What is the Universal Hooker?

The Universal Hooker is a tool to intercept execution of programs. It enables the user to intercept calls to API calls inside DLLs, and also arbitrary addresses within the executable file in memory.

Why is it 'Universal'? There are different ways of hooking functions in a program, for example, it can be done by setting software breakpoints (int 3h), hardware breakpoints (cpu regs), or overwriting the prologue of a function to jump to a 'stub', etc. All the methods mentioned required above, specially the latter, require the programmer of the code creating the hook to have certain knowledge of the function it is intercepting. If the code is written in a programming language like C/C++, the code will need to be recompiled for every function one wants to intercept, etc.

The Universal Hooker tries to create very simple abstractions that allow a user of the tool to write hooks for different API and non-API functions using an interpreted language (python), without the need to compile anything, and with the possibility of changing the code that gets executed whent the hooked function is called in run-time.

The Universal Hooker builds on the idea that the function handling the hook is the one with the knowledge about the parameters type of the function it is handling. The Universal Hooker only knows the number of parameters of the function, and obtains them from the stack (all DWORDS). The hook handler is the one that will interpret those DWORDS as the types received by the function.

The hook handlers are written in python, what eliminates the need for recompiling the handlers when a modification is required. And also, the hook handlers (executed by the server) are reloaded from disk every time a hook handler is called; this means that one can change the behavior of the hook handler without the need to recompile the code, or having to restart the application being analyzed.

How does it work?

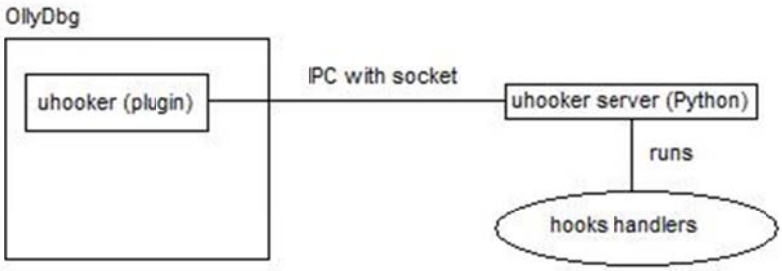
I have written several versions of the Universal Hooker, using different techniques to hook functions such as implementing my own 'debugger', modifying prologue of functions, etc. The version available at this web page is implemented as an OllyDbg plugin, so the hooking of functions are taken care by the OllyDbg Debugger (using software breakpoints).

The basic components of the Universal Hooker are:

- The universal hooker core, implemented as a OllyDbg plugin (uhooker.dll)
- A Configuration File (e.g: hook.cfg)
- a Server (server.py) written in python that handles communication with the universal hooker core
- a library written in python (proxy.py) that contains different functions to communicate with the universal hooker core.

These functions allow the developer to perform different actions on the intercepted process, for example: read memory, write memory, etc.

- A python module written by the developer that contains the code that handles the hooked functions/addresses. This module uses the python library proxy.py to perform actions on the intercepted process.



A configuration file is loaded from Ollydbg that defines what functions/addresses to hook, after parsing the configuration file the uhooker core connects to the server and sends the hook information. Every time a function hook is triggered, the uhooker core communicates with the server sending information about the function/address hooked, and the server executes the corresponding hook handler as defined in the configuration files.

The configuration file

The configuration files is a regular text file where each line defines the functions/addresses to intercept (All lines beginning with '#' are treated as comments).

There are 3 different types of hooks:

1. Hook Before Entering the function (type 'B')
2. Hook After the function returns (type 'A')
3. Hook when executing reaches this address (type '*')

To intercept functions exported from a DLL the syntax is:

name_of_dll:function_name:number_of_parametes:python_module.hook_handler_name:hook_type

for example, to hook 'CreateFileA' (which has 7 parameters) before it is executed with a handler called 'CreateFileA_handler' implemented in the file mymodule.py:

kernel32.dll:CreateFileA:7:mymodule.CreateFileA_handler:B

As explained above, it is possible to hook any 'executable' address of a process (which is basically the same as setting a breakpoint on an address):

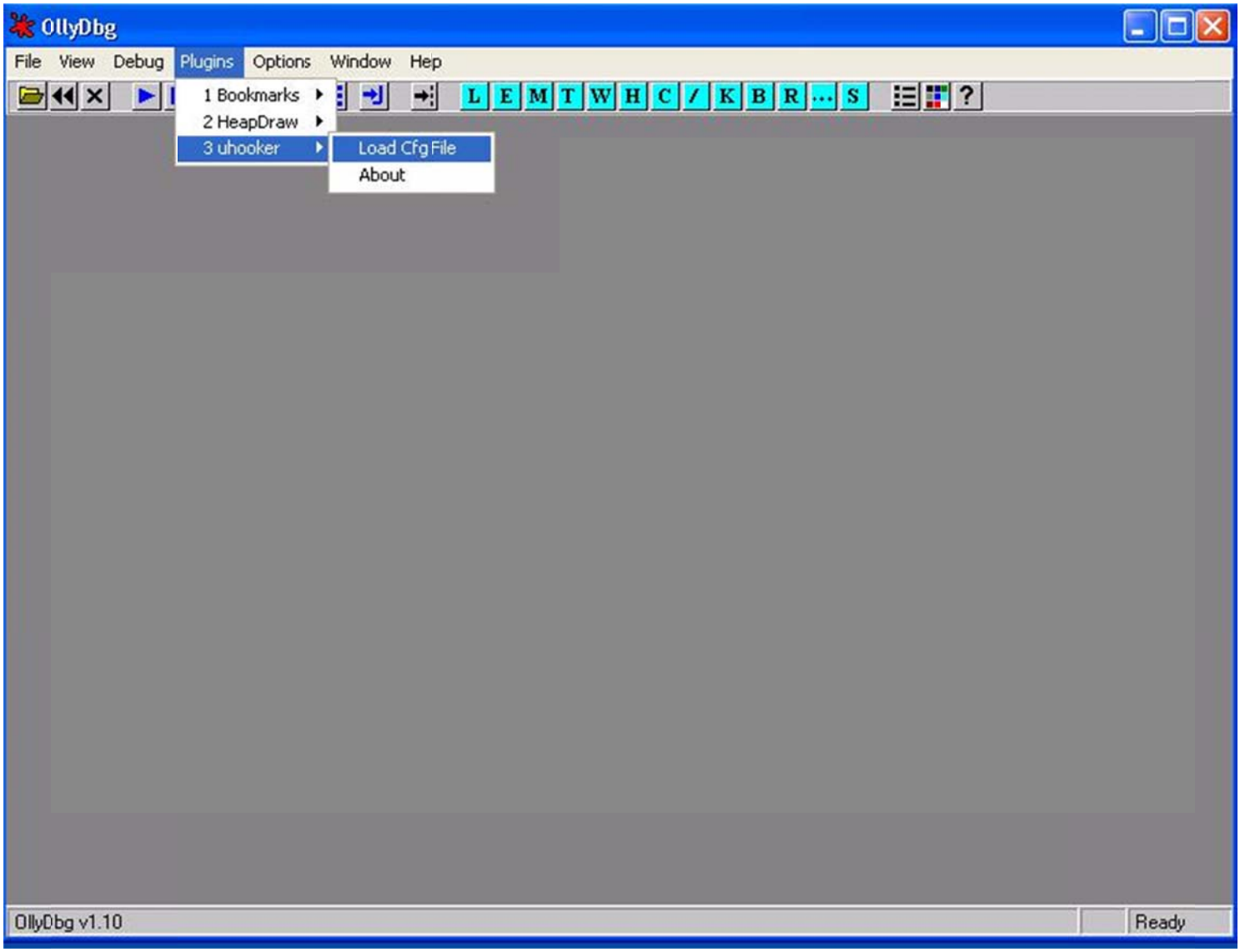
field_not_used:address_to_hook_in_hex:field_not_used:python_module.hook_handler_name:hook_type

So, for example, to hook execution at 0x401000 handled by the hook handler called 'anybp' implemented in the mymodule.py file:
dummy.dll:0x401000:0:mymodule.anybp:*

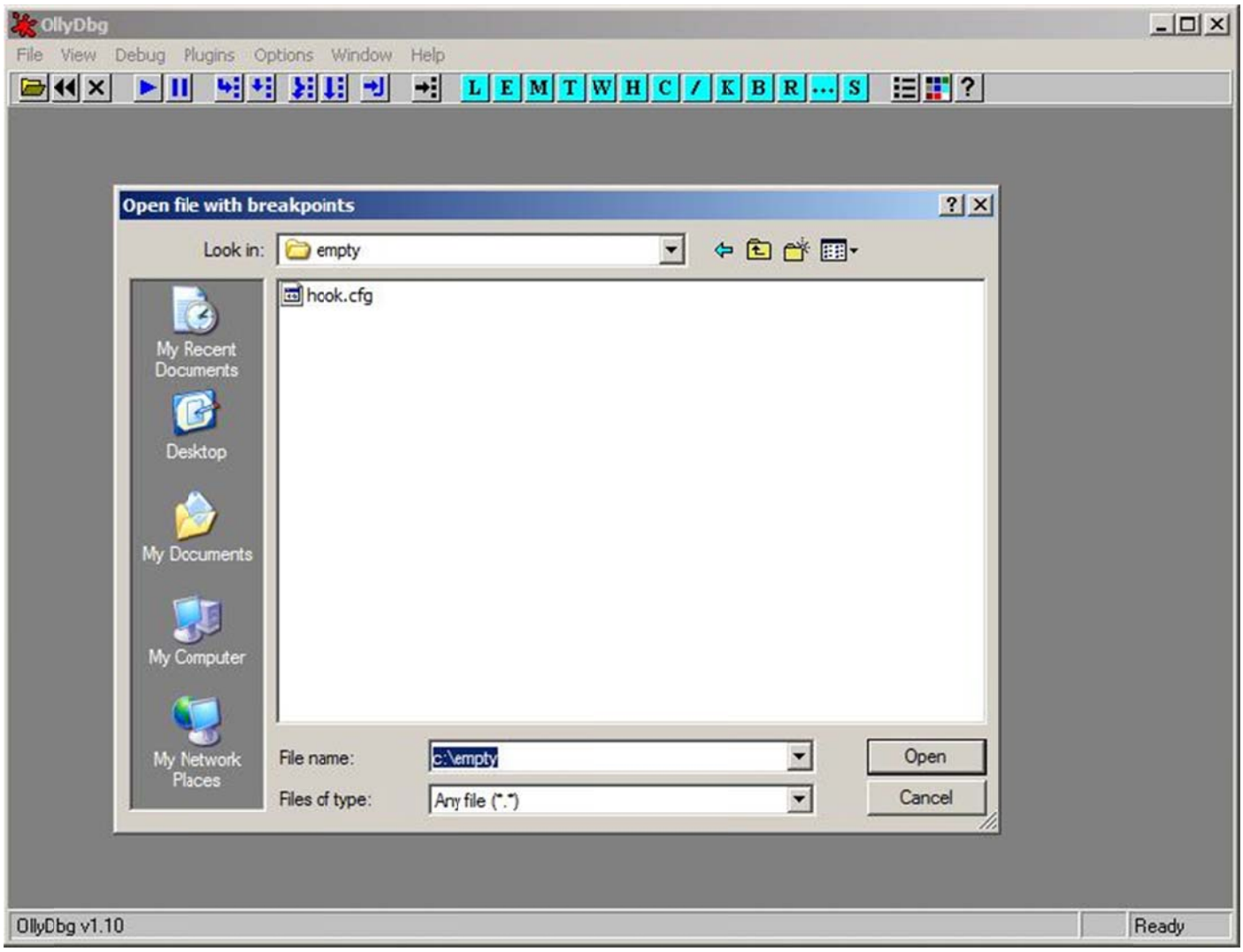
Next is a sample of a configuration file:

```
# B = hook before entering function
# A = hook AFTER function returns
# * = hook when an address is executed
kernel32.dll:CreateFileA:7:mymodule.CreateFileA_handler:B
dummy.dll:0x401000:0:mymodule.anybp:*
```

The configuration file can be loaded from OllyDbg by going to the 'plugins->uhooker' menu, and selecting 'Load Cfg File'. See the screenshot below.



A file dialog will be displayed from which a .cfg file (in fact any name can be used) can be loaded.



After parsing the configuration file, the uhooker core will try to set breakpoints on the functions/addresses indicated in the configuration file, for that reason before loading the configuration file the process to intercept should be loaded in the debugger or the debugger should be attached to the process.

The Hooks Handlers

A hook handler is basically a python script that is called every time the intercepted function/address is called. It has the following definition:
def hook_name(hookcall):
The hookcall parameter is an object passed by the server to the hook handler that contains useful information about the intercept function and process. For example:

- **hookcall.regs**: contains the content of the registers of the intecepted process (e.g.: hookcall.regs['eax']).
- **hookcall.params**: contains parameters of the intercepted function. As explained above, the parameters are only DWORDs as far as the uhooker core knows, so this 'params' list contains basically a list of DWORDs that the hook handler must interpret as char*, ints, or any other type depending on the function it is handling. The list is zero-based, meaning hookcall.params[0] is the first parameter to the function.

- **hookcall.retaddr**: contains the return address of the function.
- **hookcall.threadid**: contains thread id of current thread
- **hookcall.procid**: contains process id of current process
- **hookcall.sendack()**: All hook handlers must end with a call to this function. Otherwise, the uhooker core will hang forever. This function returns control to ollydbg and resumes execution of the program being debugged.

- **hookcall.sendacknocont()**: handlers can also end with a call to this function. This function does the same

thing as hookcall.sendack(), but it DOES NOT resume execution of the debugged program. This is good, for example, for scripts that want to check for certain condition, and then stop the debugger to allow the user to continue debugging manually.

A hook handler can also read and write memory of the intercepted process, allocate memory, etc. All these functions are available from the Proxy.py module, so all hook handler also import and create an instance of the 'Proxy' object.

Next is a sample hook handler:

```
def CreateFileA_handler(hookcall):
    myproxy = hookcall.proxy
    print "bughandler running..."
    print "esp = %X" % hookcall.regs['esp']
    print "retaddr = %X" % hookcall.retaddr
    print "arg0 = %X" % hookcall.params[0]
    buffer = myproxy.readasciiz( hookcall.params[0] )
    print buffer
    hookcall.sendack()
    return
```

This handler:

- Obtains an instance of the 'Proxy' object. (*IMPORTANT*: before uhooker v1.2, you needed to create the instance by doing a 'myproxy = proxy.Proxy()', ths is no longer the case, if you are using uhooker v1.2, please change your scripts to use this new method)
- Prints meaningless information to the console ("CreateFileA handler running...")
- Prints to the console the value of the ESP register, the return address of the function, and the value (DWORD) of the first parameter of the function.
- Since this is a handler for the CreateFileA function, and the first parameter (hookcall.params[0]) is a pointer to an ASCII string indicating the file to open, it uses the 'readascizz' function, to read that ascii string and prints it to the console.
- Finally, it calls sendack() to inform to the uhooker core that it has finished processing the hook, and returns.

Using the Universal Hooker

To use uhooker the steps that need to be done are:

- Have a valid configuration file defining what functions/address to hook and indicating which are the handlers for those hooks
- Have a .py file with the hook handlers indicated in the configuration file
- Start OllyDbg, load or attach to the process to intercept
- Load the configuration file. The uhooker core will set the breakpoints, start the python server automatically and send the hook information to the server.
- That's all. Every time the function/address is called, the proper hook handler will be executed.