Hochschule Wismar
University of Applied Sciences
Technology, Business and Design
Fakultät für Ingenieurwissenschaften, Bereich EuI

Bachelor-Thesis

Statische Codeanalyse für sicherheitskritische Softwaresysteme

Eingereicht am: 01.03.2012

von: Jan Klemkow
geboren am 09.01.1988
in Crivitz

Betreut von: Prof. Dr.-Ing. Ernst Jonas
Hochschule Wismar

Dipl.-Math. Alexander Bluhm
Gesellschaft für Netzwerk- und
Unix–Administration mbH

Kurzreferat

In dieser Arbeit werden die Möglichkeiten der statischen Quellcode–Analyse untersucht und am praktischen Beispiel eines Firewall–Software–Systems angewendet. Dabei wird untersucht, inwieweit freie Quellcode–Analyseprogramme in der Lage sind, Software–Fehler zu finden. Einleitend werden bekannte Fehler erklärt, welche dann im weiteren Verlauf dieser Arbeit mit verschiedene Analyseprogrammen gesucht werden. Danach wird der Einsatz von statischer Quellcode–Analyse im Entwicklungsprozess diskutiert.

Abstract

This document examines the possibility of static code analysis in the practical context of a firewall software system. It analyses the possibility of free source code analysing tools to find software faults. At the beginning of this document well–known software faults are described. Later on it is tried to find them with several code analysing tools. After this it discusses the possibilities of using code analysis in the development process.

Danksagung

Ich danke meinem Betreuer Prof. Dr. Ernst Jonas nicht nur für die wissenschaftliche Betreuung bei dieser Arbeit, sondern auch für die fachliche Betreuung im Rahmen des Network–Security–Management–Labors. Die Arbeit in dem Labor und die kritischen Fachgespräche haben mich sehr beflügelt und mich motiviert, Arbeiten wie die Vorliegende zu schreiben.

Zu großem Dank bin ich auch meinem betrieblichen Betreuer Dipl.-Math. Alexander Bluhm verpflichtet, für die außerordentlich kompetente Betreuung, welche weit über den Inhalt dieser Arbeit hinausging und mich auch in anderen Bereichen fachlich sehr voran gebracht hat.

Weiter gilt meinem Dank den GeNUA–Mitarbeitern Dr. Christian Ehrhardt, Dipl.-Inf. Markus Friedl, Dipl.-Inf. Alexander Fiveg sowie Dr. Stefan Fritsch, für Ihre sehr interessanten Hinweise, Fragestellungen und Fachgespräche rundum das Themenfeld dieser Arbeit.

Einen besonderer Dank geht an Dr. Roland Meister, welcher mich beim Schreiben dieser Arbeit mittels LaTeX mit viele kleinen Details und Hinweisen für eine korrekte Formatierung sehr unterstützt hat.

Inhaltsverzeichnis

Kapitel 1

Einleitung

„Most software is fragile and buggy because most programs are too complicated for a human brain to understand all at once.“ [7]

Mit diesem Satz beschreibt Eric S. Raymond sehr treffend das große Problem der zunehmenden Komplexität von Software. In seinem Essay „The Art of Unix Programming“ weist er damit auf die Notwendigkeit von möglichst einfachen Software–Entwürfen hin. Diese Arbeit verfolgt einen anderen Ansatz, um sich mit dem Problem der Komplexität von Software auseinanderzusetzen. Es wird untersucht, inwieweit automatisierte statische Quellcode–Analyse in der Entwicklung von Software dabei helfen kann, die Quellcode–Qualität zu verbessern und Fehler zu vermeiden. Viele Fehler, welche in Softwaresystemen gefunden werden, sind oft schon lange bekannte Fehler, welche immer wieder auftreten und für instabile und unsichere Softwaresysteme verantwortlich sind. So begegnet man Meldungen von Pufferüberläufen[1], Speicherlecks, SQL– und Code–Injektionen[2] sowie Format–String–Angriffen[3] schon seit mehreren Dekaden in der Fachpresse. Die Zunahme der Komplexität durch Abstraktion, Parallelisierung, Virtualisierung und Abwärtskompatibilität erschweren es dem Programmierer hierbei ein Softwaresystem zu überschauen und damit zu wissen, wie sich sein Softwaresystem im realen Einsatz verhalten wird. Statische Quellcode–Analyse kann den Programmierer dabei unterstützen, Fehler frühzeitig zu erkennen und sie damit zu vermeiden.

[1] http://heise.de/-943751
[2] http://heise.de/-1408079
[3] http://heise.de/-1425053

1.1 Verwandte Arbeiten

In der Dissertation „Static Code Analysis in Multi–Threaded Environments“ [4] wird untersucht, mit welchen Mitteln der statischen Quellcode–Analyse Probleme bei Parallelisierungen beseitigt werden können. In den Fallstudien der Dissertation werden ebenfalls Unix–artige Systeme untersucht. Die vorliegende Arbeit geht nicht auf spezielle Probleme der Parallelisierung von Software–Systemen ein, sondern befasst sich mit allgemeineren Fehlerarten und untersucht mit den Möglichkeiten freier statischer Quellcode–Analyseprogramme diese zu Finden. Der Bericht von Sun–Microsystems „Finding Bugs in Open Source Kernels using Parfait“ [3] befasst sich mit den Möglichkeiten des Einsatzes des kommerziellen Quellcode–Analyseprogrammes *Parfait*. Innerhalb dieser Arbeit werden ausschließlich freie Quellcode–Analyseprogramm untersucht. Ein andere Bericht dieser Firma, „BegBunch – Benchmarking for C Bug Detection Tools“ [1], dreht such um die Fragestellung, wie man die Effizienz von Quellcode–Analyseprogramme bewerten kann. Dafür wurde dass proprietäre Programm *BugBunch* entwickelt. Im Rahmen dieser Arbeit wird die Möglichkeiten der statischen Quellcode–Analyse untersucht, Fehler zu finden, ohne eine Aussage darüber zu treffen mit welcher Effizienz dieses geschieht.

1.2 Konventionen

In dieser Arbeit werden folgende Konventionen zur Formatierung verwendet. Im Fließtext erwähnte Funktionen werden im `Typewriter`–Stil und mit Klammerpaar formatiert, z. B. `main()`. Werden im Text Funktionen der Standard–C–Bibliothek, System-Aufrufe oder Kommandos verwendet, welche in einer Manpage beschrieben werden, dann wird in das anhängende Klammerpaar die Manpage–Kategorie geschrieben. So lassen sich z. B. die Funktionen `malloc(3)` und `malloc(9)` besser im Fließtext voneinander unterscheiden. Es wird dabei immer auf Manpages des OpenBSD–Projektes[4] verwiesen. Technische Ausdrücke sowie Programmnamen werden im *Kursiv*–Stil dargestellt.

[4] http://www.openbsd.org/cgi-bin/man.cgi

Kapitel 2

Grundlagen

2.1 Informationssicherheit

Informationssicherheit bezeichnet die Vertraulichkeit, Verfügbarkeit und Integrität von Computersystemen. Die Eigenschaft der Vertraulichkeit bedeutet in diesem Zusammenhang, dass Informationen nur einem bestimmten Personenkreis bekannt werden. Verfügbarkeit bezeichnet die Eigenschaft, dass Informationen und Computer gestützte Dienstleistungen ohne Unterbrechung zur Verfügung stehen. Die Integrität beschreibt die Unversehrtheit von Computersystemen und Informationen, sodass diese nicht von Dritten beeinflusst wurden.

2.2 Softwarefehler

Im Allgemeinen ist eine Softwarefehler jedes Verhalten eines Prozesses, welches vom Programmierer nicht beabsichtigt wurde. In seltenen Fällen können diese Verhaltensweisen nützlich sein und werden dann im Nachhinein als Funktionalität betrachtet. In der Mehrzahl der Fälle, in denen ein unvorhergesehenes Programmverhalten auftritt, handelt es sich um einen Fehler, welcher korrigiert werden sollte. Sobald ein Softwarefehler die Informationssicherheit eines Computersystems beeinträchtigt, spricht man von einer sicherheitsrelevanten Schwachstelle. Abhängig davon, welcher Teilaspekt der Informationssicherheit betroffen ist und in welchem Umfang der Fehler ausgelöst werden kann, ist ein Softwarefehler mehr oder weniger kritisch.

2.3 Schwachstellen

In diesem Abschnitt werden Kategorien von Programmfehlern erläutert. Dabei wird analysiert, inwieweit diese sicherheitsrelevant sind und welcher Schaden verursacht werden kann. Zudem wird erläutert, inwieweit Programme, welche in der Programmiersprache *C* verfasst wurden, von diesen Fehlern betroffen sind.

2.3.1 Speicherleck

Ein Speicherleck, auch *memory leak* genannt, ist ein Stabilitätsrisiko, da es ein Programm zum Absturz bringen kann. Dieses Problem tritt auf, wenn ein Programm Speicher reserviert und diesen nicht wieder frei gibt. Tritt dieses Phänomen im Programmfluss an sich nicht häufig wiederholenden Stellen auf, wird es oft nicht bemerkt und hat keinen großen Einfluss auf den Gesamtspeicherbedarf oder die Stabilität eines Prozesses. An Stellen an denen ein Speicherleck iterativ verursacht wird, ist dieses Problem schon kritischer. Wenn diese Iterationen von außen beeinflusst oder aufgerufen werden, — z. B. in einem Server-Client-Verbund — könnte ein Angreifer dieses ausnutzen. Bei zunehmenden Verbrauch von Hauptspeicher durch ein Programm, wird auch das *swapping* des Hauptspeichers zunehmen und damit das System verlangsamen. Wenn ein Programm die vom Betriebssystem zugewiesene maximale Speicherauslastung überschritten hat, wird es von ihm beendet.

Die Programmiersprache *C* sowie deren Standard-Bibliothek enthalten keine Mittel der Ressourcen-Verwaltung. Es gibt keinen Mechanismus, wie die automatische Speicherbereinigung (*Garbage Collection*), welche sich darum kümmert, nicht mehr benötigten Speicher wieder freizugeben. Der Programmierer muss sich somit selbst um das Speichermanagement seines Programmes kümmern.

2.3.2 Speicherbereichsverletzung

Bei einer Speicherbereichsverletzung versucht ein Prozess auf Teile des virtuellen Adressraums zuzugreifen, welche nicht für ihn bestimmt sind. In diesen Fällen sendet das Betriebssystem dem laufende Prozess das Signal *segmentation fault*, welches ihn im Normalfall beendet.

2.3.3 *NULL*–Zeiger–Dereferenzierung

Eine *NULL*–Zeiger–Dereferenzierung ist ein Zugriff auf Zeiger, wenn diese auf die *NULL*–Adresse zeigen. Die *NULL*–Adresse wird dazu verwendet, um einen Zeiger ausdrücklich als nicht gesetzt oder ungültig zu markieren. Ein Prozess, der auf einen solchen Zeiger zugreift, greift in der Regel auf einen ungültigen Speicherbereich zu.

Diese Art von Fehler kann bei ungenügender Fehlerbehandlung auftreten. Viele Systemaufrufe und Funktionen der Standard–*C*–Bibliothek, welche einen Zeiger zurückgeben, geben im Fehlerfall einen Zeiger auf *NULL* zurück. Wenn Zeiger aus solchen Funktionen vor ihrer Verwendung nicht geprüft werden, kommt es im Fehlerfall zu einer *NULL*–Zeiger–Dereferenzierung. Wenn ein solches Verhalten von außen durch einen Angreifer provoziert werden kann, handelt es sich um eine *Denail of Service (DoS)*–Schwachstelle, da der angebotene Dienst abstürzt. Bei einem *DoS*–Angriff ist es dem Angreifer möglich, den Dienst eines Computers in einen Zustand zu bringen, in dem er seine Aufgaben nicht mehr erfüllen kann.

2.3.4 use–after–free

Ein weiterer spezieller Fall einer Speicherbereichsverletzung ist ein sogenanntes *use–after–free*. Dabei wird auf einen Zeiger zugegriffen, desses Inhalt zuvor mit der Funktion `free(3)` freigegeben wurde. Wurde zuvor beim Freigeben des Speichers die ganze Speicherseite wieder frei gegeben, dann wird ein Zugriff auf einen solchen Zeiger vom Betriebssystem mit dem Signal *segmentation fault* behandelt. Wenn hingegen innerhalb einer Speicherseite nur ein einzelner Bereich freigegeben wurde, dann wird der Zugriff nicht unterbunden und der Speicher regulär gelesen oder beschrieben. Dieses kann je nach nach Kontext im Programm dazu führen, dass Daten nach außen hin sichtbar werden, welche vertraulich sind. Zum anderen kann es dazu führen, dass dieser Speicherbereich beim nächsten Aufruf von `malloc(3)` wieder vergeben wird und es dann zwei Zeiger gibt, die auf den selben Speicherbereich zeigen. Dieses hat dann unvorhersagbare Folgen für den weiteren Ablauf des Prozesses.

2.3.5 Pufferüberlauf

Bei einem Pufferüberlauf wird ein reservierter Speicherbereich über die Grenze der Reservierung hinaus beschrieben. Dieses kann zu sehr unterschiedlichem Verhalten von Prozessen führen. Da einem Prozess vom Betriebssystem immer nur ganze Speicherseiten zugewiesen werden, bemerkt das Betriebssystem das Überschreiben erst dann, wenn über die Grenzen der letzten Speicherseite hinaus geschrieben wird, da es dann zu einer Speicherbereichsverletzung kommt. Innerhalb einer Speicherseite können sich auch mehrere reservierte Speicherbereiche befinden. Dadurch kann ein Überschreiben auch dazu führen, dass andere Daten eines Prozesses korrumpiert werden. Das kann unvorhersehbare Folgen für den weiteren Ablauf des Prozesses haben. Ein etwas anderes Verhalten kann entstehen, wenn ein Puffer, der sich auf dem *Stack* befindet, über seine Länge hinaus beschrieben wird. Sollte dabei z. B. die Rücksprungadresse einer Funktion mit überschrieben werden, kann dieses die Ausführung von fremden Programmcode zur Folge haben.

Die Programmiersprache *C* und deren Standardbibliothek sind für Pufferüberläufe beim Verarbeiten von Zeichenketten besonders anfällig. Viele Programme verwenden dafür Funktionen, die beim Verarbeiten von Zeichenketten die Pufferlängen nicht beachten. Dadurch kann es bei bestimmten Operationen zum Überschreiben von reservierten Speicherbereichen kommen. Für viele Funktionen, bei denen es zu Pufferüberläufen kommen kann, gibt es Alternativen, bei denen die Pufferlängen beachtet werden, z. B. `strcpy(3)` und `strncpy(3)`, `strcat(3)` und `strncat(3)` sowie `strlen(3)` und `strnlen(3)`.

2.3.6 Datei-Handle-Leck

Bei einem Datei–Handle–Leck wird zu dem Systemaufruf `open(2)` kein `close(2)` aufgerufen. Der erfolgreiche Aufruf von `open(2)` führt zum Öffnen einer Datei und dem Belegen eines Datei-Handles. Erst beim Aufruf von `close(2)` mit dem Datei–Handle wird dieses wieder freigeben. Einem Prozess stehen nur eine begrenzte Anzahl an Datei-Handle zur Verfügung. Sobald diese maximale Anzahl von Datei–Handlen erreicht wurde, kann keine weitere Datei von diesem Prozess geöffnet wer-

den. Ähnlich wie die Speicherverwaltung muss auch die Verwaltung der Datei-Handle vom Programmierer manuell durchgeführt werden.

2.3.7 Fehlerhaftes Blockieren von *Interrupts*

Ein typischer Fehler im Ressourcen–Management von Betriebssystemen tritt beim Blockieren von Betriebsmitteln oder *Interrupts* auf, wenn diese nicht korrekt wieder freigegeben werden. Oft wird in Fehlerbehandlungen vergessen, die entsprechenden Blockierungen wieder freizugeben. Dieses hat dann zu Folge, dass nachfolgende Prozesse von diesen Betriebsmitteln abgeschnitten sind und damit selbst blockiert sind. Bei der Unterbrechung von *Interrupts* kann dieses fatale Folgen für das Gesamtsystem haben. Wenn ein Teil des Betriebssystems für ein kritische Operation die Interrupt–Behandlung des Prozessors unterbricht und diese nicht wieder freigibt, kann ein Zustand entstehen, indem es auf keine äußeren Ereignisse mehr reagiert und einfriert. Wenn ein solches Verhalten von einem Dritten beeinflusst werden kann, spricht man von einer *denial–of–service*–Schwachstelle.

2.3.8 Format–String–Schwachstelle

In der Standard–*C*–Bibliothek gibt es eine Reihe von Funktionen, um Zeichenketten mit einer speziellen Formatierung zu erzeugen, wie z. B. `printf(3)`. Diese Funktionen haben alle eine undefinierte Anzahl von Parametern. Der erste Parameter ist eine Zeichenkette, welche das Format zur der erzeugten Zeichenkette sowie die Typen und Anzahl der weiteren Parameter der Funktion bestimmt. Wenn in einem Programm nur eine Zeichenkette ausgegeben werden soll, wird oft die Funktion `printf(3)` mit der auszugebenden Zeichenkette als einiger Parameter benutzt. Diese Zeichenkette wird dann auf Steuerzeichen hin untersucht und interpretiert. Sobald diese Steuerzeichen auftauchen, wie z. B. `%d`, dann wird vom *Stack* der entsprechende Parameter gelesen unabhängig davon ob der Funktion ein weiterer Parameter übergeben wurde. Wenn es einem Dritten möglich ist, diesen Format–String zu verändern, dann kann er damit den Programm–*Stack* auslesen oder eine Speicherbereichsverletzung auszulösen.

2.3.9 Passwort–Leck

Bei einem Passwort–Leck ist es einem Dritten möglich durch Software–Fehler unbefugt an Passwörter zu gelangen. Wenn solche Informationen wie Passwörter oder auch Kryptographischeschlüssel in einem Programm verarbeitet werden, dann sollten sie nach ihrem Gebrauch gelöscht werden. Im praktischen Beispiel bedeutet das, dass man einen reservierten Speicher, welche solche Informationen beinhaltet, vor dem freigeben, oder dem wiederverwenden für andere Informationen überschreibt. Wenn man solche Puffer ohne sie zu überschrieben freigibt, dann bleiben die eigentlichen Daten im Speicher des Programms erhalten. Ein Dritter könnte sich durch anderen Programmfehler Zugriff auf diese Informationen verschaffen. Aus diesem Grund sollten sensible Daten nach ihrer Verwendung im Programm immer gelöscht werden.

Kapitel 3

Statische Quellcode Analyse

Die statische Quellcode-Analyse ist ein *White–Box*–Software–Testverfahren, bei dem der Quelltext eines Programmes auf Fehler untersucht wird. Dabei grenzt sich die statische Quellcode–Analyse von der dynamischen dadurch ab, dass nur der vorliegende Quelltext überprüft wird und kein Zusammenhang zum Laufzeitverhalten hergestellt wird.

Statische Quellcode–Analyse fängt bereits mit dem Compiler an. Dieser analysiert den Quellcode auf lexikalische Korrektheit. Der *GNU–C–Compiler* `gcc(1)`[1] ist ein *Compiler* für die Programmiersprache *C* und Bestandteil der *GNU–Compiler–Collection*. Die Software, welche diese Arbeit untersucht, wird mit der vom OpenBSD–Projekt angepassten Version des `gcc(1)` kompiliert. In dieser ist der Linker so modifiziert worden, dass er vor bestimmten Funktionen warnt, welche gefährlich sein können, wie z. B. `strcpy(3)` und `stpcpy(3)` Für den `gcc(1)` lassen sich verschiedene Warnstufen konfigurieren. Mit diesen lassen sich die statischen Quellcode-Analyse-Möglichkeiten des Compilers nutzen. Andere Compiler wie z. B.der *LLVM (Low Level Virtual Machine)* bieten ebenfalls erweitere statische Quellcode–Analysemöglichkeiten. Im OpenBSD–*Build–Process* wird zur Qualitätssicherung das Quellcode–Analyseprogramm *lint* [6] [2] verwendet. Die Möglichkeiten dieser Programme konnten in dieser Arbeit aus Zeitgründen leider nicht untersucht werden.

[1] http://gcc.gnu.org/

Im weiteren Verlauf der Arbeit und bei allen Untersuchungen, wird von lexikalisch korrektem Quellcode ausgegangen. Der Fokus dieser Arbeit liegt auf dem Finden von logischen Fehlern.

Kapitel 4

Bewertung von Programmen

In diesem Kapitel werden alle Programme beschrieben, mit denen im Rahmen dieser Arbeit Quellcode analysiert wurde.

4.1 Cppcheck

Das Analyseprogramm *Cppcheck*[1] analysiert den Quelltext sehr tiefgründig. Dadurch kommt es zu einem Problem beim Analysieren von Quellcode, in dem viele Präprozessoranweisungen verwendet werden. Dieses ist sehr zeitaufwändig und kann bei größeren Projekten mehrere Stunden dauern, da *Cppcheck* versucht, alle möglichen Kombinationen von definierten Präprozessorkonstanten durchzuprobieren. Es verfolgt unter anderem die Verwendung von Variablen und Zeigern und ist damit in der Lage *NULL*–Zeiger–Dereferenzierungen zu finden. Das Programm beinhaltet eine Fülle von Suchmustern, kann aber auch mittels eigener Modulen, mit denen man eigene Muster beschreibt, erweitert werden. Das Erweitern von *Cppcheck* wurde im Rahmen dieser Arbeit nicht untersucht, da es sehr komplex und zeitaufwändig ist. Die Einsatzmöglichkeiten von eigenen Suchmustern wurde mit dem Programm *coccinelle* untersucht (siehe unten).

[1] http://cppcheck.sourceforge.net/

4.2 Flawfinder

Das Programm *Flawfinder*[2] führt eine sehr grobe und einfache Prüfung des Quellcodes durch. Es orientiert sich an einer internen Datenbank, in der zu bestimmen Textmustern Hinweise hinterlegt sind. Die Hinweise des Programms sind sehr allgemein. Das Programm versteht nicht alle Sprachkonstrukte der Programmiersprache *C*. So kann es nicht zwischen den Bezeichnern von Funktionen und Variablen unterscheiden und weisst z. B. bei der Verwendung des Namen „system“ immer auf einen möglichen Fehler im Zusammenhang mit der Funktion `system(3)` hin.

4.3 Rough Auditing Tool for Security

Das Programm *Rough Auditing Tool for Security (RATS)*[3] verhält sich ähnlich wie das Programm *Flawfinder*. Es enthält eine Datenbank an Funktionen, welche als gefährlich eingestuft werden und meldet, wenn sie verwendet werden. Dabei wird kein Bezug auf die Umstände einen bekannten Fehlers genommen. So weist das Programm, beispielsweise bei der Verwendung der Funktion `getopt(3)`, auf eine fehlerhafte Implementierung[4] dieser Funktion unter Solaris im Jahre 1999 mit der Meldung aus Listing 4.1 hin und stuft das Warnlevel auf „High“ ein. *RATS* wurde in dieser Arbeit in der aktuellen Version 2.3 analysiert.

Listing 4.1: `getopt(3)` Sicherheitswarnung von RATS

```
driverstub.c:3986: High: getopt
Truncate all input strings to a reasonable length
before passing them to this function
```

4.4 grep

Das UNIX–Programm `grep(1)` eignet sich gut, um schnell nach bekannten Ausdrücken im Quellcode zu suchen. Programmierer kennzeichnen Stellen im Quelltext,

[2] http://www.dwheeler.com/flawfinder/
[3] https://www.fortify.com/ssa-elements/threat-intelligence/rats.html
[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0966

an denen aus Zeitgründen unsauber gearbeitet wurde, oft mit den Kommentaren „XXX“, „TODO“ sowie „FIXME“. Dieses sind Hinweise an sich selbst und andere Programmierer, dass Fehlerbehandlungen fehlen, Benutzereingaben nicht validiert werden oder allgemein unsauber programmiert wurde. Bei der Qualitätssicherung von Software und einem manuellen *review* von Quellcode lohnt es sich, solche Stellen zu überprüfen, da es dort leicht zu Softwarefehlern kommen kann.

4.5 coccinelle

Das Programm *coccinelle*[5] eignet sich sehr dazu eigene Programmkonstrukte zu formulieren und danach zu suchen. *coccinelle* besteht aus einer eigenen kleinen Sprache, welche es dem Programmierer erlaubt, Suchmuster für *C*–Quelltexte zu formulieren und mit diesen Suchen sowie Ersetzungen vorzunehmen. Damit lassen sich auch komplexe Suchmuster erstellen. Da das Programm die Syntax der Sprache *C* sowie *C++* bereits versteht, sind Suchmuster einfacher zu formulieren, als mit normalen regulären Ausdrücken[6]. Durch diese Eigenschaften ist *coccinelle* sehr gut dafür geeignet, um Suchmuster speziell für das eigene Software–Projekt zu entwickeln. Die Möglichkeit, Ersetzungen als Muster zu formulieren, eignet sich dazu, um ein allgemeines Programmkonstrukt in ein anderes umzuformen.

4.6 codecheck

Das Programm *codecheck* ist ein *Framework* für verschieden Quellcode–Analyseprogramme, welches im Rahmen dieser Arbeit entwickelt wurde. Es führt die verschiedenen Analyseprogramme auf ein Quellcode–Verzeichnis aus und verwaltet deren Meldungen. Die Verwaltung von Falschmeldungen ist dabei eine der Hauptaufgaben. Die unterschiedlichen Meldungen werden zunächst sortiert und in einer internen Datenbank gespeichert. Das Programm ist darauf ausgelegt, dass Falschmeldungen vom Programmierer manuell analysiert und bewertet werden. Alle ge-

[5] http://coccinelle.lip6.fr/

[6] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html

sammelten Falschmeldungen können als HTML– oder als TEX–Dokument exportiert werden, um einen Prüfbericht zu erzeugen.

Kapitel 5

Analyse der GeNUGate-Bestandteile

In diesem Kapitel werden die untersuchten Bestandteile des GeNUGates und deren gefundenen Schwachstellen erläutert. In der Tabelle 5.1 sind alle untersuchten Bestandteile, die Größe des Quellcodes und die Anzahl der im Rahmen dieser Arbeit gefundenen Fehler aufgelistet. Es wird jeweils exemplarisch ein Suchverfahren erklärt und was damit gefunden wurde. Eine genaue Auflistung aller gefundenen Fehlers und eine Erläuterung zu jedem Einzelfall befindet sich im Anhang.

Tabelle 5.1: Übersicht der analysierten Programme

Name	Sprache	Quellcodezeilen	Fehler	Lizens
Process–Master	C	1 847	5	proprietär
Accounting–Daemon	C	1 530	5	proprietär
Squid	C++	164 047	2	Open–Source
OpenBSD–Kernel	C	2 151 589	6	Open–Source
OpenBSD–Basissystem	C	7 766 113	4	Open–Source

5.1 Prozess–Master

Der *Prozess–Master* überwacht verschiedene Programme und das Logging des Firewall–Systems. Er wurde mit den Programmen *Cppcheck* und *Flawfinder* untersucht. Da diese Programme meist nur sehr allgemeine Hinweise gegeben haben, mussten diese sehr aufwändig manuell nachgeprüft werden. Dabei sind einige Schwachstellen gefunden worden. Unter anderem wurde ein mögliches Passwortleck gefunden.

Listing 5.1: Passwordleck im *Process–Master*

```
1  int pm_input_master_password(EVP_CIPHER *cipher) {
2          char *pass = NULL, *decrypted;
3          int tries = 3, fail = 0;
4
5          while( --tries >= 0 ) {
6                  pass = getpass("enter master password: ");
7  ...
8                  if( (decrypted = decryptString(cipher, pass, CFG
                      ->passwords)) != NULL ) {
9                          // Passwort ist richtig, decrypteten
                              String brauchen wir jetzt noch nicht
10                         memset(decrypted, 0, strlen(decrypted));
11                         // Passwort sichern
12                         pm_store_master_password(pass);
13                         memset(pass, 0, strlen(pass));
14                         return 0;
15                 } else {
16                         fail++;
17                 }
18         }
19 ...
20         pm_fatal(PM_WRONG_MASTER_PASSWD);
21         return -1;
22 }
```

Im Listing 5.1 wird eine Funktion, zum Einlesen eines Kennworts, gelistet. Der lokale Zeiger `pass` enthält die Benutzereingabe. Nach einer gültigen Eingabe, wird der Puffer von `pass` mittels `memset(3)` mit Nullen überschrieben. Wenn der Benutzer jedoch drei Mal ein falsches Kennwort eingibt, bleiben die Eingaben im Speicher zurück. Gelingt, es einem Dritten diesen Speicher auszulesen, kann er so an Kennwörter oder beim Vertippen des Benutzers Fragmente davon gelangen.

Das Programm *Flawfinder* hat den Hinweis aus Listing 5.2 gegeben, mit dem dieser Fehler gefunden wurde. Das Problem an dieser Meldung ist, dass sie immer angezeigt wird, sobald `getpass(3)` benutzt wird. In diesem Fall ist es so, dass das Quellcode–Analyseprogramm keine eigentlichen Fehler findet, sondern nur einen Verwendungshinweis für die Funktion `getpass(3)` gibt. Dadurch entsteht das Problem, dass nach der Korrektur der oben beschriebenen Schwachstelle, die Meldung des Programms weiter bestehen bleibt und zu einer Falschmeldung wird.

Listing 5.2: *Flawfinder*–Hinweis zu `getpass(3)`

```
pm/pmencrypt.c:283:  [4] (misc) getpass:
  This function is obsolete and not portable. It was in SUSv2
     but removed by POSIX.2.  What it does exactly varies
     considerably between systems, particularly in where its
     prompt is displayed and where it gets its data (e.g., /dev/
     tty, stdin, stderr, etc.). Make the specific calls to do
     exactly what you want.  If you continue to use it, or write
      your own, be sure to zero the password as soon as possible
      to avoid leaving the cleartext password visible in the
     process' address space.
```

5.2 Accounting–Daemon

Der *Accounting–Daemon* ist ein systemweiter und netzwerktransparenter *Loggin–*Dienst. Dieses Programm wurde auf die gleiche Art wie der *Process–Master* mit dem *Flawfinder* und dem *Cppcheck* überprüft. Mit dem Hinweis aus Listing 5.3 wird vom *Cppcheck* auf eine interessante Stelle im Quellcode verwiesen, an der mehrere Fehler zu finden sind.

Listing 5.3: *Cppcheck*–Hinweis zur *NULL*–Zeiger–Dereferenzierung

```
error nullPointer /home/jklemkow/code/acctd/testframes.h 73
Possible null pointer dereference: data - otherwise it is
   redundant to check if data is null at line 66
```

Zum einen wird in Zeile 73 auf einen *NULL*–Zeiger zugegriffen, sobald die Bedingungen in den Zeilen 66 oder 62 wahr werden. Da der Zeiger `pass` in beiden Fehlerbehandlungen auf *NULL* gesetzt wird und es vor Zuweisung in Zeile 73 keine Gültigkeitsprüfung gibt, wird es in beiden Fällen zu einer *NULL*–Zeiger–Dereferenzierung und damit zu einem Programmabsturz kommen.

Betrachtet man die anderen Stellen der Funktion `slurp()` genauer, findet man einen Datei–Handle–Leck sobald die Bedingung in Zeile 57 wahr wird. Zu diesem Zeitpunkt gibt es eine geöffnete Datei in der lokalen Variable `in`, welcher durch das Verlassen der Funktion bis zum Ende des Programms geöffnet und ungenutzt bleibt. Dieses Beispiel zeigt, dass eine gründliche Analyse von umgebenen Quellcode durch eine rein manuelle Prüfung zusätzliche Fehler hervorbringen kann. Da solch

ein Vorgehen sehr zeitaufwändig ist, sollte das Kosten–Nutzen–Verhältnis für die Qualitätssicherung eines Software–Projektes beachtet werden.

Listing 5.4: Doppelter Fehler im *Accounting–Daemon*

```
45 static char *
46 slurp(const char *fn)
47 {
48         struct stat st;
49         FILE *in;
50         char *data;
51
52         in = fopen(fn, "r");
53         if (!in) {
54                 warn("fopen");
55                 return NULL;
56         }
57         if (fstat(fileno(in), &st) < 0) {
58                 warn("stat");
59                 return NULL;
60         }
61         data = (char *) malloc(st.st_size+1);
62         if (data == NULL) {
63                 warn("malloc");
64                 data = NULL;
65         }
66         if (data && fread(data, st.st_size, sizeof(char), in) !=
               1) {
67                 warn("fread");
68                 free(data);
69                 data = NULL;
70         }
71         fclose(in);
72
73         data[st.st_size] = '\0';
74
75         return data;
76 }
```

5.3 Squid

Squid[1] ist ein Proxy–Server für das *Hypertext Transfer Protocol* (HTTP)[2]. Er ist das einzige *C++* Programm, welches in dieser Arbeit analysiert wurde. Bei der Analyse

[1] http://www.squid-cache.org/
[2] http://tools.ietf.org/html/rfc2616

wurden die Programme *Cppcheck*, *Flawfinder* und `grep(1)` eingesetzt. Eine Suche mit dem Programm `grep(1)` nach der Zeichenkombination „XXX“ ergab unter anderem den im Listing 5.5 aufgeführten Kommentar.

Listing 5.5: Auszug aus Datei src/url.cc von der Squid-Version 3.1.16

```
251 /* Then its :// */
252 /* (XXX yah, I'm not checking we've got enough data left before
        checking the array..) */
253 if (*src != ':' || *(src + 1) != '/' || *(src + 2) != '/')
254         return NULL;
```

An dieser Stelle ging der Programmierer davon aus, dass die zu untersuchende Zeichenkette in jedem Fall eine gewisse Mindestlänge haben wird, ohne dieses zu prüfen. Dadurch entsteht an dieser Stelle die Möglichkeit einer Speicherzugriffsverletzung. Das Listing 5.6 zeigt eine Längenprüfung des Puffers am Beginn der Bedingung, die das Problem behebt. In der Variablen `i` wird sich die Position des Zeigers `src` innerhalb eines Puffers gemerkt. Der Wert von `l` beinhaltet die Gesamtlänge des Puffers, welche zuvor mit der Funktion `strlen(3)` ermittelt wurde.

Listing 5.6: Loesung der Speicherzugriffsverletzung im *Squid*

```
251 /* Then its :// */
252 if (i+3 > l || *src != ':' || *(src + 1) != '/' || *(src + 2) !=
        '/')
253         return NULL;
```

5.4 OpenBSD–Kernel

Der OpenBSD–Kernel ist als Betriebssystemkern eine der zentralen Komponente des GeNUGate. Unter anderem verwaltet der Kernel alle Prozesse und die elementaren Netzwerkfunktionen. Zudem ist die OpenBSD–Firewall *Packet–Filter (PF)* im Kernel implementiert. Eine sicherheitsrelevante Schwachstelle in diesem Bereich wäre sehr kritisch. Bei der Analyse des Kernels wurden aus Zeitgründen auf eine Auswertung mit allgemeinen Quellcode–Analyse Programmen verzichtet, da das manuelle nachvollziehen der allgemeinen Hinweise im Rahmen dieser Arbeit aufgrund der Größe des zu überprüfenden Quellcodes (siehe Tabelle 5.1) nicht bewältigt werden konnte.

Häufige Fehler, welche in der Vergangenheit im OpenBSD–Kernel hin und wieder auftraten, sind Fehler beim Blockieren und Freigeben von *Interrupts*. Für das Blockieren von *Interrupts* werden die `spl(9)`–Funktionen benutzt. Die Funktion `splx(9)` gibt eine Interrupt Blockierung wieder frei. Bei der Suche nach Funktionen mit unsauberen *Interrupt*–Blockierungen hat sich das Programm *coccinelle*, mit der Möglichkeit eigene Suchmuster zu beschreiben, bewährt. Für das zuvor beschriebene Problem der fehlerhaften *Interrupt*–Blockierung wurde das Suchmuster aus Listing 5.7 entwickelt.

Listing 5.7: `splx(9)`-Suchmuster

```
1  @rule@
2  position p;
3  identifier f;
4  identifier spllock =~ "spl(high|serial|sched|clock|statclock|vm|
       tty|softtty|net|bio|softnet|softclock|lowersoftclock|0)";
5  identifier splfree =~ "splx";
6  @@
7
8  f (...)
9  {
10 ...
11 spllock
12 ... when != splfree
13 return ... ;@p
14 }
15
16 @script:python@
17 p << rule.p;
18 @@
19 print "missing␣splx:␣%s␣:␣%s" % (p[0].file, p[0].line)
```

Das Suchmuster besteht aus drei Bereichen. Im Kopfbereich, zwischen den Zeilen eins und sechs, werden Variablen mit verschiedene Bedeutungen definiert, welche dann im eigentlichen Suchmuster zwischen den Zeilen acht und 14 benutzt werden. Es werden verschiedene Bezeichner „identifier“ definiert, welche vordefinierte Namen besitzen oder allgemein sind. In diesem Fall wird nach einer allgemeinen Funktion `f` gesucht, dessen Bezeichner undefiniert ist. Innerhalb dieser Funktion wird nach der Verwendung eines Bezeichners mit den Namen der verschiedenen `spl(9)`–Funktionen gesucht. Wenn zwischen einem solchen Bezeichner und dem *C*–Keyword `return` kein `splx(9)` verwendet wird, dann trifft das Suchmuster

zu. Es wird sich mit dem Positionszeiger `@p` die Stelle des `return` innerhalb der analysierten Quellcode–Datei gemerkt und im Script–Teil verwendet. Der Script–Teil zwischen den Zeilennummern 16 und 19 ist in der Programmiersprache *Python* geschrieben und gibt eine Fehlermeldung mit dem Dateinamen und der Zeilennummern des gefunden Fehlers aus.

Mit diesem Suchmuster wurde unter anderem ein Fehler im Netzwerkkartentreiber `fxp(4)` gefunden.

Listing 5.8: Fehlerhafte *Interrupt*–Blockierung im `fxp(4)`–Treiber

```
1  void
2  fxp_init(void *xsc)
3  {
4  ...
5          s = splnet();
6  ...
7          if (!(cbp->cb_status & htole16(FXP_CB_STATUS_C))) {
8                  printf("%s: config command timeout\n", sc->
                       sc_dev.dv_xname);
9                  return;
10         }
11 ...
12         if (!(cb_ias->cb_status & htole16(FXP_CB_STATUS_C))) {
13                 printf("%s: IAS command timeout\n", sc->sc_dev.
                       dv_xname);
14                 return;
15         }
16 ...
17         splx(s);
18 ...
19 }
```

In Zeile sieben des Listings werden mit der Funktion `splnet(9)` die Netzwerk–*Interrupts* blockiert und der vorherige *Interrupt–Level* in der lokalen Variablen `s` gespeichert. Zum Ende der Funktion `fxp_init()` setzt die Funktion `splx(9)` das *Interrupt–Level* wieder auf den ursprünglichen Wert zurück. In den beiden Fehlerbehandlungen in Zeile elf und 16, wird die Funktion verlassen, ohne das *Interrupt–Level* zurück zu setzen oder vorherigen Status mitzuteilen. Somit kann die aufrufende Funktion es nicht zurücksetzen und eine Verarbeitung von Netzwerk–*Interrupts* ist nicht mehr möglich.

5.5 OpenBSD–Basissystem

Das OpenBSD–Basissystem sind alle Programme und Bibliotheken, welche beim einer Installation immer dabei sind und welche im Versionskontrollsystem des OpenBSD–Projektes selbst verwaltet werden. Es beinhaltet unter anderem eine Standard–*C*–Bibliothek, Unix–Benutzerprogramme und verschiedene System- und Netzwerkdienste. Wegen des enormen Umfangs des Quellcodes (siehe Tabelle 5.1) wurde aus Zeitlichen Gründen auf eine Analyse mit allgemeinen Quellcode–Analyseprogrammen verzichtet. Stattdessen wurde mit selbstentwickelten Suchmustern gearbeitet, um nach Fehlern zu suchen.

Ein Bestandteil des Basissystems ist das OpenSSH–Programm. Für dieses Programm wurde das Suchmuster aus Listing 5.9 entwickelt, welches nach Speicherlecks sucht.

Listing 5.9: Suchmuster für Speicherlecks im OpenSSH

```
1  @rule@
2  position p;
3  identifier f;
4  identifier bufinit =~ "buffer_init";
5  identifier buffree =~ "buffer_free";
6  @@
7
8  f (...)
9  {
10 ...
11 bufinit
12 ... when != buffree
13 return ... ;@p
14 }
15
16 @script:python@
17 p << rule.p;
18 @@
19 print "missing␣buffer_free:␣%s␣:␣%s" % (p[0].file, p[0].line)
```

Dieses Suchmuster funktioniert wie das Muster aus Listing 5.7. Es wird nach eine Funktion gesucht, in der Zwischen dem Aufruf von `buffer_init()` und einem `return` kein `buffer_free()` aufgerufen wird. Die Funktion `buffer_init()` alloziert einen Puffer, welcher dann von `buffer_free()` wieder freigegeben wird.

Diese beiden Funktionen kapseln `malloc(3)` und `free(3)`. Das macht es sehr schwierig ein allgemeines Suchmuster für Speicherlecks zu entwickeln.

Bei der Suche wurde das Speicherleck aus Listing 5.10 gefunden. Die Funktion `key_load_file()` bekommt einen Zeiger auf einen Puffer mit dem Bezeichner `blob` übergeben. Zu Beginn des Hauptteils dieser Funktion wird für den Puffer über den Aufruf der Funktion `buffer_init()`, Speicher alloziert. Die Funktion `key_load_file()` selbst, von den Funktionen `key_load_private_rsa1()`, `key_load_private_pem()`, `key_load_private_type()` und `key_load_private()` aufgerufen, die zu Beginn den Puffer allozieren und nach der Benutzung mit `buffer_free()` wieder frei geben. Durch den zweiten Aufruf der Funktion `buffer_init()` wird ein neuer Speicherbereich alloziert, auf den nun der Zeiger `blob` verweist. Der zuvor allozierte Speicher bleib bis zum Beenden des Programms ungenutzt zurück, da es keinen Verweis mehr auf ihn gibt.

Listing 5.10: Speicherleck im *OpenSSH*

```
1  int
2  key_load_file(int fd, const char *filename, Buffer *blob)
3  {
4  ...
5          buffer_init(blob);
6          for (;;) {
7  ...
8                  buffer_append(blob, buf, len);
9  ...
10         }
11 ...
12         if ((st.st_mode & (S_IFSOCK|S_IFCHR|S_IFIFO)) == 0 &&
13             st.st_size != buffer_len(blob)) {
14 ...
15                 buffer_clear(blob);
16                 return 0;
17         }
18
19         return 1;
20 }
21
22 Key *
23 key_load_private_pem(int fd, int type, const char *passphrase,
24     char **commentp)
25 {
26 ...
27         buffer_init(&buffer);
28         if (!key_load_file(fd, NULL, &buffer)) {
29                 buffer_free(&buffer);
30                 return NULL;
31         }
32         prv = key_parse_private_pem(&buffer, type, passphrase,
               commentp);
33         buffer_free(&buffer);
34         return prv;
35 }
```

Kapitel 6

Bewertung des praktischen Einsatzes

Im Idealfall sollte die statische Quellcode–Analyse ein fester Bestandteil des täglichen Entwicklungsprozesses sein, um diesen von Nacharbeiten und Fehlersuche zu entlasten. Dafür müssen einige Voraussetzungen erfüllt sein.

6.1 Verwaltung von Falschmeldungen

Eine immer wiederkehrende Analyse erfordert eine Verwaltung von Falschmeldungen und ein möglichst gutes Verhältnis von Falschmeldungen zu echten Fehlermeldungen. Falschmeldungen lassen sich leider nicht vermeiden. Sie treten auch dann auf, wenn nach einer korrigierten echten Meldung das Suchmuster immer noch auf die Stelle im Quelltext zutrifft. Das ist vor allem dann immer der Fall, wenn Analyse–Programme nur einen Hinweis geben, sobald eine bestimmte Funktion benutzt wird, ohne im Einzelfall zu überprüfen, ob eine echte Gefahr besteht. Aus diesem Grund braucht man ein Verwaltungssystem, welches Falschmeldungen handhabt.

6.1.1 Codecheck

Um die Verwaltung von Falschmeldungen zu untersuchen, wurde im Rahmen dieser Arbeit ein Verwaltungssystem für Falschmeldungen entwickelt. Mit diesem wurde prototypisch untersucht, wie eine solche Verwaltung aussehen könnte und welche weiteren Probleme sich ergeben. Dieses Programm *codecheck* nimmt die Meldungen verschiedener Quellcode-Analyseprogramme auf, normalisiert diese und speichert sie

in einer internen Datenbank. Diese Meldungen können dann vom Benutzer gesichtet und bewertet werden. Bei der Auflistung von Fehlern können bewertete Meldungen ausgeschlossen werden, damit ist das Ausblenden von bekannten Fehlermeldungen möglich. Einige Quellcode–Analyseprogramme bieten die Möglichkeit, durch Kommentare im Quellcode Fehlermeldungen zu unterdrücken. Dieses hat den Nachteil, dass der Quellcode unübersichtlich wird. Bei der Benutzung mehrerer Analyseprogramme verstärkt sich dieser Effekt. Das Programm *codecheck* umgeht dieses Problem, indem es die Meldungen mit Hilfe des Dateipfades und der Zeilennummern indiziert. Damit werden bei einer erneuten Analyse schon bekannte Meldungen den schon existierenden Bewertungen zugeordnet. Aus der Zuordnung von Meldungen mittels Dateipfad und Zeilennummer ergeben sich neue Probleme. Wenn nach der Bewertung eines Hinweises der Quelltext bearbeitet wird, dann verschieben sich die Zeilennummern und ein Hinweis, welcher schon bewertet wurde, kann nicht mehr zugeordnet werden. Somit erscheint der Hinweis erneut ohne eine Bewertung. Der vorherige Hinweis ist dann immer noch in der Datenbank und kann bei einer Sichtung der ursprünglichen Stelle im Quellcode zugeordnet werden. Für dieses Problem muss eine Lösung gefunden werden, bevor man statische Quellcode-Analyse in den Entwicklungsprozess integriert.

6.1.2 Zeilennummern relativ zur Funktion

Ein Lösungsansatz ist, die Zuordnung dahin zu verändern, die Zeilennummer relativ zum Beginn einer Funktion in Verbindung mit deren Bezeichner zur Identifikation eines Hinweises zu benutzen. Damit könnte ein Programmierer weite Teile einer Quellcode-Datei bearbeiten, ohne, durch das Verschieben von Zeilennummern, Analyse-Hinweise mehrfach bewerten zu müssen. Dieser Ansatz wäre sehr von der zu untersuchenden Programmiersprache abhängig, da durch Spracheigenschaften wie das Überladen und Überschreiben von Funktionen die Identifikation einer Funktion innerhalb einer Quellcode–Datei sehr erschwert wird.

6.1.3 Zeilennummern neu berechnen

Ein effizienterer und sprachunabhängiger Ansatz ist es, die Änderung eines Programmierers, den sogenannten *Diff*, auf das Verschieben von Zeilennummern zu untersuchen, und die Verschiebung innerhalb einer Datei mit den Zeilennummern der bekannten Hinweise in der Datenbank zu verrechnen.

Diese Lösungsansätze sollten in aufbauenden Arbeiten weiter untersucht werden.

6.2 Wenig Falschmeldungen

Da jede Meldung, die ein Quellcode–Analyseprogramm liefert, manuell überprüft werden muss, sollte eine hohe Rate an Falschmeldungen vermieden werden. Dafür sollten die Analyseprogramme so eingestellt werden, dass sie möglichst keine Falschmeldungen liefern und bei Produkten von Drittanbietern Fehlerarten wie stilistische Fehler nicht melden. Der Prüfer vernachlässigt sonst schnell seine Gründlichkeit und übersieht echte Fehler, welche dann als Falschmeldungen ignoriert werden. Fehlermeldungen, welche nur sehr allgemein die Möglichkeit eines Fehlers beschreiben, müssen gründlicher untersucht werden als sehr konkrete Meldungen.

6.3 Projektspezifische Suchmuster

Die Quellcode–Analyseprogramme *Flawfinder*, *RATS* und *Cppcheck* sind dafür geeignet, um allgemeine Fehler in *C* und *C++* Programmen zu finden. Bekannte Design- und Implementierungsfehler der Standardbibliothek und Probleme mit anderen Sprachelementen sind die Grundlagen, mit denen diese Programme erstellt wurden.

In Software–Projekten werden nicht nur diese Standardelemente der Programmiersprachen verwendet, sondern auch eigenen Funktionen und Mechanismen, welche ebenfalls Design- und Implementierungsfehler haben. Darum reicht es nicht, nach Standardfehlern zu suchen. Die Suche nach Fehlern bei der Benutzung eigener oder Dritter Komponenten ist daher unverzichtbar. Zudem werden bekannte Fehler durch Kapseln von Standardfunktionen nicht mehr von Analyseprogrammen gefun-

den, die auf die Standardbibliothek spezialisiert sind. Das Beispiel der Kapselung von `malloc(3)` und `free(3)` im *OpenSSH* sowie die *Interrupt*–Blockierung durch `spl(9)` im OpenBSD–Kernel zeigt, dass ohne auf das eigene Projekt angepasste Suchverfahren nur eine oberflächliche Quellcode–Analyse möglich ist.

Kapitel 7

Zusammenfassung

Mit freien statische Quellcode–Analyseprogrammen lassen sich auch schwere Fehler in sicherheitskritischen Software–Systemen finden und damit die allgemeine Quellcode–Qualität solcher Systeme steigern. Der Einsatz dieser Art von Qualitätssicherung ist in der heutigen Software–Entwicklung unverzichtbar, da es immer wieder zu Fehlern kommt, welche lange bekannt sind und sich im Laufe der Zeit wiederholen. So wurden die Fehler der *Interrupt*–Blockierung im OpenBSD–Kernel schon im Jahr 2001 von Mitarbeitern der Firma Sun–Microsystems gefunden, welche zu diesem Zeitpunkt an dem kommerziellen statischen Quellcode–Programm *Parfait* [3] gearbeitet haben. Will man solche bekannten Fehler zukünftig dauerhaft vermeiden, so ist man gezwungen, regelmäßig Quellcode–Analysen durchzuführen. Idealerweise findet dieses im täglichen Entwicklungsprozess statt. Um dieses auch praktisch realisieren zu können, müssen noch einige Fragen, die diese Arbeit in dem Zusammenhang offen lässt, beantwortet werden. Für eine möglichst tiefgründige Suche nach Fehlern, darf das Set an Suchmustern nicht statisch sein und muss ständig neuen und projektspezifischen Gegebenheiten angepasst werden.

Kapitel 8

Thesen

Aus der vorliegenden Arbeit haben sich folgende Thesen ergeben:

- Bei der automatischen statischen Quellcode–Analyse ist ein Methodik zur Handhabung von Falschmeldungen wichtig.
- Statische Quellcode–Analyse hebt die Qualität von Software an, wenn sie in den Entwicklungsprozess eingebunden wird.
- Eine effektive statische Quellcode–Analyse muss ständig an das zu untersuchende Softwareprojekt angepasst werden.
- Freie Quellcode–Analyseprogramme sind in der Lage auch bedeutende Fehler in Software–Systemen zu finden.
- Der Nutzen von statischer Quellcode–Analyse rechtfertigt den Aufwand nicht nur bei Sicherheitskritischen Software–Systemen.

Kapitel 9

Anhang

Dieser Arbeit liegt eine CD-ROM bei, auf der sich die Arbeit selbst, Quellcode–Listings, Suchmuster sowie Teile der verwiesenen Literatur befinden.

9.1 Gefundene Fehler

In diesem Abschnitt sind alle Fehler aufgelistet, welche im Rahmen dieser Arbeit gefunden wurden.

Tabelle 9.1: Passwordleck im GeNUGate *Process–Master*

Programmhinweis von *Flawfinder*:
`pm/pmpass.c:150: [4] (misc) getpass:` `This function is obsolete and not portable. It was in SUSv2 but removed by POSIX.2. What it does exactly varies considerably between systems, particularly in where its prompt is displayed and where it gets its data (e.g., /dev/tty, stdin, stderr, etc.). Make the specific calls to do exactly what you want. If you continue to use it, or write your own, be sure to zero the password as soon as possible to avoid leaving the cleartext password visible in the process' address space.`
Bewertung:
In der Funktion `pm_input_master_password()` wird versucht drei mal ein Passwort vom Benutzer einzulesen. Wenn das eingegeben Passwort einmal richtig ist, dann wir das *Char–Array* mit Nullen über schrieben und die Funktion verlassen. Bei drei Fehlversuchen wird das Passwort im Speicher belassen und wäre potentiell auslesbar hinterher.
Empfehlung:
Das `memset(3)` der Zeile 160 sollte auch nach der Eingabe eines falschen Passwortes aufgerufen werden, um ein Passwortleck zu verhindern.

Tabelle 9.2: Pufferüberlauf im *Squid*

Programmhinweis von *Flawfinder*:

`error, insecureCmdLineArgs,`
`squid-3.1.16/src/recv-announce.cc:102`
`Buffer overrun possible for long cmd-line args`

Bewertung:
In der Funktion `main()` wird ein statische Puffer definiert. In dieses wird dann der erste Parameter des Programms kopiert, ohne dabei auf die vorher statisch festgelegte Puffergröße zu achten. Dieses kann bei einem geeignet großem Parameter zu einem Pufferüberlauf führen.
Empfehlung:
Der statische Puffer kann durch einen Zeiger ersetzt werden, welcher dann auf das erste Argument zeigt.

Tabelle 9.3: *NULL*–Zeiger–Dereferenzierung im GeNUGate *Accounting–Daemon*

Programmhinweis von *Cppcheck*:
`error nullPointer acctd/testframes.h 73` `Possible null pointer dereference: data - otherwise it is redundant to check if data is null at line 66`
Bewertung:
In dieser Funktion gibt es zwei Fehler. Eine NULL–Zeiger–Dereferenzierung in Zeile 73, sobald das `malloc(3)` oder das `fread(3)` fehlschlagen. Zudem ein Datei-Handle-Leck, sobald `fstat(2)` fehlschlägt.
Empfehlung:
Vor dem Zugriff auf das *Array* `data` am Ende der Funktion, eine Prüfung auf *NULL* einfügen. Zudem sollte in der Fehlerbehandlung von `fstat(2)` die geöffnete Datei vor dem Beenden Funktion geschlossen werden.

Tabelle 9.4: *NULL*–Zeiger–Dereferenzierung im GeNUGate *Process–Master*

Programmhinweis von *Cppcheck*:

`file: pm/pmsys.c`
`line: 86`
`level: 0`
`severity: error`
`ID: nullPointer`
`text: Possible null pointer dereference: ci - otherwise it is redundant to check if ci is null at line 84`

Bewertung:
An dieser Stelle findet eine *NULL*–Zeiger–Dereferenzierung statt, sobald die Bedingung `ci != NULL` wahr wird.
Empfehlung:
Der Aufruf von `debug()` sollte in den Körper der vorherigen Bedingung aufgenommen werden.

Tabelle 9.5: Format–String–Schwachstelle im GeNUGate *Accounting–Daemon*

Programmhinweis von *Flawfinder*:

`file: acctd/Conversation.c`
`line: 83`
`level: 4`
`severity: format`
`id: vsnprintf`
`program: flawfinder 1.27`
`text: If format strings can be influenced by an attacker, they can be exploited, and note that sprintf variations do not always \0-terminate. Use a constant for the format specification.`

Bewertung:
An drei Stellen im Code wird die Funktion `output()` unsicher benutzt und ermöglicht einen möglichen Format–String–Angriff.
Empfehlung:
Die Funktion `output()` sollte immer mit dem Format–String „`%s`“ aufgerufen werden.

Tabelle 9.6: Speicherzugriffsverletzung im GeNUGate *Process-Master*

Programmhinweis von *RATS*:

`file: pm/pmlog.c`
`line: 295`
`level: 5`
`short: High fixed size global buffer`
`program: rats 2.3`
`time: 2011-11-14 17:19:07`
`long: Extra care should be taken to ensure that character arrays that areallocated on the stack are used safely. They are prime targets forbuffer overflow attacks.`

Bewertung:
In Zeilennummer 306 tritt ein Off–By–One-Fehler auf. Es wird versucht einen \n und eine \0 an das Ende des String zu schreiben.
Empfehlung:
Bei der Bestimmung der Größe von `output` sollte für die terminierende Null ein eins abgezogen werden.

Tabelle 9.7: Format–String–Schwachstelle im GeNUGate *Process–Master*

Programmhinweis von *RATS*:

`file: pm/pm.c`
`line: 767`
`level: 5`
`short: High fixed size global buffer`
`program: rats 2.3`
`time: 2011-11-14 17:19:07`
`long: Extra care should be taken to ensure that character arrays that areallocated on the stack are used safely. They are prime targets forbuffer overflow attacks.`

Bewertung:
Die Pufferlänge wird korrekt beachtet, aber beim Aufruf der Funktion debug(buf) gibt es einen möglichen Format–String–Angriff.
Empfehlung:
Die Funktion `debug()` sollte mit dem Format–String „`%s`“ aufgerufen werden.

Tabelle 9.8: Format–String–Schwachstelle im GeNUGate *Logmonitor*

Programmhinweis von `grep(1)`:

`# grep -Rn "ggdebug([[:alpha:]]" */*.c`
`logmonitor/logmonitor.c:740:  ggdebug(buf);`
`logmonitor/logmonitor.c:752:  ggdebug(buf);`
`logmonitor/message.c:37:ggdebug(const char *fmt, ...)`

Bewertung:
Die Funktion `ggdebug()` kapselt ein Format–String–Funktion und ermoeglicht mit der Verwendung in den Zeilen 740 und 752 der Datei `logmonitor.c` einen Format–String–Angriff, wenn es einem Angreifer gelingt, den Inhalt der Variable `buf` zu beeinflussen.
Empfehlung:
Der Funktionsaufruf `debug(buf)` sollte in `debug("%s", buf)` geändert werden.

Tabelle 9.9: Speicherzugriffsverletzung im *Squid*

Programmhinweis von `grep(1)`:

`# grep -Rn XXX *`
`...`
`url.cc:252: /* (XXX yah, I'm not checking we've got enough data left before checking the array..) */`

Bewertung:
Beim *parsen* einer URL wird beim Trenner „`://`“ einfach voraus gesetzt. Beim zweiten Schrägstrich findet ein Zugriff ausserhalb des Puffers statt, wenn die zu *parsende* Zeichenkette nur ein `http:` enthält.
Empfehlung:
Es sollte eine Längenprüfung an dieser stellen eingebaut werden.

Tabelle 9.10: Fehlerhafte Blockierung im OpenBSD–Kernel

Programmhinweis von *coccinelle* mit Suchmuster 9.2:

`missing splx: sys/dev/ic/pgt.c : 2052`
`missing splx: sys/dev/ic/fxp.c : 1170`
`missing splx: sys/dev/ic/trm.c : 294`
`missing splx: sys/arch/solbourne/solbourne/pmap.c : 911`
`missing splx: sys/arch/solbourne/solbourne/pmap.c : 1209`
`missing splx: sys/dev/usb/if_wi_usb.c : 1883`

Bewertung:
An den angegebenen Stellen im Quellcode wird jeweils eine `spl(9)` Funktion aufgerufen um eine *Interrupt–Level* zu verändern. In verschiedenen Fehlerbehandlungen, wird die Funktion verlassen, ohne das *Interrupt–Level* mittels `splx(9)` wieder zurückzusetzen.
Empfehlung:
In den Fehlerbehandlungen, in denen Funktion verlassen wird, sollte das vorherige `Interrupt-Level()` mit `splx(9)` wieder zurückgesetzt werden.

Tabelle 9.11: Speicherleck im OpenBSD–Basissystem im *OpenSSH*–Client

Programmhinweis von *coccinelle* mit Suchmuster 9.6:
`missing buffer_free: /usr/src/usr.bin/ssh/` `ssh-pkcs11-client.c  : 127`
Bewertung:
Die Funktion `pkcs11_rsa_private_encrypt()` alloziert einen Puffer und vergisst dieses vor dem verlassen der Funktion wieder freizugeben.
Empfehlung:
Vor vor dem Verlassen dieser Funktion sollte der Puffer mittels `buffer_free()` wieder freigegeben werden.

Tabelle 9.12: Speicherleck im OpenBSD–Basissystem im *OpenSSH*–Client

Programmhinweis von *coccinelle* mit Suchmuster 9.6:
`missing buffer_free: /usr/src/usr.bin/ssh/authfile.c : 330`
Bewertung:
Die Funktion `key_load_file()` wird immer mit einem schon alloziertem Puffer aufgerufen. Dieser wird dann von den aufrufenden Funktionen auch wieder frei gegeben. Die Funktion selbst, alloziert nun wieder Speicher für den übergebenen Puffer. Dadurch wird der Speicher beim ersten Allozieren in der aufrufenden Funktion nie frei gegeben und endet als Speicherleck.
Empfehlung:
Der Aufruf der Funktion `buffer_init()` innerhalb von `key_load_file()` sollte gelöscht werden.

Tabelle 9.13: Datei–Handle–Leck im OpenBSD–Basissystem im *NSD*

Programmhinweis von *coccinelle* mit Suchmuster 9.3:
`missing fclose(3): /usr/src/usr.sbin/nsd/nsd-patch.c : 171`
Bewertung:
In der Funktion `exist_difffile()` wird überprüft ob eine Datei existiert, in dem sie mit `fopen(3)` geöffnet wird um dann den Datei–Handle auf ungleich *NULL* zu prüfen. Dabei wurde das `fclose(3)` vergessen. Somit bleibt das Datei-Handle über die gesamte Prozesslaufzeit unbenutzt offen.
Empfehlung:
Im einfachen Fall könnte man jeweils vor die beiden *returns* der Funktion ein `fclose(3)` setzen. Man könnte sich aber auch überlegen, die Funktion durch *access*2 oder `stat(2)` zu ersetzen.

Tabelle 9.14: Datei–Handle–Leck im OpenBSD–Basissystem im *authpf*

Programmhinweis: *coccinelle* mit Suchmuster 9.3:
`missing close: /usr/src/usr.sbin/authpf/authpf.c : 475`
Bewertung:
In der Funktion `allowed_luser()` wird eine Datei-Handle mit `fopen(3)` geöffnet ohne es mit einem `fclose(3)` wieder zu schliessen.
Empfehlung:
Vor dem Beenden der Funktion `allowed_luser()` sollte die Datei mittels `fclose(3)` wieder geschlossen werden.

Tabelle 9.15: Datei–Handle–Leck im OpenBSD–Basissystem im `vi(1)`

Programmhinweis von *coccinelle* mit Suchmuster 9.3:
`missing close: /usr/src/usr.bin/vi/common/recover.c : 794`
Bewertung:
In dieser Funktion gibt es zum Aufruf von `open(2)` kein zugehöriges `close(2)`.
Empfehlung:
Es wird empfohlen eine `close(2)` hinzuzufügen.

9.2 Entwickelte Suchmuster

Hier sind die Listings aller im Rahmen dieser Arbeit entwickelten Suchmuster aufgelistet.

Listing 9.1: Suchmuster für Fehlerhafte Blockierung mit `mutex(9)`

```
1  @rule@
2
3  position p;
4
5  identifier f;
6  identifier spllock =~ "mtx_enter";
7  identifier splfree =~ "mtx_leave";
8  @@
9
10 f (...) {
11 ...
12 spllock
13 ... when != splfree
14 return ... ;@p
15 }
16
17 @script:python@
18 p << rule.p;
19 @@
20
21 print "missing␣mtx_leave:␣%s␣:␣%s" % (p[0].file, p[0].line)
```

Listing 9.2: Suchmuster für Fehlerhafte Interrupt–Blockierung mit `spl(9)`

```
@rule@
position p;
identifier f;
identifier spllock =~ "spl(high|serial|sched|clock|statclock|vm|
   tty|softtty|net|bio|softnet|softclock|lowersoftclock|0)";
identifier splfree =~ "splx";
@@

f (...)
{
...
spllock
... when != splfree
return ... ;@p
}

@script:python@
p << rule.p;
@@
print "missing splx: %s : %s" % (p[0].file, p[0].line)
```

Listing 9.3: Suchmuster für eine Datei–Handle–Leck

```
@rule@
position p;
identifier f;
identifier open =~ "open";
identifier close =~ "close";
@@

f (...)
{
...
open
... when != close
return ... ;@p
}

@script:python@
p << rule.p;
@@
print "missing fclose(3): %s : %s" % (p[0].file, p[0].line)
```

Listing 9.4: Suchmuster für fehlerhaften `malloc(9)` Aufruf

```
@rule@

position p;

identifier malloc =~ "malloc";

identifier flags =~ "";

expression E1, E2;
type T;

@@

(
 E2 = (T) malloc(E1, flags, types);@p
|
 malloc(E1, flags, types);@p
|
 return malloc(E1, flags, types);@p
)

@script:python@
p << rule.p;
@@

print "wrong malloc(9): %s : %s" % (p[0].file, p[0].line)
```

Listing 9.5: Suchmuster für Speicherleck mittels `realloc(3)`

```
@rule@
position p;
identifier realloc =~ "realloc";
identifier A;
expression E1;
type T;
@@

A = (T) realloc(A, E1);@p

@script:python@
p << rule.p;
@@
print "wrong realloc(3): %s : %s" % (p[0].file, p[0].line)
```

Listing 9.6: Suchmuster für Speicherleck im *OpenSSH*

```
@rule@
position p;
identifier f;
identifier bufinit =~ "buffer_init";
identifier buffree =~ "buffer_free";
@@

f (...)
{
...
bufinit
... when != buffree
return ... ;@p
}

@script:python@
p << rule.p;
@@
print "missing buffer_free: %s : %s" % (p[0].file, p[0].line)
```

Abkürzungsverzeichnis

BSD ..... Berkeley Software Distribution

DoS ..... Denial of Service

GeNUA ..... Gesellschaft für Netzwerk- und Unix–Administration

HTTP ..... Hypertext Transfer Protocol

LLVM ..... Low Level Virtual Machine

RATS ..... Rough Auditing Tool for Security

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. *BegBunch – Benchmarking for C Bug Detection Tools.* Sun Microsystems Laboratories, July 2009.

[2] Ian F. Darwin. *Checking C Programs with lint.* O'Reilly, 1988.

[3] Daniel Dawson, Nathan Hawes, Christian Hoermann, Nathan Keynes, and Cristina Cifuentes. *Finding Bugs in Open Source Kernels using Parfait.* Sun Microsystems Laboratories, 2009.

[4] Christian Ehrhardt. *Static Code Analysis in Multi-Threaded Environments.* PhD thesis, Fakultät für Mathematik und Wirtschaftswissenschaften der Universität Ulm, Oktober 2007.

[5] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code.* Addison-Wesley Professional, 2004.

[6] Stephen Johnson. Lint, a c program checker. In *Computer Science Technical Report 65.* December 1977.

[7] E.S. Raymond. *The Art of Unix Programming.* Addison-Wesley professional computing series. Addison-Wesley, 2004.

[8] D.J. Worth, C. Greenough, and L.S. Chin. *A Survey of C and C++ Software Tools for Computational Science.* Software Engineering Group, Computational Science and Engineering Department, STFC Rutherford Appleton Laboratory, Harwell Science and Innovation Campus, Didcot, December 2009.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig, ohne unerlaubte fremde Hilfe und nur unter Verwendung der aufgeführten Hilfsmittel angefertigt habe.

Ort, Datum Unterschrift