Reverse Engineering

Studiengang Digitale Forensik
Ausarbeitung im Rahmen des Modul 110

Markus Stoll

projekt20.exe

Datum 25.10.2017

Inhaltsverzeichnis

1 projekt20.exe

1.1 Einleitung

1.1.1 Kontext

Die Projektarbeit findet im Kontext des ***Modul 110 Reverse Engineering*** im Master Studiengang Digitale Forensik statt.

1.1.2 Modul M110 Reverse Engineering

Ein regelmäßiger Untersuchungsgegenstand in der digitalen Forensik ist unbekannte Software, deren Funktionsweise analysiert werden soll. Im Gegensatz zum „Software Engineering“, bei dem es um die Übersetzung von Anforderungen in Code geht, muss hier der umgekehrte Weg gegangen werden, also das Herauslesen von Anforderungen und Intentionen aus Software. Man spricht deshalb von „Software Reverse Engineering“, oder einfach nur kurz „Reverse Engineering“. [1)]

1.1.3 Projektarbeit

1.1.3.1 Zweck

Die Projektarbeit ist eine wissenschaftliche Arbeit und soll das Erlernen der Techniken und Anforderungen unterstützen, die auch für das Erstellen einer Masterthesis benötigt werden.

Im Rahmen einer konkreten Problem- bzw. Aufgabenstellung soll ein eigener Lösungsweg gefunden, umgesetzt, dokumentiert und präsentiert werden. Die Dokumentation muss in Inhalt, Gliederung und Form den Anforderungen an eine wissenschaftliche Arbeit genügen.

1.1.3.2 Aufgabenstellung

Analysieren einer individualisierten Schadsoftware (Malware) mit Hilfe von Hex-Rays IDA Disassembler statisch und dynamisch. Die Malware ist ein Binärprogramm (Windows Executable).

Die Analyse der Malware untergliedert sich in drei Teilaufgaben:

1. Durch die Eingabe eines korrekten Schlüssels (Challenge) können Sie die Malware deaktivieren. Finden und dekompilieren Sie das Prüfverfahren für den Schlüssel und formulieren Sie einen möglichen Quellcode des Verfahrens in der Sprache C. Geben Sie im Projektbericht ein Beispiel für einen gültigen Schlüssel zur Deaktivierung Ihrer Malware an.

2. Es werden mehrere Verfahren zur Kontrollfluss-Obfuskierungen angewandt. Beschreiben Sie die Verfahren und ihren Ablauf detailliert.

3. Den Ablauf der übrigen Programmteile sollen Sie auf einer höheren Abstraktionsebene rekonstruieren und darstellen, d.h. die Beschreibung der Funktionsweise und der Zusammenhänge ist hier wichtiger als beispielsweise einzelne Registerbewegungen.

1.1.4 Arbeitsumgebung

Betriebssystem

Die Malware wurde unter einer virtuellen Maschine mit der folgenden Arbeitsumgebung untersucht:

Virtualisierungssoftware:	Oracle, Virtual Box 5,1,20
Virtuelle Maschine:	Microsoft, Windows 7 Professional 32 Bit deutsch
Disassembler:	Hex-Rays SA, IDA Freeware Version 5.0
C Code:	Microsoft, Visual Studio 2017 Community
Pfad der Malware:	c:\M110\Aufgabe

1.1.5 Was ist Obfuscation?

Obfuscation (engl. „Verschleierung“) bezieht sich auf die Transformation von Programmcode. Ziel ist es, die Ermittlung der Semantik und der Funktionalität eines Programms zu erschweren, jedoch dessen Funktionalität zu erhalten.

Prinzipiell kann Obfuscation auf zwei Bestandteile eines Programms angewandt werden:

➔ Kontrollfluss

Auf den chronologischen und kausalen Ablauf des Programms (Sprungpunkte, Verzweigungen, Ausführungsreihenfolge der Anweisungen).

➔ Datenstrukturen

Einsatzgebiete von Obfuscation

Schutz vor Geistigem Eigentum innerhalb der Software. Dadurch soll verhindert werden, dass technisches Knowhow oder Lizensierungscodes gestohlen werden.

Die Verschleierung innerhalb einer Malware soll dem Analysten die Arbeit möglichst erschweren. So dass nur schwer analysiert werden kann was die Malware macht.

1.1.6 Was ist die Malwaretechnik Anti-Debugging?

Der Malwareanalyst verwendet zur Analyse einer Malware verschiedene Tools. Unter anderen wird in der Regel ein Debugger verwendet mit dem der Analyst den Programmablauf der Malware in einzelnen Schritten analysieren kann.

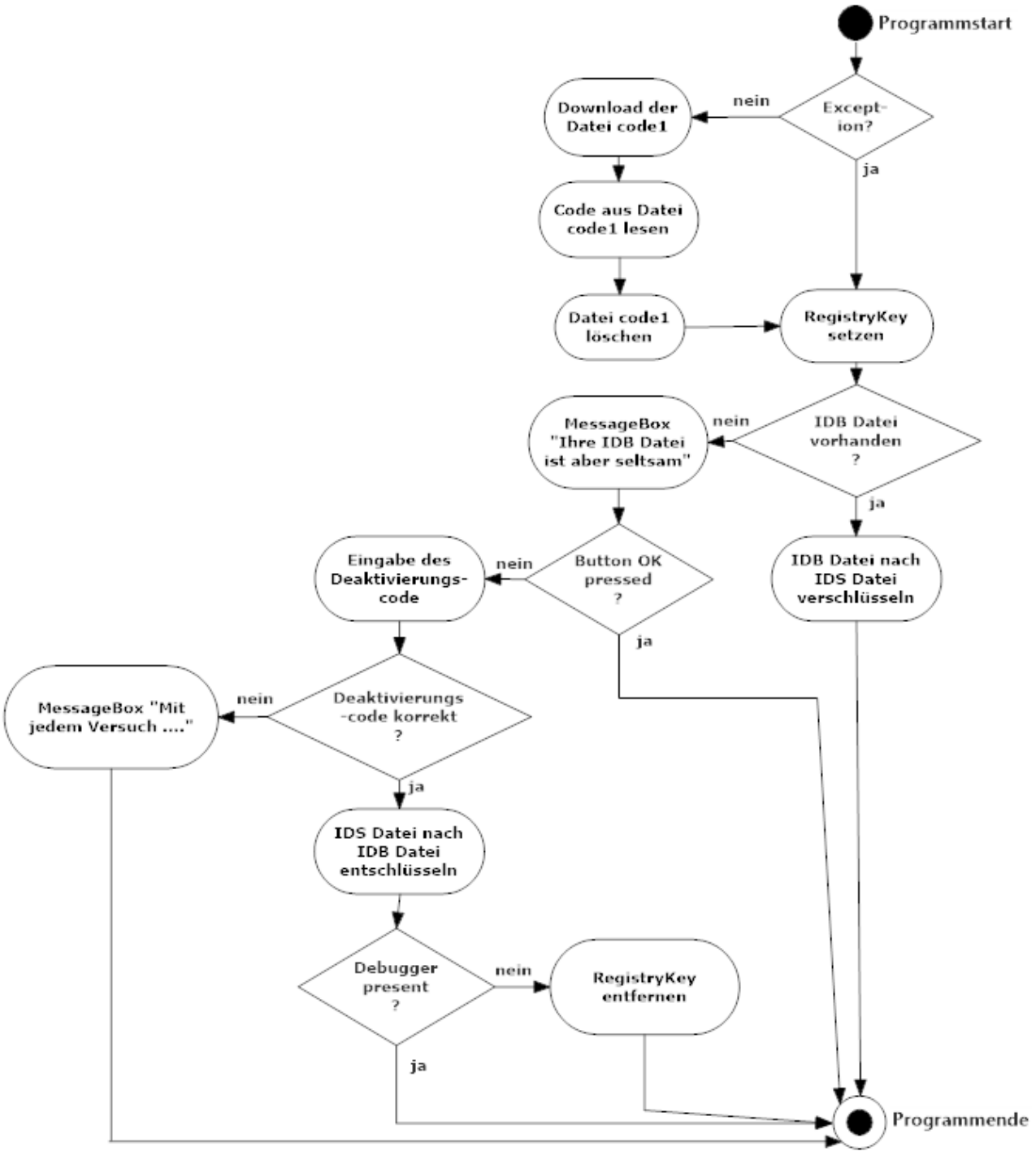
Als Anti-Debugging bezeichnet man allgemein Aktivmaßnahmen einer Malware, um sich gegen die Analyse durch einen Debugger zur Wehr zu setzen. Wird ein Debugger erkannt, so wird die Malware sich möglicherweise gutartig verhalten oder auch einen Programmabsturz provozieren, was die Analyse erschwert.

1.2 Projekttagebuch

Datum	***Stunden***	***Tätigkeit***
22.9.2017	8	Programmanalyse
23.9.2017	8	Programmanalyse
29.9.2017	8	Programmanalyse
30.9.2017	8	Programmanalyse
7.10.2017	4	Ausarbeitung
13.10.2017	4	Ausarbeitung
14.10.2017	4	Ausarbeitung
20.10.2017	4	Ausarbeitung
21.10.2017	4	Ausarbeitung
25.10.2017	2	Ausarbeitung

1.3 Ergebnisse

1.3.1 Wie funktioniert die Malware?



Flowchart 1: Funktionalität der Malware

1.3.2 Deaktivierung der Schadsoftware

- Eingabe eines gültigen Deaktivierungscodes.
  ➔ Datei wird Entschlüsselt
- Der gültige Deaktivierungscode muss außerhalb des Debuggers eingegeben werden.
  ➔ Registry Key wird entfernt

1.3.3 Gültiger Deaktivierungscode

ak12Y345AAAAAAAYYYYYYYYY

1.3.4 Bedingungen für den Deaktivierungscode

Der Deaktivierungscode wird auf die folgenden Bedingungen geprüft:

Länge	== 25 oder 26 Zeichen
2. Zeichen	== k
5. Zeichen	== 21. Zeichen
Anzahl Zahlen	>= 5
Anzahl Großbuchstaben	>= 5
21. Zeichen	== Y
9. Zeichen	== 15. Zeichen
10. Zeichen	== 14. Zeichen
11. Zeichen	== 13. Zeichen
1. Zeichen "klein"	== 13. Zeichen "groß“ //1.Zeichen == 'a' & 13. Zeichen == 'A'

1.3.5 Deaktivierungsalgorithmus in C

```
int EingabePruefen(const char* ptrEingabeString)
{
	int isEingabePruefen = 0xFFFFFFFF;	// -1 Eingabe nicht gültig

	if (ptrEingabeString != nullptr)
	{
		// Länge der eingabe prüfen
		int eingabeLaenge = strlen(ptrEingabeString);

		if (eingabeLaenge == 25 || eingabeLaenge == 26)
		{
			// 2. Zeichen muss 'k' sein
			if (ptrEingabeString[1] == 'k')
			{
				// 5. Zeichen muss gleich dem 21. Zeichen sein
				if (ptrEingabeString[4] == ptrEingabeString[20])
				{
					// mind. 5 Zahlen und mind. 5 Grossbuchstaben
					int anzahlZahl = 0;
					int anzahlGrossbuchstaben = 0;
					for (int i = 0; i < eingabeLaenge; i++)
					{
						// Zahl?
						if (ptrEingabeString[i] >= 0x30 &&
						    ptrEingabeString[i] <= 0x39)
						{
							anzahlZahl++;
						}
```

```
                                        // Grossbuchstabe?
                                        if (ptrEingabeString[i] >= 0x41 &&
                                            ptrEingabeString[i] <= 0x5A)
                                        {
                                                anzahlGrossbuchstaben++;
                                        }
                                }

                                if (anzahlZahl >= 5 && anzahlGrossbuchstaben >= 5)
                                {
                                        // 21. Zeichen muss 'Y' sein
                                        if (ptrEingabeString[20] == 'Y')
                                        {
                                                //  9. Zeichen == 15. Zeichen
                                                // 10. Zeichen == 14. Zeichen
                                                // 11. Zeichen == 13. Zeichen
                                                int inc = 8;
                                                int dec = 14;
                                                bool isZeichen9Bis15Valid = false;
                                                for (; inc != dec; inc++, dec--)
                                                {
                                                        if (ptrEingabeString[inc] ==
                                                            ptrEingabeString[dec])
                                                        {
                                                                isZeichen9Bis15Valid = true;
                                                        }
                                                        else
                                                        {
                                                                isZeichen9Bis15Valid = false;
                                                                break;
                                                        }
                                                }

                                                if (isZeichen9Bis15Valid)
                                                {
                                                        // 1. Zeichen "klein" == 13. Zeichen "gross"
                                                        // Bsp.: 1. Zeichen = 'a' und 13. Zeichen = 'A'
                                                        char zeichen13Klein =
                                                                tolower(ptrEingabeString[12]);

                                                        if (ptrEingabeString[0] == zeichen13Klein)
                                                        {
                                                                isEingabePruefen = 1; // gültig
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }

        return isEingabePruefen;
}
```

1.3.6 Verwendete Kontrollflussobfuskierungen

1.3.6.1 Structured Exception Handling

Mit einem Exceptionhandler werden Ausnahmesituationen (Exceptions) abgefangen und gehandelt. Wird versucht aus einer Datei zu lesen die nicht existiert, so wird eine FileNotFoundException ausgelöst.

Damit das Programm trotz dieses Fehlers weiterlaufen kann, hat der Programmierer die Aufgabe im Exceptionhandler notwendig Dinge durchführen, wie z.B. eine Fehlermeldung an den Benutzer auszugeben oder Transaktionen wieder rückgängig machen.

Ist kein Exceptionhandler vom Programmierer definiert, so wird der vom System gegebene Default-Exceptionhandler verwendet. Dieser beendet in der Regel das Programm.

Der Exceptionhandler kann aber auch zur Obfuskierung verwendet werden. Löst der Malwareprogrammierer eine Exception aus, so wird der normale Programmablauf unterbrochen und der Programmablauf geht im Exceptionhandler weiter. Die Codezeilen nach der die Exception ausgelöst wurde, werden nicht ausgeführt.

Beispielcode

```
public static void Main()
{
    try
    {
            int i = 10;
            i = i/0;      // Absichtliche Division durch 0
                          // Nachfolgende Zeilen werden nicht ausgeführt
                          // Weiter geht es im Catch-Block
            …
            …
            …
    }
    catch (Exception ex)
    {
            // hier geht es weiter nach der Division durch 0
            …
            …
            …
    }
}
```

Code in IDA

Adresse	***Beschreibung***
	FS:0 Zeiger auf die Liste der Exceptionhandler
0x0040209C	Exceptionhandler einhängen
0x004020A3	Exceptionhandler einhängen
…	
…	
0x004020BA	Division durch 0
…	Nachfolgende Zeilen werden nicht ausgeführt, weiter geht es im Exceptionhandler

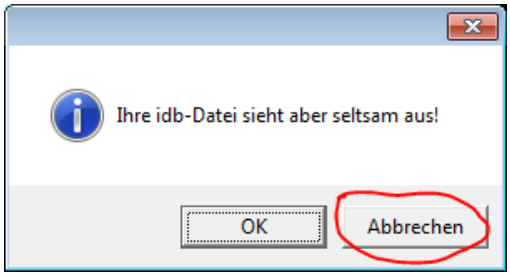
```
.text:00402097 call    sub_401EC9                          ; Call Procedure
.text:0040209C push    large dword ptr fs:0                ; Exceptionhandler - Einhängen
.text:004020A3 mov     large fs:0, esp
.text:004020AA xor     ebx, eax                            ; Logical Exclusive OR
.text:004020AC shl     eax, 3                              ; Shift Logical Left
.text:004020AF adc     eax, eax                            ; Add with Carry
.text:004020B1 adc     ecx, eax                            ; Add with Carry
.text:004020B3 and     cx, 0                               ; Logical AND
.text:004020B7 shl     ecx, 10h                            ; Shift Logical Left
.text:004020BA div     ecx                                 ; Exception ->
.text:004020BA                                             ; weiter im Exceptionhandler.
.text:004020BC mov     eax, ds:AdresseMethode_LoadLibrary ;         Adresse der Methode
.text:004020C1 mov     dword ptr [esp], offset String_Wininet_dll ; DLL Name als Parame
.text:004020C8 call    eax                                 ;                              Wi
.text:004020C8                                             ;
.text:004020CA sub     esp, 4                              ; Integer Subtraction
.text:004020CD mov     ds:Adresse_Wininet_dll, eax
.text:004020D2 mov     eax, ds:AdresseMethode_LoadLibrary ;         Adresse der Methode
.text:004020D7 mov     dword ptr [esp], offset String_urlmon_dll ;  DLL Name als Parame
```

1.3.6.2 Zweckentfremdung der GUI Controls

Diese Obfuskierung steht nicht im Studienbrief.

Die MessageBox stellt dem Programmierer mehrere Buttons bereit. Normalerweise wird beim Button „Cancel“ ein Vorgang abgebrochen und beim Button „OK“ ein Vorgang bestätigt.

Der Malwareprogrammierer hat hier die beiden Buttons vertauscht. Bei der Messagebox muss der Button „abbrechen“ gewählt werden damit der Deaktivierungscode eingegeben werden kann. Wird wie normal „OK“ gewählt, beendet sich das Programm und die Eingabe des Deaktivierungscodes ist nicht mehr möglich.



1.3.6.3 Unterprogrammaufruf auf den Folgebefehl

Der ***Call*** Befehl ruft bei normaler Verwendung eine Funktion über deren Funktionsnamen auf. Die Parameter der Funktion werden zuvor auf den Stack gelegt. Der Programmablauf geht dann in der gerufenen Funktion weiter. Hier im Beispiel ruft der Call Befehl die Funktion ***StringRot13()*** auf.

Code in IDA

Adresse	***Beschreibung***	
0x00401625	Call StringRot13	; Aufruf der Funktion ***StringRot13()***

```
.text:00401616 sub     esp, 8                          ; Integer Subtractio
.text:00401619 mov     ds:dword_4064AC, eax
.text:0040161E mov     [esp+440h+var_440], offset String_RegOpenKeyExA ; "E
.text:00401625 call    StringRot13                     ; Call Procedure
.text:0040162A mov     [esp+440h+var_440], offset String_RegistryKey ; SOFT
.text:00401631 call    StringRot13                     ; Call Procedure
```

Mit dem ***Call $+5*** Befehl wird hier nicht eine Funktion aufgerufen, sondern eine Adresse. Die Adresse berechnet sich aus der Adresse des aktuellen Befehles + 5 Bytes. Dies entspricht jedoch nur dem nächsten Befehl. Dadurch ist in IDA nicht im Klartext lesbar welche Funktion mit dem Call Befehl aufgerufen wird.

Code in IDA

Adresse	***Beschreibung***	
0x0040204E	Call $+5	; 0x0040204E + 0x5 = 0x00402053
		; Es wird nur zum nächsten Befehl gesprungen.
0x00402053	pop eax	

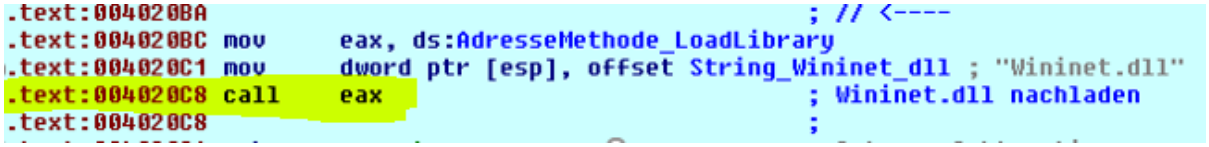
```
.text:00402043 loc_402043:
.text:00402043 cmp     [ebp+var_1C], 7
.text:00402047 jle     short loc_402026
.text:00402049 push    offset loc_402064
.text:0040204E call    $+5
.text:00402053 pop     eax
.text:00402054 mov     ebx, dword_403B4C
.text:0040205A movzx   ebx, bl
```

1.3.6.4 Pointer indirection

Mit dem Call Befehl wird hier nicht eine Funktion aufgerufen, sondern eine Adresse. Die Adresse wird zuvor in das Register eax geladen. Dadurch ist in IDA nicht im Klartext lesbar welche Funktion mit dem Call Befehl aufgerufen wird.

Code in IDA

Adresse	***Beschreibung***
0x004020BC	Laden der Funktionsadresse nach eax
0x004020C1	Parameter der Funktion auf den Stack
0x004020C8	Aufruf der Funktion über die Adresse die in EAX steht



1.3.6.5 Zweckentfremdung des ret-Befehls

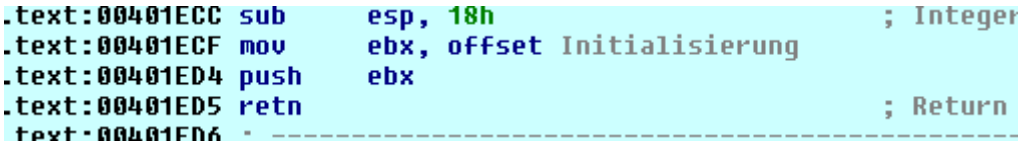
Mit dem ret-Befehl wird eine Unterfunktion beendet und es geht danach mit der Adresse weiter die auf dem Stack liegt, der Rücksprungadresse.

Zur Kontrollflussobfuskierung wird der ret-Befehl wie folgt verwendet.

Code in IDA

Adresse	***Beschreibung***
0x00401ECF	Adresse der Funktion ***Initialisierung()*** nach EBX
0x00401ED4	EBX auf den Stack, Adresse der Funktion liegt jetzt auf dem Stack
0x00401ED5	Mit dem ret Befehl wird jetzt die Funktion ***Initialisierung()*** aufgerufen

Der ret Befehl bewirkt hier eine ganz normale indirekte Verzweigung.



1.3.7 Weitere verwendete Obfuskierungen und Malwaretechniken

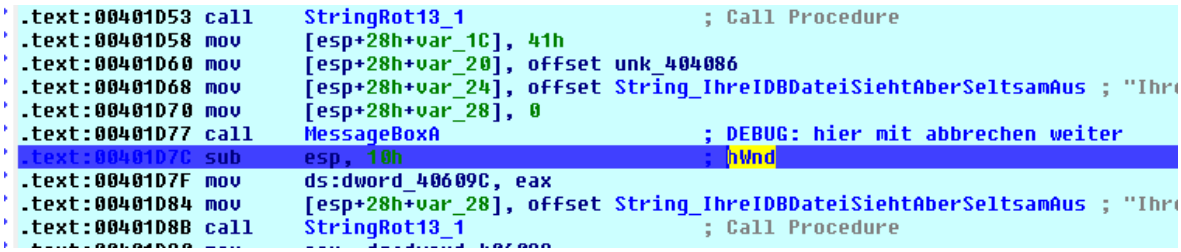
1.3.7.1 Stringobfuscation

Die Zeichenketten liegen mit Rot13 Verschlüsselt im Speicher. Diese werden direkt vor der Verwendung, z.B. in der Dialogbox, entschlüsselt und sofort danach wieder verschlüsselt. Dadurch sind die Zeichenketten nur zur Laufzeit während des Aufrufs der MessageBox in IDA im Klartext lesbar.

ROT13 (rotiere um 13 Stellen“) ist eine Caesar-Verschlüsselung (auch Verschiebechiffre genannt), mit der auf einfache Weise Texte verschlüsselt werden können. Dies geschieht durch Ersetzung von Buchstaben – bei ROT13 im Speziellen wird jeder Buchstabe des lateinischen Alphabets durch den im Alphabet um 13 Stellen davor bzw. dahinter liegenden Buchstaben ersetzt. [3)]

Code in IDA

Adresse	***Beschreibung***
0x00401D53	Entschlüsselung, Zeichenkette ist jetzt lesbar
0x00401D77	Ausgabe der Zeichenkette in der MessageBox
0x00401D8B	Verschlüsselung, Zeichenkette nicht mehr lesbar



1.3.7.2 Junkcode insertion

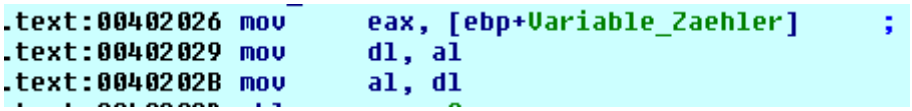
Bei Junkcode insertion hat der Malwareprogrammierer Codezeilen oder ganze Funktionen programmiert die im eigentlichen Sinn nichts bewirken.

Beispielcode

```
void ExampleJunkCodeInsertion()
{
	int wert1 = 1;
	int wert2 = 2;
	wert2 = wert1;
	wert1 = wert2;		// JunkCode
}
```

Code in IDA

Adresse	***Beschreibung***
0x00402026	AL nach DL
0x00402029	DL nach AL, Junkcode, keine Bedeutung



1.3.7.3 Import Hiding

Die beiden Bibliotheken urlmon.dll und Wininet.dll werden zur Laufzeit mittels der Funktion LoadLibrary nachgeladen. Mit der Funktion GetProcAddress kann dann die Adresse einer exportierten Funktion der Bibliothek ermittelt werden.

Durch das Nachladen werden die verwendeten Funktionen aus den Bibliotheken in IDA nicht unter den importierten Funktionen aufgelistet. Somit wird dem Programmanalyst die Verwendung dieser Methoden bei der statischen Analyse verborgen.

Code in IDA

Adresse	***Beschreibung***
0x004020BC	Adresse der Methode LoadLibrary() nach EAX
0x004020C1	DLL Name (wininet.dll) als Parameter auf den Stack
0x004020C8	wininet.dll nachladen
…	
…	
…	
0x004020E8	Adresse der Methode GetProcaddress() nach EDX
0x004020EE	DLL Name (wininet.dll) als Parameter auf den Stack
0x004020F3	Funktionsname (InternetGetConnecti…) als Parameter auf den Stack
0x004020FE	Adresse der exportierten Funktion ermitteln
	Mit der ermittelten Adresse kann die Funktion InternetGetConnectionState() mit dem Call Befehl aufgerufen werden.

```
.text:004020BC mov     eax, ds:AdresseMethode_LoadLibrary ;         Adresse der
.text:004020C1 mov     dword ptr [esp], offset String_Wininet_dll ; DLL Name als
.text:004020C8 call    eax                                 ;
.text:004020C8                                             ;
.text:004020CA sub     esp, 4                              ; Integer Subtraction
.text:004020CD mov     ds:Adresse_Wininet_dll, eax
.text:004020D2 mov     eax, ds:AdresseMethode_LoadLibrary ;         Adresse der
.text:004020D7 mov     dword ptr [esp], offset String_urlmon_dll ;  DLL Name als
.text:004020DE call    eax                                 ;
.text:004020E0 sub     esp, 4                              ; Integer Subtraction
.text:004020E3 mov     ds:dword_4064B0, eax
.text:004020E8 mov     edx, ds:Adresse_GetProcAddress  ;             Adresse der
.text:004020EE mov     eax, ds:Adresse_Wininet_dll     ;             DLL Name in
.text:004020F3 mov     dword ptr [esp+4], offset String_InternetGetConnectionSta
.text:004020FB mov     [esp], eax
.text:004020FE call    edx                                 ; // Adresse der Function
.text:00402FF                                              ; // Funktionsrückgabe is
```

1.3.7.4 Idiome

Idiome sind Programmierkniffe, bei denen Instruktionen für einen anderen als den offensichtlichen Zweck verwendet werden. Links-Shift einer Binärzahl um 1 Bit entspricht einer Multiplikation mit 2.

Idiome werden durch den Malwareprogrammierer bewusst eingesetzt um die Analyse des Programms zu erschweren. Optimierungen durch den Compiler oder Programmierer können ebenfalls Idiome entstehen lassen.

Häufiges Idiom ist:	xor eax, eax,
dass statt dieses Befehls verwendet wird:	mov eax, 0.

Die Gründe für die Verwendung des xor-Befehls sind der kürzere Opcode und die je nach Prozessormodell schnellere Befehlsausführung.

Code in IDA

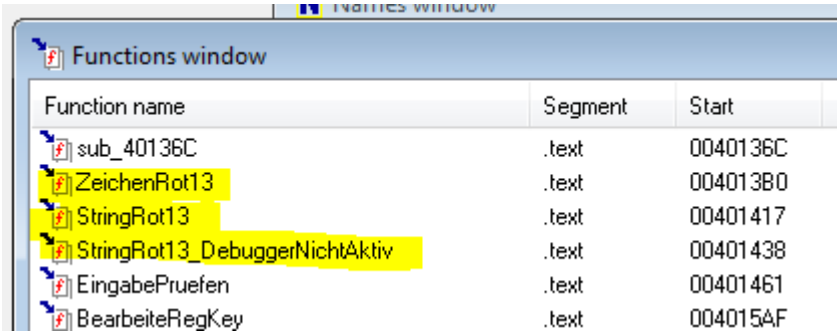
Adresse	***Beschreibung***
0x0040202D	EAX Links-Shift um 2 Bit, entspricht EAX = EAX * 2 * 2

```
.text:0040202D shl     eax, 2                          ; EAX = EAX * 2 * 2;
.text:00402030 add     eax, edx                        ; Add
```

1.3.7.5 Splitting

Bei Splitting teilt der Malwareprogrammierer eine Funktion in mehrere Funktionen auf. Dadurch wird der Zusammenhang der Codeteile verschleiert und die Funktionalität bleibt erhalten. Durch die höhere Anzahl von Funktionsaufrufen wird jedoch die Laufzeit etwas langsamer.

Die Funktion zur Verschlüsselung/Entschlüsselung der Strings mit dem Rot13 Verfahren ist auf 3 Funktionen gesplittet.



Die Funktion ***StringRot13()*** geht den String durch und ruft zur Verschlüsselung/Entschlüsselung für jedes Zeichen die Funktion ***ZeichenRot13()*** auf. Die Funktion ***StringRot13_DebuggerNichtAktiv()*** wird aufgerufen wenn der Debugger nicht aktiv ist. Diese Funktion führt aber die Rot13 Verschlüsselung/Entschlüsselung eines Zeichens selbst aus.

1.3.7.6 Persistenz

In der Regel möchte der Malwareprogrammierer, dass seine Schadsoftware so lange wie möglich auf einem infizierten Rechner bleibt und ausgeführt wird. So wird gewährleistet, dass die Schadsoftware möglichst lange Daten wie z.B. Zugangsdaten übermittelt.

Die einfachste Möglichkeit ist die Malware automatisch beim Start von Windows mit zu starten. Dies ist über die Run-Schlüssel der Registry möglich. Run-Schlüssel sorgen dafür, dass Programme automatisch ausgeführt werden, sobald sich ein Benutzer anmeldet.

In der Registrierung von Windows sind die folgenden vier Run-Schlüssel enthalten:

- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce

Der Malwareprogrammierer hat den folgenden Eintrag in der Registry eingerichtet:

Schlüssel:
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Key:
_SRV_CTRL_ cmd.exe /c start /min "C:\M110\Aufgabe\Stoll\projekt20.exe"

Beim Pfad des zu startenden Programmes hat der Malwareprogrammierer im Key, Anführungszeichen verwendet ***"C:\M110\Aufgabe\Stoll\projekt20.exe"***. Dies kann Windows 7 nicht interpretieren, so dass Windows 7 den angegebenen Pfad zu der auszuführenden Datei nicht findet und die Schadsoftware nicht gestartet wird. Es wird hier nur ein DOS Command Fenster minimalisiert ausgeführt.

Das Ziel durch die Persistenz war, dass die Malware wieder gestartet wird. Beim Start der Malware wird dann, bei vorhandener IDA Datei, diese wieder zur IDS verschlüsselt.

Siehe dazu auch den Punkt 1.3.7.2 Modifikation der Registry.

1.3.7.7 Anti-Debugging

PEB (Process Environment Block)
Die Verwaltungsinformationen zu aktiven Prozessen werden in der speziellen Datenstruktur PEB (Process Environment Block) hinterlegt. Im PEB werden prozessspezifische Informationen abgelegt. Dazu gehört die Prozess-ID, über die der Prozess eindeutig identifizierbar ist. Sowie das Byte ***BeingDebugged*** welches anzeigt ob der Prozess Debugged wird.

PEB structure

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Der Programmierer kann über die Funktionen IsDebuggerPresent, CheckRemoteDebuggerPresent und OutputDebugString das Byte BeingDebugged im PEB abfragen.

Beispielcode

```
// IDS Datei nach IDA Datei entschlüsseln
//

bool isDebuggerpresent = IsDebuggerPresent();
if (isDebuggerpresent == false)
{
      // Regkey entfernen
}
```

Code in IDA

Adresse	***Beschreibung***
0x00401FD5	Abfrage des Bytes BeingDebugged im PEB
0x00401FD8	Auswerten des Bytes
0x00401FDE	Auswerten des Bytes
0x00401FE1	Das ausgewertete Byte in einer Variablen speichern

```
.text:00401FC9 mov     [esp+14h+var_14], offset String_advapi32_dll ; "advapi32.dll"
.text:00401FD0 call    StringRot13                  ; Call Procedure
.text:00401FD5 mov     eax, large fs:18h            ; Process Environment Block -> BeingDebugged
.text:00401FD5                                      ; Wird der aktuelle Prozess debugged?
.text:00401FDB mov     eax, [eax+30h]
.text:00401FDE mov     bl, [eax+2]
.text:00401FE1 mov     IsDebuggerAktiv, bl
.text:00401FE7 leave                                ; High Level Procedure Exit
```

1.3.8 Funktionalität der Malware zur Laufzeit

1.3.8.1 Verschlüsselung / Entschlüsselung der IDB Datei

Ist die IDB Datei vorhanden, dann wird der Inhalt aus der IDB Datei ausgelesen, XOR – Verschlüsselt und als IDS Datei gespeichert. Die vorhandene IDB Datei wird anschließend gelöscht.

Als Schlüssel wird ein Byte-Array der Größe 10 mit Zufallszahlen erstellt. Die 10 Schlüsselelemente des Byte-Arrays werden XOR Verschlüsselt mit 0x0D auf den ersten 10 Bytes der neuen IDS Datei abgespeichert. So ist gewährleistet, dass der Schlüssel für das Entschlüsseln verfügbar ist.

Ist keine IDB Datei aber eine IDS Datei vorhanden, wird die IDS Datei wieder entschlüsselt und die IDB Datei erstellt. Die IDS Datei anschließend gelöscht.

Code in IDA: Schlüsselarray (Byte-Array) mit Zufallszahlen erstellen

```
.text:00401A43 ZufallszahlErstellen:                   ; CODE XREF: @@+15C↓j
.text:00401A43 call    rand                            ; Call Procedure
.text:00401A48 and     eax, 8000003Fh                  ; Logical AND
.text:00401A4D test    eax, eax                        ; Logical Compare
.text:00401A4F jns     short loc_401A56                ; Jump if Not Sign (SF=0)
.text:00401A51 dec     eax                             ; Decrement by 1
.text:00401A52 or      eax, 0FFFFFFC0h                 ; Logical Inclusive OR
.text:00401A55 inc     eax                             ; Increment by 1
.text:00401A56
.text:00401A56 loc_401A56:                             ; CODE XREF: @@+141↑j
.text:00401A56 lea     edx, [eax+21h]                  ; Load Effective Address
.text:00401A59 mov     eax, [ebp+Variable_Zaehler_Byte]
.text:00401A5C add     eax, offset Offset_SchluesselArray ; Add
.text:00401A61 mov     [eax], dl
.text:00401A63 inc     [ebp+Variable_Zaehler_Byte]     ; Increment by 1
.text:00401A66
.text:00401A66 SchluesselArrayErstellen:               ; CODE XREF: @@+133↑j
.text:00401A66 cmp     [ebp+Variable_Zaehler_Byte], 9  ; Zaehler 0 - 9
.text:00401A6A jle     short ZufallszahlErstellen      ; Jump if Less or Equal (ZF=1
.text:00401A6C mov     [esp+818h+var_814], offset aWb  ; "wb"
.text:00401A74 lea     eax, [ebp+var_7E9]              ; Load Effective Address
.text:00401A7A mov     [esp+818h+var_818], eax
.text:00401A7D call    fopen                           ; Datei projekt20.ids anlegen
.text:00401A82 mov     [ebp+var_18], eax
.text:00401A85 mov     [ebp+Variable_Zaehler_Byte], 0
.text:00401A8C jmp     short SchluesselendeErreicht    ; Jump
.text:00401A8E ; ---------------------------------------------------------------------
```

Code in IDA: Schlüsselarray in IDS Datei schreiben

Adresse	***Beschreibung***
0x00401A98	Schlüsselbyte ***XOR*** mit 0x0D

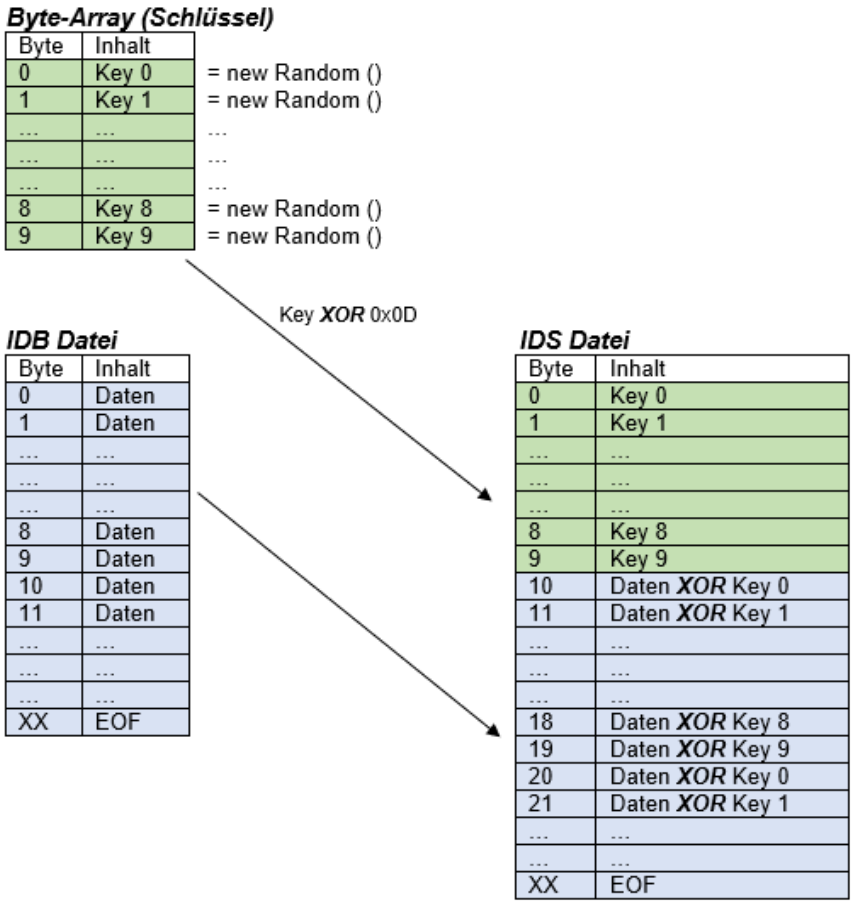
```
.text:00401A8E
.text:00401A8E SchluesselInIdsDateiSchreiben:          ; CODE XREF: @@+1A6↓j
.text:00401A8E mov     eax, [ebp+Variable_Zaehler_Byte]
.text:00401A91 add     eax, offset Offset_SchluesselArray ; Add
.text:00401A96 mov     al, [eax]
.text:00401A98 xor     eax, 0Dh                        ; XOR Verschlüsselung mit 0x0D
.text:00401A9B movzx   eax, al                         ; Move with Zero-Extend
.text:00401A9E mov     edx, [ebp+var_18]
.text:00401AA1 mov     [esp+818h+var_814], edx         ; filestream
.text:00401AA5 mov     [esp+818h+var_818], eax         ; zeichen
.text:00401AA8 call    fputc                           ; Call Procedure
.text:00401AAD inc     [ebp+Variable_Zaehler_Byte]     ; Increment by 1
.text:00401AB0
.text:00401AB0 SchluesselendeErreicht:                 ; CODE XREF: @@+17E↑j
.text:00401AB0 cmp     [ebp+Variable_Zaehler_Byte], 9  ; Compare Two Operands
```

Code in IDA: Bytes von IDB nach IDS Datei verschlüsseln und schreiben

Adresse	***Beschreibung***
0x00401AF1	Byte aus IDB Datei ***XOR*** mit Schlüsselbyte aus Schlüssel-Array

```
.text:00401ABD ZeichenVonIdbNachIds:                   ; CODE XREF: @@+1F8↓j
.text:00401ABD mov     eax, [ebp+Variable_Zaehler_Byte]
.text:00401AC0 inc     eax                             ; Increment by 1
.text:00401AC1 mov     ecx, 0Ah
.text:00401AC6 cdq                                     ; EAX -> EDX:EAX (with sign)
.text:00401AC7 idiv    ecx                             ; Signed Divide
.text:00401AC9 mov     [ebp+Variable_Zaehler_Byte], edx
.text:00401ACC mov     eax, [ebp+FileHandle_Projekt20.ids]
.text:00401ACF mov     [esp+818h+var_818], eax         ; filestream
.text:00401AD2 call    fgetc                           ; Zeichen aus IDB lesen
.text:00401AD7 mov     [ebp+VariableAusIdbAusgelesenesZeichen], al
.text:00401ADA mov     eax, [ebp+FileHandle_Projekt20.ids]
.text:00401ADD mov     eax, [eax+0Ch]
.text:00401AE0 and     eax, 10h                        ; Logical AND
.text:00401AE3 test    eax, eax                        ; Logical Compare
.text:00401AE5 jnz     short loc_401B08                ; Jump if Not Zero (ZF=0)
.text:00401AE7 mov     eax, [ebp+Variable_Zaehler_Byte]
.text:00401AEA add     eax, offset Offset_SchluesselArray ; Add
.text:00401AEF mov     al, [eax]
.text:00401AF1 xor     al, [ebp+VariableAusIdbAusgelesenesZeichen] ; Verschlüsselung
.text:00401AF1                                         ; mit Key aus dem Schlüssel-Array
.text:00401AF4 movzx   eax, al                         ; Move with Zero-Extend
.text:00401AF7 mov     edx, [ebp+var_18]
.text:00401AFA mov     [esp+818h+var_814], edx         ; filestream
.text:00401AFE mov     [esp+818h+var_818], eax         ; zeichen
.text:00401B01 call    fputc                           ; Zeichen nach IDS schreiben
.text:00401B06 jmp     short ZeichenVonIdbNachIds      ; Jump
```

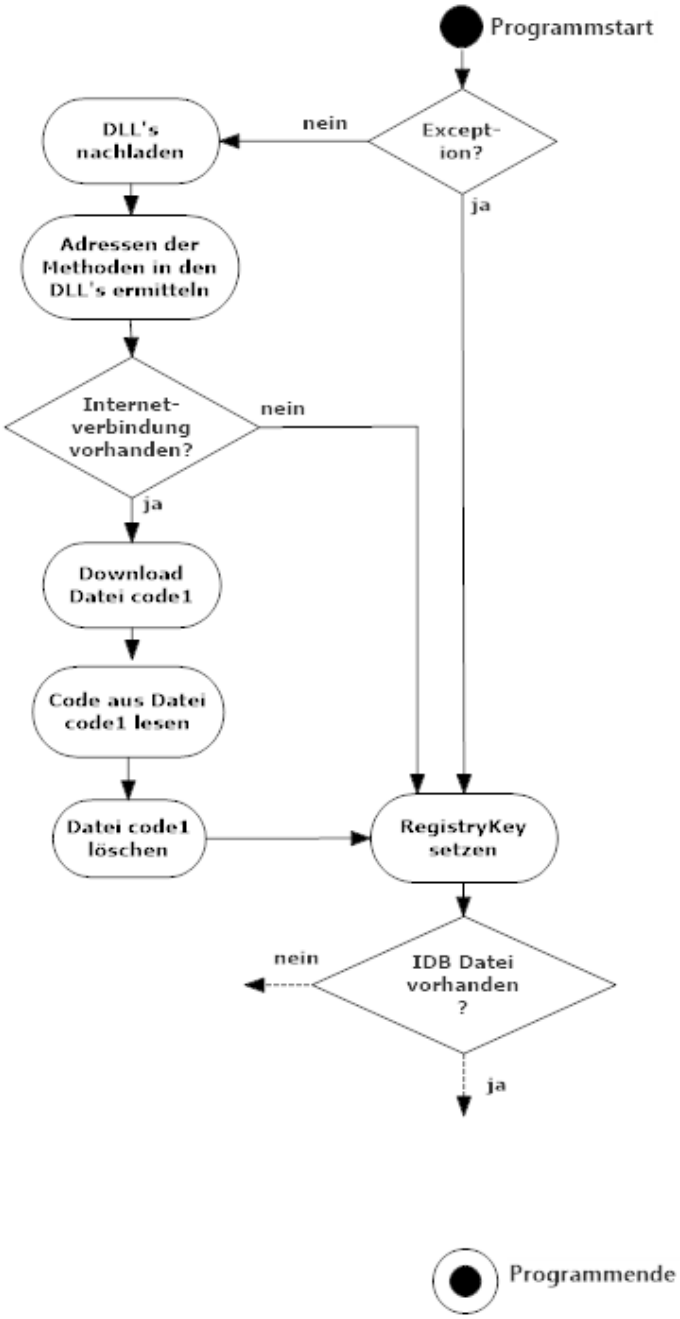
Aufbau der IDS Datei



1.3.8.2 Download der Datei code1.dat

Wird die Exception beim Programmstart nicht ausgeführt (Register ECX = 0xFFFFFFF0), dann wird die Datei ***code1.dat*** von https://faui1-videos.informatik.uni-erlangen.de/videos/public/Master-Online/M15-M110/M110-2016 heruntergeladen.

Anschließend werden die ersten 9 Zeichen herausgelesen und die Datei wieder gelöscht. Der Code und die eingelesenen Zeichen haben für den Deaktivierungscode der Malware keine Bedeutung.



Flowchart 2: Download Datei code1.dat

1.3.8.3 Modifikation der Registry

Beim Programmstart der Malware wird der folgende Key in der Registry eingetragen:

Schlüssel:
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Key:
_SRV_CTRL_ cmd.exe /c start /min "C:\M110\Aufgabe\Stoll\projekt20.exe"

Dieser Eintrag bewirkt, dass bei jedem Start von Windows die Malware in einer DOS Command Box minimalisiert ausgeführt wird. Vom Malwareprogrammierer war es aber sicher das Ziel, dass die Malware gestartet wird. Dies wird aber verhindert da der Pfad der auszuführenden Datei zwischen Anführungszeichen steht und Windows 7 dies nicht interpretiert.

Der RegistryKey wird wieder entfernt, wenn die Malware nicht im Debugger läuft und der korrekte Deaktivierungscode eingegeben wurde.

Zur Modifikation der Registry hat der Malwareprogrammierer die Funktion ***RegistryBearbeiten()*** programmiert. Die Funktion hat einen Integer Wert als Parameter. Über diesen lässt sich steuern ob der Registry Eintrag erstellt, abgefragt oder gelöscht wird. Der Aufruf der Funktion, mit dem Parameterwert 3 zum Löschen des Registry Eintrags, steht nach dem entschlüsseln der IDA Datei. Die Funktion wird aber nur aufgerufen, wenn die Malware nicht im Debugger läuft.

1.4 Zusammenfassung und Fazit

1.4.1 Was haben Sie gelernt?

Das praktische Arbeiten mit IDA und das Erkennen von Malwaretechniken.

1.4.2 Was war schwierig?

Schwierig war für mich das Erkennen der Obfuskierungen. Ich hatte mich sehr lange auf die Obfuskierung Structured Exception Handler und dem Download und Auswertung der Datei cod1.dat eingelassen.

Zusätzlich war es für mich sehr schwierig zu erkennen, wann die Analyse fertig ist. Bei einem Entwicklungsauftrag gibt es eine fest definierte Spezifikation des Funktionsumfangs. Dadurch gibt es auch ein definiertes Ende der Arbeit.

Beim Reverse-Engineering hatte ich keinen definierten Endpunkt der Analyse. Ich war mir nie sicher ob es immer noch etwas Wichtiges in der Malware gibt, dass noch nicht analysiert wurde.

1.4.3 Was war positiv? Was war negativ?

Der Lernerfolg über die eingesetzten Malwaretechniken war sehr gut. Dies ist auch nur über das Arbeiten mit IDA möglich, nicht durch die theoretische Vorlesung. Daher ist das Modul richtig aufgebaut.

Der Erfolg ging aber nur über einen großen Zeitaufwand. Dies war für ein zeitintensives Modul. Aber auch das Modul mit dem größten Lernerfolg!

1.5 Literaturverzeichnis

[1)] Dr. rer. nat. Werner Massonne, Prof. Dr.-Ing. Felix C. Freiling, Studienbrief Reverse Engineering

[2)] https://msdn.microsoft.com/de-de

[3)] https://de.wikipedia.org/wiki/Wikipedia:Hauptseite