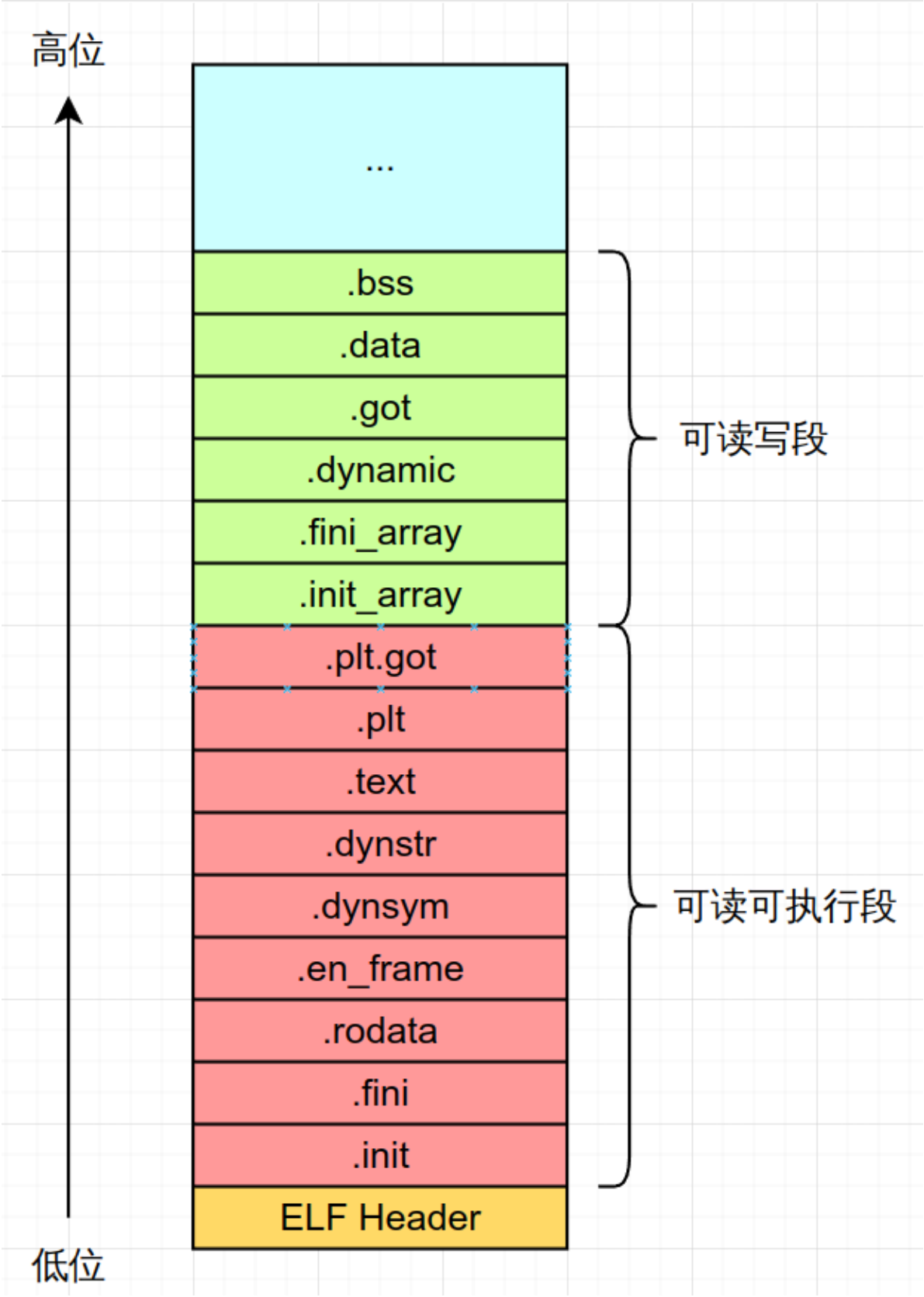
栈溢出入门以及ret2text

以下内容会涉及到一些汇编的概念

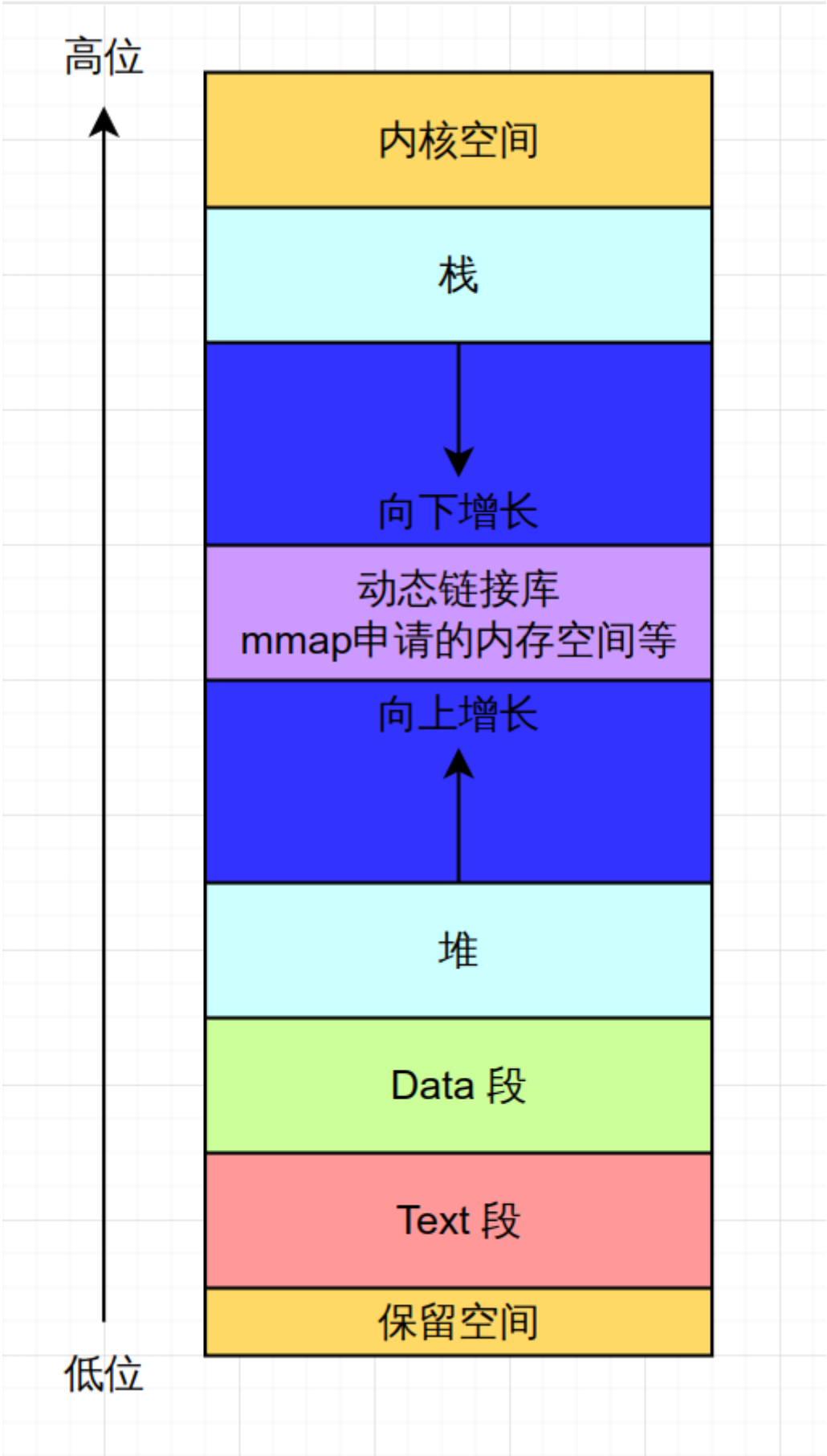
可执行ELF文件结构与Linux系统的虚拟地址空间

可执行ELF文件结构的简单介绍

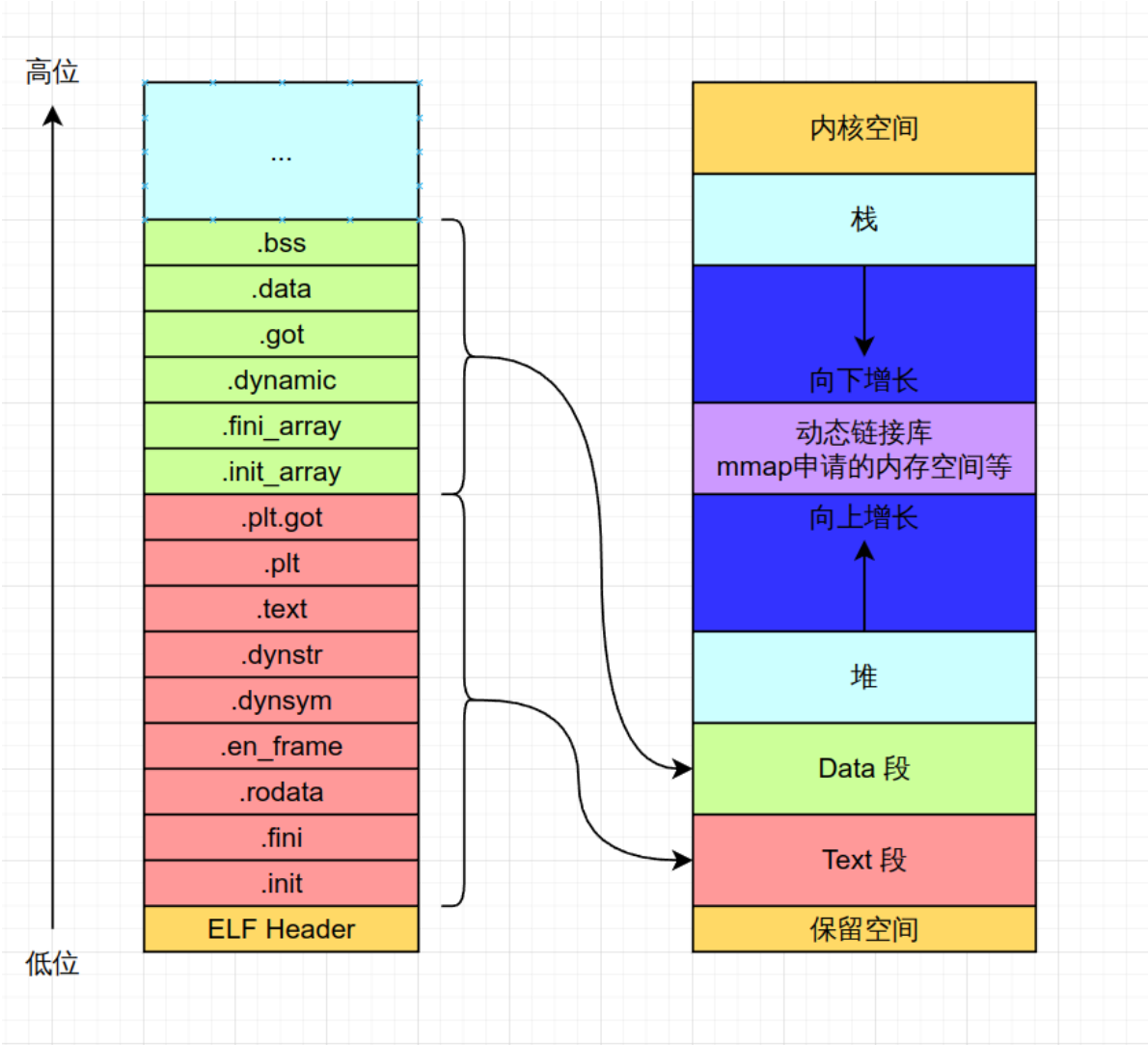
可执行ELF文件的基本结构如下图，里面主要分为两个段，一个段是只读可执行的，记录了程序的常量以及程序的编译后的代码；另一个段是可读写，但是不可执行的，记录了保证程序正常运行的一些自动生成的变量，以及程序的全局变量。



为了多进程同时运行时，进程之间互不影响，同时简化内存管理，我们引入了虚拟地址空间，在用一些方法将虚拟地址空间映射到物理内存，具体的实现方法之类的在这里就先不过多的讲解，目前只需要知道对于用户态程序来说，每一个进程运行在虚拟地址空间中，所操作的内存的下标都是从0开始，后文提到的内存都指代虚拟地址空间。下图是Linux虚拟内存空间结构（没有相关内存保护的情况下)。



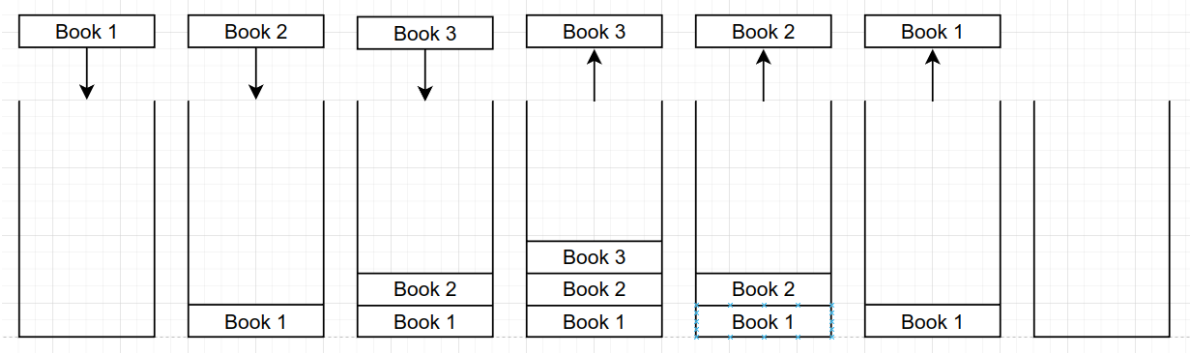
当程序运行时，ELF中的数据会被映射到内存中。映射关系如下图，可执行的部分被映射到Text段，不可执行的部分被映射到Data段。



栈溢出

简单的介绍一下栈空间

首先，栈是一种先进后出的结构。可以理解为有一叠长宽相等书堆放在一个没有顶盖的，长宽合适的容器中，先放进去的书会被压在底下，后放进去的书会在上面。当需要取出时，下面的书需要等上面的书取走后才能拿出。就如下图所示



在虚拟地址空间中，栈由数个栈帧组成，栈帧记录了函数的局部变量，返回地址等信息，每调用一次函数就会将一个栈帧压栈，函数返回是出栈，栈空间由编译器管理，在编译时就完成了分配。栈空间一般位于地址的高位，且栈底在高地址一侧，栈帧从高地址向低地址增加。

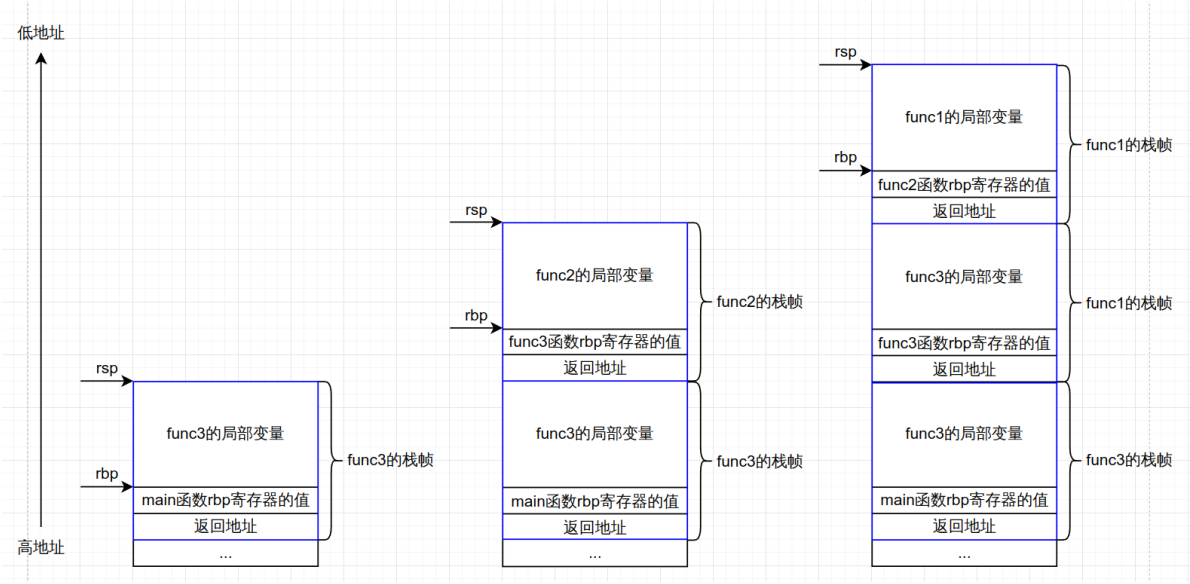
可以看下面这一个程序的例子

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

```c
void func1() {
    unsigned long a = 1;
    unsigned long b = 2;
    unsigned long c = 3;
    printf("func1 called")
}

void func2() {
    unsigned long a = 4;
    unsigned long b = 5;
    unsigned long c = 6;
    func1();
}

void func3() {
    unsigned long a = 7;
    unsigned long b = 8;
    unsigned long c = 9;
    func2();
}

int main() {
    func3();

    return 0;
}
```

因函数实现结构差不多，故选取`func3`编译产生的汇编代码为例，可以看到开头有`push rbp`，`mov rbp, rsp`，`sub rsp, 0x20`几行指令，这几行指令就是用于初始化栈帧的。

上面代码运行到func1时的栈空间变化如下（从func3开始，且为方便理解，将上方视为低地址）。从左往右可以视为函数调用的变化，从右往左则是返回时的变化。



栈溢出的原理

对于c语言来说，它运行时并不知道数组的大小，也不会检查数组的下标是否越界，仅仅在运行时判断一下目标内存的权限（读，写，执行等），而栈是可以读写的。所以说，如果在栈帧的局部变量的范围中存在一个数组，我们就可以使数组的下标越界，从而访问甚至修改一些不应该被访问或修改的值。

但是通过程序的输入直接操作数组下标的机会并不多，但是有一个东西比较的特殊——字符串。很多时候字符串可以理解为一个字符数组，又因为要输入的字符串的长度往往无法明确，有些输入函数就对输入长度不加以限制（如`gets`函数），或是默认不加限制（如`scanf`函数的`%s`占位符），又或是编写程序时，设置的读取长度过长（如Linux系统独有的`read`函数），你给多少他们，他们就会读多少，或是一直读到读取限制的长度为止。这时就有可能覆盖掉其他变量，甚至一些其他的存在栈帧中的内容，就造成了栈溢出。

ret2text

原理

在汇编层面，调用函数就是`call xxxx`，而`call`可以等价为`push rip+5; jmp xxxx`，前面被压进栈中的就是返回地址，即当被调用函数结束时，程序继续执行的位置。同时，汇编中的函数结尾有一个`ret`指令，常常与`call`成对出现，而`ret`可以等价为`pop rip`，使返回地址出栈，进入rip寄存器，从而使程序继续运行。

另外，数组存储的方向是低位向高位，字符串也是一样，那么如果不对字符串输入加以限制，就有可能覆盖掉本栈帧的返回地址，只要将返回地址控制为一个合适的值，我们就可以劫持程序流。

ret2text的利用方法就是将返回地址修改为一个text段中的函数（通常为一个后门函数）

利用实例

就比如下面这个程序，模拟了栈溢出，将返回地址修改为`backdoor`函数，从而使程序返回到了`backdoor`函数上。

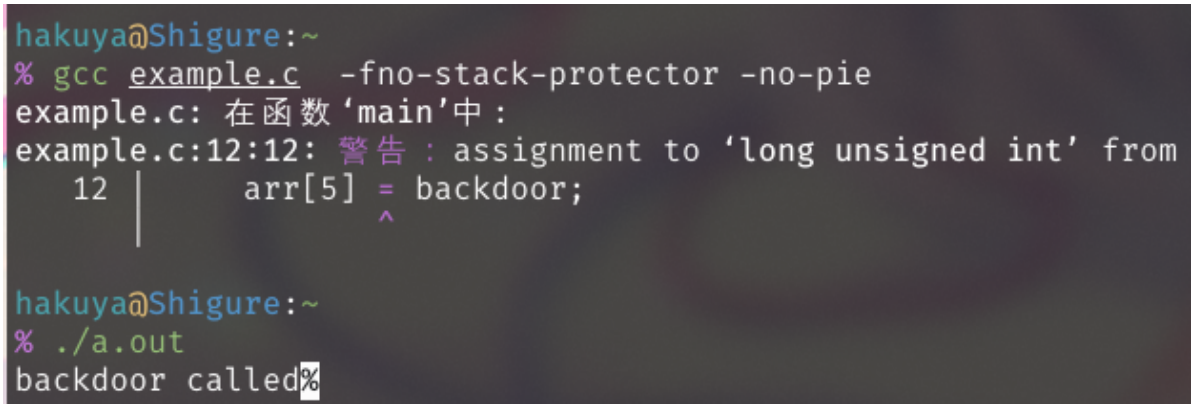
```
#include <stdio.h>
#include <stdlib.h>

void backdoor() {
    printf("backdoor called");
    exit(0);
}
```

```c
int main() {
    unsigned long arr[3];

    arr[5] = backdoor;

    return 0;
}
```

编译命令（后面两个flag是用来关闭一些内存保护）：

```
gcc example.c  -fno-stack-protector -no-pie
```

运行效果：



写脚本过程中可能会用到的东西

主要就是pwntools的`p64()`和`p32()`两个函数，`p64`能够将传入的整数转换为byte类型的字符串，并对其为64位，`p32`同理，但是对齐为32位。注意，`p32(0x1234)`并不是说将`0x1234`变为`b'4660'`，而是转换为`b'\x34\x12\x00\x00'`（以大端序为例）。另外，用这两个函数时，很多文本编辑器的静态语法检查会出现错误，说找不到该函数，这个报错可以直接无视。