Buffer Overflows

Datum: 6. Januar 2008

Christian Schreiber - 0325661
e0325661@student.tuwien.ac.at

Zusammenfassung

In dieser Arbeit werden Sicherheitslücken welche, auf stackbasierten Bufferüberläufen basieren, behandelt. Es werden einige mögliche Lösungsansätze für das Problem betrachtet und deren Vor- und Nachteile aufgezeigt. Auf die Möglichkeit den Stack in mehrere Unterstacks aufzuteilen und Daten nach gewissen Kriterien auf diese Stacks zu verteilen, wird näher eingangen.

1 Einführung

Buffer-overflow Sicherheitslücken sind ein häufiges und ernstes Sicherheitsproblem in Systemen. Die meisten Buffer-overflow Angriffe nutzen fehlerhafte Buffergrößenüberprüfungen, um die Rücksprungadresse am Stack zu manipulieren. Ein Angreifer kann so Befehle exekutieren, die mit den gleichen Rechten ausgeführt werden, wie das angegriffene Programm (One).

Es wurden bereits eine Vielzahl von Ansätzen entwickelt, um gegen Buffer-overflow Angriffen vorzugehen. Einige diese Ansätze verhindern das Einführen von fremden Code in das Programm, andere versuchen das Problem komplett zu lösen. Jedoch haben diese Lösungsansätze meist eine sehr negative Auswirkung auf die Performance des Systems. Für Stack-basierte Buffer-overflow Angriffe wurden Verfahren entwickelt, die das Problem lösen und keine große Auswirkung auf die Performance haben. Diese Lösungsansätze können in vier Kategorien eingeteilt werden: Schutz durch einen Zufallswert, Erzeugen einer Kopie der Returnadresse und Überprüfung des Wertes beim Rücksprung der Methode, korrigieren der Methoden, die oft der Grund für die Sicherheitslücken sind und Veränderungen am Betriebssystem oder der Hardware um Buffer-overflow Angriffe zu erschweren.

Im folgenden wird ein neuer Ansatz zum Schutz vor stack-basierten Bufferüberläufen erläutert. Der Stack wird, je nach den darauf abgelegten Daten, in mehrere Stacks aufgeteilt. Die Stacks werden vor gegenseitigen Veränderungen durch *Guard pages* geschützt. Jeder schreibende oder lesende Zugriff auf diese Guard pages führt zu einem sofortigen Beenden des Programms.

In Kapitel 2 wird die funktionsweise von stackbasierten Bufferüberlaufattacken erläutert. In Kapitel 3 wird auf die Möglichkeit eingegangen, den Stack in mehrere Stacks aufzuteilen und in Kapitel 4 wird die Performance dieser Schutzmaßnahme erläutert. Die Kapitel 5 und 6 beschäftigen sich mit alternativen Schutzmaßnahmen.

2 Stack-basierte Buffer Overflows

In diesem Kapitel wird die Funktionsweise von Stack-basierten Buffer-overflows erläutert.

Buffer-overflow Sicherheitslücken ergeben sich meist aus fehlerhaften Schreibzugriffen auf Arrays. Dadurch, dass mehr Daten in ein Array geschrieben werden, als darin Platz

haben, können vom Angreifer Speicherbereiche verändert werden, die sich im Speicher nach dem Array befinden. Wenn sich im Speicher nach dem Array ein Codepointer (z.B. return-Adresse) befindet kann der Angreifer durch Veränderungen dieses Codepointers Befehle ausführen.

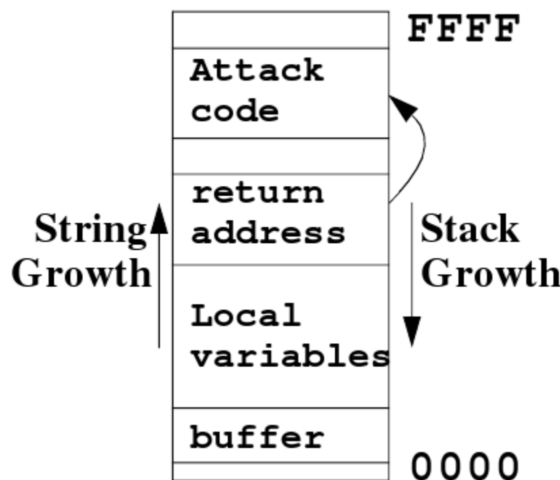

Abbildung 1: Darstellung des Stacks[1]

Abbildung 1 zeigt wie der Stack eines Programmes aussieht. In dieser Abbildung wächst der Stack von oben nach unten und Heap von unten nach oben. Der Stack ist eingeteilt in mehrere Stackframes. Jeder dieser Stackframes enthält neben den lokalen Variablen und anderer Information die Rücksprungadresse einer Methode. Wenn ein Angreifer nun mehr Daten in ein Array auf dem Stack schreiben kann als Speicher für dieses Array reserviert ist, kann die Rücksprungadresse verändert werden. Der Angreifer kann so Code zur Ausführung bringen.

Zwei der am häufigsten eingesetzten Möglichkeiten, um sich vor Buffer-overflow Angriffen zu schützen, werden im folgenden erläutert.

StackGuard(CWP+00; CPM+98) schützt die Rücksprungadresse mit einem Zufallswert. Der Zufallswert (auch canary genannt) wird beim Starten des Programms erzeugt und vor jede Rücksprungadresse geschrieben. Wenn nur die Adresse dereferenziert wird, wird überprüft, ob der Wert verändert wurde. Sollte das Canary verändert worden sein, wird die Ausführung des Programms beendet. Wenn der Angreifer das Canary erraten (oder herausfinden) kann, kann dieser Schutz jedoch leicht umgangen werden. Mittels eines "indirect pointer overwriting"(BK00) Angriffs kann StackGuard auch umgangen werden. Bei diesem Angriff wird ein lokaler Pointer so verändert, dass er auf die Returnadresse zeigt. Wenn nun auf den Pointer geschrieben wird, kann der Angreifer die Returnadresse verändern ohne das Canary zu veränder.

ProPolice(Pro) versucht die Nachteile von StackGuard zu beseitigen, indem die Reihenfolge der gespeicherten Variablen auf dem Stack verändert werden. Es werden alle Arrays vor den anderen Variablen gespeichert. Dadurch ist es nicht möglich, einen lokalen Pointer zu überschreiben. Wenn ein Angreifer das Canary kennt, kann er den Codepointer, wie bei StackGuard, verändern.

[1] Diese Abbildung wurde aus (YPPJ06) entnommen.

3 Aufteilung des Stacks

In diesen Abschnitt wird eine Möglichkeit vorgestellt, die durch Aufteilung des Stacks vor Buffer-overflow Angriffen schützt.

3.1 Lösungsansatz

Der Ansatz dieser Schutzmaßnahme ist es, Daten nach ihrem Wert für den Angreifer als Angriffsziel ("target") und wie sehr sich Daten für den Angriff eignen ("Angriffs Vector") in Kategorien einzuteilen, und diese Kategorien jeweils auf verschiedenen Stacks zu speichern. Es werden hier sechs Kategorien verwendet, jedoch wären auch andere Einteilungen möglich. Tabelle 1 zeigt eine mögliche Einteilung der Kategorien. *Kategorie 1* enthält Daten die einen großen Wert als Angriffsziel haben jedoch nicht bzw. schwer für einen Angriff genutzt werden können. Diese Daten sind das Hauptziel beim Schutz. Bei Daten in *Kategorie 3* spielt es keine Rolle, wo sie gespeichert sind, da sie kaum Wert für den Angreifer haben. Die *Kategorie 6* enthält die am schwersten zu schützenden Daten, da sie sowohl einen hohen Wert als Angriffsziel haben als auch der Ausgangspunkt eines Angriffspunkts sein können.

Vector/Target	Low	Medium	High
Low	Kat. 3	Kat. 2	Kat. 1
Medium	Kat. 5	Kat. 3	Kat. 2
High	Kat. 5	Kat. 4	Kat. 6

Tabelle 1: Einteilung der Daten in Kategorien

Alle möglichen Daten auf dem Stack werden nach folgender Einteilung in Kategorien eingeteilt. Eine genaue Erläuterung kann in (YPPJ06) gefunden werden.

Kategorie 1 return address, other saved registers, pointers

Kategorie 2 arrays of pointers, structures and unions (no arrays), integers

Kategorie 3 floating types, other arrays, structures/unions containing arrays but not arrays of characters at any levels, arrays of structures that do not contain arrays of characters at any level

Kategorie 4 structures containing array of characters, arrays of structures containing arrays of characters

Kategorie 5 arrays of characters

Kategorie 6 leer

Eine andere Einteilung dieser Daten würde nur minimale Modifikationen bei dieser Schutzmaßnahme mit sich ziehen. Die Hauptidee bei diesem Buffer-overflow Schutz ist es, Daten, die einen hohen Wert für Angreifer haben, getrennt von Daten zu Speicher die der Angriff als Ziel hat. Dadurch können die Daten, die das Angriffsziel darstellen, nicht mehr überschrieben werden.

3.2 Implementierung

Die einzelnen Stacks werden hintereinander im Speicher abgelegt und durch Guard pages getrennt[2]. Die Stacks beginnen an fixen Positionen im Speicher. Diese Punkte definieren auch die maximale Größe, die eine Stack wachsen kann (muss zu Übersetzungszeit bekannt sein).

Wenn auf eine lokale Variable zugegriffen wird, wird die Adresse des Stack Pointers in das Frame Pointer Register kopiert. Auf alle lokalen Variablen wird mit dem Frame Pointer als Offset zugegriffen. Der Compiler berechnet den Offset für lokale Variablen während dem Übersetzen und verwenden diesen Offset für die Zugriffe.

Dieser Mechanismus wurde verändert, um die verschiedenen Stacks zu realisieren. Es wird $(stacknr - 1) * (sizeofstack + pagesize)$ zum Offset dazugerechnet. Dadurch kann auf die Variablen auf den Stacks zugegriffen werden, ohne die Methoden für den Zugriff zu verändern.

4 Performance / Memory overhead

Zum Testen der Implementierung wurden Benchmarks durchgeführt.

SPEC CPU2000 Integer benchmarks				
Program	LOC	Gcc 4.1 (s)	Multistack (s)	Overhead
164.gzip	8,616	201	201	0%
175.vpr	17,729	213	212	-0.47%
176.gcc	222,182	89.7	89.8	0.11%
181.mcf	2,423	248	249	0.4%
186.crafty	21,150	116	115	-0.86%
197.parser	11,391	257	255	-0.78%
253.perlbmk	85,185	150	151	0.67%
254.gap	71,430	101	101	0%
255.vortex	67,220	169	174	2.96%
256.bzip2	4,649	204	203	-0.49%
300.twolf	20,459	291	297	2.06%
Microbenchmarks				
loop	20	9.166 ± 0.029	9.2 ± 0.015	0.37%
fibonacci	14	3.354 ± 0.00	4 3.363 ± 0.005	0.27%

Tabelle 2: Benchmark Ergebnisse

Tabelle 2 zeigt die Ergebnisse der Benchmarks. Als Macro Benchmark wurde Spec CPU2000 (Spe) Benchmark verwendet. Als Micro Benchmark ein Programm, welches in einer Schleife eine Funktion eine Million mal aufruft und ein Programm welches die 42. Fibonacci Zahl berechnet. Wie in Tabelle 2 ersichtlich ist, hat diese Schutzmaßnahme kaum bis keinen Einfluss auf die Performance der Software.

Nachteil dieser Implementierung ist der hoher Speicherbedarf. Der Mehraufwand an Speicher berechnet sich aus der Originalgröße des Stacks mal der Anzahl an Stacks. Da der Speicherplatz der Variable berechnet wird durch hinzufügen eines konstanten Wertes

[2]Für die Implementierung wurde gcc-4.1-20050902 für Linux (IA32 Architektur) verwendet.

zum Frame Pointer entstehen Lücken. Um diesen Nachteil zu Beseitigen, soll eine Version entwickelt werden, die den tatsächlichen Speicherplatz der Variable berechnet im Stack.

5 Andere Schutzmaßnahmen

In diesem Kapitel werden andere Ansätze vorgestellt, die vor Stack-basierten Bufferüberlaufen schützen.

StackGuard und ProPolice wurden bereits in Kapitel 2 eingeführt.

StackShield kopiert die Rücksprungadresse an einen anderen Speicherort und stellt den Wert vor dem Rücksprung von einer Methode wieder her. Jedoch schützt StackShield auch nicht vor dem indirekten Überschreiben eines Pointers (siehe 2).

RAD funktioniert so ähnlich wie StackShield, jedoch wird die Rücksprungadresse nur mit dem zuvor gespeicherten Wert verglichen und nicht überschrieben.

Es existiert ein weiterer Ansatz der ähnlich wie StackShield funktioniert. Dieser Ansatz teilt den Stack in einen Daten- und einen Kontrollstack. Bei einem Methodenaufruf wird die Rücksprungadresse auf den Kontrollstack kopiert und beim Rücksprung von dort zurück kopiert.

Libverify funktioniert ähnlich wie StackShield. Es wird jedoch bei dieser Methode der Sourcecode nicht benötigt. Die Überprüfungen werden dynamisch in den Prozess gelinkt.

libsafe ersetzt die Bibliotheksfunktionen von denen bekannt ist, dass sie häufig die Ursache für Buffer-overflow Sicherheitslücken sind (z.B. strcpy).

6 Alternative Lösungsansätze

In diesem Kapitel wird auf Schutzmaßnahmen vor Buffer-overflows eingegangen, die sich nicht nur auf stack-basierte Angriffe spezialiseren, sondern auch andere Bufferüberlauf Angriffe adressieren.

Einen perfekten Schutz vor Buffer-overflow Sicherheitslücken würden **Größenüberprüfungen** bei allen Array Zugriffen bieten. Dies kann jedoch in C zu großen Performanceeinbußen und Inkompatibilität mit anderen Anwendungen führen.

PointGuard verschlüsselt alle Pointer im Speicher (mittels XOR) mit einem randomisierten Wert und entschlüsselt sie wieder bevor sie in den Speicher geladen werden. Wenn der Schlüssel erraten werden kann, kann jedoch PointGuard leicht umgangen werden.

Eine Möglichkeit das Einschleusen von fremden Code zu verhindern sind **nicht ausführbare Speicher**. Das Betriebssystem verhindert das Ausführen von Code der im Textsegment des Programms liegt. Diese Schutzmaßnahme verhindert jedoch nicht das Ausführen von bereits verhandenen Code bei einem Angriff (z.B. die Funktion exec in der libc).

Ein weiter Ansatz ist das Verschlüsseln von Instruktionen für jeden Prozess. Dieser **Randomisierten Instruction Set** wird erst wieder entschlüsselt, wenn ein Befehl exekutiert werden soll. Dieser Ansatz bringt jedoch große Performancenachteile mit sich. Außerdem wäre es möglich, dass Angreifer den Schlüssel erraten.

Mittels **randomisierten Speicheradressen** werden die Speicheradressen bei jedem Ausführen der Applikation geändert. Die Methode hat wie auch der randomisierte Instruction Set Performancenachteile.

Mittels **Program shepherding** wird die Ausführung eines Programms beobachtet und es werden Aktionen verhindert, die als unsicher gelten z.B. verhindern, dass der Angreifer Sicherheitsabfragen umgeht oder sicherstellen, dass Programme nur in Bibliotheksfunktionen springen können.

Eine weiter Möglichkeit die Ausführung von Programmen zu überwachen ist die Überprüfung der **Integrität des Control-flows**. Wenn der Kontrollverlauf nicht den vorgegebenen Werten entspricht wird von einem Fehler ausgegangen. Dies wird verwirklicht indem jedem Ziel eines Kontrollflussüberganges[3] ein Identifier zugeordnet wird und die Abfolge der Identifier beobachtet wird.

7 Zusammenfassung

Es wurde in dieser Arbeit eine Einführung in stackbasierte Bufferüberlaufschwachstellen gegeben. Des weiteren wurden einige Maßnahmen vorgestellt, welche diese Schwachstellen beseitigen und es wurden Nachteile dieser Maßnahmen aufgezeigt. Auf die Möglichkeit den Stack in mehrere Unterstacks zu teilen, wurde näher eingegangen. Für diese Schutzmaßnahme werden alle Daten im Speicher, je nach ihrem Schutzbedarf und ihrer Gefahr bei einem Angriff, eingeteilt. Die Daten in den verschiedenen Kategorien werden danach in unterschiedlichen Stacks gespeichert, um sie vor gegenseitiger Veränderung zu schützen. Der Nachteil dieses Schutzes ist die große Speicherverschwendung, welche jedoch beseitigt werden soll. Die Vorteile sind die geringen Performanceeinbußen und dass diese Methode keinen geheimen Canary Wert benötigt.

Literatur

[BK00] Bulba and Kil3r. Bypassing stackguard and stackshield. 2000.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stack-Guard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[CWP+00] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pages 119–129, Jan 2000.

[3] Ein solcher Übergang ist z.B. ein Methodenaufruf oder der Rücksprung aus einer Funktion

[One] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49.

[Pro] http://www.research.ibm.com/trl/projects/security/ssp/. Stand:02.12.2007.

[Spe] http://www.spec.org/cpu/. Stand:02.12.2007.

[YPPJ06] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 429–438, Washington, DC, USA, 2006. IEEE Computer Society.