华中科技大学网络空间安全学院

2022 网络安全课程设计报告

——Linux 下状态检测防火墙的设计与实现

姓　　名_____

学　　号_____

班　　级_____

指导教师_____

2023 年 1 月 10 日

实验报告评分表

评分项目	分值	评分标准	得分
实验原理	15	15-13：统流程清晰，报文处理过程描述清楚； 12-10：系统流程比较清晰，报文处理过程描述比较清楚； 9-0：描述简单	
系统设计	25	25-21：系统结构设计描述详细、清楚、完整，前后关系清晰； 20-16：系统结构设计比较清楚，关键模块、流程等都进行了描述 15-0：系统结构设计描述比较简单或不完整	
详细设计	25	25-21：关键模块设计描述详细、清楚、完整，前后关系清晰； 20-16：关键模块设计比较清楚，关键模块、流程等都进行了描述 15-0：关键模块设计描述比较简单或不完整	
系统测试	15	15-13：任务完成，针对任务点的测试，对结果有分析 12-10：针对任务点的测试截图，没分析 9-0：测试很简单，没有覆盖任务点	
心得体会	10	10-8：体会真实具体 7-5：体会比较空洞 4-0：没有写什么体会	
格式规范	10	图、表的说明，行间距、缩进、目录等，一种不规范扣 1 分	
总　分			
评分人：			

目　录

一、 实验目的和要求

1.1 实验目的

- 结合理论课程学习，深入理解计算机网络安全的基本原理与协议，巩固计算机网络安全基本理论知识；
- 熟练掌握计算机网络编程方法，拓展应用能力；
- 加强对网络协议栈的理解；
- 提高分析、设计软件系统以及编写文档的能力；
- 培养团队合作能力。

1.2 实验要求

- 系统运行
  - ✧ 系统启动以后插入模块，防火墙以内核模块方式运行；
  - ✧ 应用程序读取配置，向内核写入规则，报文到达，按照规则进行处理。
- 界面
  - ✧ 采用图形或者命令行方式进行规则配置，界面友好。
- 功能要求
  - ✧ 能对 TCP、UDP、ICMP 协议的报文进行状态分析和过滤；
  - ✧ 每一条过滤规则至少包含：报文的源 IP（带掩码的网络地址）、目的 IP（带掩码的网络地址）、源端口、目的端口、协议、动作（禁止/允许），是否记录日志；
  - ✧ 过滤规则可以进行添加、删除、保存，配置的规则能立即生效；
  - ✧ 过滤日志可以查看；
  - ✧ 具有 NAT 功能，转换地址分按接口地址转换和指定地址转换（能实现源或者目的地址转换的任一种即可）；
  - ✧ 能查看所有连接状态信息。
- 测试
  - ✧ 测试系统是否符合设计要求；
  - ✧ 系统运行稳定；
  - ✧ 性能分析。

二、 实验原理

2.1 状态检测防火墙

状态检测防火墙采用了状态检测包过滤的技术，是传统包过滤上的功能扩展。其维护了记录所有通过防火墙的连接/虚拟连接及其相关信息的状态检测表。这些状态检测表一般使用非线性的数据结构，从而能够大幅提升系统的性能。另一方面，由于状态表是动态的，因而可以有选择地、动态地开通 1024 号以上的端口，使得安全性得到进一步提高。

2.2 Linux Netfilter 架构

Netfilter 是一个由 Linux 内核提供的框架，它允许以自定义处理程序的形式实现各种网络相关的操作。Netfilter 为数据包过滤、网络地址转换和端口转换提供了各种功能和操作，这些功能为引导数据包通过网络和禁止数据包到达网络中的敏感位置提供了必要的功能。[1]

Netfilter 代表了 Linux 内核中的一组钩子，允许特定的内核模块在内核的网络栈中注册回调函数。这些函数，通常以过滤和修改规则的形式应用于流量，对穿越网络堆栈内相应钩子的每个数据包都会被调用。

Netfilter 提供的 Hook 点如表 2-1 所示，在 Hook 函数完成了对数据包所需的任何的操作之后，它们必须返回表 2-2 中预定义的 Netfilter 返回值中的一个。

表 2-1 Netfilter 提供的 Hook 点及说明

Hook 点	调用的时机
NF_INET_PRE_ROUTING	刚刚进入网络层的数据包通过此点（刚刚进行完版本号、校验和等检测），目的地址转换在此点进行
NF_INET_LOCAL_IN	经路由查找后，送往本机的通过此检查点，INPUT 包过滤在此点进行
NF_INET_FORWARD	要转发的包经过此检测点，FORWARD 包过滤在此点进行
NF_INET_LOCAL_OUT	本机进程发出的包通过此检测点，OUTPUT 包过滤在此点进行
NF_INET_POST_ROUTING	所有马上便要通过网络设备出去的包通过此检测点，内置的源地址转换功能（包括地址伪装）在此点进行

表 2-2 Netfilter 提供的 Hook 函数返回值

返回值	含义

NF_DROP	丢弃该数据包
NF_ACCEPT	保留该数据包
NF_STOLEN	告知 Netfilter 忽略该数据包
NF_QUEUE	将该数据包插入到用户空间
NF_REPEAT	请求 Netfilter 再次调用该 Hook 函数

在不同的 Linux 内核版本中，Netfilter 相关 API 有一定的不同。以下实验所使用的 Linux 内核版本为较新的 5.15.0-50-generic，其 Netfilter API 的使用参考了 Linux 内核中其他已有模块的示例代码[2][3]。

2.3 Linux 编程

- 用户空间与内核空间交互

Linux 操作系统和驱动程序运行在内核空间（如报文过滤的内核模块），应用程序（如配置规则的程序）运行在用户空间，如图 2-1 所示，两者不能简单地使用指针传递数据，需要使用特定的信息交互方式进行交互，包括编写自己的系统调用、编写设备驱动程序、使用 proc 文件系统、使用 Netlink、使用内存映像等。以下实现中使用 Netlink 完成用户空间与内核空间交互。

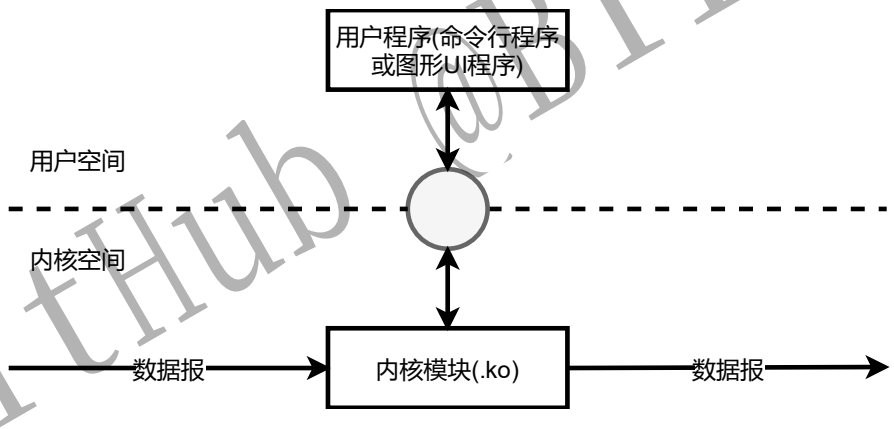

图 2-1 用户空间与内核空间交互

- Linux 内核相关编程基础及编程规范

该项目的实现使用到了 Linux 内核中的多种常用内置数据结构和功能，如 hlist，hashtable，rwlock，mutex，timer 等，将在后续设计实现部分详细展开。

为了与内核中其他模块及内核整体的编程规范保持一致，代码遵循 Linux 内核代码风格[4]。

三、 实验环境和采用的工具

3.1 实验环境

- 开发环境：Windows 11 (physical machine)
- 运行环境：Ubuntu 22.04.1 LTS, Linux 5.15.0-50-generic (virtual machine in VMware 16.2.1)

3.2 开发工具

- Visual Studio Code 1.72.2

3.3 开源项目

- cJSON[5]
- Yarn[6] 1.22.17
- Vue[7] 2.6.14 & Vue/cli[8] 4.5.13
- Element-ui[9] 2.5.16

四、 系统设计

4.1 系统结构设计

系统的整体结构如图 4-1 所示，总体分为内核空间的内核模块和用户空间的交互程序，后者又分为使用 C 语言实现的命令行界面 xwall_app.c 和使用 Python 和 VUE 语言实现的图形用户界面 xwall_app.py 及 xwall-gui-0.1.0.AppImage。

内核模块总体分为用户交互模块（Netlink Message Handler）、基本数据结构模块（Connection Table、Rule Table、Log Table、Manage Log Table、NAT Table 等）和数据包处理模块（Netfilter Filter & NAT Translator）。用户交互模块负责处理用户通过 Netlink 发送来的请求并做出相应响应，基本数据结构模块用于存储工作过程中需要的数据，数据包处理模块用于完成数据包的过滤和 NAT 转换等操作。

CLI 程序可以通过 Netlink 与内核模块进行通信，且提供了 TEXT 模式和 JSON 模式以提供用户友好的输出格式和 JSON 输出格式，分别用于不同场景。用户可以直接使用 CLI 与内核模块进行交互（TEXT 模式），也可以使用 GUI 程

序间接调用 CLI 与内核模块进行交互（JSON 模式）。

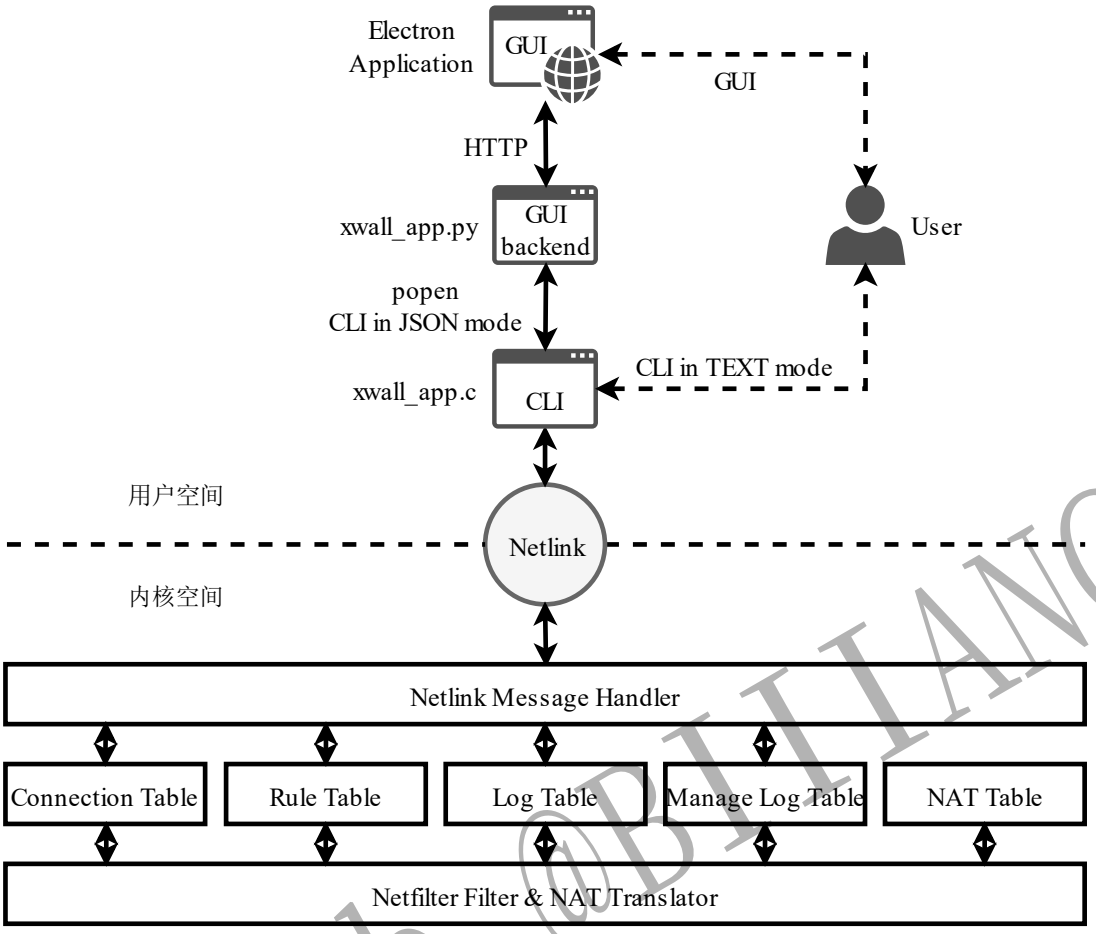

图 4-1 系统整体结构

4.2 核心数据流程

系统的核心数据分为两部分：一部分是与用户操作有关的控制数据和响应数据，这部分数据通过 Netlink 在用户空间和内核空间进行传递，并且分别由用户空间程序和内核模块中的用户交互模块（Netlink Meaasge Handler）进行处理，如图 4-1 所示；另一部分是被系统处理的数据包，这部分数据主要由图 4-1 中的数据包处理模块（Netfilter Filter & NAT Translator）进行处理，具体的处理流程如图 4-2 所示。

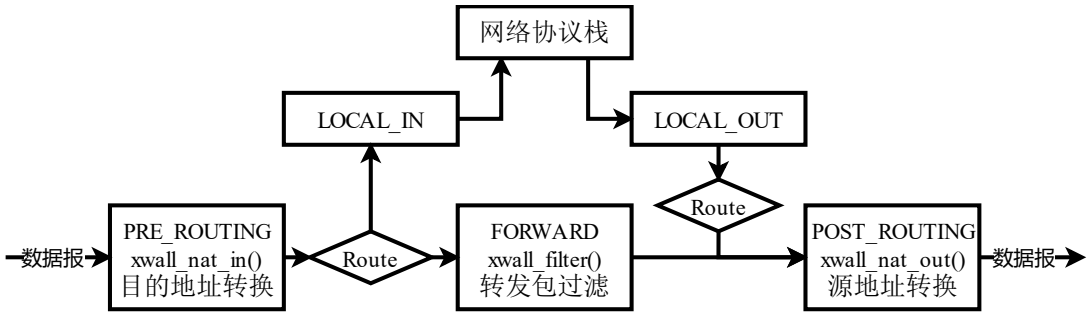

图 4-2 数据包的处理流程

由于系统设计中的 GUI 前后端通信使用的 HTTP 协议同样需要经过 Netfilter 架构，且开发时使用物理机通过 SSH 连接至虚拟机，此处为了便于开发调试，仅对 FORWARD 包进行了过滤，在 LOCAL_IN 和 LOCAL_OUT 两个过滤点未设置 Hook 函数进行过滤。在实际应用中，可以在 LOCAL_IN 和 LOCAL_OUT 处分别添加相应的 Hook 函数并管理相应的规则，也可以将 FORWARD 处的包过滤移动到 PRE_ROUTING 处经过目的地址转换后和 POST_ROUTING 处源地址转换前进行。前者可以将各种不同的数据包分开处理，后者可以在尽量少的位置进行处理。

对于 TCP 连接，要求建立连接的必须是满足 ACCEPT 规则或默认 ACCEPT 动作的 SYN 包，建立连接后双向的数据包都可以通过该连接通过防火墙，连接超时时间为 5 分钟，其状态机如图 4-3 所示。

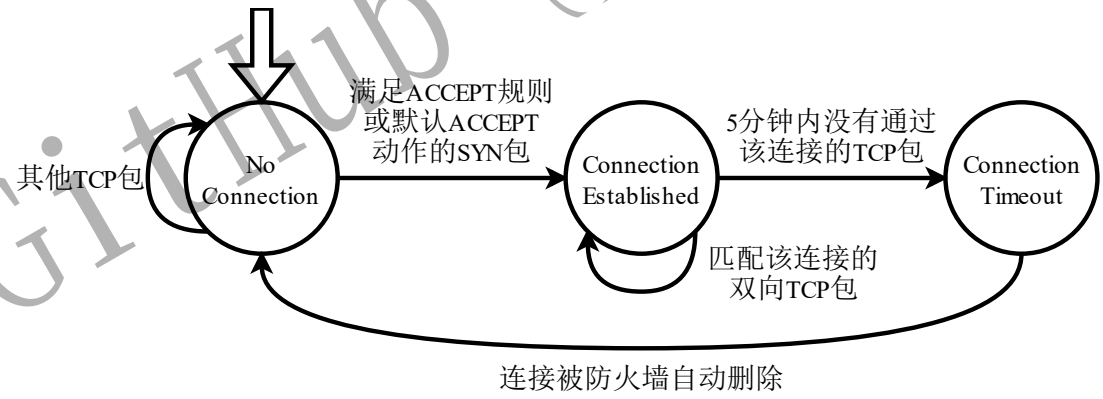

图 4-3 TCP 连接状态机

对于 UDP 虚拟连接，只要第一个数据包满足 ACCEPT 规则或默认 ACCEPT 动作即可建立虚拟连接，建立连接后双向的数据包都可以通过该虚拟连接通过防火墙，虚拟连接超时时间为 3 分钟，其状态机如图 4-4 所示。

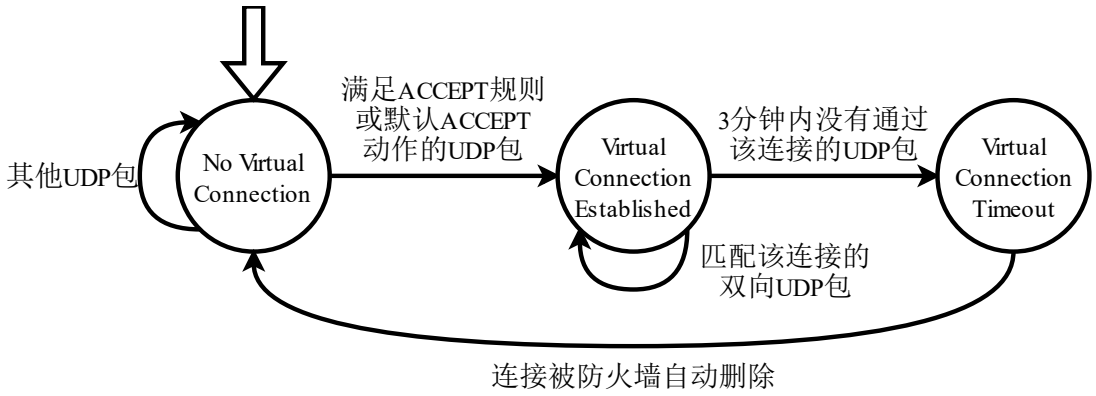

图 4-4 UDP 虚拟连接状态机

对于 ICMP 虚拟连接，要求建立虚拟连接的必须是满足 ACCEPT 规则或默认 ACCEPT 动作的 ECHO REQUEST 包，建立虚拟连接后与建立虚拟连接的包方向相同的 ECHO REQUEST 包和方向相反的 ECHO REPLY 包可以通过该虚拟连接通过防火墙，其他包无法通过该虚拟连接通过防火墙，虚拟连接超时时间为 3 分钟，其状态机如图 4-5 所示。

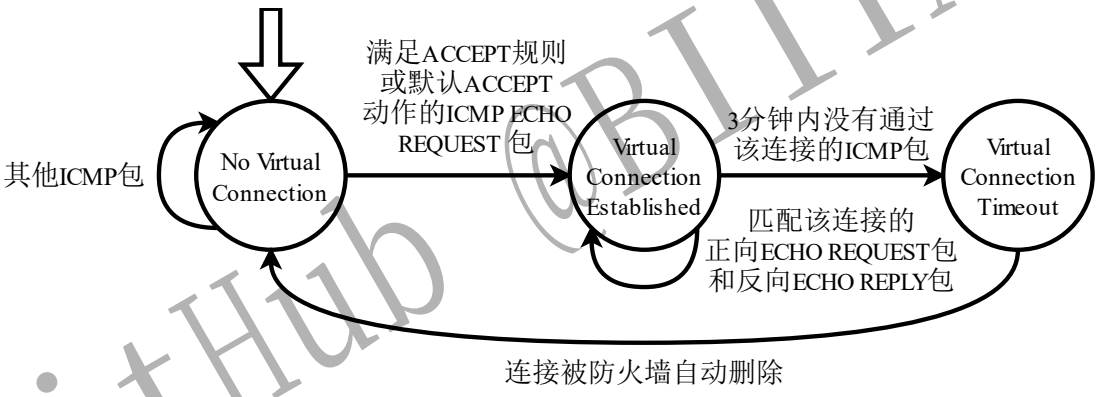

图 4-5 ICMP 虚拟连接状态机

五、 详细设计实现

5.1 关键模块流程

包过滤模块的流程图如图 5-1 所示。

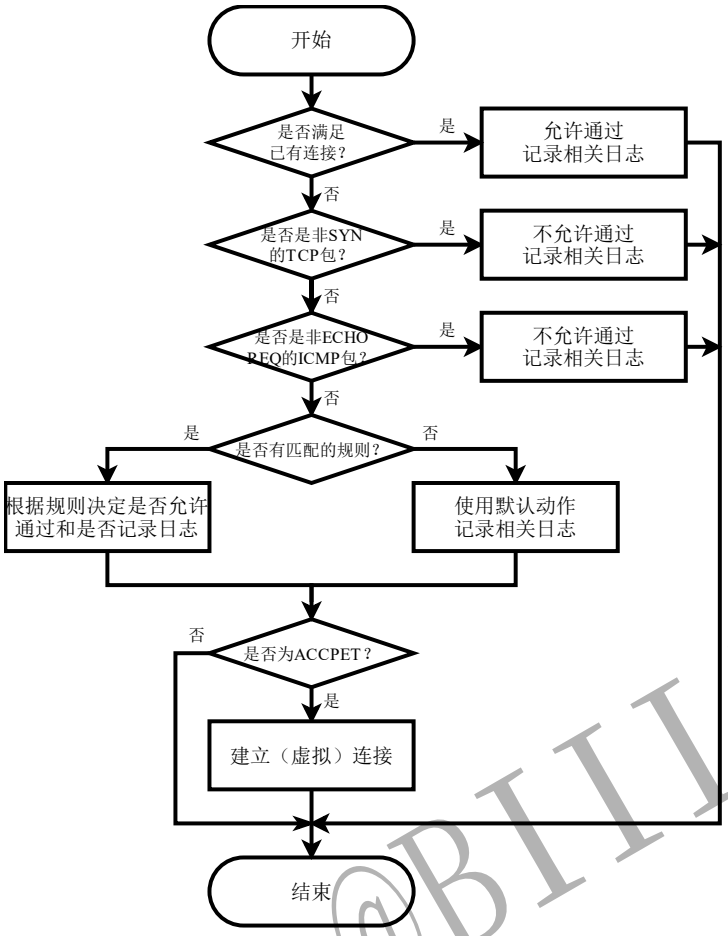

图 5-1 包过滤模块流程图

5.2 关键数据结构

基本数据结构定义源代码见 xwall.h 及 xwall_public.h。

- 连接：struct xwall_connection

表 5-1 连接结构

联合体	类型	字段名	含义
	__be32	saddr	建立该连接的源地址
	__be32	daddr	建立该连接的目的地址
	__u8	protocol	协议
union	__u8	icmp.type	ICMP 协议 TYPE
	__u8	icmp.code	ICMP 协议 CODE
	__be16	tcp.sport	TCP 协议源端口
	__be16	tcp.dport	TCP 协议目的端口
	__u8	tcp.state	TCP 协议状态
	__be16	udp.sport	UDP 协议源端口
	__be16	udp.dport	UDP 协议目的端口
	__be64	timeout	连接超时时间

	struct hlist_node	node	hlist 节点，用于构建链表

- 连接表：struct xwall_hashtable

表 5-2 连接表结构

联合体	类型	字段名	含义
	rwlock_t	lock	读写锁
	unsigned int	conn_num	当前连接数
	struct hlist_head []	hashtable	使用内核 hlist 实现的哈希表

- 规则：struct xwall_rule

表 5-3 规则结构

联合体	类型	字段名	含义
	__be32	idx	索引
	__be32	saddr	源地址
	__be32	daddr	目的地址
	__be32	smask	源地址子网掩码
	__be32	dmask	目标地址子网掩码
	__be16	sport_min	源端口最小值
	__be16	sport_max	源端口最大值
	__be16	dport_min	目标端口最小值
	__be16	dport_max	目标端口最大值
	__u8	protocol	协议
	__be32	action	采取的动作
	__u8	logging	是否记录日志
	__be64	timeout	规则超时时间
	struct hlist_node	node	hlist 节点，用于构建链表

- 规则表：struct xwall_ruletable

表 5-4 规则表结构

联合体	类型	字段名	含义
	rwlock_t	lock	读写锁
	unsigned int	rule_num	添加过的规则数
	struct hlist_head	node	hlist 节点，用于构建链表

- 日志：struct xwall_log

表 5-5 日志结构

联合体	类型	字段名	含义
	__be32	idx	索引
	__be64	ts	时间戳
	__be32	saddr	建立该连接的源地址
	__be32	daddr	建立该连接的目的地址
	__u8	protocol	协议

union	__u8	icmp.type	ICMP 协议 TYPE
	__u8	icmp.code	ICMP 协议 CODE
	__be16	tcp.sport	TCP 协议源端口
	__be16	tcp.dport	TCP 协议目的端口
	__u8	tcp.state	TCP 协议状态
	__be16	udp.sport	UDP 协议源端口
	__be16	udp.dport	UDP 协议目的端口
	__be16	len	数据包长度
	__be32	action	采取的动作
	struct hlist_node	node	hlist 节点，用于构建链表

- 日志表：struct xwall_logtable

表 5-6 日志表结构

联合体	类型	字段名	含义
	struct mutex	lock	互斥锁
	unsigned int	log_num	当前日志数
	struct hlist_head	node	hlist 节点，用于构建链表

- 管理日志：struct xwall_mlog

表 5-7 管理日志结构

联合体	类型	字段名	含义
	__be32	idx	索引
	__be64	ts	时间戳
	__u8 []	msg	管理信息
	struct hlist_node	node	hlist 节点，用于构建链表

- 日志表：struct xwall_mlogtable

表 5-8 日志表结构

联合体	类型	字段名	含义
	struct mutex	lock	互斥锁
	unsigned int	log_num	当前管理日志数
	struct hlist_head	node	hlist 节点，用于构建链表

- NAT 记录：struct xwall_nat

表 5-9 NAT 记录结构

联合体	类型	字段名	含义
	ktime_t	timeout	该记录超时时间
	__be32	lan_addr	内网 IP 地址
	__be16	lan_port	内网端口
	bool	valid	该记录是否有效

- Netlink 请求：struct xwall_request

表 5-10 日志表结构

联合体	类型	字段名	含义
	__u8	opcode	操作码
union	struct xwall_rule	rule_add	需要添加的规则
	__be32	rule_del_idx	需要删除的规则索引
	__be32	read_rule.start_idx	需要读取的规则起始索引
	__be32	read_rule.end_idx	需要读取的规则结束索引
	__be32	read_log.start_idx	需要读取的日志起始索引
	__be32	read_log.end_idx	需要读取的日志结束索引
	__be32	read_nat.start_idx	需要读取的 NAT 起始索引
	__be32	read_nat.end_idx	需要读取的 NAT 结束索引
	__be32	read_mlog.start_idx	需要读取的管理日志起始索引
	__be32	read_ mlog.end_idx	需要读取的管理日志结束索引
	__be32	def_act	默认动作

- Netlink 响应：struct xwall_response

表 5-11 日志表结构

联合体	类型	字段名	含义
	__u8	type	响应类型
	__be32	len	消息长度
	__u8 [0]	msg	携带信息的起始位置

5.3 模块接口设计

- 连接[10][11][12][13]

表 5-12 连接相关接口

接口	功能
struct xwall_index *xwall_index_create(struct iphdr *iph)	根据数据包创建在连接哈希表中的索引
struct xwall_connection *xwall_connection_create(struct iphdr *iph)	根据数据包创建连接
struct xwall_hashtable *xwall_hashtable_create(void)	初始化连接表
void xwall_hashtable_add(struct xwall_hashtable *table, struct xwall_connection *conn, struct xwall_index *idx)	向连接哈希表中添加一条连接
void xwall_hashtable_del(struct xwall_hashtable *table, struct xwall_connection *conn)	把连接从连接哈希表中删除
bool _xwall_hashtable_match(struct xwall_hashtable *table, struct xwall_connection *conn, struct xwall_index *idx, bool reverse)	在连接哈希表中寻找是否有当前数据包能够匹配的连接
bool xwall_hashtable_match(struct xwall_hashtable *table, struct xwall_connection *conn, struct xwall_index *idx)	

char *xwall_hashtable_read(struct xwall_hashtable *table, int *len)	读取连接哈希表中的数据
void xwall_hashtable_clean(struct xwall_hashtable *table)	清理连接哈希表中的超时连接
void xwall_hashtable_clear(struct xwall_hashtable *table)	清空整个连接哈希表
#define XWALL_HASH_JEN(keyptr, keylen, hashv, bits)	哈希函数（Jenkins）[14]

- 规则

表 5-13 规则相关接口

接口	功能
struct xwall_ruletable *xwall_ruletable_create(void)	初始化规则表
void xwall_ruletable_add(struct xwall_ruletable *table, struct xwall_rule *rule)	向规则表中添加规则
void xwall_ruletable_del(struct xwall_ruletable *table, struct xwall_rule *rule)	将规则从规则表中删除
bool xwall_ruletable_del_by_idx(struct xwall_ruletable *table, unsigned int idx)	删除规则表中指定索引的规则
struct xwall_rule *xwall_ruletable_match(struct xwall_ruletable *table, struct xwall_connection *conn)	在规则表中寻找当前数据包能够匹配的规则
char *xwall_ruletable_read(struct xwall_ruletable *table, unsigned int start_idx, unsigned int end_idx, int *len)	读取规则表中的数据
int xwall_ruletable_real_num(struct xwall_ruletable *table)	获得规则表中当前实际规则数
bool xwall_save_rule(struct xwall_ruletable *table)	保存当前规则表到文件
bool xwall_load_rule(struct xwall_ruletable *table)	从文件中加载规则
void xwall_ruletable_clear(struct xwall_ruletable *table)	清空整个规则表

- 日志

表 5-14 日志相关接口

接口	功能
struct xwall_log *xwall_log_create(struct sk_buff *skb, unsigned int action)	根据数据包和动作创建日志
struct xwall_logtable *xwall_logtable_create(void)	初始化日志表
void xwall_logtable_add(struct xwall_logtable *table, struct xwall_log *log)	向日志表中添加日志
char *xwall_logtable_read(struct xwall_logtable *table, unsigned int start_idx, unsigned int end_idx, int *len)	读取日志表中的数据
void xwall_logtable_clear(struct xwall_logtable *table)	清空整个日志表

- 管理日志

表 5-15 管理日志相关接口

接口	功能
struct xwall_mlog *xwall_mlog_create(void)	分配并初始化一条管理日志
struct xwall_mlogtable *xwall_mlogtable_create(void)	初始化管理日志表

void xwall_mlogtable_add(struct xwall_mlogtable *table, struct xwall_mlog *mlog)	向管理日志表中添加管理日志
char *xwall_mlogtable_read(struct xwall_mlogtable *table, unsigned int start_idx, unsigned int end_idx, int *len)	读取管理日志表中的数据
void xwall_mlogtable_clear(struct xwall_mlogtable *table)	清空整个管理日志表

- NAT[15]

表 5-16 NAT 相关接口

接口	功能
__be16 xwall_nattable_match(__be32 saddr, __be16 sport)	在 NAT 表中匹配相应的条目
void xwall_update_checksum_ip(struct iphdr *iph)	更新 IP 校验码
void xwall_update_checksum_tcp(struct sk_buff *skb, struct tcphdr *tcph, struct iphdr *iph)	更新 TCP 校验码
void xwall_update_checksum_udp(struct sk_buff *skb, struct udphdr *udph, struct iphdr *iph)	更新 UDP 校验码
void xwall_update_checksum_icmp(struct sk_buff *skb, struct icmphdr *icmph, struct iphdr *iph)	更新 ICMP 校验码
unsigned int xwall_nat_in(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)	在 PRE-ROUTING 处进行目标地址转换
unsigned int xwall_nat_out(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)	在 POST-ROUTING 处进行源地址转换

- Netlink

表 5-17 Netlink 相关接口

接口	功能
struct xwall_response *xwall_response_create(enum XWALL_RESPONSE_TYPE type, int len, char *data)	构造 Netlink 响应
int xwall_netlink_send(int pid, char *data, int len)	使用 Netlink 发送数据
int xwall_msg_hdr(int pid, char *data, int len)	处理 Netlink 接收到的请求
void xwall_netlink_recv(struct sk_buff *skb)	使用 Netlink 接受数据

- Netfilter[16][17]

表 5-18 Netfilter 相关接口

接口	功能
unsigned int xwall_filter(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)	进行数据包过滤

- 定时器[18][19]

表 5-19 定时器相关接口

接口	功能
void xwall_timer_callback(struct timer_list *t)	定时器回调函数，清理超时连接

六、 系统测试

6.1 测试环境

- 网络拓扑

  测试环境的网络拓扑如图 6-1 所示。

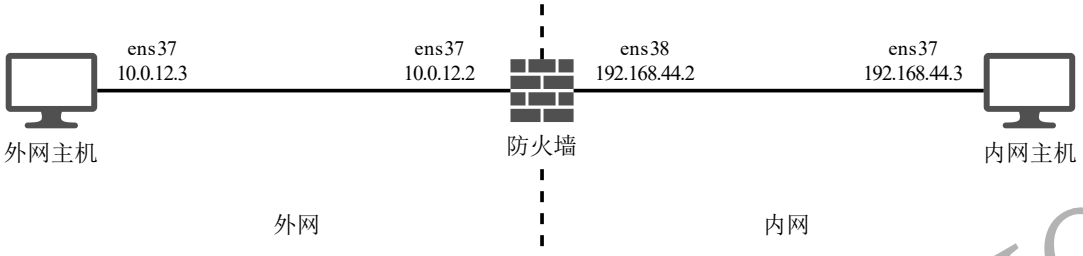

图 6-1 测试环境网络拓扑

- 网络拓扑配置 1：虚拟网络设置及 IP 分配

  外网主机、防火墙、内网主机分别为三台运行在 VMware 中的 Ubuntu 虚拟机（外网主机及内网主机为 Ubuntu 20.04，防火墙为 Ubuntu 22.04），通过以下对 VMware 和 Ubuntu 的设置构建该拓扑（仅适用于目前 Ubuntu 版本，不同版本的 Ubuntu 配置可能会有一定差异）：

  ✧ VMware 设置：在“编辑→虚拟网络编辑器→更改设置”中添加两个仅主机模式虚拟网络，其子网 IP 及掩码分别为 10.0.12.0/24 和 192.168.44.0/24，如图 6-2 中的 VMnet3 及 VMnet4 所示。

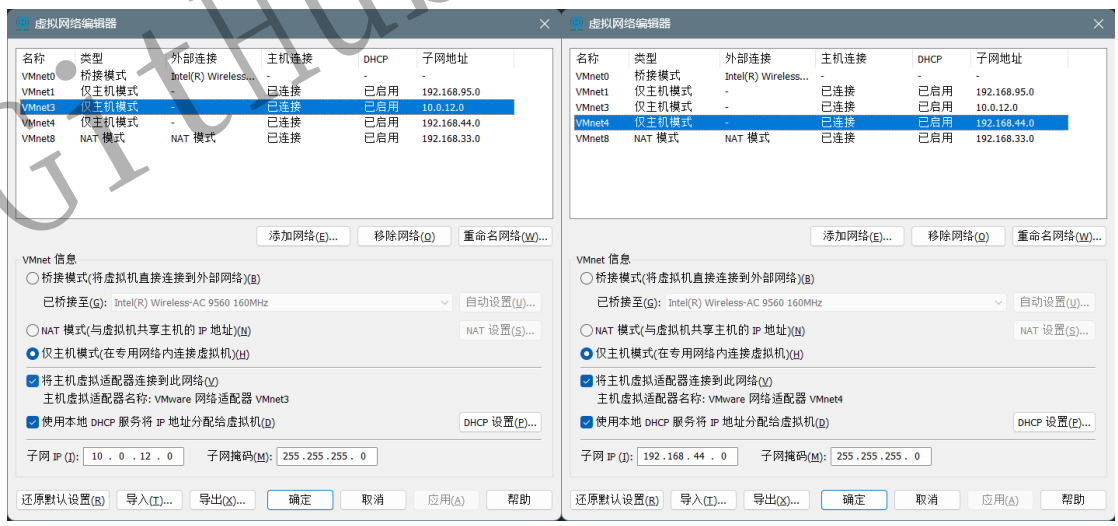

图 6-2 VMware 虚拟网络设置

  ✧ 添加网络适配器：在“虚拟机→设置→添加→网络适配器”中分别为外网主机、内网主机和防火墙主机添加网络适配器。为内网主机添加 1 个网络适配器，网络连接设置为自定义→VMnet4（仅主机模式）；为外

网主机添加 1 个网络适配器，网络连接设置为自定义→VMnet3（仅主机模式）；为防火墙主机添加 2 个网络适配器，网络连接分别设置为自定义→VMnet3（仅主机模式）和自定义→VMnet4（仅主机模式）。为了避免已有适配器的影响，可以将已有适配器移除。

- ✧ 外网主机设置：在“Settings→Network→Wired Settings→IPv4”进行如图 6-3 左侧的设置并 Apply，若修改未生效可以尝试重启这个网络；使用 route -n 及 ifconfig 查看网络信息如图 6-4 所示。
- ✧ 内网主机设置：在“Settings→Network→Wired Settings→IPv4”进行如图 6-3 右侧的设置并 Apply，若修改未生效可以尝试重启这个网络；使用 route -n 及 ifconfig 查看网络信息如图 6-5 所示。

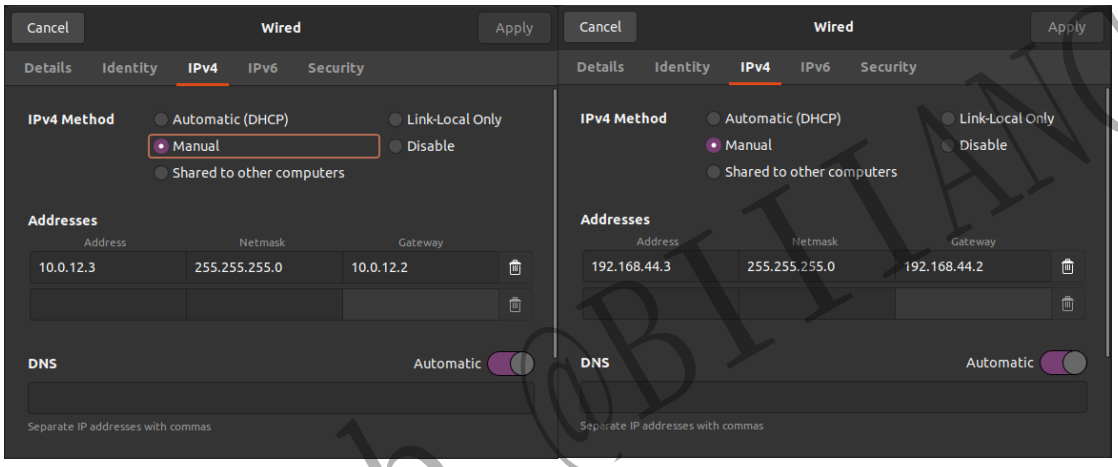

图 6-3 外网主机及内网主机的网络设置

```
xubiang@ubuntu:~/Desktop$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.12.2       0.0.0.0         UG    20100  0        0 ens37
10.0.12.0       0.0.0.0         255.255.255.0   U     100    0        0 ens37
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens37
xubiang@ubuntu:~/Desktop$ ifconfig
ens37: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.12.3  netmask 255.255.255.0  broadcast 10.0.12.255
        inet6 fe80::972f:f964:e661:bbe  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:d9:d6:30  txqueuelen 1000  (Ethernet)
        RX packets 11444  bytes 954154 (954.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6454  bytes 535408 (535.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 6-4 外网主机网络信息

```
xubiang@ubuntu:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.44.2    0.0.0.0         UG    20100  0        0 ens37
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens37
192.168.44.0    0.0.0.0         255.255.255.0   U     100    0        0 ens37
xubiang@ubuntu:~$ ifconfig
ens37: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.44.3  netmask 255.255.255.0  broadcast 192.168.44.255
        inet6 fe80::be1b:1fd0:6483:216b  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:7e:dd:48  txqueuelen 1000  (Ethernet)
        RX packets 74  bytes 5943 (5.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 84  bytes 10121 (10.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 6-5 内网主机网络信息

✧ 防火墙设置：在“Settings→Network→Wired Settings→IPv4”进行如图 6-6 的设置并 Apply，若修改未生效可以尝试重启这两个网络；为避免干扰，暂时关闭 ens33 对应的网络。此时使用 route -n 及 ifconfig 查看网络信息如图 6-7 所示。

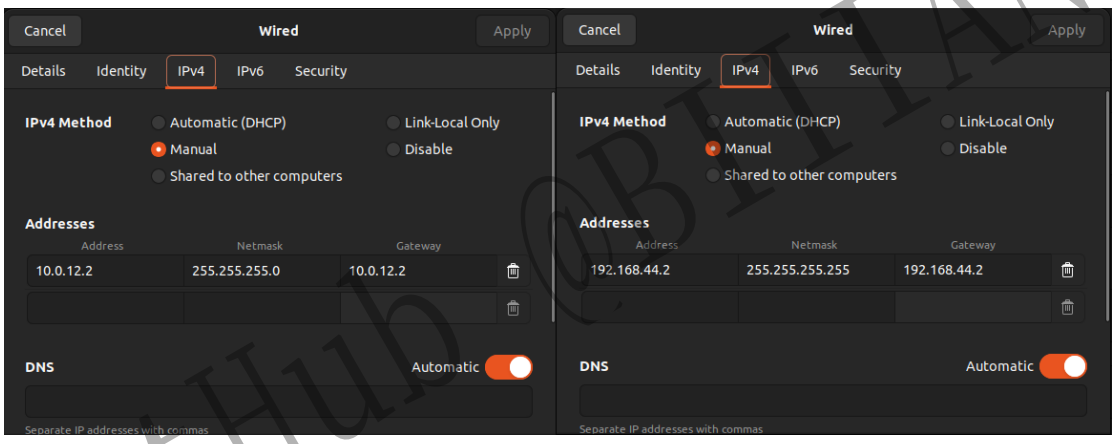

图 6-6 防火墙的网络设置

```
xubiang@xubiang:~/Desktop/CDNS/gui/dist$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.12.2       0.0.0.0         UG    20100  0        0 ens37
0.0.0.0         192.168.44.2    0.0.0.0         UG    20101  0        0 ens38
10.0.12.0       0.0.0.0         255.255.255.0   U     100    0        0 ens37
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens37
192.168.44.2    0.0.0.0         255.255.255.255 UH    20101  0        0 ens38
xubiang@xubiang:~/Desktop/CDNS/gui/dist$ ifconfig ens37
ens37: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.12.2  netmask 255.255.255.0  broadcast 10.0.12.255
        inet6 fe80::eae4:9b13:217:3b7c  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:c2:cd:e8  txqueuelen 1000  (Ethernet)
        RX packets 13115  bytes 1261047 (1.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 17433  bytes 1182943 (1.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

xubiang@xubiang:~/Desktop/CDNS/gui/dist$ ifconfig ens38
ens38: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.44.2  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::fb83:c6c3:f345:c2fa  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:c2:cd:f2  txqueuelen 1000  (Ethernet)
        RX packets 7988  bytes 873318 (873.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13545  bytes 875700 (875.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 6-7 防火墙网络信息

- 网络拓扑配置 2：防火墙主机转发设置
  - ✧ 经过以上配置后，相应的网卡及 IP 已分配完毕，之后在防火墙主机上使用命令 sudo sysctl net.ipv4.ip_forward=1 启用转发，并使用命令 sudo iptables -F 清除 iptables 规则。
  - ✧ 经过以上配置后，经过测试三台主机仍不能相互联通，原因是 Ubuntu 22.04 所使用的 iptables 对 FORWARD 包的默认策略为 DROP，如图 6-8 所示，因此还需使用命令 sudo iptables -P FORWARD ACCEPT 将其默认策略设置为 ACCEPT，如图 6-9 所示。[20]

```
xubiang@xubiang:~$ sudo iptables -F
xubiang@xubiang:~$ sudo iptables -nL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
```

图 6-8 FORWARD 策略默认为 DROP

```
xubiang@xubiang:~$ sudo iptables -P FORWARD ACCEPT
xubiang@xubiang:~$ sudo iptables -nL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
```

图 6-9 将 FORWARD 策略修改为 ACCEPT

- ✧ 经过以上配置后，内网主机与防火墙仍不可联通，经过分析，应为防火墙主机添加对应的路由表，相应命令为 sudo route add -net 10.0.12.0/24 ens37 及 sudo route add -net 192.168.44.0/24 ens38，添加路由之后防火墙主机的路由表如图 6-10 所示。

```
xubiang@xubiang:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.12.2       0.0.0.0         UG    20101  0        0 ens37
0.0.0.0         192.168.44.2    0.0.0.0         UG    20102  0        0 ens38
10.0.12.0       0.0.0.0         255.255.255.0   U     0      0        0 ens37
10.0.12.0       0.0.0.0         255.255.255.0   U     101    0        0 ens37
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens37
192.168.44.0    0.0.0.0         255.255.255.0   U     0      0        0 ens38
192.168.44.2    0.0.0.0         255.255.255.255 UH    20102  0        0 ens38
```

图 6-10 完善后的防火墙路由表

- ✧ 以上“防火墙主机转发设置”部分的设置命令可使用 shell 脚本 xwall_net_setup.sh 进行快速配置。
- ✧ 至此，三台主机可以互相联通，环境配置完毕。

- 外网主机服务配置：
  - ✧ TCP 协议测试：使用 python 自带的 HTTP SERVER 在 9000 端口启动服务：python3 -m http.server 9000；
  - ✧ UDP 协议测试：使用基于 python 的简短的 UDP SERVER 在 9001 端口启动服务：python3 udp.server.py。[21]
- 防火墙服务配置：
  - ✧ 在工作目录根目录下设置环境变量：export XWALL_PATH=$(pwd)，如图 6-11 所示。

```
xubiang@xubiang:~/Desktop/XWALL$ export XWALL_PATH=$(pwd)
xubiang@xubiang:~/Desktop/XWALL$ echo $XWALL_PATH
/home/xubiang/Desktop/XWALL
```

图 6-11 设置工作目录环境变量

- ✧ 使用 xwall_run_gui.sh 编译、加载模块和用户程序，如图 6-12 所示。

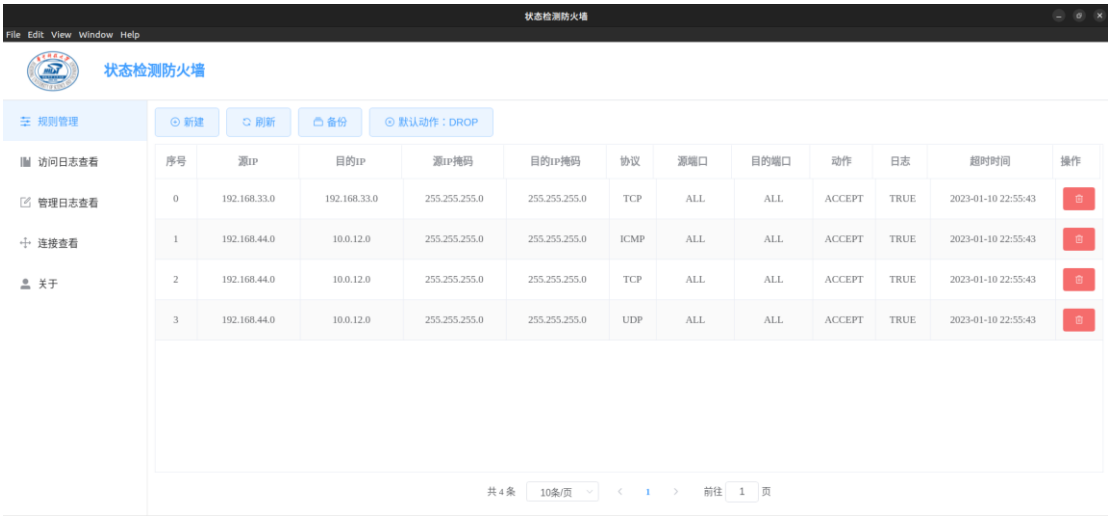

图 6-12 系统启动成功

6.2 功能测试

- 测试设计

设计的测试步骤、预期结果及测试目的如表 6-1 所示。

表 6-1 功能测试设计方案

序号	测试步骤	预期结果	测试目的
1	首次进入系统	界面正常加载、初始规则正常加载，默认策略为 DROP，允许内网主机主动向外网主机发起的 TCP、UDP、ICMP（虚拟）连接	规则维护（查看规则）
2	ICMP 测试	内网主机可以正常 ping 通外网主机且外网主机接收到的 ICMP 数据包的来源为防火墙（即经过了源地址转换）	规则过滤（ICMP） 状态检测（ICMP） NAT 转换
3	查看连接	为 ICMP 建立了虚拟连接，响应数据包正是匹配了该虚拟连接才被允许通过	连接管理（建立、匹配、查看）
4	删除 ICMP 规则后进行 ICMP 测试	ICMP 规则被正常删除，内网主机不可以正常 ping 通外网主机	规则维护（删除规则） 规则过滤（ICMP） 默认动作（执行）
5	查看并清理访问日志	访问日志记录了以上被允许和丢弃的记录	日志审计（保存、查看）
6	TCP 测试	内网主机可以正常访问外网主机的 HTTP 服务	规则过滤（TCP） 状态检测（TCP）
7	查看连接	为 TCP 建立了连接，响应数据包及后续请求数据包正是匹配了该连接才被允许通过	连接管理（建立、匹配、查看）
8	删除 TCP 规则后进行 TCP 测试	TCP 规则被正常删除，内网主机不可以正常访问外网主机的 HTTP 服务	规则维护（删除规则） 规则过滤（TCP）

			默认动作（执行）
9	查看并清理访问日志	访问日志记录了以上被允许和丢弃的记录	日志审计（保存、查看）
10	UDP 测试	内网主机可以正常访问外网主机的简单 UDP 服务且外网主机接收到的 UDP 数据包的来源为防火墙	规则过滤（UDP） 状态检测（UDP） NAT 转换
11	查看连接	为 UDP 建立了虚拟连接，响应数据包及后续请求数据包正是匹配了该连接才被允许通过	连接管理（建立、匹配、查看）
12	删除 UDP 规则后进行 UDP 测试	UDP 规则被正常删除，内网主机不可以正常访问外网主机的简单 UDP 服务	规则维护（删除规则） 规则过滤（UDP） 默认动作（执行）
13	查看并清理访问日志	访问日志记录了以上被允许和丢弃的记录	日志审计（保存、查看）
14	修改默认策略为 ACCEPT 后进行 ICMP、TCP、UDP 测试	默认策略被正常修改为 ACCEPT，内网主机可以正常 ping 通外网主机、内网主机可以正常访问外网主机的 HTTP 服务和简单 UDP 服务	默认动作（设置）
15	保存规则	规则被顺利保存（此时只剩一条规则）	规则维护（保存）
16	查看管理日志	管理日志记录了以上规则修改和查看等记录	日志审计（管理日志）
17	使用 xwall_run_gui.sh 重新运行	规则被顺利读取并加载（仅一条规则）	规则维护（加载）
18	添加一条 ICMP 的 ACCEPT 规则后进行 ICMP 测试	规则被成功添加，内网主机可以正常 ping 通外网主机	规则维护（添加）
19	等待 3 分钟（ICMP 虚拟连接的超时时间）后，查看连接	ICMP 连接因超时被删除	连接管理（超时删除）

以上测试项目基本覆盖了“互评计分表”中的任务要求 1-7；任务要求 8“基本界面：界面友好，能够完成上述设置、查询等管理操作”可通过整体过程及截图得到表现；任务要求 9“拓展任务：若实现图形界面、规则支持时间控制，额外加分”中的图形界面可以通过截图得到表现，时间控制目前只完成了规则超时失效、未完成规则超时自动删除。

以下测试基于 GUI 程序，CLI 程序同样可以达到相同的测试结果。

- 测试过程及结果

该部分序号与“测试设计”部分中的表 6-1 相对应。以下结果仅为部分展示，防火墙运行时的一些关键数据可以通过 dmesg 指令查看。

1） 首次进入系统

图 6-13 首次进入系统

2） ICMP 测试

```
xubiang@ubuntu:~$ ping 10.0.12.3 -c 2
PING 10.0.12.3 (10.0.12.3) 56(84) bytes of data.
64 bytes from 10.0.12.3: icmp_seq=1 ttl=63 time=1.14 ms
64 bytes from 10.0.12.3: icmp_seq=2 ttl=63 time=0.868 ms

--- 10.0.12.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.868/1.003/1.139/0.135 ms
```

图 6-14 内网主机可以成功 ping 通外网主机

```
xubiang@ubuntu:~/Desktop$ sudo tcpdump -i any icmp -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 bytes
07:53:27.653971 IP 10.0.12.2 > 10.0.12.3: ICMP echo request, id 51, seq 1, length 64
07:53:27.654012 IP 10.0.12.3 > 10.0.12.2: ICMP echo reply, id 51, seq 1, length 64
07:53:28.655449 IP 10.0.12.2 > 10.0.12.3: ICMP echo request, id 51, seq 2, length 64
07:53:28.655481 IP 10.0.12.3 > 10.0.12.2: ICMP echo reply, id 51, seq 2, length 64
```

图 6-15 外网主机接收到的 ICMP 数据包的来源为防火墙

3） 查看连接

图 6-16 为 ICMP 建立了虚拟连接

4） 删除 ICMP 规则后进行 ICMP 测试

图 6-17 删除了 ICMP 规则

```
xubiang@ubuntu:~$ ping 10.0.12.3 -c 2
PING 10.0.12.3 (10.0.12.3) 56(84) bytes of data.

--- 10.0.12.3 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1018ms
```

图 6-18 内网主机不可以成功 ping 通外网主机

5) 查看并清理访问日志

图 6-19 访问日志记录了以上被允许和丢弃的记录

图 6-20 清除访问日志

6) TCP 测试

图 6-21 内网主机可以正常访问外网主机的 HTTP 服务

7） 查看连接

图 6-22 为 TCP 建立了连接

8） 删除 TCP 规则后进行 TCP 测试

图 6-23 删除了 TCP 规则

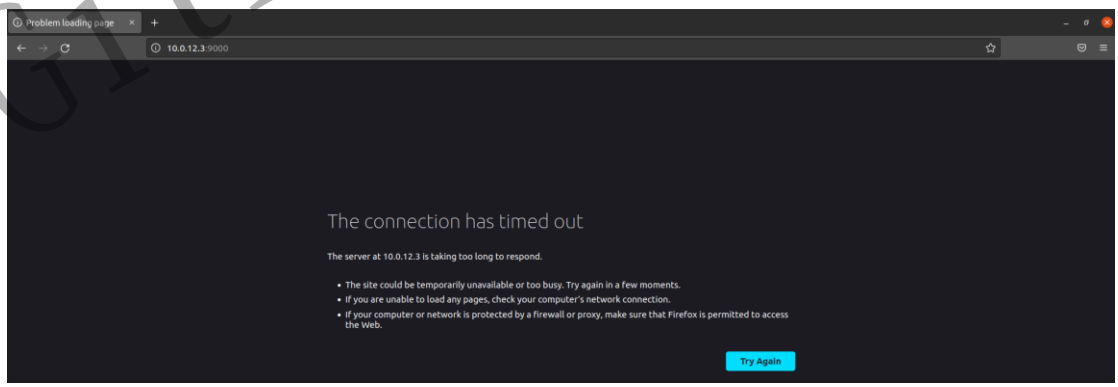

图 6-24 内网主机不可以正常访问外网主机的 HTTP 服务

9） 查看并清理访问日志

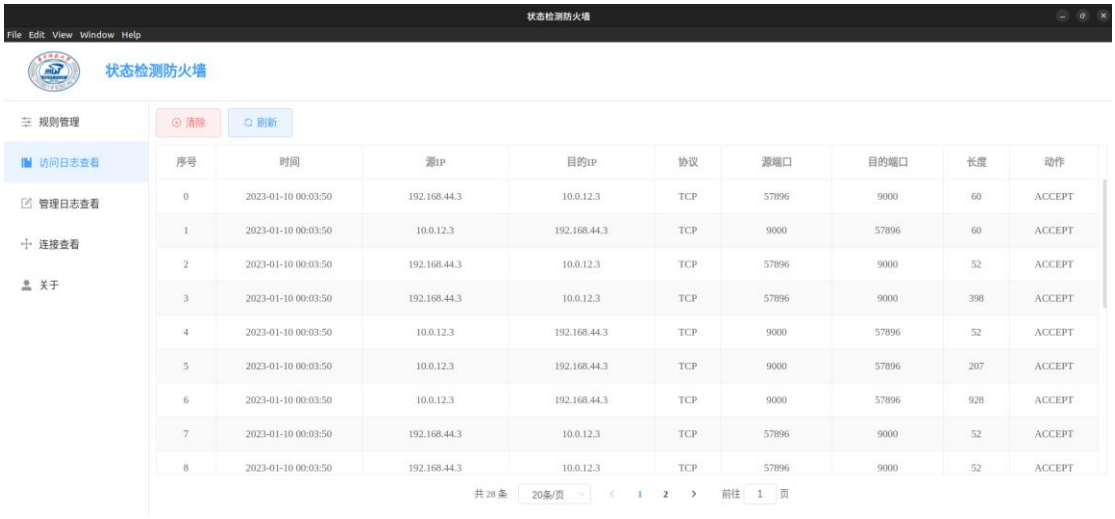

图 6-25 访问日志记录了以上被允许的记录

图 6-26 访问日志记录了以上被丢弃的记录

图 6-27 清除访问日志

10) UDP 测试

```
xubiang@ubuntu:~$ echo "hello" | nc -u 10.0.12.3 9001
Hello, udp client.
```

图 6-28 内网主机可以正常访问外网主机的简单 UDP 服务

```
xubiang@ubuntu:~/Desktop$ python3 udp.server.py
10.0.12.2,11805: hello
```

图 6-29 外网主机接收到的 UDP 数据包的来源为防火墙

11）查看连接

图 6-30 为 UDP 建立了虚拟连接

12）删除 UDP 规则后进行 UDP 测试

图 6-31 删除了 UDP 规则

```
xubiang@ubuntu:~$ echo "hello" | nc -u 10.0.12.3 9001
```

图 6-32 内网主机不可以正常访问外网主机的简单 UDP 服务

13）查看并清理访问日志

图 6-33 访问日志记录了以上被允许和丢弃的记录

图 6-34 清除访问日志

14）修改默认策略为 ACCEPT 后进行 ICMP、TCP、UDP 测试

图 6-35 修改默认策略为 ACCEPT

```
xubiang@ubuntu:~$ ping 10.0.12.3 -c 2
PING 10.0.12.3 (10.0.12.3) 56(84) bytes of data.
64 bytes from 10.0.12.3: icmp_seq=1 ttl=63 time=0.935 ms
64 bytes from 10.0.12.3: icmp_seq=2 ttl=63 time=1.00 ms

--- 10.0.12.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.935/0.967/1.000/0.032 ms
```

图 6-36 内网主机可以成功 ping 通外网主机

图 6-37 内网主机不可以正常访问外网主机的 HTTP 服务

```
xubiang@ubuntu:~$ echo "hello" | nc -u 10.0.12.3 9001
Hello, udp client.
```

图 6-38 内网主机可以正常访问外网主机的简单 UDP 服务

15）保存规则

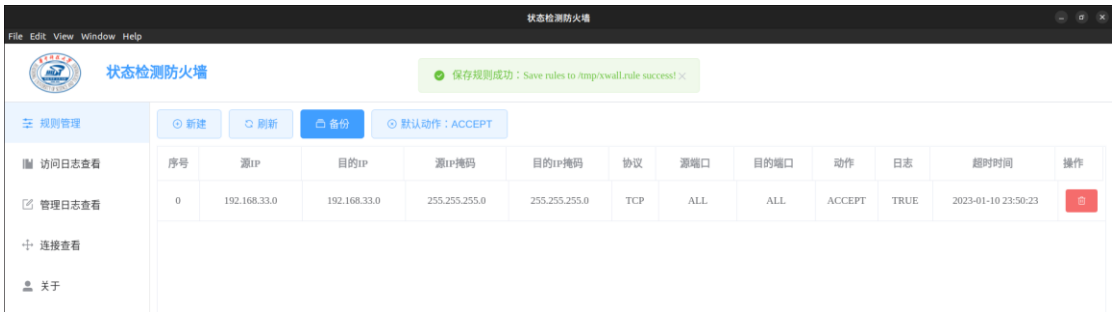

图 6-39 规则保存成功

```
xubiang@xubiang:~/Desktop/XWALL$ ll /tmp | grep xwall.rule
-rw-r--r--  1 xubiang xubiang  256  1月  10 00:25 xwall.rule
xubiang@xubiang:~/Desktop/XWALL$
```

图 6-40 保存的规则文件

16) 查看管理日志

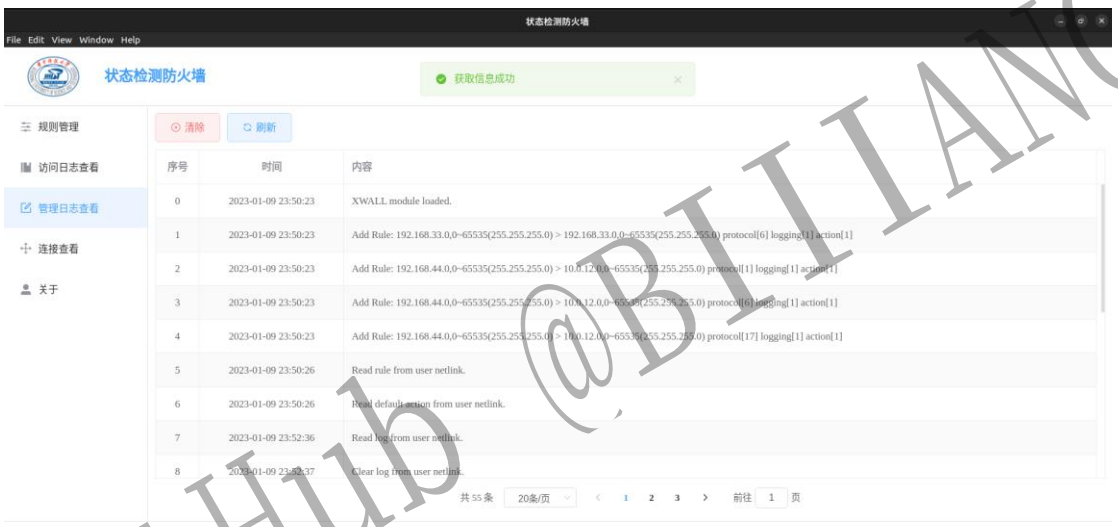

图 6-41 管理日志记录了以上规则修改和查看等记录

17) 使用 xwall_run_gui.sh 重新运行

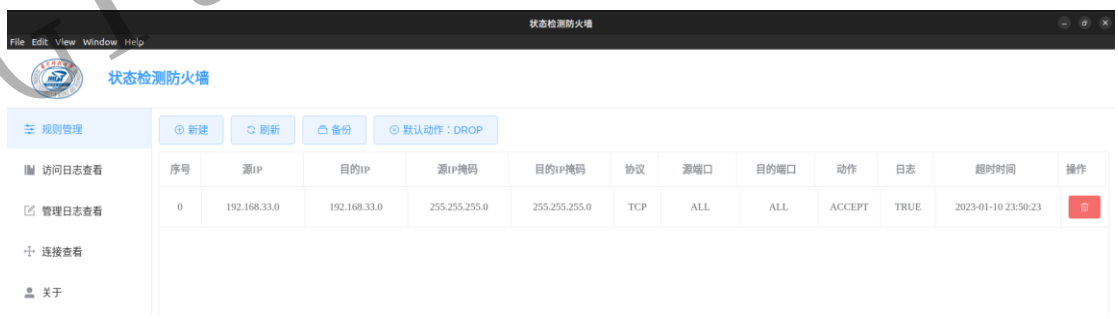

图 6-42 规则被顺利读取并加载

```
[ 6940.834967] xwall[              xwall_init]: XWALL module loaded.
[ 6940.834989] xwall[         xwall_load_rule]: Add 1 rule from file /tmp/xwall.rule done.
[ 6943.455230] xwall[       xwall_netlink_recv]: Data recv from user 7672.
[ 6943.455240] xwall[       xwall_netlink_send]: Data send to user 7672.
[ 6943.456792] xwall[       xwall_netlink_recv]: Data recv from user 7674.
[ 6943.456801] xwall[       xwall_netlink_send]: Data send to user 7674.
```

图 6-43 规则被顺利读取并加载（内核 dmesg 信息）

18）添加一条 ICMP 的 ACCEPT 规则后进行 ICMP 测试

图 6-44 添加规则所填表单

图 6-45 添加规则成功

```
xubiang@ubuntu:~$ ping 10.0.12.3 -c 2
PING 10.0.12.3 (10.0.12.3) 56(84) bytes of data.
64 bytes from 10.0.12.3: icmp_seq=1 ttl=63 time=0.924 ms
64 bytes from 10.0.12.3: icmp_seq=2 ttl=63 time=0.840 ms

--- 10.0.12.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.840/0.882/0.924/0.042 ms
```

图 6-46 内网主机可以成功 ping 通外网主机

图 6-47 为 ICMP 建立了虚拟连接

19）等待 3 分钟（ICMP 虚拟连接的超时时间）后，查看连接

图 6-48 连接超时后被自动删除

```
[ 7544.865543] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7555.105248] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7565.345364] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7575.585210] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7585.825598] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7596.065299] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7606.305580] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7616.544707] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7626.785430] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7637.024988] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7647.264729] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7657.504336] xwall[     xwall_timer_callback]: Clean the connection table...
[ 7667.744727] xwall[     xwall_timer_callback]: Clean the connection table...
```

图 6-49 内核程序每 10 秒检测并删除超时连接（内核 dmesg 消息）

6.3 性能测试

为了避免各种日志对系统性能的影响，首先将 UDP 规则、TCP 规则和内核中的 default_logging 均设为 false，并将 printk 内核输出关闭。

在以上环境下，使用 iperf3 进行 UDP 测速，统计结果如表 6-2 所示。开启防火墙的情况下，实际带宽在带宽参数为 350M 时达到最大，约 305M；关闭防火墙的情况下，实际带宽在带宽参数为 500M 时达到最大，约 487M。因此，防火墙给整个系统带来了约 40%的性能损失，推测是在数据包的处理过程中多次申请了内存以存放临时数据和连接表的读写锁带来的影响，导致性能下降较多。

表 6-2 性能测试结果

	第一次	第二次	第三次	第四次	第五次	第六次	平均值
开启防火墙	321 Mb/s	305 Mb/s	305 Mb/s	303 Mb/s	301 Mb/s	297 Mb/s	305 Mb/s
关闭防火墙	485 Mb/s	487 Mb/s	489 Mb/s	489 Mb/s	486 Mb/s	483 Mb/s	487 Mb/s

由于测试平台为三台虚拟机，且分配的 CPU 核心数已经超过了硬件具备的核心数，在测试过程中难以避免地会出现由于 CPU 分配导致的相差悬殊的结果，考虑到 iperf3 作为测试工具在 UDP 测速上也存在一些不足，以上测试结果仅作为参考，具有一定的不确定性。

七、 心得体会及意见建议

略。

[1] https://en.wikipedia.org/wiki/Netfilter
[2] https://github.com/torvalds/linux/blob/fc02cb2b37fe2cbf1d3334b9f0f0eab9431766c4/net/ipv4/netfilter/iptable_nat.c
[3] https://github.com/torvalds/linux/blob/6f2b76a4a384e05ac8d3349831f29dff5de1e1e2/security/smack/smack_netfilter.c
[4] https://www.kernel.org/doc/html/v4.10/process/coding-style.html#
[5] https://github.com/DaveGamble/cJSON
[6] https://github.com/yarnpkg/yarn
[7] https://github.com/vuejs/vue
[8] https://github.com/vuejs/vue-cli
[9] https://github.com/ElemeFE/element
[10] https://kernelnewbies.org/FAQ/Hashtables
[11] https://lwn.net/Articles/510202/
[12] https://github.com/torvalds/linux/blob/5bfc75d92efd494db37f5c4c173d3639d4772966/include/linux/hashtable.h
[13] https://github.com/torvalds/linux/blob/5bfc75d92efd494db37f5c4c173d3639d4772966/drivers/net/wireguard/peerlookup.c
[14] https://troydhanson.github.io/uthash/userguide.html#hash_functions
[15] https://github.com/gtungatkar/NAT
[16] https://github.com/torvalds/linux/blob/fc02cb2b37fe2cbf1d3334b9f0f0eab9431766c4/net/ipv4/netfilter/iptable_nat.c
[17] https://github.com/torvalds/linux/blob/6f2b76a4a384e05ac8d3349831f29dff5de1e1e2/security/smack/smack_netfilter.c
[18] https://blog.csdn.net/hhhhhyyyyy8/article/details/102885037
[19] https://blog.csdn.net/qq_42174306/article/details/122770059
[20] https://blog.csdn.net/weixin_45623111/article/details/117325657
[21] https://blog.csdn.net/chenfei3306/article/details/119103292