



Zertifikatsprogramm - Z103

Reverse Engineering

- Einführung
- Assemblerprogrammierung
- Windows-Strukturen, Formate und Prozesse
- Codeerzeugung und Programmanalyse
- Tools zur Programmanalyse
- Malware und Malwareanalyse

Dr. rer. nat. Werner Massonne

Dr. rer. nat. Werner Massonne

werner.massonne@cs.fau.de

+49 (0) 9131 / 85 - 69916

Modul Z103

Reverse Engineering

Studienbrief 1: Einführung

Studienbrief 2: Assemblerprogrammierung

Studienbrief 3: Windows - Strukturen, Formate und Prozesse

Studienbrief 4: Codeerzeugung und Programmanalyse

Studienbrief 5: Tools zur Programmanalyse

Studienbrief 6: Malware und Malwareanalyse

Autor:

Dr. rer. nat. Werner Massonne

5. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

© 2021 Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

5. Auflage (25. Mai 2021)

Didaktische und redaktionelle Bearbeitung:
Romy Rahnfeld, Patrick Eisoldt

Das Werk einschließlich seiner Teile ist urheberrechtlich geschützt. Jede Verwendung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verfasser unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Inhaltsverzeichnis

Einleitung zu den Studienbriefen	6
I. Abkürzungen der Randsymbole und Farbkodierungen	6
II. Zu den Autoren	7
III. Modullehrgziele	8
Studienbrief 1 Einführung	9
1.1 Lernergebnisse	9
1.2 Advance Organizer	9
1.3 Was ist Reverse Engineering?	9
1.4 Wozu Reverse Engineering?	11
1.5 Grenzen und Schwierigkeiten von Reverse Engineering	13
1.5.1 Code vs. Daten	13
1.5.2 Selbstmodifizierender Code	15
1.5.3 Idiome	15
1.5.4 Generierter Code	16
1.5.5 Verschleierung (Obfuscation)	16
1.6 Zusammenfassung	16
1.7 Übungen	18
Studienbrief 2 Assemblerprogrammierung	19
2.1 Lernergebnisse	19
2.2 Advance Organizer	19
2.3 Assembler und Rechnerarchitektur	19
2.3.1 Von-Neumann-Architektur	21
2.3.2 Programmausführung auf einer Von-Neumann-Architektur	22
2.4 Die Prozessorarchitektur IA-32	24
2.4.1 Register	26
2.4.2 Instruktionssatz	31
Datentransferbefehle	32
Arithmetische und logische Befehle	33
Shift- und Rotationsbefehle	33
Kontrolltransferbefehle	37
Speicher und Adressierung	41
Subroutinen	45
Speicherverwaltung	52
String-Befehle	60
Sonstige Befehle	62
Interrupts und Exceptions	63
Maschinenbefehlsformat	64
2.5 Zusammenfassung	65
2.6 Übungen	66
Studienbrief 3 Windows - Strukturen, Formate und Prozesse	69
3.1 Lernergebnisse	69
3.2 Advance Organizer	69
3.3 Einführung	69
3.4 Anwendungen und Bibliotheken	71
3.5 Windows API	72
3.6 Systemaufrufe	75
3.7 Das PE-Format	75
3.7.1 DOS Header	77
3.7.2 PE Header	78
3.7.3 Section Table	81

3.7.4	Export Directory	81
3.7.5	Import Directory	83
3.8	Windows-Prozesse	86
3.8.1	PEB und TEB	87
3.9	Exceptions	90
3.10	Zusammenfassung	92
3.11	Übungen	93
Studienbrief 4 Codeerzeugung und Programmanalyse		97
4.1	Lernergebnisse	97
4.2	Advance Organizer	97
4.3	Einführung	97
4.4	Compiler, Interpreter, Linker	98
4.5	Dekompilierung	99
4.5.1	Kontrollflussstrukturen	100
	Bedingte Anweisung	100
	Verzweigung	101
	Mehrfachauswahl	102
	Schleifen	104
	Funktionsaufrufe	106
4.5.2	Datenstrukturen	107
	Elementare Datentypen	108
	Felder (Arrays)	108
	Strukturen und Verbunde	110
4.6	Optimierung	111
4.6.1	Propagierung von Konstanten	112
4.6.2	Dead Code Elimination	112
4.6.3	Inlining	113
4.6.4	Schleifenoptimierung	114
	Unswitching	114
	Loop Unrolling	115
	Loop Inversion	116
4.6.5	Kontrollflussoptimierung	117
	Branch-to-Branch Elimination	117
	Sub-Expression Elimination	118
	Branchless Code	119
4.6.6	Idiome	119
4.6.7	Frame Pointer Omission	121
4.7	Zusammenfassung	121
4.8	Übungen	123
Studienbrief 5 Tools zur Programmanalyse		127
5.1	Lernergebnisse	127
5.2	Advance Organizer	127
5.3	Arten des Reverse Engineering	127
5.4	Statische Analyse	128
5.4.1	Tools in der statischen Analyse	128
5.5	Statische Analyse mit IDA	130
5.5.1	Ein erster Eindruck	130
5.5.2	Navigation in IDA	133
5.5.3	Reverse Engineering mit IDA	135
5.6	Dynamische Analyse	140
5.6.1	Allgemeines zur dynamischen Analyse	141
5.6.2	OllyDbg	142
5.6.3	Dynamische Analyse mit IDA	149
5.6.4	IDA und OllyDbg im Zusammenspiel	150
	Die Geheimnummer	152

	Dynamische Änderung des Kontrollflusses	152
	Programmmanipulation	152
5.7	Zusammenfassung	153
5.8	Übungen	154
Studienbrief 6 Malware und Malwareanalyse		155
6.1	Lernergebnisse	155
6.2	Advance Organizer	155
6.3	Einführung	155
6.4	Obfuscation	156
6.4.1	String Obfuscation	157
6.4.2	Junk Code Insertion	158
6.4.3	Code Permutation	158
6.4.4	Opaque Predicates	159
	Bogus Control Flow / Dead Code	159
	Fake Loops	160
	Random Predicates	160
6.4.5	Merging	160
6.4.6	Splitting	160
6.4.7	Zahlen „verschleiern“	161
6.4.8	Aliasing	163
6.4.9	Pointer Indirection	163
6.4.10	Control Flow Flattening (Chenxify)	164
6.4.11	Import Hiding	164
6.4.12	Hashing	166
6.4.13	Structured Exception Handling	167
6.5	Verhinderung von Disassemblierung	168
6.5.1	Polymorphie, Metamorphie	168
6.5.2	Selbstmodifizierender Code	170
6.5.3	Unaligned Branches	171
6.5.4	Kontrollfluss-Obfuscation	172
6.6	Malware-Techniken	173
6.6.1	Virtuelle Maschinen	174
6.6.2	Packer	175
6.6.3	Anti-Unpacking	180
	Anti-Dumping	180
	VM-Obfuskatoren und Anti-Emulation	181
	Anti-Debugging	182
6.6.4	Erkennen Virtueller Maschinen	184
6.6.5	Malware Launching	185
6.6.6	Persistenz-Mechanismen	187
6.7	Analyse realer Malware	187
6.7.1	Automatische Malware-Analysetools	187
6.7.2	Fallbeispiel	188
6.8	Zusammenfassung	191
6.9	Übungen	192
Stichwörter		197
Liste der Lösungen zu den Kontrollaufgaben		201
Verzeichnisse		207
I.	Abbildungen	207
II.	Definitionen	209
III.	Exkurse	209
IV.	Tabellen	209
V.	Literatur	210

Einleitung zu den Studienbriefen**I. Abkürzungen der Randsymbole und Farbkodierungen**

Definition	D
Exkurs	E
Merksatz	M
Quelltext	Q
Übung	Ü

II. Zu den Autoren



Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Modulehrziele

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“. In diesem Modul werden Sie mit dem Thema „Reverse Engineering“ bei der Softwareanalyse vertraut gemacht.

In der Praxis hat man es standardmäßig mit Software zu tun, deren Quellcode nicht verfügbar ist, die also nur in Binärform vorliegt. Hierauf, insbesondere auf die 32-Bit-Intel-Architektur IA-32 der x86-Prozessorfamilie und das Windows-Betriebssystem, spezialisiert sich auch dieses Modul. Sie werden nach Abschluss dieses Moduls in der Lage sein, vom Umfang her überschaubare ausführbare Programme (Binaries) unter Windows mit entsprechenden Werkzeugen (im Wesentlichen IDA und OllyDbg) zu analysieren.

Zunächst wird Reverse Engineering definiert und in sein begriffliches Umfeld eingeordnet. Es wird aufgezeigt, welche Anwendungsfälle es für Reverse Engineering gibt. Es wird aber auch verdeutlicht, mit welchen Schwierigkeiten Reverse Engineering zu kämpfen hat und wo seine Grenzen liegen.

Im Mittelpunkt dieses Moduls steht die Binärcodeanalyse, also die Analyse von Maschinenprogrammen. Um diese verstehen zu können, müssen die hardwaretechnischen Details einer ausführenden Rechnerarchitektur bekannt sein. Ebenso muss der Reverse Engineer die Assemblerprogrammierung verinnerlicht haben. IA-32 ist die heute dominierende Plattform im Bereich der Arbeitsplatzrechner. In Studienbrief 2 wird IA-32 detailliert vorgestellt, und die Assemblerprogrammierung dieser Architektur wird intensiv erlernt.

In den Studienbriefen 3 und 4 werden die darüber hinausgehenden Grundlagen vermittelt, die ein Reverse Engineer beherrschen muss. Im Studienbrief 3 wird das Betriebssystem Windows näher betrachtet. Es werden die wesentlichen Elemente von Windows detailliert vorgestellt, die für eine Programmanalyse wichtig sind. Dieser Studienbrief soll Ihnen die Fähigkeit vermitteln, Ablauf und Aufbau von Programmen und Verwaltungsstrukturen unter Windows zu verstehen und zu analysieren. Kenntnisse zu den allgemeinen Grundlagen von Betriebssystemen werden hierbei vorausgesetzt.

Im anschließenden Studienbrief 4 wird gezeigt, welche Codesequenzen auf Maschinenebene bei der Kompilierung von Hochsprachenprogrammen entstehen. Damit werden Ihnen die für die Erkennung und Rekonstruktion von Hochsprachenkonstrukten in Assembler erforderlichen Fähigkeiten vermittelt. Darüber hinaus enthält der Studienbrief eine Übersicht über compilerseitige Optimierungsverfahren. Von einem Compiler optimierte Programme sind oftmals schwerer zu analysieren als nicht optimierte Programme.

Im Studienbrief 5 werden Ihnen Tools vorgestellt, die heute bei der Analyse von Maschinenprogrammen zum Einsatz kommen. Es werden sowohl die statische als auch die dynamische Analyse von Programmen an praktischen Beispielen vorgestellt. Nach dieser Einführung in die Benutzung der Tools sollten Sie selbstständig in der Lage sein, Reverse Engineering einfacher Programme durchzuführen.

Der abschließende Studienbrief 6 widmet sich einem wesentlichen Einsatzgebiet von Reverse Engineering, der Analyse von Malware. Es werden die Charakteristika und Besonderheiten von Malware gezeigt und die Schutzmaßnahmen, die Malware benutzt, um sich gegen ihre Analyse zu schützen. Zum Schluss wird anhand eines Fallbeispiels real existierende Malware exemplarisch analysiert.

Das Material dieses Moduls entstammt zum Teil einer Vorlesung, die im Sommersemester 2010 von Felix Freiling, Carsten Willems und Ralf Hund an der Universität Mannheim gehalten wurde [Freiling et al., 2010] und von Thorsten Holz [Holz, 2012] an der Ruhr-Universität Bochum fortgeführt wurde. Das Material wurde grundsätzlich überarbeitet und ergänzt. Der Autor dankt Felix Freiling, Thorsten Holz, Carsten Willems und Ralf Hund sowie den Studierenden der Vorlesungen für die ausgezeichnete Unterstützung.

Studienbrief 1 Einführung

1.1 Lernergebnisse

Sie können den Begriff *Reverse Engineering* definieren und gegenüber verwandten Begriffen abgrenzen und einordnen. Sie können die wichtigsten Anwendungsgebiete von Reverse Engineering nennen und begründen, warum Reverse Engineering dort jeweils wichtig ist. Sie können begründen, warum Reverse Engineering Grenzen grundsätzlicher Art gesetzt sind.

1.2 Advance Organizer

(Software) Reverse Engineering beschäftigt sich allgemein mit der Analyse von Programmen und hat das Ziel, diese zu verstehen und daraus die Intention von Programmen und ihrer Autoren abzuleiten. Im engeren Sinne steht Reverse Engineering für Binärcodeanalyse, also für die Analyse von Programmen, die nur im Maschinencode eines Rechners vorliegen. In der digitalen Forensik ist die Untersuchung unbekannter Schadsoftware (Malware) ein wichtiges Thema. Ein *Reverse Engineer* hat hier die Aufgabe, die Funktionsweise der Malware zu ergründen.

Reverse Engineering zur Analyse von Maschinenprogrammen erfordert Basiswissen bzgl. Rechnerarchitektur und Betriebssystemen. Dieses Basiswissen setzen wir teilweise voraus. In diesem Modul beschäftigen wir uns mit weiterführenden Themen und Techniken zur Programmanalyse auf realen Rechnern. Der einleitende Studienbrief führt zum Thema „Reverse Engineering“ hin. Reverse Engineering wird genauer definiert und eingeordnet. Die typischen Einsatzgebiete werden aufgezeigt, es werden aber auch einige prinzipielle Grenzen von Reverse Engineering herausgearbeitet.

1.3 Was ist Reverse Engineering?

Allgemein gesprochen möchte man beim Reverse Engineering verstehen, wie ein gegebenes System funktioniert. Mit „System“ meint man nicht unbedingt nur eine Software, sondern es kann ein beliebiges technisches oder soziales System sein. Genauer gesagt bezeichnet Reverse Engineering allgemein ein Verfahren zur Analyse eines bestehenden Systems mit den folgenden Zielen:

1. Identifikation der Systemkomponenten und ihrer wechselseitigen Abhängigkeiten und Beziehungen.
2. Die Generierung einer Repräsentation des Systems in einer anderen Form oder auf einem höheren Abstraktionslevel.

Beide Punkte spiegeln wider, dass es sich um Verfahren handelt, die ein tieferes Verständnis für das untersuchte System herstellen sollen: In der Regel werden Systeme arbeitsteilig erstellt, d. h. die Gesamtaufgabe wird in mehrere Teilaufgaben zerlegt, die getrennt voneinander in unterschiedlichen Systemkomponenten bearbeitet werden. Versteht man, welche Systemkomponenten es gibt und wie diese zusammenarbeiten, versteht man mehr über das System. In der Praxis ist es bspw. für das Verständnis der Funktionsweise einer Behörde oder Verwaltung sehr hilfreich, die inneren Abteilungs- oder Arbeitsgruppenstrukturen dieser Organisation zu kennen.

Analyse von Systemkomponenten

Der zweite Punkt bezieht sich auf den Nutzen von Abstraktion zum Verständnis von Systemen. Praktische Systeme müssen sich mit vielen Details befassen, die für das grobe Verständnis des Systems nicht wichtig sind. Für das Verständnis der

Abstraktion

inneren Funktionsweise einer Behörde oder Verwaltung ist es bspw. unerheblich, welche konkreten Personen etwa in der Auftragsbearbeitung oder der Finanzbuchhaltung sitzen. Viel ist gewonnen, wenn man die Abteilungsstruktur dieser Organisation herausgefunden hat. Die Abteilungsstruktur ist eine Repräsentation der Organisation auf einer anderen Abstraktionsebene. Beide Ziele sind also nicht überdeckungsfrei.

Reverse Engineering
vs. Re-Engineering

Reverse Engineering hat nicht zum Ziel, ein bestehendes System z. B. aus kommerziellen Gründen nachzubauen. Ziel von Reverse Engineering ist es vielmehr, das bestehende System zu *verstehen*. Wenn das primäre Ziel darin besteht, ein gegebenes System nachzubauen, spricht man von *Re-Engineering*. Das Verständnis eines „re-engineerten“ Systems spielt dabei eine untergeordnete Rolle, im Vordergrund steht die Kopie des Originals.

Die Begriffe Reverse Engineering und Re-Engineering entstammen keineswegs der Computertechnik, sondern finden allgemein Verwendung in unterschiedlichsten Bereichen.

B

Beispiel 1.1

Wenn ein bestimmtes Automodell eines bestimmten Herstellers quasi form- und funktionsgleich von einem Konkurrenten auf dem Markt erscheint, dann liegt der Verdacht nahe, dass es sich um einen Fall von Re-Engineering handeln könnte. Da es sicher nicht so leicht möglich ist, ein Automobil 1:1 nachzubauen, ohne zuvor ein tieferes Verständnis der technischen Systeme des Originals gewonnen zu haben, waren dazu sicherlich auch jede Menge Techniken von Reverse Engineering im Sinne der obigen Definition nötig.

Forward Engineering
und Reverse Engineering

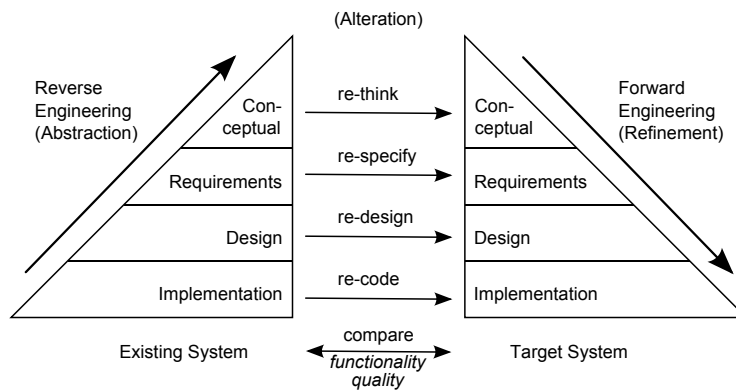
Reverse Engineering ist das Gegenteil von *Forward Engineering*. Forward Engineering ist der traditionelle Entwicklungsprozess von einem hohen logischen und implementierungsunabhängigen Abstraktionslevel ausgehend hin zu einer konkreten (physikalischen) Implementierung eines Systems [Chikofsky and Cross, 1990].

Software Reverse
Engineering

Mit dem Begriff *Reverse Engineering* meinen wir in diesem Modul immer *Software Reverse Engineering*, also die Analyse von Computerprogrammen. In diesem Kontext kennt man eher den Begriff *Software Engineering* (deutsch auch *Software-Technik*). Damit meint man aber in der Regel *Software Forward Engineering*, also den strukturierten Prozess zur Entwicklung von Software aus natürlichsprachlich formulierten Anforderungen, der schließlich über einen Entwurf hin zu einer konkreten Implementierung führt. Abb. 1.1 zeigt auf der rechten Seite die prinzipiellen Schritte beim Software Engineering vom Konzept hin zur Implementierung. Auf der linken Seite sind die korrespondierenden Schritte beim Software Reverse Engineering dargestellt.

Die Person, die Software Engineering betreibt (deutsch auch der *Software-Ingenieur*), wird heute in der Regel seine Programme in einer Hochsprache wie C oder Java implementieren und diese von einem automatischen Übersetzer (Compiler) in sogenannten Maschinen- oder Binärcode umwandeln lassen. Dieser Maschinencode kommt letztendlich auf einem Computer zur Ausführung. Abb. 1.2 zeigt die Vorgänge etwas detaillierter. Im Folgenden gehen wir auf die einzelnen Schritte ein.

Abb. 1.1: Software Engineering und Reverse Engineering [Byrne, 1990]



Definition 1.1: Dekompilierung und Disassemblierung

Dekompilierung ist die Rückübersetzung eines Maschinencodeprogramms in eine Hochsprache.

Disassemblierung ist die Umwandlung eines Maschinencodeprogramms in eine symbolische Schreibweise, den Assemblercode. Wenn der Programm-einstiegspunkt gegeben ist, so ist diese Umwandlung bei reinem Programm-code eindeutig und automatisierbar.¹

D

Der technisch anspruchsvollste Schritt beim Software Reverse Engineering ist die Dekompilierung. Deshalb liegt auch der Fokus dieses Moduls auf diesem Schritt. Wir betrachten die Grundlagen und Methoden, um ausführbare Programme zu dekompile und um so zu verstehen, was diese tun.

Die Dekompilierung eines Maschinenprogramms führt im Idealfall wieder zum Originalprogramm. Abb. 1.3 zeigt das Szenario: Ausgangspunkt ist Programm A in einer Hochsprache wie C, das durch einen Compiler in ein ausführbares Programm B übersetzt wird. Ein Decompiler nimmt nun B als Eingabe und erzeugt wieder ein Programm A' in einer Hochsprache. Idealerweise stimmen A und A' überein. Dieses Idealziel ist aber aus verschiedenen Gründen automatisiert meist nicht erreichbar. Die Mindestanforderung an eine erfolgreiche Dekompilierung von B ist etwas schwächer als das Idealziel. Es reicht bereits aus, dass die erneute Kompilierung von A' wieder – semantisch, nicht buchstäblich – Maschinenprogramm B ergibt.

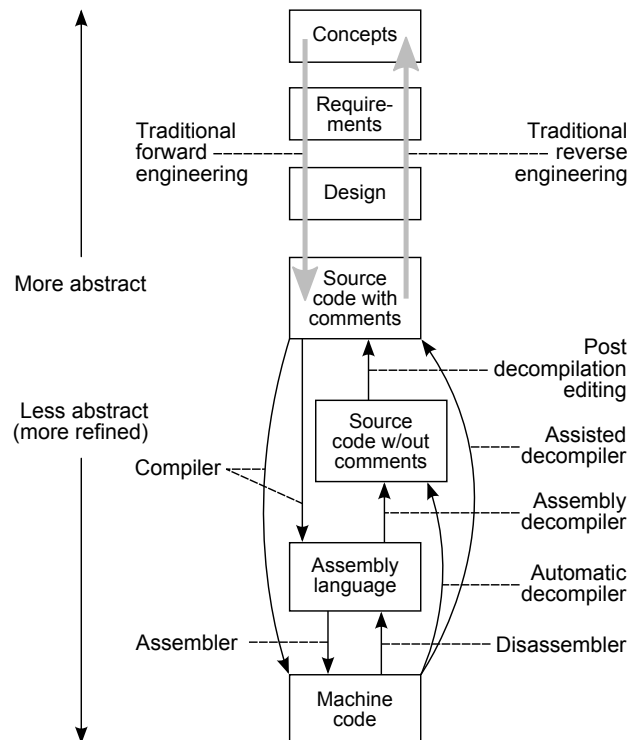
1.4 Wozu Reverse Engineering?

Die Haupteinsatzgebiete von Reverse Engineering sind:

- Software-Wartung
- Software-Analyse
- Malware-Analyse

¹ Keine Regel ohne Ausnahmen: In Studienbrief 6 werden Verfahren vorgestellt, die eine automatische Disassemblierung behindern. Dies ist aber nur unter bestimmten Voraussetzungen möglich.

Abb. 1.2: Detaillierte Darstellung des Übersetzungsprozesses und der entsprechenden Schritte beim Reverse Engineering [van Emmerik, 2012]



Software-Wartung	Das klassische Anwendungsgebiet von Reverse Engineering ist die Software-Wartung. Bei vielen gewachsenen Software-Systemen existiert heute häufig nur noch der Binärcode. Quellcode oder kommentierter Assemblercode fehlen vollständig, und die Programmierer der Software haben das Unternehmen längst verlassen. Wenn es nun erforderlich wird, solche Programme zu verstehen, um eventuelle Fehler zu beseitigen, bleibt nichts anderes übrig, als die Programme zu analysieren und selbst herauszufinden, was sich die Programmierer dabei dachten. Ein globales Problem dieser Art entstand bspw. beim Jahrtausendwechsel und ist als Jahr-2000-Problem oder Millenium-Bug bekannt. ² Ein anderes Beispiel ist die nachträgliche Beseitigung von Sicherheitslücken.
Software-Analyse	Neben der Software-Wartung ist die <i>Software-Analyse</i> ein wichtiges Einsatzgebiet von Reverse Engineering. Software enthält oftmals wichtige Informationen, die finanziell sehr wertvoll sein können, z. B. Registrierungsschlüssel, Passwörter oder Forschungsergebnisse in Form von Algorithmen. Beispielsweise könnten legal erworbene Registrierungsschlüssel in Papierform verloren gegangen sein, während sie innerhalb eines Programms noch vorhanden sind. Mit Mitteln von Reverse Engineering können sie möglicherweise wieder zum Vorschein gebracht werden.
Malware-Analyse	Das dritte und immer mehr an Bedeutung gewinnende Einsatzgebiet von Reverse Engineering ist die <i>Malware-Analyse</i> . Als Malware (Schadsoftware) werden Programme bezeichnet, die einem Computer und damit dem Betreiber des Computers Schaden zufügen. Hiermit ist keineswegs nur ein direkter materieller oder finanzieller Schaden gemeint, sondern auch ein indirekter Schaden, der bspw. durch den zeitlichen Aufwand zur Neuinstallation eines infizierten PC entsteht. Die Malware-Analyse ist ein wichtiger Bestandteil der digitalen Forensik. Bei der digitalen Forensik stehen Straftaten im Vordergrund, die mittels Software begangen wurden, oder bei denen ein Programmverhalten eine wichtige Rolle gespielt haben

² Viele alte Programme berücksichtigten nur die beiden letzten Ziffern der Jahreszahl und hätten sich ohne Korrektüreingriffe im Jahr 1900 statt im Jahr 2000 wiedergefunden. Eine kurze Abhandlung dazu findet man unter „<http://de.wikipedia.org/wiki/Jahr-2000-Problem>“.

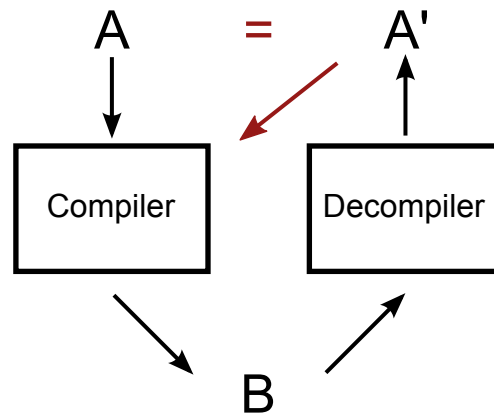


Abb. 1.3: Kompilierung
vs. Dekompilierung.

könnte. Eine forensische Untersuchung solcher Delikte gliedert sich in die Bereiche Beweismittelsicherung und Beweismittelanalyse, was letztendlich eine Analyse von Programmen erfordert.

Im Laufe der forensischen Untersuchung eines Programms werden häufig Fragen der folgenden Art aufgeworfen, die mit Hilfe von Reverse Engineering beantwortet werden können:

1. Welche Veränderungen hat ein Programm auf einem Rechner möglicherweise vorgenommen?
2. Mit welchen Rechnern kann ein Programm potenziell kommunizieren?
3. Welche Daten liest ein Programm, während es auf einem Rechner läuft?
4. Wer hat ein bestimmtes Programm geschrieben?

1.5 Grenzen und Schwierigkeiten von Reverse Engineering

Reverse Engineering bzw. Dekompilierung ist kein vollständig automatisierbarer Prozess, führt also i. Allg. nicht zu einem eindeutigen Resultat. Dies liegt einmal an immanenten Schwierigkeiten, die dem Verfahren grundsätzliche Probleme bereiten, und zum anderen an vielen „Fallstricken“, die die Analyse – teilweise vom Programmierer beabsichtigt – erschweren. Einige dieser Grenzen und Probleme werden im Folgenden näher beschrieben und auch später immer wieder in den Studienbriefen aufgegriffen.

1.5.1 Code vs. Daten

Moderne Rechner folgen dem Konzept der Von-Neumann-Architektur. Im Speicher werden Codesequenzen, also Programme und Daten nicht grundsätzlich getrennt voneinander gehalten. Eine unbekannte Datei, die zur Ausführung in den Hauptspeicher des Rechners geladen wird, besteht zunächst nur aus Bitmustern. Ein solches Bitmuster ist genau dann Programmcode, wenn es aus gültigen Maschinenbefehlen besteht, und wenn diese Befehle auch irgendwann zur Ausführung kommen.

Auch Zufallsmuster im Speicher enthalten gültige Maschinenbefehle, die mit entsprechenden Tools aufgespürt werden können. Die entscheidende Frage ist also, ob ein erkannter Maschinenbefehl jemals zur Ausführung kommt.

B**Beispiel 1.2**

Betrachten Sie die folgenden Programme, die ein Dekompilierungstool aus Bitmustern im Hauptspeicher gewonnen haben könnte:

Programm A:

```
01 int i = 50
02 if (i < 1) goto 5
03 i = i - 1
04 goto 2
05 i = 42
```

Programm B:

```
01 int i = 50
02 if (i < 1) goto 5
03 i = i / 1
04 goto 2
05 i = 42
```

Wie leicht zu erkennen ist, wird Zeile 5 bei Programm A irgendwann ausgeführt, da Variable *i* fortlaufend dekrementiert wird. Bei Programm B ist dies nicht der Fall. Demnach ist Zeile 5 bei Programm A Code, bei Programm B jedoch nicht und möglicherweise nur ein bei der Dekompilierung fälschlicherweise als Code fehlinterpretiertes Datum.

So einfach wie im Beispiel gezeigt, ist es jedoch in der Realität leider nicht zu entscheiden, ob ein Programmbefehl jemals zur Ausführung kommt oder nicht. Im Allgemeinfall ist dieses Problem sogar unentscheidbar, weil es auf das *Halteproblem* reduziert werden kann.

E**Exkurs 1.1: Halteproblem**

Gesucht ist ein Programm *H* mit folgendem Verhalten:

- Eingabe: beliebiges Programm *P*, beliebige Eingabe *E*
- Ausgabe:
 - **true**, falls *P* angewandt auf *E* hält (terminiert), und
 - **false**, falls *P* angewandt auf *E* nicht hält.

Ein wichtiges Resultat der theoretischen Informatik besagt nun: Programm *H* kann es nicht geben, d. h. das Halteproblem ist nicht berechenbar! Das bedeutet, dass es nicht allgemein entscheidbar ist, ob ein beliebiges Programm mit einer beliebigen Eingabe jemals beendet wird. Demnach ist auch nicht allgemein entscheidbar, ob eine beliebige Programmzeile jemals zur Ausführung kommt.

Wir greifen das Halteproblem erneut auf und schlagen die Brücke zum Problem, Code und Daten zu unterscheiden. Betrachten Sie dieses kleine Programmfragment in C-Notation:

```
while (Bedingung) {
    // Schleifenrumpf
}
AnweisungX
```

Falls die Schleife terminiert, ist AnweisungX Code, ansonsten ist AnweisungX kein Code. Das Halteproblem besagt nun, dass die Terminierung der Schleife nicht allgemein entscheidbar ist. Dies bedeutet natürlich im Umkehrschluss nicht, dass die Terminierung im Einzelfall nicht voraussagbar wäre. Meistens wird sie sogar voraussagbar sein, aber eben nicht immer. Die Folgerungen aus dem Halteproblem führen zu der allgemeinen und aus Sicht von Reverse Engineering bedauerlichen Schlussfolgerung, dass nicht allgemein entscheidbar ist, ob eine Speicherzelle Code oder Daten enthält.

1.5.2 Selbstmodifizierender Code

Eine ganz besondere Herausforderung stellt die Analyse von selbstmodifizierendem Code dar. Selbstmodifizierender Code ist wiederum eine direkte Folge des Von-Neumann-Prinzips. Da Daten- und Codebereiche nicht klar voneinander getrennt sind, ist es möglich, dass ein Programm zur Laufzeit Daten in den eigenen Codebereich schreibt, die dann von der CPU als Programmcode interpretiert und ausgeführt werden. Die Veränderung des Programms kann dabei von den Programmeingaben abhängig sein, wodurch wieder direkt gefolgert werden kann, dass selbstmodifizierender Code nicht allgemein analysierbar ist.

Beispiel 1.3

Ein kleines Beispiel für selbstmodifizierenden Code in x86-Assembler (nach [Cifuentes, 1994, S. 3]):

```
. . .          ; other code
mov [inst], E920 ; E9 = jmp, 0x20 = offset
. . .
inst nop        ; 0x90
nop             ; 0x90
```

Der Befehl `mov` schreibt den Wert `0xE920` an Adresse `inst`. Dies verändert die `nop`-Befehle (*no operation* mit Opcode `0x90`) in einen unbedingten Sprung mit Offset `0x20` (`jmp 0x20` mit Opcode `0xE920`).

B

Kontrollaufgabe 1.1

Durch welche technische Maßnahme wäre die Ausführung von selbstmodifizierendem Code leicht zu verhindern?

K

1.5.3 Idiome

Als *Idiome* bezeichnet man Programmierkniffe, bei denen Instruktionen für einen anderen als den offensichtlichen Zweck verwendet werden. Beispiele dafür sind:

1. Links-Shift einer Binärzahl um 1 Bit entspricht einer Multiplikation mit 2.
2. Addition zweier Doppelworte durch folgende Befehlskette:
 - Addition der beiden niederwertigen Worte.
 - Addition der beiden höherwertigen Worte unter Berücksichtigung des Übertrags aus der ersten Addition.

Diese beiden Beispiele werden in Studienbrief 4 im Abschnitt 4.6.6 detaillierter betrachtet. Es gibt sehr viele solcher Idiome. Zum richtigen Erkennen trägt im Wesentlichen der Erfahrungsschatz des Reverse Engineer bei.

1.5.4 Generierter Code

Der von einem Compiler generierte Code enthält Teile, die der Verfasser eines Hochsprachenprogramms nicht explizit ausprogrammiert hat, d. h. es gibt Codestücke im Assemblerprogramm, die keine direkte Entsprechung in der Hochsprache haben. Beispiele hierfür sind:

1. Startup-Code, der Bibliotheken einbindet und die Laufzeitumgebung initialisiert. Dies wird in Studienbrief 3 näher behandelt.
2. Code zur Rettung von Prozessorregistern vor dem Sprung in eine Subroutine. Details dazu sind im Studienbrief 4 zu finden.

Generierter Code hängt stark vom verwendeten Compiler und der Systemumgebung ab. Um ihn zu erkennen, benötigt man etwas Erfahrung mit den entsprechenden Werkzeugen und Systemen.

1.5.5 Verschleierung (Obfuscation)

Die Methode der *Verschleierung* (engl. Obfuscation) verfolgt die Absicht, einen Code komplizierter zu machen, um die Analyse zu erschweren. Die Absicht des Programmiers ist leicht nachzuvollziehen: Die eigentliche Semantik eines Programms soll vor dem menschlichen Betrachter verschleiert werden, um Reverse Engineering zu verhindern. Diese Methode wird vor allem in Schadsoftware angewandt, aber auch in vielen kommerziellen Programmen wird Verschleierung eingesetzt, um bspw. Firmengeheimnisse (Algorithmen) zu schützen oder einen Kopierschutz zu gewährleisten.

gepackte Malware

Ein praktisches Beispiel für den Einsatz von Verschleierungstechniken ist sogenannte *gepackte Malware*. Die Malware besteht dabei aus einem vorgeschalteten Entpacker, während sich der Kern der Malware (Payload) in einem gepackten und möglicherweise verschlüsselten „Container“ befindet. Bei der Ausführung der Malware wird zuerst der Entpacker ausgeführt, also der Payload entpackt und entschlüsselt. Die eigentliche Schadsoftware, d. h. das eigentliche schädliche Binärprogramm der Malware, existiert nur zur Laufzeit. Die Analyse der Schadsoftware vor der Entpackung wird durch den Packer und die Verschlüsselung nahezu unmöglich gemacht. Demzufolge ist die Programmanalyse nur zur Laufzeit möglich. In Studienbrief 5 werden Werkzeuge zur dynamischen Laufzeitanalyse vorgestellt. Mit Obfuscation-Verfahren beschäftigen wir uns im ersten Teil von Studienbrief 6.

K

Kontrollaufgabe 1.2

Lässt Obfuscation immer auf Schadsoftware schließen?

1.6 Zusammenfassung

Reverse Engineering ist ein wesentlicher Bestandteil der Analyse und Wartung von Software. Von zentraler Bedeutung ist dabei die Dekompilierung von Programmen in Maschinencode, also die Rückübersetzung in den Quellcode einer Hochsprache.

Maschinencode ist „unleserlich“ und damit sehr schwer zu verstehen. Liegt jedoch der Quellcode vor, so ist die Chance groß, ein Programm in seiner Gesamtheit zu erfassen, um so auch Ziele und Absichten des Programms bzw. des Programmierers nachzuvollziehen.

Die Dekompilierung von Maschinencode ist ein sehr komplexer Vorgang. Zum einen sind der Dekompilierung grundsätzlich Grenzen gesetzt, und zum anderen sind hierfür tiefgehende Kenntnisse über den Aufbau und die Funktionsweise von Rechnern und Programmen auf einem Hardware-nahen Level erforderlich.

1.7 Übungen

Ü

Übung 1.1

Beschreiben Sie den Unterschied zwischen „Reverse Engineering“ und „Re-Engineering“. Recherchieren und beschreiben Sie ein Beispiel für Re-Engineering aus der Geschichte.

Hinweis: Typische Beispiele finden Sie im Automobilbereich (z. B. BMW X5 vs. CEO) oder in anderen Hochtechnologiebereichen (z. B. Space Shuttle vs. Raumfähre Buran).

Ü

Übung 1.2

Beschreiben Sie im Detail die Schritte von Software Engineering und die korrespondierenden Schritte von Software Reverse Engineering.

Ü

Übung 1.3

Was ist der Unterschied zwischen Dekompilierung und Disassemblierung. Was von beiden ist schwieriger und warum ist das so?

Ü

Übung 1.4

Erklären Sie die Haupteinsatzgebiete von Reverse Engineering. Versuchen Sie, jeweils praktische Beispiele zu finden.

Ü

Übung 1.5

Als Dead Code bezeichnet man Teile eines Programms, die niemals zur Ausführung kommen können. Ein Programm kann damit angereichert sein, um den Analysten aufs Glatteis zu führen, weil er unter Umständen sehr viel Zeit in die Analyse von sinnlosen Codesequenzen investiert. Kann es ein allgemeines Verfahren geben, das Dead Code komplett entfernt? Begründen Sie Ihre Antwort möglichst genau.

Stichwörter

Abstraktion, 9

Forward Engineering, 10

gepackte Malware, 16

Idiom, 15

Malware-Analyse, 12

Obfuscation, 16

Re-Engineering, 10

Software Engineering, 10

Software Reverse Engineering, 10

Software-Analyse, 12

Software-Ingenieur, 10

Software-Wartung, 12

Liste der Lösungen zu den Kontrollaufgaben

Lösung zu Kontrollaufgabe 1.1 auf Seite 15

Durch strikte Trennung von Programm- und Datenspeicher. Diese Variante der Von-Neumann-Architektur wird auch als Harvard-Architektur bezeichnet. Bei einer Harvard-Architektur kann der Programmspeicher zur Laufzeit nicht manipuliert werden. Zusätzlich muss natürlich unterbunden werden, dass Programmcode in den Datenspeicher geschrieben und dort ausgeführt wird. Reine Harvard-Architekturen werden vor allem im Bereich von Mikrocontrollern und Embedded Systems eingesetzt. Ihre Nachteile sind die erhöhten Hardware-Kosten (doppelte Bussysteme) sowie weniger Flexibilität.

Lösung zu Kontrollaufgabe 1.2 auf Seite 16

Nein, keineswegs. Obfuscation ist primär reiner Softwareschutz und dient dazu, die Funktionsweise eines Programms zu verschleiern. Hersteller kommerzieller Software setzen bspw. Obfuscation ein, um die Umgehung eines Kopierschutzes zu verhindern.

Verzeichnisse

I. Abbildungen

Abb. 1.1:	Software Engineering und Reverse Engineering [Byrne, 1990]	11
Abb. 1.2:	Detaillierte Darstellung des Übersetzungsprozesses und der entsprechenden Schritte beim Reverse Engineering [van Emmerik, 2012]	12
Abb. 1.3:	Kompilierung vs. Dekompilierung.	13
Abb. 2.1:	Von-Neumann-Architektur	21
Abb. 2.2:	Befehlsausführung bei der Von-Neumann-Architektur [Wisman, 2012]	23
Abb. 2.3:	Intel 80386-CPU	25
Abb. 2.4:	Registersatz des 80386	26
Abb. 2.5:	Registeraufteilung beim 80386	26
Abb. 2.6:	Flags der 80386-CPU [Intel, 1987]	28
Abb. 2.7:	Shift- und Rotationsbefehle	36
Abb. 2.8:	Endianness	41
Abb. 2.9:	Offset-Adressierung im Überblick	44
Abb. 2.10:	Auf- und Abbau eines Stack Frame	48
Abb. 2.11:	Aufbau des Stack Frame nach dem Prolog	49
Abb. 2.12:	Implementierung der Aufrufkonventionen in Assembler	52
Abb. 2.13:	Privilegienringe	53
Abb. 2.14:	Logischer, linearer und physikalischer Adressraum	54
Abb. 2.15:	Physikalischer Speicher eines Intel PC [Duarte, 2008]	54
Abb. 2.16:	Segmentierung im Protected Mode [Intel, 2012]	56
Abb. 2.17:	MMU: Pages und Frames [Freiling et al., 2010]	58
Abb. 2.18:	Paging (80386) bei 4 KB großen Pages [Intel, 2012]	58
Abb. 2.19:	Adress- Umwandlung mit Segmentierung UND Paging [Intel, 2012]	60
Abb. 2.20:	Aufbau der Interrupt Descriptor Table	64
Abb. 2.21:	IA-32-Befehlsformat [Intel, 2012]	64
Abb. 2.22:	ModRM und SIB [Freiling et al., 2010]	65
Abb. 2.23:	Befehlsformat	66
Abb. 3.1:	Logischer Aufbau von Windows [Rusinovich and Solomon, 2012]	70
Abb. 3.2:	Statische und dynamische Bindung	72
Abb. 3.3:	DLL-Hierarchie	73
Abb. 3.4:	Übergang in den Kernelmode	75
Abb. 3.5:	Alignment in Datei und Speicher	76
Abb. 3.6:	Offset, virtuelle Adresse und relative virtuelle Adresse	76
Abb. 3.7:	PE-Format	77
Abb. 3.8:	DOS Header	78
Abb. 3.9:	Aufbau PE Header [Holz, 2012]	79
Abb. 3.10:	Export Directory mit EAT, ENT und EOT	82
Abb. 3.11:	EAT, ENT und EOT von <i>kernel32.dll</i>	83
Abb. 3.12:	Import Directory	84
Abb. 3.13:	Vor dem Aufbau der IAT	85
Abb. 3.14:	Nach dem Aufbau der IAT	85
Abb. 3.15:	Prozesskomponenten (Prozessrahmen)	86
Abb. 3.16:	Prozess im Speicher	87
Abb. 3.17:	TEB/PEB	88
Abb. 3.18:	Anfang des TEB	88
Abb. 3.19:	PEB	89
Abb. 3.20:	Structured Exception Handling (SEH)	91
Abb. 3.21:	Implementierung SEH	91
Abb. 3.22:	Frame Based Exception Handling (FBEH)	92
Abb. 4.1:	Hochsprachen, ein zeitlicher Abriss	98
Abb. 4.2:	Übersetzung eines Hochsprachenprogramms	99

Abb. 4.3:	Kompilieren und Reversing	100
Abb. 4.4:	Bedingte Anweisung	101
Abb. 4.5:	Verzweigung	101
Abb. 4.6:	Mehrfach-Verzweigung	102
Abb. 4.7:	Zusammengesetzte Bedingung	102
Abb. 4.8:	Mehrfachauswahl mit Switch Table	103
Abb. 4.9:	Switch Table mit Lücken	104
Abb. 4.10:	Mehrfachauswahl über Binärbaum	104
Abb. 4.11:	Mehrfachauswahl: Baum	105
Abb. 4.12:	<i>While-do</i> -Schleife	105
Abb. 4.13:	<i>Do-while</i> -Schleife	105
Abb. 4.14:	Schleifenvariante mit <i>continue</i> und <i>break</i>	106
Abb. 4.15:	<i>For</i> -Schleife	106
Abb. 4.16:	Funktionsaufruf	107
Abb. 4.17:	Lokales Array	109
Abb. 4.18:	Globales Array	110
Abb. 4.19:	Globale und lokale Struktur	111
Abb. 4.20:	Zeiger auf eine Struktur	111
Abb. 4.21:	Verbund	112
Abb. 4.22:	Übersetzung ohne Inlining	114
Abb. 4.23:	Übersetzung mit Inlining	114
Abb. 4.24:	Branchless Code	119
Abb. 5.1:	PEview Übersicht	130
Abb. 5.2:	Import Directory	130
Abb. 5.3:	IAT von <i>hostname.exe</i>	131
Abb. 5.4:	Unterstützte Binärformate	132
Abb. 5.5:	Das Programm aus Beispiel 5.2, disassembliert in IDA	132
Abb. 5.6:	Der Call Graph von <i>calc.exe</i>	133
Abb. 5.7:	IDA-Codebalken	134
Abb. 5.8:	Imports von <i>hostname.exe</i>	135
Abb. 5.9:	Disassembly von <i>hostname.exe</i>	136
Abb. 5.10:	Aufruf von <i>gethostname</i> in <i>hostname.exe</i>	136
Abb. 5.11:	Wesentlicher Teil von <i>rechnen1.exe</i>	138
Abb. 5.12:	Flow Graph mit Umbenennungen	140
Abb. 5.13:	Die Menüleiste von OllyDbg	143
Abb. 5.14:	OllyDbg - die Fenster im Überblick	143
Abb. 5.15:	Abgelegte Werte bleiben häufig auch nach Auflösen des Stack Frame eine Zeit lang auf dem Stack erhalten	144
Abb. 5.16:	Die Rücksprungadresse zu der Stelle im Code, die für das Ausgeben von Text verantwortlich ist, befindet sich im ungültigen Bereich des Stack und wurde nicht überschrieben.	145
Abb. 5.17:	Die Stelle im Code von <i>cmd.exe</i> , an der das Passwort an die API-Funktion zur Ausgabe in der Konsole übergeben wird	145
Abb. 5.18:	Nachdem der Breakpoint gesetzt wurde, stoppt OllyDbg das Programm im richtigen Moment, um den gesuchten Wert abzulesen.	145
Abb. 5.19:	Die beiden Pfade nach der Ausgabe des Passworts	146
Abb. 5.20:	Der (Default-)Pfad von <i>echo</i> nach Eingabe beliebiger Zeichen	147
Abb. 5.21:	Der Alternativpfad für die Fehlerbehandlung	148
Abb. 5.22:	Nach Verändern des Kontrollflusses wird statt der gewöhnlichen Ausgabe die Fehlerroutine aktiv.	149
Abb. 5.23:	Register, Flags und Referenzen	150
Abb. 5.24:	Call Graph von <i>crackme2.exe</i>	151
Abb. 5.25:	Wesentlicher Codeabschnitt von <i>crackme2.exe</i>	152
Abb. 6.1:	Obfuskator	157
Abb. 6.2:	Merging von Funktionen	161
Abb. 6.3:	Splitting von Funktionen	161
Abb. 6.4:	Pointer Indirection	164
Abb. 6.5:	Control Flow Flattening	164
Abb. 6.6:	IAT ohne aufgelöste Importinformationen	165

Abb. 6.7: IAT mit aufgelösten Importinformationen	166
Abb. 6.8: Sichtbare Imports	166
Abb. 6.9: Versteckten von Imports	166
Abb. 6.10: Unsichtbare Imports	167
Abb. 6.11: Shellcode zur Manipulation des SEH	168
Abb. 6.12: Polymorphie [Holz, 2012]	169
Abb. 6.13: Packen	176
Abb. 6.14: Entpacken bei UPX	177
Abb. 6.15: UPX - Anfang des Entpacker-Stub	178
Abb. 6.16: UPX - Sprung zum OEP	179
Abb. 6.17: Code Injection	186
Abb. 6.18: Polymorpher Shellcode	192
Abb. 19: Kontrollaufgabe 2.3	202
Abb. 20: Kontrollaufgabe 2.5	203

II. Definitionen

Definition 1.1: Dekompilierung und Disassemblierung	11
Definition 2.1: Assembler als Programmiersprache	19
Definition 2.2: Assembler als Übersetzer	20

III. Exkurse

Exkurs 1.1: Halteproblem	14
Exkurs 2.1: System-Flags	30
Exkurs 2.2: Zweierkomplementdarstellung	38
Exkurs 2.3: Befehlssynonyme	40
Exkurs 3.1: Manuelle Exportsuche	83
Exkurs 4.1: Dead Code Elimination	113
Exkurs 4.2: Pipelining	116
Exkurs 5.1: Nicht erkannte Codebereiche	134
Exkurs 5.2: Detours	149
Exkurs 6.1: Verschlüsselungsverfahren bei Malware	170
Exkurs 6.2: VirtualBox	175
Exkurs 6.3: Gepackte DLLs	180

IV. Literatur

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. *16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

E. J. Byrne. A conceptual foundation for software re-engineering. *ICSM 1992*, pages 226–235, 1990.

E. Chikofsky and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7(1), pages 13–17, 1990.

Cristina Cifuentes. *Reverse Compilation Techniques*. Doktorarbeit, Queensland University of Technology, Australien, 1994.

Gustavo Duarte. *Motherboard Chipsets and the Memory Map*.
<http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map>, 2008.

Chris Eagle. *The IDA Pro Book*. No Starch Press, 2008.

E. Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2005.

Peter Ferrie. Anti-unpacker tricks.

<http://pferrie.tripod.com/papers/unpackers.pdf>, 2008.

Felix Freiling, Ralf Hund, and Carsten Willems. *Software Reverse Engineering*. Vorlesung, Universität Mannheim, 2010.

Thorsten Holz. *Program Analysis*. Vorlesung, Ruhr-Universität Bochum, 2012.

Intel. *Intel 80386, Programmers Reference Manual*.

<http://css.csail.mit.edu/6.858/2011/readings/i386.pdf>, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>, 2012.

Kip R. Irvine. *Assembly Language for Intel-Based Computers (5th Edition)*. Prentice Hall, 2006.

Tobias Klein. *Aus dem Tagebuch eines Bughunters. Wie man Softwareschwachstellen aufspürt und behebt*. dpunkt Verlag, 2010.

Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. *SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

Microsoft. Microsoft pe and coff specification.

<http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx>, 2010.

Gary Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, 2000.

Matt Pietrek. An in-depth look into the win32 portable executable file format. *February 2002 issue of MSDN Magazine*, 2002.

<http://msdn.microsoft.com/en-us/magazine/bb985992.aspx>.

Joachim Rohde. *Assembler GE-PACKT, 2. Auflage*. Redline GmbH, Heidelberg, 2007.

Rolf Rolles. Binary literacy – optimizations.

<http://www.openrce.org/repositories/users/RolfRolles/Binary%20Literacy%20-%20Static%20-%206%20-%20Optimizations.ppt>, 2007.

Rolf Rolles. Unpacking virtualization obfuscators. *WOOT'09 Proceedings of the 3rd USENIX conference on Offensive technologies*, 2009.

M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft Press Corp, 2012.

Justin Seitz. *Hacking mit Python: Fehlersuche, Programmanalyse, Reverse Engineering*. dpunkt Verlag, 2009.

Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

Sebastian Strohäcker. Malicious code: Code obfuscation.

www.reverse-engineering.info/0BF/strohhaecker.pdf, 2004.

Mike van Emmerik. Decompilation and reverse engineering.

<http://www.program-transformation.org/Main/MikeVanEmmerik>, 2012.

Carsten Willems and Felix Freiling. Reverse code engineering - state of the art and countermeasures. *Oldenburg Wissenschaftsverlag*, 2012.

Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 2007.

Ray Wisman. *Processor Architecture*.

<http://homepages.ius.edu/RWISMAN/C335/HTML/Chapter2.htm>, 2012.

Fort- und Weiterbildung

Neue Bedrohungsszenarien stellen Sicherheitsexperten und IT-Verantwortliche in Unternehmen und einschlägigen Behörden vor immer größere Herausforderungen. Neue Technologien und Anwendungen erfordern zusätzliches Know-how und personelle Ressourcen.

Zur Erhöhung des Fachkräftepools und um neues Forschungswissen schnell in die Praxis zu integrieren, haben sich die im Bereich lehrenden und forschenden Verbundpartner zum Ziel gesetzt, ein hochschuloffenes transdisziplinäres Weiterbildungsprogramm im Sektor Cyber Security zu entwickeln. Auf der Grundlage kooperativer Strukturen werden wissenschaftliche Weiterbildungsmodul im Verbund zu hochschulübergreifenden Modulpaketen und abschlussorientierten Ausbildungslinien konzipiert und im laufenden Studienbetrieb empirisch getestet.

Die Initiative soll High Potentials mit und ohne formale Hochschulzugangsberechtigung über innovative Weiterbildungsangebote (vom Zertifikat bis zum Masterprogramm) zu Sicherheitsexperten aus- und fortbilden. Hierzu werden innovative sektorale Lösungen zur Optimierung der Durchlässigkeit von beruflicher und hochschulischer Bildung entwickelt und für eine erfolgreiche Implementierung vorbereitet. Unter prominenter Beteiligung einschlägiger Verbände, der Industrie sowie Sicherheits und Ermittlungsbehörden verfolgt die Initiative das Ziel, im deutschsprachigen Raum eine Generation von Fachkräften wissenschaftlich aus- und weiterzubilden, die unser Internet schützen kann.

Open Competence Center for Cyber Security

Open C³S ist aus dem Verbundvorhabens Open Competence Center for Cyber Security entstanden. Das Gesamtziel des Programms war die Entwicklung eines hochschuloffenen transdisziplinären Programms wissenschaftlicher Weiterbildung im Sektor Cyber Security. Das Bundesministerium für Bildung und Forschung (BMBF) fördert das Großprojekt im Rahmen des Wettbewerbs „Aufstieg durch Bildung: offene Hochschulen“, der aus BMBF-Mitteln und dem Europäischen Sozialfonds finanziert wird.

Neun in Forschung und Lehre renommierte Hochschulen und Universitäten aus dem gesamten Bundesgebiet haben sich zum Ziel gesetzt, Online-Studiengänge auf dem Gebiet der Cybersicherheit zu entwickeln. Dieses Konzept soll den Studierenden ermöglichen, sich berufsbegeleitend auf hohem Niveau wissenschaftliche Qualifikationen anzueignen und akademische Abschlüsse zu erlangen. Beruflich erworbene Kompetenzen können eingebracht werden. Die Bezeichnung „Open“ steht auch für die Öffnung des Zugangs zu akademischer Bildung ohne klassischen Hochschulzugang.

Mission der Initiative ist es, dringend benötigte Sicherheitsexperten aus- und fortzubilden, um mit einer sicheren IT-Infrastruktur die Informationsgesellschaft in Deutschland und darüber hinaus zu stärken.

Umsetzungsnahes Wissen ist ein wesentlicher Schlüssel um der wachsenden digitalen Bedrohung zu begegnen. Solange wir nicht in der Lage sind, Systeme hinreichend zu härten, Netzwerke sicher zu designen und Software sicher zu entwickeln, bleiben wir anfällig für kriminelle Aktivitäten. Unser Ziel ist es, die Mitarbeiter von heute zu Sicherheitsexperten und Führungskräften von morgen auszubilden und dafür zu sorgen, dass sich die Zahl und die Fertigkeiten dieser Experten nachhaltig erhöht.

Z103 Reverse Engineering

Reverse Engineering ist ein wichtiger Teilbereich der digitalen Forensik und beschäftigt sich mit der Analyse unbekannter Software, deren Funktionsweise analysiert werden soll. Die zu untersuchende Software liegt dabei in der Regel nur in Form von Maschinenprogrammen vor. 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“. In diesem Modul werden Sie mit dem Thema „Reverse Engineering“ bei der Softwareanalyse vertraut gemacht.

In der Praxis hat man es standardmäßig mit Software zu tun, deren Quellcode nicht verfügbar ist, die also nur in Binärform vorliegt. Hierauf, insbesondere auf die 32-Bit-Intel-Architektur IA-32 der x86-Prozessorfamilie und das Windows-Betriebssystem, spezialisiert sich auch dieses Modul. Sie werden nach Abschluss dieses Moduls in der Lage sein, einfache ausführbare Programme unter Windows mit entsprechenden Werkzeugen (im Wesentlichen IDA und OllyDbg) zu analysieren.

Zunächst wird Reverse Engineering definiert und in sein begriffliches Umfeld eingeordnet. Es wird aufgezeigt, welche Anwendungsfälle es für Reverse Engineering gibt. Es wird aber auch verdeutlicht, mit welchen Schwierigkeiten Reverse Engineering zu kämpfen hat und wo seine Grenzen liegen.

Im Mittelpunkt dieses Moduls steht die Binärcodeanalyse, also die Analyse von Maschinenprogrammen. Um diese verstehen zu können, müssen die hardwaretechnischen Details einer ausführenden Rechnerarchitektur bekannt sein. Ebenso muss der Reverse Engineer die Assemblerprogrammierung verinnerlicht haben. IA-32 ist die heute dominierende Plattform im Bereich der Arbeitsplatzrechner. Dieses Modul baut auf dem Modul „Systemnahe Programmierung“ auf, in dem IA-32 und ihre Programmierung auf Assemblerebene sehr eingehend behandelt werden.

Zertifikatsprogramm

Die Zertifikatsmodule auf wissenschaftlichem Niveau und mit hohem Praxisbezug bilden ein passgenaues Angebot an Qualifikation und Spezialisierung in der nebenberuflichen Weiterbildung. Damit können einzelne Module nebenberuflich studiert werden. Durch die Vergabe von ECTS-Punkten können sie auf ein Studium angerechnet werden.

<https://zertifikatsprogramm.de>