# Interacting with IDA through IPC channels

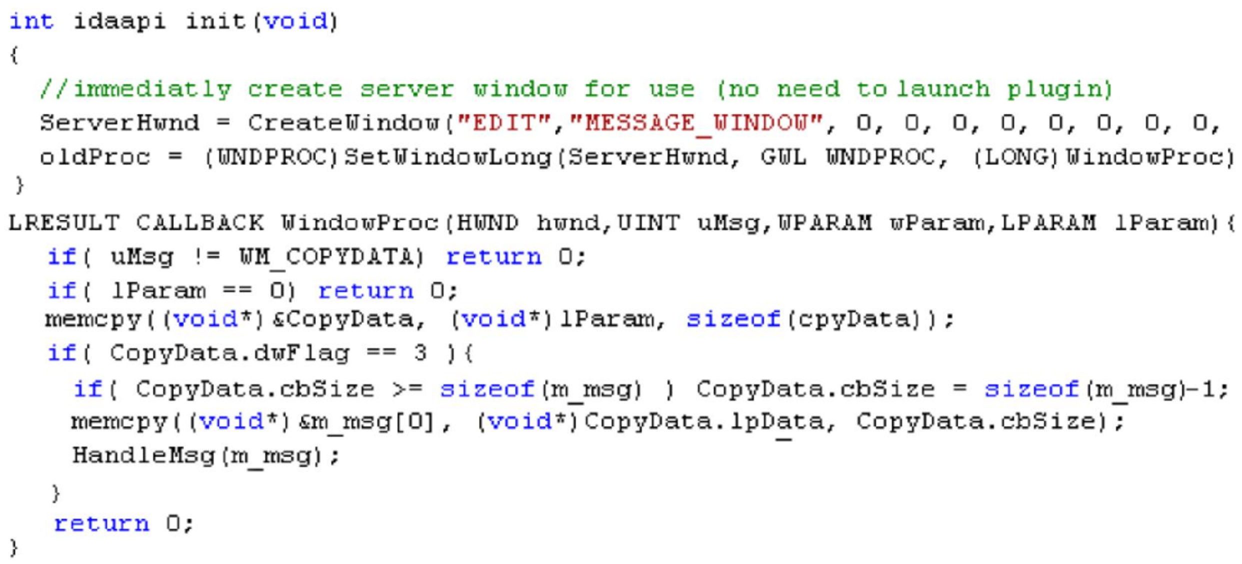
David Zimmer <dzzie@yahoo.com>

In this article we are going to discuss a mechanism that can be used to interact with IDA through external applications.

The reason this technique was developed was to provide a convenient way for utility applications to query information from IDA databases, and automate its interface.

Over the years, several methods have been tried such as pipes, and sockets. In the end, the easiest Inter-Process Communication (IPC) technique I have found is the Windows specific SendMessage API along with the WM_COPYDATA message. This technique was chosen for its simplicity, reliability, and its inherently synchronous nature.

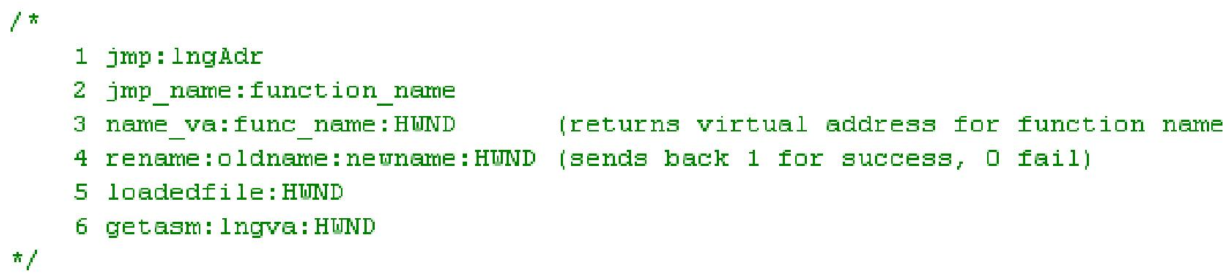
With this technique, the IDA server plugin creates a window and watches for command messages to come in.



```
int idaapi init(void)
{
  //immediatly create server window for use (no need to launch plugin)
  ServerHwnd = CreateWindow("EDIT","MESSAGE_WINDOW", 0, 0, 0, 0, 0, 0, 0, 0,
  oldProc = (WNDPROC)SetWindowLong(ServerHwnd, GWL WNDPROC, (LONG)WindowProc)
 }
LRESULT CALLBACK WindowProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam){
   if( uMsg != WM_COPYDATA) return 0;
   if( lParam == 0) return 0;
   memcpy((void*)&CopyData, (void*)lParam, sizeof(cpyData));
   if( CopyData.dwFlag == 3 ){
     if( CopyData.cbSize >= sizeof(m_msg) ) CopyData.cbSize = sizeof(m_msg)-1;
     memcpy((void*)&m_msg[0], (void*)CopyData.lpData, CopyData.cbSize);
     HandleMsg(m_msg);
   }
   return 0;
}
```

(abbreviated C source to CreateWindow and receive messages)

The plugin contains its own text based API which can be used to automate actions and return queried data back to the calling process. Example commands and data transfer routines are shown below:



```
 /*
      1 jmp:lngAdr
      2 jmp_name:function_name
      3 name_va:func_name:HWND       (returns virtual address for function name
      4 rename:oldname:newname:HWND (sends back 1 for success, 0 fail)
      5 loadedfile:HWND
      6 getasm:lngva:HWND
 */
```

(hwnd arguments specify which window handle receives data callback)

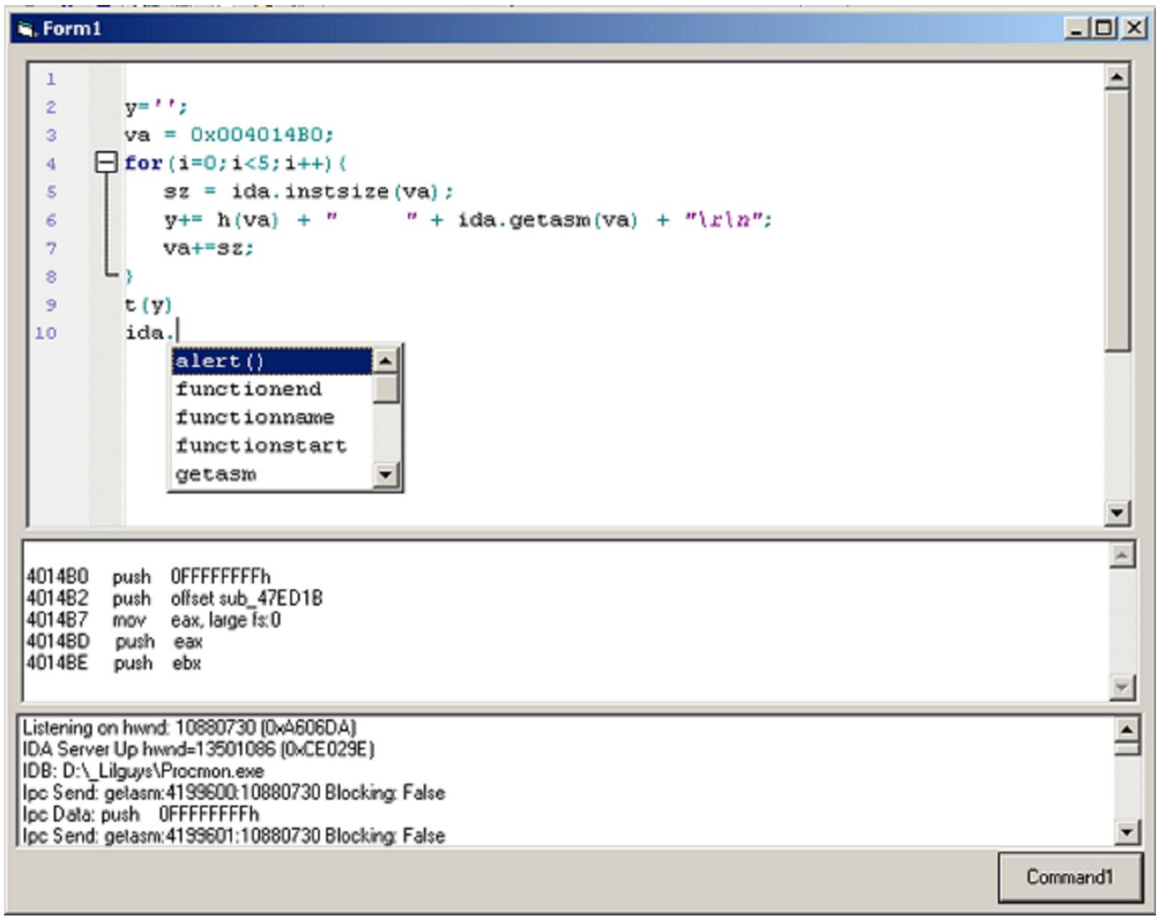
```
bool SendIntMessage(int hwnd, int resp){
    char tmp[30]={0};
    sprintf(tmp, "%d", resp);
    return SendTextMessage(hwnd,tmp, strlen(tmp));
}

bool SendTextMessage(int hwnd, char *Buffer, int blen)
{
    cpyData cpStructData;
    cpStructData.cbSize = blen ;
    cpStructData.lpData = (int)Buffer;
    cpStructData.dwFlag = 3;
    SendMessage((HWND)hwnd, WM_COPYDATA, (WPARAM)hwnd,(LPARAM)&cpStructData);
    return true;
}
```

There are several advantages to this technique. Traditional plugin development usually requires compiling your code and launching the plugin from the host application. Debugging often entails wading through runtime debug logs, inferring problems through observed behavior, and chasing crashs in a debugger.

This technique allows you to debug your executable in the development IDE as normal, harnessing all of its powerful capabilities such as edit and continue, call stack viewing, variable watches, breakpoints etc.

Complex projects can be run directly while you iron out interface behaviors and are not dependant upon the host. Datasets can even be loaded from test files so that many routines can be debugged independently of IDA.

One project where this technique was put to good use was an experiment to see if a Javascript IDE could be created for IDA automation tasks. The desire was for a scriptable interface that supported intellisense, function prototype tool tips, and syntax highlighting.



```

y='';
va = 0x004014B0;
for(i=0;i<5;i++){
    sz = ida.instsize(va);
    y+= h(va) + "      " + ida.getasm(va) + "\r\n";
    va+=sz;
}
t(y)
ida.
```

```
4014B0   push   0FFFFFFFFh
4014B2   push   offset sub_47ED1B
4014B7   mov    eax, large fs:0
4014BD   push   eax
4014BE   push   ebx
```

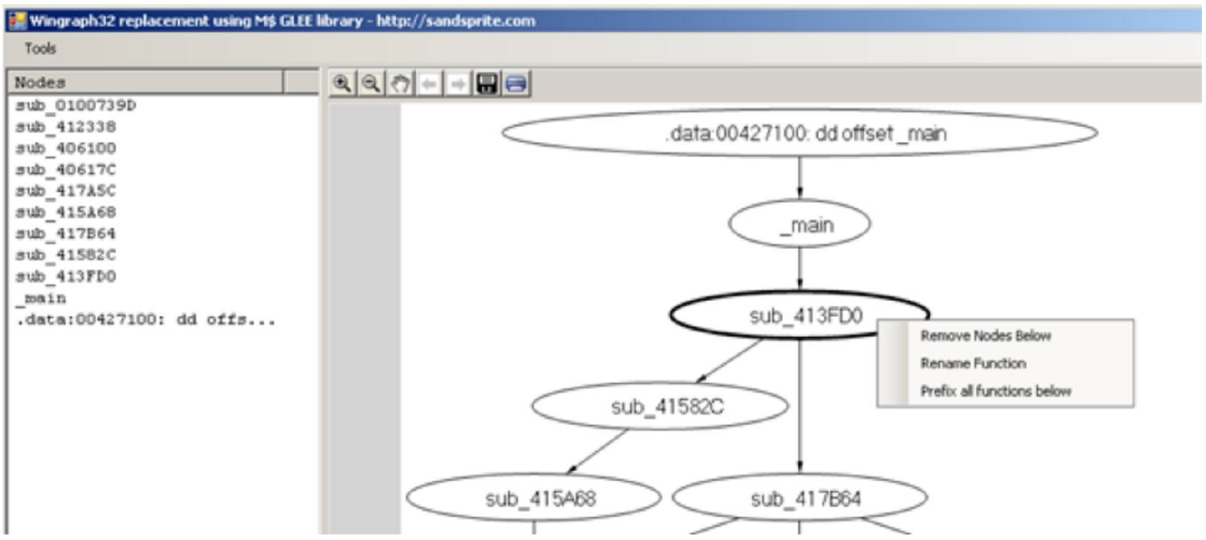
```
Listening on hwnd: 10880730 (0xA606DA)
IDA Server Up hwnd=13501086 (0xCE029E)
IDB: D:\_Lilguys\Procmon.exe
Ipc Send: getasm:4199600:10880730 Blocking: False
Ipc Data: push    0FFFFFFFFh
Ipc Send: getasm:4199601:10880730 Blocking: False
```

With a project such as this, coding for the IDE behaviors is a major part of the development task. To develop such an application strictly as a plugin would be a daunting endeavor. Developing it as a standalone application was a great advantage where it could be run directly from the Visual Studio IDE without any intermediate steps.

One other advantage to this technique is that it opens up plugin development to include higher level languages that do not have native support for the IDA API*. Languages such as C#, Delphi, and VB6 can easily interact with IDA through this mechanism. These languages have excellent rapid GUI development capabilities and a wealth of complex components already available to them. This technique is even open to Java developers through the JNI.

Once the intermediate API and the client access library is written, creating utilities to integrate with IDA becomes a pretty quick process. Another example project that has come in handy is a Wingraph32 replacement that was coded in about a day.

The interface shown below automatically syncs the IDA disassembly when graph nodes are clicked and can perform several renaming operations.



There are some limitations to this technique as well. The particular implementation detailed in this article is Windows specific. It also requires both applications to be running on the same machine.

Another consideration is that the data is crossing process boundaries which can be a performance hit depending on what is being transferred and how often it is being invoked. For example, extracting or patching large blocks of data would best be optimized by reserving the use of the SendMessage API as the command channel, and utilizing files or shared memory for the data transfer. This will likely be an optimization included in future revisions.

The example IDASRVR project linked to below currently supports 36 API and uses a simple text based command protocol. Sample code is provided in C and C#. Client libraries are also available for C# and VB6 in the associated projects below.

Sample Projects:

IDASrvr – IDA IPC Server example

IDA_Jscript – Javascript IDE PoC for IDA (VB6)

GleeGraph – C# Wingraph32 replacement

* You can create C# and VB6 in process plugins for IDA through COM. This was my first approach, and was used in my IDACompare plugin.