

Malicious Code : Code Obfuscation

Simon Bierbaum

bierbaum@in.tum.de

Abstract. Diese Arbeit gibt einen Einblick in die Code Obfuscation und beschreibt ihre Anwendung in der Konstruktion von metamorphen Viren. Es wird dargestellt, wie Viren ihre Erkennung durch automatische Virens Scanner mittels Code Obfuscation vermeiden können, und welche Methoden Virens Scannern zur De-Obfuscation zur Verfügung stehen. Nach einer Übersicht über relevante Code Obfuscation- und De-Obfuscation-Techniken werden jeweils zwei Ansätze detailliert dargestellt.

1. Einleitung

1.1. Code Obfuscation

Code Obfuscation (engl. „Verschleierung“, „Verwirrung“) bezeichnet die Transformation von Programmcode mit dem Ziel, die Ermittlung der Semantik und der Funktionalität zu erschweren, die wesentliche Funktionalität aber zu erhalten. *Code De-Obfuscation* bezeichnet den umgekehrten Vorgang.

Code Obfuscation kann auf zwei Bestandteile eines Programms angewandt werden: Auf den *Kontrollfluß*, also den chronologischen und kausalen Ablauf des Programms, und auf seine *Datenstrukturen*. Der Kontrollfluß eines Programms wird bestimmt durch Sprungpunkte und Verzweigungsstellen und gibt die Ausführungsreihenfolge der Anweisungen eines Programms an.

Code Obfuscation kann auf die Verwirrung von Menschen als auch Maschinen ausgelegt sein. Die Analyse eines Programms wird für einen Menschen durch andere Methoden erschwert als für Maschinen wie z.B. Virens Scanner oder automatische De-Obfuscation-Programme. Ein Maß für die Güte einer Obfuscation ist hierbei der Aufwand, der für die De-Obfuscation betrieben werden muß. Menschen lassen sich vor allem durch Transformationen verwirren, die die Lesbarkeit des Programms beeinträchtigen: Uneingängige Bezeichner, Mißachtung von Programmierkonventionen, uneinheitliche Formatierung des Quelltextes und Verteilung von logisch zusammenhängenden Strukturen über mehrere Dateien sind nur einige von zahllosen Beispielen, die im Internet von Programmierern humorvoll in einem langen Essay unter dem Titel „How to write unmaintainable code“ zusammengetragen wurden¹. Maschinen hingegen sind für die genannten Methoden,

¹ <http://mindprod.com/unmain.html>

2 Simon Bierbaum

die dem Menschen das Parsen von Quelltexten erschweren, unempfindlich. Dafür laufen sie auf Probleme auf, für deren Lösung der „gesunde Menschenverstand“ nötig ist. Ein menschlicher Analyst kann beispielsweise Annahmen über Eingaben in Programme machen, die bei der Entscheidung helfen, ob bestimmte Programmteile ausgeführt werden oder nicht. Verschiedene Methoden der Obfuscation sind gegen Menschen und Maschinen unterschiedlich widerstandsfähig.

1.2. Metamorphe Viren und Motivation

Code Obfuscation wurde Mitte der 90er Jahre für Computerviren interessant. Viren verhielten sich bis dahin statisch, Dateien wurden mit exakten Kopien von sich selbst infiziert. Somit war es für Virens Scanner leicht, durch einfachen Bitmustervergleich bekannte Viren aufzuspüren und zu eliminieren. Um dem zu entgehen, setzten findige Programmierer auf *Polymorphie* bzw. *Metamorphie*. Ein polymorpher Virus verändert sich mit jeder Kopie, die er von sich erzeugt. Die Güte dieses Morph-Vorgangs entscheidet darüber, wie gut ein Virens Scanner mittels Mustererkennung noch in der Lage ist, die Kopie als Kopie des ursprünglichen Virus zu erkennen. Polymorphe Viren erreichten dies z.B. durch Verschlüsselung ihres Körpers mit zufällig erzeugten Schlüsseln, wobei ein kleiner Teil, der zum Entschlüsseln nötig war, jeweils unverschlüsselt blieb. Letztendlich konnten Virens Scanner durch geringe Abstriche in der Genauigkeit der Erkennung auch diesen kleinen konstanten Teil sehr gut ausmachen. Metamorphe Viren transformieren sich komplett, so daß im Idealfall Virus und Kopie völlig unterschiedlich aussehen, ihre Funktionalität aber gleich ist. Code Obfuscation kann verwendet werden, um Metamorphie zu implementieren: Ein Virus erzeugt eine Kopie von sich selbst und wendet Code Obfuscation-Techniken auf die Kopie an, bevor sie in eine Zieldatei eingebaut wird.

1.3. Methoden der Code Obfuscation und der Virenerkennung

Für die Bewertung von Obfuscation- und De-Obfuscation-Techniken ist es hilfreich, gängige Methoden der Viruserkennung und generelle Möglichkeiten in der Code Obfuscation zu kennen. Die (In-)Effektivität der hier kurz vorgestellten Methoden wird nach und nach in dieser Arbeit ersichtlich werden.

Code Obfuscation umfaßt, wie bereits gesagt, eine Vielzahl von Methoden, wobei die meisten jedoch für die automatische De-Obfuscation kein besonderes Hindernis darstellen. Zu den potentiell interessanten Methoden zählt das Einfügen von zusätzlichem, möglicherweise totem Code und überflüssigen Variablen. Besonders das Aufblähen von Prädikaten, z.B. in if- oder while-Anweisungen, erhöht die Komplexität einer Analyse. *Inlining* und *Outlining* beschreiben das Auflösen bzw. Erzeugen von Unterprogrammaufrufen. Beim *Inlining* wird der Aufruf einer Funktion durch den Funktionskörper ersetzt, während beim *Outlining* ein beliebiges Codefragment in eine Prozedur ausgelagert wird. Schließlich können mehrere Variablen zu einem Datenkonstrukt zusammengefaßt oder größere Datenobjekte in mehrere kleine Teile aufgespaltet werden.

Für die Erkennung eines Virus mittels Mustervergleich reichen möglicherweise schon sehr kleine Stücke (mehrere Bytes) aus. Aktuelle eMail-Würmer tragen Dateipfade in sich, um bestimmte, auf dem befallenen System vorhandene Dateien zu finden. Auf Ebene der Maschinensprache kann sich ein Virus schon beim Suchen nach zu infizierenden Programmen verraten, wenn er die ersten Bytes eines Programms mit einer bestimmten Konstante vergleicht². Neben dem Mustervergleich spielen die Simulation und die statische Analyse eine zentrale Rolle bei der Erkennung von Viren. Bei einer Simulation wird der potentielle Virus in einer kontrollierten Umgebung Schritt für Schritt ausgeführt und das Verhalten beobachtet. Verhält er sich auffällig, z.B. durch das Ausführen von schädlichen Operationen, kann die Ausführung jederzeit abgebrochen werden. Bei der statischen Analyse hingegen wird der Code betrachtet, ohne ihn auszuführen. Basierend auf dem Wissen über die Semantik der Sprache und eventuell genutzter Funktionen können Schlussfolgerungen über das Verhalten und die Belegungen von Variablen an bestimmten Stellen im Programm gezogen werden.

Prinzipiell kann ein potentieller Virus immer erkannt werden, wenn genügend Zeit und Ressourcen zur Verfügung stehen. Deshalb beschränken sich die hier vorgestellten Maßnahmen auf die automatische Virenerkennung durch spezialisierte Software. Solche Virens Scanner können üblicherweise nicht auf menschliche Hilfe zurückgreifen und unterliegen engen Zeitschranken bei der Analyse – schließlich soll der Virenschutz möglichst unsichtbar arbeiten und den Benutzer nicht bei seiner Arbeit am Rechner stören.

Im Folgenden werde ich zwei Methoden der Code Obfuscation vorstellen, die für die Zwecke von metamorphen Viren gut geeignet erscheinen: Deckende Strukturen und virtuelle Maschinen. Weiterhin stelle ich mit der Klischeebasierten Analyse und dem Program Slicing zwei Analysetechniken vor, die die besprochene Obfuscation teilweise brechen können.

2. Deckende Strukturen

2.1. Definition und Motivation

Bei der statischen Analyse von Programmen spielen verzweigende Kontrollstrukturen eine entscheidende Rolle. Faßt man den Ablauf eines Programms als einen gerichteten Graphen auf, bei dem jeder Knoten einem Zustand des Programms entspricht (ein sog. Kontrollflussgraph), so werden durch verzweigende Kontrollstrukturen wie `if`, `while`, `for` usw. Verzweigungen im Graph erzeugt. Für die Analyse ist eine solche Verzweigung solange unproblematisch, wie das Prädikat, das zur Wahl des Ausführungspfades verwendet wird, eindeutig und effizient entscheidbar ist. Diese Entscheidung kann extrem erschwert werden.

² Ausführbare Dateien beginnen zumindest unter Windows und Mac OS X mit einer kurzen, charakteristischen Bytefolge.

4 Simon Bierbaum

Ein *deckendes Prädikat* ist ein Prädikat, das entweder nicht eindeutig oder nicht effizient (oder beides) entscheidbar ist. Eine deckende Variable ist eine Variable, die Teil eines deckendes Prädikats ist. Deckende Strukturen schließlich sind beliebige Datenstrukturen, die dazu beitragen, ein Prädikat deckend zu machen. Jedes deckende Prädikat an einer Verzweigungsstelle im Ablaufgraphen des Programms zwingt den Analysierenden dazu, entweder mit potentiell großem Aufwand das Prädikat zu entscheiden oder aber beide Möglichkeiten zu betrachten. Werden genügend viele deckende Prädikate verschachtelt, führt dies zur praktischen Unentscheidbarkeit hinsichtlich der Wirkungsweise des zu analysierenden Programms. Da ohne Beschränkung der Allgemeinheit die relevante Wirkung eines Programms erst nach einer solchen Kaskade an deckenden Prädikaten auftreten kann, besteht die Möglichkeit, einen Virus effektiv vor der Entdeckung zu schützen.

2.2. Nichtdeterminismus und Annahmen

Das Schlüsselement zur Konstruktion von deckenden Prädikaten, Variablen und Strukturen ist der *Nichtdeterminismus*. Während wohldefinierte Berechnungen und Funktionsaufrufe gut simuliert werden können und somit auch die Entscheidung von Prädikaten im Wesentlichen so komplex ist wie für die ausführende Maschine, kann der Einsatz von nichtdeterministischen Werten dazu führen, dass sämtliche möglichen Werten einer Variable oder einer Struktur zu betrachten sind. Zu den möglichen nichtdeterministischen Bausteinen gehören explizite Zufallsfunktionen, die aktuelle Uhrzeit, Werte in nicht initialisierten Speicherbereichen, Aktivitäten des Benutzers, der Inhalt von beliebigen Dateien und andere Werte, über deren Entstehung das Programm nur beschränkte Kontrolle hat und deren möglicher Wertebereich genügend groß ist. Über ein solches Prädikat kann der Autor des Virus beim Anlegen Annahmen treffen, die der Analysierende nicht kennt. Ein Virus könnte beispielsweise dem Benutzer ein Dialogfeld präsentieren und ihn zum Anklicken eines Buttons auffordern. Unmittelbar nach dem Mausklick kann die Position des Mauszeigers auf dem Bildschirm ziemlich genau auf die Fläche des Buttons eingeschränkt werden. Ein automatisches Analyseprogramm ist aber nicht imstande, die Aufforderung „Bitte klicken Sie auf den Button“ zu verstehen. Während also das Prädikat immer einen bestimmten Wert annimmt, ist es für die Analyse unentscheidbar.

2.3. Parallelismus

Eine weitere Möglichkeit, durch Nichtdeterminismus Prädikate unentscheidbar zu machen, ist die Parallelität. Der Virus erzeugt eine beliebige Anzahl an Threads oder Kindprozessen, die alle auf die gleiche Datenstruktur schreiben. Da das Scheduling durch andere auf dem System rechnende Prozesse für die Analyse praktisch nichtdeterministisch ist, ergibt sich eine fakultative Anzahl von Sequentialisierungen der verschiedenen Ausführungsstränge. Der Autor des Virus kann wieder mittels Ausnutzung von Invarianten erreichen, daß deckende Prädikate, die auf von den Threads erzeugten Werten basieren, immer einen bestimmten Wert annehmen.

3. Klischeebasierte Analyse

3.1. Definition und Motivation

Um festzustellen, ob es sich bei einem Programm und einen Virus handelt, kann ein Virens Scanner versuchen zu „verstehen“, welchen Zweck es erfüllt. Mit der *Klischeebasierten Analyse* kann er dies tun, ohne den Code simulieren zu müssen.

Die Klischeebasierte Analyse beruht darauf, daß gewisse Programmfragmente typisch für eine bestimmte Anwendung sind. Psychologische Studien haben gezeigt, daß Programmierer aus Fleisch und Blut beim Lesen von unbekanntem Code intuitiv keine formale Analyse vornehmen, sondern aus ihrem Erfahrungsschatz schöpfen und aus vorhandenen, bekannten Programmfragmenten, sog. Klischees, auf die Funktionsweise schließen. Dabei werden Kombinationen von Klischees zu weiteren Klischees abstrahiert, wodurch sich nach und nach immer allgemeinere Aussagen über einen immer größeren Programmabschnitt treffen lassen.

Solche Klischees gibt es in allen Programmiersprachen: `for(i=0; i<x.size(); i++)` deutet in C++ beispielsweise auf die Iteration durch eine Liste hin, `struct x { x* a, b; type_y y };` wäre eine einfache Implementation eines Knotens einer Baumstruktur. Eine Klischeebasierte Analyse könnte nun aus den beiden verschachtelten Klischees *Iteration-durch-Liste* und *Baumstruktur* schließen, daß hier eine Baumtraversierung vorgenommen wird.

Bei der klischeebasierten Analyse treten zunächst einige Probleme auf: Üblicherweise gibt es viele syntaktisch verschiedene Möglichkeiten, ein Klischee zu implementieren, und viele Algorithmen für ein Problem. Meistens werden also Klischees nicht durch exakte Übereinstimmung erkannt werden können. Zudem können die Bestandteile eines Klischees mit Anweisungen vermischt auftreten, die nicht zum Klischee gehören. Ein Klischee muß insbesondere nicht lokal sein, sondern kann sich über das ganze Programm erstrecken.

Für die Klischeebasierte Analyse bietet es sich deshalb an, ein Programm zunächst in einen Kontrollflußgraphen umzuwandeln und somit von der Syntax völlig und von der Implementation teilweise unabhängig analysieren zu können. Für die Eliminierung von Anweisungen, die nicht zum Klischee gehören, kann bei der Überprüfung einer potentiellen Fundstelle z.B. jede Anweisung ignoriert werden, die keine Variable beeinflusst, die im Klischee vorkommt.

3.2. Virenerkennung durch klischeebasierte Analyse

Ein Virens Scanner würde also mit einer Bibliothek an Klischees ausgestattet ein Programm analysieren und könnte Schritt für Schritt immer abstraktere Aussagen über die Funktion des Programms machen. Da Viren üblicherweise mehrere Funktionen erfüllen bzw. nur wenige Klischees existieren, die einen Virus eindeutig identifizieren, ist eine Bewertungsfunktion sinnvoll. Ähnlich dem Bayes'schen Filterverfahren zum Erkennen von eMail-Spam werden für bestimmte Klischees Punkte vergeben und für ein Programm aufsummiert. Auffällige Klischees, die

typischerweise in Viren auftreten, werden dabei mit vielen Punkte bewertet, Klischees, die häufig in harmlosen Programmen vorkommen, dagegen mit wenigen Punkten. Eine Punkteschwelle entscheidet schließlich, ob das Programm als Virus angesehen wird oder nicht.

Prinzipiell eignet sich die Klischeebasierte Analyse also, die einfachen Methoden der Code Obfuscation aus der Einleitung zu überlisten. Deckende Prädikate müssen kein Problem darstellen, da nicht versucht wird, den Wert desselben zu ermitteln, sondern lediglich die Verzweigung des Kontrollflusses am Prädikat an sich betrachtet wird. Jedoch können viele eingestreute Prädikate und Anweisungen, die für das Klischee relevante Werte ändern, die Analyse verwirren und die Erkennung von Klischees erschweren. Der folgende Abschnitt zeigt eine wirkungsvolle Methode, um Klischeeerkenntung vollständig unmöglich zu machen.

4. Virtuelle Maschinen

4.1. Konzept und Motivation

Die virtuelle Maschine kann in diesem Kontext genauso aufgefasst werden wie beispielsweise die Java Virtual Machine. Es handelt sich um einen Interpreter, der eine beliebige, festgelegte Sprache interpretiert. Der Interpreter und die Sprache müssen keineswegs die Gegebenheiten der vorhandenen Hardware widerspiegeln, sondern können prinzipiell beliebige Konzepte modellieren. Der Trick mit der virtuellen Maschine besteht nun darin, daß der Kontrollflußgraph eines Programms praktisch jegliche Aussagekraft verliert, wenn es sich lediglich um die Implementation einer virtuellen Maschine nebst einem Programm in der zugehörigen Sprache handelt. Schließlich wird ja das eigentliche Programm nicht mehr als Anweisungsblock, sondern als Datenblock aufgefasst. Während sich ein reguläres Programm mittels Klischeebasierter Analyse mehr oder weniger gut abstrahieren lässt, steht bei der Analyse einer virtuellen Maschine höchstens die Aussage „Es handelt sich um eine virtuelle Maschine“ am Ende. Somit können Funktionen, die typisch für ein Virus sind und in der Klischee-Datenbank eines Virenschanners gespeichert sind, in einer virtuellen Maschine effektiv vor Entdeckung geschützt werden.

Besonders geeignet sind virtuelle Maschinen, um im Programm verwendete Konstanten zu verbergen. Ein konstanter Wert kann durch eine virtuelle Maschine und ein passendes Programm ersetzt werden, das ihn erzeugt. Beispiele für potentiell verräterische konstante Werte sind Pfadangaben von auszulesenden oder zu ändernden Dateien wie `/etc/passwd`, und IP-Adressen, die zur Kommunikation mit dem Schöpfer des Virus dienen.

4.2. Programmiersprachen für Virtuelle Maschinen

Bei der Ausgestaltung der virtuellen Maschine und der dazugehörigen Maschinensprache sind einige Einschränkungen zu beachten. Da die Maschine als Teil eines metamorphen Virus eingesetzt werden soll, muß es möglich sein, auch auf die Maschinensprache automatische Obfuscation-Techniken anzuwenden. Weiterhin sollte die Semantik der Sprache in der Allgemeinheit unbekannt sein. Ansonsten bestünde für den Virenschreiber die Möglichkeit, einen metamorphen Virus anhand seiner z.B. in C verfassten, mitgeführten Quelltexte zu identifizieren.

Es existieren viele unbekannte Sprachen, die alle genannten Kriterien erfüllen. Eine einfache Sprache ist *Cow*³. *Cow* ähnelt einer Turingmaschine: Das Konzept umfaßt ein in Bytes unterteiltes Speicherband, besitzt einen Schreib/Lesekopf, ein Register sowie eine Ein-/Ausgabemöglichkeit. Programme werden als einfache Textdateien abgelegt. Die Sprache besteht aus elf Anweisungen, die aus Variationen des Strings „Moo“ bestehen. Bei der Ausführung eines Programms wird jedes Zeichen ignoriert, das nicht Bestandteil einer Anweisung ist. Somit ist die Obfuscation eines *Cow*-Programms sehr einfach, denn es kann in einen beliebigen Text eingebettet werden. Zudem wurde bewiesen, daß *Cow* Turing-vollständig ist und damit prinzipiell alle Berechnungen ausführen kann, die für einen Virus relevant sein könnten (siehe Literaturhinweis im Anhang). Der C-Quellcode eines 6kb großen *Cow*-Interpreters ist auf der Website des Autors verfügbar.

5. Program Slicing

Program Slicing beschreibt eine Technik, die aus einem Programm nur diejenigen Anweisungen extrahiert, die für die Bestimmung eines Werts an einer bestimmten Stelle im Programm notwendig sind. Ein Slicing-Algorithmus nimmt ein Programm, eine Stelle in diesem Programm sowie eine oder mehrere Variablen entgegen und berechnet daraus ein lauffähiges Programm, das lediglich für die Bestimmung der Variablen relevante Bestandteile enthält.

Beim Program Slicing wird zwischen statischem und dynamischem Slicing unterschieden: Statische Slices werden direkt aus dem Programmcode erzeugt und verhalten sich für jede Eingabe wie das ursprüngliche Programm. Ein dynamisches Slice wird für eine konkrete Eingabe an das Programm erstellt. Der Vorteil hierbei ist, daß an Verzweigungen im Kontrollfluß Entscheidungen einfach basierend auf den gegebenen Werten durchgeführt werden können und somit das entstehende Slice wesentlich kleiner sein kann. Nachteilig ist vor allem, daß eine sinnvolle Eingabe nicht bekannt sein muß und sie möglicherweise extrem groß ist. Die Aussagekraft eines Slices für den allgemeinen Fall wäre damit stark begrenzt.

Ein Vorgehen beim Slicing ist der Einsatz von Zusagekalkülen, die Aussagen darüber machen, welche Variablen in einer Anweisung geändert werden und welche anderen Variablen zur Bestimmung des neuen Wert herangezogen werden. Mit dieser Information kann ein Program Slicer von der gegebenen Stelle aus rückwärts durch

³ <http://www.bigzaphod.org/cow/>

das Programm iterieren und in jedem Schritt diejenigen Anweisungen entfernen, die nicht zur Bestimmung der betrachteten Variablen beitragen. Modelliert man diese Abhängigkeiten zwischen Anweisungen mit einem gerichteten Graph, so können alle Anweisungen entfernt werden, von denen aus kein Pfad zum gegebenen Knoten führt. In der Virenerkennung eignet sich das Program Slicing vor allem zum Finden von totem Code, also Code, der niemals ausgeführt wird. Mittels Program Slicing kann sich also ein Virens Scanner auf die offenbar wesentlichen Teile eines potentiellen Virus konzentrieren. Für die Simulation eines Programmteils stellt das Program Slicing eine sinnvolle Vorarbeit dar, da ein ausführbares und somit auch simulierbares Programm erzeugt wird. Bei der Konzentration auf relevanten Code fallen möglicherweise auch deckende Prädikate weg, die sich für die Simulation als problematisch hätten herausstellen können.

6. Literatur

Code Obfuscation

- Collberg, Thomborson, Low, "A Taxonomy of Obfuscating Transformations", University of Auckland, 1997

Metamorphie

- Péter Ször, Peter Ferrie, "Hunting For Metamorphic", *Symantec Corporation*, 2003

Deckende Strukturen

- Collberg, Thomborson, Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998

Klischeebasierte Analyse

- Wills, "Automated Program Recognition – a feasibility demonstration", *Artificial Intelligence Vol. 45*, 1990

Virtuelle Maschinen

- Monden, Monsifrot, Thomborson, "A Framework for Obfuscated Interpretation", University of Auckland, 2004
- van Oostenrijk, van Beek, "Semantics of COW", Radboud Universiteit Nijmegen, 2004

Program Slicing

- Tip, "A Survey of Program Slicing Techniques", Centre for Mathematics and Computer Science, Amsterdam, 1994