Defekte in Software: Buffer-Overflows, Format-String-Fehler und Race-Conditions

Marcel Noe

Zusammenfassung

Die meisten erfolgreichen Angriffe auf Computersysteme basieren auf Fehler in der verwendeten Software. Die häufigsten Fehler, die in Server-Software zu finden sind, lassen sich in die Klassen Stack- oder Heap-Overflow, Format-String Fehler oder Race-Condition einteilen. In dieser Seminararbeit wird ein Überblick über diese Arten von Fehlern gegeben, sowie gängige Abwehrmechanismen vorgestellt und eine Bewertung dieser vorgenommen.

1 Einleitung

Die Kommunikation über das Internet hat in den letzten Jahren immer mehr an Bedeutung gewonnen. Fast jeder Vorgang des realen Lebens, seien es nun Einkäufe, Finanztransaktionen oder Behördengänge, wie z.B. das Abgeben einer Steuererklärung, kann mittlerweile über das Internet vorgenommen werden. Hierdurch steigen aber auch die Auswirkungen von Fehlern in Computersoftware und das wirtschaftliche Interesse daran, diese auszunutzen. Wo es vor einigen Jahren vielleicht nur ärgerlich war, wenn man aufgrund eines Computerviruses oder gehackten Rechners eine Zeit lang auf gewisse Dienste wie z.B. E-Mail nicht zugreifen konnte, so ziehen Fehler in Computerprogrammen mittlerweile immer mehr auch finanzielle Verluste nach sich.

Um das Auffinden und Ausnutzen von Sicherheitslücken in Computerprogrammen hat sich mittlerweile eine ganze Industrie etabliert – Schwarzmärkte für so genannte *Exploits*, also kleine Scripte oder Programme, die dazu dienen, eine bestimmte Sicherheitslücke in einem verwundbaren Programm auszunutzen, sind längst Realität[1].

Umso wichtiger wird es, gängige Sicherheitslücken in Software zu kennen, zu wissen auf welche Arten sie ausgenutzt werden können, und sich über Abwehrmaßnahmen zu informieren.

Im Abschnitt 2 werden zuerst einige Grundbegriffe erläutert, die zum späteren Verständnis notwendig sind. Danach werden die vier häufigsten Arten von Sicherheitslücken vorgestellt und es wird kurz darauf eingegangen, wie diese üblicherweise ausgenutzt werden. Zum Schluss jedes Abschnitts werden noch die bekanntesten Abwehrmaßnahmen vorgestellt, und es wird eine kurze Bewertung über deren Wirksamkeit vorgenommen.

Besonderer Schwerpunkt dieser Arbeit liegt auf *Stack-Buffer-Overflows*, weil es sich hierbei um die bekanntesten und wahrscheinlich am besten erforschten Sicherheitslücken handelt.

Bei *Heap-Buffer-Overflows* und *Format-String-Fehlern* handelt es sich um vergleichsweise neue Arten von Sicherheits-Lücken, deren Ausnutzbarkeit erst seit einigen Jahren bekannt ist.

Auf *Race-Conditions*, also Fehler bei denen der Erfolg einer Operation von dem zeitlichen Verlauf mehrerer Einzel-Operationen abhängt, wird nur kurz eingegangen.

[1] Bei `https://wslabi.com` handelt es sich so z.B. um eine Art E-Bay für Zero-Day-Exploits.

In den meisten Teilen dieser Arbeit wird von der Verwendung einer GNU/Linux-Plattform auf gängigen PC-Systemen mit x86-Prozessor sowie der Programmiersprache C ausgegangen, an einigen Stellen werden jedoch auch Besonderheiten der Windows Plattform von Microsoft beschrieben.

2 Grundlagen zum Verständnis

2.1 Übersicht x86

Die *x86*-Architektur ist die bis heute am weitesten verbreitete Prozessorarchitektur der Welt. Bei dem ersten Vertreter dieser, dem *8086*-Mikroprozessor, handelte es sich um eine 16-Bit CPU mit CISC-Befehlssatz, die Ende der 1970er Jahre von Intel auf den Markt gebracht wurde.

Enorme Bedeutung bekam diese Architektur als der ab 1981 von IBM produzierte IBM-PC zu großem Erfolg gelangte. Heutzutage haben PCs mit x86 CPU bis auf wenige Ausnahmen alle anderen Architekturen in den Sparten Notebook-, Desktop- und Server-Computer vom Markt verdrängt. Anzumerken ist, dass der Erfolg der x86 nicht auf besonderer technischer Innovation beruhte, sondern vor allem durch den geringen Stückpreis der CPUs begründet war.

Die x86-Architektur verwendet durchgehend eine sogenannte *Little-Endian-Codierung*, auf die in Abschnitt 4.3 noch genauer eingegangen wird.

2.2 Wichtige Register

Im Gegensatz zu z.B. der MIPS-Architektur, die vor allem über `general purpose` Register verfügt, haben die meisten Register der x86-Architektur eine speziellen Zweck. Für die Zwecke dieser Ausarbeitung ist vor allem das Register %eip interessant. In diesem wird der so genannte *Instruction Pointer*, also ein Zeiger auf den nächsten auszuführenden Befehl, gespeichert. Von weiterem Interesse ist das Register %esp, das einen Zeiger auf die aktuelle Position des *Stacks* enthält.

2.3 Organisation des Speichers eines Prozesses

Linux verwendet – wie jedes moderne Betriebssystem – virtuellen Speicher. Jedem Prozess steht damit sein eigener, linear adressierbarer Adressraum zu (bei einem 32-Bit System ist dieser in der Regel 4 GB groß). Der Speicherraum eines Prozesses ist in 3 Bereiche aufgeteilt: *Textsegment* (manchmal auch *Codesegment* genannt), *Datensegment* und *Stack*.

Der Schematische Aufbau ist nochmal kurz in Tabelle 1 dargestellt.

Text-Segment
Daten-Segment (inkl. Heap)
... Freier Speicherbereich ...
Stack

Tabelle 1: Schematische Anordnung des Adressraums eines Prozesses

2.3.1 Textsegment

Das Textsegment enthält den ausführbaren Code eines Programms. Wird ein Programm mehrmals ausgeführt, so muss sein Textsegment nur einmal in den Speicher geladen werden. Das Textsegment ist meistens als *read-only* markiert, und jeder Versuch, es zu verändern, führt zu einer Schutzverletzung (*Segmentation Fault*).

2.3.2 Datensegment und Heap

Das Datensegment ist nochmal in mehrere Unterbereiche untergliedert. Der *Datenteil* enthält alle globalen Variablen, die nicht mit Null initialisiert wurden. Die mit Null initialisierten globalen Variablen werden dagegen im *BSS-Segment* innerhalb des Datensegments gespeichert. Am Ende des Datensegments befindet sich der *Heap*. Aus diesem werden alle Anfragen nach mehr Speicher bedient, die das Programm während seiner Laufzeit stellt. Hierfür steht dem Programmierer der Befehl `malloc()` zur Verfügung.

Die genaue Organisation des *Heaps* hängt von der verwendeten *libc* ab. Mehr zu diesem Thema findet sich in Abschnitt 3.2.

2.3.3 Stack

Der Sinn des *Stacks* ist es, das Betriebssystem beim Ausführen von *Function Calls* (siehe dazu auch Abschnitt 3.1) zu unterstützen. Hierzu wird mit dem *Stack* eine *Last-in-First-out* (*LIFO*) Datenstruktur zur Verfügung gestellt.

Der *Stack* ermöglicht es, verschachtelte Funktionsaufrufe mit nahezu beliebiger Tiefe einfach und effizient zu realisieren.

Bemerkenswert ist, dass der *Stack* am oberen Ende – also bei der höchsten dem Prozess zur Verfügung stehenden Adresse – beginnt und nach unten in Richtung des *Heaps* wächst. Zwischen *Heap* und *Stack* befindet sich der unbenutzte Speicher, und *Heap* und *Stack* wachsen aufeinander zu.

Da Feldvariablen (*Arrays*) von der niedrigsten Adresse zur höchsten Adresse geordnet werden, führt dies dazu, dass man bei der Benutzung von *Arrays* entgegen der Wachstumsrichtung des *Stacks* schreibt.

3 GNU/Linux & GCC

Bei *GNU/Linux* handelt es sich um ein weit verbreitetes, freies, *UNIX*-artiges Betriebssystem.

Große Teile dieses Systems sind in der Programmiersprache *C* geschrieben und werden standardmäßig mit dem ebenfalls freien Compiler-Paket *GCC* (*GNU Compiler Collection*) übersetzt.

Es gibt auch andere *Compiler* (z.B. den *ICC* von Intel), diese werden hier allerdings nicht betrachtet.

3.1 Funktionsaufrufe

Eine wichtige Anforderung an Funktionsaufrufe ist, dass man diese bis zu theoretisch beliebiger Tiefe verschachteln kann.

Natürlich ist man in der Realität durch Rahmenparameter, wie verfügbaren Hauptspeicher eingeschränkt, jedoch ist die Anzahl der möglichen Verschachtelungen immer noch sehr hoch (ein für diese Zwecke geschriebenes Testprogramm, in dem sich eine Testfunktion immer wieder sich selbst aufrief, brach erst bei einer Rekursionstiefe in der Größenordnung von 690.000 verschachtelten Aufrufen ab).

Bei dem hier betrachteten System sieht ein Funktionsaufruf folgendermaßen aus: Das aufrufende Programm legt die Parameter für die aufzurufende Funktion nacheinander auf den Stack, und benutzt dann die Assembler-Instruktion `Call`, um diese Funktion anzuspringen.

`Call` legt zuerst den aktuellen Befehlszeiger auf den Stack, damit das Programm beim Verlassen der Funktion an der Codestelle nach dem Funktionsaufruf die Arbeit wieder aufnehmen kann (daher nennt man den gespeicherten Befehlszeiger auch die Rücksprungadresse). Danach springt `Call` die aufgerufene Funktion an.

Beim Eintritt in die angesprungene Funktion legt diese zuerst den aktuellen Framepointer auf den Stack, und alloziert danach Speicherplatz für alle in ihr deklarierten lokalen Variablen (ebenfalls auf dem Stack), indem sie den *Stack-Pointer* um den für die Variablen benötigten Platz dekrementiert[2]. Da das *Allozieren* und *Deallozieren* des Speicherplatzes für diese Variablen automatisch beim Betreten und Verlassen der Funktion geschieht, nennt man diese *automatische Variablen.*

Das Aussehen des *Stacks* während eines Funktionsaufrufs wird in Tabelle 2 noch einmal visualisiert.

Rücksprungadresse
Framepointer
Parameter 1
Parameter 2
Automatische Variable 1
Automatische Variable 2

Tabelle 2: Belegung des *Stacks* während eines Funktionsaufrufs

Hierbei muss beachtet werden, dass bestimmte Architekturen *Speicherausrichtung* verlangen, d.h. dass der Speicher nur in Blöcken gewisser Größe adressiert werden kann, also evtl. mehr Platz reserviert werden muss, als für die Speicherung der Variablen eigentlich notwendig wäre.

Bei der hier betrachteten x86-Architektur ist Speicherausrichtung zwar prinzipiell nur bei der Verwendung von *SIMD* Instruktionen notwendig und ansonsten optional, wird allerdings aus Performancegründen trotzdem oft vorgenommen.

Ist die Vorbereitungsphase abgeschlossen, beginnt die aufgerufene Funktion ihre Arbeit und schließt diese irgendwann durch ein explizites oder implizites `return()` ab.

Hierzu werden die Assembler-Instruktionen *leave*, welche den von der Funktion allozierten Speicher aufräumt, und *ret*, die die auf dem *Stack* gespeicherte Rücksprungadresse anspringt, indem sie diese in *%eip* lädt, verwendet.

3.2 glibc

Die Programmiersprache C besitzt keine eingebauten Funktionen. Diese werden erst durch externe Bibliotheken zur Verfügung gestellt. Die wichtigste dieser Bibliotheken ist die so

[2]Dekrementieren deshalb, weil der *Stack* nach unten wächst.

genannte *Standard C Library*, in der sich unter anderem Funktionen für Ein- und Ausgabe, die Speicherverwaltung sowie ein Interface zu den *System-Calls* des verwendeten Betriebssystems befinden.

Es gibt viele verschiedene Implementierungen dieser Standard-Bibliothek, die auch kurz *libc* genannt wird – die bekannteste dürfte allerdings die *GNU-C-Bibliothek*, kurz *glibc* sein.

Darüber hinaus gibt es jedoch noch unzählige weitere Implementierungen, wie z.B. die der Betriebssysteme Free-, Net- und OpenBSD, die *dietlibc* sowie die *msvcrt.dll* von Microsoft. Aufgrund ihrer großen Verbreitung, wird hier jedoch nur die *glibc* betrachtet.

Sofern eine C-Bibliothek für die Plattform, für die ein Programm geschrieben wird, verfügbar ist, kann die Bibliothek beim Linken des Programms frei gewählt werden. Wurde das Programm dynamisch gelinkt, muss die entsprechende Library zum Zeitpunkt der Ausführung auf dem Zielsystem vorhanden sein.

Unabhängig davon, ob man ein Programm statisch oder dynamisch linkt, werden beim Starten des Programms alle aus der *libc* verwendeten Funktionen in den Hauptspeicher geladen, so dass sie verwendet werden können.

Konkret bedeutet dies, dass sich alle verwendeten *libc*-Funktionen zur Laufzeit des Programms im Adressraum des selbigen befinden.

Darüber hinaus ist die verwendete *libc* für die Verwaltung des *Heap*-Speichers verantwortlich. Der *Kernel* stellt lediglich Speicher-Seiten zur Verfügung, die genaue Organisation muss allerdings im *Userspace* vorgenommen werden.

4 Stack-Buffer-Overflows

4.1 Einleitung

Bei *Stack-Buffer-Overflows* handelt es sich um die bekanntesten Vertreter der Kategorie *Buffer-Overflow* und somit um die häufigsten Sicherheits-Lücken überhaupt. Hierbei handelt es sich um Fehler, bei denen die Kapazitäten von auf dem *Stack* liegenden Variablen überschritten werden.

4.2 Funktionsweise

Stack-Buffer-Overflows entstehen dadurch, dass mehr Daten in eine auf dem *Stack* liegende Variable eingelesen werden, als in dieser eigentlich gespeichert werden können.

Sehr häufig passiert so etwas bei der Verwendung von *Strings*. *Strings* sind in *C* als Array von *chars* realisiert. Damit Funktionen, die Strings verarbeiten, wissen, wann diese Strings zu Ende sind, wird an das Ende eines Strings eine *Binäre Null* angehängt. Problematisch hierbei ist allerdings, dass nirgendwo gespeichert ist, wie viel Platz für die Variable, in der ein String gespeichert wird, reserviert wurde. Werden zur String-Verarbeitung Funktionen wie `gets()` oder `strcpy()` eingesetzt, die die Länge der eingegebenen Strings nicht beschränken, so kann es passieren, dass ein Schreiben über das Ende eines Strings erfolgt.

Damit ein *Stack-Buffer-Overflow*, und somit eine für einen Angreifer ausnutzbare Lücke, entsteht, muss sich die Variable, in die geschrieben wird, auf dem *Stack* befinden. Dies ist allerdings für jede automatische Variable, die innerhalb einer Funktion verwendet wird, bereits erfüllt.

Um dies zu verdeutlichen, ein Beispiel:

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[80];

    // Buffer 2 lesen
    fgets(buffer2, 19, stdin);

    // Buffer 2 in Buffer 1 kopieren
    strcpy(buffer1, buffer2);
}
```

Hier werden zunächst zwei Variablen, *buffer1* und *buffer2*, deklariert. Da es sich um Variablen innerhalb einer Funktion handelt, landen diese auf dem Stack.

In der Variable *buffer1* können nun bis zu 5 Zeichen gespeichert werden (Zeile 2) – von denen allerdings, wegen der geforderten Terminierung mit einer Binären Null, nur 4 verwendet werden können. In der Variable *buffer2* können allerdings bis zu 80 Zeichen (Zeile 3) gespeichert werden – von denen auch wiederum nur 79 nutzbar sind.

In Zeile 6 werden nun Daten von *stdin* (was in der Regel die Tastatur ist) in die Variable *buffer2* eingelesen. Der zweite Parameter der Funktion `fgets()` stellt sicher, dass das Einlesen nach 19 Zeichen abgebrochen wird.

Es wurden nun bis zu 19 Byte in die Variable *buffer2* eingelesen. Zusätzlich hängt *fgets* nun noch eine *binäre Null* an *buffer2* an, um das Ende des eingegebenen Strings zu markieren. Die Gesamtlänge des Strings in *buffer2* kann nun also bis zu 20 Byte betragen.

Problematisch wird es nun in Zeile 9: Hier wird der Inhalt von *buffer2* in die Variable *buffer1* kopiert. Wie wir sehen, wird der Funktion `strcpy()` kein Parameter übergeben, der die Anzahl der kopierten Bytes beschränken würde.

Diese Funktion kopiert nun so lange Zeichen aus *buffer2* in *buffer1*, bis sie in *buffer2* auf eine *binäre Null* trifft, die das Ende der in *buffer2* gespeicherten Zeichenkette markiert.

buffer1 kann lediglich 4 Zeichen aufnehmen, in *buffer2* sind allerdings bis zu 19 Zeichen gespeichert. Dies kann zu einem Problem werden: Ein String in *C* ist nichts anderes, als ein *Array* von *chars*. Die dazugehörige Variable ist ein Pointer auf das erste Element dieses *Arrays*. Nun tut die Funktion `strcpy()` nichts anderes, als diesen Pointer für jedes zu schreibende Zeichen jeweils um ein Byte zu inkrementieren. Dabei hat sie gar keine Möglichkeit festzustellen, wenn der Pointer, den sie inkrementiert, gar nicht mehr in den für die Variable vorgesehen Speicherplatz sondern auf den Speicherplatz einer ganz anderen Variable zeigt. Für den Fall, dass die gelesene Variable länger ist, als die, in die geschrieben wird, wird `strcpy()` andere Variablen üeberschreiben.

4.3 Exploits

Wie wir gesehen haben, ist es möglich, in *Array-Variablen* mehr Daten einzulesen, als Platz für diese reserviert wurde.

Die Frage ist nun, wie sich dieses Verhalten dazu ausnutzen lässt, das betroffene Programm zur Ausführung von Schadcode zu bringen.

In Abschnitt 3.1 wurde gezeigt, dass die Adresse, an die ein Programm nach dem Ausführen einer Funktion zurückkehrt, auch auf dem Stack abgelegt wird.

Array-Variablen werden in aufsteigender Richtung geschrieben, wohingegen der *Stack* in absteigende Richtung wächst. Konkret bedeutet dies, dass ein Schreibvorgang über die Grenzen eines *Arrays* dazu führt, dass Variablen, die in der Vergangenheit auf den *Stack* gelegt wurden, überschrieben werden.

Da die Rücksprungadresse zeitlich vor dem Anlegen der automatischen Variablen auf dem *Stack* abgelegt wurde, ist es möglich, dass durch das Schreiben hinreichend vieler Zeichen in eine automatische Variable irgendwann auch die Rücksprungadresse der Funktion, in der wir uns momentan befinden, überschrieben wird.

Wie lässt sich dies nun zum Ausführen, beliebigen Codes ausnutzen? Hierzu gibt es mehrere Möglichkeiten:

- Die bekannteste ist, den Code, den man ausführen möchte, vor der Rücksprungadresse in die überlaufende Variable zu schreiben und danach die Rücksprungadresse auf den Anfang dieses Codes zu setzen. Da dieser Code meistens eine *Shell* startet, nennt man ihn auch *Shell-Code*.

  Da oft nicht bekannt ist, an welcher Adresse im Speicher man sich gerade befindet, ist dies nicht immer einfach. Hierbei kann es hilfreich sein, den Code am Anfang mit *NOP*-Befehlen aufzufüllen, so dass man den Code nicht genau treffen muss.

  Ein weiteres Problem ist, dass oft nur sehr wenig Platz zur Verfügung steht, so dass der Code sehr kompakt gestaltet sein muss.[3]

- Eine elegantere Möglichkeit zum Ausführen beliebigen Codes ist es, an Stelle des Codes Parameter für eine Funktion aus der *libc* (sinnvollerweise *execve* oder *system*) auf den *Stack* zu legen, und dann die Rücksprungadresse auf diese Funktion zu setzen. Dies nennt man auch *Return-to-libc-Angriff* (vgl. Abschnitt 4.4.1).

Insgesamt muss also einen spezieller *String* erzeugt werden, der dem verwundbaren Programm als Eingabe übegeben wird. Zunächst muss ein geeigneter Wert für die Rücksprungadresse gefunden werden.

Dieser Wert wird dann hexadezimal in den zu erzeugenden String geschrieben. Da es sich bei x86 um eine *Little-Endian-Architektur* handelt, muss die Reihenfolge der Bytes in der Rücksprungadresse umgedreht werden, so dass das Byte mit der geringsten Wertigkeit (*Least significant Byte*, *LSB*) am Anfang, und das mit der höchsten Wertigkeit (*Most significant Byte*, *MSB*) am Ende steht.[4]

Nun fügt man, je nach gewählten Angriff, entweder den Shell-Code auf der linken Seite, oder die Parameter für die *libc-Funktion* auf der rechten Seite der Rücksprungadresse in den *String* ein.

Als letzten Schritt füllt man diesen *String* so lange von links mit Füllzeichen auf, bis man die manipulierte Rücksprungadresse so weit verschoben hat, dass sie später genau den Wert der Rücksprungadresse auf dem Stack überschreibt.

Der gesamt Vorgang wird in Abbildung 1 noch einmal verdeutlicht.

Für einen Angriff mit *Shellcode-Injection* sollte der manipulierte String nun folgendes Format haben:

```
<Fuellzeichen><Shellcode><Neue Ruecksprungadresse>
```

[3] Mittlerweile gibt es Toolkits wie *Metasploit*, mit denen sich Exploits und Shellcodes im Baukastenverfahren herstellen lassen.

[4] Dies bereitete dem Autor beim erstellen eines Demo-Exploits einiges Kopfzerbrechen.

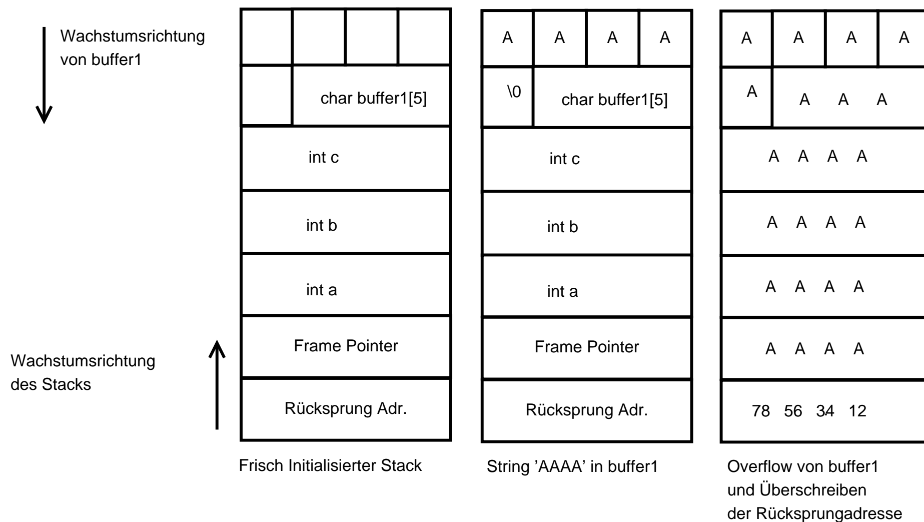

Abbildung 1: Schematische Darstellung eines Stack-Buffer-Overflows

Bzw. folgendes im Falle eines *Return-to-libc-Angriffs*:

```
<Fuellzeichen><Neue Ruecksprungadresse><Paramter fuer libc-Funktion>
```

4.4 Gegenmaßnahmen

4.4.1 Data-Execution-Prevention

Data Execution Prevention, von manchen Herstellern auch *Write or Execute*, *Exec Shield* oder *Executable space protection* genannt, ist eine Methode, Speicherbereiche als nicht ausführbar zu markieren. Dies kann entweder *Hardware-Unterstützt* mittels des *NX-Bits* von AMD oder des *XD-Bits* von Intel oder aber mittels Software-Emulation erreicht werden. Die *Hardware-Unterstützte-Variante* ist allerdings nur auf x86-Prozessoren mit 64-Bit Erweiterungen verfügbar.

Wird so z.B. der gesamte *Stack* als nicht ausführbar markiert, führt die oben beschriebene Attacke, bei der *Shell-Code* auf den *Stack* gelegt wird, und die Rücksprungadresse auf diesen gesetzt wird, zu einem Fehler und das Programm terminiert anormal.

Dies bringt allerdings auch Probleme mit bestimmter Anwendungs-Software mit sich, falls diese Teile ihres Codes wie z.B. Plugins auf dem *Stack* ablegt. Als so z.B. mit *Service Pack 2 Data Execution Prevention* in *Windows XP* eingeführt wurde, funktionierte einige Software wie *Microsoft Office*, *Adobe Photo-Shop* oder der *DivX-Codec* nicht mehr, so dass diese Software von dieser Schutzmaßnahme ausgeschlossen werden musste – was natürlich dazu führte, dass die betroffene Software auch nicht mehr geschützt wurde.

Lange Zeit wurde angenommen, dass mit *Data Execution Prevention* u.ä. Methoden *Stack-Buffer-Overflows* nicht mehr auftreten können. Dies lag daran, dass das Ablegen von *Shell-Code* auf dem *Stack* die verbreitetste Möglichkeit war, *Stack-Buffer-Overflow-Lücken* auszunutzen.

Mithilfe der in Abschnitt 4.3 beschriebenen *Return-to-libc* Attacke ist es jedoch möglich, *Data Execution Prevention* komplett auszuhebeln: Da direkt in einen Speicherbereich gesprungen

wird, der ausführbar sein muss, ist ein ein Ablegen von ausführbarem Code gar nicht mehr erforderlich.

Ohne weitere Schutzmaßnahmen bietet Data Execution Prevention also keinen ausreichenden Schutz vor *Stack-Buffer-Overflow-Lücken*.

4.4.2 ASLR

Address Space Layout Randomization oder kurz *ASLR* ist eine Technik, die die Position von Elementen wie z.B. *Heap*, *Stack* und die Adresse von Funktionen im Speicher zufällig verändert. Hierdurch soll das Ausnutzen eines *Buffer-Overflows* erschwert bis unmöglich gemacht werden, da ein Angreifer idealerweise nicht mehr vorhersagen kann, wo sich was im Speicher befindet und seine manipulierten Sprünge daher ins Leere laufen.

Die Position der Elemente im Speicher wird entweder beim Starten des Programms oder zur *Compile-Zeit* festgelegt. Hierdurch soll insbesondere die *Return-to-libc* Attacke erschwert werden, da es dem Angreifer nicht möglich sein soll, die Position der *libc*-Funktionen im Speicher vorherzusagen. Daher ist diese Technik besonders sinnvoll in Kombination mit der oben genannten *Data-Execution-Prevention*.

Implementiert wird diese Technik unter anderem von dem *PaX*-Patch für den *Linux* Kernel, *OpenBSD*, *Windows Vista* sowie *Mac-OS X 10.5* („Leopard“)[5].

Wie Shachman et. al. in [SGPM+04] zeigen, lässt sich mit dieser Technik auf 32-Bit Rechnern lediglich eine Entropie von 16-20 Bit erreichen, was in der Praxis einen Angriff zwar verlangsamen, aber nicht effektiv verhindern kann. Die Autoren schlussfolgern daher, dass diese Technik nur auf 64-Bit Rechnern einen sinnvollen Schutz darstellt. Weist die fehlerhafte Software über den *Buffer-Overflow* hinaus eine weitere Lücke auf, durch die es möglich wird, die Adresse von Funktionen im Speicher zu bestimmen (In Frage käme hierfür z.B. der in Abschnitt 6.1 beschriebener *Format-String-Fehler*), so ist es möglich, den Schutz von *ASLR* ohne großen Aufwand zu umgehen.

4.4.3 Stack-Smashing-Protection

Traditionelle Angriffe auf *Stack-Buffer-Overflow-Lücken* basieren alle auf dem Überschreiben der Rücksprungadresse auf dem Stack. Als *Stack-Smashing-Protection* werden eine Reihe von Maßnahmen bezeichnet, die zur *Compile-Zeit* eines Programms getroffen werden können, um dies zu verhindern.

Hier wird zwischen zwei Möglichkeiten unterschieden, dies zu realisieren:

- Speichern der Rücksprungadresse außerhalb des *Stacks*:

  Spezielle Programme wie z.B. *Stack-Shield* verändern den Programmcode dahingehend, dass die Rücksprungadresse außerhalb des *Stacks* in einer weiteren, stack-artigen Datenstruktur abgelegt wird. Je nach Einstellung wird nun entweder immer die Rücksprungadresse aus dieser Kontrolldatenstruktur genommen oder es wird beim Rücksprung der Wert auf dem *Stack* wird mit dem der Kontrolldatenstruktur verglichen und bei Nicht-Übereinstimmung wird die Ausführung des Programms abgebrochen.

- Schützen der Rücksprungadresse mittels eines Kontrollwerts („*Canary*“):

[5] Wobei die *Mac-OS* Implementierung bereits im Vorfeld als nicht effektiv gilt.

Programme wie *StackGuard*, der *Microsoft Visual Studio C/C++ Compiler* sowie *GCC* sind dazu in der Lage, zwischen der Rücksprungadresse und der auf dem *Stack* gespeicherten Variablen einen Kontrollwert (ein so genanntes „*Canary*") abzulegen, und zum Zeitpunkt des Rücksprungs zu überprüfen, ob dieser Kontrollwert immer noch unverändert auf dem *Stack* liegt. Die Idee dahinter ist, dass der Angreifer beim Überschreiben der Rücksprungadresse immer auch den Kontrollwert überschreibt, wodurch diese Manipulation beim Überprüfen des selbigen sofort auffällt und die Ausführung des Programms abgebrochen wird. Dies wird in Tabelle 3 noch einmal verdeutlicht.

Rücksprungadresse
Kontroll-Wert (Canary)
Framepointer
Parameter 1
Parameter 2
Automatische Variable 1
Automatische Variable 2

Tabelle 3: Ausschnitt eines mit Canary-Werten geschützten Stacks

[Rich02] beschreibt diverse Angriffe gegen *Stack-Smashing-Protection*. Fazit dieses Artikels ist, dass Maßnahmen, die lediglich die Rücksprungadresse, nicht aber den auf dem *Stack* gespeicherten *Frame-Pointer* schützen, einen erfolgreichen Angriff zwar erschweren, nicht jedoch unterbinden. Die Vermutung liegt daher nahe, dass auch *Stack-Smashing-Protection* keinen ausreichenden Schutz gegen *Stack-Buffer-Overflow*-Angriffe bietet.

4.4.4 Verwendung sicherer Funktionen

Keine der oben genannten Techniken bietet einen perfekten Schutz gegen *Stack-Buffer-Overflow-Angriffe*. Dies zeigt deutlich, dass die Verantwortung für die Sicherheit eines Programms letztendlich beim Programmierer liegt, der dieses erstellt hat.

Sichere Alternativen für Funktionen wie `strcpy()` oder `gets()` sind verfügbar und die Gefährlichkeit von Funktionen, die die Eingabe Länge von Zeichenketten nicht überprüfen ist seit vielen Jahren bekannt.

Aus sicherheitstechnischer Sicht wäre es angebracht, solche unsicheren Funktionen komplett aus der *libc* zu verbannen, und die Programmierer so zur Verwendung sicherer Funktionen zu zwingen.

5 Heap-Buffer-Overflows

5.1 Einleitung

Programme haben neben der Verwendung von statisch definierten Variablen die Möglichkeit, zur Laufzeit mittels `malloc()` und ähnlichen Funktionen, dynamisch Speicher vom Betriebssystem anzufordern. Solche Anfragen werden dann aus dem *Heap* bedient. In Abschnitt 2.3.2 wurde beschrieben, dass es sich bei dem *Heap* um eine zur Laufzeit des Programms wachsende Datenstruktur am Ende des Datensegments handelt.

Um die Verwaltung des *Heap*-Speichers zu vereinfachen und eine übermäßige Fragmentierung zu verhindern, wird der *Heap*-Speicher in Form von Blöcken vergeben. Hierzu befindet sich

am Anfang eines Blockes ein Header, in dem zwei Pointer, einen auf den vorherigen, einen auf den nächsten Block, die Größe des Blocks sowie die Angabe, ob der Block benutzt oder unbenutzt ist, gespeichert wird. Dies ist in Abbildung 2 noch einmal illustriert.

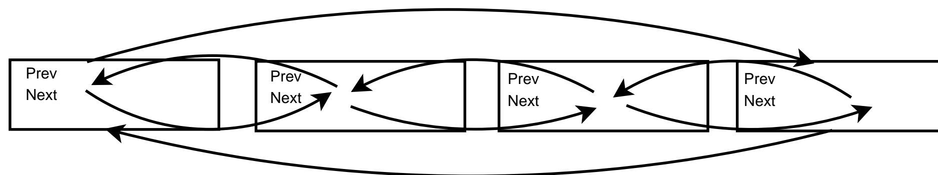

Abbildung 2: Alloziierter Speicher als doppelt verlinkte Liste.

Zwei gleich große Blöcke können mit einer *merge-Operation* zu einem größeren Block verschmolzen werden. Üblicherweise wird dies bei der Freigabe von Speicher von der Funktion `free()` durchgeführt.

Wird die Funktion `free()` auf einen allozierten Block ausgeführt, so reicht diese den Block an das `unlink()-Makro` weiter, welches den Block aus der Liste der zugewiesenen Speicher-Blöcke entfernt. Die Funktionsweise könnte in etwa folgendermaßen aussehen:

```
1    #define unlink(block){
2         // Temporaere Variablen
3         (Header *) prevblock;
4         (Header *) nextblock;
5
6         // Aus vorherigem Block aushaengen
7         prevblock         = block->prev;
8         prevblock->next = block->next;
9
10        // Aus nachfolgendem Block aushaengen
11        nextblock         = block->next;
12        nextblock->prev = prevblock;
13
14        // Pointer in ausgehaengtem Block auf Null setzen
15        block->prev       = NULL;
16        block->next       = NULL;
17   }
```

Außerdem kann ein großer Block mit einer *split* Operation in zwei kleinere Blöcke zerlegt werden. Dies ist typischerweise notwendig, wenn `malloc()` keinen Block findet, der klein genug ist, um die Anfrage zu erfüllen. Zwar wäre es ohne weiteres möglich, dem Programm einen größeren Block zurückzugeben, als es eigentlich angefordert hat, dies wäre aber natürlich eine Verschwendung kostbaren Arbeitsspeichers.

5.2 Funktionsweise

Vergebene Blöcke werden innerhalb des *Heaps* als doppelt verkettete Liste gespeichert. Zu Beginn eines jeden Blocks befindet sich ein *Header* mit Kontrollinformationen über diesen Block. Wenn es nun einem Angreifer gelingt, mehr Daten in einen Block zu schreiben, als dieser eigentlich aufnehmen kann, so kann er über die Grenzen dieses Blocks in den angrenzenden Block schreiben. Da sich hier der Header des angrenzenden Blocks befindet, wird es dem Angreifer so möglich, diesen *Header* zu überschreiben.

Die Ursachen dafür, dass mehr Daten in einen Block geschrieben werden als darin Platz vorhanden ist, sind meist die selben wie bei *Stack-Buffer-Overflows*, nämlich eine falsche Kontrolle der Länge von Zeichenketten.

5.3 Exploits

Wie kann nun aber das Umsetzen der *Header*-Felder dazu verwendet werden, schadhaften Code auf dem angegriffenen System auszuführen?

Gelingt es einem Angreifer, den *Header* eines zugewiesenen Blocks erfolgreich zu überschreiben und ihn derart zu manipulieren, dass der *next-Zeiger* anstatt auf den nächsten Block, auf eine Stelle im Speicher zeigt, in der vom Angreifer eingeschleuster Code liegt, dann ist er evtl. in der Lage, Schadcode auf dem angegriffenen System auszuführen.

Hierzu muss er den *prev-Zeiger* so verändern, dass er nicht auf einen vorherigen Block im Speicher, sondern auf die Rücksprungadresse der aktuellen Funktion im *Stack* zeigt. Wird nun auf den so modifizierten Block die Funktion `free()`, und damit auch das `unlink()`-Makro angewendet, so versucht dieses, den modifizierten Block aus der verketteten Liste des zugewiesenen Speicherbereichs auszuhängen.

Dazu folgt das Makro dem *prev*-Pointer des modifizierten Blocks – der im obigen Beispiel ja auf die Rücksprungadresse im *Stack* zeigt – und ersetzt diesen durch den *next*-Pointer, also durch den Pointer auf den eingeschleusten *Shell-Code*.

5.4 Gegenmaßnahmen

5.4.1 Canary-based-Heap-Protection

Analog zu dem in Abschnitt 4.4.3 beschriebenen Verfahren, ist es möglich, zwischen die einzelnen Blöcke ein *Canary*, also einen Kontrollwert zu legen. Wird ein Block überschrieben, so wird auch der Kontrollwert überschrieben. Wird nun der so manipulierte Block wieder freigegeben, überprüft das *Memory-Management* das *Canary*, was natürlich fehlschlägt und einen Abbruch des Programms veranlasst.

Dieser Art von Schutz wird beispielsweise von *Microsoft Windows XP* ab *Service Pack 2* implementiert.

In [Anis04] wird jedoch eine Möglichkeit beschrieben, diesen Schutz in *Windows* auszuhebeln. Der Angriff basiert darauf, dass eine Überprüfung nicht stattfindet, wenn sich der betroffene Bereich des *Heaps* in einer sogenannten *Lookaside-List*, also einer Art in Software implementiertem *Cache* befindet. Da es sich hier jedoch um einen *Implementierungsfehler* handelt, ist davon auszugehen, dies in absehbarer Zeit von *Microsoft* behoben wird.

5.4.2 Konsistenzprüfung des Heaps

In [Lind06] wird vorgeschlagen, bei jedem Aufruf von `unlink()` zu überprüfen, ob die *next*- und *prev*-Pointer überhaupt auf den *Heap* zeigen. Hierzu ist es evtl. notwendig, einmal rekursiv allen Pointern zu folgen. Dies würde zwar einen gewissen Overhead erzeugen, dafür würde allerdings auch das Überschreiben von Daten außerhalb des *Heaps* unterbunden werden. Implementierungen dieser Technik waren zum Zeitpunkt dieser Ausarbeitungen jedoch noch nicht bekannt.

6 Format-String-Fehler

6.1 Einleitung

Angriffe auf *Format-String-Fehler* sind relativ neu und kamen erst vor einigen Jahren auf. Obwohl schon seit vielen Jahren bekannt war, dass *Format-String-Fehler* zu Programmabstürzen führen können, dachte man bis vor kurzem, dass es nicht möglich ist, diese Fehler dazu auszunutzen, um Code auf ein Zielsystem einzuschleusen. Dass es sich hierbei um einen Irrtum handelte, demonstrierte ein Hacker mit dem Nickname *tf8* auf sehr dramatische Weise mit einem *Remote-Root-Exploit* für *WU-FTP*. Seitdem haben sich sehr viele unterschiedliche und vielseitige Angriffe auf diese Art von Lücken entwickelt.

6.2 Funktionsweise

Format-Strings dienen dazu, den Ablauf bestimmter Klassen von Funktionen wie z.B. `printf()` zu steuern. Prinzipiell handelt es sich bei einem Format-String um eine Anweisung, wie ein String zusammenzubauen ist. Hierzu enthalten Format-String spezielle Marken, die durch weitere, der Funktion übergebene Parameter ersetzt werden. Hier ein Beispiel:

```
1   int alter          = 36;
2   char vorname[]     = "Peter";
3   char nachname[]    = "Mayer";
4   printf("Name: %s %s, Alter: %i\n", vorname, nachname, alter);
```

In diesem Beispiel wird das erste %s durch die *String-Variable Vorname*, das zweite %s durch *Nachname* und das %i durch die *Integer-Variable alter* ersetzt. Die Ausgabe würde also konkret „`Name: Peter Mayer, Alter: 36`“ lauten.

Wie werden nun aber *Format-Strings* verarbeitet? Argumente für Funktionen werden über den *Stack* übergeben, d.h. als erstes Argument bekommt die Funktion – in unserem Beispiel also `printf()` – den *Format-String* übergeben. Diesen parst sie nun und liest für jedes gefundene Format-Zeichen die zu dem Format-Zeichen entsprechende Anzahl von Bytes[6] vom Stack, wo eigentlich die zu den Format-Zeichen passenden Parameter liegen sollten.

Problematisch wird dies immer dann, wenn es möglich wird, *Format-Strings* von außen einzuschleusen. Betrachten wir dazu folgendes fehlerhafte Programm:

```
1   #include <stdio.h>
2
3   int main(int argc, char * argv[]){
4       if(argc)
5           printf(argv[1]);
6   }
```

Wenn diesem Programm Parameter übergeben werden, gibt es diese auf *stdout* aus. Diese Art der Verwendung von `printf()` wurde lange Zeit als einfache Methode, *String-Variablen* auszugeben, benutzt.

Werden im ersten Argument beim Aufruf *Format-Zeichen* übergeben, so werden diese von `printf()` interpretiert. Da `printf()` aber keine weiteren Parameter übergeben wurden, wird das angezeigt, was sich als nächstes auf dem *Stack* befindet. Mit einer geschickten Abfolge von Format-Zeichen ist es so z.B. möglich, den Inhalt des *Stacks* zu lesen:

[6] Für %i würden so z.B. 4 Bytes gelesen werden. Für %s wird so lange gelesen, bis das erste mal eine binäre Null auftritt, die einen String terminiert.

```
‘--> ./a.out "%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x"
bfe069b4.bfe06938.080483f9.bfe06940.bfe06940.bfe06988.b7e94ea8.00000000%
```

Bei einem Vergleich dieser Ausgabe mit der Ausgabe des Programms *objdump* ergibt sich, dass es sich bei dem dritten Parameter, hier mit dem Wert 0x080483f9, um die Rücksprungadresse handelt.

6.3 Exploits

Mit dem im vorherigen Kapitel gezeigten Weg, sich den Inhalt des *Stacks* anzeigen zu lassen, lässt sich bereits sehr viel Schaden anrichten. So ist es z.B. möglich, *Canaries* vom *Stack* auszulesen und so die in Abschnitt 4.4.3 beschriebene *Stack-Smashing-Protection* auszuhebeln. Auch das in Abschnitt 4.4.2 beschriebene *ASLR* verliert so seine Wirkung, da nun bekannt ist, um welchen Offset der *Stack* verschoben wurde, woraus sich sehr leicht die Position anderer Funktionen im Speicher berechnen lässt.

Dies ist zwar alles bereits sehr hilfreich, die entscheidende Frage ist jedoch, ob es möglich ist, mithilfe von geschickt erzeugten *Format-Strings* Inhalte im Speicher zu überschreiben. Hierzu wäre ein Formatzeichen notwendig, das z.B. `printf()` dazu veranlasst, etwas in den Speicher zu schreiben. Ein Zeichen welches genau dies tut ist %n, das dazu dient, die Anzahl der bisher geschriebenen Zeichen in einen als Parameter übergebenen Zeiger auf einen Integer zu schreiben. Mit einer geschickten Kombination aus Formatzeichen und dem mehrfachen Verwenden von %u, %x und %n ist es so möglich, beliebiger Werte jeweils Byte für Byte in den Speicher zu schreiben. Eine genaue Anleitung hierzu findet sich unter [Scut01].

6.4 Gegenmaßnahmen

Mithilfe einiger Modifikationen der im vorherigen Abschnitt beschriebenen Attacke ist es möglich, praktisch beliebige Stellen im Speicher zu schreiben. *Format-String-Angriffe* werden somit zu einer vielseitigen Bedrohung, da gängige Abwehrmechanismen wie z.B. *Stack-Smashing-Protection* komplett umgangen werden. Glücklicherweise sind *Format-String-Fehler* relativ einfach aufzuspüren, hierfür gibt es eine Reihe von Tools wie z.B. *TESOgcc* oder *pscan*.

7 Race-Conditions

7.1 Einleitung

Race-Conditions sind Fehler, die dadurch entstehen, dass gewisse Operationen, die voneinander abhängen, nicht atomar durchgeführt werden. Hängt z.B. die Ausführung einer gegebenen *Funktion B* von einer Überprüfung eines Sachverhaltes von einer *Funktion A* ab, und verändert sich dieser Sachverhalt nach dem Ausführen von *Funktion A* und vor dem Ausführen von *Funktion B*, so spricht man von einer *Race-Condition*.

7.2 Funktionsweise

Race-Conditions treten oft auf, wenn Dateirechte überprüft werden sollen. Stellen wir uns z.B. ein Programm vor, das mit *Super-User* Rechten läuft und Usern ermöglichen soll, gewisse Operationen auf ihren eigenen Dateien auszuführen, die ihnen als normalen Usern nicht zur Verfügung stehen (ein typisches Beispiel hierfür ist z.B. der *Drucker-Spooler*). Hierzu

überprüft das Programm zuerst, ob die Datei wirklich dem entsprechenden User gehört und führt danach eine bestimmte Operation (also z.B. Ausdrucken) darauf aus:

```
1   int main(int argc, char * argv[]){
2       if(access(argv[1], R_OK) != 0)
3           // Kein Zugriff
4           exit(1);
5
6       // Zugriff OK
7       do_something(argv[1]);
8   }
```

Verändert sich dieser Sachverhalt nun während des Ausführens von `access()` und `do_something()`, so führt das Programm trotzdem die Funktion `do_something()` aus, obwohl der User eigentlich gar keinen Zugriff mehr auf die Datei haben sollte.

7.3 Exploits

Ein typischer Angriff auf *Race-Conditions* erfolgt durch geschicktes Setzen von *Links*: Zuerst wird eine Datei erzeugt, auf die der User Zugriff hat und diese wird dem von der *Race-Condition* betroffenen Programm als Parameter übergeben.

Nachdem das Programm seine Sicherheitsüberprüfungen abgeschlossen hat, wird die Datei entfernt und durch einen Link auf eine Datei, auf die man eigentlich zugreifen möchte (z.B. die Passwort Datei des Systems) ersetzt. Das anfällige Programm öffnet somit irrtümlich die Datei, auf die der Link gesetzt wurde.

Dies setzt natürlich ein sehr genaues Timing der Angriffe voraus und erfordert evtl., dass die Ausführung des anfälligen Programms verlangsamt wird. Durch die Verwendung von *nice*, sowie dem Beobachten der letzten im Dateisystem vermerkten Zugriffszeit der Fake-Datei, lässt sich dies erreichen.

7.4 Gegenmaßnahmen

In [BJSW05] werden zwei verschiedene Methoden vorgeschlagen, wie *Race-Conditions* zu vermeiden sind. Die eine Möglichkeit sieht vor, den *System-Call* `open()` derart zu verändern, dass man ihm über einen weiteren Parameter die UID des Users übergibt, mit dessen Rechten eine Datei geöffnet werden soll. Hat dieser User die erforderlichen Rechte nicht, schlägt die Operation fehl.

Eine weitere Möglichkeit besteht darin, einen weiteren Prozess mit den eingeschränkten Rechten des ausführenden Users zu forken, diesen die Datei öffnen zu lassen und den *File-Deskriptor* auf die Datei z.B. über eine Pipe an den Vater-Prozess mit *Super-User-Rechten* durch zu reichen.

Literatur

[Anis04] Alexander Anisimov. Defeating Microsoft Windows XP SP2 Heap protection, 2004.

[BJSW05] Nikita Borisov, Rob Johnson, Naveen Sastry und David Wagner. Fixing Races for Fun and Profit. In *USENIX*, 2005.

[c0nt] c0ntex. Bypassing non-executable stack during exploitation using return-to-libc.

[Cono99] Matt Conover. w00w00 on Heap Overflows, 1999.

[dRaa05] Theo de Raadt. Exploit Mitigation Techniques. In *OpenCON*, 2005.

[HaFi01] Dr. Richard Hall und Torsten Fink. Linux memory management, 2001.

[HKVW07] Hans-Joachim Hof, Stephan Krause, Lars Völker und Uwe Walter (Hrsg.). Hacking und Hackerabwehr. Seminarband WS06/07. Technischer Bericht TM-2007-1, Institut für Telematik, Universität Karlsruhe (TH), März 2007.

[I.E.99] I.E.C.C. Dynamic Linking and Loading, Juni 1999.

[Kaem01] Michael 'MaXX' Kaempf. Vudo - An object superstitiously believed to embody magical powers. *Phrack Magazine*, 2001.

[Lind06] Felix „FX“ Lindner. A heap of risk. *Heise Security*, Juni 2006.

[Löh05] Klaus-Peter Löhr. Sicherer Programmieren, 2005.

[One96] Aleph One. Smashing The Stack for Fun and Profit. *Phrack Magazine* 49(49), August 1996.

[Rich02] Gerado Richarte. Four different Tricks to bypass StackShield and StackGuard Protection, April 2002.

[Scut01] Scut. Exploiting Format String Vulnerabilities, März 2001.

[SGPM+04] Hovav Shacham, Eu-Jin Goh, Matthew Page, Nagendra Modadugu, Ben Pfaff und Dan Boneh. On the Effectiveness of Address-Space Randomization, 2004.