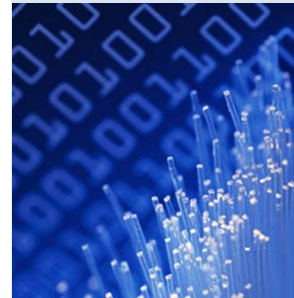




High-Level Reverse Engineering

An IRM Research White Paper by
Matthew Lewis



IRM Research

Information technology constantly changes and advances. IRM is dedicated to keeping pace with new technology and continuing to innovate in the field of information security. This ensures that we are all well informed of new issues and technologies, expanding our knowledge and providing world class services to our clients.

High-Level Reverse Engineering

This paper aims to present a methodical framework for high-level reverse engineering. The methodology is a culmination of existing tools and techniques within the IT security research community, which presents ways to identify process operation at a higher-level of abstraction than traditional binary reversing. Here, we focus our attention on application DLLs and the functions that they implement and export, which includes process interactions with other applications and various operating system function calls. We use existing tools and techniques to derive ways of quickly identifying how applications are constructed, the functions that they use and how they use them. Following this high-level reverse engineering, the researcher is then free to take further steps at reversing specific functions with the more traditional lower-level binary analysis.

It is anticipated that analysing applications in this way will allow the researcher to focus his/her attentions on specific functions that appear more “interesting” from a security perspective. For example, identifying a commonly called function within an application that processes user-supplied input is a far better choice for further analysis at lower levels, as the function may be susceptible to inappropriate data handling resulting in one or more of the common overflow-type vulnerabilities that can lead to system compromise. The techniques described within this paper will also allow the researcher to identify specific points within the application which may be fuzzed, allowing for further vulnerability investigations into the application and its functions.

This whitepaper is specific to Microsoft Windows operating system applications; however the methods and techniques could be developed and applied to any other operating system capable of function hooking. A worked example is presented throughout this paper, providing an investigation of the *Microsoft Fingerprint Reader* functionality (manufactured by *Digital Persona*). The device connects to a Microsoft Windows operating system via USB, and is used to provide fingerprint-based authentication for system and web application logon. We identify the DLLs and system calls that the device and application invokes, providing methods to intercept and manipulate these calls in ways that allow the researcher to perform security investigations on specific components.

The key tools required and used throughout the methodology are the Universal Hooker (*uhooker*) by Core Security Technologies [1], the Interactive Disassembler (IDA) [2] and the OllyDbg debugger [3]. It is assumed that the reader is already familiar with these tools. Further information on these tools and their operation can be found from the references section at the end of this document.

Brief Overview of the Universal Hooker

The Universal Hooker (*uhooker*) is a tool to intercept execution of programs. It enables the user to intercept calls to API functions inside DLLs and also arbitrary addresses within the executable file in memory. *Uhooker* builds on the idea that the function handling the hook is the one with the knowledge about the parameter types of the function it is handling. *Uhooker* only knows the number of parameters of the function and obtain those (DWORDS) from the stack. Hook handlers are written in python, which eliminate the need for recompiling the handlers when modification is required. *Uhooker* is implemented as an OllyDbg plugin, which takes care of function hooking using software breakpoints [1].

Uhooker components comprise the *uhooker* core (OllyDbg plugin), a configuration file, a server (*server.py*) which handles communication with the *uhooker* core, a python library (*proxy.py*) containing functions to communicate with *uhooker*, such as read/write memory, and a python handler module written by the researcher that contains the code to execute on hooked functions and/or addresses [1].

As part of this whitepaper, we present methods for automating the generation of configuration and handler files. For the handler files, stub code is generated, which allows the researcher to quickly implement the desired hook functionality on specific functions or addresses.

Phase One: Identifying Relevant Components

For this whitepaper, we embark on investigation of the *Microsoft Fingerprint Reader* from a “black-box” perspective, meaning that our first phase demands identification of the core components of the system under investigation. A number of methods are available to us at this stage; which primarily depend on the nature of the system being investigated. For example, open-source research from Internet resources can often yield valuable insight into implementation details of specific systems.

The obvious starting points include inspection of specific drivers that are used. Here, the operating system itself will often provide much information on the drivers and their system locations, as shown in Figure 1 below:

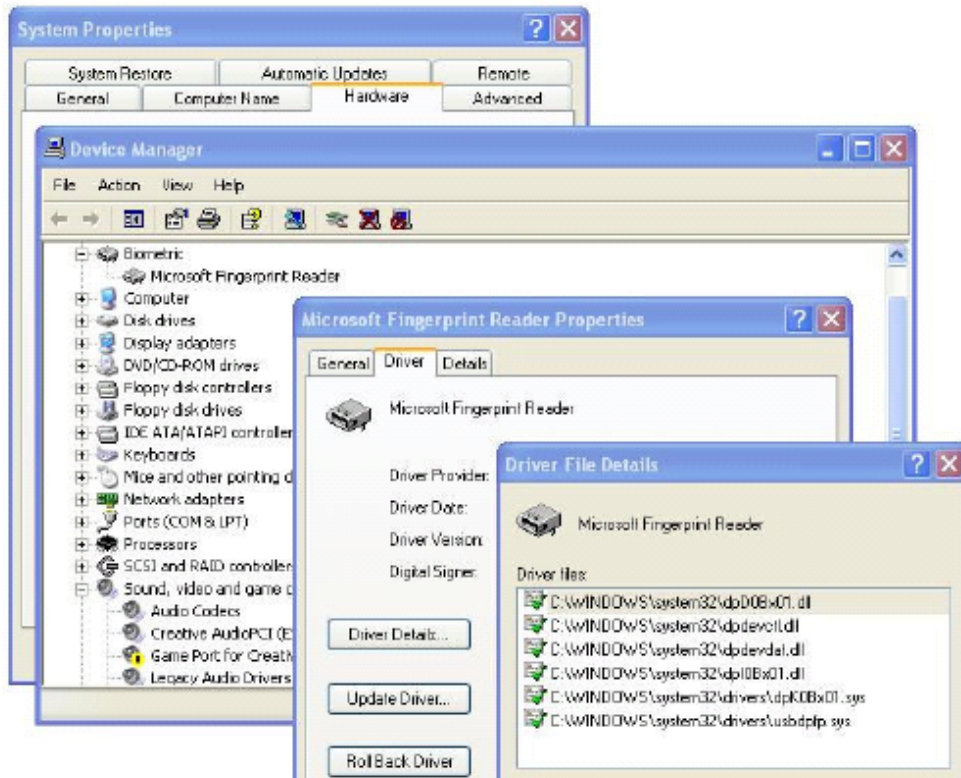


Figure 1. Identification of core driver modules of the Fingerprint Reader from System Manager

Here, we see the various DLLs and drivers that are used to control the device, which will serve as a good starting point to our High-Level understanding of the device and system operation.

The next step in this phase should typically include examination of the system’s interaction with the underlying operating system. Again, a number of tools exist for this purpose – the well-known *Sysinternals* tools [4] *regmon*, *filemon* and *process explorer* provide a great avenue for exploring process interaction with the registry, file system and other processes respectively. Findings from this step should be documented by the researcher as they will form the basis of later phases.

Once the relevant processes, DLLs and files have been identified, simply the names of these components can often yield clues as to their implemented functionality. In our worked example, the following table presents just some of the components identified during this phase, and their likely functionality (purely based on the filename):

System Component/File Name	Likely Functionality
DPhost.exe	Digital Persona Host – Main host application
Crypt32.dll and DPSecret.dll	Encryption/Decryption Functionality (Fingerprint images are purportedly encrypted between device and host)
Dpdevctl.dll	Digital Persona Digital Device – Control commands for the fingerprint device
Dpdevdat.dll	Digital Persona Feature Extraction – functions for handling data received from the device
DPCFtrEx.dll	Digital Persona Feature Extraction – functions for extracting biometric features from fingerprint images
DpCmpMgt.dll	Digital Persona Comparison/Component Management
DPCRecEn.dll	Digital Persona Recognition Engine – functionality relating to the biometric matching algorithm

Table 1: Identifying possible system functions from filenames alone

The minor information leakage in the filenames above might go some way in assisting the researcher to quickly identify the functionality of the system that he/she is more interested in researching further.

Having identified a core process or application within our system (such as DPhost.exe listed in Table 1 above), one final method noted here for identifying the components used by the application involves the use of OllyDbg's Executable Modules Window. By attaching the debugger to the interested process, this window lists all executable modules currently loaded by the debugged process. Figure 2 below demonstrates the modules used within our *Microsoft Fingerprint Reader* system. This window confirms the system modules identified in other stages mentioned above, while it also identifies those system modules that are loaded by our process, such as *kernel32.dll* and *advapi32.dll*. Identification of these modules will prove useful during the next phase of our investigation.

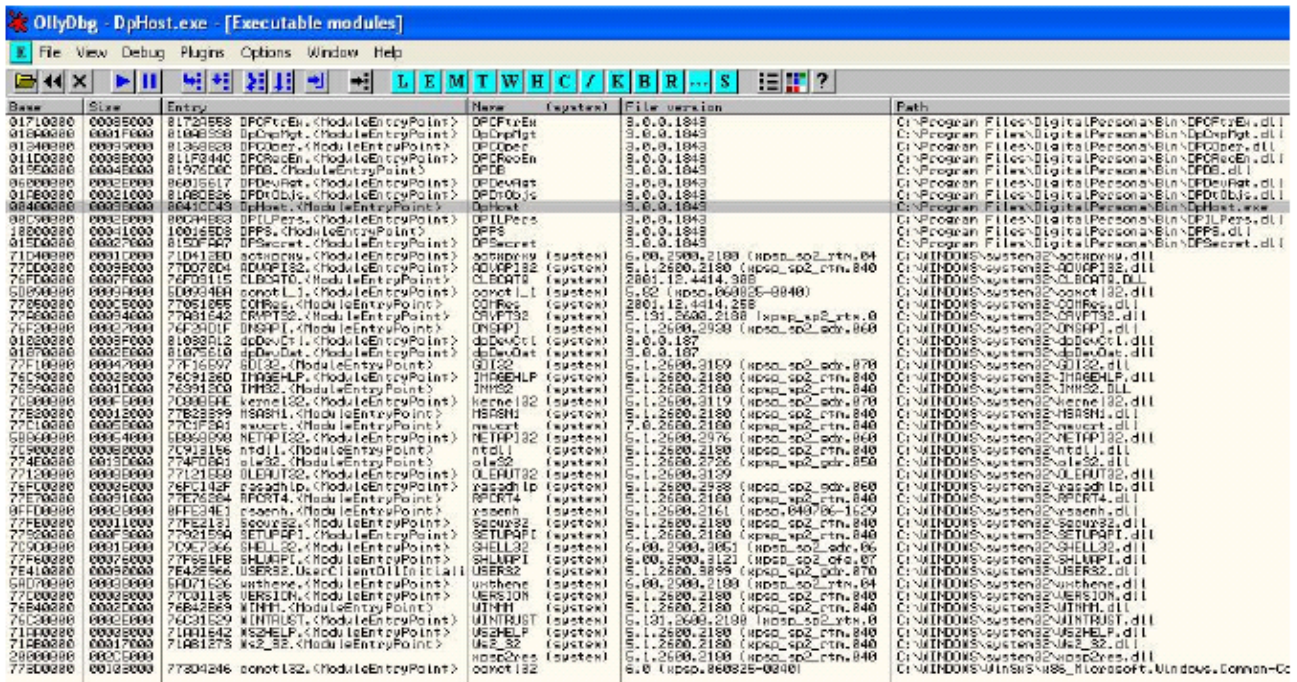


Figure 2. The OllyDbg Executable Modules window identifies modules loaded by our debugged process

Phase Two: Identifying Component Functions

To further our high-level reverse engineering, the next phase involves examination of the components identified during Phase One for those functions that are exported, or may appear interesting for further analysis. Again, a number of tools and techniques are available to us at this stage. Initially, we are interested in identifying name and exported functions; and various API calls. One such tool that can be used here is the DLL Export Viewer, which displays a list of all exported functions and their virtual memory addresses for a specified DLL file [5], as shown in Figure 3 below:

Function Name	Address	Relative Address	Ordinal	Filename
FD_CloseDevice	0x10009570	0x00009570	3 (0x3)	dpdevctl.dll
FD_CloseDeviceManager	0x10009020	0x00009020	4 (0x4)	dpdevctl.dll
FD_DlGetVersion	0x100014b0	0x000014b0	1 (0x1)	dpdevctl.dll
FD_Entry	0x10009810	0x00009810	5 (0x5)	dpdevctl.dll
FD_EnumerateDevice	0x100091a0	0x000091a0	6 (0x6)	dpdevctl.dll
FD_GetDataFormat	0x10009380	0x00009380	7 (0x7)	dpdevctl.dll
FD_GetDeviceInfo	0x100092b0	0x000092b0	8 (0x8)	dpdevctl.dll
FD_GetParameter	0x100095f0	0x000095f0	9 (0x9)	dpdevctl.dll
FD_OpenDevice	0x10009450	0x00009450	10 (0xa)	dpdevctl.dll
FD_OpenDeviceManager	0x10008d60	0x00008d60	11 (0xb)	dpdevctl.dll
FD_SetParameter	0x10009770	0x00009770	12 (0xc)	dpdevctl.dll
FD_TestDevice	0x100098f0	0x000098f0	2 (0x2)	dpdevctl.dll

Figure 3. DLL Export Viewer identifies the exported functions of a given DLL (dpdevctl.dll)

IDA Pro can also be used to identify this information, by examining the *Exports* window of an auto-analysed DLL. At this stage, we present an automated tool to assist in the identification of functions, and auto-generation of the required configuration and code handler files for use with *uhooker*. The following IDC script (*uhooker.idc*) identifies each function within a DLL/module, and generates a *uhooker*-compatible configuration file. A python stub code file is also generated, which includes a simple function handle for each identified function. If a function is not exported, the configuration file is appended with a hook on the function's address. That way, we can still hook on undocumented or non-exported functions, as identified within IDA. The function handlers simply print to standard output when an exported function is called, or the memory location of a non-exported function is accessed. Figure 4 below details the IDC script. We note that the script is extremely generic, and further functionality or automation is left as an exercise to the motivated reader.

```
// IDC auto-generation of uhooker configuration and handler files
// Matt Lewis - IRM plc 2007
#include <idc.idc>
static main() {
    auto ea,x,num_args,func_name,file,filename;
    auto hook_type,config_file,stub_file,config_line;
    auto stubl1, stubl2, stubl3, stubl4, stubl5;
    // these are stub code lines for our python handlers
    // the stubs simply print to stdout when the handler is called
    stubl1 = "def ";
    stubl2 = "\tmyproxy = hookcall.proxy\n";
    stubl3 = "\tprint \"\"";
    stubl4 = "\thookcall.sendack()\n";
    stubl5 = "\treturn\n\n";
    // we set this to the type of hooks that we want to invoke:
    // Type A: Hook after the function returns
    // Type B: Hook before entering the function
    // Type *: Hook when execution reaches this address
    hook_type = 'B';
    Message("\n\n----- Creating uhooker configuration and handler files... -----\n");
    file = GetInputFile();
    filename = substr(file,0,strstr(file, "."));
    // the uhooker configuration file is written to...
    config_file = fopen(filename + ".cfg", "w");
    // the uhooker python handler stub code is written to...
    stub_file = fopen(filename + ".py", "w");
    for ( ea=NextFunction(0); ea != BADADDR; ea=NextFunction(ea) ) {
        // get the function name
        func_name = GetFunctionName(ea);
        // get the number of function arguments
        // divide by four for 32-bit machine
        num_args=(GetFunctionAttr(ea,FUNCATTR_ARGSIZE)) / 4;
        Message("%s:%s:%d:%s.%s_handle:%c\n",file,func_name,num_args,filename,func_name,hook_type);
        x = GetFunctionFlags(ea);
        if(strstr(func_name,"sub_") == 0) {
            // this is not an exported function, therefore we hook on its address, rather than name
            config_line = "dummy.dll:0x" + ltoa(ea,16) + ":0:" + filename + "." + func_name +
                "_handle:" + "\n\n";
        } else {
            // this is an exported function, therefore we hook on its exported name
            config_line = file + ":" + func_name + ":" + ltoa(num_args,10) + ":" + filename + "." +
                func_name + "_handle:" + hook_type + "\n\n";
        }
        // write to config file
        writestr(config_file,"# Insert Comment Here...\n");
        writestr(config_file,config_line);
        // write to stub code file
        writestr(stub_file,stubl1 + func_name + "_handle(hookcall):\n" + stubl2 + stubl3 + func_name
            + "_handle called"\n" + stubl4 + stubl5);
    }
    fclose(config_file);
    fclose(stub_file);
}
}
```

Figure 4. IDC code for auto-generation of uhooker configuration and handler files

An excerpt of the auto-generated configuration file of the *Dpdevctl.dll* module can be seen in Figure 5 below. The difference can be seen between those functions that are exported, and those that are not. In its default state, the above script generates hooks to functions before they are entered or their start memory addresses are accessed. This can be changed in the script, or manually via the configuration file, should the researcher wish to hook after the function returns for example. A default comment placeholder is also added, should the researcher wish to pass comment on specific function calls.

```
# Insert Comment Here...
dummy.dll:0x10001000:0:dpdevctl.sub_10001000_handle:*
# Insert Comment Here...
dummy.dll:0x10001160:0:dpdevctl.sub_10001160_handle:*
# Insert Comment Here...
dummy.dll:0x10001270:0:dpdevctl.sub_10001270_handle:*
# Insert Comment Here...
dummy.dll:0x10001350:0:dpdevctl.sub_10001350_handle:*
# Insert Comment Here...
dummy.dll:0x100013F0:0:dpdevctl.sub_100013F0_handle:*
# Insert Comment Here...
dummy.dll:0x100014A0:0:dpdevctl.sub_100014A0_handle:*
# Insert Comment Here...
dpdevctl.dll:FD_DllGetVersion:1:dpdevctl.FD_DllGetVersion_handle:B
# Insert Comment Here...
dummy.dll:0x10001550:0:dpdevctl.sub_10001550_handle:*
...<remainder omitted for brevity>
```

Figure 5. Example of an auto-generated configuration file for use with *uhooker*

For each function calls and hook handles generated within the *uhooker* configuration file, the IDC script generates a corresponding python handler file. Figure 6 below shows an example of the auto-generated stub code. As explained earlier, initially, the stub code merely prints to standard output when a function call is made. This will allow us to quickly identify the system calls that a process or application makes.

```
def sub_10001000_handle(hookcall):
    myproxy = hookcall.proxy
    print "sub_10001000_handle called"
    hookcall.sendack()
    return
def FD_DllGetVersion_handle(hookcall):
    myproxy = hookcall.proxy
    print "FD_DllGetVersion_handle called"
    hookcall.sendack()
    return
```

Figure 6. Example of auto-generated stub code for each function

System Libraries and Functions

Until now, we have primarily been focused on identifying the functions that specifically relate to the device or system under investigation. We noted in Phase One that applications will often implement calls to operating system functions, and so it would be wise to include as much detail as possible about such calls within our *uhooker* configuration and handler files, in order to examine how the system truly interacts with its underlying operating system.

For this purpose, we can use our IDC script above on any system modules that we choose, such as *kernel32.dll* and *advapi32.dll*. The huge benefit of generating configuration files and handlers for the functions within these modules is reuse. This allows the researcher to quickly build reusable libraries to be used in future High-Level reverse engineering tasks. Another benefit in this area relates to the wealth of information within the public domain on specific libraries. Many of the system functions provided by Microsoft are well documented both by Microsoft themselves [6] and open-source forums [7]. Knowing the type of function arguments will prove extremely useful in the final phase of our research.

For example, Figure 7 below shows a configuration file for some of the registry and cryptographic functions exported by *advapi32.dll*. These are wise choices to include in our investigations as Windows applications will commonly interact with the system registry. In the example below, our comments have been extended with information gained from [6] and [7], whereby it has been possible to identify the function names, their arguments and types.

```
# LONG WINAPI RegCreateKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult);
# Creates the specified registry key. If the key already exists in the registry, the function
opens it.
advapi32.dll:RegCreateKeyA:3:advapi32.RegCreateKeyA_handle:B
# LONG WINAPI RegCreateKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD Reserved, LPTSTR lpClass, DWORD
dwOptions, REGSAM samDesired,
# LPSECURITY_ATTRIBUTES lpSecurityAttributes, PHKEY phkResult, LPDWORD lpdwDisposition);
# Creates the specified registry key. If the key already exists, the function opens it. Note that
key names are not case # sensitive.
advapi32.dll:RegCreateKeyExA:9:advapi32.RegCreateKeyExA_handle:B
# BOOL CryptGenKey(HCRYPTPROV hProv, ALG_ID Algid, DWORD dwFlags, HCRYPTKEY* phKey)
# Generates a random cryptographic session key or a pub/priv key pair.
advapi32.dll:CryptGenKey:4:advapi32.CryptGenKey_handle:B
# BOOL CryptGenRandom(HCRYPTPROV hProv, DWORD dwLen, BYTE* pbBuffer)
# Fills a buffer with cryptographically random bytes.
advapi32.dll:CryptGenRandom:4:advapi32.CryptGenRandom_handle:B
```

Figure 7. A sample of advapi32.dll functions to be used with *uhooker*

Bringing it all together

Once we have identified as much information as possible regarding the components and functions that our system under research executes, we can begin to bring these components together for the next phase of investigation. While our code handlers reside in distinct python modules, we note here that *uhooker* works with one configuration file only, meaning that before we progress, we must concatenate all of the required configuration files into one. Again, this brief manual process is yet another example of further automation that could be introduced within the IDC script presented above.

Phase Three: High-Level Functional Analysis

Now that we have defined our *uhooker* configuration file and python handlers, we are ready to begin examining system operation at the function call level. A major benefit with what we are doing is that our investigation is focused on a specified process. Once attached to our process of interest, *uhooker* will hook onto the addresses and function calls that relate to that process only, meaning that we will not need to perform any filtering on the results. We note that this could become problematic with truly multi-threaded applications.

Figure 8 below presents a screenshot of a minimal *uhooker* configuration file while attached to the *DPHost.exe* application. The screenshot shows some of the system calls made while attempting to enroll an addition finger within the system.

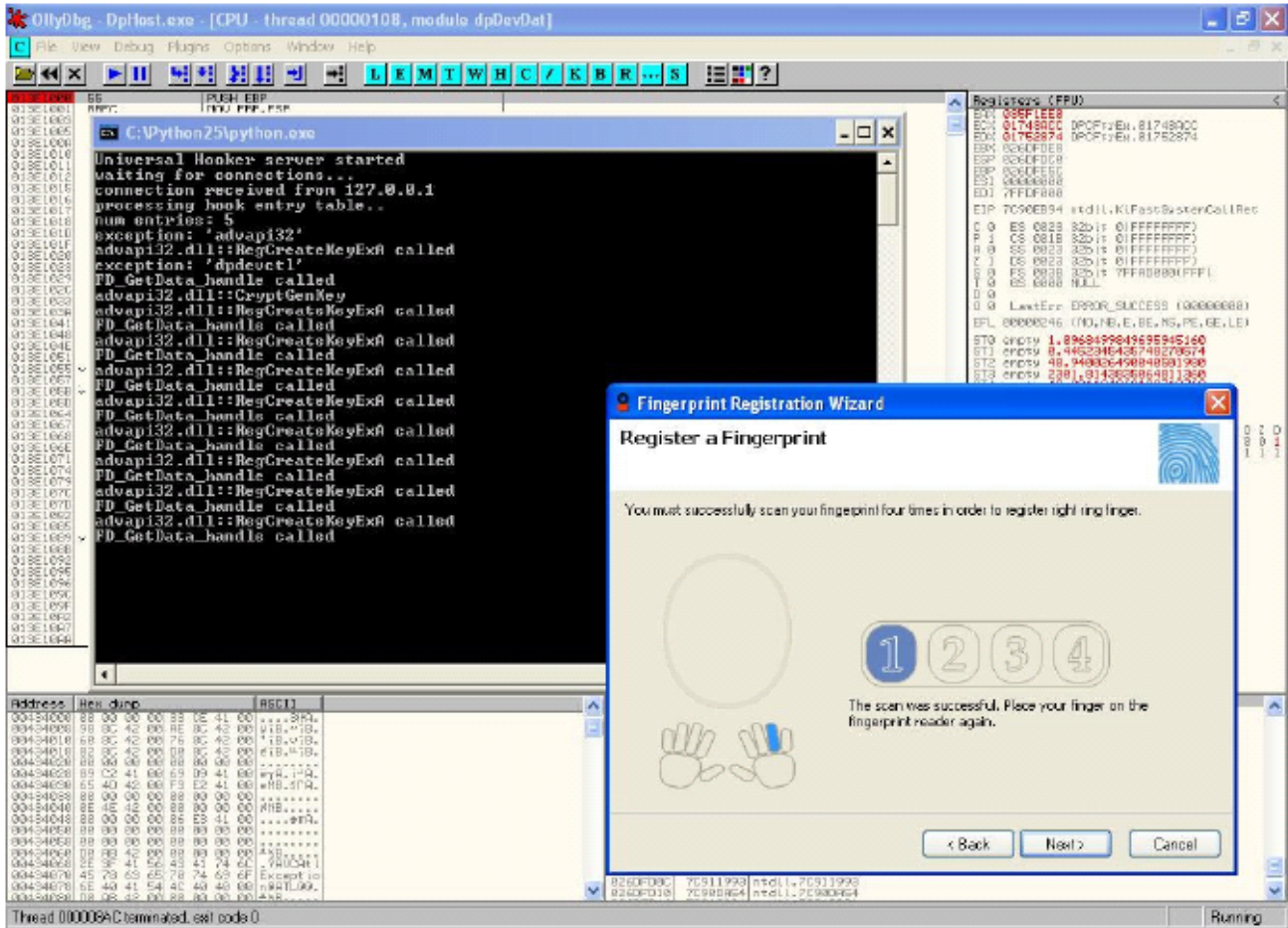


Figure 8. Example of uhooker examining function calls with the Microsoft Fingerprint Reader

We see that calls to *RegCreateKeyExA*, *CryptGenKey* and *FD_GetData* are made. While on the surface this may not seem like much, this information is extremely useful in identifying the sequence of function calls. The benefit of this High-Level approach, as opposed to diving into low-level reversing, is that the application or process being researched tells us how it operates. We are able to quickly identify which functions are called, and when.

With the method above, we are able to freely use the system and it to tell us which function calls it is making. It is during this “learning” phase that the researcher is likely to become *au fait* with the system, and is able to identify those functions that may be interesting for further investigation (e.g. functions that process user-supplied input).

Where Next?

Uhooker implements an API for use within hook handlers. While until now we have merely been printing when our functions are called, we can take our research further by implementing more specific functionality within our handlers. Again, the reader is referred to [1] for more information on the API – fundamentally, we can use the API to read/write memory and CPU registers, convert memory bytes to ASCII strings, create and free memory.

The next step is to identify the types of the parameters passed to the functions. For some of the system functions, we can ascertain this from a number of sources. For example, the following handler code for *RegCreateKeyExA* would allow us to identify the actual registry keys that are created/opened during execution.

```
def RegCreateKeyExA_handle(hookcall):
    myproxy = hookcall.proxy
    print "advapi32.dll::RegCreateKeyExA called"
    keyname = hookcall.params[1]
    print "Attempting to create/open: " + myproxy.readasciiz(keyname)
    hookcall.sendack()
    return
```

Figure 9. Reading String arguments within handlers

With the CryptGenKey handler, we might be able to identify the algorithm ID, the key type and key:

```
def CryptGenKey_handle(hookcall):
    myproxy = hookcall.proxy
    print "advapi32.dll::CryptGenKey"
    algorithmID = hookcall.params[1]
    keyType = hookcall.params[2]
    keyAddress = hookcall.params[3]
    print "Algorithm ID: " + str(algorithmID)
    print "Key Type: " + str(KeyType)
    myproxy.readmemory(keyAddress, 32)
    hookcall.sendack()
    return
```

Figure 10. Reading memory addresses within handlers

While the two examples above show only reading of memory, the implications of the ability to write/set memory and registers in this way should be obvious. The researcher is able to use these techniques to test boundaries and program control flow in ways that might be difficult via other means. Again, we note the possibilities of writing fuzzers around specific handlers for example, which provides for a more focused and direct approach than blind fuzzing or bounds checking. From this point, the potential possibilities are only limited by the imagination of the researcher with respect to further investigations into the system being assessed.

Conclusions

In concluding this whitepaper, we re-iterate our initial point that this paper does not present any new tools, but is a culmination of existing tools and techniques that are already available within the IT security research community. We have merely scratched the surface of the possibilities within this domain, and it is envisaged that an entire suite of security auditing tools could be developed around the way of working presented in this document.

Function hooking can provide an extremely quick method of identifying system implementation and operation, while it also provides direct access to the application's memory and CPU space in ways that can be both read and written. As a matter of responsibility, we note here the intentions of this whitepaper as a document of methods for vulnerability research only. The aim of such research should be conducted in order to identify vulnerabilities so that subsequent patching and fixing can be invoked in order to mitigate any problems discovered.

References

- [1] The Universal Hooker (uhooker) – <http://oss.coresecurity.com/uhooker/doc/index.html>
- [2] The Interactive Disassembler (IDA) – <http://www.datarescue.com/>
- [3] The OllyDbg Debugger – <http://www.ollydbg.de/>
- [4] Sysinternals Tools – <http://www.microsoft.com/technet/sysinternals/default.mspx>
- [5] DLL Export Viewer – http://www.nirsoft.net/utils/dll_export_viewer.html
- [6] Microsoft Technet – <http://technet.microsoft.com>
- [7] Open Source Microsoft Windows API - <http://source.winehq.org/WineAPI/advapi32.html>

About the Author

Matthew Lewis is a Security Consultant at Information Risk Management Plc (IRM) where he performs a range of consultancy services including providing advice to clients on the use of biometrics. Prior to working at IRM, Matthew spent three years at CESG (the UK Government's Information Assurance arm) researching the security capabilities of biometric systems and advising Government about their use. Matthew has presented at many international conferences on the subject of biometrics and co-administered the UK Biometrics Working Group.

About IRM

Information Risk Management Plc (IRM) is a vendor independent information risk consultancy, founded in 1998. IRM has become a leader in client side risk assessment, technical level auditing and in the research and development of security vulnerabilities and tools. IRM is headquartered in London with Technical Centres in Europe and Asia as well as Regional Offices in the Far East and North America. Please visit our website at www.irmplc.com for further information.