Technologic Papers

עקיפת מחסנית הצללים (RFG) של Microsoft

מאת אייל איטקין

הקדמה

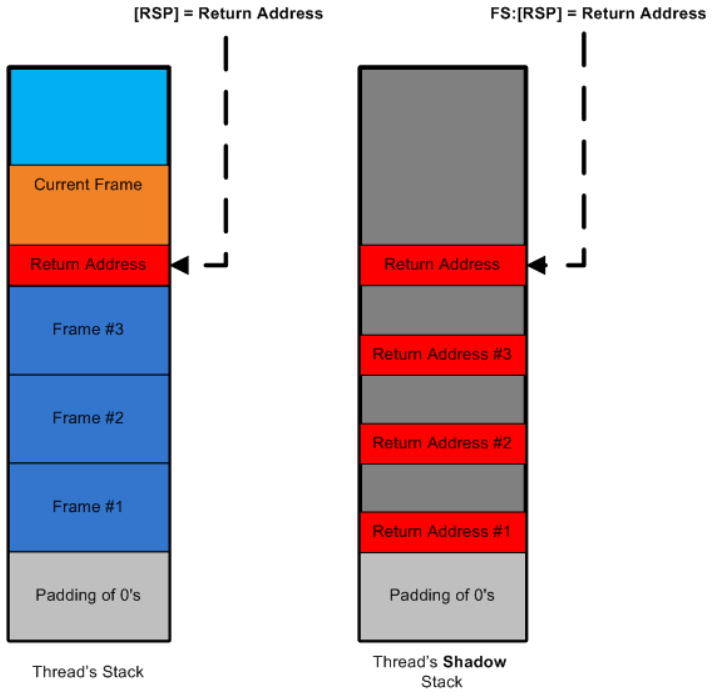
בסוף שנת 2016, בזמן שבדקתי האם יש עדכונים כלשהם לתוכנית ה-Bug Bounty של Microsoft (Mitigation Bypass and Bounty for Defense), הבחנתי באזכור למנגנון הגנה חדש בשם "Return Flow Guard" או "RFG" בקצרה. מכיוון שבאותו הזמן בדיוק סיימתי פרויקט קודם (הצעה הגנתית לשיפור CFG) כחלק מאותה תוכנית Bug Bounty, החלטתי לנסות ולבדוק אם אוכל לעקוף את מנגנון ההגנה החדש. מאמר זה יתאר את התקיפה שלי על מנגנון ה-RFG, תקיפה שמשיגה **עקיפה מלאה** של המנגנון.

הצגת המטרה - Return Flow Guard

הצעד הראשון במחקר היה לאסוף כמה שיותר מידע אודות מנגנון ההגנה המסתורי של Microsoft. חיפוש מהיר באינטרנט הוביל אותי למאמר הטכני המצויין הזה, שנכתב על ידי Tencent Xuanwu Lab. המסקנה הראשונה הייתה כי RFG הוא למעשה מימוש תוכנתי מקביל למנגנון הגנת המחסנית החומרתי של Intel, אשר טרם יצא לשוק. על רגל אחת, על ידי שימוש במחסנית כפולה, אשר לרוב תיקרא גם "מחסנית צללים", קוד מיוחד בסיומה של כל פונקציה יוכל להשוות בין ערך החזרה במחסנית המקורית לזה ששמור במחסנית הצללים, זאת במטרה לחסום תקיפות הבאות לשנות את כתובת החזרה השמורה במחסנית.

הערה: לשם הפשטות, מאמר זה הולך להתייחס אך ורק לארכיטקטורות של 64 ביט. יש לשים לב כי מעבדי אינטל בארכיטקטורת 64 ביט, משתמשים לרוב ברגיסטרים מורחבים אשר שמם מתחיל ב-R (בשביל לציין גישה לרגיסטר מורחב בגודל 64 ביט) במקום ב-E (אשר מציין גישה לחצי התחתון, בגודל 32 ביט).

האיור הבא מתאר את המימוש התוכנתי של Microsoft למנגנון מחסנית הצללים:



כלומר, הגישה למחסנית הצללים נעשית על ידי רגיסטר המחסנית (RSP) ועל ידי שימוש בסגמנט FS. בצורה זו ניתן לבצע גישות מהירות ויעילות למחסנית מצד אחד, ומהצד השני אין צורך להחזיק מצביע למחסנית, דבר שעלול להסגיר את מיקומה לתוקף.

תיאור תרחיש התקיפה

מנגנון ההגנה החדש של Microsoft נועד להגן על המחסנית מפני תקיפות, ובכך הוא משלים את מנגנון ההגנה Control Flow Guard (CFG) שנועד לאכוף את תקינות ריצת התוכנית בכל הקשור למצביעים לפונקציות. על כן, תרחיש התקיפה שנתאר במאמר זה הוא תרחיש של אויב שהשיג שליטה על "מכונה וירטואלית" בתוך התהליך הנתקף. לדוגמא, תוקף שניצל חולשת Flash או Javascript בדפדפן, וכעת באמצעות פרימיטיבים של Read-Where ו-Write-What-Where מנסה "לברוח" מהמכונה שמריצה את פקודות השפה, ולהשיג השתלטות מלאה על התהליך הנתקף.

תרחיש זה דומה לתרחיש התקיפה שתיארתי במאמר הקודם שעסק בבריחה ממכונה וירטואלית של MRuby.

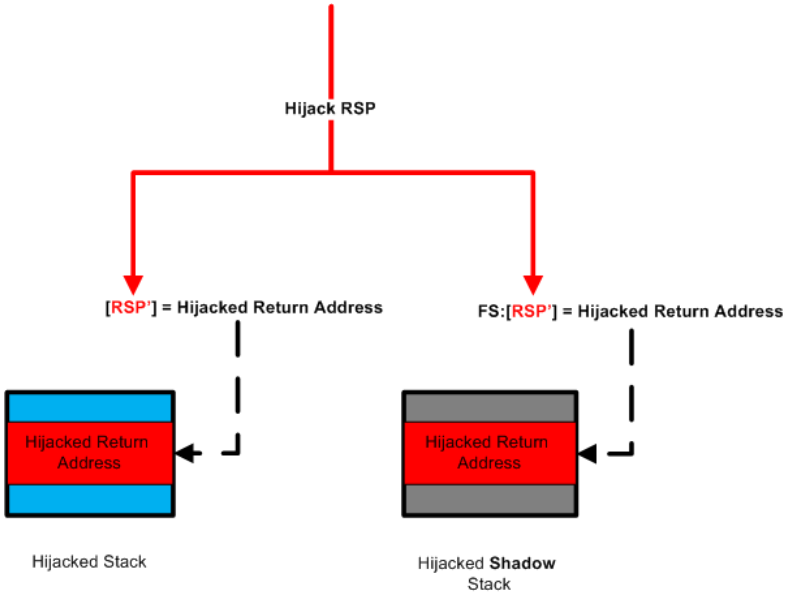
תוכנית תקיפה מס' 1 - השתלטות על RSP

הדבר הראשון שקופץ לעין בעיצוב של Microsoft הוא העובדה כי RSP, מצביע המחסנית, מתפקד למעשה כמצביע ל**שתי** המחסניות (כפי שמוצג באיור הקודם). בנוסף, נשים לב כי בארכיטקטורות של 32 ביט נהוג להשתמש בתבנית האסמבלי השכיחה הבאה:

- תחילת הפונקציה:
  - שמירת EBP למחסנית
  - ביצוע ההשמה EBP := ESP

- סיומת הפונקציה:
  - ביצוע השחזור ESP := EBP
  - טעינת EBP מהמחסנית

לכן, דריסת זכרון המחסנית יכולה לשנות את ערכו השמור של EBP, שבתורו ישפיע על ערכו של ESP של הפונקציה הקוראת. דריסה שכזו תוכל לאפשר את התקיפה הבאה:

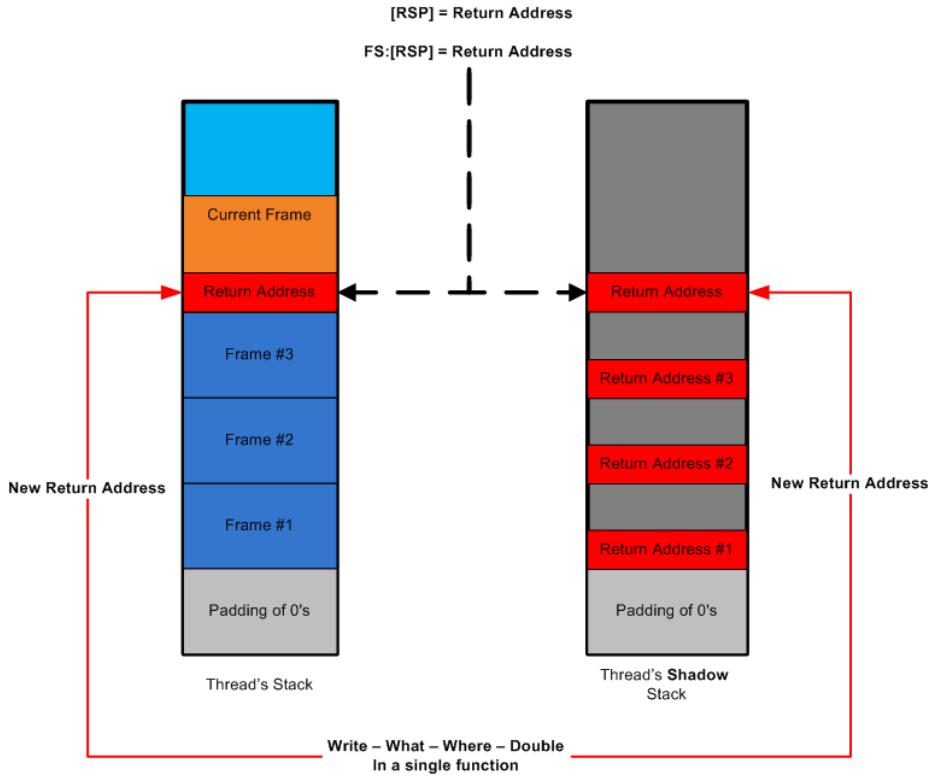


למרבה הצער, בארכיטקטורות של 64 ביט, התבנית הנ"ל היא די נדירה. הקומפיילר מעדיף לבצע גישות ישירות מהסגנון "SS:[RSP + X]" על מנת לגשת למשתנים על המחסנית, כאשר סיומת הפונקציה פשוט תוסיף חזרה ל-RSP גודל התואם לסך הזכרון שהוקצה על המחסנית בתחילת הפונקציה. ללא השימוש ב-"base pointer", יהיה משמעותית קשה יותר "לחטוף" את RSP, ולכן נאלץ לחשוב על תוכנית טובה יותר.

תוכנית תקיפה מס' 2 - דריסה כפולה (Controlled Pair)

חסרון משמעותי במנגנון ההגנה של Microsoft, בהשוואה למנגנון החומרתי של Intel, נובע מכך שהמימוש התוכנתי מחייב את שמירת מחסנית הצללים במרחב הכתובות של ה-User. זאת משום ששמירת המחסנית ב-Kernel תהיה איטית ולא תעמוד בסטנדרטי הביצועים הנדרשים ממנגנון הגנה. בהיעדר תמיכה חומרתית מיוחדת, משמעות אפיון זה היא ש**לכל פיסת קוד** שרצה בתהליך **יהיו הרשאות קריאה וכתיבה** (RW) **למחסנית הצללים**.

היות וזכרון מחסנית הצללים הוא כתיב, נוכל לנסות ולבצע את התקיפה הבאה:



אם נוכל למצוא פרימיטיב מסוג "Write-What-Where-Double", כלומר דריסה נשלטת כפולה (Controlled Pair) מתוך **קריאה לפונקציה בודדת**, הרי שנוכל לדרוס **סימולטנית** את **שני** ערכי החזרה, ובכך נעקוף את הבדיקות בסיומה של הפונקציה.

תוכנית זו דורשת שנעמוד ב-3 יעדים:

1. מציאת המחסנית (ה"רגילה") במרחב הזכרון
2. מציאת מחסנית הצללים במרחב הזכרון
3. מציאת פרימטיב ניצול מסוג Controlled Pair

כפי שהזכרתי קודם לכן, Microsoft הקפידו שלא יהיו מצביעים אל מחסנית הצללים במרחב הזכרון של המשתמש. כדי להתגבר על מגבלה זו, נראה כעת כיצד ניתן לאתר **כל** דף זכרון רצוי, בצורה יעילה ואמינה גם ב-Windows וגם ב-Linux.

שיעור בחשבון - ספירת ביטים

מערכות הפעלה בארכיטקטורה של 64 ביט מגדירות מרחב זכרון וירטואלי שלעיתים נראה כמעט אינסופי, ובוודאי שאינו נראה כמרחב קטן מספיק למנייה בפרק זמן סביר. אולם, ישנן מספר נקודות טכניות אשר מקלות עלינו משמעותית:

- מעבדי Intel תומכים רק בכתובת של עד 48 ביט, ובכך כבר חתכנו 16 ביט ממרחב הכתובות
- נהוג כי ה-Kernel ממוקם בחציו העליון של מרחב הזכרון, ולכן מדובר בעוד ביט שאינו נגיש למשתמש

כלומר, למשתמש יש מרחב זכרון של 47 ביט "בלבד", ולמעשה לא מדובר בהגדלה כה משמעותית אל מול הארכיטקטורות ה"ישנות" של 32 ביט, כפי שנשמע תחילה כשמדברים על 64 ביט.

אפשרות 1 - מיפוי הזכרון ע"י הקצאות שווא

רוב מערכות ההפעלה תומכות בהקצאות זכרון לפי כתובת לבקשתינו (כתובות **קבועות**), למשל באמצעות הפונקציות *mmap()* ו-*VirtualAlloc()* ב-Linux וב-Windows בהתאמה. כל נסיון הקצאה שכזה יכול ללמד על מרחב הזכרון הוירטואלי שלנו:

1. אם ההקצאה הצליחה, טווח הזכרון הנ"ל היה פנוי
2. אחרת, אם ההקצאה נכשלה, סימן שהייתה הקצאת זכרון קודמת כלשהי בתוך המרחב שביקשנו להקצות, והיא חסמה את ההקצאה שלנו

בתיאוריה, נוכל להתחיל לכסות את כלל מרחב הזכרון מתחילתו, הכתובת 0, ועד סופו, 47 ביט, על ידי הקצאות קבועות. כל הקצאה שכזו תלמד אותנו על מידע חדש אודות אזורי הזכרון שכבר מוקצים לתהליך שלנו.

לאחר מספר בדיקות מסתמן כי הפונקציות *mmap()* ו-*VirtualAlloc()* מאפשרות לנו **להקצות** עד 1GB של זכרון בכל קריאה לפונקציה. הקצאת הזכרון היא "זולה" משום שעד שלא נבצע בו שימוש וניגש אליו, מערכת ההפעלה לא תבצע את ההקצאה בפועל עבור התהליך. על ידי הקצאת בלוק אחד בכל פעם, ושחרורו מיד לאחר הבדיקה, נוכל לסרוק את מרחב הזכרון כולו מבלי שהאלגוריתם יקצה כמויות גדולות של זכרון.

היות ונוכל לבצע "דגימה" של עד 1GB (30 ביט) בכל צעד, הרי שנוכל לסרוק את **כלל** מרחב הזכרון של התהליך באמצעות 17 = 30 - 47 ביט קריאות לפונקציה. ובארכיטקטורות מודרניות, 17 ביט (128K) קריאות מערכת יתבצעו בפרק זמן קצר למדי ופתאום סריקת כלל המרחב נראית אפשרית.

בעבור כל הקצאת זכרון שנכשלה, אנו מבצעים חיפוש בינארי כדי להבין אילו אזורים בטווח ההקצאה המקורית היו תפוסים. מכיוון ומרחב הזיכרון הוא דליל למדי, הרי שעלות הרקורסיה זניחה.

אפשרות 2 - פשוט נבקש מפה של הזכרון (בלעדי ל-Windows)

מערכת ההפעלה Windows תומכת בפונקציה *VirtualQuery()*, אשר (לפי MSDN) מקבלת כתובת ו"מחזירה מידע אודות טווח דפים ממרחב הזכרון הוירטואלי של התהליך הקורא". פרטים אלו הם:

- PVOID BaseAddress
- PVOID AllocationBase
- DWORD AllocationProtect
- SIZE_T RegionSize
- DWORD State
- DWORD Protect
- DWORD Type

ובקצרה, על ידי תשאול על הכתובת X נוכל ללמוד כמעט כל מה שנרצה:

1. האם הכתובת היא חלק מדף זכרון ממופה? (protect == 1)
2. כמה בתים, החל מהכתובת X (מיושרת לגבולות דף זכרון) ישנם באזור זכרון זה? (RegionSize)

בונוס: מערכת ההפעלה מתייחסת לזכרון משוחרר כמו לכל סוג אחר של זכרון, ולכן תשאול על הכתובת "0" יחזיר את כמות הבתים הפנויים עד לטווח הזכרון הממופה הראשון. בנוסף, התוצאות מכילות גם את ההרשאות המלאות של כל אחת מההקצאות. זה אומר שנוכל לבצא קריאות VirtualQuery לכל קטע זכרון, לחלץ מהתוצאה את גודל הקטע הנוכחי, ולבצע קריאה נוספת מיד אחריו כדי לדעת את המרחק עד לקטע הבא - ללא צורך בחיפוש בינארי כמו מקודם.

לסיום, שימו לב כי הפונקציה *VirtualQuery()* **אכן מאופשרת** על ידי CFG, ולכן נוכל להשתמש בה מבלי ש-CFG יחסום את ניסיונותנו למיפוי הזכרון.

הפללת מחסנית הצללים

שתי שיטות מיפוי הזכרון שהוצגו מאפשרות בניית מפת זכרון מלאה של כל מרחב הכתובות הוירטואלי של התהליך בו אנו רצים - אנחנו אפילו יודעים את גודלה של כל הקצאה לוגית, ואת ההרשאות שניתנו לה. כעת נרצה לבדוק את ההתאמה של כל הקצאה לפרופיל של מחסנית הצללים:

- מחסנית הצללים תהיה בגודל זהה למחסנית הרגילה
- הערכים במחסנית הצללים הם:
  - אפסים
  - כתובות חזרה - אותן ניתן להשוות לאלו שבמחסנית ה"רגילה"

ולסיכום שלב זה, מערכת ההפעלה Windows מאפשרת חיפוש יעיל ומהיר של מחסנית הצללים, זאת גם ללא שום קצה חוט שיצביע על מחסנית זו.

איתור המחסנית ה"רגילה"

בעוד שישנן מספר טכניקות למציאת המחסנית של החוט (Thread) הנוכחי, הפעם ניסיתי גישה חדשה. במהלך השיטוטים ב-MSDN נתקלתי בפונקציה השימושית להפליא GetCurrentThreadStackLimits. התיאור שלה מציין כי "הפונקציה מחזירה את גבולות המחסנית שהוקצתה על ידי המערכת עבור החוט הנוכחי".

במקום לחפש ידנית מצביע כלשהו למחסנית במרחב הזכרון של הקורבן, ניתן פשוט לבצע קריאה לפונקציה ולקבל את המידע בחינם. והדובדבן שבקצפת - שיטה זו תעבוד **בכל** תהליך קורבן, ולכן היא מציעה פתרון נוח וגנרי במקום לבצע שלב איסוף יעודי עבור כל קורבן.

מציאת פרימיטיב ניצול - Controlled Pair

בזמן ששיחקתי עם פונקציית ה-API החדשה שמצאנו, התברר כי היא מחזירה שתי תוצאות:

- גבולה התחתון של המחסנית
- גבולה העליון של המחסנית

ומכיוון והפונקציה מחזירה שתי תוצאות, החתימה שלה היא:

```
VOID WINAPI GetCurrentThreadStackLimits(
  _Out_ PULONG_PTR LowLimit,
  _Out_ PULONG_PTR HighLimit
);
```

בצירוף מקרים משעשע, מצאנו פונקציה מועמדת לפרימיטיב הדריסה שרצינו. כעת נשאר רק לבדוק:

1. איך הפונקציה מחשבת את הערכים אותם היא מחזירה?
2. האם אנו יכולים לשלוט בערכים אלו?

בדיקה מהירה בדיבאגר העלתה ששני הערכים נשלפים מהמידע הייחודי של החוט (TEB):

```
00007FF9091CD010   65 4C 8B 04 25 30 00   mov r8,qword ptr gs:[30]          GetCurrentThreadStackLimits
00007FF9091CD019   49 8B 80 78 14 00 00   mov rax,qword ptr ds:[r8+1478]
00007FF9091CD020   48 89 01               mov qword ptr ds:[rcx],rax
00007FF9091CD023   49 8B 40 08            mov rax,qword ptr ds:[r8+8]
00007FF9091CD027   48 89 02               mov qword ptr ds:[rdx],rax
00007FF9091CD02A   C3                     ret
```

מידע נוסף אודות מבנה זה ניתן למצוא בקישור הזה.

לכן, ענינו לעצמנו על השאלה הראשונה, וכעת ננסח מחדש את השאלה השנייה:

1. האם אזור הזכרון היעודי של החוט כתיב?
2. האם ניתן לאתר את אזור הזכרון הנ"ל של החוט?

התשובה הקצרה לשאלה הראשונה היא: כן. אזור זה כולל בין היתר את ערך השגיאה האחרון בו נתקלנו, ובפרט האזור כתיב. כעת נשאר לנו רק למצוא את המבנה במרחב הזכרון.

מציאת ה-TEB

השלב הראשון היה למצוא את הכתובת באמצעות הדיבאגר. כעת נשאר רק לחפש ברחבי הזכרון, בתקווה שנמצא כתובות דומות שיעידו על קיומם של מצביעים למבנה זה. לאחר מספר חיפושים ברחבי הזכרון, חיפושים שהתמקדו בעיקר ב-heap, החלטתי להציץ גם על המחסנית. באופן מפתיע, ישנם **שלושה מצביעים** ל-TEB שיושבים ברצף על המחסנית. תבנית זו מאפשרת לנו גם לקבל אינדיקציה ברורה ל"הצלחה".

להלן צילום מסך מתהליך iexplorer.exe:

```
00000057B0BFF7F8   00000194D4CEC5E0
00000057B0BFF800   0000000000000000
00000057B0BFF808   0000000000000000
00000057B0BFF810   0000000000000000
00000057B0BFF818   0000000000000000
00000057B0BFF820   0000000000000000
00000057B0BFF828   00000194D4CE7858
00000057B0BFF830   0000000000000000
00000057B0BFF838   0000000000300000
00000057B0BFF840   0000000100000000
00000057B0BFF848   0000000000000000
00000057B0BFF850   000000000000006C
00000057B0BFF858   000000000000006C
00000057B0BFF860   0000000000000000
00000057B0BFF868   0000000000000000
00000057B0BFF870   0000000000000000
00000057B0BFF878   0000000000000000
00000057B0BFF880   0000000000000000
00000057B0BFF888   0000000000000000
00000057B0BFF890   00000057B0897000
00000057B0BFF898   00000057B0897000
00000057B0BFF8A0   00000057B0897000
00000057B0BFF8A8   0000000000000000
00000057B0BFF8B0   0000000000000000
```

לאחר בדיקה קצרה התברר המקור לתבנית הזכרון המועילה שמצאנו. רוב תהליכי המשתמש, או לפחות אלו שמעניינים תוקפים, נעזרים בממשק של מערכת ההפעלה ליצירת ושימוש במאגר חוטי עבודה (Thread Pool). כחלק מיצירת חוטי העבודה, המצביע למבנה ה-TEB שלהם נשמר על המחסנית 3 פעמים, זאת ככל הנראה כחלק ממבנה כלשהו המתאר אותם.

סיכום התקיפה

אם נסכם את השלבים שראינו, הרי שהתקיפה תיראה כך:

0. נתחיל מריצה בתוך "מכונה וירטואלית" בתוך התהליך הנתקף
1. נקרא ל-*GetCurrentThreadStackLimits()* למציאת המחסנית ה"רגילה"
2. באמצעות מאפיינים אלו, נבצע חיפוש במרחב הזכרון (למשל באמצעות *VirtualQuery()*) ונאתר את מחסנית הצללים
3. נסרוק את המחסנית ה"רגילה" מבסיסה, עד שניתקל באותה הכתובת 3 פעמים - זהו מצביע ה-TEB
4. לצורך אימות, נשווה את ערכי המחסנית ב-TEB לאלו שקיבלנו מקריאת הפונקציה בשלב 1
5. נעדכן את ערכי המחסנית ב-TEB כך שיצביעו שניהם לאזור הקוד אותו נרצה להריץ
6. נקרא בשנית לפונקציה *GetCurrentThreadStackLimits()* עם שני פרמטרים:
    a. כתובת ערך החזרה במחסנית ה"רגילה"
    b. כתובת ערך החזרה במחסנית הצללים

צעד אחרון זה יבצע דריסה כפולה (Controlled Pair) לאחריה ריצת החוט תוסט אל הקוד שלנו

7. ניצחון

הסטאטוס של RFG

לאחר שסיימתי את המחקר, חיכיתי ליציאת גרסה רשמית של Windows 10 (Creators Update) אשר בה יופעל מנגנון ההגנה. גרסאות ניסוי קודמות כללו את המנגנון, אבל מחסנית הצללים מופתה לאותן הכתובות כמו המחסנית הרגילה, ולכן בפועל המנגנון לא באמת פעל.

לצערי, ב-31.01.2017, זמן קצר יחסית לאחר השקת מנגנון ההגנה, Microsoft עדכנה את תוכנית ה-Bug Bounty והודיעה כי RFG אינו כלול יותר בתוכנית. מאוחר יותר הם הוסיפו כי הצוות האדום של החברה איתר בעיה במנגנון זה, ולכן החברה החליטה לבטל את המימוש התוכנתי ולחכות למימוש החומרתי של Intel.

סיכום

במאמר זה הצגתי מספר תקיפות על Return Flow Guard, מנגנון ההגנה הניסיוני של Microsoft להגנה על המחסנית. בעוד שתוכנית התקיפה הראשונה יכולה להתאים במספר מועט של מקרים, הרי שתוכנית התקיפה השנייה מציגה תקיפה מלאה, שלב-אחרי-שלב, אשר נעזרת בפרימיטיב דריסה חזק (Controlled Pair) ובכך עוקפת בצורה מלאה את מנגנון ההגנה.

למרות שהוחלט להפסיק את הפיתוח של RFG, אני מאמין שנוכל למצוא שימושים מועילים נוספים לפונקציה *GetCurrentThreadStackLimits()* גם בתרחישים אחרים, הן בשלב האיסוף והן בשלב הניצול עצמו.

בנוסף, ניכר שבאפיון הנוכחי של מערכת ההפעלה Windows, וברמה מסוימת גם באפיון הנוכחי של Linux, לתוקף המריץ קוד במתכונת "מכונה וירטואלית" ישנן יכולות חזקות למדי בכל הקשור למיפוי מרחב הזכרון של התהליך הנתקף. למעשה, השימוש ב-*VirtualQuery()* מאפשר מיפוי מהיר ופשוט יחסית של כלל מרחב הזכרון, דבר שלמעשה מאפשר להשלים עקיפה מלאה של ASLR, ושל מנגנוני הגנה עתידיים נוספים כדוגמאת מחסנית הצללים אותה Microsoft ניסתה להחביא, ללא הצלחה רבה.

על המחבר

אייל איטקין: חוקר אבטחת מידע העוסק בעיקר בתחומי ה-Embedded. מפעם לפעם עוסק בציד חולשות בפרויקטי Bug bounty, ברשימה הכוללת את: Microsoft (Liberation Guard - Bounty For Defense), (C)Python, (C)Ruby, MRuby, PHP, Perl, ועוד.

- בלוג אבטחה: https://eyalitkin.wordpress.com
- אימייל: eyal.itkin@hotmail.com
- פרופיל hackerOne: https://hackerone.com/aerodudrizzt