Source-Code-Manipulationen Erkennbarkeit und Abwehr

Frank Hoffmeister, Dortmund
Michael Schneider, Bonn

Maßnahmen zur Abwehr von Software-Manipulationen setzen zumeist im Bereich des Object-Code an. Dabei wird jedoch häufig übersehen, daß Software im Verlauf ihres Lebenszyklus in unterschiedlichen Darstellungsformen vorliegt und eine unberechtigte Einflußnahme auf allen Sprachebenen denkbar ist. Der vorliegende Beitrag zeigt auf, in welcher Weise Manipulationen des Source-Code vorgenommen werden können. Darüberhinaus werden mögliche Gegenmaßnahmen und der mit ihnen verbundene Aufwand geschildert.

Man geht heute allgemein davon aus, daß die erstmalige Erstellung eines komplexen Systems weniger Aufwand verursacht, als spätere Korrekturen, Änderungen oder Erweiterungen[1]. In jeder Phase, in der es notwendig wird, den Source-Code einer Software zu modifizieren, ergibt sich aber auch die Gefahr einer unkontrollierten Einflußnahme durch den mit der Änderung befaßten Programmierer.
Hält man sich vor Augen, daß Quellenprogramme, wenn sie einmal erstellt sind, selten einer Codeinspektion unterzogen werden, wird deutlich, daß Manipulationen kaum bemerkt werden können[2]. Findet eine solche Kontrolle dennoch statt, stehen dem potentiellen Angreifer einfache Mittel zur Verfügung, die ihn in die Lage versetzen, Überprüfungen zu unterlaufen.

I. Semantische Programm-Manipulationen

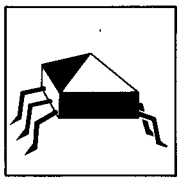
Der Quellcode läßt sich bereits durch semantische Manipulationen, das sind Veränderungen, die den Bedeutungsgehalt von Programmteilen verfälschen, sabotieren. Aus dieser überaus trivialen Vorgehensweise ergeben sich ernste Konsequenzen, wenn es darum geht, Sabotagemaßnahmen zu erkennen.
Dies soll anhand eines einfachen Beispiels, einer iterativen PASCAL-Implementation des ggT[3], erläutert werden:

Semantische Manipulationen = Änderung des Bedeutungsgehalts von Programmteilen

```
function ggT(a,b:integer):integer;
begin
    repeat
        if a < b then swap(a,b);[4]
        a:= a mod b;
    until a = 0;
    ggT:= b;
end;
```

Eine offensichtliche Verfälschung besteht im Austausch des Programmstücks durch eine zufällig gewählte Zeichenkette, die nicht der PASCAL-Syntax entspricht. Derartige Veränderungen können aber leicht mit Hilfe eines PASCAL-Syntax-Checkers festgestellt werden, der die betreffende Manipulation als Syntax-Fehler ausweisen würde. Aber schon kleinste Änderungen des Quellcodes können die Funktionalität der Funktion zerstören, ohne die PASCAL-Syntax zu verletzen.

1 Gerhard Weck; Datensicherheit: Methoden, Maßnahmen und Auswirkungen des Schutzes von Informationen; Stuttgart 1984, S. 253
2 W. Gleißner et al.; Manipulation in Rechnern und Netzen. Risiken, Bedrohungen und Gegenmaßnahmen; Addison-Wesley, 1989, S.132
3 Größter gemeinsamer Teiler zweier natürlicher Zahlen, hier in der Divisionsform des euklidischen Algorithmus
4 Die Hilfsfunktion swap(a,b) vertauscht den Inhalt der Variablen a und b.



Durch Austausch eines Zeichens läßt sich der obige Algorithmus so verändern, daß die Funktion ggT bei jeder Parameterbelegung mit a <> b; a,b > 0 in eine Endlosschleife gerät[5]:

```
function ggT(a,b:integer):integer;
begin
    repeat
        if a > b then swap(a,b);
        a:= a mod b;
    until a = 0;
    ggT:= b;
end;
```

Gerade an wichtigen und unübersichtlichen Programmteilen lassen sich durch Hinzufügen, Entfernen oder Manipulation einfacher Operationen und/oder Anweisungen Fehler erzeugen, die die Funktionalität des gesamten Programms untergraben und zudem nur von eingearbeiteten Programmierern erkannt werden können.
Diese Form der Source-Code-Sabotage ist einfach und den meisten Programmierern durch ihre Erfahrungen bei der Suche nach Programmfehlern bekannt. Zuweilen werden abgedruckte Programme in Zeitschriften sogar auf diese Weise „zensiert"[6].
Derartige „Fehler" programmgestützt zu erkennen, hieße die Korrektheit von Programmen mittels eines Algorithmus beweisen zu können. Grundlegende Sätze der theoretischen Informatik weisen jedoch darauf hin, daß dies ein unentscheidbares Problem ist[7].

Semantische Prüfung des Source- Codes durch Menschen unverzichtbar

Es gibt keinen Algorithmus, der die Korrektheit beliebiger Programme feststellen kann. Will man also derartige Sabotagemaßnahmen ausschließen, ist eine semantische Prüfung des Source-Code durch den Menschen unverzichtbar. Der hierzu notwendige Aufwand steht jedoch in keinem vernünftigen Kosten/Nutzen-Verhältnis, da der damit verbundene Aufwand einer Neuentwicklung des Software-Produkts nahekommt.
Versucht man dagegen Sabotagmaßnahmen phänomenologisch, also durch intensives Testen der Software zu erkennen, so kann dies zu einer trügerischen Sicherheit führen, wenn selten benutzte, aber vitale Systemfunktionen ungetestet bleiben[8]. Sicherheit durch Testen erreicht man also nur, wenn durch Tests wenigstens eine vollständige Abdeckung aller Programmzweige erreicht wird.
Während bereits das Erzeugen typischer Testfälle das Nachvollziehen des Designs und der Implementation eines Programms voraussetzt, ist das rechnergestützte Generieren von Testfällen zur Abdeckung aller Programmzweige wiederum unmöglich, da das Erkennen unerreichbarer Programmzweige ein unentscheidbares Problem ist[9]. Eine Kontrollinstanz wird sich wegen des erforderlichen Aufwandes also darauf beschränken müssen, die Funktionalität einer Software anhand ausgewählter, exemplarischer Testfälle zu überprüfen.
Exemplarische Tests, die nicht den vollständigen Nachweis der Funktionalität einer Software erbringen, decken aber bestenfalls halbherzig durchgeführte Sabotagemaßnahmen auf. Eine Gewähr, daß die Software die erklärten Eigenschaften besitzt, können sie nicht bieten.

---

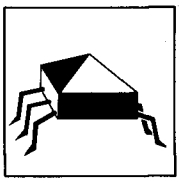
5 Die Umkehrung des Vergleichs zu a > b bewirkt, daß nach dem if-statement immer gilt: a <= b. Bei a <> b und a,b > 0 folgt a mod b = a, was wiederum zur Folge hat, daß die repeat-until-Schleife nicht abbricht.

6 Eckhard Krabel; Die Viren kommen; c't Heft 4 1987, S.108-117 enthält das Listing eines Computervirus, das in der abgedruckten Form nicht lauffähig ist. Die Beseitigung des Fehlers erzwingt, daß sich der Leser eingehend mit dem Problem und seiner Implementierung auseinandersetzen muß, bevor er leichtfertig mit einem Computervirus experimentieren kann.

7 J.E. Hopcroft / J.D. Ullman; Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie; Addison-Wesley, 1988, S.185-203

8 Beispielsweise die Steuersatzanpassung innerhalb eines Buchhaltungspakets.

9 J.C. Hunag; An Approach to Program Testing; in: Software Testing and Validation Techniques, E.Miller, W.E. Howden (Edt.), S. 246-261 M.R. Woodward, et al.; Experience with Path Analyses and Testing of Programs; in: IEEE Transactions on Software Engineering, Vol. SE-6 No.3, May 1980, S. 278-286



II. Trojanische Pferde

Aus den zuvor genannten Gründen wird in der Praxis die Überprüfung der Korrektheit und Integrität einer Software immer unvollständig bleiben. Damit bieten sich einem Saboteur Möglichkeiten, sogenannte Trojanische Pferde in die Software einzubringen[10]. Das Erweitern der Login-Routine eines Betriebssystems, das neben dem eigentlichen Password eines Benutzers ein geheimes Master-Password akzeptiert, ist ein Beispiel für ein Trojanisches Pferd, das seinem Programmierer erlaubt, sich Zugang zu jedem Benutzer-Account zu verschaffen.

Prohibitiver Entdeckungsaufwand bei „Trojanischen Pferden"

Beliebig destruktive Funktionen können als Trojanische Pferde an bestimmte Programmfunktionen oder externe Bedingungen wie das Verstreichen eines bestimmten Datums gekoppelt werden. Das Erkennen von Trojanischen Pferden ist ein ebenso schwieriges Problem wie das Erkennen fehlerhafter Programmteile, auch hier ist der erforderliche Aufwand zur Entdeckung prohibitiv. Aber selbst wenn man bereit ist, eine semantische Analyse des Quellcodes durch Menschen durchführen zu lassen, so gibt es unter bestimmten Rahmenbedingungen Trojanische Pferde, die sich sogar dieser Kontrolle entziehen. Dies wird im folgenden anhand eines Beispiels demonstriert werden, das Thompson 1984 in seiner Turing Award Lecture[11] vorgetragen hat.

Basis für sein Beispiel ist das UNIX-Betriebssystem, das den Vorteil besitzt, mit verhältnismäßig geringen Aufwand auf andere Rechner portierbar zu sein, da fast alle Systemfunktionen und -Utilities in der (höheren) System-Implementierungssprache C[12] implementiert sind.

Bei dem Versuch, ein Trojanisches Pferd in das Login-Programm einzusetzen, erkennt derjenige, der das Trojanische Pferd einbringen will, daß seine Programmänderungen sofort von einem fremden Programmierer bemerkt werden müssen, wenn dieser den Quellcode inspiziert (beispielsweise gelegentlich einer Portierung). Damit wäre sein Trojanisches Pferd wertlos. Wie die übrigen Systemfunktionen ist das Login-Programm in der Programmiersprache C geschrieben; um sein Trojanisches Pferd der Entdeckung zu entziehen, verändert Thompson den C-Compiler seines Rechners so, daß immer dann, wenn der Compiler ein bestimmtes Codefragment des Login-Programms erkennt, er dieses gegen ein modifiziertes ersetzt, welches das Trojanische Pferd enthält:

Ein im Compiler verstecktes Trojanisches Pferd

```
compile(s)
char *s;                              /* next program line */
{
        /* match login-code  */
      if (match(s,loginPattern))
      {
           /* replace it */
           compile(„Trojan Horse");
           return;
      }
      ...                             /* normal processing */
{
```

Obwohl der Quellcode des Login-Programms kein Trojanisches Pferd enthält, erzeugt jede Übersetzung mit dem modifizierten C-Compiler ein Login-Programm, in dem das Trojanische Pferd enthalten ist, denn die Codierung des Trojanischen Pferdes wird durch den Compiler in das entstehende Maschinenprogramm eingefügt. Eine spätere Analyse des Quellcodes kann die Modifikation nicht entdecken, da sie durch den Compiler zur Compile-Zeit durchgeführt wird.

Dieser Kunstgriff hat das prinzipielle Problem der Entdeckung des Trojanischen Pferdes bei einer Quellcodeinspektion nicht beseitigt, sondern nur verlagert. Eine Inspektion der compile-Routine des C-Compilers würde die Existenz des Trojanischen Pferdes offenlegen. Aber auch diesem (aus der Sicht eines potentiellen Saboteurs „unangenehmem") Umstand läßt sich durch einen neuerlichen Kunstgriff abhelfen.

---

10 Zur Definition des Begriffs vgl.: Michael Schneider; Zur Terminologie im Bereich softwaretechnischer Dysfunktionen, jur-pc 1989, S.133

11 Ken Thompson; Reflections on Trusting Trust; Communications of the ACM, Aug.1984, Vol.27(8), S.761-763

12 B.W. Kernighan, D.E. Ritchie; The C Programming Language; Prentice-Hall, New Jersey, 1978

Ein Trojanisches Pferd zur Tarnung des Trojanischen Pferdes

Die meisten der modernen Compiler sind in der Programmiersprache implementiert, die sie selbst übersetzen: Ein C-Compiler ist demgemäß in der Programmiersprache C geschrieben. Damit eröffnet sich die Möglichkeit ein zweites Trojanisches Pferd einzusetzen, dessen Aufgabe es ist, die Modifikation des C-Compilers zu verbergen:

```
compile(s)
char *s;                                    /* next program line */
{
    if (match(s,compilePattern))/* match compile-code */
    {
        compile(„these add-ons“);/* replace it */
        return;
    }
    if (match(s,loginPattern))/* match login-code  */
    {
        compile(„Trojan Horse“);/* replace it */
        return;
    }
    ...                                     /* normal processing */
}
```

Diese Vorgehensweise erscheint zunächst widersinnig, erweist sich bei genauerer Betrachtung jedoch als äußerst wirkungsvoll, wenn folgendes Vorgehen damit verknüpft wird:

1. Der modifizierte C-Compiler mit seinen beiden Trojanischen Pferden wird übersetzt und ersetzt den Original-C-Compiler.
2. Die Modifikationen im Quellcode des C-Compilers werden entfernt. Der Quellcode enthält nun keinerlei Indizien, die auf die erfolgten Modifikationen hinweisen.

Genau wie im Fall des Login-Programms wird bei jeder anschließenden Übersetzung des C-Compilers (durch sich selbst) die Modifikation des C-Compiler-Quellcodes zur Compile-Zeit vom Compiler selbst durchgeführt. Die Informationen zur Modifikation des C-Compiler-Quellcodes befinden sich nur in der Binärversion des C-Compilers und werden bei der Erstellung einer neuen Binärversion an diese weitergegeben. Jede Quellcode-Analyse bleibt also bei dieser Einsatzform von Trojanischen Pferden wirkungslos.

Bedenkt man, daß UNIX-Portierungen mit Hilfe sog. Cross-Compiler erfolgen, die sich nur in der Code-Generierungskomponente vom normalen C-Compiler unterscheiden, so erkennt man, daß die Trojanischen Pferde im C-Compiler und im Login-Programm ebenfalls auf das andere Rechnersystem „portiert“ werden, und so seinem Autor auch dort den gewünschten „Dienst“ erweisen.

III. Fazit

Die Maßnahme, den Quellcode eines Programmes einer Inspektion zu unterziehen, bevor er compiliert, mit einem Interpreter abgearbeitet oder auf andere Weise in eine Form umgesetzt wird, in der er zur Ausführung gelangt, kann durchaus sinnvoll sein. Auf diese Weise läßt sich die Gefahr, daß eine manipulierte oder manipulierbare Software entsteht, erheblich reduzieren.

Dennoch können gut durchdachte Angriffe auf den Quellcode ebensowenig mit letzter Sicherheit abgewehrt werden wie Manipulationen ausführbarer Programme. Bereits die Einfachheit der Sabotage durch semantische Programm-Manipulationen zeigt die Grenzen dessen, was durch Kontrollen praktisch machbar ist. Mit ausgefeilteren Techniken lassen sich selbst ausgeklügelte Schutzmechanismen wirkungsvoll unterlaufen.

Keine endgültige Sicherheit trotz Quellcode-Analyse

Diese Erkenntnis sollte zu einer kritischen Einschätzung der eigenen Möglichkeiten führen, Manipulationen des Source-Code abzufangen[13]. Aus dem Umstand, Quellenprogramme verstehen und überprüfen zu können, läßt sich nicht ableiten, daß eine Kontrolle erfolgreich und mit vertretbarem Aufwand durchgeführt werden kann.

13 Insbesondere im Zusammenhang mit der Softwarehinterlegung wird allzu unkritisch davon ausgegangen, daß es ohne weiteres möglich sei, Trojanische Pferde aus Quellprogrammen zu entfernen. Vgl. dazu beispielsweise: W. Loseries / E. Spahr Carneiro; Sicherung der dauernden Einsetzbarkeit der Software für den Fall des Konkurses des Software-Lieferanten; in: D.J. Hildebrand / T. Hoene (Hrsg.); Software in der Insolvenz; Stuttgart 1989, S. 7(15)