Dynamic Buffer Overflow Detection

Helmut Petritsch
Matr.Nr. 0105154
helmut@petritsch.co.at

Zusammenfassung

Dieses Artikel befasst sich mit dem Paper *"A Practical Dynamic Buffer Overflow Detector"* [1], welches das Tool CRED (C Range Error Detector) beschreibt, das Programme um eine Laufzeitumgebung zur Verwaltung von Buffern erweitert und damit Überläufe in diesen Buffern erkennen und verhindern kann.

Dieser Ansatz wird mit jenem im Paper *"Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software"* [2] beschriebenen verglichen. Beide Ansätze versprechen, eine zuverlässige Erkennung von Bufferoverflows zu gewährleisten.

1 Einführung

Tools für Dynamic Buffer Overflow Detection sollen es ermöglichen, Prozesse zur Laufzeit zuverlässig gegen Overflow Attacken zu schützen. Am Besten ist daher ein möglichst generischer Ansatz zu wählen, der sämtliche Bufferüberläufe (gleichgültig, ob der Buffer z.B. auf dem Stack oder auf dem Heap liegt) erkennen kann. Lösungen wie das Executeable Bit[3] in neueren Prozessoren bieten nur teilweise Schutz gegen Bufferüberläufe. So verhindert das Executeable Bit nur, dass in einen Prozess injizierter Code ausgeführt werden kann. Dies erschwert zwar das Schreiben eines Exploits für Vulnerabilities, verhindert aber nicht Bufferüberläufe an sich.

Bei der Statische Analyse wird versucht, Fehler im (Source) Code zu finden, ohne das entsprechende Programm auszuführen. Im Gegensatz dazu wird bei der dynamischen Analyse das Programm ausgeführt und auf die eine oder andere Art "beobachtet". Dadurch können verdächtige Aktivitäten zur Laufzeit erkannt werden um gegebenenfalls die Ausführung des Programms (im Optimalfall mit einer detaillierten Fehlermeldung) zu beenden.

Da für die dynamische Analyse zur Laufzeit zusätzlicher Code ausgeführt werden muss, bedeutet dies Performance Einbußen; eine bessere Qualtität und Genauigkeit der Ergebnisse erfordert meist einen höheren Aufwand für die Analyse. Des weiteren gilt, dass, je vielseitiger ein Tool ist (also je mehr Arten von Angriffen erkannt werden kann), umso aufwändiger wird die Implementierung und damit schlechter das Laufzeitverhalten. Tools sind daher für unterschiedliche Aufgaben unterschiedlich geeignet bzw. auf diese hin optimiert.

Eine weitere Herausforderung besteht darin, dass jede Form von Beobachtung das beobachtete Objekt beeinflusst[1]. Eine weitere Anforderung an die Dynamische Analyse ist daher, dass existierender Code weiterhin funktionieren muss und sich Exploits, auch wenn sie beobachtet werden, (möglichst) gleich verhalten sollen.

Des weiteren ist für derartige Werkzeuge relevant, wie aufwändig die Analyse an sich ist. Dies bedeutet einerseits, dass das Setup für die Testumgebung möglichst einfach zu bewerkstelligen sein sollte, andererseits sollten aber auch möglichst wenig Voraussetzungen zu erfüllen sein, um das Tool erfolgreich einsetzen zu können - so wäre ein erneutes Kompilieren sämtlicher dynamisch eingebundener Bibliotheken ein aufwändiger Fall, im Optimalfall wird nur das zu testende Executable benötigt.

[1] Einige Techniken basieren z.B. darauf, dass der zu beobachtende Prozess in einer emulierten Umgebung ausgeführt wird. Allerdings sind sowohl softwarebasierte Emulationstechniken wie Qemu [4] erkennbar [5], als auch Emulationen die auf Emulationstechniken der CPU beruhen[6].

1.1 Existierende Lösungen

Buffer Overflows waren lange Zeit einer der vordringlichsten Themen im Security Bereich. Mehrere Würmer wie Slammer etc. zeigten, dass mit Exploits für Buffer Overflows massiver Schaden anzurichten ist. Es existieren daher zu diesem Thema verschiedenste Ansätze, wie Buffer Overflows verhindert und erkannt werden können. Obwohl keiner dieser Ansätze eine einfache, alles umfassende Antwort geben kann, so scheinen sie in ihrer Gesamtheit doch bewirkt zu haben, dass Overflows heutzutage nicht mehr derart relevant sind wir noch vor einigen Jahren.

Lösungen wie StackGuard oder StackShield fokussieren, wie der Name bereits andeutet, primär auf Overflows von Buffern, die am Stack liegen (da hier mit dem Überschreiben der Return Adresse am Einfachsten der Programmfluss zu beeinflussen ist), gegen Buffer Overflows in static Buffers oder am Heap sind sie nicht ausgelegt. Ansätze, die Pointer an sich modifizieren (z.B. CCured), erfordern meist ein Ändern des Source Codes und erhöhen sowohl den Implementierungsaufwand als auch den Speicherbedarf der laufenden Applikation.

2 CRED

CRED[1] (C Range Error Detector) ist ein Tool, bei dem es sich um eine Modifikation des gcc Kompiliers handelt, der das kompilierte Programm um eine Laufzeitumgebung zur Verwaltung von Buffern (Objekten, Arrays, Speicher mit `malloc()` alloziert, etc.) erweitert. Wenn zur Laufzeit vom einem Pointer auf einen zugehörigen Speicher bzw. dessen Größe geschlossen werden kann, ist es möglich, eine Verifikation von lesenden und schreibenden Zugriffen auf diese Buffer zu ermöglichen und damit Bufferüberläufe zu verhindern. Sowohl das Tool als auch das Paper sind als Erweiterung eines bestehenden Tools bzw. Papers [7] von den Autoren Jones und Kelly zu betrachten.

2.1 Bound Checkers

CRED erfasst zur Laufzeit alle Buffer mit Größe und Adresse und kann somit Pointer auf Gültigkeit prüfen. Dies wird erreicht, ohne dass der Programmierer der Anwendungen davon Kenntnis besitzen muss, bestehnder Code kann weiter verwendet werden. Dies bedeutet vor allem auch, dass Programme sowohl "normal" als auch "modifiziert" (z.B. für Tests) gebaut werden können. Die dafür nötigen Modifikationen werden im Zuge des Kompilierens in das Zielprogramm eingefügt, für CRED wurde der Gnu C Compiler (gcc) in Version 3.3.1 als Basis Kompiler verwendet.

Die Idee hinter diesem Tool ist, dass Pointer immer innerhalb eines Buffers weisen müssen - dieser Buffer wird als "Referent Object" bezeichnet. Nach jeder arithmetischen Operation auf einen Pointer, muss dieser in den gleichen Buffer, auf das gleiche Referent Object, weisen. Wenn die Basis Adressen und die Größen aller Buffer zur Laufzeit bekannt sind, kann bei arithmetischen Berechnungen auf Pointer die Korrektheit dieser Pointer einfach überprüft werden. Daraus folgt einerseits, dass die Operationen zur Berechnung und zum Dereferenzieren von Pointern modifiziert werden müssen, um mittels eines Funktionsaufrufes in die "Runtime Library" die Korrektheit des Pointers zu überprüfen. Damit diese Überprüfung ausgeführt werden kann muss des Weiteren sichergestellt werden, dass alle Buffer mit Basisadresse und Größe zur Laufzeit bekannt sind. Wie dies für die unterschiedlichen Arten von Buffern erreicht wird, soll im Folgenden beschrieben werden.

2.1.1 Heap Based Buffers

Buffers auf dem Heap werden durch den `new` Operator bzw. Funktionen wie `malloc()` alloziert und durch den ˜ Operator und Funktionen wie `free()` wieder freigegeben. Durch ein modifizieren all dieser Operatoren und Funktionen kann sichergestellt werden, dass alle auf dem Heap allozierten Speicherbereiche der Laufzeitumgebung bekannt sind[2].

[2]Dies gilt natürlich nur, wenn das eigentliche Executable mit CRED kompiliert wurde - sollte eine shared Library damit kompiliert werden, ist es möglich, dass vor deren Laden bereits Buffer am Heap alloziert wurde und damit nicht erfasst wurden. Wie sich das Tool bei Libraries verhalten würde, wird im Paper aber nicht beschrieben

2.1.2 Stack Based Buffers

Stack Buffers werden anders wie Heap Buffers nicht generell durch Operatoren oder Funktionen erzeugt und können daher auch nicht generell getrackt werden. Dem Kompiler ist jedoch bekannt, wann Objekte auf den Stack gelegt werden, ein Eintragen bzw. Löschen aus der Laufzeitumgebung ist damit implizit möglich. Dies gilt jedoch nur für jenen Code, der mit CRED kompiliert wird - Stack Buffers von externen Libraries können auf diese Art und Weise nicht erfasst werden[3].

Das Problem das sich damit ergibt ist, dass es Buffer geben kann, die zu lesen und zu beschreiben eine gültige Operation ist. Da sie aber der Laufzeitumgebung unbekannt sind, würde diese das Dereferenzieren derartiger Pointer verhindern. Für diesen Spezialfall überprüft die Laufzeitumgebung, wenn sie einen Pointer keinem Buffer zuweisen konnte, ob der Pointer zwischen Stackpointer (`%esp` Register) und dem Top of Stack liegt.

Dieses Vorgehen erklärt, wieso sowohl beim Erzeugen eines neuen Pointers (also bei arithmetischen Operationen auf Pointer) ebenfalls eine Verifikation durchgeführt werden muss, und nicht nur beim Dereferenzieren (was rein intuitiv ausreicht, da erst das Dereferenzieren eines ungültigen Pointers zu undefiniertem Verhalten führt)[4]: da die Darstellung von Pointern aus Kompatibilitätsgründen nicht verändert wird, könnte beim Dereferenzieren nicht mehr zwischen einem ungültigen Pointer auf einen der Laufzeitumgebung bekannten Stackbuffer und einem (gültig oder ungültigen) Pointer auf einen der Laufzeitumgebung nicht bekannten, von einer Library angelegten Stackbuffer, unterschieden werden. Damit würden Pointer, die eigentlich als ungültig erkannt werden könnten (also jene, die aus einem bekannten Stackbuffer hinauszeigen), als gültig verifiziert werden, da die Laufzeitumgebung davon ausgehen würde bzw. müßte, dass es sich eventuell um einen nicht bekannten Stackbuffer handelt.

2.1.3 Static Buffers

Ein weiteres Problem gibt es mit statischen Buffern. Diese können ebenfalls vom Compiler erkannt werden und der Laufzeitumgebung gemeldet werden. Hier ergibt sich das gleiche Problem wie bei Stack Buffern, da nur jene Speicherbereiche auf diese Art und Weise erkannt werden können, die mit CRED kompiliert wurden. Dem entsprechend ist das Verhalten zur Laufzeit ähnlich, nur dass jene Bereiche externer Libraries dadurch identifiziert werden können, dass diese zwischen der virtuellen Speicheradresse `0x0` und der `_BSS` Section von linked Libraries liegen.

2.2 Implementierung von Jones, Kelly

Die Implementierung von Jones und Kelly basiert auf den bisher vorstellten Prinzipien. Informationen über neue Buffer (Basisadresse und Bufferlänger) werden in einen Splay Tree ("Object Table") eingetragen, der darauf optimiert ist, für Speicheradressen schnell feststellen zu können, auf welchen Speicher sie zeigen. Da nur die Speicher selbst getrackt werden und nicht die auf sie zeigenden Pointer, ist der Aufwand linear zur Anzahl von allozierten Speicherbereichen.

Um bestehende Libraries nicht neu kompilieren zu müssen, gleichzeitig aber für das Überschreiben von Buffern "beliebte" Funktionen sicher zu machen, werden jene Funktionsaufrufe an die Standard Library abgefangen und die Buffergrößen kontrolliert. So wird z.B. überprüft ob bei der Funktion `strcpy()` der Buffer auf den geschrieben wird genug Speicher im Zielspeicher vorhanden ist[5].

Bei arithmetischen Operationen auf Pointer wird der jeweils zugehörige Buffer, auf die der ursprüngliche Pointer zeigt, gesucht. Zeigt der Pointer nach dieser Operation nicht mehr auf jenen Speicherbereich, wird er auf den Wert von `ILLEGAL` (definiert als `(void*) -2`, also `0xFFFFFFFE` in 32 bit Systemen) gesetzt.

Beim Dereferenzieren bzw. neuerlichen Operationen auf diesen Pointer wird die Ausführung des Prozesses mit einer Fehlermeldung beendet.

[3] Bufferüberläufe in von Libraries auf den Stack gelegten Buffern können daher nicht erkannt werden, was im Paper nicht erwähnt wird

[4] Auch eher theoretisch anmutende Schwächen, wo ein Pointer auf ein anderes Referent Object geshifted wird, können nur dadurch erkannt werden

[5] Obwohl im paper nicht erwähnt, sind nach Meinung des Autors durch dieses Verhalten False Positives möglich: Wenn z.B. bei `strcpy()` ein 128 Byte Buffer übergeben wird, der String aber nach 32 Byte Null-terminiert ist, der Zielbuffer 64 Byte lang ist, wird unkorrekter weise ein Fehler erkannt

2.3 Implementierung CRED

Mit der Implementierung von [7] gibt es allerdings ein Wesentliches Problem.

Listing 1: Gültige Pointer aus ungültigen Pointern

```
1  char *r, *s, t;
2  char *p = malloc(4);
3  s = p + 10;  //s out of bounds
4  r = s - 8;
5  t = *r;
6  t = *s;      //dies sollte nicht funktionieren
```

Dieser Code wird als gültig kompiliert, obwohl in Zeile 3 eine Verletzung des ANSI C Standards stattfindet: Der Variable `s` wird ein ungültiger Pointer zugewiesen, was bedeutet, dass die Laufzeitumgebung in Zeile 3 dieser Variable `ILLEGAL` zuweist. In Zeile 4 ist es daher nicht mehr möglich, den gültigen Pointer zu errechnen, da über Pointer `s` keine Informationen mehr vorliegen (außer, dass dieser Pointer ungültig ist). Das Programm wird daher in Zeile 4 unkorrekter Weise mit einer Fehlermeldung abgebrochen. Eine korrekte Implementierung sollte diesen Code erst in Zeile 6 abbrechen, wenn der ungültige Pointer dereferenziert wird.

Diese theoretisch anmutende Überlegung führt bei 60% der für dieses Paper getesteten Programme zu einem fehlerhaftem Abbruch. Um diesem Problem entgegen zu treten haben die Autoren von CRED das Konzept der OOB - Out Of Bounds - Objects eingeführt.

2.3.1 Out Of Bounds Objects

OOB Objects repräsentieren Pointer, die aus einem gültigen Pointer berechnet wurden, zur Zeit aber nicht gültig sind. Sie beinhalten eine Referenz auf den Buffer, auf den sie ursprünglich gezeigt haben. Sollten sie, wie im Listing 1 dazu verwendet werden, wieder einen gültigen Pointer zu finden, so ist die Information über den ursprünglichen Pointer noch nicht verloren, der neue Pointer kann errechnet und als gültig verifiziert werden.

OOB Objects werden in einer Hashmap gespeichert, wo schnell auf sie zugegriffen werden kann. Daraus ergibt sich folgendes Verhalten für arithmetische Operationen auf Pointer:

1. Es wird gesucht, ob ein Referent Object in der Object Table vorhanden ist
2. Wird hier nichts gefunden wird untersucht, ob die Adresse in einem unchecked Object liegt (Stack based Buffer, etc).
3. Wird auch hier nichts gefunden, wird ein OOB Object gesucht. Wird die Laufzeitumgebung hier fündig und der neue Pointer ist gültig, wird dieser als regulärer Pointer gespeichert

Ein Problem lässt allerdings auch dieser Ansatz offen: Bei arithmetischen Operationen auf transformierte Pointer(z.B. Pointer, Integer, Pointer Casts) gehen die OOB Informationen verloren.

Eine weitere Verbesserung betrifft die Optimierung: da Bufferüberläufe primär auf String Buffers basieren (da auf diese Art und Weise am einfachsten User Daten in den Prozess gelangen), kann mit einem Compilerflag angegeben werden, dass die Checks nur auf String Buffer angewendet werden sollen. Dies reduziert vor allem in Prozessen, wo String Buffer im Gegensatz zu anderen Buffern nur sparsam eingesetzt werden, den Overhead für die Checks.

2.3.2 Performance

Im Direktvergleich mit der Implementierung von Jones und Kelly ist CRED bis zu 12 mal schneller - vor allem in Programmen, wo kaum String Buffers verwendet werden. Das OOB Konzept bringt einen Overhead von max. 15%, zusätzlicher Speicherbedarf konnte nur im Bereich von weniger als einem Kilobyte beobachtet werden.

Absolut gesehen bringt der Einsatz von CRED für 14 der 20 Programme einen Overhead von 26% oder weniger, für die restlichen Programme ist ein Anstieg des Overheads zwischen 160 und 230% zu verzeichnen.

3 Vergleich mit TaintCheck

In diesem Bereich soll ein kurzer Vergleich mit TaintCheck[2] angestellt werden. Dafür wird zunächst der Ansatz von TaintCheck kurz erklärt um dann im zweiten Teil die Unterschiede zu beschreiben.

3.1 TaintCheck

TaintCheck hat einen gänzlich anderen Ansatz als CRED. Für TaintCheck werden nur die Binaries benötigt, welche in einem modifizierten Prozessemulator (Valgrind[8]) ausgeführt werden.

Daten, die als potentiell gefährlich eingestuft werden, werden beim Eintritt in den Prozess (als Ergebnisse von System Calls - z.B. beim Lesen vom Netzwerkstack etc.) getainted. Dies ist eine häufig verwendete Technik, um Daten innerhalb eines Prozesses zu kennzeichnen, wobei die Kennzeichnung durch verarbeitende Operationen weiter transportiert wird. An sensiblen Stellen kann dann festgestellt werden, ob es sich z.B. bei zu verarbeitenden Daten um gekennzeichnete Daten handelt.

Das transportieren der Taint-Informationen (also das Weiterverfolgen der getainteten Daten) erfolgt bei TaintCheck auf Assembler Ebene bzw. auf Ebene des "UCode", der innerhalb von Valgrind als "Pseudo-Assembler" fungiert. Die auszuführenden Binaries werden in diesen Zwischencode übersetzt, der dann auf einer virtuellen CPU ausgeführt wird. Durch Änderungen bei der Ausführung werden die Taint-Informationen während der Ausführung aktuell gehalten (bei Kopieroperationen, arithmetischen Operationen etc. werden die Zielbereiche ebenfalls getaintet bzw. wird protokolliert, wie welche Daten von wo nach wo gelangt sind).

Durch dieses Verfahren ist immer bekannt, welche Daten vom User kommen und damit potentiell gefährlich sind. Sollte Userinput für sensible Operationen verwendet werden (z.B. als `JUMP` Target, Format String, etc) wird der Prozess mit einer Fehlermeldung beendet. Durch das lückenlose Aufzeichnen des Weges der Daten vom Eintritt in den Prozess bis zur unerlaubten Verwendung ist es möglich konkrete Aussagen über den Trace der Daten innerhalb des Prozesses zu treffen: welche Daten wie in den Prozess gelangt sind und wie genau diese vom Eintritt in den Prozess bis zur ausgenutzten Vulnerability gelangt sind.

3.2 CRED vs. TaintCheck

Der Ansatz von CRED und TaintCheck sind im Grunde auf unterschiedliche Ziele gerichtet: CRED möchte einen jederzeit verfügbaren Schutz bieten, TaintCheck sowohl Echtzeitschutz als auch die Möglichkeit, neue Angriffe frühzeitig zu erkennen und eine präzise Analyse des neu entdeckten Angriffs zu liefern.

Der klare Vorteil von CRED ist damit dessen bessere Performance - die Emulation von Prozessen ist mit einem nicht unwesentlichen Aufwand verbunden, die Modifikation von Assembler Code verlangsamen die Ausführung von TaintCheck zusätzlich. Allerdings ist, vor allem bei Programmen die viel mit Strings arbeiten, der Overhead auch mit CRED signifikant, sodass ein Einsatz mit recht hohen Kosten verbunden ist.

TaintCheck widerum liefert den klaren Vorteil, mehr Angriffe zuverlässiger erkennen zu können und eine präzise Analyse über den Angriff selbst liefern zu können. Der Einsatz von TaintCheck als Honeypot System, der entweder über Loadbalancing in einen bestehenden Verbund eingebunden wird oder über aufgezeichnete und abgespielte Requests aktuellen Traffic abarbeitet und damit neue Angriffe erkennt, ist nach Ansicht des Autors dieses Artikels jene Methodik, die sich im täglichen Einsatz eher bewährt[6].

4 Kritik

Beim Lesen des Papers sind einige Fragen aufgetaucht, die sich teilweise lösen ließen, teilweise aber auf Probleme stoßen, die nicht erwähnt wurden.

[6] Dies ist vor allem für Antiviren Software Hersteller oder Große Webseiten interessant, die Contentfiltering System wie Snort eingesetzten und mit Hilfe von TaintCheck neue Signaturen erstellen können, die weitere Angriffe blockieren

So ist einer der Hauptpunkte, die Optimierung des bestehenden Tools, im Wesentlichen durch den recht einfach Schritt erreicht worden, dass nicht alle Buffer getrackt werden, sondern nur String Buffer. Der Innovationsgehalt dieser Optimierung ist an und für sich nicht sonderlich hoch; des Weiteren bietet diese Optimierung gerade für schützenswerte Programme wie `apached` oder `hypermail`, die viel mit String Buffern arbeiten, kaum Performance Vorteile.

Außerdem wird im Paper nicht darauf eingegangen, wieso nicht nur die Dereferenzierungs-Operationen überwacht werden[7]. Nach Ansicht des Autors dieses Artikels wäre es des Weiteren eine elegantere Lösung, auch "checked" und "unchecked" Stackbereiche (bzw. Bereiche für static Daten) mitzutracken. Damit wären bei sämtlichen arithmetischen Operationen auf Pointer keine Lookups im Object Table mehr nötig. Dies würde auch das OOB Konzept unnötig machen, und auch die Problematik von Pointer, Integer, Pointer Casts würde entfallen. Diese Vorteile könnten nach Meinung des Autors den Nachteil dieser Lösung, "Sprünge" von Pointern von einem Referent Object zu einem anderen nicht erkennen zu können, aufwiegen.

Was ebenso unerwähnt bleibt ist die Tatsache, dass Buffer Overflows in von externen Libraries auf den Stack gelegten Buffern unerkannt bleiben. Ob und wie mit Pointern auf (wiederum von externen Libraries) Stack based Buffers in anderen Threads umgegangen wird (in diesem Fall liegt der Speicher nicht zwischen `%esp` Register und der Stackbase), wird auch nicht näher beschrieben.

5 Evaluierung

Für die Evaluierung wurden 20 Open Source mit in Summe 1,2 Millionen Line of Code getestet. Das Tool von Jones und Kelly lieferte bei 12 der 20 False Positives, mit CRED trat kein einziges False Positive auf. Sieben der 20 getesteten Open Source Programme wurden in Versionen getestet, in denen sie über bekannte Vulnerabilities verfügten. Die darauf angewendeten Exploits konnten erfolgreich von CRED erkannt werden. Ein von Wilander und Kamkar bereitgestelltes Set von Programmen mit verschiedenen Vulnerabilities an Buffer Overflows [9] konnte erfolgreich ausgeführt und alle Exploits erkannt werden.

6 Conclusio

CRED und TaintCheck verfolgen das gleiche Ziel: Buffer Overflows zur Laufzeit erkennen zu können. Allerdings sind die intendierten Einsatzgebiete unterschiedlich - CRED möchte für alle, inklusive produktive, Systeme einen effektiven Schutz zur Laufzeit bieten. TaintCheck verfolgt das Ziel, eine umfassende Erkennung von Buffer Overflows und auch printf-Vulnerabilities zuverlässig zu erkennen und einen präzisen Report für Analysten liefern zu können. Bei dem Paper über CRED bleibt ein wenig der enttäuschende Beigeschmack einer unvollständig wirkenden Arbeit.

Literatur

[1] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. 2004.

[2] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[3] No execute bit. `http://de.wikipedia.org/wiki/NX-Bit`.

[4] Fabrice Bellard. Qemu, a Fast and Portable Dynamic Translator. *Usenix annual technical conference*, 2005.

[5] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. 2007.

[6] Joanna Rutkowska. Red pill. `http://invisiblethings.org/papers/redpill.html`.

[7] Begründung siehe "Stack Based Buffers"

[7] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. page 1997.

[8] N. Nethercote and J. Seward. Valgrind: A program supervision framework. 2003.

[9] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. 2003.