WINDOWS VISTA UIPI

User Interface Privilege Isolation

Speaker Info

- Edgar Barbosa
- Security researcher
- Currently employed at COSEINC
- Experience with reverse engineering of Windows kernel executables
- Published some articles at rootkit.com
- Participated in the creation of bluepill, a hardware-virtualization based rootkit

Windows Vista UIPI

- Shatter attacks
- Integrity Levels
- UIPI initialization
- Process and thread initialization
- Window messages
- UIPI in action
- Special cases

Why UIPI?

- Reasons to learn about UIPI internals:
  - It’s not well documented
  - The new book “Writing secure code for Windows Vista” dedicates only one page to discuss UIPI
  - UIPI affects several applications and API functions
  - There are some possible vectors of attacks against UIPI
  - To understand how windows messages are internally processed
  - Understand how some internal function works under the hood

Shatter attacks

- Published by Chris Paget in 2002
- Process by which one application could execute arbitrary code in another application
- Uses the window messaging system to send messages from an unprivileged app to the window message procedure of a privilege application.
- WM_TIMER could be sent with a callback pointer parameter

UIPI

- Stands for “User Interface Privilege Isolation”
- New for Windows Vista Operating System
- Vista now uses Integrity Levels for each running process:
  - 0x1000 - Low integrity
  - 0x2000 - Medium integrity
  - 0x3000 - High integrity
  - 0x4000 - System Integrity
- Each application runs with one assigned IL.

UIPI

- A lower privilege application cannot [1] :
  - Perform a window handle validation of higher process privilege
  - SendMessage or PostMessage to higher privilege application windows (block “Shatter attacks”)
  - Use thread hooks to attach to a higher privilege process
  - Use journal hooks to monitor a higher privilege process
  - Perform DLL injection to a higher privilege process

UIPI initialization

Windows Vista UIPI

UIPI initialization

- The UIPI initialization process occurs in the load time of the graphical subsystem module (win32k.sys)
- DriverEntry will call Win32UserInitialize function which will call:
  - InitUIPI (win32k)
    - RtlQueryElevationFlags (ntoskrnl)
  - InitClipFormatExceptionList (win32k)

UIPI initialization KUSER_SHARED_DATA

- Is a memory area shared between the user mode and kernel mode space
- Mapped for each running process
- Initialized by the OS executive - ntoskrnl
- It contains very important system variables:
  - KdDebuggerEnabled, SystemCall, TickCount
  - DbgElevationEnabled, DbgVirtEnabled, ….
- It is protected:
  - Read-only access for user mode code

UIPI initialization RtlQueryElevationFlags

- Undocumented function exported by ntoskrnl.exe and the ntdll.dll
- Prototype:
  - NTSTATUS RtlQueryElevationFlags(OUT ULONG *Flags);
- Retrieve UAC system information from the shared memory area (KUSER_SHARED_DATA)
- Return flags:
  - ElevationEnabled (0x01) - UAC is enabled
  - VirtEnabled (0x02) - Virtualization is enabled
  - InstallerDetectEnabled (0x04) - Detection of installers

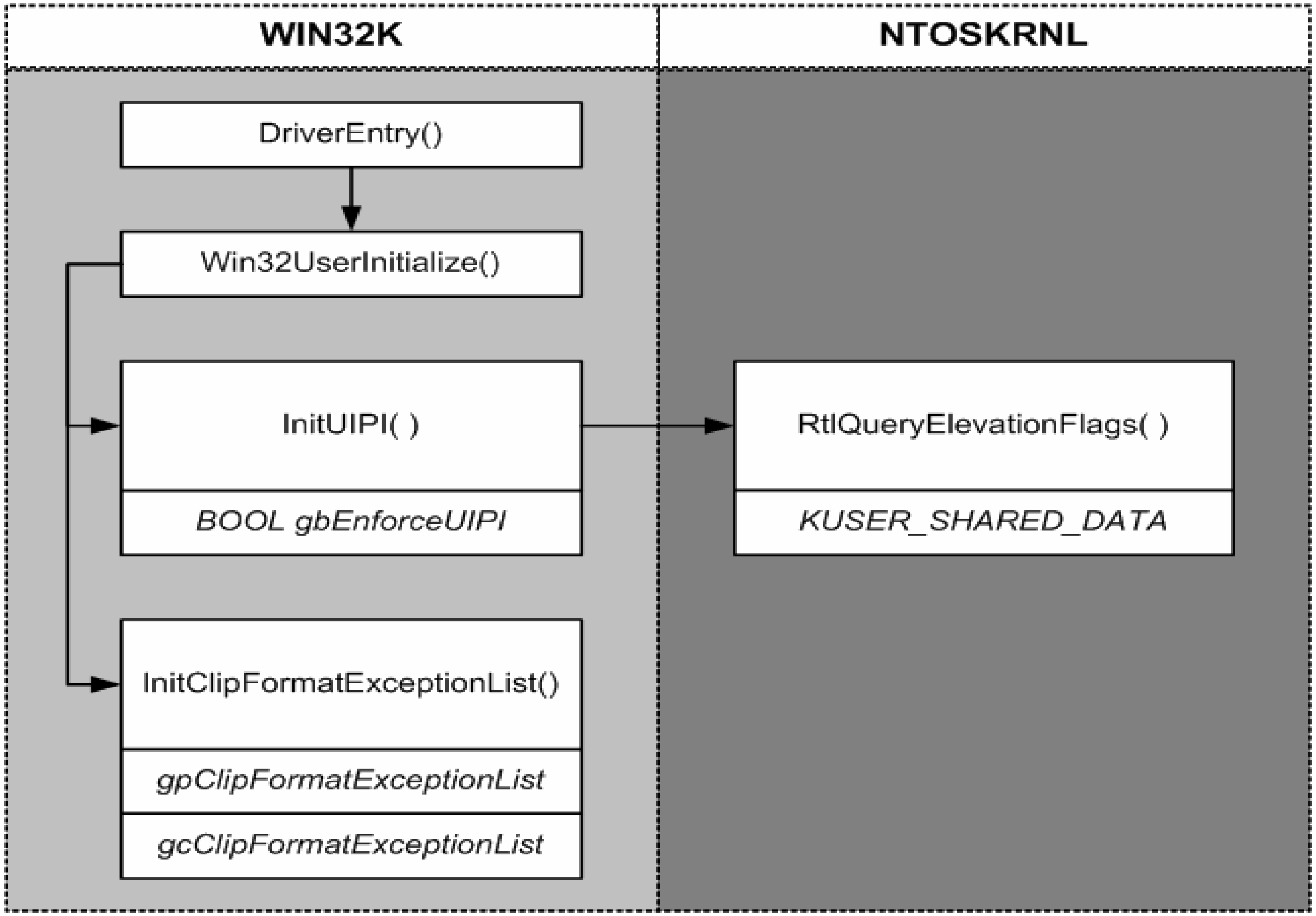
UIPI initialization VOID InitUIPI();

- UIPI is initialized by the InitUIPI function implemented inside the win32k kernel module
- Calls the RtlQueryElevationFlags to check if elevation is enabled (ElevationEnabled).
- If elevation is enabled, then check the value “EnableUIPI” inside the system policies registry key.
- If EnableUIPI value is set to true, the system set the global variable gbEnforceUipi to TRUE.
- UIPI is only active if gbEnforceUipi is equal TRUE.

UIPI initialization InitClipFormatExceptionList

- Called immediately after InitUIPI if gbEnforceUipi value is true.
- Responsible for reading the values of the registry key:
  - HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System\UIPI\Clipboard\ExceptionFormats
- gpClipFormatExceptionList:
  - Pointer to a list of Clipboard Format values, e.g. CF_OEMTEXT (0x07)
- gcClipFormatExceptionList:
  - Global counter of the number of elements inside the list pointed by the gpClipFormatExceptionList pointer.

UIPI Initialization


WIN32K
NTOSKRNL
DriverEntry()
Win32UserInitialize()
InitUIPI( )
BOOL gbEnforceUIPI
InitClipFormatExceptionList()
gpClipFormatExceptionList
gcClipFormatExceptionList
RtlQueryElevationFlags( )
KUSER_SHARED_DATA

Processes, threads and UIPI

Windows Vista UIPI

Process initialization and UIPI

- Data structures:
  - EPROCESS
  - PROCESSINFO
- Functions:
  - xxxInitProcessInfo()

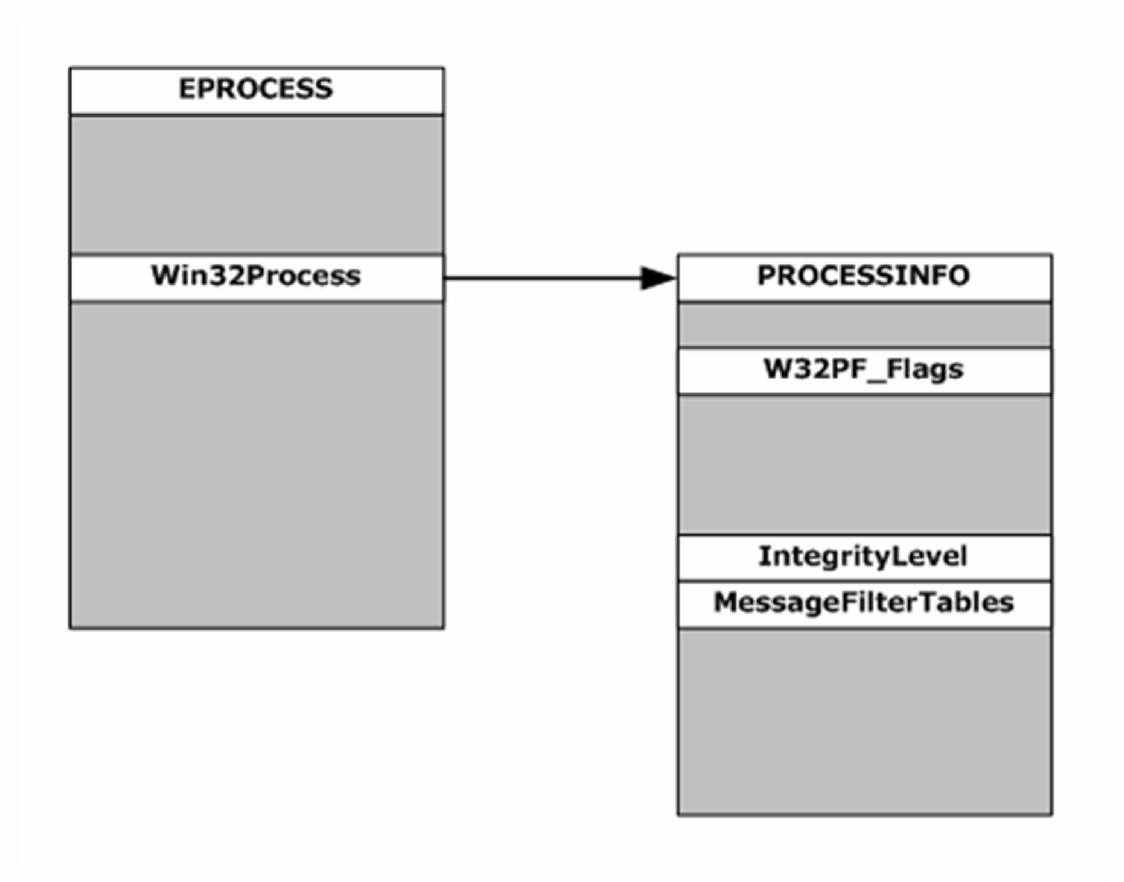
EPROCESS

- Each process is represented by an executive process object structure (EPROCESS)
- Accessible only for kernel mode code
- Some EPROCESS data information includes:
  - Handle table pointer
  - Virtual Address space Descriptors (VAD)
  - Access token
  - PROCESSINFO pointer

PROCESSINFO

- Structure used by win32k to store all the USER32 related information about a process.
- Created at the first time that the process calls a USER32 syscall and it is initialized by the xxxInitProcessInfo function
- Address stored at EPROCESS->Win32Process
- Data information includes:
  - Pointer to EPROCESS
  - Desktop, UserHandleCount, WinStation information
  - ….

EPROCESS and PROCESSINFO



xxxInitProcessInfo

- Prototype
  - NTSTATUS xxxInitProcessInfo(PROCESSINFO *pi);
- Responsible for the initialization of the PROCESSINFO structure
- Open the primary token of the process with PsReferencePrimaryToken and
- Get the Integrity Level of the process by calling SeQueryInformationToken with the TOKEN_INTEGRITY_LEVEL information class

xxxInitProcessInfo

- If gbEnforceUIPI == TRUE
  - Copy the Integrity Level value at pi->IntegrityLevel
- Check for the TokenUIAccess flag using SeQueryInformationToken function. If the flag is set, set the correspondent flag at pi->Flags
- The TokenUIAccess importance will be presented soon.

Thread initialization and UIPI

- Data structures:
  - ETHREAD
  - THREADINFO
- Functions:
  - xxxInitThreadInfo
  - NtUserCheckAccessForIntegrityLevel

ETHREAD

- Structure used by the kernel to represent a thread (ETHREAD)
- Each ETHREAD is always owned by an EPROCESS structure (ETHREAD->Tcb.Process)
- Linked to other threads by the ThreadListEntry pointer
- Some fields:
  - InitialStack, StartAddress, ThreadListEntry, ApcState
  - Priority, Affinity, THREADINFO, ...

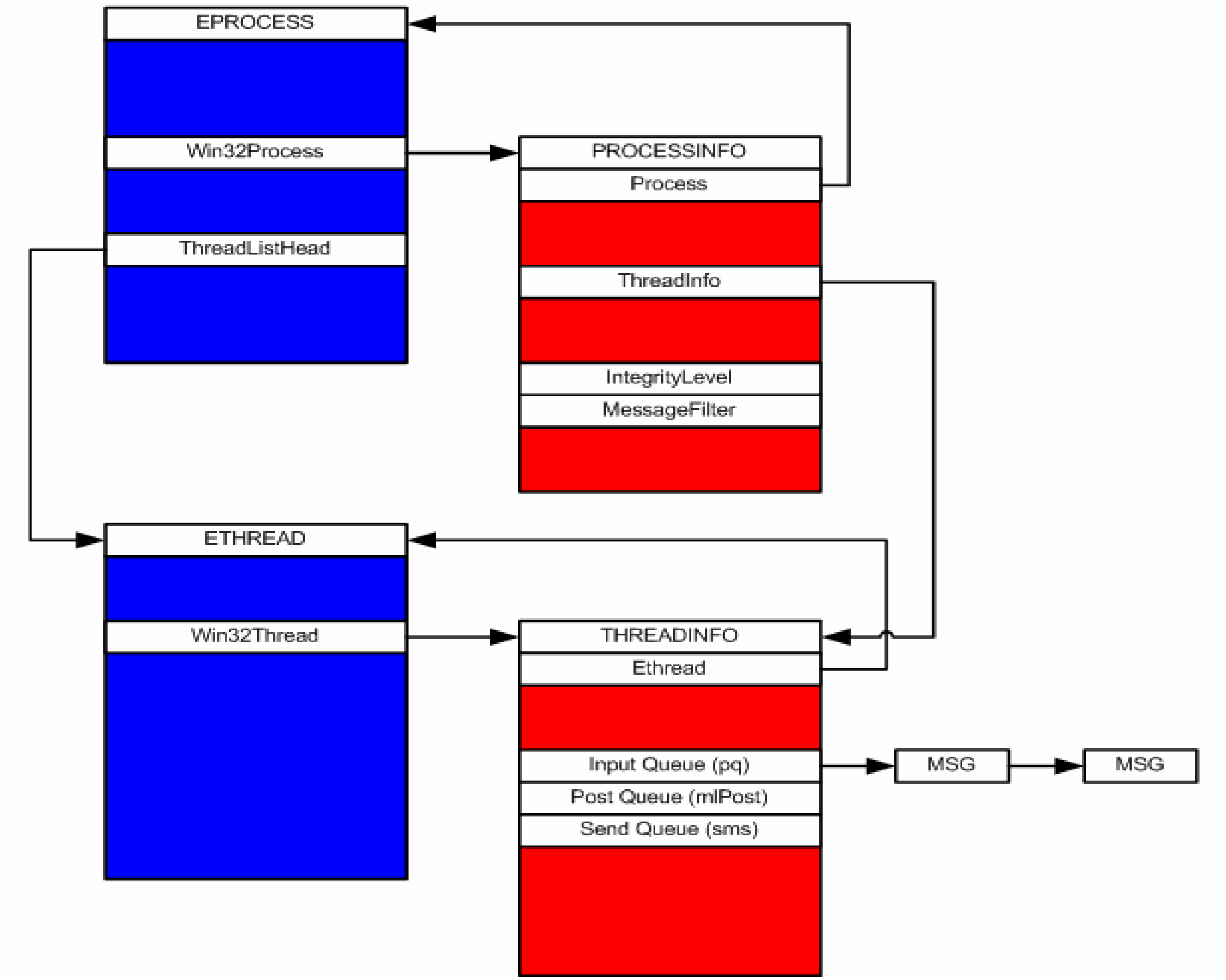
THREADINFO

- Undocumented structure used by win32k to store USER32 information related to a thread
- Created only if the thread calls a USER32 function
- THREADINFO address stored at ETHREAD->Tcb.Win32Thread
- Some fields:
  - pq (Input message queue)
  - mlPost (Post message queue)
  - Send message queue
  - Windows hook information...

xxxCreateThreadInfo()

- Function called to allocated and initialize the THREADINFO structure
- Some responsibilities:
  - Allocate and initialize the input, send and post message queues.
  - Set desktop
  - Set integrity level of the message queue
  - Set foreground priority

Kernel structures



NtUserCheckAccessForIntegrity Level

- Undocumented syscall which can be used by usermode programs (not officially supported)
- It's used by the USER32 usermode functions:
  - TileWindows()
  - CascadeWindows()
- NTSTATUS NtUserCheckAccessForIntegrityLevel (Pid1, Pid2, *BOOL Result);
- It checks the current process PROCESSINFO integrity level against the integrity level of the target process PROCESSINFO.

Window messages

Windows Vista UIPI

Window messages

- Data structures:
  - Message Queue
- Functions:
  - ChangeWindowMessageFilter

Window messages

- Each window has a window procedure.
- Window messages are used by the system to send events to a window procedure.
- Each windows, which is identified by a window handle (HWND), is always owned by a thread.
- But each thread can own more than one window
- Range 0x0 -> 0xffff (available to programmer)
- 0x10000 -> 0x1ffff (reserved to system)

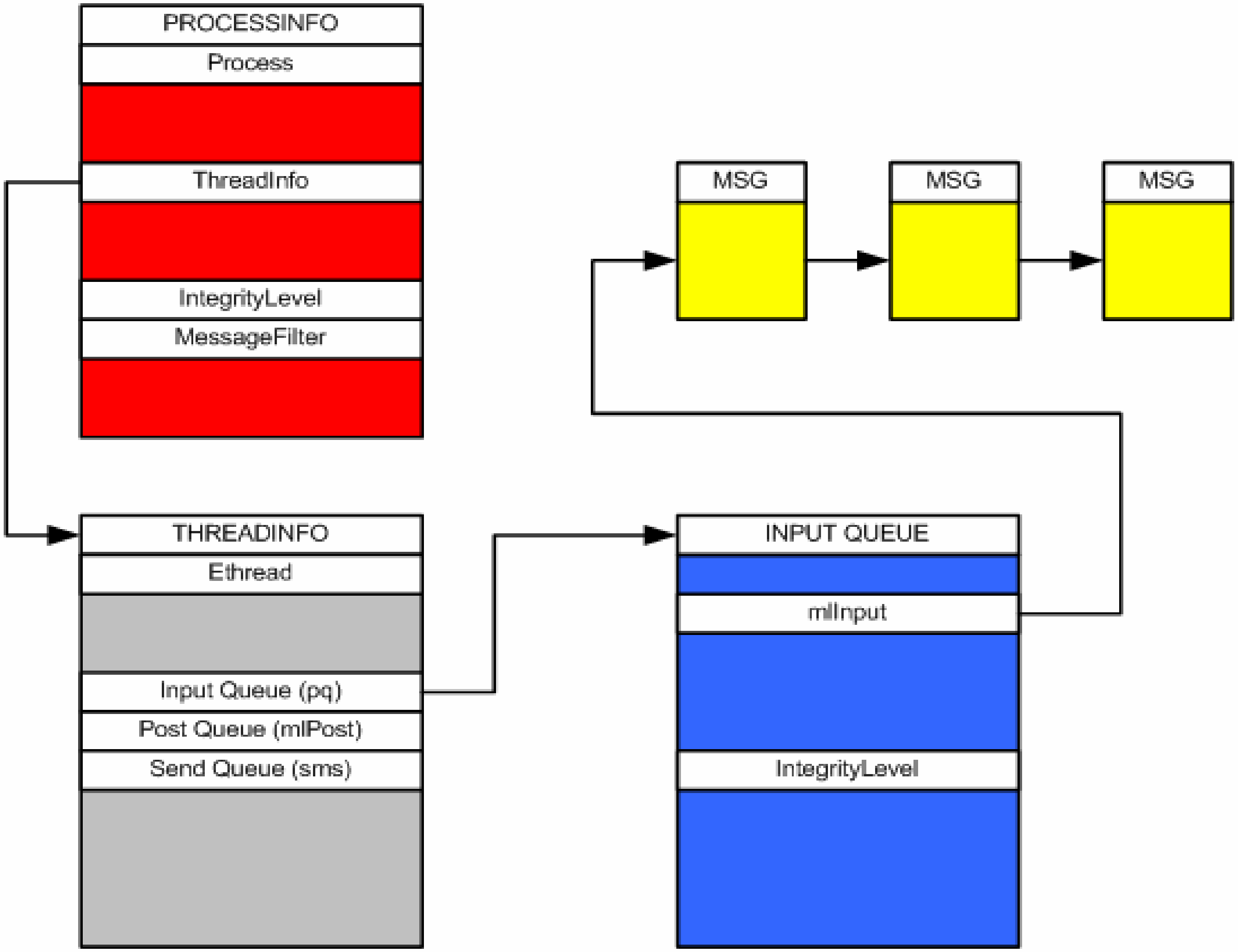
Message Queues

- Each received message is sent to a message queue
- The window message queues are implemented in the THREADINFO structure
- There are 3 queues:
  - Input queue (SendInput, mouse and keyboard msgs)
  - Post queue (PostMessage)
  - Send queue (SendMessage)
- The Input Queue structure, a.k.a Virtualized Input Queue (VIQ), Integrity Level field.

Message Queues

- The VIQ structure is allocated by the AllocQueue() function, which is called by the xxxCreateThreadInfo() function.
- AllocQueue() returns a pointer to the VIQ address.
- If the owner process of the Queue is the CSRSS process, the Queue->IntegrityLevel will be set equal 0x2000 (MEDIUM_INTEGRITY). If not, it will be set equal the PROCESSINFO->IntegrityLevel.

VIQ, IntegrityLevel and Messages



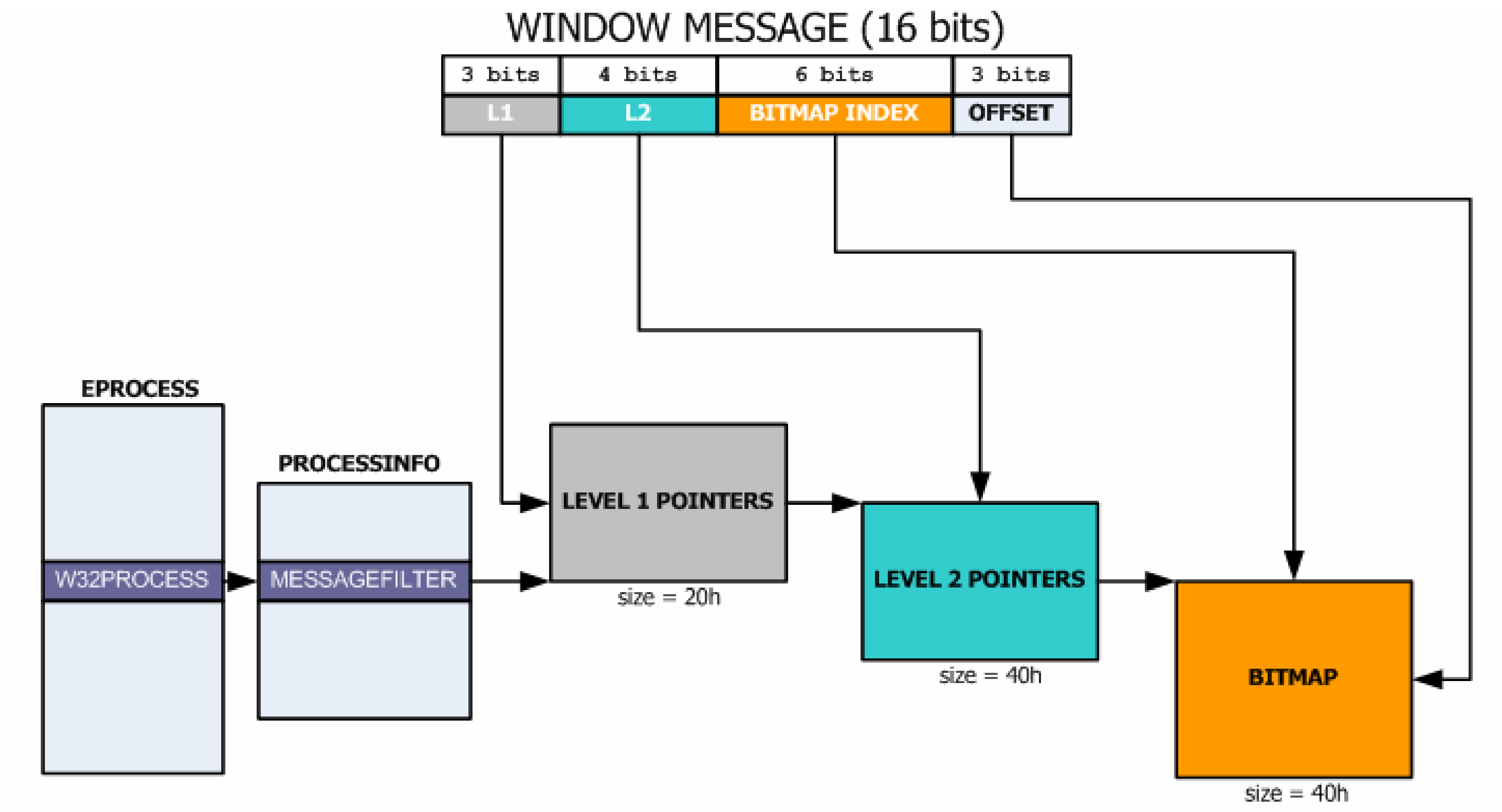
ChangeWindowMessageFilter

- Documented API directly related to UIPI
- New for Window Vista.
- Can be used to add or remove Window messages from the message filter
- If a message is added to the filter, any other process is able to send that message to the process, regardless of the integrity level of the processes.
- Prototype:
  - BOOL ChangeWindowMessageFilter(UINT msg, DWORD dwFlag);
- dwFlag:
  - MSGFLT_ADD: Adds the message to the filter
  - MSGFLT_REMOVE: Removes the message from the filter

Window Message Filter implementation

- Message Filter is implemented in the PROCESSINFO.
- Message queues are implemented per-thread, but all threads (windows) share the same Message Filter.
- Can’t be used by LOW_INTEGRITY processes!
- win32k!_ChangeWindowMessageFilter function is the real code responsible for the filter management.
- The filter is implemented using bitmap structures:
  - 0 = message is not allowed
  - 1 = message is allowed
- The address of the bitmap tables is stored at PROCESSINFO->MessageFilter

Message Filter implementation



WINDOW MESSAGE (16 bits)

3 bits	4 bits	6 bits	3 bits
L1	L2	BITMAP INDEX	OFFSET

UIPI in action

Windows Vista UIPI

UIPI in action

- How PostThreadMessage function works internally in Windows Vista?
- Functions:
  - PostThreadMessage
  - NtUserPostThreadMessage
  - AllowMessageAcrossIL
  - CheckForMessageAccessCrossIL
  - CheckAccessForIntegrityLevel

PostThreadMessage

- PostThreadMessage function posts a message to the message queue of the specified thread.
- It returns without waiting for the thread to process the message.
- It uses the syscall NtUserPostThreadMessage

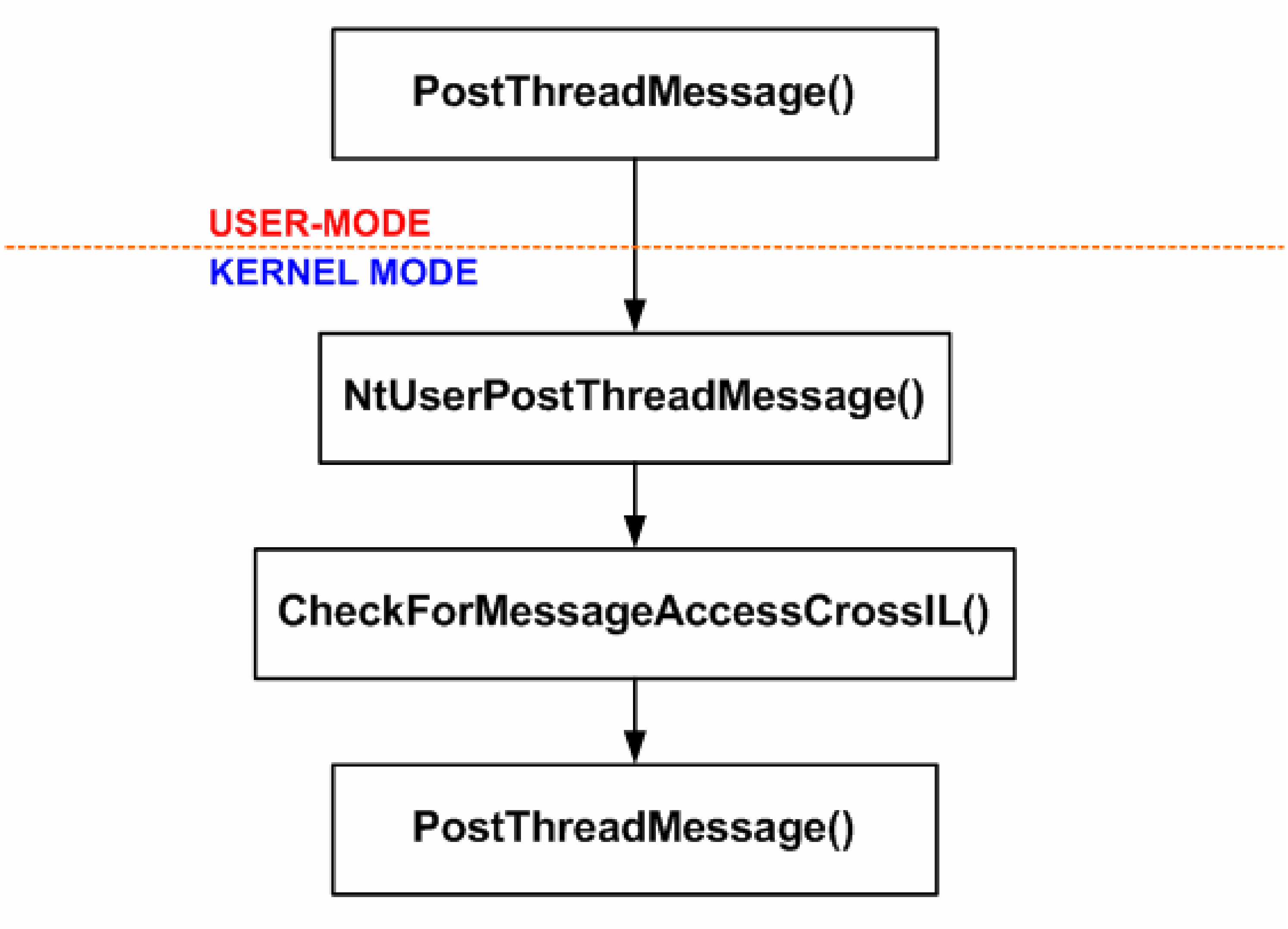
NtUserPostThreadMessage

- Kernel mode implementation of PostThreadMessage
- Prototype:
  - BOOL NtUserPostThreadMessage(DWORD idThread, UINT msg, WPARAM wParam, LPARAM lParam);
- Flow:
  - Get the address of the THREADINFO structure of the current thread and save it in the _gptiCurrent variable.
  - Get the THREADINFO of the target idThread using the PtiFromThreadId( ) function

NtUserPostThreadMessage

- If they are from the same desktop, it calls the **CheckForMessageAccessCrossIL**() function, which will verify if the destination thread allows the window message to be posted in the target thread post message queue.
- If the return code of CheckForMessageAccessCrossIL is TRUE, the PostThreadMessage is called
- PostThreadMessage allocates a new queue entry (*AllocQEntry*), stores the message (*StoreQMessage*) and set the wake message flags (*SetWakeBit*)

NtUserPostThreadMessage



CheckForMessageAccessCrossIL

- CheckForMessageAccessCrossIL is a complex function
- We will analyze now each function used by CheckForMessageAccessCrossIL:
  - CheckAccessForIntegrityLevel()
  - AllowMessageCrossIL()

CheckAccessForIntegrityLevel

- Internal win32k function extremelly used
- Prototype:
  - BOOL CheckAccessForIntegrityLevel(ULONG SourceIntegrityLevel, ULONG TargetIntegrityLevel);
- If the SourceIntegrityLevel < TargetIntegrityLevel, returns FALSE
- If SourceIntegrityLevel >= TargetIntegrityLevel, returns TRUE.

AllowMessageCrossIL

- Internal win32k function
- Prototype:
  - BOOL AllowMessageCrossIL(PROCESSINFO *pi, UINT msg);
- This function uses the MessageFilter bitmap structures created by the ChangeWindowMessageFilter function.
- Return:
  - If TRUE, the target thread will accept the message
  - If FALSE, doesn't mean nothing! Why?

CheckForMessageAccessCrossIL

- Prototype:
  - BOOL CheckForMessageAccessCrossIL (PROCESSINFO *piSource, PROCESSINFO *piTarget, UINT msg, WPARAM wParam);
- How it works?
  - It compares the piSource against piTarget. If equal, return TRUE. There’s no reason in checking threads of the same processinfo.
  - If different, calls the AllowMessageAcrossIL(piTarget, msg), but it do not check the return value immediately!

CheckForMessageAccessCrossIL

- It now compares the msg with a list of always allowed messages:
  - 0x000 - WM_NULL
  - 0x003 - WM_MOVE
  - 0x005 - WM_SIZE
  - 0x00D - WM_GETTEXT
  - 0x00E - WM_GETTEXTLENGTH
  - 0x033 - WM_GETHOTKEY
  - 0x07F - WM_GETICON
  - 0x305 - WM_RENDERFORMAT
  - 0x308 - WM_DRAWCLIPBOARD
  - 0x30D - WM_CHANGECBCHAIN
  - 0x31A - WM_THEMECHANGED
  - 0x313, 0x31B (WM_???)

CheckForMessageAccessCross IL

- If the message is in the list of always allowed, return TRUE immediately.
- If not, checks now the returned value of the CheckForMessageAccessCrossIL() . If equal true, then returns TRUE.
- If false, compare the current IL against the target IL using the CheckAccessForIntegrityLevel. If the current thread has a IL greater or equal than the target thread IL, returns TRUE.
- Check if CSRSS is the owner process of the target thread and allow the message if it is.

UIPI special cases

Windows Vista UIPI

Special cases

- CSRSS
- TokenUIAccess

CSRSS threads

- The CSRSS process is responsible for the creation of the console window for console-mode applications.
- Each console window is controled by a CSRSS thread (*ConsoleInputThread* function) inside winsrv.dll
- It registers the window class "ConsoleWindowClass"
- *ProcessCreateConsoleWindow* creates the console window
- *ConsoleWindowProc* is the window procedure.

CSRSS threads

- As exception, CSRSS threads that creates windows will have the THREADINFO->IntegrityLevel set equal 0x2000 (MEDIUM_INTEGRITY) by the xxxCreateThreadInfo function, regardless of the CSRSS IL.
- The address of the PROCESSINFO structure of csrss process is stored at win32k.sys in the global variable _gpepCSRSS
- How is this related to UIPI?

CSRSS threads

- Windows owned by CSRSS threads are the great exception rule in UIPI!
- In February 2007, Joanna Rutkowska published in her blog [2] that is possible to send WM_KEYDOWN messages to a open Administrative Shell (cmd.exe) running at HIGH IL from a LOW IL program.
- This is not only possible with cmd.exe, but with any console mode program running regardless of the its integrity level.

CSRSS threads

- The CheckForMessageAccessCrossIL() and xxxInterSendMsgEx() functions checks if the process owner of the target thread is CSRSS (gpepCSRSS)
- If CSRSS is the owner, the message is sent or posted even if the IL of the source thread is lesser than the IL of the target thread.
- Message is allowed even if the target’s message filter do not allow the message.

TokenUIAccess

- TokenUIAccess is a new token flag (at enum TOKEN_INFORMATION_CLASS)
- At the PROCESSINFO initialization process, the system checks for the TokenUIAccess flag in the primary token of the process.
- If the flag is present, the TokenUIAccess is set in the Flags field of PROCESSINFO.
- This flag allows application to potentially override some UIPI restrictions [3]

MSCTF.DLL and UIPI

Windows Vista UIPI

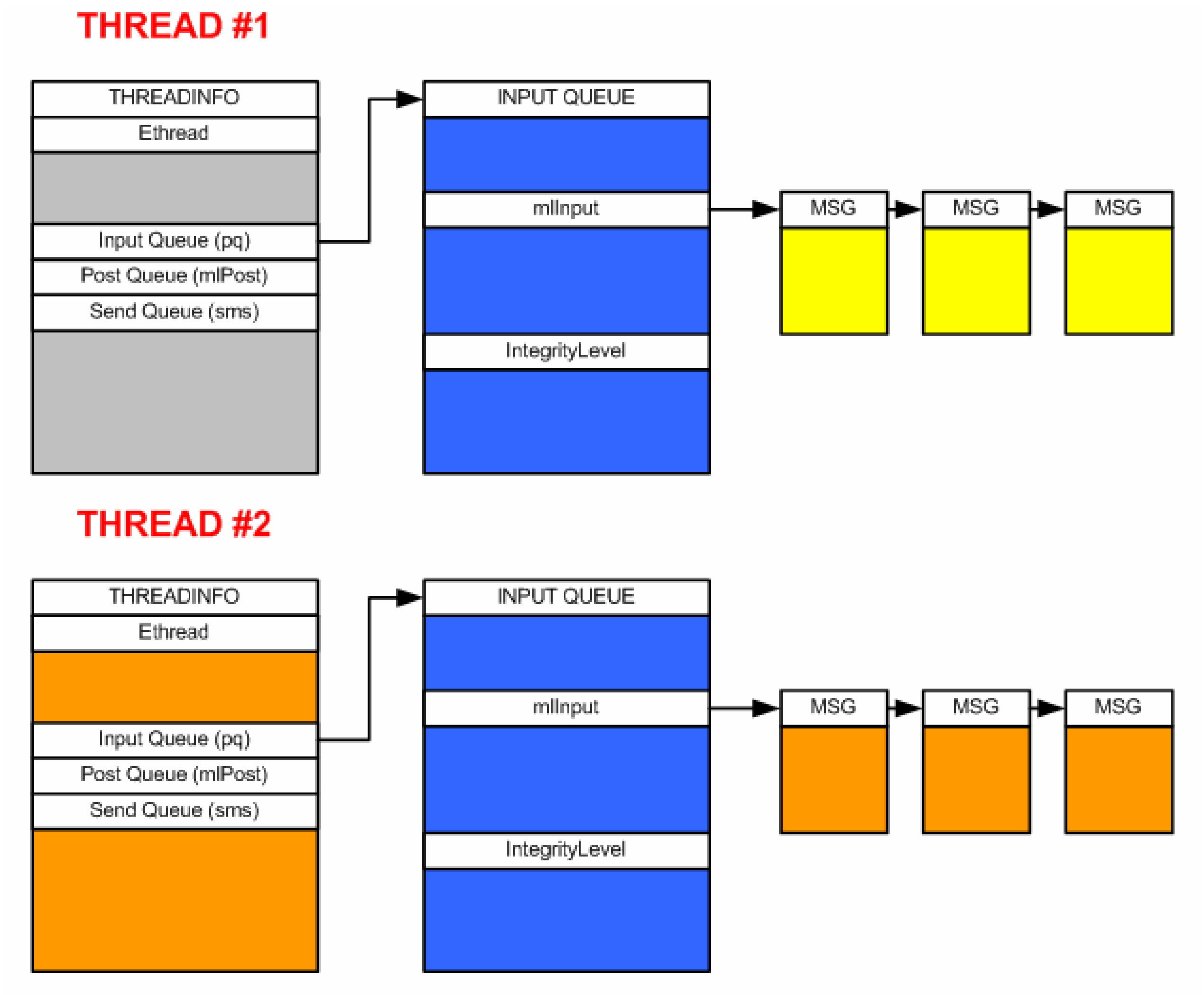
MSCTF.dll

- AttachThreadInput
- MSCTF.DLL
- DoS attack
- Queue integrity level elevation

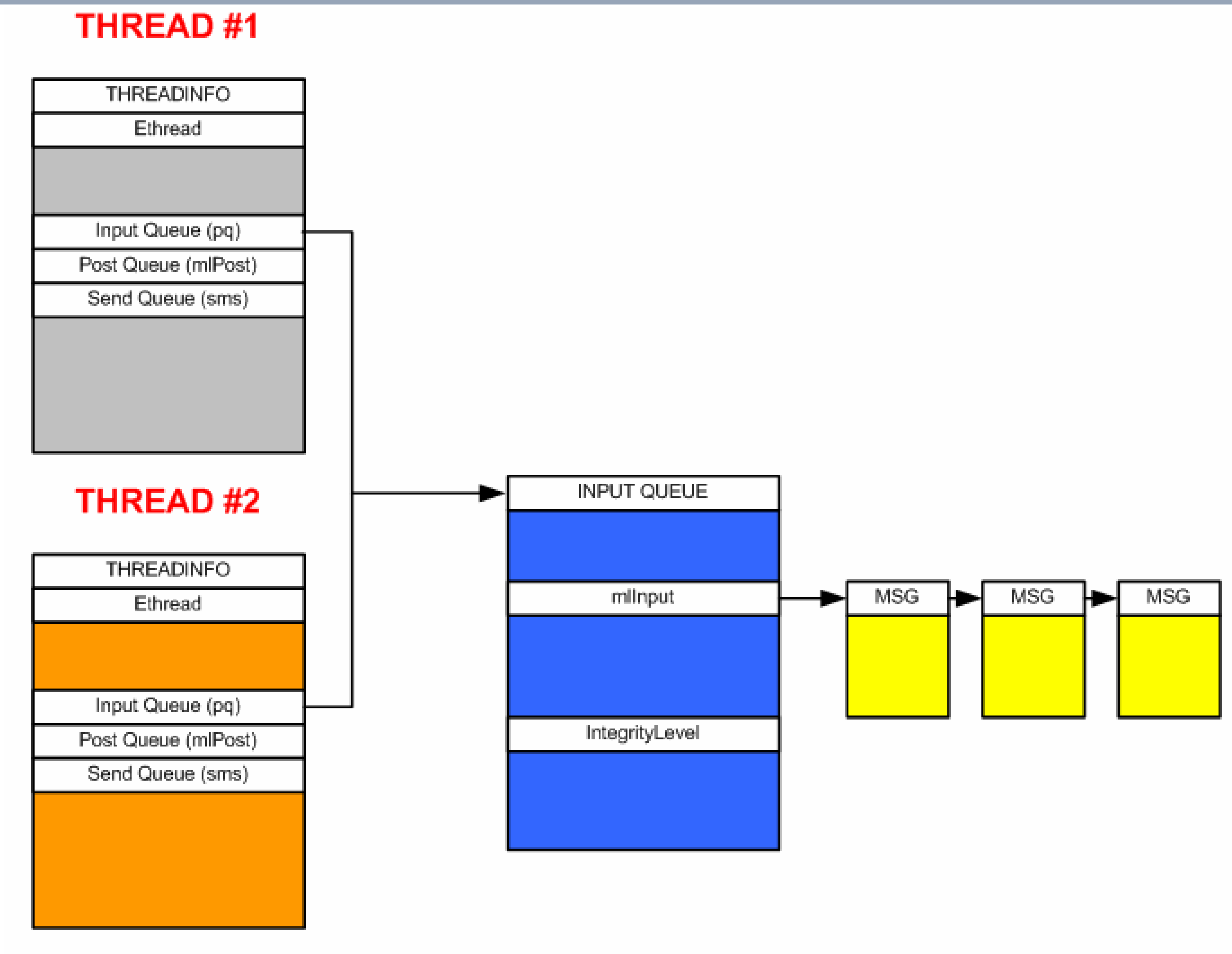
AttachThreadInput

- Input messages (keyboard, mouse, SendInput) are inserted at the Virtualized Input Queue of the THREADINFO.
- With the AttachThreadInput function, two threads can share their VIQ.
- Prototype:
  - BOOL AttachThreadInput(tidAttach, tidAttachTo, fAttach);
- When 2 thread share their input queues, input messages will be processed synchronously.
- If one thread hangs, the other thread will hang too.

Before AttachThreadInput



After AttachThreadInput

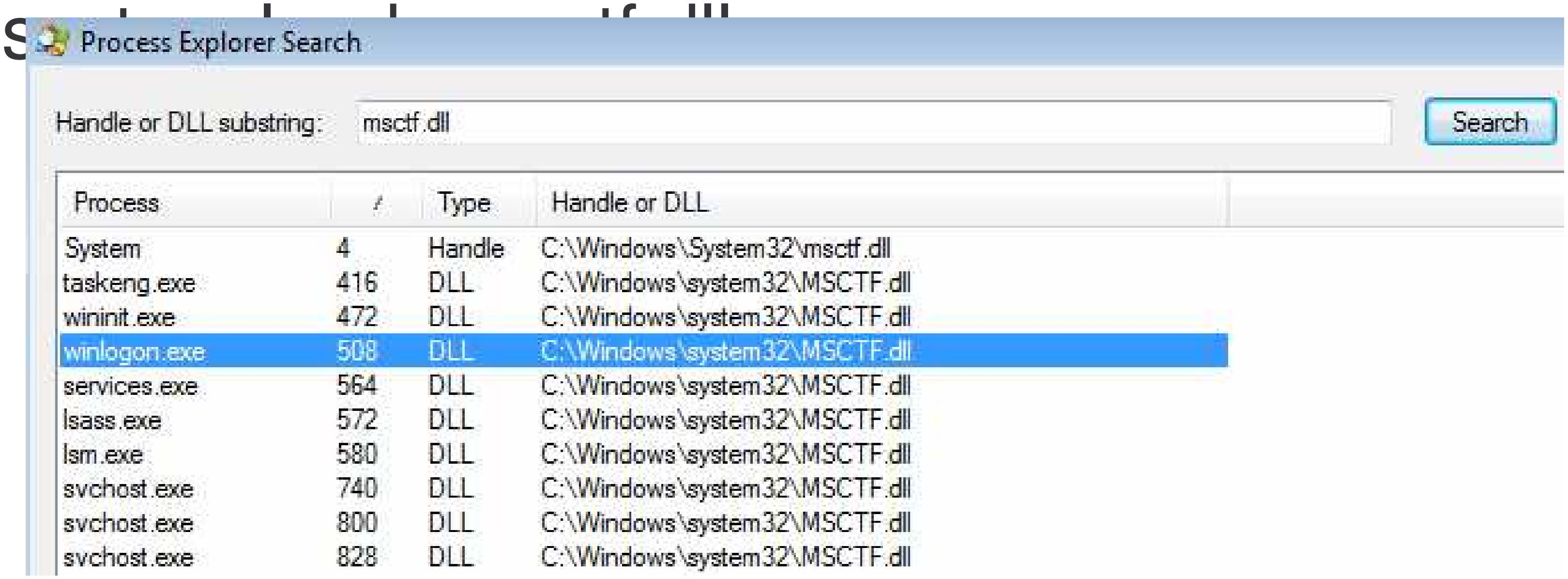


AttachThreadInput

- Must be used very carefully because it affects the robustness of the window message processing of the threads.
- NtUserAttachThreadInput is the syscall used by the user32 AttachThreadInput
- It is affected by UIPI, because the zzzAttachThreadInput function will check the Queue->IntegrityLevel of calling thread against the ProcessInfo->IntegrityLevel of the target thread.

MSCTF.DLL

- There are a DLL located in the system32 folder called MSCTF.dll which is used by the user32.dll.
- Practically all running programs in the Vista s



Process Explorer Search

Handle or DLL substring: msctf.dll

Process		Type	Handle or DLL
System	4	Handle	C:\Windows\System32\msctf.dll
taskeng.exe	416	DLL	C:\Windows\system32\MSCTF.dll
wininit.exe	472	DLL	C:\Windows\system32\MSCTF.dll
winlogon.exe	508	DLL	C:\Windows\system32\MSCTF.dll
services.exe	564	DLL	C:\Windows\system32\MSCTF.dll
lsass.exe	572	DLL	C:\Windows\system32\MSCTF.dll
lsm.exe	580	DLL	C:\Windows\system32\MSCTF.dll
svchost.exe	740	DLL	C:\Windows\system32\MSCTF.dll
svchost.exe	800	DLL	C:\Windows\system32\MSCTF.dll
svchost.exe	828	DLL	C:\Windows\system32\MSCTF.dll

MSCTF.DLL

- MSCTF.dll is one of the few DLLs in Vista that uses the new ChangeWindowMessageFilter() function.
- How this new API is being used by the MSCTF?
- The first step is to call the internal function EnsurePrivateMessages()
- EnsurePrivateMessages() internals:
  - Uses the USER32!RegisterWindowMessage function to register several window messages
  - RegisterWindowMessage takes as parameter a string and returns a window message that is guaranteed to be unique througout the system.
  - It now uses the returned window message and calls ChangeWindowMessageFilter() function

MSCTF.DLL

- Example:

```
gAttMsg =
  RegisterWindowMessage(“MSUIM.Msg.AttachThreadInput”);
If (gAttMsg)
    ChangeWindowMessageFilter( gAttMsg, MSGFLT_ADD );
```

- After EnsurePrivateMessages(), MSCTF will register a window class “CicMarshallWndClass” with the window procedure CicMarshallWndProc();
- Where is the problem?
  - The problem is inside the CicMarshallWndProc() code responsible for the processing of gAttMsg message

MSCTF.DLL

- The code is:

```
curThread = GetCurrentThreadId();
AttachThreadInput( curThread, wParam, lParam);
```

- If a low integrity thread calls AttachThreadInput against a higher integrity thread, the call will fail.
- But now, if the higher IL thread uses the MSCTF.dll, we can simply send a gAttMsg message and the target thread will call AttachThreadInput for us
- SendMessage(targetwnd, gAttMsg, GetCurrentThreadId(),TRUE);

DoS attack

- Using the AttachThreadInput message created by msctf.dll, we can create a Denial of Service tool which will hang all application running on the system with the msctf.dll loaded.
- Due to the fact that attach thread processes input messages synchronously, a low IL program is able to hang even higher integrity applications.
- One interesting consequence of attaching the virtualized input queue of two thread of different integrity levels is the elevation of the integrity level of the queue of the less privileged thread.

Queue integrity level elevation

- Each input queue (VIQ) has it’s own Integrity Level
- VIQ IL is assigned by xxxCreateThreadInfo function
- If a medium integrity level (0x2000) thread uses the Msctf AttachThreadInput message to attach to a high integrity level (0x3000) thread, the AttachThreadInput function (zzzAttachToQueue) will elevate the queue IL from 0x2000 to 0x3000

Final notes

- UIPI has some weird rules
- The always allowed window messages list is not public and documented
- We will probably see malwares using the UIPI exception rules, like the CSRSS case
- ChangeWindowMessageFilter function must be carefully used to avoid unprivileged processes to control privileged processes, like the MSCTF AttachThreadInput example.

References

- [1]http://blogs.msdn.com/vishalsi/archive/2006/11/30/what-is-user-interface-privilege-isolation-uipi-on-vista.aspx
- [2] http://theinvisiblethings.blogspot.com/2007/02/running-vista-every-day.html
- [3] M. Howard and D. LeBlanc - Writing Secure Code for Windows Vista - 2007. page 24.

Questions?

Thank you for your time!