MELTDOWN UND SPECTRE

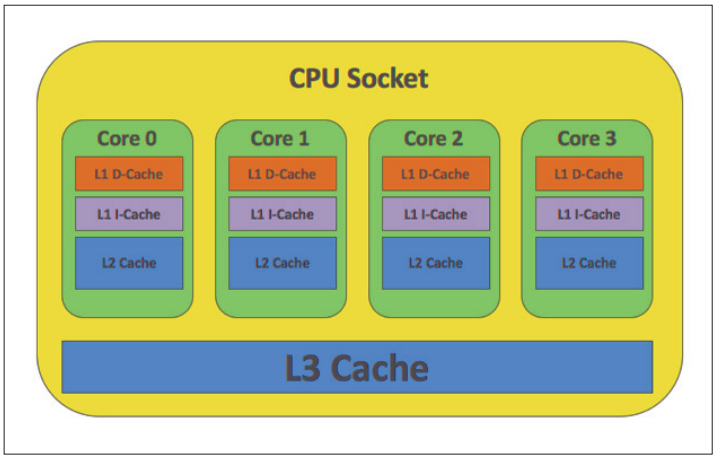
Kernschmelze von Intel-CPUs

So funktioniert das Auslesen von Kernel Memory von einem Anwendungsprogramm aus.

Wie Sie sicher erfahren haben, wurden Anfang des Jahres von IT-Security-Forschern der Technischen Universität Graz zwei der folgenschwersten Fehler in der Architektur von Intel-CPUs der letzten 20 Jahre veröffentlicht.

Durch diese beiden Designfehler – besser bekannt unter den Namen Meltdown und Spectre – ist es aus normalen Benutzerprogrammen heraus möglich, auf geschützte Speicherbereiche des Betriebssystems beziehungsweise auf andere virtuelle Maschinen in einer virtualisierten Umgebung zuzugreifen.

Dieser Artikel soll Ihnen einen technischen Überblick über Meltdown geben und zeigen, wie auf CPU-Ebene über eine sogenannte Side-Channel-Attacke sämtliche Sicherheitsvorkehrungen des Betriebssystems ausgehebelt werden können.



Die verschiedenen Caches einer CPU (Bild 1)

Das Schreckliche an den Bugs, die von Meltdown und Spectre aufgezeigt wurden, ist die Tatsache, dass es sich hierbei um designtechnische Probleme auf CPU-Ebene handelt, die in einer gewissen Art und Weise by design sind.

Das heißt, dass hier Intel für Performance-Vorteile gewisse Abstriche hinsichtlich der Sicherheit bewusst in Kauf genommen hat.

Daher können diese beiden Bugs auch nur effektiv auf Hardware-Ebene gelöst werden.

Glücklicherweise stehen für die Meltdown-Attacke bereits Betriebssystem-Patches von Microsoft, Linux, Apple und VMware zur Verfügung. Inwieweit sich diese Patches auf die Betriebssystem-Performance auswirken werden, kann leider noch nicht gesagt werden, da es aktuell hierfür noch keine Erfahrungswerte gibt (Januar 2018).

Teilweise wird aber spekuliert, dass die Performance-Einbußen durchaus zwischen fünf und 30 Prozent liegen könnten, abhängig von der ausgeführten Workload.

Sehr stark beeinträchtigt können hierbei Datenbank-Anwendungen und Virtualisierungslayer sein (ESXi von VMware und HyperV von Microsoft).

Katastrophaler sieht es mit der Spectre-Attacke aus, da diese sich höchstwahrscheinlich nicht vollständig durch Software-Patches beheben lässt.

Hier kann nur ein kompletter Austausch der CPU vorgenommen werden, der aber logistisch nicht möglich ist, da fast sämtliche CPUs (Intel, AMD, ARM) der letzten 20 Jahre davon betroffen sind.

Des Weiteren ist noch zu erwähnen, dass die Meltdown-Attacke nur auf CPUs durchgeführt werden kann, die von Intel stammen.

Auf der anderen Seite kann jedoch die Spectre-Attacke auch auf AMD- und ARM-Prozessoren durchgeführt werden, das heißt, sie betrifft sämtliche Prozessoren, die Sie in Ihrem täglichen Leben einsetzen – inklusive Ihrer Smartphones und Ihrer Tablets.

Bevor die Details der Meltdown-Attacke nun beleuchtet werden, soll der nächste Abschnitt einen generellen Überblick darüber geben, wie eine CPU aufgebaut ist und wie CPUs Assembly-Befehle ausführen. Das Verständnis dieser Punkte ist eine Grundvoraussetzung, um die Meltdown-Attacke nachvollziehen zu können.

CPU-Architekturen

Vereinfacht gesagt besteht eine CPU aus zwei Komponenten, die für eine Befehlsausführung unbedingt notwendig sind:

- Execution Units
- Register

Execution Units können Sie sich wie das Gehirn der CPU vorstellen, da durch die unterschiedlichen Units die eigentliche Arbeit verrichtet wird, die die CPU ausführen muss. Eine wichtige und auch sehr bekannte Execution Unit ist die ALU – die Arithmetic Logic Unit, welche arithmetische Berechnun-

Instruction	Stage 1	Stage 2	Stage 3
#1	LOAD Register1	*<Empty>*	*<Empty>*
#2	LOAD Register2	LOAD Register1	*<Empty>*
#3	ADD	LOAD Register2	LOAD Register1
#4	STORE Register3	ADD	LOAD Register2
#5	*<Empty>*	STORE Register3	ADD
#6	*<Empty>*	*<Empty>*	STORE Register3
#7	*<Empty>*	*<Empty>*	*<Empty>*

Beispiel für eine Three-Stage-Pipeline (Bild 2)

gen ausführt – zum Beispiel das Addieren und das Subtrahieren von Zahlen.

Die meiste Zeit verbringt die CPU während der Befehlsausführung in der ALU – auch für einfachste Aufgaben wie beispielsweise das Schreiben dieses Artikels.

Neben den verschiedenen Execution Units benötigt die CPU auch noch temporären Speicher, in dem Daten zwischengespeichert werden.

Im Kontext der CPU sind das die sogenannten Register. Eine CPU kann nur dann eine Operation wie das Addieren von zwei Zahlen ausführen, wenn sich die dafür notwendigen Daten in Registern befinden.

Eine CPU kann niemals direkt mit Daten arbeiten, die im Hauptspeicher (RAM) liegen. Für eine CPU ist der Hauptspeicher nichts anderes als ein Storage-Subsystem.

Zuerst müssen also die Daten aus dem Hauptspeicher in Register geladen und nach der Ausführung der Operation wieder in den Hauptspeicher zurückgeschrieben werden.

Stellen Sie sich zum Beispiel vor, dass Sie den folgenden C-Code ausführen möchten, der zwei Zahlen addiert, deren Werte sich aktuell im Hauptspeicher befinden:

```
int c = a + b;
```

Wenn Sie dieses C-Statement in (Pseudo)-Assembly-Code übersetzen, den die CPU ausführen kann, könnte der generierte Code wie folgt aussehen:

```
LOAD Register1 from MemoryLocationA
LOAD Register2 from MemoryLocationB
Register3 = ADD(Register1, Register2)
STORE Register3 in MemoryLocation3
```

Das Problem mit diesem Ansatz ist, dass der Hauptspeicherzugriff sehr langsam ist. Hauptspeicher basiert auf sogenannten DRAM-Speicherzellen, welche beim Zugriff Latenzzeiten von ca. 100 bis 200 Nanosekunden haben. Dadurch verlangsamt jeder Hauptspeicherzugriff Ihr Programm.

Bei einer CPU mit einer Taktrate von einem GHz würde die Ausführung einer Instruktion daher genau eine Nanosekunde dauern.

Aber Sie müssen die ganze Zeit immer wieder auf den Hauptspeicherzugriff warten. Die Codesequenz von vorher würde daher wie folgt auf dieser fiktiven CPU ausgeführt werden:

```
Clock Cycle #1
LOAD Register1 FROM MemoryLocationA
Wartezeit von 100 Clock Cycles...
Clock Cycle #102
LOAD Register FROM MemoryLocationB
Wartezeit von 100 Clock Cycles...
Clock Cycle #203
Register3 = ADD(Register1, Register2)
Keine Wartezeiten auf Hauptspeicher
Clock Cycle #204
STORE Register3 in MemoryLocationC
Wartezeit von 100 Clock Cycles
Clock Cycle #305
Nächster Befehl...
```

Wie Sie anhand dieses einfachen Beispiels erkennen können, dauert die Ausführung der vier Assembly-Befehle ganze 304 Nanosekunden.

Aber von diesen 304 Nanosekunden verbringt die CPU 300 Nanosekunden mit Warten auf den Hauptspeicher. Wie Sie erkennen können, führt dieser einfache Ansatz zu extremsten Performance-Problemen.

Daher müssen sich CPUs cleverer Ansätze bedienen, um diese Performance-Probleme zu vermeiden:

- CPU-Caches
- Pipelining
- Speculative Execution

CPU-Caches und Pipelining

Eine moderne CPU besitzt eine Vielzahl unterschiedlicher Caches, welche auf der einen Seite sehr schnell, aber auf der anderen Seite im Vergleich zum Hauptspeicher sehr klein ▶

Tabelle 1: Die Latenzzeiten der CPU-Caches

Cache	Größe	Latenzzeit
L1 Data Cache	32 KB	4 Taktzyklen
L1 Instruction Cache	32 KB	4 Taktzyklen
L2 Cache	256 KB	10 Taktzyklen
L3 Cache	Bis zu 45 MB (Intel Xeon E7-8880 v3)	40 Taktzyklen

Instruction	Stage 1	Stage 2	Stage 3
#1	DoSomeWork	<Empty>	<Empty>
#2	LOAD Register1	DoSomeWork	<Empty>
#3	LOAD Register2...	LOAD Register1	DoSomeWork
#4	UNDO	LOAD Register2...	LOAD Register1
#5	UNDO	UNDO	LOAD Register2...
#6	<Empty>	<Empty>	UNDO

Die CPU befindet sich wieder in einem konsistenten Zustand (Bild 3)

sind. Bild 1 gibt Ihnen einen Überblick über die verschiedenen CPU-Caches.

Wie Sie erkennen können, wird hier eine komplette Cache-Hierarchie aufgebaut. Je näher der Cache an der CPU ist, desto schneller und desto kleiner ist er. Tabelle 1: Die Latenzzeiten der CPU-Caches gibt einen Überblick über die Größen und die durchschnittlichen Latenzzeiten der unterschiedlichen CPU-Caches.

Wie Sie ebenfalls anhand von Bild 1 erkennen können, erstreckt sich der L3-Cache über sämtliche CPU-Cores eines CPU-Sockets. Das hat auch Auswirkungen und Implikationen in Kombination mit Hyperthreading.

Beim L3-Cache handelt es sich auch um einen sogenannten Last Level Cache, da danach bereits der langsame Zugriff auf den Hauptspeicher erfolgt, wenn die Daten im Cache nicht gefunden wurden.

Die Idee der CPU-Caches ist es nun, dass Daten vom Hauptspeicher, die immer wieder angefordert werden, sich direkt in den CPU-Caches speichern lassen. Dadurch muss nicht immer wieder auf den langsamen Hauptspeicher zugegriffen werden.

Selbstverständlich kann jetzt in einem kleineren CPU-Cache nicht der komplette Hauptspeicherinhalt zwischengespeichert werden.

Beim Zugriff auf Daten, die sich nicht in einem CPU-Cache befinden, handelt es sich dann um einen sogenannten Cache Miss, der schlussendlich wiederum zu einem langsamen Hauptspeicherzugriff führt.

Anhand der CPU-Caches kann nun der Zugriff auf den Hauptspeicher minimiert werden, ist aber trotzdem immer noch notwendig.

Das heißt, dass immer noch sehr lange Wartezeiten aufgrund der hohen Latenzzeit des Hauptspeichers anfallen. Um dieses Problem ebenfalls lösen zu können, besteht eine CPU intern aus sogenannten Pipelines, die parallel CPU-Befehle ausführen können.

Stellen Sie sich zum Beispiel eine sehr einfache CPU mit einer Three-Stage-Pipeline vor (Bild 3). In Wirklichkeit hat eine CPU eine viel größere Anzahl von Stages.

Wenn nun eine CPU, die auf einer Pipeline basiert, einen Befehl ausführt, kann dieser Befehl in mehrere unabhängige Einzelschritte unterteilt werden. Und jede Stage der CPU-Pipeline führt einen solchen Einzelschritt aus. Dadurch können mehrere Befehle innerhalb eines CPU-Cores (Single-Threaded!) gleichzeitig abgearbeitet werden. Je mehr Stages die CPU hat, desto größer ist die erzielte Parallelität.

Solange nun die CPU einen seriellen Stream von CPU-Befehlen zur Abarbeitung hat, ist eine Pipelined CPU höchst effizient. Die Realität schaut aber komplett anders aus.

Tatsächlich werden im Programmcode sehr oft Verzweigungen durchgeführt, die von bestimmten Ausdrücken abhängig sind:

```
if (x > 0)
{
   Instruction Stream #1
}
else
{
   Instruction Stream #2
}
```

Mit welchem Instruction Stream soll nun aber die CPU-Pipeline nach dem *if*-Statement befüllt werden? *Instruction Stream #1* oder *Instruction Stream #2*? Eine CPU kann das vorzeitig nicht wissen, da im ersten Schritt das *if*-Statement abgearbeitet und evaluiert werden muss.

Daher haben sämtliche modernen CPUs einen sogenannten Branch Predictor, der intern in einem sogenannten Branch Target Buffer (BTB) vermerkt, wo in den letzten Ausführungen bei diesem *if*-Statement die Codeausführung fortgeführt wurde.

Intern werden dazu die Hauptspeicheradresse des *if*-Statements und die Ziel-Hauptspeicheradresse des bedingten Sprunges abgelegt.

Der Branch Target Buffer kann ebenfalls im Rahmen der raffinierteren Spectre-Attacke gezielt manipuliert und damit für bösartige Zwecke missbraucht werden.

Normalerweise ist der Branch Predictor äußerst effizient und schätzt mit einer hohen Wahrscheinlichkeit den richtigen Instruction Stream ab. Dadurch kann die CPU-Pipeline bereits vorzeitig Befehle ausführen, ohne auf die Evaluierung des *if*-Statements zu warten.

Hat sich aber der Branch Predictor verschätzt und die CPU-Pipeline mit dem falschen Instruction Stream aufgebaut, wird ein sogenannter Pipeline Flush durchgeführt.

Dadurch wird seitens der CPU Arbeit verworfen, die bereits durch die falsch aufgefüllte Pipeline abgearbeitet wurde.

Ja, Sie haben richtig gelesen: Eine moderne CPU kann spekulativ Befehle ausführen, die zu einem späteren Zeitpunkt (gleich wie eine Datenbanktransaktion) wieder zurückgerollt werden. Schlussendlich ist die CPU wiederum in einem kon-

sistenten Status und es wurden alle Spuren dieser spekulativen Codeausführung beseitigt. Fast ...

Wurden Daten während einer spekulativen Codeausführung vom Hauptspeicher in ein Prozessorregister gelesen, werden diese Register entsprechend zurückgesetzt.

Was aber nicht passiert, ist das Zurücksetzen der CPU-Caches. Das heißt, dass die gelesenen Daten vom Hauptspeicher nach wie vor in den CPU-Caches zwischengespeichert werden!

Das Löschen der CPU-Caches wäre viel zu aufwendig und teuer, da anschließend sämtliche Instruktionen wiederum Hauptspeicherzugriff durchführen müssten, was sich negativ auf die Performance auswirken würde.

Und genau diese CPU-Caches sind nun der Angriffspunkt der Meltdown-Attacke.

Speculative Execution

Der folgende Code zeigt, wo der Angriffspunkt liegt.

```
// Access some protected Kernel Memory
// This triggers an Access Violation Exception!
LOAD Register1, [SomeKernelMemoryAddress]

// This code is never, ever executed! Is it?
LOAD Register2, [BaseArrayAddress + Register1]
```

Dieser Pseudo-Assembly-Code ist ziemlich einfach aufgebaut: Im ersten Schritt wird versucht, auf einen Hauptspeicherbereich des Betriebssystem-Kernels zuzugreifen.

Dieser Zugriff löst natürlich eine Access Violation Exception seitens des Betriebssystems aus.

Nach der Access Violation Exception wird weitere Arbeit verrichtet.

Mit all dem, was wir über Computerprogrammierung wissen, können wir mit Sicherheit behaupten, dass der *LOAD-Register2*-Befehl niemals ausgeführt werden kann, da mit dem Zugriff auf das Kernel Memory eine Access Violation Exception ausgelöst wird.

Der *LOAD-Register2*-Befehl ist nichts anderes als unreachable code. Im Worst Case würde die unbehandelte Access Violation Exception das Programm zum Absturz bringen.

Das Problem ist nur, dass das alles gelogen ist. Die Praxis sieht nämlich anders aus, denn die CPU macht etwas komplett anderes.

Wie bereits erwähnt, muss die CPU ihre Pipeline mit Befehlen auffüllen, damit so viel Arbeit wie möglich parallel abgearbeitet werden kann. Daher wird von der CPU spekulativ auch der *LOAD-Register2*-Befehl nach (!) der Access Violation Exception ausgeführt!

Zu einem späteren Zeitpunkt sieht nun die CPU, dass eine Access Violation Exception ausgelöst wurde, und rollt die spekulativ ausgeführten Befehle einfach zurück. Das heißt, dass das Überprüfen der Access Violation Exception asynchron (!) ausgeführt wird – auf Intel-CPUs.

Andere CPUs, wie zum Beispiel von AMD, machen eine solche Überprüfung synchron. Daher funktioniert die Meltdown-Attacke auch nur auf Intel-CPUs.

Nach dem Zurückrollen der spekulativ ausgeführten Befehle befindet sich die CPU schlussendlich wiederum in einem konsistenten Zustand (Bild 3).

Die CPU hat einfach aufgrund von hardwareseitigen Performance-Optimierungen angenommen, dass keine Access Violation Exception generiert wird, und hat spekulativ mit der Codeausführung weitergemacht, damit die CPU-Pipeline so effektiv wie möglich ausgenutzt werden kann.

Im Prinzip ist das nichts anderes als eine Branch Misprediction des Branch Predictors.

Bevor die spekulative Ausführung gestartet wurde, wurde seitens der CPU ein Checkpoint durchgeführt, das heißt, dass sich die CPU die aktuellen Registerwerte gemerkt hat.

Und anhand dieser Checkpoint-Daten kann die CPU wiederum in einen konsistenten Zustand zurückgebracht werden. Das klingt alles ganz gut.

Das Problem sind allerdings die CPU-Caches. Die Daten, die vom *LOAD-Register2*-Befehl spekulativ geladen wurden, befinden sich jetzt immer noch in den CPU-Caches, obwohl sie niemals angefordert wurden. Sie wurden nur spekulativ angefordert.

Und genau dieses Verhalten führt nun zur ultimativen Kernschmelze bei Intel-CPUs, weil dadurch das komplette physische Kernel Memory anhand des Meltdown-Exploits ausgelesen werden kann.

Kernel Memory Address	String Content
1	K
2	L
3	A
4	U
5	S

Das Layout des fiktiven Prozessors (Bild 4)

Meltdown: Die ultimative Kernschmelze

Wie Sie anhand des vorherigen Abschnitts gesehen haben, kann nun eine CPU Daten von einer geschützten Kernel-Memory-Adresse lesen, und diese gelesenen Daten sind nach wie vor in den CPU-Caches vorhanden, nachdem die spekulative Codeausführung zurückgerollt wurde.

Die CPU bietet Ihnen jedoch in keinster Art und Weise eine Möglichkeit an, die Daten direkt von einem CPU-Cache zu lesen. Diese Möglichkeit besteht auf technischer Ebene zu keinem Zeitpunkt. Das ist schlichtweg unmöglich. Dazu existieren (glücklicherweise) keine entsprechenden Assembly-Befehle. Also alles gut?

Wie können jetzt aber trotzdem geschützte Kernel-Memory-Daten über einen sogenannte Side Channel in eine normale Benutzeranwendung transferiert werden?

Im Prinzip müssen Sie nur ein wenig in die Trickkiste greifen und die Tatsache ausnutzen, dass der Zugriff auf gecachte Hauptspeicherdaten schnell und der Zugriff auf nicht ▶

gecachte Hauptspeicherdaten langsam ist. Ein konkretes Pseudo-Beispiel soll zeigen, wie es geht.

Stellen Sie sich vor, dass geschützte Kernel-Memory-Daten über eine Side-Channel-Attacke in eine Benutzeranwendung übertragen werden sollen.

Um das Beispiel einfach zu halten, wird angenommen, dass das Kernel Memory insgesamt nur aus fünf Byte besteht, die sich an den (fiktiven) Hauptspeicheradressen eins bis fünf befinden. Das Memory-Layout sieht daher wie in Bild 4 aus.

Wie Sie anhand von Bild 4 ebenfalls erkennen können, wird hier der String *KLAUS* im Kernel Memory abgelegt. Jedes Zeichen ist intern in Form eines ASCII-Codes abgebildet und auch über diesen ASCII-Code im Hauptspeicher abgelegt.

Wenn der String anhand einer ASCII-Tabelle zu seinen ASCII-Werten konvertiert wird, sehen die Daten im Kernel Memory wie in Bild 5 aus.

Und nun sehen Sie sich nochmals den bereits zuvor gezeigten Code näher an.

```
// Access some protected Kernel Memory
// This triggers an Access Violation Exception!
LOAD Register1, [SomeKernelMemoryAddress]

// This code is never, ever executed! Is it?
LOAD Register2, [BaseArrayAddress + Register1]
```

Kernel Memory Address	Decimal Content
1	75 -- 'K'
2	76 -- 'L'
3	65 -- 'A'
4	85 -- 'U'
5	83 -- 'S'

Der Name Klaus in Form der ASCII-Entsprechung im Hauptspeicher abgelegt (Bild 5)

Die eckigen Klammern bei der Variablen *[SomeKernelMemoryAddress]* bedeuten, dass hier eine sogenannte Indirect Memory Addressing Operation durchgeführt werden soll.

Der Code sagt der CPU, dass sie die Daten an der Hauptspeicheradresse *[SomeKernelMemoryAddress]* in *Register1* laden soll. Ersetzen Sie zum Beispiel diese Variable mit der ersten Kernel-Memory-Adresse aus folgendem Beispiel:

```
// Access some protected Kernel Memory
// This triggers an Access Violation Exception!
LOAD Register1, [1]
```

Der Code lädt den Wert 75 der Kernel-Memory-Adresse 1 in das *Register1*.

Dieser Hauptspeicherzugriff löst jedoch eine Access Violation Exception aus, da vom User Space nicht auf den Kernel-Space zugegriffen werden darf. Diese security-technische Unterteilung des Hauptspeichers zwischen User Space und Kernel-Space wird der CPU während des Betriebssystemstarts über sogenannte Global Descriptor Tables (GDTs) mitgeteilt.

Wie zuvor erwähnt, wird diese Access Violation Exception jedoch nicht sofort ausgelöst, da sie ja auch durch die CPU-Pipeline abgearbeitet werden muss.

Während dieser Abarbeitung wird jetzt aber bereits spekulativ der nächste Befehl in die CPU-Pipeline gestellt, und es wird damit begonnen, diesen ebenfalls abzuarbeiten:

```
// This code is never, ever executed! Is it?
LOAD Register2, [BaseArrayAddress + Register1]
```

Wie Sie erkennen können, wird hier wiederum eine Indirect Memory Addressing Operation durchgeführt, um *Register2* zu laden.

Dieses Mal wurde jedoch ein Array im User Space deklariert, welches an der Hauptspeicheradresse *BaseArrayAddress* abgelegt wurde.

Nehmen Sie zum Beispiel an, dass das Array im User Space an der (fiktiven) Hauptspeicheradresse 100 liegt. Wenn Sie die Variablen im Code ersetzen, bekommen Sie folgendes Ergebnis:

```
// This code is never, ever reached! Is it?
LOAD Register2, [100 + 75]
```

Wenn wir die Addition schlussendlich durchführen, lautet der Code, der ausgeführt wird, wie folgt:

```
// This code is never, ever reached! Is it?
LOAD Register2, [175]
```

Es wird daher der Speicherinhalt der Hauptspeicheradresse 175 des User Space in das *Register2* geladen. Hierbei handelt es sich um eine komplett andere Hauptspeicheradresse, die auch mit dem Inhalt des Kernel-Space überhaupt nichts gemeinsam hat.

Es wurden auch niemals Daten vom Kernel Memory gelesen, da diese ja durch die Access Violation Exception geschützt sind. Sämtlicher Code ist im User Space ausgeführt worden, aber:

Der Code hat Daten von einer Kernel-Memory-Hauptspeicheradresse als einen Index-Zeiger in ein Array verwendet, welches im User Space abgelegt ist!

Wenn Sie nun versuchen, ein Byte (acht Bits) von einer geschützten Kernel-Memory-Hauptspeicheradresse zu lesen, gibt es insgesamt 256 mögliche Index-Zeiger (2 ^ 8). Daher gibt es auch 256 verschiedene Möglichkeiten, wie *Register2* mit der *LOAD Operation* geladen wird.

```
LOAD Register2, [100 + 0]
LOAD Register2, [100 + 1]
LOAD Register2, [100 + 3]
...
...
```

```
...
LOAD Register2, [100 + 253]
LOAD Register2, [100 + 254]
LOAD Register2, [100 + 255]
```

Wie bereits erwähnt, wurde diese *LOAD Operation* spekulativ ausgeführt. Daher wird diese auch zu einem späteren Zeitpunkt von der CPU wieder zurückgerollt. Es handelt sich ja um eine Art von Branch Misprediction. Nur befinden sich die gelesenen Daten der Kernel-Memory-Hauptspeicheradresse nach wie vor in den CPU-Caches.

Sie können auf einen CPU-Cache zwar nicht direkt zugreifen, es gibt jedoch einen Workaround für diese technische Limitation: Sie können in einem separaten Thread messen, wie lange der Zugriff auf sämtliche 256 möglichen Array-Entries dauert.

Ist der Zugriff langsam, wurden die Daten vom Hauptspeicher gelesen. Ist aber der Zugriff auf einen Array-Entry schnell, wurden die Daten direkt vom CPU-Cache gelesen. Probieren Sie Folgendes aus:

```
LOAD AnotherRegister, [100 + 0]   SLOW
LOAD AnotherRegister, [100 + 1]   SLOW
LOAD AnotherRegister, [100 + 3]   SLOW
...
...
...
LOAD AnotherRegister, [100 + 73]  SLOW
LOAD AnotherRegister, [100 + 74]  SLOW
LOAD AnotherRegister, [100 + 75]  FAST!
LOAD AnotherRegister, [100 + 76]  SLOW
LOAD AnotherRegister, [100 + 77]  SLOW
...
...
...
LOAD AnotherRegister, [100 + 253] SLOW
LOAD AnotherRegister, [100 + 254] SLOW
LOAD AnotherRegister, [100 + 255] SLOW
```

Und siehe da: Der Zugriff auf den Array-Entry an der Stelle 176 ist schnell! Wenn Sie nun die Basis-Hauptspeicheradresse 100 von 175 subtrahieren, bekommen Sie den Wert 75 als Ergebnis. Und der Dezimalwert 75 ist der ASCII-Code des Zeichens *K*.

Gratulation, Sie haben gerade ein geschütztes Byte des Kernel-Space in den User-Space einer Anwendung gelesen – über eine Side-Channel-Attacke! Das ist Meltdown!

Sie haben sich die Tatsache zunutze gemacht, dass der Zugriff auf bereits gecachte Daten schnell und der Zugriff auf nicht gecachte Daten langsam ist.

In Kombination mit einer spekulativen Codeausführung führt das zu heftigsten Sicherheitsproblemen.

Im Grunde genommen können Sie über Meltdown das komplette Hauptspeicherabbild des Kernels in Ihre User-Space-Anwendung laden.

Sie müssen nur den Inhalt jeder möglichen Kernel-Hauptspeicheradresse über eine spekulative Ausführung in den CPU-Cache laden und danach alle 256 möglichen Probe-Operationen gegen das Array ausführen.

Die IT-Security-Forscher der TU Graz konnten hier mit einer Übertragungsrate von bis zu 500 KB/Sekunde den Kernel auslesen. Gar nicht mal so schlecht ... Na ja, eigentlich katastrophal ...

Fazit

Der Jahr-2000-Bug oder der Intel-Pentium-Floating-Point-Bug des Jahres 1994 sind im Vergleich zu Meltdown und der darauf aufbauenden Spectre-Attacke nichts anderes als eine Tüte Peanuts.

Wie Sie anhand dieses Artikels gesehen haben, ist die Meltdown-Attacke darauf zurückzuführen, dass Intel Performance-Aspekte vor Sicherheit gestellt hat. Könnte die spekulative Codeausführung für kritische Codepfade unterdrückt werden, hätte Meltdown niemals so große Wellen geschlagen. Und zusätzlich ist es nicht geschickt, Sicherheitsüberprüfungen wie den Zugriff auf geschützten Kernel-Memory-Space asynchron auszuführen.

Des Weiteren ist es noch wichtig zu erwähnen, dass in diesem Artikel sehr viele Aspekte der grundlegenden CPU-Architektur und des grundlegenden Meltdown-Exploits verallgemeinert wurden, damit es auch möglich ist, diese kritische Sicherheitslücke einfach nachvollziehen zu können.

In der Realität ist es um einiges schwieriger, mit Meltdown an Kernel-Memory-Daten heranzukommen, da auch beim Probing des Arrays sichergestellt werden muss, dass keine anderen Daten fälschlicherweise gecacht wurden. Und auch das Messen der Latenzzeit beim Array-Probing ist keine triviale Angelegenheit, da Sie hier ebenfalls die CPU austricksen müssen.

Wenn Sie an einer vertiefenderen Beschreibung von Meltdown interessiert sind, empfiehlt der Autor Ihnen die Lektüre des offiziellen Whitepapers dazu [1], das auch die Quelle für diesen Artikel war – neben dem Wikipedia-Eintrag [2]. Zusätzlich möchte der Autor Moritz Lipp, Michael Schwarz und Daniel Gruss von der TU Graz seine Hochachtung und seinen Respekt aussprechen, die parallel zu anderen IT-Sicherheitsforschern die Meltdown- und Spectre-Attacken aufgezeigt haben. ■

[1] Whitepaper zu Meltdown,
www.dotnetpro.de/SL1803Meltdown1
[2] Wikipedia zu Meltdown,
www.dotnetpro.de/SL1803Meltdown2

Klaus Aschenbrenner
berät Unternehmen in Europa beim Einsatz des Microsoft SQL Server und beschäftigt sich mit Windows-Programmierung und .NET. Zweimal zeichnete ihn Microsoft für sein Engagement als Microsoft MVP aus. Er ist zu erreichen über **www.SQLpassion.at**

dnpCode A1803Meltdown