Fachhochschule Aachen
Campus Jülich

Faculty: Medical Engineering and Applied Mathematics
Degree program: Scientific Programming

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Reverse Engineering

An exemplary approach to the fundamentals of reverse engineering

Bachelorthesis

in partial fulfillment of the requirements for the Bachelor of Science

by

Volker Mauel
Student no. 855252

Jülich in August 2014

Eidesstattliche Erklärung

Hiermit versiche ich, dass ich die Bachelorarbeit mit dem Thema

Reverse Engineering

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name:_____

Jülich, den _____

Unterschrift der Studentin / des Studenten

Diese Arbeit wurde betreut von:
1. Prüfer Prof. Ulrich Stegelmann
2. Prüfer Josef Heinen

Abstract

Reverse engineering describes the process of taking an existing product and disassembling it in order to understand how parts of it and the product as a whole works. In software development, it is a common technique to understand certain parts of software and to implement similar features in a new project. Additionally, it is often used in malware analysis. Reverse engineers can figure out how malware spreads and what it does on the target system. This does not necessarily mean that reverse engineering itself can only be applied to causing and preventing damage to a system, but also to extend an existing software system.

For instance one could imagine a measurement device for radioactivity with software on a computer that logs the results. The software used to work correctly with an earlier Windows version (e.g. Windows XP), but does not work with a newer Windows version (e.g. Windows 7). Additionally, the company that created the measurement device does not offer support for it any more. This is where reverse engineering helps. It enables programmers to understand how the software communicates with the device and allows them to implement its functionality in a new program. Furthermore, reverse engineering gives developers the opportunity to extend existing software to match the needs of the users. If the aforementioned measurement device software writes a text file at the end of the measurement and then simply closes, reverse engineers could identify which instructions create the file. They can then inject a Dynamic-link library (Dll), detour the existing application flow, and, in addition to writing the file to the hard drive, make the application send an e-mail containing the results to the user.

This thesis focuses on giving a general overview of the possibilities as well as showing how to apply the mentioned techniques to find out more information on a program. Therefore an example application has been created, explained in detail in chapter 4, which simulates the measurement device software. It connects via sockets to a server, which simulates the measurement device itself, that sends the acquired data to the application. It is added to a list and, at the end of the measurement, the whole list is saved into a text-file.

Additionally the legal situation for reverse engineering of software in the United States and the European Union is clarified in chapter two.

Contents

List of Figures

List of abbreviations

API	Application Programming Interface
CLR	Common Language Runtime
CPU	Central Processing Unit
Dll	Dynamic-link library
DMCA	Digital Millennium Copyright Act
EULA	End-user license agreement
FASM	Flat Assembler
IDA	Interactive Disassembler
IEEE	Institute of Electrical and Electronics Engineers
JIT	just-in-time
MSDN	Microsoft Developer Network
TLS	thread-local storage

Chapter 1

Introduction

Reverse engineering is a useful technique in software development to understand the internals of a program. This thesis explains the fundamentals of reverse engineering computer software.

Chapter two discusses the definition of Reverse Engineering provided by the Institute of Electrical and Electronics Engineers (IEEE). Different methods of reverse engineering are explained. The legal landscape of reverse engineering in the United States and the European Union are also discussed.

Since reverse engineering to a great extent consists of reading and understanding assembly, the third chapter focuses on explaining the quintessences of the assembly language and calling conventions. Additionally, the tool Interactive Disassembler (IDA), which has been used as part of this thesis, is described.

The fourth chapter overviews an example application which has been written as part of this thesis. This application simulates the connection to a measuring device, receives data from it and writes the output into a text-file.

The fifth chapter shows how to reverse engineer an unknown application on the basis of the aforementioned example application. The knowledge of the internal processes of the application is disregarded. The reader will understand how to find out more information on the internal program flow and how to structure the disassembled output in the tool IDA to discover the relevant parts of the software.

The sixth chapter focuses on using the knowledge from chapter five to acquire the data from the measuring device in a new application by reading from the original one.

The seventh chapter is a conclusion on whether reverse engineering is a viable option in software development.

Chapter 2

Reverse Engineering - Theory

This chapter explains the required theory to understand reverse engineering. It illustrates the definition of reverse engineering and the different types. Additionally, legal aspects are clarified for the European union and the United States and it explains how to use the acquired information of the application.

2.1 Definition

The IEEE describes reverse engineering as follows: "reverse engineering means using engineering techniques to discover the underlying ideas and principles governing how a machine, computer program, or other technological device works".[IEE11] Figure 2.1 shows the process of reverse engineering in the context of classic forward engineering, as a form of abstraction that begins at the implementation of an existing system and ends at the conceptual layer. The section in the middle of the figure shows that with reverse engineering it is possible to alter all layers of the software and finally, by using forward engineering, create a new software which contains these alterations.

2.2 Types of Reverse Engineering

Reverse engineering cannot only be applied to programs as a whole, but also to different parts and aspects of it. These can be categorized into **File Structure**, **Protocols** and **Functions**. Usually the process begins with reverse engineering the functions of a program and meanwhile learning about file structures and protocols, but one could imagine using network analyzing software, like **Wireshark**, to understand the protocol. To examine the structure of an unknown file, the tool **Binwalk** can be used.

2.2.1 File Structure

Programs often use their own file structures to save the data. Proprietary software often uses proprietary file structures that are not documented to obligate the users and make switching to another program inconvenient or even impossible. An example of this is Microsoft Office, whose file structures remained undocumented for a long time.

Reverse engineering can help to identify the file structure and to write a program that converts files from the proprietary format into a documented structure.

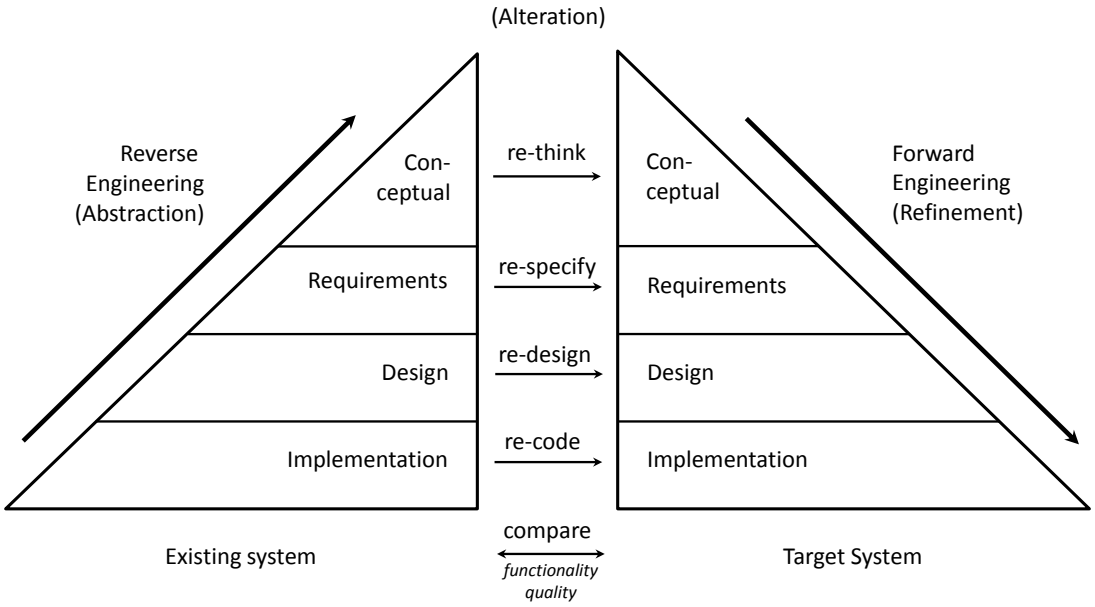


Figure 2.1: Reverse engineering and traditional software development[Pro]

2.2.2 Protocols

Applications that use other devices or services from a server have to implement a specific protocol to ensure that the request is correct and that the response can be used. These protocols are rarely documented.

Reverse engineering enables developers to find out the relevant aspects of the communication protocol and implement these in a new program which can be extended easily.

2.2.3 Function

Software sometimes contains functionality that a programmer wants to adopt in their own application. An example could be image editing software which contains image filters which the developer wants to have in their own application.

Reverse engineering can help to find out the way the original application implemented the function and gives the programmer the opportunity to implement the feature in their application. The respective law or End-user license agreement (EULA) has be to observed. (see ch. 2.3)

2.3 Legal Aspects

Since reverse engineering is such a controversial topic and potentially allows plagiarism, the law has to regulate under which circumstances it is legal to use reverse engineering techniques.

2.3.1 European Union

In the European Union it is permitted to reverse engineer software for the purpose of interoperability, but not to create a competing product. Additionally it is prohibited to release the information obtained through reverse engineering.[Mus98]

2.3.2 United States

The Digital Millennium Copyright Act (DMCA) states that reverse engineering is legal for programs if it is for the purpose of interoperability.[Usc] Contrary to the law in the European Union, the EULA overrides the copyright law for this aspect.[1] Therefore most EULAs prohibit reverse engineering explicitly.

2.4 Tools

Reverse engineering without supporting tools is possible but very time-consuming. The most popular software for reverse engineering is IDA. It offers a huge list of features and aims at professional users. In this thesis IDA Pro in version 6.1 was used, but it should be mentioned that there are free tools like **OllyDbg**[Oll] which can be used instead of IDA.

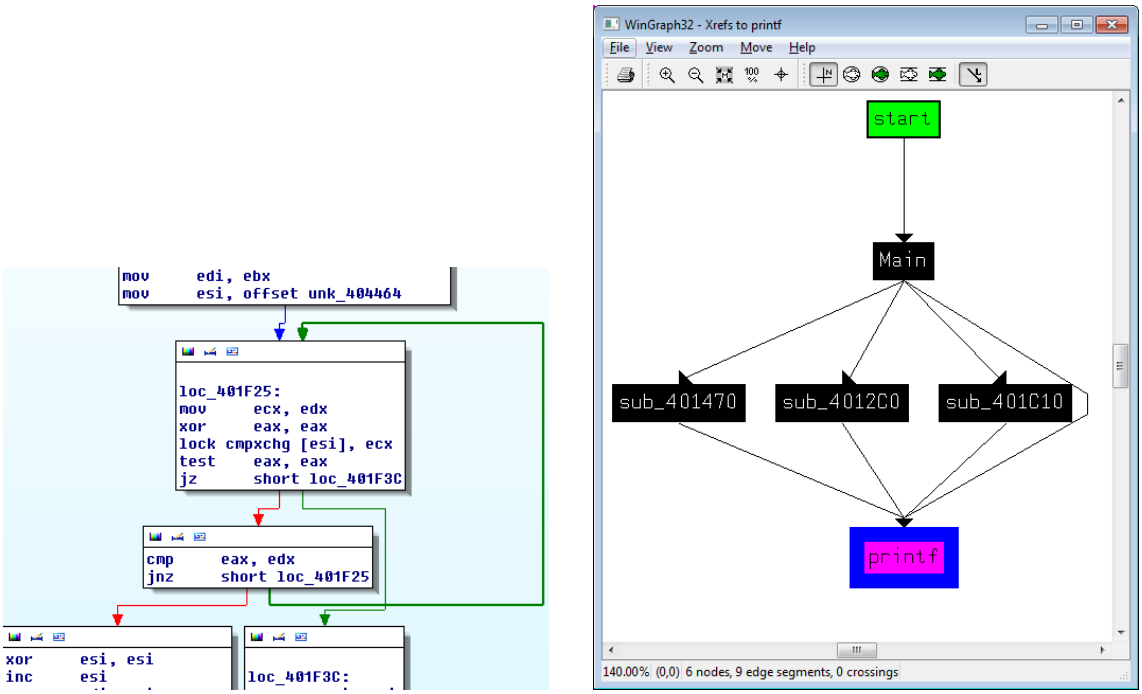
2.4.1 IDA - The Interactive Disassembler

The Interactive Disassembler (IDA) is a debugger and disassembler that is often used by reverse engineers to analyze programs. It features a static code analyzer that automatically parses the functions inside the executable and names them according to their position (*sub_400000* e.g.). Additionally IDA creates a flow chart for functions to help the developer understand which code paths are used in which case. This can be seen in figure 2.2(a). IDA can use several different debuggers on the local machine as well as attach to processes on remote systems. This allows reverse engineers to run potentially malicious code in virtual machines to analyze it from the host computer. Furthermore IDA analyzes the exported and imported functions and automatically creates so-called XRefs[2] between them. This allows the user to search for a specific import like *printf* and

[1] precedent Bowers v. Baystate Technologies http://www.infoworld.com/d/developer-world/contract-case-could-hurt-reverse-engineering-337

[2] XRefs are cross-references between different parts of the application e.g. a string and a function can be cross-referenced, so the developer knows that the string is used in this specific function and in which other functions it is used.

automatically list all occurrences of that method call in a window. Figure 2.2(b) shows the references to *printf* in an application.



(a) An example of a flowchart in ID

(b) An example of an XRef-graph in IDA

Figure 2.2: Flowcharts and XRef-graphs in IDA

.rdata:00403368	00000014	C	Measuring %d values
.rdata:0040337C	00000005	C	1234
.rdata:00403384	0000000A	C	127.0.0.1
.rdata:00403390	0000002C	C	Successfully connected!\r\nFetching Values!\r\n
.rdata:004033BC	00000011	C	error %d occured
.rdata:004033D0	0000000C	C	measure %d\n
.rdata:004033E0	0000000D	C	./output.txt
.rdata:004033F0	00000015	C	Failed to save file!
.rdata:00403408	00000011	C	list<T> too long
.rdata:0040341C	0000000D	C	%d\t\t%f\t\t%f\r\n
.rdata:0040342C	0000000C	C	%d\t\t%f\t\t%f\n

Figure 2.3: An example of the strings in IDA

Another important point is that IDA automatically parses all strings in the executable and lists them in a seperate subview shown in figure 2.3. This allows quick navigation through the code by looking for a string that is known to be used at a specific point in the application.

2.5 General Approach

The general approach without any knowledge of the target program is opening it in a disassembler and looking at the disassembled output. The entry point of an application is the most interesting part since it is the root of all function calls. In Microsoft Windows based applications it is either named *start* for console applications or *WinMain* for applications with a graphical user interface. After initialization code for variables and exception handling and run-time environments, the originally named *main*-function can be found.

Renaming different parts of the disassembled output makes it easier to get an overview of the functions and variables. All dissemblers support renaming of functions and variables.

For further analysis, the used datastructures should be mapped.[3] Once these are mapped and named, the variables using these structures should have their type changed to the type of the structure, to see which members are accessed.

2.6 Using the Acquired Information

Depending on the type of reverse engineering employed (see 2.2), the gathered information from an application can be used in various ways. This part of the thesis focuses on using the results of analyzing the application as a whole and getting information out of it while it is being executed by using ReadProcessMemory.

2.6.1 Memory Reading/Writing

The Windows Application Programming Interface (API) offers methods to read and write the memory of other processes. This allows the reverse engineer to read from the previously acquired memory locations and copy the data into another program. The required functions are *OpenProcess*, *ReadProcessMemory* and *WriteProcessMemory*. First, the process has to be opened by using *OpenProcess(int Access, bool InheritHandle, int ProcessId)*. The process-id can be obtained by using one of the different process enumeration options and filtering the result for the correct process name. The result of OpenProcess is a handle that can be passed to *ReadProcessMemory(int handle, int baseaddress, void* buffer, int size, int* numberOfBytesRead)*.

ReadProcessMemory(int handle, int baseaddress, void buffer, int size, int* numberOfBytesRead)* accepts this *handle* as first argument. The *baseAddress* is the memory location to read from, *buffer* is where the data from the other application will be copied to, *size* is the number of bytes to read and *numberOfBytesRead* is the number of bytes that have been read from the target process.

The following code shows how to use *OpenProcess* and *ReadProcessMemory*.

```
1  IntPtr handle = OpenProcess(0x10,false,123); //open process with id 123
2  byte[] buffer = new byte[4]; //create a buffer
```

[3] When working with C++ classes, the reverse engineer has to make sure that they have enough room for the data including potential virtual-method-table pointers.

```
3  int readBytes=0;
4  var result = ReadProcessMemory(handle,0xCBADE,buffer,4,ref readBytes); //
       reads 4 bytes from address CBADE and copies them into the buffer.
5  //buffer now contains the 4 bytes and can be used like any byte-array
```

WriteProcessMemory works in a similar way.

2.6.2 Hooking

Hooking describes the act of detouring the applications control flow into a user-written code-section, that will be executed before or instead of the original method. In assembly this would appear as the following:

```
1 method:
2 add eax,5 ;eax+=5
3 mov ebx,10 ;ebx=10
4 imul eax,ebx ;eax*=10
5 ret ; summary: eax=(eax+5)*10
```

A hooked function would look like this:

```
1 method:
2 jmp hook ;the hook-code is executed before the original code
3 mov ebx,10
4 imul eax,ebx
5 ret
6 hook:
7 add eax,5 ;restore the statement ''jmp hook'' overwrote
8 ;do something else e.g. modify eax or print its value to the console
9 jmp 3
```

2.6.3 Injection

Injection describes the process of injecting new code into an already running application, which is normally not part of it. Since it is running in the context of the target application, it can access the memory by normal dereferencing and referencing operations without the need for a WinAPI call.

2.6.3.1 Code-Injection

Code-Injection uses the Windows API function *WriteProcessMemory* (see 2.6.1) to write code into the target process' memory. This action requires previously allocated memory in the target process that has to be acquired by using the function *VirtualAllocEx*. Only writing into the target process does not change any of its functionality. Either a new thread has to be started by using the function *CreateRemoteThread* or an existing thread has to be detoured. Most of the time the main thread of the target application is used, as it also removes the need for synchronization and allows access to the thread-local storage (TLS)[4]. The injected code often only consists of some basic assembler instructions. Since these instructions have to be translated by libraries such as **AsmJit**

[4]The TLS is a memory region only accessible by that specific thread. To access its information, the thread has to be detoured.

and **FasmManaged**[5] before being injected into the target process. This makes the resulting code more maintainable and easier to read compared to raw byte-sequences.

2.6.3.2 Dll-Injection

Dll-Injection works by loading a user-written library into the target process' address space and using *CreateRemoteThread* to start a method inside this library. It is important to consider that *DllMain* must not start threads or load libraries itself, since this could cause a deadlock. [Msda]

To perform a Dll-Injection the function *OpenProcess* is used to get a handle to the process. Next, with *VirtualAllocEx* memory for the name and path of the Dll is allocated. With *WriteProcessMemory*, the path is then written into the target process. By calling *LoadLibrary* via *CreateRemoteThread* in the target process, the Dll is loaded and *DllMain* is executed. By using *WaitForSingleObject*, the injecting process will wait until the Dll loaded successfully. Then, *GetExitCodeThread* can be used to get the return value of *LoadLibrary* which is the base address of the loaded Dll in the target process. To avoid memory-leaks the memory for the filename has to be freed.

Now, the Dll is inside the target process and the only part left to do is to call an exported function inside that Dll with *CreateRemoteThread*. To find that function in the target process, the Dll is loaded into the local process and *GetProcAddress* is called to find the offset from the modules base-address to the function. Then the local version of the Dll can be freed and in the target process a thread can be started by using *CreateRemoteThread* with the address of the module base (LoadLibrary return value) plus that offset.[Dll]

It is also possible to make a process load a modified library of another library by copying it into the Dynamic-Link Library Search-path and renaming it to match the original file name. The Dll is required to have the same entry-points or loading it will fail. This removes the need to inject a Dll during run-time.

[5] AsmJit and FasmManaged both support just-in-time (JIT) translation of assembler code (e.g. mov eax,ebx) into the corresponding byte-sequences.

Chapter 3

Background

This chapter focuses on giving an overview of the basics needed to understand the process of reverse engineering. It contains an introduction to the Assembly-Language and an overview of the different calling-conventions.

3.1 The Environment

Reverse engineering is a highly compiler-dependent and platform specific topic. Since Microsoft Windows is the most popular operating system and supports binary compatibility between different versions, most software is written for it. Therefore this thesis describes reverse engineering of Windows-based software. The computer used for developing the software and reverse engineering it as part of this thesis uses Microsoft Windows 7 Professional x64.

3.2 Assembly-language

All computer programs consist of code that is executed by the Central Processing Unit (CPU), but it can only execute certain commands which are predefined in the so-called **Instruction Set**. As this thesis targets a Windows based PC-application, the only considered instruction set is x86.[1]

Upon loading a program, the instruction pointer (EIP) is initialized and set to the applications entry point.[2] Then the first instruction is loaded, executed and the instruction pointer is modified to match the executed instruction.

The assembly-syntax used in the examples is Flat Assembler (FASM), which is easier to understand than the Intel-syntax used in the IDA code snippets. For basic instructions and operations they are identical, but for more complex situations FASM is more readable and easier to maintain. Additionally, the library **FasmManaged** (see Chapter 2.6.3.1), which supports JIT translation of instructions to byte code, uses the FASM syntax.

[1] This does not include the x86_64 instruction set, which is mostly an extension of the x86 instruction set

[2] Usually this will be the *start* or the *WinMain* function of an application

3.2.1 Operations

x86 assembly supports a number of operations like adding, subtracting, multiplying, dividing, bit-wise operations like AND, OR, XOR and comparisons. Furthermore memory manipulations are possible to load and store data.

Examples:

```
mov eax,[ebx] ;reads the value at address EBX and move it into EAX
```

```
add eax, 10 ;adds 10 to the value in EAX
```

```
sub ebx, 15 ;subtracts 15 from EBX
```

```
mov eax, 15 ;moves 15 in the EAX register
mov edx, 16 ;moves 16 in the EDX register
mul edx ;multiplies EAX*EDX and stores the result in edx:eax.
```

3.2.2 The Stack

The stack of a program contains arguments for function calls and return addresses. The EBP register always points to the base of the stack (or stackframe) and the ESP register points to the top of the stack. Values and arguments can be stored on the stack by using the *push* operation and retrieved from the stack by using the *pop* operation. Alternatively stack-variables can be accessed by adding or subtracting the matching offset from the base pointer EBP.

Example:

```
push 5 ;pushes 5 on the stack
push 6 ;pushes 6 on the stack
push 7 ;pushes 7 on the stack
mov eax, [ebp] ;gets 5 from the stack, but does not remove it
pop ebx ;ebx now contains the value 7
pop esi ;esi contains the value 6
pop edi ;edi now contains the value 5
```

The corresponding stack during run-time looks like this:

0xF8	7	⇐ ESP
0xFC	6	
0x100	5	⇐ EBP

3.2.3 Stackframe

The stack is a data structure that grows upwards. It might begin at 0x100 and the next value would be inserted at 0xFC. A stackframe is a boundary created by a function to preserve the state of the stack and to allow the usage of local variables without unintentionally modifying the wrong arguments. It is created by saving the previous base-pointer on the stack and setting the base-pointer to the value of the stack-pointer.[Eag11, p. 91]

The typical sequence to achieve this is as follows:

```
0x12344 call method ;call the method
0x12345 jmp AnotherSection ;jump to a different section

0x12346 push ebp ;saves previous basepointer
0x12347 mov  ebp, esp ;sets ebp to esp to create the new stackframe
```

The resulting stack looks like this:

0x100 | 0x12345 | ⇐ EBP,return address

Example:

```
start:
push 0x2 ;pushes the argument onto the stack
call method ;calls the method

method:
push ebp
mov ebp,esp
mov eax, [ebp+8] ;loads the argument into eax
mov ebx,10 ;loads 10 into ebx
imul eax,ebx ;multiplies ebx*eax and store the result in eax
mov esp,ebp ;removes the current stackframe
pop ebp ;restores the old stackframe
ret 4 ;removes the argument from the stack
```

It is important to remember that the return address is pushed onto the stack after the last argument, so the correct offset to read that argument is 8.

In this example an empty stackframe was created. To actually fill the stack with data like local variables, the size of the stack frame has to be increased. This is achieved by subtracting from the ESP register.

Example:

```
method:
push ebp
mov ebp,esp
sub esp,0xc ;gets space for 3 local variables on the stack
mov [ebp-4], 123 ;saves the variable in the first free slot on the stack
mov [ebp-8], 234
mov [ebp-12], 345
;DoSomething
mov esp,ebp
pop ebp
```

This is what the stack looks like before initializing it with any data:

0x100 | 0x12345 | return address ⇐ EBP,return address

And this is what it looks like after increasing the size of the stack frame and adding the three local variables to it:

0xF4	345	⇐ ESP
0xF8	234	
0xFC	123	
0x100	0x12345	⇐ EBP,return address

3.2.4 Flow Control

The instruction pointer is modified to allow flow control. This does not happen directly, but indirectly by using operations like *jmp* or *call*. This allows to structure the code into different subroutines. These can be executed by using the *call*-Instruction. The main difference between *jmp* and *call* is, that *jmp* only modifies the instruction pointer to set the next instruction which will be executed its operand. *Call*, on the other hand, in addition to modifying the instruction pointer also pushes the address of the next instruction onto the stack. This allows the execution to continue there after the *Call*. Example:

```
start:
0x00 mov eax, SomeMethod
0x01 push 5
0x02 call eax
0x03 push 4
```

At first the instruction pointer is set to 0x0. The subroutine-pointer is moved into the EAX register. Then the instruction pointer is increased to 0x01 so the *push 5* will be executed next. Then the argument (5) is pushed onto the stack and the instruction pointer is increased again. Next the subroutine call follows, so the instruction pointer is set to the beginning of *SomeMethod* so it will be executed next. Additionally, the address of the instruction that comes after the call will be pushed onto the stack so the program knows where it has to continue after executing the subroutine. In this case, the value 0x03 will be pushed on the stack.

Furthermore, assembly offers control statements to compare values. As opposed to high-level programming languages, it return the result of the comparison a special register (ZF - zero flag). This register can be checked by conditional jump statements such as *jnz* (jump not zero), *jlt* (jump less than) and *jge* (jump greater equals). If the zero flag is set, it will stay set until the next comparison changes it.

3.2.5 Return Values

By convention, the return value of a function is stored in the EAX register. For data too large to fit into a single register, it is required to return a pointer to the result.

3.3 Calling Conventions

Calling conventions define in which order arguments are pushed onto the stack upon a subroutine call and who (the caller or the callee) has to clean the stack from the pushed arguments. The most commonly used calling conventions are cdecl, stdcall, fastcall and thiscall.

3.3.1 Cdecl

When using the cdecl calling convention, the caller has to clean the stack after the subroutine call. Arguments are pushed from right to left onto the stack. Furthermore it is the only calling convention that supports variadic functions such as *printf* since only the caller knows how many arguments have been pushed onto the stack and have to be removed again.[Fri14][Eag11, p. 85]

Example:

```
push arg1
push arg2
push arg3
call function
add  esp,12 ;cleans the stack
```

3.3.2 Stdcall

The stdcall calling convention assumes that the callee clean the stack before returning. Besides that, it behaves like to cdecl.

Example:

```
push arg1
push arg2
push arg3
call function
; no stack cleanup - callee does this
```

3.3.3 Fastcall

The fastcall calling convention has not been standardized and is very compiler-dependent. The Microsoft Visual C(++) Compiler (MSVC) passes the first 2 DWORD or smaller arguments from left to right in the ECX and EDX register. The other arguments are passed from right to left on the stack. The callee is responsible for cleaning the stack.[Msdc]

Example:

```
mov ecx, arg1
mov edx, arg2
push arg3
call function
```

3.3.4 Thiscall

The thiscall calling conventions is used for calling subroutines on objects. The this-pointer is passed in the ECX-register, the other arguments are passed via the stack from right to left. The callee cleans the stack.

```
mov ecx,object
push arg1
push arg2
call function
; no stack cleanup - callee does this
```

Chapter 4

The Example Application

As mentioned in Chapter 2.3 reverse engineering can be prohibited by the EULA and is only allowed for reasons of interoperability. In order to not violate any copyright laws, an example application has been developed as part of this thesis to demonstrate the process of reverse engineering.

This chapter briefly describes the internal structure of that application.

4.1 The Application

The application simulates the software of a measurement device (e.g. to measure radiation). It connects to the device and receives measurement values every second. These values are printed to the console and also, at the end of the execution, saved into a text-file. (as detailed in figure 4.1)

4.2 Starting the Application

The software is started by passing a command-line argument with it that contains the amount of values that should be measured. If no argument is passed, the default amount is 100. The application then connects to the server-application (that simulates the measuring device itself) and receives data from it which consists of:

- A Timestamp
- Value1
- Value2

4.3 Network Protocol

The clients first message to the server has to be "measure X" where X is the amount of values that should be measured. The data is not encoded and is transferred as little-endian byte-sequences.

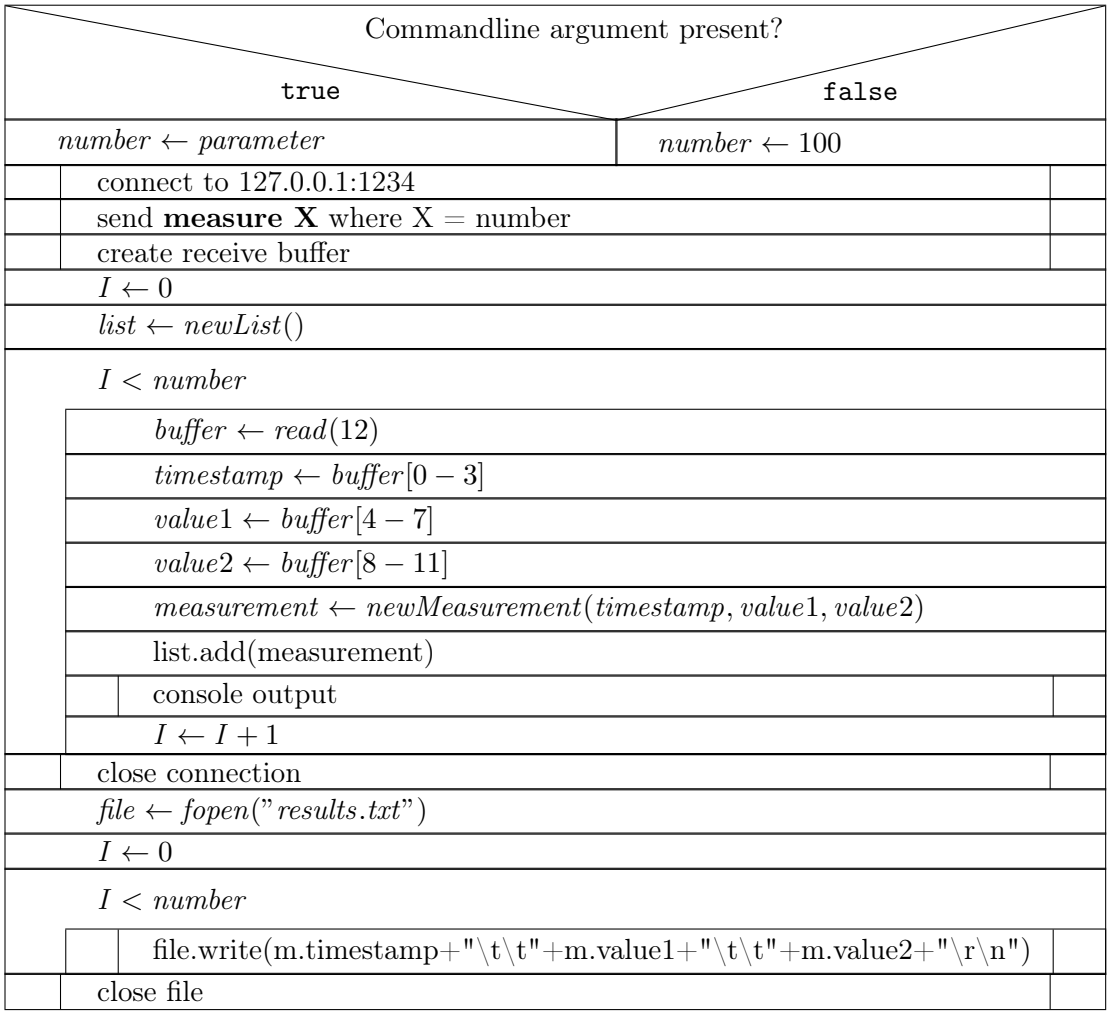
After the desired amount of values has been transmitted, the server closes the connection.

4.4 File Structure

The data received from the server application is written into a text file called **results.txt**. The delimiter symbol is two tabulators. After the time-stamp and one set of values, there is a newline. The values are written as ASCII-encoded text by using the *fprintf* function.

4.5 Diagrams

The following structogram helps understanding the main-function.



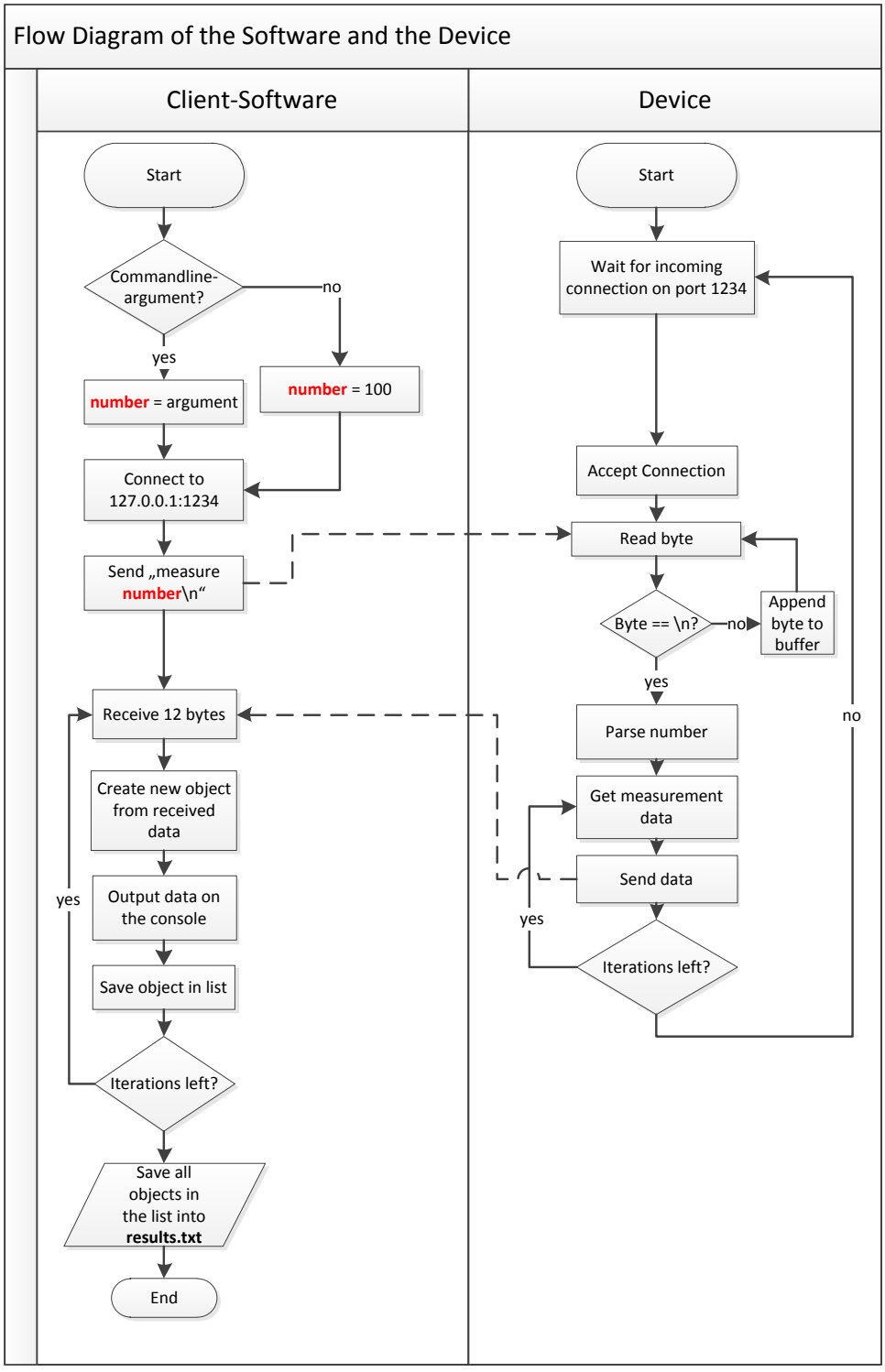


Figure 4.1: Flowdiagram of the Application and the Device

Chapter 5

The Process of Reverse Engineering

This chapter explains the necessary steps to find out the important information of a program. The example application has been compiled with Microsoft Visual Studio 2013 and the pdb-file containing the debug symbols has been removed to simulate the work on an unknown application.

5.1 Beginning

After opening the executable in IDA the user sees a dialog in which they can set different options to analyze the file. This includes the architecture as well as different naming options.[1] After clicking the OK button, IDA will begin analyzing the file. It will map out its strings, imports, exports, the entry point and all other methods. Upon finishing the initial auto analysis, the user sees the flowchart of the entry point function of the application, a window listing all the functions of the binary, a graph overview and an output window which mainly shows errors that might have occurred.

By default, IDA rebases the loaded file to offset 0x400000. This means that in the application itself this offset has to be subtracted from the values to read the correct address.

5.1.1 The Main-function pt. 1

As mentioned in chapter 2.4.1, IDA has the option to list the strings used inside the application in a specific subview. Since the first output in the console is "Measuring %d values", the user can look for this specific string and follow its DATA-XRef to find the function using it (as detailed in Figure 2.3). This leads to the function sub_401920. This method can be renamed to "Main", because it is only referenced once and this is at the end of the "start" function. The documentation of the software already explained, that by default, the application will measure 100 values if no command-line argument is given.

```
401956 cmp     [ebp+arg_0], 2; if commandline arguments count == 2
40195A mov     ebx, 64h; move 100 (default no. of values) into ebx
40195F jnz     short loc_401972; jmp if not equal
401961 mov     eax, [ebp+arg_4]; else put address of 2nd arg into eax
401964 push    dword ptr [eax+4]; push pointer to value on the stack
401967 call    ds:atoi; convert the value to integer
```

[1] IDA can not only analyze x86 binaries, but can also works with ARM-files and many other architectures

```
40196D add     esp, 4; clean the stack
401970 mov     ebx, eax; save the integer value in ebx
```

This aspect is shown on line 40195A, where the default value of 100 (64 hex) is put into the EBX register and on line 40195F a jump occurs, if there is no command-line argument.

```
401980 push  4; get 4 bytes for the new object
401982 mov   [ebp+var_4], 0
401989 call  ds:??2@YAPAXI@Z; operator new(uint)
40198F add   esp, 0Ch; clean the stack
401992 mov   [ebp+arg_0], eax; save ptr to local variable
401995 mov   byte ptr [ebp+var_4], 1
401999 test  eax, eax        ; alloc successful?
40199B jz    short loc_4019AE ; if not, jmp
40199D sub   esp, 8          ; get 8 bytes space on the stack
4019A0 mov   ecx, eax        ; put the object-pointer into ecx
4019A2 call  sub_4012C0      ; thiscall on object
4019A7 mov   edi, eax        ; move the return value into edi
4019A9 mov   [ebp+arg_0], eax ; save ptr to local variable
4019AC jmp   short loc_4019B3 ; jump to the next section
```

The next interesting part to see is that object creation and initialization is split into two parts. First, new memory for the variables of the object is allocated by calling new (which could be exchanged with a *malloc*) and in the next step, initialize the new object with a thiscall function. This is shown on line 401980 where the size of the to-be-created object is pushed onto the stack and at 401989 where new is called to allocate that amount of memory. Since the Object itself has a size of 4 bytes and it is a 32-bit application there is a high possibility that a pointer is stored there.

5.1.2 Subroutine sub_4012C0

In subroutine sub_4012C0 one can see that functions like *WSAStartup*[2] are used, so this is the code that initialized the socket, does name-resolution, and connects. This also explains that the previously created object is some type of connection wrapper.

The key-parts of the intialization are as follows:

```
401321 push    offset pServiceName ; "1234"
401326 push    offset pNodeName ; "127.0.0.1"
40132B mov     [esp+1E0h+pHints.ai_socktype], 1
401333 mov     [esp+1E0h+pHints.ai_protocol], 6
40133B call    ds:getaddrinfo
```

This part resolves the name and returns the IP-address to that name. In this case an IP-address has been passed to function, so it can immediately return that IP-address.

```
401349 mov     esi, [esp+1D0h+pAddrInfo]
40134D test    esi, esi
40134F jz      short loc_401396
401351
401351 loc_401351:             ; CODE XREF: sub_4012C0+D0
401351 push    dword ptr [esi+0Ch] ; protocol
401354 push    dword ptr [esi+8] ; type
```

[2]WSAStartup is a Windows specific function to initialize the WinSock functions.[Win]

```
401357 push    dword ptr [esi+4] ; af
40135A call    ds:socket
401360 mov     [edi], eax
```

Since the initialization of the socket is complete now, it is time to create a socket object. This object is created on line 40135A and afterwards saved in the first (and only) object-variable.

```
401362 cmp     eax, 0FFFFFFFFh
401365 jz      loc_401439
40136B push    dword ptr [esi+10h] ; namelen
40136E push    dword ptr [esi+18h] ; name
401371 push    eax             ; s
401372 call    ds:connect
401378 cmp     eax, 0FFFFFFFFh
40137B jnz     short loc_401392
40137D push    dword ptr [edi] ; s
```

Next, the connect-function is called to open the previously-created socket and connect to the target host. If that operation was successful, the AddrInfo struct will be freed, as it is no longer needed.

```
401392 loc_401392:
401392 mov     esi, [esp+1D0h+pAddrInfo] ;move the ptr in esi
401396
401396 loc_401396:
401396 push    esi; push the ptr
401397 call    ds:freeaddrinfo ;free the AddrInfo
40139D cmp     dword ptr [edi], 0FFFFFFFFh;check if [edi] is valid
4013A0 jnz     short loc_4013A4;if it is, jump
4013A2 jmp     short loc_40140E;if it is not, jmp to throw clause
4013A4 loc_4013A4:
4013A4 mov     ecx, [esp+1D0h+var_4]
4013AB mov     eax, edi;save the socketPtr in eax
4013AD pop     edi;restore register
4013AE pop     esi;restore register
4013AF xor     ecx, esp;bufferoverflow protection
4013B1 call    sub_401D50;same
4013B6 mov     esp, ebp;restore the stack
4013B8 pop     ebp;restore the stack
4013B9 retn    8;clean the stack and return
```

5.1.3 The Main-function pt. 2

The second part of the main function focuses on beginning the transaction, letting the server know how many values the client wants to receive and receiving these values.

```
4019B3 push    offset aSuccessfullyCo ; "Successfully connected!\r\nFetching Value
    "...
4019B8 call    esi ; printf
4019BA push    1               ; SizeOfElements
4019BC push    64h             ; NumOfElements
4019BE mov     [ebp+var_4], 0FFFFFFFFh
4019C5 call    ds:calloc       ; get a buffer for sending
4019CB push    ebx             ; push number of values
4019CC push    offset aMeasureD ; "measure %d\n"
4019D1 push    eax             ; Dest
4019D2 mov     [ebp+Memory], eax ; save the result in a local var
```

```
4019D5 call     ds:sprintf      ; put together the request-string
4019DB mov      edx, [ebp+Memory] ; put the result into edx
4019DE add      esp, 18h        ; clean the stack
4019E1 mov      ecx, edx        ; move the result also into ecx
4019E3 lea      eax, [ecx+1]    ; string min length = 1
4019E6 mov      [ebp+var_18], eax
4019E9 lea      esp, [esp+0]    ; alignment operation. NOP
4019F0
4019F0 loc_4019F0:
4019F0 mov      al, [ecx]       ; put the value of ecx in al
4019F2 inc      ecx             ; increase ecx by 1
4019F3 test     al, al          ; test if al equals 0
4019F5 jnz      short loc_4019F0 ; repeat. find string end
4019F7 sub      ecx, [ebp+var_18];calculate the strlen
4019FA push     0               ; flags
4019FC push     ecx             ; len
4019FD push     edx             ; buf
4019FE push     dword ptr [edi] ; push the socket
401A00 call     ds:send         ; send the buffer to the server
401A06 cmp      eax, 0FFFFFFFFh ; if the operation was successful
401A09 jnz      short loc_401A3F ; jump
```

In line 4019CC one can see that the first message is sent from the client to the server and has to be "measure %d\n" where %d is the number of values one wants to receive. That string is written into a buffer (see 4019D5) and that buffer is being sent (see 401A00) via the socket that was previously opened and connected.

```
401A3F loc_401A3F:
401A3F push     [ebp+Memory]    ; Memory
401A42 call     ds:free   ; free the send buffer
401A48 push     0Ch       ; SizeOfElements
401A4A push     1         ; NumOfElements
401A4C call     ds:calloc ; create a buffer to receive values
401A52 add      esp, 0Ch  ; clean the stack
401A55 mov      esi, eax  ; save ptr to buffer in esi
```

In line 401A42 the send-buffer is freed and in line 401A4C a receive-buffer with 12 bytes in size is created.

```
401A63 loc_401A63:              ; CODE XREF: Main+20B
401A63 push     ecx             ; save ecx in case we need it later
401A64 push     esi             ; push the ptr to the buffer
401A65 mov      ecx, edi        ; set the class instance to the previously created
    object (connection wrapper)
401A67 call     sub_401470      ; read data from the socket
401A6C movss    xmm0, dword ptr [esi+4] ; load bytes 4 to 7 as floatingpoint value
    into register xmm0
401A71 mov      edi, [esi]      ; read the first 4 bytes from the buffer and put
    them into edi
401A73 movss    [ebp+var_18], xmm0 ; save the float variable into a local variable
401A78 movss    xmm0, dword ptr [esi+8] ; load bytes 8 to 11 as floatingpoint value
     into register xmm0
```

In this part of the application the data is received from the host. It consists of 3 variables, of which the first one is an integer and the two others are floating-point variables.

```
401A7D push    0Ch             ; push the size of the object to be created onto
    the stack
401A7F movss   [ebp+Memory], xmm0 ; save the second float variable in another
    local variable
401A84 call    ds:??2@YAPAXI@Z ; operator new(uint)
401A8A add     esp, 4          ; remove one value from the stack
401A8D test    eax, eax        ; test if the "new" was successful
401A8F jz      short loc_401AA9 ; if not, jump
```

Next, a new object is created to save the variables. If the creation of the new object was not successful, there is a jump to error-handling code.

```
401A91 movss   xmm0, [ebp+var_18] ; copy the first variable into the register
401A96 movss   dword ptr [eax+4], xmm0 ; save the first variable in the new
    objects second local variable
401A9B movss   xmm0, [ebp+Memory] ; copy the second variable into the register
401AA0 movss   dword ptr [eax+8], xmm0 ; save the second variable in the new
    objects third local variable
401AA5 mov     [eax], edi      ; save the first variable from the buffer in the
    objects first local variable
401AA7 jmp     short loc_401AAB ; dont null eax
```

The data is copied from the previously used local variables into the new object

```
401AA9 loc_401AA9:             ; CODE XREF: Main+16F
401AA9 xor     eax, eax        ; null eax
401AAB
401AAB loc_401AAB:             ; CODE XREF: Main+187
401AAB movss   xmm0, dword ptr [eax+8] ; load the second float variable into the
    register
401AB0 sub     esp, 10h        ; get 16 bytes of space on the stack for local
    variables
401AB3 cvtps2pd xmm0, xmm0     ; convert the singleprecision floatingpoint value
    into a double precision
401AB6 mov     [ebp+var_18], eax ; save the object into a local variable
401AB9 movsd   [esp+40h+var_38], xmm0 ; add the second float variable to the stack
401ABF movss   xmm0, dword ptr [eax+4] ; load the first float variable into the
    register
401AC4 cvtps2pd xmm0, xmm0     ; convert the variable into double precision
401AC7 movsd   [esp+40h+var_40], xmm0 ; add the first float variable to the stack
401ACC push    dword ptr [eax] ; push the first variable in the object (timestamp)
401ACE push    offset aDFF     ; "%d\t\t%f\t\t%f\r\n"
401AD3 call    ds:printf       ; make the call to print the variables on the
    console
```

This part of the code converts the floating-point variables to double-precision values, moves them onto the stack and calls *printf* to create the console output.

```
401AD9 mov     eax, dword_404458 ; load the address of the beginning of the list
401ADE lea     ecx, [ebp+var_18] ; load the address of the object into ecx to
    prepare the thiscall
401AE1 add     esp, 18h        ; reduce the size of the stack by 24 bytes
401AE4 mov     [ebp+Memory], eax ; save the beginning of the list into a local
    variable
401AE7 push    ecx             ; push the address of the object onto the stack
401AE8 push    dword ptr [eax+4] ; push the endadresse of the list onto the stack
401AEB push    eax             ; push the beginning of the list onto the stack
401AEC call    sub_401D30      ; wrap the element in a list-object. returns the
    address to that list-object
401AF1 mov     edx, dword_40445C ; move the current length of the list into edx
```

```
401AF7 mov     ecx, 15555554h  ; move the maximal length of a list into ecx
401AFC sub     ecx, edx        ; ecx=maximalLength-currentlength
401AFE mov     edi, eax        ; copy the address of the new end of the list into
    edi
401B00 cmp     ecx, 1          ; if the new length of the list exceeded the
    maximum
401B03 jb      loc_401BD8      ; jump
401B09 mov     eax, [ebp+Memory] ; load the beginning of the list into eax again
401B0C inc     edx             ; increase edx (the length of the list)
401B0D mov     dword_40445C, edx ; save the new length into the global variable
401B13 mov     [eax+4], edi    ; save the new end of the list in the struct at [
    BASE+0x404458]+0x4
401B16 mov     eax, [edi+4]    ; get the previous element
401B19 mov     [eax], edi      ; set the correct next-element in the previous
    element
401B1B mov     eax, dword_404450 ; load the counting variable
401B20 mov     edi, [ebp+arg_0] ; load the connection wrapper-object
401B23 inc     eax             ; increase the counting variable
401B24 mov     dword_404450, eax ; save the modified counting variable back into
    memory
401B29 cmp     eax, ebx        ; if (eax <= ebx). if we still need to receive
    values, jump
401B2B jle     loc_401A63      ; repeat the loop
```

Finally, the data is wrapped into a list-element-object and appended to the list of measured values.

The part from 401A63 to 401B2B contains the main-loop of the application that receives the data and saves it into a list. This list is at 0x404458 in memory and points to two variables; at offset 0 the beginning of the list and at offset 0x4 the end of the list. Additionally, at address 0x40445C the current length of the list is saved so there is no need to iterate through the list to find out its size. This list is based on a double-linked-list implementation, which can be seen at line 401AEC where the new object is wrapped into a list-node element that contains a next and a previous pointer. These pointers are modified on lines 401B13 and following, to insert the new item into the list and maintain its consistency. Finally, the counting variable on address 0x404450 is loaded, increased, and written back into memory before checking if there are any interations left to be executed. If there are iterations left, the procedure begins again.

5.2 Conclusion

In this part of the thesis we found out:

- The client tries to connect to host 127.0.0.1 on port 1234.
- The connection is initialized by sending a message in the format "measure XXXX\n", where XXXX is the number of values to be measured.
- The receive-buffer should be 12 bytes in size to fit at least one measurement completely.
- One measurement consists of an integer (4 bytes) that contains a time-stamp and 2 float values (each 4 bytes).
- The model inside the application is a double-linked-list of a custom data-type with one integer and two floats.
- This double-linked-list can be accessed by reading from the memory at 0x404458. There follow 2 pointers, one leads to the first element of the list at offset 0x0, the other one leads to the last element of the list at offset 0x4.
- Each list-element consists of a next-pointer, a previous-pointer and a pointer to the content.
- The double-linked-list is circular, what means that the previous-pointer of the first item will point to the last element, and the next pointer of the last element will point to the first element.
- The size of the list can be found at address 0x40445C.
- The counting-variable is at 0x404450 and increased by one each iteration.

Chapter 6

Using the Acquired Information

In this chapter an application will be created that reads from the example application's memory to demonstrate the use of reverse engineering.

6.1 The Application

The application is written in an object-oriented programming language, so the different aspects are covered in different classes. Following are the different classes and a brief explanation of their usage.

6.1.1 The Structs

```
1  using System;
2  using System.Runtime.InteropServices;
3
4  namespace ProcessReader.Structs
5  {
6      [StructLayout(LayoutKind.Sequential)]
7      struct LinkedList
8      {
9          public IntPtr ListStart;
10         public uint Length;
11     }
12     [StructLayout(LayoutKind.Sequential)]
13     struct ListStart
14     {
15         public IntPtr First;
16         public IntPtr Last;
17     }
18     [StructLayout(LayoutKind.Sequential)]
19     struct MeasuredValue
20     {
21         public int Timestamp;
22         public float Value1;
23         public float Value2;
24     }
25     [StructLayout(LayoutKind.Sequential)]
26     struct ListElement
27     {
28         public IntPtr Next;
29         public IntPtr Previous;
30         public IntPtr Content;
31     }
32
```

```
33 }
```

The structs have been created according to the findings in chapter 5.2. These do not have to be created, but it makes the code maintainable and reduces the number of calls to WinAPI functions which results in faster execution.[Bos][1]

6.1.2 WinAPI Imports

```
1  using System;
2  using System.Runtime.InteropServices;
3
4  namespace ProcessReader
5  {
6      /// <summary>
7      /// Contains the imports to the WinAPI functions
8      /// </summary>
9      public static class Imports
10     {
11         [DllImport("kernel32.dll")]
12         public static extern IntPtr OpenProcess(int dwDesiredAccess, bool
               bInheritHandle, int dwProcessId);
13         [DllImport("kernel32.dll")]
14         public static extern bool ReadProcessMemory(IntPtr hProcess,
15           IntPtr lpBaseAddress, byte[] lpBuffer, int dwSize, ref int
                 lpNumberOfBytesRead);
16         [DllImport("kernel32.dll", SetLastError = true)]
17         public static extern bool CloseHandle(IntPtr hObject);
18
19         public const int PROCESS_WM_READ = 0x0010;
20     }
21 }
```

In order to use the WinAPI functions, they have to be imported and their signature has to be declared. In this example, only the functions *OpenProcess*, *ReadProcessMemory* and *CloseHandle* are used.[Msdb][Pin]

More information about the usage of these functions can be found in the Microsoft Developer Network (MSDN).

[1] On a test of 1000000 iterations reading 1 * 12 bytes took 1125ms while reading 3 * 4 bytes took 3343ms.

6.1.3 The Helper-Class

```
1  using System.Runtime.InteropServices;
2
3  namespace ProcessReader
4  {
5      /// <summary>
6      /// This class provides a helper-function that marshals a byte-array into
           a struct
7      /// Sourcecode from http://stackoverflow.com/questions/14465722/which-
           marshalling-method-is-better
8      /// </summary>
9      class Helper
10     {
11         public static T ByteArrayToStructure<T>(byte[] bytes) where T : struct
12         {
13             var handle = GCHandle.Alloc(bytes, GCHandleType.Pinned);
14             var result = Marshal.PtrToStructure<T>(handle.AddrOfPinnedObject()
                   );
15             handle.Free();
16             return result;
17         }
18     }
19 }
```

The Helper-Class provides a generic method that marshals a byte-array into a struct of the given type.

6.1.4 The Offsets-Class

```
1  namespace ProcessReader
2  {
3      /// <summary>
4      /// This class contains the offsets used inside the application
5      /// </summary>
6      internal static class Offsets
7      {
8          public static int LinkedList
9          {
10             get { return 0x4458; }
11         }
12
13         public static int CountingVariable
14         {
15             get { return 0x4450; }
16         }
17
18     }
19 }
```

The Offsets-Class contains the offsets from chapter 5.2.

6.1.5 The Wrapper-Class

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using ProcessReader.Structs;
5
6  namespace ProcessReader
7  {
8      /// <summary>
9      /// This class hides the WinAPI calls to OpenProcess, CloseHandle and
           ReadProcessMemory and provides an API to access the applications
           information
10     /// </summary>
11     internal class Wrapper
12     {
13         private IntPtr ProcessHandle { get; set; }
14         private IntPtr BaseAddr { get; set; }
15
16         /// <summary>
17         /// Creates a new instance of the Wrapper-Class and implicitly calls
               OpenProcess
18         /// </summary>
19         /// <param name="targetProcess">The Process to read from</param>
20         public Wrapper(Process targetProcess)
21         {
22             ProcessHandle = Imports.OpenProcess(Imports.PROCESS_WM_READ, false
                   , targetProcess.Id);
23             BaseAddr = targetProcess.MainModule.BaseAddress;
24         }
25
26         /// <summary>
27         /// Closes the handle to the process
28         /// </summary>
29         ~Wrapper()
30         {
31             Imports.CloseHandle(ProcessHandle);
32         }
33
34         /// <summary>
35         /// Returns the Loop Counter of the Application
36         /// </summary>
37         public uint Count
38         {
39             get
40             {
41                 var buffer = new byte[4];
42                 int readBytes=0;
43                 Imports.ReadProcessMemory(ProcessHandle, IntPtr.Add(BaseAddr,
                       Offsets.CountingVariable), buffer, 4, ref readBytes);
44                 if (readBytes == 4)
45                     return BitConverter.ToUInt32(buffer, 0);
46                 return 0;
47             }
48         }
49
50         /// <summary>
51         /// Reads the lists information and marshals it into structs
52         /// </summary>
53         public unsafe LinkedList List
54         {
55             get
```

```
56              {
57                  var buffer = new byte[sizeof(LinkedList)];
58                  int readBytes = 0;
59                  Imports.ReadProcessMemory(ProcessHandle, IntPtr.Add(BaseAddr,
                        Offsets.LinkedList), buffer, buffer.Length, ref readBytes)
                        ;
60                  if (readBytes == buffer.Length)
61                      return Helper.ByteArrayToStructure<LinkedList>(buffer);
62                  return new LinkedList();
63
64              }
65          }
66
67          /// <summary>
68          /// Returns the ListStart containing the First and Last-Pointer of the
                 list
69          /// </summary>
70          private unsafe ListStart Liststart
71          {
72              get
73              {
74                  var buffer = new byte[sizeof(ListStart)];
75                  int readBytes = 0;
76                  Imports.ReadProcessMemory(ProcessHandle, List.ListStart,
                        buffer, buffer.Length, ref readBytes);
77                  if (readBytes == buffer.Length)
78                      return Helper.ByteArrayToStructure<ListStart>(buffer);
79                  return new ListStart();
80              }
81          }
82
83          /// <summary>
84          /// Reads a ListElement and returns it. On failure it returns a new (
                empty) element.
85          /// </summary>
86          /// <param name="addr">The address to read from</param>
87          /// <returns>The read element</returns>
88          private unsafe ListElement ReadListElement(IntPtr addr)
89          {
90              var buffer = new byte[sizeof(ListElement)];
91              int readBytes = 0;
92              Imports.ReadProcessMemory(ProcessHandle, addr, buffer, buffer.
                    Length, ref readBytes);
93              if (readBytes == buffer.Length)
94                  return Helper.ByteArrayToStructure<ListElement>(buffer);
95              return new ListElement();
96          }
97
98          /// <summary>
99          /// Read the MeasuredValue and returns it. On failure it retuns a new
                (empty) element.
100         /// </summary>
101         /// <param name="addr">The address to read frm</param>
102         /// <returns>The read element</returns>
103         private unsafe MeasuredValue ReadMeasuredValue(IntPtr addr)
104         {
105             var buffer = new byte[sizeof(MeasuredValue)];
106             int readBytes = 0;
107             Imports.ReadProcessMemory(ProcessHandle, addr, buffer, buffer.
                    Length, ref readBytes);
108             if (readBytes == buffer.Length)
109                 return Helper.ByteArrayToStructure<MeasuredValue>(buffer);
```

```
110             return new MeasuredValue();
111         }
112 
113         /// <summary>
114         /// Returns a List<> of all MeasuredValues inside the
                TargetApplication
115         /// </summary>
116         public List<MeasuredValue> Values
117         {
118             get
119             {
120                 var count = (int) Count;
121                 var result = new List<MeasuredValue>(count);
122                 var first = ReadListElement(Liststart.First);
123                 result.Add(ReadMeasuredValue(first.Content));
124                 for (int i = 1; i < count; i++)
125                 {
126                     first = ReadListElement(first.Next);
127                     result.Add(ReadMeasuredValue(first.Content));
128                 }
129                 return result;
130             }
131         }
132 
133 
134     }
135 }
```

The Wrapper-Class provides an easily accessible interface to interact with the target application. It abstracts the raw WinAPI calls and exposes plain C# properties for the relevant information in the application.

Internally, this is where *ReadProcessMemory* is called to read from the target application. Then, by using the method *Helper.ByteArrayToStructure* from the Helper-Class, the bytes are marshalled into the appropriate structs. The Values-Property uses the combined information from the other properties to enumerate the list containing the measured values in the target application and returns a native List<MeasuredValue>, which can be used like any other List<T>.

6.1.6 The Main-Function

```
1  using System;
2  using System.Diagnostics;
3  using System.Linq;
4 
5  namespace ProcessReader
6  {
7      class Program
8      {
9          /// <summary>
10         /// The mainfunction. It searches for the targetapplication, creates a
                wrapper object
11         /// and uses that to access the internal information of the target
                application
12         /// </summary>
13         static void Main()
14         {
```

```
15              Process targetProcess = Process.GetProcessesByName("ExampleClient"
                    ).First(); //find the application to read from
16              if (targetProcess == null)
17              {
18                  //if it is not found, write a message and abort further
                        execution
19                  throw new Exception("Targetapplication not found. Aborting");
20              }
21              var w = new Wrapper(targetProcess); //create a wrapper-object to
                    access the internal information of the target application
22
23              foreach (var l in w.Values)
24              {
25                  //write all the measured values to the console so we can see
                        that reading worked successfully
26                  Console.WriteLine("Timestamp:{0}\tValue1:{1}\tValue2:{2}", l.
                        Timestamp, l.Value1, l.Value2);
27              }
28              w = null;//free the handle
29              Console.WriteLine("To close the Application press Enter");
30              Console.ReadLine();
31
32
33          }
34      }
35  }
```

The *Main*-function enumerates all available processes to find one with the matching name (ExampleClient). This process is passed as a parameter to the constructor of the Wrapper-Class to create an object of that class.

After that, the Values-Property is accessed to enumerate all the measured values and their information is printed on the console. This would be the point where anything else could be done to the data, like transforming it, or even e-mailing it.

Finally, the object is destroyed and the handle to the process is freed.

Chapter 7

Conclusion

This thesis gave an overview of the different phases of reverse engineering, beginning with the binary file with no knowledge of the internals of the application and ending with the implementation of a new application that interacts with the first one. It has been shown that it is possible to perform this process, but it requires a far-ranging knowledge in different fields of computer science. Reverse engineers must not only know the assembly language and object-oriented programming, but also have a strong understanding of the optimization done by the compiler, such as inlining of functions to reduce the number of call stacks. Additionally, it is important for reverse engineers to have tools such as IDA on hand to make it easier for them to focus on the task of reversing and minimize time performing repetitive interpretive tasks decompiling the program.

Furthermore this thesis covered the legal aspects on the topic of reverse engineering computer software in the United States and the European Union. In the European Union and the United States, reverse engineering is generally allowed for reasons of interoperability, but contrary to the situation in the EU, in the US this right can be overriden by the EULA.

Finally, this thesis showed how to use the insight gained from reverse engineering to create an application that is able to interact with the original application. It is not only possible to read memory from that application, but also write to it and execute code in the context of the target application. Since this requires an even deeper understanding of the topic, only reading from the other process has been covered as part of this thesis.

7.1 Future work/perspective

Reverse engineering is a wide topic and this thesis only gave an overview of the basics. It should be mentioned, that there are more advanced parts to it. First, there is obfuscation which is making the code harder to reverse engineer by outlining/inlining functions and adding operations like *mov eax,eax* which do not alter the result but add additional complexity the reverse engineers have to overcome. Then there are packers which try to make reverse engineering impossible by packing the code and having an unpacker extract the code during runtime.

7.1.1 Hex-Rays Decompiler

Since reverse engineering is time-consuming, there is a Decompiler-plugin for IDA to make it easier and give the reverse engineers more time to focus on the important parts. This decompiler automatically generates pseudo C-code for selected methods or the whole application. The resulting C-code is not like user-written code, but still understandable. Additionally, the decompiler is able to automatically detect inlined functions, like those for string manipulation.[Hex]

This thesis should illustrate the basics of reverse engineering. Due to the fact that the Hex-Rays decompiler aims at professional users and that it is expensive, only those features were used, that can be found in other tools too.

7.1.2 Managed Dll Injection

It is not only possible to inject regular Dlls (written in C++) into the target process, but also managed Dlls which are written in C#. To achieve this, a bootstrap-Dll has to be written in C++ which starts the Common Language Runtime (CLR) inside the process and then loads the desired managed Dll in the target process. This implies an overhead for the additional layer of code, but offers faster development and easier debugging with Visual Studio, since its debugger can be attached to the target process and configured to only step through managed code. Combined with unsafe code and marshalling, it is also possible to create hooks with managed code and call native (internal) functions of the target application by calling the methods from managed code.

Bibliography

Book Sources

[Eag11] Chris Eagle. *The ida pro book: the unofficial guide to the world's most popular disassembler*, 2nd edition, 2011.

Online Documentation

[Msda] *Microsoft Developer Network Dynamic-Link Library Best Practices*. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/dn633971(v=vs.85).aspx.

[Msdb] *Microsoft Developer Network OpenProcess*. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684320(v=vs.85).aspx.

[Msdc] *Microsoft Developer Network VS2013 Fastcall*. URL: http://msdn.microsoft.com/en-us/library/6xa169sk.aspx.

[Pin] *PInvoke.net*. URL: http://www.pinvoke.net/.

Other Sources

[Dll] Three ways to inject your code into another process. URL: http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces.

[Fri14] Steve Friedl. Steve friedls unixwiz.net tech tips. 2014. URL: http://unixwiz.net/techtips/win32-callconv.html.

[Hex] Decompilation vs. disassembly. URL: https://www.hex-rays.com/products/decompiler/compare_vs_disassembly.shtml.

[Mus98] David C. Musker. Protecting and exploiting intellectual property in electronics. 1998. URL: http://www.jenkins.eu/articles-general/reverse-engineering.asp.

[Oll] Ollydbg. URL: http://www.ollydbg.de/.

[Pro] Decompilation and reverse engineering. URL: http://www.program-transformation.org/Transform/DecompilationAndReverseEngineering.

[Usc] 17 u.s. code § 1201 - circumvention of copyright protection systems. URL: http://www.law.cornell.edu/uscode/text/17/1201.

[Win] Wsastartup function. URL: http://msdn.microsoft.com/de-de/library/windows/desktop/ms742213(v=vs.85).aspx.

[IEE11] IEEE USA. Position statement reverse engineering. 2011. URL: http://www.ieeeusa.org/policy/positions/reverseengineering1111.pdf.

Unpublished

[Bos] Julian Bosch. Speed of ReadProcessMemory on reading 1x12 and 3x4 bytes.