digital whisper
Technologic Papers

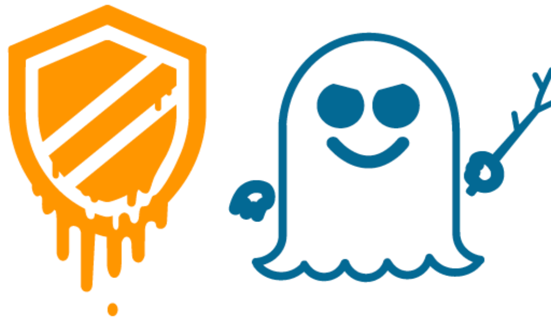
Meltdown & Spectre

מאת דור דנקנר (Ddorda)

הקדמה

בשבועות האחרונים נוצר הייפ מטורף סביב שתי חולשות חדשות שהתפרסמו:

Meltdown (CVE20170-5754) ו-**Spectre** (CVE-2017-5753 + CVE-2017-5715)



ולא בכדי - לא בכל יום מתפרסמת חולשה חומרתית שרלוונטית לרוב המעבדים שנוצרו בעשור האחרון. לתיקון החולשות מחיר כבד: החלפת כל מעבדי המחשבים בחברה עשוי להגיע לסכומי עתק, בעוד שתיקון תוכנתי יוריד את ביצועי המחשב בכ-30%. ובגלל מחירו הכבד, כנראה שהחולשות ימשיכו להיות רלוונטיות גם בעשרות השנים הקרובות.

במהלך השבועיים האחרונים כחלק מהעבודה שלי בחברת SentinelOne למדתי את הנושא לעומק יחד עם רן בן שטרית, כדי למצוא פתרון לחולשה, שיאפשר ללקוחות גם לא לקנות מעבדים חדשים וגם לא לאבד משאבים בעקבות עדכון קרנל.

בסיום המחקר הרגשתי שזו חובה לשתף ולחלוק את הידע שנצבר, גם כיוון שזה נושא בוער בתקופה האחרונה, אבל בעיקר כיוון שמצאתי שמדובר בנושא מרתק, ששבר לי הרבה מהנחות היסוד שהיו לי לגבי דרך פעולת המעבד.

במאמר זה אשתדל להסביר באופן טכני כיצד החולשות והמנגנונים המאפשרים אותן עובדים.

לתחושתי כיוון שהשמות Meltdown ו-Spectre תמיד מגיעים יחדיו, נוצר המון בלבול סביב הנושא - מהו Spectre ומהו Meltdown ומה הם ההבדלים בינהם.

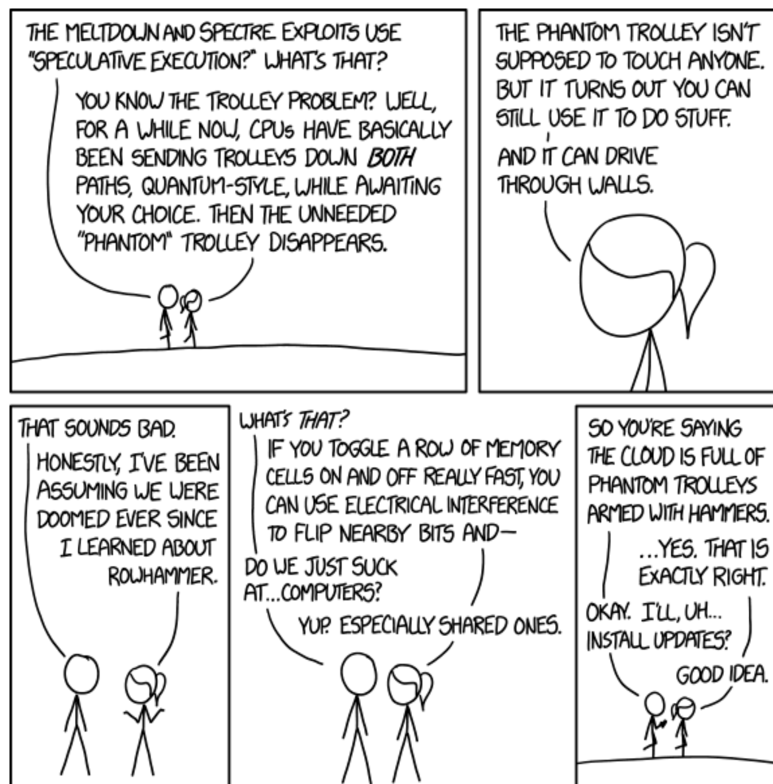
חלק ממטרת המאמר הינה לעזור ולהפיג את הבלבול שנוצר סביב ההבדלים בין החולשות, ולהנגיש את המידע הזה עבור הקהל הישראלי.

כמו-כן, חשוב לי להדגיש שיש פינות שאעגל על מנת להקל על ההבנה. אתם מוזמנים להמשיך לקרוא במקורות שצירפתי, באינטרנט או לשאול אותי באופן פרטי ואשמח להרחיב. אני משתדל לרקוד פה על שתי חתונות: מצד אחד להעביר חומר מאוד טכנולוגי שהפן המורכב שלו הוא מה שמעניין, ומצד שני להסביר את הנושא גם לאנשים שלא שוחים בחומר.

בכל אופן, במאמר השתדלתי להתאים למגוון קהלים, ולכן אם אתם מרגישים שאתם צריכים לקרוא עוד אני מזמין אתכם בחום להמשיך ולחקור על זה. מקום טוב להתחיל יהיה בפרסום של zero project, משם גם נלקחו חלק מדוגמאות הקוד במאמר הזה, או אפילו יותר טוב, מומלץ במיוחד לקרוא ישירות מן המחקרים שהתפרסמו, וראוי לציין שכתובים בצורה מאוד ברורה: המחקר של Meltdown והמחקר של Spectre.

בנוסף חשוב לי לציין שכדי להבין את נושא המאמר לעומק, חובה להכיר מונחים בסיסיים במבנה המעבד ולדעת אסמבלי במידה בסיסית. אם יש מונח שאני זורק כאן ואתם לא מכירים, אל תתביישו לעצור רגע וללכת לקרוא על המונח החדש...

והכי חשוב - תהנו מהקריאה! (:



[Meltdown and Spectre, מתוך XKCD האגדי]

Spectre: Branches speculations

Spectre הינה דוגמה אדירה למתקפת Side Channel Attack מוצלחת במיוחד. מתקפת Side Channel (או התקפת ערוץ צדדי, אם תרצו) היא מתקפה שמוכרת בעיקר מעולם הקריפטוגרפיה, בה לא תוקפים את המידע ישירות (הרבה פעמים כיוון שפשוט לא ניתן), אלא תוקפים מידע שמושג בעקבות האלגוריתם עצמו.

כדי להבין לעומק כיצד החולשה עובדת, צריך להבין כיצד המעבד עובד בכל הנוגע ליישום וקבלת החלטות.

בפנינו הקטע קוד הבא:

```
if (15 == a * b)
      c = 4;
else
      c = 93;
```

כולנו יודעים איך המעבד עובד, ולכן כולנו יודעים שבקטע הנ"ל המעבד יעשה את רצף הפעולות הבאות:

א) יחשב את a * b

ב) ישווה את התוצאה ל-15

ג) יקפוץ לקטע הקוד הבא בהתאמה

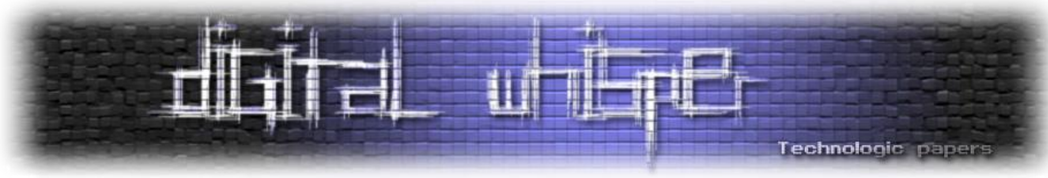
ד) יעשה השמה ל-c בהתאמה.

אבל האמת היא, שזה שקר גמור. על פני השטח זה אכן מה שהמעבד יעשה, אך מאחורי הקלעים עומד מנגנון מטורף לגמרי שנקרא branch speculation שמטרתו להאיץ את פעילות המעבד על ידי ניצול כל cycle אפשרי.

מה שבפועל קורה הוא שלפני שכל הקטע הזה רץ (א' - ד'), המעבד כבר "עשה ספקולציות" וניסה לנחש איזה אפשרויות סבירות שניכנס אליהם. לצורך הפשטות נניח פה שהוא החליט ששתי האפשרויות סבירות בהחלט, ולכן הוא נכנס לשניהם (!) מה שעכשיו מותיר אותנו בסיטואציה בה c שווה גם ל-4 וגם ל-93(!!). כלומר שהמעבד יצא מה-true execution path לשני branch-ים של שתי האפשרויות.

המנגנון הזה למעשה מנצל את העובדה שישנן "בועות" של cycle-ים לא מנוצלים, ומנצל את הזמן המת בכדי לחשב כל נתונים שבשלב הנוכחי לא רלוונטיים עדיין ל-Flow הריצה הליניארי (ה- True Execution Path), אבל יהיו רלוונטיים בהמשך. המעבד מנסה לנחש איזה קטע קוד הולך לרוץ ואיזה מידע אנחנו נרצה להשתמש, ומבצע את הפעולות לפני זמן הריצה האמיתי שלהן.

כשהמעבד יגיע אל ה-if, הוא יוכל לאמת איזה branch הוא הנכון מבין האפשרויות, למזג את ה-branch עם ה-execution path - ואת שאר ה-branch-ים לזרוק. הפעולה הזו נקראת Retire.



עד כאן, מגניב מאוד. בואו ונסתכל גם על קטע הקוד הבא:

```
struct array {
       unsigned long length;
       unsigned char data[];
};
struct array *arr1 = ...;
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
       unsigned char value = arr1->data[untrusted_offset_from_caller];
       ...
}
```

בקוד הנ"ל אנחנו מייצרים מערך תווים כלשהו, ואנחנו מקבלים משתנה בגודל לא ידוע שמייצג offset.

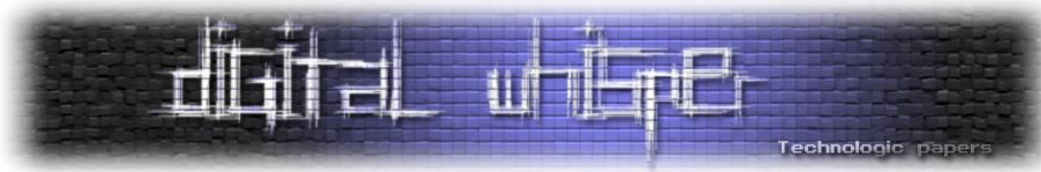
במידה ש-arr1->length לא נמצא ב-cache של המעבד, הוא לא יוכל לדעת בשלב הספקולציות אם הוא הולך להיכנס אל התנאי או לא, ולכן הוא באופן ספקולטיבי ייכנס אל הבלוק של התנאי ויקרא את arr1->data[untrusted_offset_from_caller]. כמובן שכמו שמרמז שם המשתנה, ה-offset עשוי להיות יותר גדול מ-arr1->length, מה שיגרום לכך שבתוך הספקולציה אנחנו ניגש לכתובת שאנחנו לא אמורים להיות מסוגלים לגשת אליה. כמובן שאין עם כך כל בעיה, כיוון שכשנגיע לשלב ה-retire אנחנו לעולם לא ניכנס למצב כזה, והמידע הזה לא יהיה נגיש לנו. אם מבחינת הקוד כן היה ניתן לגשת לשם, היינו מקבלים Segmentation Fault מה-MMU שיגן מפני גישה לא חוקית.

```
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
     unsigned char value = arr1->data[untrusted_offset_from_caller];
     unsigned long index2 = ((value&1)*0x100)+0x200;
     if (index2 < arr2->length) {
         unsigned char value2 = arr2->data[index2];
     }
}
```

במקרה הזה, במידה ש-arr1->length לא נמצא ב-cache, המעבד באופן ספקולטיבי ייכנס לשני ה-if-ים, כמו בדוגמה הקודמת יקרא את הערך של arr1->data[untrusted_offset_from_caller], בהתאם לערך של untrusted_offset_from_caller (שכן נמצא ב-cache כאמור) יחשב את index2, שעשוי להיות 0x220 או 0x300, ויגדיר את value2 בהתאם.

פה הקסם מתחיל:

פעולת ההשמה של value2, שמתרחשת ב-branch גורמת לערך של arr2->data[index2] להיכנס אל ה-cache. שוב, תאורטית לא אמורה להיות בעיה, שהרי אנחנו לא יכולים לגשת אל המידע הזה ב-execution path.



מה אנחנו כן יכולים לעשות?

כמו שאמרנו לפני, index2 יכול להיות רק אחד משני ערכים: 0x200 או 0x300. אנחנו יכולים לגשת אל שתי האפשרויות הבאות: **arr2->data[0x200]** ו-**arr2->data[0x300]**. אחת מהכתובות האלה נכנסה אל ה-L1 Cache בשלב הספקולטיבי, ולכן הגישה אל אותה אופציה אמורה להיות מהירה באופן **משמעותי** מאשר האפשרות השנייה. גישה אל ה-L1 Cache אמורה לקחת באיזור 5 clock cycles ולעומת זאת גישה אל ה-DRAM אמורה לקחת באיזור 500 clock cycles.

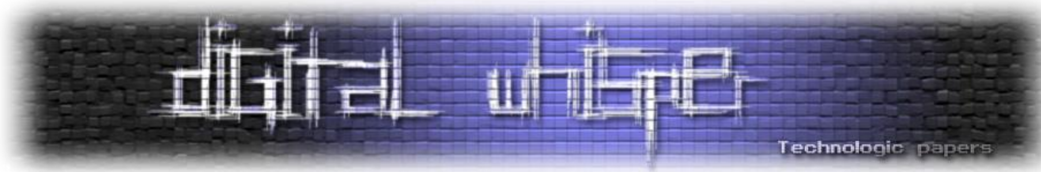
אם ניגש אל שתי הכתובות האפשריות, נמדוד את הזמן שלקח לגשת אל הכתובת, ונשווה בין שתי האפשרויות נוכל להבין איזה כתובת נמצאת כבר ב-cache ומתוך כך להסיק האם value&1 שווה ל-0 או ל-1.

בצורה זו נוכל לרוץ על כל הביטים של התוכן שאנחנו רוצים לקרוא. בשיטה זו למעשה נוכל לקרוא כל חלק שנרצה מזיכרון של תוכנה, ובכך להוציא פרטים שאנו לא אמורים להיות חשופים אליהם כמו לדוגמה פרטים אישיים או סיסמאות.

החוקרים נתנו אף דוגמאות עוצמתיות ביותר לשימושים אפשריים לחולשה, כמו לדוגמה שימוש ב-JavaScript בכדי לאסוף נתונים מהדפדפן של גולשי האתר, החל מ-Cookies, סיסמאות, גרסת דפדפן, מערכת הפעלה וכו'

דוגמה נוספת שהציגו במאמר, היא ניצול של מנגנון JIT נוסף, שנמצא בלינוקס, בשם eBPF, כדי להריץ את החולשה בקרנל ובכך להגדיל את טווח התקיפה מתהליך בודד לכלל מערכת ההפעלה.

החיסרון העיקרי והבולט ביותר של החולשה לעומת Meltdown (עליה - בהמשך), הם שאנחנו מוגבלים לאיזור זיכרון של התהליך שלנו (כאמור, למעט אם מצליחים להריץ את החולשה בקרנל). עם זאת, אין ספק שמדובר מחולשה עוצמתית ומפחידה.



Meltdown: Melting the walls between user and kernel space

החולשה השנייה, Meltdown, נחשבת לחולשה העוצמתית יותר מבין השתיים. היא מורכבת יותר ודורשת הבנה יותר עמוקה של דרך הפעולה של המעבד, וגם ה-impact שלה חזק יותר, בכך שהיא שוברת את ההפרדה בין User Space לבין Kernel Space, ולמעשה מאפשרת לנו לקרוא כל קטע קוד שנרצה מהזיכרון, לא משנה איפה הוא נמצא.

למעשה, בפרסום החולשה החוקרים התגאו בכך שבמהלך הבדיקות שלהם, הם הצליחו לעשות dump מלא של הקרנל, בקצב לא פחות ממדהים של 503 ק"ב לשנייה (!!).

תאוריה

הקונספט הראשון שאנחנו צריכים להכיר כדי להבין איך Meltdown עובד, נקרא Instruction Pipelining.

אזהרה - המוח שלכם עומד להתפוצץ.

נתחיל משאלה רעיונית: נניח שלוקח למכונת הכביסה שלכם 30 דק' להרצה, למייבש לוקח עוד 40 דק' ולקפל את הכביסה לוקח לכם עוד 20 דק'. סה"כ - 90 דק' לתהליך כביסה מלא.

כעת, רצה הגורל ואתם גדלתם בבית של שני הורים וששה ילדים, לכן יש לכם הרבה מאוד כביסה. נאמר, 8 סטים של כביסה בשבוע. כמה זמן היה לוקח לכם לטפל בכל הכביסה? התשובה פשוטה, לא? 90 דק' כפול 8 סטים, סה"כ 720 דק', שזה 12 שעות רצוף של עבודה. הגיוני נכון?

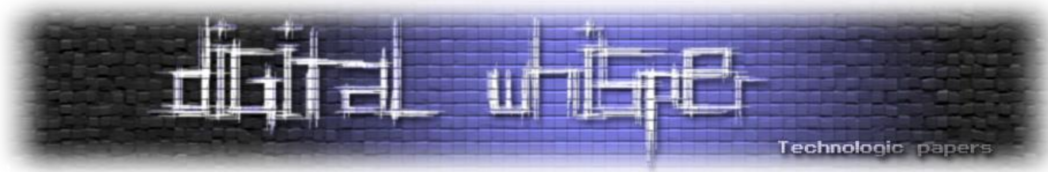
אז אולי ברגע הראשון, אבל כנראה אחרי פעם אחת שתעבדו מ-6 בבוקר עד 6 בערב על כביסה, אתם תרצו להתייעל, ולכן מה שתעשו, הוא שבזמן שהמייבש ייבש את הסט הראשון, מכונת הכביסה תכבס את הסט השני, ובזמן שהמייבש ייבש את הסט השני, אתם כבר תקפלו את הסט השלישי.

בצורה הזו מ-12 שעות, אנחנו מצליחים לרדת ל-300 דק'! 5 שעות!! פחות מחצי מהזמן שעשינו בפעם הראשונה! כיף גדול לכל ששת הילדים שעסוקים מבוקר עד ליל במטלות הבית ☺

חשוב לציין שבתהליך לא צמצמנו את הזמן שלוקח לטפל בכל סט כביסה, אלא צמצמנו את הזמן הממושך שלוקח למספר רב של סטים לרוץ, בכך שאנחנו מנצלים בכל רגע נתון את הכמות המקסימלית של המשאבים שלנו.

לתהליך הזו קוראים pipelining. ואת אותו תהליך בדיוק ניתן לעשות עם הפעולות במחשב, כל עוד ניתן להפריד את המשאבים השונים. אבל במקום מכונת כביסה ומייבש, אנו מנצלים את המשאבים הבאים (בצורה מאוד מופשטת):

- Load - לקריאת מידע מהזיכרון/אוגרים. חשוב לציין שיש שני משאבים כאלה.
- Execute - להרצת פעולות חישוביות.
- Store - לשמירת מידע בזיכרון/אוגרים.



אלגוריתם ה-pipelining שמשתמשים היום במרבית המעבדים נקרא Tomasulo algorithm, והוא ממש פועל באותה שיטה, עם עוד כמה שיפורים קטנים:

טומסולו הבין שבמחשב אין רק חלוקה בין משאבים. הרבה פעמים בין פעולות לוגיות שונות יש תלות מסוימת. אם רק נפריד למשאבים אנו עשויים לעשות פעולות שיפגעו בפעולות אחרות עתידיות. הוא פתר את הבעיה הזו בכל שהוסיף מנגנון נוסף בשם reorder buffer:

לאחר פרסור ה-Instruction, כל פעולה מתחלקת למיקרו-פעולות, לפי שימוש במשאבים, ולאחר מכן, המיקרו-פעולות מסודרות מחדש בתור (ה-reorder buffer), שם הן ממתינות עד שכל התלויות שלהן מבוצעות, ורק אז המיקרו-פעולות מתבצעות. בכל פעם שמיקרו-פעולה מסיימת, ערך ההחזרה שלה או ה-exception status שלה מוחזר ונשמר ברשומה הרלוונטית ב-reorder buffer, ובצורה זו פותר את התלויות הבאות בתור.

ברגע שכל מיקרו-הפעולות של Instruction הסתיימו בהצלחה ה-Instruction מתחיל תהליך שנקרא retirement (זוכרים?): האוגרים ושאר המשאבים הרלוונטיים יעודכנו בהתאם. אך, **אם בעקבות אחת מהמיקרו-פעולות נוצר exception, ה-interrupt ייזרק רק בשלב ה-retirement - לפעמים זמן רב לאחר שהמיקרו-פעולה הרלוונטית התבצעה (!!)**.

בנוסף - כאשר נוצר interrupt, כל ה-reorder buffer מתאפס לחלוטין, כלומר שבמידה שיש עוד מיקרו-פעולות שמחכות לביצוע, הן חודלות לחלוטין, ואנחנו חוזרים לשלב פירסור ה-Instructions.

נראה לי בשלב הזה חשוב לציין עבור מי שמתכוון להמשיך ולקרוא על הנושא, למה שתיארתי הרגע קוראים בשלל מקורות המידע גם out-of-order execution. אם תקראו את המונח הזה - תדעו על מה מדובר.

די עם התאוריה, בואו נגיע כבר לתכל'ס

יש בפנינו את ה-instruction הבא, שרץ ב-usermode:

```
mov rax, [some_kernel_mode_addr]
```

מה אנחנו יודעים להגיד על הפעולה הזאת בשלב הזה?

אנחנו יודעים להגיד שהיא תפורק למיקרו-פעולות, שיחכו ב-reorder buffer וירוצו לפי התלויות שלהם. כמובן שבשלב ה-retire ייזרק interrupt מסוג segmentation fault שיווצר עקב ניסיון הגישה אל הקרנל ספייס.

אנחנו יודעים גם להגיד שבעקבות ה-interrupt ה-reorder buffer יתאפס לקראת ה-instructions הבאים שצריכים להתפרסר.

מצוין. בואו נמשיך:

```
mov rax, [some_kernel_mode_addr]
mov rbx, [some_user_mode_addr]
```

כיוון שישנם משאבים עבור הפעולה load, אנחנו יודעים להגיד ששתי הפעולות הולכות לרוץ במקביל. הפעולה הראשונה הולכת לזרוק interrupt כמו בדוגמה הקודמת, אך בינתיים הפעולה הראשונה תספיק ככל הנראה לקרוא את המידע ולשמור אותו ב-Cache.

ה-interrupt שנוצר יגרום לאיפוס ה-reorder buffer, וה-instruction השני יהיה מוכרח לרוץ שוב כדי לשמור על True Execution Path תקין. למזלנו כיוון שהמידע כבר נשמר ב-cache, המחיר הוא מאוד זול.

נמשיך לדוגמה שלישית:

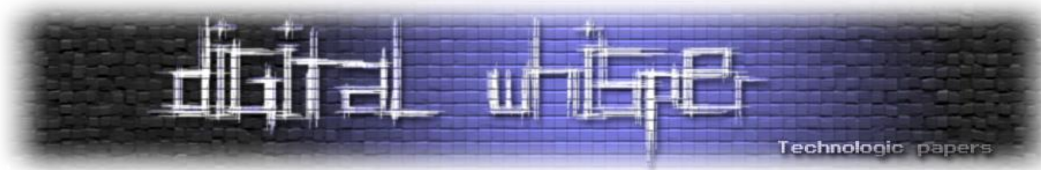
```
mov rax, [some_kernel_mode_addr]
and rax, 1
mov rbx, [rax+some_user_mode_addr]
```

פה זה כבר מתחיל להיות ממש מעניין.

אנחנו רוצים ששתי הפעולות האחרונות ירוצו לפני ה-retire של ה-instruction הראשון. מה עושים? על מנת לפתור את הבעיה הזו מה שניתן לעשות הוא למלא את ה-reorder buffer בפעולות עם הרבה תלויות מצד אחד ומצד שני שלא ישתמשו במשאבים שאנחנו צריכים עבור הריצה של שתי הפעולות האחרונות.

לדוגמה, נוכל לשים המון add rax,0xdd, שיתפוס את המשאב של כתיבה אל rax.

בצורה זו ה-instruction הראשון יתעכב ב-reorder buffer בזמן שהמעבד יריץ את שתי הפעולות האחרות בצורה ספקולטיבית.



כשה-instruction הראשון יעשה retire (יפרוש...?), אכן ייזרק interrupt בשל הניסיון לגשת לאיזור קרנלי, כמו בדוגמאות הקודמות, אך בגלל שהריצה שלו לקחה כל כך הרבה זמן, הגישה כבר התבצעה ולמרות שהמידע לא נשמר באוגרים, המידע כן נמצא ב-**L1 cache** (!).

כעת, אנחנו יכולים להשתמש ב-Side Channel Attack על ה-Cache, כמו שלמדנו על Spectre כדי למשוך את המידע הזה מה-Cache.

Veni, Vidi, Vici.

[לינק ל-POC מוצלח ב-GitHub]

סיכום

שתי החולשות החדשות שהתפרסמו, מראות לנו את העוצמה הפוטנציאלית שטמונה ב- Side Channel Attacks ובחולשות מעבד. אני משוכנע שבעקבות החולשות שהתפרסמו המחקר סביב הנושאים האלו הולך להעלות הילוך, ובוודאי שנראה חולשות דומות בעתיד.

אני מקווה מאוד שהצלחתי לעמוד במטרות שהצבתי לעצמי בעת כתיבת המאמר, ובאמת הצלחתי להסביר את ההבדלים בין Spectre לבין Meltdown, ובאופן כללי לחשוף אתכם ואתכן לעולם המופלא של ספקולציות מעבד.

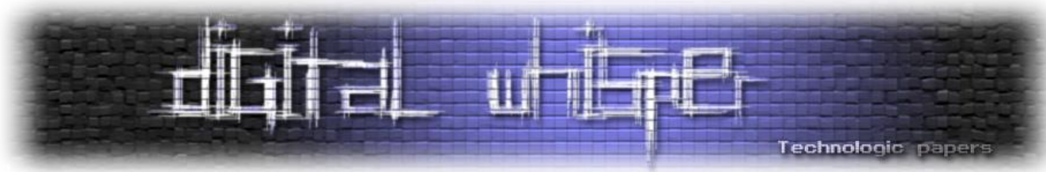
לכל שאלה:

http://ddorda.net/contact

מעבר לכך, חשוב לי להגיד כמה מילות תודה.

- אני אולי זה שכתב את המאמר, אך כמו שאתם מבינים כדי לכתוב אותו הייתי צריך להבין כל מני נושאים מורכבים מאוד. אין ספק שהיה לי לעונג לעשות את כל המחקר והלימוד שלי עם ה-coworker התותח שלי: **רן בן-שטרית**, חוקר ומפתח בחסד.
- את המחקר והלימוד עשיתי במסגרת פרוייקט של החברה הכי טובה שאני מכיר בישראל - SentinelOne. מוזמנים לבוא לעבוד איתי :)
- תודה להורים שלי ולחמשת אחיי ואחותי, בלעדיהם לא הייתי יודע מה הדרך היעילה ביותר לטפל בכביסה.

Dor Aya Po!



מקורות

https://googleprojectzero.blogspot.co.il/2018/01/reading-privileged-memory-with-side.html

https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/

https://spectreattack.com/spectre.pdf

https://meltdownattack.com/meltdown.pdf

https://simple.wikipedia.org/wiki/Instruction_pipelining

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html

https://en.wikipedia.org/wiki/Tomasulo_algorithm