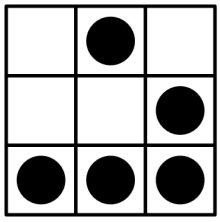
Hacking
in C



Reinhard Oertel

Inhaltsverzeichnis

1 Einleitung

Sucht man in den Nachrichten der bekannten Agenturen nach Hacker oder Hacking, so wird man schnell fündig. In letzter Zeit häufen sich die Meldungen zu immer neuen Attacken. Sei es das Auslesen der Informationen von hundert Millionen Nutzer des Playstation-Netzwerks, das Knacken vom Zertifikatsausteller Diginotar bei dem mehr als 500 "vertrauenswürdige" SSL-Zertifikate unrechtmäßig erstellt wurden um sich dann als Google, CIA oder Facebook ausgeben zu können, oder die Attacke auf die populäre MySQL Website, die dazu verwendet wurde Schadsoftware zu verbreiten. Die Angriffe häufen sich und werden zu einer ernstzunehmenden Bedrohung nicht nur für Internetnutzer.

2 Definition Hacker / Hacking

2.1 Gebräuchliche Definition

Schaut man sich die Berichte über Hacker bzw. Hacking in den Medien an, so zeichnet sich ein negatives Bild über Hacker. Sie dringen in Computersysteme ein und beschaffen sich unrechtmäßig Informationen. Informationen über Personen, Unternehmen und Regierungen. Sie zerstören Daten durch den Einsatz von Viren oder legen industrielle Anlagen lahm.

Ein Hacker ist also, glaubt man den Medien, "jemand der über das Netzwerk in ein Computersystem eindringt um sich Informationen zu beschaffen oder diese zu zerstören".

2.2 Definition

Geht es allerdings nach dem "Jargon File"[1], der "Bibel" des Hackertums, so ist diese Verallgemeinerung schlichtweg falsch und respektlos. Eine allgemeine Definition ist jedoch schwer zu geben. Es gibt viele Strömungen innerhalb der Subkultur und diese haben ihre jeweils eigenen Überzeugungen.

Das "Jargon File" versucht jedoch diese Subkultur zu erfassen. Es wird seit 1975 von freiwilligen Autoren kontinuierlich erweitert und beschreibt das Selbstverständnis der Hacker und ihrer Szene. Laut dem Jargon File sind Hacker zusammengefasst "Experimentierfreudige Personen, die mit Fachkenntnissen eine Technologie beliebiger Art außerhalb ihrer normalen Zweckbestimmung oder ihres gewöhnlichen Gebrauchs benutzen bzw. anpassen". Sie unterstreichen vehement die Unterscheidung von Hacker und den sog. Cracker. Diese seien es nämlich, die in Computersysteme eindringen oder Sicherheitssysteme umgehen.

Diese Trennung in gute und böse Hacker reicht jedoch nicht aus um die Strömungen der Hackerkultur zu erfassen.

2.3 Einteilung

2.3.1 Die „wahren“ Hacker

Die wahren Hacker sind die Hacker wie sie im Jargon File beschrieben werden. Die wahren Hacker der ersten Stunde waren IT-Spezialisten an amerikanischen Elite-Universitäten mit Beginn 70er. Heute sind sie hauptsächlich in der Linux Szene tätig und engagieren sich in Open Source und der Free Software Szene.

2.3.2 Die „aufklarerischen“ Hacker

Die aufklärerischen Hacker finden sich hauptsächlich im CCC tätig. Sie versuchen die Öffentlichkeit und die Verbraucher vor möglichen Gefahren oder Unsicherheiten zu schützen. Beispielsweise bewies der Chaos Computer Club die Manipulierbarkeit des elektronischen Wahlstifts[2], welcher für die Wahlen in Hamburg eingesetzt werden sollte.

1 Jargon File → siehe http://www.catb.org/jargon/html/
2 http://www.ccc.de/updates/2007/wahlstift-hack

Aber auch Hacker rund um Metasploit[3], SecurityFocus[4], u.a. zählen zu dieser Gruppe. Sie suchen nach bestehenden Sicherheitslücken in verbreiteter Software und veröffentlichen diese auf den genannten Websites. Diese Lücken werden dann (meist) von den Herstellern geschlossen. Sie wirken damit aktiv bei der Abwehr der Ausnutzung der Sicherheitslücken mit und erhöhen somit den Schutz der Verbraucher.

2.3.3 Die „Cracker“

Die Cracker, auch Black-Hat-Hacker gennant, sind die Hacker, die ihr Können für böse und schlechte Machenschaften einsetzen. So sehen es jedenfalls die "wahren Hacker".

Cracker sind hauptsächlich in der Warez Szene tätig. Sie umgehen Sicherheitsmechanismen und sehen dies jedes Mal als Herausforderung an. Cracker sind es auch, die Viren, Trojaner und Ähnliches entwickeln.

Aus diesen Tätigkeiten hat sich sogar ein Gewerbe entwickelt. So kann man heutzutage mit unentdeckten Schwachstellen in bekannten und häufig eingesetzten Programmen eine große Menge Geld machen. Diese Sicherheitslöcher werden an den Höchstbietenden versteigert und dienen dann später der Industriespionage, dem Aufbau von Bot-Netzen und Ähnlichem. Die Frankfurter Rundschau berichtet[5] sogar, dass die Größe des Hacker-Gewerbes in China beispielsweise bei einer Milliarde Euro liegt.

2.3.4 Die „Polithacker“

Die Polithacker verfolgen meist politische Ziele. So wurde beispielsweise der Virus Stuxnet entwickelt um das iranische Atomprogramm lahmzulegen. Stuxnet hat aber auch deutsche Industrieanlagen befallen. Auch gibt es immer wieder Hackerangriffe die als "Vergeltungsschlag" anzusehen sind. So gibt es zahlreiche Angriffe auf Behörden und Polizei als Antwort auf Verhaftungen innerhalb der Szene.

Ein anderes Beispiel sind die jüngsten Angriffe chinesischer Hacker[6] auf Google. Betroffen waren "US-Regierungsvertretern, Aktivisten aus China, US-Militärs, Journalisten sowie Regierungsvertretern aus mehreren asiatischen Ländern"[7].

3 http://metasploit.com/
4 http://www.securityfocus.com/
5 http://www.fr-online.de/digital/cyberkriminalitaet-in-china-hacker-gewerbe-macht-milliardenumsatz,1472406,3046202.html
6 http://www.welt.de/wirtschaft/article13408272/Chinesische-Hacker-greifen-Google-Dienst-Gmail-an.html
7 Autor unbekannt, "Chinesische Hacker greifen Google-Dienst Gmail an" auf welt.de Stand: 01.06.2011. http://www.welt.de/wirtschaft/article13408272/Chinesische-Hacker-greifen-Google-Dienst-Gmail-an.html (abgerufen am 26.09.2011)

3 Techniken

3.1 Buffer-Overflow

Buffer-Overflow (Pufferüberlauf) ist eine der ältesten und am häufigsten angewendete Technik beim Hacking. Dabei werden die Eigenschaften der Von-Neumann-Architektur und der Programmiersprache C ausgenutzt.

Gemäß der Architektur Von-Neumanns befinden sich ausführbarer Code und die Daten in einem gemeinsamen Speicherbereich. Dadurch wird es möglich Code als Daten zu interpretieren und Daten als Code auszuführen. C verzichtet zu Gunsten der Performance auf eine automatische Speicherverwaltung und -Überwachung, die eine Überschreitung von Speicherbereichen verhindern würde.

3.1.1 Hintergrundwissen - Stack

Der Stack, auch Stapelspeicher oder Kellerspeicher, ist eine einfache Datenstruktur funktionierend nach dem Last-In-First-Out Prinzip. Ein Stack besitzt als wichtigste Basisoperationen *push* und *pop*.

push legt dabei ein Objekt auf den Stapel, *pop* nimmt das oberste Objekt vom Stapel.

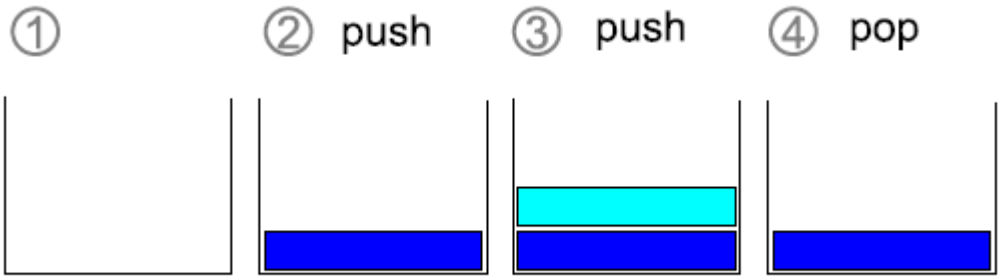


Abbildung 1: Stack-Operationen

In der Programmiersprache C ist der Stack die Ablage für die lokalen Variablen einer Funktion wenn der Speicherplatz für diese nicht mit *malloc()* alloziert wurde. Auch die Aufruf-Parameter von Funktionen werden auf dem Stack übergeben. Bei dem Aufruf einer Funktion wird weiterhin die Rücksprungadresse auf dem Stack gespeichert, damit die aufgerufenen Unterprogramme dorthin zurück springen können.

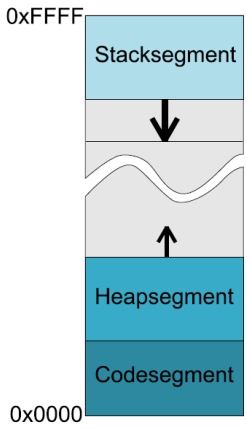


Abbildung 2: Segmente

Ein Programm in Ausführung besteht aus drei wichtigen Bereichen: dem Stacksegment, dem Heapsegment und dem Codesegment. Stack- und Heapsegment wachsen dabei in unterschiedliche Richtungen. Das Stacksegment wächst in Richtung der niedrigen Adressen des Adressraums, das Heapsegment wächst in Richtung der hohen Adressen.

Jede Funktion hat dabei ihren eigenen Stack-Ausschnitt innerhalb des Stacksegments. Dieser ist begrenzt durch den EBP (Base Pointer), die Basisadresse des Ausschnitts und

durch den Zeiger ESP (Stack Pointer) auf das aktuell "oberste" Objekt. Der Stack Pointer wird bei Verwendung von *push* und *pop* jeweils angepasst.

Am Beispiel eines Funktionsaufrufs lässt sich dies nachvollziehen, wenn man den erzeugten Assembler-Code betrachtet.

```
void function(int a, int b, int c)
{
    int* pReturn;
    // ...
}

int _tmain(int argc, _TCHAR* argv[])
{
    function (1, 2, 3);
}
```

Abbildung 3: Beispiel: Funktionsaufruf (I)

In Abbildung 3 wird aus der *main*-Funktion (*_tmain*) eine einfache Funktion mit drei Parametern aufgerufen. Betrachtet man sich den Assembler Code, kann man die Verwendung des Stacks ersehen.

```
int _tmain(int argc, _TCHAR* argv[])
{
 push        ebp
 mov         ebp,esp
 sub         esp,0C0h
 push        ebx
 push        esi
 push        edi
 lea         edi,[ebp-0C0h]
 mov         ecx,30h
 mov         eax,0CCCCCCCCh
 rep stos    dword ptr es:[edi]
    function(1,2,3);
 push        3
 push        2
 push        1
 call        function (135113Bh)
 add         esp,0Ch
}
```

```
void function(int a, int b, int c)
{
 push        ebp
 mov         ebp,esp
 sub         esp,0CCh
 push        ebx
 push        esi
 push        edi
 lea         edi,[ebp-0CCh]
 mov         ecx,33h
 mov         eax,0CCCCCCCCh
 rep stos    dword ptr es:[edi]
    int* pReturn;
    // ...
}
```

Abbildung 4: Bespiel: Funktionsaufruf (II)

Die wichtigen des Funktionsaufruf sind die Zeilen unter `function(1,2,3);`. Die Parameter für den Aufruf der Funktion werden in umgedrehter Reihenfolge auf den Stack gelegt. Danach wird die gewünschte Funktion per *call* aufgerufen. Der Aufruf *call* legt die nächst höhere Befehlsadresse der Funktion *main* (*_tmain*) ebenfalls auf den Stack. Nach Abarbeitung der Unterfunktion wird zu dieser Adresse zurück gesprungen und das Hauptprogramm fortgesetzt.

In der Funktion *function* wird durch den Befehl *push ebp* der alte Base Pointer des Stack-Ausschnitt der aufrufenden Funktion gesichert. Per darauf folgendem *mov ebp, esp* wird der aktuelle Stack Pointer zum neuen Base Pointer. Es wird also ein neuer Stack-Ausschnitt hinter den bereits bestehenden Ausschnitten festgelegt. Die darauf folgenden Befehle sind für uns nicht relevant. Wird nun eine lokale Variable angelegt, so wird diese auch auf den Stack gelegt.

Nach dem Aufruf der Unterfunktion und den Vorbereitungen für die Erzeugung eines neuen Stack-Ausschnitts sieht der Stack (vereinfacht) wie in Abbildung 5.

Die Parameter sind in umgekehrter Reihenfolge auf dem Stack und können so von der Funktion ausgelesen werden. Zur Rücksprung-Adresse (RET) wird nach Beendigung der Unterfunktion zurückgesprungen um den Programmfluss aufrecht zu erhalten. Der Base Pointer der aufrufenden Funktion und somit dessen Stack-Ausschnitt wurde ebenfalls gesichert. Danach folgen die lokalen Variablen.

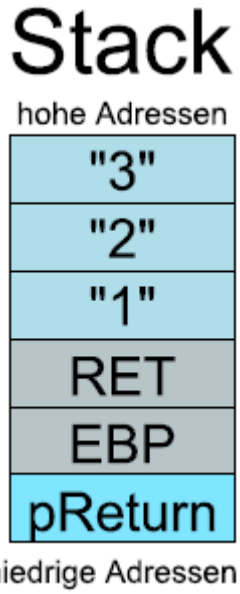


Abbildung 5: Bespiel: Funktionsaufruf - Stack

3.1.2 Stack-Overflow

Bei einem Stack-Overflow - oder Pufferüberlauf im Stack-Speicher macht man sich die Eigenschaften des Stacks bei einem Funktionsaufruf zu Nutze. Dabei werden zu große Datenmengen in einen zu kleinen reservierten Speicherbereich im Stack geschrieben. Da C keine automatische Überwachung der Speicherbereichsgrenzen besitzt, werden die Speicherstellen hinter dem Puffer überschrieben.

Abbildung 6: Arraypuffer im Stack

Abbildung 6 zeigt ein Array im Stack. Dieses enthält 12 Elemente. Befinden sich nun an der relativen Adresse 13 und 14 der RETURN-Pointer und der Base Pointer, so können diese Informationen beim Befüllen des Arrays mit mehr als 12 Elementen überschrieben werden.

Ursache dafür ist, wie schon erwähnt, die Von-Neumann-Architektur, die Code und Daten nicht trennt. Im Stack befinden sich neben den Daten auch Verwaltungsinformationen.

Werden diese überschrieben, so ändert sich das Programmverhalten oder das Programm stürzt ab.

Eine weitere Ursache ist, dass C keine Strings kennt. Strings werden als *char* Arrays realisiert. Viele C-Funktionen wie z.B. *strcpy()* kopieren Daten (Strings) ohne Längenüberprüfung bis zum lesen der abschließenden 0 (\0). Die Verwendung dieser unsicheren Funktionen ist die häufigste Ursache der Stack-Overflows.

Die Möglichkeiten die sich durch das Überschreiben des Stack-Inhalts ergeben ist zum einen die Möglichkeit zum Überschreiben von lokalen Variablen. Auch die Rücksprung-Adresse kann dadurch geändert werden. Dies führt zu einer Änderung des Programmverhaltens. Es ist beispielsweise möglich, die Rücksprung-Adresse in den Puffer im Stack zeigen zu lassen. Dadurch kann beliebiger Code ausgeführt werden - wenn der Puffer groß genug ist.

Abbildung 7 zeigt ein Beispiel zur Veränderung der Rücksprung-Adresse. Dies ist zwar nur konstruiert, es zeigt jedoch, dass die Möglichkeit gegeben ist.

```
void function(int a, int b, int c)
{
    int* pReturn;

    pReturn = &a;
    --pReturn;
    (*pReturn) += 10;
}


int _tmain(int argc, _TCHAR* argv[])
{
    int x;

    x = 0;
    printf("\n x = %d\n\n",x);
    function(1,2,3);
    x = 1;
    printf(" x sollte jetzt 1 sein.\n\n x = %d\n\n",x);

    return 0;
}
```

Abbildung 7: Beispiel: Änderung der Rücksprung-Adresse

Nach Deklaration der Variable x in der Funktion *main* (*_tmain*) wird diese auf 0 gesetzt und der Wert auf der Konsole ausgegeben. Die aufgerufene Funktion *function* speichert in dem int-Pointer *pReturn* die Adresse des Parameters *a*. Diese liegt im Stack (siehe Abbildung 5) und befindet sich vor der Rücksprung-Adresse (RET Pointer).

Durch Verringerung der Adresse in pReturn haben wir also Zugriff auf die Rücksprung-Adresse und können diese Verändern. Ziel ist es in die *main* Funktion zurück zu springen, jedoch nach der Zuweisung *x = 1;* Dazu muss zu dem Inhalt der Variable *pReturn* der Wert 10 addiert werden. Dies entspricht eine Erhöhung der Rücksprung-Adresse um 10 Befehle. Der Wert 10 ist nicht willkürlich und ergibt sich aus der Anzahl der Assembler-Befehle, die nach Ausführung der Unterfunktion *funktion* noch im Hintergrund ausgeführt werden, ehe wieder mit der Abarbeitung der programmierten Befehlen begonnen wird. Diese Zahl kann je nach verwendeten Compiler und System unterschiedlich sein. Führt man den obigen Code (unter Windows) aus, wird die Variable x nie den Wert 1 besitzen.

Ein prominentes Beispiel für die Ausnutzung der Stack-Overflows war beispielsweise der "Twilight Hack“ für die Wii. Dabei musste ein angepasstes Savegame für das Spiel "The Legend of Zelda: Twilight Princess" geladen werden. In dem Savegame wurde dem Pferd „Epona“ ein zu langer „Name“ gegeben. Dies führte zu einem Stack Overflow. Es konnte beliebiger Schadcode ausgeführt werden.

3.1.2.1 Schutzmaßnahmen

Einen 100% Schutz vor Buffer-Overflows ist nur schwer zu realisieren oder bedeutet einen erheblichen Mehraufwand. Aus diesem Grund wird das Thema Schutz und Sicherheit von vielen Programmierern eher stiefmütterlich behandelt. In sicherheitskritischen Programmen sollten wenigstens die folgenden Aspekte bedacht werden.

Der Umgang mit Arrays, wenn diese nicht im dynamischen Speicher gehalten werden, sollte genau geprüft werden. Vergisst man die abschließende *0*, kann dies eine Sicherheitslücke bedeuten. Werden auf String-Arrays ohne abschließende 0 die String Funktionen ohne Längenüberprüfung (z.B. *strcpy(), strcat(), strlen()*) verwendet, kann es passieren, dass zusätzlicher Datenmüll gelesen oder geschrieben wird. Handelt es bei dem String-Arrays um Eingabedaten - aus einer Datei oder einem Eingabefeld - kann beliebiger Schadcode in das Programm eingeschleust werden.

Es ist daher ratsam die statt der Funktionen ohne Längenüberprüfung deren Pendant mit Längenüberprüfung zu verwenden. Diese haben ein *n* im Funktionsnamen (*strncpy(), strncat(), strnlen()*). Zusätzlich sollte man die zu verarbeitenden Daten überprüfen. Dabei sollte man die erlaubte Länge begrenzen um nicht beliebige Datenmengen zu verarbeiten. Kennt man die Gestalt der zu verarbeitenden Daten - sind es beispielsweise nur alphanumerische Zeichen - sollte man die eingelesenen Daten formatieren und alle unerlaubte (Steuer-) Zeichen vor der Verarbeitung filtern.

3.1.3 Hintergrundwissen - Heap Verwaltung

Um die Grundlagen von Heap-Overflows zu verstehen, muss man wissen wie ein Heap verwaltet wird.

Der Speicher im Heap kann über Bibliotheksfunktionen dynamisch reserviert werden. Dies geschieht häufig über die Funktionen wie *malloc(), calloc() und realloc()*. Die eigentliche Verwaltung geschieht durch die jeweilige Systembibliothek und ist von der Laufzeitumgebung abhängig.

Die Implementierung innerhalb der Systembibliothek ist durch doppelt verkettete Listen realisiert in denen der freie sowie der reservierte Speicher verwaltet wird. Die einzelnen Elemente der verketteten Liste bestehen dabei aus einem Header mit Verwaltungsinformationen und einem Speicherblock mit den reservierten Daten.

Der Header besteht unter anderem aus einem Zeiger zum jeweils nächsten und vorherigen Speicherblock. Weiterhin wird die Größe des reservierten Speichers gespeichert und ein Flag, ob dieser Speicherblock verwendet wird oder frei ist.

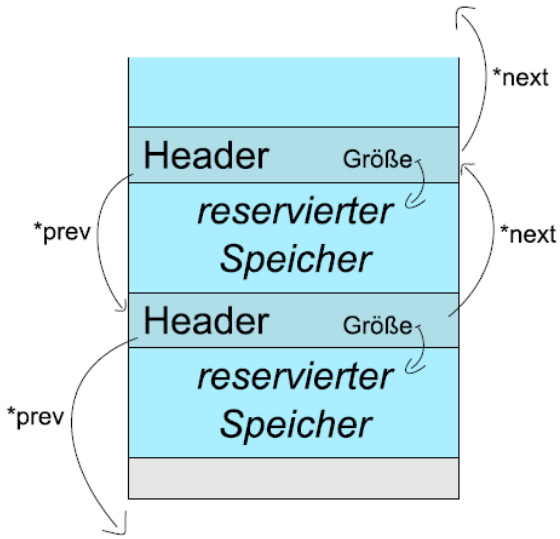


Abbildung 8: Speicherverwaltung im Heap

Der Grund für die notwendige Verwaltung ist die Fragmentierung des Speichers beim Freigeben

von reserviertem Speicher. Bei fortgeschrittener Fragmentierung kann es vorkommen, dass zwar noch genügend freier Speicher vorhanden ist, dieser aber nicht vergeben werden kann da der angeforderte Speicher größer ist als der größte zusammenhängend freie Speicherbereich. Die freigewordenen Speicherblöcke müssen also zusammengefasst werden. Diese Defragmentierung lässt sich dank der verketteten Liste durch die Anpassung der Zeiger auf den nächsten bzw. vorherigen Speicherblock relativ leicht realisieren.

3.1.4 Heap-Overflow

Der Heap-Overflow ist die zweite Generation der Buffer-Overflow Technik. Genau genommen müsste die Technik Buffer-Overlow im Heap heißen. Schätzungen zu Folge ist der Heap-Overflow der Grund für jede 2. Sicherheitslücke in Software. Er ist allerdings auch deutlich komplizierter zu realisieren als beispielsweise der Stack-Overflow.

Die Erzeugung eines Heap-Overflows wird meist über zu ladende Dateien realisiert. So ist es möglich, dass Schadcode über geladene Bilder, Dokumente, Savegames, PDFs, etc. ausgeführt wird.

Zwei Fehlerquellen können einen Heap-Overflow verursachen: Die Größe des reservierten Speichers beim Laden von Daten ist statisch. Man kann dann durch Anpassung der Daten (z.B. der zu ladenden Datei) dafür sorgen, dass mehr als die vorgesehene Menge Daten hineingeschrieben wird. Die andere Fehlerquelle ist, dass die Berechnung des benötigten Speichers bei der Reservierung fehlerhaft ist oder auf Angaben des Angreifers beruht.

Für einen erfolgreichen Heap-Overflow Angriff, ist es jedoch nötig, genau zu wissen, welche Heap-Implementierung das Programm zur Laufzeit verwendet, um die Verwaltungsinformationen gezielt mit manipulierten Werten zu überschreiben. Anders als auf dem Stack kann ein Angreifer auf dem Heap keine Speicherbereiche überschreiben, deren Inhalt das Programm direkt als Sprungadresse nutzt. Bei eine Heap-Overflow hat der Angreifer lediglich Kontrolle über die Daten für die Heap-Verwaltung, also die Zeiger **next* und *prev und die Felder der Größe und des Benutzungsflag im Header. Dies alles erschwert die Ausführung eines Heap-Overflows.

Um eigenen Code zur Ausführung zu bringen, muss der Angreifer also zusammengefasst dafür sorgen, dass das System einen Speicherbereich überschreibt, von dem es zu einem späteren Zeitpunkt eine Sprungadresse lädt.

Prominente Beispiele für den Einsatz eines Heap-Overflow ist der Nachfolger des "Twilight Hack" "*Bannerbomb*". Dieser nutze eine Schwachstelle in der JPEG Dekomprimierung beim Laden eines Banners. Weitere Beispiele sind die unzähligen iPhone Jailbreaks und der PS3 Exploit.

3.1.4.1 Schutzmaßnahmen

Da der Buffer-Overflow meistens über zu ladende Dateien ausgeführt wird, ist dies die erste Stelle zur Schutzmaßnahmen. Man sollte als Programmierer bei sicherheitskritischen Systemen immer die Konsistenz der eingelesenen Daten prüfen. Also die Richtigkeit der angegeben Verwaltungsinformationen zum Laden der Datei. Auch die Verwendung der GNU libc kann zum Schutz beitragen. Diese prüft die verwendeten Pointer auf Gültigkeit (vergleichbar mit Smart Pointern).

Eine andere Möglichkeit liegt in der Verhinderung der Ausführung von Heap-Speicher. Ab Windows XP SP2 wird dies durch die DEP-Technologie (*Data Execution Prevention, Datenausführungsverhinderung*) unterstützt. Dabei werden Speicherbereiche als ausführbar bzw nicht ausführbar gekennzeichnet.

Eine weitere Möglichkeit ist die so genannte Address Space Layout Randomization (ASLR). ASLR vergibt den Programmen zufällig die Adressbereiche, somit ist das System praktisch nicht mehr deterministisch.

4 Quellen

- Smashing The Stack For Fun And Profit
  - http://www.phrack.org/issues.html?id=14&issue=49

- Vermeiden von Sicherheitslochern beim Entwickeln einer Applikation - Teil 3: Buffer Overflow
  - http://www.linuxfocus.org/Deutsch/May2001/article190.shtml

- w00w00 on Heap Overflows
  - http://packetstormsecurity.org/files/view/13877/w00w00-heap-overflows.txt

- Eingelocht: Buffer-Overflows und andere Sollbruchstellen
  - http://www.heise.de/security/artikel/Eingelocht-270148.html

- Buffer overflow
  - http://en.wikipedia.org/wiki/Buffer_overflow
  - ff.

- Ein Haufen Risiko
  - http://www.heise.de/security/artikel/Ein-Haufen-Risiko-270800.html