Hauptseminar Malicious Code: Viren, Würmer und Co.

Statische Programmanalyse

Matthias Langhammer

Technische Universität München,langhamm@in.tum.de

Zusammenfassung Um ihre Existenz zu verschleiern enthalten moderne Viren neben den eigentlichen Schadroutinen auch noch Methoden, die ein automatisches Auffinden durch Antivirensoftware verhindern sollen. Dieser Artikel befasst sich mit Möglichkeiten der statischen Analyse von ausführbaren Binärdateien, um diese Verschleierungen zu erkennen und unwirksam zu machen.

1 Einleitung

Seitdem es Virenscanner gibt, versuchen Virenauthoren, ihre Werke möglichst unauffällig zu verbergen. Dazu steht ihnen eine Vielzahl von Möglichkeiten zur Verfügung, den eigentlichen Sinn oder die Präsenz ihres Codes zu verschleiern. Meist wird dazu der gefährliche Code so modifiziert, daß er durch reines Vergleichen mit dem Original nicht mehr wiedererkannt werden kann. Es existieren aber auch noch aufwändigere „code-obfuscations“, bei denen der Viruscode mit dem Code der befallenen Datei vermischt wird, oder sogar durch Kryptographie unkenntlich gemacht wird.

Die folgenden Abschnitte führen einige Verfahren zur Erkennung solcher Verschleierungen auf. Alle diese Verfahren basieren auf statischer Analyse von Binärdateien mit maschinenausführbarem Inhalt, d.h. die Dateien werden gelesen und auf schädlichen Code überprüft, ohne sie auszuführen.

Alle Verfahren, bis auf das in Abschnitt 5 vorgestellte, gehören zum Standardrepertoir aktueller Virenscanner. Die Beispiele wurden zum Teil aus [1] übernommen.

2 Signaturbasierte Virensuche

Die einfachste Art der statischen Programmanalyse ist die Suche nach Virensignaturen. Eine solche Signatur besteht im Allgemeinen aus einem Teil des Binärcodes des Virus. Ein Virendetektor, der auf Signaturen basiert, vergleicht eine zu prüfende Datei mit allen ihm bekannten Signaturen. Findet er eine Übereinstimmung, deutet dies auf eine verseuchte Datei hin.

Um mit der Signatursuche befriedigende Ergebnisse zu erzielen, wird eine möglichst vollständige Datenbank aller bekannter Viren und deren Signaturen

benötigt. Das größte Problem dabei ist, daß selbst minimale Veränderungen an dem schadhaften Code die Signatur des Virus verändern, wodurch ein weiterer Eintrag in der Signaturdatenbank erforderlich wird. Daher finden sich in diesen Datenbanken oft zig Versionen des gleichen Virus, die sich jeweils nur um wenige Bytes unterscheiden.

Zur Bestimmung der Signatur eines bestimmten Codefragments liest man diese lediglich an dem Assemblercode in Hexadezimaldarstellung ab. Listing 1.1 zeigt ein Fragment des unveränderten Chernobyl/CIH-Virus mit den zugehörigen hexadezimalen Byte-Sequenzen.

```
call 0h                 ;E800 0000 00
pop ebx                 ;5B
lea ecx, [ebx + 42h]    ;8D4B 42
push ecx                ;51
push eax                ;50
push eax                ;50
sidt [esp - 02h]        ;0F01 4C24 FE
pop ebx                 ;5B
add ebx, 1Ch            ;83C3 1C
cli                     ;FA
mov ebp, [ebx]          ;8B2B
```

Listing 1.1. Codefragment des originalen Chernobyl/CIH Virus

Daraus ergibt sich beispielsweise die in Listing 1.2 abgebildete Signatur.

```
E800 0000 005B 8D4B 4251 5050 0F01 4C24
FE5B 83C3 1CFA 8B2B
```

Listing 1.2. Signatur des Original Chernobyl/CIH Virus

Allerdings ist eine Analyse einer ausführbaren Datei, basierend auf der Suche nach einer solchen Signatur, leicht durch simpelste Tricks der „code obfuscation“angreifbar. Beispielsweise genügt es schon, zwischen den eigentlichen Viruscode unnütze Befehle einzustreuen (dead-code-insertion, trash-insertion, NOP-insertion), um ihn für Detektoren unkenntlich zu machen.

Listing 1.3 zeigt eine modifizierte Variante des Chernobyl-Codes aus Listing 1.1, wobei an 3 Stellen der Befehl NOP eingefügt wurde (NOP-insertion), um den Virus zu tarnen.

```
call 0h                 ;E800 0000 00
pop ebx                 ;5B
lea ecx, [ebx + 42h]    ;8D4B 42
nop                     ;90
push ecx                ;51
push eax                ;50
```

```
push eax              ;50
nop                   ;90
sidt [esp - 02h]      ;0F01 4C24 FE
pop ebx               ;5B
add ebx, 1Ch          ;83C3 1C
nop                   ;90
cli                   ;FA
mov ebp, [ebx]        ;8B2B
```

Listing 1.3. Code des Chernobyl/CIH, verschleiert durch NOP-insertion

Obwohl nur sehr geringe Änderungen vorgenommen wurden und das Programm semantisch vollkommen unverändert geblieben ist, hat sich die Signatur des Code verändert. Listing 1.4 zeigt die neue Signatur des veränderten Chernobyl-Fragments aus Listing 1.3.

```
E800 0000 005B 8D4B 4290 5150 5090 0F01
4C24 FE5B 83C3 1C90 FA8B 2B
```

Listing 1.4. Neue Signatur des veränderten Chernobyl/CIH Virus

Mit der ursprünglichen Signatur aus Listing 1.2 könnte man diesen veränderten Virus nicht finden, da die Signaturen nicht übereinstimmen. Kommt eine so modifizierte Variante in Umlauf, wird es also nötig, die neue Signatur auch in die Datenbank aufzunehmen, oder robustere Suchverfahren zu verwenden.

Damit die Rate der falschen positiven Treffer nicht zu groß wird, sollte man darauf achten, die Länge der zu suchenden Signaturen nicht zu kurz zu wählen.

3 Analyse mit Regulären Ausdrücke

In Abschnitt 2 wurde gezeigt, daß die Binary-Analyse mittels Signatursuche leicht ausgetrickst werden kann. Es reicht also oft nicht aus, nach festen Signaturen zu suchen.

Bessere Ergebnisse erzielt man, indem man nach regulären Ausdrücken sucht. Im einfachsten Fall entspricht ein solcher regulärer Ausdruck genau der Signatur. Will man nun z.B. gegen das Einfügen beliebig vieler NOP-Befehle (NOP-insertion) gewappnet sein, verändert man den Ausdruck derart, daß zwischen den Instruktionen auch noch beliebig viele NOPs akzeptiert werden.

Aus der Signatur in Listing 1.2 würde man also einen regulären Ausdruck wie den in Listing 1.5 erzeugen, um den Detektor robust gegen solche Angriffe zu machen.

```
E800      0000      00(90)* 5B(90)* 8D4B      42(90)*
51(90)* 50(90)* 50(90)* 0F01      4C24      FE(90)*
5B(90)* 83C3      1C(90)* FA(90)* 8B2B
```

Listing 1.5. Gegen NOP-insertion robuster regulärer Ausdruck, der den Chernobyl/CIH Virus beschreibt

Außer trash-insertion kann die Analyse mit regulären Ausdrücken aber auch eingeschränkt Codesequenzen erkennen, die mit den folgenden Methoden der „code–obfuscation“unkenntlich gemacht wurden:

1. Instruction-Substitution: Opcode-Sequenzen werden durch äquivalente Sequenzen ersetzt. Besonders der aufgeblähte x86-Befehlssatz mit seinen vielen Redundanzen bietet hier viel Freiraum. Listing 1.6 zeigt einen exemplarischen regulären Ausdruck, der ein paar sehr einfache Fälle von Befehlsersetzungen erkennt.

```
(MOV EAX, 0)       | (SUB EAX, EAX)        | \
(XOR EAX, EAX)  | (PUSH 0 ; POP EAX)
```

Listing 1.6. Beispiel eines regulären Ausdrucks, der einen sehr eingeschränkten Fall von Instruction-Substitution erkennt

2. „weaving“: Verweben / Vermischen des Viruscode mit dem Programmcode. Dabei wird der Virus in kleine Codeblöcke zerstückelt, die entweder auf ungenutzte Stellen in einer Binärdatei (z.B. entstehen manchmal Alignment-Lücken beim Linken zwischen Code aus verschiedenen Objekt-Dateien) verteilt, oder mit diesen aufwändiger vermischt werden. Danach müssen die einzelnen Blöcke nur noch mit angepassten Jump-Befehlen verbunden werden. Geschieht die Verteilung der Blöcke in fester aufsteigender Reihenfolge kann z.B. ein regulärer Ausdruck wie der in Listing 1.7 helfen.

```
<Sequenz_1> JMP .* <Sequenz_2> JMP .* <Sequenz_3>
```

Listing 1.7. Beispiel eines regulären Ausdrucks, der einen zerstückelten Programmabschnitt erkennen kann

3. Register-Reassignment: Soweit nicht Spezialregister vorausgesetzt werden, lassen sich viele Operationen auf beliebigen Registern ausführen. Vertauscht man geeignet die Registerbelegung, so bleibt der Viruscode semantisch intakt und voll funktionsfähig. Listing 1.8 zeigt einen Ausdruck, der einen solchen Fall von Register-Reassignment erkennt, allerdings zeigt dieser bereits, daß die hier reguläre Ausdrücke auch schnell an ihre Grenzen stoßen.

```
((MOV EAX, 5)+(SUB ECX, EAX)) | \
((MOV EBX, 5)+(SUB ECX, EBX))
```

Listing 1.8. Beispiel eines regulären Ausdrucks, der einen sehr eingeschränkten Fall von Register-Reassignment erkennt

Wie man schon an den Listings 1.6 und 1.8 erkennen kann, würde man eine Vielzahl komplizierter Ausdrücke benötigen, um zuverlässig alle erdenklichen Vertauschungen und Umordnungen zu erkennen. Außerdem können entsprechende Datenbanken nur schwer automatisch erzeugt, aktualisiert und auf Konsistenz geprüft werden, da das nötige Hintergrundwissen Maschinen meist verborgen bleibt.

4 Statische Heuristische Analysemethoden

Bei den vorangegangenen Methoden müssen die Beschreibungen der Viren ständig aktuell gehalten werden, um wirksamen Schutz zu bieten. Heuristische Analysemethoden versuchen nun, auch bisher noch unbekannte Viren zu entdecken, indem sie nach typischen Merkmalen oder Verhaltensmustern suchen. Solche Merkmale können z.B. verdächtige Befehle im Bootsektor eines Datenträgers sein. Die Suche nach solchen verdächtigen Mustern kann ihrerseits wieder mittels Signaturen oder regulären Ausdrücken implementiert werden.

Eine solche heuristische Binärdatenanalyse kann z.B. Code entdecken, der gebräuchliche Methoden des „Entry–Point–Obscuring“verwendet,wie z.B. das Nachladen weiterer Sektoren vom Bootsektor eines Datenträgers aus, oder untypische Sprünge in einem Binary, das ansonsten eindeutig einem Compiler zugeordnet werden kann. Denkbar wäre auch, Code von Dekomprimierungs- oder Decodierungsalgorithmen zu erkennen, der von gebräuchlichen Viren–Bau–Kits stammt.

Aber auch Code der speicherresidente Programmteile (TSR, Terminate and Stay Resident) hinterlässt, kann mit einer gewissen Treffsicherheit in Binärdateien aufgespürt und als verdächtig eingestuft werden.

Man sollte allerdings auch beachten, daß heuristische Verfahren zu Fehlalarmen (Fehler 1. Art) führen können. Routinen in Treibern und Binärdateien eines Betriebssystems können beispielsweise Befehlssequenzen beinhalten, die große Ähnlichkeit mit denen in Viren haben, was zu solchen falschen positiven Ergebnissen führen kann.

Außerdem bieten auch heuristische Algorithmen keine 100%-ige Garantie, jeden neuen Virus zu entdecken (Fehler 2. Art).

Es gibt viele weitere heuristische Analysemethoden, die meisten sind allerdings eher als dynamisch zu klassifizieren, und fallen deshalb aus dem Rahmen dieses Artikels. Detektoren dieser Klasse suchen z.B. regelmäßig den gesamten Hauptspeicher eines Computers nach verdächtigen Mustern (Signaturen oder reguläre Ausdrücke) ab, oder verwenden Sandboxing, um potentiell gefährlichen Code zuerst in einer gesicherten Umgebung auszuführen und zu prüfen, bevor er als „sicher“eingestuft wird.

Solche Runtime-Scans bieten dann aber auch eine gewisse Sicherheit gegen kryptographische Verschleierungen, da auch jeder verschlüsselte Virus vor seiner Vervielfältigung oder Triggerung irgendwann einmal im Klartext im Hauptspeicher eines befallenen Rechners erscheinen muss.

5 Semantische Analyse mit einem Malicous Code Automaton

Die bisher vorgestellten Verfahren zum Aufspüren von schädlichem Code in Binärdateien sind weit verbreitet und gehören zum Standardrepertoir beinahe aller gängigen Virenscanner. Doch wie man gesehen hat, sind sie alle relativ

einfach zu täuschen. Besonders mit Code-Transpositionen haben alle so ihre Probleme, weil dabei nicht einmal mehr sichergestellt ist, daß der schädliche Code in der ursprünglichen Reihenfolge bzw. überhaupt zusammenhängend ist.

Man braucht also ein Verfahren, welches mehr auf Semantik als auf Syntax des zu untersuchenden Codes achtet und möglichst robust gegen die bereits aufgezählten Code-Verschleierungen ist.

Ein solches System wurde 2003 unter dem Namen SAFE (Static Analyzer For Executables) [2] von Mihai Christodorescu und Somesh Jha an der University of Wisconsin entwickelt und mit gängigen Virenscannern verglichen [1].

Das System prüft die Semantik eines kompletten Programms, indem es dessen Kontrollflussgraphen (control flow graph, CFG) erzeugt und analysiert. Dazu wird dieser erst einmal aus dem Code einer binären ausführbaren Datei erzeugt. Im nächsten Schritt ist es die Aufgabe eines sog. Program-Annotators, den CFG Knotenweise mit Metacode zu kommentieren. Besonderes Augenmerk richtet der Annotator auf überflüssige Befehle und Sprünge, um Trash-Insertion und Code-Transpositionen zu erkennen. Außerdem ist es seine Aufgabe, Registernamen und Speicheradressen in abstrakte, typisierte Variablen zu überführen, um auch kompliziertere Fälle von Register-Reassignment zu erkennen. Während der Annotationsphase ist es auch möglich, mehrere Knoten des originalen CFG zu vereinen, was z.B. bei aufeinander folgenden NOP-Befehlen (NOP-slides) Sinn macht, da es genügt, die ganze Sequenz einfach mit einem Kommentar wie „Irrelevante Befehle"zu versehen.

Abbildung 1 zeigt, wie ein annotierter Kontrollflussgraph aussehen könnte. Der verwendete Metacode sollte eigentlich selbsterklärend sein, weitere Erklärungen würden den Ramen dieses Textes sprengen. Wer trotzdem mehr darüber wissen möchte, sei auf die ausführlichen Erklärungen in [1] verwiesen.

Zur Beschreibung einer schadhaften Routine dient bei SAFE nicht mehr ihre Signatur oder ein regulärer Ausdruck. Stattdessen kommt ein abstrakteres Modell zur Beschreibung der Routinen zum Einsatz. Und zwar wird ein sogenanntes Malicious-Code-Automaton (MCA) erzeugt, welches zum Auffinden der in ihm beschriebenen Codesequenz dient. Ein MCA A ist ein 6-Tupel $(V, \Sigma, S, \delta, S_0, F)$ mit folgenden Eigenschaften:

1. $V = \{v_1 : \tau_1, \ldots, v_k : \tau_k\}$ ist eine Menge typisierter Variablen v_i vom Typ τ_i
2. $\Sigma = \{\Gamma_1, \ldots, \Gamma_n\}$ ist ein endliches Alphabet aus abstrakten Mustern, die die Gültigkeitsbereiche von Variablen festlegen. Auf eine genauere Erklärung muß hier wegen dem Umfang verzichtet werden. Es sei wieder auf [1] verwiesen.
3. S ist eine endliche Menge von Zuständen
4. $d : S \times \Sigma \rightarrow 2^S$ ist eine Übergangsfunktion
5. $S_0 \subseteq S$ ist eine nichtleere Menge von Startzuständen
6. $F \subseteq S$ ist eine nichtleere Menge von Endzuständen

Abbildung 2 zeigt, wie ein solcher MCA graphisch veranschaulicht werden kann.

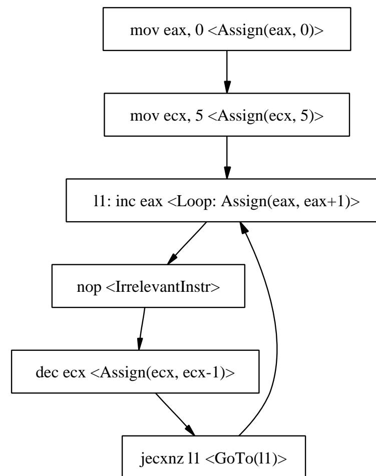

Abbildung 1. Beispiel eines annotierten CFG.

Eine solche abstrakte Virenbeschreibung gilt als gefunden, wenn eine zu untersuchende Binärdatei ein Wort der von dem Automaten akzeptierten Sprache enthält.

Das von Christodorescu und Jha entwickelte System braucht trotz seines noch recht frühen Stadiums den Vergleich mit kommerziellen Virenscannern nicht zu scheuen. In ihrer Ausarbeitung [1] haben sie mehrere kommerzielle Virenscanner und ihr eigenes Tool SAFE, das nach der hier beschriebenen Methode arbeitet, mit einigen weit verbreiteten, jedoch leicht modifizierten Viren getestet. Das Ergebnis war, daß alle kommerziellen Programme ausnahmslos versagten, jedoch ihr eigenes Tool alle Viren trotz Modifikationen erkannt hat. Die falsch-positiven und falsch-negativen Raten von 0.0 bei dem Tool SAFE verwundern in diesem Zusammenhang nicht weiter, da deren Tool ja genau auf die vorgenommenen Arten der code-obfuscation zugeschnitten waren.

Literatur

1. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium (Security'03), USENIX Association, USENIX Association (2003) 169–186
2. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Technical Report 1467, University of Wisconsin, Madison (2003)

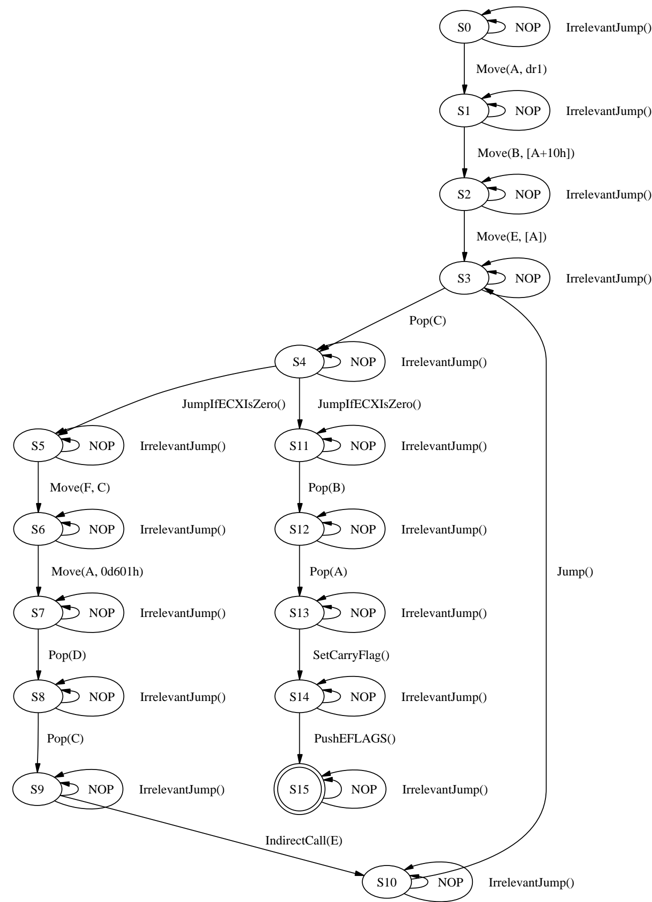

Abbildung 2. Ein Malicious-Code-Automaton