Buffer Overflows und Lösungen dazu

Stephan Neuhaus

21. Mai 2003

1 Einführung

Am 2. November 1988 startete ein Informatikstudent an der Cornell-Universität, Robert T. Morris Jr., ein Programm auf einem Rechner, der mit dem Internet verbunden war [17, 13]. Dieses Programm war ein *Wurm*, der sich durch verschiedenste Sicherheitslücken schnell verbreitete. Von den damals sechzigtausend mit dem Internet verbundenen Computern wurden innerhalb der zwei Tage andauernden Infektion etwa 10–20% lahmgelegt. Der Wurm nutzte einen irrtümlich standarmäßig in Produktionsversionen von *sendmail* verbliebenen debug-Modus, das Vertrauensverhältnis von Rechnern untereinander und einen *Buffer Overflow* in dem (damals) standardmäßig eingesetzten Unix-Dienst *finger*. Die Funktionsweise von Buffer Overflows ist mittlwerweile seit Jahren gut verstanden [1, 30] und es gibt auch diverse Schutzmechanismen dagegen [3, 4, 14, 35]. Dennoch tauchen immer wieder solche Overflows auf. Und es werden immer mehr [10].

Die Tabelle 1 zeigt eine Statistik der CERT-Advisories [7] der letzten fünf Jahre. CERT Advisories (CAs) werden herausgegeben, um auf besonders kritische Sicherheitslöcher hinzuweisen. Aus dieser Tabelle wird ersichtlich, daß sich der Prozentsatz der durch Buffer Overflows verursachten CAs in den letzten fünf Jahren im wesentlichen nicht geändert hat. (Für 2000 werden kaum Buffer Overflows berichtet, einige Advisories wurden jedoch wegen „Input validation errors“ herausgegeben, was möglicherweise ebenfalls Buffer Overflows sind.) Wie kann so etwas geschehen? CERT Advisories werden selten genug herausgegeben, daß man annehmen sollte, daß die betroffenen Hersteller (oder die Open-Software-Gemeinde) schnellstens Maßnahmen ergreift, um dieses Problem ein für allemal aus der Welt zu schaffen. Offenbar ist das Problem aber seit 1988 ungelöst.

Wie die meisten Sicherheitslücken, die nicht auf Konfigurationsfehlern beruhen (wie z.B. der versehentlich in *sendmail* belassene Debug-Modus) oder etwa auf schwachen benutzergewählten Paßwörtern oder ähnlichem beruhen, ist ein Buffer Overflow ein Software-Problem. Das Perfide an solchen Sicherheitslücken ist, daß sie sich nur schwer aufspüren lassen, weil der Abstand zwischen der verursachenden Eingabe und

Jahr	**1998**	**1999**	**2000**	**2001**	**2002**
CAs	13	17	22	37	37
BOs	5	6	1(?)	13	13
%	38	35	4.6(?)	35	35

Tabelle 1: Buffer Overflows in CERT Advisories

```
static void copy_buf(char *buf, size_t len)
{
  int i = 3;
  char dst[12];
  int j = 5;

  /* i == 3 && j == 5 */
  memcpy(dst, buf, len);
  /* Annahme: i == 3 && j == 5 */
}
```

Abbildung 1: Buffer Overflow

dem Buffer Overflow selbst oft groß ist und Modul- oder auch Prozeßgrenzen problemlos überschreiten kann.

Buffer Overflows sind ein Beispiel für ungesicherte Manipulationen des Programmzustands, also Manipulationen des internen Zustands ohne vorherige Prüfung durch das Programm. Bei diesen Manipulationen sind Buffer Overflows am weitesten verbreitet (weil am leichtesten auszunutzen). Eine weitere Klasse von ungesicherten Manipulationen ist die große Klasse der *Code Injection Vulnerabilities*. Diese entstehen, wenn Eingaben ungesichert als Teile von ausführbarem Code verwendet werden.

2 Funktionsweise von Buffer Overflows

Ein Buffer Overflow funktioniert, indem ein Speicherbereich fester Größe, üblicherweise aber nicht immer ein Buffer für Zeichenketten, mit einem Puffer überschrieben wird, dessen Kapazität größer ist, als die des Ziels. Die häufigsten Fälle finden sich, wenn der Zielpuffer auf dem Stack liegt. Siehe dazu im Folgenden Abbildung 1.

Nehmen wir an, der Parameter *buf* enthielte 16 Zeichen (und *len* wäre daher 16), dann könnte der Wert von *i* überschrieben werden. Das liegt daran, daß lokale Variablen in der Regel hintereinander auf dem Stack abgelegt werden und daß *memcpy*(3) die Kapazität des Zielpuffers nicht prüfen kann. Damit können über das Ende von *dst* hinaus Speicherzellen manipuliert werden, die ebenfalls lokale Variablen enthalten (siehe Abbildung 2). Der Wert von *j* ist jedoch in aller Regel nicht vom Buffer Overflow betroffen. Diese Manipulation ist in allen heute gängigen Rechnerarchitekturen auf diese Art möglich. Die Details hängen dabei von den genauen Gegebenheiten der Rechnerzrchitektur ab, z.B. ob Padding nötig ist und wenn ja, auf welchen Grenzen usw.

Das Fragment in Abbildung 3 zeigt ein anderes Problem. Hier könnte der Wert von *i* durch einen Buffer Overflow verändert werden. Allerdings hängen die Auswirkungen dieser Veränderung auf die `for`-Schleife von den Compilereinstellungen zur Optimierung ab. Optimiert der Compiler gar nicht, wird der Wert von *i* spätestens in der 13. Iteration überschrieben, womit die Schleife möglicherweise abbricht. Der Compiler könnte aber auch die Variable *i* komplett wegoptimieren und die Schleifenzählung vollständig in Prozessorregistern machen. In diesem Fall gibt es gar keinen Platz für *i* auf dem Stack und der Aufruf von *memcpy*(3) überschreibt womöglich die Rücksprungadresse oder den geretteten Frame-Pointer.

Buffer Overflows sind grundsätzlich immer sicherheitsrelevant, weil sie programminterne Prüfungen aushebeln. Besonders gefährlich sind sie, weil Angriffe über Ka-

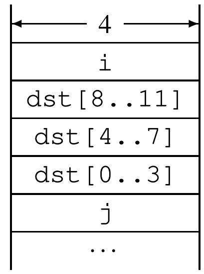

Abbildung 2: Lage der lokalen Variablen auf dem Stack

```
static void copy_buf(char *buf, size_t len)
{
  int i;
  char dst[12];

  for (i = 0; i <= len; i++)
    dst[i] = buf[i]);
}
```

Abbildung 3: Unklare Auswirkungen, falls der Compiler `i` wegoptimiert

näle kommen können, die von den üblichen Sicherheitsmaßnahmen nicht betrachtet werden. So wurde im April 2003 eine Meldung veröffentlicht, nach der im Windows QuickTime Player ein Buffer Overflow existiert [19]. Der QuickTime Player ist ein Programm, mit dem man Video-Dateien in Apples Filmformat abspielen kann. Der Buffer Overflow besteht darin, daß der Puffer, der die abzuspielende URL enthält, nur 400 Zeichen lang ist. Diese Meldung ist deshalb besonders besorniserregend, weil ein solcher Buffer Overflow es ermöglicht, eine besondere URL auf einer HTML-Seite unterzubrigen, die ausführbaren Code enthält. Somit kann durch Anklicken eines Hyperlinks, der eine solche URL enthält, auf dem angegriffenen Rechner beliebiger Code zur Ausführung gebracht werden. Wenn das Herunterladen von Videos überhaupt freigegeben ist, wird dieser Angriffscode alle Firewalls und in der Regel auch alle Virenscanner ungehindert passieren. Man könnte diesem speziellen Problem mit einem Proxy entgegentreten, der angeforderte URLs grundsätzlich in der Länge beschränkt. Dazu müßte man aber erst mal wissen, daß solche Buffer Overflows überhaupt möglich sind.

3 Stand der Forschung

Es existieren eine Reihe guter Artikel zur Funktionsweise von Buffer Overflows. Exemplarisch seien hier nur der allgemeine Artikel [24], der Artikel zu Windows [12] und der „Klassiker“ [1] hervorgehoben.

Ansätze zur Behandlung von Buffer Overflows lassen sich in vier Kategorien einteilen: Statische Methoden, die durch Analyse des Quelltexts (oder auch Maschinencodes, siehe [39]) solche überläufe finden wollen, Methoden, die auf einer Instrumentierung des Quelltexts beruhen, stackbasierte Methoden, die zwar nicht die überläufe selbst, wohl aber das überschreiben der Rücksprungadresse verhindern wollen und Methoden auf Betriebssystemebene, die mit Unterstützung der MMU verhindern wollen, daß der Stack unangemessen verwendet wird. Niemand versucht, mit experimentellen Methoden zu Aussagen über die Sicherheit von Programmen zu gelangen.

Im Artikel [9] lautet das ernüchternde Fazit:

> Fundamentally, the attacks that StackGuard prevents are not very interesting. They are serious security faults that result from minor programming errors. Once discovered, fixing each error is easy.

Daran ist vieles wahr. Wenn das aber so ist, wieviele andere Klassen ernsthafter Sicherheitsfehler aufgrund von geringfügigen Programmierfehlern gibt es dann noch? Allgemein gefragt: Welche Möglichkeiten gibt es denn überhaupt, den Zustand eines Programms von außen schädlich zu manipulieren?

3.1 Statische Analyse

In [37] zerlegen die Autoren einen Quelltext in Token, um danach mit Textmustern nach potentiell gefährlichen Stellen im Quelltext zu suchen.

All diesen Ratschlägen und Tools ist gemeinsam, daß sie nur dann gut funktionieren, wenn die Entwicklungsprojekte klein oder recht neu sind, weil dann der Aufwand, alle Quellcodes umzuschreiben, noch vertretbar ist.

In [31] stellen die Autoren eine Methode vor, Slices zu präzisieren. Dabei werden Pfadbedingungen erstellt, die Aussagen über die Abhängigkeit von Daten von anderen Daten oder Eingaben machen. Die dort entwickelte Software ValSoft wird u.a. eingesetzt, um bei der Physikalisch-Technischen Bundesanstalt in Braunschweig zu überprüfen, ob der Datenpfad einer Waage von der Wiegevorrichtung zur Anzeige von nachträglichen Manipulationen frei bleibt. Diese Methoden funktionieren allgemein, nicht nur bei Buffer Overflows.

Andere statische oder symbolische Methoden, die speziell auf Sicherheitsbelange ausgerichtet sind, findet man in [27, 28, 15, 22].

Ein Problem, das Buffer Overflows symbolischen Methoden in der Regel nicht zugänglich macht ist, daß dabei bestimmte im Programmiermodell verankerte Invarianten verletzt werden. So wird man beispielsweise bei der Untersuchung des Programmfragments in Abbildung 1 davon ausgehen, daß der Aufruf von *memcpy*(3) den Wert von *i* nicht verändert. Deshalb sind symbolische Methoden in der Regel ungeeignet, um die Konsequenzen eines Buffer Overflows außer in besonders einfachen Fällen festzustellen und zu bewerten.

3.2 Instrumentierung des Quelltexts

Eine weitere dynamische Methode ist es, Zugriffsfehler durch Code-Instrumentierung festzustellen [2]. Der Titel “Efficient Detection of *All* Pointer and Array Access Errors” ist dabei etwas mißverständlich. Gemeint sind nicht alle *potentiellen* Zugriffsfehler, sondern alle *während eines bestimmten Laufs tatsächlich auftretenden* Fehler. Die Methode ist hier, den Quellcode so zu instrumentieren, daß jeder Pointer-Zugriff vor Ausführung auf Gültigkeit überprüft wird. Pointer erhalten nemen ihrem Wert noch die Anfangsadresse und Größe des Speicherbereichs zugewiesen, auf den sie zeigen. Da Referenten von Zeigern entweder feste Größen haben (die also zur Kompilierzeit bereits festliegen) oder durch z.B. *malloc*(3) erzeugt werden, wobei die Größe ebenfalls festliegen muß, ist das auch prinzipiell nicht schwierig.

Das Problem beim Einsatz dieser Methodik zur Entdeckung von Buffer Overflows ist, daß das so ausgeführte Programm solange „sauber“ bleibt, wie nicht tatsächlich ein Buffer Overflow auftritt. (Die Methode ist auch gar nicht dazu entwickelt worden. Sie wurde entwickelt, um die Findung einer Ursache-Wirkung-Kette, die bis zum Programmabsturz führt, zu vereinfachen.) Man erhält keine Aussage über *potentielle* Buffer Overflows. Das ist insbesondere deshalb unglücklich, weil sich eine dauerhafte Instrumentierung (und damit auch ein dauerhafter Schutz) auch im Produktionsbetrieb verbietet. Das liegt am hohen Verwaltungsaufwand, da wirklich *jeder* Pointerzugriff (also z.B. in C auch jeder Array-Zugriff) geprüft werden muß. Die Autoren nennen in ihrem Aufsatz Overheads zwischen 130% und 500%, die Laufzeit wird also um einen Faktor 2 bis 6 erhöht. Möglicherweise wäre eine Unterstützung in Hardware sinnvoll. Das würde allerdings eine Anpassung des Betriebssystems und der Compiler bedingen.

3.3 Stackbasierte Methoden

Methoden, die die Ausnutzung von Buffer Overflows zur Laufzeit unmöglich machen sollen, bestehen oft darin, spezielle Bibliotheken anzubieten, die statt der Standardbibliotheken verwendet werden sollen oder die diese ersetzen [3, 4, 8, 21, 26, 29, 34].

Diese Methode funktioniert in [3, 4] durch die Feststellung, daß ein Puffer, der auf dem Stack abgelegt wird, durch die Größe des Stackframes beschränkt ist, in dem er enthalten ist. Daher ist es möglich, beispielsweise bei *memcpy*(3) oder *strcpy*(3) durch Betrachtung des entsprechenden Stackframes ein Überschreiben der Rücksprungadresse zu verhindern. Dieses Vorgehen scheint gegen eine große Zahl von Overflows zu helfen. Buffer Overflows entstehen jedoch nicht nur durch unsichere Bibliotheksaufrufe (wenn auch diese den größten Teil ausmachen dürften). Außerdem funktioniert die Methode nur, wenn es verläßlich möglich ist, den Frame-Pointer innerhalb eines beliebigen Stack-Frames ausfindig zu machen. Das ist dann nicht gegeben, wenn aus Optimierungsgründen auf die explizite Errichtung eines Stackframes verzichtet wird, wie es beispielsweise bei Verwendung der gcc-Option `-fomit-frame-pointer` passiert. Die Autoren behaupten weiterhin:

> [By using `libsafe`], the return address from the function, which is located on the stack, cannot be overwritten and control of the process cannot be commandeered.

Das ist so nicht ganz richtig. Zwar mag die Rücksprungadresse vor überschreiben geschützt sein, andere Variablen im momentanen Stackframe aber nicht. Eine ungesicherte Beeinflussung dieser Variablen kann aber bereits zur Kompromittierung des Prozesses ausreichen. Eine andere Behauptung der Autoren leuchtet ebenfalls nicht ein: Daß nämlich die Größe eines Puffers, der auf dem Stack zu liegen kommt, zur Kompilierzeit noch nicht feststehe. Das wird aber in aller Regel der Fall sein. Kaum ein Programm verwendet heute noch Funktionen wie *alloca*(3), die die Größe des Stackframes über die zur Kompilierzeit feststehenden Wert hinaus zur Laufzeit verändern. Diese Kritik betrifft aber nicht den Kern der Arbeit.

Im Bergbau wurden bis zur Einführung nichtletaler Methoden Kanarienvögel eingesetzt, die bei der Anwesenheit gefährlicher Gase zu zwitschern aufhörten und so die Bergleute rechtzeitig warnen konnten [18]. In Anlehung daran werden in StackGuard bestimmte Werte (canary values) auf den Stack unmittelbar vor die Rücksprungadresse geschrieben [8, 9]. Damit kann die Veränderung der Rücksprungadresse zwar nicht verhindert, deren änderung jedoch bemerkt werden. Diese Technik wird von der Compilermodifikation in [14] implementiert. Die Rücksprungadresse wird außerdem durch ein MemGuard genanntes System zur Laufzeit geschützt. Dabei werden bestimmte Speicherbereicht als Read-Only markiert und Zugriffe darauf abgefangen. Die meisten Kritikpunkte werden von den Autoren in ihrem Paper vorweggenommen:

- MemGuard ist bezüglich der Ausführungszeit ziemlich teuer und beinhaltet einen nicht akzeptablen Performance-Overhead.
- Ein Angreifer, der einen canary value raten kann, kann einen erfolgreichen Angriff vortragen.
- Angreifer können mit MemGuard geschützte Programme durch den großen Performance-Overhead einem DoS-Angriff aussetzen.
- Die aktuelle Implementierung schützt nur die vier letzten Stackframes.
- Gegen Buffer Overfows, die keine „stack smashing attacks“ sind, hilft MemGuard nicht.

Auch hier findet sich wieder die irreführende Behauptung, durch den Schutz der Rücksprungaadresse im Fall eines Buffer Overflows könnte zuverlässig verhindert werden, daß ein Angreifer unerwünschte Privilegien erhält. Dennoch ist das Paper sehr löblich, weil es eine ausführliche Beschreibung *experimenteller* Resultate enthält. Die Autoren versuchen jedoch nicht, ihr Verfahren gezielt auszuhebeln. Die Behauptung, durch die Anwendung bereits bekannter Buffer Overflows auf ein ansonsten unmodifiziertes Programm würde simuliert, wie ihr Verfahren beim Aufkommen eines *unbekannten* Exploits reagiert, ist aber sicher nicht richtig. StackGuard wurde mit bestimmten Angriffen im Hinterkopf konzipiert. Daß eine große Anzahl bereits vorhandener Angriffe auf das StackGuard-Muster passen, ist zwar löblich, aber kein verläßliches Indiz dafür, wie StackGuard auf gänzlich unbekannte Angriffe reagiert. Dieser methodische Fehler wird von mehreren Autoren begangen.

Die Autoren machen aber auf einen wichtigen Punkt aufmerksam: Wer Buffer Overflows studieren will, braucht anfälligen Code und darauf passende Exploits. Letzterer sei nach Aussage der Autoren schwierig zu erhalten, weil z.B. über CERT keine Exploits herausgegeben werden und man sich stattdessen mit Web-Suchen behelfen muß. Aus der Tatsache, daß die Autoren offenbar davor zurückschrecken, eigene Exploits zu schreiben, kann man schließen, daß sie darin kaum Erfahrung haben dürften. Ich glaube, daß das die Einschätzungen über die Gefährlichkeit von Angriffen zugunsten der Verteidigungsmöglichkeiten trübt.

Anfälliger Code sei nach der Herausgabe eines Patches (was meist zeigleich mit der Ankündigung der Anfälligkeit selbst geschehe) kaum mehr verfügbar, weshalb ältere Softwareversionen auf jeden Fall archiviert werden sollten.

Ein weiterer Fehler der Autoren ist die Annahme, nur administrative Programme benötigten Schutz gegen Buffer Overflows. Gelingt es beispielsweise, durch den Quicktime Player beliebigen Code auszuführen [19], kann ein Angreifer die Identität des angegriffenen Benutzers übernehmen und danach Angriffe von innen starten oder eine Hintertür im Stil von Back Orifice [6] öffnen. Gegen solche Angriffe sind Rechner und Netzwerke oft deutlich schlechter abgesichert, als gegen externe Angriffe.

Ein weiterer Punkt:

> Using StackGuard does not eliminate the need to fix buffer overflow vulnerabilities, but by converting `root` vulnerabilities into mild degradation-of-service attacks, it does eliminate the urgency to fix them. This

> gives software developers the breathing room to fix buffer overflows when it is convenient (i.e. when the next release is ready) rather than having to rush to create and distribute a patch. More importantly, StackGuard eases security administration by relieving the system administrators of the need to apply these patches as soon as they are released, often several times a month.

Ob sich beispielsweise Amazon.com dieser Sichtweise anschließen würde?

Wenn ein Exploit von StackGuard zuverlässig verhindert wird, warum muß man dann überhaupt noch patchen? Die Autoren scheinen von ihrer eigenen Lösung nicht sehr überzeugt zu sein, und das aus gutem Grund: Wenn Administratoren nicht genau wissen, daß ein Exploit von StackGuard verhindert wird, muß nämlich doch unverzüglich gepatcht werden. Auch die Entwickler können sich nur dann Zeit lassen, wenn sie wissen, daß ein Exploit gegen ihr StackGuard-geschütztes Programm nicht funktioniert. (Wenn aber ihr Programm schon mit StackGuard-Schutz entwickelt wurde, wer würde dann einen solchen Exploit bauen, der nicht einmal erfolgreich getestet werden kann?) Bei einem Sicherheitsloch, das nicht durch StackGuard gefunden wird, müssen auch die Entwickler wieder sofort einen Patch herausgeben, besonders wenn anzunehmen ist, daß ein Exploit bereits in freier Wildbahn kursiert. (Performance-Verschlechterungen wie die von den Autoren gemessenen einige tausend Prozent als „mild degradation-of-service attacks“ zu bezeichnen, ist außerdem ziemlich stark.)

Andere Tools versuchen lediglich, die Rücksprungadresse zu schützen, damit nicht länger beliebiger Code ausgeführt werden kann [35, 36]. Diese Tools schreiben die Rücksprungadresse an einen „sicheren“ Ort und stellen sie kurz vor dem Rücksprung wieder her. Da Funktionen andere Funktionen aufrufen und für diese Funktionen dieselben Mechanismen gelten müssen, wird hier ein Stack benötigt, um alle Rücksprungadressen zu speichern. In der Praxis bedeutet eine solche Lösung die Separation des Stacks für Rücksprungadressen vom Stack für lokale Variablen. Das erfordert eine Menge änderungen beim Compiler.[1] Außerdem steht zu befürchten, daß dieses Verfahren nicht gegen speziell auf diese Verfahren zugeschnittene Exploits schützt.

Eine andere Möglichkeit ist es, einen Compiler zu verwenden, der Code erzeugt, der gegen Buffer Overflows weniger anfällig ist [14, 5, 23]. Hier reicht es jedoch, wenn eine Fremdbibliothek, die man möglicherweise aus Geschäftsgründen einsetzen muß und über die man keine Kontrolle hat, einen Buffer Overflow aufweist. Nachteil: Solche Methoden sind *reaktiv*.

3.4 Stack-Schutz mit Hilfe der MMU

Einige Maßnahmen arbeiten auf Betriebssystemebene. Ein Beispiel ist die Einführung nicht-ausführbarer Stacks [11, 32]. Dabei werden die Speicherbereiche, die den Stack beinhalten, als nicht ausführbar markiert. Kommt es zu einem Programmlesezugriff auf eine solchen Bereich, wird dieser Zugriff von der MMU mit einem Trap abgewiesen. Einige Linux-Distributionen bieten mittlerweile einen nicht ausführbaren Stack an, andere zumindest Patches, wie z.B. vom Openwall-Projekt [25]. Dennoch bieten sie verläßlichen keinen Schutz gegen Angriffe. Ein Exploit [38] (auch ausgenutzt in [33, 20]) funktioniert gegen einen ungepatchten X11-Server bei nicht ausführbarem Stack. Der Trick ist, daß man nicht den Exploit-Code selbst auf dem Stack plazieren, sondern lediglich einen möglichst ähnlichen Code im Programm finden muß. Viele Programme enthalten z.B. die Möglichkeit, eine Shell zu starten. Gelingt es, die

[1] Diese Beschreibung wurde dem übersichtsartikel [16] entnommen.

Rücksprungadresse so zu ändern, daß sie auf diesen Code zeigt, erhält man eine Shell auch ohne ausführbaren Stack. Andere Wege, um an nicht ausführbaren Stacks vorbei zu kommen, findet man in [11]. Nicht-ausfürbare Stacks schüzen zudem nicht gegen das überschreiben von Variablen oder gegen Code-Injection.

3.5 Zusammenfassung

Diese Ansätze sind entweder noch zu neu, um verbindliche Aussagen zu ermöglichen oder aber werden nicht breit genug eingesetzt, um die angesprochenen Probleme wirksam bekämpfen zu können. (Immerhin setzt OpenBSD ab Mai 2003 nun ProPolice [14] ein, eine Variante der StackGuard-Methode [9].) Das liegt an mehreren Eigenheiten:

- Statische Methoden ergeben zu viele falsche Positivmeldungen, weil sie den Aufrufkontext nicht richtig bewerten können
- *Ein* Programm, das *einen* ausnutzbaren Buffer Overflow enthält, genügt u.U. bereits, um einen Rechner zu kompromittieren
- Schutzmechanismen werden nicht flächendeckend eingesetzt
- Manchmal gibt es sogar technische Hürden, die den Einsatz von Schutzmechanismen verhindern
- Fokussierung auf das einzelne *Programm*, selten auf das *System*, bestehend aus Programm, Prozeß, Betriebssystem, dynamischen Bibliotheken, Netzwerkkomponenten etc.
- Schutzmechanismen haben das Problem, daß sie meist durch besondere Angriffe umgangen werden können. Solange das so ist, gibt es eine Art Rüstungswettlauf, bei dem die Angreifer immer in der besseren Position sind, weil sie beliegig viel Zeit haben, in den Maßnahmen Lücken zu finden und sich einen neuen Exploit zu überlegen.

Dennoch bleiben alle Arten von ungesichertern Manipulationen wichtige Sicherheitslücken.

Der Übersichtsartikel [16] schließt:

> There are at least three possible areas of future development for buffer overflow tools:
>
> 1. Many of the compiler patches and libraries only support Linux or other Unix derivatives and the gcc compiler. There is a need for developing similar tools with support for other compilers and operating systems.
> 2. There is a need for a powerful source analyzer to assist security auditors in locating potential buffer overflow problems. This is a major programming project.
> 3. The possibility of allocating all buffers on the heap should be investigated. In particular, it should be examined how this affects the performance of real-world programs.

Literatur

[1] ALEPH ONE: *Smashing the Stack for Fun and Profit*. Phrack Magazine, 7(49), 1996. `http://www.phrack-dont-give-a-shit-about-dmca.org/phrack/49/P49-14`.

[2] AUSTIN, TODD M., SCOTT E. BREACH und GURINDAR S. SOHI: *Efficient detection of all pointer and array access errors*. In: *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, Seiten 290–301. ACM Press, 1994.

[3] BARATLOO, A., N. SINGH und T. TSAI: *Transparent run-time defense against stack smashing attacks*. In: *Proceedings of the USENIX Annual Technical Conference*, Juni 2000.

[4] BARATLOO, A., T. TSAI und N. SINGH: *libsafe: Protecting Critical Elements of Stacks*. `http://www.avayalabs.com/project/libsafe/`.

[5] BRUGGE, HERMAN TEN: *GCC bounds checking*. `http://web.inter.nl.net/hcc/Haj.Ten.Brugge/`.

[6] (CDC), CULT OF THE DEAD COW: *Back Orifice 2000*. `http://www.bo2k.com`.

[7] COMPUTER EMERGENCY RESPONSE TEAM: *CERT/CC Advisories*. `http://www.cert.org/advisories/`.

[8] COWAN, C., M. BARRINGER, S. BEATTIE und G. KROAH-HARTMAN: *Formatguard: Automatic protection from printf format string vulnerabilities*. In: *2001 USENIX security symposium*, Washington, D.C., August 2001.

[9] COWAN, CRISPAN, CALTON PU, DAVE MAIER, JONATHAN WALPOLE, PEAT BAKKE, STEVE BEATTIE, AARON GRIER, PERRY WAGLE, QIAN ZHANG und HEATHER HINTON: *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. In: *Proc. 7th USENIX Security Conference*, Seiten 63–78, San Antonio, Texas, jan 1998.

[10] COWAN, CRISPIN, PERRY WAGLE, CALTON PU, STEVE BEATTIE und JONATHAN WALPOLE: *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. In: *Proceedings of the DARPA information survivability conference and Expo*. Defense Advanced Research Projects Agency, 1999.

[11] DIK, CASPER H. S.: *Re: protect_stack script for Solaris 2.5.1*. Usenet article, `news:comp.security.unix`, Message-ID <casper.899327081@nl-usenet.sun.com>, Juli 1998.

[12] DILDOG: *The Tao of Windows Buffer Overflow*. `http://www.cultdeadcow.com/cDc_files/cDc-351/`.

[13] EICHIN, M. W. und J. A. ROCHLIS: *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*. In: *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, 1989.

[14] ETOH, HIROAKI: *GCC extension for protecting applications from stack-smashing attacks*. http://www.trl.ibm.com/projects/security/ssp/, 2001.

[15] FRADET, PASCAL, RONAN CAUGNE und DANIEL LE MÉTAYER: *Static Detection of Pointer Errors: An Axiomatisation and a Checking Algorithm*. In: *European Symposium on Programming*, Seiten 125–140, 1996.

[16] FRYKHOLM, N.: *Countermeasures against buffer overflow attacks*. http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.htm%l, November 2000.

[17] HAFNER, KATIE und JOHN MARKOFF: *Cyberpunk: Outlaws and Hackers on the Computer Frontier*. Simon & Schuster, 1991.

[18] HEUER, ANDREAS: *Geschichte des Kanarienvogels*. http://www.kanarienvogelzucht.de/id17.htm.

[19] iDEFENSE SECURITY, INC.: *iDEFENSE Security Advisory 03.31.03*. http://www.idefense.com/advisory/03.31.03.txt.

[20] KIL3R OF LAM3RZ: *3xterm.c*. http://emsi.it.pl/sploits/3xterm.c.

[21] KOLISHAK, ANDREY: *BOWall*. http://www.security.nnov.ru/bo/eng/BOWall/, 2000.

[22] LAROCHELLE, DAVID und DAVID EVANS: *Statically Detecting Likely Buffer Overflow Vulnerabilities*. In: *10th USENIX Security Symposium*, Seiten 177–190, August 2001. to appear.

[23] MCGARY, GREG: *Bounds Checking Projects*. http://gcc.gnu.org/projects/bp/main.html.

[24] MIXTER: *Writing buffer overflow exploits—a tutorial for beginners*.

[25] OPENWALL PROJECT MEMBERS: *The OpenWall Project*. http://www.openwall.com.

[26] RATIONAL SOFTWARE CORPORATION: *Rational Purify for NT*. http://www.rational.com/products/purify_nt/index.jsp.

[27] RUGINA, RADU und MARTIN RINARD: *Symbolic bounds analysis of pointers, array indices, and accessed memory regions*. In: *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, Seiten 182–195. ACM Press, 2000.

[28] SCHOLZ, BERHARD, JOHANN BLIEBERGER und THOMAS FAHRINGER: *Symbolic pointer analysis for detecting memory leaks*. In: *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, Seiten 104–113. ACM Press, 1999.

[29] SHANKAR, UMESH, KUNAL TALWAR, JEFFREY S. FOSTER und DAVID WAGNER: *Detecting Format String Vulnerabilities with Type Qualifiers*. In: *Proceedings of the 10th USENIX Security Symposium*, Seiten 201–220, 2001.

[30] SMITH, NATHAN P.: *Stack smashing vulnerabilities in the unix operating system.* `http://www.hack.gr/users/dawn/archives/ps/nate-buffer.ps.gz`.

[31] SNELTING, GREGOR, TORSTEN ROBSCHINK und JENS KRINKE: *Efficient Path Conditions in Dependence Graphs for Software Safety Analysis.* In: *Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, Mai 2002. ACM Press.

[32] SOLAR DESIGNER: *Linux kernel patch from the Openwall project.* `http://www.openwall.com/linux/`.

[33] SOLAR DESIGNER: *Getting around non-executable stack (and fix).* `http://ouah.bsdjeunz.org/solarretlibc.html`, August 1997.

[34] TSAI, TIMOTHY und NAVJOT SINGH: *Libsafe 2.0: Detection of format string vulnerability exploits.* `http://www.avayalabs.com/project/libsafe/`.

[35] VENDICATOR: *Stack shield: A "stack smashing" technique protection tool for linux.* `http://www.angelfire.com/sk/stackshield/`.

[36] VENDICATOR: *Stack Shield technical info file.* `http://www.angelfire.com/sk/stackshield/download.html`.

[37] VIEGA, JOHN, J. T. BLOCH, TADAYOSHI KOHNO und GARY MCGRAW: *Token-based scanning of source code for security problems.* ACM Transactions on Information and System Security, 5(3):238–261, 2002.

[38] WOJTCZUK, RAFAL: *Defeating Solar Designer non-executable stack patch.* `http://www.insecure.org/sploits/non-executable.stack.problems.html`, Januar 1998.

[39] XU, ZHICHEN, BARTON P. MILLER und THOMAS REPS: *Safety checking of machine code.* In: *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, Seiten 70–82. ACM Press, 2000.