Sicherheitslücken Meltdown und Spectre

Im Juni 2017 wurde Intel vom Google Project Zero, welches sich mit dem Finden von unbekannten Lücken in Computern beschäftigt, im Geheimen informiert, dass es ein großes Problem mit ihren Prozessoren gibt. Am 04.01.2018 wurden die bislang geheim gehaltenen Informationen öffentlich und die Sicherheitslücken Meltdown und Spectre waren geboren. Betroffen sind wohl alle Prozessoren der letzten 10 bis 15 Jahre und alle Betriebssysteme und Plattformen.

Momentan werden sowohl von den Betriebssystemherstellern als auch von den Prozessorherstellern Intel, AMD und Nvidia Patches für diese Lücke gebaut, getestet und ausgeliefert. Diese Patches haben aber wahrscheinlich eine merkbare Reduzierung der Performance der Systeme zur Folge. Momentan pendelt sich der erwartete Verlust an Geschwindigkeit zwischen 1% und 5% ein. Dies kann man natürlich nie genau sagen, da es sehr davon abhängt, welche Art von Arbeit der Prozessor tatsächlich durchführt.

Für diejenigen, die technische Hintergründe im Detail interessieren, folgt nun eine leicht vereinfachte Erklärung von Meltdown und Spectre. Fangen wir mit Meltdown an. Spectre ist hier sehr ähnlich.

Sicherheitslücke Meltdown

Dieser Angriff nutzt das Zusammenspiel von vier essentiellen Systemen in modernen Computern aus. Diese Systeme sind keine Programme, sondern wirklich in Silizium gegossene Hardware. Deshalb ist eine einfache Anpassung der Systeme sehr kompliziert!

1. Die Memory Management Unit (MMU) des Prozessors

Die MMU sorgt für eine Trennung zwischen sogenanntem virtuellen Arbeitsspeicher und

pyhsikalischem Arbeitsspeicher. Der physikalische Arbeitsspeicher sind die RAM-Riegel, die Sie in Ihrem Computer eingebaut haben. Diese haben 4 GB, 8 GB, 16 GB und zusammen vielleicht 32 GB, weil es mehrere sein können.

Der virtuelle Arbeitsspeicher ist ein vorgetäuschter Arbeitsspeicher, den jeder Prozess bekommt. Wenn Sie nämlich nur 2 GB Ram haben, reicht dieser gerade mal von Adresse 0x0000 0000 - 0x8000 0000.

Was aber nun, wenn ein Programm erwartet, dass es alle möglichen Adressen, die mit 32Bit adressiert werden können, von 0x0000 0000 - 0xFFFF FFFF haben kann? Oder wenn mehrere Programme die Adresse 0x0000 1000 verwenden wollen?

Klar könnten wir jedem Programm vorher sagen „Du bekommst die Adressen von X bis Y.“ und erwarten, dass es sich daran hält. Dann haben wir aber noch keine Lösung gefunden zu verhindern, dass es trotzdem von anderen Adressen liest und somit andere Programme ausspioniert. Dies war früher tatsächlich normal. Zu Zeiten von MS-DOS konnte jedes Programm jederzeit jeden Speicher beschreiben.

Die Lösung für all diese Probleme ist nun die MMU. Die MMU ist eine Hardware-Einheit auf dem Prozessor, die nicht nur sehr schnell virtuelle Adressen in physikalische Adressen umwandeln kann, sondern sie kann die Adressen zudem auch noch mit Berechtigungen versehen. Bei Linux ist es nun zum Beispiel so, dass jeder Prozess mit den Adressen 0x0000 0000 - 0xBFFF FFFF tun und lassen kann was er will. Die Adressen 0xC000 0000 - 0xFFFF FFFF hat aber auch jeder Prozess. Diese gehören jedoch dem Betriebssystem (Linux Kernel) und sind trotzdem aus Geschwindigkeitsgründen von jedem Prozess aus zugreifbar. Das war vor Meltdown nie ein Problem, denn die MMU verwaltet die Berechtigungen auf die Speicheradressen des Kernels. Möchte also ein Prozess wissen, welches Passwort der aktuelle Benutzer hat (solche Sachen merkt sich der Kernel), so kann er einfach alle Adressen des Kernels durchsuchen bis er das Passwort gefunden hat. Er kommt hier aber nicht weit, denn beim ersten Zugriff auf eine Speicherseite, die nicht dem Prozess, sondern dem Kernel gehört, blockiert die MMU den Prozess und informiert das Betriebssystem über einen möglichen Sicherheitsbruch - bis jetzt Meltdown kam…

2. Der Prozessor-Cache

Der physikalische Arbeitsspeicher ist zwar schon sehr schnell, man kann ihn aber noch schneller bauen - er wird dann nur deutlich teurer. Außerdem kann man ihn auch noch näher an den Prozessor bauen, denn auch die Lichtgeschwindigkeit, mit der Signale von und zum Speicher übertragen werden, benötigt Zeit. Noch dazu besteht die Möglichkeit, dass der Weg zum Arbeitsspeicher (der Bus) momentan blockiert ist, weil z.B. die Festplatte gerade mit dem Arbeitsspeicher Daten austauscht. Das Ergebnis des Ganzen ist ein sehr, sehr schneller Speicher, der ganz nah neben dem Prozessorkern steckt: der sogenannte Cache. Er wird unterteilt in Level 1 (L1), Level 2 (L2) und Level 3 (L3). Der Prozessor entscheidet selbst, was er sich hier zwischenspeichert, aber im Regelfall nutzt er das Prinzip der zeitlichen und räumlichen Nähe aus. Greife ich auf Adresse 0x1000 zu, so ist es wahrscheinlich, dass ich auch auf Adresse 0x1001 zugreife → räumliche Nähe. Greife ich auf Adresse 0x2000 zu, so ist es wahrscheinlich, dass ich kurze Zeit später wieder darauf zugreife → zeitliche Nähe.

3. Out of Order Execution

Im Gegensatz zu uns Menschen, die mit zwei Armen eher dürftig ausgestattet sind, hat ein Prozessor nicht nur viele Kerne, sondern auch in seinen Kernen viele Bauteile mehrfach verfügbar. Die Recheneinheit zum Beispiel, die Arithmetisch-logische Einheit (ALU), welche mathematische Berechnungen durchführt, kann mehrmals vorkommen. Deshalb ist ein Prozessor nicht unbedingt daran gebunden, jeden Programmcode tatsächlich Zeile für Zeile auszuführen, so wie ihn der Programmierer eigentlich vorgibt.

Wir nehmen folgenden Beispielprogrammcode:

1. a = 5 + 1

2. b = a + 4

3. c = 10

4. d = a + 4

So hat der Prozessor, auch wenn er momentan an Zeile 1 steht, keinerlei Probleme, die Zeile 3 schonmal gleichzeitig auszuführen. Sobald er an Zeile 2 steht, hat a auch einen festen Wert und Zeile 4 kann schon einmal berechnet werden. Prozessoren machen das dauerhaft und auch wenn der Programmierer denkt, es wird jeder Programmcode Zeile für Zeile abgearbeitet, so macht ein moderner Prozessor ein Dutzend oder mehr Befehle gleichzeitig. Wir Menschen setzen auch das Wasser auf die Herdplatte und schneiden nebenbei das Gemüse, während wir darauf warten, dass das Wasser kocht. Es würde keinen Sinn machen, zu warten bis das Wasser kocht und dann erst mit dem Gemüse schneiden anzufangen, obwohl es genau in dieser Reihenfolge im Kochbuch steht. Prozessoren sind sehr gut darin zu merken, welche Befehle sie bereits vorbereiten können, während sie auf die Ausführung der aktuellen Zeile warten.

4. Branch Prediction Unit

In jedem Programm gibt es Tausende und Abertausende von Entscheidungen. Der Prozessor muss ständig Entscheidungen abarbeiten, die sein können: „Wenn Eingabe größer 0, dann berechne den Wert.“ oder „Wenn der eingegebene Name länger als 10 Zeichen ist, dann springe zu einer Fehlermeldung.“

Im Programmcode sieht das zum Beispiel so aus:

1. x = Eingabe des Users

2. Wenn x > 0 gehe zu 4.

3. Fehlermeldung „x ist zu klein“

4. Ergebnis = x + 10

Hier kann der Prozessor in Zeile 2 entweder einfach mit Zeile 3 weitermachen, oder aber er springt zu Zeile 4. Die Branch Prediction Unit versucht nun herauszufinden, welche der beiden Möglichkeiten wahrscheinlicher ist. Die Befehle, die am wahrscheinlichsten sind, werden der Out of Order Execution zum Vorbereiten gegeben. Unter anderem merkt sie sich, wie die letzten Male bei der Entscheidung gesprungen wurde (also ob gesprungen wurde oder nicht) und geht davon aus, dass es diesmal wieder genauso sein wird.

Diese 4 Punkte beenden unseren Crashkurs über das Innenleben eines modernen Prozessors. Mit diesem Wissen können wir problemlos die Meltdown Attacke erklären:

Wie wir mittlerweile wissen, könnte es einen Prozess geben, der die virtuellen Adressen 0x0000 0000 - 0xB000 0000 belegt. In „seiner“ MMU gibt es aber auch die Adressen 0xC000 0000 - 0xFFFF FFFF, welche das Betriebssystem belegt und auf welche er nicht zugreifen darf.

Nun machen wir Folgendes:

1. Wir leeren den kompletten Prozessorcache. Dafür gibt es einen speziellen Befehl.

2. Wir merken uns einen Speicherbereich, auf den wir zugreifen dürfen. Zum Beispiel 0x1000.

3. Wir lesen von der verbotenen Adresse 0xB000 0001. Das Ergebnis ist 0x15, welches wir uns merken. Dieser Schritt ist natürlich nicht erlaubt! Er wird vom Prozessor nicht ausgeführt werden, weil die MMU weiß, dass wir keine Leserechte auf diesen Speicherbereich haben. Außerdem wird das Programm hier abgebrochen werden.

4. Wir addieren nun das Ergebnis vom Punkt 3 (0x15) zu unserer Adresse (0x1000) und schreiben dort einen beliebigen Wert hin. Auch eine 0 ist vollkommen ausreichend.

5. Obwohl wir die Anweisung 3 nicht ausführen dürfen und wir deshalb sofort von der MMU gestoppt werden, hat die Out of Order Execution des Prozessors die Anweisungen 3 + 4 bereits vollständig ausgeführt. Der Knackpunkt hierbei ist, dass in Anweisung 4 an die Adresse 0x1015 tatsächlich eine 0 geschrieben wurde - und zwar von der Out of Order Execution Einheit des Prozessors. Der Prozessor sorgt zwar quasi im Nachhinein dafür, dass die 0 nicht wirklich an die Speicheradresse geschrieben wird, aber die Speicheradresse wurde aus Performancegründen trotzdem in den Cache geladen.

6. Ich kann nun Stück für Stück die Speicheradressen 0x1000 - 0x10FF auslesen und die Zeit messen, wie schnell das Ergebnis kommt. Bewegt sich die Zeit für den Zugriff zwischen 400 und 500 Prozessorzyklen, so ist das ein normaler Speicherzugriff. Ist die Zugriffszeit aber plötzlich um die 200 Prozessorzyklen schnell, so weiß ich, dass diese Adresse nicht vom Speicher, sondern direkt aus dem Cache kam. Dies passiert tatsächlich bei Adresse 0x1015! Ich kann damit darauf schließen, dass die Out of Order Execution diese Adresse in den Cache geladen hat. Und somit weiß ich, dass in der geheimen Speicherzelle der Wert 0x15 stand, obwohl ich sie eigentlich nicht lesen durfte.

Was ist nun der Unterschied zwischen Spectre und Meltdown?

Spectre trainiert die Branch Prediction Einheit, so dass immer ein Sprung genommen (oder nicht genommen) wird. Wenn dann das Gegenteil erfolgt, kann man sicher sein, dass die Out of Order Execution trotzdem den Programmcode ausführen wird, der bislang immer ausgeführt wurde. Auch hier kann ich somit Programmcode ausführen, den das Programm eigentlich nicht ausführen wollte.

Wenn in diesem Programmcode User Eingaben verarbeitet werden, die ich kontrollieren kann, kann ich hier schon eine ganze Menge geheime Daten aus dem Speicher des Programmes auslesen. Auch hier nutze ich das Messen der Zugriffszeit auf die Speicheradressen, die ich geschickt gewählt habe, um auf diese Daten zuzugreifen.

Das ganze Vorgehen bezeichnet man übrigens als Seitenkanal-Attacke. Das ist quasi so, als würde ich nicht durch die hoch gesicherte Eingangstür gehen, sondern das offen stehende Fenster daneben zum Einbruch nutzen!

Patches der Sicherheitslücken

Was ist die Lösung für Meltdown?

Für Linux ist die erste Lösung der sogenannte KAISER Patch (kernel address isolation to have side-channels efficiently removed). Dieser macht nichts anderes, als für jeden Prozess die Adressen des Betriebssystems (0xC000 0000 - 0xFFFF FFFF) aus der MMU zu entfernen - und zwar jedesmal, wenn dieser Prozess ausgeführt wird. Das kann viele hundert Mal in der Sekunde sein. Dies kostet Performance und macht das System langsamer.

Was ist die Lösung für Spectre?

Hier helfen nur Patches, die den Prozessor betreffen. Prozessoren werden teilweise auch mit sogenanntem Microcode programmiert. Diese Programmierung kann angepasst werden.

Eine kreative Lösung hat hier übrigens der Webbrowser Mozilla Firefox eingebaut - er macht einfach seine Zeitmessungen ungenauer. Damit kann man nicht mehr zwischen einem Zugriff vom Arbeitsspeicher oder Cache unterscheiden. Dass das aber keine dauerhafte Lösung ist, sollte klar sein! Was ist, wenn ich so kurze Zeiträume einmal messen muss?

Was wurde in dieser Erklärung vereinfacht?

Es wird normalerweise beim Testen und Speichern der Adressen nicht +1 gerechnet, sondern +4096, denn der Prozessor cacht nicht einzelne Bytes, sondern immer gleich ganze Seiten. Und eine Seite ist 4096 Bytes groß. Außerdem geht der Adressraum eines Prozessors heutzutage nicht mehr von 0x0000 0000 bis 0xFFFF FFFF (32 Bit), sondern ist auf modernen 64 Bit Betriebssystemen deutlich größer.

Quelle Meltdown: https://meltdownattack.com/meltdown.pdf

Quelle Spectre: https://spectreattack.com/spectre.pdf