杭州電子科技大学

网络安全理论与技术实验

实验一　基于 snort 的入侵检测系统实验

一、　实验目的

1、熟悉 NIDS 的基本概念与功能。

2、学习 Snort 工具基础命令及原理，学习三种工作方式（嗅探、记录数据包、入侵检测）的使用。

3、基于 Snort 规则实现 NIDS，当有人通过任何方法以 root 用户身份在计算机外部登录计算机时，实现邮件报警。

二、　实验原理

1.　理解 NIDS 的基本工作原理

掌握网络入侵检测系统通过被动监听网络流量、解析协议字段并依据规则进行匹配检测的基本思想，理解其在主机安全防护体系中的作用与局限。

2.　熟悉 Snort 基础使用

Snort 是一个开源的轻量级入侵检测系统（NIDS），使用 C 语言编写，支持 windows、Linux 平台，有三种工作模式，包括：嗅探、记录数据包、入侵检测。嗅探器模式仅仅是从网络上读取数据包并作为连续不断的流显示在终端上。数据包记录器模式把数据包记录到硬盘上。入侵检测模式是最复杂的，而且是可配置的。可以让 Snort 分析网络数据流以匹配用户定义的一些规则，并根据检测结果采取一定的动作。

3. 实现基于 Snort 规则的异常行为检测与告警

针对“外部主机以 root 用户身份登录本机”的高危行为，基于 Snort 规则对异常数据包进行检测，并在检测到入侵行为时触发告警.

三、 实验环境

宿主机环境：

Ubuntu 22.04 LTS

虚拟机环境：

虚拟化平台：Oracle VM VirtualBox 7.x

操作系统：Ubuntu 22.04 LTS

Snort 版本：2.9

网络适配器配置：

- 网卡 1：桥接网络（Bridged Adapter）enp0s3
- 网卡 2：Host-Only 网络（用于与宿主机通信） enp0s8

四、 主要操作步骤及实验结果记录

1、嗅探：使用命令 snort -dev -i enp0s3 -v 在实验主机上对网络接口 enp0s3 采用详细模式进行监听，在终端显示数据包网络层、链路层、应用层信息。

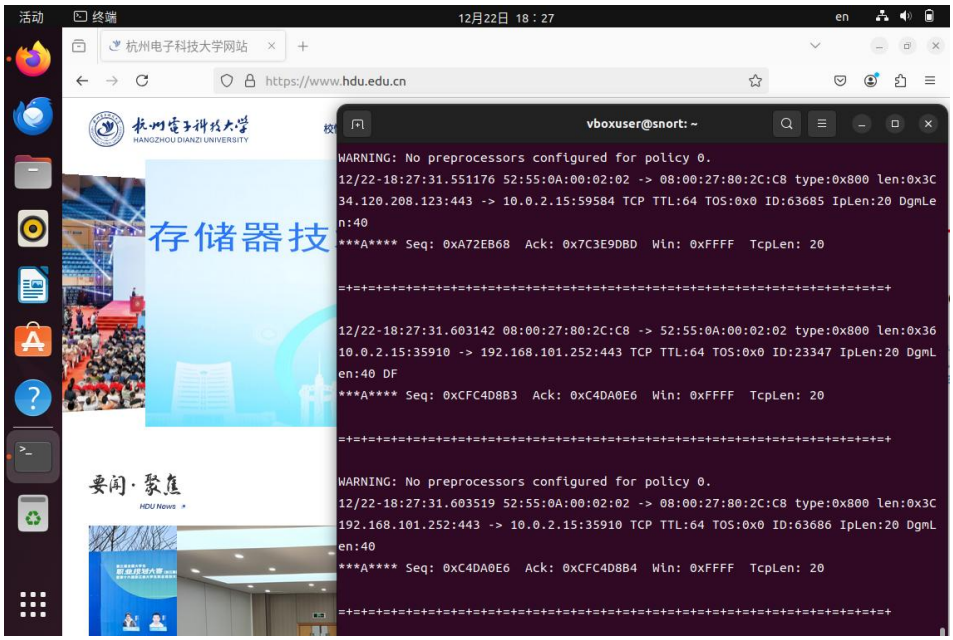


图 1 访问 hdu.edu.cn 时嗅探到 tcp 包

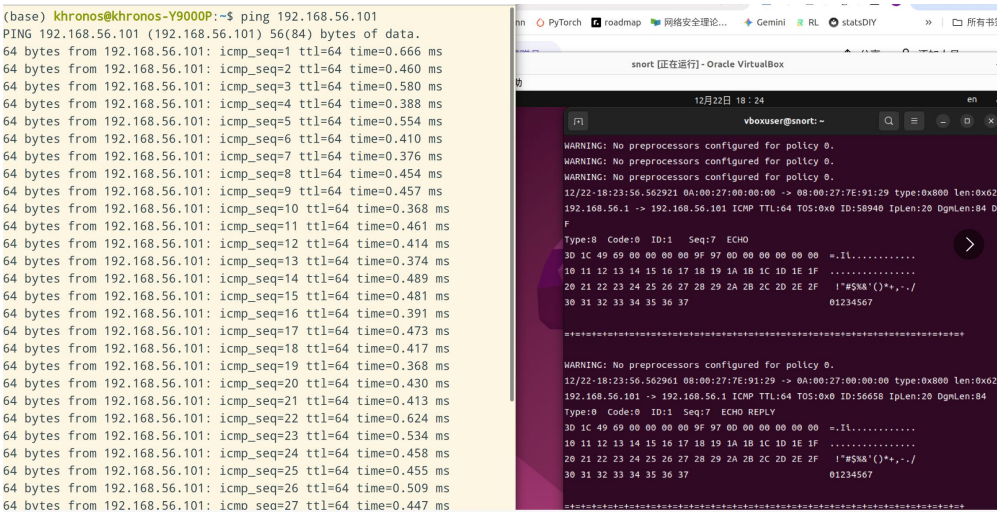


图 2 外部计算机 ping 时收到 icmp 包

2、数据包记录：在 snort 命令后加 -l /var/log/snort -b，将嗅探数据包以二进制格式记录到日志中，后续使用 snort -r 进行读取和分析。

```
vboxuser@snort:~$ sudo snort -v -d -e -i enp0s3 -l /var/log/snort/ -b
Running in packet logging mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Log directory = /var/log/snort/
```

图 3 详细模式监听并存入日志

```
vboxuser@snort:~$ sudo snort -r /var/log/snort/snort.log
Running in packet dump mode

        --== Initializing Snort ==--
Initializing Output Plugins!
```

图 4 使用-r 命令在终端读取

3、入侵检测。在$RULE_PATH 添加 new.rules 增加对 telnet root 登录警报的规则，在 snort.conf 末尾添加 include new.rules 后, 使用 snort -c /etc/snort/snort.conf -i enp0s8 -A console 开启入侵检测, 将警报输出在终端控制台中。

```
alert tcp any any -> $HOME_NET 23 (msg:"TELNET root login attempt"; content:"roo
t"; nocase; sid:1000001; rev:1;)
```

图 5 new.rules 定位网络流量方向、23 端口的 telnet 和 root 负载

```
# Event thresholding or suppression commands. See threshold.conf
include threshold.conf

include $RULE_PATH/new.rules

-- 插入 --                                              759,1         底端
```

图 6 snort.conf 引入新规则

```
vboxuser@snort:~$ sudo vim /etc/snort/rules/new.rules
vboxuser@snort:~$ sudo vim /etc/snort/snort.conf
vboxuser@snort:~$ sudo snort -c /etc/snort/snort.conf -i enp0s8 -A console
Running in IDS mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Initializing Preprocessors!
```

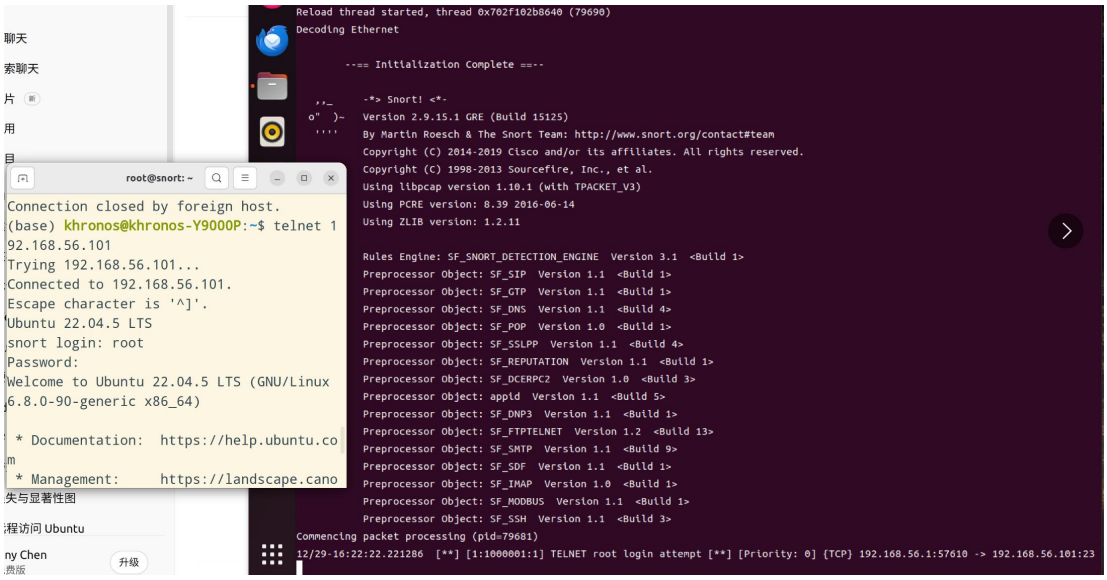


图 7 外部 telnet 后在终端输出报警

五、 实验分析总结及心得

遇到和解决了很多困难:

首次尝试在 ubuntu 宿主机用 virtualbox 装 ubuntu 虚拟机，过程比 win11 下使用 vmware 平台繁琐和困难，例如由于 Linux 内核默认拒绝未签名模块而必须在 bios 暂时关闭 Secure Boot，以及虚拟机创建后上半屏幕频闪需要额外调整显存设置、默认的 vboxuser 不在 sudoers 文件中、虚拟机并不默认安装 telnet 服务等。

也有充分的收获，通过实操熟悉 snort 的三种模式使用加深了对 NIDS 网络嗅探与规则匹配原理的更细致理解，在错误排查过程中通过抓包并逐层解析协议寻找问题，也是对计算机网络理论知识的补充。

实验二 基于 snort 规则实现简单 NIDS

一、 实验目的

1. 理解 NIDS 的基本工作原理

掌握网络入侵检测系统通过被动监听网络流量、解析协议字段并依据规则进行匹配检测的基本思想，理解其在主机安全防护体系中的作用与局限。

2. 熟悉 Snort 规则结构与匹配机制

通过下载并分析官方 Snort 规则文件，理解规则中对协议、端口、方向、内容匹配及告警动作的描述方式，为后续规则复用与扩展奠定基础。

3. 掌握网络数据包捕获与多层协议解析方法

通过抓包工具捕获外部主机以 Telnet 方式登录实验主机的通信数据，对链路层、网络层和应用层协议字段进行解析，建立对网络协议分层结构的直观认识。

4. 实现基于 Snort 规则的异常行为检测与告警

针对“外部主机以 root 用户身份登录本机”的高危行为，基于 Snort 规则对异常数据包进行检测，并在检测到入侵行为时触发告警，将告警信息发送至个人邮箱。

二、 实验原理

基于 Snort 规则的网络入侵检测思想，通过对实验主机网络流量的被动监听与分析，实现对外部主机以 root 用户身份远程登录行为的识别与告警。

利用抓包工具对经过实验主机网卡的网络数据进行捕获，程序按照网络协议逐层对数据包进行解析。首先在链路层解析以太网帧头，获取源与目的 MAC 地址等基础信息；随后在网络层解析 IP 头部，提取源 IP、目的 IP 以及所使用的上层协议类型；在传输层与应用层中，对 TCP 连接及其负载数据进行分析，重点关注 Telnet 会话中的应用层明文内容。

从 Snort 官方网站下载规则文件，并结合 Telnet 登录场景，对规则进行分析与复用。当捕获到的数据包满足“来自外部主机、使用 Telnet 协议 (即端口 23) 、会话内容中包含 root 用户登录特征”等条件时，检测模块依据 Snort 规则判定该行为为潜在入侵或高危操作。

在入侵行为被识别后，系统触发告警机制生成安全事件信息。告警内容包括事件发生时间、通信双方地址、所匹配的规则描述及异常行为类型等，并通过邮件发送模块将告警信息自动发送至预设的个人邮箱，实现了从网络流量监测、异常行为判定到安全告警输出的完整检测链路。

三、 实验环境

宿主机环境：

Ubuntu 22.04 LTS

虚拟机环境：

虚拟化平台：Oracle VM VirtualBox 7.x

操作系统：Ubuntu 22.04 LTS

Snort 版本：2.9

网络适配器配置：

- 网卡 1：桥接网络（Bridged Adapter）enp0s3
- 网卡 2：Host-Only 网络（用于与宿主机通信） enp0s8

四、 主要操作步骤及实验结果记录

1、数据包捕捉。使用 scapy 进行抓包并解析不同协议层，使用 ip 和端口二元组定义不同 session 编号，区分不同来源的数据包。

```
def packet_handler(pkt):
    if not (pkt.haslayer(IP) and pkt.haslayer(TCP) and pkt.haslayer(Raw)):
        return

    ip = pkt[IP]
    tcp = pkt[TCP]

    if tcp.sport != 23 and tcp.dport != 23:
        return
    try:
        payload = pkt[Raw].load.decode(errors="ignore")
    except:
        return

    session_key = (ip.src, tcp.sport)
    if session_key not in sessions:
        sessions[session_key] = {
            "stage": "INIT",
            "attempt_alerted": False,
            "success_alerted": False,
            "login_time": None
        }
        log("SESSION", f"新 Telnet 会话 {session_key}")
    session = sessions[session_key]
    stage = session["stage"]
    payload_l = payload.lower()

    log("PKT", f"{ip.src}:{tcp.sport} -> {ip.dst}:23 | stage={stage} | payload={repr(payload[:60])}")
```

2、规则匹配与报警模块，使用 23 端口和 login 后 root 负载匹配有远程 telnet root 登录，并设计等待窗口期 20s，在收到登录请求后该会话等待 20s，防止对同一来源短期重复发送登录尝试和成功登录警告邮件，细化交互体验。

```
    if stage in ("INIT", "LOGIN_ATTEMPT", "ROOT", "PASS") and not session["success_alerted"]:
        if "root@" in payload or "# " in payload or "welcome" in payload_l or "last login" in payload_l:
            session["success_alerted"] = True
            session["stage"] = "DONE"
            log("ALERT", f"root 用户 Telnet 登录成功 {session_key}")
            alert_msg = f"""
【来自外部主机的Telnet root 用户登录成功警报】

时间: {datetime.datetime.now()}
源 IP: {ip.src}
目标 IP: {ip.dst}
源端口: {tcp.sport}
目标端口: 23

Payload:
{repr(payload)}
"""
            send_email_alert("🚨 有来自计算机外部的成功root登录", alert_msg)
            return  # 成功告警发送后，不再发送尝试告警

    if "login:" in payload_l and not session["attempt_alerted"]:
        session["stage"] = "LOGIN_ATTEMPT"
        session["login_time"] = time.time()  # 记录首次尝试时间
        log("STATE", f"{session_key} 进入登录尝试阶段，等待 {LOGIN_GRACE_PERIOD}s 确认是否成功")
```

3、邮件发送模块。使用 email 模块，着重增加了错误输出文本，便于使用时定位错误。

```
def send_email_alert(subject, content):
    log("MAIL", f"准备发送邮件 | Subject='{subject}'")

    try:
        msg = MIMEText(content, "plain", "utf-8")
        msg["From"] = f"Telnet-NIDS <{EMAIL_USER}>"
        msg["To"] = EMAIL_TO
        msg["Subject"] = Header(subject, "utf-8")
    except Exception:
        log("ERROR", "构造邮件内容失败")
        traceback.print_exc()
        return

    try:
        server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT, timeout=10)
        server.starttls()
        server.login(EMAIL_USER, EMAIL_PASS)
        server.sendmail(EMAIL_USER, EMAIL_TO, msg.as_string())
        log("MAIL", "邮件发送成功")
    except Exception:
        log("ERROR", "SMTP 发送邮件失败")
        traceback.print_exc()
    finally:
        try:
            server.quit()
        except:
            pass
```

五、 实验分析总结及心得

遇到和解决了很多困难：在上个实验 snort 规则理解与使用的基础上增加了邮件报警功能，在如何具体设计报警逻辑上需要斟酌，例如监测到登录尝试与被成功登录的情境安全威胁不同，是否应该在邮件内容上有所区分，又如何根据网络抓包情况做出规则适配上的区分，

也有所收获，邮件报警功能具有充分的现实实用性，可以继续复用在其他代码如模型训练等长期待机场景中，很有启发。

实验二 实现简单 HIDS

一、 实验目的

1. 理解 HIDS 的基本概念、工作机制及其在主机安全防护体系中的作用；
2. 掌握在主机环境中对进程资源占用情况进行监控的方法，能够获取并展示各应用程序的内存使用情况；
3. 学习对主机中应用程序网络行为进行统计分析，能够在指定时间窗口内（如 10 分钟）统计各进程的网络流量使用情况；
4. 实现对关键文件读写行为的监控，理解文件访问行为在主机入侵检测中的重要性；
5. 了解系统启动进程的组成与启动流程，能够获取并展示系统中的启动进程信息；
6. 通过综合实验，提高对主机异常行为分析与安全事件发现能力，为后续深入研究主机安全防护技术奠定基础。

二、 实验原理

本实验围绕主机层面的资源使用、系统行为和关键对象访问情况，实现一个基础的 HIDS 功能原型。

在操作系统层面，进程、文件系统和网络通信均由内核统一管理。HIDS 通过调用操作系统提供的接口获取主机中各类运行信息，实时感知系统状态。

首先，通过对当前运行进程的监控，系统能够获取各应用程序的内存占用情况，从而分析进程资源使用是否存在异常。当某一进程出现持续、异常的高内存占用时，可能表明该进程存在恶意行为或运行异常。

其次，主机中的所有网络通信最终均由具体进程发起。通过将进程信息与网络连接状态进行关联，HIDS 可以在设定的时间窗口内（如 10 分钟）统计各应用程序的网络流量使用情况。对进程网络行为的持续统计与分析，有助于发现异常的网络访问模式，例如隐蔽的外联通信或异常的数据传输行为。

在文件系统层面，关键文件的读写行为直接反映系统的安全状态。HIDS 对指定文件的访问操作进行监控，记录其读写行为变化。当重要配置文件或敏感文件被非预期进程修改或频繁访问时，系统可据此判断可能存在入侵或篡改风险。

此外，系统启动过程中的自启动进程和服务是攻击者实现长期驻留的重要途径。通过获取并展示系统中的启动进程信息，HIDS 能够对系统启动项进行审计，及时发现异常或未知的启动进程，从而增强对持久化攻击的检测能力。

三、 实验环境

操作系统：ubuntu22.04LTS

python3.11

四、 主要操作步骤及实验结果记录

1、应用程序内存占用情况监控。使用后台线程定期调用 psutil 遍历所有进程并累计每个进程名的常驻内存计算占比，结果以字典形式保存在 self.aggregated 中，供外部通过 get_top_memory() 读取。

```python
class ProcessMonitor:
    def __init__(self, sample_interval=2):
        self.sample_interval = sample_interval
        self._running = False
        self._thread = None
        self._lock = threading.Lock()
        # aggregated by process name => {'mem_rss': int, 'mem_percent': float, 'pids':
        self.aggregated = {}

    def _sample(self):
        # 后台采样循环：只要 self._running 为 True 就不断采样
        while self._running:
            by_name = defaultdict(lambda: {'mem_rss': 0, 'mem_percent': 0.0, 'pids': |
            for p in psutil.process_iter(['pid', 'name', 'memory_info', 'memory_percer
                try:
                    info = p.info
                    name = info.get('name') or str(info.get('pid'))
                    mem = info.get('memory_info')
                    rss = mem.rss if mem else 0
                    by_name[name]['mem_rss'] += rss
                    by_name[name]['mem_percent'] += (info.get('memory_percent') or 0.0
                    by_name[name]['pids'].append(info.get('pid'))
                except Exception:
                    continue
```

2、应用程序一段时间内网络流量监控。使用 scapy 抓包，将观察到的数据包尝试归因到系统中的某个进程（pid），并按时间序列记录（timestamp, pid, bytes），以便后续按时间窗口统计各进程流量。

```python
class NetMonitor:
    def __init__(self):
        self._running = False
        self._thread = None
        # deque of (ts, pid, bytes)
        self.records = deque()
        self._lock = threading.Lock()
        # cached maps
        self._inode_pid = {}
        self._last_map_time = 0

    def start(self):
        if sniff is None:
            print('scapy is not available; network monitoring disabled')
            return
        if self._running:
            return
        self._running = True
        self._thread = threading.Thread(target=self._run_sniff, daemon=True)
        self._thread.start()
```

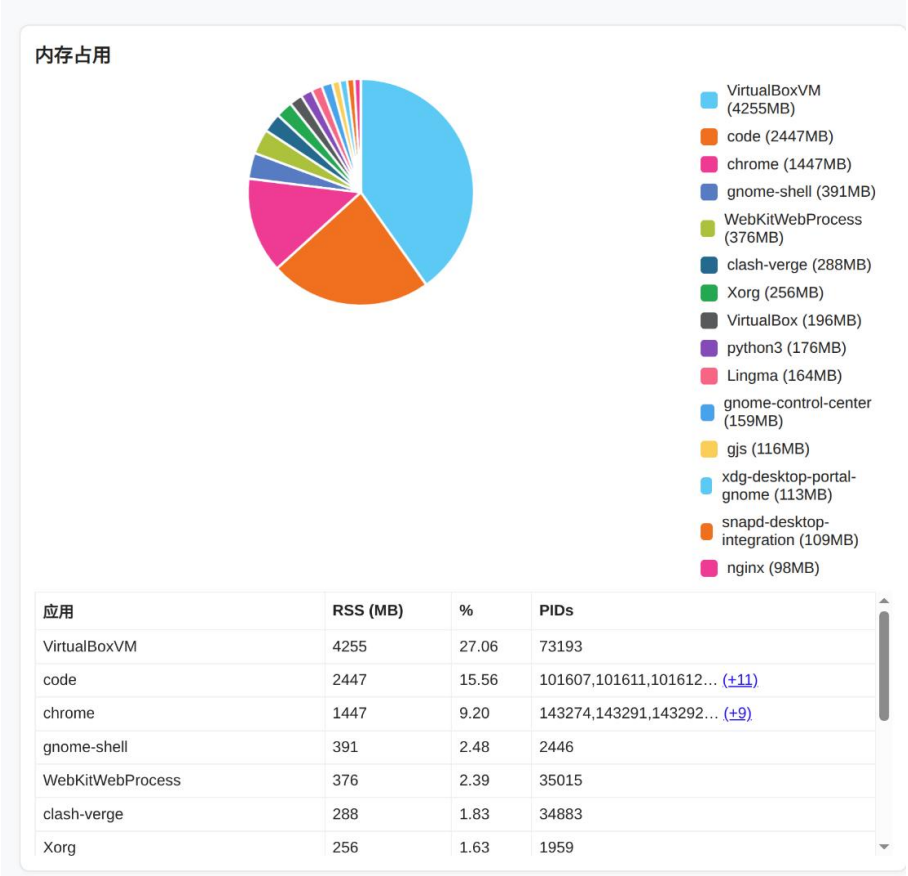
3、某路径下文件读写情况监控。使用 watchdog（基于 inotify）监控指定文件或目录，将发生的事件记录到内存列表中供外部读取

```
class _Handler(FileSystemEventHandler):
    def __init__(self, event_list, path):
        super().__init__()
        # 事件将被追加到外部传入的 list 对象
        self.event_list = event_list
        self.path = path

    def on_any_event(self, event):
        # 当发生任何文件系统事件时，记录简要信息到事件列表中
        # 记录字段：time（时间戳），event_type（字符串），is_directory（bo
        try:
            self.event_list.append({'time': time.time(), 'event_ty
        except Exception:
            # 保持鲁棒性：监控不应因为个别事件导致崩溃
            pass


class FileMonitor:
    def __init__(self, path_to_watch=None):
        # 默认监控路径；可在运行时通过 set_path 修改
        self.path = path_to_watch or '/tmp/monitored_file'
        # 存储事件的列表（append-only），外部通过 get_events 读取
        self.events = []
        self.observer = None
        self._lock = threading.Lock()
```

4、系统进程启动情况监控，直接调用 systemctl 查看当前各项服务状态。

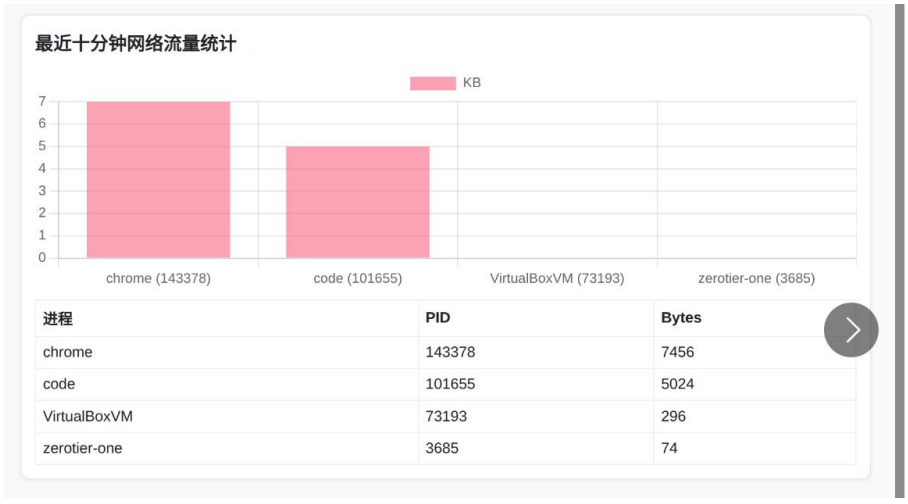
```
def api_startup():
    # gather systemd enabled services and PPID=1 processes
    services = []
    try:
        out = subprocess.check_output(['systemctl', 'list-unit-files', '--state=enable
        for line in out.splitlines():
            parts = line.split()
            if not parts:
                continue
            name = parts[0]
            try:
                active = subprocess.check_output(['systemctl', 'is-active', name], te
            except Exception:
                active = 'unknown'
            services.append({'name': name, 'active': active})
    except Exception:
        # systemd might not be present; fallback empty
        services = []

    ppid1 = []
```

5、前端仪表盘展示设计。在 html 页面上引入了简单的响应式网格和图表渲染，设计了四个主要卡片以展示四项信息。



应用	RSS (MB)	%	PIDs
VirtualBoxVM	4255	27.06	73193
code	2447	15.56	101607,101611,101612… (+11)
chrome	1447	9.20	143274,143291,143292… (+9)
gnome-shell	391	2.48	2446
WebKitWebProcess	376	2.39	35015
clash-verge	288	1.83	34883
Xorg	256	1.63	1959

图 8 使用饼状图展现内存占用，对具体信息设计了折叠



进程	PID	Bytes
chrome	143378	7456
code	101655	5024
VirtualBoxVM	73193	296
zerotier-one	3685	74

图 9 使用柱状图展现流量统计

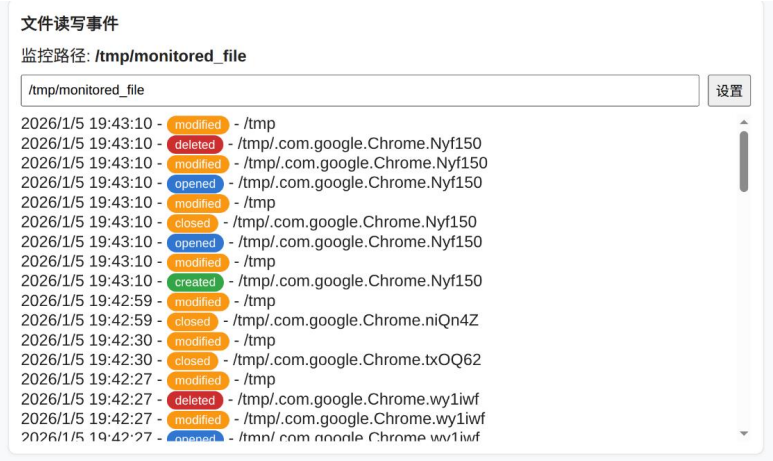


图 10 文件读写事件监控模块，可选择更改具体监控路径，设计了不同的色彩标签对应不同事件

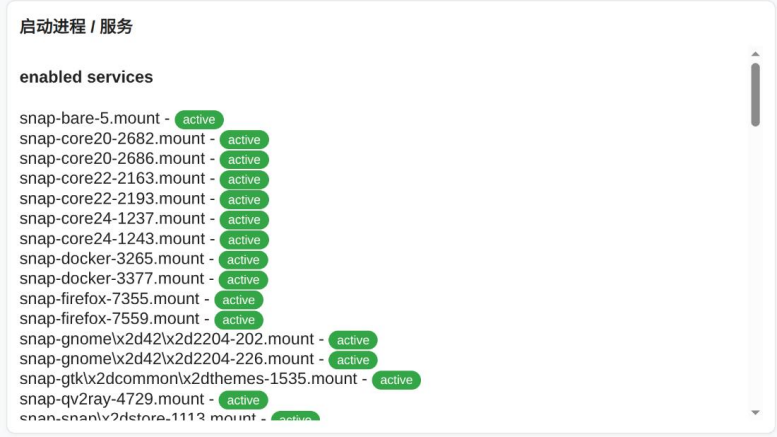
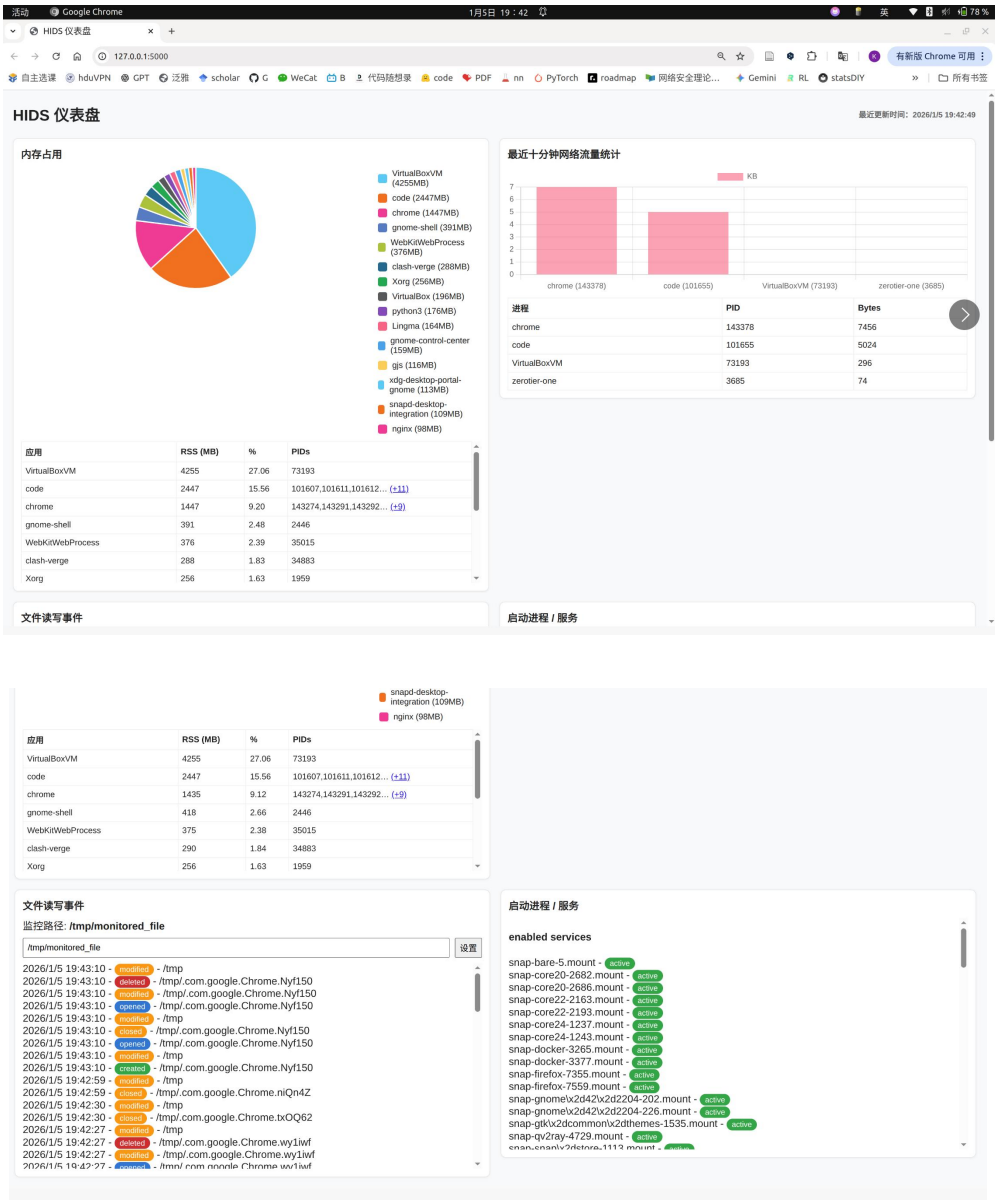


图 11 服务启动状态监控模块

6、整体页面表现



五、 实验分析总结及心得

遇到和解决了很多困难：后端监听、记录、汇总数据主要靠调包和系统 API，在如何视觉展现监听效果上有所斟酌，考虑过直接在终端输出文本数据，难度低但不够直观，于是上网参考了现有 HIDS 视觉页面，决定多使用饼状图和色彩标签展示状态，结合此前终端绘图惨痛经验，选择了成熟的前端 chart 组件进行网页渲染，此后便是视觉表现和使用体验的细节优化，例如模块在网页的排版和自适应、如何合并和折叠过多同类细节项、增加最新更新时间以带来实时效用感受等。

也有所收获：熟悉了系统底层进程/事件等收集调用，意识到结合进程资源使用、网络行为和文件操作等多种信息能更有效地发现异常行为，体会到顺畅的前端视觉交互设计在技术实现上的复杂性，