Dissecting Windows XP Svchost Internals

[Craniology of RPC Svchost by Reverse Engineering]

Aditya K Sood aka 0kn0ck
Security Researcher
SecNiche Security
http://www.secniche.org

Foreward

This document serves to be as research layout of Sec Niche security. The analysis holds information that is specific in the context in which it is presented. I would like to thank Mr. ***Pedram Amini*** [Lead Researcher, Tipping Point, OpenRCE] for continuous discussion and support in completion of this analysis.

Author

Contents

Abstract

The paper solely relates to the core internals that build up the Windows XP Svchost. The Svchost internals have not been disseminated into informative elements yet. The anatomy of Svchost has got complexity in its own term. This pushes me to write a specific analysis over it. The analysis provides a structural design with concept wise dissection. The point is to understand the hidden artifacts and how it affects the working aspect of prime service host controller.

[2] Anatomy of Service Host (svchost) Process:

Every process is disseminated into primary process and secondary process. In terms related to operating system there is a parent process and its child. If one look at the implementation scenario then child processes are undertaken as thread internally. The kernel level implementation is subjugated like this. The parent process is always termed as services. Under it number of service threads is hosted as different child process. A unique thread Id is given to every individual There is no interdependency between threads. Thread run as a unique entity.

In windows XP SP2 the Svchost child process and parent process mainly runs under DEP ie Data Execution Protection. The DEP flag is always switched on. This is because DEP marks every single memory location in a process as non executable unless location explicitly contains executable code. It depends on processor hardware to define the working implementation of DEP. The DEP protects the process memory space to be exploited by malicious code insertion. The windows services and svchost is also protected by the DEP as DEP functions work on a per-virtual memory page basis, and DEP typically changes a bit in the page table entry (PTE) to mark the memory page. At first point we have concluded that svchost is protected by DEP.

The Svchost is termed as Generic Host process for win32 services. The second part we are going to analyze is the types of services are hosted under it before getting to the core. The svchost supports mainly runs as:

1] Local Service.
2] Network Services.
3] System.

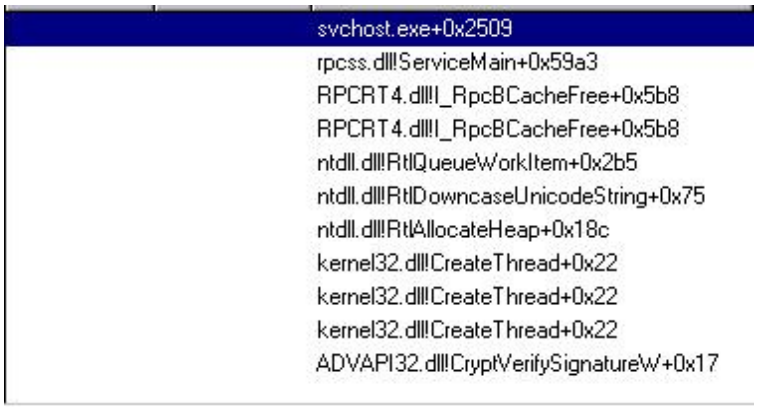
The above mentioned processes are undertaken as specific USER that are recognized by the OS internally.It runs under NT-Authority system. The OS provides a execution context entity in which local, system and network services are defined.The implementation is based on it. You can say these entities define the execution characteristic of the hosted services. Let's look at some of the examples:

```
C:\WINDOWS\System32\svchost.exe -k imgsvc [System]
C:\WINDOWS\system32\svchost -k DcomLaunch [System]
C:\WINDOWS\System32\svchost.exe -k NetworkService [Network Service]
C:\WINDOWS\system32\svchost -k rpcss [ Network Service]
C:\WINDOWS\System32\svchost.exe -k LocalService [ Local Service]
```

The hosted services are Image Service, Dcom , RPC , Network and Local Services. It is a structural view of implementing a service in Generic Host process.

[3] Dissecting Working Stature of Svchost:

Let's dissect it internally. At kernel level the Service process is divided into characteristic entities as Thread ID, State, Context Switch , Base priority and Dynamic priority. The thread identifier is used for its identification at kernel level. The state can be any Wait:Executive ,Wait:UserRequest etc. The Context Switching is intra process sharing of memory when one thread is free and handle of execution is shifted to another thread for some time. It is undertaken by a timer that sets the context switching time. The svchost thread execution working functionality is defined as base priority or dynamic priority. These all entities combine together to defined the svchost mechanism. For analysis I will disseminate the rpcss Network service to understand the Windows Svchost functionality.



This is the Starting Address structural view of RPC service hosted as svchost under Generic Host Controller. The very first thread is main svchost executable which loads and executes a specific service. The svchost executable provides memory to load the RPC service for execution as a service. The next part is Service Main handle for RPC to load it as a Network service. Firstly we will look at the stack trace of svchost.exe to understand the kernel level structures used in it. Lets see:

[Stack Trace: svchost.exe]

```
nkrnlpa.exe!KiUnexpectedInterrupt+0xf0
ntkrnlpa.exe!ZwYieldExecution+0x1900
ntkrnlpa.exe!ZwYieldExecution+0x196c
ntkrnlpa.exe!NtWriteFile+0x2b00
ntkrnlpa.exe!NtReadFile+0x580
ntkrnlpa.exe!KeReleaseInStackQueuedSpinLockFromDpcLevel+0xb14
ntdll.dll!KiFastSystemCallRet
ADVAPI32.dll!SetServiceStatus+0x238
ADVAPI32.dll!SetServiceStatus+0xcc
ADVAPI32.dll!StartServiceCtrlDispatcherW+0x8b
svchost.exe+0x2585
```

At first the kernel process ntkrnlpa performs the execution initialization function at kernel level. It also defines the handling of exception interrupts. The kernel has to perform some base level functions for secondary execution of defined modules. Afterwards the Advapi32.DLL calls certain services specific modules for checking service status and Dispatch functions. The Control Dispatcher is used to send control messages to service execution queue for the requests being generated. The dispatching is always considered as call back routine for managing requests through defined service handles.

Now we will look at the stack trace of Service Main to see the internal module calling.

```
ntkrnlpa.exe!KiUnexpectedInterrupt+0xf0
ntkrnlpa.exe!ZwYieldExecution+0x1900
ntkrnlpa.exe!ZwYieldExecution+0x196c
ntkrnlpa.exe!KeReleaseInStackQueuedSpinLockFromDpcLevel+0xb14
ntdll.dll!KiFastSystemCallRet
rpcss.dll!ServiceMain+0x453b
kernel32.dll!GetModuleFileNameA+0x1b4
```

The above trace is the Service Main starting routine. If you see then at the end a module from kernel32.dll is called. **GetModuleFileNameA.** This function retrieves the fully qualified path for the file that contains the specified module that a current process owns. The function is undertaken as:

```
DWORD GetModuleFileName(
                HMODULE hModule,
                LPTSTR lpFilename,
                DWORD nSize
        );
```

So this function is called mainly to load a specific module related to a specific service. Now we will look at the service main module crafted in Microsoft API. The code somewhat looks like this:

```
void WINAPI service_main(DWORD dwArgc, LPTSTR *lpszArgv)
  {
        // register our service control handler:
     sshStatusHandle = RegisterServiceCtrlHandler( TEXT(SZSERVICENAME),
        service_ctrl);

     if (!sshStatusHandle)
      goto cleanup;
    // SERVICE_STATUS members that don't change in example
```

```
ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
ssStatus.dwServiceSpecificExitCode = 0;
// report the status to the service control manager.

    if (!ReportStatusToSCMgr(
            SERVICE_START_PENDING, // service state
            NO_ERROR, // exit code 3000)) // wait hint
goto cleanup;

ServiceStart( dwArgc, lpszArgv );
cleanup:

    if (sshStatusHandle)
          (VOID)ReportStatusToSCMgr(
          SERVICE_STOPPED,
          dwErr, 0);

                return;}
```

This provides a structural view and implementation scenario of Service Main.We have talked about Service control Dispatcher functions. Now lets see through API code how the dispatching routine is called.

```
     void __cdecl main(int argc, char **argv)
{
   char opt;
   SERVICE_TABLE_ENTRY dispatchTable[] =
   {
     { TEXT(SZSERVICENAME), (LPSERVICE_MAIN_FUNCTION)service_main},
     { NULL, NULL}
   }; // Code

    if (!StartServiceCtrlDispatcher(dispatchTable))
  AddToMessageLog(TEXT("StartServiceCtrlDispatcher failed.")); }
```

The Service control manager plays a crucial role in service implementation. This is because what ever the intermediate messages like controlling the services are handle by the control manager. Killing and stopping of service have to be reported to service control manager prior to perform that function. When ever a debugging operation is undertaken the SC manager is notified.

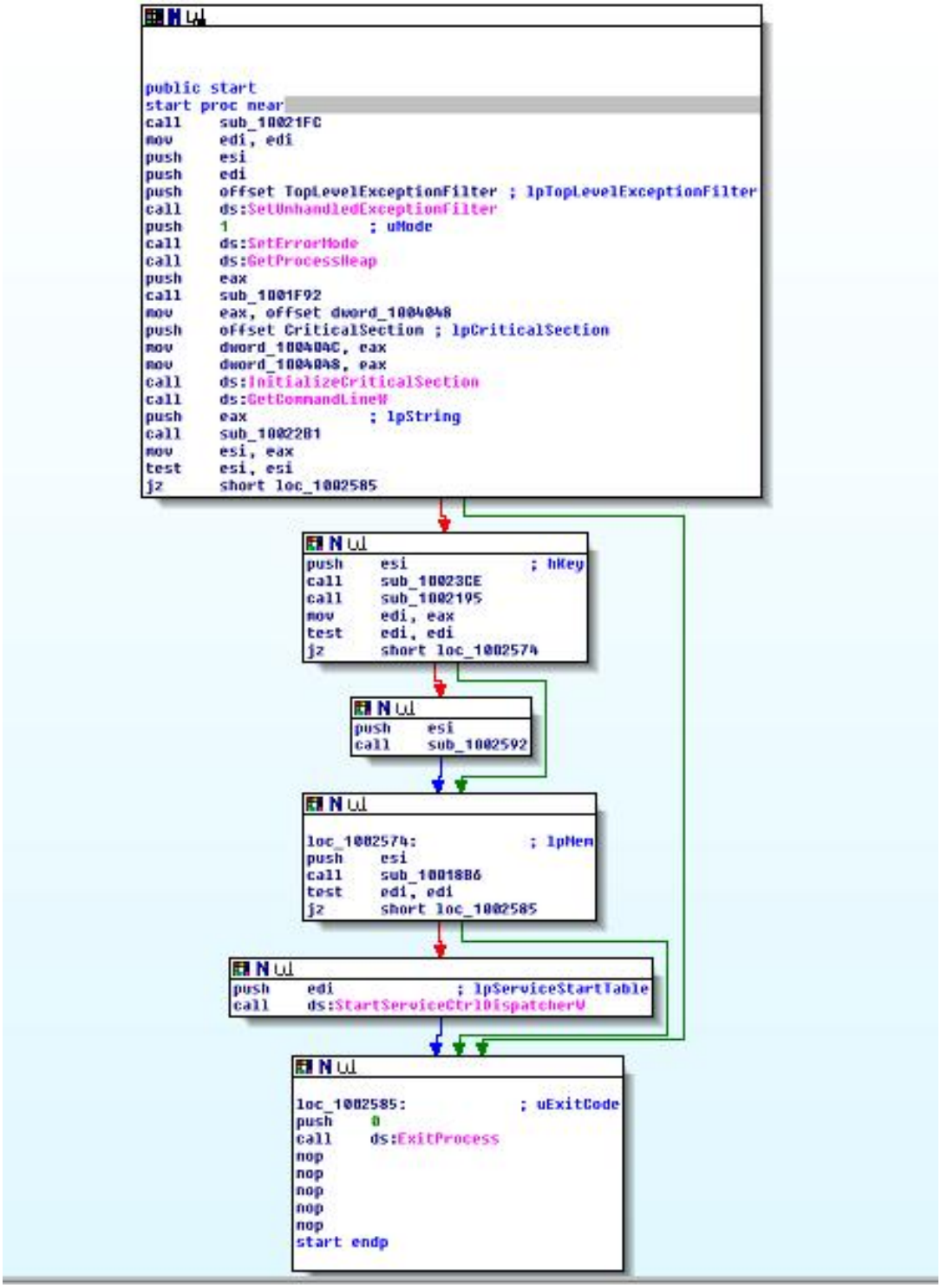
Let's look at the code:

```
BOOL ReportStatusToSCMgr(DWORD dwCurrentState,
DWORD dwWin32ExitCode,
DWORD dwWaitHint)

    { static DWORD dwCheckPoint = 1;

BOOL fResult = TRUE;
if ( !bDebug ) // when debugging we don't report to the SCM
 {
     if (dwCurrentState == SERVICE_START_PENDING)
     ssStatus.dwControlsAccepted = 0;
 else
     ssStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP;
     ssStatus.dwCurrentState = dwCurrentState;
     ssStatus.dwWin32ExitCode = dwWin32ExitCode;
     ssStatus.dwWaitHint = dwWaitHint;

    if (
         ( dwCurrentState == SERVICE_RUNNING ) ||
    ( dwCurrentState == SERVICE_STOPPED ) )

    ssStatus.dwCheckPoint = 0; else
    ssStatus.dwCheckPoint = dwCheckPoint++;

if (!(fResult = SetServiceStatus( sshStatusHandle, &ssStatus)))
        {
       AddToMessageLog(TEXT("SetServiceStatus")
         );

  }
 }

   return fResult;

}
```

The above derived C codes are presented to understand the working realm of SCManger and Control dispatcher.

[4] IDAG Cross Structural Dissection of Svchost.

Now the Swiss army knife IDA is undertaken. We are going to look at the work flow of svchost.exe. The graph will provide a overall interdependency of various modules. Let's have a look over it.



The flow chart depicts the same structure which we have enumerated above. We are going to enumerate some of the internal code used by the svchost executable to monitor more over the incore internals. Step by Step we will analyze code and relative specification of it.

[5] Disassembling Svchost Registry Paradigm.

```
      .text:010011DE mov edi, edi
     .text:010011E0 push ebp ; ulOptions
     .text:010011E1 mov ebp, esp
     .text:010011E3 push ecx ; lpSubKey
     .text:010011E4 push ecx ; hKey
     .text:010011E5 push esi
 .text:010011E6 mov esi, ds:__imp__RegOpenKeyExW@20 ;    RegOpenKeyExW(x,x,x,x,x)
     .text:010011EC push edi ; hKey
     .text:010011ED lea eax, [ebp+hKey]
     .text:010011F0 push eax ; phkResult
     .text:010011F1 mov edi, 20019h
     .text:010011F6 push edi ; samDesired
     .text:010011F7 push 0 ; ulOptions
 .text:010011F9 push offset SubKey ; "System\\CurrentControlSet\\Services"
     .text:010011FE push 80000002h ; hKey
 .text:01001203 call esi ; RegOpenKeyExW(x,x,x,x,x) ; RegOpenKeyExW(x,x,x,x,x)
     .text:01001205 test eax, eax
     .text:01001207 mov [ebp+var_4], eax
     .text:0100120A jnz short loc_1001247
     .text:0100120C push ebx
     .text:0100120D lea eax, [ebp+phkResult]
     .text:01001210 push eax ; phkResult
     .text:01001211 push edi ; samDesired
     .text:01001212 push 0 ; ulOptions
     .text:01001214 push [ebp+phkResult] ; lpSubKey
     .text:01001217 push [ebp+hKey] ; hKey
 .text:0100121A call esi ; RegOpenKeyExW(x,x,x,x,x) ; RegOpenKeyExW(x,x,x,x,x)
     .text:0100121C test eax, eax
     .text:0100121E mov ebx, ds:__imp__RegCloseKey@4 ; RegCloseKey(x)
     .text:01001224 mov [ebp+var_4], eax
     .text:01001227 jnz short loc_1001241
     .text:01001229 push [ebp+arg_4] ; phkResult
     .text:0100122C push edi ; samDesired
     .text:0100122D push 0 ; ulOptions
     .text:0100122F push offset aParameters ; "Parameters"
     .text:01001234 push [ebp+phkResult] ; hKey
.text:01001237 call esi ; RegOpenKeyExW(x,x,x,x,x) ; RegOpenKeyExW(x,x,x,x,x)
     .text:01001239 push [ebp+phkResult] ; hKey
     .text:0100123C mov [ebp+var_4], eax
     .text:0100123F call ebx ; RegCloseKey(x) ; RegCloseKey(x)
     .text:01001241
.text:01001241 loc_1001241: ; CODE XREF: OpenServiceParametersKey(x,x)+49#j
     .text:01001241 push [ebp+hKey] ; hKey
      .text:01001244 call ebx ; RegCloseKey(x) ; RegCloseKey(x)
      .text:01001246 pop ebx
```

This code holds registry oriented operations. The code depicts that registry is being queried for some sub keys. Once the sub keys are verified, the service name is undertaken by the service controller to load that service in a system memory for execution. You must have seen presence of rpcss , dcom etc services in the registry sub keys. These are all represented as strings in the registry. It means the registry possesses a direct entry of service type to be loaded in memory for execution. The registry functions are used to check the authentic structure of these services. This is

done to check whether the operating system hold information or not. Look at the code presented as:

```
.text:010011F9 push offset SubKey ; "System\\CurrentControlSet\\Services"
.text:010011FE push 80000002h ; hKey
.text:01001203 call esi ; RegOpenKeyExW(x,x,x,x,x) ; RegOpenKeyExW(x,x,x,x,x)
.text:01001205 test eax, eax
```

A registry key is opened under: HKLM\\Software\\System\\CurrentControlSet\\Services. It is a real registry entry. When I enumerated this entry the strings undertaken are:

1] rpcss
2] dcom
3] imgsvc
4] Local service
5] termsvc

This proves our point as the specified services are run under Generic Host Controller with different Thread ID. The service host checks the service entries in the registry first before starting execution. Let's have a look at some of entries:

Type	Name
Key	HKLM\SYSTEM\ControlSet003\Services\WinSock2\Parameters\Protocol_Catalog9
Key	HKLM\SYSTEM\ControlSet003\Services\WinSock2\Parameters\NameSpace_Catalog5
Key	HKLM\SYSTEM\ControlSet003\Services\Tcpip\Linkage
Key	HKLM\SYSTEM\ControlSet003\Services\Tcpip\Parameters
Key	HKLM\SYSTEM\ControlSet003\Services\NetBT\Parameters\Interfaces
Key	HKLM\SYSTEM\ControlSet003\Services\NetBT\Parameters
Key	HKCR
Key	HKCR
Key	HKLM
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKU
Key	HKCR
Key	HKU
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKCR\CLSID
Key	HKCR
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKU
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKLM\SOFTWARE\Microsoft\COM3
Key	HKCR\CLSID
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Drivers32

A simple cross check is performed by querying rpcss service through sc tool. Lets see what the tool say:

```
SERVICE_NAME: rpcss
TYPE : 20 WIN32_SHARE_PROCESS
STATE : 4 RUNNING
(NOT_STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
WIN32_EXIT_CODE : 0 (0x0)

SERVICE_EXIT_CODE : 0 (0x0)
CHECKPOINT : 0x0
WAIT_HINT : 0x0
```

Another Example of Dcom

```
C:\tools>sc query dcomlaunch
SERVICE_NAME: dcomlaunch
TYPE : 20 WIN32_SHARE_PROCESS
STATE : 4 RUNNING
(NOT_STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
WIN32_EXIT_CODE : 0 (0x0)
SERVICE_EXIT_CODE : 0 (0x0)
CHECKPOINT : 0x0
WAIT_HINT : 0x0
```

So services are categorized over same parameters as queried above. This covers our analysis regarding registry operation used by svchost executable.

[6] Calling Kernel Land Modular Functions

```
.text:01001483 ; FUNCTION CHUNK AT .text:010029E2 SIZE 00000023 BYTES
.text:01001483
.text:01001483 mov edi, edi
.text:01001485 push ebp
.text:01001486 mov ebp, esp
.text:01001488 push esi
.text:01001489 mov esi, [ebp+arg_0]
.text:0100148C mov eax, [esi+8]
.text:0100148F test eax, eax


.text:01001491 jnz short loc_10014AA
.text:01001493 push 8 ; dwFlags
.text:01001495 push eax ; hFile
.text:01001496 push dword ptr [esi+0Ch] ; lpLibFileName
.text:01001499 call ds:__imp__LoadLibraryExW@12 ; LoadLibraryExW(x,x,x)
```

```
.text:0100149F test eax, eax
.text:010014A1 jz loc_10029EF
.text:010014A7 mov [esi+8], eax
.text:010014AA
.text:010014AA loc_10014AA: ; CODE XREF: GetServiceDllFunction(x,x,x)+E#j
.text:010014AA push edi
.text:010014AB push [ebp+lpProcName] ; lpProcName
.text:010014AE push eax ; hModule
.text:010014AF call ds:__imp__GetProcAddress@8 ; GetProcAddress(x,x)
.text:010014B5 mov edi, eax
.text:010014B7 test edi, edi
.text:010014B9 jz short loc_10014E1
.text:010014BB
.text:010014BB loc_10014BB: ; CODE XREF: GetServiceDllFunction(x,x,x)+63#j
.text:010014BB ; GetServiceDllFunction(x,x,x)+1567#j
.text:010014BB mov eax, edi
.text:010014BD pop edi
.text:010014BE
.text:010014BE loc_10014BE: ; CODE XREF: GetServiceDllFunction(x,x,x)+157D#j
.text:010014BE pop esi
.text:010014BF pop ebp
.text:010014C0 retn 0Ch
.text:010014C0 _GetServiceDllFunction@12 endp
```

This code itself is very much clear because a LoadLibrary and GetProcAddress function is used to call module addresses from remote location ie kernel level from the user land to perform certain functions that are defined in the required DLL to be used. This is bit easy code to understand.

[7] Disseminating Svchost Critical Section Object Usage

This part will present the working functionality of critical section. There are number of intermodular calls that are used for the implementation of critical section. Every single modular function provides specific functioning. Let's enumerate the modules:

```
OpenServiceParametersKey(x,x) .text 010011DE 00000072 R . . . B T .
MemAlloc(x,x) .text 010012B1 0000001B R . . . B T .
GetServiceMainFunctions(x,x,x,x) .text 010012D1 00000157 R . . . B . .
GetServiceDllFunction(x,x,x) .text 01001483 00000040 R . . . B T .
FindDll(x,x) .text 010015CA 0000005F R . . . B T .
AddDll(x,x,x) .text 0100162E 00000072 R . . . B T .
ServiceStarter(x,x) .text 010016A5 00000176 R . . . B . .
SvcNetBiosOpen() .text 0100182D 0000001E R . . . . . .
MemFree(x) .text 010018B6 0000001A R . . . B T .
SvcNetBiosInit() .text 01001A56 00000022 R . . . . . .
RpcpInitRpcServer() .text 01001A7D 00000015 R . . . . . .
InitializeSecurity(x,x,x,x) .text 01002717 00000056 R . . . B T .
SvcNetBiosStatusToApiStatus(x) .text 01002F38 0000007A R . . . B . .
```

```
SetLanaFlag(x) .text 01002FB7 0000002A R . . . B . .
LanaFlagIsSet(x) .text 01002FE6 0000002E R . . . B . .
SvcNetBiosReset(x) .text 01003019 00000081 R . . . B . .
DelayLoadFailureHook(x,x) .text 01003197 00000006 R . . . . . .
Netbios(x) .text 010031BF 00000006 R . . . . T .
RpcpStopRpcServer(x) .text 010031CA 0000004C R . . . B T .
RpcpStopRpcServerEx(x) .text 0100321B 0000004C R . . . B . .
```

The above stated functions use the Critical section in its implementation. The objects are mainly created as critical section objects which are further used for synchronization. It's better to traverse the code first.

```
; FUNCTION CHUNK AT .text:0100193C SIZE 00000019 BYTES
.text:010015CA
.text:010015CA mov edi, edi
.text:010015CC push ebp
.text:010015CD mov ebp, esp
.text:010015CF push ecx
.text:010015D0 and [ebp+var_4], 0
.text:010015D4 push esi ; lpString2
.text:010015D5 push edi ; lpString1
.text:010015D6 push offset _ListLock ; lpCriticalSection
.text:010015DB call ds:__imp__EnterCriticalSection@4 ; EnterCriticalSection(x)
.text:010015E1 mov esi, _DllList.text:010015E7 mov edi, offset _DllList
.text:010015EC cmp esi, edi
.text:010015EE jz short loc_1001615
.text:010015F0 push ebx
.text:010015F1 mov ebx, ds:__imp__lstrcmpW@8 ; lstrcmpW(x,x)
.text:010015F7
.text:010015F7 loc_10015F7: ; CODE XREF: FindDll(x,x)+48#j
.text:010015F7 push [ebp+lpString2] ; lpString2
.text:010015FA mov [ebp+var_4], esi
.text:010015FD push dword ptr [esi+0Ch] ; lpString1
.text:01001600 call ebx ; lstrcmpW(x,x) ; lstrcmpW(x,x)
.text:01001602 test eax, eax
.text:01001604 jz loc_100193C
.text:01001615 loc_1001615: ; CODE XREF: FindDll(x,x)+24#j
.text:01001615 push offset _ListLock ; lpCriticalSection
.text:0100161A call ds:__imp__LeaveCriticalSection@4 ; LeaveCriticalSection(x)
.text:01001620 mov eax, [ebp+var_4]
.text:01001623 pop edi
.text:01001624 pop esi
.text:01001625 leave
.text:01001626 retn 8
.text:01001626 _FindDll@8 endp
```

The critical section objects allow two or more threads to compete for a single resource. The above stated modules are applied in this aspect.Threads that are blocked are put into wait state by the kernel and is not released till the wait state is satisfied. The scheduler plays a crucial role in applying the synchronization objects. So once the state is satisfied the thread can continue its execution. The synchronization objects are suited to specific type of data structure.

A critical section is an optimized implementation of mutex. A mutex is an object that can only be acquired by one thread at any given moment. If any thread tries to acquire a mutex, when it is already owned by another thread will enter a wait state. To acquire mutex the previous thread has to release mutex or terminate itself for further control. Critical section is considered to be as logically identical to mutex. The only difference is that it is process private and is implemented mostly in user mode. The critical section object is managed by kernel's object manager and implemented in kernel mode. It means the system should switch to kernel mode prior to implement any function on these objects. As we know critical section is implemented in user mode so system only switches to kernel mode when condition wait is applied.

The critical section objects are implemented as:

```
#include <windows.h>
CRITICAL_SECTION cs; /* This is the critical section object -- once initialized, it cannot
                                be moved in memory */

/* Initialize the critical section -- This must be done before locking */
InitializeCriticalSection(&cs);

/* Enter the critical section -- other threads are locked out */
EnterCriticalSection(&cs);

/* Do some thread-safe processing! */

/* Leave the critical section -- other threads can now EnterCriticalSection() */
LeaveCriticalSection(&cs);

/* Release system object when all finished -- usually at the end of the cleanup code */
DeleteCriticalSection(&cs);
```

The stated functions are used for implementation of critical section objects. As we have already discussed earlier that svchost threads run under services process with unique Thread ID. So a critical section object is implemented by every single svchost thread used for rpcss, dcom, termsvc etc. The critical objects provide a faster mechanism for mutual exclusion synchronization because the parent process is same for all svchost threads. The critical section objects are implemented in user mode .

A critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. When a thread owns a critical section, it can make additional calls to EnterCriticalSection or TryEnterCriticalSection without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. Any thread of the process can use the DeleteCriticalSection function to release the system resources that are allocated when the critical section object is initialized. After this function is called, the critical section object cannot be used for synchronization.

Let’s have look at the snapshot as:

This layout presents a number of svchost threads that run under services process. So svchost with Thread ID 1212, 1368, 1432 etc implements Critical section objects. The objects are called in a definitive module designed in particular DLL. So the svchost uses a proper Critical Section concept. Have a look at the calling code:

```
        // Global variable
CRITICAL_SECTION CriticalSection;
void main()
 {
     // Initialize the critical section one time only.
    if (!InitializeCriticalSectionAndSpinCount(&CriticalSection, 0x80000400) )
    return;..

          // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection)
 }
    DWORD WINAPI ThreadProc( LPVOID lpParameter )
 {
.          // Request ownership of the critical section.
      EnterCriticalSection(&CriticalSection);
     // Access the shared resource.
    // Release ownership of the critical section.
LeaveCriticalSection(&CriticalSection); }
```

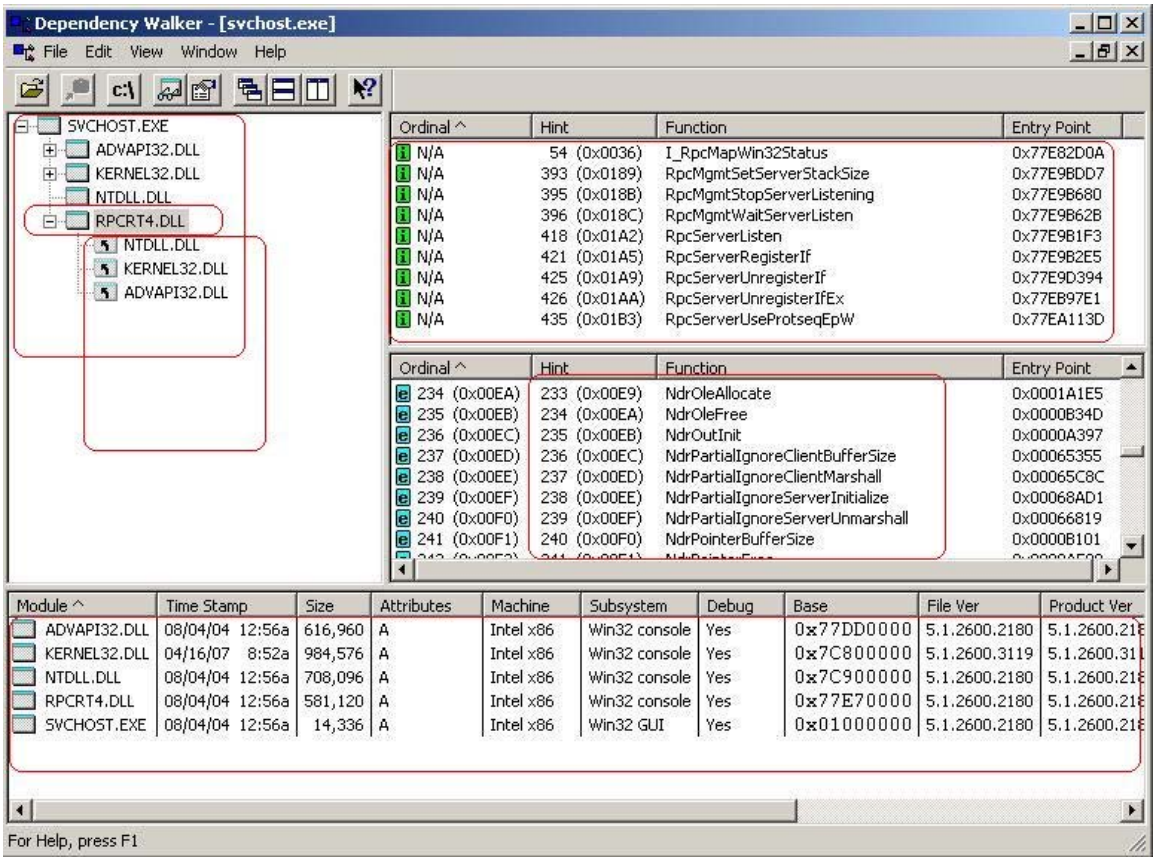
The implementations of code define the practical aspect of calling Critical Objects.

[8] Dependency Walking of RPC svchost Process

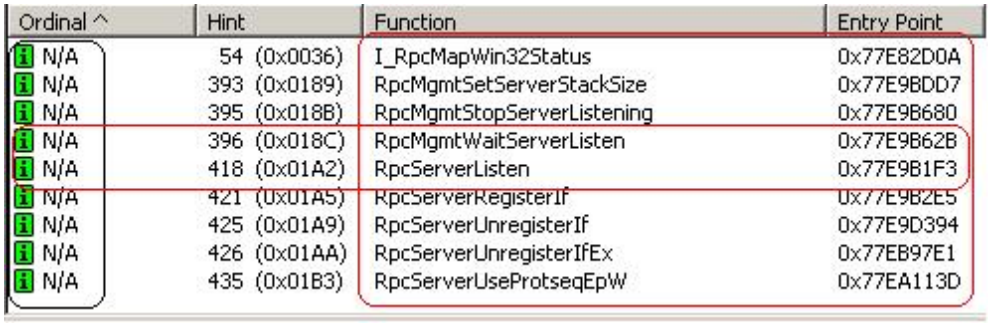
So we have dissected the basic realm of RPC svchost process. For further analysis we will look into dependency status of the svchost process. The dependency walking means to check various import and export functions that are used by RPC svchost. This technique is useful in looking at entry point addresses of various modules. Also one can look into the DLL's that are used by the process and inherited functions. The interdependencies of modules can be analyzed easily by looking at ordinal based import / export status. It leverages lot of information regarding svchost process. Why Dependency Walking to be checked? The answer is to crawl along the inter dependency of various modules and how the functionality is applied in a required

DLL. It also uncovers the inheritance aspect of RPC svchost process. If you look at the snapshot the RPCRT4.DLL holds the reference of NTDLL.DLL, KERNEL32.DLL and ADVAPI32.DLL

Our talk is entirely adheres to RPC svchost process. It's necessary to look into the functions used by this process in system context. The ordinal defines a standard number that maintains the export table. As soon as the function is exported an entry point is provided to it. This entry point holds the address through which that functions is called in a process. The svchost process is using the above mentioned Dynamic Link libraries to call certain modules. An export list of modules is provided. The functions present in module are exported by ordinal or by name.



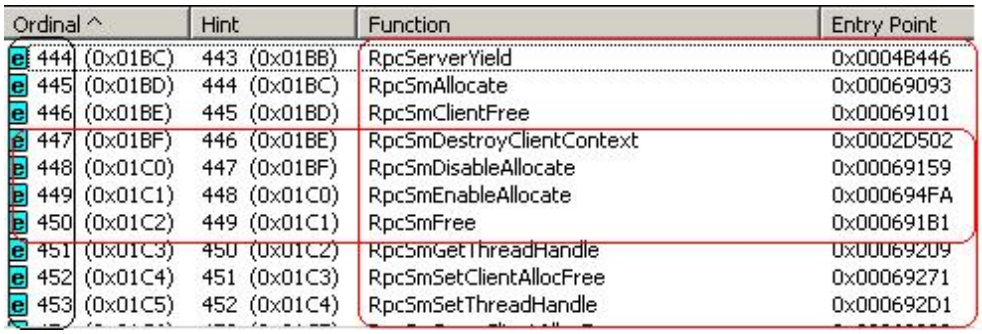
If one look at the tree of RPCRT4.dll then one can find number of functions is imported by name with [N/A] and some are with ordinal values. The functions present in the RPCRT4.dll are considered to be as Forwarded Functions because the real code is present in another module. The functions is referenced directly from a specific module.It provides a concept of Modular Interface. It means through an interface modules can be used in a cross reference manner between various dynamic link libraries. The functions like RpcServerListen, RpcServerRegisterIf, RpcServerUnregisterIf etc are imported by name and ordinal value is not applicable.



Ordinal ^	Hint	Function	Entry Point
N/A	54 (0x0036)	I_RpcMapWin32Status	0x77E82D0A
N/A	393 (0x0189)	RpcMgmtSetServerStackSize	0x77E9BDD7
N/A	395 (0x018B)	RpcMgmtStopServerListening	0x77E9B680
N/A	396 (0x018C)	RpcMgmtWaitServerListen	0x77E9B62B
N/A	418 (0x01A2)	RpcServerListen	0x77E9B1F3
N/A	421 (0x01A5)	RpcServerRegisterIf	0x77E9B2E5
N/A	425 (0x01A9)	RpcServerUnregisterIf	0x77E9D394
N/A	426 (0x01AA)	RpcServerUnregisterIfEx	0x77EB97E1
N/A	435 (0x01B3)	RpcServerUseProtseqEpW	0x77EA113D

These functions are termed as **Parent Module Functions** because of direct calling in RPCRT4.dll parent module. It sets dependency among various number of modules and usage. The every single function listed above has been provided with an entry point. It means the imported functions that are called by name consist of **PRE Specified Address** and the parent module is bounded by BIND program. If the entry point address is not defined, it means the address is not known until load time. The Bind defines an entry address.

The bind program scans the **Import Address Table [IAT]** and stores the most favorable entry point for each function in a module. This process is done to trigger up the Loading process. It is because a module is loaded directly from the preferred base address. Let's have a look at some of the exported functions in RPCRT4.dll.

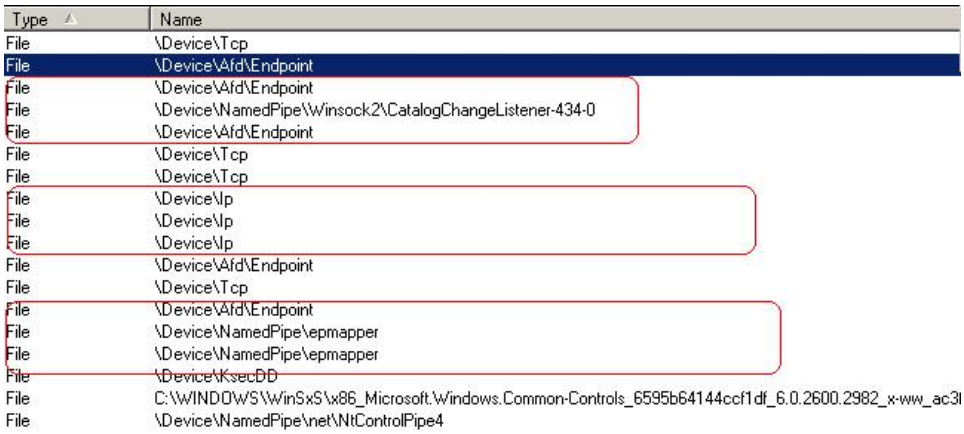


Ordinal ^	Hint	Function	Entry Point
444 (0x01BC)	443 (0x01BB)	RpcServerYield	0x0004B446
445 (0x01BD)	444 (0x01BC)	RpcSmAllocate	0x00069093
446 (0x01BE)	445 (0x01BD)	RpcSmClientFree	0x00069101
447 (0x01BF)	446 (0x01BE)	RpcSmDestroyClientContext	0x0002D502
448 (0x01C0)	447 (0x01BF)	RpcSmDisableAllocate	0x00069159
449 (0x01C1)	448 (0x01C0)	RpcSmEnableAllocate	0x000694FA
450 (0x01C2)	449 (0x01C1)	RpcSmFree	0x000691B1
451 (0x01C3)	450 (0x01C2)	RpcSmGetThreadHandle	0x00069209
452 (0x01C4)	451 (0x01C3)	RpcSmSetClientAllocFree	0x00069271
453 (0x01C5)	452 (0x01C4)	RpcSmSetThreadHandle	0x000692D1

The functions listed above are exported functions by ordinal. The **Entry Address** [0x0004B446] is specified for RpcServerYield function. It means address is hard coded prior to run time process. When a loader traverse through the **Export Address Table**, the base address for a RpcServerYield is already present and the loading is done with pace. So this process is repeated with other exported modules in RPCRT4.dll in a same manner. It enhances the functionality of address resolving and module loading. Another point if a forwarded string is present at entry point it means the function is forwarded to another module. The pattern of the string is [ModuleName.FunctionName]. As such no module forwarding is done right here. This explains the dependency status of RPC svchost process internally.

[9] Mapping EpMapper EndPoints – RPC svchost Inheritance

The system devices are represented as files. The RPC svchost process is dependent on certain system devices. These device define the inbuilt working of RPC svchost process related to TCP, Afd drivers etc. Let's have a look the system devices that RPC is using.

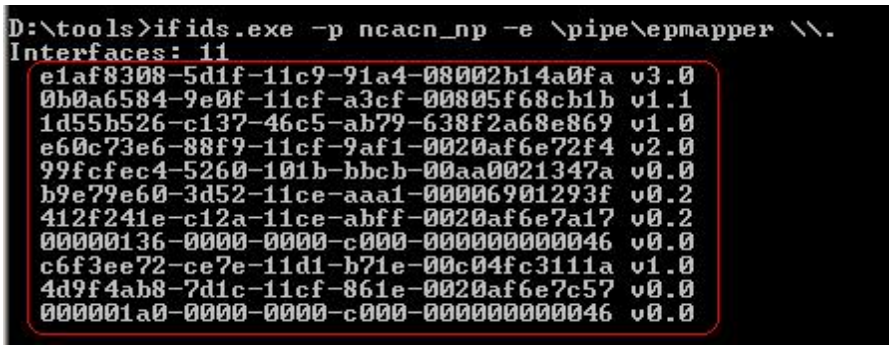


Type	Name
File	\Device\Tcp
File	\Device\Afd\Endpoint
File	\Device\Afd\Endpoint
File	\Device\NamedPipe\Winsock2\CatalogChangeListener-434-0
File	\Device\Afd\Endpoint
File	\Device\Tcp
File	\Device\Tcp
File	\Device\Ip
File	\Device\Ip
File	\Device\Ip
File	\Device\Afd\Endpoint
File	\Device\Tcp
File	\Device\Afd\Endpoint
File	\Device\NamedPipe\epmapper
File	\Device\NamedPipe\epmapper
File	\Device\KsecDD
File	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2982_x-ww_ac3l
File	\Device\NamedPipe\net\NtControlPipe4

The above snapshot depicts only network devices that are instantiated by the RPC svchost process. So mainly device is composed of Tcp , Ip , NamedPipe , etc. In general it is cleared the RPC svchost provides epmapper service on port 135. The flag is always provided in Listening state. The epmapper is configured with different end points. This service is considered as an instance of RPC service running with different end points stated as:

1. LPC port Epmapper [ncalrpc].
2. Epmapper Named Pipe [ncacn_np].
3. 135/TCP [ncacn_ip_tcp].
4. 135/UDP [ncadg_ip_udp].
5. 593/TCP [ncacn_http].

So question arises, why epmapper is configured with different end points? Actually when ever a client establishes a TCP connection on port 135, a RPC service instance is started with it. The client closes the connection and issue new connection with the port returned by the epmapper. The end point database is maintained by the **PORT MAPPER** service. The database consists of various RPC interfaces. The registration is done by calling RpcEpRegister function. So we are going to look at number of interfaces provided by epmapper.



The rpcss service not only runs the RPC subsystem but also the COM Service Control Manager (SCM), which hold the core of COM/DCOM infrastructure. The interfaces can be used as locally or remotely based on the request initiated by client. So this clears the epmapper end points dependency on RPC svchost. Let's see the disassembly layout:

After loading all exported and imported intermodular functions, I extract RpcEpRegister function. The function is exported with base module RPCRt4.dll. The stats that debugger provided are:

Address=**77E945FA** RPCRT4
Section=.text
Type=Export
Name=**RpcEpRegisterA**

The disassembly structured from entry point address of this specific function is:

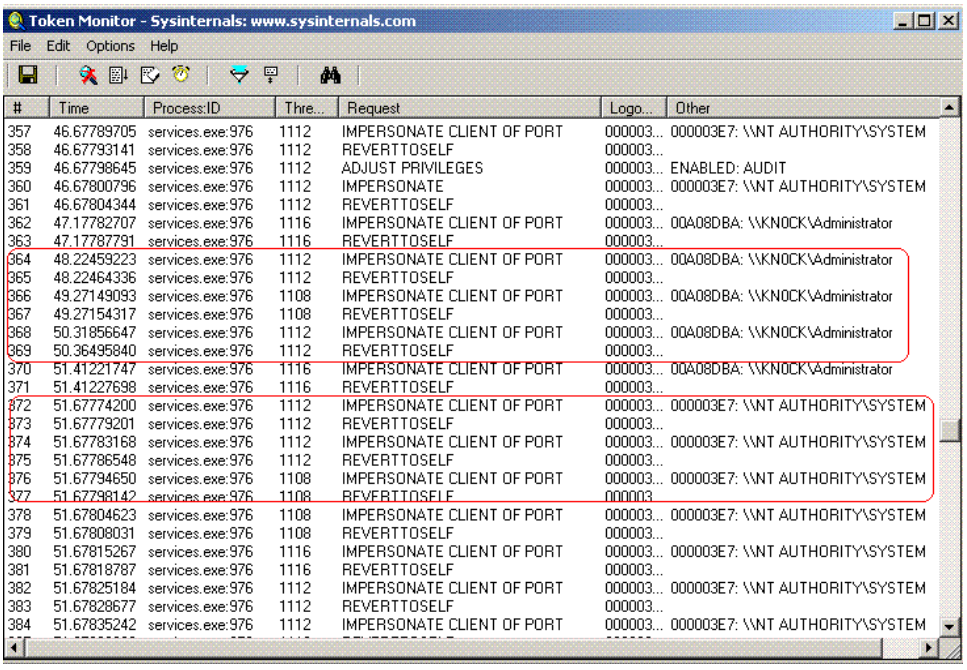
```
77E945FA > 8BFF MOV EDI,EDI
77E945FC 55 PUSH EBP
77E945FD 8BEC MOV EBP,ESP
77E945FF 83EC 28 SUB ESP,28
77E94602 A1 ACA2EF77 MOV EAX,DWORD PTR DS:[77EFA2AC]
77E94607 8B55 10 MOV EDX,DWORD PTR SS:[EBP+10]
77E9460A 53 PUSH EBX
77E9460B 8945 FC MOV DWORD PTR SS:[EBP-4],EAX
77E9460E 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
77E94611 56 PUSH ESI
77E94612 8B75 14 MOV ESI,DWORD PTR SS:[EBP+14]
77E94615 85F6 TEST ESI,ESI
77E94617 8945 E0 MOV DWORD PTR SS:[EBP-20],EAX
77E9461A 8B45 0C MOV EAX,DWORD PTR SS:[EBP+C]
77E9461D 57 PUSH EDI
77E9461E 8945 E4 MOV DWORD PTR SS:[EBP-1C],EAX
77E94621 8D5D D8 LEA EBX,DWORD PTR SS:[EBP-28]
77E94624 C645 EB 00 MOV BYTE PTR SS:[EBP-15],0
77E94628 75 03 JNZ SHORT RPCRT4.77E9462D
77E9462A 8D75 EB LEA ESI,DWORD PTR SS:[EBP-15]
77E9462D 8BC6 MOV EAX,ESI
77E9462F 8D78 01 LEA EDI,DWORD PTR DS:[EAX+1]
77E94632 8A08 MOV CL,BYTE PTR DS:[EAX]
77E94634 40 INC EAX
```

```
77E94635 84C9 TEST CL,CL
77E94637 ^75 F9 JNZ SHORT RPCRT4.77E94632
77E94639 2BC7 SUB EAX,EDI
77E9463B 83F8 40 CMP EAX,40
77E9463E 0F83 1A880100 JNB RPCRT4.77EACE5E
77E94644 837D E4 00 CMP DWORD PTR SS:[EBP-1C],0
77E94648 0F84 1A880100 JE RPCRT4.77EACE68
77E9464E 85D2 TEST EDX,EDX
77E94650 0F85 1C880100 JNZ RPCRT4.77EACE72
77E94656 33C0 XOR EAX,EAX
77E94658 8D7D EC LEA EDI,DWORD PTR SS:[EBP-14]
77E9465B AB STOS DWORD PTR ES:[EDI]
77E9465C AB STOS DWORD PTR ES:[EDI]
77E9465D AB STOS DWORD PTR ES:[EDI]
77E9465E AB STOS DWORD PTR ES:[EDI]
77E9465F 8D45 EC LEA EAX,DWORD PTR SS:[EBP-14]
77E94662 C745 D8 01000000 MOV DWORD PTR SS:[EBP-28],1
77E94669 8945 DC MOV DWORD PTR SS:[EBP-24],EAX
77E9466C E8 0F3FFEFF CALL RPCRT4.77E78580
```

The disassembly clearly depicts the calling of various modules in RPCRT4 dll. The calls are made directly.

[10] Tokens Anatomy in RPC svchost Process.

The security tokens provide a context of executing a process in specific account of system. The token should be initialized which enables the RPC svchost to run under main services.exe process. Why it is necessary to look into the security tokens of a system. The answer relies in understanding the execution context of that specific process. Let's see what the token monitor is displaying:



#	Time	Process:ID	Thre...	Request	Logo...	Other
357	46.67789705	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
358	46.67793141	services.exe:976	1112	REVERTTOSELF	000003...	
359	46.67798645	services.exe:976	1112	ADJUST PRIVILEGES	000003...	ENABLED: AUDIT
360	46.67800796	services.exe:976	1112	IMPERSONATE	000003...	000003E7: \\NT AUTHORITY\SYSTEM
361	46.67804344	services.exe:976	1112	REVERTTOSELF	000003...	
362	47.17782707	services.exe:976	1116	IMPERSONATE CLIENT OF PORT	000003...	00A08DBA: \\KNOCK\Administrator
363	47.17787791	services.exe:976	1116	REVERTTOSELF	000003...	
364	48.22459223	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	00A08DBA: \\KNOCK\Administrator
365	48.22464336	services.exe:976	1112	REVERTTOSELF	000003...	
366	49.27149093	services.exe:976	1108	IMPERSONATE CLIENT OF PORT	000003...	00A08DBA: \\KNOCK\Administrator
367	49.27154317	services.exe:976	1108	REVERTTOSELF	000003...	
368	50.31856647	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	00A08DBA: \\KNOCK\Administrator
369	50.36495840	services.exe:976	1112	REVERTTOSELF	000003...	
370	51.41221747	services.exe:976	1116	IMPERSONATE CLIENT OF PORT	000003...	00A08DBA: \\KNOCK\Administrator
371	51.41227698	services.exe:976	1116	REVERTTOSELF	000003...	
372	51.67774200	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
373	51.67779201	services.exe:976	1112	REVERTTOSELF	000003...	
374	51.67783168	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
375	51.67786548	services.exe:976	1112	REVERTTOSELF	000003...	
376	51.67794650	services.exe:976	1108	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
377	51.67798142	services.exe:976	1108	REVERTTOSELF	000003...	
378	51.67804623	services.exe:976	1108	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
379	51.67808031	services.exe:976	1108	REVERTTOSELF	000003...	
380	51.67815267	services.exe:976	1116	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
381	51.67818787	services.exe:976	1116	REVERTTOSELF	000003...	
382	51.67825184	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM
383	51.67828677	services.exe:976	1112	REVERTTOSELF	000003...	
384	51.67835242	services.exe:976	1112	IMPERSONATE CLIENT OF PORT	000003...	000003E7: \\NT AUTHORITY\SYSTEM

The Process ID of RPC svchost process is 976. This describes the security tokens that are used by process are KNOCK\Administrator and \\NT Authority System. The administrative context is used in process execution. As one knows the port 135 is always is in listening state to provided service to incoming client request. A client is impersonated to use system port in an administrative context.

The RPC process is in direct interface with different threads that are running inside it. The impersonation encompasses the ability of thread to run and execute using different security information. It means impersonation depends on the specific threads that are present in the server.

Here it is a client server mechanism .The server impersonates a client request. This is because it provides a virtual implementation in which the server thread is accessing resources on the behalf of the client. It actually sets a validation control over server objects by client request. In this way impersonation of server thread is done in the context of client. If you look a Revert to Self function, request is initiated suddenly after the impersonation. This is because no impersonation of thread occurred so it reverts back. No active connection is subjugated on port 135 for services. A continuous process is going on and on as Listening state is waiting for client connection. The impersonation and reverting of thread is undertaken by functions that are structured below:

Let see what the disassembly says:

All names, item 12663
Address=**77E7A80E** RPCRT4
Section=.text
Type=Export
Name=RpcImpersonateClient

The RpcImpersonateClient is exported as :

```
77E7A80E > 8BFF MOV EDI,EDI
77E7A810 55 PUSH EBP
77E7A811 8BEC MOV EBP,ESP
77E7A813 833D DCA0EF77 00 CMP DWORD PTR DS:[77EFA0DC],0
77E7A81A 0F84 EC5B0100 JE RPCRT4.77E9040C
77E7A820 E8 5BDDFFFF CALL RPCRT4.77E78580
```

The impersonation function is called in this manner to impersonating a client with server thread to access the contextual objects for functioning.

```
RPC_STATUS RPC_ENTRY RpcImpersonateClient(
        RPC_BINDING_HANDLE BindingHandle
        );
```

The binding handle defines how exactly server impersonates a client. After impersonation is done the reverting process occurs.

All names, item 12711
Address=**77E7892A** RPCRT4
Section=.text
Type=Export
Name=RpcRevertToSelf

The module is extracted from the intermodular call section.

```
77E7892A > 6A 00 PUSH 0
77E7892C E8 651F0000 CALL RPCRT4.RpcRevertToSelfEx
```

The function takes no parameters as the assembly code is showing. The RpcRevertToSelf function is called with no arguments. This function is called by RPC server to retain its own security identity.

```
RPC_STATUS RPC_ENTRY RpcRevertToSelf(void);
```

Finally, the server calls RpcImpersonateCleint to overwrite the security for the server thread with the client security context. After the task is completed, the server calls RpcRevertToSelf to restore the security context defined for the server thread.

This now completes our dissection of XP svchost internals.

Conclusion

The incore analysis will help us to unveil the hidden artifacts of system internals. The core of svchost is dissected to understand the working functionality of operating system entities and its usage. The concepts of thread management, Critical Objects, Registry Simulation, dispatching routines are reversed fully. The analysis is a reverse engineering layout of Generic Host Controller used by windows XP to queue up svchost processes. To understand the core we have to dismantle the internals related to system. The analysis is full structured example of this.