以 Demo 程序为例的解题思路参考和标准答案

注意：Demo 程序只包含一个恶意代码功能和一个漏洞，Demo 程序只作为学习理解题目使用，课设要求的还原样本为和 demo 有差别。

任务一：使用 IDA Pro 或者 Ghirda 等二进制逆向分析工具手动分析样本软件中的恶意行为和漏洞，要求至少找到存在漏洞的函数名称、地址、漏洞类型、漏洞成因以及存在恶意功能的函数名称、地址和恶意功能的具体行为危害信息。

任务一解题思路：

如指导书所描述的，无论是漏洞还是恶意代码功能，在本课设中均假设其需要依赖系统调用函数完成。因此，核心思路是 1.检测程序中敏感系统调用函数的使用 2.对这些函数的参数进行分析，通过参数分析判断其是否确认是恶意功能或者是漏洞。

步骤 1.系统调用函数识别

漏洞函数识别：

缓冲区溢出漏洞，如指导书描述的缓冲区溢出漏洞是使用拷贝函数但没有对拷贝数据长度进行正确检查导致的。

因此，以 Demo 程序为例，利用 IDA Pro 逆向还原程序以后很容易发现 vulnerable_function 中使用了 strcpy 函数的，如图 1 所示。

```
1 char *__cdecl vulnerable_function(char *src)
2 {
3   char dest[14]; // [esp+6h] [ebp-12h] BYREF
4
5   return strcpy(dest, src);
6 }
```

图 1

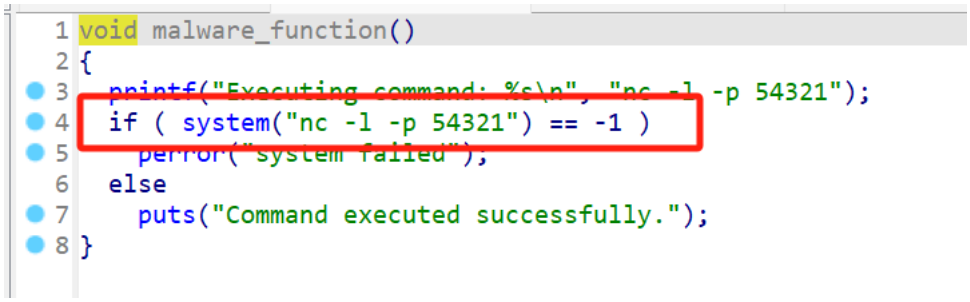
并且通过反汇编代码可以得出 strcpy 被调用的地址是 0x1435，如下图 2 所示。

```
.text:00001414                          ; char *__cdecl vulnerable_function(char *src)
.text:00001414                          public vulnerable_function
.text:00001414                          vulnerable_function proc near           ; CODE XREF: main
.text:00001414
.text:00001414                          dest= byte ptr -12h
.text:00001414                          var_4= dword ptr -4
.text:00001414                          src= dword ptr  8
.text:00001414
.text:00001414                          ; __unwind {
.text:00001414 F3 0F 1E FB              endbr32
.text:00001418 55                       push    ebp
.text:00001419 89 E5                    mov     ebp, esp
.text:0000141B 53                       push    ebx
.text:0000141C 83 EC 14                 sub     esp, 14h
.text:0000141F E8 F7 02 00 00           call    __x86_get_pc_thunk_ax
.text:0000141F
.text:00001424 05 78 2B 00 00           add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:00001429 83 EC 08                 sub     esp, 8
.text:0000142C FF 75 08                 push    [ebp+src]                       ; src
.text:0000142F 8D 55 EE                 lea     edx, [ebp+dest]
.text:00001432 52                       push    edx                             ; dest
.text:00001433 89 C3                    mov     ebx, eax
.text:00001435 E8 76 FD FF FF           call    _strcpy
.text:00001435
.text:0000143A 83 C4 10                 add     esp, 10h
.text:0000143D 90                       nop
.text:0000143E 8B 5D FC                 mov     ebx, [ebp+var_4]
.text:00001441 C9                       leave
.text:00001442 C3                       retn
.text:00001442                          ; } // starts at 1414
.text:00001442
```

图 2

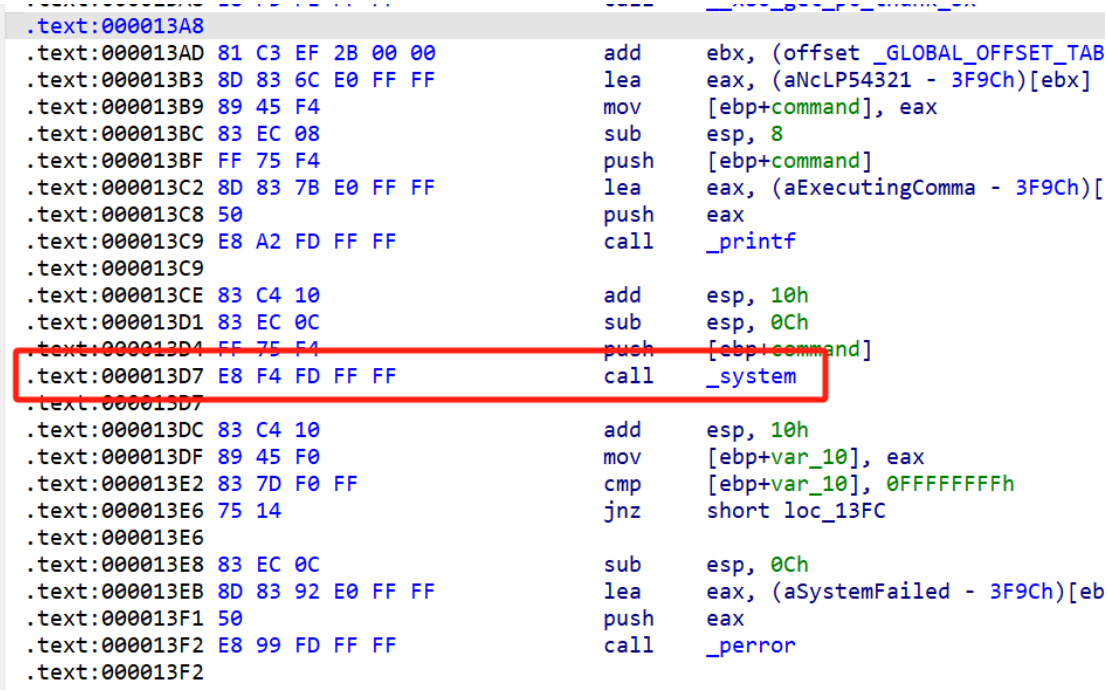
其他漏洞函数也是类型，比如堆相关漏洞，要首先识别 free 和 malloc 函数的被调用位置。

恶意代码功能函数识别： 如指导书中所述，各种恶意代码功能也是借助系统功能函数来实现，以 demo 程序为例，我们很容易发现在 malware_function 中，调用了 system 系统函数，进而可以定位其被调用的位置是 0x13D7。其他恶意代码功能同样可以参考指导书找到响应的函数，如图 3 和图 4 所示。



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

图 3



```
.text:000013A8
.text:000013AD 81 C3 EF 2B 00 00        add     ebx, (offset _GLOBAL_OFFSET_TAB
.text:000013B3 8D 83 6C E0 FF FF        lea     eax, (aNcLP54321 - 3F9Ch)[ebx]
.text:000013B9 89 45 F4                 mov     [ebp+command], eax
.text:000013BC 83 EC 08                 sub     esp, 8
.text:000013BF FF 75 F4                 push    [ebp+command]
.text:000013C2 8D 83 7B E0 FF FF        lea     eax, (aExecutingComma - 3F9Ch)[
.text:000013C8 50                       push    eax
.text:000013C9 E8 A2 FD FF FF           call    _printf
.text:000013C9
.text:000013CE 83 C4 10                 add     esp, 10h
.text:000013D1 83 EC 0C                 sub     esp, 0Ch
.text:000013D4 FF 75 F4                 push    [ebp+command]
.text:000013D7 E8 F4 FD FF FF           call    _system
.text:000013D7
.text:000013DC 83 C4 10                 add     esp, 10h
.text:000013DF 89 45 F0                 mov     [ebp+var_10], eax
.text:000013E2 83 7D F0 FF              cmp     [ebp+var_10], 0FFFFFFFFh
.text:000013E6 75 14                    jnz     short loc_13FC
.text:000013E6
.text:000013E8 83 EC 0C                 sub     esp, 0Ch
.text:000013EB 8D 83 92 E0 FF FF        lea     eax, (aSystemFailed - 3F9Ch)[eb
.text:000013F1 50                       push    eax
.text:000013F2 E8 99 FD FF FF           call    _perror
.text:000013F2
```

图 4

步骤 2：参数分析

步骤 1 识别出了潜在的风险函数，但不代表每个这些函数调用就一定是漏洞或者是恶意功能，正常软件也可以正确使用这些函数，所以第二步要追踪分析这个函数参数的使用。以 Demo 程序为例，如图 5 中所示，利用 IDA 的 xref 可以得出 vulnerable_fucntion 函数在 main 函数中被调用，然后 strcpy 的第一个参数是在 vulerable_function 函数中定义的一个字符串数组大小为 14B 如图 1 所示，第二个参数是来自于被调用的 main 函数，然后我们进一步逆向追踪分析 main 函数中参数的赋值和传递（图 6），最后如图 7 所示找到其定义数组大小定义为为 100B，那么可以判断出来 100B 拷贝到 14B 会发生缓冲区溢出漏洞。

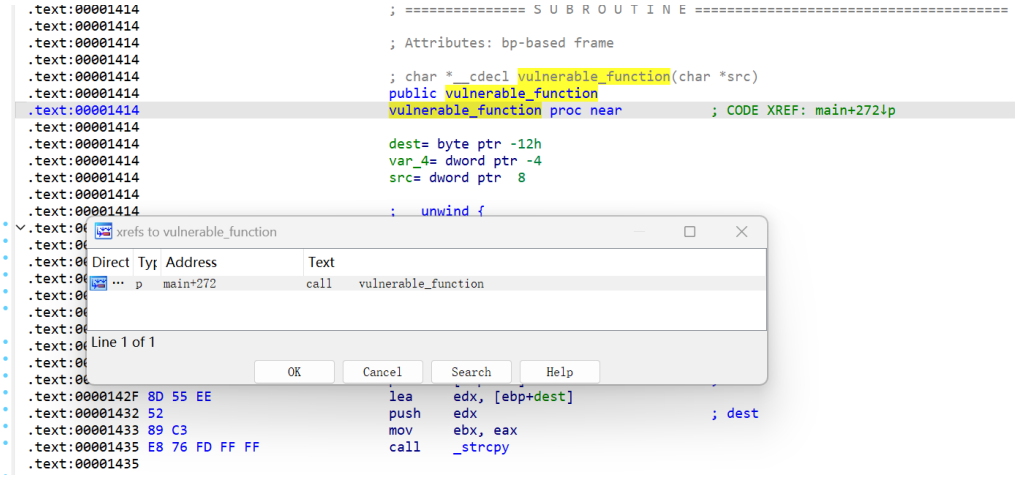


图 5

```
  while ( 1 )
  {
    memset(s, 0, sizeof(s));
    n = recv(v10, s, 0xFFu, 0);
    if ( (int)n <= 0 )
      break;
    s[n] = 0;
    printf("Received: %s\n", s);
    if ( (int)n > 99 )
      goto LABEL_18;
    memset(dest, 0, sizeof(dest));
    strncpy(dest, s, n);
    if ( dest[1] == 65 && dest[2] == 66 )
    {
      malware_function();
      send(v10, "mal test\n", 9u, 0);
    }
    else if ( dest[8] == 88 )
    {
      vulnerable_function(dest);
      send(v10, "vul test\n", 9u, 0);
    }
    else
    {
LABEL_18:
      send(v10, "Try Again\n", 0xAu, 0);
    }
  }
  puts("Client disconnected.");
  close(v10);
  close(fd);
  return 0;
}
```

图 6

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   char dest[100]; // [esp+0h] [ebp-19Ch] BYREF
4   char s[256]; // [esp+64h] [ebp-138h] BYREF
```

图 7

同理，分析恶意代码功能的参数一样，例如，demo 程序中使用了 system 函数，这个函

数参数 command 直接看汇编代码是从栈中提取的，然后追踪器定义来自如下图 8 的全局数据表中。具体参数内容为 nc -l -p 54321 由此可以判断该函数包含一个创建隐藏端口的恶意行为。

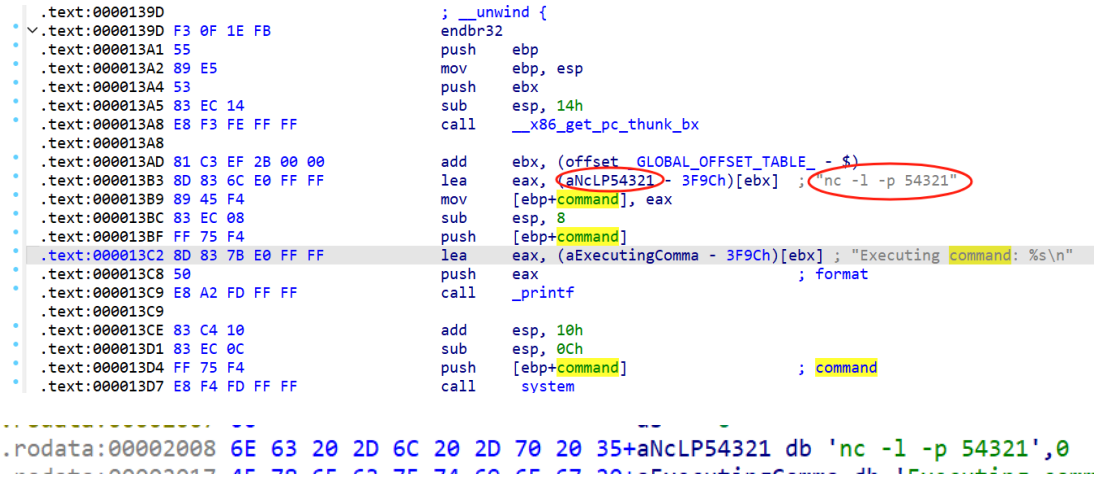


图 8

最终通过上两步骤的分析可以得出，DEmo 程序包含一个漏洞和一个恶意代码功能，具体需要输出的标准答案如下：

漏洞 1.

漏洞类型：缓冲区溢出/栈溢出 导致该漏洞函数名称：**strcpy**

该函数被调用的地址：0x1435

漏洞成因：使用 strcpy 字符拷贝时，没有做边界检查，目标数组大小只有 14B 但拷贝数据大小可以最多为 100B。

恶意代码功能 1.

功能类型：开启后门 使用的系统调用函数名称：system

该函数被调用的地址：0x13D7

具体功能描述：利用 system 函数调用 NC 指令创建额外监听端口 端口号为 54321

任务二：编写 IDA Pro 或者 Ghirda 等二进制逆向分析工具插件自动化检测样本软件中的恶意行为和漏洞（自动化输出（1）中的信息）

注意： 本次课设假设参数定义只会在敏感函数调用函数二层之内，例如，B 函数调用了 A 函数，然后 A 函数调用了敏感函数 X，对于敏感函数中所有参数的最原始定义范围可能存在于 A 函数以及调用 A 函数的 B 函数中，不考察再深层次的参数追溯。如果编写的脚步能够支持更深层次的参数追踪会酌情加分。

另外由于整数溢出漏洞需要检测所有运算指令并进行参数分析，难度较高，所以只需要任务一人工分析即可，不强制要求实现自动化分析，如果实现自动化分析会酌情加分

任务二解题思路：

根据任务一的人工解题思路，完成任务二。即第一步先确定潜在敏感函数有哪些？利用脚本自动化识别这些函数，然后定位这些函数的引用（xref） 第二步是关键也是难点，需要对这些关键函数中的参数进行追踪分析，找到其定义的内容，特别是对于堆漏洞比

如 UAF 和 DF，要跟踪指针参数的操作，从而实现自动化判断，最后要求自动化输出任务一答案内容则可判对。
这里先不给出具体脚本代码，请同学们参考思路自行完成，同学们也可以思考更好的检测方法。

任务三：使用 IDA Pro 或者 Ghirda 等二进制逆向分 的触发条件，并在基于 QEMU 的虚拟环境中进行动态验证。（人工分析即可不需要实 现工具自动化）
任务三结题思路：通过完成任务 1 和 2 以后其实已经锁定了漏洞和恶意代码的函数，那么进一步可以判断其调用的控制条件，例如 demo 程序中通过反编译我们可以看出当 dest 数组的第 2 个元素为 65 （'A'）且第 3 个元素为 66（'B'）时候，则必然会调用 malware_function 触发恶意代码功能。
同理，当 dest 数组第 9 个元素为 88（'X'）的时候则会调用 vulnerable_fucntion 函数触发漏洞。dest 数组内容通过分析可以得出就是来自于交互的 recv 端口的输入内容。

```
45  while ( 1 )
46  {
47    memset(s, 0, sizeof(s));
48    n = recv(v10, s, 0xFFu, 0);
49    if ( (int)n <= 0 )
50      break;
51    s[n] = 0;
52    printf("Received: %s\n", s);
53    if ( (int)n > 99 )
54      goto LABEL_18;
55    memset(dest, 0, sizeof(dest));
56    strncpy(dest, s, n);
57    if ( dest[1] == 65 && dest[2] == 66 )
58    {
59      malware_function();
60      send(v10, "mal test\n", 9u, 0);
61    }
62    else if ( dest[8] == 88 )
63    {
64      vulnerable_function(dest);
65      send(v10, "vul test\n", 9u, 0);
66    }
67    else
68    {
69LABEL_18:
70      send(v10, "Try Again\n", 0xAu, 0);
71    }
72  }
73  puts("Client disconnected.");
```

图 9

因此对于 Demo 程序任务三的标准答案为：（写 ascii 码字符或者数字都算对）
触发缓冲区漏洞的输入条件为：输入的第二个元素为字符 A(65)且第三个元素为 B（66）
触发开启端口的恶意代码功能条件为：输入的第九个元素为字符 X（88）

注：如果步骤三可以用脚本或者其他工具（自研）自动化得出也会考虑酌情加分