The Unlimited

31. Januar 2018

Kapitel 1

Speicher

1.1 Virtuelle Speicherverwaltung (32-Bit)

In der virtuellen Speicherverwaltung benutzt man für Prozesse das Flat Memory Model, auch bekannt als Protected Mode. Die Zusammenarbeit zwischen dem Prozessor und dem Betriebssystemkern ermöglichen es, dass jedem Prozess einen Adressraum von 4 GB (32-Bit) zur Verfügung steht. Die virtuelle Speicherverwaltung arbeitet nicht mehr kombiniert mit einer Segment- und einer Offsetadresse, wie es unter DOS üblich war, sondern man benutzt eine einzige 32-Bit große (Offset-)Adresse für den kompletten Adressraum.

Der Kernel setzt beim Start einer Anwendung einen 4 GB umfassenden Adressraum, auch wenn gar keine 4 GB RAM zur Verfügung stehen. Dabei belegt eine Anwendung nur so viel Speicher, wie der Programmcode und die zugehörigen Daten benötigen. Ist noch zusätzlichen Speicher nötig, so kann dieser, dank der virtuellen Speicherverwaltung, nachträglich angefordert werden. Verwaltet wird die virtuelle Speicherverwaltung durch den Kernel und basiert auf einem grundlegenden Mechanismus der Speicheradressierung des Prozessors.

So ist oft nicht der gesamte virtuelle Speicher einer Anwendung im Arbeitsspeicher verfügbar, da sich eine Anwendung in der Regel nur in Teilen des Codes oder der Daten bewegt. Dieser Umstand macht es Möglich, das man Teile des Arbeitsspeichers auf einer Festplatte auslagern kann. Zugriffe auf ausgelagerten Speicher werden vom Prozessor abgefangen und ausgelagerte Teile werden von ihm erst wieder in den Arbeitsspeicher geladen. Die Anwendung selbst bekommt von diesem Vorgang nichts mit. Damit das Ein-

und Auslagern möglichst effizient ist, teilt der Prozessor den virtuellen Speicher in 4 KB große Abschnitte, sogenannte Pages, auf.

Wann immer Maschinencode auf einen Adressraum zugreifen will, berechnet der Prozessor zuerst einmal aus der Adresse die Nummer der Page. Er bedient sich dazu der Page-Tabellen auch bekannt als Page-Directorys. Über sie wird der Adressraum Stück für Stück auf Pages im physischen Speicher verteilt. Der Prozessor gibt zwar das Format dieser Tabellen vor, aber verwaltet werden sie vom Betriebssystem. Der Eintrag der Page-Tabelle informiert den Prozessor auch darüber, ob sich eine Page im physischen Speicher befindet oder ob diese eventuell auf eine Festplatte ausgelagert wurde. Ein Eintrag in einer Page-Tabelle enthält für diesen Zweck verschiedene Flags, die unter anderem definieren, auf welche Art auf die Pages zugegriffen werden darf (Read-Only, ...). Sollte Speicher ausgelagert sein, so wird dies dem Kernel gemeldet, welcher sich dann um das Nachladen der Pages kümmert. Sollte es passieren das der komplette physische Speicher in Benutzung ist, so müssen erst andere Pages ausgelagert werden.

Was ein Programm als durchgängigen Adressraum sieht, verteilt sich so in Wirklichkeit über viele Pages, welche an ganz unterschiedlichen Stellen im physischen Speicher existieren. Der Speicher einer Anwendung kann bis zu Unkenntlichkeit fragmentiert im physischen Speicher vorliegen, doch es wird ihr verborgen bleiben, weil Sie keinen Speicherzugriff ausführen kann, ohne das der Prozessor dies bemerken würde. Da manche ausgelagerte Daten eventuell zeitnah wieder benötigt werden, versucht das Betriebssystem dies zu erahnen und lädt manche Daten schon wieder in den Speicher, bevor versucht wurde auf diese zuzugreifen. Zu beachten ist auch, dass nicht der komplette Adressraum von 4 GB einer Anwendung zur Verfügung steht, sondern Pages von z. B. Bibliotheken können in mehreren Anwendungen verwendet werden. Somit ist auch nicht zu vergessen, dass eine Instanz einer Bibliothek pro Prozess auch einen Datenbereich benötigt.

Abschließend sei erwähnt das Adressen größer 3 GB dem Kernel vorbehalten sind. Wenn die CPU in diesen Adressraum wechselt, dann wechselt sie auch oft von einem in den anderen Ring. In den verschieden Ringen stehen der CPU unterschiedlich ausgeprägte Befehlssätze zur Verfügung. So kann z. B. ein User-Prozess in Ring 3 nicht auf Hardware zugreifen, der Kernel in Ring 0 aber sehr wohl. Die CPU ermittelt den Ring, in welchem sie sich gerade befindet, über die Supervisor-Bits im Status-Register.

1.2 Paging

Paging ist ein integraler Bestandteil des Protected Mode und seit dem 80386 verfügbar. Für den Einsatz des Paging-Mechanismus ist das PG-Bit im Control-Register 0 (CR0) des Prozessors verantwortlich. Dieses Bit steht auf 0 nach dem das System im Real-Mode startet, im Real-Mode werden lineare Speicheradressen direkt auf physikalische Adressen abgebildet. Paging findet hier noch nicht statt. Schalte das Betriebssystem den Prozessor in den Protected-Mode, so setzt er dieses Bit auf 1. Jede Adresse, die der Prozessor bezüglich eines Maschinenbefehls verarbeitet, wird jetzt einer Page zugeordnet.

Die Größe einer Page beträgt 4 kB (4096 Byte) und jede Page beginnt an einer durch 4 kB teilbaren physischen Adresse. Der Adressraum wird dadurch in 1048576 (20-Bit) verschiedene Pages aufgeteilt, die jeweils 4 kB (12-Bit) umfassen. Die Ausrichtung auf 4 kB ermöglicht es, dass die untere 12 Bitgrenze dafür sorgt, dass lineare Adressen und physikalische Adressen identisch sind, sie stellen eine Art Offset einer Page da. Die oberen 20-Bit der linearen Adresse werden für die Nummerierung der Pages verwendet. Diese 20-Bit werden isoliert und als Index in der Page-Table verwendet, aus welcher die physischen Adresse entnommen wird.

Die Einträge in der Page-Table sind 32-Bit breit, es werden aber nur 20-Bit benötigt, denn sie muss, wie bereits erwähnt, an einer durch 4 kB teilbaren physischen Speicherstelle beginnen. Egal welche Page man nimmt, die unteren 12-Bit des Index sind deshalb eigentlich immer 0. Um diese 12-Bit aber doch sinnvoll zu nutzen, werden dort verschiedenen Flags untergebracht, welche z. B. Definieren, ob eine Page ausgelagert ist oder als Read-Only deklariert wurde.

Damit für eine komplette Page-Table keine kompletten 4 MB verwendet werden müssen, wurde der Mechanismus von Intel etwas verfeinert. Anstelle einer großen Page-Table für den gesamten 32-Bit-Adressraum wurde eine zweistufige Organisation mit mehreren Page-Tables gewählt. Ausgangspunkt ist das Page-Directory, über das mehrere kleinere Page-Tables verwaltet werden. Es werden die oberen 10-Bit der linearen Adresse als Index im Page-Directory verstanden, welches aus 1024 Einträgen besteht. Eine Adresse des Page-Directory wird über das CR3-Register bereitgestellt. Die einzelnen Einträge einer Page-Directory enthalten Zeiger mit den Adressen der eigentlichen Page-Tables. Die Nummer des Eintrags, welcher für die Umrechnung der Adresse herangezogen wird, ergibt sich aus den Bits 12–21 der linearen Adresse. Die lineare Adresse wird so in 2 Teile aufgespalten, Directory und

Table.

Das Page-Directory und jede Page-Table nimmt so die Adressen von 1024 Einträgen auf. Jeder Eintrag in der Page-Directory deckt so einen Bereich von 4 MB (4096x1024) innerhalb des linearen Adressraums ab. Der Vorteil ist, dass das System so die Page-Tables nach Bedarf anlegen kann.

Besonders wichtig für die Zusammenarbeit mit dem Betriebssystem sind die unteren 20-Bit der Page-Table-Einträge. Hier werden wie schon erwähnt die Flags der Pages untergebracht. Besonders wichtig ist das Present-Flag, es muss vom Betriebssystem auf 0 gestellt werden, wenn eine Page ausgelagert wurde. Für den Prozessor ist, dass das Signal um eine Exception auszulösen. Dadurch erlangt das Betriebssystem die Kontrolle über die Programmausführung und erhält die Möglichkeit die Page nachzuladen und den Page-Table-Eintrag mit einer neuen Basisadresse der Page im Speicher zu initialisieren.

Wird auf eine Page das erste Mal zugegriffen, so wird im Zuge der Aktualisierung der Basisadresse, das Access-Flag auf 1 gesetzt. So kann das Betriebssystem später besser entscheiden, welche Pages, sollte der Speicher mal knapp werden, zuerst ausgelagert werden. Hier werden die bevorzugt, wo das Access-Flag noch auf 0 steht. Pages, welche einen Schreibzugriff hatten, bei denen wird zudem das Dirty-Bit auf 1 gesetzt, weil es einfacher ist Pages zu laden, welche noch unverändert sind.

Andere Flags realisieren ein Schutzmechanismus zwischen System-Code und User-Code oder markieren eine Page, wie oben erwähnt, als schreibgeschützt. Bei Missachtung dieser Flags, wird eine Exception im Prozessor ausgelöst.

1.3 Prozessspeicher

Ein Prozess ist grundlegend in 3 Regionen aufgebaut. Diese Regionen sind das Codesegment, das Datensegment und der Stack. Diese Segmente ergeben sich in der Regel direkt durch das Parsen einer ausführbaren Datei.

1.3.1 Textsegment

Das Textsegment beinhaltet das Codesegment für den auszuführenden Code und Kostanten. Konstanten sind Daten, welche zur Laufzeit des Programms nicht verändert werden dürfen. Das komplette Textsegment ist schreibgeschützt. Ein Versuch, hier Daten zu ändern, würde zu einer Speicherzu-

griffsverletzung führen. Das Textsegment beginnt in der Regel im niedrigen Adressraum.

1.3.2 Datensegment

Im Datensegment sind initialisierte und nicht initialisierte globale Variablen angesiedelt. Der dynamische Speicher wird hier auch hinzugezählt, dieser Bereich ergibt sich aber nicht durch das Parsen einer ausführbaren Datei sonders erst zur Laufzeit.

Dynamischer Speicher

Anwendungen können über verschiedene Möglichkeiten, während der Ausführung Speicher allozieren. Hierfür verwendet man in der Regel die Funktionen der entsprechenden Bibliotheken, welche die Speicheranforderung dieser Funktionen aus dem virtuellen Speicher bedienen.

Es ist möglich Speicher zu allozieren, welcher sich direkt in das Paging-Konstrukt eingefügt. Diese Art wird oft benutzt, um Dateien in den Speicher einzulesen oder große Arrays anzulegen. Es ist so möglich sich regelrecht Abschnitte im linearen Adressraum zu reservieren. Das Betriebssystem nutzt diese Möglichkeit, um Teile des Adressraums für sich und den geladenen Prozess zu reservieren. Auch der User-Code kann davon profitieren, überall dort, wo Datenstrukturen während der Programmausführung dynamisch wachsen. Dies ergibt Sinn, da der Erweiterungsspeicher beim Vergrößern einer Datenstruktur, wie z. B. einer verketteten Liste, nicht einfach hinten an gehangen werden kann. Andernfalls würde ein höherer Aufwand mit Zeigern und verknüpften Listen betrieben werden oder man müsste Speicher beim Vergrößern wandern lassen. Allozierter virtuelle Speicher des User-Codes steht somit dem Kernel oder verwendeten Bibliotheken nicht zur Verfügung. Benötigt man Speicher für Bäume und Listen, so kann man über die Heap-Funktionen Speicher Byte-genau allozieren. So kann man dem Problem entgehen, das große reservierte Speicherabschnitte den physikalischen Speicher oder die Auslagerungsdatei regelrecht sprengen.

Ein Prozess muss den Speicher, welchen er wirklich nutzen will noch commiten. Das hat den Vorteil, dass Arrays trotzdem sequenziell wachsen können, obwohl hier erst mal nur ein kleiner Teil des reservierten Speichers benutzt wird. Nach dem Commit wird der dynamische Speicher im Page-Konstrukt

wirklich angelegt. Sollte der committete Bereich für die anfallenden Daten nicht groß genug sein, so wird der Prozessor eine Exception auslösen.

Heap steht für Halde oder Haufen und wird benötigt, wenn man viele kleine Speicherblöcke verwalten möchte, ohne sich die Frage stellen zu müssen, ob dieser Speicher verfügbar ist. Die C-Standardbibliothek stellt hier die Funktionen malloc(), realloc() und free() bereit, welche auf dem entsprechenden Betriebssystem-API basieren. Über das Betriebssystem-API ist es in der Regel möglich, mehrere Heaps mit unterschiedlichen Größen anzulegen. Der Speicher für Heaps wird aus dem virtuellen Adressraum der Anwendung bezogen. Die jeweiligen Heaps werden dann über ein Handle (Zeiger) angesprochen. Den Prozess-Heap gibt es bereits beim Start eines jeden Prozesses, da der Loader in ihm Informationen ablegt, die der Kernel verwalten muss. Der angefragte Speicher wird immer auf eine Page-Größe aufgerundet.

1.3.3 Stack

Der Stack befindet sich am Ende des Prozessadressraumes und wächst in Richtung Code und Datensegment. Der Stack kann mit den Prozessor- bzw. Assemblerbefehlen PUSH und POP be- und entladen werden. Das heißt sein Aufbau ähnelt hier einem Stapel von Tellern, welchen der Stack nach dem Last-In-First-Out-Prinzip (LIFO) abarbeitet. So kann mit POP nur das Element entladen werden, welches zuletzt auf dem Stack abgelegt wurde.

Der Stack wird in der Regel benutzt, um Parameter und Rückgabewerte zu übergeben und diverse Rücksprungadressen zu vorherigen Prozeduren vorzuhalten. Üblich ist auch das die lokalen Variablen einer Prozedur (Funktion), auf dem Stack angelegt werden. Das vorhalten der lokalen Variablen auf dem Stack ermöglicht es, dass verschiedene Prozeduren verschieden oft aufgerufen werden können. Der Bereich im Stack, indem alle relevanten Daten einer Prozedur enthalten sind, wird als Stack Frame bezeichnet. Wurden mehrere Prozeduren aufgerufen, so existieren auch mehrere dieser Frames im Stack, für jeden Aufruf eine eigene.

Der Stack Pointer (SP) ist ein Register, ähnlich dem Instruction Pointer (IP) und zeigt immer auf die Adresse des obersten Elements im Stack. Das Ende des Stacks ist eine feste Adresse und für gewöhnlich auch das Ende des kompletten Programms. Es gibt auch Ausnahmefälle in der Stackimplementierungen, hier wächst der Stack dann zu einer größeren Adresse hin. Bei gängigen Prozessoren ist dies dennoch nicht der Fall. Als Zusatz, um eine

Variable oder Parameter referenzieren zu können, braucht der Stack Pointer noch den Base Pointer (BP), welcher auf irgend einen Wert im Stack zeigen kann. Auch als Frame Pointer (FP) kommt er hier ggf. zum Einsatz, welcher auch in vielen Texten als Local Base Pointer (LB) bezeichnet wird.

Der Base Pointer wird benutzt um lokale Variablen über einen Offset zu referenzieren. Dieser Offset beginnt bei der Adresse, welche im Stack Pointer gespeichert ist. Bei weiteren Prozeduraufrufen oder einfach nur weiteren PUSH-Anweisungen wird der Offset aber quasi größer und muss verändert werden, diese Veränderungen werden in der Regel vom Compiler vorgenommen. Muss der Compiler die Offsetadresse im Base Pointer oft aktualisieren, ist das je nach dem ein nicht zu unterschätzender Aufwand, da bei zu großen Offsets ggf. auch noch weitere Code ausgeführt werden muss. In Manchen Fällen ist der Compiler gar nicht in der Lage diese Offset-Korrekturen fehlerfrei vorzunehmen. Eine andere und oft genutzte Strategie um lokale Variablen zu referenzieren ist hier die Benutzung des Base Pointers als Frame Pointers. Der Zugriff auf Variablen und Parameter wird bei dieser Vorgehensweise mit einem Offset zu dem Wert im Frame Pointer ermöglicht, da der Frame Pointer nicht abhängig von dem Wert im Stack Pointer ist, muss auch der Compiler hier keine nachträglichen Korrekturen vornehmen. Der Wert im Frame Pointer ist so gewählt, dass für gewöhnlich Parameter, Rücksprungadresse und der alte Frame Pointer einen positiven Offset besitzen und lockale Variablen einen negativen.

Das erste was eine Prozedur tun muss, nachdem sie aufgerufen wurde, ist sie speichert den aktuellen Wert des Frame Pointers auf dem Stack und kopiert die aktuelle Stack Pointer Adresse in den Frame Pointer. Darauffolgend wird der Stack Pointer so verändert, dass genug Platz für die lokale Variablen entsteht, unser sogenannter Stack Frame wird hier angelegt. Oft fällt die größe des Stack Frames etwas größer aus als die Summe der Bytes der lokalen Variablen, bei einer 32-Bit Architektur ist ein Element des Stacks genau 4 Bytes groß. Das hat den Sinn, das Werte in die Register der CPU passend abgelegt werden können und nicht zusätzlicher Code erzeugt werden muss, der z. B. Werte zweier Variablen in einem Register erkennen und verwalten muss.

Soll eine Prozedur beendet werden, so muss der vorherige Zustand natürlich wieder genau hergestellt werden, Prozessoren bringen hier die Instruktionen ENTER und LEAVE bzw. LINK und UNLINK mit, bzw. kann man hier auch einfach wieder mit POP und verändern der Adressen den Urzustand herstellen.

Diese Vorgehensweisen werden als Prozedur-Prolog bzw. -Epilog verstanden und können bei verschiedenen Prozessoren etwas abweichen.

Kapitel 2

Speicherüberlauf

2.1 Stack-Based Buffer Overflow

Bei vielen C Implementationen ist es möglich den vorhandenen Stack zu korrumpierten, indem man über das Ende eines im Stack existierenden Puffers schreibt. Ein Puffer ist ein Bereich im Speicher, welcher nacheinander verschiedene Instanzen desselben Datentypen beinhaltet. In C sind dies für gewöhnlich char-Arrays um Strings zu speichern.

Man muss hier unterscheiden zwischen Globale Variablen, welche beim Laden der Anwendung im Datensegment angelegt werden und lokale Variablen aus Funktionen die dynamisch auf dem Stack erzeugt werden.

Verlässt man diesen sogenannten Puffer, so überschreiben die folgenden Daten Werte im Stack, welche sich hinter dem Puffer befinden. Somit ist es möglich, die Rücksprungsadressen von z. B. Prozeduren zu verändern und dafür zu sorgen, dass der Instruction Pointer an einer anderen Stelle den Code fortführt.

In Assembler bzw. eigentlich eher im Objektcode werden Prozeduren bzw. Funktionen über ein CALL aufgerufen und mit einem RET ist man dann in der Lage zur vorherigen zurück zu springen. Ein CALL tut dabei nichts anderes, als die Adresse des Befehls, welcher sequentiell quasi direkt nach dem CALL ausgeführt würde als Rücksprungsadresse auf dem Stack abzulegen und das Register des Instruction Pointer auf die Anfangsadresse der neuen Prozedur zu setzen, ähnlich wie mit einem MOV. Ein direktes Ändern des Instruction Pointer ist MOV aber nicht erlaubt. Um zum ursprünglichen Code zurück zu gelangen, entfernt RET die Rücksprungsadresse vom Stack

und ändert den Instruction Pointer wieder auf genau diese.

Nun passiert es oft, dass Quelltexte so geschrieben werden, dass Strings von dem ein Puffer in einen anderen Puffer byteweise kopiert werden, so lange bis ein Stringendezeichen (der Wert 0) in einem Byte des Quellstring vorkommt. Ist der Zielpuffer z. B. aber nur 16 Byte groß, die Quelle aber 256 Byte groß, so würde die veraltete Funktion strcpy (die Alternative ist strncpy), aus der C-Standardbibliothek, erbarmungslos bis zum Ende des Quellstrings alle Bytes kopieren. Auf Maschinencodeebene wird hier vermutlich nur ein Register inkrementiert und mit einem MOV kopiert man dann schrittweise die Daten. Hier sollte angemerkt sein das MOV in verschiedenen Varianten vorliegt und auch in der Lage ist Words (16-Bit), Doublewords (32-Bit) und Quadwords (64-Bit) zu kopieren.

Passiert zur Laufzeit ein solcher Fehler, so bekommt man oft, ein Segmentation Fault. Die Rücksprungadresse, welche RET dann benutzt, wurde hier vermutlich mit einem Wert überschreiben der nicht existiert.

Findet man nun solch einen Fehler in ein Programm, so kann man durch die richtige Anzahl von Bytes, so viel vom Stack überschreiben, das nur die Rücksprungsadresse betroffen ist und so kein Speicherzugriffsfehler ausgelöst wird. Die Bytes, welche wir hier von strcpy kopieren lassen, enthalten natürlich schon unseren Code, welcher später ausgeführt werden soll.

2.2 Global-Based Buffer Overflow

2.3 Heap-Based Buffer Overflow

2.4 Stackoverflow

Sollte der Platz zwischen Datensegment und Stack einmal zu gering werden, so wird der Prozess angehalten und kurz darauf mit zusätzlichem Speicher fortgesetzt.

2.5 Integeroverflow

2.6 Format-String-Attacke

2.7 Return to Libc

2.8 Use after Free

2.9 Shellcode (Payload)

Kapitel 3

Gegenmaßnahmen

3.1 NX-Bit

3.1.1 Angriffe NX-Bit

3.2 ASLR

3.2.1 KASLR

3.2.2 KARL

3.2.3 KAISER/KPTI (Kernel- und User-Space "page table splitting")

3.2.4 PIE-Flag

3.2.5 Angriffe ASLR

Heap-Spraying

(

Double-Page-Fault)

(

Sidechannel-Angriff)

3.3 Stack-Cookie

3.4 Shadow Stack

3.5 Non Excutable Stack

3.6 Canary

3.7 CFI

3.8 Stack Smashing Protector (ehemals ProPolice)

3.9 Stack Guard

3.10 Syscall-Filtering (https://en.wikipedia.org/wiki/Se

3.11 Sandboxing/Container (Chroot, LXC, Jails, Apps, Browser, Java, ...)

3.11.1 Jailbreak

Kapitel 4

Finden von Overflows

Kapitel 5

Vermeiden von Overflows