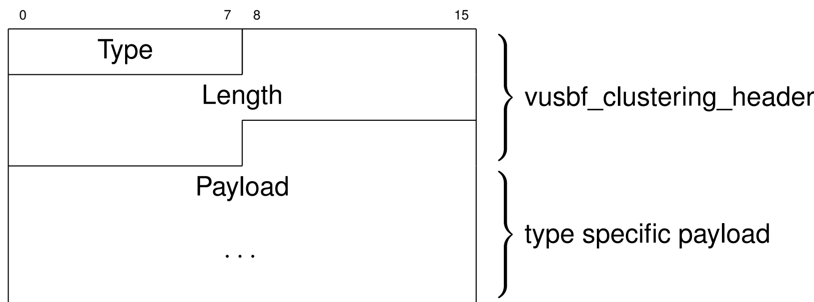

Abbildung 31: Protokoll-Header des Clustering-Protokolls

Im Feld *Type* wird der jeweilige Pakettyp spezifiziert. Das Feld *Length* gibt die Länge der *Payload* an, und entspricht der gesamten Paketlänge exklusive des Protokollheaders (8 Byte). Das entwickelte Clustering-Protokoll kennt in der aktuellsten Version des vUSBf-Frameworks acht spezielle Pakettypen:

(0) hello-Paket:
Dieser Pakettyp verfügt über keine Payload, sondern dient nur dem Verbindungsaufbau.

(1) task_request-Paket:
Dieses Paket verschickt der Client für eine Anfrage nach neuen Testcases. Die Payload von diesem Pakettyp ist wie folgt spezifiziert:
4 Byte: Anzahl der benötigten Testcases.

(2) task_response-Paket:
Dieses Paket enthält die angeforderten Testcases. Der Aufbau sieht wie folgt aus:
4 Byte: Anzahl der gelieferten Testcases.
Payload: Payload in Form eines serialisierten Python Objektes, das mit Hilfe von Python-Pickle erstellt wird. Das Python Objekt enthält eine Python-Liste aller angefragten Testcases.

(3) sync_request-Paket:
Damit der Server die Anzahl aller berechneten Testcases als Information ausgeben kann, schickt dieser in regelmäßigen Abständen (der Wert ist konfigurierbar und entspricht standardmäßig 5 Sekunden) ein sogenanntes sync_request-Paket. Dieses fordert einen Client auf, in Form eines sync_response-Paket, die Anzahl aller, bis zum aktuellen Zeitpunkt, erledigten Testcase zuschicken. Dieses Paket enthält keine spezielle Payload.

(4) sync_response-Paket:
Die Antwort auf ein sync_request-Paket enthält die Anzahl aller, auf dem jeweiligen Client, angewandten Testcases. Der Aufbau sieht wie folgt aus:
4 Byte: Die Anzahl aller erledigten Testcases.

(5) check_request-Paket:
Eine check_request-Paket wird beim Verbindungsaufbau verschickt. Dieses Paket dient dem Abgleichen von verschiedenen Konfigurationen zwischen dem Server und dem Client. In der aktuellsten Version werden nur die MD5-Hashes der Overlays- bzw. des RAM-Image abgeglichen. Somit ist sichergestellt, dass der Client auch die Testcases auf die gewünschte Konfiguration anwendet. Das Paket enthält die folgende Payload:

16 Byte: MD5_RAM
16 Byte: MD5_Overlay

(6) check_response-Paket:

Der Server bekommt den check_request und überprüft diesen mit seinen MD5-Hashes. Es wird je nach Übereinstimmung ein Paket mit einem Boolean-Wert verschickt. War der Abgleich erfolgreich, wird die Verbindung fortgesetzt. Ansonsten wird die Verbindung nach diesem Paket abgebrochen. Die Payload sieht wie folgt aus:
1 Byte: check_passed? (0x00 für nein bzw. 0x01 für ja)

(7) close-Paket:

Ohne spezielle Payload, wird dieses Paket bei einem gewünschten Verbindungsabbruch verschickt. Das Paket kann vom Server bzw. vom Client verschickt werden.

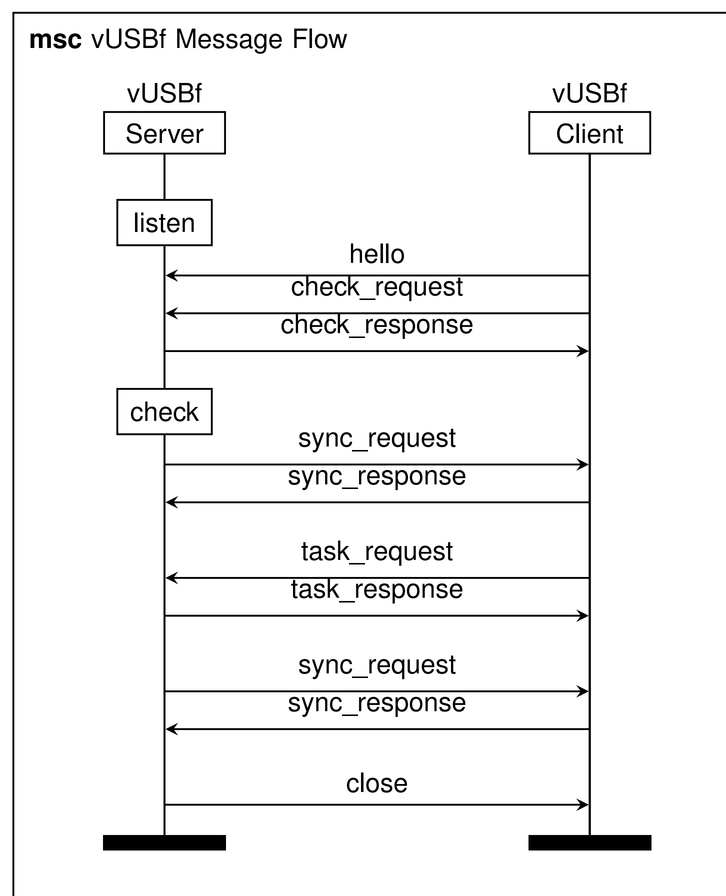

Abbildung 32: Beispielhafter Message Flow zwischen einem Server und Client

Die Abbildung 32 stellt eine beispielhafte Kommunikation zwischen einem Client und dem Server da. Da die MD5-Hashes passend sind, wird die Verbindung nicht beendet. Der Server schickt in regelmäßigen Abständen sync_request-Pakete, um die Anzahl der erledigten Testcases ausgeben zu können. Der beispielhafte Message Flow stellt auch eine Anfrage für neue Testcases dar, auf die der Server mit dem entsprechenden Paket und der Payload in Form eines serialisierten Python-Objektes antwortet. Die Verbindung wird nach einem letzten sync-Abgleich vom Server mit dem close-Paket beendet.

6.7 Dynamische Testcase-Generierung

Da das USB-Fuzzing möglichst nachvollziehbar sein soll und darüber hinaus auch ein hohes Maß an Reproduzierbarkeit bieten soll, wurde für das Framework eine eigene Fuzzing-Engine entwickelt. Die Fuzzing-Funktion des Scapy-Frameworks ist äußerst beschränkt und erlaubt nur das Fuzzing einzelner Paketfelder. Jedoch ist auch diese Funktionalität sehr limitiert, da die Werte für die Paketfelder, nur mit Zufallswerten beschrieben werden können und das Fuzzing somit nicht gesteuert werden kann. Des Weiteren wäre in diesem Fall das Fuzzing nicht reproduzierbar und kaum nachvollziehbar.

6.7.1 Verknüpfungen

Die entwickelte Fuzzing-Engine verwendet Testcase-Objekte. Diese Objekte enthalten alle benötigten Informationen über die anzuwendenden Fuzzing-Anweisungen. Außerdem enthält jeder Testcase eine eigene eindeutige Identifikationsnummer. Anhand dieser Identifikationsnummer lässt sich jeder Testcase reproduzieren. Des Weiteren erlaubt das Framework auch den Export von Sequenzen mehrerer Testcases und das Reproduzieren von diesen Testcases.

Da das Framework auf eine hohe Ausführgeschwindigkeit ausgelegt ist und die Intention eine umfassende systematische Untersuchung von einer Vielzahl von USB-Treibern durch USB-Fuzzing ist, macht es Sinn, dass hierfür automatisiert eine große Anzahl an Testcases in Form eines Testpools[23] generiert werden. Hierfür werden mehrere Testpools miteinander verknüpft. Implementiert sind in der aktuellsten Version zwei verschiedene Verknüpfungsarten. Die erste Verknüpfungsart hat den Namen „union[24]“ und erlaubt es zwei Testpools so miteinander zu verknüpfen, dass diese hintereinander ausgeführt werden. Die Anzahl der Testcases im ausgehenden Testpool entspricht im Idealfall, der Summe der Anzahl der beiden eingehenden Testpools. Die zweite Variante, welche auch als „pairwise union[25]“ bezeichnet wird, also der paarweisen Vereinigung, erlaubt es aus zwei Testpools eine Kombination von beiden zu erstellen. Die folgende Abbildung stellt die beiden Verknüpfungsarten grafisch dar:

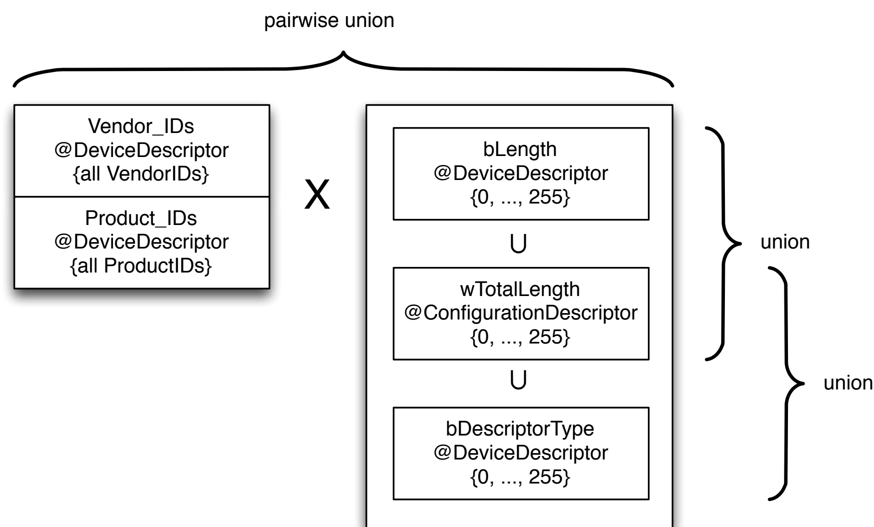

Abbildung 33: Beispielhafte Verknüpfung von mehreren Testpools

[23] Mengen bzw. Listen von Testcases, werden im folgenden als Testpool bezeichnet.
[24] $A \cup B = \{q | q \in A \vee q \in B\}$ mit A, B sind Testpools und q ist Testcase
[25] $A\ x\ B = \{X \cup Y | X \in A, Y \in B\}$ mit A, B sind Testpools und X, Y sind Testcases

Wie in Abbildung 33 dargestellt, werden drei Testpools (rechts in der Abb.) mit der Verknüpfungsart „union“ zu einem neuen Testpool verknüpft. Technisch gesehen verfügt der neue Testpool über alle drei vorherigen Testpools und diese liegen sequenziell in dem neuen Testpool vor. Im Anschluss wird per „pairwise union“ der neue Testpool mit einem Testpool, welcher über alle geordnete Paare von Vendor- und Product-ID Anweisungen verfügt[26], verknüpft. Dies führt dazu, dass der finale Testpool, pro Testcase eine Anweisung aus dem linken Testpool und dem rechten zusammengefassten Testpool verfügt. Würde der linke Testpool über alle Vendor- und Product-ID Kombinationen beispielsweise 15.000 Testcases enthalten, so lässt sich die Anzahl der Testcases in dem finalen Testpool wie folgt berechnen:

Beispielhafte Anzahl aller Testcases im finalen Testpool:

$$\underbrace{15.000}_{\substack{Vendor-ID\&Product-ID\\ Testpool}} *(\underbrace{256}_{\substack{bLength\\ Testpool}} + \underbrace{256}_{\substack{wTotalLength\\ Testpool}} + \underbrace{256}_{\substack{bDescriptorType\\ Testpool}}) = \underbrace{11.520.000}_{\substack{finaler\\ Testpool}} \quad (1)$$

Der finale Testpool würde somit über 11.520.000 Testcases verfügen. Der Vorteil an dieser Generierung von Testpools, ist die gegebene Flexibilität. Beispielsweise erlaubt dieses Vorgehen die leichte Generierung von zukünftigen Testpools. Als Beispiel sei hierbei die Verknüpfung von einer Menge von Vendor- und Product-IDs, welche nur mit der USB-Mass-Storage-Class referenziert sind, und einem SCSI-spezifischen[27] Testpool erwähnt. Somit könnte diese USB-Klasse umfassend mit Hilfe von USB-Fuzzing untersucht werden, ohne dabei Treiber zu laden, welche nicht über SCSI auf dem Function Layer kommunizieren.

6.7.2 Fuzzing-Instruktion

Ein Testcase entspricht einer Liste von Fuzzing-Instruktionen. Eine Fuzzing-Instruktion entspricht einem Tupel mit den folgenden drei Elementen:

{**Paketname**, **Feldname**, **Wert**}

Paketname:
Dieser String gibt das zu verändernde Paket an.

Feldname:
Dieser String gibt das zu verändernde Feld an.

Wert:
Dieses Feld gibt den neuen Wert für das ausgewählte Paket an.

Alternativ erlaubt das vUSBf-Framework noch das Angeben von speziellen Kommandos in Fuzzing-Instruktionen, welche das Device-Descriptor-Fuzzing (siehe. 5.3.2) ermöglichen. Technisch gesehen werden diese Fuzzing-Instruktionen durch Typüberprüfungen der einzelnen Tupel-Elemente von den anderen Fuzzing-Instruktionen unterschieden.

[26]Diese geordneten Paare bzw. Listen von Vendor- und Product-ID Kombinationen wurden aus der offiziellen Linux-USB Datenbank entnommen[vgl. 3].
[27]Einige der USB-Mass-Storage-Subklasse kommunizieren über einen SCSI-Layer[vgl. 19].

Das Tupel hat dabei den folgenden Inhalt:

{**Deskriptor-Name**, **Nummer**, **Aktion**}

Deskriptor-Name:
Dieser String gibt den zu verändernde Deskriptor-Typen an.

Nummer:
Dieser Integer-Wert gibt die Nummer des zu verändernden Deskriptor an. Falls der Wert größer als 255 ist, wird die auswählte Aktion auf jeden entsprechenden Deskriptor angewandt.

Aktion:
Dieser Integer-Wert gibt an, welches Kommando ausgeführt werden soll. Zum aktuellen Zeitpunkt sind noch nicht alle Aktionen implementiert. Geplant sind aber die folgenden Aktionen:

Value	Aktion	Beschreibung
0x0	**delete**	Lösche den ausgewählten Deskriptor bzw. den ganzen Zweig.
0x1	**add**	Füge einen neuen ausgewählten Deskriptor mit Standardwerten hinzu.
0x2	**copy**	Kopiere einen bestehenden Deskriptor.
...		

Tabelle 8: Auflistung aller Fuzzing-Aktionen

Das Fuzzing-Modul wendet Fuzzing-Instruktionen, die für das Device-Descriptor-Fuzzing erstellt wurden, direkt nach dem Laden des USB-Emulators an. Die anderen Fuzzing-Instruktionen werden während der USB-Emulation angewandt. Das Fuzzing während der USB-Emulation ist im Kapitel 6.3 näher beschrieben.

6.7.3 XML-Beschreibung

Die Testcase-Generierung wird mit Hilfe von drei verschiedenen XML-Dateien beschrieben:

test.xml:
Diese XML-Datei beschreibt alle Fuzzing-Instruktionen und die entsprechenden Werte, des hieraus resultierenden Testpools. Ein Testpool mit fünf Testcases, die bis auf den Wert absolut identisch sind, wird wie folgt in XML definiert:

```
...
    <testpool name="conf_bNum_Interface_invalid">
        <fuzz>
            <packet name="USB_Configuration_Descriptor"/>
            <field name="bNumInterfaces"/>
            <value>0</value>
            <value>5</value>
            <value>10</value>
            <value>25</value>
            <value>100</value>
        </fuzz>
    </testpool>
...
```

Alternativ lassen sich auch Fuzzing-Instruktionen für Device-Descriptor-Fuzzing in XML wie folgt beschreiben:

```
...
    <testpool name="dev_desc_fuzzing_1">
        <action type="delete">
            <descriptor name="configuration" num="1"/>
        </action>
    </testpool>
...
```

testpool:
Diese XML-Datei beschreibt die angewandten Verknüpfungen von Testpools, welche in der test.xml beschrieben sind. Eine Verknüpfung wie in der Abbildung 33. dargestellt, lässt sich in XML wie folgt beschreiben:

```
...
    <testpools name="final_testpool_1" >
        <operation type="pairwise_union" >
            <testpool name="all_vendor_product_ids" />
            <operation type="union" >
                <testpool name="dev_desc_blength_invalid" />
                <testpool name="conf_wTotalLength_invalid" />
                <testpool name="dev_bDescriptorType_invalid" />
            </operation >
        </operation >
    </testpools >
...
```

execution.xml:
Die execution.xml enthält alle Testpools, welche für das USB-Fuzzing erstellt werden sollen. Darüber hinaus bietet diese Datei noch die Möglichkeit einen USB-Emulator, den *Device Descriptor* und den Fuzzing-Modus (Reload bzw. Non-Reload) auszuwählen. Ein Ausschnitt einer beispielhaften execution.xml sieht wie folgt aus:

```
...
    <execute name="ex1">
        <testpools name="testcase_1"/>
        <emulator name="abortion_enumeration" descriptor="desc3.txt" reload−vm="yes"/>
    </execute>
...
```

7 Ergebnisse

Da zum Zeitpunkt dieser Arbeit die Entwicklung des vUSBf-Framework noch nicht abgeschlossen ist, ist davon auszugehen, dass sich die Performance weiter verbessern lässt und die Anzahl der gefundenen Fehler bzw. Sicherheitslücken weiter ansteigt.

7.1 Performance

Das vUSBf-Framework ermöglicht eine signifikante Steigerung der Ausführgeschwindigkeit, im Vergleich zu anderen USB-Fuzzing-Lösungen. Hierbei ist jedoch zu unterscheiden zwischen den beiden USB-Fuzzing-Modi und der zur Verfügung stehender Rechenleistung.

Das vUSBf-Framework bietet den Ausführmodus „Reload“, der dafür sorgt, dass nach jedem virtuellen „Einstecken“ eines virtuellen USB-Gerätes der Zustand der virtuellen Maschine durch einen Snapshot wiederhergestellt wird. Der Vorteil bei dieser Variante ist, dass damit Testcases und deren Wirkung auf das zu untersuchende System isoliert werden. Dieser Modus erkennt Testcases, die alleine ohne weitere Mitwirkung eine Reaktion auslösen.
Die zweite Variante ist der Modus „Non-Reload“, welcher dem Facedancer-artigen Fuzzing nachempfunden ist. Bei diesem Modus wird der Zustand der virtuellen Maschine nur im Fehlerfall wiederhergestellt. Das heißt, falls der USB-Stack nicht mehr auf ein virtuelles „einstecken“ reagiert, wird der definierte Zustand, mithilfe eines Snapshots, wiederhergestellt. Dies hat unter anderem den Vorteil, dass mit diesem Modus Fehler oder Schwachstellen gefunden werden können, welche nur dann zu Abstürzen oder anderen fehlerhaften Reaktionen führen, falls eine Sequenz von bösartigen USB-Geräten eingesteckt wird. Gleichzeitig ist dieser Modus schneller.

Die folgenden Performance-Benchmarks wurden auf einem System mit je vier Intel Xeon E5-2630L Prozessoren und je 64GB RAM durchgeführt. Jeder Prozessor verfügt dabei über vier Kerne. Es standen je drei baugleiche Systeme zur Verfügung. Für das Clustering wurden alle drei Systeme verwendet, wobei einer parallel auch als Server, im Sinne des vUSBf-Clustering, diente. Anzumerken sei jedoch, dass in der aktuellen Implementierung nicht die komplette Leistung genutzt werden kann.

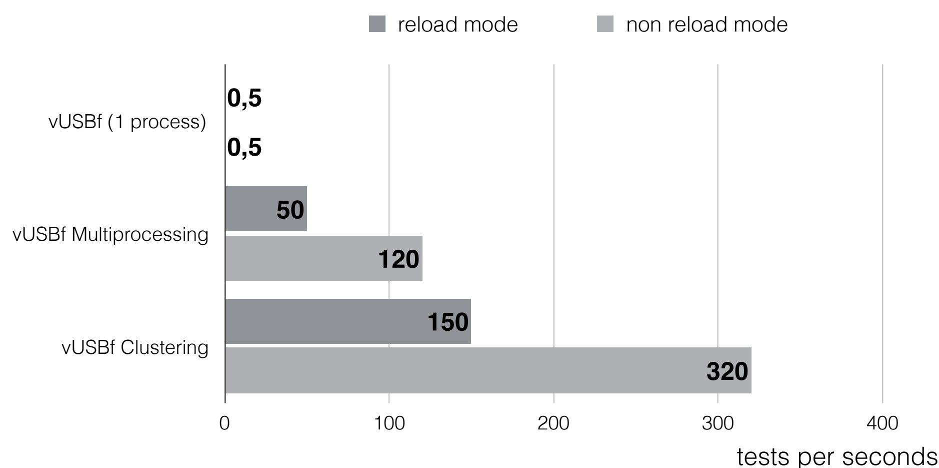

Abbildung 34: Performance im Vergleich

Die aufgeführten Werte zeigen eine Leistungssteigerung mithilfe von Multiprocessing um das etwa

100- bzw. 240-fache in Relation zur sequentiellen Ausführung. Die Performance skaliert beim Clustering mit baugleichen Servern um das 3-fache. In diesem Fall ist das eine Leistungssteigerung um das 300- bzw. 640-fache.

Die Relevanz der Leistungssteigerung wird anhand der benötigten Berechnungszeit eines beispielhaften Testpools der Größe von 1.000.000 Testcases deutlich:

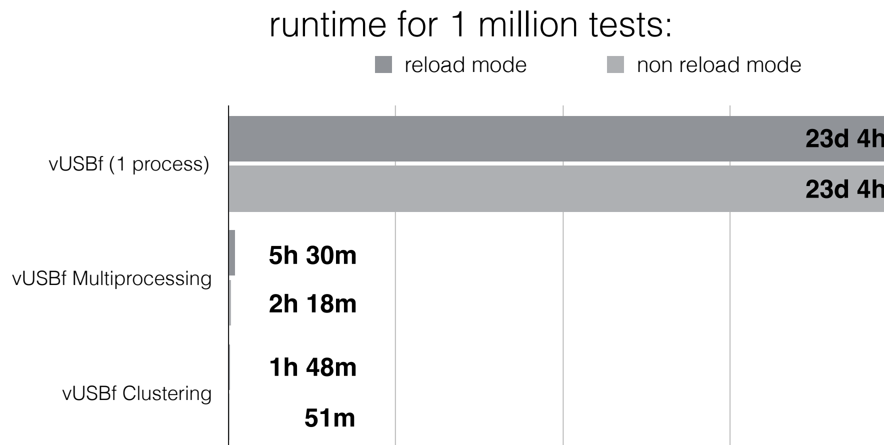

Abbildung 35: Performance in Relation zur Laufzeit

Die dargestellten Laufzeiten verdeutlichen den Nutzen des vUSBf-Frameworks und die Möglichkeit, eine systematische Untersuchung von mehreren Gerätetreibern zu realisieren. Der limitierende Faktor entspricht hierbei nur der zur Verfügung stehenden Anzahl der Rechner bzw. der gegebenen Rechenleistung.
Die aktuelle Version stößt an die Grenze der C-Funktion `select()`, welche auch von CPython verwendet wird. Die Funktion ist nur in der Lage, eine endliche Menge an Dateideskriptoren zu vergeben. Dieses Limit ist als Konstante in der Linux Header-Datei
`Linux/include/linux/posix_types.h` definiert:

```
...
#undef __FD_SETSIZE
#define __FD_SETSIZE 1024
 ...
```

Die Man-Page von `select()` beschreibt ebenfalls dieses Limitierung: „An fd_set is a fixed size buffer. Executing FD_CLR() or FD_SET() with a value of fd that is negative or is equal to or larger than FD_SETSIZE will result in undefined behavior“[vgl. 4]. In Zukunft wird diese Limitierung im vUSBf-Framework wahrscheinlich durch die Verwendung von der Funktion `poll()` umgangen.

7.2 Fehler in USB-Geräte-Treibern

Zum aktuellen Zeitpunkt wurden etliche Abstürze und Fehlverhalten von Linux-USB-Treibern beobachtet. Es lassen sich somit bereits erste Erkenntnisse, im Bezug auf die Eingabevalidierung von

USB-Geräte-Treibern im Linux-Kernel, schliessen. Das USB-Fuzzing mit dem vUSBf-Framework beschränkt sich zum aktuellen Zeitpunkt nur auf den Linux-Kernel. Die beobachteten Abstürze und Fehlverhalten lassen sich mit dem vUSBf-Framework fast alle reproduzieren. Dies erlaubt die zukünftige Untersuchung von Schwachstellen in den jeweiligen USB-Treibern.

Alle beobachteten Fehlverhalten beziehen sich nur auf „Vendor-Specific“-Treiber, und somit nicht auf Treiber für generische USB-Klassen. Außerdem beschränkt sich das USB-Fuzzing momentan nur auf die USB-Enumeration. Durch das USB-Fuzzing im Enumeration-Vorgang ist nur die Treiber-Initialisierung betroffen und somit betrifft die Code-Coverage auch nur diesen kleinen Teil. Weitere Untersuchungen lassen sich durch die Implementierungen von zusätzlichen, spezifischen USB-Emulatoren erreichen.

7.2.1 Fehler reproduzieren

Um einen gefunden Fehler zu reproduzieren, lässt sich die benötigte Payload an Hand der erstellten Log-Dateien feststellen und dem vUSBf-Framework als Parameter übergeben. In den Log-Dateien ist hierfür die eindeutige Testcase-Identifikationsnummer, die Kernel-Ausgaben nach dem Abschicken der jeweiligen Payload und der Zeitpunkt des letzten Ladens eines Snapshots hinterlegt.

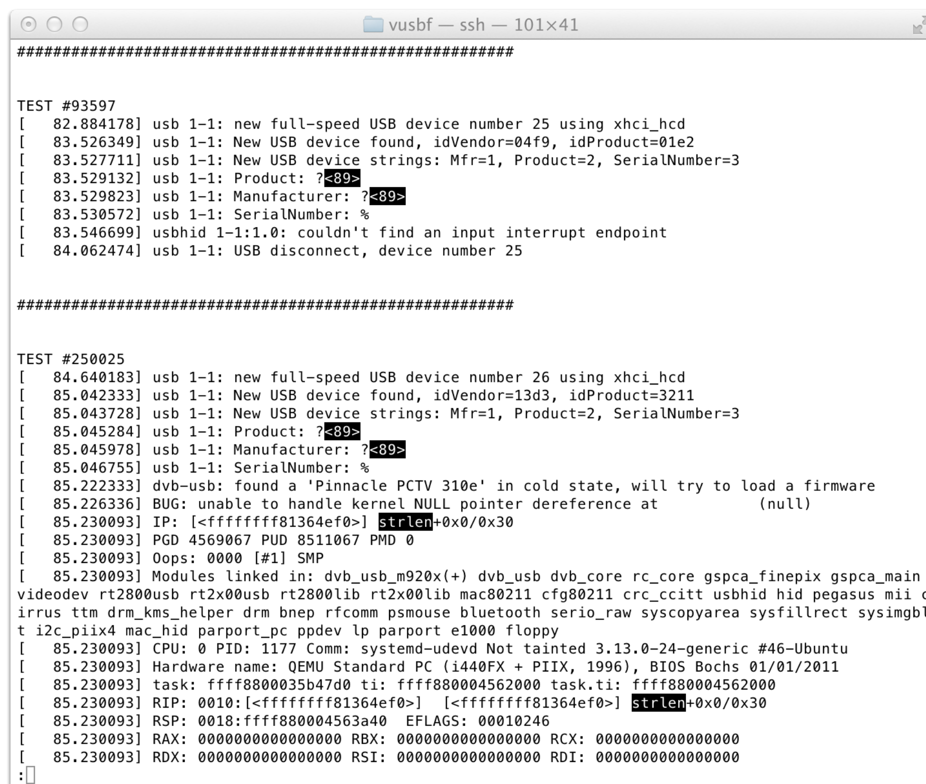

Abbildung 36: Ausschnitt aus einer vUSBf Log-Datei

Mithilfe dieser Informationen lassen sich, in Abhängigkeit des verwendeten Modi, die benötigten Testcases bestimmten. Im Falle des „Reload“-Modus wird für die Reproduzierbarkeit eines Fehlers nur die entsprechende Testcase-ID, welche mit dem Kernel-Fehler referenziert ist, benötigt. Im Falle des alternativen „Non-Reload“-Modus muss unter Umständen jede Testcase-ID zwischen dem Fehlereintritt und dem vorherigen Laden eines Snapshots, bestimmt werden.

Danach lassen sich mithilfe dieser Informationen spezielle vUSBf-kompatible Payload-Dateien erstellen. Diese Dateien entsprechen einer Liste von den angegebenen Testcases, sowie allen darin enthaltenen Fuzzing-Instruktionen. Diese werden mithilfe des Python-Pickle Modul[vgl. 2] serialisiert und danach in eine Datei geschrieben. Das Framework kann diese Dateien jederzeit einlesen, deserialisieren und daraus die benötigte Payload auslesen.

Die aktuellste Version erlaubt auch das Auslesen und Ausgeben von allen bisher gespeicherten Payloads. Zusätzlich lässt sich auch ein Beschreibungstext für jede archivierte Payload hinterlegen. Die Payloads lassen sich jeder Zeit laden, anwenden und auflisten (siehe Abb. 37.).

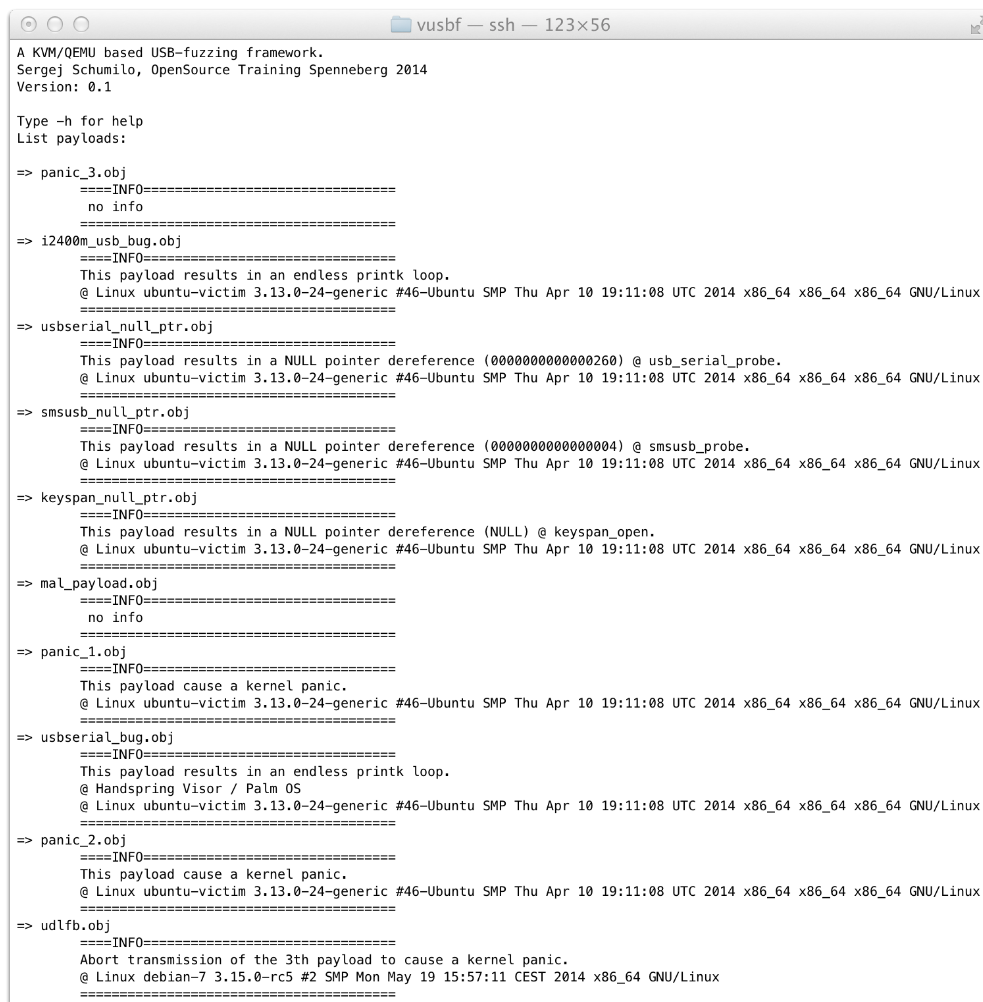

Abbildung 37: Payload-Archiv der aktuellsten Version

7.2.2 Auswertung

Es wurden etliche Fehler in aktuellen Linux-Kernel Versionen gefunden. Hauptsächlich ist hierbei nur ein kleiner Teil der Treiber anfällig. Zu den gefundenen Fehlern zählen die folgenden Klassifizierungen:

- Null-pointer dereferences
- Kernel paging requests
- Bad page state
- Segfault
- Kernel panic
- Kernel BUG

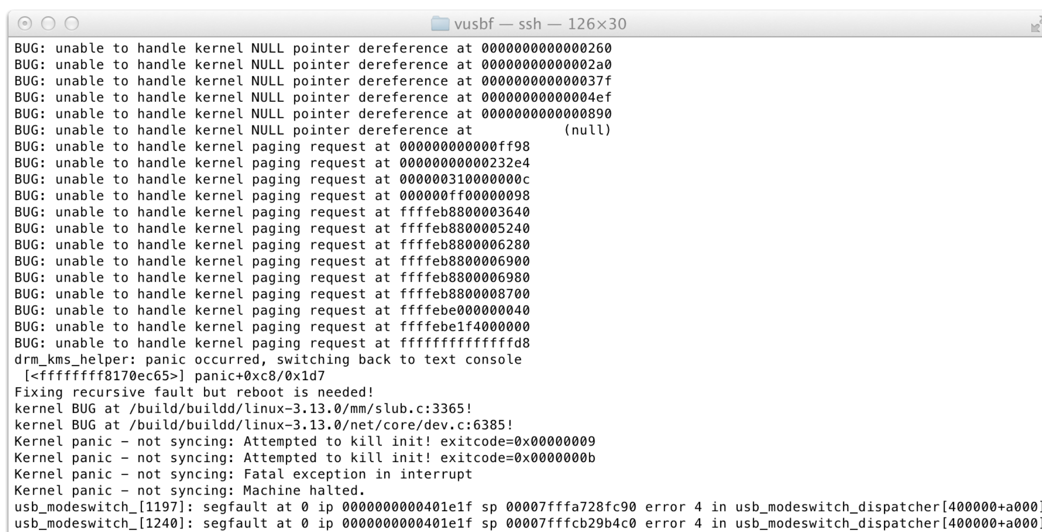

Abbildung 38: Selektierter Auszug aus den Log-Dateien (Ubuntu 14.04 LTS Desktop)

Im Modi „Non-Reload“ wurden sogar Fehler gefunden, die sich durch die Abhängigkeit geladener Treiber triggern lassen. Hierzu zählen unter anderem Fehler in den Treibern `udlfb`, `r8192u_usb` und `hfa384x` welche zu einem Absturz des Systems und zu einer Kernel Panic führen. Auch wenn einige dieser Treiber im „Staging“-Status sind, so werden diese von vielen Linux-Distributionen mit dem Kernel ausgeliefert. Des Weiteren wurden teilweise neue Fehler in neueren Kernel-Versionen entdeckt. Es kann daher sinnvoll sein, auch einen Fokus auf neue Kernel-Versionen zu legen. Da sämtliche Fehler zum Zeitpunkt dieser Arbeit noch nicht untersucht sind, ist die Einschätzung über ihre Ausnutzbarkeit nur schwer möglich.

Anzumerken ist auch die Tatsache, dass die meisten gefundenen Fehler nur zu einer Kernel Oops führen und somit im schlimmsten Fall nur Teile eines Kernel-Subsystems zum Absturz bringen. Kritisch ist aber die Tatsache, dass der Linux-Kernel eine Build-Option namens „Panic-on-Oops“ hat, welche bei jeder auftretenden Kernel Oops zu einer Kernel Panic führt und das System somit zum Absturz bringt. Unter anderem verwendet Red Hat Enterprise Linux und CentOS diese einkompilierte Kernel-Option. Sie sind somit anfällig für eine Vielzahl von eigentlich harmlosen Fehlern, da diese Fehler sich in diesem Fall als „Denial of Service“-Attacke verwenden lassen.

```
Call Trace:
 [<ffffffff815271fa>] ? panic+0xa7/0x16f
 [<ffffffff8152b534>] ? oops_end+0xe4/0x100
 ...
```

7.2.3 KGDB Debugging

In Kombination von QEMU, der Verwendung von Snapshots, zum Wiederherstellen eines festgelegten Zustands, und der Nutzung der vUSBf-Payload-Dateien, ist die Reproduzierbarkeit im hohen Maße gegeben. Diese Tatsache ermöglicht auch, in Kombination mit der QEMU-eigenen KGDB[28]-Schnittstelle, die einfache Untersuchung der Auswirkung verschiedenster Payloads auf die jeweiligen Treiber.

KGDB[29] ist ein Kernel Debugger, der sich zum Debuggen des Linux-Kernel, FreeBSD und NetBSD eignet und auch Teil der vorgestellten Betriebssystem-Kerne ist. Die Kommunikation geschieht per seriellen Port oder in der Variante KGDBoE[30] auch per UDP/IP. QEMU bringt hierfür bereits einen eigenen GDB-Server mit und kommuniziert intern per virtuellen seriellen Port. Das Debuggen per QEMU und KGDB ist vergleichbar mit dem Debuggen einer Applikation im User-Space. Um die Funktionalität zu verwenden, genügt es bereits zum Startbefehl noch den folgenden Parameter hinzuzufügen:

```
-gdb tcp::[PORT]
```

In Abhängigkeit des verwendeten Linux-Kernels, fehlen möglicherweise die Debugsymbole. In diesem Fall ist ein erneute Kompilierung des Kernels unabdingbar. Hierfür muss vorher jedoch die Option „Compile the kernel with debug info“ im Menu „Kernel Hacking“ aktiviert werden. Die Einstellungen lassen sich mit dem Konfigurationsprogramm `make menu-config` bzw. `make menuconfig` einstellen.

[28] https://kgdb.wiki.kernel.org/index.php
[29] http://www.linux-magazine.com/Online/Features/Qemu-and-the-Kernel
[30] KGDB over Ethernet

8 Fazit

In der Vergangenheit wurden immer wieder die fehlenden Sicherheitskonzepte und Gefahren bei der Verwendung des USB aufgezeigt. Spätestens seit der BadUSB-Veröffentlichung, fand eine umfassende Sensibilisierung für konzeptionelle Sicherheitsschwachstellen, auch in der Öffentlichkeit, statt. Darüberhinaus wurden immer wieder neue Schwachstellen in den Implementierungen der USB-Gerätetreiber gefunden, welche in Einzelfällen schwerwiegende Kompromittierungen erlauben.

Bewährt hat sich für die Suche nach neuen Fehlern die USB-Fuzzing-Methode. Hierfür werden unerwartete oder ungültige Daten an den entsprechenden USB-Treiber geschickt, um Fehlverhalten oder Abstürze zu provozieren, wodurch dann auf Fehler in der Implementierung geschlossen und durch nachträgliche Untersuchungen die Ausnutzbarkeit bestimmt werden kann. Leider ist die Suche mit Hardware-basierten Lösungen sehr zeitaufwendig.

Das entwickelte USB-Fuzzing-Framework vUSBf ermöglicht eine automatisierte Suche nach solchen Schwachstellen und erlaubt eine Steigerung der Ausführgeschwindigkeit um fast drei Größenordnungen, im Vergleich zu sequentiell arbeitenden USB-Fuzzing Lösungen. Realisiert wird dies durch den Einsatz von etlichen parallel arbeitenden virtuellen Maschinen, welche hierfür QEMU in Kombination mit KVM nutzen, und die Verwendung von USB-Emulatoren, welche die benötigten USB-Daten bereitstellen, ohne physikalische Hardware vorauszusetzen. Durch zusätzliche Rechenleistung und der Nutzung des Clustering-Protokolls, ist die Ausführgeschwindigkeit beliebig skalierbar. Des Weiteren werden gefundene Fehler in virtuellen Umgebungen umfassend dokumentiert und sind fast immer reproduzierbar.

Bereits während der ersten Testläufe des Frameworks, wurden etliche Fehler in verschiedensten USB-Treibern entdeckt. Es ließen sich auch bereits bekannte Gerätetreiber-Bugs, welche von anderen Sicherheitsforschern entdeckt wurden, ebenfalls mit dem Framework reproduzieren. Jedes Betriebssystem, welches sich mit Hilfe von KVM und QEMU virtualisieren lässt, lässt sich auch mit Hilfe von diesem Framework systematisch untersuchen. Hardware-Lösungen, wie zum Beispiel das Facedancer-Board, stellen derzeit immer noch die einzig nutzbare Möglichkeit für das USB-Fuzzing von nicht virtualisierbaren Systemen dar.

Insgesamt lässt sich zusammenfassend sagen, dass das entwickelte vUSBf-Framework eine umfassende, automatisierte und systematische Untersuchung von USB-Gerätetreibern auf Implementierungsschwachstellen bietet und auch die Endeckung bisher unbekannten Fehlern in der Implementierung ermöglicht. Das vUSBf-Framework ist darüberhinaus vielseitig einsetzbar, wie zum Beispiel für Regressionstests bei der Treiberentwicklung oder für die Suche nach neuen Schwachstellen, und lässt sich durch die zur Verfügung gestellten Schnittstellen um weitere Funktionalitäten erweitern. Dank des Datei-Exportes einer Sequenz von Testcases, kann der Debug-Prozess für Kernel- oder Treiber-Entwickler in Zukunft erleichtert werden bzw. der Fehler-Report kann für sie nachvollziehbarer sein.

9 Ausblick

vUSBf stellt momentan nur ein Framework zur Verfügung. Dieses muss für seine komplette Funktionalität um weitere Bausteine ergänzt werden. Mögliche Bausteine sind die folgenden.

9.1 Facedancer-Interface

Eine vielversprechende Funktionalität könnte die Implementierung einer Facedancer-Schnittstelle sein. Der Facedancer hat den großen Vorteil, dass gefundene Fehler in USB-Geräte-Treibern auch unter reellen Bedienungen auf echter Hardware verifiziert werden können. Es kann somit ausgeschlossen werden, dass ein gefundener und reproduzierbarer Bug nur ein Artefakt der Virtualisierung ist.

Problematisch ist jedoch die Interaktion zwischen der Facedancer-API und dem vUSBf-Framework. Das vUSBf-Framework wurde, unter anderem wegen dem Scapy-Framework, in Python 2.x implementiert Die Facedancer-API ist jedoch in der neuen Python 3 Version geschrieben. Aktuell stellen die unterschiedlichen Versionen die größte Schwierigkeit dar. Eine Möglichkeit wäre jedoch eine generische Kommunikation zwischen den beiden Programmen. Als Beispiel wäre hier die Kommunikation per IPC zu nennen. Des Weiteren müssen alle USB-Emulatoren des vUSBf-Frameworks auch kompatibel zur Facedancer-API sein.

9.2 Arduino-Binary Generierung

Ein alternativer Weg der Verifikation von gefundenen Fehlern mit Hilfe von physikalischer Hardware, ist die Nutzung des Arduino-Boards. Da das Arduino-Board, anders als der Facedancer, kein Schnittstelle zur direkten Direktion von USB-Daten hat, muss die Payload in Form einer Firmware bzw. einer Binärdatei für das Board vorliegen.

Die Idee ist die automatisierte Erstellung von Firmware-Dateien für das Arduino-Board. Diese Firmware würde der Logik von einer oder mehreren Testcase in Kombination mit einem USB-Emulators entsprechen. Hierzu müssen die in Python implementierten USB-Emulatoren in C-Code übersetzt werden, um hieraus dann eine entsprechende Binärdatei zu generieren.

Inwiefern diese Variante realisierbar ist, ist momentan noch nicht ganz abzuschätzen. Jedoch wäre der Nutzen, der hieraus resultiert, besonders groß, da mit der automatisierten Erstellung, gefundene Fehler direkt verifiziert werden könnten und die Hardware absolut portabel wäre. Anders als beim Facedancer wäre kein zweiter Rechner erforderlich. Das Argument der Portabilität geht sogar so weit, dass mit den erstellten Binärdateien auch Arduino-kompatible USB-Sticks, wie zum Beispiel der LeoStick[31], verwendet werden könnte. In Kombination mit einem Gehäuse, wäre der LeoStick von anderen USB-Speichersticks optisch nicht zu unterscheiden und könnte für Penetrationstests eingesetzt werden.

9.3 Microsoft Windows Unterstützung

Ein weiterer wichtiger Punkt ist die Unterstützung von alternativen Betriebssystemen. Das vUSBf-Framework bietet momentan nur eine Unterstützung für Linux-Systeme bzw. anderen UNIX-artigen

[31] http://www.freetronics.com/products/leostick

Systeme. Die Unterstützung ist zum aktuellen Zeitpunkt abhängig von der Möglichkeit, Kernel-eigene Informationen über den virtuellen seriellen Port nach Außen verfügbar zu machen.

Für die Unterstützung von Microsoft Windows, muss eine Überwachung des Systems während des USB-Fuzzings gewährleistet werden. Da Windows keine Möglichkeit besitzt, Kernel-Ausgaben von Hause aus direkt an den seriellen Port zu übergeben[32], müssen hierfür andere Lösungskonzepte erarbeitet werden. Ein Möglichkeit würde zum Beispiel das automatisierte Auswertung eines Bildschirmfoto darstellen. Dieses könnte über eine kurze Verbindung zum VNC-Port der laufenden QEMU-Instanz geschehen und würde, mithilfe einer Farbwert-Analyse, Aufschluss über den aktuellen Zustand des Betriebssystem liefern. Über den Farbwert des Bildschirmfotos lässt sich auf einen typischen Bluescreen schliessen. Eine weitere Möglichkeit wäre die Implementierung eines eigenen Treibers, welcher im Kernel-Space laufen und Kernel-eigene Informationen über den seriellen Port ausgeben würde. Beide Varianten müssen noch evaluiert werden.

9.4 Bug Reports

Aufgrund der hohen Anzahl der gefunden Fehler in verschiedensten Kernel-Versionen, werden in absehbarer Zukunft diese auch auf ihre Ausnutzbarkeit hin untersucht. Es werden darüber hinaus auch „Bug Reports“ und gegebenenfalls auch eigene Patches folgen. Aufgrund der Tatsache, dass der Schwerpunkt in dieser Arbeit in der Implementierung der USB-Fuzzing Infrastruktur lag, wurden zum aktuellen Zeitpunkt noch keine Fehler gemeldet bzw. eigene Patches entwickelt. Die hohe Anzahl der gefunden Fehler erfordert viel Zeit für die Untersuchung und es ist noch nicht abzusehen wann dieser Prozess abgeschlossen ist. Da das vUSBf-Framework auch auf der Blackhat Europe 2014 vorgestellt wird, ist davon auszugehen, dass sich weitere Entwickler und Sicherheitsforscher mit den gefundenen Fehlern beschäftigen werden. Aufgrund der Funktionalität zum Erstellen von portablen vUSBf-kompatiblen Payload-Dateien, könnten diese auch den Kernel-Entwicklern helfen, den Debug-Prozess zu beschleunigen bzw. möglichst transparent zu gestalten.

[32] Bis auf den Einsatz von WinDBG. Leider ist diese Methode in dem vUSBf-Framework aktuell nicht praktikabel, da WinDBG über ein proprietäres Protokoll mit dem Zielsystem kommuniziert. Es existieren jedoch inoffizielle Dokumentationen des Protokolls[vgl. 13]

Literaturverzeichnis

[1] AutoRun changes in Windows 7. http://blogs.technet.com/b/srd/archive/2009/04/28/autorun-changes-in-windows-7.aspx.

[2] Python 2.x documentation: Python object serialization. "https://docs.python.org/2/library/pickle.html".

[3] The USB ID Repository. http://www.linux-usb.org/usb-ids.html.

[4] Ubuntu Manpage: select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O. "http://manpages.ubuntu.com/manpages/raring/man2/select.2.html".

[5] USB-Tastatur kapert Linux-Kern. http://www.heise.de/security/meldung/USB-Tastatur-kapert-Linux-Kern-1947516.html.

[6] Adrian Crenshaw. Programmable HID USB Keystroke Dongle, DEF CON 18. https://www.defcon.org/images/defcon-18/dc-18-presentations/Crenshaw/DEFCON-18-Crenshaw-PHID-USB-Device.pdf, 2010.

[7] Andy Davis. Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions, Black Hat USA 2013. "https://media.blackhat.com/us-13/US-13-Davis-Deriving-Intelligence-From-USB-Stack-Interactions-WP.pdf", 2013.

[8] N. G. Andy Davis. Lessons learned from 50 bugs: Common USB driver vulnerabilities. https://www.nccgroup.com/media/190706/usb_driver_vulnerabilities_whitepaper_january_2013.pdf.

[9] Andy Greenberg. Why the Security of USB Is Fundamentally Broken. http://www.wired.com/2014/07/usb-security/.

[10] Darrin Barrall and David Dewey. ”Plug and Root,” the USB Key to the Kingdom, Black Hat USA 2005. "https://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Barrall-Dewey.pdf", 2005.

[11] Hans de Goede. USB Network Redirection protocol description. "https://github.com/SPICE/usbredir/blob/master/usb-redirection-protocol.txt".

[12] Hans-Joachim Kelm. USB 2.0: Studienausgabe, 2006.

[13] Joe Stewart. Just Another Windows Kernel Perl Hacker, Black Hat USA 2007. https://www.blackhat.com/presentations/bh-usa-07/Stewart/Presentation/bh-usa-07-stewart.pdf.

[14] Karsten Nohl, Sascha Krißler and Jakob Lell. BadUSB — On accessories that turn evil. https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf, 2014.

[15] A. G. Michael Sutton and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison Wesley Pub Co Inc, Auflage 1, Seite 28, 2007.

[16] Moritz Jodeit, Martin Johns. USB Device Drivers: A Stepping Stone into your Kernel. "http://www.jodeit.org/research/EC2ND_USB_Device_Drivers.pdf", 2010.

[17] Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/, 2010.

[18] Travis Goodspeed, Sergey Bratus. Emulating USB Devices with Python. http://travisgoodspeed.blogspot.de/2012/07/emulating-usb-devices-with-python.html, 2012.

[19] USB Implementers Forum. Universal Serial Bus Mass Storage Class Specification Overview. "http://www.usb.org/developers/docs/devclass_docs/Mass_Storage_Specification_Overview_v1.4_2-19-2010.pdf".

[20] USB Implementers Forum. USB 2.0 Dokumentation. "http://www.usb.org/developers/docs/usb20_docs/".

[21] USB Implementers Forum. USB - Language Identifiers. "http://www.usb.org/developers/docs/USB_LANGIDs.pdf", 2000.

[22] I. van Sprundel. Fuzzing: Breaking software in an automated fashion. http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf, 2005.

[23] Wikimedia Foundation. The two types of hypervisors for virtualization. "http://commons.wikimedia.org/wiki/File:Hyperviseur.png", 2011.

Alle Weblinks wurden am 14. Oktober 2014 auf ihre Erreichbarkeit geprüft.