0.1 四大核心保护机制概览

保护机制	全称	作用	开启方式（编译）
RELRO	Relocation Read-Only	使 *GOT*[1]表只读，防 GOT 覆写	`-Wl,-z,relro`（Partial） `-Wl,-z,relro,-z,now`（Full）
Stack *Canary*[2]	Stack Guard / Cookie	防栈溢出覆盖返回地址	`-fstack-protector` 系列
NX[3]	No-eXecute (DEP)	禁止执行栈/堆上的代码	默认开启（现代系统）
PIE[4]	Position Independent Executable	使程序基地址随机化（配合 ASLR）	`-pie -fPIE`

Table 1

[1]GOT（Global Offset Table，全局偏移表）是 Linux ELF（Executable and Linkable Format）程序中一个非常关键的数据结构，尤其在动态链接和 Pwn 利用中扮演核心角色，程序调用外部函数（如 `printf`, `puts`）时，会通过 GOT 表查找真实地址。GOT 表在程序运行时是可以被修改的（为了支持动态链接）
[2]想象你在栈上放了一个“警报器”（canary）。如果发生缓冲区溢出，攻击者要覆盖返回地址，就必须先覆盖这个 canary。程序在函数返回前会检查它是否被改动，如果变了就直接 crash，阻止攻击。
[3]NX（No-eXecute） 或 DEP（Data Execution Prevention） 是现代操作系统中一项非常重要的内存安全保护机制。它的核心思想很简单，但对漏洞利用（尤其是 Pwn）影响巨大
NX（No-eXecute） = 禁止在“数据区域”（如栈、堆）执行代码；即可以读写栈和堆（比如放字符串、数组），但不能把栈或堆上的数据当作指令来运行
[4]PIE（Position Independent Executable，位置无关可执行文件） 是现代 Linux 系统中一项重要的安全机制，主要用于配合 ASLR（Address Space Layout Randomization，地址空间布局随机化） 来增加漏洞利用的难度。

1. 1. RELRO（Relocation Read-Only)

1.1 原理

- 程序调用外部函数（如 `printf`）时，通过 **GOT** 动态解析地址。
- RELRO 控制 GOT 表是否在程序启动后变为**只读**。

1.2 两种级别

类型	行为	安全性
No RELRO	GOT 全程可写	极不安全
Partial RELRO	`.got` 只读，`.got.plt` 仍可写	中等（GOT 可被劫持）
Full RELRO	整个 GOT 启动后立即只读	安全

Table 2

1.3 CTF 利用点（Partial RELRO）

- **GOT Hijacking**：覆盖 `.got.plt` 中某函数指针（如 `puts`）为 `system` 地址。
- 示例：

```
# 假设 puts@got = 0x404020，system_addr = 0x401230
payload = overwrite(0x404020, p64(system_addr))
```

1.4 如何检查?

```
checksec ./fileName
# 输出：RELRO: Partial RELRO
```

2. 2. Stack Canary（栈金丝雀）

2.1 原理

- 在函数栈帧中插入一个随机值（canary），位于局部变量和返回地址之间。
- 函数返回前检查 canary 是否被修改 → 若被覆盖（溢出），程序 abort。

2.2 开启条件

- 编译时加：
  - `-fstack-protector`：仅保护含 `char[8+]` 的函数
  - `-fstack-protector-strong`：更广泛保护
  - `-fstack-protector-all`：所有函数

2.3 CTF 影响

状态	攻击难度
No Canary	⭐ 极易：直接覆盖返回地址
Canary Enabled	⭐ ⭐ ⭐ 需先泄露 canary 值（如格式化字符串漏洞）

Table 3

2.4 绕过思路（若存在其他漏洞）

- 格式化字符串 → 泄露 canary
- 信息泄露 → 读取栈上 canary 值
- 然后再进行栈溢出

3. 3. NX（No-eXecute） / DEP

3.1 原理

- 标记内存页属性：**数据段不可执行**（栈、堆为 RW，代码段为 RX）。
- 阻止攻击者在栈上部署并执行 shellcode。

3.2 CTF 影响

- **不能直接执行 shellcode**
- 必须使用ROP（Return-Oriented Programming）：
  - 利用程序中已有的代码片段（gadgets）
  - 拼接成恶意逻辑（如调用 `system("/bin/sh")`）

3.3 工具推荐

```
# 查找 gadgets
ROPgadget --binary ./vuln --only "pop|ret"
```

3.4 小知识

- 若 NX 关闭（罕见）：可直接写 shellcode 到栈，跳转执行（经典 exploit）

4. 4. PIE（Position Independent Executable）

4.1 原理

- 使主程序像共享库一样加载，**基地址每次运行随机化**（需 ASLR 支持）。
- 所有函数、字符串、gadgets 地址都不固定。

4.2 CTF 影响

状态	攻击难度
No PIE	⭐ 地址固定，ROP 直接硬编码
PIE Enabled	⭐⭐⭐ 需先泄露某个地址（如 `__libc_start_main`）→ 计算基址

Table 4

4.3 绕过思路（PIE 开启时）

1. 利用 `puts(puts@got)` 泄露 libc 地址
2. 泄露程序自身地址（如返回地址低字节）
3. 计算程序/ libc 基地址 → 得到所有 gadget 真实地址

> 提示：即使 PIE 开启，**libc 本身也是 PIE 的**，所以通常要同时泄露 libc 和程序地址。

5. 实战判断流程图（拿到题目后）

```
1. checksec ./vuln
   ↓
2. 是否有 Canary?
   ├─ 有 → 需找信息泄露（如格式化字符串）
   └─ 无 → 可直接栈溢出
   ↓
3. NX 是否开启？
   ├─ 关 → 写 shellcode 到栈
   └─ 开 → 准备 ROP
   ↓
4. PIE 是否开启？
   ├─ 开 → 需地址泄露（leak）再 ROP
   └─ 关 → 直接硬编码 gadget 地址
   ↓
5. RELRO 状态？
   ├─ Full → 不能改 GOT
   └─ Partial/No → 可考虑 GOT hijacking（备用方案）
```

6. 常用命令 & 工具

```
# 查看保护
checksec ./vuln

# 反汇编
objdump -d ./vuln
readelf -s ./vuln      # 查符号表
readelf -r ./vuln      # 查重定位表（GOT 相关）

# 找 gadgets
ROPgadget --binary ./vuln

# 调试
gdb ./vuln
```

```
gef           # 推荐 GDB 插件
pwndbg        # 另一个强大插件

# 写 exp(Python)
from pwn import *
```

7. 学习路线建议

1. **阶段一：无保护栈溢出**
   - 题目特征：No Canary, No NX, No PIE
   - 目标：覆盖返回地址 → 跳转到 shellcode
2. **阶段二：NX 开启，无 PIE**
   - 题目特征：NX enabled, No PIE, No Canary
   - 目标：ROP 调用 `system("/bin/sh")`
3. **阶段三：加入 Canary 或 PIE**
   - 学会信息泄露（leak）
   - 结合格式化字符串 or puts 泄露地址
4. **阶段四：Full RELRO + Canary + NX + PIE**
   - 综合利用：leak + ROP + ret2libc / SROP