DLL Injection – A Simple Message Box

by R4ndom on Aug.13, 2012, under Intermediate, Tutorials

Requirements

In this tutorial, we will go over adding a message box using DLL injection. This is meant to be a gentle introduction to the subject and not a detailed analysis (which will be in later tutorials).

For this tutorial, you will need OllyDBG v.1.10, IIDKing, and MASM. IIDKing is included in the download of this tutorial, available on the tutorials page.

You may also want to use an IDE for the assembly- I am using RadASM which you can get here. You can also download MASM on the RadASM site. You may also want my version of OllyDBG if you want the tutorial to match your version. You can get that on the tools page.

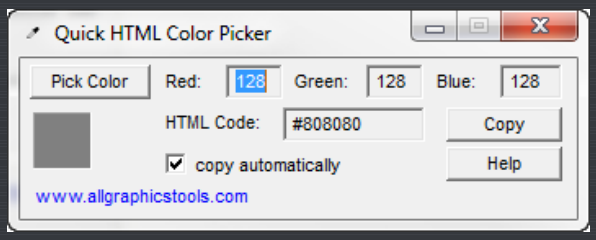
Lastly, because of the nature of DLL injection, your virus scanner may give a false positive on the downloaded files, or files you create during this tutorial. Because many viruses use the technique in this tutorial, the false positive is to be expected. If you downloaded the tutorials directly from my site, you have nothing to worry about as all files have been scanned (many times). If, however, you do not download from my site, you may run the suspicious file through www.VirsuTotal.com. The benefit of this site is that it runs the file through the top 42 virus scanners, instead of just one. It is a good way of detecting false positives. Generally, if VirusTotal shows less than 3-4 positives, I consider it safe. Any more than that and I run it in a virtual machine, as the file may be infected.

Introduction

If you have read my other tutorials on modifying binaries, you know that you can add functionality to an executable by modifying the code directly, using code caves. This method is perfectly legitimate, if not a little time consuming. There is an alternative to this method, however, in that your added functionality code can be put into a separate file, a DLL file, and this code can be called when the target app is loaded. This saves the trouble of finding a code cave, worrying about offsets etc. Though DLL injection also has it's limitations (especially when using resources).

In this tutorial we will be adding a Message Box to a freeware program called "ColorPicker". It is a very simple freeware program that simply allows you to select a pixel anywhere on the screen and it will show the color values of that pixel. Using a simple program like this helps us not get bogged down in other details.

Running the app, we see the main screen:

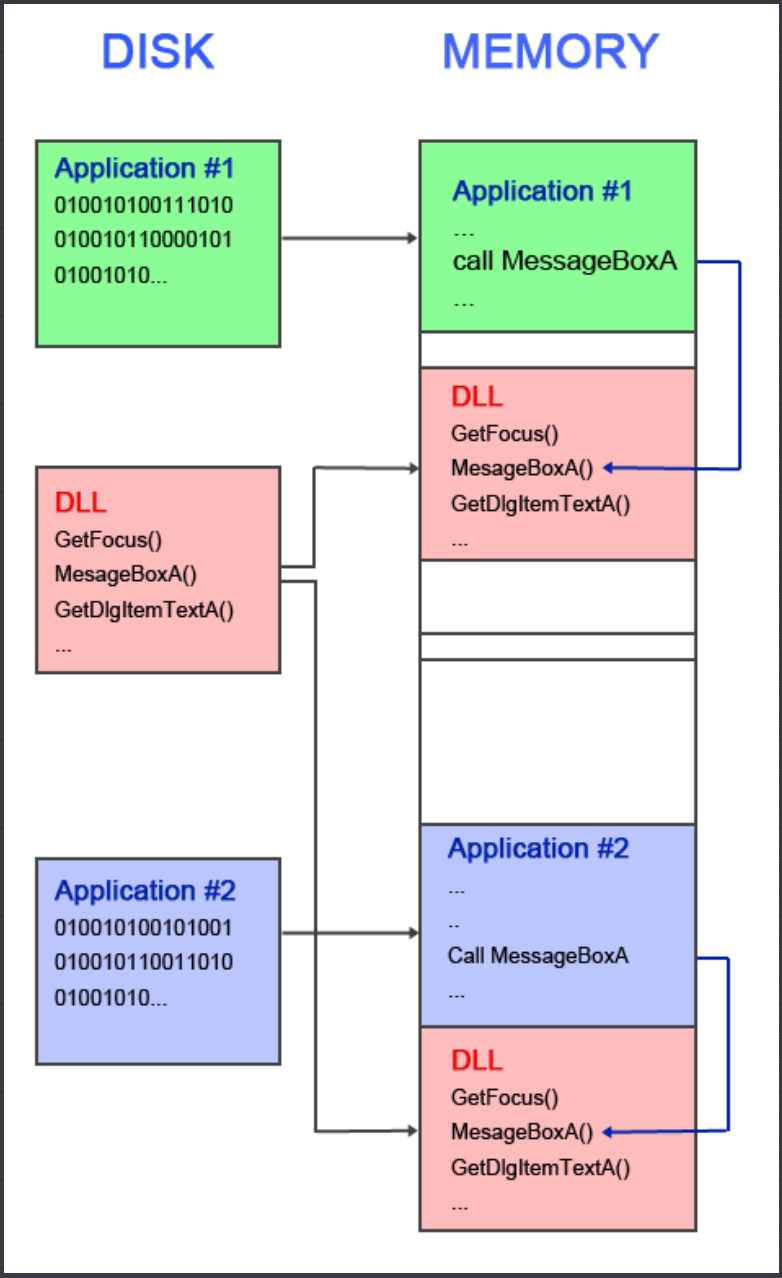


What we want to do is add a nag message box before the main screen is displayed. In this case we will use DLL injection.

DLLs

A DLL file is a collection of functions that, theoretically anyway, many applications need. Instead of having the same function in every application, each application that needs a specific function can call the required function inside of a DLL file, so that there only has to be one DLL file that every app uses. This cuts down on code and memory use.

The name, Dynamic Link Library gives some of this away. It is a 'library', a collection of functions that multiple apps use. These functions, available in the DLL, are 'linked' in with the application that needs them. This is done 'dynamically' when the app is first loaded into memory. Perhaps a picture will help:

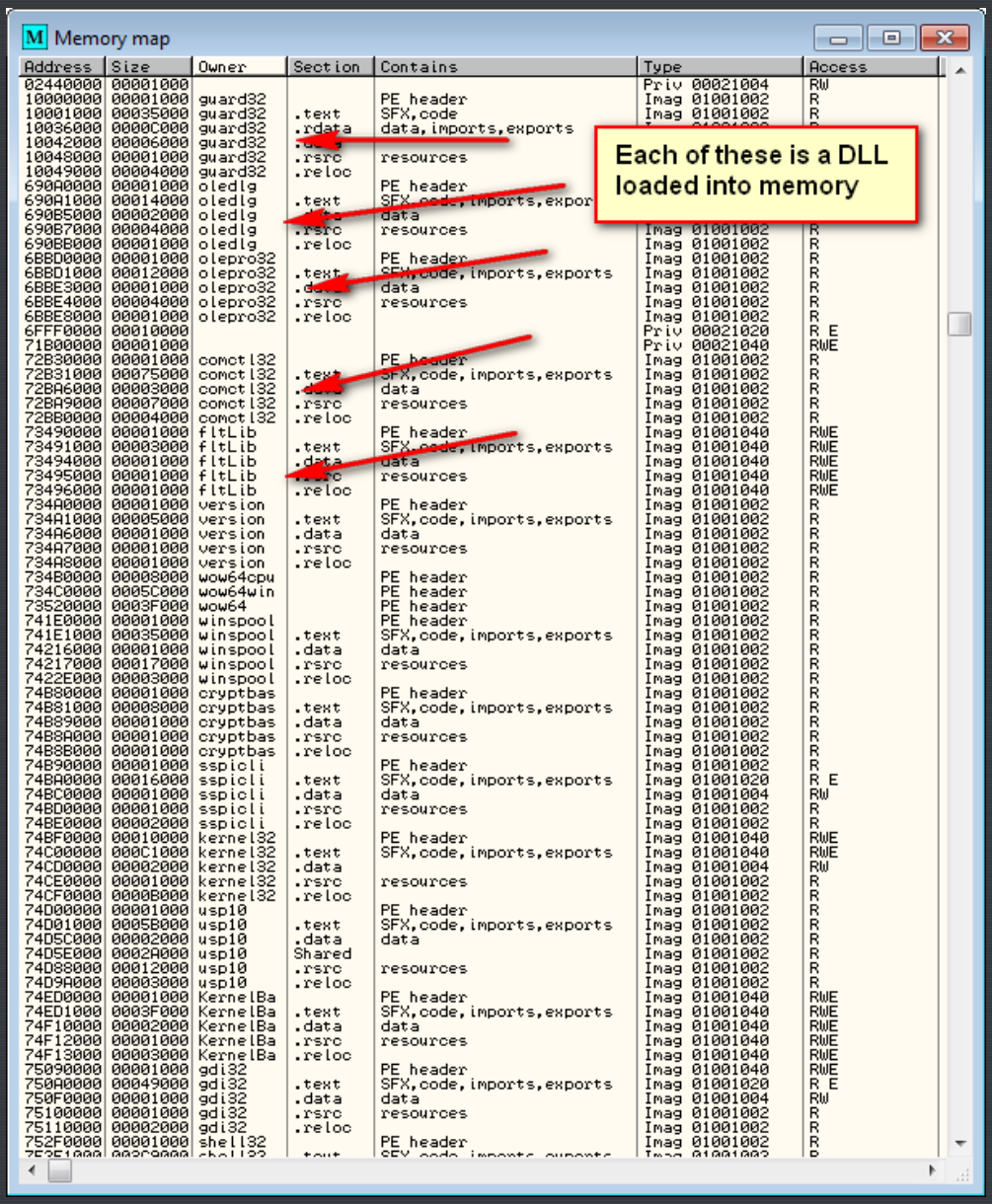


When a normal file is loaded, any DLL files that the application needs are loaded into the address space of that application, so that the app can make calls to APIs inside that DLL. If another Windows program is started that also needs a function from the same DLL, the DLL is also copied into the address space of this new app. For example, Kernel32.dll is loaded with (just about) every application in Windows. Kernel32.dll offers several methods that an application can call, for example ExitProcess that closes a window. Any application that needs to be able to close itself will have Kernel32.dll loaded into its address space, and will have the ExitProcess function available to call.

You may say to yourself, "If a DLL is loaded into every program's memory space that needs it, it doesn't really cut down on memory as every program has a copy of it, so they are not sharing." well, Windows has a special way of dealing with DLLs that even though the applications think that they each have a copy of the DLL all to themselves, they all really share the same code. Windows just makes it seem like they all

have a copy.

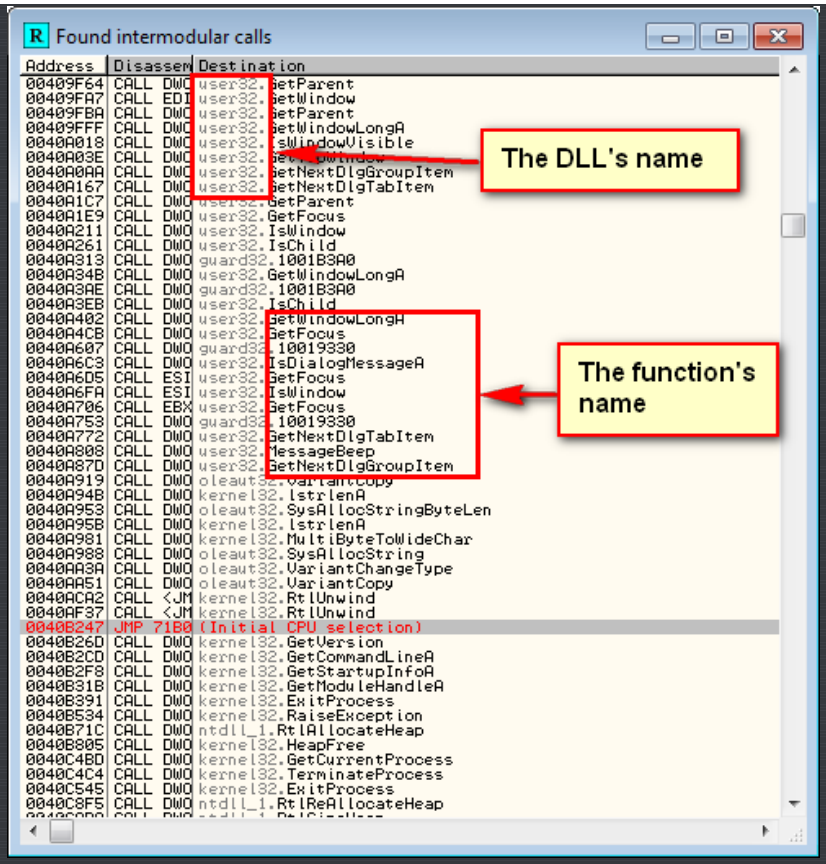
If you load up the target into Olly and look in the memory window, you will see all of the DLLs loaded into the applications memory space:



Address	Size	Owner	Section	Contains	Type	Access
02440000	00001000				Priv 00021004	RW
10000000	00001000	guard32		PE header	Imag 01001002	R
10001000	00035000	guard32	.text	SFX,code	Imag 01001002	R
10036000	0000C000	guard32	.rdata	data,imports,exports		
10042000	00006000	guard32	.data			
10048000	00001000	guard32	.rsrc	resources		
10049000	00004000	guard32	.reloc			
690A0000	00001000	oledlg		PE header		
690A1000	00014000	oledlg	.text	SFX,code,imports,expor		
690B5000	00002000	oledlg	.rdata	data		
690B7000	00004000	oledlg	.rsrc	resources	Imag 01001002	R
690BB000	00001000	oledlg	.reloc		Imag 01001002	R
6BBD0000	00001000	olepro32		PE header	Imag 01001002	R
6BBD1000	00012000	olepro32	.text	SFX,code,imports,exports	Imag 01001002	R
6BBE3000	00001000	olepro32	.data	data	Imag 01001002	R
6BBE4000	00004000	olepro32	.rsrc	resources	Imag 01001002	R
6BBE8000	00001000	olepro32	.reloc		Imag 01001002	R
6FFF0000	00010000				Priv 00021020	R E
71B00000	00001000				Priv 00021040	RWE
72B30000	00001000	comctl32		PE header	Imag 01001002	R
72B31000	00075000	comctl32	.text	SFX,code,imports,exports	Imag 01001002	R
72BA6000	00003000	comctl32	.data	data	Imag 01001002	R
72BA9000	00007000	comctl32	.rsrc	resources	Imag 01001002	R
72BB0000	00004000	comctl32	.reloc		Imag 01001002	R
73490000	00001000	fltLib		PE header	Imag 01001040	RWE
73491000	00003000	fltLib	.text	SFX,code,imports,exports	Imag 01001040	RWE
73494000	00001000	fltLib	.data	data	Imag 01001040	RWE
73495000	00001000	fltLib	.rsrc	resources	Imag 01001040	RWE
73496000	00001000	fltLib	.reloc		Imag 01001040	RWE
734A0000	00001000	version		PE header	Imag 01001002	R
734A1000	00005000	version	.text	SFX,code,imports,exports	Imag 01001002	R
734A6000	00001000	version	.data	data	Imag 01001002	R
734A7000	00001000	version	.rsrc	resources	Imag 01001002	R
734A8000	00001000	version	.reloc		Imag 01001002	R
734B0000	00008000	wow64cpu		PE header	Imag 01001002	R
734C0000	0005C000	wow64win		PE header	Imag 01001002	R
73520000	0003F000	wow64		PE header	Imag 01001002	R
741E0000	00001000	winspool		PE header	Imag 01001002	R
741E1000	00035000	winspool	.text	SFX,code,imports,exports	Imag 01001002	R
74216000	00001000	winspool	.data	data	Imag 01001002	R
74217000	00017000	winspool	.rsrc	resources	Imag 01001002	R
7422E000	00003000	winspool	.reloc		Imag 01001002	R
74B80000	00001000	cryptbas		PE header	Imag 01001002	R
74B81000	00008000	cryptbas	.text	SFX,code,imports,exports	Imag 01001002	R
74B89000	00001000	cryptbas	.data	data	Imag 01001002	R
74B8A000	00001000	cryptbas	.rsrc	resources	Imag 01001002	R
74B8B000	00001000	cryptbas	.reloc		Imag 01001002	R
74B90000	00001000	sspicli		PE header	Imag 01001002	R
74BA0000	00016000	sspicli	.text	SFX,code,imports,exports	Imag 01001020	R E
74BC0000	00001000	sspicli	.data	data	Imag 01001004	RW
74BD0000	00001000	sspicli	.rsrc	resources	Imag 01001002	R
74BE0000	00002000	sspicli	.reloc		Imag 01001002	R
74BF0000	00010000	kernel32		PE header	Imag 01001040	RWE
74C00000	000C1000	kernel32	.text	SFX,code,imports,exports	Imag 01001040	RWE
74CD0000	00002000	kernel32	.data		Imag 01001004	RW
74CE0000	00001000	kernel32	.rsrc	resources	Imag 01001002	R
74CF0000	0000B000	kernel32	.reloc		Imag 01001002	R
74D00000	00001000	usp10		PE header	Imag 01001002	R
74D01000	0005B000	usp10	.text	SFX,code,imports,exports	Imag 01001002	R
74D5C000	00002000	usp10	.data	data	Imag 01001002	R
74D5E000	0002A000	usp10	Shared		Imag 01001002	R
74D88000	00012000	usp10	.rsrc	resources	Imag 01001002	R
74D9A000	00003000	usp10	.reloc		Imag 01001002	R
74ED0000	00001000	KernelBa		PE header	Imag 01001040	RWE
74ED1000	0003F000	KernelBa	.text	SFX,code,imports,exports	Imag 01001040	RWE
74F10000	00002000	KernelBa	.data	data	Imag 01001040	RWE
74F12000	00001000	KernelBa	.rsrc	resources	Imag 01001040	RWE
74F13000	00003000	KernelBa	.reloc		Imag 01001040	RWE
75090000	00001000	gdi32		PE header	Imag 01001040	RWE
750A0000	00049000	gdi32	.text	SFX,code,imports,exports	Imag 01001020	R E
750F0000	00001000	gdi32	.data	data	Imag 01001004	RW
75100000	00001000	gdi32	.rsrc	resources	Imag 01001002	R
75110000	00002000	gdi32	.reloc		Imag 01001002	R
752F0000	00001000	shell32		PE header	Imag 01001002	R

We can see that this application uses several DLLs, with names like Guard32.dll, Oledlg.dll, and Olepro32.dll.

If you then do a search for all intermodular calls, you will see all of the functions available from each of the DLLs available to this application:



A DLL file is very similar to a normal executable file except all (or most) of it's functions are set up to be used by other applications instead of itself. When you create a DLL, you set up a certain file, called a DEF file (for 'definition'), that lists all of the function names that will be available for other applications to use. When you then assemble and link this DLL, the compiler makes a table that can be used, with all of the names and associated addresses of all of the functions. This is how the application finds these functions (well, not exactly, but close enough).

Just like a 'normal' .exe file, a DLL has a "main" function, called DllEntry, that is executed when the DLL is loaded into any program's address space. The difference between a DLL and a normal binary is that the main function of a DLL is called whenever the DLL is loaded into ANY applications memory space - you do not need to actually run the DLL. This allows the DLL to set up any housekeeping it needs before making itself available to the application. This is where DLL injection come is.

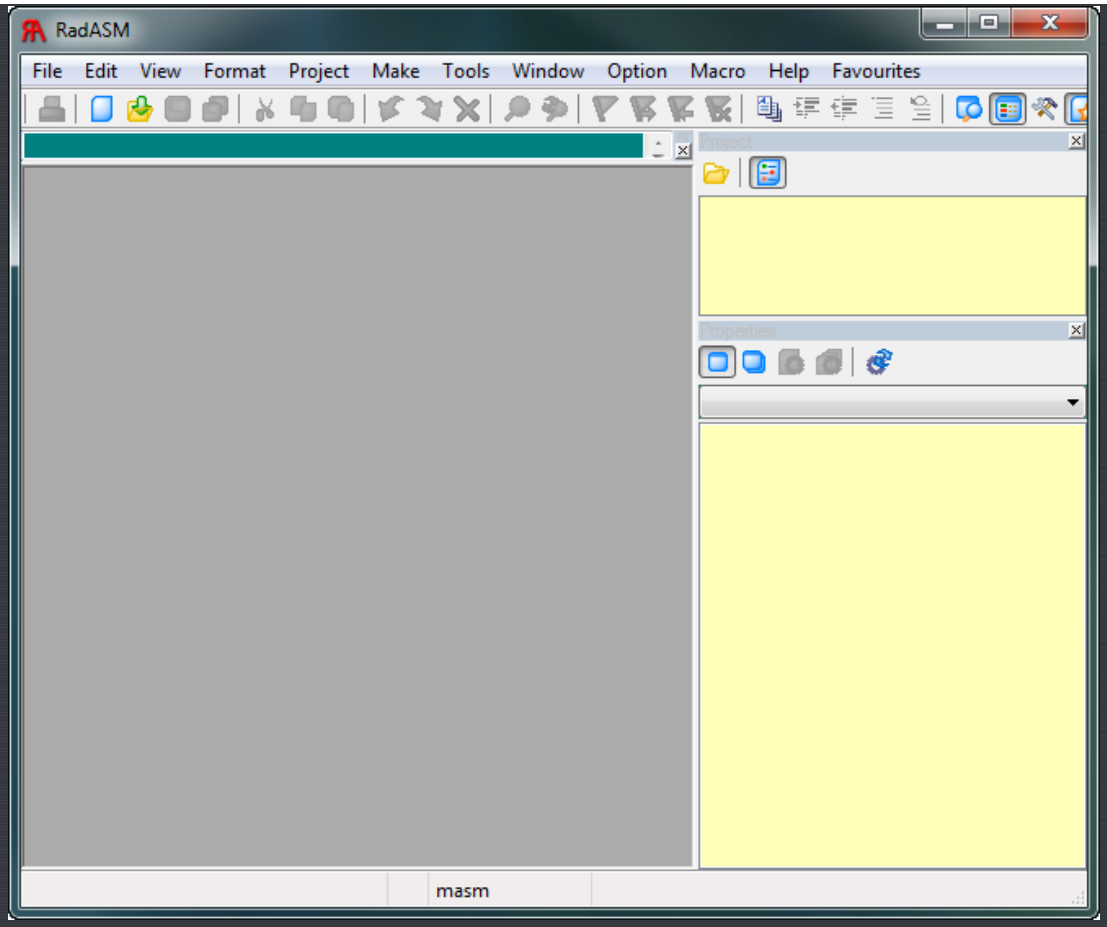
DLL Injection

DLL Injection is a way of injecting our own DLLs into an executable that wasn't initially set up to use it. You first create your own DLL, and then you add it to an application. You can do this programatically, though we won't do so in this tutorial, as that is quite detailed. Here, we will use a program called IIDKing by Santmat. What it does is loads the target application and injects our DLL into it, saving the modified target after. When the target app loads, before any code of the application is run, the Windows loader will load our DLL along with any others the application needs. Anything we put inside the DLL main function will be run automatically (as long as we tell the compiler that is should be run automatically-see below). We can also modify the application to call our DLL whenever we want. Therefore, our DLL is injected into another binary.

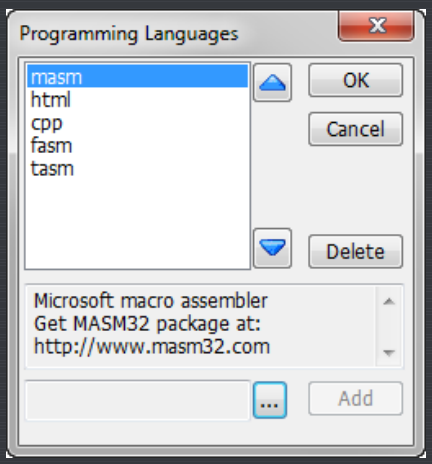
DLL Injection can be used for a lot of things; patching a binary, cracking an application, keygenning, unpacking, virus writing.

Writing a DLL is not that hard. In fact, it is very much like writing a normal .exe file. Really, the only difference is how you compile it. In this tutorial, we will add a message box when the application is first started.

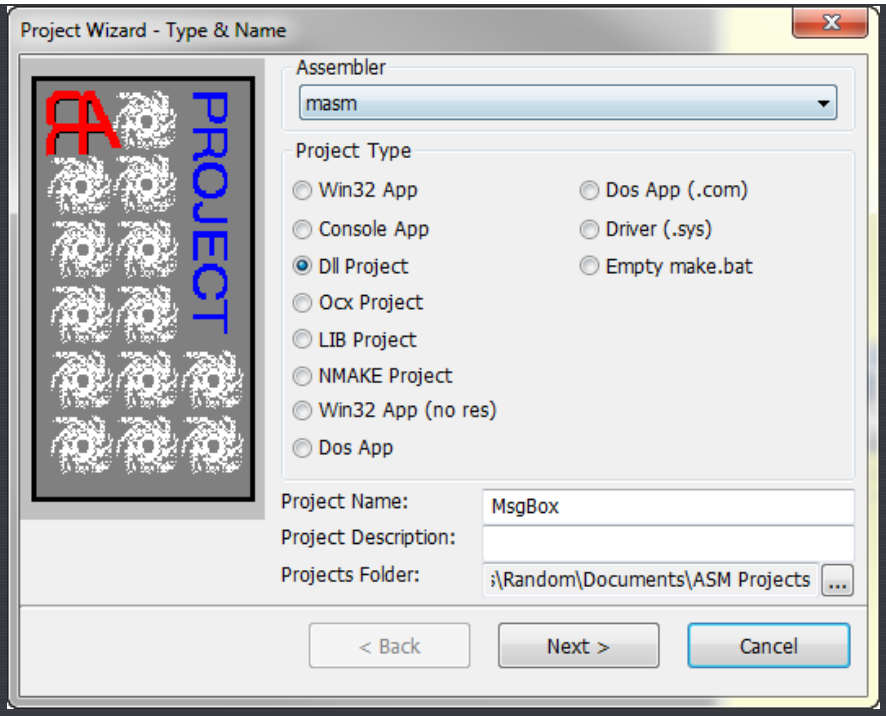
Let's get started creating a DLL. Start up RadASM:



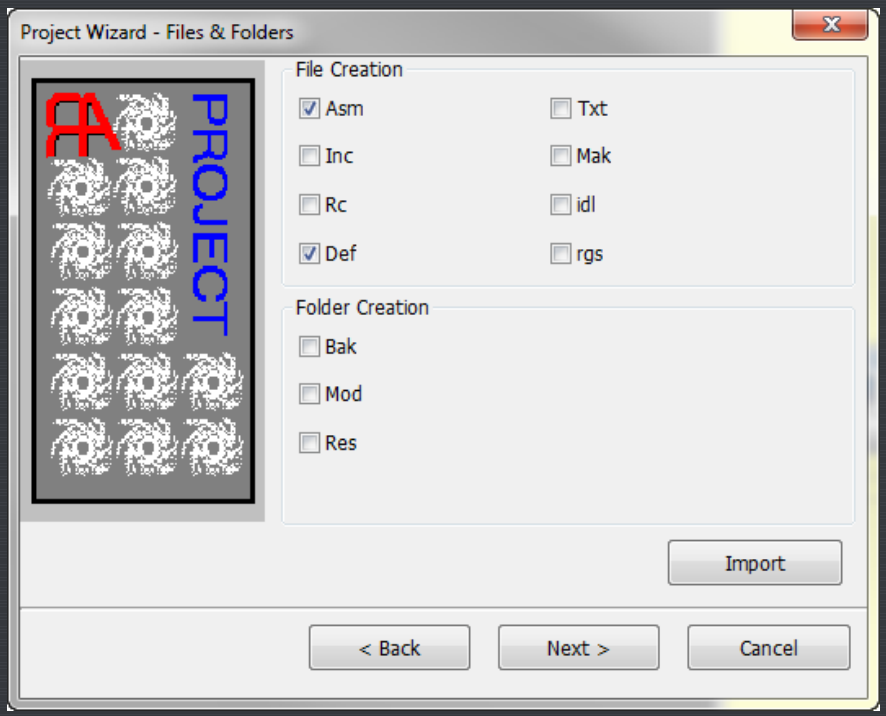
If you haven't loaded MASM as a language yet, do so by selecting Options -> Programming Languages. Select the three dots ("…") to add a language and choose the masm.ini file:



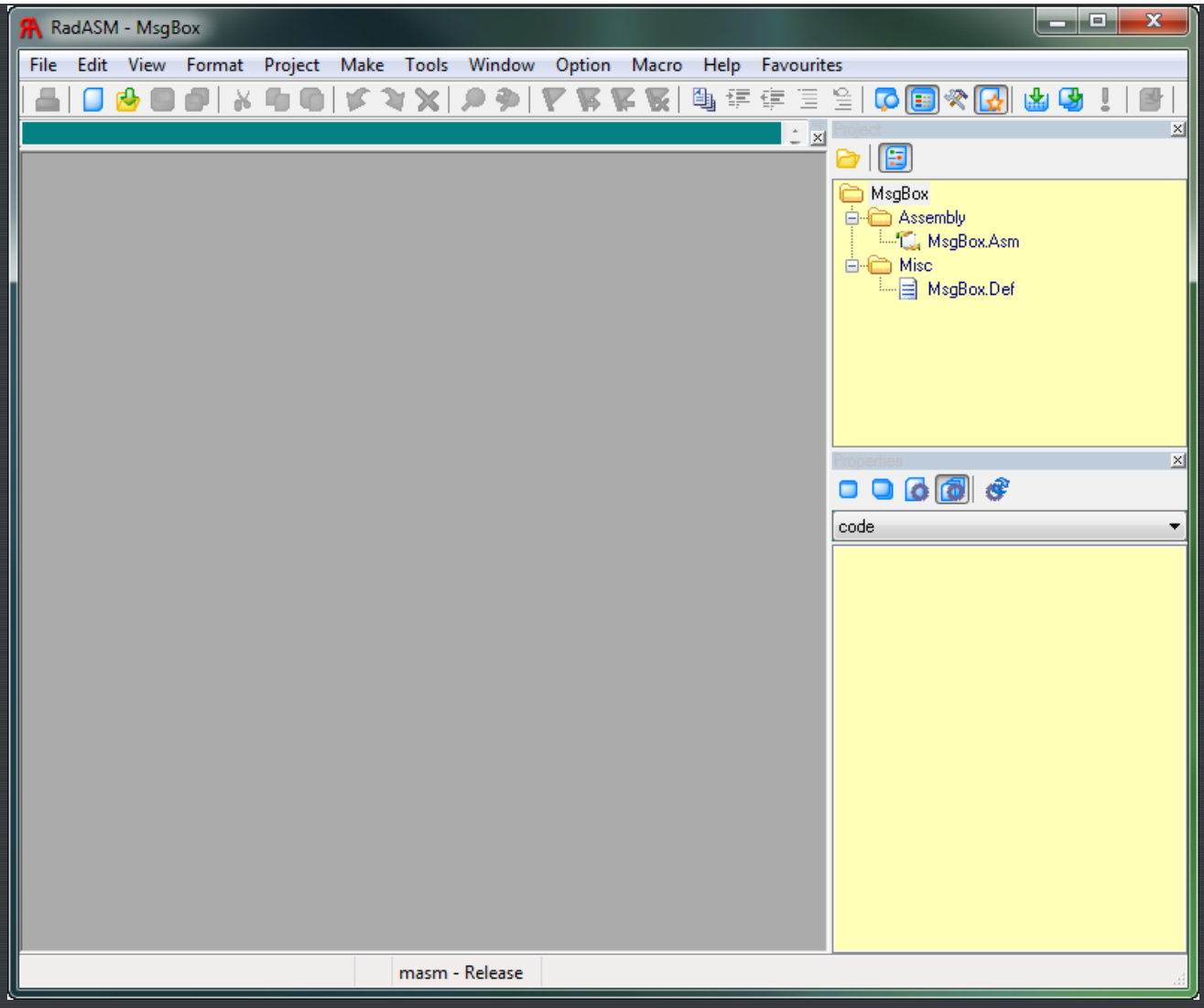
Now, create a new project by selecting File -> New Project. Make sure MASM is selected in the Assembler drop down, and choose "Dll Project". Type in a name for your project and click next:



Select “None” for the template and click Next. Make sure “Def” is selected, along with “Asm”:



Click Next. Leave all the options the same in the “Make” window and click Finish. You should now have a blank project:



Now open the MsgBox.asm file by double-clicking it in the project tree. Right now, there is nothing there, so you will get a blank screen. Now, let's add the DLL code in:

****The code is also included in the download of this tutorial****

We'll go over this code piece by piece. First we declare some housekeeping stuff, telling the compiler which CPU we're running on and what kind of calling conventions to use:

```
.386
.model flat,stdcall
option casemap:none
```

Next, we define any files that this DLL needs, namely some Windows files that contain the code for the MessageBox and other behind-the-scene functions:

```
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

Now we declare some strings that we will use, namely the title of the opening nag and the message of the opening nag:

```
.data
AppName1        db "Windows format!",0
LoadMsg         db "Click OK to format your hard drive…",0
```

Finally, we define the code for the DllEntry function. First is the actual definition:

```
.code
DllEntry proc hInstance:HINSTANCE, reason:DWORD, reserved1:DWORD
```

Next we have an IF statement for when we load the DLL. anything we put under the "reason==DLL_PROCESS_ATTACH" statement will be run when the application laods in the DLL In this case we just bring up a dialog box:

```
.if reason==DLL_PROCESS_ATTACH
invoke MessageBox,NULL,addr LoadMsg,addr AppName1,MB_OK
.endif
```

At the end, we return a true in the EAX register. This is a normal way for a DLL to return:

```
mov  eax,TRUE
ret
```

And finally, we end the procedure definition:

```
DllEntry Endp
End DllEntry
```

The last thing we need to do is create an actual function that could theoretically be called. I say 'theoretically' because in our case it will not be called. The reason we're creating it is you need at least one callable function in the DLL in order to inject it. So we will create a dummy function. Insert it between the last two lines, (the "DLLEntry endp" and "End DllEntry" lines):

```
TestProc proc
   invoke MessageBox,NULL,addr LoadMsg,addr AppName1,MB_OK
   ret
TestProc endp
```

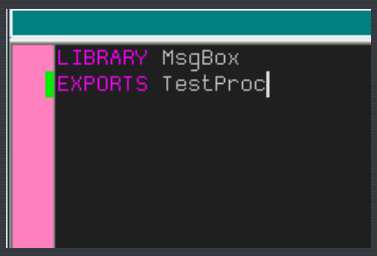
This just invokes another message box, but we'll never see it as it's never called.

```
.386
.model flat,stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
AppName1         db "Windows format!",0
LoadMsg          db "Click OK to format your hard drive...",0


.code
DllEntry proc hInstance:HINSTANCE, reason:DWORD, reserved1:DWORD

    ; DLL loaded
    .if reason==DLL_PROCESS_ATTACH
        invoke MessageBox,NULL,addr LoadMsg,addr AppName1,MB_OK
    .endif

    mov  eax,TRUE
    ret

DllEntry endp

TestProc proc

    invoke MessageBox,NULL,addr LoadMsg,addr AppName1,MB_OK
    ret
TestProc endp

End DllEntry
```

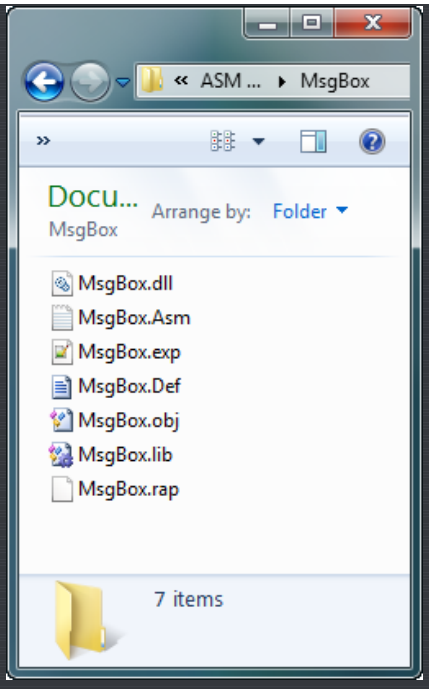
Now let's create the DEF file. Right click in the project window tree and choose "Add New" -> "File". Save the file as 'MsgBox.Def'. Now let's put in our definition:

```
LIBRARY MsgBox
EXPORTS TestProc
```

This tells the compiler that the name of our library will be "MsgBox" and the function that can be called in it is called "TestProc". That's it. Now this DLL will make the TestProc function available to all applications that use this DLL:



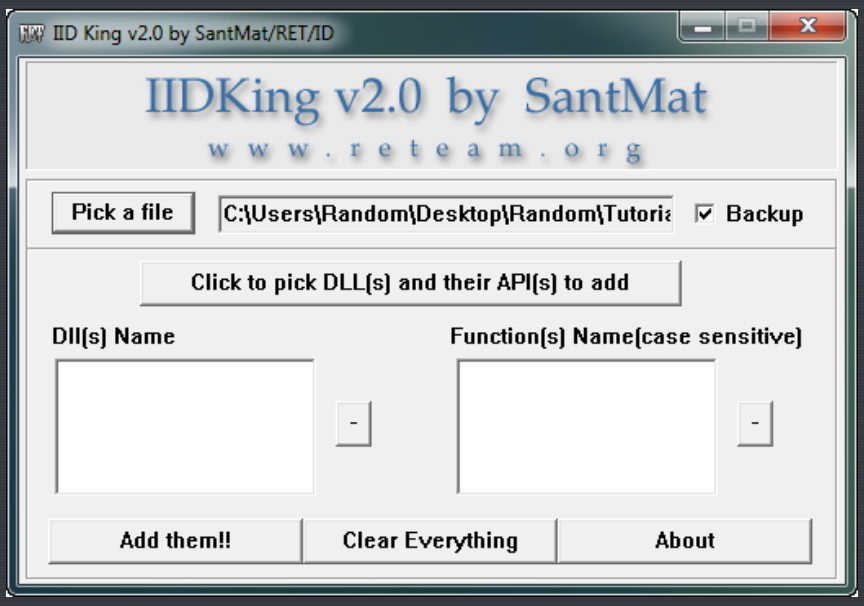
It's time to compile our DLL. Hit F5 to assemble the project, creating the .obj file. If there are any errors, they will appear at the bottom of the screen. Otherwise it will say the build was made. Now link it by selecting "Make" -> "link". If everything ran as expected, there will be a message saying the make was done. You will also have a DLL file in the project folder:



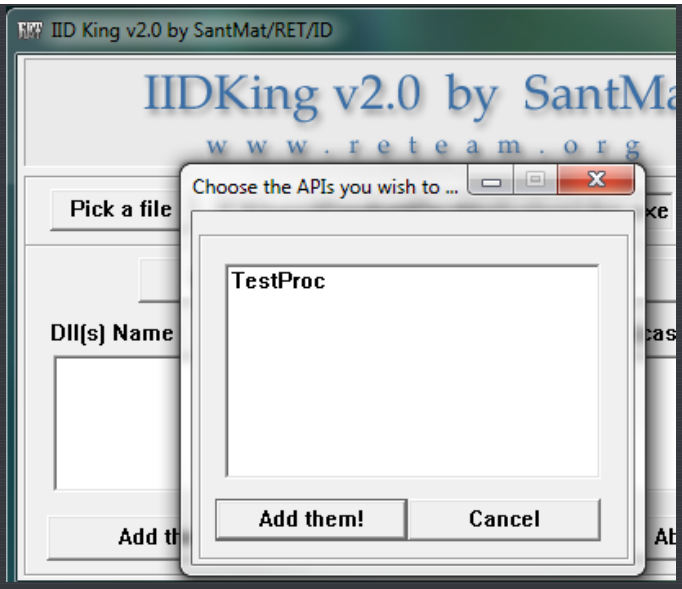
We now have a legitimate DLL file 😀 .

Injecting Our DLL File

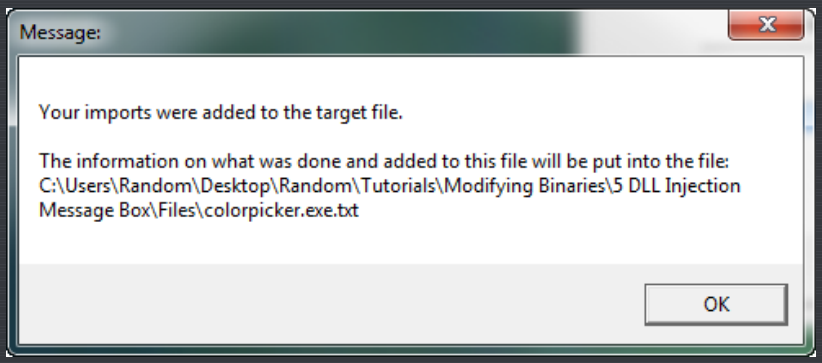
Now that we have our DLL, we will inject it into our target. Start up IIDKing. Click the “Pick a file” button and select the target, in this case the ColorPicker.exe file:



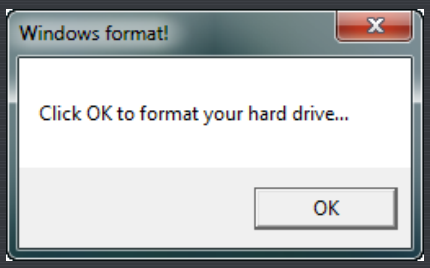
Now we need to select our DLL. Click on the “Click to pick DLL” button and select our DLL file. That will bring up the API selection box:



Select our TestProc function and click the “Add them!” button. Then, on the main IIDKing screen, click “Add them!”. You will get a message saying it was successfully added to the target:



Now for the coup d’grace…Start the target. You will see our first message box appear:



Clicking OK will bring up the main application’s window.

You have now injected a DLL into an executable…

-Till next time

R4ndom

[=-Docendo discimus-=]