Zusammenfassung

Statische Erkennung von Buffer Overflows mit BOON

SE 185.307 Seminar aus Programmiersprachen

von

Alex Hörmandinger, 0026782
Wien, 7. Januar 2008

Abstract

BOON ist ein statisches Analysewerkzeug um Buffer Overflows mittels Integer-Bereichs-Analyse zu erkennen. Es ermöglicht die vollautomatische Generierung und Überprüfung von Constraints auf String-Puffern, deren Verletzung auf einen Buffer Overflow hinweist. BOON entlastet damit den Entwickler vom Spezifizieren expliziter Bedingungen im Quellcode. Die vorhandene Implementierung des Werkzeugs ist ein Proof-of-Concept, der nur eingeschränkte Analysefähigkeiten besitzt.

1 Einführung

Die vorliegende Arbeit versucht, die wesentlichen Aspekte des Artikels „A First Step Towards Automated Detection Of Buffer Overrun Vulnerabilities“ von David Wagner [6] zusammenzufassen und im Kontext anderer Arbeiten aus dem Forschungsfeld darzustellen.

Das Ausnützen von Buffer Overflows, um von außen eingeschleusten Code auszuführen, stellt seit mehr als 15 Jahren eine zentrales Sicherheitsproblem dar. Zwar ist das Feld der möglichen Attacken genau erforscht, dennoch wurde bis in die Gegenwart keine Möglichkeit gefunden, Pufferüberläufe zu verhindern, ohne Nachteile wie Performance-Verlust, mögliche Denial-Of-Service-Attacken, etc. in Kauf zu nehmen.
Um Buffer Overflows zu verhindern bzw. zu erkennen, wurden zahlreiche Ansätze erforscht und publiziert, wobei sich folgende Einteilung treffen läßt:

- Laufzeit-Lösungen: Systeme wie StackGuard [1], etc. versuchen das Überschreiben der Return-Adresse einer Methode zu erkennen bzw. zu verhindern.
- Non-Executable-Stack: Stacks werden von verschiedenen Betriebsystemen als Read-Only markiert, wodurch das Überschreiben von Return-Adressen unmöglich wird.
- Sicherheits-Architekturen: Architekturen wie SELinux [5] erlauben die Definition einer feinkörnigen Sicherheits-Politik, um die Verwendung von System-Calls, etc. zu beschränken.
- Statische Analyse

2 Statische Analyse

Statische Werkzeuge [4] [2] analysieren in den meisten Fällen den Quellcode eines Programmes und setzen daher seine Verfügbarkeit voraus. Im Gegensatz zu anderen Lösungen ermöglichen sie die Erkennung von Überläufen zur Entwicklungszeit und unterstützen somit die Entwickler aktiv.
Die Entdeckung von Buffer Overflows durch statische Analyse ist aus Perspektive der mathematischen Logik ein *unentscheidbares Problem*. Folglich sind alle auf ihr basierenden Ansätze logisch betrachtet *nicht abgeschlossen*

und *nicht korrekt*. Das bedeutet in der Konsequenz, dass jedes Analysetool *False Positives* (vermeintlich erkannte Buffer Overflows, die aber keine sind) bzw. *False Negatives* (tatsächliche Buffer Overflows, die jedoch nicht erkannt werden) produziert.
Es wurden zahlreiche Analysewerkzeuge für Buffer Overflows im akademischen Umfeld entwickelt, wobei die Mehrzahl auf der Verwendung von Annotationen basiert. Dabei werden semantische Kommentare - sogenannte Annotationen - verwendet, die der Entwickler im Quellcode verwendet, um explizit Constraints für Puffergrössen anzugeben. Die definierten Constraints werden vom Werkzeug während der Analyse verwendet, um weitere Constraints abzuleiten und zu überprüfen, ob Statements im Code diese verletzen.

3 BOON

Ein anderer Weg wurde bei der Entwicklung von BOON (*Buffer Overrun DetectiON*) [6] gewählt. Die Grundlage des Vorgehens besteht darin, das Problem der Erkennung von Buffer Overflows als Bereichsanalyse-Problem (*integer range analysis problem*) aufzufassen. Dieser Ansatz besticht vorallem dadurch, dass keinerlei Meta-Informationen (vgl. Annotationen) vom Entwickler angegeben werden muss. Dazu wird jeder Puffer durch zwei Integer beschrieben:

1. Die allokierte Anzahl der Bytes eines Puffers (*allocated size*).
2. Die Anzahl der gegenwärtig verwendeten Bytes eines Puffers (*length*).

Dadurch können alle Funktionen der C Standard-Bibliothek in Bezug auf Veränderung der allokierten Länge bzw. der verwendeten Länge von Puffern modelliert werden. Der Inhalt der Puffer ist dabei nicht von Interesse. Abbildung 1 veranschaulicht die Modellierung von Funktionen der Standard-Bibliothek anhand von Beispielen.
Die statische Analyse von Code-Fragmenten, die Puffer mittels Zeiger manipulieren, ist sehr komplex. Da die überwiegende Mehrzahl der Buffer Overflows aber in String-Arrays passiert und dabei meist die Funktionen der Standard-Bibliothek verwendet werden, können Puffer als abstrakte Datentypen modelliert werden, die Operationen wie *strcpy()* und *strlen()* besitzen. Dadurch wird die Komplexität der Analyse stark reduziert, allerdings mit Einschränkung der Mächtigkeit der Analyse: Buffer Overflows die durch gewöhnliche Zeiger-Operationen entstehen, können nicht erkannt werden.

C Code	Constraints
char s[n];	$n \subseteq alloc(s)$
strlen(s)	$len(s) - 1$
strcpy(dst,src);	$len(src) \subseteq len(dst)$
strncpy(dst,src,n);	$min(len(src), n) \subseteq len(dst)$
s = "foo";	$4 \subseteq lens(s), 4 \subseteq alloc(s)$
p = malloc(n);	$n \subseteq alloc(p)$
p = strdup(s);	$len(s) \subseteq len(p), alloc(s) \subseteq alloc(p)$
strcat(s,suffix);	$len(s) + len(suffix) - 1 \subseteq len(s)$
strncat(s,suffix,n);	$len(s) + min(len(suffix) - 1, n) \subseteq len(s)$

Abbildung 1: BOON Constraints

3.1 Definition von Constraints

Um effektive Range Constraints auf Puffern beschreiben zu können, definieren die Autoren von BOON eine abgeschlossene Constraint Sprache.
Dabei ist ein Bereich *range* eine Teilmenge $R \subseteq \mathbb{Z}$ gegeben in Form eines Intervalls $[m, n] = \{i \in \mathbb{Z} : m \leq i \leq n\}$. Für eine Menge S bezeichnet $infS$ das kleinste Element, bzw. $supS$ das grösste Element der Menge. Für Ranges gilt damit natürlich: $inf[m, n] = m$ und $sup[m, n] = n$.
Die Bereichshülle *range closure* einer Menge S ist definiert als der kleinste Bereich R der S umfasst, genauer: $R = [infS, supS]$ So hat zum Beispiel die Menge $S = \{-4, 0, 1, 3\}$ die Hülle $[-4, 3]$.
Die arithmetischen Operatoren lassen sich folgendermaßen auf Mengen erweitern:

$$S, T \subseteq \mathbb{Z} :$$

$$S + T = \{s + t : s \in S, t \in T\}$$
$$S - T = \{s - t : s \in S, t \in T\}$$
$$S \times T = \{s \times t : s \in S, t \in T\}$$
$$min(S, T) = \{min(s, t) : s \in S, t \in T\}$$
$$max(S, T) = \{min(s, t) : s \in S, t \in T\}$$

Ist das Ergebnis einer Operation kein Bereich, wird die Hülle des Ergebnisses verwendet. Zum Beispiel: $S = \{1, 2, 3, 4\} = [1, 4]$ und $T = \{3, 4, 5, 6\} = [3, 6]$, dann ist $S - T = \{-5, -4, -3, -2, -1, 0, 1\} = [-5, 1]$
Eine Integer-Bereichs-Ausdruck *integer range expression* ist definiert als:

$$e ::= v \mid n \mid n \times v \mid e + e \mid e - e \mid max(e, \ldots, e) \mid max(e, \ldots, e)$$

■

wobei $n \in \mathbb{Z}$ und $v \in Vars$, eine Menge aus Bereichsvariablen.
Ein Integer-Bereich-Constraint *integer range constraint* hat die Form:

$$e \subseteq v$$

wobei v eine Variable ist.
Formal ist eine Wert-Zuweisung (*C assignment*) an eine Variable in BOON eine Funktion der Form:

$$\alpha : v \to \alpha(v) \subseteq \mathbb{Z}$$

Eine Zuweisung (*assignment*) erfüllt eine gegebene Menge von Constraints, falls alle Constraints erfüllt sind, wenn man die formale Variable v durch ihre möglichen zugewiesenen Werte $\alpha(v)$ ersetzt.

3.2 Constraint Erzeugung

Die grundsätzliche Architektur von BOON lässt sich in 3 Komponenten gliedern: Parsen des Quellcodes, Erzeugung der Constraints, Lösen des Constraint-Systems.
In einem ersten Schritt wird der Quellcode des Programms geparst um einen Parse Tree aufzubauen. In einem weiteren Schritt wird dieser Baum traversiert, um für allen Knoten Integer-Bereich-Constraints zu erzeugen, welche die im obigen Abchnitt 3.1 beschriebene Form besitzen. Dabei ist mit jeder Integer-Variable v im Programm (v ist also ein C `int`) eine Range-Variable w assoziiert, die beschreibt welchen Wertebereich v annehmen kann. Mit jeder String-Variable s im Programm (s ist also ein C `char`-Buffer) sind wie bereits beschrieben zwei Variablen assoziiert: die allokierte und die tatsächliche Länge: $alloc(s)$ und $len(s)$.
Um sicherzustellen dass auf einem String-Array s kein Buffer Overflow auftritt, muss folgende Aussage im gesamten Programm gelten:

$$len(s) \leq alloc(s)$$

Für jedes C-Statement werden in der Analyse Integer-Bereichs-Constraints erzeugt. So wird zum Beispiel für `v = e` der Constraint $e \subseteq v$ und für `i =`

`i + j` der Constraint $i + j \subseteq i$ generiert. String-Operationen werden wie in Abbildung 1 gezeigt, auf Constraints abgebildet.
Um die Analyse zu vereinfachen und eine bessere Skalierbarkeit zu gewährleisten, ist die Implementierung des BOON-Prototypen nicht flow-sensitiv; auf Bedingungen und Schleifen wird keinerlei Rücksicht genommen. Das wirft zudem Probleme bei Funktionen wie `strcat()`auf, die in einer Schleife beliebig oft ausgeführt werden können und dabei kein idempotentes Verhalten aufweisen. Aus diesem Grund wird jeder Aufruf von `strcat()` als potentieller Buffer Overflow ausgegeben. Ähnliche Vereinfachungen gelten in Bezug auf Methodenaufrufe. Von der Analyse vollkommen ignoriert werden: Funktionenzeiger, doppelte Zeiger (`char**`) und Unions.

3.3 Auflösung der Constraints

Um das erzeugte System von Constraints zu lösen existieren vielfältige Möglichkeiten. Der am häufigsten beschrittene Weg ist die Verwendung von Techniken aus der Linearen Programmierung, zum Beispiel das Simplex-Verfahren. Im Gegensatz dazu verwendet BOON einen ungenaueren, auf Fixpunkt-Theoremen aufbauenden Bounding-Box-Algorithmus, der jedoch mit der Anzahl von Constraints linear skaliert. In [3] wird eine zu BOON eng verwandten Ansatz beschrieben, jedoch unter Verwendung von Algorithmen aus der Linearen Programmierung.

3.4 Evaluierung und Performance

In [7] wurde eine detaillierte Evaluierung fünf verschiedener statischer Analysetools (PolySpace, Splint, BOON, Archer, UNO) vorgenommen. Dabei wurden eine Analyse des Quellcodes der Projekte Sendmail, Bind und WU-FTPD durchgeführt. Betrachtet man die Ergebnisse, fällt die schlechte Erkennungsqualität von BOON ins Auge. Abbildung 2 zeigt die Ergebnisse im Überblick. $P(d)$ bezeichnet die Wahrscheinlichkeit des Erkennens eines vorhandenen Buffer Overflows, $P(f)$ bezeichnet die Wahrscheinlichkeit von False Positives. Von den frei verfügbaren Werkzeugen (PolySpace ist ein kommerzielles System, zu dem keine genauen Informationen frei verfügbar sind), bietet gegenwärtig nur Splint gute Ergebnisse. In Bezug auf BOON müssen jedoch Relativierungen angeführt werden:

- Es handelt sich bei BOON um einen Prototyp, der laut den Autoren

System	$P(d)$	$P(f)$	$P(\neg f \mid d)$
PolySpace	0.87	0.5	0.37
Splint	0.57	0.43	0.30
BOON	0.05	0.05	-
Archer	0.01	0	-
UNO	0	0	-

Abbildung 2: Evaluierung von Statischen Analysetools

neue Ansätze zur statischen Analyse von Buffer Overflows aufzeigen soll, diese allerdings noch nicht vollständig implementiert.

- Die schlechte Precision von BOON ist durch die gegenwärtige flow-insensitive Analyse (wie in Abschnitt 3.2 beschrieben) begründet.
- Buffer Overflows treten besonders an den Extremwerten von Programm-Schleifen auf, die von der derzeitigen Implementierung nicht berücksichtigt werden.

4 Zusammenfassung

BOON ist eine Prototyp-Implementierung um neue Konzepte bzw. Ansätze in der statischen Erkennung von Buffer Overflows auf ihre Tauglichkeit zu überprüfen. Die grundlegende Idee besteht darin, die Erkennung von Buffer Overflows in ein Range Analysis Problem zu überführen, für dessen Lösung vielfältige, hochentwickelte Methoden zur Verfügung stehen. Im Gegensatz zu annotations-basierten Werkzeugen ist BOON gegenwärtig noch sehr unausgereift. Dennoch hat sich die Tauglichkeit des Konzepts bereits dadurch erwiesen, dass mehrere Forschungsgruppen den vorgestellten Ansatz weiterentwickelt bzw. verfolgt haben.

Literatur

[1] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference*

on USENIX Security Symposium, 1998, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[2] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM.

[3] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 345–354, New York, NY, USA, 2003. ACM.

[4] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.

[5] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. http://www.nsa.gov/selinux/papers/freenix01-abs.cfm, 2001.

[6] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[7] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM.