

# Hauptseminar

## Malicious code:

### Verhindern von Bufferoverflows

Axel Rimanek

Technische Universität München

## 1 Kurzfassung

Buffer Overflows kommen immer häufiger vor und stellen ein immer größeres Sicherheitsproblem dar. Mit dieser Ausarbeitung soll ein Überblick über existierende Methoden zur Verhinderung von Buffer Overflows gegeben werden.

## 2 Einleitung

1988 trat der erste Internet Wurm auf, der sich mit Hilfe eines Buffer Overflows ausbreitete indem er hauptsächlich auf Unix-Systemen eine Sicherheitslücke des "finger daemons" ausnutzte. Es wurden tausende Maschinen infiziert und das Problem erlangte die öffentliche Aufmerksamkeit. 1998 veröffentlichten die CERT's (Computer Emergency Response Teams), dass mittlerweile über die Hälfte aller gemeldeten Sicherheitsprobleme auf Buffer Overflows zurückzuführen sind und die Tendenz weiter steigend ist. Der Grund dafür dürfte einerseits die Unterschätzung des Problems bei der Entwicklung betroffener Programmiersprachen wie C oder C++ sein und andererseits die Verharmlosung der Problematik bei den Programmierern, denn ein Buffer Overflow ist, lokal betrachtet, meist ein triviales Problem. Dass es im Moment aber nicht in den Griff zu bekommen ist, zeigt sich beispielsweise bei großen Projekten wie sendmail, bei denen mehrere tausend Zeilen Quelltext von Experten per Hand analysiert wurden und danach immer noch Buffer Overflows aufwiesen. Im Folgenden soll deshalb ein Überblick über die derzeitig verfügbaren Methoden verschafft werden, welche eingesetzt werden können, um Buffer Overflows beim Entwickeln zu verhindern oder im Betrieb unschädlich zu machen.

## 3 Strategien zur Vermeidung von Buffer Overflows

Der erste Abschnitt befasst sich mit den Ursachen der Problematik und zeigt zugleich Vermeidungsstrategien auf. Danach werden zwei Methoden vorgestellt mit denen Buffer Overflows automatisch verhindert werden können. Der letzte Abschnitt befasst sich schließlich mit einer Hardware-Lösung, bei der durch Veränderung der Systemarchitektur Abhilfe verschafft werden soll.

### 3.1 String Funktionen

Um eine Programmiersprache möglichst effizient zu gestalten, müssen Kompromisse eingegangen werden, die Rechenzeit einsparen oder den Programm Overhead möglichst gering halten. Darunter leiden in C und C++ besonders die String-Funktionen der "Standard Template Library", denn sie können die Speicher-kapazität der ihnen übergebenen Parameter nicht überprüfen und arbeiten deshalb bis zum typischen Ende eines Strings, welcher durch eine Null oder einen Zeilenvorschub repräsentiert wird. Für einen Angriff übergibt man einem Programm dann eine Eingabe mit deren Länge Entwickler nicht gerechnet haben könnten und die Funktion überschreibt unkontrollierbar den angrenzenden Speicherbereich. Andere Programmiersprachen wie Java schützen vor diesem Problem indem sie die Ausführung eines Programms in der "Java-Virtual-Machine" überwachen. Um die Problematik auch in C oder C++ zu entschärfen wurden kritische Funktionen durch verbesserte Varianten ergänzt, die dem Entwickler mehr Kontrolle geben. Diese lassen sich ihrer Funktion nach in zwei Gruppen aufteilen:

#### **A: Funktionen ohne Einfluss auf die Eingabe wie z.B. gets()**

Funktionen die von der Eingabe lesen, arbeiten meist bis zum Erreichen des Zeilenvorschubs da dies das Ende der Eingabe kennzeichnet. Beispiel:

```
char buf[1024];
gets(buf);
```

Da über die zu erwartende Eingabe und Ihre Länge keinerlei Aussage gemacht werden kann, gilt diese Funktion als äußerst kritisch und sollte möglichst nicht verwendet werden. Die Variante fgets() ist von der Funktionalität identisch, außer dass man mit einem zusätzlichen Parameter die maximale Eingabelänge festlegen kann und somit ein Überlaufen des vorgesehenen Puffers verhindert:

```
char buf[1024];
fgets(buf, 1024, stdin);
```

Für die Funktion scanf(), gibt es mit sscanf() auch die Alternative, die Arbeitslänge mit einem weiteren Parameter zu spezifizieren. Existiert diese Variante auf einer Plattform nicht, so kann man die maximale Länge auch im Format angeben:

```
void main(int argc, char **argv){
    char parameter_one[1024];
    scanf(argv[1], "%1023s", &parameter_one);
}
```

**B: Allgemeine Stingfunktionen wie z.B. strcpy()**

Da man hier die volle Kontrolle über seine Variablen hat, sind mehrere Varianten möglich:

Falls man mit kleinen Konstanten arbeitet kann man auch ohne Überprüfungen arbeiten da hier das Risiko beschränkt ist:

```
strcpy(dst, "Text");
```

Beim nächsten Schritt überprüft man seine Variablen selbst:

```
if(strlen(src) > dst_size)
    * Fehlerbehandlung *
else
    strcpy(dst, src);
```

Oder man fordert den benötigten Speicherplatz dynamisch an:

```
dst = (char *) calloc(strlen(src)+1);
strcpy(dst, src);
```

Vorteil dieser Variante ist, dass durch calloc() der Zielpuffer bereits mit Null initialisiert wird und man sich somit nicht mehr um die Terminierung kümmern muss. Die beste Möglichkeit jedoch ist die Verwendung verbesserter Funktionen, denen man über einen weiteren Parameter die maximale Länge des zu verarbeitenden Strings übergibt:

```
strncpy(dst, src, dst_size-1);
dst[dst_size-1] = "\0"
```

Dies ist zugleich aber ein Beispiel, dass selbst die verbesserten Funktionen noch große Verwirrung bei den Entwicklern hervorrufen und in der Vergangenheit eine Vielzahl so genannter "off by one errors" entstanden sind, bei denen also ein Fehler von genau einem Zeichen gemacht wurde. Denn strncpy() terminiert den Ziel-String nur, wenn die Quelle kleiner oder gleich der übergebenen Arbeitslänge ist. Die sicherste Methode ist deshalb ein Zeichen weniger zu kopieren und die Null am Ende immer selbst einzufügen. Des weiteren stellt strncpy() ein Performance Problem dar, da ein (evtl. sehr großer) Zielpuffer immer vollständig mit Null aufgefüllt wird. Dieses Problem wurde bereits für viele Plattformen endgültig gelöst indem wiederum neue Funktionen eingeführt wurden, etwa strlcpy() für Linux oder die neuen "C Runtime Libraries", die mit dem neuen VisualStudio 2005 erscheinen.

Für einen Programmierer ist es wichtig sich bei der Verarbeitung von Strings über die Risiken der verwendeten Funktionen im Klaren zu sein, verbesserte Funktionen zu verwenden oder wenn dies nicht möglich oder verfügbar ist, selbst geeignete Überprüfungen vor der Ausführung durchzuführen. Man beachte aber, dass die verbesserten Funktionen dem Programmierer zwar die Möglichkeit geben mehr Kontrolle zu erlangen, doch sie stellen noch kein Sicherheitskonzept

dar. Will jemand mutwillig in einen fremden Speicherbereich schreiben oder hat sich mit den Parametern der Funktionen verrechnet, so wird das Programm nach wie vor ohne Hinderung ausgeführt. Eine Technik, welche zumindest die Ausführung eines durch Bereichsüberschreitung eingeschleusten Codes verhindert ist das NX-Flag neuester Prozessoren, auf das später eingegangen wird.

### 3.2 Automatische Analyse Tools

Um in Projekten, die mehrere tausend Zeilen Code umfassen nicht alles von Hand überprüfen zu müssen entstanden zwei unterschiedliche Ansätze, dies automatisch durchführen zu können: Statische und dynamische Analyse.

Bei der Statischen Analyse versucht man Sicherheitslücken zu entdecken indem man den Quelltext beim Compilieren analysiert und somit dem Programmierer die Möglichkeit zur Verbesserung gibt. Ein solches Tool ist beispielsweise ITS4. Es liest den Quelltext zeichenweise ein und vergleicht ihn mit seiner Datenbank, die bekannte, kritische Funktionen enthält. Beim Auffinden eines dem Tool bekannten Problems werden dem Entwickler dann Fehlerbeschreibungen, Vermeidungsstrategien und Risikoeinstufungen angezeigt, so dass er das Problem verstehen und vermeiden kann. Da bei diesem Ansatz der Code lediglich auf lexikographischer Ebene analysiert wird, kann auch nur eine lokale Aussage erzeugen werden, die dafür sehr präzise ist und aufgrund des geringen Rechenaufwands zum Einsatz in Code Editoren prädestiniert ist, die "on-the-fly" beim Programmieren beraten. Momentan existiert eine Emacs-Version in die ITS4 integriert wurde und eine Datenbank mit 131 kritischen Funktionen enthält.

Will man eine Globalere Betrachtung der Variablen des Programms erlangen, so muss man das gesamte Programm einlesen und analysieren. Dies wird z.B von BOON (Buffer Overrun detectiON) versucht, welches eine statische Analyse auf der syntaktischen Ebene beim Parsen des Quelltextes durchführt. Da die meisten Überläufe bei Strings auftreten konzentriert es sich ausschließlich auf sie, modelliert diese als abstrakten Datentyp und verwaltet für Stringpuffer zwei (integer) Werte, die die Gesamtgröße und die bereits belegte Größe des Puffers abspeichern. Für jede Anweisung wird somit beim Parsen eine (integer) Ungleichung aufgestellt und anschließend, ähnlich wie bei der normalen Compiler Codeanalyse, in einem Ungleichungssystem gelöst. Dadurch lassen sich Variablen durch das gesamte Programm hindurch überwachen und für jede Anweisung Aussagen treffen ob ein möglicher Buffer Overflow vorliegt. Da das Tool auf die Analyse großer Softwareprojekte optimiert wurde hat man vielfach heuristische Annahmen gemacht um eine ausreichende Performanz zu erlangen - leider auf Kosten der Präzision. Damit erkennt das Tool vielfach Fehler nicht oder alarmiert bei sicheren Stellen. Außerdem kann es nur sehr beschränkt mit Pointern und gar nicht mit Format-Strings umgehen. Trotzdem stellt es für einen Entwickler eine schnellere Alternative dar als die manuelle Analyse großer Projekte.

Im Gegensatz zur statischen Analyse versucht man bei der dynamischen Analyse die Ausführungsumgebung derartig zu verändern, dass Sicherheitslücken oder infizierter Code keine Sicherheitsrelevanten Auswirkungen mehr haben. Ein solches dynamisches Tool ist z.B. "StackGuard", bei dem jeder Funktionsaufruf durch eine Compilererweiterung überwacht wird. Da ein Angreifer beim Einschleusen seines auszuführenden Codes im Stack die Rücksprungadresse verändert, damit sein eigener Code ausgeführt wird, muss er dessen Position kennen. Diese variiert jedoch, da die Größe des Stackframes auf unterschiedlichen Plattformen und Implementierungen variieren kann. Um dieses Problem zu umgehen wird der Angreifer deshalb erst seinen Puffer überlaufen lassen, dann alle weiteren lokalen Variablen im Stack überschreiben und schließlich mehrfach um die vermutete Position der alten Rücksprungadresse seine neue platzieren. Genau hier setzt das Projekt an, indem es vor die Rücksprungadresse ein extra Wort setzt, das "canary-word". Vor dem Aufruf der Rücksprungadresse wird nun überprüft ob dieses durch einen Angriff überschrieben wurde, und es wird ggf. der Prozess beendet.

Um diesen Schutz zu umgehen müsste ein Angreifer nun also die Rücksprungadresse überschreiben ohne das "canary-word" zu verändern, indem er es entweder mit dem korrekten Wert überschreibt oder es mit einem Pointer umgeht. Dagegen wurden nacheinander drei Strategien entwickelt: Als erstes wurde das "canary-word" zufällig gewählt um ein Erraten zu verhindern, dann wurde ein Wort generiert, welches aus allen vier möglichen Sting Terminatoren besteht (Null, CarriageReturn, -1 und LineFeed) und somit unmöglich durch einen String Buffer Overflow generierbar ist. Zuletzt wurde ein XOR des "canary-words" und der Rücksprungadresse verwendet, so dass jegliche Veränderungen der Rücksprungadresse erkannt werden konnten.

Ein anderes Konzept von StackGuard entstand aus dem MemGuard-Projekt, welches versucht, ähnlich der neuen NX-Prozessor-Flags, virtuelle Speicherseiten als schreibgeschützt zu markieren um somit z.B. ein Überschreiben der Rücksprungadresse zu verhindern. Der einzige Nachteil dieser Softwarelösung ist jedoch, dass zur Programmausführung schreibbare Speicherstellen aus einer schreibgeschützten Speicherseite sehr rechenintensiv emuliert werden müssen. Der Faktor liegt hierbei 1800 Mal über der Rechenzeit einer normalen Schreiboperation. Diese wesentlich sicherere Variante eignet sich daher leider nur für "ruhige Speicherbereiche" der Kernprozesse oder Debugging Zwecke. Die Entwickler denken jedoch bereits an eine Variante, bei der durch vermehrte Angriffe eine Maschine von der günstigen "canary" auf die sicherere MemGuard-Methode umgeschaltet wird und ein optimaler Kompromiss zwischen Geschwindigkeit und Performance erreicht wird.

### 3.3 Hardwarelösungen

Einen völlig neuen Weg beschreiten im Moment Microsoft zusammen mit Intel und AMD. Die immer weiter ansteigenden Zahlen an Buffer Overflows haben gezeigt, dass die ständig veröffentlichten Sicherheitspatches zur Beseitigung der Sicherheitslücken in Betriebssystemen dauerhaft keine Lösung darstellen. Mit der Datenausführungsverhinderung (Data-Execution-Prevention DEP) wurde ein völlig neues Konzept entwickelt, das nun auf der Hardwareseite für Abhilfe sorgen soll. Die DEP unterscheidet zwischen Speicherstellen die nur Daten enthalten und Speicherstellen die Ausführbaren Code enthalten. So kann eine Ausführung von Code in Speicherbereichen, die als Datenspeicher markiert sind, verhindert werden. Dies geschieht indem in den "page table entries" ( PTE ) entsprechende Bits gesetzt werden. Diese Funktion wird auch als "No Execute" (NX) oder Ausführungsschutz bezeichnet. Wenn ein Versuch unternommen wird, Code von einer markierten Speicherseite auszuführen, tritt sofort eine (höchstwahrscheinlich) unbehandelte Ausnahme auf und verhindert die Ausführung des Codes. Dadurch wird verhindert, dass ein Angreifer einen Buffer Overflow mit Code herbeiführt und den Code anschließend ausführt. Würde gar der Kernel auf eine so geschützte Speicherstelle zugreifen, so stürzt das ganze Betriebssystem ab. Ein Software-Kompatibilitätsproblem bei diesem Ansatz stellen jedoch Programme dar, die während der Laufzeit neuen Programmcode generieren (wie Just- In-Time-Compiler) und diesen nicht ausdrücklich mit Ausführungsberechtigung markieren. Die Software müsste überarbeitet werden und die entsprechenden Funktionen, welche Speicher anfordern, durch neue Spezialvarianten ersetzt werden (VirtualAlloc( ) mit einem der PAGE\_EXECUTE-Speicherattribute).

Bei der Hardware ist momentan noch die übliche Einlaufphase zu beobachten: Während Intel bisher nur seine Server-CPU "Itanium" mit NX-Technologie ausstattet ist AMD mit sämtlichen seiner aktuellen 64-bit CPUs vorraus. Alle Windows User mit einer 64- bit-CPU und dem neuen Servicepack 2 erhalten somit vollen Schutz des Stacks und des Heaps - beim Einsatz einer 32-bit CPU leider nur noch des Stacks. Für Linux und OpenBSD kommt das Feature in Form der Erweiterung "PaX". Es bleibt abzuwarten ob mit dieser neuen Technologie die Häufigkeit klassischer Attacks durch Buffer Overflows eingeschränkt werden kann.

## 4 Schlussfolgerung

Die aktuell verfügbaren Methoden zur Verhinderung von Buffer Overflows zeigen sehr gute Ansätze zu Lösung des Problems, doch scheitern fast alle in der Praxis an individuellen Problemen, nicht zuletzt durch die Vielzahl der existierenden Varianten durch die ein Buffer Overflow erzeugt werden kann. Somit ist derzeit ein absoluten Schutz vor Buffer Overflows, selbst durch eine Kombination, nicht realisierbar.

Sehr wohl aber wird durch die vorgestellten Methoden die Problematik zumindest eingedämmt und ein Entwickler erhält mehr Informationen und Werkzeuge um seinen Quelltext effizienter vor Buffer Overflows zu schützen.

## References

1. Gary McGraw, John Viega,: Make your software behave: Preventing buffer overflows
2. David Wagner Jeffrey S. Foster Eric A. Brewer Alexander Aiken: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities
3. John Viega, J.T. Bloch, Tadayoshi Kohno, Gary McGraw: ITS4: A Static Vulnerability Scanner for C and C++ Code
4. Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks
5. Starr Andersen:Changes to Functionality in Microsoft Windows XP Service Pack 2 Part 3: Memory Protection Technologies