x86matthew - CreateSvcRpc - A custom RPC client to execute programs as the SYSTEM user

x86matthew.com/view_post

CreateSvcRpc - A custom RPC client to execute programs as the SYSTEM user

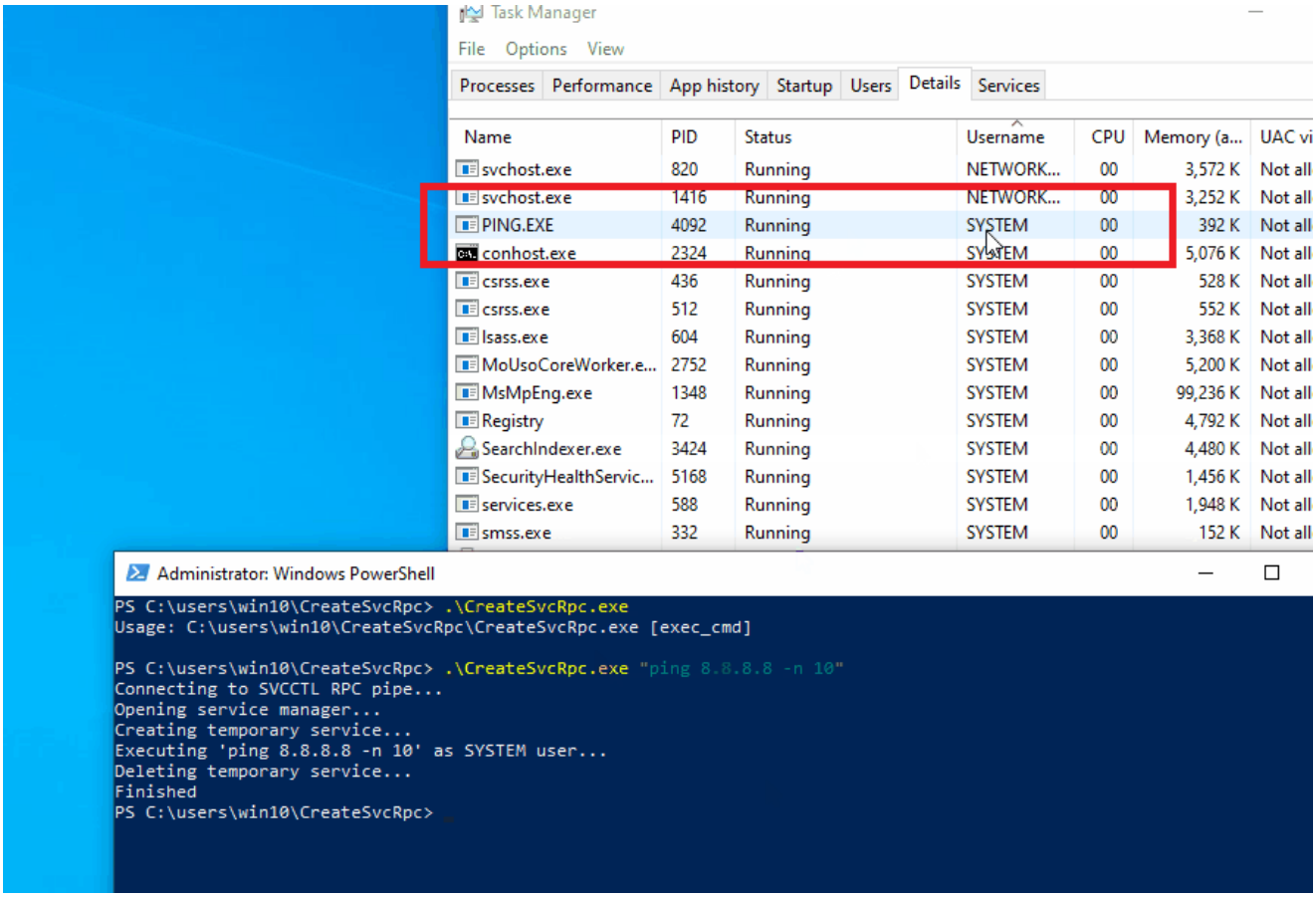
04/02/2022

The Windows RPC protocol is an area that I haven't previously experimented with very much. I have now created a custom RPC client which communicates with the ntsvcs pipe using raw data. This means it is possible to create and execute a Windows service using only the CreateFile and WriteFile APIs.

The RPC protocol seems to be somewhat documented, but the relevant information is so fragmented that I found it easier to reverse-engineer it from scratch.

I logged the communication of the Windows service APIs by hooking the NtWriteFile, NtReadFile, and NtFsControlFile functions. I analysed this data flow to gradually build my own RPC client.

After I got my first version working, I found some useful information in the Wireshark documentation which helped me label the remaining unknown fields in the RPC headers.



Full program code below:

#include <stdio.h> #include <windows.h> // rpc command ids #define RPC_CMD_ID_OPEN_SC_MANAGER 27 #define RPC_CMD_ID_CREATE_SERVICE 24 #define RPC_CMD_ID_START_SERVICE 31 #define RPC_CMD_ID_DELETE_SERVICE 2 // rpc command output lengths #define RPC_OUTPUT_LENGTH_OPEN_SC_MANAGER 24 #define RPC_OUTPUT_LENGTH_CREATE_SERVICE 28 #define RPC_OUTPUT_LENGTH_START_SERVICE 4 #define RPC_OUTPUT_LENGTH_DELETE_SERVICE 4 #define MAX_RPC_PACKET_LENGTH 4096 #define MAX_PROCEDURE_DATA_LENGTH 2048 #define CALC_ALIGN_PADDING(VALUE_LENGTH, ALIGN_BYTES) ((((VALUE_LENGTH + ALIGN_BYTES - 1) / ALIGN_BYTES) * ALIGN_BYTES) - VALUE_LENGTH) struct RpcBaseHeaderStruct { WORD wVersion; BYTE bPacketType; BYTE bPacketFlags; DWORD dwDataRepresentation; WORD wFragLength; WORD wAuthLength; DWORD dwCallIndex; }; struct RpcRequestHeaderStruct { DWORD dwAllocHint; WORD wContextID; WORD wProcedureNumber; }; struct RpcResponseHeaderStruct { DWORD dwAllocHint; WORD wContextID; BYTE bCancelCount; BYTE bAlign[1]; }; struct RpcBindRequestContextEntryStruct { WORD wContextID; WORD wTransItemCount; BYTE bInterfaceUUID[16]; DWORD dwInterfaceVersion; BYTE bTransferSyntaxUUID[16]; DWORD dwTransferSyntaxVersion; }; struct RpcBindRequestHeaderStruct { WORD wMaxSendFrag; WORD wMaxRecvFrag; DWORD dwAssocGroup; BYTE bContextCount; BYTE bAlign[3];

```
RpcBindRequestContextEntryStruct Context; }; struct RpcBindResponseContextEntryStruct { WORD wResult; WORD wAlign; BYTE bTransferSyntax[16]; DWORD dwTransferSyntaxVersion; }; struct RpcBindResponseHeader1Struct { WORD wMaxSendFrag; WORD wMaxRecvFrag; DWORD dwAssocGroup; }; struct RpcBindResponseHeader2Struct { DWORD dwContextResultCount; RpcBindResponseContextEntryStruct Context; }; struct RpcConnectionStruct { HANDLE hFile; DWORD dwCallIndex; DWORD dwInputError; DWORD dwRequestInitialised; BYTE bProcedureInputData[MAX_PROCEDURE_DATA_LENGTH]; DWORD dwProcedureInputDataLength; BYTE bProcedureOutputData[MAX_PROCEDURE_DATA_LENGTH]; DWORD dwProcedureOutputDataLength; }; DWORD RpcConvertUUID(char *pString, BYTE *pUUID, DWORD dwMaxLength) { BYTE bUUID[16]; BYTE bFixedUUID[16]; DWORD dwUUIDLength = 0; BYTE bCurrInputChar = 0; BYTE bConvertedByte = 0; DWORD dwProcessedByteCount = 0; BYTE bCurrOutputByte = 0; // ensure output buffer is large enough if(dwMaxLength < 16) { return 1; } // check uuid length dwUUIDLength = strlen("00000000-0000-0000-0000-000000000000"); if(strlen(pString) != dwUUIDLength) { return 1; } // convert string to uuid for(DWORD i = 0; i < dwUUIDLength; i++) { // get current input character bCurrInputChar = *(BYTE*)((BYTE*)pString + i); // check if a dash character is expected here if(i == 8 || i == 13 || i == 18 || i == 23) { if(bCurrInputChar == '-') { continue; } else { return 1; } } else { // check current input character value if(bCurrInputChar >= 'a' && bCurrInputChar <= 'f') { bConvertedByte = 0xA + (bCurrInputChar - 'a'); } else if(bCurrInputChar >= 'A' && bCurrInputChar <= 'F') { bConvertedByte = 0xA + (bCurrInputChar - 'A'); } else if(bCurrInputChar >= '0' && bCurrInputChar <= '9') { bConvertedByte = 0 + (bCurrInputChar - '0'); } else { // invalid character return 1; } if((dwProcessedByteCount % 2) == 0) { bCurrOutputByte = bConvertedByte * 0x10; } else { bCurrOutputByte += bConvertedByte; // store current uuid byte bUUID[(dwProcessedByteCount - 1) / 2] = bCurrOutputByte; } dwProcessedByteCount++; } } // fix uuid endianness memcpy((void*)bFixedUUID, (void*)bUUID, sizeof(bUUID)); bFixedUUID[0] = bUUID[3]; bFixedUUID[1] = bUUID[2]; bFixedUUID[2] = bUUID[1]; bFixedUUID[3] = bUUID[0]; bFixedUUID[4] = bUUID[5]; bFixedUUID[5] = bUUID[4]; bFixedUUID[6] = bUUID[7]; bFixedUUID[7] = bUUID[6]; // store uuid memcpy((void*)pUUID, (void*)bFixedUUID, sizeof(bUUID)); return 0; } DWORD RpcBind(RpcConnectionStruct *pRpcConnection, char *pInterfaceUUID, DWORD dwInterfaceVersion) { RpcBaseHeaderStruct RpcBaseHeader; RpcBindRequestHeaderStruct RpcBindRequestHeader; DWORD dwBytesWritten = 0; DWORD dwBytesRead = 0; BYTE bResponseData[MAX_RPC_PACKET_LENGTH]; RpcBaseHeaderStruct *pRpcResponseBaseHeader = NULL; RpcBindResponseHeader1Struct *pRpcBindResponseHeader1 = NULL; RpcBindResponseHeader2Struct *pRpcBindResponseHeader2 = NULL; BYTE *pSecondaryAddrHeaderBlock = NULL; WORD wSecondaryAddrLen = 0; DWORD dwSecondaryAddrAlign = 0; // set base header details memset((void*)&RpcBaseHeader, 0, sizeof(RpcBaseHeader)); RpcBaseHeader.wVersion = 5; RpcBaseHeader.bPacketType = 11;
```

```
RpcBaseHeader.bPacketFlags = 3; RpcBaseHeader.dwDataRepresentation = 0x10;
RpcBaseHeader.wFragLength = sizeof(RpcBaseHeader) + sizeof(RpcBindRequestHeader);
RpcBaseHeader.wAuthLength = 0; RpcBaseHeader.dwCallIndex = pRpcConnection-
>dwCallIndex; // set bind request header details memset((void*)&RpcBindRequestHeader, 0,
sizeof(RpcBindRequestHeader)); RpcBindRequestHeader.wMaxSendFrag =
MAX_RPC_PACKET_LENGTH; RpcBindRequestHeader.wMaxRecvFrag =
MAX_RPC_PACKET_LENGTH; RpcBindRequestHeader.dwAssocGroup = 0;
RpcBindRequestHeader.bContextCount = 1; RpcBindRequestHeader.Context.wContextID =
0; RpcBindRequestHeader.Context.wTransItemCount = 1;
RpcBindRequestHeader.Context.dwTransferSyntaxVersion = 2; // get interface UUID
if(RpcConvertUUID(pInterfaceUUID, RpcBindRequestHeader.Context.bInterfaceUUID,
sizeof(RpcBindRequestHeader.Context.bInterfaceUUID)) != 0) { return 1; }
RpcBindRequestHeader.Context.dwInterfaceVersion = dwInterfaceVersion; // {8a885d04-
1ceb-11c9-9fe8-08002b104860} (NDR) if(RpcConvertUUID("8a885d04-1ceb-11c9-9fe8-
08002b104860", RpcBindRequestHeader.Context.bTransferSyntaxUUID,
sizeof(RpcBindRequestHeader.Context.bTransferSyntaxUUID)) != 0) { return 1; } // write
base header if(WriteFile(pRpcConnection->hFile, (void*)&RpcBaseHeader,
sizeof(RpcBaseHeader), &dwBytesWritten, NULL) == 0) { return 1; } // write bind request
header if(WriteFile(pRpcConnection->hFile, (void*)&RpcBindRequestHeader,
sizeof(RpcBindRequestHeader), &dwBytesWritten, NULL) == 0) { return 1; } // increase call
index pRpcConnection->dwCallIndex++; // get bind response
memset((void*)&bResponseData, 0, sizeof(bResponseData)); if(ReadFile(pRpcConnection-
>hFile, (void*)bResponseData, sizeof(bResponseData), &dwBytesRead, NULL) == 0) {
return 1; } // get a ptr to the base response header pRpcResponseBaseHeader =
(RpcBaseHeaderStruct*)bResponseData; // validate base response header
if(pRpcResponseBaseHeader->wVersion != 5) { return 1; } if(pRpcResponseBaseHeader-
>bPacketType != 12) { return 1; } if(pRpcResponseBaseHeader->bPacketFlags != 3) { return
1; } if(pRpcResponseBaseHeader->wFragLength != dwBytesRead) { return 1; } // get a ptr to
the main bind response header body pRpcBindResponseHeader1 =
(RpcBindResponseHeader1Struct*)((BYTE*)pRpcResponseBaseHeader +
sizeof(RpcBaseHeaderStruct)); // get secondary addr header ptr
pSecondaryAddrHeaderBlock = (BYTE*)pRpcBindResponseHeader1 +
sizeof(RpcBindResponseHeader1Struct); wSecondaryAddrLen = *
(WORD*)pSecondaryAddrHeaderBlock; // validate secondary addr length
if(wSecondaryAddrLen > 256) { return 1; } // calculate padding for secondary addr value if
necessary dwSecondaryAddrAlign = CALC_ALIGN_PADDING((sizeof(WORD) +
wSecondaryAddrLen), 4); // get a ptr to the main bind response header body (after the
variable-length secondary addr field) pRpcBindResponseHeader2 =
(RpcBindResponseHeader2Struct*)((BYTE*)pSecondaryAddrHeaderBlock + sizeof(WORD)
+ wSecondaryAddrLen + dwSecondaryAddrAlign); // validate context count
if(pRpcBindResponseHeader2->dwContextResultCount != 1) { return 1; } // ensure the result
value for context #1 was successful if(pRpcBindResponseHeader2->Context.wResult != 0) {
```

```
return 1; } return 0; } DWORD RpcConnect(char *pPipeName, char *pInterfaceUUID,
DWORD dwInterfaceVersion, RpcConnectionStruct *pRpcConnection) { HANDLE hFile =
NULL; char szPipePath[512]; RpcConnectionStruct RpcConnection; // set pipe path
memset(szPipePath, 0, sizeof(szPipePath)); _snprintf(szPipePath, sizeof(szPipePath) - 1,
"\\\\.\\pipe\\%s", pPipeName); // open rpc pipe hFile = CreateFile(szPipePath,
GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL); if(hFile ==
INVALID_HANDLE_VALUE) { return 1; } // initialise rpc connection data
memset((void*)&RpcConnection, 0, sizeof(RpcConnection)); RpcConnection.hFile = hFile;
RpcConnection.dwCallIndex = 1; // bind rpc connection if(RpcBind(&RpcConnection,
pInterfaceUUID, dwInterfaceVersion) != 0) { return 1; } // store connection data
memcpy((void*)pRpcConnection, (void*)&RpcConnection, sizeof(RpcConnection)); return 0;
} DWORD RpcSendRequest(RpcConnectionStruct *pRpcConnection, DWORD
dwProcedureNumber) { RpcBaseHeaderStruct RpcBaseHeader; RpcRequestHeaderStruct
RpcRequestHeader; DWORD dwBytesWritten = 0; BYTE
bResponseData[MAX_RPC_PACKET_LENGTH]; RpcBaseHeaderStruct
*pRpcResponseBaseHeader = NULL; RpcResponseHeaderStruct *pRpcResponseHeader =
NULL; DWORD dwProcedureResponseDataLength = 0; DWORD dwBytesRead = 0; BYTE
*pTempProcedureResponseDataPtr = NULL; // ensure rpc request has been initialised
if(pRpcConnection->dwRequestInitialised == 0) { return 1; } // clear initialised flag
pRpcConnection->dwRequestInitialised = 0; // check for input errors if(pRpcConnection-
>dwInputError != 0) { return 1; } // set base header details memset((void*)&RpcBaseHeader,
0, sizeof(RpcBaseHeader)); RpcBaseHeader.wVersion = 5; RpcBaseHeader.bPacketType =
0; RpcBaseHeader.bPacketFlags = 3; RpcBaseHeader.dwDataRepresentation = 0x10;
RpcBaseHeader.wFragLength = sizeof(RpcBaseHeader) + sizeof(RpcRequestHeader) +
pRpcConnection->dwProcedureInputDataLength; RpcBaseHeader.wAuthLength = 0;
RpcBaseHeader.dwCallIndex = pRpcConnection->dwCallIndex; // set request header details
memset((void*)&RpcRequestHeader, 0, sizeof(RpcRequestHeader));
RpcRequestHeader.dwAllocHint = 0; RpcRequestHeader.wContextID = 0;
RpcRequestHeader.wProcedureNumber = (WORD)dwProcedureNumber; // write base
header if(WriteFile(pRpcConnection->hFile, (void*)&RpcBaseHeader,
sizeof(RpcBaseHeader), &dwBytesWritten, NULL) == 0) { return 1; } // write request header
if(WriteFile(pRpcConnection->hFile, (void*)&RpcRequestHeader,
sizeof(RpcRequestHeader), &dwBytesWritten, NULL) == 0) { return 1; } // write request body
if(WriteFile(pRpcConnection->hFile, (void*)pRpcConnection->bProcedureInputData,
pRpcConnection->dwProcedureInputDataLength, &dwBytesWritten, NULL) == 0) { return 1; }
// increase call index pRpcConnection->dwCallIndex++; // get bind response
memset((void*)&bResponseData, 0, sizeof(bResponseData)); if(ReadFile(pRpcConnection-
>hFile, (void*)bResponseData, sizeof(bResponseData), &dwBytesRead, NULL) == 0) {
return 1; } // get a ptr to the base response header pRpcResponseBaseHeader =
(RpcBaseHeaderStruct*)bResponseData; // validate base response header
if(pRpcResponseBaseHeader->wVersion != 5) { return 1; } if(pRpcResponseBaseHeader-
>bPacketType != 2) { return 1; } if(pRpcResponseBaseHeader->bPacketFlags != 3) { return
```

```
1; } if(pRpcResponseBaseHeader->wFragLength != dwBytesRead) { return 1; } // get a ptr to the main response header body pRpcResponseHeader = (RpcResponseHeaderStruct*) ((BYTE*)pRpcResponseBaseHeader + sizeof(RpcBaseHeaderStruct)); // context ID must be 0 if(pRpcResponseHeader->wContextID != 0) { return 1; } // calculate command response data length dwProcedureResponseDataLength = pRpcResponseBaseHeader->wFragLength - sizeof(RpcBaseHeaderStruct) - sizeof(RpcResponseHeaderStruct); // store response data if(dwProcedureResponseDataLength > sizeof(pRpcConnection->bProcedureOutputData)) { return 1; } pTempProcedureResponseDataPtr = (BYTE*)pRpcResponseHeader + sizeof(RpcResponseHeaderStruct); memcpy(pRpcConnection->bProcedureOutputData, pTempProcedureResponseDataPtr, dwProcedureResponseDataLength); // store response data length pRpcConnection->dwProcedureOutputDataLength = dwProcedureResponseDataLength; return 0; } DWORD RpcInitialiseRequestData(RpcConnectionStruct *pRpcConnection) { // initialise request data memset(pRpcConnection->bProcedureInputData, 0, sizeof(pRpcConnection->bProcedureInputData)); pRpcConnection->dwProcedureInputDataLength = 0; memset(pRpcConnection->bProcedureOutputData, 0, sizeof(pRpcConnection->bProcedureOutputData)); pRpcConnection->dwProcedureOutputDataLength = 0; // reset input error flag pRpcConnection->dwInputError = 0; // set initialised flag pRpcConnection->dwRequestInitialised = 1; return 0; } DWORD RpcAppendRequestData_Binary(RpcConnectionStruct *pRpcConnection, BYTE *pData, DWORD dwDataLength) { DWORD dwBytesAvailable = 0; // ensure the request has been initialised if(pRpcConnection->dwRequestInitialised == 0) { return 1; } // calculate number of bytes remaining in the input buffer dwBytesAvailable = sizeof(pRpcConnection->bProcedureInputData) - pRpcConnection->dwProcedureInputDataLength; if(dwDataLength > dwBytesAvailable) { // set input error flag pRpcConnection->dwInputError = 1; return 1; } // store data in buffer memcpy((void*)&pRpcConnection->bProcedureInputData[pRpcConnection->dwProcedureInputDataLength], pData, dwDataLength); pRpcConnection->dwProcedureInputDataLength += dwDataLength; // align to 4 bytes if necessary pRpcConnection->dwProcedureInputDataLength += CALC_ALIGN_PADDING(dwDataLength, 4); return 0; } DWORD RpcAppendRequestData_Dword(RpcConnectionStruct *pRpcConnection, DWORD dwValue) { // add dword value if(RpcAppendRequestData_Binary(pRpcConnection, (BYTE*)&dwValue, sizeof(DWORD)) != 0) { return 1; } return 0; } DWORD RpcDisconnect(RpcConnectionStruct *pRpcConnection) { // close pipe handle CloseHandle(pRpcConnection->hFile); return 0; } int main(int argc, char *argv[]) { RpcConnectionStruct RpcConnection; BYTE bServiceManagerObject[20]; BYTE bServiceObject[20]; DWORD dwReturnValue = 0; char szServiceName[256]; DWORD dwServiceNameLength = 0; char szServiceCommandLine[256]; DWORD dwServiceCommandLineLength = 0; char *pExecCmd = NULL; printf("CreateSvcRpc - www.x86matthew.com\n\n"); if(argc != 2) { printf("Usage: %s [exec_cmd]\n\n", argv[0]); return 1; } // get cmd param pExecCmd = argv[1]; // generate a temporary service name memset(szServiceName, 0, sizeof(szServiceName)); _snprintf(szServiceName, sizeof(szServiceName) - 1,
```

```
"CreateSvcRpc_%u", GetTickCount()); dwServiceNameLength = strlen(szServiceName) + 1;
// set service command line memset(szServiceCommandLine, 0,
sizeof(szServiceCommandLine)); _snprintf(szServiceCommandLine,
sizeof(szServiceCommandLine) - 1, "cmd /c start %s", pExecCmd);
dwServiceCommandLineLength = strlen(szServiceCommandLine) + 1; printf("Connecting to
SVCCTL RPC pipe...\n"); // open SVCCTL v2.0 if(RpcConnect("ntsvcs", "367abb81-9844-
35f1-ad32-98f038001003", 2, &RpcConnection) != 0) { printf("Failed to connect to RPC
pipe\n"); return 1; } printf("Opening service manager...\n"); // OpenSCManager
RpcInitialiseRequestData(&RpcConnection);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, SC_MANAGER_ALL_ACCESS);
if(RpcSendRequest(&RpcConnection, RPC_CMD_ID_OPEN_SC_MANAGER) != 0) { // error
RpcDisconnect(&RpcConnection); return 1; } // validate rpc output data length
if(RpcConnection.dwProcedureOutputDataLength !=
RPC_OUTPUT_LENGTH_OPEN_SC_MANAGER) { // error
RpcDisconnect(&RpcConnection); return 1; } // get return value dwReturnValue = *
(DWORD*)&RpcConnection.bProcedureOutputData[20]; // check return value
if(dwReturnValue != 0) { printf("OpenSCManager error: %u\n", dwReturnValue); // error
RpcDisconnect(&RpcConnection); return 1; } // store service manager object
memcpy(bServiceManagerObject, (void*)&RpcConnection.bProcedureOutputData[0],
sizeof(bServiceManagerObject)); printf("Creating temporary service...\n"); // CreateService
RpcInitialiseRequestData(&RpcConnection);
RpcAppendRequestData_Binary(&RpcConnection, bServiceManagerObject,
sizeof(bServiceManagerObject)); RpcAppendRequestData_Dword(&RpcConnection,
dwServiceNameLength); RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, dwServiceNameLength);
RpcAppendRequestData_Binary(&RpcConnection, (BYTE*)szServiceName,
dwServiceNameLength); RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, SERVICE_ALL_ACCESS);
RpcAppendRequestData_Dword(&RpcConnection, SERVICE_WIN32_OWN_PROCESS);
RpcAppendRequestData_Dword(&RpcConnection, SERVICE_DEMAND_START);
RpcAppendRequestData_Dword(&RpcConnection, SERVICE_ERROR_IGNORE);
RpcAppendRequestData_Dword(&RpcConnection, dwServiceCommandLineLength);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, dwServiceCommandLineLength);
RpcAppendRequestData_Binary(&RpcConnection, (BYTE*)szServiceCommandLine,
dwServiceCommandLineLength); RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0);
```

```
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0); if(RpcSendRequest(&RpcConnection,
RPC_CMD_ID_CREATE_SERVICE) != 0) { // error RpcDisconnect(&RpcConnection); return
1; } // validate rpc output data length if(RpcConnection.dwProcedureOutputDataLength !=
RPC_OUTPUT_LENGTH_CREATE_SERVICE) { // error RpcDisconnect(&RpcConnection);
return 1; } // get return value dwReturnValue = *
(DWORD*)&RpcConnection.bProcedureOutputData[24]; // check return value
if(dwReturnValue != 0) { printf("CreateService error: %u\n", dwReturnValue); // error
RpcDisconnect(&RpcConnection); return 1; } // store service object memcpy(bServiceObject,
(void*)&RpcConnection.bProcedureOutputData[4], sizeof(bServiceObject)); printf("Executing
'%s' as SYSTEM user...\n", pExecCmd); // StartService
RpcInitialiseRequestData(&RpcConnection);
RpcAppendRequestData_Binary(&RpcConnection, bServiceObject, sizeof(bServiceObject));
RpcAppendRequestData_Dword(&RpcConnection, 0);
RpcAppendRequestData_Dword(&RpcConnection, 0); if(RpcSendRequest(&RpcConnection,
RPC_CMD_ID_START_SERVICE) != 0) { // error RpcDisconnect(&RpcConnection); return 1;
} // validate rpc output data length if(RpcConnection.dwProcedureOutputDataLength !=
RPC_OUTPUT_LENGTH_START_SERVICE) { // error RpcDisconnect(&RpcConnection);
return 1; } // get return value dwReturnValue = *
(DWORD*)&RpcConnection.bProcedureOutputData[0]; // check return value
if(dwReturnValue != 0 && dwReturnValue != ERROR_SERVICE_REQUEST_TIMEOUT) {
printf("StartService error: %u\n", dwReturnValue); // error RpcDisconnect(&RpcConnection);
return 1; } printf("Deleting temporary service...\n"); // DeleteService
RpcInitialiseRequestData(&RpcConnection);
RpcAppendRequestData_Binary(&RpcConnection, bServiceObject, sizeof(bServiceObject));
if(RpcSendRequest(&RpcConnection, RPC_CMD_ID_DELETE_SERVICE) != 0) { // error
RpcDisconnect(&RpcConnection); return 1; } // validate rpc output data length
if(RpcConnection.dwProcedureOutputDataLength !=
RPC_OUTPUT_LENGTH_DELETE_SERVICE) { // error RpcDisconnect(&RpcConnection);
return 1; } // get return value dwReturnValue = *
(DWORD*)&RpcConnection.bProcedureOutputData[0]; // check return value
if(dwReturnValue != 0) { printf("DeleteService error: %u\n", dwReturnValue); // error
RpcDisconnect(&RpcConnection); return 1; } printf("Finished\n"); // disconnect from rpc pipe
if(RpcDisconnect(&RpcConnection) != 0) { return 1; } return 0; }
```