Technologic Papers

על מירוצים, תוכנה ומה שביניהם

מאת דוד סטרינסקי

הקדמה

מה אתם חושבים על המצב הבא?

אתם רוצים לרכוש כרטיס לסרט בקולנוע, נשאר מושב אחד אחרון באולם.

אדם נוסף גם רוצה לרכוש את הכרטיס האחרון שנשאר, שניכם מגיעים לקופה ביחד והמערכת מוכרת את הכרטיס לשניכם, אז מי מקבל את הכרטיס בפועל? המצב הזה, נקרא מצב מירוץ או באנגלית Race Condition. שירותי אינטרנט כיום עובדים מהר ומשרתים לקוחות רבים, והעוצמה שלהם הופכת גם לחולשה.

race conditions קורים לרוב בעקבות כשל לוגי, כשאין סדר בדברים הכאוס מתחיל. מוצר שנרכש פעמיים, קופון חד פעמי שנוצל יותר מפעם אחת, על כל אלו ועוד נדבר במאמר הבא.

אז מה זה race condition בתוכנה?

כשמבצעים פעולה באתר או באפליקציה, נוצרת בקשה שנשלחת לשרת. השרת מעבד אותה, משנה מידע אם צריך, ומחזיר תשובה מעודכנת. לפעמים כחלק מהתהליך השרת ניגש למשאב רגיש, מעדכן אותו, ואז שולח תוצאה סופית.

אבל אם שתי בקשות שמנסות לעדכן את אותו משאב מגיעות כמעט באותו רגע, עלול להיווצר בלגן לוגי. השרת לא תמיד יודע איזו בקשה צריכה להיכנס קודם, והתוצאה הסופית תלויה בסדר הטיפול. זה דומה לשני אנשים שכותבים על אותו פתק בלי לתאם, ובסוף נשארים עם משהו לא ברור.

בדרך כלל מצב כזה יוצר חולשה לוגית, למרות שלא תמיד. כשזה כן קורה, ההשפעה יכולה להיות מפתיעה ומשמעותית.

עד כמה זה באמת נפוץ?

race conditions הן לא החולשה הכי נפוצה, אך פוטנציאל הנזק שלהן גבוה מאוד.

מחקרים מראים שכ-80% מהבאגים בתחום המקביליות הם בעצם race conditions.

גם בעולם ה-Bug Bounty, שבו חברות מציעות פרסים לפי חומרת החולשה, מדווחים שקרוב ל-1% מכלל הדיווחים הם על race conditions. רק בחצי הראשון של 2025 דווחו יותר מ-21,500 חולשות, כך שגם אם רק אחוז אחד מהן קשור ל-race conditions מדובר במספר גדול ומשמעותי.

הן אולי פחות נפוצות מחולשות מוכרות אחרות, אבל כשהן מופיעות הן מסוגלות לעקוף מנגנוני אבטחה שלא תמיד ניתן לעקוף בדרך אחרת. עם פוטנציאל נזק כזה, אין ספק שמגיע להן מאמר משל עצמן.

כלים בהם השתמשתי להדגמות במאמר

רוב העבודה נעשה באמצעות מודולים שונים בכלי Burp Suite, הכי מייצר לנו סוג של "מתווך" בין הדפדפן לבין האפליקציה כך שניתן "לתפוס" את הבקשות בטרם יעברו הלאה לעיבוד, לראות את היסטוריית הבקשות שנשלחו, לערוך אותם ובאמצעות מודולים מובנים ותוספים ניתן לבצע מתקפות שונות. המודולים והתוספים בהם השתמשתי הם:

Burp Repeater - המודול מאפשר לגשת לתוכן הבקשה לערוך אותה ולשלוח אותה שוב ושוב, ניתן ליצור העתקים של הבקשה, ליצור קבוצה של בקשות ולשלוח אותם במקביל או אחת אחרי השניה ועוד.

Turbo intruder - תוסף Burp Suite המאפשר באמצעות קוד לשלוח מספר רב של בקשות במקביל, התוסף שימושי מאוד כאשר צריכים לשלוח מספר בקשות בזמן כמעט זהה והוא חזק יותר מהמודול Intruder המובנה ששולח את הבקשות באופן טורי.

ניגש לעניינים

מאחר וקיימים מספר סוגים שונים של race conditions נרצה להרחיב על כל אחד מהם בנפרד.

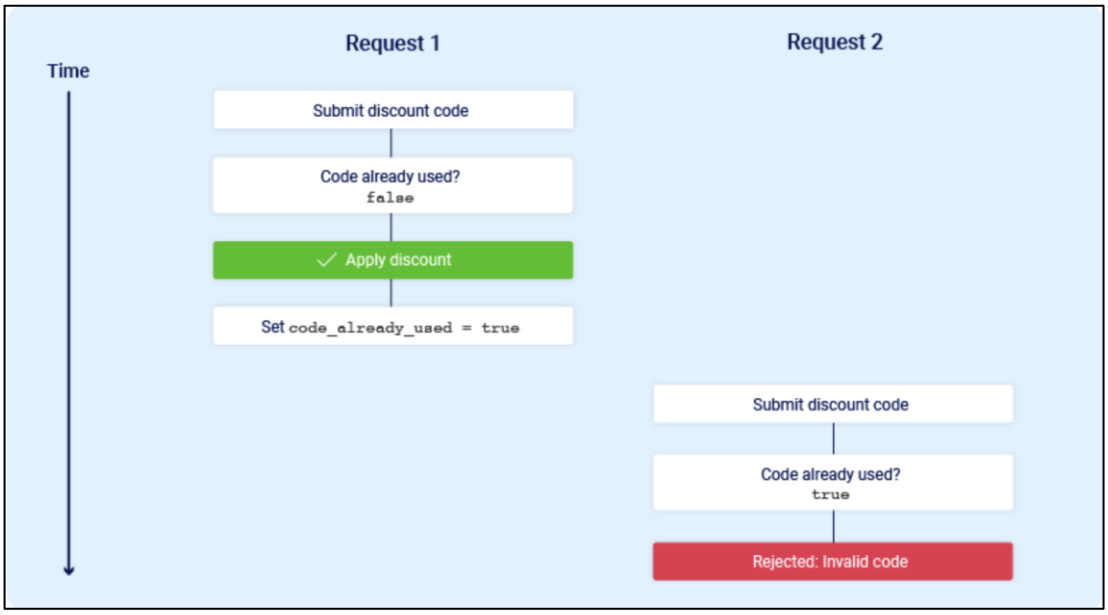
נתחיל מהמצב הראשון: דריסת מנגנוני הגבלה.

המוכר ביותר מבין כל המצבים הוא המצב בו יש לנו הגבלה מסוימת בלוגיקה העסקית. ניקח לדוגמא חנות אינטרנטית המאפשרת להזין קופון חד פעמי לפני התשלום ולקבל הנחה על סל הקניות.

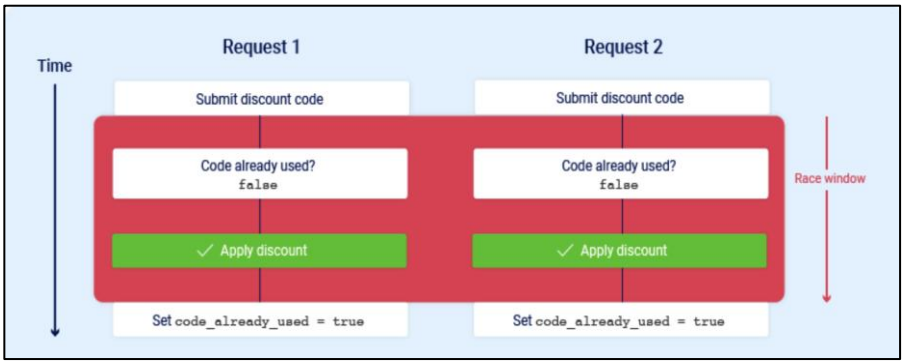
לוגיקה כזאת תראה בערך באופן הבא:

- נבדוק שהקופון טרם נוצל.
- נפעיל את ההנחה על סל הקניות.
- נעדכן במסד הנתונים שהקופון נוצל כדי שלא ינוצל שוב.

עם הלוגיקה הזאת, כל ניסיון לשימוש חוזר מאוחר יותר בקופון לא יתקבל.



אם כך, מה יקרה אם אותו לקוח ישלח את הבקשה פעמיים באותו הזמן עם אותו קוד הקופון?



בתרשים מעלה ניתן לראות כי לפי הלוגיקה שבחרנו, בשתי הבקשות ראשית נבצע בדיקה אם הקופון נוצל ומאחר ולא נוצל נעבור לשלב הבא, נפעיל את ההנחה פעמיים, לבסוף נעדכן בשתי הבקשות כי הקופון נוצל (לא משנה מי יעדכן ראשון מאחר והשדה לא משתנה במסד הנתונים).

שתי הבקשות עוברות באותו הזמן את המצב שבודק האם הקופון נוצל, זהו מצב המירוץ שלנו ומשך הזמן בו הוא קורה נקרא חלון המירוץ או באנגלית Race window (מסומן באדום בתרשים). זהו מושג חשוב מאוד שנשתמש בו המון בהמשך.

קיימות וריאציות שונות למתקפה שכזו, למשל:

- שימוש בקופון חד פעמי יותר מפעם אחת.
- דירוג מוצר מספר פעמים.
- משיכה או העברת כספים מעבר ליתרה בחשבון.
- שימוש חוזר באותו פתרון (מנגנון לזיהוי בוטים) CAPTCHA.
- מעקף על מנגנוני הגנה מפני Brute force עם הגבלת קצב.

חולשות דריסת הגבלות הן תת טיפוס של חולשות מסוג "time-of-check to time-of-use", בהמשך נראה דוגמאות לחולשות שלא נופלות תחת הקטגוריה הזו.

אז איך נוכל לגלות ולנצל חולשות מהסוג הזה?

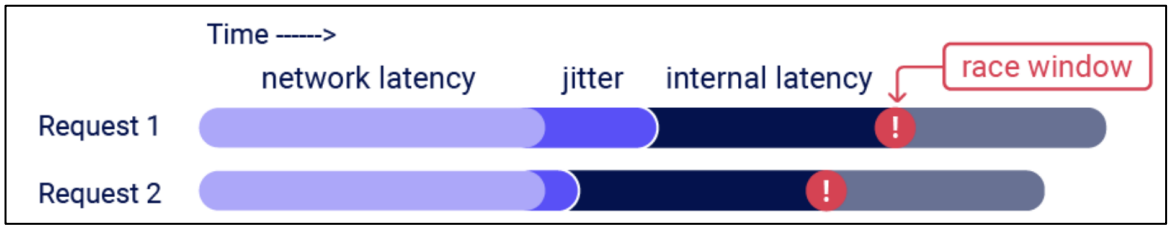
הרעיון פשוט מאוד, ואפשר לחלק אותו לשלבים הבאים:

נזהה נקודת קצה או הגבלת קצב על משאב יקר ערך או בעל השפעה גבוהה על האבטחה.

נשלח מספר בקשות במקביל לאותה הנקודה ונבדוק אם הצלחנו לדרוס.

הקושי העיקרי הוא לשלוח את הבקשות כך שלפחות 2 חלונות מירוץ יסתדרו במקביל ונקבל מצב מירוץ ביניהם, לפעמים מדובר במרווח כל כך קטן ברמת מילישניות ואפילו פחות.

וגם אם נשלח את כל הבקשות בדיוק באותו הזמן קיימים גורמים חיצוניים נוספים שיכולים להשפיע על הזמן של החלון ועל הסדר שבו השרת מעבד את הבקשות.



וכאן נכנסים לתמונה הכלים החיצוניים כמו Burp Suite שמספק לנו את המודול של ה-Repeater.

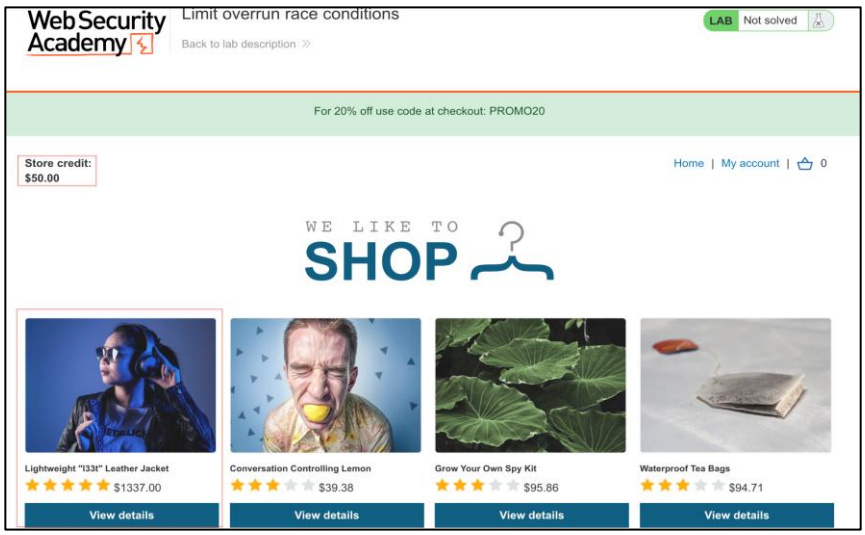
ה-Repeater מאפשר לנו ליצור קבוצה של בקשות ולשלוח אותן במקביל בשיטות שונות לגרסאות HTTP שונות:

- עבור HTTP/1 - המודול משתמש בטכניקה שנקראת last-byte synchronization. הטכניקה היא בעצם לשמור את הבייט האחרון של כל אחת מהבקשות ולאחר העברת הבקשות להעביר יחד את כל הבייטים האחרונים שישמרו מכל בקשה. לרוב שרתים מעבדים בקשות רק לאחר שהסתיימה קבלת הבקשה במלואה, כך באמצעות הטכניקה הזאת ייתכן ונצליח לתמרן את השרת לעבד את כל הבקשות שלנו ביחד.
- עבור HTTP/2 - המודול משתמש בטכניקה שנקראת single-packet attack. הרעיון הכללי של הטכניקה אומר "לארוז" את כל הבקשות ביחד לסגמנט TCP אחד ולשלוח לשרת ובכך לנסות צמצם את הפרשי הזמנים בין עיבוד של כל בקשה. הטכניקה הוצגה לראשונה על ידי חוקרי PortSwigger בכנס Black Hat 2023 USA.

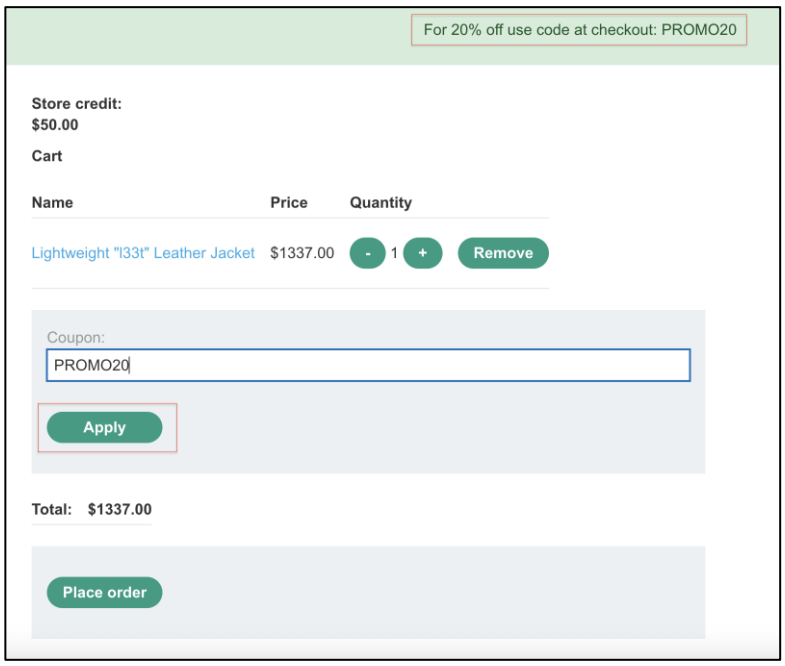
אמנם לרוב מספיק לשלוח שתי בקשות במקביל כדי לזהות את החולשה אך שליחה של מספר גדול יותר של בקשות עוזר לצמצם את ההשהיות והתנודות ועיכובים נוספים או בשמם המקצועי server-side jitter.

הדגמה

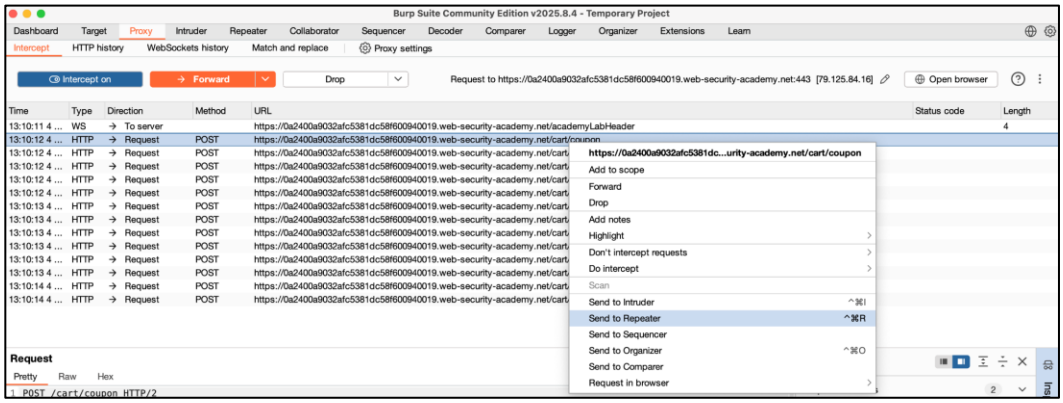
הגיע הזמן, הדגמה באמצעות מעבדה. במעבדה קיבלנו משתמש לחנות אינטרנטית שמוכרת מעיל עור במחיר שגבוה מהקרדיט שיש לנו בארנק, בנוסף קיבלנו קוד קופון חד פעמי להנחה של 20%, המטרה שלנו היא לקנות את המעיל עם הקרדיט הקיים וספוילר קטן, נשתמש בקופון שקיבלנו.



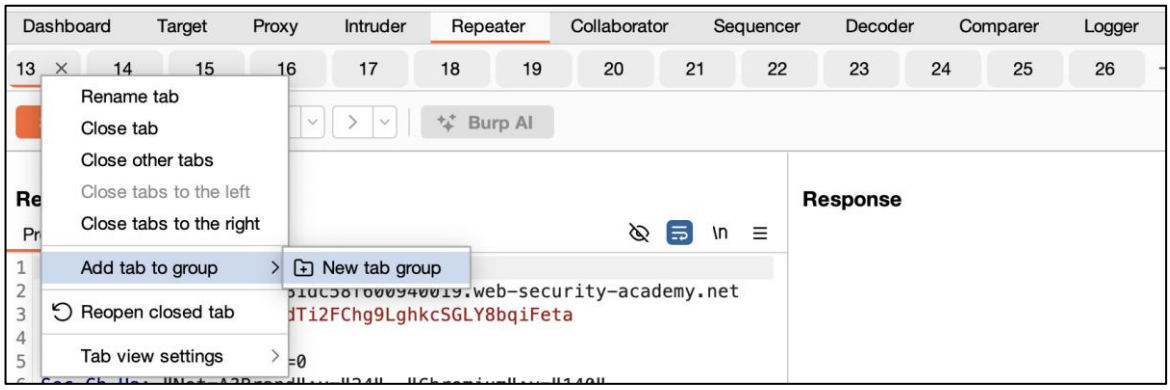
כפי שניתן לראות קיבלנו קרדיט של 50$ אבל המעיל עולה 1337$, נתחיל את הבדיקה שלנו על ידי כך שננסה להבין קודם כל איך עובד תהליך הרכישה ואיך עובד תהליך הפעלת הקופון.



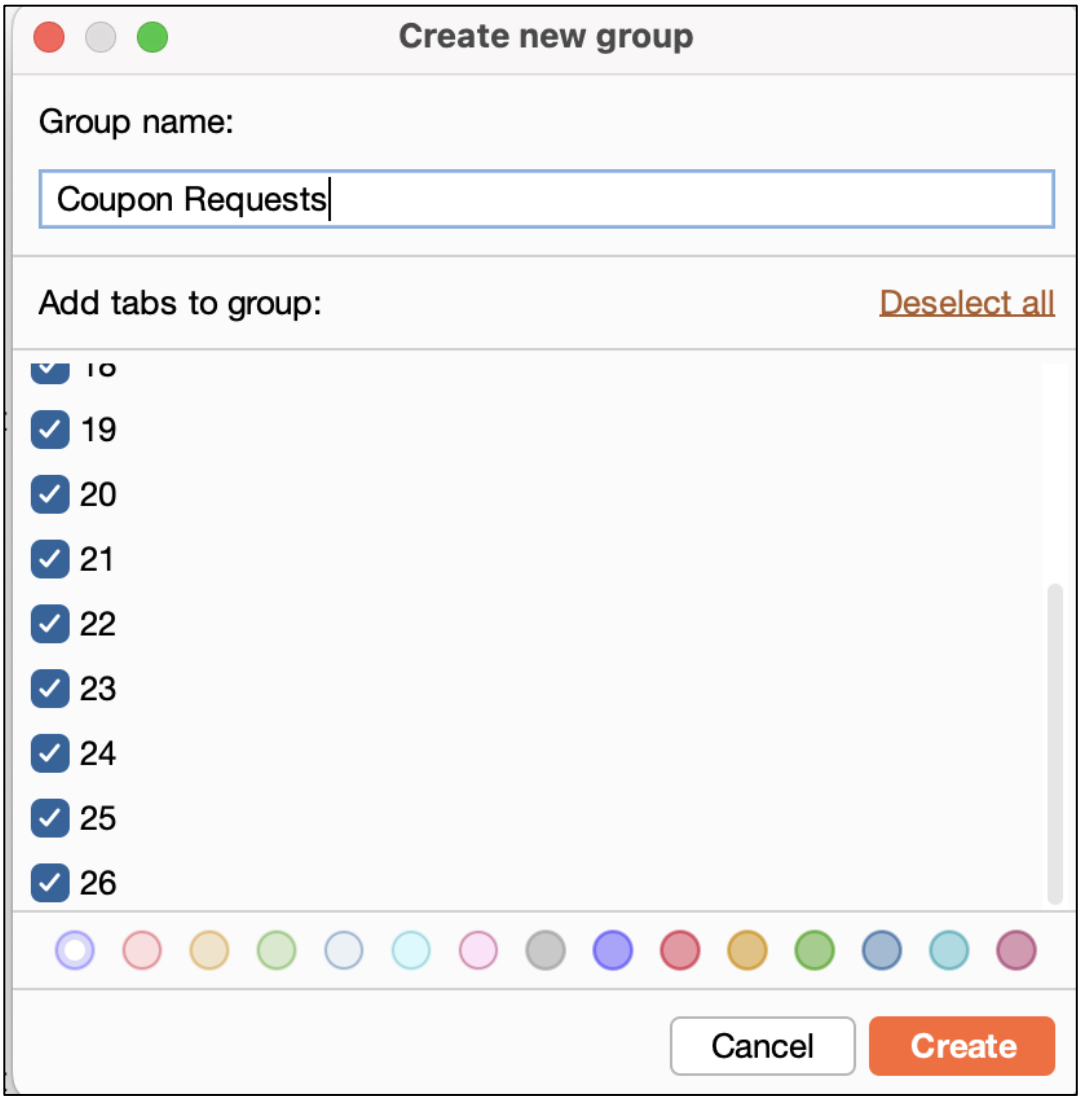
לאחר לחיצה על Apply, ניגש ל-BurpSuite Proxy ונאתר את הבקשה שנשלחה. את הבקשה נעביר ל-Repeater, נעשה זאת מספר פעמים כדי ליצור מספר עותקים של הבקשה (כמספר הפעמים שנצטרך להפעיל את הקופון כדי להגיע למחיר הרצוי)



כעת נעבור ללשונית ה-Repeater, שם ניצור קבוצה חדשה של בקשות מהבקשות שהעברנו מה-Proxy.

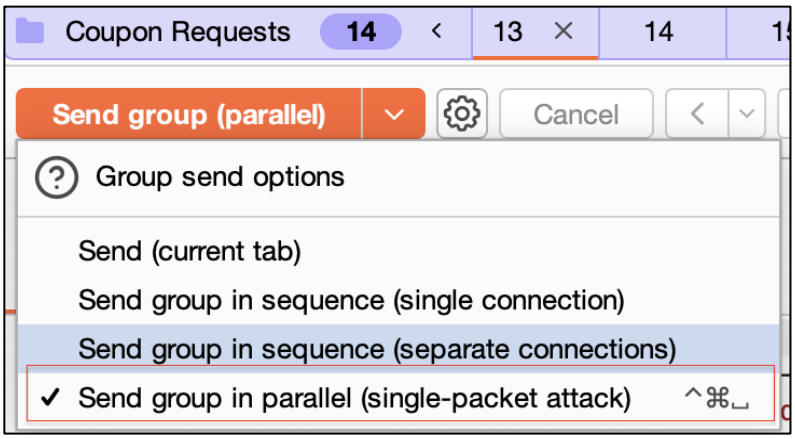


נסמן את הבקשות מתוך הרשימה וניתן שם לקבוצה (לא חובה, יש שם המופיע כברירת מחדל), ונלחץ על Create.

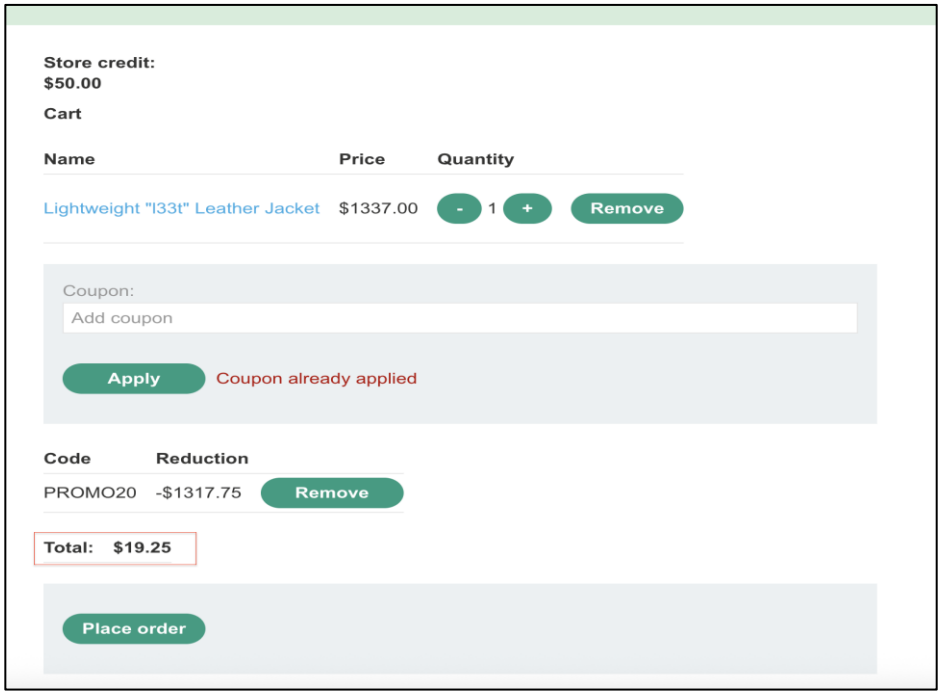


כעת ניגש אל לשונית הקבוצה שיצרנו ונלחץ על החץ שמופיע כדי לבחור את שיטת השליחה לכל הקבוצה, במעבדה הזו השתמשתי בשיטת single-packet בגלל פרוטוקול ה-Http.

לאחר בחירת שיטת השליחה, נלחץ על Send group.



נחזור אל המעבדה, לאחר השליחה קיבלנו מהשרת שהבקשות עברו ואפשר לראות כי ההנחות עם הקופון עבדו והגענו למחיר רצוי.



נחמד לקנות מעיל שעולה $1337 רק ב-$19.25 לא? זו רק ההתחלה, נתקדם לחקור מקרים נוספים.

ומה עושים כאשר נשלחות מספר בקשות למספר נקודות קצה?

תתארו לכם את המצב הבא, לקוח מוסיף מוצר לעגלה, משלם עליו ותוך כדי המתנה לאישור התשלום מוסיף עוד מוצרים לעגלה.

יש מספר וריאציות שהחולשה יכולה להופיע כשהשרת מאמת את התשלום ומאשר את ההזמנה עבור בקשה אחת שנשלחה. נניח כי התהליך הזה קורה בצורה הבאה:

- הסל בהמתנה עם מוצרים
- נשלחת בקשה לנקודת הקצה בשרת, למשל: POST /makePayment
- מתבצע בשרת אימות לתשלום
- מתבצע אישור להזמנה

במקרה הזה, נוצר חלון מירוץ שניתן לנצל בין הזמן שהתשלום מאומת לזמן שההזמנה מאושרת.

כאשר בודקים חלונות מירוץ עם מספר נקודות קצה, אפשר להיתקל בבעיות שנוצרות בעקבות תזמון חלונות המירוץ הנוצרים בשליחת הבקשות, גם אם הן נשלחות במקביל.

בעיות אלו נוצרות לרוב בעקבות שתי גורמים:

- השהיות שנוצרות בעקבות ארכיטקטורת הרשת - למשל יכולה להיווצר השהייה כשה-Client יוצר חיבור חדש עם ה-Server, גם לסוג הפרוטוקול בו הם מתקשרים יכולה להיות השפעה על ההשהייה.
- השהיות הנוצרות על ידי עיבוד בנקודת הקצה - לנקודות קצה שונות יש זמני עיבוד שונים, תלוי באילו פעולות השרת צריך לבצע כשפונים לאותה הנקודה.

למזלנו, גם לבעיות האלו יש פתרון, והוא נקרא חימום החיבור (Connection Warming).

חימום החיבור (Connection Warming)

החיבור הראשוני בין ה-Client ל-Server בדרך כלל לא מפריע למתקפות על race conditions מאחר ושרתים לרוב משהים בקשות מקבילות באופן שווה, לכן הבקשות נשמרות בסנכרון.

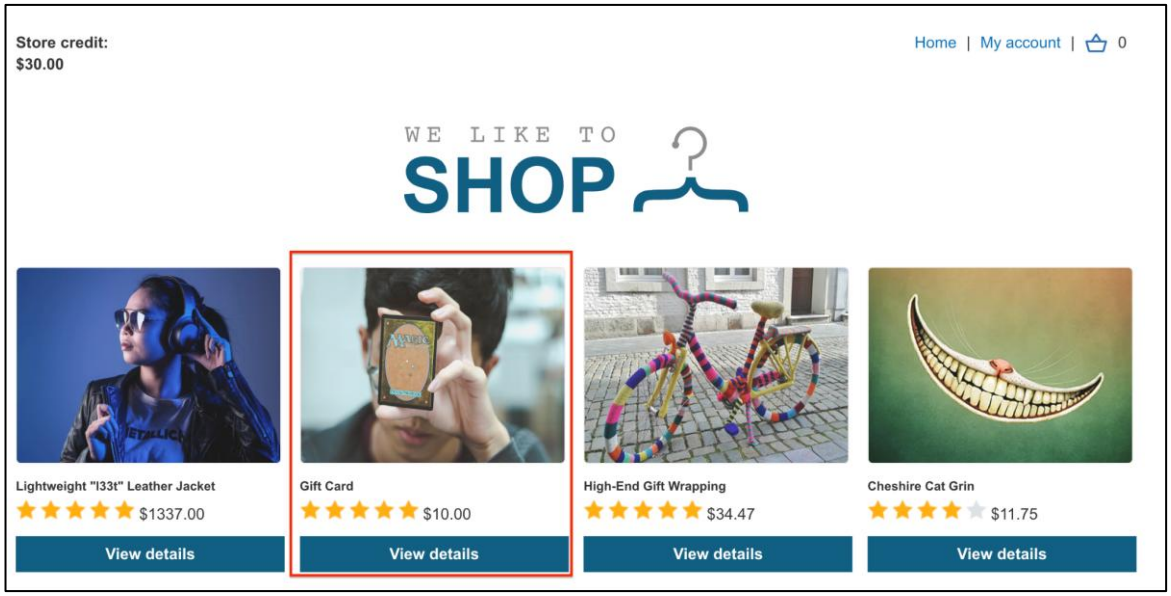
חשוב להבדיל השהיות אלו הנוצרות בגלל החיבור לעומת השהיות הנוצרות בעקבות זמן עיבוד בנקודת הקצה.

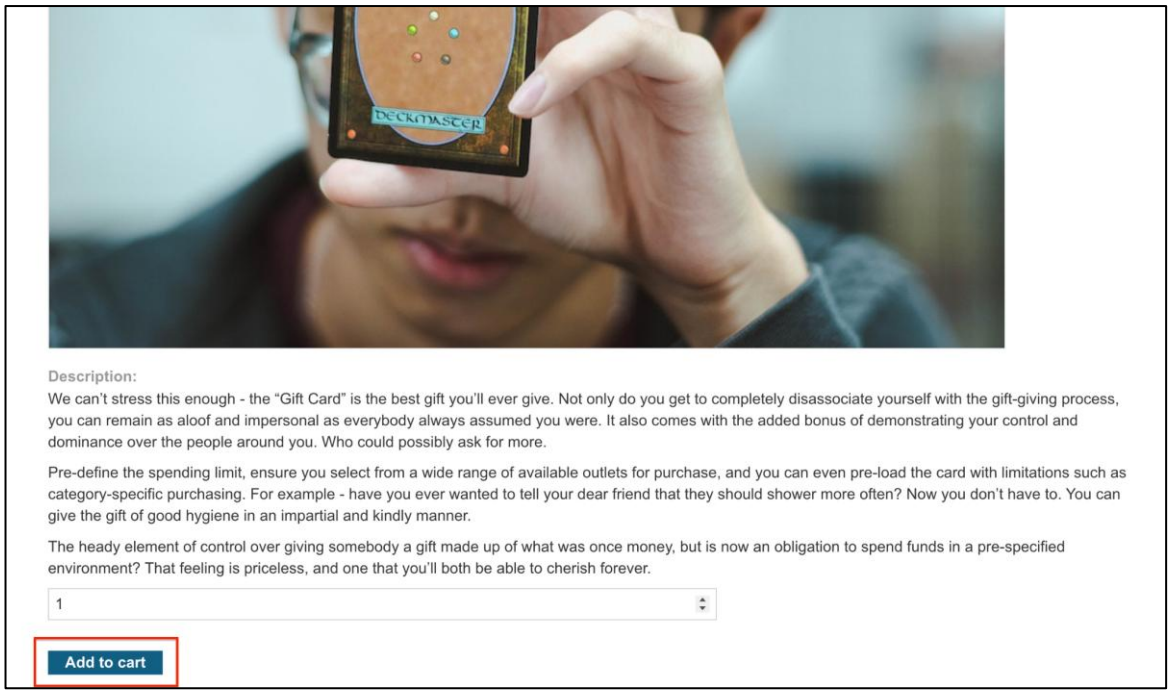
דרך אחת להתגבר על בעיית סנכרון החלונות הזו היא באמצעות "חימום" החיבור על ידי הוספה לקבוצה של בקשת ה-GET הנשלחת כשמבקשים מהשרת את עמוד הבית לדוגמא ושליחת הבקשות באופן סדרתי (אחת אחרי השניה) , כך בעצם אנחנו יוצרים מצב בו הבקשה הראשונה שנשלחה תיקח הכי הרבה זמן מאחר והיא יוצרת את החיבור, ושאר הבקשות יהיו תלויות רק בזמני העיבוד.

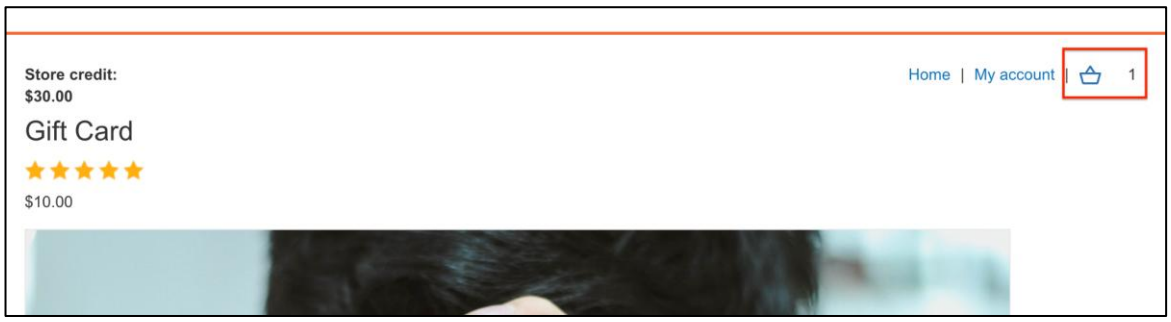
אם עדיין מקבלים זמני תגובה לא עקביים בנקודת קצה מסוימת, גם אם השתמשנו בשיטת ה-Single packet, נוכל להסיק כי השהיות בשרת מפריעות לנו לביצוע התקיפה.

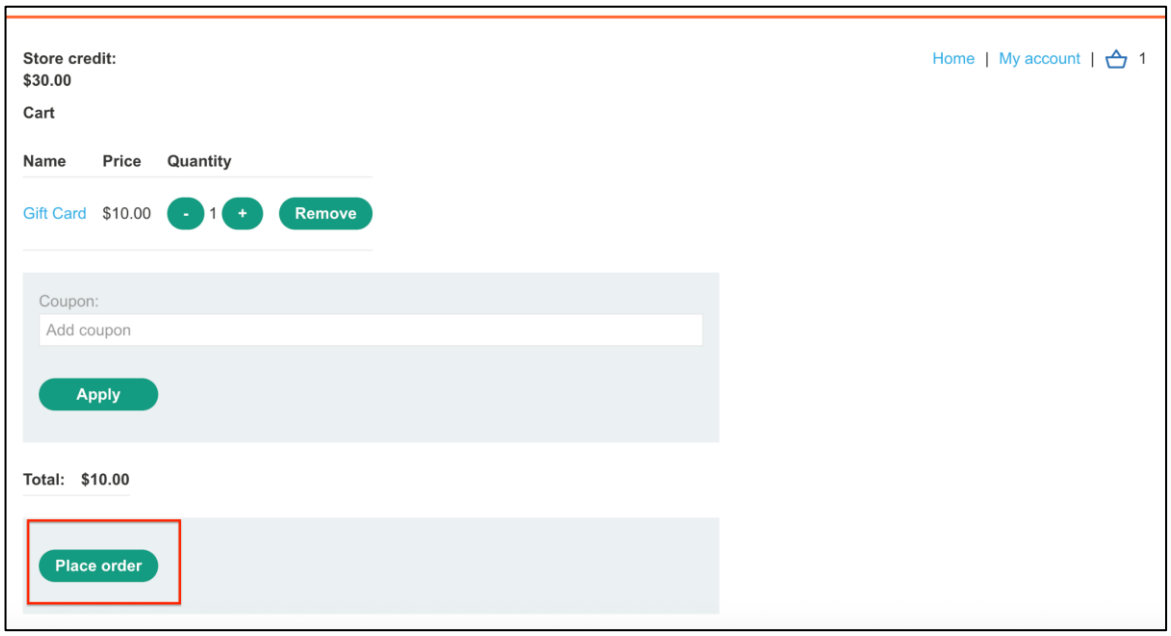
ניתן לפתור את זה עם שימוש בתוסף Turbo Intruder של BurpSuite שישלח כמה בקשות חיבור כאלו כדי לחמם את החיבור לפני שיוציא את הבקשות העיקריות לתקיפה.

והפעם במעבדה שלנו, המטרה היא שוב לקנות את מעיל העור היקר, נתחיל בהתחברות עם הפרטים שקיבלנו ונעבור על תהליך הקניה עם פריט זול יותר, גיפטקארד במקרה הזה וננסה לראות האם נוכל לאתר race conditions.

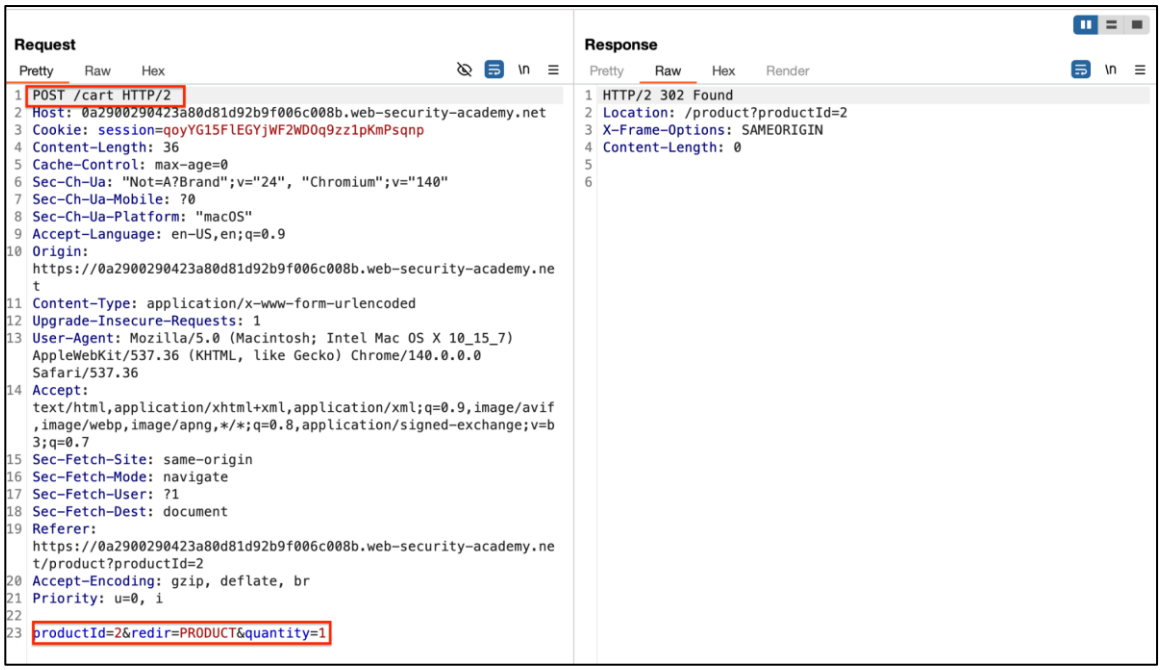


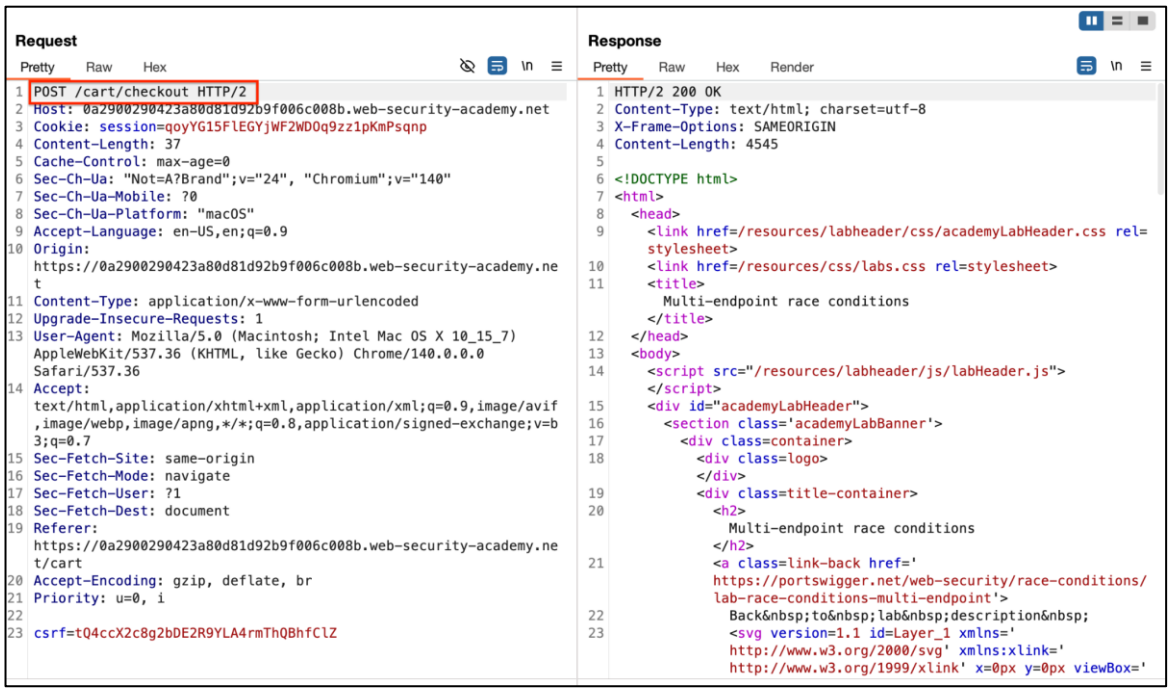






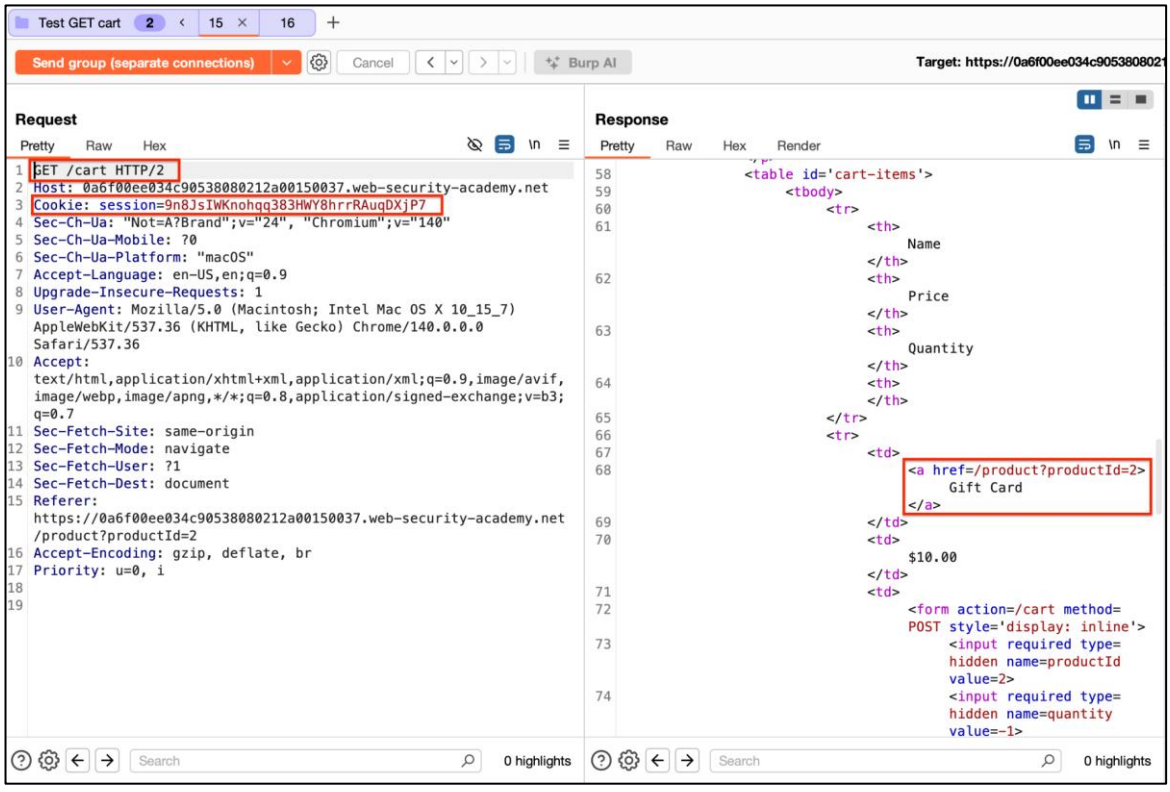
לאחר השלמת ההזמנה, ניגש להיסטוריית הפרוקסי שלנו ב-Burpsuite ונבחן אילו בקשות משתתפות בתהליך הרכישה.

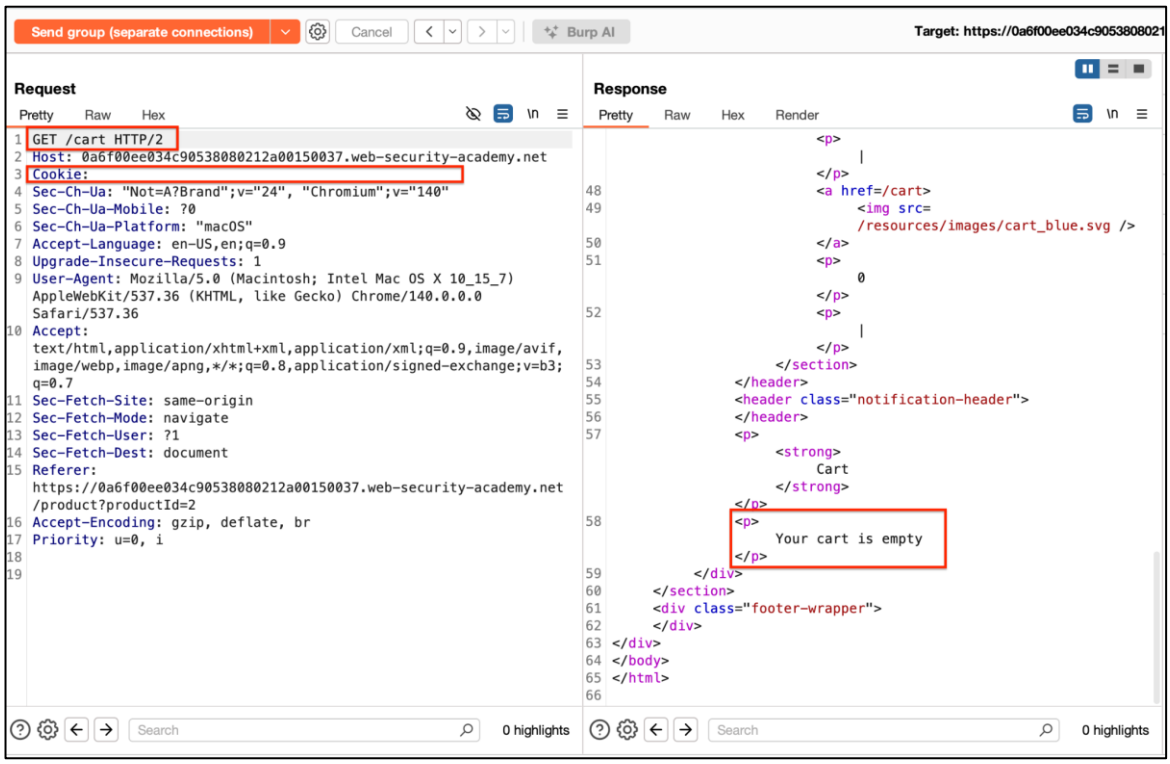




מצאנו שנשלחות שתי בקשות, אחת לאחר הוספת הפריט לעגלת הקניות, והשניה לאחר הרכישה עצמה מתוך סל הקניות.

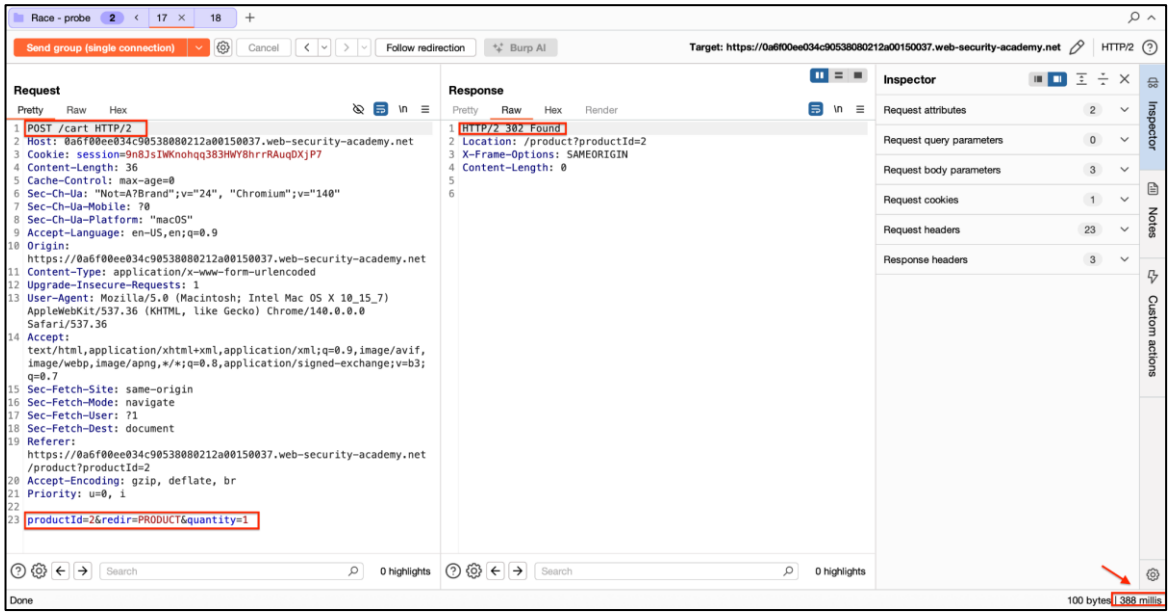
נעשה ניסוי קטן, ניקח את הבקשה שנשלחת כשעוברים לעמוד של עגלת הקניות ונשלח אותו פעמיים, פעם אחת עם Cookie שקיבלנו מהשרת ופעם שניה ללא.

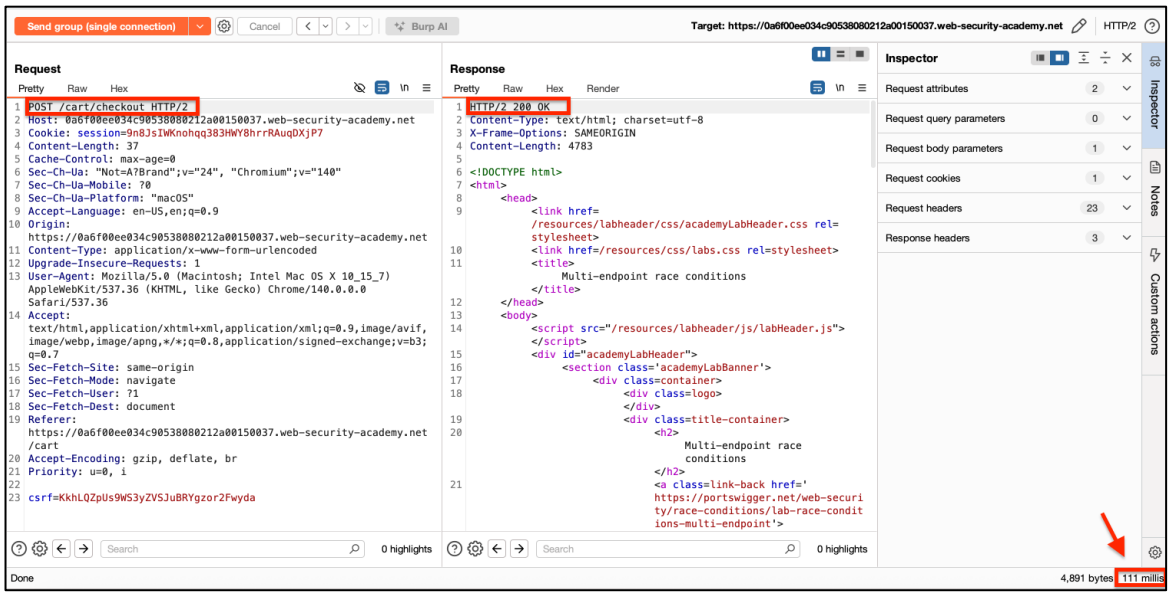




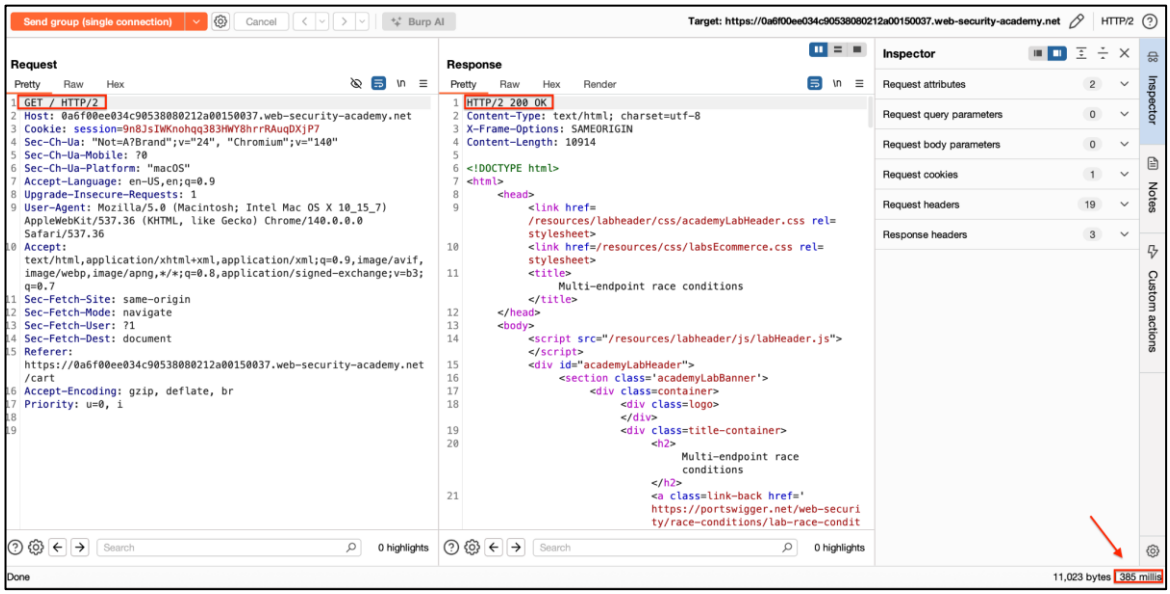
כששלחנו את הבקשה עם ה-Cookie קיבלנו את העגלה שלנו עם המוצרים, וכששלחנו ללא קיבלנו עגלה ריקה. נסיק מזה שפרטי עגלת הקניות נשמרים בצד השרת, מצב כזה יכול להוביל להתנגשות אותה אנחנו מחפשים. נחזור לתהליך הקניה שלנו וננסה לנצל את המצב הזה.

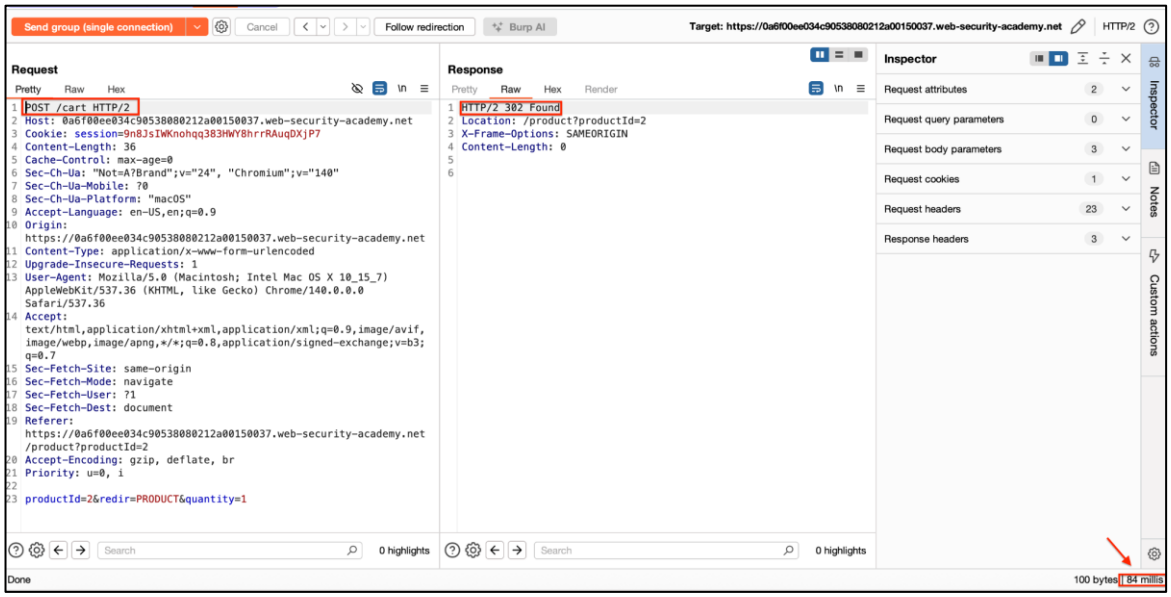
ניקח את שתי הבקשות שראינו קודם, הבקשה עבור הוספה לעגלה והבקשה לרכישה, ונעביר אותם ל-Repeater. נשלח את שתיהן לשרת באופן באופן סדרתי בחיבור אחד:

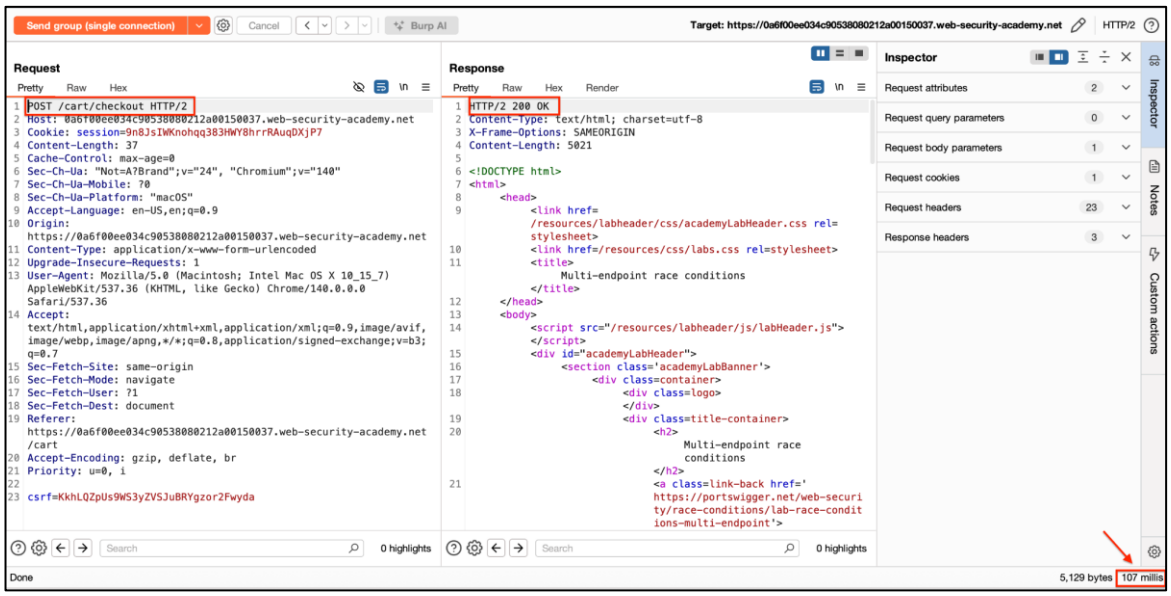




הבקשות עברו לשרת עברו וקיבלנו תשובה תקינה מהשרת, אך אם נשים לב לזמני התגובה המסומנים, יש ביניהם הפרש יחסית גדול, לכן נרצה להשתמש בשיטה שראינו קודם ולחמם את החיבור על ידי הוספת בקשת ה-GET שנשלחת בתחילת החיבור לאתר.

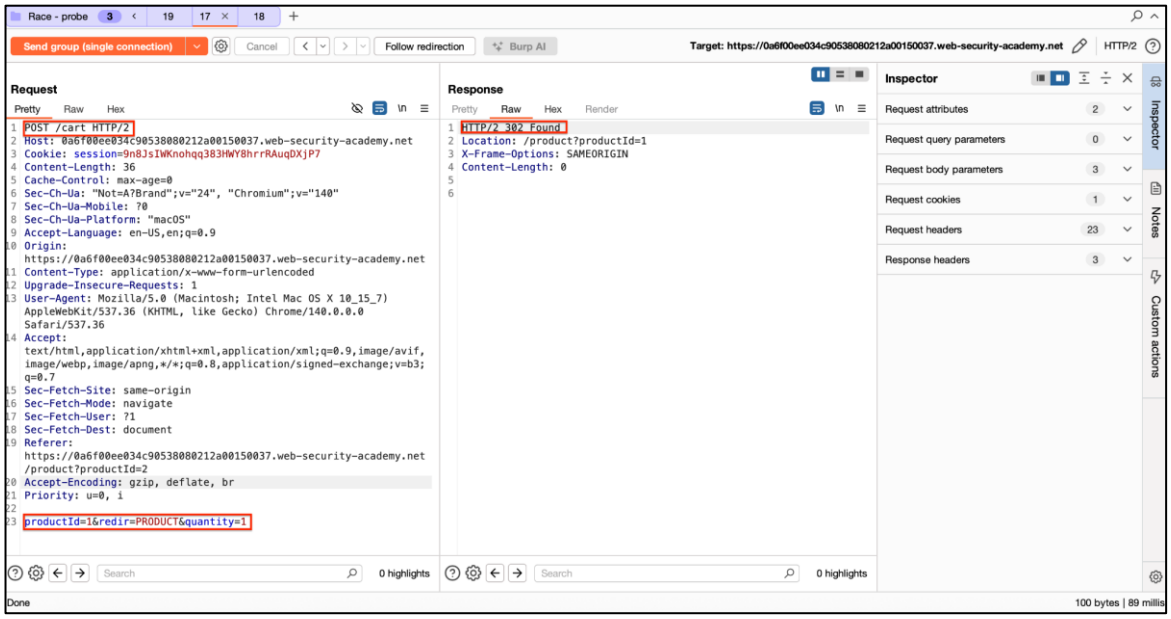


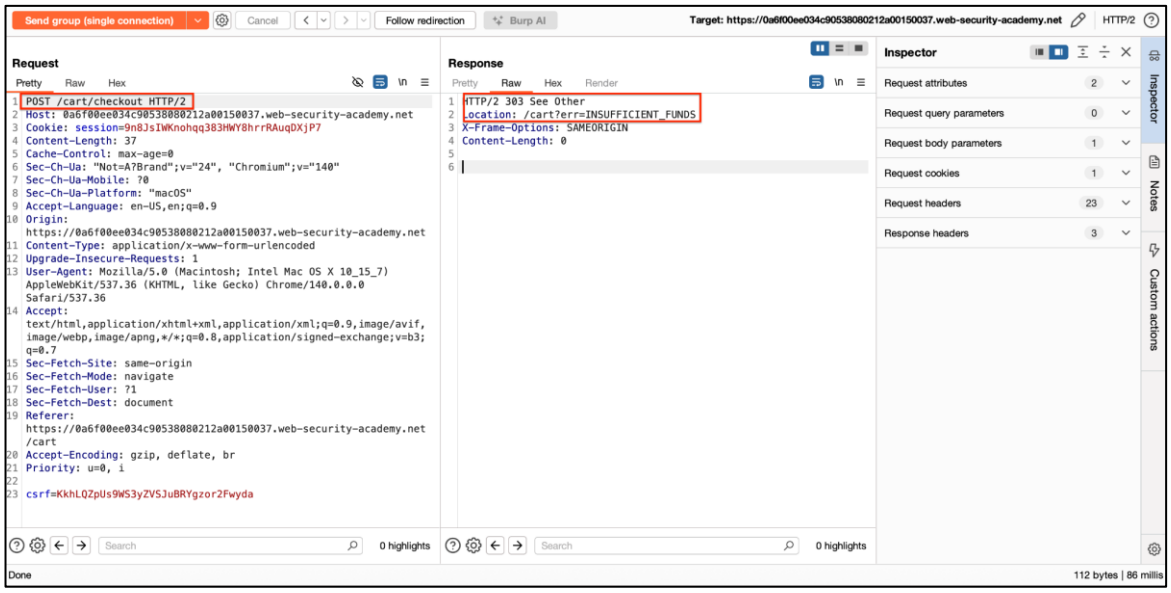




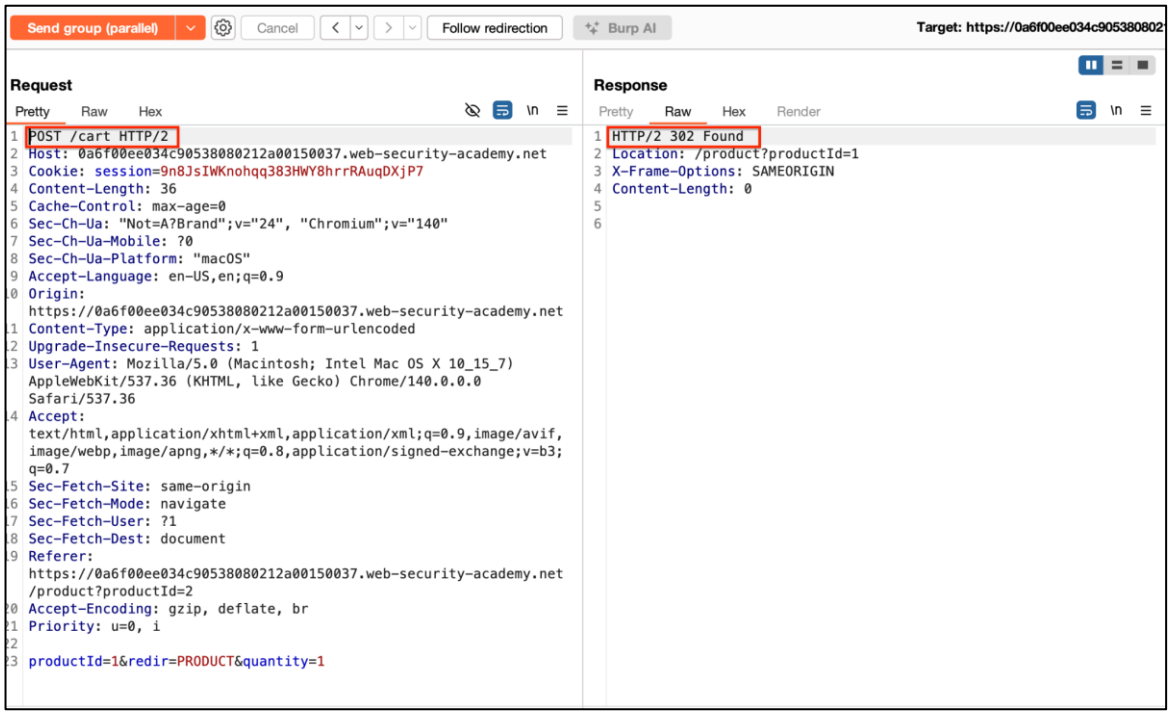
כעת אפשר לראות שהבקשה שלקחה הכי הרבה זמן היא הבקשה הראשונה שיוצרת את החיבור וההפרש בין זמני התגובה של הבקשות השניות קטן והחלונות יותר מסונכרנים.

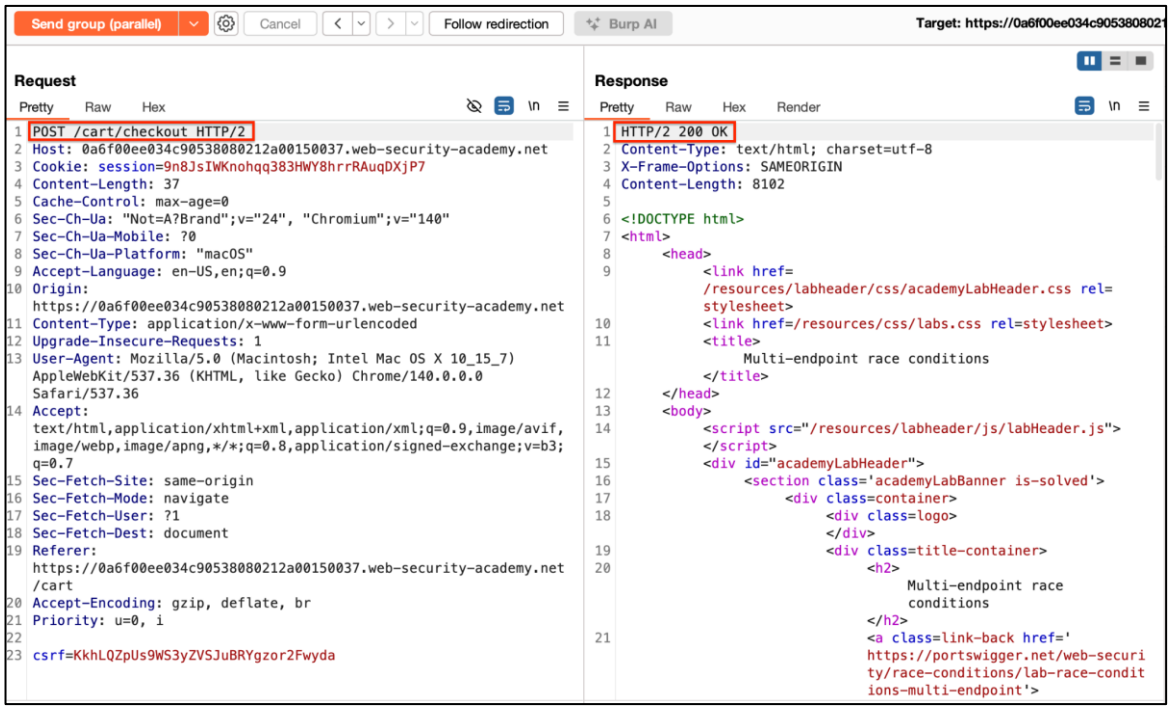
לאחר בדיקה קטנה באתר החנות מצאנו שה-productID של המעיל הוא 1, ננסה לשנות בבקשה שנשלחת לעגלה את ה-productID מ-2 (שזה הגיפט קארד) ל-1 ולשלוח את הכל שוב ביחד באופן סדרתי כמו שעשינו בניסוי מעלה.





הבקשה להוספה לקופה עברה בהצלחה, אך הבקשה לביצוע הרכישה נכשלה בעקבות חוסר של קרדיט לקניה, הגיוני סה"כ. אבל מה יקרה אם נשלח את כל הבקשות ביחד באופן מקבילי בשיטת single packet?


Send group (parallel)
Cancel
Follow redirection
Burp AI
Target: https://0a6f00ee034c905380802
Request
Pretty Raw Hex
POST /cart HTTP/2
Host: 0a6f00ee034c90538080212a00150037.web-security-academy.net
Cookie: session=9n8JsIWKnohqq383HWY8hrrRAuqDXjP7
Content-Length: 36
Cache-Control: max-age=0
Sec-Ch-Ua: "Not=A?Brand";v="24", "Chromium";v="140"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "macOS"
Accept-Language: en-US,en;q=0.9
Origin: https://0a6f00ee034c90538080212a00150037.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://0a6f00ee034c90538080212a00150037.web-security-academy.net/product?productId=2
Accept-Encoding: gzip, deflate, br
Priority: u=0, i
productId=1&redir=PRODUCT&quantity=1
Response
Pretty Raw Hex Render
HTTP/2 302 Found
Location: /product?productId=1
X-Frame-Options: SAMEORIGIN
Content-Length: 0


Send group (parallel)
Cancel
Follow redirection
Burp AI
Target: https://0a6f00ee034c905380802
Request
Pretty Raw Hex
POST /cart/checkout HTTP/2
Host: 0a6f00ee034c90538080212a00150037.web-security-academy.net
Cookie: session=9n8JsIWKnohqq383HWY8hrrRAuqDXjP7
Content-Length: 37
Cache-Control: max-age=0
Sec-Ch-Ua: "Not=A?Brand";v="24", "Chromium";v="140"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "macOS"
Accept-Language: en-US,en;q=0.9
Origin: https://0a6f00ee034c90538080212a00150037.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://0a6f00ee034c90538080212a00150037.web-security-academy.net/cart
Accept-Encoding: gzip, deflate, br
Priority: u=0, i
csrf=KkhLQZpUs9WS3yZVSJuBRYgzor2Fwyda
Response
Pretty Raw Hex Render
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 8102
<!DOCTYPE html>
<html>
<head>
<link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
<link href=/resources/css/labs.css rel=stylesheet>
<title>Multi-endpoint race conditions</title>
</head>
<body>
<script src="/resources/labheader/js/labHeader.js"></script>

<section class='academyLabBanner is-solved'>

<h2>Multi-endpoint race conditions</h2>
<a class=link-back href='https://portswigger.net/web-security/race-conditions/lab-race-conditions-multi-endpoint'>

```
<p>
     <strong>
          Your order is on its way!
     </strong>
</p>
<table>
     <tbody>
          <tr>
               <th>
                    Name
               </th>
               <th>
                    Price
               </th>
               <th>
                    Quantity
               </th>
               <th>
               </th>
          </tr>
          <tr>
               <td>
                    <a href=/product?productId=1>
                         Lightweight &quot;l33t&quot;
                         Leather Jacket
                    </a>
               </td>
               <td>
                    $1337.00
               </td>
```

אכן כעת גם הבקשה לביצוע הרכישה התקבלה והמעיל היקר שלנו כבר בדרך אלינו!

רצפי שלבים מוסתרים

למעשה, ייתכן שבקשה אחת תיזום רצף של מספר שלבים שקורים מאחורי הקלעים, נתייחס לשלבי ביניים אלו כתת שלבים. אם נוכל לזהות בקשה אחת או יותר שנוגעות באותם הנתונים נוכל לנסות לנצל את תתי השלבים לטובתינו כדי לחשוף כשלים לוגיים רגישים לזמן אשר נפוצים בתהליכים רבי שלבים. כך בעצם מתאפשר לנו בעקבות פירצות של race conditions לקבל כוח בהרבה יותר גדול מדריסת מנגנוני הגבלה.

ניקח לדוגמא את מנגנון ה-MFA, אם אינו מיושם כראוי, נוכל לעקוף אותו.

נביט ב-Pseudo code הבא:

```
session['userid'] = user.userid
if user.mfa_enabled:
        session['enforce_mfa'] = True
 generate and send MFA code to user
 redirect browser to MFA code entry form
```

לפי קטע הקוד, אם עברנו את החלק הראשון של האימות באמצעות פרטי ההזדהות, המשתמש קיבל את הסשן מהשרת, כלומר שהוא כביכול כבר מאומת מבלי שביצע את השלבים הנוספים הדרושים.

כעת תוקף יכול לנצל את הסשן הזה כדי לפנות לנקודות קצה רגישות ולמשוך מידע שרק משתמשים מאומתים יכולים לגשת אליהם.

אז מה המתודולוגיה?

כדי לזהות ולנצל רצפי פעולות עם מספר תתי שלבים מוסתרים נשתמש בשיטה המסוכמת במאמר:

Smashing the state machine: The true potential of web race conditions by PortSwigger Research

השלב הראשון: חיזוי

- בדיקת כל נקודות הקצה היא לא מעשית, לכן נצמצם את הנקודות לפי שתי השאלות הבאות:
- האם נקודת הקצה חשובה מבחינה אבטחתית?
- האם יש פוטנציאל להתנגשות בעיבוד הבקשות?

שלב שני: ניסוי

לאחר שמיפינו את נקודות הקצה הרלוונטיות, נבדוק תחילה כיצד הן מתנהגות במצב רגיל.

לאחר מכן נתחיל לנסות בעזרת ה-Repeater לשלוח מספר בקשות כסדרת בקשות, ואז גם ננסה גם לשלוח במקביל, וננסה למצוא רמזים בתשובות המתקבלות.

כל דבר יכול להיות רמז, למשל זמני תגובה שונים ושינויים בהתנהגות אפליקציה עצמה, נרצה לאתר את אלו כדי לנסות ולנצל את זה לטובתנו.

שלב שלישי: הוכחת יכולת

ננסה להבין מה קורה בפועל, נסיר בקשות מיותרות וננסה לשחזר את ההשפעה על האפליקציה.

לרוב ההשפעה המירבית לא תהיה כל ברורה מאליו ולכן החוקרים של PortSwigger ממליצים להתייחס לכל מצב מירוץ כחלק מסדרה ולא כחולשה מבודדת.

לפעמים חימום החיבור כפי שראינו לא עוזר לנו, אך גם לזה יש פתרונות. לרוב שרתי WEB יוצרים השהיות בעיבוד הבקשות אם נשלחות יותר מדי כאלו בזמן צפוף מדי, נוכל לעקוף את זה על ידי שליחת מספר גדול של Dummy Requests, אלו בקשות סתמיות ללא משמעות, ובכך נעורר באופן מכוון את מנגנון ההשהייה בשרת, וכעת התקפות single packet יכולות לעבוד למרות שזמן הביצוע שלהן עם השהייה.

race conditions בנקודת קצה אחת

העמסת נקודת קצה אחת במספר גדול של בקשות יכולה לעורר מצב מירוץ, נניח במנגנון של איפוס סיסמא השומר את מזהה המשתמש ואת הטוקן שלו בסשן של אותו משתמש.

במקרה הזה שליחת שתי בקשות מאותו הסשן אך עם שתי משתמשים שונים יכולה להוביל להתנגשות כך שטוקן איפוס יישלח לתוקף במקום לקורבן.

בשביל שמתקפות כאלו יעבדו, על הפעולות להתבצע בדיוק בסדר הנכון, לכן לפעמים צריך לבצע את הפעולה מספר פעמים ועם קצת מזל נשיג את המטרה הרצויה.

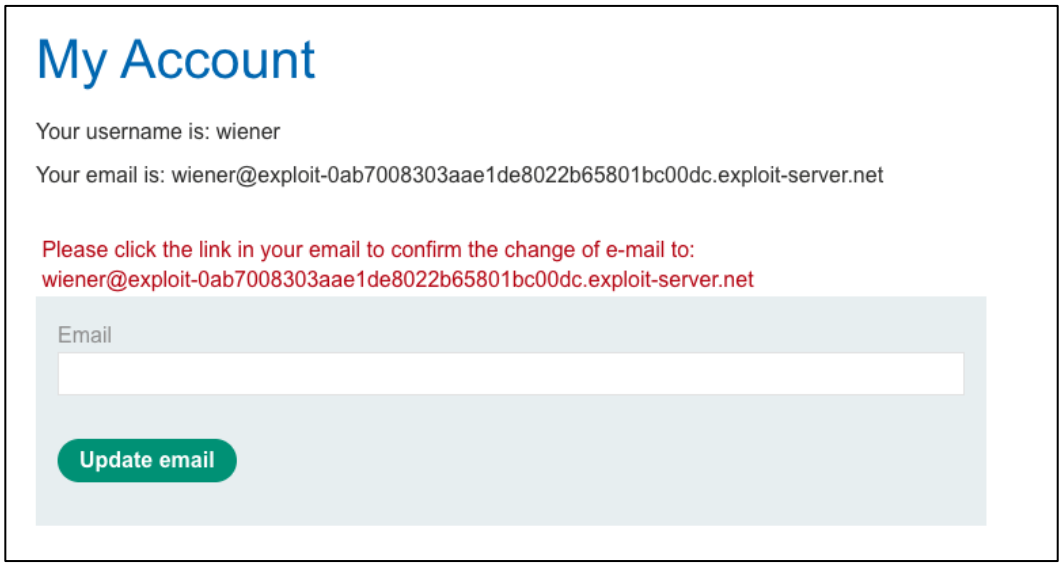
פעולות המבוססות על מייל כמו אימות שנשלח וכדומה יהיו קרקע טובה למתקפה מאחר ולרוב הן קורות ב-Thread נפרד ולאחר שכבר עובדה הבקשה בשרת.

במעבדה הבאה נשלחה הזמנה להיות אדמיניסטרטור באתר לכתובת המייל:

carlos@ginandjuice.shop

אך עדיין לא נוצר חשבון באתר עם המייל הזה, המטרה שלנו לקשר את המייל למשתמש שלנו, לקבל יכולות של אדמיניסטרטור באתר ולמחוק את היוזר carlos.

בשלב הראשון נרצה למצוא נקודת קצה קריטית לתקיפה.



לאחר ההתחברות למשתמש שקיבלנו, בעמוד הפרופיל יש אפשרות לעדכון המייל, ננסה לנתח איך התהליך מתנהג במצב רגיל.

Sent	To	From	Subject	Body	
2025-10-08 12:34:18 +0000	wiener@exploit-0ab7008303aae1de8022b65801bc00dc.exploit-server.net	no-reply@0a9d0059031ee1d78037b77d006b0035.web-security-academy.net	Please confirm your e-mail	To confirm your email change to wiener@exploit-0ab7008303aae1de8022b65801bc00dc.exploit-server.net, click the link below Click here to confirm.	View raw

לאחר עדכון כתובת המייל בפרופיל באתר, נדרש אישור נוסף במייל כפי שמופיע מעלה, לאחר האישור מקבלים הודעה שהכתובת עודכנה.

```
Your email has been successfully updated
```

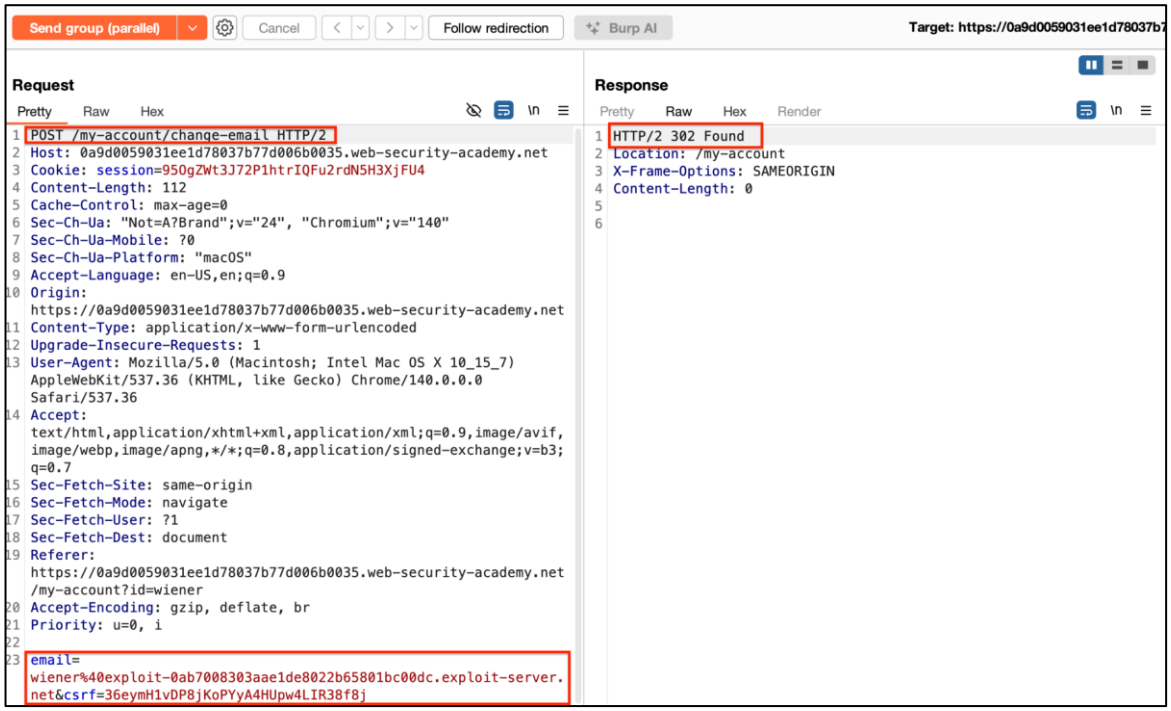
ננסה לחקור את הבקשה שנשלחת לנקודת הקצה מהאתר, נבדוק את התוכן שלה וננסה לדלות משם רמזים.

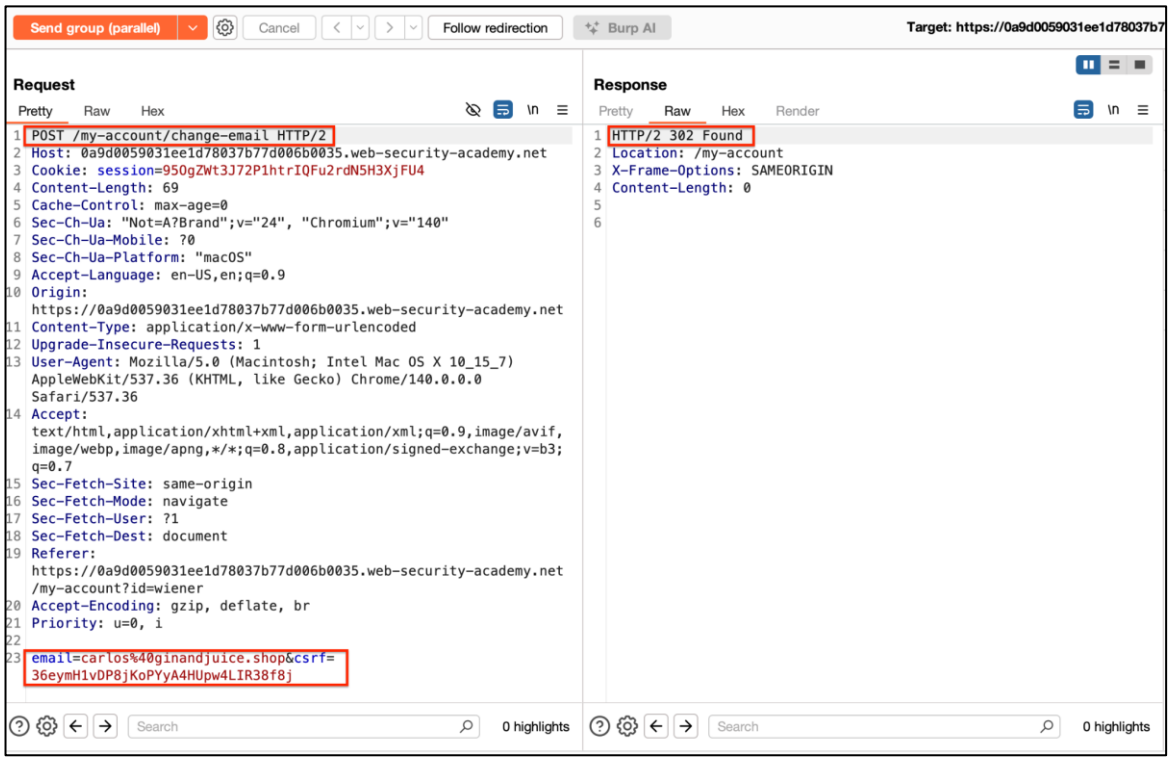
Request

Pretty Raw Hex

```
1  POST /my-account/change-email HTTP/2
2  Host: 0a9d0059031ee1d78037b77d006b0035.web-security-academy.net
3  Cookie: session=95OgZWt3J72P1htrIQFu2rdN5H3XjFU4
4  Content-Length: 112
5  Cache-Control: max-age=0
6  Sec-Ch-Ua: "Not=A?Brand";v="24", "Chromium";v="140"
7  Sec-Ch-Ua-Mobile: ?0
8  Sec-Ch-Ua-Platform: "macOS"
9  Accept-Language: en-US,en;q=0.9
10 Origin:
   https://0a9d0059031ee1d78037b77d006b0035.web-security-academy.net
11 Content-Type: application/x-www-form-urlencoded
12 Upgrade-Insecure-Requests: 1
13 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0
   Safari/537.36
14 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
   image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;
   q=0.7
15 Sec-Fetch-Site: same-origin
16 Sec-Fetch-Mode: navigate
17 Sec-Fetch-User: ?1
18 Sec-Fetch-Dest: document
19 Referer:
   https://0a9d0059031ee1d78037b77d006b0035.web-security-academy.net
   /my-account?id=wiener
20 Accept-Encoding: gzip, deflate, br
21 Priority: u=0, i
22
23 email=
   wiener%40exploit-0ab7008303aae1de8022b65801bc00dc.exploit-server.
   net&csrf=36eymH1yDP8iKoPYvA4HUpw4LIR38f8i
```

אז מצאנו שכתובת המייל נמצאת בתוך הבקשה הנשלחת לנקודת הקצה, וגם שקיים סשן מסוים למשתמש שמחובר, ננסה לבדוק מה יקרה אם נשלח שתי בקשות במקביל עם אותו הסשן אבל כתובת מייל שונה, אחת תהיה של המייל שאליו נרצה לשנות והשניה תהיה כתובת המייל אותה רצינו לשייך למשתמש שלנו במטרת המעבדה.

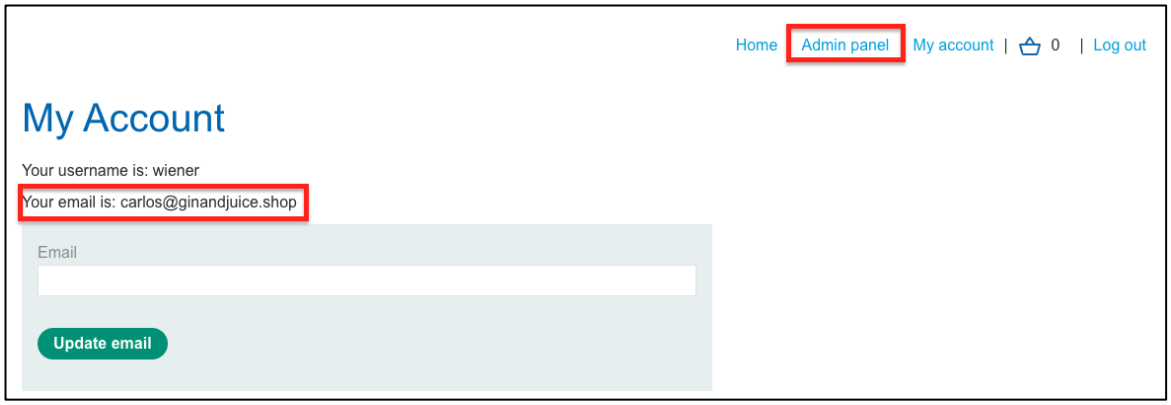




לפי תגובת השרת שתי הבקשות עובדו בהצלחה, נבדוק את תיבת המייל כעת.

Sent	To	From	Subject	Body
2025-10-08 12:44:37 +0000	wiener@exploit-0ab7008303aae1de8022b65801bc00dc.exploit-server.net	no-reply@0a9d0059031ee1d78037b77d006b0035.web-security-academy.net	Please confirm your e-mail	To confirm your email change to carlos@ginandjuice.shop, click the link below Click here to confirm. View raw

קיבלנו מייל לאישור של הכתובת החדשה אותה רצינו לייחס בשונה מהמייל הראשוני שקיבלנו לאישור כשבדקנו את התהליך. נאשר את החלפת המייל ונחזור לפרופיל המשתמש באתר.



אכן הצלחנו להשתלט על כתובת המייל של קרלוס וכעת נפתח לנו גם פאנל האדמין שם ניתן למחוק משתמשים, נגלוש לפאנל ונסיר את המשתמש של קרלוס.

סוגי מצבי מרוץ

- TOCTOU (Time-of-Check to Time-of-Use) - מצב בו תוכנה בודקת משאב מסוים, למשל קובץ או הרשאה, ומשתמשת בו מאוחר יותר בהנחה שהוא עדיין נכון, אך תוקף יכול להחליף או לשנות אותו בטווח הזמן הזה.
- Data race - המצב קורה כאשר שתי תהליכים או יותר מנסים לגשת לאותו חלק בזיכרון בו זמנית ולפחות אחד מהם כותב אליו, המצב מוביל להתנהגות לא צפויה ויכול להשחית את התוצאה.
- Race in file operations - כאשר מספר תהליכים מנסים ליצור, לפתוח או למחוק קובץ מסוים בו זמנית, מה שיכול להוביל לדריסה, הסלמת הרשאות (Privilege Escalation) או דליפת מידע רגיש.
- Race in privilege escalation - קורה כאשר המערכת משנה הרשאות, למשל החלפה בין משתמש רגיל למשתמש root, והפעולה לא מתבצעת בסנכרון מתאים, תוקף יכול לנסות בזמן הזה לבצע פעולות המצריכים הרשאות גבוהות יותר.

- Race in authentication or session handling - כשהמערכת מאמת זהות מסוימת או טוקן לסשן, אבל לא נועלת (נפרט על נעילות בהמשך) את המשאב בהתאם ומאפשרת לתוקף לעשות שימוש חוזר או לחטוף את הסשן לפני שפג תוקפו.
- Race in resource allocation - כשמספר משתמשים או תהליכים מנסים לתפוס משאב מוגבל מסוים, למשל בלוקים בזכרון או Sockets, והמערכת לא מאלצת נעילות מתאימות, זה יכול להוביל ל-DOS או דליפת מידע.

עוד במהלך הפיתוח כדאי לשים לב ולהעריך את הסיכויים להיווצרות אחד מהמצבים האלו, לעיתים קשה לשחזר לאחר הפיתוח אך תכנון נכון של הלוגיקה של תהליכים יכול להקטין את הסיכוי שיקרו.

הגיע הזמן לנתח ביחד חולשה אמיתית שהתגלתה

אם חשבתם ש-race conditions קיימים רק בעולמות ה-Web, הרשו לי להפתיע אתכם, יוני שנת 2024, מיקרוסופט משחררת Patch לחולשה עם חומרה גבוהה (7.0) בגרעין של ווינדוס (Windows Kernel), חולשה שלימים הוכח שנוצלה על ידי על ידי קבוצת תוקפים בשם APT3.

הכירו את חולשה:CVE-2024–3008, אך לפני שנצלול לניתוח החולשה נבין קודם כמה מושגים, הראשון הוא Kernel. ה-Kernel הוא הגרעין של מערכת ההפעלה, חלק מרכזי האחראי על ניהול משאבי המחשב המשמש כמעין גשר בין החומרה לתוכנה. הגרעין אחראי על שליטה בניהול המעבד, הזיכרון, רכיבים שמחוברים למחשב, איזה תהליכים רצים ומתי ומנהל את הגישה לקבצים, חלק מתפקידיו גם לספק גבולות אבטחה מסוימים. אפשר להתייחס לגרעין כאל "המוח" של מערכת ההפעלה שדואג שהכל ירוץ חלק ובאופן בטוח.

אז אחרי שהבנו מה תפקידו של הגרעין, נעבור לניתוח החולשה. החולשה נובעת ממצב של כשל מסוג TOCTOU, מה שהוביל את החוקרים למצוא שניתן לבצע הסלמת הרשאות. במקרה הנ"ל הגרעין מוודא את המצב של המשאב, למשל כתובת בזיכרון או חוצץ מסוים (Buffer), ולאחר מכן עובר להשתמש בו. אך ייתכן ומצב הזה התעדכן או שונה בפרק הזמן הקצר הזה ובמיוחד שהמשאב נשלט על ידי תהליך בשליטת המשתמש, מה שפותח דלת לתוקפים להשיג שליטה ברמת מערכת או להזריק מידע זדוני. מהבדיקה עולה כי רצף הפעולות קרה באופן הבא: הגרעין בודק את גודל ופריסת החוצץ המסופק על ידי המשתמש, ומתחיל להעתיק מידע ממנו (למשל שמות, טיפוסים, ערכים ועוד). בפועל, בין בדיקת המצביע לחוצץ וכתיבת המידע, לא מתבצעת נעילה כראוי של המשאב, מה שיוצר חלון מירוץ קלאסי לתוקפים.

תהליכים זדוניים עלולים:

1. להעביר מצביע תקף בזמן הבדיקה.
2. בזמן החלון, לשנות את המצביע או את התוכן.
3. להחליף את מיקום הכתיבה למיקום בשליטת התוקף.

מה ששובר את הבידוד בין הגרעין למשתמש ונותן לתוקף למעשה שליטה ברמת הגרעין.

מבין המערכות שנפגעו היו גרסאות של ווינדוס 10, ווינדוס 11 ובפוטנציאל גם גרסאות שונות של שרתים החולקים גרעין בסיסי משותף עם מערכות אלו.

ב-Patch ששוחרר מיקרוסופט התייחסו לנושאים הבאים, הוספת ולידציה נוספת מפורשת למצביעים שמגיעים ממשתמש טרם ביצוע כתיבה, הוספת נעילות זיכרון מתאימות למניעת מצבי המירוץ ועוד.

וכמו שאנחנו כבר יודעים, תוקפים זדוניים לא מחכים יותר מדי, ומהר מאוד חוקרים מ-Trend Micro קישרו ניצול בפועל של החולשה לקבוצת התקיפה APT34.

שיטת הפעולה שלהם הייתה למצוא תשתיות קריטיות בתחומי האנרגיה, הגז ועוד, להשתמש ב- Multi Stage Implants הכוללים בתוכם דלתות אחוריות וגונבי טוקנים, לחפש עקביות באמצעות שינויים ב-Registry וניצול לרעה של כלי מובנים במערכת ההפעלה, נוסף על כך שילבו את התקיפה עם מתקפות phishing נרחבות כדי ליצור את הכניסה הראשונית למערכות.

הקבוצה מנצלת את מערכות שלא עודכנו באופן אגרסיבי, מה שמדגיש את החשיבות של ביצוע עדכונים בזמן.

אז איך מתגוננים ממצבים כאלו?

לפני שניכנס לעובי הקורה, חשוב להבין מספר מושגים, הראשון יהיה קטע קריטי.

קטע קריטי הוא חלק בקוד בו מתבצעת גישה או שינוי של משאב משותף, למשל משתנה בזכרון או קובץ.

השני יהיה פעולה אטומית, אטומית הכוונה שפעולה מתרחשת בשלמותה או לא מתרחשת כלל.

והשלישי יהיה נעילה, מטרתה של הנעילה להגן על הקטע הקריטי בכך שהיא מבטיחה שרק תהליך אחד יכול להיכנס לקטע בכל רגע נתון, מה שייצור סינכרוניזציה בין תהליכים ומונע פגיעה בעקביות הנתונים.

נניח למשל מערכת שמנהלת חשבון בנק, כערך התחלתי יש בחשבון יתרה של 100 ש"ח. במערכת יצרנו פונקציה בשם update_balance שתפקידה לעדכן את היתרה בכל פעולת הפקדה או משיכה מהחשבון בהתאם לסכום שהיא מקבלת כפרמטר (הפקדה תהיה עם מספר חיובי, ומשיכה תהיה עם מספר שלילי), היא תיראה בערך ככה:

```
update_balance(amount):
        current_balance+=balance
```

כעת שתי תהליכים ניגשים לפונקציה זו זמנית, האחד ימשוך 50 ש"ח והשני יפקיד 100 ש"ח, מה תהיה היתרה בסוף הפעולות? אז מאחר ואין אנו יכולים לדעת מי ביצע את הפעולה קודם, התוצאה תהיה לא צפויה וייתכן מקרה בו משיכת הכסף מהחשבון תהיה על היתרה המקורית במקום על המעודכנת בסוף נקבל שהיתרה בחשבון תהיה 950 ש"ח כשזו אינה היתרה האמיתית. עם נעילה מסביב לפונקציה הזו, רק תהליך אחד ייגש לחשבון, יעדכן את היתרה בהתאם, והתהליך השני יבצע את הפעולה שלו על יתרה מעודכנת. הנעילה מבטיחה לנו רצף פעולות תקין גם כשיש ריבוי תהליכים.

קיימים מספר סוגי נעילה שונים:

- Mutex - הסוג הפשוט ביותר, רק תהליך אחד יכול להחזיק בנעילה ולשחרר אותה בכל רגע נתון, אחד נעל ורק הוא יכול לפתוח.
- Semaphore - כאשר מספר מוגבל מראש של תהליכים צריכים לגשת למשאב מסוים בו זמנית, ניתן להתייחס לMutex כמקרה פרטי של Semaphore כשהמספר המוגבל הוא 1, שימושי עבור הגבלת משאבים כמו חיבורי רשת.
- Read-Write Lock - מאפשר להרבה קוראים לגשת למשאב אך הכתיבה יכולה להתבצע רק כאשר כל הקוראים מסיימים, שימושי במצבים בהם יש קריאה מרובה אך העדכונים מעטים.
- Spinlock - גורם לתהליך להמתין לשחרור המנעול באמצעות לולאה במקום "לישון", יכול להקטין את זמן ההמתנה אך גם "לבזבז" הרבה זמן מעבד אם ההמתנה ארוכה.
- Reentrant lock - נעילה רקורסיבית המאפשרת לאותו התהליך לנעול שוב את המשאב מבלי להוביל למבוי סתום, שימושי כאשר פונקציה מסוימת צריכה להפעיל פונקציה אחרת מתוכה אף מאחר והן מגיעות באותו התהליך עם מנעול אחר היא תחסם, למשל פונקציה אחת פותחת את הקובץ ומפעילה פונקציה שמעדכנת נתונים בתוכו.

שיטה נוספת להתגונן תהיה תכנון קוד ללא שיתוף מצב (shared state), כלומר כל תהליך עובד על עותק משלו של הנתונים וכך לא צריך לעשות סינכרוניזציה ביניהם. זהו עקרון נפוץ בתכנות פונקציונלי ומערכות מבוססות message passing כמו Actor model, כל "שחקן" מחזיק מצב פנימי ומתקשר עם "שחקנים" אחרים באמצעות הודעות.

נבחן ביחד תרחיש מציאותי

בתרחיש הזה יש לנו מנעול חכם לדלת שאפשר לפתוח דרך אפליקציה, ובמקביל רץ עליו מנגנון נעילה אוטומטית אחרי כמה דקות. הקוד מחזיק משתנה משותף lock_state שמייצג את מצב הדלת, ומשתמש בשתי פונקציות שונות: אחת שפותחת את הדלת דרך האפליקציה, ואחת שסוגרת אותה אוטומטית.

```
lock_state = "locked"  # shared state in memory
def unlock_via_app(user_id):
    # Step 1: check if user is allowed to unlock
    if has_access(user_id):
        global lock_state
        # Step 2: update internal state
        lock_state = "unlocked"
        # Step 3: send command to the physical lock
        send_command_to_lock("UNLOCK")
        return "door unlocked"
    else:
        return "access denied"
def auto_lock():
    global lock_state
    # Step 1: check current state
    if lock_state == "unlocked":
        # Step 2: update internal state
        lock_state = "locked"
```

הבעיה מתחילה כששתי הפעולות האלו רצות כמעט באותו הזמן. למשל, בדיוק כשהמשתמש פותח את הדלת דרך האפליקציה, תהליך הנעילה האוטומטי נכנס לפעולה. אחת הפונקציות קוראת את המצב כ-"unlocked" ומנסה לנעול, השנייה משנה ל-"unlocked" ושולחת פקודת פתיחה.

בגלל שאין כאן פעולה אטומית אחת שמבטיחה סדר ברור, ייתכן מצב שבו המנעול הפיזי יישאר פתוח בזמן שהמערכת חושבת שהוא נעול, או להפך.

זהו Race Condition קלאסי במכשיר חכם, חוסר תיאום בין כמה תהליכים גורם לכך שמצב האמת בעולם הפיזי והמצב הלוגי בקוד מתנתקים אחד מהשני, ומהר מאוד הופכים לחולשה לוגית עם השלכות אבטחה מאוד לא נעימות.

איך מתגברים על זה? נשתמש במנעול ונכריח את המערכת לעבוד בצורה שרק פעולה אחת יכולה לגשת למצב המנעול בכל רגע נתון, כלומר שאם תהליך אחד מחזיק את הנעילה, כל שאר התהליכים לא יוכלו להתבצע עד שהוא ישחרר את המנעול.

בכך אנחנו מעלימים את החלון הקטן שנוצר בין שתי פעולות קוראות וכותבות בו זמנית. מנגנונים מסוג זה נפוצים במכשירים חכמים ומערכות IoT בהן תזמון תהליכים קריטי וחשוב למניעת בלאגן גדול.

```python
from threading import Lock
state_lock = Lock()
lock_state = "locked"
def unlock_via_app(user_id):
    if has_access(user_id):
        global lock_state
        # Acquire the lock before touching shared state
        with state_lock:
            lock_state = "unlocked"
            send_command_to_lock("UNLOCK")
        return "door unlocked"
    return "access denied"
def auto_lock():
    global lock_state
    # Acquire the lock before reading and updating
    with state_lock:
        if lock_state == "unlocked":
            lock_state = "locked"
            send_command_to_lock("LOCK")
```

סיכום

לסיכום, ניתן להשתמש גם בפעולות אטומיות כפי שצוין קודם ומבני נתונים יעודיים שנקראים lock-free structures, מבנים אלו מבטיחים שהפעולה תתבצע בשלמותה מבלי התנגשויות בין התהליכים, מתאים בדרך כלל לפעולות פשוטות כמו עדכון מונה או דגל.

במערכות וב המצב קצת שונה, המירוץ קורה בעיקר בין בקשות HTTP שונות המגיעות במקביל, לצורך כך ניתן להשתמש במנגנוני idempotent keys, מזהים חד פעמיים לכל בקשה (למשל בעת תשלום) כך שגם אם אותה הבקשה נשלחת פעמיים רק אחת מהן תתבצע בפועל.

כמו כן גם שימוש ב-database transactions או row-level locking מבטיחים כי רק תהליך אחד משנה את השורה בכל פעם.

יש גם גישות נוספות כמו למשל optimistic concurrency control, בגישה הזאת הבקשות נשלחות כרגיל אך לפני ביצוע העדכון המערכת בודקת אם הנתונים לא השתנו מאז שנקראו, אם בוצע שינוי הבקשה נדחית או נשלחת שוב.

ולבסוף, ניתן להשתמש גם בשכבת היישום עצמה, שימוש בתורים או מערכות הודעות למיניהם מאפשרות ניהול סדר מסוים לבקשות מה שיכול למנוע מקביליות בטיפול בבקשות.

בשורה התחתונה, אין פתרון אחד שמתאים לכל מערכת, תכנון ושילוב נכון בין השיטות והפתרון הוא הדרך האידיאלית להגן על הקוד והמערכת מrace conditions בזמן אמת.

על המחבר

אז למי שעדיין לא מכיר, אני דוד סטרינסקי סטודנט למדעי המחשב, והנדסאי תוכנה עם התמחות בסייבר.

מדריך בקורס רשתות תקשורת בתוכנית מגשימים וחובב טכנולוגיה. משתדל להישאר בתנועה ולהרחיב את הידע שלי בתחום ושמח תמיד לשמוע וללמוד דברים חדשים, אז מי שמעוניין לשמור על קשר בלינקדאין מוזמנים לשלוח קונקשן ונדבר: https://www.linkedin.com/in/david-starinsky/

מקורות מידע

- https://cwe.mitre.org/data/definitions/362.html
- https://cwe.mitre.org/top25/archive/2024/2024_key_insights.html
- https://www.hackerone.com/blog/how-race-condition-vulnerability-could-cast-multiple-votes
- https://www.bugcrowd.com/wp-content/uploads/2024/01/Inside-the-Platform-Vulnerability-Trends-Report.pdf
- https://portswigger.net/web-security/race-conditions
- https://portswigger.net/research/smashing-the-state-machine
- https://nvd.nist.gov/vuln/detail/cve-2024-30088
- https://medium.com/@shira.borochovich/cve-2024-30088-kernel-level-toctou-vulnerability-abused-by-apt34-for-privilege-escalation-in-5a75035bf076
- https://hackerone.com/reports/2598548
- https://www.guidepointsecurity.com/blog/race-conditions-in-modern-web-applications
- https://www.bugcrowd.com/blog/racing-against-time-an-introduction-to-race-conditions