**Race conditions**

Race conditions là 1 loại lỗ hổng phổ biến liên quan đến lỗi logic. Chúng xảy ra khi các trang web xử lí các yêu cầu đồng thời mà không có biện pháp bảo vệ thích hợp. Điều này có thể dẫn đến nhiều luồng khác nhau tương tác với cùng 1 dữ liệu trong cùng 1 thời gian, dẫn đến 1 va chạm sinh ra các hành vi không mong muốn của ứng dụng. Một cuộc tấn công race condition sử dụng các yêu cầu được định thời gian cẩn thận để gây ra va chạm có chủ ý và khai thác hành vi không mong muốn này cho các mục đích xấu xa.



Khoảng thời gian có thể xảy ra va chạm là race window. Điều này có thể là 1 phần của giây giữa hai tương tác với cơ sở dữ liệu.

Giống như lỗi logic khác, tác động của race condition phụ thuộc nhiều vào ứng dụng và chức năng mà nó xảy ra.

Trong mục này, bạn sẽ học cách để xác định và khai thác các loại race condition khác nhau. Chúng tôi sẽ dạy bạn cách các công cụ được dựng sẵn trong Burp Suite có thể giúp bạn vượt qua các thử thách mà thực hiện tấn công cổ điển, cộng với một phương pháp đã được thử nghiệm cho phép bạn phát hiện các loại race condition trong quá trình nhiều bước ẩn. Những điều này vượt xa các lần vượt quá giới hạn mà bạn có thể đã quen thuộc.

**Limit overrun race conditions**

Loại race condition được biết đến nhiều nhất cho phép bạn vượt quá một số loại giới hạn do logic của ứng dụng áp đặt.

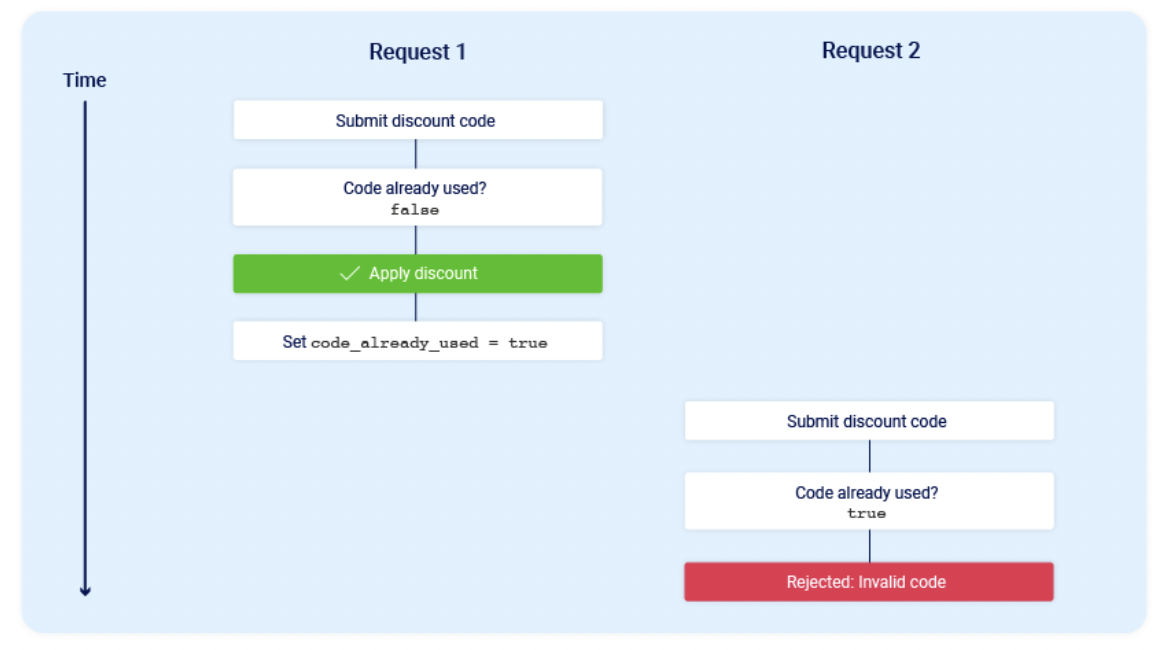
Ví dụ, xem xét 1 cửa hàng online cho phép bạn nhập 1 mã code trong suốt quá trình thanh toán để nhận giảm giá 1 lần trên đơn đặt hàng của bạn. Để áp dụng giảm giá này, ứng dụng có thể thực hiện các bước sau:

1. Kiểm tra xem bạn đã sử dụng code chưa

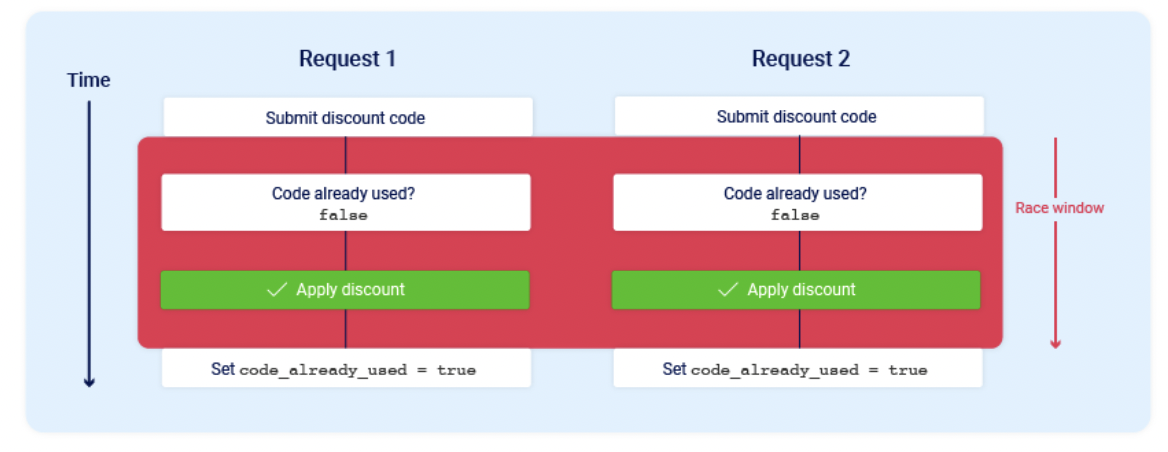
2. Áp dụng giảm giá cho tổng đơn hàng

3. Chỉnh sửa record trong cơ sở dữ liệu để biết bạn đã dùng code rồi

Nếu sau đó bạn sử dụng lại code, bước kiểm tra ban đầu sẽ ngăn chặn bạn làm điều đó:



Bây giờ xem xét điều gì sẽ xảy ra nếu 1 người dùng chưa bao giờ áp dụng mã giảm giá này trước khi cố gắng áp dụng nó hai lần gần như chính xác cùng 1 thời gian:



Như bạn có thể thấy, ứng dụng chuyển tiếp qua 1 trạng thái con tạm thời; nó là 1 trạng thái mà nó nhập vào và sau đó thoát ra lần nữa trước khi quá trình xử lí yêu cầu hoàn tất. Trong trường hợp đó, trạng thái con bắt đầu khi máy chủ bắt đầu xử lí yêu cầu đầu tiên và kết thúc khi nó chỉnh sửa cơ sở dữ liệu để chỉ ra rằng bạn đã sử dụng code đó. Điều này đem lại 1 race window nhỏ trong đó bạn có thể liên tục yêu cầu giảm giá bao nhiêu lần tùy thích.

Có rất nhiều biến thể của loại tấn công này:

* Đổi thẻ quà tặng nhiều lần
* Đánh giá 1 sản phẩm nhiều lần
* Rút hoặc chuyển tiền mặt vượt quá số dư tài khoản của bạn
* Sử dụng lại CAPTCHA
* Bỏ qua 1 bộ lọc giới hạn tỉ lệ chống brute-force

Limit overruns là 1 loại phụ của cái gọi là lỗi “time-of-check to time-of-use”. Ở phần sau của chủ đề này, chúng ta sẽ xem các ví dụ của lỗ hổng race condition không rơi vào hai trường hợp trên.

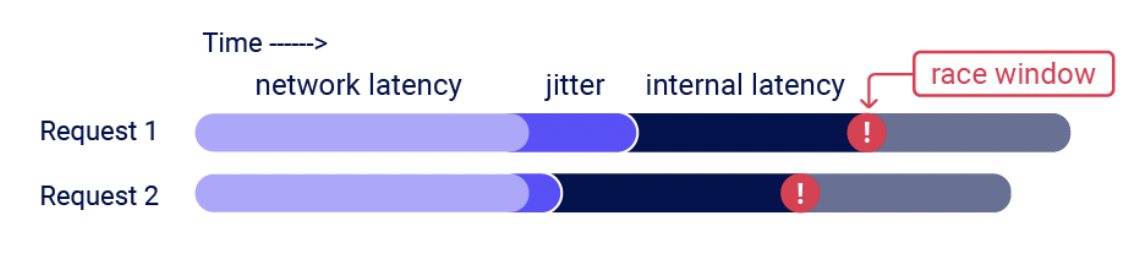
**Phát hiện và khai thác limit overruns race conditions với Burp Repeater**

Quá trình phát hiện và khai thác limit overrun race conditions khá đơn giản. Tất cả những việc bạn cần làm là:

* Xác định 1 điểm vào sử dụng 1 lần hoặc bị giới hạn lượt sử dụng
* Gửi nhiều yêu cầu đến điểm vào đó liên tiếp nhanh chóng để xem bạn có thể vượt qua giới hạn đó không

Thách thức chính là thời gian các yêu cầu để mà ít nhất hai race window xếp hàng, gây ra va chạm. Window này chỉ thường vài mili giây thậm chí có thể ngắn hơn.

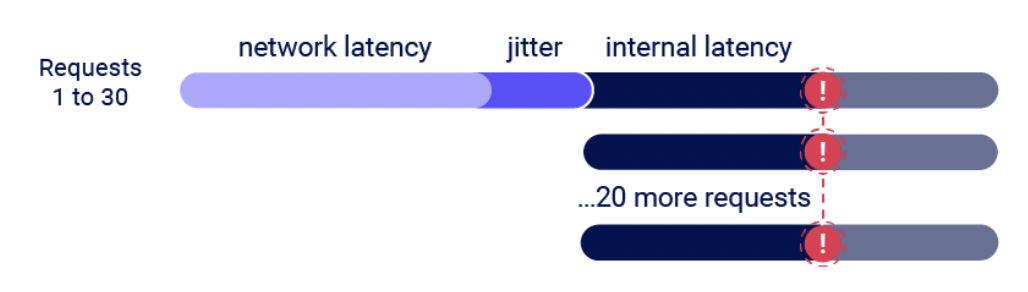
Thậm chí nếu bạn gửi tất cả yêu cầu cùng 1 lúc, trong thực tế có nhiều yếu tố bên ngoài không thể kiểm soát và dự đoán được ảnh hưởng khi máy chủ xử lí từng yêu cầu và theo thứ tự nào.



Burp Suite 2023.9 đã thêm 1 tính năng mới đến Burp Repeater cho phép bạn dễ dàng gửi 1 nhóm các yêu cầu song song theo 1 cách mà nó giảm sự tác động của các yếu tố bên ngoài như network jitter. Burp tự động điều chỉnh kĩ thuật để phù hợp với phiên bản của http được hỗ trợ bởi máy chủ:

* Đối với HTTP/1, nó dùng kĩ thuật đồng bộ byte cuối cổ điển.
* Đối với HTTP/2, nó dùng kĩ thuật tấn công từng gói, lần đầu được mô tả bởi PortSwigger Research ở Black Hat USA 2023.

Tấn công từng gói cho phép bạn vô hiệu hóa hoàn toàn sự can thiệp từ network jitter bằng cách dùng 1 gói TCP để hoàn thành 20-30 yêu cầu cùng 1 lúc.



Mặc dù bạn chỉ cần dùng hai yêu cầu để thực thi 1 khai thác, việc gửi 1 lượng lớn các yêu cầu như vậy sẽ giúp giảm thiểu độ trễ bên trong như server-side jitter. Điều này đặc biệt hữu ích trong giai đoạn khám phá đầu tiên.

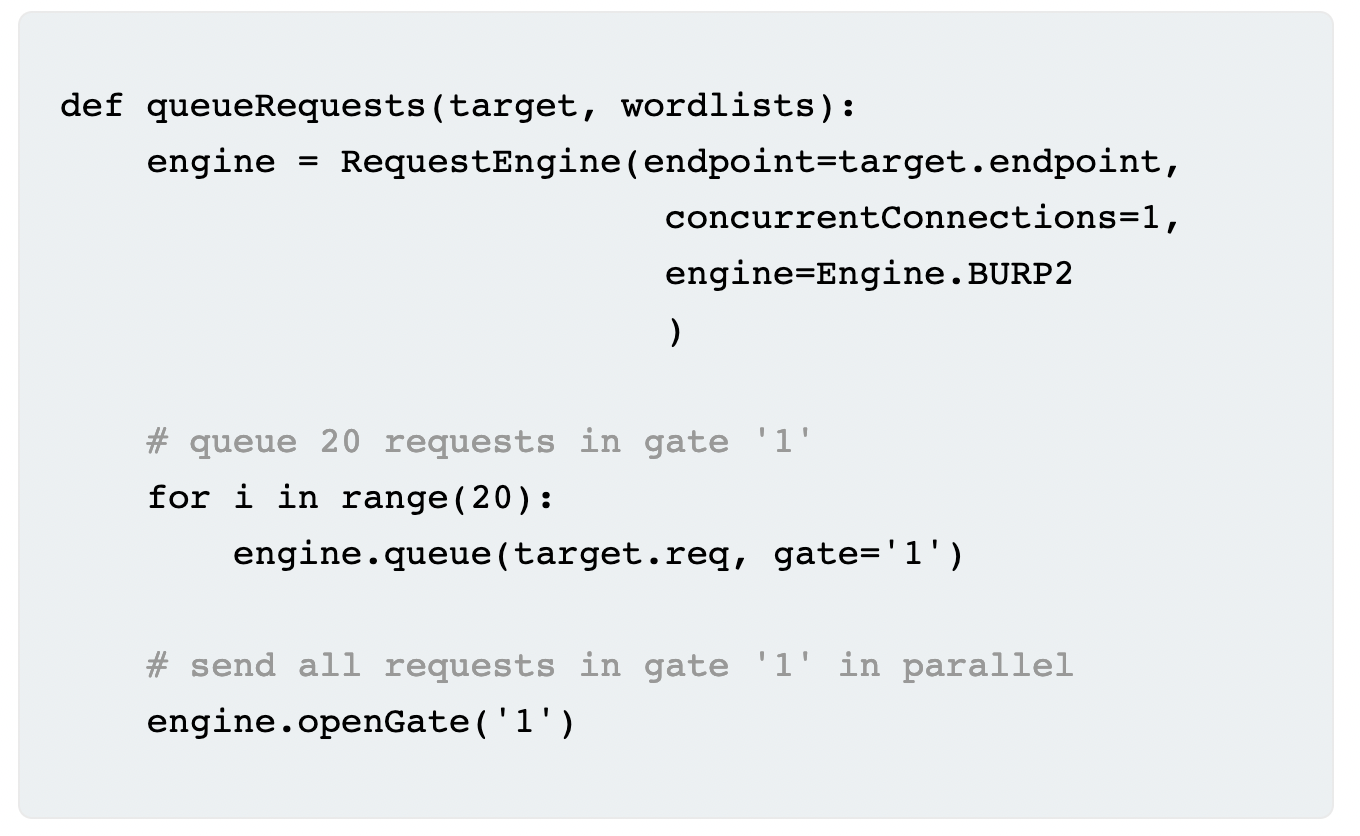
**Phát hiện và khai thác limit overrun race conditions with Turbo Intruder**

Ngoài việc tấn công single-packet trong Burp Repeater, chúng tôi cũng nâng cao Turbo Intruder extension để hỗ trợ kĩ thuật này. Bạn có thể tải phiên bản mới nhất từ Bapp Store.

Turbo Intruder yêu cầu thành thạo về Python, nhưng phù hợp với những tấn công phức tạp hơn như các cuộc tấn công yêu cầu thử lại nhiều lần, thời gian của mỗi yêu cầu so le hay số lượng request cực lớn.

Để sử dụng single-packet attack trong Turbo Intruder:

* Đảm bảo đối tượng hỗ trợ HTTP/2. Single-packet attack không tương thích với HTTP/1.
* Đặt lại cấu hình cho request engine engine=Engine.BURP2 và concurrentConnections=1
* Khi xếp hàng đợi các yêu cầu của bạn, nhóm chúng bằng cách chỉ định cho chúng 1 gate được đặt tên trong đối số gate của phương thức engine.queue()
* Để gửi tất cả yêu cầu trong 1 nhóm chỉ định, mở cổng tương ứng với phương thức engine.openGate()



Để hiểu chi tiết hơn, xem thư viện race-single-packet-attack.py được cung cấp trong thư mục mặc định của Turbo Intruder.

**Hidden multi-step sequences**

Trong thực tế, 1 yêu cầu đơn có thể bắt đầu 1 chuỗi nhiều bước, chuyển tiếp ứng dụng qua nhiều trạng thái ẩn mà nó đi vào và thoát ra lần nữa trước khi quá trình xử lí yêu cầu được hoàn tất. Chúng ta sẽ gọi nó là những trạng thái con.

Nếu bạn có thể xác định được một hoặc nhiều yêu cầu http gây ra sự tương tác với cùng dữ liệu, bạn có thể lợi dụng các trạng thái con để biết được nhiều time-sensitive của các loại lỗi logic thường phổ biến trong quy trình làm việc nhiều bước. Điều này cho phép khai thác race condition vượt xa limit overruns.

Ví dụ, bạn có thể thấy quen với lỗi của xác thực nhiều lớp (MFA) cho phép bạn thực hiện phần đăng nhập đầu tiên bằng tài khoản mình có, sau đó chuyển hướng thẳng đến ứng dụng, thành công vượt qua MFA.

Đoạn mã sau mô tả cách 1 trang web có thể bị dính lỗ hổng race:

session['userid'] = user.userid

if user.mfa\_enabled:

session['enforce\_mfa'] = True

# generate and send MFA code to user

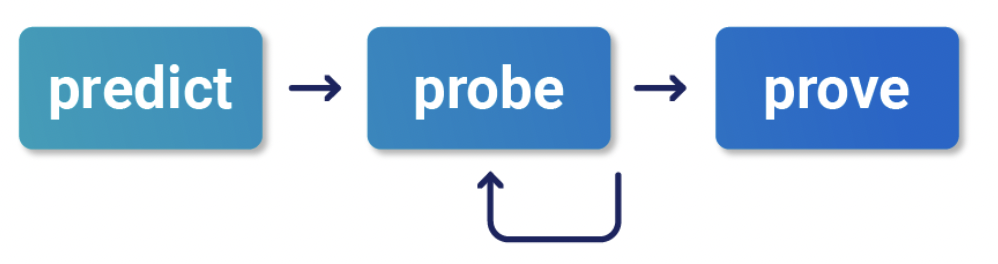
# redirect browser to MFA code entry form

Như bạn có thể thấy, trên thực tế, đây là một chuỗi gồm nhiều bước trong khoảng 1 yêu cầu. Quan trọng nhất, nó chuyển qua trạng thái phụ trong đó người dùng tạm thời có phiên đăng nhập hợp lệ nhưng MFA chưa được thực thi. Kẻ tấn công có thể khai thác lỗ hổng này bằng cách gửi 1 yêu cầu đăng nhập cùng với 1 yêu cầu đến 1 điểm vào nhạy cảm, đã xác thực.

Tuy nhiên, vì những lỗ hổng này khá cụ thể cho từng ứng dụng, điều quan trọng trước tiên là phải hiểu phương pháp rộng hơn bạn sẽ cần để áp dụng trong xác định các lỗ hổng, cả trong labs và thực tế.

**Phương pháp**

Để phát hiện và khai thác chuỗi nhiều bước bị ẩn, chúng tôi đề xuất theo phương pháp sau, được tổng kết từ trang “Smashing the state machine: The true potential of web race conditions” bởi PortSwigger Research.

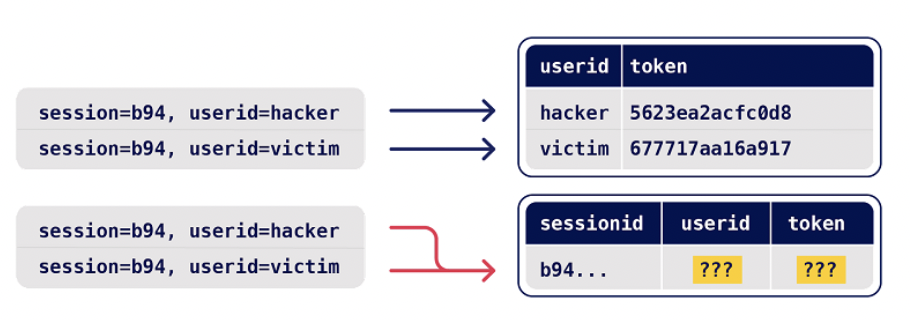


**1. Dự đoán các va chạm có thể xảy ra**

Kiểm tra mọi điểm cuối là không thực tế. Sau khi vạch ra trang mà mình nhắm đến như bình thường, bạn có thể giảm số lượng điểm cuối cần kiểm tra bằng cách tự hỏi mình những câu hỏi sau:

**Bảo mật điểm cuối này có quan trọng không?** Nhiều điểm cuối không đóng vai trò quan trọng nên không đáng để kiểm tra

**Có khả năng xảy ra va chạm không?** Để va chạm thành công, bạn cần 2 hay nhiều yêu cầu thực thi các hoạt động trên cùng 1 bản ghi. Ví dụ, xem xét các biến thể sau đây của việc triển khai đặt lại mật khẩu:



Với ví dụ đầu tiên, các yêu cầu đặt lại mật khẩu song song đối với hai người dùng khác nhau không gây ra va chạm vì nó dẫn đến những thay đổi đối với hai bản ghi khác nhau. Tuy nhiên, cách thực thi thứ hai cho phép bạn chỉnh sửa trên cùng 1 bản ghi với các yêu cầu đặt lại mật khẩu cho hai người dùng khác nhau.

**2. Thăm dò manh mối**

Để nhận ra các manh mối, đầu tiên bạn cần xem cách hoạt động của điểm cuối trong điều kiện bình thường có đúng với tiêu chuẩn không? Bạn có thể làm điều đó trong Burp Repeater bằng cách nhóm tất cả yêu cầu và dùng Send group in sequence (separate connections).

Tiếp theo, gửi 1 nhóm các yêu cầu trên bằng tấn công single-packet (hoặc last-byte sync nếu HTTP/2 không được hỗ trợ) để giảm thiểu network jitter. Bạn có thể làm điều này trong Burp Repeater bằng cách chọn Send group in parallel.

Bất cứ điều gì cũng có thể là manh mối. Chỉ cần tìm một số dạng sai lệch so với những gì bạn quan sát được trong quá trình đánh giá. Điều đó bao gồm sự thay đổi trong một hoặc nhiều phản hồi, nhưng đừng quên các ảnh hưởng phụ như nội dung thư khác nhau hoặc sự thay đổi rõ ràng trong hành vi của ứng dụng sau đó.

**3. Chứng minh ý tưởng**

Cố gắng hiểu những gì đang diễn ra, loại bỏ những yêu cầu không cần thiết và đảm bảo bạn có thể thực hiện lại ý tưởng đó.

Race condition nâng cao có thể làm cho các hành vi của ứng dụng trở nên khác thường và độc đáo vì vậy con đường dẫn đến tác động tối đa không phải lúc nào cũng rõ ràng. Có thể bạn sẽ thấy hữu ích khi coi mỗi race condition như 1 điểm yếu về cấu trúc chứ không phải là 1 lỗ hổng riêng lẻ.

**Race conditions nhiều điểm cuối**

Có lẽ hình thức trực quan nhất của race condition thì liên quan đến việc gửi yêu cầu đến nhiều điểm cuối cùng một lúc.

Nghĩ về lỗi logic kinh điển trong cửa hàng trực tiếp khi bạn thêm 1 vật vào trong giỏ hàng, thanh toán nó, sau đó thêm nhiều vật nữa vào giỏ hàng trước khi bị ép chuyển sang trang xác nhận đơn hàng.

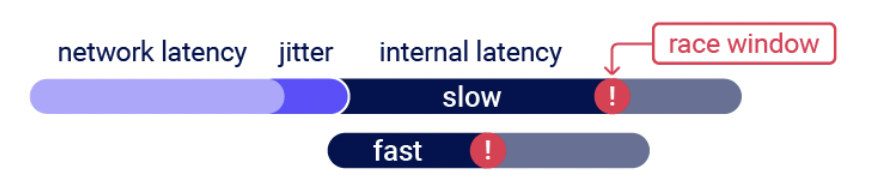
Một biến thể của lỗ hổng này có thể xảy ra khi xác nhận thanh toán và xác nhận đơn hàng được thực hiện trong quá trình xử lí 1 yêu cầu. Trạng thái của máy tính cho trạng thái của đơn hàng có thể nhìn giống như sau:



Trong trường hợp này, bạn có thể thêm nhiều vật vào trong giỏ hàng trong race window giữa lúc xác nhận thanh toán và khi đơn hàng được xác nhận

**Căn chỉnh race windows nhiều điểm cuối**

Khi kiểm tra race conditions nhiều điểm cuối, bạn có thể gặp phải sự cố khi cố gắng sắp xếp race windows cho từng yêu cầu, ngay cả khi bạn gửi tất cả yêu cầu vào cùng một lúc bằng kĩ thuật single-packet:



Vấn đề phổ biến này chủ yếu được gây ra bởi hai yếu tố sau:

* **Trì hoãn bởi kiến trúc mạng:** Ví dụ, sẽ có trì hoãn khi máy chủ front-end thiết lập kết nối đến back-end. Giao thức được sử dụng có thể có tác động lớn.
* **Trì hoãn bởi quá trình xử lí điểm cuối:** Các điểm cuối khác nhau sẽ có thời gian xử lí khác nhau, thường rất lớn dựa vào hoạt động mà chúng được thực thi.

**Connection warming**

Trì hoãn trong kết nối back-end thường không ảnh hưởng race conditions vì chúng thường trì hoãn các yêu cầu song song như nhau nên các yêu cầu vẫn được đồng bộ hóa.

Phân biệt được trì hoãn do các điểm cuối gây ra rất quan trọng. Một cách để làm điều này là làm nóng kết nối với một hoặc nhiều yêu cầu không quan trọng để xem điều này có làm trơn tru thời gian xử lí còn lại hay không. Trong Burp Repeater, bạn có thể thêm yêu cầu GET cho trang chủ vào đầu nhóm, sau đó dùng Send group in sequence (single connection).

Nếu yêu cầu đầu tiên có thời gian xử lí lâu hơn, nhưng phần còn lại của nhóm yêu cầu được xử lí trong 1 short window, bạn có thể bỏ qua độ trễ rõ ràng và kiểm tra lỗ hổng như bình thường.

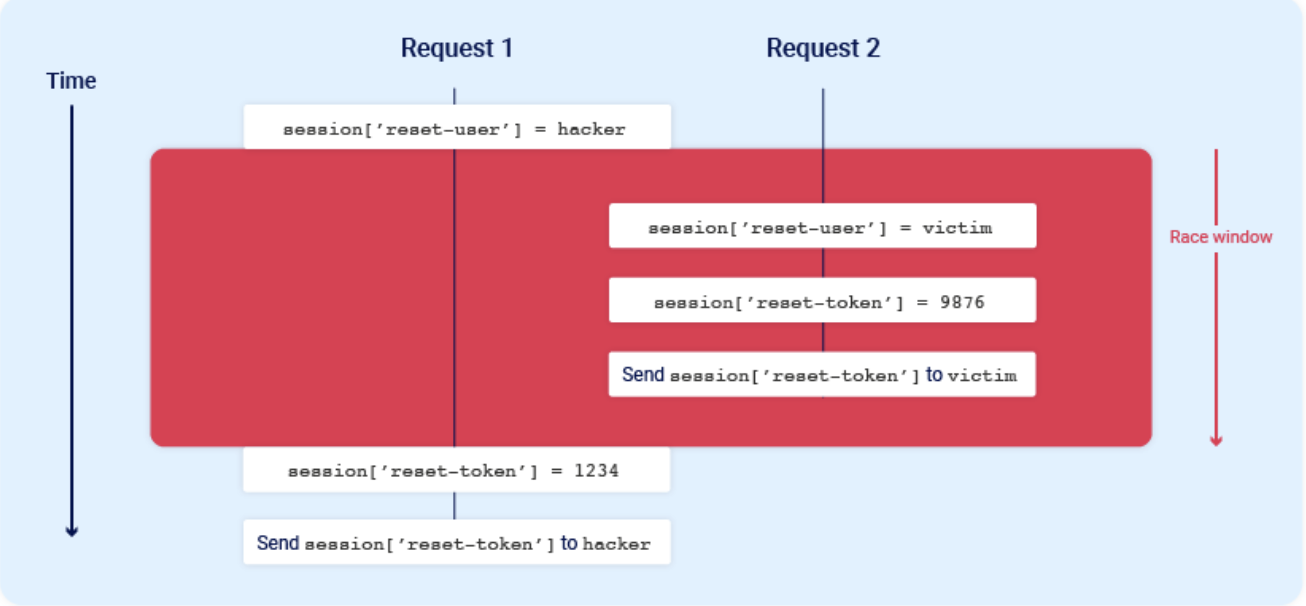
Nếu bạn vẫn thấy thời gian phản hồi không đồng nhất trên cùng 1 điểm cuối, thậm chí khi dùng kĩ thuật single-packet, đây là 1 bằng chứng cho thấy sự trì hoãn back-end ảnh hưởng đến race condition. Bạn có thể giải quyết vấn đề này bằng Turbo Intruder để gửi 1 vài yêu cầu để làm nóng kết nối trước khi tiếp tục các yêu cầu tấn công chính.

**Race conditions 1 điểm cuối**

Việc gửi các yêu cầu song song với các giá trị khác nhau đến 1 điểm cuối có thể kích hoạt race conditions dễ dàng.

Xem xét cơ chế đặt lại mật khẩu lưu user ID và token đặt lại trong phiên của người dùng.

Trong tình huống đó, gửi hai yêu cầu đặt lại mật khẩu song song từ cùng 1 phiên nhưng với hai tên người dùng khác nhau có thể gây ra sự va chạm sau:



Để ý trạng thái cuối cùng khi tất cả yêu cầu hoàn tất:

* session['reset-user'] = victim
* session['reset-token'] = 1234

Session bây giờ chứa user ID của nạn nhân, nhưng token đặt lại được gửi cho kẻ tấn công

Để tấn công này hoạt động, các hoạt động khác nhau được thực hiện bởi mỗi tiến trình phải xảy ra theo đúng trình tự. Nó sẽ cần nhiều nỗ lực, một chút may mắn để đạt được kết quả mong muốn.

Xác thực địa chỉ email, các hoạt động liên quan đến email thường được nhắm đến để khai thác race conditions. Các emails thường được gửi ở 1 luồng ngầm sau khi máy chủ gửi phản hồi HTTP cho máy khách, làm cho race conditions dễ tấn công hơn.

**Các cơ chế khóa dựa trên phiên**

Một số thư viện cố gắng ngăn chặn sự gián đoạn dữ liệu bằng cách khóa yêu cầu. Ví dụ, module xử lý phiên của PHP chỉ xử lý 1 yêu cầu mỗi phiên vào 1 thời điểm.

Điều cực kỳ quan trọng là phát hiện ra loại hành vi này vì nó có thể che giấu các lỗ hổng có thể khai thác được. Nếu bạn để ý thấy tất cả yêu cầu được xử lý theo tuần tự, thử gửi từng yêu cầu bằng các phiên khác nhau.

**Race conditions từng phần**

Các ứng dụng tạo đối tượng qua nhiều bước, có thể xuất hiện 1 trạng thái trung gian mà các đối tượng có thể khai thác được.

Ví dụ, khi đăng ký 1 người dùng mới, 1 ứng dụng có thể tạo người dùng trong cơ sở dữ liệu và thiết lập API key bằng cách dùng 2 câu truy vấn SQL. Điều này để lại 1 window mà người dùng tồn tại, nhưng API key của họ chưa được khởi tạo.

Loại hành vi này mở đường cho các khai thác mà bạn tiêm vào điểm cuối sẽ trả về một cái gì đó mà cơ sở dữ liệu chưa được khởi tạo, giống như một chuỗi rỗng hoặc null trong JSON.

Các thư viện thường để bạn truyền vào 1 mảng và cấu trúc dữ liệu không phải là chuỗi. Ví dụ, trong PHP:

* param[]=foo param = ['foo']
* param[]=foo&param[]=barparam = ['foo', 'bar']
* param[]param = []

Ruby on Rails cũng cho bạn làm điều tương tự bằng cách cung cấp 1 truy vấn hoặc tham số POST với 1 key nhưng không có giá trị. Nói cách khác, param[key] sẽ là

params = {"param"=>{"key"=>nil}}

Ở ví dụ trên, trong suốt race window, bạn có thể gửi các yêu cầu đến API yêu cầu xác thực:

GET /api/user/info?user=victim&api-key[]= HTTP/2

Host: vulnerable-website.com

Lưu ý là các va chạm từng phần với mật khẩu sẽ dễ xảy ra với mật khẩu hơn là API Key. Tuy nhiên, khi mật khẩu được hash, bạn cần tiêm 1 giá trị làm cho hash digest phù hợp với giá trị chưa khởi tạo.

**Các cuộc tấn công time-sensitive**

Thông thường bạn sẽ không tìm được race conditions, nhưng các kĩ thuật để vận chuyển các yêu cầu với thời gian chính xác vẫn có thể tiết lộ sự hiện diện của các lỗ hổng khác.

Ví dụ như timestamps được sử dụng thay thế cho các chuỗi ngẫu nhiên bảo mật bằng mã hóa để tạo token.

Xem xét 1 token đặt lại mật khẩu chỉ ngẫu nhiên bằng timestamp. Trong trường hợp này, có thể đặt lại mật khẩu cho 2 tài khoản khác nhau vì chúng dùng chung token. Tất cả những gì bạn cần làm canh thời gian các yêu cầu để chúng tạo ra cùng timestamp.