The Unlimited

8. September 2017

Kapitel 1

Speicher

1.1 Virtuelle Speicherverwaltung (32-Bit)

Man arbeitet mit dem flat memory model bzw. dem protected Mode. Die Zusammenarbeit zwischen dem Prozessor und dem Betriebssystemkern ermöglicht, dass jedem Prozess einen Adressraum von 4 GB ($2\hat{3}2$) zur Verfügung steht. Die virtuelle Speicherverwaltung arbeitet nicht mehr kombiniert mit einer Segment- und einer Offsetadresse (DOS), sondern man benutzt eine einzige 32-Bit große (Offset-)Adresse für den kompletten Adressraum. Der Systemkern setzt beim Start einer Anwendung einen 4 GB umfassenden Adressraum. Dies ist auch der Fall, wenn gar keine 4 GB RAM zur Verfügung stehen. Eine Anwendung belegt nur so viel Speicher, wie der Programmcode und die zugehörigen Daten benötigen. Eine Anwendung kann später weiteren Speicher anfordern, dies ist der virtuellen Speicherverwaltung zu verdanken. Verwaltet wird die virtuelle Speicherverwaltung durch den Systemkern und basiert auf einem grundlegenden Mechanismus der Speicheradressierung des Prozessors. Oft ist nicht der gesamte virtuelle Speicher einer Anwendung physikalisch verfügbar, da sich eine Anwendung in der Regel nur in Teilen des Codes oder der Daten bewegt. So ist 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 erst ein mal aus der Adresse die Nummer der Page. Er bedient sich dazu der Page-Tabellen auch Page-Directorys genannt. Über sie wird der Adressraum Stück für Stück auf Pages im physikalischen Speicher verteilt. Der Prozessor gibt das Format dieser Tabellen vor aber sie werden vom Betriebssystem verwaltet. In dem jeweiligen Eintrag der Page-Tabelle schaut der Prozessor, ob sich die Page im physikalischen Speicher befindet. 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 Systemkern gemeldet, welcher sich um das Nachladen der Pages kümmert. Ist der komplette physikalische Speicher in Benutzung, so müssen erst andere Pages ausgelagert werden. Was ein Programm an durchgängigen Adressraum sieht erstreckt sich so in Wirklichkeit über viele Pages, welche an ganz unterschiedlichen Stellen im physikalischen Speicher existieren. Der Speicher einer Anwendung kann bis zu Unkenntlichkeit fragmentiert sein, 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. Nicht der komplette Adressraum von 4 GB steht aber der Anwendung zur Verfügung. Pages von Bibliotheken z. B. können in mehreren Anwendungen zur Verfügung stehen und eine Instanz einer Bibliothek benötigt auch einen Datenbereich. Adressen größer 3 GB sind dem Systemkern vorbehalten, somit ist anzunehmen, das, wenn die CPU in diesen Adressraum wechselt, sie auch von Ring-3 zu Ring-0 wechselt.

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, hier werden lineare Speicheradressen direkt auf physikalische Adressen abgebildet und Paging findet noch nicht statt. Schalte das Betriebssystem den Prozessor in den Protected-Mode, so setzt es dieses Bit auf 1. Jede Adresse, die der Prozessor bezüglich eines Maschinenbefehls verarbeitet, wird in diesem Modus erst einer Page zugeordnet.

Ein Speicherzugriff wird ab jetzt auf eine Page umgeleitet. Die Größe einer Page beträgt 4 kB und jede Page beginnt an einer durch 4 kB teilbaren physikalischen Adresse. Der Adressraum wird dadurch in $2\hat{2}0$ verschiedene Pages aufgeteilt, die jeweils $2\hat{1}2$ Byte (4 kB) umfassen. Die Ausrichtung auf 4 kB ermöglicht, dass die untere 12 Bitgrenze dafür sorgt, dass lineare Adressen und physikalische Adressen identisch sind. Sie stellen eine Art Offset der Page da. Die oberen 20 Bit der linearen Adresse werden für die Nummerierung der Page verwendet. Diese 20 Bit werden isoliert und als Index in der Page-Table verwendet, aus welcher die physikalische Adresse entnommen wird. Die Einträge in der Page-Table sind 32 Bit breit, es werden aber nur 20 Bit benötigt, denn sie muss an einer durch 4 kB teilbaren physikalischen Speicherstelle beginnen. Egal welche Page man nimmt, die unteren 12 Bit des Index sind deshalb immer 0. Somit werden dort die verschiedenen Flags untergebracht, welche z. B. definieren, ob eine Page ausgelagert ist. 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 in 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 verschiedenen Page-Tables. Die Nummer des Eintrags, welcher für die Umrechnung der Adresse herangezogen wird, ergibt sich aus dem Besitz 12–21 der linearen Adresse. Eine lineare Adresse wird so in 2 Teile aufgespalten. Das Page-Directory und jede Page-Tabelle nimmt so die Adressen von 1024 Pages auf. Jeder Eintrag in der Page-Directory deckt so einen Bereich von 4 MB innerhalb des linearen Adressraums ab. Das System kann so die Page-Tables nach Bedarf anlegen. 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. Das Acccess-Flag wird gesetzt, wenn auf eine Page das erste Mal zugegriffen wurde. So

kann das Betriebssystem später besser entscheiden, welche Pages zuerst ausgelagert werden sollen, insofern der physikalische Speicher knapp wird. Hier werden die bevorzugt, wo das Access-Flag noch auf 0 steht. Pages, welche einen Schreibzugriff hatten, bei denen wird das Dirty-Bit auf 1 gesetzt. Es ist einfacher Pages zu laden, welche unverändert sind. Andere Flags realisieren ein Schutzmechanismus zwischen System-Code und User-Code oder markieren eine Page als schreibgeschützt. Bei Missachtung dieser Flags wird eine Exception ausgelöst.

1.3 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 System 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 Struktur nicht einfach hinten angehangen 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 Systemkern oder verwendeten Bibliotheken nicht zur Verfügung. Benötigt man Speicher für Bäume und Listen, so kann man über die Heap-Funktionen Speicher Byte-genau allozieren. Um dem Problem zu 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. Ab hier wird der dynamische Speicher im Page-Konstrukt wirklich angelegt. Ist der committete Bereich für die anfallenden Daten nicht groß genug, so wird der Prozessor eine Exception auslösen.

1.4 Heap

Wenn man viele kleine Speicherblöcke verwalten möchte, ohne sich die Frage stellen zu müssen, ob dieser Speicher verfügbar ist, dann benutzt man die verschiedenen Heap-Funktionen. Die C-Standard-Funktionen hierzu sind zum Beispiel malloc(), realloc() und free(), 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 angesprochen. Dem Prozess-Heap gibt es bereits beim Start eines jeden Prozesses. Der Loader legt in ihm Informationen ab, die der Systemkern verwalten muss. Der angefragte Speicher wird immer auf eine Page-Größe aufgerundet.

1.5 Code-/Textsegment

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. Das Codesegment, welches oft auch als Textsegment betitelt wird, beinhaltet, wie der Name schon sagt, den auszuführenden Code und ist schreibgeschützt. Ein Versuch, hier Daten zu ändern, würde zu einer Speicherzugriffsverletzung führen. Das Codesegment beginnt in der Regel im niedrigen Adressraum.

1.6 Datensegment

Im Datensegment sind initialisierte und nicht initialisierte globale Variablen angesiedelt, sowie Konstanten. Auch der dynamische Speicher und der Heap werden hier hinzugezählt, diese Bereiche ergeben sich aber nicht aus der ausführbaren Datei.

1.7 Stack

Der Stack befindet sich am Ende des Prozessadressraumes und wächst in Richtung Code und Datensegment. Der Stack kann mit den Prozessorbefehlen PUSH und POP be- und entladen werden. Sein Aufbau ähnelt hier einem

Stapel von Tellern. Der Stack arbeitet nach dem Last-In-First-Out-Prinzip (LIFO). 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 Rücksprungsadressen zu vorherigen Prozeduren vorzuhalten. Üblich ist auch das die lokalen Variablen einer Prozedur, auf dem Stack angelegt werden, somit ist es möglich verschiedene Prozeduren verschieden oft aufzurufen. Der Stackzeiger (SP) ist ein Register ähnlich wie der Instruktionszeiger und zeigt immer auf das oberste Element im Stack. Das Ende des Stacks ist eine feste Adresse und für gewöhnlich auch das Ende des kompletten Programms. In Ausnahmefällen gibt es auch Stackimplementierungen, wo der Stack zu einer größeren Adresse hin wächst, bei gängigen Prozessoren ist dies dennoch nicht der Fall. Als Zusatz zu dem Stackzeiger benutzt man oft noch den Base Pointer (BP), welcher auf irgend einen Wert im Stack zeigen kann. Zusätzlich wird der Framezeiger (FP) benutzt, welche auch in vielen Texten als Local Base Pointer (LB) betitelt wird. Der Base Pointer wird benutzt um lokale Variablen über einen Offset zu referenzieren. Dieser Offset beginnt bei der Adresse, welche im SP gespeichert ist. Bei diversen PUSHs und POPs kann sich dieser Offset aber verändern. In der Regel wirft der Compiler ein Auge hierauf und korrigiert die entsprechenden Werte. Viele Compiler benutzen aber das zweite Register, den Framezeiger und referenzieren damit lokale Variablen sowie Parameter, da der Wert im Framezeiger nicht abhängig vom Stackzeiger ist. Der Zugriff auf Variablen sowie Parameter wird hier nun mit einem Offset zu dem Wert im Framezeiger ermöglicht. Das erste was eine Funktion bzw. Prozedur tun muss, nachdem sie aufgerufen wurde, sie speichert den aktuellen Wert des älten"Framezeiger auch auf dem Stack und kopiert die aktuelle Adresse des Stackzeiger in den Framezeiger, danach wird der Stackzeiger verändert, um Platz für neue lokale Variablen zu schaffen. Dieser Platz, ist der berüchtigte Puffer, aus dem man ausbrechen könnte, um eine Rücksprungadresse zu verändern. Soll eine Prozedur beendet werden, so muss der vorherige Zustand wieder genau hergestellt werden, Prozessoren bringen hier die Instruktionen ENTER und LEAVE bzw. LINK und UNLINK mit, natürlich kann man hier auch einfach wieder mit POP und verändern der Adresse den Urzustand wieder herstellen. Diese beiden Vorgehensweisen werden auch als Prozedur-Prolog bzw. -Epilog verstanden. Das gesamte Vorgehen kann aber bei verschiedenen Prozessoren etwas abweichen. Sollte der Platz zwischen Datensegment und Stack einmal zu gering werden, so wird der Prozess angehalten und kurz darauf mit zusätzlichem Speicher fortgesetzt.

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. Globale Variablen werden beim Laden der Anwendung im Datensegment angesiedelt. Lokale Variablen in Funktionen werden aber dynamisch im Stack angelegt. Verlässt man den Puffer, so überschreiben weitere Daten die Werte im Stack, welche sich hinter diesem Puffer befinden. Somit ist es möglich die Rücksprungsadressen von z. B. Prozeduren zu verändern und dafür zu sorgen, dass der Instruktionszeiger an einer anderen Stelle den Code fortführt. In Assembler bzw. eigentlich eher im Objektcode ist es möglich Prozeduren bzw. Funktionen über ein CALL anzuspringen und über einen RET wieder zurückzukehren. Ein CALL tut dabei nichts anderes, als die Adresse nach dem CALL als Rücksprungsadresse auf dem Stack abzulegen und das Register des Instruktionszeigers auf die Anfangsadresse der Prozedur zu setzen, ähnlich wie bei einem MOV. Ein direktes Ändern des Instruktionszeigers ist MOV aber nicht erlaubt. Um zum ursprünglichen Code zurück zu gelangen, entfernt RET die Rücksprungsadresse vom Stack und ändert den Instruktionszeiger auf genau diese. Nun passiert es oft, dass Quelltexte so geschrieben worden, dass Strings von dem ein Puffer in einen anderen Puffer byteweise kopiert werden, so lange bis ein Stringendezeichen (der Wert 0) in ein Byte des Quellstring vorkommt.

Ist der Zielpuffer z. B. aber nur 16 Byte groß, die Quelle aber z. B. 256 Byte groß, so würde die schon etwas obsolete Funktion strcpy (die Alternative ist strncpy), aus der C-Standardbibliothek, erbarmungslos bis zum Ende kopieren. Da hier vermutlich auf Maschinencodeebene immer nur ein Zeiger inkrementiert wird und mit einem MOV die Daten kopiert werden. Passiert zur Laufzeit ein solcher Fehler, so bekommt man oft, ein segmentation fault. Die Adresse, welche in der Quellzeigervariable steht, wird mit Bytes überschrieben und somit überschreibt dieser Fehler sogar die Quelladresse von strcpy, danach versucht MOV, Daten von einer Adresse zu lesen, die nicht zur Verfügung steht. 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. 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

2.5 Integeroverflow

2.6 Shellcode (Payload)

Kapitel 3

Gegenmaßnahmen

3.1 NX-Bit

3.1.1 Angriffe NX-Bit

3.2 ASLR

3.2.1 PIE-Flag

3.2.2 Angriffe ASLR

Spraying

3.3 Stack-Cookie

3.4 CFI

3.5 Stack Smashing Protector (ehemals ProPolice)

3.6 Stack Guard

3.7 Syscall-Filtering (https://en.wikipedia.org/wiki/Sec

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

3.8.1 Jailbreak

Kapitel 4

Finden von Overflows

Kapitel 5

Vermeiden von Overflows