Verwundbarkeiten durch Buffer-Overflow und Format-Strings

Martin Ofner
(ofner@net.in.tum.de)

Seminar „Internetsicherheit“
Technische Universität München

WS 2003 (Version 1. März 2004)

Zusammenfassung

Verwundbarkeiten durch Buffer-Overflow und Format-Strings sind de facto die grössten Sicherheitslücken, welche zur erfolgreichen Kompromittierung eines Systems ausgenutzt werden können. Dieses Dokument soll einen Überblick über die technischen Grundlagen, Ursachen und Gegenmaßnahmen geben.

1 Einleitung

Fast täglich kann man in den diversen Mailinglisten und Security-Online-Archiven Nachrichten von neuen Einbruchsmöglichkeiten und Verwundbarkeiten in Anwendungen oder Betriebssystemen lesen. In vielen Fällen ist die Ursache ein Pufferüberlauf, englisch Buffer-Overflow. Mit diversen Tricks können Angreifer solche Buffer-Overflows nutzen, um in ein System einzubrechen. Buffer-Overflow Attacken treten nur bei C-Programmen auf, bedingt durch das Konzept der binären Null. In C sind Strings Zeiger auf eine Zeichenarray. Das Ende des Strings wird lediglich durch das Null Byte erkannt. Die Problematik ensteht hier, dass ein zu langer String das Null Byte und darauf folgenden Code überschreiben kann und somit eigener Code in fremde Systeme eingeschleust werden kann. Was passiert nun wirklich bei einem Buffer-Overflow? Wie können Angreifer einen solchen Buffer-Overflow ausnutzen? Gibt es Möglichkeiten, diese Art des Systemeinbruchs zu verhindern?

2 Verwundbarkeit des Buffer-Overflow

Innerhalb der letzten Jahre sind Angriffe auf Systeme, die sich der Verwundbarkeit des Buffer-Overflow bedienen, sprunghaft angestiegen.

In den meisten Fällen handelt es sich dabei um einen Überlauf von dynamischen Puffern (engl. *buffer*), auch bekannt unter dem Begriff *stack based buffer*. Diesen Puffern wird zur Programmlaufzeit Speicher auf dem Keller (engl. *stack*) reserviert. Um zu verstehen, was *stack buffers* sind, muss erst verstanden werden, wie ein laufender Prozess im Speicher organisiert wird.

2.1 Speichermanagement eines Prozesses

Prozesse teilen sich den Speicher in drei Bereiche auf:

- text
- data
- stack

In dem *text* Bereich befindet sich der auszuführende Code (Instruktionen). Dieser Datenbereich ist schreibgeschützt. Sollte dennoch versucht werden in diesen Bereich zu schreiben gibt es eine *segmentation violation* (Speicherzugriffsverletzung).

Bevor wir uns die anderen Bereiche ansehen, noch einige Dinge zu Variablen in C. Die globalen Variablen werden im ganzen Programm benutzt, wohingegen die lokalen Variablen nur in der Funktion, in der sie deklariert wurden, benutzt werden können. Die statischen Variablen haben eine feste Gröe, sobald sie deklariert wurden. Die Größe hängt vom Typ ab. Typen sind z.B char, int, double, pointer, etc. Der *data* Bereich enthält initialisierte globale statischen Daten (z.B. Programmkonstanten). Dieser Bereich wird zur Kompilezeit reserviert, da die Größe bekannt ist.

Was passiert aber mit lokalen Daten lokaler Varibalen? Da Funktionen rekursiv aufgerufen werden können, ist die Anzahl der benötigten lokalen Variablen und deren Speicherplatz von vorneherein nicht bekannt. Ihr Speicherplatz wird erst zur Laufzeit auf dem Keller reserviert.

2.2 Der Keller

Ein Keller ist ein abstrakter Datentyp. Auf diesem sind folgende Operation definiert:

- PUSH: Legt ein Element auf dem Keller ab
- POP: Nimmt das erste Element von dem Keller herunter

Diese charakteristische Eigenschaft wird als last in, first out (LIFO) bezeichnet. Der Keller ist organisiert in logischen Elementen, den Keller Frames. Der Kellerzeiger SP (engl. *stack pointer*) zeigt dabei immer auf das zuletzt abgelegte Element. Bei jedem Funtionsaufruf wird ein neuer Frame (engl. *stack frames*) auf dem Keller erstellt. Dieser enthält alle lokalen Variablen und Parameter, die für die Funktion benötigt werden, sowie den Wert Befehlszähler IP (engl. *instruction pointer*) zum Zeitpunkt des Unterfunktionsaufrufs (um später wieder zum Hauptprogramm zurückkehren zu können). Man bedient sich innerhalb der

Frames noch eines Framezeigers EPB (engl. *frame pointers*) zur relativen Adressierung der lokalen Variablen und Parametern. Alle lokalen Variablen werden relativ zum Framezeiger adressiert, anstatt relativ zum Kellerzeiger. Ansonsten würden sich, beispielsweise nach jedem Unterprogrammaufruf (ändert den Kellerzeiger) die relativen Adressen der lokalen Variablen bzw. Parametern ändern. Der Keller wird vor allem benutzt, um Unterprogrammaufrufe zu realisieren. Vor jedem Funktionsaufruf wird der Prolog (engl. *procedure prolog*) ausgeführt:

- Ablegen der Funktionsparameter auf dem Keller
- Ablegen des aktuellen Befehlszähler auf dem Keller
- Speichern des aktuellen Framezeigers
- Reservieren des Speichers für die lokalen Variablen

Die noch folgenden Beispiele basieren auf einer Intel-Architektur. Hier wird der Speicher nur in Viellfachen der Wortlänge (4 Bytes) adressiert, d.h ein Puffer der Länge 5 belegt tatsächlich 8 Bytes.

2.3 Buffer-Overflow

Ein Buffer-Overflow liegt vor, sobald versucht wird mehr Daten in einen Puffer zu schreiben, als dieser zu handhaben bereit ist. Benutzt die Anwendung dabei Funktionen, die keine Längenüberprüfung vornehmen, werden Adressen überschrieben, die nicht zu dem Puffer gehören (typischerweise Rücksprungadresse). Bei geschickter Wahl der Eingabedaten ist es möglich die Rücksprungadresse mit einem gültigen Wert zu überschreiben, und man erhält somit Kontrolle über den Programmfluss. In den meisten Fällen möchte man natürlich einen Shellzugriff (Kommandozeile) auf dem Zielsystem erreichen, mit dem man dann alle Möglichkeiten ausschöpfen kann. Aber wie können wir jetzt die notwendigen Instruktionen ausführen, die uns eine Shell bereitstellen? Man legt den Shellcode in den Puffer ab, den man überlaufen lassen kann und überschreibt die Rücksprungadresse, so dass sie wiederum auf den Anfang unseres Puffers und damit auf den Shellcode zeigt. Nach Ablauf der Unterfunktion und vermeintlichem Rücksprung zum Hauptprogramm wird tatsächlich zu dem Shellcode gesprungen und dieser ausgeführt. Der Shellcode ist natürlch plattform- und prozessorabhängig. Der Shellcode wird normalerweise einem verwundbaren Programm über ein Kommandozeilenargument, eine Umgebungsvariable oder über eine Benutzereingabe übergeben.

2.4 Beispiel

Zur Veranschaulichung jetzt ein Beispiel, das die theoretischen Ausführungen verdeutlichen soll. Das Beispiel ist eine Kommandozeilenanwendung, der man als Parameter einen String beliebiger Länge übergeben kann.

```
vulnerable.c
void main(int argc, char *argv[])
```

```
{
  char buffer[512];

  if( argc > 1 )
    /* schreiben in buffer, ohne Laenge zu ueberpruefen*/
    strcpy(buffer, argv[1]) /*argv[1] ist hierbei der */
                            /*erste Kommandozeilenparameter*/
}
```

Auch für main() wird der Prolog ausgeführt und es ergibt sich folgendes Bild auf dem Keller (bei jedem Funktionsaufruf landen Parameter, Rücksprungadresse und lokale Variablen auf dem Keller)

Jetzt muss nur noch ein geeigneter String dem Programm übergeben werden. Der String sollte den Shellcode und unsere Rücksprungadresse enthalten. Die restlichen Bytes werden mit beliebigen Werten aufgefüllt.

Die Funktion strcpy() kopiert nun den 520 Bytes langen String in einen Puffer, für den nur 512Bytes auf dem Keller reserviert sind. Die Folge ist das Überschreiben des Framezeigers und natürlich der Rücksprungadresse.

Die Schwierigkeit ist natürlich jetzt, die Rücksprungadresse mit einem geschickten Wert zu überschreiben (am besten natürlich eine Adresse die in den überlaufenden Buffer zeigt, wo der Shellcode bereitliegt). Angreifer schreiben sich kleine Programme, die als Parameter eine Pufferlänge und den Abstand zur Rücksprungadresse (von da an, wo man den überlaufenden Puffer auf dem Keller erwartet) annemhem. Das Programm konstruiert den String mit den Informationen aus den Parameter, und ¨ubergibt ihn der anzugreifenden Anwendung. Der Angreiferlässt sein Programm solange laufen, bis er die richtigen Werte ermittelt hat. Um die Parameter in den richtigen Dimensionen einschätzen zu können, hilft ihm auch die Analyse der Quellcodes und der Assemblercodes der anzugreifenden Anwendungen [1].

Da die Rücksprungadresse auf vier Bytes absolut exakt sein muss, kann das Raten der richtigen Abständen sehr aufwendig werden. Um die Effizienz zu erhöhen, bedient man sich sogenannter *null operations* (NOP) der Prozessoren. Diese werden für gewöhnlich benutzt, um Zeitverzögerungen während der Programmausführung zu realisieren. Man platziert den Shellcode in der Mitte des Strings, und füllt den Rest

mit *null operations*. Sobald die vom Angreifer bereitgestellte Adresse in den überlaufenden Puffer zeigt, werden solange vom Prozessor *null operations* ausgeführt, bis der Befehlszähler den Shellcode erreicht.

Den Shellcode erhält man, indem man für die Zielumgebung ein Programm schreibt, dass eine Shell startet. Anschliessend analysiert man den compilierten Code und erhält schliesslich den Binärcode in hexadezimal. Unter Linux (Intel-Prozessor) ist dies beispielsweise:

```
shell.c
void main(int argc, char *argv[])
{
  execv();
}

char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Zu einem umfassenden Einstieg zur Generierung von Shellcodes eignet sich der Artikel von Aleph One [1].

2.5 Zusammenfassung und Gegenmaßnahmen

Was braucht man, um einen Buffer-Overflow ausnützen zu können?

- Buffer, den man überlaufen lassen kann
- Abstand der Pufferadresse zur Rücksprungadresse, um die Rücksprungadresse überschreiben zu können
- Shellcode für das Zielsystem

Der beste Schutz gegen Buffer-Overflows ist sicherheitsbewusste Programmierung. Software-Entwickler sollten sich unbedingt über die entsprechenden Fallstricke informieren und eine bewusst defensive Programmierung anstreben. Bei der Programmierung mit C/C++ kann man ausserdem auf diverse Hilfsmittel zurückgreifen, die zumindest helfen, solche Sicherheitslücken zu vermeiden. StackShield [6] sichert in Linux-Programmen bei jedem Funktionsaufruf die Rücksprungadresse und korrigiert sie bei Bedarf vor dem Rücksprung. Das Programm fügt dazu entsprechenden Code am Beginn und Ende jedes Funktionsaufrufs ein. Auch StackGuard [7] versucht, auf Unix-Systemen die Rücksprungadresse bei Funktionsaufrufen zu schützen. Dazu platziert es auf dem Keller direkt daneben ein so genanntes Canary. Dahinter verbirgt sich ein spezielles Kontrollzeichen, dessen Wert StackGuard vor jedem Rücksprung aus einer Funktion überprüft. Hat er sich geändert, schreibt das Programm eine Warnmeldung ins Syslog und beendet sich. Um das Problem an der Wurzel zu packen, könnte man den Keller als nicht ausführbar markieren. Damit löst der Versuch, Code auf dem Keller auszuführen, eine Speicherschutzverletzung aus. Dies muss allerdings auf Betriebssystemebene implementiert sein und bringt diverse Kompatibilitätsprobleme mit sich. Von Solar Designer gibt es einen

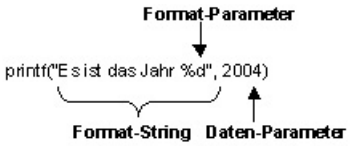
entsprechenden Patch für Linux [8]. Allerdings haben Sicherheitsexperten bereits darauf hingewiesen, dass auch dies nicht der Weisheit letzter Schluss ist. Außerdem gibt es von SecureWave mit SecureStack eine Implementierung für Windows NT/2000, bei der allerdings derzeit die Windows-2000-Version noch übermäßige Geschwindigkeitseinbußen mit sich bringt [9].

3 Verwundbarkeit des Format-Strings

Im Gegensatz zum Buffer-Overflow ist die Verwundbarkeit des Format-Strings erst seit Mitte 1999 in das Interesse der Öffentlichkeit gerückt. Verwundbarkeiten des Format-Strings können sehr gut durch entsprechende Tools im Quellcode identifiziert werden. Die Gefahr liegt hier eher in der Unterschätzung des Sicherheitsrisiko seitens der Entwickler. Dadurch, dass bislang nur wenige Angriffe bekannt geworden sind (im Gegensatz zu Buffer-Overflows) ist das Sicherheitsbewusstsein bei Format-Strings eher gering.

3.1 Format-String Funktionen

Format-String Funktionen wandeln trivialerweise einfache C-Datentypen in einen String um. Dabei ist es möglich anzugeben, wie die Daten ausgegeben werden sollen (mittels des Format-Strings) und welche Ausgabe benutzt werden soll (z.B. Standardausgabe). Der erste Parameter einer Format-Funktion ist immer der Format-String, die restlichen Parameter geben die auszugebenden Daten an. Innerhalb des Format-Strings dienen Format-Parameter (beginnt mit %) als Platzhalter und Interpretierungsanweisung für die auszugebenden Daten. Die Anzahl



der Format-Parameter muss gleich der Anzahl der auszugebenden Daten sein. Dabei ist zu beachten, dass zum Zeitpunkt der Ausführung einer Format-Funktion die Daten-Paramter auf den Keller gelegt werden. Die Format-Funktion holt sich dann für jeden Format-Paramter den nächsten Daten-Paramter vom Keller.

Sobald ein potentieller Angreifer die Möglichkeit hat, den Inhalt des Format-Strings teilweise oder komplett selbst zu bestimmen, liegt eine Verwundbarkeit des Format-Strings vor. Dabei wird das Verhalten der Format-Funktion beeinflusst, und es kann so der Programmfluss unter Kontrolle gebracht werden.

3.1.1 Die Rolle des Kellers

Wie bereits angesprochen, werden die auszugebenden Werte zuerst auf dem Keller abgelegt. Die Format-Funktion holt sich dann zur Ausga-

%d	Dezimal
%x	Hexadezimal
%s	String
%n	Anzahl der bisher geschrieben Bytes

Tabelle 1: Format-Paramter

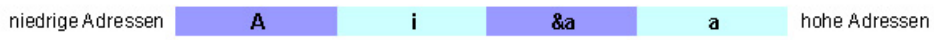
printf	Ausgabe auf die Standardausgabe
sprintf	Ausgabe in einen String
snprintf	Ausgabe in einen String mit Längenüberprüfung

Tabelle 2: Übersicht der wichtigsten Format-String benutzenden Funktionen

be die vom Format-String gewünschten Parameter vom Keller in der angebenden Reihenfolge.

```
printf("Zahl %d hat keine Adresse, Zahl %d schon: %08x", i, a, &a);
```

Zum Zeitpunkt des Ausführens von printf() sieht der Keller folgendermassen aus.




```
A = Adresse des Format-Strings
i = Wert von der Variable i
a = Wert von der Variable a
&a = Adresse der Variable a
```

Die Format-Funktion liest den String bei A (Format String) nun Zeichen für Zeichen, und prüft, ob das Zeichen ein '%' ist. Ist dies nicht der Fall, so wird es einfach kopiert (in die Ausgabe oder den String), ansonsent wird der nächste Wert vom Keller gelesen und anhand des Format-Paramters interpretiert und dementsprechend in die Ausgabe eingefügt.

3.1.2 Beispiel

Hierzu ein erläuterndes Beispiel. Die Variable *user* wird vom Benutzer (z.B. als Komandozeilenparameter) bereitgestellt.

Bei *printf(user)* wird nicht explizit ein Format-String angegeben. Falls der Benutzer nun einen String übergibt, der gültige Format-Paramter enthält (z.B. %s%s), interpretiert die Format-Funktion die Benutzereingabe als Format-String. Demzufolge wird auch versucht so oft vom Keller zu lesen, wie Format-Parameter vorkommen. Da aber keine Daten-Parameter übergeben werden (die auf dem Keller

Richtig	**Falsch**
printf(„%s“,user)	printf(user)

gelegt werden), liest die Format-Funktion einfach die nächsten Daten auf dem Keller und versucht sie dem Format-Parameter entsprechend zu interpretieren und auszugeben. Im einfachsten Fall endet dies in einer Speicherverletzung, im schlimmsten Fall erhält ein Angreifer die Kontrolle über den Prozessablauf (mittels Manipulation des Kellers. Bei printf(„%s“,user) wird die Benutzereingabe immer als Daten-Parameter interpretiert. Eine Manipulation von außen ist somit ausgeschlossen.

3.2 Was kann man kontrollieren?

Nachdem die Grundlagen der Verwundbarkeit des Format-String nun bekannt sind, stellt sich jetzt die Frage, welche Möglichkeiten ein Angreifer jetzt tatsächlich besitzt.

3.2.1 Anzeigen des Prozessspeichers

Der wohl interessanteste Aspekt der Verwundbarkeit des Format-Strings ist die Möglichkeit *beliebige* Speicheradressen auslesen zu können. Voraussetzung ist natürlich, dass man die Ausgabe der Format-Funktion sehen kann.

Den Keller kann man beispielsweise mit folgenden Format-String auslesen:

```
"%08x.%08x.%08x.%08x.%08x"
```

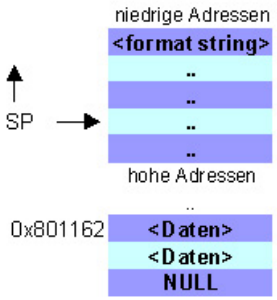
Jedes %08x bewirkt hier, dass die Format-Funktion den nächsten Wert vom Keller holt. So werden hier die nächsten fnf Adressen vom Keller gelesen und achtstelig ausgegeben. Je nachdem wie lang der Format-String sein darf, ist es möglich den kompletten Keller zu rekonstruiren. Dies ist nützlich, um Informationen über den Programmfluss zu erhalten und ist enorm hilfreich um u.a. korrekte Abstände für Buffer-Overflows zu ermitteln.

Um beliebigen Speicher anzuzeigen, muss man die Format-Funktion dazu bringen, als aktuellen Daten-Parameter eine von uns gegebene Adresse zu benutzen. Wenn dass der Fall ist, kann man mit dem Format-Parameter %s den String bei dieser Adresse (entspricht dem Speicherinhalt) bis zum nächsten Null Byte, anzeigen lassen (das Ende eines Strings wird immer mit einem Null Byte gekennzeichnet).

Beispiel:

```
"\x62\x11\x80\x00_%08x.%08x.%08x.%s"
```

In den Format-String legen wir die Adresse, ab der wir den Speicher anzeigen lassen wollen in hexadezimaler Schreibweise (x62x11x80x00). Läßt man nun genügend oft vom Keller lesen (jedes %08x im Format-String), so zeigt der Kellerzeiger irgendwann in den vom Angreifer bereitgestellten Format-String (der Format-String selbst liegt ja auch auf



dem Keller). Zu dem Zeitpunkt, zu welchem der Kellerzeiger auf die anzuspringende Adresse zeigt, muss auch der interne Format-Parameter-Zeiger auf %s stehen. Das Ergebnis ist, dass die Format-Funktion unsere Adresse als Zeiger auf einen String interpretiert und ihn demzufolge bis zum ersten Null Byte ausgibt. Erhöht man geschickt die gewünschte Adresse im Format-String, kann nach und nach der gesamte Prozessspeicher ausgelesen werden. Diese Technik wird u.a. genutzt, um Ursachen für fehlgeschlagene Angriffe herauszufinden.

3.2.2 Überschreiben des Speichers

Hier unterscheidet man zwei Ansätze:

- Angriffe ähnlich wie Buffer-Overflows

Man benutzt (analog zu Buffer-Overflows) den Format-String, um über die Grenzen eines Puffers hinwegzuschreiben, um die Rücksprungadresse zu ändern. Der Format-String selbst liegt ja auch auf dem Keller. Dann springt man abermals wieder in den Format-String, wo der Shellcode bereitliegt.

- Angriffe duch pure Format-Strings

In manchen Situationen ist es nicht möglich den Format-String zu verlängern, um so über die Grenzen des Buffers zu kommen. Hier behilft man sich durch den %n Format-Parameter. Er erlaubt es uns, an die Adresse die durch den aktuellen Kellerzeiger angegeben wird, eine Integer Zahl, nämlich die Anzahl der bereits durch die Format-Funktion geschriebenen Bytes zu schreiben.

```
"aAA0_%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n"
```

Dabei verwenden wir wieder den %08x Format-Parameter. Der Format-String wird so gewählt, dass wenn das %n beim Lesen erreicht wird, der aktuelle Parameter-Zeiger der Format-Funktion auf den Begin unseres Format-Strings zeigt (denn der Format-String liegt auf dem Keller). Das %n bewirkt nun, dass an die Adresse 0x30414141 (aAA0 ist die String-Repräsentation), die Anzahl der bereits geschriebenen Bytes geschrieben wird. Wir können also an beliebige Adressen schreiben, aber nur die Anzahl der bereits geschriebenen Bytes. Das ist natürlich eine Einschränkung. Wie kann der %n Format-Parameter effektiv und

sinnvoll beeinflusst werden? Der Artikel von Scut01 [2] befasst sich ausführlich mit dieser Thematik.

3.2.3 Absturz des Programms

In bestimmten Situationen kann es Angreifern nützlich sein, laufende Dienste zum Absturz zu bringen. Beispielsweise kann somit ein *core dump* (Abbild des Prozessspeichers) erzeugt werden, der interessante Daten enthält (z.B Passwörter) oder aber ein laufender Dienst würde bestimmte Aktionen verhindern oder erschweren (z.B. bei DNS Spoofing[1]). Zu diesem Zweck genügt es schon, im Programmablauf einen illegalen Zeigerzugriff zu provozieren. Dazu übergibt man einfach eine Kette von %s Format-Parametern im Format-String:

```
"%s%s%s%s%s%s%s%s"
```

Die Format-Funktion versucht jetzt nacheinander acht Adressen vom Keller zu Strings zu referenzieren. Die Wahrscheinlichkeit ist natürlich gross, dass dabei eine ungültige Adresse auftaucht und das Programm mit *segmentation fault*(Speicherzugriffsfehler) abstürzt. Der Angreifer hat sein Ziel erreicht.

3.3 Spezialfall: Global Offset Tabelle (GOT)

Durch den großen Erfolg der Angriffe mittels Buffer-Overflows glauben viele Leute, dass es das Beste sei, immer die Rücksprungadresse auf dem Keller zu überschreiben, um Kontrolle über die Anwendung zu erlangen. Obwohl dies offensichtlich gut funktioniert, ist es eher eine aus dem Zwang heraus gewählte Möglichkeit. Bei herkömmlichen Buffer-Overflows hat man auch nur die Möglichkeit, die auf dem Keller gespeicherten Daten zu überschreiben. Mit der Verwundbarkeit des Format-Strings haben wir jedoch die Möglichkeit, beliebige Daten im beschreibbaren Speicher des Prozesses zu überschreiben. Beliebtes Ziel ist hier die Adresse einer Bibliotheksfunktion in der Global Offset Tabelle (GOT). Jede binäre ausführbare Anwendung enthält eine Global Offset Tabelle. In der GOT werden zu allen im Programm gebundenen Funktionen die Adressen, die die jeweilige Funktion referenziert, gespeichert. Beispiel:

```
exit(EXIT_FAILURE);
```

Ein zuverlässier Weg, um Kontrolle über den Prozess zu erhalten, ist es den GOT Eintrag, hier am Beispiel für exit(), zu überschreiben. Bei jedem Aufruf von exit() im Programm wird hierbei die von dem Angreifer bereitgestellte Adresse angesprungen. Die GOT Tabelle für Binärdateien kann man sich mittels Tools [2] ausgeben lassen. Ausserdem ist die GOT Adresse per Binärdatei fest, das heisst, sie verändert sich nicht wie eine Kelleradresse (wenn das Program zum Beispiel mit

[1] Umleitung von Datenpakete an eine durch den Angreifer besetzte Adresse

einer anderen Umgebung gestartet wird). Man muss nicht umständlich irgendwelche Abstände bestimmen und ereicht trotzdem einen zuverlässigen Angriff der Anwendung. Die Manipulation der GOT Tabelle ist eine bequeme und einfache Abkürzung.

3.4 Zusammenfassung und Gegenmaßnahmen

Damit eine Verwundbarkeit des Format-Strings vorliegt muss die Voraussetzung gelten, dass der Format-String durch den Beuntzer bereitgestellt werden kann. Desweiteren muss der potentielle Angreifer folgende Werte herausfinden:

- Abstand vom Format-String zur Rücksprungadresse (um die Rücksprungadresse zu manipulieren; nur wenn der Format-String aus seinen Grenzen ausbrechen kann)
- Die Adresse, die auf den Format-String zeigt (um ihn, z.B. für Shellcode anzuspringen)

Aus der Sicht des Entwicklers kann man sich sehr gut gegen die Verwundbarkeit des Format-Strings schützen.

- Bewusstsein, dass hier eine potentielle Gefahr liegt
- Durchsuchen des Quellcodes nach entsprechenden Schwachstellen durch Tools [2]

4 Ausblick

Ein ganz anderes Konzept, um gegen Buffer-Overflows vorzugehen, verfolgen gesicherte Betriebssysteme wie Argus Pitbull [4] oder das Security-Enhanced Linux der NSA [5]. Diese unternehmen gar nicht erst den Versuch, Buffer-Overflows zu verhindern, sondern versuchen deren Konsequenzen durch sehr detaillierte Rechtevergabe unter Kontrolle zu halten. So hat ein Angreifer selbst mit einer Root-Shell, die er durch einen Buffer-Overflow im Web-Server erlangt hat, nur sehr eingeschränkte Zugriffsmöglichkeiten auf das System. Dass sich auch solche Systeme kompromittieren lassen, zeigte im Frühjahr ein erfolgreicher Einbruch bei einem Hacking-Contest von Argus, bei dem die Angreifer ein Sicherheitsproblem des eingesetzten Solaris-Kernel ausnutzten - was ihnen 100 000 Mark Preisgeld einbrachte. Buffer-Overflows und Format-String Verwundbarkeiten werden sicher auch in Zukunft eines der zentralen Sicherheitsprobleme darstellen. Neben den kellerorientierten Angriffen treten immer häufiger auch vermehrt Angriffe auf, die sich Pufferüberlaufe in statischen Variablen auf dem Heap zu Nutze machen. Erste Veröffentlichungen dazu sind bereits im Internet erschienen [3]. Solange Entwickler und Programmierer unter Zeitdruck immer neue Funktionen und Features in Programme einbauen, wird sich an dieser Situation auch wohl nichts ändern.

Literatur

[1] Aleph One - Smashing The stack For Fun And Profit, *Phrack 49*, 1996, http://www.phrack.com/phrack/49/P49-14

[2] scut / team teso - Exploiting format string vulnerabilities, *17th CCC Congress / Berlin* , 2001, http://teso.scene.at/articles/formatstring

[3] Matt Conover & w00w00 Security Team - w00w00 on Heap Overflows, 1999, http://www.cs.ucsb.edu/ vigna/courses/CS595/heapoverflows/heaptut.txt

[4] Products Overview PitBull Foundation and Foundation Suite, http://www.argus-systems.com/product/overview/pitbull/

[5] Security-Enhanced Linux, http://www.nsa.gov/selinux/

[6] StackShield, http://www.angelfire.com/sk/stackshield/

[7] StackGuard, http://immunix.org/

[8] Solar Designers Linux-Patch, www.openwall.org

[9] SecureStack, http://www.securewave.com/products/securestack/