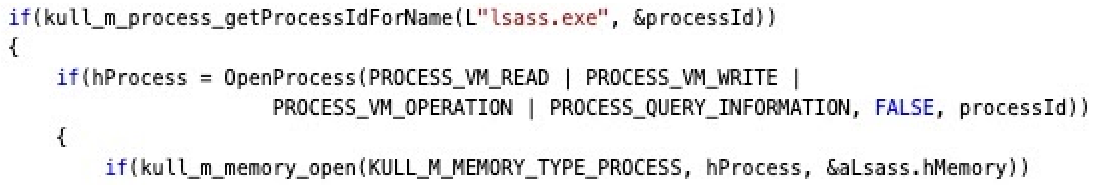
Memory Forensics R&D Illustrated: Detecting Mimikatz's Skeleton Key Attack

volatility-labs.blogspot.com/2021/10/memory-forensics-r-illustrated.html



```
if(kull_m_process_getProcessIdForName(L"lsass.exe", &processId))
{
    if(hProcess = OpenProcess(PROCESS_VM_READ | PROCESS_VM_WRITE |
                      PROCESS_VM_OPERATION | PROCESS_QUERY_INFORMATION, FALSE, processId))
    {
        if(kull_m_memory_open(KULL_M_MEMORY_TYPE_PROCESS, hProcess, &aLsass.hMemory))
```

In this blog post, we are going to walk you through the research and development process that leads to new and powerful memory analysis capabilities. We are often asked about what this workflow looks like, and how the abuse of an API by malware or a new code injection technique can be successfully uncovered by a Volatility plugin. To showcase this process, we are going to analyze the Skeleton Key feature of Mimikatz, and then develop a brand-new Volatility 3 plugin that can successfully detect this backdoor technique across memory samples. While Volexity Volcano customers have had this capability, we wanted to contribute this back to the Volatility community, since there was no publicly available plugin. This post will also reveal a number of entirely new features.

To reach this goal, we will first study the relevant Mimikatz source code; then we will reverse engineer the API that Mimikatz uses to locate its victim data structure; and then we will write a plugin that can replicate this search and look for signs of tampering. As you will see shortly, the new Skeleton Key detection plugin is fully documented and shows how to perform a wide range of tasks using the APIs of Volatility 3.

Our hope with this blog post is to inspire more members of the community to challenge themselves to develop their own new capabilities, and to experience what real-world malware and operating systems investigations entail. If you find this work interesting and decide to develop your own plugin(s), please consider submitting them to our 2021 Volatility Plugin Contest and take a chance at winning several prizes, including cash or a free spot in our popular Malware and Memory Forensics training.

Skeleton Key Background

The Skeleton Key technique was first detected in the wild by the DFIR team at SecureWorks. Their blog post walks through the steps taken by the malware sample they uncovered. They also worked with Microsoft on a follow-up paper about the attack type and its variations. The implementation in Mimikatz is very similar to the one described in their research.

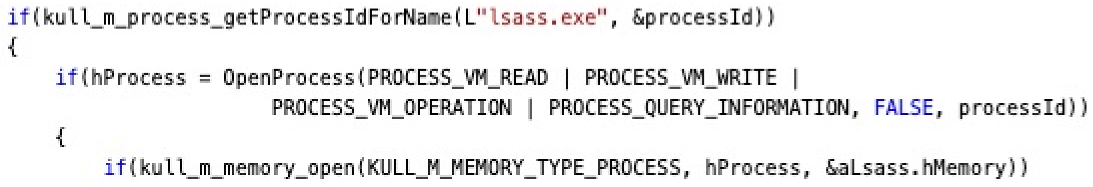
The idea behind the Skeleton Key technique is to backdoor the authentication subsystem of Windows Active Directory domain controllers. This is accomplished by injecting code into the running lsass.exe process, and then hooking the routines used when verifying a domain account's credentials. With the hooks in place, attackers are able to authenticate as any valid user in an AD domain by using a hard-coded password (termed the *Skeleton Key* by SecureWorks).

The ability of attackers to log in as any user makes several traditional incident remediation procedures largely ineffective. As an example, it is very common during incidents to temporarily disable accounts that attackers are/were using or to at least force a password reset of these accounts. When a Skeleton Key is active, these procedures are not helpful, since any account can be used. This problem is also compounded by the fact that all user accounts in a domain could potentially be abused by attackers, and a significant amount of log review (assuming logs are available) is necessary to trace the abuse of accounts and any associated lateral movement. This ability to authenticate as *any* user to *any* system is incredibly powerful and significantly expands the scope of DFIR engagements.

Analyzing the Skeleton Key Capability of Mimikatz

Activating the Skeleton Key attack of Mimikatz requires using its *misc::skeleton* command after running the usual *privilege::debug* command. There are many great blog posts that document this process by showing the related Mimikatz output and other related information, such as here, here, and here. Cycraft also documented malware from the Chimera APT group that used a significant amount of code from *misc::skeleton* to implement its own Skeleton Key attack. The end result of this command is a Skeleton Key attack being active on the system; the attacker is able to authenticate with the malware-controlled credentials.

Running the *misc::skeleton* command will lead to the *kuhl_m_misc_skeleton* function being called inside the active Mimikatz instance. This function is responsible for patching the needed code and data inside the lsass.exe process to make the Skeleton Key active. The first steps in this process are shown in the following image:

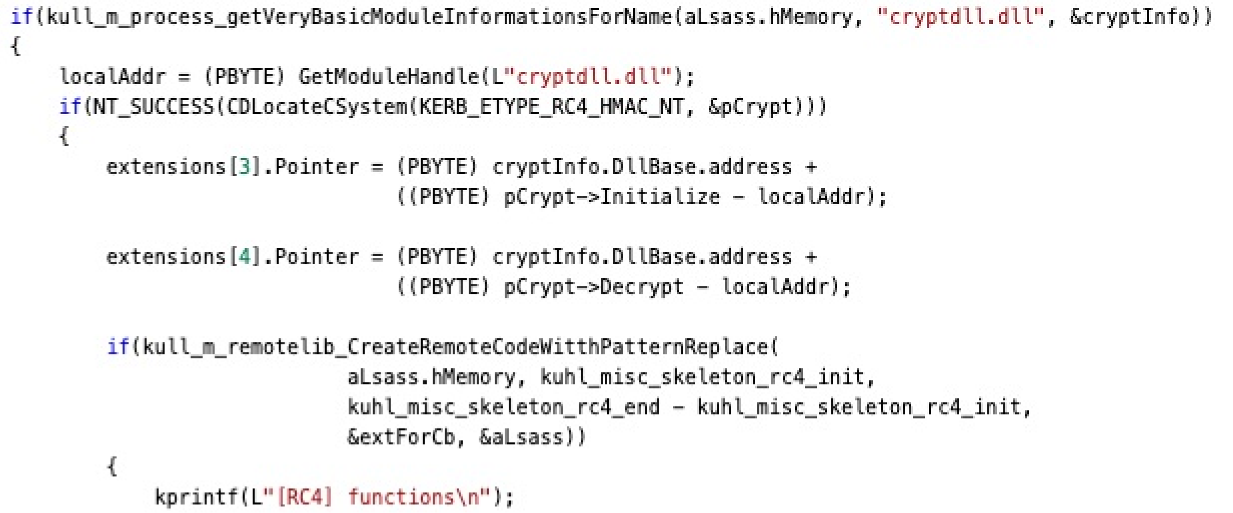


```
if(kull_m_process_getProcessIdForName(L"lsass.exe", &processId))
{
    if(hProcess = OpenProcess(PROCESS_VM_READ | PROCESS_VM_WRITE |
                    PROCESS_VM_OPERATION | PROCESS_QUERY_INFORMATION, FALSE, processId))
    {
        if(kull_m_memory_open(KULL_M_MEMORY_TYPE_PROCESS, hProcess, &aLsass.hMemory))
```

In this code, Mimikatz first gets the process ID of lsass.exe and stores it in the *processId* variable. Next, it calls OpenProcess to obtain a handle to the lsass.exe process. This handle gives the ability to read and write the memory of the lsass.exe process from the calling process. Mimikatz then calls *kull_m_memory_open*, which is an internal Mimikatz function that stores the handle for later use.

After Mimikatz is able to read and write memory of the lsass.exe process, it then searches for the *Kerberos-Newer-Keys* string in memory so that it can find the data structure related to AES-based authentication. It then manipulates this structure so that authentication is downgraded to the weaker RC4 without the use of a salt. Note that this is the same approach described in the previously linked Cycraft report. Older reports, such as the one from Microsoft, describe how malware can also achieve the same result by hooking the *SamIRetrieveMultiplePrimaryCredentials* function and forcing it return an error when the *Kerberos-Newer-Keys* package is used. The end result of both methods is the same: all authentication attempts to an infected domain controller will use the weaker RC4 algorithm.

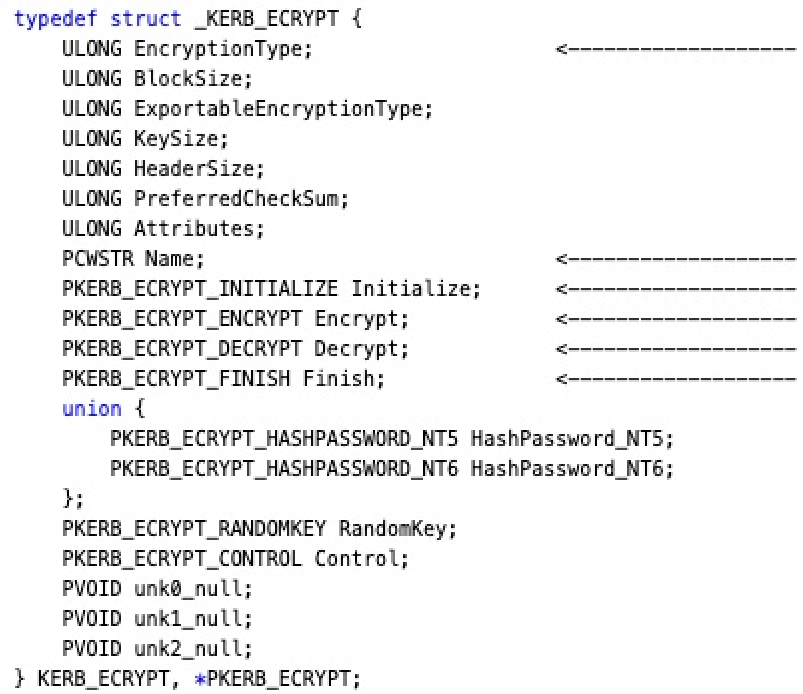
After downgrading the domain controller to RC4, Mimikatz will then attempt to locate and patch the data structure that handles RC4-based authentication. The following image shows the beginning of this code:



```
if(kull_m_process_getVeryBasicModuleInformationsForName(aLsass.hMemory, "cryptdll.dll", &cryptInfo))
{
    localAddr = (PBYTE) GetModuleHandle(L"cryptdll.dll");
    if(NT_SUCCESS(CDLocateCSystem(KERB_ETYPE_RC4_HMAC_NT, &pCrypt)))
    {
        extensions[3].Pointer = (PBYTE) cryptInfo.DllBase.address +
                                ((PBYTE) pCrypt->Initialize - localAddr);

        extensions[4].Pointer = (PBYTE) cryptInfo.DllBase.address +
                                ((PBYTE) pCrypt->Decrypt - localAddr);

        if(kull_m_remotelib_CreateRemoteCodeWitthPatternReplace(
                            aLsass.hMemory, kuhl_misc_skeleton_rc4_init,
                            kuhl_misc_skeleton_rc4_end - kuhl_misc_skeleton_rc4_init,
                            &extForCb, &aLsass))
        {
            kprintf(L"[RC4] functions\n");
```

First, the module information is gathered for the “cryptdll.dll” module loaded within the lsass.exe process. This module is responsible for implementing the different encryption packages, which are also known as *systems*. Next, the address of where cryptdll.dll is loaded within the Mimikatz process is gathered by calling *GetModuleHandle.*

The undocumented *CDLocateCSystem* function is then called with an argument of KERB_ETYPE_RC4_HMAC_NT and the address to store the resulting lookup *(&pCry*pt). *CDLocateCSystem* determines the address of the data structure that handles the given (KERB_ETYPE_RC4_HMAC_NT) authentication system and copies its contents into the passed-in *pCrypt* address. In this instance, it will be for the *system* that implements RC4-based authentication, and it will contain the information shown below:



```
typedef struct _KERB_ECRYPT {
    ULONG EncryptionType;                    <-------------------
    ULONG BlockSize;
    ULONG ExportableEncryptionType;
    ULONG KeySize;
    ULONG HeaderSize;
    ULONG PreferredCheckSum;
    ULONG Attributes;
    PCWSTR Name;                             <-------------------
    PKERB_ECRYPT_INITIALIZE Initialize;      <-------------------
    PKERB_ECRYPT_ENCRYPT Encrypt;            <-------------------
    PKERB_ECRYPT_DECRYPT Decrypt;            <-------------------
    PKERB_ECRYPT_FINISH Finish;              <-------------------
    union {
        PKERB_ECRYPT_HASHPASSWORD_NT5 HashPassword_NT5;
        PKERB_ECRYPT_HASHPASSWORD_NT6 HashPassword_NT6;
    };
    PKERB_ECRYPT_RANDOMKEY RandomKey;
    PKERB_ECRYPT_CONTROL Control;
    PVOID unk0_null;
    PVOID unk1_null;
    PVOID unk2_null;
} KERB_ECRYPT, *PKERB_ECRYPT;
```

The actual data structure definition for this type is not documented by Microsoft, so the above image is directly from the Mimikatz source code. The arrows point to the members relevant to our plugin, which include the encryption type; the function pointers for the initialization, encryption, decryption, and finish operation handlers; and the pointer to the string name of the system.

After finding the *_KERB_ECRYPT* instance for RC4 through the use of *CDLocateCSystem*, Mimikatz then hooks the legitimate initialization (*Initialize*) and decryption (*Decrypt*) members of the structure. These hooks point the handlers to Mimikatz's malicious handlers (*kuhl_misc_skeleton_rc4_init* and *kuhl_misc_skeleton_rc4_init_decrypt*) instead of the legitimate ones. The malicious handlers are injected into the address space of lsass.exe through the use of the WriteProcessMemory function. Combined, these malicious handlers are what implement the Skeleton Key attack, as they give Mimikatz control over all future authentication attempts to the infected domain controller.

Devising a Detection Strategy

Now that we understand how Mimikatz implements its attack—forcing a downgrade to RC4 followed by hooking the RC4 initialization and decryption routines—we can devise a strategy to detect the attack in memory.

We could start by attempting to detect the RC4 downgrade, but this has a few limitations. First, the string needed to find this data structure (*Kerberos-Newer-Keys*) is zeroed out as part of the attack, removing the possibility of a scanning-based approach to finding it. Second, attempting to detect that this string has been zeroed out would lead to many false positives due to paging of data out to disk, as well as the possibility of the page holding the string being smeared. Third, there are other methods to force a downgrade to RC4 without directly altering this string (as discussed in earlier references), meaning several approaches would be needed to completely detect it. Finally, finding proof of the downgrade only gives a clue that a Skeleton Key attack might have been performed, but it does not offer direct evidence.

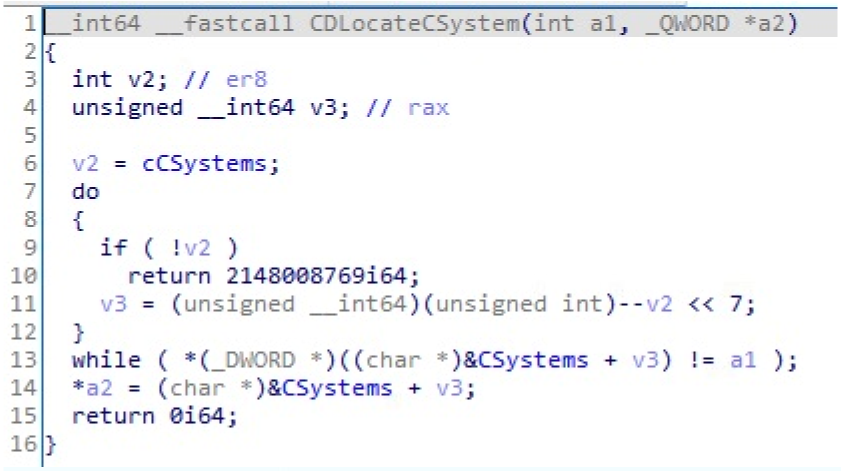
On the other hand, by examining the RC4 data structure directly, we can inspect the handlers for the initialization and decryption routines and determine if they were altered at runtime. This not only definitively tells us if a Skeleton Key attack occurred, but it also tells us exactly where the malicious handlers are inside of the infected lsass.exe process. Given that this approach gives direct evidence of the attack, as well as directly points out the malicious code, inspecting these handlers was chosen as the detection method for our plugin.

Reverse Engineering CDLocateCSystem

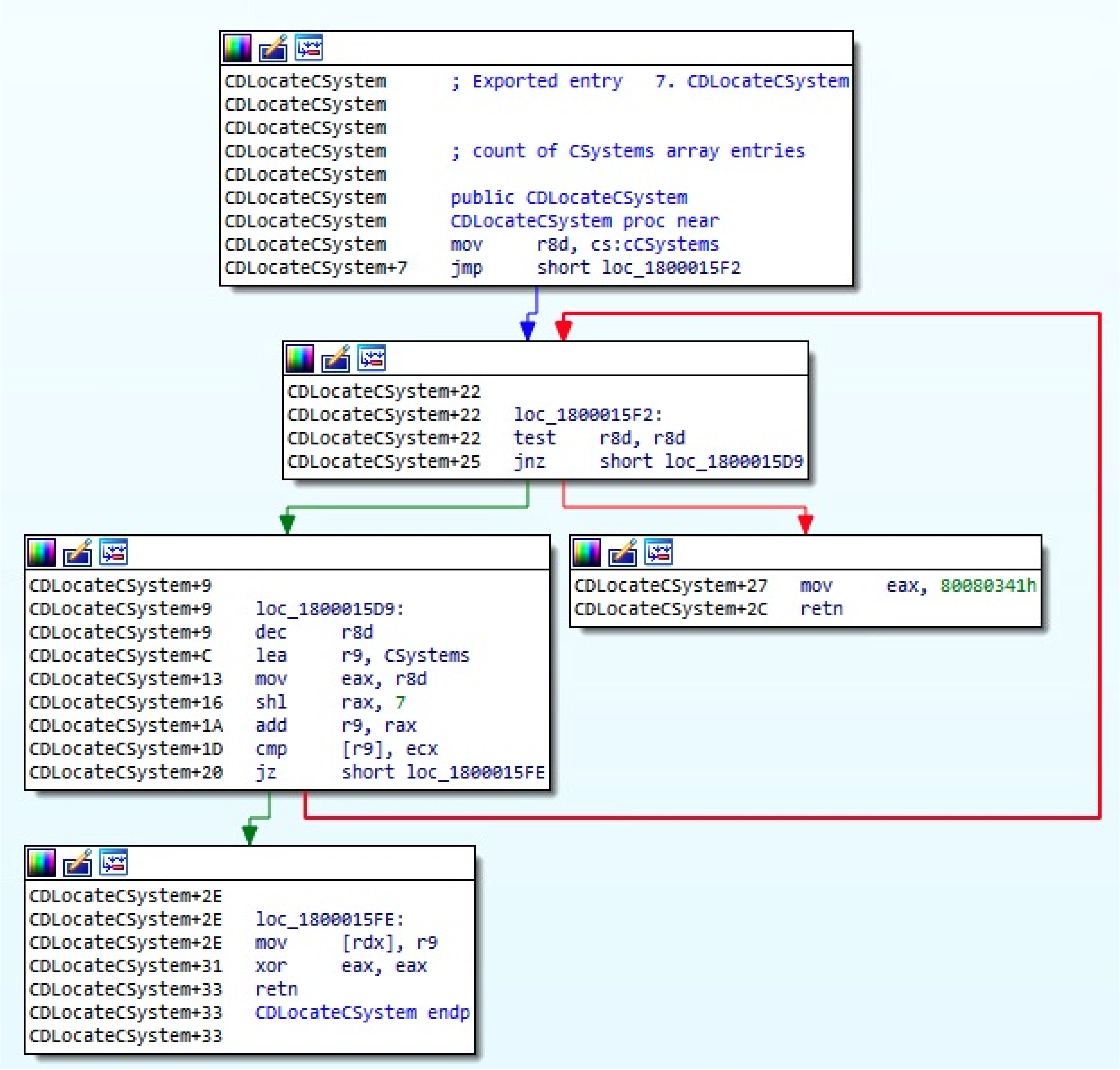
Before we can locate the handlers to then verify them, we need to be able to find the RC4 data structure in a repeatable and consistent manner. As shown previously, Mimikatz locates the address of the RC4 data structure by calling *CDLocateCSystem*. This tells us that if we can replicate the algorithm of *CDLocateCSystem*—or at least build an algorithm that is equal —we can reliably locate the RC4 structure to then verify its handlers.

Since cryptdll.dll contains the *CDLocateCSystem* function implementation and is closed source, we will need to reverse engineer the function to determine its algorithm. As you will see next, this function is pretty simple, so do not panic if you have never reverse engineered before; the concept will be straightforward.

The following images show the IDA Pro decompiler and graph view of *CDLocateCSystem*:



```
__int64 __fastcall CDLocateCSystem(int a1, _QWORD *a2)
{
  int v2; // er8
  unsigned __int64 v3; // rax

  v2 = cCSystems;
  do
  {
    if ( !v2 )
      return 2148008769i64;
    v3 = (unsigned __int64)(unsigned int)--v2 << 7;
  }
  while ( *(_DWORD *)((char *)&CSystems + v3) != a1 );
  *a2 = (char *)&CSystems + v3;
  return 0i64;
}
```



```
CDLocateCSystem      ; Exported entry   7. CDLocateCSystem
CDLocateCSystem
CDLocateCSystem
CDLocateCSystem      ; count of CSystems array entries
CDLocateCSystem
CDLocateCSystem      public CDLocateCSystem
CDLocateCSystem      CDLocateCSystem proc near
CDLocateCSystem      mov     r8d, cs:cCSystems
CDLocateCSystem+7    jmp     short loc_1800015F2
```

```
CDLocateCSystem+22
CDLocateCSystem+22   loc_1800015F2:
CDLocateCSystem+22   test    r8d, r8d
CDLocateCSystem+25   jnz     short loc_1800015D9
```

```
CDLocateCSystem+9
CDLocateCSystem+9    loc_1800015D9:
CDLocateCSystem+9    dec     r8d
CDLocateCSystem+C    lea     r9, CSystems
CDLocateCSystem+13   mov     eax, r8d
CDLocateCSystem+16   shl     rax, 7
CDLocateCSystem+1A   add     r9, rax
CDLocateCSystem+1D   cmp     [r9], ecx
CDLocateCSystem+20   jz      short loc_1800015FE
```

```
CDLocateCSystem+27   mov     eax, 80080341h
CDLocateCSystem+2C   retn
```

```
CDLocateCSystem+2E
CDLocateCSystem+2E   loc_1800015FE:
CDLocateCSystem+2E   mov     [rdx], r9
CDLocateCSystem+31   xor     eax, eax
CDLocateCSystem+33   retn
CDLocateCSystem+33   CDLocateCSystem endp
CDLocateCSystem+33
```

As seen above, the function is pretty small and simple. It begins (the first instruction of *CDLocateCSystem* in the disassembly view) by copying the current value of *cCSystems* global variable into the r8d register, which is an alias for the lower 32 bits of 64-bit r8 register.

It then tests (*CDLocateCSystem*+22) if the value is zero and bails with an error (+27) if it is. If the value is anything but zero, then it moves to basic block, starting at offset +9. This basic block begins by decrementing the r8d register (+9). This code pattern of storing a variable, checking if it is zero, and then decrementing the value tells us that this is likely a counter for looping (iterating) through a data structure. Looking ahead, the red line leading from +20 back to +22 in the graph confirms this, as the code at +22 will be evaluated every time the basic block starting at +9 fails to exit. This is exactly how loops look in IDA Pro and other basic block graphing tools.

Further studying the basic block starting at +9, we see the address of the *CSystems* global variable copied into the r9 register. Next (+13), the value in r8d is copied into eax, and then rax is shifted left by 7
(+16), which is the same thing as being multiplied by 128 (2 to the 7th power). This computed value is then stored into r9, and the data r9 points to is compared with the value in ecx (+1D). If this comparison matches, then the function returns. Otherwise, the flow starting at +9 repeats.

Breaking this down, the code is using the current value of r8d multiplied by 128 (shifted left by 7) as an index into *CSystems*. This is exactly what iterating through an array looks like. As each array element is stored contiguously in memory, by knowing the size and count, you can successfully locate each element. This understanding of the code now tells us two things:

1. *cCSystems* holds the number of elements in *CSystems.*
2. The size of each *CSystems* element is 128 bytes.

For the basic block at +9, the only remaining parts to understand are which values are being compared at +1D and the purpose of that comparison. Since the loop breaks dependent on that comparison, it is likely critical to the function's overall purpose. Looking at the two values, [r9] and ecx, we know a few things. First, the comparison will be comparing two 32-bit values, as that is the size of ecx, which is the lower 32 bits of rcx. Second, the brackets around r9 mean to treat the value of r9 as an address in memory and then retrieve the value at that address, which is known as dereferencing an address (pointer). From our previous discussion, we know that r9 holds the address of the current *CSystems* element being inspected. Dereferencing it as [r9] is equivalent to dereferencing [r9+0], which tells us that the first 4 bytes (32 bits) of the referenced structure are being accessed.

As for ecx, the instruction at +1D is the first time ecx (or any of rcx) is referenced. This means the value must have been set before the function was called. Consulting the Microsoft documentation on function-calling conventions, we see that the rcx register is used on 64-bit systems to store the first parameter sent to a function. Earlier, when we examined how Mimikatz called *CDLocateCSystem*, we noted that the first argument was the *KERB_ETYPE_RC4_HMAC_NT* constant, which is defined in NTSecAPI.h of the Windows

SDK as 0x17 hex (23 decimal). This means that *CDLocateCSystem* will be searching for an element of *CSystems* that has 0x17 (23) as the first integer.

Looking at the end of the function (+2E -> +33), we see that r9 is stored into the address pointed to by rdx; the previous Microsoft documentation tells us rdx stores the second parameter to a function. We know for *CDLocateCSystem* that this is the address of where the calling code (Mimikatz) wants Windows to store the address of the requested authentication system (RC4).

Seeing that the address of the *CSystems* element found in the loop is directly returned to the caller tells us that the data structure returned is also of type *_KERB_ECRYPT*, since we know that is the type of the second parameter to *CDLocateCSystem*. This then tells us that the integer at offset 0, that is compared in the loop, is actually the *EncryptionType* member of structure. This makes sense, since it holds the integer value for the particular authentication system type. It also means that the elements in *CSystems* are the ones actually used by Windows during the authentication process, since these are the ones directly targeted by Skeleton Key attacks.

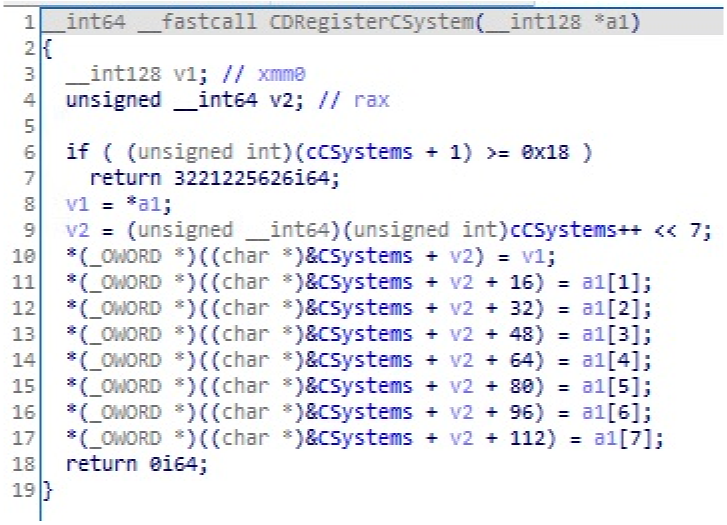
In summary, reverse engineering has showed us that the active RC4 authentication system structure can be located by enumerating *CSystems* and then looking for the element that has an *EncryptionType* of 0x17 (23). This precisely matches how *CDLocateCSystem* uses its first parameter to determine which element of the *CSystems* array to return to the caller. It also tells us that the type of each element is *KERB_ECRYPT*, which is very handy since we already have the definition for this type.

Reverse Engineering the RC4 Structure Origin

After learning how *CDLocateCSystem* operated, the next analysis step taken was to determine if the RC4 structure inside the *CSystems* array could be found directly. While enumerating the array is not difficult nor time consuming, in memory forensics research we aim to find the most direct path to data to avoid analysis issues that can be caused by smear.

To begin this analysis, we wanted to determine how elements of *CSystems* were registered, with particular interest in the RC4 system. Examining cross-references (meaning, finding code that references), *CSystems* showed only a few locations inside of cryptdll.dll. Of these, the *CDRegisterCSystem* function sounded the most promising, as it would hopefully lead us to RC4 being registered.

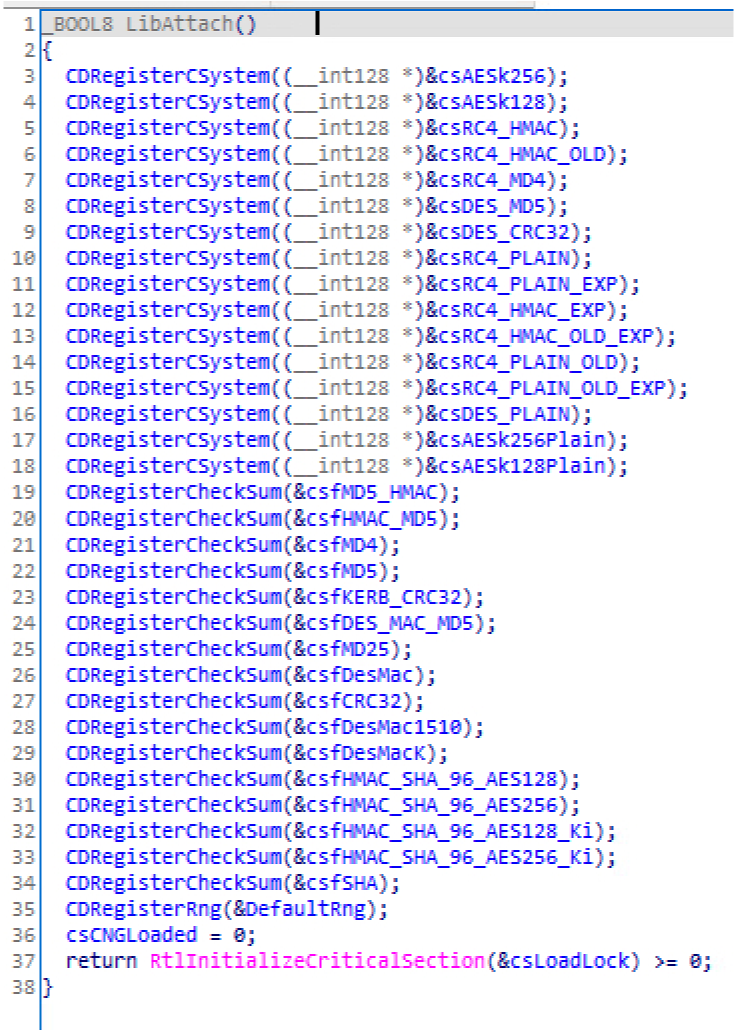
The following image shows the decompiled view of this function:



```
 1 __int64 __fastcall CDRegisterCSystem(__int128 *a1)
 2 {
 3   __int128 v1; // xmm0
 4   unsigned __int64 v2; // rax
 5 
 6   if ( (unsigned int)(cCSystems + 1) >= 0x18 )
 7     return 3221225626i64;
 8   v1 = *a1;
 9   v2 = (unsigned __int64)(unsigned int)cCSystems++ << 7;
10   *(_OWORD *)((char *)&CSystems + v2) = v1;
11   *(_OWORD *)((char *)&CSystems + v2 + 16) = a1[1];
12   *(_OWORD *)((char *)&CSystems + v2 + 32) = a1[2];
13   *(_OWORD *)((char *)&CSystems + v2 + 48) = a1[3];
14   *(_OWORD *)((char *)&CSystems + v2 + 64) = a1[4];
15   *(_OWORD *)((char *)&CSystems + v2 + 80) = a1[5];
16   *(_OWORD *)((char *)&CSystems + v2 + 96) = a1[6];
17   *(_OWORD *)((char *)&CSystems + v2 + 112) = a1[7];
18   return 0i64;
19 }
```

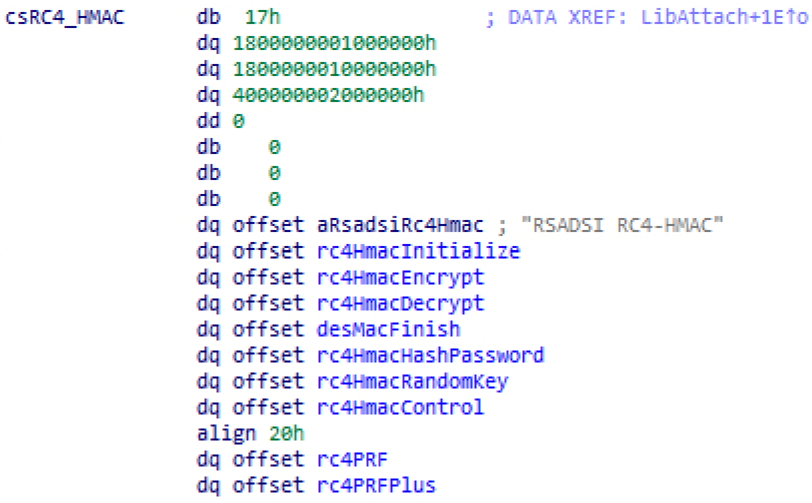
As can be seen, this is a pretty simple function that first (line 6) checks against the maximum number of registered systems (0x18), and then bails if already at the maximum. Next, the function determines the offset into C*Systems* (line 9) by using *cCSystems* shifted by 7. This matches our understanding of *cCSystems* and the shifting by 7 from earlier. The function then simply copies in the values from the passed in data structure (*a1*) into the correct offsets of C*Systems*. In summary, whatever values are in the system being registered are copied separately inside of *CSystems*, duplicating them in memory.

Following cross-references to *CDRegisterCSystem* leads us to many references inside of *LibAttach; a* decompiled view is shown below:



```
 1 _BOOL8 LibAttach()
 2 {
 3   CDRegisterCSystem((__int128 *)&csAESk256);
 4   CDRegisterCSystem((__int128 *)&csAESk128);
 5   CDRegisterCSystem((__int128 *)&csRC4_HMAC);
 6   CDRegisterCSystem((__int128 *)&csRC4_HMAC_OLD);
 7   CDRegisterCSystem((__int128 *)&csRC4_MD4);
 8   CDRegisterCSystem((__int128 *)&csDES_MD5);
 9   CDRegisterCSystem((__int128 *)&csDES_CRC32);
10   CDRegisterCSystem((__int128 *)&csRC4_PLAIN);
11   CDRegisterCSystem((__int128 *)&csRC4_PLAIN_EXP);
12   CDRegisterCSystem((__int128 *)&csRC4_HMAC_EXP);
13   CDRegisterCSystem((__int128 *)&csRC4_HMAC_OLD_EXP);
14   CDRegisterCSystem((__int128 *)&csRC4_PLAIN_OLD);
15   CDRegisterCSystem((__int128 *)&csRC4_PLAIN_OLD_EXP);
16   CDRegisterCSystem((__int128 *)&csDES_PLAIN);
17   CDRegisterCSystem((__int128 *)&csAESk256Plain);
18   CDRegisterCSystem((__int128 *)&csAESk128Plain);
19   CDRegisterCheckSum(&csfMD5_HMAC);
20   CDRegisterCheckSum(&csfHMAC_MD5);
21   CDRegisterCheckSum(&csfMD4);
22   CDRegisterCheckSum(&csfMD5);
23   CDRegisterCheckSum(&csfKERB_CRC32);
24   CDRegisterCheckSum(&csfDES_MAC_MD5);
25   CDRegisterCheckSum(&csfMD25);
26   CDRegisterCheckSum(&csfDesMac);
27   CDRegisterCheckSum(&csfCRC32);
28   CDRegisterCheckSum(&csfDesMac1510);
29   CDRegisterCheckSum(&csfDesMacK);
30   CDRegisterCheckSum(&csfHMAC_SHA_96_AES128);
31   CDRegisterCheckSum(&csfHMAC_SHA_96_AES256);
32   CDRegisterCheckSum(&csfHMAC_SHA_96_AES128_Ki);
33   CDRegisterCheckSum(&csfHMAC_SHA_96_AES256_Ki);
34   CDRegisterCheckSum(&csfSHA);
35   CDRegisterRng(&DefaultRng);
36   csCNGLoaded = 0;
37   return RtlInitializeCriticalSection(&csLoadLock) >= 0;
38 }
```

This function is exactly what we were looking for, as we can see all the different systems being registered. We also see our system of interest, *csRC4_HMAC,* being registered on line 5. If we examine the data at this address, we can verify this with seeing 0x17 (23) as the first integer. We learned earlier that this is the *EncryptionType* targeted by Mimikatz.



```
csRC4_HMAC      db  17h                  ; DATA XREF: LibAttach+1E↑o
                dq 1800000001000000h
                dq 1800000010000000h
                dq 400000002000000h
                dd 0
                db    0
                db    0
                db    0
                dq offset aRsadsiRc4Hmac ; "RSADSI RC4-HMAC"
                dq offset rc4HmacInitialize
                dq offset rc4HmacEncrypt
                dq offset rc4HmacDecrypt
                dq offset desMacFinish
                dq offset rc4HmacHashPassword
                dq offset rc4HmacRandomKey
                dq offset rc4HmacControl
                align 20h
                dq offset rc4PRF
                dq offset rc4PRFPlus
```

As seen above, not only is the 0x17 (23) present at the first offset, but a little further down we also see the string defined for the system (*RSADSI RC4-HMAC*), as well as the handlers for events the system must support. Looking at the list of functions, we find the legitimate handlers for the initialization (*rc4HmacInitialize)* and decryption (*rc4HmacDecrypt*) routines that Mimikatz targets. This gives us the specific symbol names that should correspond to the handlers we find inside of analyzed memory samples.

In summary, this reverse-engineering effort to find the origin structure led us to two import conclusions. First, even though we know the symbol name of the static RC4 structure (*csRC4_HMAC*), we cannot analyze this directly, as a copy of its values will be placed inside of *CSystems*. This means we will still need to enumerate *CSystems* to get the “active” values, but it also means that we can potentially choose to leverage the duplicate, original data in our plugin. Second, by knowing the symbol names of the legitimate initialization and decryption handlers, we can make the sanity checks performed by our plugins as specific as possible.

With these two reversing efforts complete, we can now start to develop our plugin!

Designing the *windows.skeleton_key_check Plugin*

Our previous analysis gave us all the information we need to design and implement our plugin; we saw exactly how the operating system retrieves our desired data structure. As a direct approach, this would include the following steps:

1. Find the address of *CSystems*
2. Walk each element to find the active RC4 system
3. Compare its initialization and decryption handlers to the known-good symbols

After the handlers are processed, the plugin would then report whether the handler’s value is legitimate or if a Skeleton Key attack has been performed.

Creating a New Plugin

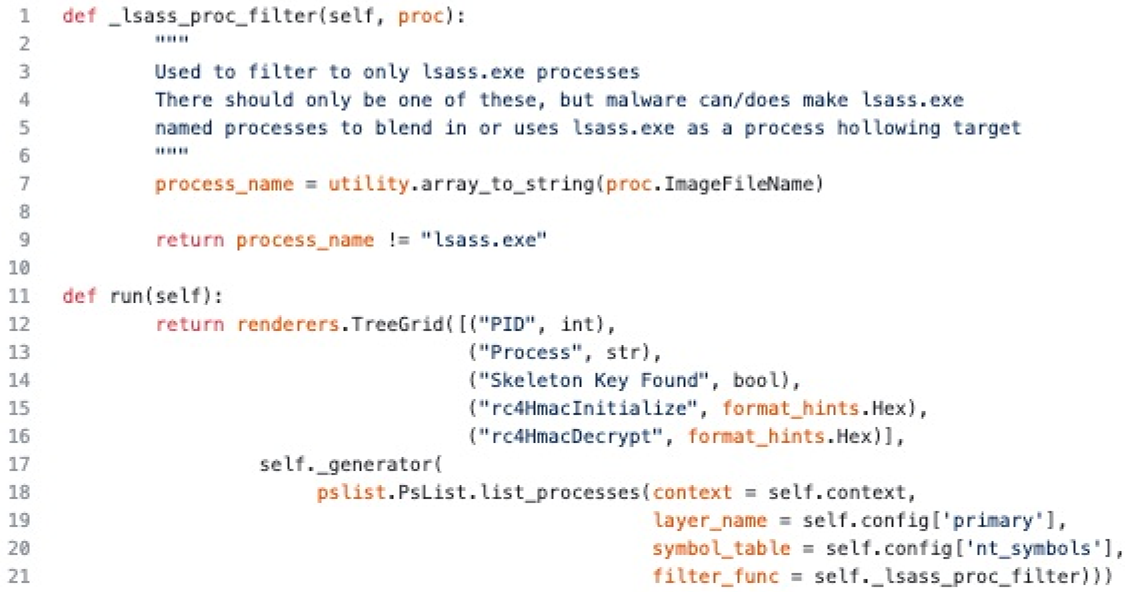
To start, we must create a base Volatility 3 plugin that is capable of processing Windows samples. A major goal of Volatility 3 was to have significant and always-up-to-date documentation for both users and developers. This documentation is stored on the Volatility 3 page of readthedocs. There is also a section specifically on writing a basic plugin here.

At a high level, all plugins must define their *requirements*, a *run* method, and a *generator* method. The *run* method executes first and calls the *generator* method to create the data sets that will be displayed on the terminal (or output in whatever format other interfaces support). For more information, please see the documentation above.

For our Skeleton Key plugin, we use the basic starting form to then implement the steps listed previously. Note that the plugin being described in this blog post is already available in Volatility 3 here. Since line numbers change after each new commit, we instead will be referencing portions of the plugin by the function name. Also, we will be showing screenshots of code portions being discussed with the line numbers starting at 1. This will guide the discussion in a consistent manner.

Implementation - Writing the *run* Function

The *run* function is called first when a plugin's execution begins. The expected return value is a TreeGrid that the calling user interface will then display for the analyst. The following image shows the *run* function, along with the process filter from our Skeleton Key plugin:



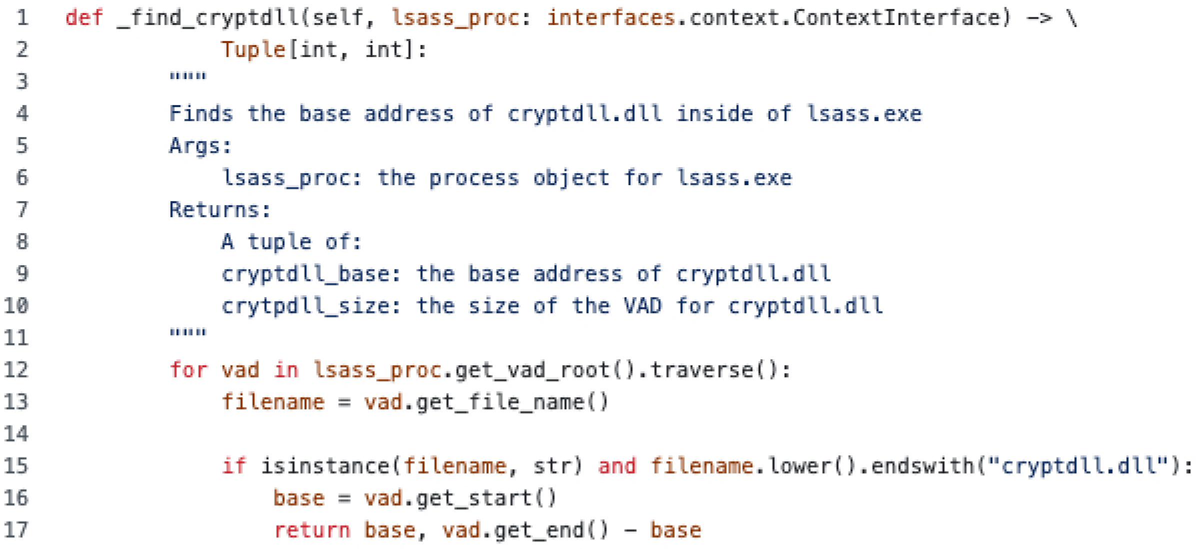
```
1   def _lsass_proc_filter(self, proc):
2           """
3           Used to filter to only lsass.exe processes
4           There should only be one of these, but malware can/does make lsass.exe
5           named processes to blend in or uses lsass.exe as a process hollowing target
6           """
7           process_name = utility.array_to_string(proc.ImageFileName)
8
9           return process_name != "lsass.exe"
10
11  def run(self):
12          return renderers.TreeGrid([("PID", int),
13                                     ("Process", str),
14                                     ("Skeleton Key Found", bool),
15                                     ("rc4HmacInitialize", format_hints.Hex),
16                                     ("rc4HmacDecrypt", format_hints.Hex)],
17                        self._generator(
18                             pslist.PsList.list_processes(context = self.context,
19                                                          layer_name = self.config['primary'],
20                                                          symbol_table = self.config['nt_symbols'],
21                                                          filter_func = self._lsass_proc_filter)))
```

On line 12, the return statement begins with the construction of the required TreeGrid instance. The first parameter to the TreeGrid constructor is the list of columns that the plugin will display. Each column is specified with its name and type. For this plugin, we have chosen to display the process ID and name of analyzed lsass.exe instances; whether or not a Skeleton Key attack was found; and the addresses of the initialization and decryption handlers. Note that the handlers are listed by their address in memory, which Volatility 3 will automatically print in hexadecimal due to the *format_hints.Hex* specifier. This is similar to the *[addrpad]* specifier of Volatility 2.

Next, the *generator* function is called. For plugins that operate on data not made available by another plugin, the *generator* function will be called with no arguments. For Skeleton Key, since we only want to analyze lsass.exe processes, we can leverage *list_processes* to perform the filtering for us. This filtering occurs through the use of the *filter_func* argument, which specifies a callback that evaluates if a process object should be yielded to the caller. Our filtering function, *lsassproc_filter*, is very simple; it only needs to evaluate if the process name is lsass.exe.

Implementation – Leveraging PDBs

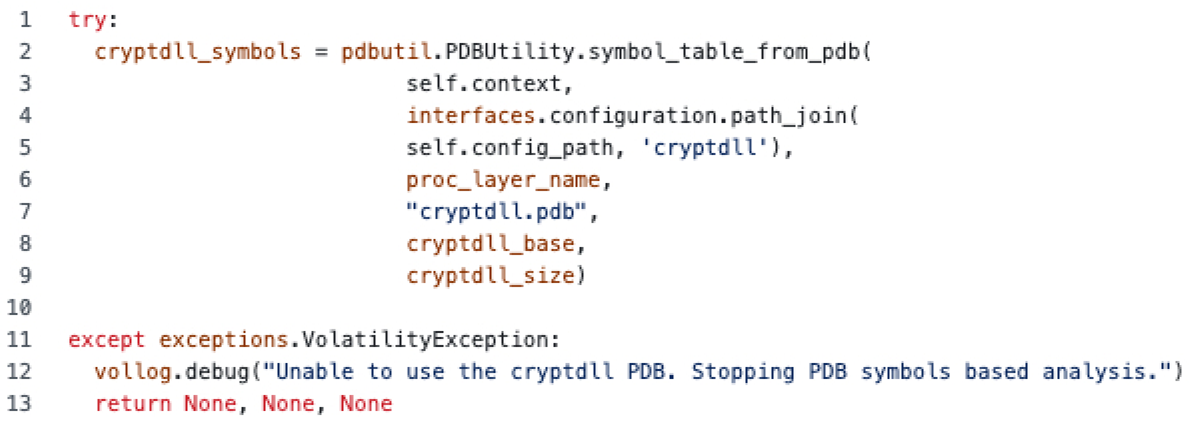
Our reverse-engineering effort showed us that four symbols—*cSystems*, *cCSystems*, *rc4HmacInitialize*, *rc4HmacDecrypt*—hold the key data we need to write a complete plugin. Luckily, one of the new features of Volatility 3 is the ability to automatically download and incorporate PDB (symbol) files into the analysis flow of plugins. This is accomplished by locating the PE file (.exe, .dll, .sys) of interest and parsing it with the PDB utility API. Since cryptdll.dll holds the symbols our plugins need, the first step is to find the DLL within the address of lsass.exe:



```
1  def _find_cryptdll(self, lsass_proc: interfaces.context.ContextInterface) -> \
2              Tuple[int, int]:
3          """
4          Finds the base address of cryptdll.dll inside of lsass.exe
5          Args:
6              lsass_proc: the process object for lsass.exe
7          Returns:
8              A tuple of:
9              cryptdll_base: the base address of cryptdll.dll
10             crytpdll_size: the size of the VAD for cryptdll.dll
11         """
12         for vad in lsass_proc.get_vad_root().traverse():
13             filename = vad.get_file_name()
14
15             if isinstance(filename, str) and filename.lower().endswith("cryptdll.dll"):
16                 base = vad.get_start()
17                 return base, vad.get_end() - base
```

The above image shows that *_find_cryptdll*—a function that receives the process object for lsass.exe—iterates through its memory regions (line 12), retrieves the filename for the current region (line 13), and checks for the file of interest (13-15). Once cryptdll.dll is found, its base address and size are returned (16-17).

Once cryptdll.dll has been located, its information can then be passed to the PDB utility APIs:

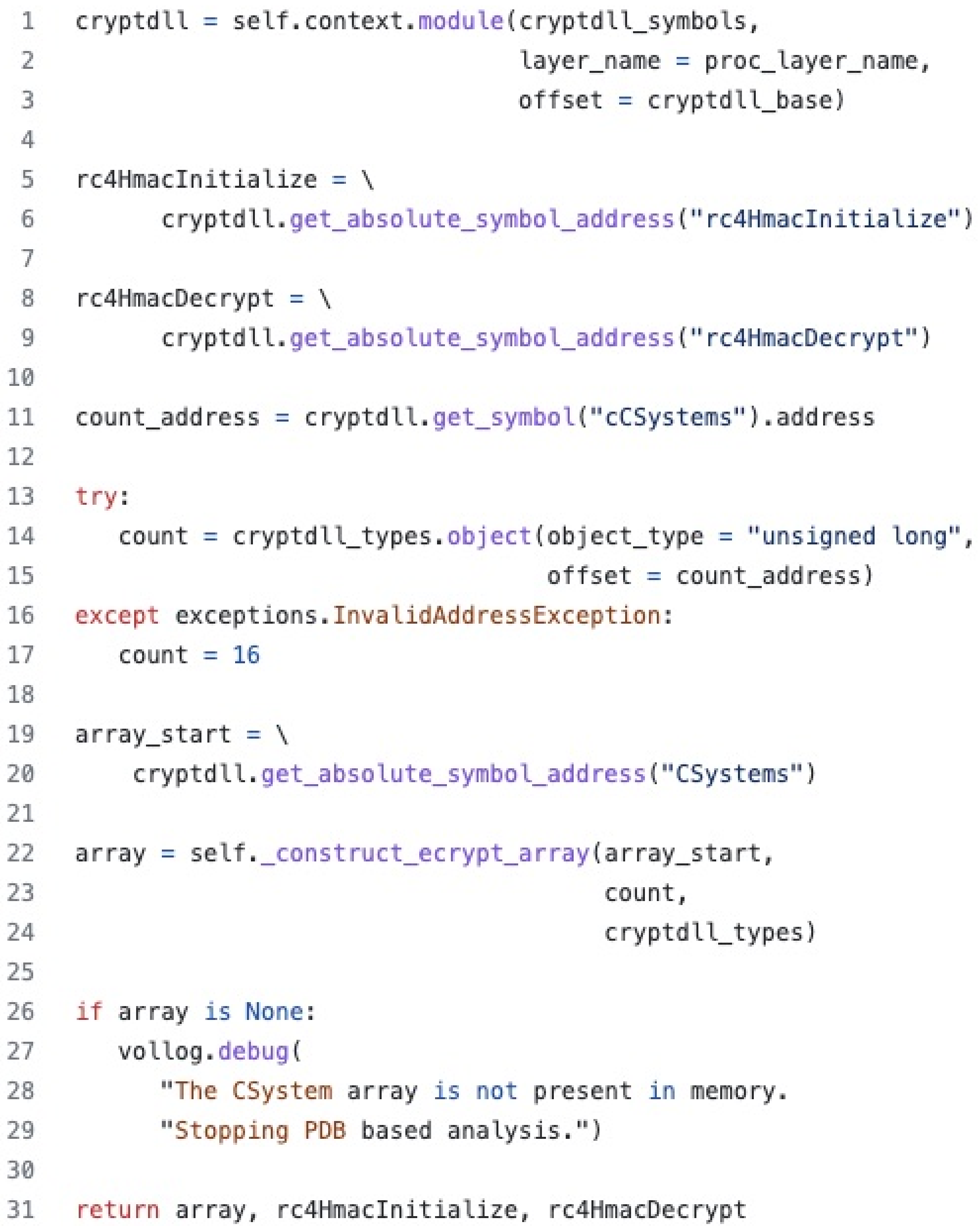


```
try:
  cryptdll_symbols = pdbutil.PDBUtility.symbol_table_from_pdb(
                        self.context,
                        interfaces.configuration.path_join(
                        self.config_path, 'cryptdll'),
                        proc_layer_name,
                        "cryptdll.pdb",
                        cryptdll_base,
                        cryptdll_size)

except exceptions.VolatilityException:
  vollog.debug("Unable to use the cryptdll PDB. Stopping PDB symbols based analysis.")
  return None, None, None
```

As shown, calling into the PDB API is straightforward, but this actually triggers quite a bit of activity inside the core of Volatility 3. First, the memory range specified for the PE file is scanned to find its GUID, which is unique identifier for the file. Next, the local Volatility cache is checked to see if the PDB for this GUID has already been downloaded and processed during previous plugin runs. If so, then the cached file is parsed and returned to the caller.

If the GUID is not in the cache, then Volatility will attempt to download the PDB file from the Microsoft symbol server. If successful, then the PDB will be parsed, converted to Volatility's symbol table format, and stored within the cache.

Assuming the PDB is successfully downloaded and parsed, then our plugin has direct access to the offsets of the needed symbols within the particular version of cryptdll.dll. This allows us to trivially find their values within a particular memory sample:

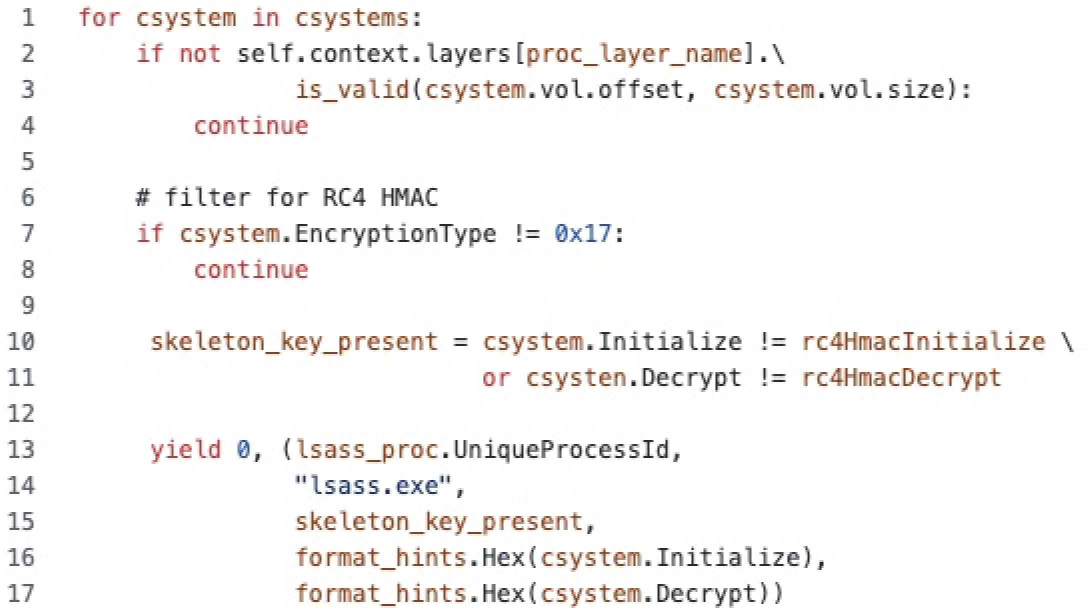


```
cryptdll = self.context.module(cryptdll_symbols,
                               layer_name = proc_layer_name,
                               offset = cryptdll_base)

rc4HmacInitialize = \
      cryptdll.get_absolute_symbol_address("rc4HmacInitialize")

rc4HmacDecrypt = \
      cryptdll.get_absolute_symbol_address("rc4HmacDecrypt")

count_address = cryptdll.get_symbol("cCSystems").address

try:
   count = cryptdll_types.object(object_type = "unsigned long",
                                 offset = count_address)
except exceptions.InvalidAddressException:
   count = 16

array_start = \
    cryptdll.get_absolute_symbol_address("CSystems")

array = self._construct_ecrypt_array(array_start,
                                     count,
                                     cryptdll_types)

if array is None:
   vollog.debug(
      "The CSystem array is not present in memory.
      "Stopping PDB based analysis.")

return array, rc4HmacInitialize, rc4HmacDecrypt
```

The code shown gathers the runtime address for each of the four desired symbols. For the handlers, we only need their address in memory to compare to the ones in the active RC4 system. For *cCSystems*, we treat it separately, as we do not want processing to fail simply because the page holding the count is unavailable.

We also treat *CSystems* separately, as we need to construct an array type to cleanly enumerate its elements. Constructing this object requires not only the address of where C*Systems* is in memory, but also the structure definition for the array elements. Unlike the PDB file for the kernel, which includes both symbol offsets and type information, the PDB file for cryptdll.dll only includes the symbol offsets. This means we need to manually inform Volatility of the data structure layout. This is performed in Volatility 3 by creating a JSON file that describes the data structure(s) a plugin requires. You can view this file for the *_KERB_ECRYPT* structure here, which was based on the definition from Mimikatz discussed earlier.

Once the array is constructed, it can then be enumerated as shown below:

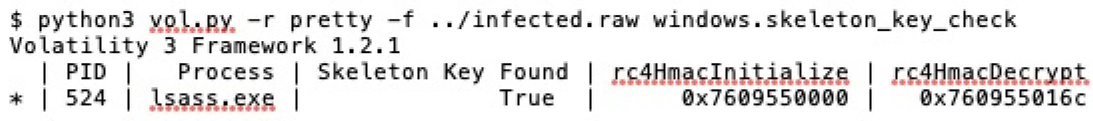


```
1   for csystem in csystems:
2       if not self.context.layers[proc_layer_name].\
3                  is_valid(csystem.vol.offset, csystem.vol.size):
4           continue
5
6       # filter for RC4 HMAC
7       if csystem.EncryptionType != 0x17:
8           continue
9
10       skeleton_key_present = csystem.Initialize != rc4HmacInitialize \
11                              or csysten.Decrypt != rc4HmacDecrypt
12
13       yield 0, (lsass_proc.UniqueProcessId,
14                 "lsass.exe",
15                 skeleton_key_present,
16                 format_hints.Hex(csystem.Initialize),
17                 format_hints.Hex(csystem.Decrypt))
```

Volatility has built-in support for enumerating arrays, so the for loop will walk each element, creating the *csystem* variable as the *_KERB_ECRYPT* type. Before processing an element, it is checked for being valid (mapped) into the process address space (lines 2-3). Next, the *EncryptionType* value is compared with our type of interest (lines 6-7). To determine if a Skeleton Key is present, we compare the *Initialize* and *Decrypt* members of the system found in memory to the expected values from the PDB file. If either of these have been modified, then a Skeleton Key attack has occurred, or at a minimum, a modification has occurred that an analyst would want to know about.

With all of the values computed, displaying the results to the analyst requires just a simple yield of the data. This can be seen in lines 13-17 and will result in the process name and PID, presence of a Skeleton Key, and handler addresses being displayed. This immediately informs the analyst if a Skeleton Key was found, and if so, where the malicious handler values are in memory.

The following image shows a run of our new plugin against an infected memory sample:



```
$ python3 vol.py -r pretty -f ../infected.raw windows.skeleton_key_check
Volatility 3 Framework 1.2.1
  | PID |   Process | Skeleton Key Found | rc4HmacInitialize | rc4HmacDecrypt
* | 524 | lsass.exe |               True |      0x7609550000 |   0x760955016c
```

Adding Resiliency to *windows.skeleton_key_check*

So far, our plugin is able to successfully detect Skeleton Key attacks by leveraging the cryptdll.dll PDB file to determine where our four symbols of interest are located in memory. Unfortunately, real-world memory forensics is not always this straightforward, and the data we would like may not be memory resident or it may be smeared. Thus, it is also advantageous to consider other approaches.

In the case of leveraging a PDB file for analysis, there are a few situations that could prevent us from determining which PDB file is needed for analysis, as well as obtaining that PDB file.

1. The page containing cryptdll.dll's GUID could be paged out or smeared.
2. The analysis system may be offline and unable to download the PDB file from Microsoft's symbol server.
3. Although rare, Microsoft has published corrupt/broken PDB files for modules shipped with stable versions of Windows.

In these situations, we would still like to be able to detect Skeleton Key attacks, but we need a different approach to gather the required data.

Finding CSystems Without a PDB File

Using knowledge gained from previous work on the plugin, we know that the *CDLocateCSystem* function directly references two of the four symbols we need: C*Systems* and *cCSystems*. This means that by performing static binary analysis of *CDLocateCSystem*, we should be able to determine the address of these symbols, since the function's instructions will reference the addresses themselves. This is a common tactic in memory analysis and reverse engineering tasks to find symbols that are not exported or where a symbol file cannot be obtained.

To attempt to find *CDLocateCSystem* without the use of a PDB file, we parse the export directory of cryptdll.dll, since it exports *CDLocateCSystem* by name. The following image shows how this is performed in Volatility 3:

```python
create_table = intermed.IntermediateSymbolTable.create

pe_table_name = create_table(self.context,
                             self.config_path,
                             "windows",
                             "pe",
                             class_types = pe.class_types)

cryptdll = self._get_pefile_obj(pe_table_name,
                                proc_layer_name,
                                cryptdll_base)
if not cryptdll:
    return None

cryptdll.parse_data_directories(directories = \
          [pefile.DIRECTORY_ENTRY["IMAGE_DIRECTORY_ENTRY_EXPORT"]])

if not hasattr(cryptdll, 'DIRECTORY_ENTRY_EXPORT'):
    return None

# find the location of CDLocateCSystem and then perform static analysis
for export in cryptdll.DIRECTORY_ENTRY_EXPORT.symbols:
    if export.name != b"CDLocateCSystem":
        continue

    function_start = cryptdll_base + export.address

    try:
        function_bytes = \
             self.context.layers[proc_layer_name].read(function_start,
                                                       0x50)
    except exceptions.InvalidAddressException:
        vollog.debug("The CDLocateCSystem function is not present. " \
                     "Stopping export based analysis.")
        break

    array = self._analyze_cdlocatecsystem(function_bytes,
                                          function_start,
                                          cryptdll_types,
                                          proc_layer_name)
    if array is None:
        vollog.debug("The CSystem array is not present in memory. "
                     "Stopping export based analysis.")

    return array
```

First, a reference is obtained to the type information for PE files (lines 1-7). Next, a Volatility 3 PE file object is constructed starting at the base address of cryptdll.dll (lines 9-11). This object contains a number of convenience methods for accessing common data, such as the data directories. This is leveraged on line 15 to parse the export directory, and then loop through its exported symbols starting on line 22. The body of this loop then looks for *CDLocateCSystem*, and when found, attempts to read the bytes (opcodes of the instructions) from its location in memory.

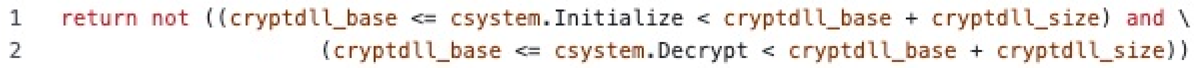
If these bytes can be read, then the *_analyze_cdlocatecsystem* function is called, which leverages capstone to perform the static disassembly necessary to locate both symbols. After locating them, it will construct the array object using the same method as described when the PDB file symbols were used.

Assuming the export table and opcodes for *CDLocationCSystem* are present, this method will successfully find CSystems and allow us to locate the RC4 structure as we did previously.

Finding *rc4HmacInitialize* and *rc4HmacDecrypt*

So far, we have been able to locate the RC4 structure without the PDB file. Unfortunately, there are no direct references to the legitimate initialize and decrypt handlers that we can leverage. This leaves us with two options. The first option is to verify the memory region holding the handlers, which will be discussed in this section. The second option is to attempt to scan for the values, which is discussed in the next section. Each has advantages and drawbacks, as we will discuss.

Each memory region within a process's address space is tracked by a virtual address descriptor (VAD). Information in the VAD includes the starting and ending address of the region; the initial protection of the region; and the number of committed pages. For executables, such as lsass.exe and cryptdll.dll, one VAD will track all regions of the executable, including its code and data. Knowing this, we can check if the values of the initialization and decryption handlers are within the region for cryptdll.dll. This is shown in the following image:



```
return not ((cryptdll_base <= csystem.Initialize < cryptdll_base + cryptdll_size) and \
                    (cryptdll_base <= csystem.Decrypt < cryptdll_base + cryptdll_size))
```

This simple check ensures that the value of the handler is within the starting and ending range of the VAD for cryptdll.dll. Although this check is not as precise as having the exact, legitimate values from the PDB file, this method still detects all forms of Skeleton Key attacks found in the wild, as they all allocate new VADs to hold the shellcode of the malicious handlers.

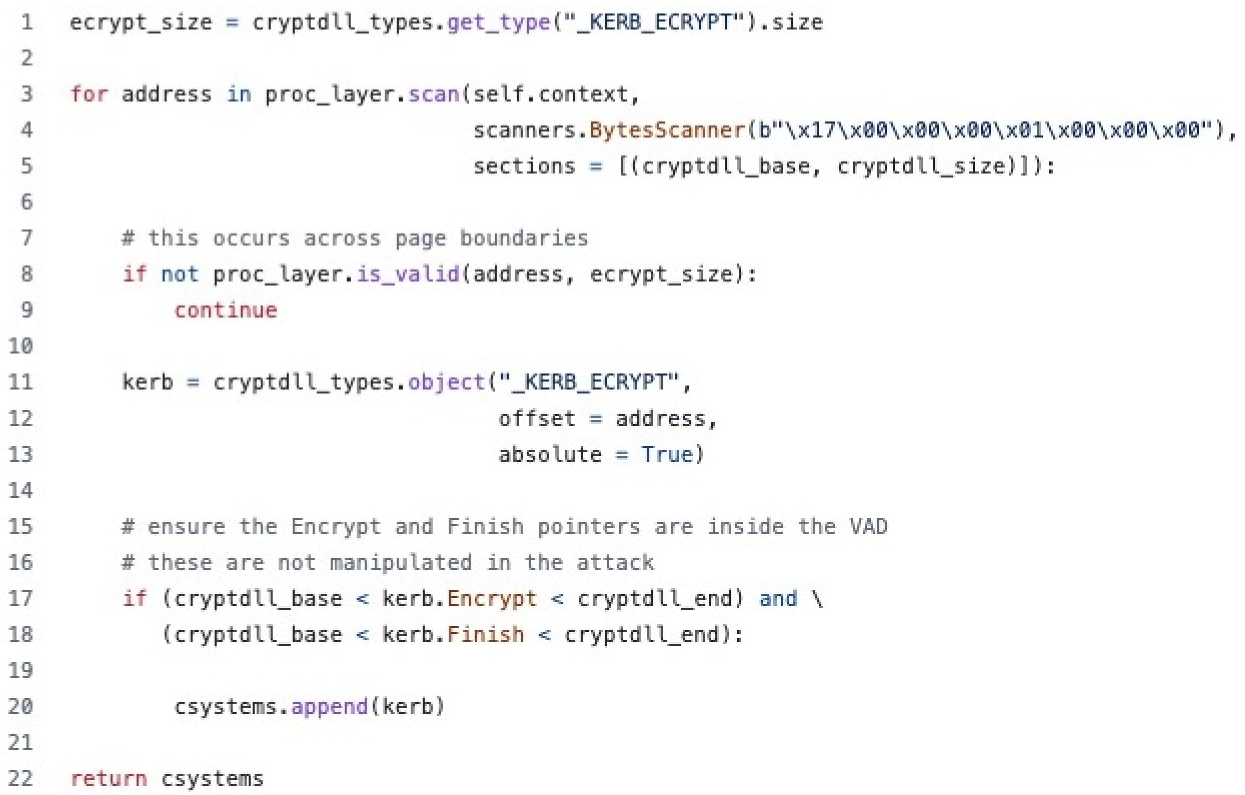
Note: Theoretically, an in-memory code cave could be used to place redirection stubs within cryptdll.dll and this check would be bypassed, but no malware—in the wild or proof-of-concept—has leveraged this approach. Furthermore, the PDB-based method and the one described in the next section would still detect these, rendering them not particularly stealthy. These types of attacks are also much less portable to differing operating system versions, which is one of the reasons they are uncommon in the real world.

Adding Scanning as a Last Resort to *windows.skeleton_key_check*

We currently have two methods to gather the data needed for Skeleton Key attacks: PDB files and export table analysis. As discussed previously, the PDB file method can be unavailable for a number of reasons, and unfortunately, the export table method can be as well. The most common reason for this is the PE header metadata being paged out or the page(s) holding the export table information are paged out. The end result is that we cannot use the export table to tell us directly where to look for our needed information.

In these situations, there is a long history of memory forensic tools *scanning* for the data they need. Since we have access to all pages that are present within a process's address space, we can simply scan them in hopes of finding what we need. In the case of our Skeleton Key plugin, we were able to develop a highly effective and efficient scanner to meet our needs.

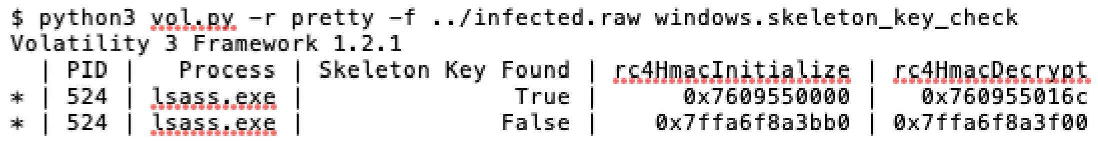
To begin, we used our knowledge that the data we need is contained within cryptdll.dll. This means we only have to scan a very small space (the size of the DLL). Second, as shown before, the layout of the active structure starts with the integer for the encryption type, which we know is 0x17 for RC4. Other research showed that the second member, *BlockSize*, had a value of 1 in all of our test samples. Using this knowledge, we developed a scanner based on Volatility 3's scanning API:



```
ecrypt_size = cryptdll_types.get_type("_KERB_ECRYPT").size

for address in proc_layer.scan(self.context,
                               scanners.BytesScanner(b"\x17\x00\x00\x00\x01\x00\x00\x00"),
                               sections = [(cryptdll_base, cryptdll_size)]):

    # this occurs across page boundaries
    if not proc_layer.is_valid(address, ecrypt_size):
        continue

    kerb = cryptdll_types.object("_KERB_ECRYPT",
                                 offset = address,
                                 absolute = True)

    # ensure the Encrypt and Finish pointers are inside the VAD
    # these are not manipulated in the attack
    if (cryptdll_base < kerb.Encrypt < cryptdll_end) and \
       (cryptdll_base < kerb.Finish < cryptdll_end):

        csystems.append(kerb)

return csystems
```

The scanner is configured to look for an 8-byte pattern of 0x17 followed by 1 in little-endian integer format. It attempts to instantiate a *_KERB_ECRYPT* type at each address where this pattern is found. To strengthen the check, we also verify that the *Encrypt* and *Finish* members our potential structure reference addresses are inside of cryptdll.dll. Neither of these are targeted by Skeleton Key attacks and validating their values provides a strong check against false positives.

The following shows the output of our plugin when the scanning method is used:



```
$ python3 vol.py -r pretty -f ../infected.raw windows.skeleton_key_check
Volatility 3 Framework 1.2.1
  | PID |   Process | Skeleton Key Found | rc4HmacInitialize | rc4HmacDecrypt
* | 524 | lsass.exe |               True |      0x7609550000 |   0x760955016c
* | 524 | lsass.exe |              False |    0x7ffa6f8a3bb0 | 0x7ffa6f8a3f00
```

Note that there are two lines of output. This occurs beause the scanner finds both the active version of the RC4 structure and the version that is statically compiled into the application. Having both outputs provides some advantages: there is direct visual confirmation that the active structure is hooked, and the statically compiled version reveals the addresses for the legitimate handlers, even without PDB usage.

Wrap Up

In this blog post, we have walked through the entire process for memory forensics research and development. We analyzed a target (Mimikatz’s Skeleton Key attack), analyzed the subsystem it abuses (the authentication systems managed by cryptdll.dll), and developed a new Volatility plugin that can automatically analyze this subsystem for abuse. This is a common workflow used to develop Volatility plugins.

If you find this type of research interesting, please consider developing a new plugin and submitting it to our Volatility Plugin Contest. Note that your submission does not have to be anywhere near as thorough as the plugin presented here; even submitting a new capability with just one of the discovery methods (PDB files, export analysis, scanning) used would be sufficient for an entry. We showed the full range here to display many of Volatility 3's new capabilities, but we certainly do not expect all plugins to meet this level of complexity.

We hope you have enjoyed this post. If you have any questions or comments, please let us know. You can find us on Twitter (@volatility) and our Slack server.

-- The Volatility Team