基于嵌入式 Linux 的 USB 中间人攻击设计与实现

摘要

USB（Universal Serial Bus，指的是通用串行总线）经过多年的发展，USB 接口已经在各种设备中的普及，成为入侵和窃取数据的主要途径之一。近年来，攻击者逐渐将目光转移到 USB 固件、驱动和硬件层面的攻击，如 BadUSB 等通过专用硬件或篡改设备固件方式实现的恶意硬件攻击。

本文使用树莓派和嵌入式 Linux 设计并实现了一种 USB 中间人攻击设备。依托于 Linux 内核良好的软件工程设计完成硬件层面与软件代码层面的部分解耦，脱离传统且复杂的硬件固件攻击方式。通过树莓派拦截在 USB 设备与主机设备之间，由树莓派的 USB 外围设备（peripheral）模式模拟 USB 设备，实现记录、转发、篡改和重放 USB 数据，达到 USB 中间人攻击的效果。

通过 USB 中间人攻击设备的设计与实现，探讨当下大量应用 USB 协议的计算机生态环境下，所面临的严峻信息安全问题。

关键词：嵌入式，Linux，USB，BadUSB，中间人攻击

目 录

1 绪论

1.1 引言

USB（Universal Serial Bus，指的是通用串行总线）是一种计算机与外设数据通讯的串行通讯总线，自 1994 年 USB 0.7 推出以来，成功取代了传统且兼容性较差的并口和串口。随着科技的不断发展，USB 因其强大的兼容性和扩展性逐渐成为各种终端设备的主流接口，在个人电脑和移动设备中被广泛应用。USB 接口目前已经发展到 USB 4.0 版本，其强大的功能和丰富的优点，为用户提供了许多如 USB 键鼠、USB 存储、USB 摄像头等多种类型的输入输出设备。

USB 接口和其设备已经成为计算机系统中不可或缺的一部分，其种类和数量极其丰富，而 USB 接口也成为入侵和窃取数据的途径之一。

1.2 课题研究背景与现状

传统的 USB 攻击主要是利用 U 盘、移动硬盘等储存设备作为载体，通过恶意软件作为有效荷载进行攻击。这种形式的攻击往往是利用操作系统漏洞和用户的安全意识匮乏，然而近年来随着操作系统和安全软件的发展以及安全意识的普及，恶意软件和传统 USB 存储设备的攻击方式逐渐失去生存空间。攻击者也逐渐放弃这种传统的攻击方式，逐渐将目光转移到 USB 固件、驱动和硬件层面的攻击。

2014 年曝光的“棱镜门”事件中，披露了一种使用 USB 接口作为攻击途径的小型设备，可以在计算机物理隔离的情况下窃听敏感数据。

2014 年美国黑帽子大会上,由研究人员 Jakob Lell 和 Karsten Nohl 首次展示了 BadUSB 攻击。该攻击属于 HID (Human Interface Device, 指的是计算机直接与人交互的设备, 例如键盘、鼠标等) 攻击的一种,通过改写 USB 设备底层固件将其伪装成无害的硬件外设, 在骗取操作系统交互权限后自动输入攻击代码, 造成危害后果。[1]

2016 年，David Kierznowski、Keith Mayes 等人在论文《BadUSB 2.0: Exploring

USB Man-In-The-Middle Attacks》[2]中提出了一种 USB 中间人攻击的概念，并在 GitHub 上传了论文的 PoC 代码（Proof Of Concept，指的是观点验证代码）。David Kierznowski 提出了一种通过每个成本约 100 美元的定制 USB 硬件设备实现 USB 低速设备的 USB 中间人攻击的概念。

目前 USB 攻击方式通常使用专用的可编程 USB 设备或者对 USB 设备固件进行逆向工程，篡改为恶意固件并重新烧录后实施攻击。从操作系统和安全软件角度来看，所有 USB 设备均是合法的，只有在检查 USB 设备物理连接的情况下才可以检查到。

1.3 课题研究内容

本文使用树莓派和嵌入式 Linux 设计并实现了一种 USB 中间人攻击设备。依托于 Linux 内核良好的软件工程设计完成硬件层面与代码层面的部分解耦，脱离传统且复杂的硬件固件攻击方式。

通过树莓派一端的 USB 接口连接 USB 设备，另一端 USB 接口的外围设备（peripheral）模式模拟 USB 设备，拦截在 USB 设备与主机设备之间，以实现记录、转发、篡改和重放 USB 数据，达到 USB 中间人攻击的效果。USB 中间人攻击设备示意如图 1-1 所示。

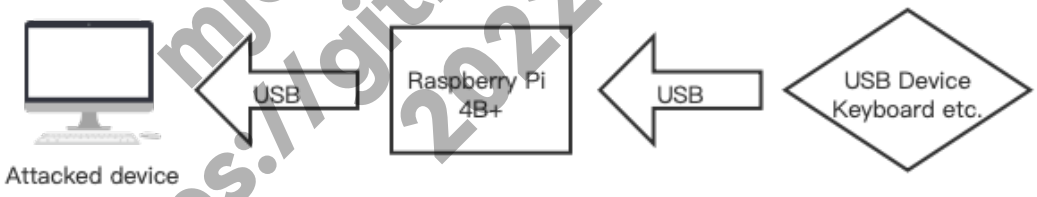

图1-1 **USB 中间人攻击设备示意图**

在解析和篡改 USB 数据时，对常见的 USB 数据协议进行简单分析，如键盘、鼠标等设备的 HID 协议。

使用.NET Core 和 C#语言作为用户空间应用的开发技术，实现应用核心功能并开放 RESETful API 用于前端操作界面调用。使用 Vue.js 编写设备的前端操作界面调用开放的 RESETful API，在网页中操作设备模拟、设备复制和中间人攻击功能。

通过 USB 中间人攻击设备的设计与实现，讨论当下大量应用 USB 协议的计算机生态环境下，USB 协议所面对的严峻信息安全挑战，以及 USB 中间人攻击技术在信息安全防御方面的应用。

2 相关技术介绍

通过上一章中对于研究现状和研究内容的描述，课题主要通过以下概念和技术的结合运用实现基于嵌入式 Linux 的 USB 中间人攻击。

2.1 中间人攻击

中间人攻击（Man-In-The-Middle Attacks）也被简写为 MITM，是一种在信息安全中被普遍应用的攻击手段。传统的中间人攻击主要被应用于各种网络层面的攻击，如通过一台不可信的中间计算机拦截在两台计算机之间以监听或篡改两台计算机的网络通信。在中间人攻击中，不可信的中间计算机对于通信的双方计算机是透明的，双方计算机无法感知到中间计算机，如图 2-1 所示。

中间人攻击技术同时也是一把“双刃剑”，被广泛应用于软件开发调试、网络安全设备等计算机领域。在网络安全设备中，如防火墙和流量监控设备就是中间人攻击技术的应用，通过在转发数据包给收件人的同时对数据包进行判断，做出丢弃、转发或者拒绝的响应。然而随着计算机技术的发展，中间人攻击技术也逐渐从传统的网络应用延展到多方面，如 WiFI、蓝牙、USB 等数据通信。

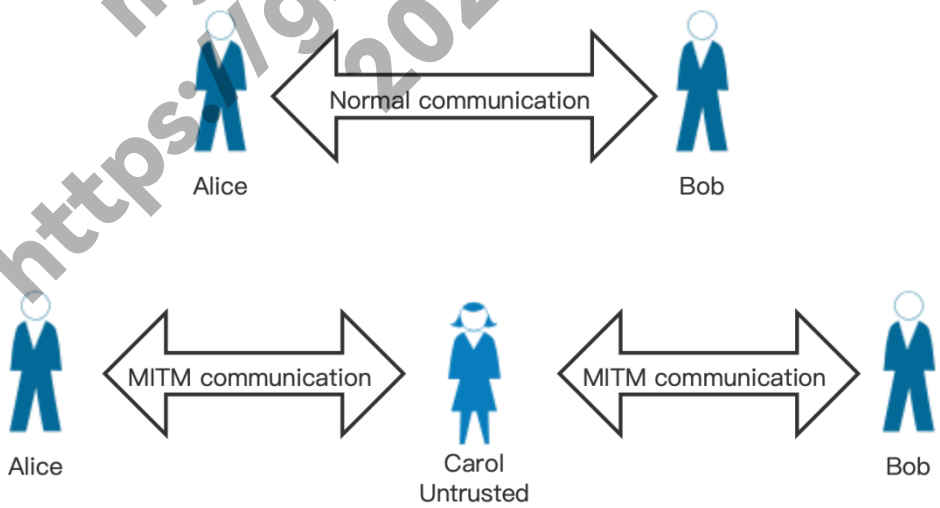

图2-1 正常通信与中间人攻击通信示例

2.2 Linux 设备树

操作系统在启动时需要知道当前设备的硬件和相关信息，对于传统的 x86 架构个人计算机可以通过 BIOS 枚举和 ACPI 表（指的是 Advanced Configuration and Power Management Interface，高级配置和电源管理接口）的方式向操作系统告知硬件相关信息。

而 Linux 设备树（指的是 Device Tree）在嵌入式设备中主要被用于向操作系统描述硬件信息，开发者需要通过编写 DTS 文件（指的是 Device Tree Source）进行定义设备的硬件信息。DTS 文件会被计算机使用 DTC 编译器（指的是 Device Tree Compiler）编译为 DTB 文件（指的是 Device Tree Blob），而 DTB 文件在硬件启动时会由 Bootloader 载入到内存中，Linux 内核会在启动过程中对内存中的设备树进行解析。

2.3 USB 设备框架

USB 2.0 标准由 Compaq、HP 和英特尔等公司组成的 USB 标准化组织（指的是 USB Implementers Forum）于 2000 年定义，其编写的《Universal Serial Bus Specification Revision 2.0》[3]对 USB 2.0 标准进行了详细的描述。

USB 2.0 标准中第九章“USB Device Framework”（指的是 USB 设备框架）对 USB 设备端的中间层进行了介绍，详细阐述了 USB 请求与各类描述符的数据结构及含义。在 USB 2.0 标准中，USB 标准化组织共制定了 8 种描述符，而其中相对重要的是设备、配置、字符串、接口和端点 5 种描述符。

2.4 USB HID 协议

HID 协议（指的是 Human Interface Devices）是在 USB 协议之上运行的一种人机设备协议，如 USB 键盘、USB 手柄等交互设备。HID 协议具备双向通信能力，部分传感器和智能卡设备也使用该协议进行通信。现代操作系统中均默认安装有 HID 设备驱动，因此使用 HID 协议的设备可以免安装驱动程序使用，具备较强的兼容性。

2.5 USB Gadget 子系统

USB Gadget 子系统是 Linux 内核提供的一系列内核模式的 API，用于外围

设备和其他嵌入式 Linux 的 USB 设备中使用。通过 USB Gadget API 可以在 Linux 内核中编写模块实现 USB 设备的模拟功能，Linux 内核源代码中提供了许多 USB 设备的实现。通过这种方式的 USB 设备模拟实现了逻辑代码与硬件的解耦，硬件有关的代码由 USB Gadget 子系统中负责 USB 控制器的驱动代码实现。

2.6 USB Raw Gadget 接口

USB Raw Gadget 是使用 USB Gadget 子系统实现的一个模块，提供了一系列 API 让用户空间可以对 USB Gadget 进行直接访问和底层操作。用户空间使用系统 IOCTL 调用与 Raw Gadget 模块定义的虚拟设备进行通信，使开发者能够脱离内核空间独立在用户空间中模拟 USB 设备，而在用户空间编写代码的方式提高了编写时代码的容错性和可调试性。

2.7 .NET Core 介绍

.NET Core 是由微软公司于 2014 年一改以往的商业模式，向社区开源的.NET 平台，具有开源、跨平台、高性能的特点，完整支持 C#语言。其兼容多种架构（如 x86、ARM 等）和操作系统（如 Windows、Linux、macOS 等）的特点，在开源社区中引起了广泛的支持。2020 年，微软发布最新的稳定版本与闭源的.NET 4 平台宣布合并，并正式命名为.NET 5。

P/Invoke（指的是平台调用）是.NET 架构中用于托管代码访问非托管库中的结构、函数等代码的技术。.NET 架构中的托管代码需要直接访问系统调用和数据结构时，需要通过 P/Invoke 技术使用 C#语言定义函数原型和数据结构。然而系统调用使用 C 语言定义数据结构，使用时需要在 C#语言定义的类和结构体中使用 StructLayout 标签和 MarshalAs 标签定义相应数据结构在内存中存储的类型和形式。

.NET 架构下的 C#语言在其默认情况下不允许开发者使用指针等不安全代码，而在使用 P/Invoke 技术在访问内核空间非托管内存时需要使用到指针、定长数组甚至直接操作内存等不安全代码。

在.NET 架构和 C#语言中需要使用不安全代码时，需要在项目中开启 unsafe 选项，并在进行相关不安全代码段使用 unsafe 关键字，告知编译器和.NET 运行时允许该段不安全代码。

3 系统选型与设计

根据前文描述的研究现状、研究内容和相关技术，本章结合实际需求对实现 USB 中间人攻击所需的软硬件技术从选型和架构设计两方面进行详细分析和阐述。

3.1 硬件与操作系统

3.1.1 硬件选型

硬件选型使用树莓派 4B+，树莓派是一款 ARM 指令集芯片的微型计算机，主要用于计算机教育。树莓派 4B+的核心处理器为博通 BCM2711(四核 1.5 GHz,Cortex A72 架构)，其使用 LPDDR4 内存，供电方式为 5V/3A USB-C 或 GPIO 5V。其他配置方面，支持双频 Wi-Fi、蓝牙 5.0、2 个 Micro HDMI 2.0 接口、千兆网口、MIPI DSI 接口、MIPI CSI 相机接口、立体声耳机接口、2 个 USB 3.0、2 个 USB 2.0、40 Pin GPIO 扩展接口。[5]其中用于供电的 USB-C 接口支持外围设备（peripheral）模式，可以用于模拟 USB 设备。同时树莓派 4B+强大的外设接口与 CPU 内存性能也使其成为了目前的最佳选择。

树莓派支持多种供电方式，包括 USB-C 接口供电、GPIO 接口供电、POE 供电等，其中一般使用 USB-C 接口进行供电，如图 3-1 所示。本课题中需要使用 USB-C 接口作为外围设备（peripheral）模式模拟 USB 设备，为方便开发和调试选择了 GPIO 接口供电方式。

通常嵌入式设备需要增加硬件或者修改一个接口特性时经常需要修改对应的 DTS 文件中的设备树定义并重新编译。而树莓派提供了一种设备树覆写（dtoverlay）的机制，允许开发者通过简单的配置文件配置常见的设备树修改，开发者也可以自行编写修改部分的设备树文件进行替换。

树莓派预置了许多设备树覆写（dtoverlay）所需的文件，包括 RTC（实时时钟）、I2C 接口和 USB 外围设备（peripheral）模式。通过在树莓派启动参数添加设备树覆写（dtoverlay）的方式可以使其 USB-C 接口以外围设备（peripheral）模

式工作。

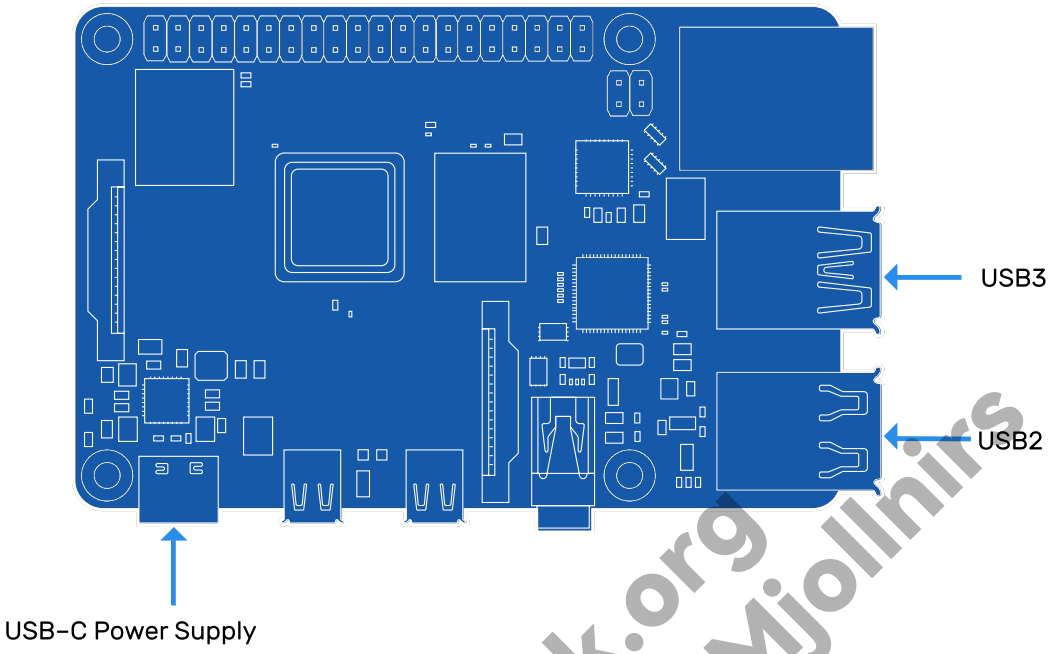

图3-1　**树莓派 4B+的 USB 接口**

3.1.2 操作系统选型

使用树莓派基金会配套的 Raspberry Pi OS，64 位版本作为操作系统。Raspberry Pi OS 是一款基于 Debian 的开源 Linux 系统，专门为树莓派硬件进行过优化，并且是使用树莓派所推荐的操作系统。Raspberry Pi OS 长期保持活跃开发中，重点提高 Debian 软件包在树莓派上的稳定性和性能。[6]

目前版本 Raspberry Pi OS 的 Linux 内核为 5.10 版本，其 USB Gadget 子系统能够支撑 USB Raw Gadget 模块的运行，但需要通过 Git 下载 USB Raw Gadget 内核模块源码编译并加载。

3.2 架构设计

3.2.1 攻击模型

在 USB 中间人攻击中，由树莓派作为中间人角色，负责转发、监听和篡改 USB 通信数据。树莓派的 USB-C 口与被攻击计算机通过 USB 数据线缆进行连接，而正常的 USB 设备与树莓派的 USB 2.0 或 USB 3.0 接口进行连接。树莓派通过在 USB-C 接口模拟连接在正常 USB 接口上的设备信息和行为，USB 设备和被攻击计算机互相都无法感知到中间人设备（指的是树莓派）的存在。

实际攻击者则可以通过多种方式操作中间人设备（树莓派），如 WiFi、网络、蓝牙等方式。通过这种形式的攻击往往具备很强的隐蔽性，而被攻击者往往需要通过检查 USB 物理连线情况才可以发现，如图 3-2 所示。

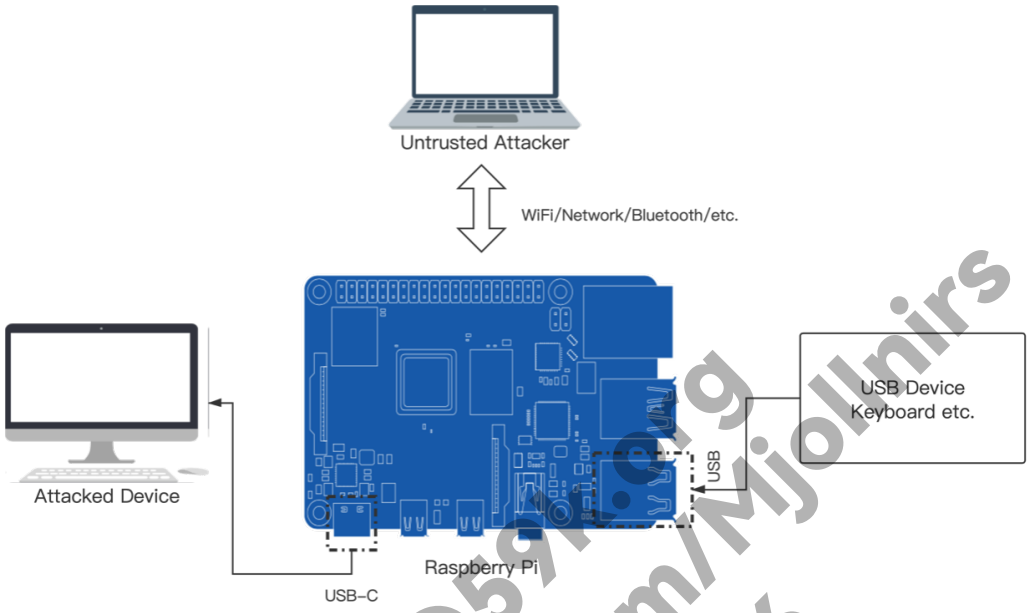

图3-2　**USB 中间人攻击模型**

3.2.2 整体架构

Linux 内核及其驱动和模块工作 Linux 内核空间，用户空间与内核空间无法直接进行数据交换。而 USB Raw Gadget 模块通过其定义的虚拟硬件文件（/dev/raw-gadget）与用户空间应用进行双向通信。

用户空间使用 P/Invoke 技术实现的 USB Raw Gadget 兼容层库负责调用虚拟硬件文件（/dev/raw-gadget）以实现与 USB Raw Gadget 模块的通信。USB Raw Gadget 兼容层库以.NET Standard 2.1 标准开发，负责主体逻辑的.NET 应用通过依赖方式调用 USB Raw Gadget 兼容层库。

而第三方 LibUSB 库则负责与连接在树莓派的 USB 设备进行通信，读取 USB 设备信息和数据。由于.NET 完善的生态链体系，有第三方开发的 LibUSB 兼容库，不需要自行使用 P/Invoke 技术编写兼容层。负责主体逻辑的.NET 应用通过 USB Raw Gadget 兼容层库转发到模拟的 USB 设备上，以实现中间人攻击效果。整体架构如图 3-3 所示。

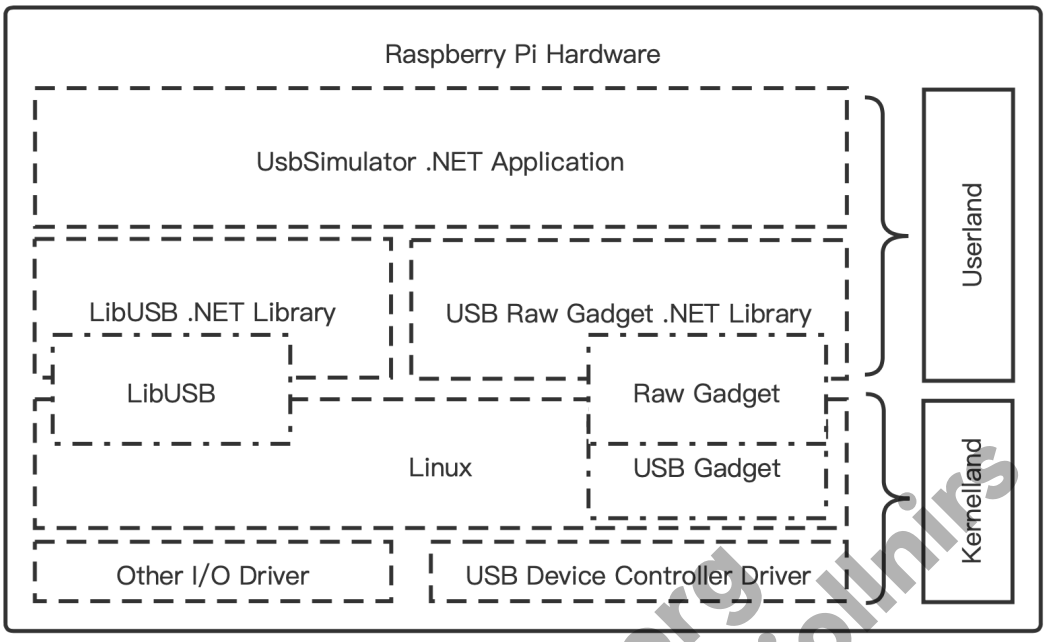

图3-3 **整体架构图**

3.2.3 应用架构

应用架构在逻辑上分为四层，各层次负责不同的主体任务，层次从下至上分别为非托管层（Unmanaged）、基础设施层（Infrastructure）、框架层（Framework）和应用层（Application）如图 3-4 所示。其中基础设施层、框架层和应用层在.NET 运行环境中运行。

(1) 非托管层（Unmanaged）

非托管层不属于.NET 运行环境的一部分，却是整个程序的最底层和最关键部分。非托管层主要由需要使用的第三方库和 Linux 内核 API 组成，包括 LibUSB、LibC 和 Raw Gadget 模块。其中 LibUSB 提供了一系列 USB 设备操作和通信的 API 接口；LibC 提供了一系列系统底层调用 API，如 open、close、ioctl 等 API 接口；Raw Gadget 模块则将 Linux 内核空间的 USB Gadget 子系统通过虚拟硬件接口形式开放给用户空间调用。

(2) 基础设施层（Infrastructure）

基础设施层由一些.NET 库构成，包括需要编写的 Raw Gadget 兼容层也属于这一层。基础设施层还包括 LibUSB 和 LibC 的兼容层，以及 LiteDB 数据库、NLog 日志库等第三方库构成。

(3) 框架层（Framework）

框架层是基础设施层与应用层之间的连接层，负责整个架构的具体实现，包括 ASP.NET Core 框架和 IoC 容器组成。ASP.NET Core 框架为应用层提供实现 RESETful API 的基本架构，是整个应用的核心框架部分。Castle Windsor 为应用层提供依赖注入（DI）和控制反转（IoC）容器，ASP.NET Core 框架中内置有依赖注入容器，Castle Windsor 容器可以与内置的依赖注入容器进行整合，提供更加强大的功能。

引入依赖注入（DI）和控制反转（IoC）的目的是使基础设施层各个库中的类与应用层中的具体实现解耦，降低各个组件之间的耦合度。

(4) 应用层（Application）

应用层是整体业务逻辑实现层，通过调用下层提供的类和接口实现 USB 设备模拟、USB 数据转发、USB 中间人攻击等具体业务逻辑。应用层不实现具体界面，仅提供 RESETful API 接口给独立的 Web 前端应用调用，也便于未来拓展其他类型的前端应用。

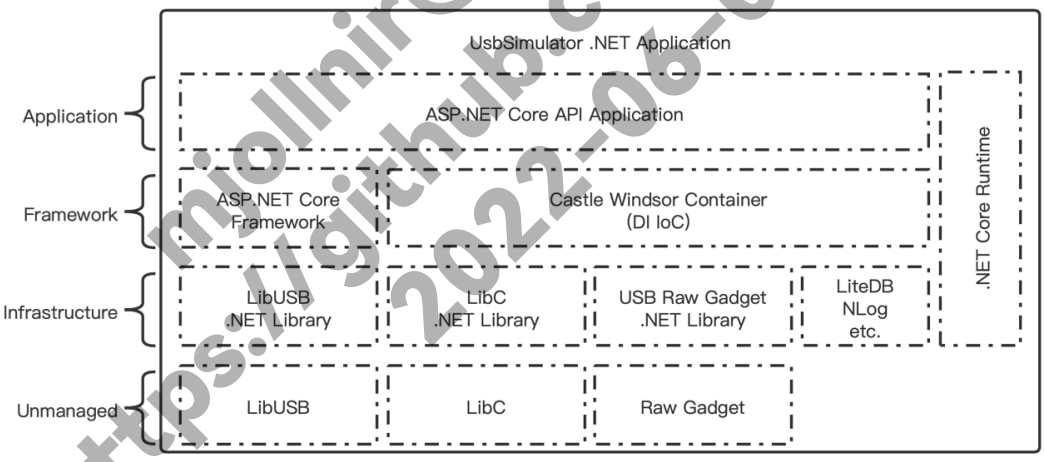

图3-4 用户空间应用架构图

4 主要功能实现

根据上一章阐述的硬件选型和架构设计，本章介绍 USB 中间人攻击主要功能的设计思路和实现方案，并通过对 USB 键盘实施 USB 中间人攻击的方式测试程序的核心功能。

4.1 系统环境

在树莓派官方网站下载最新 64 位版本的 Raspberry Pi OS 并写入树莓派 SD 卡后系统可以正常启动并使用，但还需要配置一些启动参数和加载内核模块，使树莓派和 USB-C 接口可以正常模拟 USB 设备。

4.1.1 启动参数

树莓派 4B+的 USB-C 接口默认情况工作在 Host 模式下，即用于连接其他 USB 设备，而在后续开发中需要使用外围设备（peripheral）模式。通过树莓派在其启动参数中提供的一种设备树覆写（dtoverlay）方式，修改设备树的定义，使 USB-C 接口以外围设备（peripheral）模式工作。

设备树覆写功能需要通过修改 boot 分区中的 config.txt 文件，通过 SSH 连接到树莓派并编辑/boot/config.txt 文件。

```
1.  pi@raspberrypi:~ $ sudo vi /boot/config.txt
```

在 config.txt 文件末尾添加设备树覆写（dtoverlay），并设置其模式为外围设备（peripheral）模式，如图 4-1 所示。

```
# Additional overlays and parameters are documented /boot/overlays/README

# Enable audio (loads snd_bcm2835)
dtparam=audio=on

# Enable DRM VC4 V3D drive
dtoverlay=vc4-kms-v3d
max_framebuffers=2
arm_64bit=1
gpio=26=pu,dh,op
dtoverlay=i2c3,pins_4_5,baudrate=400000
dtoverlay=dwc2,dr_mode=peripheral
"/boot/config.txt" 65L, 1808C                                    58,0-1
```

图4-1　**添加设备树覆写（dtoverlay）**

修改完成后需要重启并通过命令查看内核启动记录，确认已经在外围设备模式工作，同时通过内核日志可以看到 USB 控制器的可用端点数（EPs）为 8 个，如图 4-2 所示。

```
pi@raspberrypi:~ $ dmesg |grep dwc2
[    5.468443] dwc2 fe980000.usb: supply vusb_d not found, using dummy regulator
[    5.468836] dwc2 fe980000.usb: supply vusb_a not found, using dummy regulator
[    5.571807] dwc2 fe980000.usb: EPs: 8, dedicated fifos, 4080 entries in SPRAM
pi@raspberrypi:~ $
```

图4-2　**确认外围设备模式已启用**

4.1.2 内核模块编译

USB Raw Gadget 内核模块在 5.7 版本的 Linux 源码中已经被合并进入主线分支，但 Raspberry Pi OS 的内核分支中还未合并，需要通过 Git 下载内核模块代码并编译载入。USB Raw Gadget 内核模块的兼容性较强，兼容 4.14 以上版本 Linux 内核。

通过 Git 命令从 Github 下载 USB Raw Gadget 内核模块代码，执行 update.sh 脚本自动从 Linux 内核主分支下载最新 USB Raw Gadget 代码，如图 4-3、图 4-4 所示。

```
pi@raspberrypi:~ $ git clone https://github.com/xairy/raw-gadget.git
正克隆到 'raw-gadget'...
remote: Enumerating objects: 106, done.
remote: Counting objects: 100% (42/42), done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 106 (delta 15), reused 24 (delta 8), pack-reused 64
接收对象中: 100% (106/106), 68.84 KiB | 1.15 MiB/s, 完成.
处理 delta 中: 100% (37/37), 完成.
pi@raspberrypi:~ $
```

图4-3　**Git 下载 USB Raw Gadget 内核模块代码**

```
pi@raspberrypi:~/raw-gadget/raw_gadget $ ./update.sh
+ REPO=https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain
+ wget https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/drivers/usb/gadget/legacy/raw_gadget.c -O raw_gadget.c
--2021-11-02 10:28:16--  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/drivers/usb/gadget/legacy/raw_gadget.c
正在解析主机 git.kernel.org (git.kernel.org)... 145.40.73.55, 2604:1380:40e1:4800::1
正在连接 git.kernel.org (git.kernel.org)|145.40.73.55|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度： 31188 (30K) [text/plain]
正在保存至："raw_gadget.c"
```

图4-4　**从 Linux 内核主分支下载最新代码**

执行 make 命令编译内核模块，编译后得到文件名为 raw_gadget.ko 的内核模块，如图 4-5 所示。

```
pi@raspberrypi:~/raw-gadget/raw_gadget $ make
make -C /lib/modules/5.10.52-v8+/build M=/home/pi/raw-gadget/raw_gadget SUBDIRS=/home/pi/
aw-gadget/raw_gadget modules
make[1]: 进入目录“/usr/src/linux-headers-5.10.52-v8+”
  CC [M]  /home/pi/raw-gadget/raw_gadget/raw_gadget.o
  MODPOST /home/pi/raw-gadget/raw_gadget/Module.symvers
  CC [M]  /home/pi/raw-gadget/raw_gadget/raw_gadget.mod.o
  LD [M]  /home/pi/raw-gadget/raw_gadget/raw_gadget.ko
make[1]: 离开目录“/usr/src/linux-headers-5.10.52-v8+”
pi@raspberrypi:~/raw-gadget/raw_gadget $ ls -lah
总用量 244K
drwxr-xr-x 2 pi pi 4.0K 11月  2 10:28 .
drwxr-xr-x 7 pi pi 4.0K 11月  2 10:27 ..
-rw-r--r-- 1 pi pi  372 11月  2 10:27 include.patch
-rwxr-xr-x 1 pi pi   74 11月  2 10:27 insmod.sh
-rw-r--r-- 1 pi pi  148 11月  2 10:27 Makefile
-rw-r--r-- 1 pi pi   45 11月  2 10:28 modules.order
-rw-r--r-- 1 pi pi  179 11月  2 10:28 .modules.order.cmd
-rw-r--r-- 1 pi pi    0 11月  2 10:28 Module.symvers
-rw-r--r-- 1 pi pi  221 11月  2 10:28 .Module.symvers.cmd
-rw-r--r-- 1 pi pi  31K 11月  2 10:28 raw_gadget.c
-rw-r--r-- 1 pi pi 8.1K 11月  2 10:28 raw_gadget.h
-rw-r--r-- 1 pi pi  29K 11月  2 10:28 raw_gadget.ko
-rw-r--r-- 1 pi pi  264 11月  2 10:28 .raw_gadget.ko.cmd
-rw-r--r-- 1 pi pi   45 11月  2 10:28 raw_gadget.mod
-rw-r--r-- 1 pi pi 2.3K 11月  2 10:28 raw_gadget.mod.c
-rw-r--r-- 1 pi pi  163 11月  2 10:28 .raw_gadget.mod.cmd
-rw-r--r-- 1 pi pi 5.9K 11月  2 10:28 raw_gadget.mod.o
-rw-r--r-- 1 pi pi  36K 11月  2 10:28 .raw_gadget.mod.o.cmd
-rw-r--r-- 1 pi pi  24K 11月  2 10:28 raw_gadget.o
-rw-r--r-- 1 pi pi  44K 11月  2 10:28 .raw_gadget.o.cmd
-rw-r--r-- 1 pi pi  168 11月  2 10:27 README.md
-rwxr-xr-x 1 pi pi  258 11月  2 10:27 update.sh
pi@raspberrypi:~/raw-gadget/raw_gadget $
```

图4-5　**编译完成的 USB Raw Gadget 内核模块**

4.1.3 模块自动加载

自行编译的 Linux 内核模块需要每次使用 insmod 命令载入模块，并且无法使用 modprobe 命令载入模块。为了方便使用和调试，需要对系统进行配置，在系统启动时自动加载 raw_gadget.ko 内核模块。

Linux 的内核模块存放在/lib/modules 目录下，将 raw_gadget.ko 文件拷贝到该目录下。配置 modprobe 需要在/lib/modprobe.d 目录下创建一个新的 raw_gadget.conf 文件，用于描述加载 raw_gadget 模块需要执行的操作。

```
1. pi@raspberrypi:~/raw-gadget/raw_gadget $ sudo cp -f raw_gadget.ko /lib/modules/
2. pi@raspberrypi:~ $ sudo vi /lib/modprobe.d/raw_gadget.conf
```

在 raw_gadget.conf 文件中描述 modprobe 在加载 raw_gadget 模块时自动使用

insmod 载入 raw_gadget.ko 文件，如图 4-6 所示。

```
install raw_gadget insmod /lib/modules/raw_gadget.ko
~
~
"/lib/modprobe.d/raw_gadget.conf" 1L, 53C
```

图4-6　**配置 modprobe 加载 raw_gadget 模块**

此时已经可以通过 modprobe 命令载入模块，还需要修改/etc/modules-load.d/modules.conf 添加 raw_gadget 模块，使系统每次重启后可以自动载入 raw_gadget 模块，如图 4-7 所示。

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.

i2c-dev
raw_gadget
~
~
"/etc/modules-load.d/modules.conf" 7L, 214C                    7,1
```

图4-7　**配置自动加载 raw_gadget 模块**

重启设备后通过 lsmod 命令和确认是否存在/dev/raw-gadget 文件确认 raw_gadget 模块被系统正确加载，如图 4-8 所示。

```
pi@raspberrypi:~ $ lsmod |grep raw
raw_gadget             28672  0
pi@raspberrypi:~ $ ls /dev/raw-gadget
/dev/raw-gadget
pi@raspberrypi:~ $ ls -lah /dev/raw-gadget
crw------- 1 root root 10, 59 10月  28 12:17 /dev/raw-gadget
pi@raspberrypi:~ $
```

图4-8　**确认 raw_gadget 模块正确加载**

4.2　USB Raw Gadget 兼容层

在 USB Raw Gadget 兼容层中需要使用 P/Invoke（指的是平台调用）技术对 USB 框架和 USB Raw Gadget 模块中所定义的数据结构转换为可以在.NET 架构下可以运行的托管代码定义。使用 C#语言对数据结构正确定义之后，才可以在托管和非托管内存之间进行传输。

4.2.1 Raw Gadget 数据结构

Raw Gadget 的数据接口在其 raw_gadget.h 文件中定义，共涉及到 7 个结构体与若干个常量、枚举和函数等。由于 C#语言不支持 C 语言中宏定义形式，在 C#代码中将会把 C 语言的宏定义形式变更为函数形式。

(1) UsbRawInit 结构体

对应 raw_gadget.h 代码文件中的 usb_raw_init 结构体，用于初始化 USB Raw Gadget 模块，向内核空间传递需要使用的 USB 设备控制器（UDC）、控制器驱动和设备速度等信息。通过 StructLayout 标签设置结构体在内存中按照字段顺序布局，MarshalAs 标签设置字段非托管状态下为数组头指针，数组最大长度为常量 UDC_NAME_LENGTH_MAX。

```
1.  [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
2.  public unsafe struct UsbRawInit
3.  {
4.      [MarshalAs(UnmanagedType.ByValArray, SizeConst =
    RawGadgetConst.UDC_NAME_LENGTH_MAX, ArraySubType = UnmanagedType.U1)]
5.      public byte[] DriverName;
6.
7.      [MarshalAs(UnmanagedType.ByValArray, SizeConst =
    RawGadgetConst.UDC_NAME_LENGTH_MAX, ArraySubType = UnmanagedType.U1)]
8.      public byte[] DeviceName;
9.      [MarshalAs(UnmanagedType.U1)]
10.     public UsbDeviceSpeed Speed;
11. }
```

MarshalAs 中的 UnmanagedType 参数用于定义非托管内存中字段的类型，例如 U1 代表 1 个字节的无符号整数，UnmanagedType 常用类型如表 4-1 所示。

表4-1 UnmanagedType 常用类型

序号	名称	说明
1	ByValArray	定长/变长数组类型，需要指定数组元素类型
2	I1	1 字节长度的有符号整数类型
3	I2	2 字节长度的有符号整数类型
4	I4	4 字节长度的有符号整数类型
5	I8	8 字节长度的有符号整数类型
6	U1	1 字节长度的无符号整数类型

7	U2	2 字节长度的无符号整数类型
8	U4	4 字节长度的无符号整数类型
9	U8	8 字节长度的无符号整数类型
10	FunctionPtr	函数指针

(2) UsbRawEvent 结构体

对应 usb_raw_event 结构体，用于从内核空间读取 USB Raw Gadget 模块事件。结构体主要包括 2 个 32 位无符号整数，用于表达事件类型和事件信息的长度，其中事件类型 UsbRawEventType 单独定义为枚举类型。

```
1.  [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
2.  public unsafe struct UsbRawEvent
3.  {
4.      [MarshalAs(UnmanagedType.U4)]
5.      public UsbRawEventType EventType;
6.      [MarshalAs(UnmanagedType.U4)]
7.      public uint Length;
8.  }
9.  public enum UsbRawEventType : uint
10. {
11.     USB_RAW_EVENT_INVALID = 0,
12.     USB_RAW_EVENT_CONNECT = 1,
13.     USB_RAW_EVENT_CONTROL = 2,
14. }
```

(3) UsbRawEpIo 结构体

对应 usb_raw_ep_io 结构体，用于读取和写入端点数据。其结构由端点号，标记，长度和具体数据组成，其内存内存中的大小不定。受到 C#语言的限制，UsbRawEpIo 结构体需要根据具体传输的数据与其他结构体组合，以正常读取数据部分内容。

```
1.  public unsafe struct UsbRawEpIo
2.  {
3.      [MarshalAs(UnmanagedType.U2)]
4.      public ushort Ep;
5.      [MarshalAs(UnmanagedType.U2)]
6.      public ushort Flags;
7.      [MarshalAs(UnmanagedType.U4)]
8.      public uint Length;
```

```
9.  }
```

(4) UsbRawEpCaps 结构体

对应 usb_raw_ep_caps 结构体，用于描述端点功能信息。其 C 语言的结构体采用位域形式（低位至高位）描述端点功能，每一位表达一个布尔类型，如图 4-9 所示。

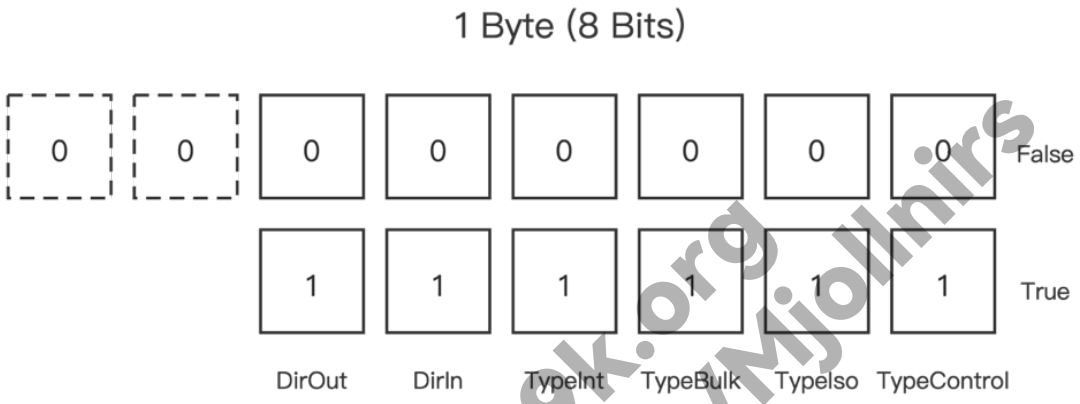

图4-9 **UsbRawEpCaps 存储形式**

C#中的枚举类型可以通过 Flags 标签实现相同功能，将结构体声明为枚举类型，其大小占用为 1 个字节。

```
1.  namespace UsbSimulator.RawGadget.LowLevel.RawGadget
2.  {
3.      [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
4.      public unsafe struct UsbRawEpCaps
5.      {
6.          public UsbRawEpCapsEnum Enum;
7.      }
8.
9.      [Flags]
10.     public enum UsbRawEpCapsEnum
11.     {
12.         None = 0,
13.         TypeControl = 1 << 0, TypeIso = 1 << 1, TypeBulk = 1 << 2, TypeInt = 1 << 3,
14.         DirIn = 1 << 4, DirOut = 1 << 5,
15.         All = TypeControl | TypeIso | TypeBulk | TypeInt | DirIn | DirOut,
16.     }
17. }
```

(5) UsbRawEpLimits 结构体

对应 usb_raw_ep_limits 结构体，用于描述端点的限制信息。结构体中描述端点的最大包尺寸、最大数据流数量和一个 32 位无符号整型作为保留字段。

```
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
public unsafe struct UsbRawEpLimits
{
    [MarshalAs(UnmanagedType.U2)]
    public ushort MaxpacketLimit;

    [MarshalAs(UnmanagedType.U2)]
    public ushort MaxStreams;

    [MarshalAs(UnmanagedType.U4)]
    public uint Reserved;
}
```

(6) UsbRawEpInfo 结构体

对应 usb_raw_ep_limits 结构体，用于表述端点信息。结构体中描述了端点名称，地址以及包含了 UsbRawEpCaps 结构体和 UsbRawEpLimits 结构体，完整的描述了端点的全部信息。

```
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
public unsafe struct UsbRawEpInfo
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst =
RawGadgetConst.USB_RAW_EP_NAME_MAX, ArraySubType = UnmanagedType.U1)]
    public byte[] Name;

    [MarshalAs(UnmanagedType.U4)]
    public uint Address;

    public UsbRawEpCaps Caps;
    public UsbRawEpLimits Limits;
}
```

(7) UsbRawEpsInfo 结构体

对应 usb_raw_ep_info 结构体，是对 UsbRawEpInfo 结构体的数组封装。结

构体中数组最大长度由 USB_RAW_EPS_NUM_MAX 常量限制，通常用于从内核空间中读取该 USB 控制器的全部可用端点。

```
1.  [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
2.  public unsafe struct UsbRawEpsInfo
3.  {
4.      [MarshalAs(UnmanagedType.ByValArray, SizeConst =
    RawGadgetConst.USB_RAW_EPS_NUM_MAX)]
5.      public UsbRawEpInfo[] Eps;
6.  }
```

4.2.2 USB 框架数据结构

USB Raw Gadget 模块中还需要使用 USB 框架中的的一些通用数据结构，主要包括各种描述符和标准请求结构体，如表 4-2 所示。这些通用数据结构使用 Raw Gadget 数据结构相同的方式进行编写。

表4-2　USB 框架数据结构清单

序号	结构体名称	说明
1	UsbDeviceDescriptor	USB 设备描述符结构体
2	UsbConfigDescriptor	USB 配置描述符结构体
3	UsbInterfaceDescriptor	USB 接口描述符结构体
4	UsbEndpointDescriptor	USB 端点描述符结构体
5	UsbStringDescriptor	USB 字符串描述符结构体
6	UsbCtrlRequest	USB 标准请求结构体

(1) USB 设备描述符

设备描述符用于描述 USB 设备的基本信息，每一个 USB 设备有且只有一个 USB 设备描述符，用于描述版本、类型、厂商、产品和配置数量等。

(2) USB 配置描述符

配置描述符用于特定配置信息，如接口数量、特性、耗电量等。每一个 USB 设备至少有一个 USB 配置描述符，配置描述符的数量由设备描述符定义。

(3) USB 接口描述符

接口描述符用于具体描述某个配置描述符中的接口信息，如类型、协议、端

点数量等。每一个配置描述符下至少有一个 USB 接口描述符。

(4) USB 端点描述符

端点是 USB 通信中的接受点和发送点，端点描述符用于描述接口描述符下的某个具体端点信息，如地址、方向、支持的功能等。每一个接口描述符可以无端点或者拥有多个端点。USB 设备通常使用编号为 0 的端点作为 USB 设备的控制管道，用于 USB 的基本控制信息传递。

(5) USB 字符串描述符

字符串描述符通过设备描述符中的厂商、序列号等信息的索引号读取，是一种可选描述符。

(6) USB 标准请求

每一个 USB 请求被封装为一个 Setup 包，设备端通过解析和处理 Setup 包后向主机响应正确的数据。Setup 包在内存中的大小为 8 个字节，其中包含 5 个字段，如表 4-3 所示。

表4-3 USB 请求 Setup 包数据结构

序号	字段名	占用空间	说明
1	bmRequestType	1 字节	类型，用于表达方向、类型、接收者等
2	bRequest	1 字节	命令字段，用于获取或设置描述符等
3	wValue	2 字节	根据请求标识传递不同参数
4	wIndex	2 字节	根据请求标识传递不同参数
5	wLength	2 字节	数据传输长度

4.2.3 USB 描述符之间关系

USB 2.0 中常用的 5 种描述符之间存在着互相引用的关系，如图 4-10 所示。设备描述符中定义的 bNumConfigurations 字段描述配置描述符总数量，将诱导主机根据需要向设备读取配置描述符。

设备描述符中的 iManufacturer、iProduct、iSerialNumber 字段则描述的是字符串描述符的索引编号，由主机根据需要向设备读取字符串描述符。当这些字段值为 0 时则不会尝试读取字符串描述符。

配置描述符由主机读取后，其 bNumInterfaces 字段描述接口描述符在当前配置下的总数量，配置描述符向主机传递时会一次性发送接口描述符、端点描述符

以及其他描述符。

而接口描述符被主机读取之后，其 bNumEndpoints 字段描述在当前接口描述符下的端点描述符数量，同时主机将根据需要解析端点描述符。

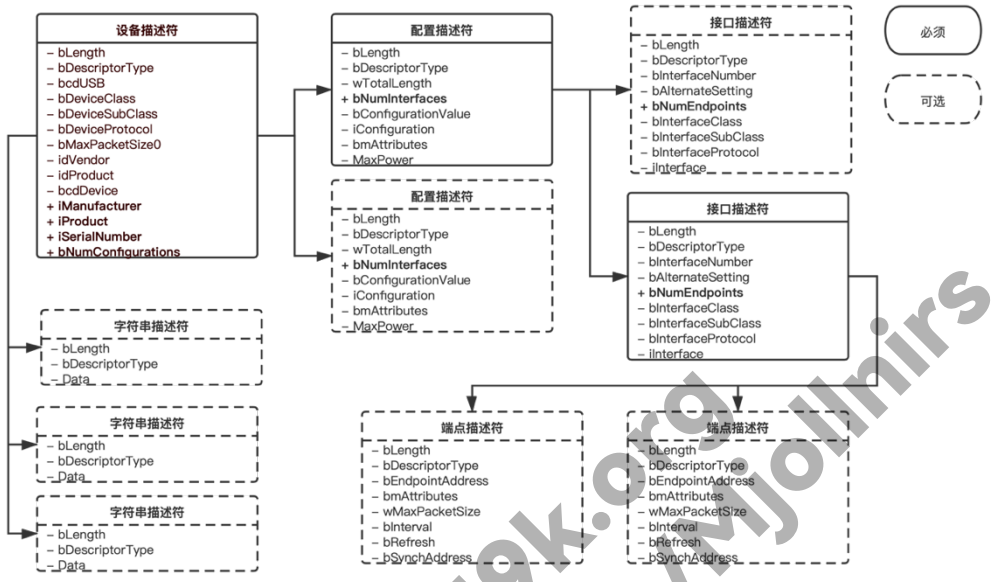

图4-10 **USB 描述符关系**

4.2.4 IOCTL 系统调用

用户空间应用需要使用 IOCTL 系统调用与 Raw Gadget 模块进行通信，而 IOCTL 分为函数、命令和数据结构组成，其中数据结构已经定义完成。IOCTL 函数需要使用 P/Invoke 技术定义外部函数原型。

```
1.  [DllImport(libc, SetLastError = true)]
2.  public static extern int ioctl(int fd, int request);
3.
4.  [DllImport(libc, SetLastError = true)]
5.  public static extern int ioctl(int fd, int request, int arg);
6.
7.  [DllImport(libc, SetLastError = true)]
8.  public static extern int ioctl(int fd, int request, void* arg);
```

IOCTL 命令为整数类型，在 C 语言代码中通过宏进行定义，C#语言中转换为函数和静态变量的形式。IOCTL 命令包括无数据（_IO）、只读(_IOR)、只写(_IOW)和读写(_IOWR)四种常用类型，通过不同的宏进行定义。其中只读、只写、读写三种类型的命令在定义时还与传输的数据结构大小有关。

```
1.  public static int _IO(int type, int nr) => _IOC(_IOC_NONE, (type), (nr), 0);
2.  public static int _IOR<T>(int type, int nr) => _IOC(_IOC_READ, (type), (nr), (_IOC_TYPECHECK<T>()));
3.  public static int _IOW<T>(int type, int nr) => _IOC(_IOC_WRITE, (type), (nr),
    (_IOC_TYPECHECK<T>()));
4.  public static int _IOWR<T>(int type, int nr) => _IOC(_IOC_READ | _IOC_WRITE, (type), (nr),
    (_IOC_TYPECHECK<T>()));
```

Raw Gadget 模块一共包含 16 个 IOCTL 命令完成全部功能，包括初始化、数据的传输等功能，如表 4-4 所示。

表4-4　Raw Gadget IOCTL 命令列表

序号	名称	说明
01	USB_RAW_IOCTL_INIT	初始化 Raw Gadget 实例
02	USB_RAW_IOCTL_RUN	绑定 UDC 并开始模拟
03	USB_RAW_IOCTL_EVENT_FETCH	拉取事件数据
04	USB_RAW_IOCTL_EP0_WRITE	端点 0 写入数据
05	USB_RAW_IOCTL_EP0_READ	端点 0 读取数据

续表 4-4

序号	名称	说明
06	USB_RAW_IOCTL_EP_ENABLE	启用指定的端点
07	USB_RAW_IOCTL_EP_DISABLE	禁用指定的端点
08	USB_RAW_IOCTL_EP_WRITE	指定端点写入数据
09	USB_RAW_IOCTL_EP_READ	指定端点读取数据
10	USB_RAW_IOCTL_CONFIGURE	设置 Raw Gadget 已配置
11	USB_RAW_IOCTL_VBUS_DRAW	设置 UDC VBUS 电源
12	USB_RAW_IOCTL_EPS_INFO	读取 UDC 可用端点
13	USB_RAW_IOCTL_EP0_STALL	挂起端点 0
14	USB_RAW_IOCTL_EP_SET_HALT	设置端点停止状态
15	USB_RAW_IOCTL_EP_CLEAR_HALT	清除端点停止状态
16	USB_RAW_IOCTL_EP_SET_WEDGE	设置端点停止（不可恢复）

在对 Raw Gadget 模块的 16 个 IOCTL 命令定义完成后，还需要编写单元测试检查定义的 IOCTL 命令是否正确，通过对比 C#定义的 IOCTL 命令计算结果与 C 语言宏的结果，确认定义的结果是否正确。

```
1.  public void RawGadgetIOCTLTest()
2.  {
3.      Dictionary<string, uint> consts = new Dictionary<string, uint>()
4.      {
5.          { "USB_RAW_IOCTL_INIT", 1090606336 },
6.          //省略其他值
```

```
7.          };
8.          foreach (var item in consts)
9.          {
10.             var member = typeof(RawGadgetConst).FindMembers(MemberTypes.Method, BindingFlags.Static
    | BindingFlags.Public, null, null);
11.             var method = member.Where(x => x.Name.EndsWith(item.Key)).FirstOrDefault() as MethodInfo;
12.             Assert.NotNull(method);
13.             var output = (int)method.Invoke(null, new object[] { });
14.             Assert.Equal(item.Value, BitConverter.GetBytes(output));
15.         }
16. }
```

4.2.5 Raw Gadget 初始化流程

用户空间通过虚拟硬件文件（/dev/raw-gadget）与内核空间进行通信是需要遵循一系列初始化流程，主要包括启动 Raw Gadget 模块和端点循环等步骤，而端点循环中端点 0 作为默认控制管道较为特殊。用户空间调用 Raw Gadget 模块进行初始化的主体流程如图 4-11 所示。

(1) 启动 Raw Gadget 模块

a) 通过 open 系统调用打开虚拟硬件文件（/dev/raw-gadget）

b) 通过 ioctl 系统调用发送 USB_RAW_IOCTL_INIT 命令和 UDC 信息，包括 UDC 名称、UDC 驱动。告知 Raw Gadget 模块使用哪一个 UDC，可以同时使用多个实例控制多个 UDC。

c) 通过 ioctl 系统调用发送 USB_RAW_IOCTL_RUN 命令，启动 Raw Gadget 模块。

(2) 端点 0 循环

在发送 USB_RAW_IOCTL_RUN 命令后即可进入端点 0 循环完成 USB 控制信息与设备信息的初始化和传递，USB 的各类描述符在该循环中向主机端发送。

a) 通过 ioctl 系统调用发送 USB_RAW_IOCTL_EVENT_FETCH 命令拉取事件信息，无事件则会一直等待。

b) 当事件类型为 USB_RAW_EVENT_CONNECT 时，则 c)；当事件类型不是 USB_RAW_EVENT_CONTROL 时，则跳出循环 a)；其它则 d)

c) 通过 ioctl 系统调用发送 USB_RAW_IOCTL_EPS_INFO 命令读取 UDC

的全部可用端点，并根据需要处理端点，如将端点分配给接口描述符等。

d) 对事件数据进行解析和处理，事件数据中的 Requst 字段为 USB 标准请求。根据 USB 标准请求的 bRequest 字段和 wValue 字段判断主机当前请求的描述符和数据返回响应描述符，例如当 bRequest 为 USB_REQ_GET_DESCRIPTOR，同时 wValue 为 USB_DT_DEVICE 则返回 USB 设备描述符。完成则回到 a）。

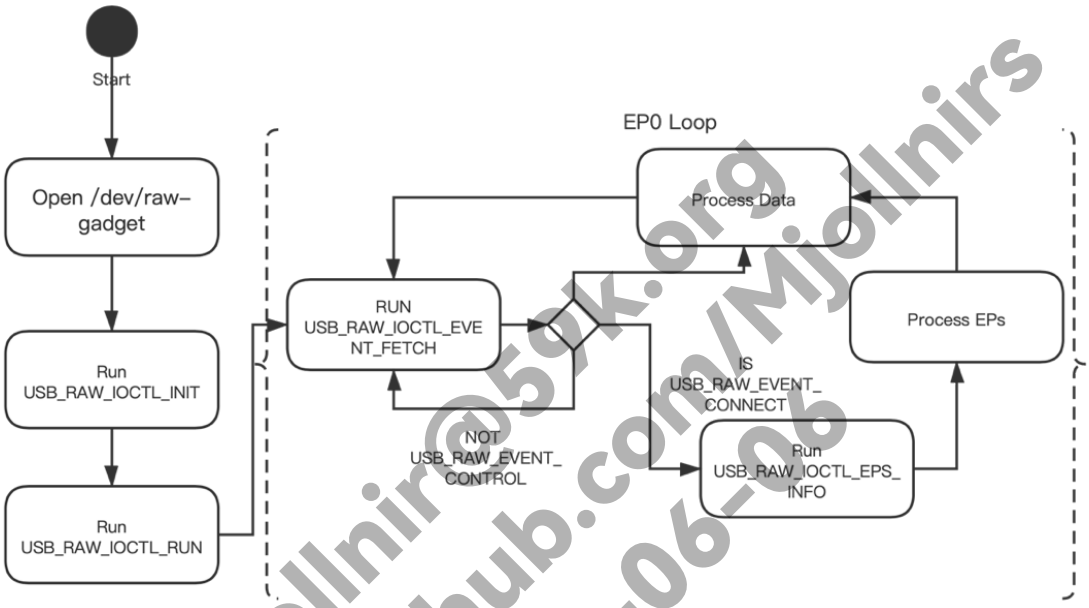

图4-11 **Raw Gadget 初始化流程**

4.2.6 Raw Gadget 兼容层测试

根据 Raw Gadget 初始化流程编写代码后对代码效果进行测试，由树莓派模拟一个名称为“UsbSimulator Product”的 USB 设备并在计算机中成功识别。在 macOS 下进行测试，并且使用 Wireshark 软件监听 USB 控制器数据包。Wireshark 中会自动将 USB 数据包进行编排解码，便于分析 USB 设备初始化过程中获取设备描述符和字符串描述符的过程。

```
1.  //设备信息
2.  public class UsbDeviceInfo
3.  {
4.      public int Vendor { get; set; } = 0x01;
5.      public string Manufacturer { get; set; } = "ZhouXin";
```

```
6.          public int Product { get; set; } = 0x01;
7.          public string ProductName { get; set; } = "UsbSimulator Product";
8.          public string SerialNumber { get; set; } = "UsbSimulator SerialNumber";
9.        //省略代码
10.   }
```

通过 dotnet run 命令启动应用程序，并将树莓派的 USB-C 接口连接至 macOS 计算机后，计算机将成功识别 USB 设备。计算机识别的 USB 设备信息按照预先代码中定义的产品名称、厂商、序列号、厂商 ID 与产品 ID 等内容显示，如图 4-12、图 4-13 所示。

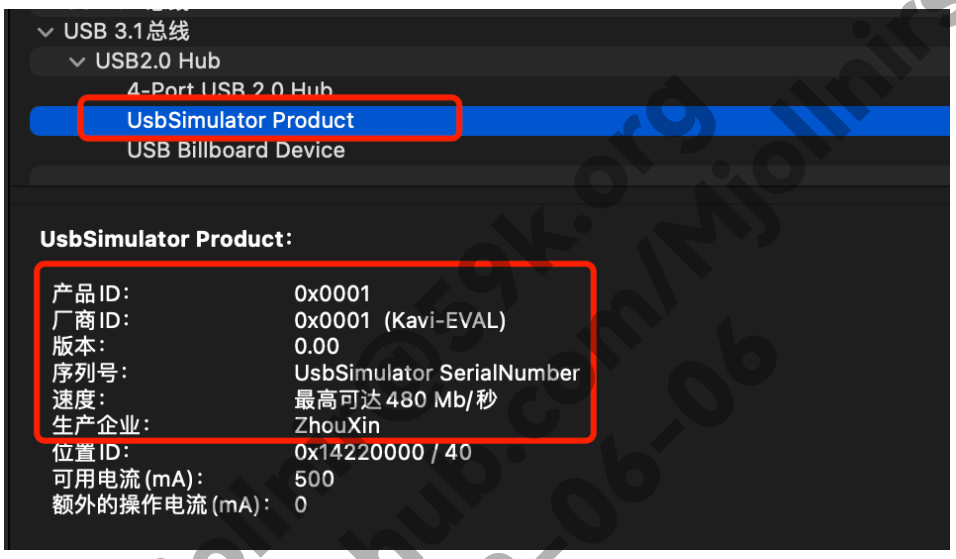

图4-12　成功识别模拟的 USB 设备

```
root@raspberrypi:/home/pi/SourceCode/UsbSimulator/UsbSimulator# dotnet run
/home/pi/SourceCode/UsbSimulator/UsbSimulator.RawGadget/UsbRawGadget.cs(214,25): warning CS0219: 变量“rv”已被赋值，但从未
使用过它的值 [/home/pi/SourceCode/UsbSimulator/UsbSimulator.RawGadget/UsbSimulator.RawGadget.csproj]
2021-11-02 10:39:25.4023 | Default | DEBUG | Start Application...
2021-11-02 10:39:25.6003 | Default.UsbRawGadget | DEBUG | Open Raw Gadget...
2021-11-02 10:39:25.6091 | Default.UsbRawGadget | DEBUG | Opened Raw Gadget: 84
2021-11-02 10:39:25.6356 | Default.UsbRawGadget | DEBUG | Init Raw Gadget...
2021-11-02 10:39:25.6499 | Default.UsbRawGadget | DEBUG | Ioctl Write Structure, Cmd: 1090606336, Structute: UsbRawInit, S
ize: 257
2021-11-02 10:39:25.6559 | Default.UsbRawGadget | DEBUG | Ioctl Start, Cmd: 1090606336, Ptr: 367211761072
2021-11-02 10:39:25.6597 | Default.UsbRawGadget | DEBUG | Ioctl End, Cmd: 1090606336, Ptr: 367211761072, Rv: 0
2021-11-02 10:39:25.6753 | Default.UsbRawGadget | DEBUG | Ioctl Read Structure, Cmd: 1090606336, Structute: UsbRawInit, Si
ze: 257
2021-11-02 10:39:25.6817 | Default.UsbRawGadget | DEBUG | Run Raw Gadget...
2021-11-02 10:39:25.6857 | Default.UsbRawGadget | DEBUG | Ioctl Start, Cmd: 21761, Data: 0
2021-11-02 10:39:25.6905 | Default.UsbRawGadget | DEBUG | Ioctl End, Cmd: 21761, Data: 0, Rv: 0
2021-11-02 10:39:25.7080 | Default.UsbRawGadget | DEBUG | Run Ep0 loop...
2021-11-02 10:39:25.7199 | Default.UsbRawGadget | DEBUG | Ioctl Write Structure, Cmd: -2146937598, Structute: UsbRawContro
lEvent, Size: 16
2021-11-02 10:39:25.7199 | Default.UsbRawGadget | DEBUG | Ioctl Start, Cmd: -2146937598, Ptr: 544655552752
2021-11-02 10:39:25.7199 | Default.UsbRawGadget | DEBUG | Ioctl End, Cmd: -2146937598, Ptr: 544655552752, Rv: 0
2021-11-02 10:39:25.7274 | Default.UsbRawGadget | DEBUG | Ioctl Read Structure, Cmd: -2146937598, Structute: UsbRawControl
Event, Size: 16
2021-11-02 10:39:25.7355 | Default.UsbRawGadget | DEBUG | Fetched USB Event USB_RAW_EVENT_CONNECT, Length: 0
2021-11-02 10:39:25.7531 | Default.UsbRawGadget | DEBUG | Ioctl Write Structure, Cmd: -2084547317, Structute: UsbRawEpsInf
o, Size: 960
2021-11-02 10:39:25.7531 | Default.UsbRawGadget | DEBUG | Ioctl Start, Cmd: -2084547317, Ptr: 544454228624
2021-11-02 10:39:25.7531 | Default.UsbRawGadget | DEBUG | Ioctl End, Cmd: -2084547317, Ptr: 544454228624, Rv: 14
2021-11-02 10:39:25.7600 | Default.UsbRawGadget | DEBUG | Ioctl Read Structure, Cmd: -2084547317, Structute: UsbRawEpsInfo
, Size: 960
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG | Endpoint #0
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Name: ep1in
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Addr: 1
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Type: TypeIso, TypeBulk, TypeInt, DirIn
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       MaxPacketLimit: 1024
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       MaxStreams: 0
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Reserved: 0
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG | Endpoint #1
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Name: ep1out
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Addr: 1
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       Type: TypeIso, TypeBulk, TypeInt, DirOut
2021-11-02 10:39:25.7627 | Default.UsbRawGadget | DEBUG |       MaxPacketLimit: 1024
```

图4-13 **启动.NET 程序**

通过 Wireshark 记录的 USB 控制器数据包可以看到，在设备建立连接且操作系统识别到该设备时，USB 控制器会通过端点 0 立即发送 GET_DESCRIPTOR 的 USB 请求，请求的类型为设备描述符（DEVICE），如图 4-14 所示。

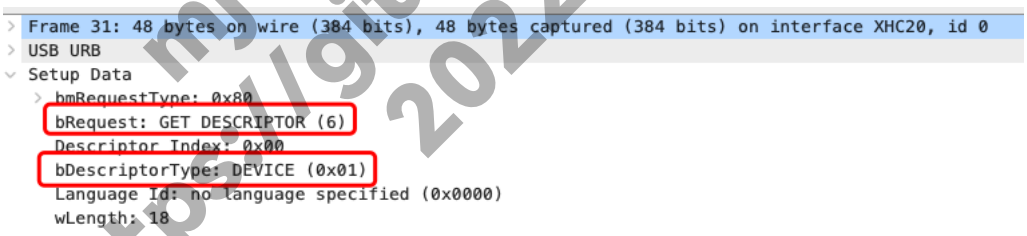

图4-14 **设备描述符请求包**

设备在收到请求设备描述符的数据后，会响应自身设备描述符，包含厂商、型号、字符串描述符的索引等信息，如图 4-15 所示。

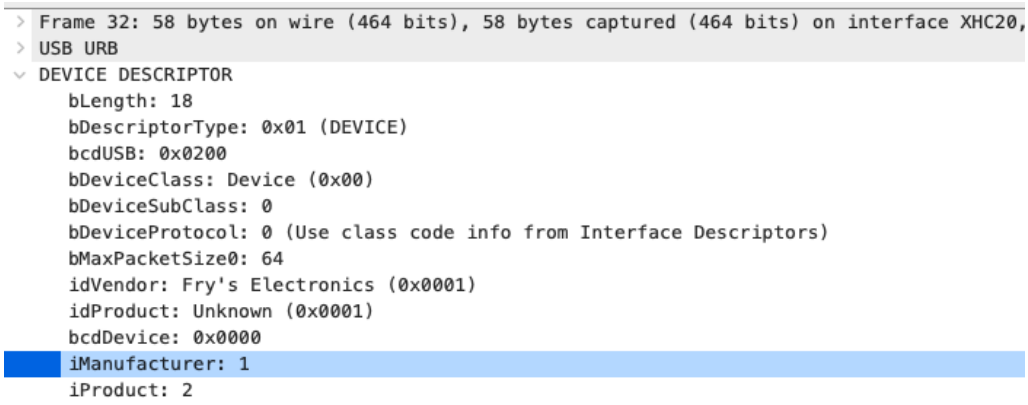

图4-15 **设备描述符响应包**

操作系统和 USB 控制器根据响应的设备描述符依次向设备再次发送 GET_DESCRIPTOR 的 USB 请求，此时请求的类型则分别为字符串描述符（STRING）和配置描述符（CONFIGURATION）。其中字符串描述符在请求时还会带有这次请求的字符串描述符索引，如图 4-16 所示。

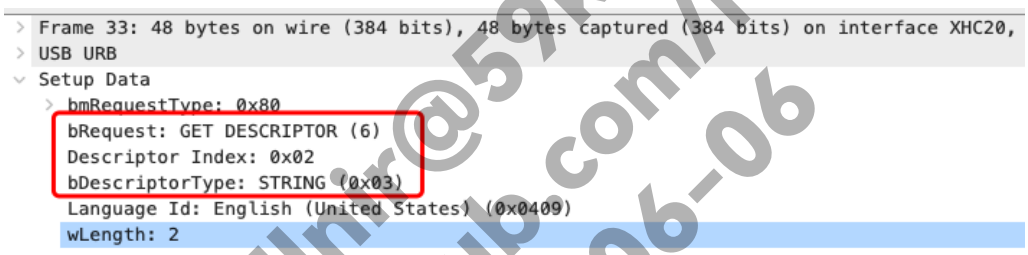

图4-16 **字符串描述符请求包**

设备则会根据请求的字符串描述符索引对数据进行编码后发送给 USB 控制器，而操作系统将会根据响应的数据在系统中显示相应的设备和信息，如图 4-17 所示。

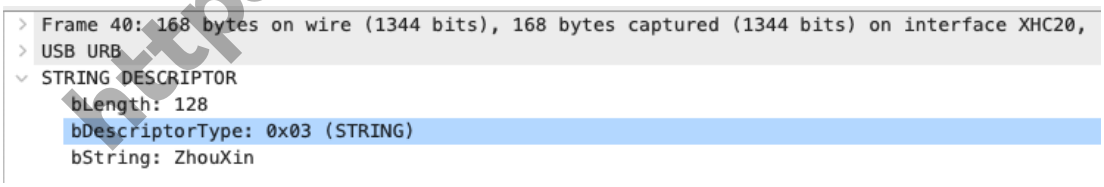

图4-17 **字符串描述符响应包**

4.3 USB 中间人攻击实现

通过对 USB Raw Gadget 兼容层的实现可以完成对 USB 设备的模拟功能，但要实现 USB 中间人攻击，还需要使用 LibUSB 库实现读取 USB 设备信息和

USB 数据转发功能。第三方.NET 封装的 LibUSB 库可以实现对具体端点 USB 数据的读取和写入，配合 Raw Gadget 兼容层模拟 USB 设备即可实现 USB 中间人攻击。

4.3.1 USB 信息复制

USB 信息复制的主要目的是让 Raw Gadget 兼容层所模拟的设备更加真实，这里的信息主要包括设备描述符中的厂商、产品等。部分厂商的 USB 驱动还使用版本来区分不同子产品线，需要确保复制的信息尽可能与原始信息匹配。通过 LibUSB 库读取全部 USB 设备，并记录设备的 VendorId 和 ProductId，如图 4-18 所示。

```
1.  UsbContext context = new UsbContext();
2.  foreach (IUsbDevice item in context.FindAll(x => true))
3.  {
4.      item.Open();
5.      Console.WriteLine("Device: {0}", item.Info.Device);
6.      Console.WriteLine("DeviceClass: {0}", item.Info.DeviceClass);
7.      Console.WriteLine("\tVendorId: 0x{0:x}", item.VendorId);
8.      Console.WriteLine("\tProductId: 0x{0:x}", item.ProductId);
9.      Console.WriteLine("\tProduct: {0}", item.Info.Product);
10.     Console.WriteLine("\tManufacturer: {0}", item.Info.Manufacturer);//省略部分代码
11. }
```

```
root@raspberrypi:/home/pi/SourceCode/UsbSimulator/UsbSimulator# dotnet run
Device: 1296
DeviceClass: 9
        VendorId: 0x1d6b
        ProductId: 0x3
        Product: xHCI Host Controller
        Manufacturer: Linux 5.10.52-v8+ xhci-hcd
        SerialNumber: 0000:01:00.0
Device: 256
DeviceClass: 0
        VendorId: 0x46a
        ProductId: 0x11
        Product:
        Manufacturer:
        SerialNumber:
Device: 1057
DeviceClass: 9
        VendorId: 0x2109
        ProductId: 0x3431
        Product: USB2.0 Hub
        Manufacturer:
        SerialNumber:
Device: 1296
DeviceClass: 9
        VendorId: 0x1d6b
        ProductId: 0x2
        Product: xHCI Host Controller
        Manufacturer: Linux 5.10.52-v8+ xhci-hcd
        SerialNumber: 0000:01:00.0
root@raspberrypi:/home/pi/SourceCode/UsbSimulator/UsbSimulator#
```

图4-18 **LibUSB 输出 USB 信息**

除了通过构造描述方式模拟设备外，也可以直接通过 LibUSB 的 ControlTransfer 函数直接转发端点 0 数据，在 Raw Gadget 兼容层接收到端点 0 数据时，通过 ControlTransfer 函数读取并转发真实设备响应的描述符，减少了构造描述符的过程。

```
1. byte[] buf = new byte[RawGadgetConst.EP0_MAX_DATA];//缓存区，保存设备响应数据
2. UsbRawHidIo io = new UsbRawHidIo(){
3.         Inner = new UsbRawEpIo() { Ep = (ushort)int_id, Length = (uint)data.Length, },
4.         Data = data.PadRight(8),
5. };
6. int uTransferLength = UsbDevice.ControlTransfer(setupPacket, buf, 0, buf.Length);//转发请求到真实 USB 设备
```

4.3.2 USB 数据转发

通过 LibUSB 的 UsbEndpointReader、UsbEndpointWriter 两个类配合 Raw Gadget 模块的 USB_RAW_IOCTL_EP_READ、USB_RAW_IOCTL_EP_WRITE 两个命令对其他端点数据进行转发（除端点 0），如图 4-19 所示。而对于中间人设备，模拟 USB 设备的读取操作对应于另一端真实 USB 设备则是写入操作。

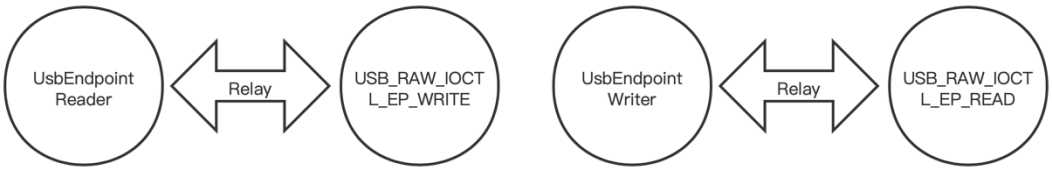

图4-19 **USB 数据转发示意图**

数据的记录、篡改、分析和重放功能均在这一步进行实现，每次转发数据时将数据分为响应和请求直接记录在数据库中。

```
1. var reader = UsbDevice.OpenEndpointReader(ReadEndpointID.Ep01); //端点 1 读取类
2. byte[] data = new byte[8]; //缓存区，缓存端点 1 数据，HID 协议一般 8 字节
3. reader.Read(data, 0, 8, 300, out tr); //读取端点 1 数据
4. //存储和分析代码
5. EpWrite<UsbRawHidIo>(io).Wait(); //Raw Gadget 写入端点 1 数据
```

4.3.3 HID 协议

HID 协议中增加了两种 USB 描述符，包括 HID 描述符和报告描述符。HID 描述符是较标准的 USB 描述符，按照 USB 描述符形式描述报告描述符长度、版本等信息。而对于 HID 报告描述符，USB 标准化组织提供了一款工具用于快速生成和编辑报告描述符，如图 4-20、图 4-21 所示。

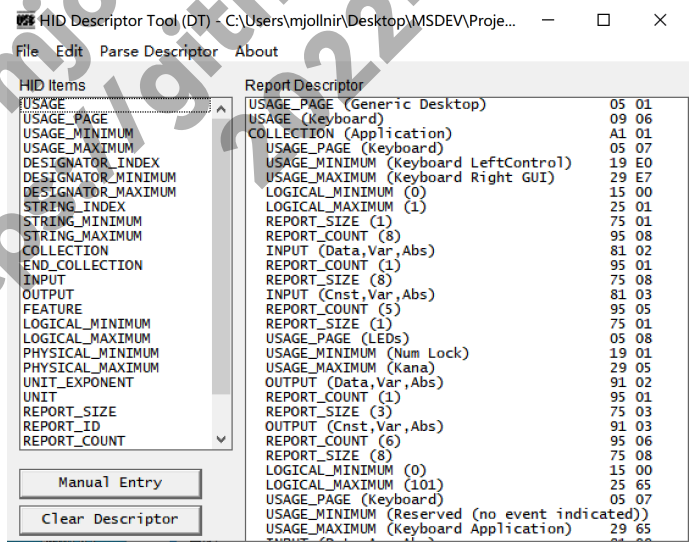

图4-20 **HID 报告描述符工具（键盘）**

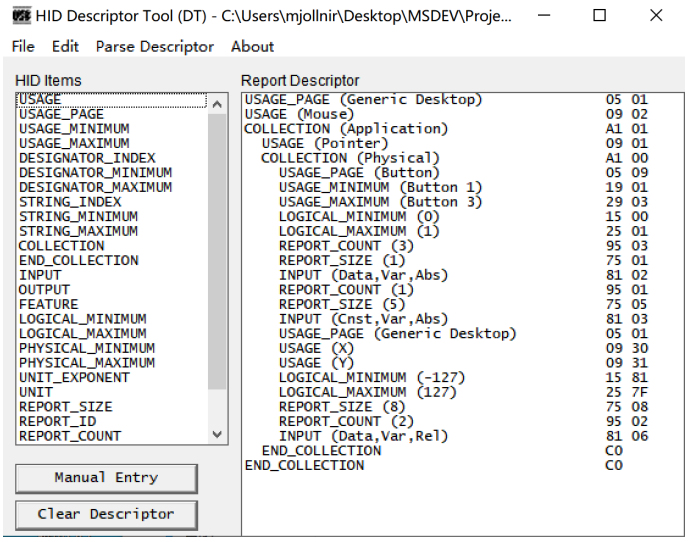

图4-21 **HID 报告描述符工具（鼠标）**

HID 报告描述符较为复杂，用于描述详细的设备报告报文信息，包括键盘、鼠标、手柄等功能信息。以大多数的 HID 键盘为例，键盘敲击信息和数据通常会通过端点 1 进行传输，而数据则会以报文形式传输，报文的内容和格式则在 HID 报告描述符中进行详细描述，因此 HID 报告描述符也被称为 USB 协议中最复杂的描述符。

在 HID 键盘中，向计算机系统输入一个字符需要发两次数据报文，分别为按下按键和松开按键。HID 报文中的按键具体值则需要遵循 USB 标准化组织的《HID Usage Tables FOR Universal Serial Bus (USB)》文档中“Keyboard/Keypad Page”部分。例如主机输出字符 x，则需要分别向发送“0x00, 0x1b, 0x00, 0x00, 0x00”（按下 x 键）和“0x00, 0x00, 0x00, 0x00, 0x00”（松开按键），如图 4-22 所示。

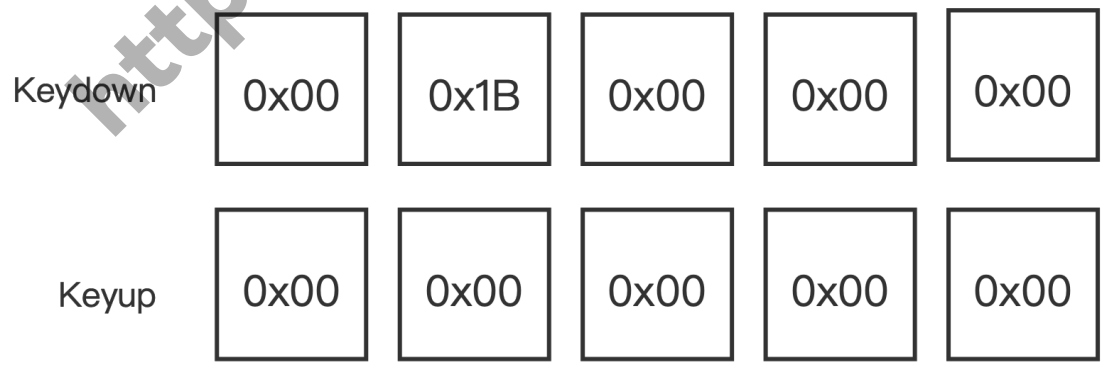

图4-22 **HID 报文示例**

4.3.4 操作 API（RESETful）

通过 ASP.NET Core 实现程序具体操作代码，通过 RESETful API 操作设备进行模拟设备、攻击设备选择、数据记录等功能，如表 4-5 所示。通过 Postman 和 Vue.js 对 RESETful API 接口进行测试。

RESETful API 标准通过 HTTP 协议中的请求类型进行 URL 的复用，包括 GET、POST、PUT、PATCH、DELETE 五种请求类型。通过这五种请求类型实现服务器数据的获取、新建、更新和删除操作。

表4-5　API 接口列表

序号	接口	说明
1	GET /Usb/Device	列出连接在设备上的 USB 设备
2	GET /Usb/Device/{id}	查看指定 id 的 USB 设备详细信息
3	POST /Usb/Device/{id}	选择指定 id 的 USB 设备信息模拟
4	GET /Gadget/Device	读取当前模拟的设备描述符信息
5	PUT /Gadget/Device	修改当前模拟的设备描述符信息
6	POST /Gadget/Usb/{id}	指定 id 的 USB 设备进行转发模拟
7	GET /Gadget/Run	开始设备模拟/转发
8	GET /Gadget/Monitor	读取当前数据记录状态
9	POST /Gadget/Monitor	修改当前数据记录状态
10	GET /Monitor	列出已有数据记录
11	GET /Monitor/{id}	显示/解析指定 id 的数据记录
12	GET /Monitor/{id}/Download	以二进制下载指定 id 的数据记录
13	DELETE /Monitor/{id}	删除指定 id 的数据记录
14	GET /Monitor/{id}/Relay	模拟并重放指定 id 的数据记录

4.4　USB 中间人攻击测试

对 USB 键盘进行中间人攻击测试，树莓派一端连接真实 USB 键盘，另一端连接计算机。通过 Raw Gadget 模块模拟连接的真实 USB 键盘设备信息，模拟前后的 USB 键盘设备信息如图 4-23 所示。

(a) 原始 USB 键盘信息

(b) 模拟 USB 键盘信息

图4-23 原始 USB 键盘信息

通过对比两次系统中显示的设备信息可以看到，原始设备没有序列号信息，而模拟的设备具有序列号信息。通过 Wireshark 分析模拟数据后，将设备描述符中的 iSerialNumber 字段修改为 0，使主机不读取序列号的字符串描述符，如图 4-24 所示。

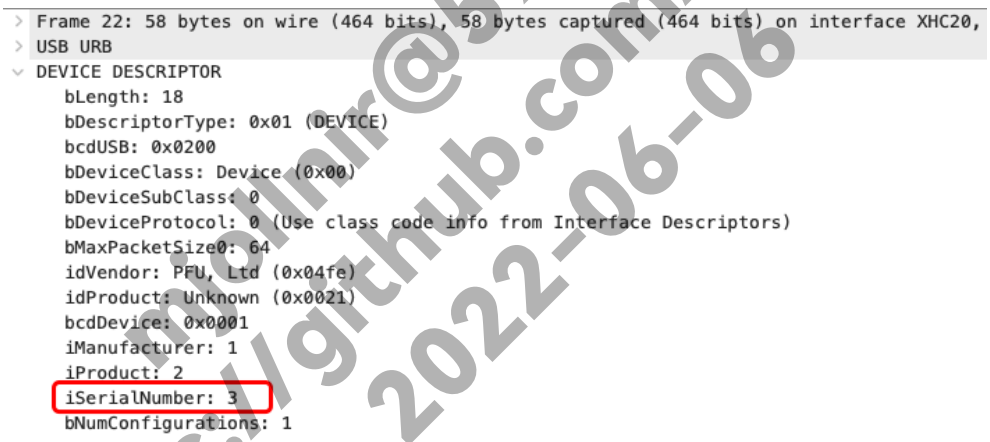

图4-24 模拟的设备描述符信息

使用 USB 键盘在依次按下“hello world”，计算机屏幕中正常输出“hello world”字符串。程序按照时间顺序转发数据至端点 1，同时在控制台中以 16 进制输出端点 1 所监听到的数据。

```
1.  Logger.InfoFormat("EP: {{{0}}}, Data: {1}", int_id, byteToHexStr(data));
```

使用 USB 键盘在依次按下“hello world”键后所输出的 16 进制数据，如图 4-25 所示。

```
root@raspberrypi:/home/pi/SourceCode/UsbSimulator/UsbSimulator# dotnet run
2021-11-02 10:52:59.4712 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:01.5128 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 2c 00 00 00 00 00
2021-11-02 10:53:01.6888 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:14.5387 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:18.0029 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 0b 00 00 00 00 00
2021-11-02 10:53:18.1782 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:20.2045 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 08 00 00 00 00 00
2021-11-02 10:53:20.3485 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:21.0506 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 0f 00 00 00 00 00
2021-11-02 10:53:21.1788 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:21.5576 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 0f 00 00 00 00 00
2021-11-02 10:53:21.6937 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:22.1967 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 12 00 00 00 00 00
2021-11-02 10:53:22.3088 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:22.7088 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 2c 00 00 00 00 00
2021-11-02 10:53:22.8770 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:23.2699 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 1a 00 00 00 00 00
2021-11-02 10:53:23.3900 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:23.8851 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 12 00 00 00 00 00
2021-11-02 10:53:24.0370 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:24.4530 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 15 00 00 00 00 00
2021-11-02 10:53:24.5891 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:25.0051 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 0f 00 00 00 00 00
2021-11-02 10:53:25.1412 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:25.6373 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 07 00 00 00 00 00
2021-11-02 10:53:25.7493 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:28.6567 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  01 00 00 00 00 00 00 00
2021-11-02 10:53:29.2737 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  01 00 06 00 00 00 00 00
2021-11-02 10:53:29.4178 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  01 00 00 00 00 00 00 00
2021-11-02 10:53:29.5378 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  00 00 00 00 00 00 00 00
2021-11-02 10:53:33.3590 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  01 00 00 00 00 00 00 00
2021-11-02 10:53:33.5504 | Default.UsbRawGadget |  INFO | EP: {1}, Data:  01 00 06 00 00 00 00 00
^Croot@raspberrypi:/home/pi/SourceCode/UsbSimulator/UsbSimulator#
```

图4-25 **端点 1 监听数据**

通过对程序输出的数据和 Wireshark 抓包的数据进行对比，确认抓取的数据为 USB 通讯中端点 1 的通讯数据。Wireshark 中也自动对 HID 数据进行了编排解码，更加直观的显示和分析 HID 数据所对应的键值信息，如图 4-26、图 4-27 所示。

```
> Frame 3458: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface XHC20,
> USB URB
v HID Data: 0000070000000000
    0... .... = Key: LeftControl (0xe0) = UP
    .0.. .... = Key: LeftShift (0xe1) = UP
    ..0. .... = Key: LeftAlt (0xe2) = UP
    ...0 .... = Key: LeftGUI (0xe3) = UP
    .... 0... = Key: RightControl (0xe4) = UP
    .... .0.. = Key: RightShift (0xe5) = UP
    .... ..0. = Key: RightAlt (0xe6) = UP
    .... ...0 = Key: RightGUI (0xe7) = UP
    Padding: 00
  v Keys: 070000000000
      0000 0111 = Key: d (0x07)
```

图4-26 **Wireshark 抓包的部分数据 1**

```
> Frame 3460: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface XHC20, i
> USB URB
v HID Data: 0000000000000000
    0... .... = Key: LeftControl (0xe0) = UP
    .0.. .... = Key: LeftShift (0xe1) = UP
    ..0. .... = Key: LeftAlt (0xe2) = UP
    ...0 .... = Key: LeftGUI (0xe3) = UP
    .... 0... = Key: RightControl (0xe4) = UP
    .... .0.. = Key: RightShift (0xe5) = UP
    .... ..0. = Key: RightAlt (0xe6) = UP
    .... ...0 = Key: RightGUI (0xe7) = UP
    Padding: 00
    Keys: 000000000000
```

图4-27 **Wireshark 抓包的部分数据 2**

对抓取的数据进行分析，数据中的数据值（键值）对应 USB 标准化组织的《HID Usage Tables FOR Universal Serial Bus (USB)》文档中“Keyboard/Keypad Page”部分，分析结果如表 4-6 所示。

表4-6　端点 1 数据分析（部分）

原始数据	有效数据	键值
00 00 0b 00 00 00 00 00	0b	Keyboard h and H
00 00 00 00 00 00 00 00	-	松开按键（后续省略）
00 00 08 00 00 00 00 00	08	Keyboard e and E
00 00 0f 00 00 00 00 00	0f	Keyboard l and L
00 00 0f 00 00 00 00 00	0f	Keyboard l and L
00 00 12 00 00 00 00 00	12	Keyboard o and O
00 00 2c 00 00 00 00 00	2c	Keyboard Spacebar（空格）
00 00 1a 00 00 00 00 00	1a	Keyboard w and W
00 00 12 00 00 00 00 00	12	Keyboard o and O
00 00 15 00 00 00 00 00	15	Keyboard r and R
00 00 0f 00 00 00 00 00	0f	Keyboard l and L
00 00 07 00 00 00 00 00	07	Keyboard d and D

通过对程序代码的测试和对 USB 端点中 USB 键盘的 HID 协议数据进行分析，完成基于嵌入式 Linux 的 USB 中间人攻击。而对于其他还未测试的 USB 设备类型（如 USB 存储设备等），仅仅在端点和端点所传输的数据协议有所不同，而攻击实现的整体思路是相同的。在 USB 中间人攻击的影响下， USB 传输的数据将可能会被攻击者所监听。

4.5　本章小结

本章介绍了 USB 中间人攻击设备中部分主要功能的实现，分析并实现了 Raw Gadget 兼容层、LibUSB 中间人数据转和 API 接口等主要功能，通过 Wireshark

对 USB 模拟设备连接初始化过程进行了简单分析。

在本章最后也通过对 USB 键盘实施中间人攻击的测试，基本证明了使用 Raw Gadget 技术实现 USB 中间人攻击的可实施性和严重危害性。

参考文献

[1] 秦玉海,李懿攀,斯嘉懿.BadUSB 攻击的实验与防范[J].中国刑警学院学报,2018(04):119-123.
[2] Kierznowski D,Mayes K.BadUSB 2.0: Exploring USB Man-In-The-Middle Attacks[D]. ISG, Royal Holloway, University of London, 2016.
[3] Compaq,Hewlett-Packard,Intel,Lucent,Microsoft,NEC,Philips. Universal Serial Bus Specification Revision 2.0[S],2000:239-274.
[4] Linux Kernel Organization,Inc.,David Brownell. USB Gadget API for Linux[EB/OL].[2021].https://www.kernel.org/doc/html/latest/driver-api/usb/gadget.html.
[5] The Raspberry Pi Foundation. Raspberry Pi 4 Tech Specs[EB/OL]. [2021]. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.
[6] The Raspberry Pi Foundation. Raspberry Pi OS Documentation[EB/OL]. [2021]. https://www.raspberrypi.org/documentation/computers/os.html.
[7] 郭永帅,王胜和,满超.BadUSB 的分析与侦查防范[J].警察技术, 2020(02):68-71.
[8] USB Implementers Forum. Device Class Definition for Human Interface Devices (HID) Firmware Specification Version 1.11[S], 2001.
[9] USB Implementers Forum. HID Usage Tables FOR Universal Serial Bus (USB)[S].2021:82-89.
[10] Mathias Claussen,禾沐.树莓派 4：全新的背后，依然是优秀的表现吗？[J].单片机与嵌入式系统应用,2020,20(06):1-4.
[11] 康云川,代彦.恶意 USB 设备原理及防护措施研究[J].计算机技术与发展,2020,30(01):112-117.
[12] 徐绍峰,陈静轩,刘都鑫,蔡希昌,鲍志平,陈钰婷,刘通,于东池,乔子凌.基于树莓派 4B 及 Jetson Nano 的多路音频采集传输系统[J].工业技术创新,2020,07(06):40-44.
[13] 姜建国,常子敬,吕志强,董晶晶.USB HID 攻击与防护技术综述[J].信息安全研究,2017,3(02):129-138.
[14] Microsoft. Native interoperability[EB/OL].[2021]. https://docs.microsoft.com/en-us/dotnet/standard/native-interop/.
[15] 吕志强,薛亚楠,张宁,冯朝雯,金忠峰.WHID Defense:USB HID 攻击检测防护技术[J].信息安全学报,2021,6(02):110-128.