Proseminar

Detecting Buffer Overruns Using Static Analysis

Marcel Johannfunke

31. Januar 2014

Technische Universität Dortmund
Fakultät Informatik

Lehrstuhl 14 Software Engineering – Prof. Dr. Jan Jürjens

betreut durch: Prof. Dr. Jan Jürjens

Marcel Johannfunke

Marcel.Johannfunke@udo.edu
Matrikelnummer: 148617
Studiengang: Bachlor Informatik

Werkzeugunterstützung für sichere Software
Thema: Detecting Buffer Overruns Using Static Analysis

Eingereicht: 16.12.2013

Betreuer: Prof. Dr. Jan Jürjens

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Sämtliche aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Dortmund, den 31. Januar 2014 _____

Unterschrift

Abstract

We describe a design idea of a tool to find buffer overruns in C source code using static analysis. We translate the detection of buffer overruns into an integer range problem which we will solve with linear programming. There we will see two different approaches how to solve the linear program we create.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Zuerst werden wir den Grund betrachten, wieso es überhaupt wichtig ist, sich mit dem Thema der Pufferüberläufe zu beschäftigen. Danach legen wir fest, was erreicht werden soll und zum Schluss wie wir die so gesteckten Ziele erreichen.

1.1 Motivation

„Buffer overflow attacks may be today's single most important security threat." ([LE01] S.1) Im Jahr 2013 hat Yves Younan in seinem Bericht für Sourcefire geschrieben: „It is now safe to declare the buffer overflow the vulnerability of the quartercentury." ([Yve12] S.7) Es war und ist also der größte bekannte Typ einer Sicherheitslücke.

1.2 Ziele und Nutzen

Um die Sicherheitslücke, die die Ausnutzung eines Pufferüberlaufs darstellt und die von einem einzelnen unachtsamen Programmierer erzeugt werden kann (vgl. [GJC+03] S.1, [DJN+10] S.35), flächendeckend zu schließen, stellen wir nun das Design für ein Tool vor, um Pufferüberläufe zu finden.

1.3 Aufbau der Arbeit

Dazu werden wir in Kapitel 2 die Grundlagen der statischen Analyse und linearen Programmierung erklären. In dem Kapitel 3 werden wir lineare Einschränkungen für die Variablen im Quellcode erzeugen und uns mit Zeigern beschäftigen, sowie mit den Makeln, die dabei auftreten können. Zuletzt zeigen wir im Kapitel 4, wie wir die erzeugten linearen Einschränkungen lösen und somit Werte für die einzelnen Einschränkungen der Variablen des Quellcodes ermitteln können. Mit diesen können wir dann bestimmen, ob es Pufferüberläufe gegeben hat oder nicht.

2 Grundlagen

Mit Hilfe der statischen Analyse können die verschiedenen Zustände eines Programmes ermittelt werden, die beim Ausführen auftreten können, ohne das Programm auszuführen. Für diese Ausarbeitung reicht es zu wissen, dass sie auf dem Quellcode arbeitet. Genauere Informationen dazu finden sich in [DJN+10] im Kapitel 2.1.

Ein lineares Programm ist ein Optimierungsproblem, dass wie folgt ausdrückt werden kann:

$$\begin{aligned} \text{Minimize} &: cx \\ \text{Subject To} &: Ax \geq b \end{aligned}$$

wo A eine m×n Matrix aus Konstanten, b und c Vektoren aus Konstanten und x ein Vektor aus Variablen ist. Lineare Programmierung rechnet auf endlichen realen Zahlen. Die Variablen in x dürfen entsprechend auch nur endliche reelle Werte annehmen und daher ist auch jede Optimierung, wenn sie existiert, aus endlichen Werten bestehend. Auch hierzu finden sich weitere Details in [DJN+10] im Kapitel 3.2.

3 Einschränkung für Variablen generieren

Um Pufferüberläufe zu finden, benötigen wir eine Darstellung für unsere Variablen, anhand welcher wir Überläufe erkennen können. Wir nutzen dafür die in [DJN+10][GJC+03] genutzte Darstellung, bei der für jede Variable 4 Einschränkungen gesetzt werden. Für einen Zeichenpuffer `buf` gibt es `buf!used!min` und `buf!used!max`, die die minimale und maximale Nutzung des Puffers in Bytes angeben, sowie `buf!alloc!min` und `buf!alloc!max`, die die minimale und maximale Allokation des Puffers in Bytes beschreiben.

Für eine Integer Variable `i` nutzen wir die beiden Einschränkungen `i!min` und `i!max`, welche den minimalen und maximalen Wert für `i` darstellen. Programmaufrufe werden durch lineare Einschränkungen und nicht Einschränkungsvariablen modelliert. Diese werden im folgenden Einschränkungsvariablen genannt.

Diese Einschränkungsvariablen sind jedoch weder fluss- noch kontextsensitiv. Die fehlende Flusssensitivität bedeutet, dass die Reihenfolge der Aufrufe nicht beachtet wird, und aus der nicht vorhanden Kontextsensitivität folgt, dass mehrere Aufrufe derselben Funktion nicht unterschieden werden.

Alle Einschränkungsvariablen werden durch einen Lauf über den gesamten Quellcode generiert. Dabei werden 4 Anweisungen betrachtet: Deklarationen, Zuweisungen, Funktionsaufrufe und Rückgabeanweisungen. Eine Deklaration eines Puffers wirkt sich auf die Länge des allokierten Puffers aus. Wenn es für einen Zeichenpuffer `buf` nur die Deklaration `char buf[1024]` gibt, folgt daraus $\texttt{buf!alloc!min} \leq 1024$ und $\texttt{buf!alloc!max} \geq 1024$. Anweisungen wie `buf[i]='c'` haben Auswirkungen auf die Einschränkungen `buf!used!min` und `buf!used!max`. Anweisungen, die eine ganzzahlige Variable `i` verändert, wirken sich auf `i!min` und `i!max` aus. Diese Ungleichungen werden auch lineare Einschränkungen genannt.

Funktionsaufrufe der Standardbibliothek von C werden durch ihren Effekt auf den genutzten Puffer modelliert. Zum Beispiel zählen dazu die Funktionen `strcpy` und `strcat`. Ein Aufruf `strcpy(puffer, quelle)` führt zu den folgenden linearen Einschränkungen:

$$\texttt{puffer!used!max} \geq \texttt{quelle!used!max}$$
$$\texttt{puffer!used!min} \leq \texttt{quelle!used!min}$$

Aber selbst die „sicheren“ Methoden der C String API sind nicht unbedingt sicher, sondern müssen auch richtig angewendet werden. Genauere Ausführungen dazu finden sich in [WFB+00].

Für alle von einem Benutzer angelegte Funktionen, deren Rückgabewert ein Integer oder Zeichenpuffer ist, werden Einschränkungen für den Rückgabewert generiert. Für eine Funktion `foo` folgen die Einschränkungen dem Muster `foo$return!alloc!min`. Die Einschränkungen der Parameter einer benutzerdefinierten Funktion können somit ähnlich wie die Funktionsaufrufe der Standardbibliothek beschränkt werden.

Auf Grund der Ignorierung des Kontrollflusses kommt es zu Problemen. So werden die Einschränkungen in und um `if`-Anweisungen und `for`-Schleifen nicht immer richtig berechnet. Als einfaches Beispiel soll hier dienen:

```
for (int i=0; i<10; i++)
    ;
```

Abbildung 1: Codebeispiel for-Schleife

Das `i` wegen der Bedingung `i<10` nur bis 10 läuft wird nicht weiter beachtet. Weiterhin würde die Anweisung `i++` direkt übersetzt ergeben:

$$i!max \geq i!max + 1$$

Diese lineare Einschränkung kann jedoch nicht von einem Solver für lineare Programme interpretiert werden. Deswegen modellieren wir diese Anweisung wie die zwei getrennte Anweisungen `i‘ = i + 1` und `i = i‘`. Die durch die zwei Anweisungen generierten Einschränkungen können von einem Solver interpretiert werden, auch wenn sie unlösbar sind. Unlösbare Programme werden in Kapitel 3.2 und 4.2 weiter betrachtet.

Ein weiteres Problem sind Werte für Variablen, die durch die Umgebung oder den Nutzer erstellt werden, wenn sie über einen unsicheren Weg ermittelt werden. So kann der Aufruf `getenv(„PATH“)` einen beliebig langen String zurückgeben. Um dies zu modellieren werden die Einschränkungen $getenv\$return!used!max \geq \infty$ und $getenv\$return!used!min \leq 0$ generiert. Ein ganzzahliger Wert `i`, der durch den Nutzer eingegeben wird, wird entsprechend mit $i!max \geq \infty$ und $i!min \leq -\infty$ beschrieben.

3.1 Handhabung von Zeigern

Idealer Weise sollten die generierten Einschränkungen in sich schlüssig und robust sein. Wenn eine Einschränkung für die Variable `var` gegeben ist, so sollte die Einschränkung alle möglichen Werte für `var` umfassen. Jedoch ist die vorgestellte Darstellung für Einschränkungen unsicher unter Nutzung von Aliasen und Zeigern.

Es werden für jedes einzelne Alias Einschränkungen erstellt und festgelegt, jedoch werden diese nicht zusammengefasst, auch wenn sie alle den gleichen Puffer bezeichnen. Daraus können falsch negative Resultate entstehen, also Pufferüberläufe nicht erkannt werden. So wird der Pufferüberlauf in dem folgenden Codefragment aus [WFB+00] nicht erkannt, bei dem ein 13-Byte String in ein 10-Byte Puffer kopiert wird:

```
Char s[20], *p, t[10]
 strcpy(s, „Hello“);
      p = s + 5;
strcpy(p, “ world!”);
    strcpy(t, s);
```

Abbildung 2: Nicht erkannter Pufferüberlauf

Was aber auf keinen Fall ignoriert werden kann sind benutzerdefinierte Structs. Weil diese die einzige Möglichkeit sind für Abstraktion oder Erzeugung von Datenstrukturen, würde die Ignorierung dieser zur Nutzlosigkeit einer Überprüfung nach Pufferüberläufen bei realen Programmen führen. Eine Lösung des Problems ist der folgende Ansatz: Wenn ein Zeiger `p` auf ein Struct `s` zeigt, welches einen Zeichenpuffer `buf` enthält, so werden alle Aufrufe von `p->buf` in Einschränkungen `s.buf` übersetzt.

3.2 Makel identifizieren und entfernen

Wie schon in Kapitel 3 erwähnt, können die generierten Einschränkungen teilweise unlösbar sein. Lineare Programmierung kann nur mit endlichen Werten arbeiten, und weiter ist es wichtig, dass die `max` Einschränkungen eine endliche untere Schranke besitzen und die `min` Einschränkungen eine endliche obere Schranke. Daher werden nun zwei Aspekte betrachtet:

Identifizieren und entfernen aller Einschränkungen, die unendliche Werte annehmen: Alle Einschränkungen `es`, die einen unendlichen Wert annehmen können ($es \geq \infty$ oder $es \leq -\infty$), müssen entfernt werden.

Identifizieren und entfernen aller nicht initialisierten Einschränkungen: Alle Einschränkungen müssen der Bedingung genügen, dass sie endliche untere und obere Schranken für die entsprechenden `max` und `min` Einschränkungen besitzen. Einschränkungen, die diese Bedingung nicht erfüllen, bezeichnen wir als nicht initialisiert. Das kann dadurch geschehen, dass die entsprechenden Variablen im Quellcode nie initialisiert wurden oder dass Anweisungen, die den Wert der Variable bestimmen, nicht erfasst wurden. Der zweite Fall kann durch Systemfunktionsaufrufe geschehen, für die der Effekt auf die Variable nicht bekannt ist.

4 Lösung der Einschränkungen mit Hilfe von Linearer Programmierung

Nachdem nun Makel identifiziert und entfernt wurden, können die Einschränkungen, die nun weiterhin bestehen, mit Hilfe von Linearer Programmierung gelöst werden. Dazu werden die zwei verschiedenen Methoden aus [DJN+10][GJC+03] vorgestellt. Die Erste nutzt einen Solver auf dem kompletten Satz von Einschränkungen um die Lösung jeder einzelnen Einschränkung zu berechnen. Die zweite Methode analysiert und reduziert den kompletten Satz in kleinere Untersätze.

Das Ziel ist bei beiden das gleiche: Erreichen der bestmöglichen Ergebnisse für die Anzahl der genutzten Bytes aller Puffer und Werte aller ganzzahligen Variablen.

4.1 Direkte Anwendung auf Einschränkungen

Da unsere generierten Einschränkungen lineare Einschränkungen sind, können sie auch als lineares Programm formuliert werden. Wir wollen die bestmöglichen Ergebnisse für unsere Einschränkungen erreichen, die alle Abhängigkeiten der jeweiligen Einschränkung beachten. Bestmöglich gilt bei den Einschränkungen, die einen minimalen Wert beschreiben (`buf!alloc!min`), der größte mögliche Wert und bei solchen, die einen maximalen Wert beschreiben (`buf!alloc!max`), der kleinste mögliche Wert. Dass es diese gibt, haben wir in Kapitel 3.2 sichergestellt, denn sonst würden sie als nicht initialisiert gelten. Die zu lösenden linearen Programme sind dann wie folgt:

```
Minimize: buf!alloc!max
Maximize: buf!alloc!min
```

Diese müssen für jeden Puffer `buf` sowie deren Einschränkungen für `used` gelöst werden. Nach [DJN+10][GJC+03] kann gezeigt werden, dass man alle einzelnen Programme in folgendes Programm zusammenfassen kann:

$$Minimize\colon \sum_{buf} (buf!alloc!max - buf!alloc!min + buf!used!max - buf!used!min)$$

Man sollte erwähnen, dass ganzzahlige Lösungen gesucht sind. Dieses Problem nennt sich ganzzahlige lineare Programmierung bzw. Optimierung. Weil es ein bekanntes NP-vollständiges Problem ist, wird ein Approximationsverfahren genutzt, dass mit dem normalen Solver für lineare Programme arbeitet.

Dafür müssen die Einschränkungen aber in der Form $A * x \geq b$ sein, wo A eine unimodulare Matrix[1] ist. A ist eine n×n Matrix mit ganzzahligen Konstanten, x ein n×1 Vektor mit Variablen und b ein n×1 Vektor mit ganzzahligen Konstanten. [DJN+10][GJC+03] zu Folge werden damit immer ganzzahlige Lösungen erzielt.

4.2 Nicht berechenbare Programme

Jetzt erscheint es so, als ob wir eine Lösung gefunden hätten. Das stimmt nicht unbedingt. Wie schon in Kapitel 3 erwähnt, können lineare Einschränkungen so generiert sein, dass sie unlösbar sind. Wir erinnern uns, dass die Anweisung `i++` in zwei lineare Einschränkungen übersetzt wurde. Das lineare Programm dazu sieht aus wie folgt:

```
Minimize  :    i!max
Subject To:    i'!max ≥ i!max + 1
               i!max ≥ i'!max
```

Abbildung 3: Nicht berechenbares Programm

In Kapitel 3.2 haben wir sichergestellt, dass alle `max` Einschränkungen endliche untere Schranken besitzen und umgekehrt. Jedoch gilt ein Programm als nicht berechenbar, wenn keine endliche Lösung gefunden werden kann, die alle Einschränkungen erfüllt und das Ergebnis optimiert. Eine sichere Variante zur Lösung wäre nun, alle `max` Einschränkungen auf ∞ zu setzen und alle `min` Einschränkungen auf –∞ zu setzen.

Wie im vorherigen Kapitel erwähnt, möchten wir nur eine einzelne Formel nutzen, um den kompletten Satz von Einschränkungen zu lösen. Wenn ein solcher Satz nicht berechenbar ist, kann dies auch nur durch einen kleinen Teil des Satzes ausgelöst werden. Daher ist es viel zu konservativ, alle Einschränkungen mit ∞ zu beschreiben, denn dies würde jegliche Information vernichten. Weil sich solche Anweisungen in fast jedem Programm finden lassen, müssen wir einen anderen Weg finden, um den Satz von Einschränkungen zu lösen.

Wir versuchen, einen Teil des Satzes zu entfernen, der für die Nichtberechenbarkeit verantwortlich ist. Dazu werden *Irreducibly Inconsistent Sets* (kurz IIS) ausfindig gemacht. Ein IIS ist ein Satz von linearen Einschränkungen, der minimal aber nicht berechenbar ist und die Entfernung einer einzelnen Einschränkung den Satz berechenbar macht. So sind in Abbildung 3 die beiden Einschränkungen zusammen ein IIS, denn zusammen ergeben sie ein nicht berechenbares Programm, einzeln aber lassen sie sich berechnen.

[1] Eine unimodulare Matrix ist eine quadratische Matrix, deren Werte alle ganzzahlig sind und die Determinante 1 oder -1 ist.

In [DJN+10][GJC+03] wird der *Elastic Filtering algorithm* genutzt, um einzelne IIS zu finden. Dieser bekommt ein Satz von linearen Einschränkungen als Parameter übergeben und identifiziert ein IIS, wenn es ein solches gibt. Um alle IIS in einem Satz von Einschränkungen zu finden, müssen wir daher den Algorithmus so lange anwenden, bis dieser kein IIS mehr findet. Wir speichern dabei alle entfernten IIS in einem zweiten Satz S' und setzen alle Einschränkungen in unserem neuen, kleineren Satz, die auch in S' vorkommen, entsprechend auf ∞ und -∞.

Weil wir soeben einige Einschränkungen auf ∞ bzw. -∞ gesetzt haben, müssen wir nochmal, wie in Kapitel 3.2, alle Makel entfernen. Das daraus entstehende lineare Programm ist nun berechenbar, alle linearen Einschränkungen haben obere bzw. untere endliche Schranken und das Programm wird nun nur noch optimale Lösungen ergeben. Unser Ziel ist erreicht.

4.3 Einschränkungen hierarchisch lösen

Nach dem Ansatz, den kompletten Satz von Einschränkungen direkt zu lösen, möchten wir nun einen alternativen Ansatz betrachten. Bei diesem ist die Idee den kompletten Satz in kleinere Untersätze zu unterteilen und diese einzeln zu lösen. Zur Unterteilung konstruieren wir einen *directed acyclic graph* (kurz DAG), wo jeder Knoten ein Untersatz ist, und wir die Ergebnisse der Knoten in topologischer Sortierung berechnen.

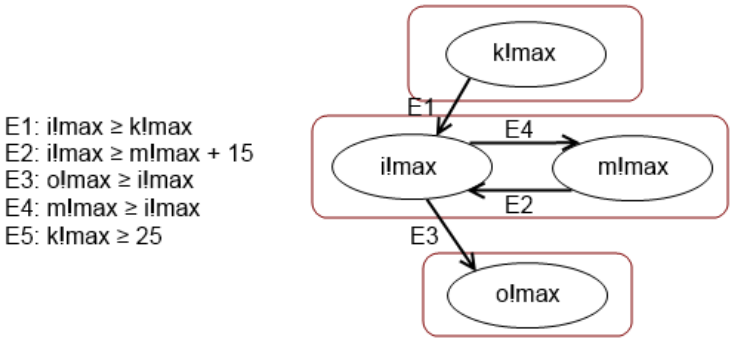

Abbildung 4: Einschränkungen Abhängigkeitsgraph Beispiel

Alle Einschränkungen in einem Untersatz eines Knotens sind dabei von einander abhängig. Als Beispiel schauen wir uns Abbildung 4 an. `E4` beschränkt `m!max` untere Schranke auf `i!max`, jedoch ist `i!max` selbst bestimmt durch `E1` und `E2`. Daher gilt `E4` von `E1` und `E2` abhängig. Formal erstellen wie einen Graphen wie folgt:

Wir erstellen jeweils einen Knoten für alle Einschränkungen, die auf der linken Seite der Ungleichung stehen. Falls die gleiche Einschränkung öfters auf der linken Seite vorkommt, so werden all diese mit dem entsprechenden Knoten verknüpft. Wir ziehen dann eine Kante von einem Knoten `k` zu einem Knoten `l`, falls die entsprechende Einschränkung zu `k` auf der rechten Seite und `l` auf der linken Seite einer Ungleichung stehen. Dann werden *Strongly*

Connected Components (kurz SCCs) gefunden. Ein Satz von linearen Einschränkungen in einem SCC gilt als von einander abhängig. In Abbildung 4 werden solche SCCs durch die roten Umrandungen identifiziert.

Wenn wir nun nur den Graphen der SCCs betrachten, so ist dieser ein DAG. Die topologische Ordnung[2] in einem DAG definiert uns sogleich eine Hierarchie. Dieser Hierarchie folgend können wir nun die einzelnen SCCs lösen und somit den kompletten anfänglichen Satz von linearen Einschränkungen. Da ein SCC aus einem Satz von linearen Einschränkungen besteht, können wir diese in ein lineares Programm wie in Kapitel 4.1 lösen. Falls dabei sich das Programm als nicht berechenbar herausstellt, so können wir direkt alle Einschränkungen in dem SCC auf ∞ und -∞ setzen. Daher ist es nicht nötig, einzelne IIS zu identifizieren.

Nachdem wir jetzt alle Werte für die Einschränkungen in einem SCC berechnet haben, können wir die Kinder dieser SCC berechnen. Dafür substituieren wir alle Werte für Einschränkungen in den Kindern. So berechnen wir alle SCCs des Graphen und daher den ganzen anfänglichen Satz der linearen Einschränkungen.

Dazu seien noch einige Dinge gesagt:

- Durch die Substitution der Werte kann es dazu kommen, dass die Verwendung eines Solvers überflüssig wird. So können wir zum Beispiel für `k!max` in Abbildung 4 einfach bestimmen, dass `k!max` den Wert 25 zugeordnet bekommt. Der Wert kann nun in der linearen Einschränkungen E1 substituiert werden und diese damit vereinfachen.
- Der erste vorgestellte Ansatz, der mit Hilfe der Erkennung von IIS gearbeitet hat, ist nur ein Approximationsverfahren. Es kann passieren, dass unnötig viele lineare Einschränkungen entfernt und die damit verbundenen Einschränkungen entsprechend auf ∞ bzw. -∞ gesetzt werden. Nach [DJN+10] kann gezeigt werden, dass der hierarchische Ansatz präzise ist, dass die kleinste nötige Anzahl von Einschränkungen den Wert ∞ bzw. -∞ erhalten. Es stehen hier also die schnellere, aber ungenauere Variante aus Kapitel 4.1 gegenüber der genaueren, aber langsameren Methode des hierarchischen Lösens gegenüber.
- Da wir für jeden einzelnen Knoten im Graphen den Satz lösen, bietet sich die Möglichkeit an, verschiedene Solver zu nutzen. Je nach Art und Aufbau des Satzes gibt es verschiedene Solver, die es effizienter lösen können.
- Für breite DAGs kann man auch die SCCs einer Ebene parallel lösen. Somit würde ein DAG der Tiefe T in T Schritten gelöst werden können.

[2] Beschreibt die Reihenfolge der Dinge durch vorgegebene Abhängigkeiten. Hier ist der Anfang an dem Knoten zu finden, der nur ausgehende Kanten hat.

4.4 Pufferüberläufe erkennen

Nun schauen wir uns noch an, woran man eigentlich Pufferüberläufe erkennen kann. Nehmen wir an, wir haben ein Programm komplett durchlaufen lassen und Abbildung 5 sei unser Ergebnis.

buf1!used!max	1024
buf1!used!min	0
buf1!alloc!max	1024
buf1!alloc!min	1024
buf2!used!max	2048
buf2!used!min	1024
buf2!alloc!max	1024
buf2!alloc!min	0
buf3!used!max	2048
buf3!used!min	1024
buf3!alloc!max	2048
buf3!alloc!min	1024

Abbildung 5: Beispielergebnis einer Analyse

Die Verwendung mit buf1 ist sicher, denn es werden nie mehr Bytes genutzt als minimal allokiert wurden. buf2 ist ein sicherer Pufferüberlauf, denn es wurde maximal mehr genutzt als maximal dem Puffer zugewiesen wurden. buf3 ist ein möglicher Pufferüberlauf. Es könnte sein, dass zu einem Zeitpunkt nur die minimale Anzahl an Bytes allokiert waren, aber schon die maximale Anzahl an Bytes genutzt wurden (alloc = 1024, used = 2048).

5 Fazit und Ausblick

Mit einem Tool das nach dem gezeigten Design arbeitet lassen sich Pufferüberläufe in realen Programmen finden. Was noch verbessert werden kann ist, dass Kontextsensitivität bei der Generierung von Einschränkungen hinzugefügt wird, um noch genauere Ergebnisse erzielen zu können. Außerdem muss noch eine Möglichkeit bereitgestellt werden, um nach der Erkennung von Pufferüberläufen die entsprechenden Zeilen im Quellcode zu finden, um diese zu eliminieren.

5.1 Zusammenfassung

Wir haben für alle Variablen lineare Einschränkungen erzeugt und aus diesen Einschränkungen entfernt, die entweder unendliche Werte angenommen haben oder als nicht initialisiert galten. Auf den übrig gebliebenen linearen Einschränkungen haben wir dann zwei Methoden gezeigt, um Ergebnisse für die Einschränkungen zu berechnen. Die erste Methode war, zuerst IIS im kompletten Satz zu finden und zu entfernen, um daraufhin für den restlichen Satz eine Lösung zu finden. Die zweite Methode war, dass wir einen DAG bilden und die SCCs in einem Knoten des Graphen lösen. Die Ergebnisse davon haben wir dann genutzt, um Pufferüberläufe für die einzelnen Variablen im Quellcode auszuschließen, sicher zu bestätigen oder auch die Möglichkeit eines Überlaufes zu sehen.

5.2 Kritische Würdigung

Wir haben ein Design vorgestellt, das statische Analyse nutzt. Diese arbeitet auf dem Quellcode und die Fehler müssen durch Programmierer behoben werden. Doch offensichtlich muss es, wenn es statische Analyse gibt, auch dynamische Analyse geben. Diese versucht zur Laufzeit zu erkennen, wenn es zu einem Pufferüberlauf kommt. Ein darauf basierendes Tool kann auch von einem Anwender genutzt werden, um die Ausnutzung von Pufferüberläufen zu verhindern. Doch oft muss dabei das Programm beendet werden, was ein deutlicher Nachteil ist. Außerdem werden die Fehler nicht behoben, sondern nur die Symptome behandelt. Daher ist es auf lange Sicht besser, mit Hilfe der statischen Analyse die Ursachen zu finden und zu eliminieren. Um aber kurzfristig vor Angriffen durch Ausnutzung von Pufferüberläufen geschützt zu sein, ist auch die Anwendung von Tools basierend auf der dynamischen Analyse sinnvoll.

Literaturverzeichnis

[DJN+10] DATTA, Anupam ; JHA, Somesh ; NINGHUI, Li ; MELSKI, David ; REPS, Thomas: *Analysis Techniques for Information Security*. Morgan & Claypool, 2010 (Synth. Lect. on Inform. Sec., Priv., and Trust)

[WFB+00] WAGNER, David ; FOSTER, Jeffrey S. ; BREWER, Eric A. ; AIKEN, Alexander: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security* (2000)

[GJC+03] GANAPATHY, Vinod ; JHA, Somesh ; CHANDLER, David ; MELSKI, David ; VITEK, David: Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM conference on Computer and communications security (*2003)

[LE01] LAROCHELLE, David ; EVANS, David: Statically Detecting Likely Buffer Overflow Vulnerabilities. In *2001 USENIX Security Symposium (2001)*

[Yve12] YVES, Younan: *25 Years of Vulnerabilities: 1988-2012.* Sourcefire, 2013