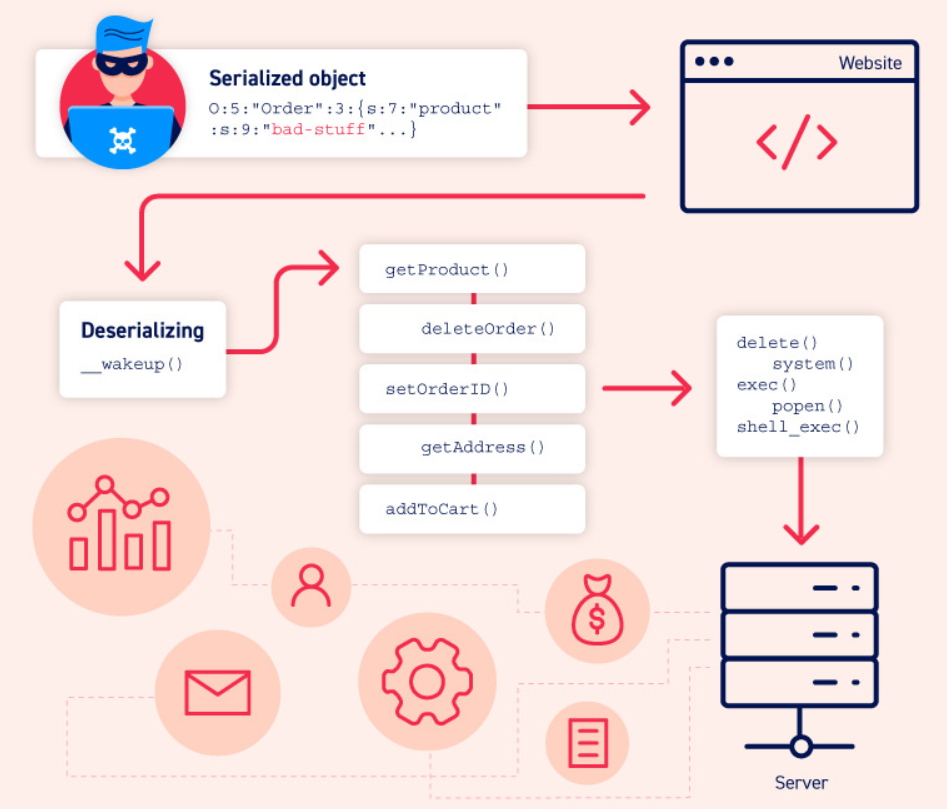
**Checkpoint WEB 11 đến WEB 17**

**WEB11: Insecure deserialization**



**A. Tìm hiểu về Insercure deserialization**

**1) Serialization (Đối tượng hóa) là gì?**

- Là quá trình chuyển đổi cấu trúc dữ liệu phức tạp, chẳng hạn như các object và trường của chúng sang định dạng **“phẳng hơn”** để có thể gửi và nhận dưới dạng luồng byte tuần tự. Đối tượng hóa dữ liệu làm cho nó đơn giản hơn nhiều để:

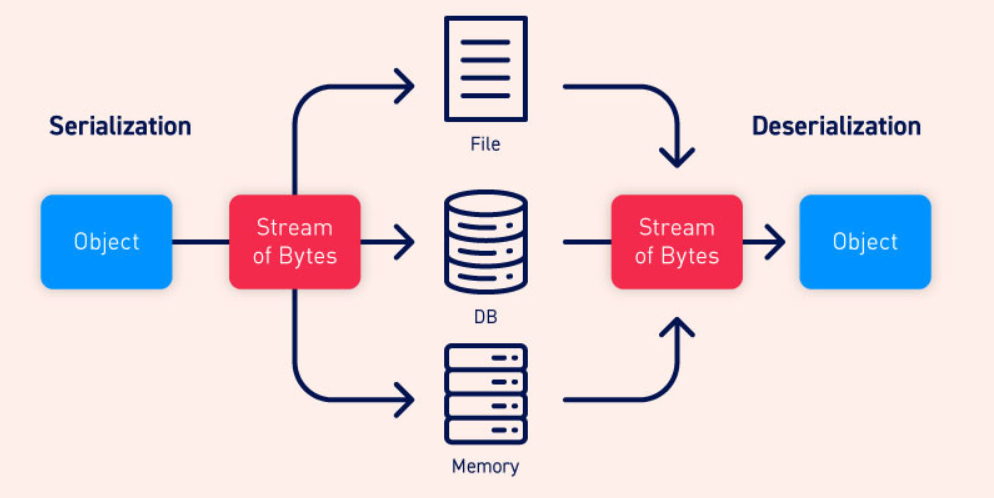
* Ghi dữ liệu phức tạp vào bộ nhớ tiến trình, tệp hoặc cơ sở dữ liệu
* Gửi dữ liệu phức tạp, chẳng hạn như qua mạng, giữa các thành phần khác nhau của ứng dụng hoặc trong lệnh gọi API

- Khi đối tượng hóa, các thuộc tính được giữ nguyên, cùng với giá trị được gán của chúng

- Định dạng **“phẳng”**: Định dạng tệp không có cấu trúc rõ ràng, không chứa các đối tượng. Chỉ là dữ liệu tĩnh được sắp xếp theo một định dạng cụ thể. **Định dạng tệp hoặc chuỗi ký tự.** JSON, XML, YAML.

**2) Serialization vs deserialization**

- **Deserialization** là quá trình khôi phục luồng byte thành một bản sao đầy đủ chức năng của object ban đầu, ở trạng thái chính xác như khi nó được **serialization**



- Các đối tượng được **serialization** phụ thuộc vào ngôn ngữ. Tất cả các thuộc tính của đối tượng gốc được lưu trữ trong luồng dữ liệu khi **serialization** bao gồm private fields. Để ngăn một trường được **serialization,** nó phải được đánh dấu rõ ràng là **“transient”** trong class.

- **Serialization:** tương đương marshalling (Ruby) hoặc pickling (Python)

**3) Insecure deserialization là gì?**

- **Insecure deserialization** là khi dữ liệu kiểm soát bởi người dùng được giải đối tượng hóa (deserialized) bởi trang web. Điều này cho phép kẻ tấn công thao túng các đối tượng được **serialization** để đưa dữ liệu độc hại vào.

- Thậm chí có thể thay thế một đối tượng được **serialization** bằng một đối tượng của một lớp hoàn toàn khác. Đáng lo ngại là các object của bất kỳ class nào có sẵn sẽ được **deserialized** và khởi tạo => **object injection.**

- Một object của một class không mong đợi có thể gây exception. Nhiều cuộc tấn công dựa trên "**deserialization**" được thực hiện trước khi quá trình "**deserialization**" hoàn thành. Điều này có nghĩa là quá trình "**deserialization**" chính nó có thể khởi động một cuộc tấn công, ngay cả khi chức năng của trang web không tương tác trực tiếp với đối tượng độc hại.

**4) Lỗ hổng insecure deserialization phát sinh thế nào?**

- Thường do thiếu hiểu biết về tầm quan trọng của việc deserializing dữ liệu do người dùng có thể điều khiển được.

- Thực hiện một số hình thức kiểm tra bổ sung trên dữ liệu được deserialized -> không hết được các trường hợp hoặc quá muộn

- Đối tượng được deserialized thường được cho là đáng tin cậy. Đặc biệt là khi sử dụng các ngôn ngữ với định dạng serialization nhị phân (Hackvertor extension)

- Số lượng dependencies và library của một trang web nhiều -> quá nhiều class và method -> khó quản lý an toàn

**5) Tác động của insecure deserialization**

Nó cho phép kẻ tấn công tái sử dụng code ứng dụng hiện có một cách độc hại, dẫn đến nhiều lỗ hổng khác, thường là RCE hoặc DoS

**B. Exploit insecure deserialization**

**1) Cách nhận dạng insecure deserialization**

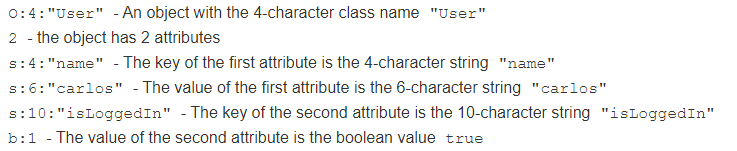
**a) PHP serialization forma**

PHP sử dụng định dạng chuỗi hầu như con người có thể đọc được: **chữ cái** đại diện cho kiểu dữ liệu và **số** đại diện cho độ dài



Serialized:





method: *serialize()* và *unserialize().*

**b) Java serialization format**

Định dạng binary serialization.

Các đối tượng Java serialized luôn bắt đầu với cùng một byte, được mã hóa là *ac ed* trong hệ thập lục phân và *rO0* trong Base64.

Bất kỳ class nào implements interface *java.io.Serializable* đều có thể serialized và deserialized. *readObject()* method, được sử dụng để đọc và deserialized dữ liệu từ *InputStream*

**2) Thao túng serialized objects**

- Đơn giản nhất là thay đổi một thuộc tính của object đã được serialize.

- Có 2 cách tiếp cận:

* Chỉnh sửa đối tượng trực tiếp trong dạng byte stream của nó (chỉnh sửa thuộc tính hoặc kiểu dữ liệu)
* Viết đoạn script ngắn để tạo vào serialize object của chính mình. Thường dùng khi làm việc với các định dạng binary serialization

**a) Sửa đổi thuộc tính của object**

Miễn là kẻ tấn công giữ nguyên một đối tượng được serialized hợp lệ, quá trình deserialized sẽ tạo ra một đối tượng phía máy chủ với các **giá trị thuộc tính đã bị thay đổi.**



Sau khi login thành công, **session cookie** được tạo. Sau khi URL decoded session cookie thu được mã base64 (ký tự cuối là dấu "="). base64 decoded thu được byte stream của php object.

**Ví dụ byte stream:**



Thay đổi giá trị thuộc tính *admin* thành "1" và mã hóa base64 sẽ thu được giá trị session cookie mới truy cập được vào site admin.

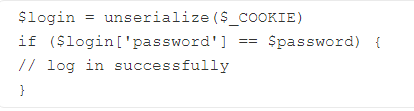
**b) Thay đổi data types**

PHP-based logic đặc biệt dễ bị tấn công bởi loại này do hành vi của toán tử so sánh (“==”) lỏng lẻo khi so sánh các loại dữ liệu khác nhau. Ví dụ so sánh string với int, php sẽ cố gắng chuyển string thành int => **5=="5"** là true.

**PHP có thể chuyển đổi một chuỗi bất kỳ bắt đầu bằng số thành giá trị số nguyên bằng cách bỏ qua phần còn lại của chuỗi. Nếu không có số thì chuyển thành 0.**



Trường hợp toán tử "==" được sử dụng cùng với dữ liệu do người dùng kiểm soát từ mội deserialized object.

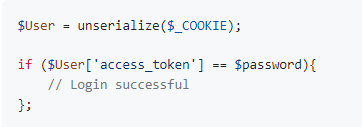


Sửa đổi thuộc tính mật khẩu để chứa số nguyên 0 thay vì chuỗi dự kiến. Miễn là mật khẩu được lưu trữ không bắt đầu bằng một số, điều kiện sẽ luôn trả về true, cho phép vượt qua xác thực. Chỉ có thể thực hiện được vì việc deserialized object giữ nguyên kiểu dữ liệu.



Sử dụng *access\_token* để kiểm tra xem có được xác thực hay không. Bài này có một list *access\_token*, check trùng với đúng user đó thì sẽ được xác thực.





Lấy *access\_token* để so sánh và vì sử dụng toán tử "==" lỏng lẻo nên có thể sửa đổi deserialized object để bypass authentication.

Sửa giá kiểu dữ liệu thuộc tính *access\_token* thành *int* với **giá trị = 0** sẽ so sánh 0 với string. Thì khi so sánh kết quả sẽ luôn *true* vì không có số nào trong *access\_token*.

=> Điều kiện để cách này thành công:

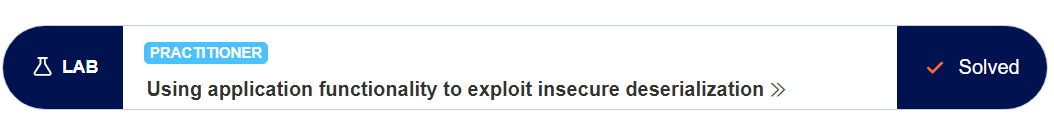
* Phải là quá trình deserialization vì deserialized object giữ nguyên kiểu dữ liệu.
* Sử dụng toán tử “==” so sánh lỏng lẻo

Khi làm việc với các định dạng binary, dùng **Hackvertor extension**: Sửa đổi serialized data dưới dạng string, và nó sẽ tự cập nhật dữ liệu binary, điều chỉnh độ lệch (offset) phù hợp.

**c) Using application functionality**

- Có thể sử dụng insecure deserialization để chuyền vào dữ liệu không mong muốn và tận dụng chức năng liên quan để gây thiệt hại

- Ví dụ: Lợi dụng hàm **Delete user** để xóa một file bất kỳ**:** Khi xóa user thì ảnh đại diện của user chính là giá trị của thuộc tính $user->image\_location (là một đường dẫn tới file) cũng sẽ bị xóa. Nếu kẻ tấn công sửa đường dẫn tới file avatar thành một đường dẫn tệp tùy ý thì khi xóa user, tệp này cũng sẽ bị xóa



Khi xóa user thì avatar\_link cũng bị xóa theo. Chỉ cần thay đổi giá trị avatar\_link thành file muốn xóa là được.

- Tạo ra exploit để tự động chuyển dữ liệu vào các method nguy hiểm. Điều này được kích hoạt bởi việc sử dụng **magic methods**

**3) Magic methods**

**-** Magic methods là một tập con của methods mà không cần phải gọi nó ra. Chúng sẽ được tự động gọi mỗi khi xảy ra một sự kiện hoặc kịch bản cụ thể. Magic methods đôi khi được chỉ định bằng cách thêm tiền tố hoặc bao quanh tên method bằng 2 dấu gạch dưới liên tiếp.

- PHP **\_\_construct()** được gọi mỗi khi object của class được khởi tạo, tương tự như \_\_init\_\_ trong Python. Thông thường dùng để khởi tạo các thuộc tính, tuy nhiên magic methods có thể được tùy chỉnh bởi dev để thực thi bất kỳ code nào mà họ muốn

- Magic methods có thể nguy hiểm khi nó xử lý dữ liệu bị kiểm soát bởi kẻ tấn công.

- Một số ngôn ngữ có các magic methods được tự động gọi trong quá trình deserialization. Ví dụ: phương thức *unserialize()* của PHP tìm và gọi magic methods *\_\_wakeup()* của đối tượng.

- Trong Java deserialization, *ObjectInputStream.readObject()* method được sử dụng để đọc dữ liệu từ byte stream ban đầu và hành động như một hàm khởi tạo (constructor) để “**khởi tạo lại**” object đã được serialization. Các class *Serializable* (các class implement interface Serializable)cũng có thể khai báo phương thức readObject() của riêng chúng theo cách sau:

*private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException*

*{*

*// implementation*

*}*

Method *readObject()* được gọi bởi quá trình deserialization sau khi deserialization mặc định đã xảy ra.

- Một method *readObject()* được khai báo chính xác như vậy hoạt động như một magic method được gọi trong quá trình deserialization. Nên chú ý đến magic method kiểu này vì chúng cho phép truyền data từ serialized object vào code của trang web trước khi object được deserialized hoàn toàn.

**4) Tiêm objects bất kỳ**

- Thao túng class của object đang được truyền vào dưới dạng serialized data.

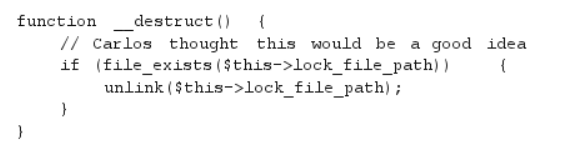
- Deserialization methods thường không kiểm tra object nào được deserialized, nếu serializtion thì sẽ được deserialization => cho phép kẻ tấn công tạo ra các instance của các class tùy ý => khởi tạo object độc hại

- Có source code -> nghiên cứu class có sẵn -> tìm các class có deserialization magic methods -> kiểm tra xem có thực hiện hành động nguy hiểm với dữ liệu được kiểm soát không -> tiêm object đã được serialization



Các máy chủ web thường được cấu hình để chạy các tệp như **.php, .phtml**,... bằng trình thông dịch PHP và không gửi chúng tới trình duyệt. Với **.php.bak** hoặc **.php~**, có thể đọc code để tìm lỗ hổng. Trong bài lab này dùng **.php~** để đọc source code của **CustomTemplate.php**

Có class: **CustomTemplate** với hàm **\_\_destruct()** là magic method.



**unlink()** là một hàm trong PHP được sử dụng để xóa một tệp tin.

**Giải thích code:** Hàm \_\_destruct() được gọi tự động khi object của class CustomTemplate bị hủy. Hàm if() sẽ kiểm tra nếu file mà object của CustomTemplate đang sử dụng vẫn tồn tại trên hệ thống nó sẽ sử dụng hàm unlink() để xóa file này đi.

=> nếu có thể tạo ra một instance của lớp CustomTemplate với nội dung do mình tự định nghĩa, thì có thể xóa bất kỳ file nào.

Tạo ra một object của class CustomTemplate. Sau khi serialization, lưu trong cookie và reload lại trang web. Object của class CustomTemplate đã được deserialized và khởi tạo. Trang web sẽ trả ra một Exception. Khi hủy, tệp được tham chiếu trong cookie bị xóa.

**5) Gadget chains**

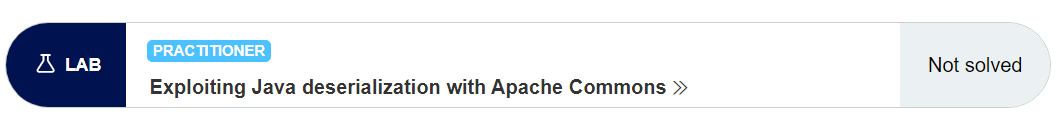
- Gadget là một đoạn code tồn tại trong ứng dụng có thể giúp kẻ tấn công đạt được một mục tiêu cụ thể. Một gadget có thể không có hại, nhưng kẻ tấn công có thể lợi dụng để chuyển input vào một gadget khác. Bằng cách nối các gadget với nhau theo chuỗi như thế này, kẻ tấn công có thể tiềm ẩn đầu vào của họ vào một "sink gadget" nguy hiểm.

- Điều duy nhất mà kẻ tấn công kiểm soát là **dữ liệu được truyền vào gadget chain**. Thông thường điều này được thực hiện bằng cách sử dụng một **magic method** được gọi trong quá trình deserialization, đôi khi được biết đến với tên gọi là "**kick-off gadget**".

**6) Làm việc với gadget chains dựng sẵn**

**a) ysoserial**

- Công cụ này cho phép chọn một trong những chuỗi gadget được cung cấp cho một thư viện mà cho rằng ứng dụng mục tiêu đang sử dụng, sau đó truyền vào một lệnh muốn thực thi. Nó sau đó tạo ra một đối tượng đã được serialize phù hợp dựa trên chuỗi đã chọn. Việc này vẫn bao gồm một số mức độ thử và lỗi.



Response chứa cookie, nó bắt đầu với **rO0** là dấu hiệu cho thấy đây là đối tượng java serialized được má hóa base64.

Vì ứng dụng web sử dụng thư viện Apache Commons Collections, nên chỉ tập trung vào tải trọng **CommonsCollections**.

*java -jar path/to/ysoserial.jar CommonsCollections4 'rm /home/carlos/morale.txt' | base64*

Sử dụng gadget chain "CommonsCollections4" để tạo ra một chuỗi byte serialized của đối tượng có lỗ hổng.

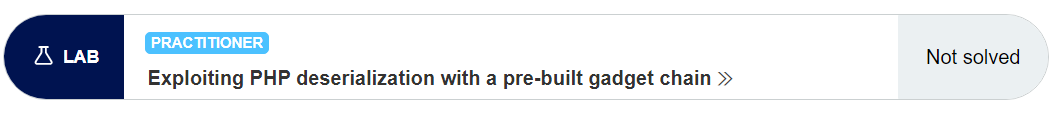
Đầu ra của chuỗi byte serialized được mã hóa base64.

- Phát hiện lỗ hổng deserialization không an toàn trên hầu hết các máy chủ:

+ Gadget chain *URLDNS* gây ra một DNS lookup cho một URL được cung cấp. Không dựa vào ứng dụng mục tiêu sử dụng một thư viện dễ bị tấn công cụ thể và hoạt động trên bất kỳ phiên bản Java đã biết nào. Nếu phát hiện một đối tượng đã được serialize trong lưu lượng truy cập, có thể thử sử dụng chuỗi gadget này để tạo ra một đối tượng gây ra tương tác DNS với máy chủ Burp Collaborator. Nếu như vậy, bạn có thể chắc chắn rằng việc deserialization đã xảy ra trên mục tiêu.

+ *JRMPClient.* Nó khiến máy chủ cố gắng thiết lập kết nối TCP đến địa chỉ IP được cung cấp. Chuỗi này có thể hữu ích trong các môi trường mà tất cả lưu lượng đi ra đều bị chặn bởi tường lửa, bao gồm cả các truy vấn DNS. Sự khác biệt thời gian tinh tế giữa các phản hồi có thể giúp phát hiện xem liệu việc deserialization đã xảy ra trên máy chủ, ngay cả trong trường hợp mù.

**b) PHP (PHPGGC) Generic Gadget Chains**

****

Token là serialized PHP object, và HMAC signature

HMAC (hash-based message authentication code), là một phương thức xác thực sử dụng hàm băm, trong trường hợp này là SHA-1 cùng với một khóa bí mật làm đầu vào bổ sung (HMAC = SHA-1 + secret key) => không thể tạo token hợp lệ nếu không biết secret key.

Gửi invalid token (sửa hoặc xóa chữ ký). Kiểm tra và xác thực chữ ký, nhưng không xử lý việc này một cách khéo léo -> một Exception hiển thị làm lộ thông tin framework được sử dụng: **Symfony 4.3.6**

Để lộ đường dẫn tới **phpinfo.php** => lộ SECRET\_KEY

Khi biết khóa bí mật -> tạo được token đúng chứa dữ liệu tùy ý chẳng hạn như serialization object của mình.

Sử dụng **PHP Generic Gadget Chains** để tìm gadget chain đúng cho framework Symfony 4.3.6 đó là **Symfony/RCE4**



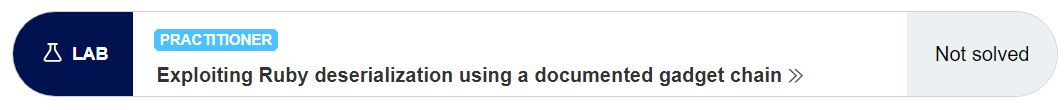
Dòng đầu tiên tạo object serialized PHP và mã hóa base64

Dòng thứ 2 tạo HMAC signature: **obj** được băm sử dụng **SHA1** sau đó kết hợp với **key** để tạo ra HMAC signature

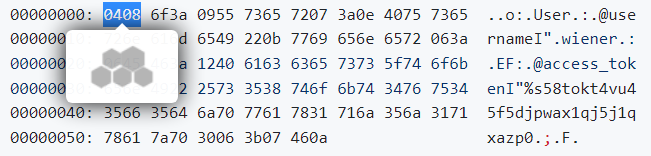
Cách khác dùng **hash\_hmac:** *$signedHMACSHA1Key = hash\_hmac('sha1', $gadgetChains, $secretKey);*

**7) Làm việc với documented gadget chains**

Không có tool thì đọc document 😊.

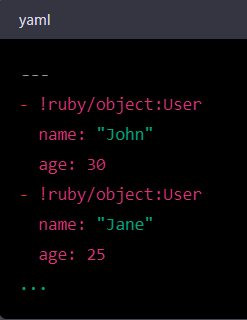


Base64 decode cookie thu được:



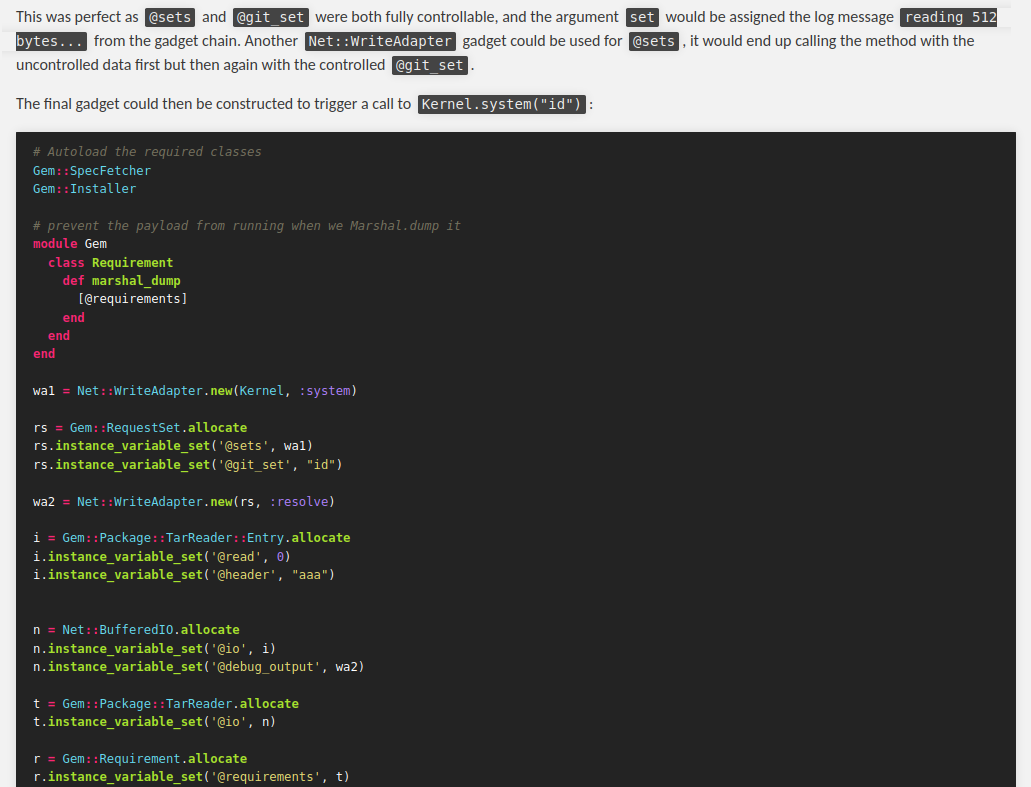
4 byte đầu tiên “0408” chỉ ra đây là một **serialized Ruby object** cụ thể là **Serialized ("marshaled") Ruby objects**

**Bổ sung: Serialized ("yaml") Ruby object** thường bắt đầu bằng chuỗi "---", có cấu trúc dạng key-value pairs và kết thúc bằng "..." hoặc "!!null".



<https://devcraft.io/2021/01/07/universal-deserialisation-gadget-for-ruby-2-x-3-x.html>

Có một tập lệnh Ruby tạo ra payload Ruby serialized object gadget chain



Sửa **id** thành **rm /home/carlos/morale.txt** và base64 encode output.

------------------------------------------------------------------------------------------------------------------------------------------

**WEB12: Server-side template injection (SSTI)**

**A. Tìm hiểu SSTI là gì?**

**1) Server-side template injection là gì?**

**-** Khái niệm: Kẻ tấn công dựa vào **template syntax** ( cú pháp của khuôn mẫu mà mục tiêu đang sử dụng để dựng web page) để tiêm payload độc hại vào template từ đó thực thi payload ngay trên phía server.

- Engine: Kết hợp các **yếu tố cố định trên template** với các **dữ liệu được đưa vào** tại thời điểm đó để **render** ra bản web page tương ứng.

- SSTI xảy ra khi đầu vào của người dùng được kết hợp trực tiếp vào template, thay vì được truyền dưới dạng dữ liệu đơn thuần. Khi đó, attacker sẽ có thể đưa vào các **template directive** nhằm thao túng **template engine** từ đó có thể chiếm trọn quyền kiểm soát server.

- **Template directive:** Là các lệnh đặc biệt có thể nhúng vào trong template để kiểm soát hành vi của nó và tạo các nội dung động. Thường được cung cấp bởi **template engine** và được sử dụng để xác định cách xử lý và hiển thị template.

**2) Tác động của SSTI**

- Khiến trang web hứng chịu nhiều cuộc tấn công khác nhau tùy thuộc vào template engine.

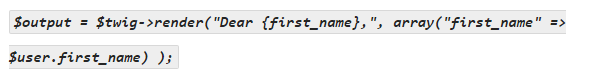
- Mức nghiêm trọng nhất là RCE, chiếm toàn quyền kiểm soát back-end server và sử dụng nó để thực hiện các cuộc tấn công khác vào cơ sở hạ tầng nội bộ

- Có khả năng đọc vào dữ liệu nhạy cảm và các tệp tùy ý trên máy chủ

**3) SSTI phát sinh như thế nào?**

- Phát sinh khi đầu vào của người dùng được nối vào các mẫu thay vì được chuyển vào dưới dạng dữ liệu

- Static template chỉ cung cấp các **placeholder** để **engine** lấy các dữ liệu liên quan rồi render ra web page. Kiểu này thì sẽ không phát sinh SSTI vì dữ liệu chỉ là đơn thuần. Ví dụ Twig template:



Không bị SSTI vì tên người dùng chỉ được truyền vào dưới dạng dữ liệu

- Template dạng string cho phép kết hợp trực tiếp input của người dùng vào template trước khi render ra web page sẽ phát sinh SSTI



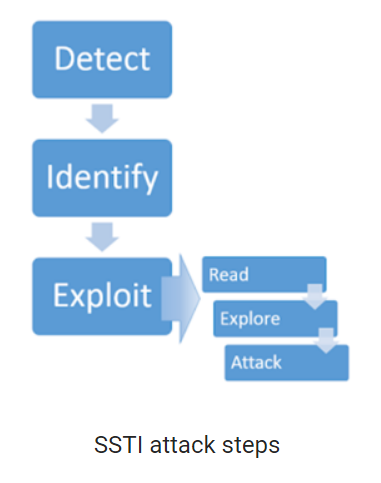
Một phần của template đã không còn static mà đã chuyển sang dynamic (động) tùy thuộc vào **GET parameter name.** Điều này có khả năng cho phép kẻ tấn công tiêm payload bên trong tham số name:



Theo một số cách, điều này **tương tự** như các lỗ hổng **SQL injection** xảy ra trong các câu lệnh được chuẩn bị kém.

**4) Xây dựng một cuộc tấn công SSTI**

- Các bước triển khai SSTI attack: SSTI attack sẽ diễn ra qua các bước bao gồm **Detect** (Phát hiện), **Identify** (Nhận diện) và **Exploit** (khai thác).



- **Detect:**

**+** Phương pháp thử nghiệm đơn giản ban đầu là **fuzzing** (quá trình thử nghiệm với nhiều dạng thông tin đầu vào và quan sát đầu ra để đánh giá), đưa vào một chuỗi các ký tự đặc biệt thường được sử dụng trong các template expression ví dụ như: **${{<%[%'"}}%\.** Nếu có một exception được đưa ra => input vào đã được máy chủ diễn giải theo một cách nào đó => có thể tồn tại SSTI

+ SSTI xảy ra trong 2 bối cảnh (context) riêng biệt, mỗi context đều yêu cầu phương pháp phát hiện riêng.

**Plaintext context: Ngữ cảnh text đơn thuần**

**Input được chèn trực tiếp vào template**

Hầu hết các template language cho phép tự do nhập nội dung bằng cách sử dụng trực tiếp các HTML tags hoặc sử dụng cú pháp của template, cú pháp này được hiển thị thành HTML ở back-end trước khi HTTP response được gửi

Đôi khi có thể khai thác XSS.

Sử dụng các phép toán làm giá trị của tham số để kiểm tra xem có bị SSTI không.

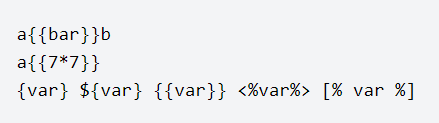




Nếu kết quả là **Hello 49** => có thể bị SSTI

Cú pháp tùy thuộc vào template sử dụng

Bước đầu tiên trong việc thử nghiệm SSTI trong plaintext context là xây dựng các biểu thức phổ biến được sử dụng bởi các template engines khác nhau dưới dạng payload và theo dõi response của server để xem biểu thức nào được thực thi



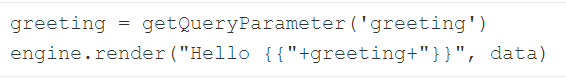
**=> Test biểu thức toán học trong template expression nếu có thực thi thì là plaintext context**

**Code context**

**Input được saved vào một variable trước và cho vào template expression**

Lỗ hổng bị lộ do input được đặt trong một **template expression** (là một đoạn code sự dụng trong template để tạo nội dung động. Thường được đặt trong các dấu phân cách đặc biệt như `**{{ }}`** hoặc `**{% %}**` phụ thuộc vào template đang được sử dụng. Cho phép chèn nội dung động vào mẫu bằng cách chuyển dữ liệu từ phía máy chủ sang phía máy khách. Chúng có thể được sử dụng để hiển thị dữ liệu, thực hiện các phép tính đơn giản và thực thi logic lập trình cơ bản.). Điều này có thể ở dạng tên biến do người dùng kiểm soát được đặt bên trong một tham số, ví dụ như:

Có một **variable** (biến số) mà có thể kiểm soát (**greeting** trong trường hợp bên dưới) được đặt vào bên trong của một **parameter** như sau:



URL:



Thì kết quả xuất ra sẽ có dạng “Hello {username}“.

Không bị XSS



Nếu có lỗi hoặc output trống => sai syntax. Nếu không có syntax nào phù hợp thì không bị SSTI

Nếu output có kèm HTML tùy ý thì có thể khai thác SSTI



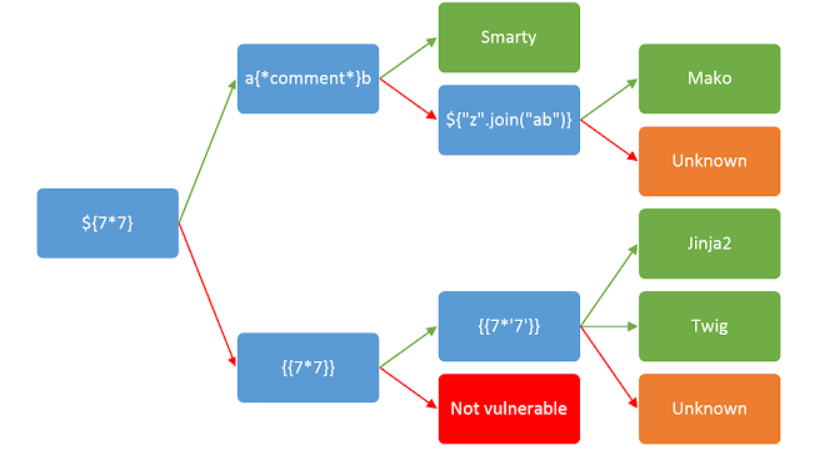
**=> Chèn HTML tag tùy ý vào nếu output có kèm HTML thì là code context**

**- Identify**

Đa phần đều có nhiều điểm tương đồng về mặt syntax.

Chỉ cần gửi cú pháp không hợp lệ thường là đủ vì thông báo lỗi kết quả sẽ cho biết chính xác template engine là gì và đôi khi là phiên bản nào.

Nếu không có thông tin template từ thông báo lỗi thì cần kiểm tra thủ công các payload dành riêng cho ngôn ngữ khác nhau và quan sát kết quả để đánh giá **template engine** thông qua phương án loại trừ. Một cách phổ biến để thực hiện việc này là đưa vào các phép toán tùy ý bằng cách sử dụng cú pháp từ các template engine khác nhau



Cùng một payload đôi khi nhiều template language đều trả response thành công => không kết luận vội

- **Exploit**

Với việc đã phát hiện **SSTI vulnerability** tồn tại và nhận diện được **template engine** tương ứng, có thể triển khai tiếp phần exploit bao gồm các bước **read** (đọc chi tiết về **template syntax**, thông tin về phương án triển khai bao mật và phương án **exploit** mà những người đi trước đã triển khai), **explore** (khám phá môi trường tác nghiệp và các đối tượng liên quan) và **attack** (triển khai tấn công).

**B. Khai thác lỗ hổng SSTI**

Sau khi detect và xác định được template đang sử dụng , việc khai thác thành công thường bao gồm quy trình sau

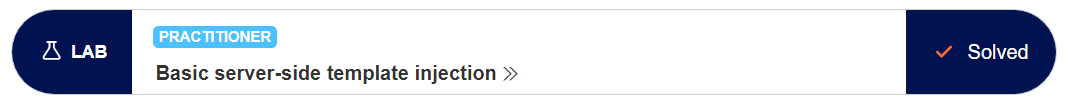


**1) Read**

Đọc tài liệu của nó

**a) Học các cú pháp đơn giản của template**

Học cú pháp cơ bản rõ ràng là quan trọng, cùng với các **chức năng chính và xử lý các biến.**



Detect: Gửi request với message chứa **nhiều ký tự đặc biệt** được mã hóa URL: **${{<%[%'"}}%\**

**<%= foobar %>**

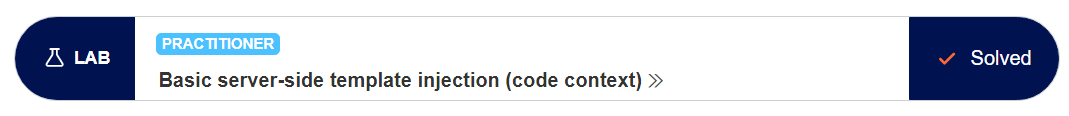
****

Thông báo lỗi cho biết sử dụng Ruby.

Có thể thấy các ký tự **<%** không có trong phản hồi. Bài lab sử dụng ERB template. Document chỉ ra rằng **<%** và **%>** được sử dụng làm phần đầu và cuối của template expression.

Một biểu thức có thể được đánh giá bằng cú pháp **<%= Ruby expression %>**

Sử dụng method **system(): <%= system("rm morale.txt") %>**

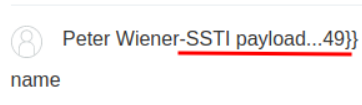


Thử test SSTI trong comment nhưng không được. username có thể kiểm soát bới attacker nên có thể gây SSTI.

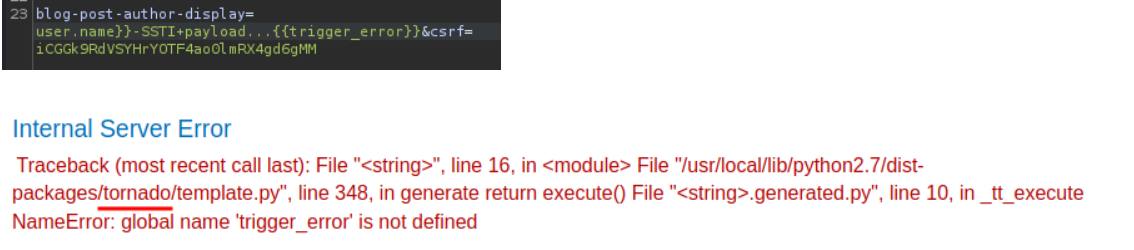
Thử với biểu thức toán học

**user.name}}-SSTI payload...{{7\*7}}**

****

****

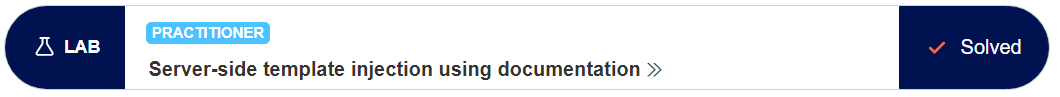
Xác định template sử dụng: **{{trigger\_error}}**



Template sử dụng là **tornado.** Có thể import module **os** cho phép thực thi OS command.

**b) Read about the security implications**

Trong document cung cấp Security section, thường phác thảo tất cả những nguy hiểm tiềm ẩn nên tránh thực hiện với template. Ví dụ như một đối tượng hoặc chức năng tích hợp cụ thể có thể gây ra rủi ro bảo mật.



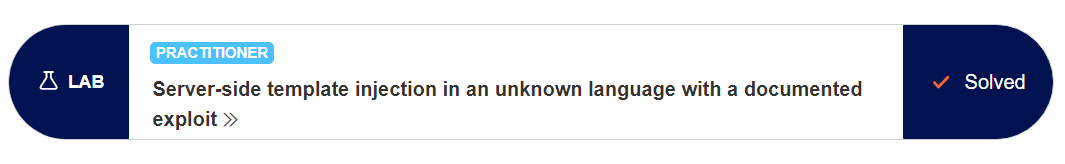
Cho phép chỉnh sửa template. Thử biểu thức toán học và đã thành công

Kích hoạt lỗi => trong thông báo lỗi cho biết sử dụng FreeMarker (Java).

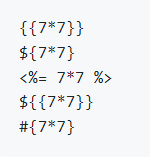
Có thể thực thi OS command thông qua: Class **Execute, exec() method.**

**c) Tìm kiếm các khai thác đã biết - Look for known exploits**

Nếu đã xác định được template sử dụng, có thể tham khảo các khai thác có sẵn trên mạng



Sử dụng Burp Intruder để test với bộ payload



Nếu cú pháp không hợp lệ thì lỗi máy chủ sẽ xảy ra và cung cấp một số thông tin => biết được sử dụng **Handlebars template**

Sử dụng payload có sẵn trên mạng.

**2) Explore -Khám phá**

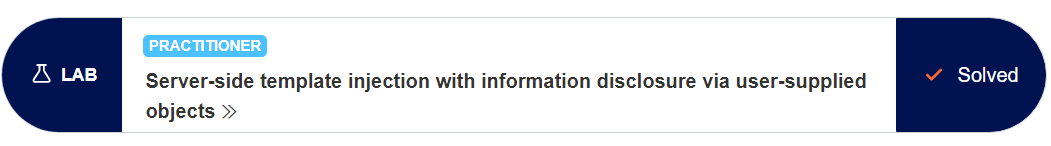
Khám phá môi trường và cố gắng khám phá tất cả các đối tượng mà bạn có quyền truy cập.

Nhiều template hiển thị một số **“self”** hoặc **"environment"** object, hoạt động như một không gian chứa tất cả các object, method và attributes được template engine hỗ trợ. Nếu một đối tượng như vậy tồn tại, bạn có thể sử dụng nó để tạo danh sách các đối tượng nằm trong phạm vi.

**a) Object do dev cung cấp**

Các trang web sẽ chứa cả các **đối tượng tích hợp sẵn do template cung cấp** và **các đối tượng tùy chỉnh, dành riêng cho trang web do nhà phát triển web cung cấp (non-standard objects).** Non-standard objects có khả năng chứa thông tin nhạy cảm hoặc các phương pháp có thể khai thác

Có thể bị **directory traversal**



Cho phép chỉnh sửa template. Thử biểu thức toán học và đã thành công

Thử với cú pháp khác thì hiện thông báo lỗi => biết là dùng **Django template.**

Theo như document thì có thể sử dụng tag **debug** để lấy thông tin debug



Kết quả có thể xem được danh sách các đối tượng và thuộc tính trong template này.

Theo như document thì đối tượng **settings** có thuộc tính **SECRET\_KEY**



------------------------------------------------------------------------------------------------------------------------------------------

**WEB13: Web cache poisoning**

**A. Tìm hiểu Web cache poisoning**

**1) Web cache poisoning là gì?**

- Kẻ tấn công lợi dụng hành vi của một máy chủ web và bộ nhớ đệm để cung cấp một phản hồi HTTP có hại cho các người dùng khác.

- Bao gồm hai giai đoạn:

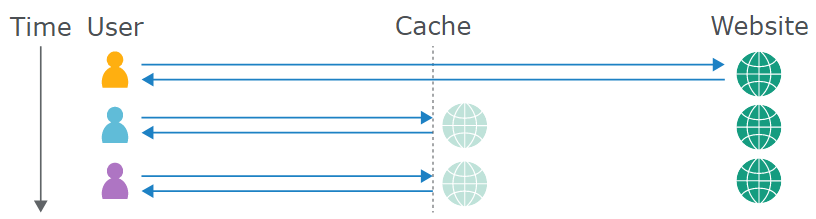
* Kẻ tấn công phải tìm cách tạo ra phản hồi từ máy chủ backend mà vô tình chứa một loại tải trọng nguy hiểm nào đó
* Đảm bảo rằng phản hồi của họ được lưu trữ trong bộ nhớ đệm và được cung cấp cho các nạn nhân dự kiến

- Tiềm ẩn nhiều tấn công khác nhau: XSS, JS injection, redirection,...

**2) Web cache hoạt động thế nào?**

- Caching là phương tiện dùng để giảm thiểu việc một máy chủ phải gửi một phản hồi mới cho mỗi yêu cầu HTTP riêng biệt.

- cache nằm giữa user và server, nơi nó lưu trữ (caches) các phản hồi cho các yêu cầu cụ thể, thường trong một khoảng thời gian cố định. Nếu có yêu cầu tương đương đến thì cache gửi bản sao của phản hồi được lưu trữ trong cache lại mà không cần tương tác phía backend.



**Cache keys**

- Khi cache nhận được một HTTP request thì nó đầu tiên xác định xem có response được lưu trữ trong cache mà có thể cung cấp trực tiếp không hay phải chuyển request đến server.

- cache xác định các request tương đương bằng **cache key**. Cache key thường chứa **request line** và **Host header**. Các thành phần của request không được bao gồm trong **cache key** gọi là "**unkeyed**"

- Nếu cache key khớp => request tương đương => cung cấp một bản sao của phản hổi được lưu trữ.

- **!!!Các thành phần khác của yêu cầu được bỏ qua hoàn toàn bởi bộ nhớ đệm!!!**

**3) Tác động của tấn công web cache poisoning?**

Phụ thuộc nhiều vào hai yếu tố:

* **Chính xác những gì kẻ tấn công có thể lưu trữ thành công:** poisoned cache là một phương tiện phân phối hơn là một cuộc tấn công độc lập => tác động liên quan mật thiết với mức độ nguy hiểm của payload.
* **Lưu lượng truy cập trên page bị ảnh hưởng:** Tác động có thể không tồn tại hoặc cực lớn, phụ thuộc vào việc trang web có phổ biến hay không.

Thời gian tồn tại không nhất thiết ảnh hưởng vì một cuộc tấn công thường có thể được viết kịch bản theo cách nó tái đầu độc bộ đệm vô thời hạn.

**4) Xây dựng một cuộc tấn công web cache poisoning**

Bao gồm các bước:

+ Xác định và đánh giá **unkeyed** inputs

+ Tạo ra phản hồi có hại từ back-end server

+ Khiến cho response cached

**a) Xác định và đánh giá unkeyed inputs**

Cache bỏ qua các unkeyed inputs => có thể sử dụng chúng để tiêm payload và tạo ra một phản hồi độc hại.

Có thể xác định các unkeyed inputs bằng cách thêm các input ngẫu nhiên vào request và xem nó có ảnh hưởng đến response không (phản ánh input trong phản hồi hoặc kích hoạt một phản hồi khác). Đôi khi tác động của chúng khá lảng tránh và yêu cầu phải tìm hiểu. Có thể sử dụng Burp Comparer để so sánh phản hồi có và không có injected input.

**Param Miner**

- Sử dụng extension này để xác định unkeyed inputs

- Thêm **cache buster** (ví dụ như một parameter unique) vào request line để tạo ra cache key của riêng mình.

**b) Tạo ra phản hồi có hại từ back-end server**

Xác định được unkeyed input -> đánh giá cách trang web xử lý nó -> tạo response có hại

**c) Khiến cho response cached**

Việc phản hồi có được lưu vào bộ nhớ cache hay không có thể phụ thuộc vào các yếu tố khác nhau, chẳng hạn như file extention, content type, route, status code và response header => gửi request tới các page khác nhau và nghiên cứu cách hoạt động của cache để tìm ra cách

**B. Exploiting web cache poisoning vulnerabilities**

**1) Khai thác lỗi thiết kế bộ đệm - Exploiting cache design flaws**

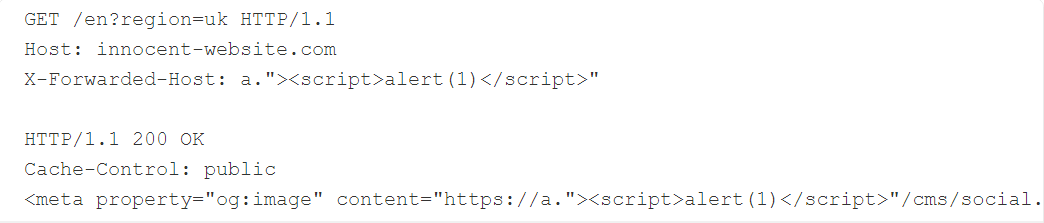
Các trang web bị web cache poisoning nếu xử lý unkeyed input theo cách không an toàn và cho phép các HTTP response tiếp theo được lưu vào cache. Lỗ hổng này có thể được sử dụng làm phương thức phân phối cho nhiều cuộc tấn công khác nhau.

**a) Sử dụng web cache poisoning để thực hiện một cuộc tấn công XSS**

Lỗ hổng web cache poisoning đơn giản nhất để khai thác là khi unkeyed input được phản ánh trong response có thể lưu vào cache.

****

Giá trị của *X-Forwarded-Host* header được phản ánh trong response. Đối với web cacher poisoning, *X-Forwarded-Host* header thường là **unkeyed**. => cache có thể bị đầu độc bằng 1 phản hồi chứa XSS đơn giản

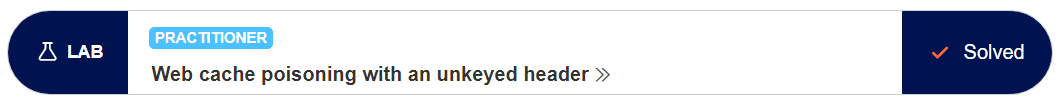


Nếu response được lưu vào trong cache, tất cả người dùng truy cập **/en?region=uk** sẽ thực thi payload XSS này

**b) Sử dụng web cache poisoning để khai thác việc xử lý không an toàn resource imports**

Một số trang web sử dụng unkeyed headers để tự động tạo URLs cho việc import resource, ví dụ như các tệp JS lưu trữ bên ngoài. Nếu kẻ tấn công thay đổi giá trị header thành domain mà họ kiểm soát, họ có thể làm thay đổi URL để trỏ đến tệp JS của riêng họ. Nếu response chứa URL độc hại được lưu vào cache, tệp JS của kẻ tấn công sẽ được import và thực thi trong phiên trình duyệt của người dùng có request với cache key hợp lệ.





Phát hiện máy chủ web đang sử dụng web cache

Ứng dụng web chấp nhận unkeyed header *X-Forwarded-Host*. Giá trị của *X-Forwarded-Host* được phản ánh trong đường dẫn tải tệp JS.





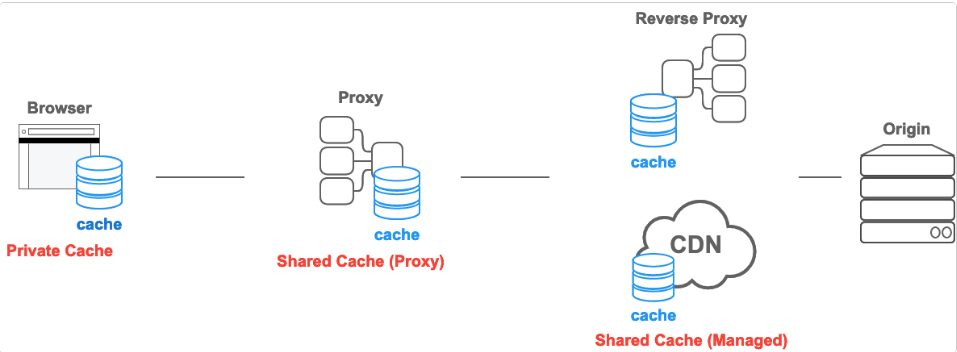


Trong exploit server: thay đổi tên tệp để khớp với đường dẫn được sử dụng









- **Cache-control header:**

**+** Chưa các chỉ thị (hướng dẫn) trong cả request và response để kiểm soát caching trong browsers và shared cache (ví dụ: Proxies, CDNs).

+ Cú pháp: Tuân theo các nguyên tắc sau:

* Không phân biệt chữ hoa, chữ thường
* Được ngăn cách bởi dấu phẩy
* Một số chỉ thị có tham số tùy chọn

+ Một số thuật ngữ mới:

* **(HTTP) cache**: Triển khai chứa các requests và responses để sử dụng lại trong các request tiếp theo. Nó có thể là **shared cache** hoặc **private cache**
* **Shared cache**: Cache nằm giữa server gốc và client (ví dụ: Proxy, CDN). Nó **lưu trữ một response duy nhất** và sử dụng lại với nhiều người dùng -> nên tránh lưu trữ nội dung được cá nhân hóa để lưu trữ vào shared cache.
* **Private cache:** Cache tồn tại trong client. Còn được gọi là *local cache* hoặc *browser cache*. Nó có thể được lưu trữ và sử dụng lại nội dung được cá nhân hóa cho một người dùng (không được chia sẻ với các ứng dụng client khác)
* **Store response:** Response được lưu trữ trong cache
* **Reuse response:** Sử dụng lại cách reponse được lưu trong cache cho request tiếp theo
* **Revalidate response:** Hỏi server xem response được lưu trữ có còn mới hay không. Thông thường được thực hiện thông qua một yêu cầu có điều kiện
* **Fresh response:** Cho biết response là mới
* **Stale response:** Response là cũ. Điều này thường có nghĩa là không thể sử dụng lại response nguyên trạng.
* **Age:** Thời gian đã trôi qua kể từ khi response được tạo.

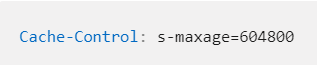
**+** Cache directives: Các chỉ thị ảnh hưởng đến cache, chủ yếu là response directives:

* **max-age:** Response có hai trạng thái là: fresh(mới) và stale(cũ). max-age=N chỉ ra rằng response vẫn còn mới cho đến N giây sao khi phải hồi được tạo.



=> response sẽ cho thấy nó còn mới trong 604800-86400=518400. Đấy là sự khác biệt giữa **max-age** và **age**

* **s-maxage:** cũng cho biết thời gian response fresh trong bao lâu (giống max-age) nhưng chỉ dành riêng cho shared cache

****

* **no-cache:** Response có thể được lưu trữ trong cache, nhưng response phải được xác thực với server trước mỗi lần sử dụng lại, ngay cả khi cache ngắt kết nối với server

****

* **private:** response chỉ có thể lưu trữ trong private cache.

****

* **public:** response có thể lưu trữ trong shared cache.

****

**- X-Forwarded-Host(XFH):**

**+** Là header để xác định host ban đầu được client request trong **Host** HTTP request header

+ Host name và port của reverse proxies (load balancers, CDNs) có thể khác với server gốc xử lý request -> **X-Forwarded-Host** được sử dụng để xác định máy chủ nào được sử dụng ban đầu.

**- X-cache header:**

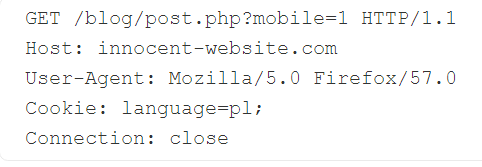
**+** Là một header HTTP được thêm vào HTTP response để cung cấp thông tin về trạng thái của các tài nguyên được yêu cầu từ bộ nhớ cache. Header này có thể được sử dụng để xác định xem một tài nguyên được lấy từ bộ nhớ cache hay từ máy chủ gốc.

+ Giá trị của X-cache là thông tin về việc cache của trang web đã được sử dụng để phục vụ request hay không. Bao gồm một trong các giá trị sau:

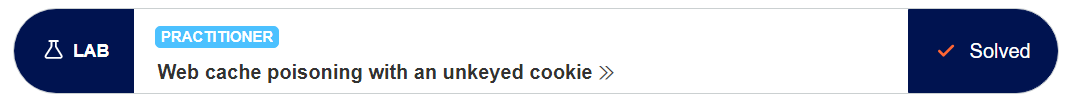
* **HIT**: Nội dung đã được lưu trữ trong bộ nhớ cache và được trả về trực tiếp từ bộ nhớ cache mà không cần phải tạo lại request đến máy chủ.
* **MISS**: Nội dung không tồn tại trong bộ nhớ cache và yêu cầu phải được gửi đến máy chủ để lấy dữ liệu mới.
* **EXPIRED**: Nội dung đã được lưu trữ trong bộ nhớ cache, nhưng đã hết hạn. Yêu cầu mới được tạo để lấy nội dung mới từ máy chủ.
* **BYPASS**: Nội dung được yêu cầu bị bỏ qua và không được lưu trữ trong bộ nhớ cache

**c) Sử dụng web cache poisoning để khai thác lỗ hổng xử lý cookie**

- Cookie thường được sử dụng để tự động tạo nội dung trong response. Một ví dụ phổ biến là cookie cho biết ngôn ngữ ưu thích của người dùng.



- Lưu ý thông tin về phiên bản ngôn ngữ chỉ được chứa trong Cookie header. Nếu cache key chỉ chứa **request line** và **host header** mà không chứa cookie header thì nếu được lưu vào cache thì bất kể người dùng nào khi truy cập cũng sẽ nhận được phiên bản tiếng Ba Lan (pl=Polish) dù họ chọn ngôn ngữ nào

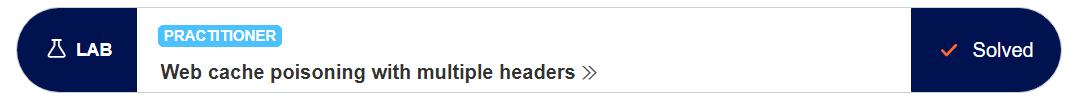


Cookie không được bao gồm trong cache key, và giá trị của cookie được ánh xạ trong response theo cách không an toàn -> chèn được alert().

**d) Sử dụng multiple headers để khai thác lỗ hổng web cache poisoning**

- Kẻ tấn công tạo ra một request thao túng nhiều unkeyed inputs

- Ví dụ trang web yêu cầu HTTPS, nếu nhận request sử dụng giao thức khác -> điều hướng đến chính nó sử dụng HTTPS. Nếu kết hợp với lỗ hổng trong URL thì có thể điều hướng người dùng đến URL độc hại



- **X-Forwarded-Proto(XFP):** Là header để xác định giao thức (HTTP hoặc HTTPS) mà client sử dụng để kết nối với proxy hoặc bộ cân bằng tải.

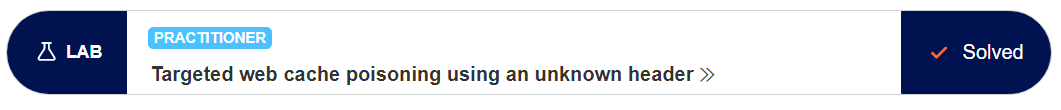
- **X-Forwarded-Scheme:** Tương tự X-Forwarded-Proto nhưng chỉ sử dụng để chỉ ra scheme (http hoặc https).

**X-Forwarded-Scheme: nothttps**

**e) Khai thác các response tiết lộ quá nhiều thông tin**

- **Cache-control directives:** response chứa thông tin tần xuất xóa cache hoặc tuổi của response được lưu vào cache

- **Vary header:** Chỉ định danh sách các header bổ sung mà nên được coi là một phần của cache key ngay cả khi chúng là unkeyed. Thường được sử dụng để chỉ định rằng header User-Agent là một phần của cache key.



Sử dụng **Param Miner** tìm được header **X-host** khả thi

Giá trị của **X-Host** được sử dụng động để tạo một URL tuyệt đối để nhập tệp JavaScript được lưu trữ tại /**resources/js/tracking.js**

Gửi request đã được chỉnh sửa đi. Trong response, **Vary** header được sử dụng chỉ định rằng **User-Agent** là một phần của cache key => lấy User-Agent của nạn nhân là xong

Tạo một phần tử <img> trỏ đến máy chủ khai thác. Bằng cách đó, có thể in ra giá trị User-Agent của nạn nhân

**2) Khai thác lỗ hổng triển khai bộ đệm - Exploiting cache implementation flaws**

**a) Lỗi cache key**

Trong thực tế, nhiều trang web và CDNs thực hiện các biến đổi khác nhau trên các thành phần khi chúng được lưu trong cache key. Có thể bao gồm:

* Loại trừ query string
* Lọc ra các query parameters cụ thể
* Bình thường hóa đầu vào trong các thành phần

Có thể gây ra một vài tính năng lạ lẫm không mong đợi. Chủ yếu dựa trên sự khác biệt giữa dữ liệu được ghi vào cache key và dữ liệu được truyền vào code mặc dù cùng một đầu vào. Có thể khai thác để độc chiếm cache.

**b) Phương pháp thăm dò cache**

Các kỹ thuật mới này phụ thuộc vào các lỗ hổng trong cài đặt và cấu hình cụ thể của bộ nhớ cache.

Phương pháp bao gồm các bước sau:

1. Xác định một nguồn thông tin về cache thích hợp (cache oracle)
2. Khảo sát xử lý cache key
3. Xác định một gadget có thể khai thác được (exploitable gadget)

**Xác định một nguồn thông tin về cache thích hợp (cache oracle)**

- Là một page hoặc endpoint cung cấp thông tin phản hồi về hành vị của cache. Nó phải có khả năng được lưu vào cache và phải có dấu hiệu nhận biết là nhận respone từ cache hay từ server. Response này có thể có nhiều hình thức khác nhau:

* Tiêu đề HTTP
* Sự thay đổi quan sát được đối với nội dung động
* Thời gian phản hồi khác biệt

Lý tưởng nhất là phản ánh toàn bộ URL và ít nhất một query parameter trong response.

third-party cache -> đọc document.

**Khảo sát xử lý cache key**

- Kiểm tra xem bộ nhớ cache có thực hiện bất kỳ xử lý bố sung nào trên input khi tạo cache key không. Ví dụ phổ biến:

+ Loại bỏ query parameter cụ thể hoặc cả tất cả query string

+ Bỏ port trong Host header

=> Cache thì nó sẽ loại payload đi để tạo cache key nên cache key sẽ vẫn bình thường nhưng ứng dụng web vẫn nhận payload và được phản hồi trong response => **Viết payload và phần bị cache bỏ. Loại hành vi này là khái niệm chính đằng sau tất cả các khai thác trong phần này.**

**Xác định một gadget có thể khai thác được (exploitable gadget)**

**-** Gadget thường là những lỗ hổng phía client như reflected XSS, open redirect + web cache posoining => lỗ hổng reflected thành lỗ hổng stored.

**c) Khai thác lỗi cache key**

**Unkeyed port**

Có thể cho phép xây dựng một cuộc tấn công từ chối dịch vụ. Cuộc tấn công kiểu này có thể được leo thang hơn nữa nếu trang web cho phép chỉ định một cổng không phải là số. Có thể sử dụng điều này để tiêm một XSS payload.

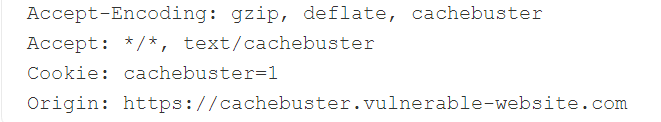
**Unkeyed query string**

Một trong những chuyển đổi cache-key phổ biến nhất là loại trừ toàn bộ query string

**\*Phát hiện unkeyed query string**

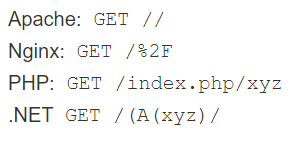
Nếu response cho biết có nhận được **cache hit** hay không thì dễ dàng nhận biết

Có cách khác nhau để thêm **cache buster,** chẳng hạn như thêm nó vào keyed header mà không gây ảnh hưởng đến hành vi ứng dụng. Một số ví dụ điển hình:

****

Sử dụng Param Miner có thể tự động thêm cache buster vào các keyed headers phổ biến.

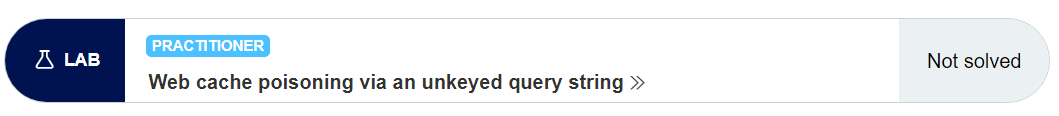
Một phương pháp khác là xem có sai khác nào giữa cách cache và back-end chuẩn hóa path của request vì path gần như chắc chắn có khóa. Có thể khai thác điều này để gửi request với khóa khác nhau nhưng vẫn trùng endpoint. Ví dụ các mục sau có thể được lưu cache riêng biệt nhưng được coi là tương đương với **GET /** trên back-end:



Sự chuyển đổi này có thể che giấu lỗ hổng reflected XSS.

**\*Exploit unkeyed query string**

Unkeyed query string có thể làm cho các lỗ hổng reflected XSS trở nên nghiêm trọng hơn



Thêm các parameter tùy ý vào request, vẫn nhận được lần truy cập vào cache ngay cả khi đã thay đổi parameter => chúng không được bao gồm trong cache key

**Origin HTTP header** được bao gồm trong cache key

Khi cache miss, parameter được tiêm vào được reflected trong response. Khi xóa parameter đi thì nó vẫn được reflected trong response.

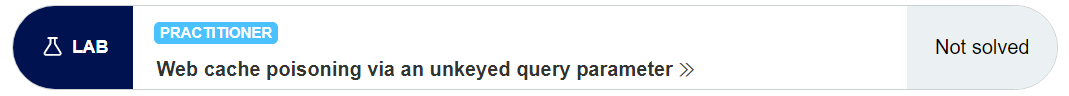
Tiêm payload vào parameter : *?evil='/><script>alert(1)</script>*

=> Việc Origin HTTP header được bao gồm trong cache key và có thể thêm **cache buster** vào chỉ để tạo cache key riêng của mình để phục vụ cho việc test. **Còn bản chất thì vẫn là query string bị bỏ qua bởi cache trong quá trình tạo cache key nhưng vẫn được reflected trong response.**

**Unkeyed query parameters**

Loại bỏ các query parameters không liên quan đến back-end như phân tích dữ liệu hoặc phục vụ quảng cáo. UTM parameter như **utm\_content** là ứng cử viên tốt trong quá trình thử nghiệm

Các tham số bị loại khỏi cache key không có khả năng ảnh hưởng đến response, không có gadget nào chấp nhận. Tuy nhiên, một số trang xử lý toàn bộ URL một cách dễ bị tấn công, khiến nó có thể khai thác các tham số tùy ý.



Các trang web thường loại trừ các **tham số phân tích UTM** nhất định khỏi cache key

Query string là một phần của cache key.

Sử dụng **Guess GET parameters** của Param Miner để xác định **utm\_content** được hỗ trợ.

Add parameter **utm\_content** vào query string vào gửi đến khi có **X-cache: hit** và nhận thấy nó được reflected trong respone. Thay đổi giá trị **utm\_content** và gửi lại vẫn nhận được response từ cache => **utm\_content** là unkeyed.

**Che dấu cache parameter**

Có khả năng cho phép đưa các tham số tùy ý vào logic ứng dụng bằng cách "che giấu" chúng trong một tham số bị loại trừ bởi cache.

Trước parameter đầu tiên của query string là dấu “?”, các parameter sau sẽ được đi trước bởi dấu “&”. Một số thuật toán yếu sẽ xử lý bất kỳ dấu “?” nào như là bắt đầu của một parameter mới, bất kể là parameter đầu tiên hay không.

Giả sử thuật toán để loại bỏ parameter khỏi cache key hoạt động theo cách này nhưng thuật toán của server chỉ chấp nhận dấu “?” đầu tiên là **dấu phân cách**



Cache sẽ xác định có 2 tham số và sẽ loại trừ tham số thứ 2 khỏi cache key. Nhưng server không chấp nhận dấu “?” thứ 2 là **dấu phân cách** và chỉ thấy có một tham số là **example,** giá trị của nó là toàn bộ chuỗi đằng sau bao gồm cả payload.

**\*Exploiting parameter parsing quirks**

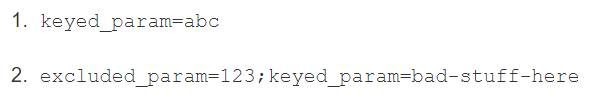
**Bản chất:** Cách xác định parameter của cache và server là khác nhau.

**Quirks:** đề cập đến hành vi không mong muốn hoặc bất thường theo cách mà bộ đệm ẩn hoặc ứng dụng phân tích cú pháp và xử lý các tham số truy vấn.

Trong Ruby on Rails, dấu “&” và dấu “;” là **dấu phân cách =>** có khả năng ghi đè giá trị của keyed parameter trong logic ứng dụng

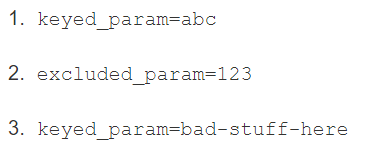


Cache: 2 tham số là:



Sau khi thuật toán phân tích cú pháp xóa **excluded\_param,** cache key chỉ chứa **keyed\_param=abc.**

Back-end: 3 tham số là:



Bị lặp **keyed\_param.** Sử dụng đến quirk thứ hai: Nếu có tham số trùng lặp, Ruby on Rails ưu tiên lần xuất hiện cuối cùng.

=> cache key lưu parameter có giá trị vô hại, response được lưu vào cache và phản hồi cho người dùng bình thường. Nhưng ở phía back-end, cũng là parameter đó nhưng mang một giá trị khác có hại sẽ được reflected trong response.

Khai thác này đặc biệt mạnh nếu cho phép kiểm soát một chức năng sẽ được thực thi. Ví dụ: nếu một trang web đang sử dụng JSONP để thực hiện cross-domain request, điều này thường sẽ chứa tham số **callback** để thực thi một chức năng nhất định trên dữ liệu được trả về

**JSONP (JSON with Padding):** Là một kỹ thuật được sử dụng để vượt qua hạn chế về chính sách Same-Origin của trình duyệt web. Các truy vấn Ajax có thể truy cập vào các tài nguyên từ origin khác nhau bằng cách sử dụng một đối tượng script để truy xuất dữ liệu JSON từ một tài nguyên nằm ở tên miền khác với tên miền hiện tại của trang web. **Kỹ thuật này cho phép các trang web lấy dữ liệu từ các nguồn khác nhau mà không cần phải lo lắng về Same-Origin Policy.**

**Parameter callback:** Đề giúp trình duyệt xử lý phản hồi JSON từ một tên miền khác, tham số callback được truyền vào truy vấn. Tham số này là một hàm JavaScript được định nghĩa trên trang web hiện tại và sẽ được gọi lại khi dữ liệu JSON được trả về.



Khi phản hồi từ tên miền khác được trả về, dữ liệu JSON sẽ được đóng gói trong một lời gọi hàm "handleResponse" và được truyền vào hàm này để xử lý.

**=> Giá trị của parameter callback là tên hàm dùng để xử lỷ dữ liệu JSON từ miền khác trả về**



=> có thể sử dụng các kỹ thuật này để ghi đè chức năng **callback** dự kiến ​​và thay vào đó thực thi JavaScript tùy ý.



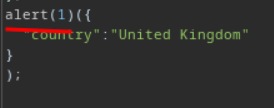
Hỗ trợ **utm\_content** parameter và là một unkeyed. Nếu sử dụng dấu “;” nối thêm một parameter khác vào **utm\_content** thì cache coi đây là một tham số duy nhất.

Param Miner -> Bulk scan -> Rails parameter cloaking scan => Thấy được mọi page đều import */js/geolocate.js?callback=setCountryCookie*

**Có thể kiểm soát tên của hàm được gọi trên dữ liệu trả về bằng cách chỉnh sửa parameter callback.**

*GET /js/geolocate.js?callback=setCountryCookie&utm\_content=foo;callback=alert(1)*

Cách thực thi giống như phần lý thuyết. Kết quả thu được

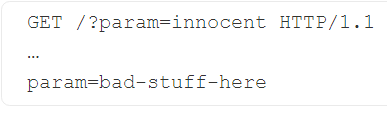


**Exploiting fat GET support - Khai thác tính năng GET dày đặc**

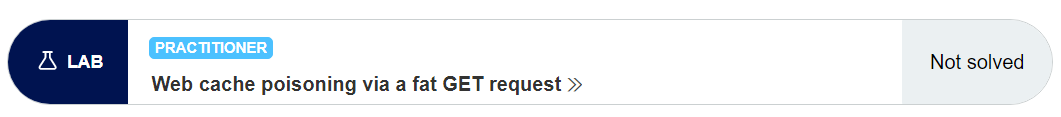
Kỹ thuật này thường được sử dụng để tấn công các ứng dụng web bằng cách gửi các yêu cầu GET có kích thước lớn hoặc chứa nhiều tham số, từ đó tạo ra các tình huống quá tải hoặc lỗi trên máy chủ đích.

Trong một số trường hợp, HTTP method không phải là keyed.

Thêm body vào GET request để tạo “fat” GET

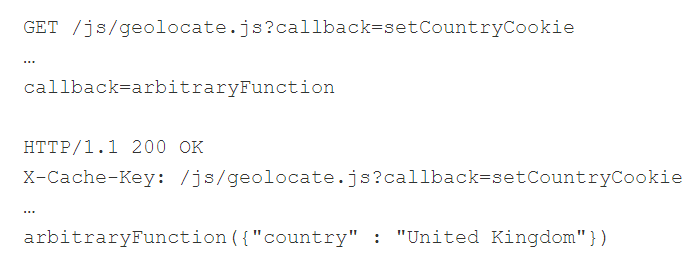


Cache dựa trên request line nhưng phía server sẽ lấy giá trị từ body.

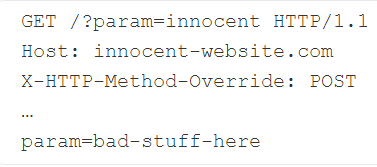


Mọi page đều import */js/geolocate.js?callback=setCountryCookie*

Kiểm soát tên của hàm được gọi bằng cách truyền parameter callback lặp lại trong body.



- Đôi khi có thể xử lý “fat” GET bằng cách ghi đè HTTP method



- Miễn là header **X-HTTP-Method-Override** là unkeyed, có thể gửi request giả POST trong khi vẫn giữ cach key GET lấy từ request line

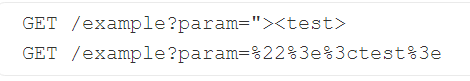
**Exploiting dynamic content in resource imports**

Imported resource files thường là tĩnh nhưng một số reflec input từ query string => có thể inject content vào resource file.

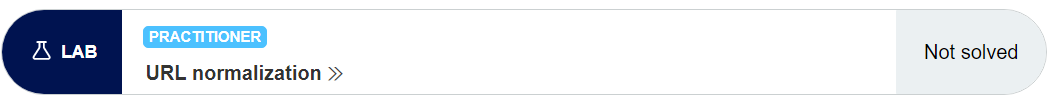
**Cache keys được chuẩn hóa – Normalized cache keys**

**-** Dữ liệu độc hại được reflected trong response.

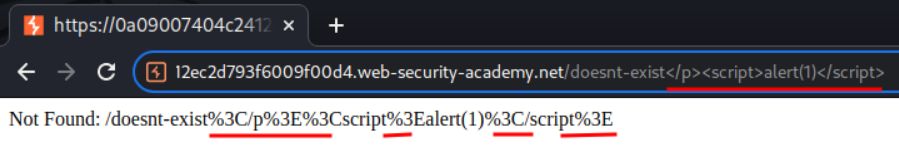
- Một số triển khai cache chuẩn hóa keyed input khi thêm nó vào cache key. Trong trường hợp đó, 2 request dưới đây là cùng một key

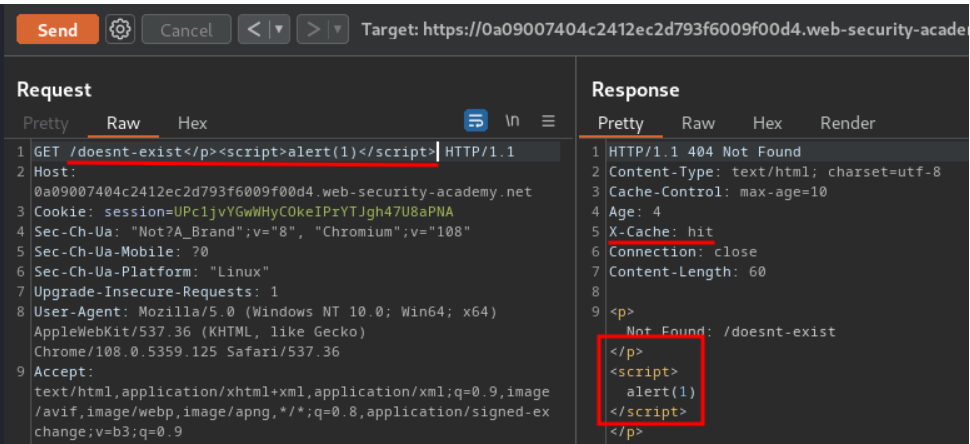


Nếu gửi request bằng Burp Repeater, có thể poison web cache với payload XSS chưa được mã hóa. Khi victim truy cập URL độc hại, payload sẽ vẫn được trình duyệt của họ **mã hóa URL**. Vì **cache chuẩn hóa keyed input khi thêm nó vào cache key** => Cache đã chuẩn hóa cái **URL** => nó **vẫn cùng cache key** với response chứa payload chưa được mã hóa được lưu trong cache.



Đường dẫn được cung cấp được phản ánh tới trang web. Nhưng bị mã hóa URL các ký tự cần thiết khi gửi request và máy chủ không giải mã chúng. Phản hồi mà nạn nhân nhận được sẽ chỉ chứa một chuỗi đã mã hóa URL vô hại.





Có thể thấy payload không bị mã hóa. Truy cập lại URL thì sẽ pop up alert(1)

Nó được thực thi vì payload được mã hóa của trình duyệt đã được bộ đệm giải mã URL, gây ra lỗi bộ đệm với yêu cầu trước đó.

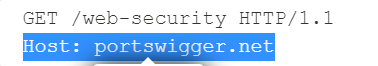
------------------------------------------------------------------------------------------------------------------------------------------

**WEB14: HTTP Host header attacks**

**A. Tìm hiểu HTTP Host header attacks**

**1) HTTP Host header là gì?**

Là tiêu đề bắt buộc kể từ HTTP/1.1. Nó chỉ định tên miền mà client muốn truy cập.



Trong một số trường hợp, chẳng hạn như khi yêu cầu đã được chuyển tiếp bởi một hệ thống trung gian, Host value có thể bị thay đổi trước khi nó đến được back-end dự kiến.

**2) Mục đích của HTTP Host header**

- Giúp xác định thành phần back-end nào mà client muốn giao tiếp với.

- Nếu không có Host header hoặc dị dạng có thể ảnh hưởng đến việc định tuyến request đến ứng dụng dự kiến

\* **Virtual hosting (máy chủ ảo)**

Một tình huống có thể xảy ra là khi một máy chủ web lưu trữ nhiều trang web hoặc ứng dụng. Đây có thể là:

**+** Nhiều trang web có một chủ sở hữu duy nhất

+ Các trang web có nhiều chủ sở hữu khác nhau được lưu trữ trên một nền tảng được chia sẻ duy nhất

=> Mỗi trang web riêng biệt có một tên miền khác nhau nhưng tất cả chúng đều chia sẻ một địa chỉ IP chung với máy chủ. Các trang web được lưu trữ theo cách này trên một máy chủ duy nhất được gọi là “máy chủ ảo”

Đối với người dùng bình thường truy cập trang web, máy chủ ảo thường không thể phân biệt được với trang web được lưu trữ trên máy chủ chuyên dụng của chính nó.

\* **Định tuyến lưu lượng qua một bên trung gian**

Lưu lượng giữa client và server được định tuyến thông qua một hệ thống trung gian. (bộ cân bằng tải, reverse proxy,…)

Các trang web được lưu trữ trên các back-end server riêng biệt, nhưng tất cả các tên miền của chúng đều phân giải thành một địa chỉ IP duy nhất của thành phần trung gian.

=> Vấn đề tương tự máy chủ ảo vì reverse proxy hoặc bộ cân bằng tải cần biết back-end thích hợp mà nó sẽ định tuyến từng yêu cầu.

**Tóm lại: Vấn đề là do nhiều ứng dụng có thể truy cập từ một địa chỉ IP**

**3) HTTP Host header giải quyết vấn đề**

Host header được đưa vào để chỉ định người nhận dự kiến. Khi gửi request, URL sẽ phân giải thành địa chỉ IP của một máy chủ cụ thể. Khi máy chủ nhận được request, nó sẽ tham chiếu đến Host header để xác định backend dự định và gửi request tương ứng.

**4) Tấn công HTTP Host header attacks là gì?**

Nếu máy chủ hoàn toàn tin tưởng vào Host header và không xác thực hoặc thoát khỏi tiêu đề đó đúng cách, thì kẻ tấn công có thể sử dụng thông tin đầu vào này để đưa vào các tải trọng có hại thao túng hành vi phía máy chủ. Gọi là **Host header injection**

Các ứng dụng web có sẵn thường không biết chúng được triển khai trên miền nào trừ khi miền đó được chỉ định thủ công trong tệp cấu hình trong quá trình thiết lập. Ví dụ: để tạo một URL tuyệt đối có trong email, họ có thể sử dụng cách truy xuất tên miền từ Host header



Host header có thể kiểm soát bởi người dùng. Nếu đầu vào không được thoát hoặc xác thực đúng cách, có thể gây ra một số lỗ hổng khác:

* Web cache poisoning
* Business logic flaws trong chức năng cụ thể
* Routing-based SSRF
* Lỗ hổng cổ điển phía máy chủ: SQL injection,…

**5) Lỗ hổng HTTP Host header phát sinh thế nào?**

Giả định sai lầm rằng Host header không thể kiểm soát bởi user => Tin tưởng tuyệt đối Host header.

Host header có thể bị ghi đè bằng cách đưa vào các header khác

Cấu hình không an toàn của một hoặc nhiều thành phần trong cơ sở hạ tầng liên quan. Các sự cố cấu hình này có thể xảy ra do các trang web tích hợp công nghệ của bên thứ ba vào kiến ​​trúc của chúng mà không nhất thiết phải hiểu các tùy chọn cấu hình và ý nghĩa bảo mật của chúng.

**B. Cách nhận biết và khai thác lỗ hổng HTTP Host header**

**1) Cách kiểm tra lỗ hổng sử dụng HTTP Host header**

Thay đổi Host header và xem xét các phản hồi

**2) Cung cấp Host header tùy ý**

Bước đầu tiên là kiểm tra điều gì sẽ xảy ra khi cung cấp một tên miền tùy ý, không được nhận dạng thông qua Host header.

Một số Proxy chặn lấy trực tiếp địa chỉ IP từ Host header => loại thử nghiệm này gần như không thể vì bất cứ thay đổi nào với Host header khiến yêu cầu được gửi đến địa chỉ IP hoàn toàn khác.

**Burp Suite tách biệt giữa địa chỉ IP và Host header => thay đổi Host header tùy ý mà yêu cầu vẫn được gửi đến mục tiêu đã định**

Đôi khi có thể truy cập trang web mục tiêu ngay cả khi cung cấp Host header không mong muốn.

**3) Kiểm tra xác nhận thiếu sót - Check for flawed validation**

**Nên cố gắng hiểu cách trang web phân tích cú pháp của Host header.** Điều này đôi khi có thể tiết lộ những sơ hở có thể được sử dụng để bỏ qua việc xác thực.

Ví dụ: Chỉ xác thực domain, bỏ qua port. Có thể cung cấp một cổng không phải là số => chèn payload vào



Cho phép các tiên miền phụ tùy ý theo một logic nhất định. Bỏ qua xác thực bằng cách đăng ký một tên miền tùy ý.



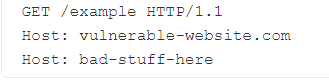
Sử dụng tên miền phụ kém an toàn hơn đã xâm phạm



**4) Gửi request mơ hồ**

**a) Tiêm Host header trùng lặp**

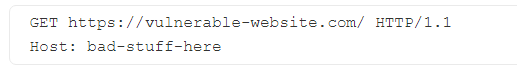
Thông thường một trong hai tiêu đề sẽ được ưu tiên hơn tiêu đề còn lại, ghi đè giá trị của nó một cách hiệu quả



Giả sử FE ưu tiên header 1 nhưng BE ưu tiên header 2 => có thể sử dụng header 1 để đảm bảo yêu cầu được chuyển đến mục tiêu đã định và header 2 để truyền payload vào server

**b) Cung cấp một URL tuyệt đối**

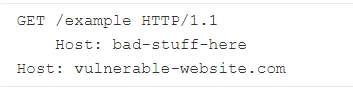
Sự không rõ ràng do cung cấp cả URL tuyệt đối và Host header cũng có thể dẫn đến sự khác biệt giữa các hệ thống khác nhau. Request line được ưu tiên khi định tuyến yêu cầu => URL tuyệt đối ở request line



Cũng có thể cần thử nghiệm với các giao thức khác nhau. Máy chủ đôi khi sẽ hoạt động khác nhau tùy thuộc vào việc request line có chứa URL HTTP hay HTTPS hay không.

**c) Add line wrapping**

Một số máy chủ sẽ diễn giải header được thụt lề dưới dạng một dòng được bao bọc và do đó, coi như một phần của giá trị header trước đó.

****

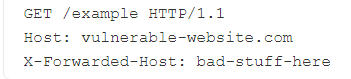
**5) Inject host override headers**

Các trang web thường được truy cập thông qua một số loại hệ thống trung gian, ví dụ như bộ cân bằng tải hoặc proxy ngược. Host header mà máy chủ nhận được có thể chứa domain của một trong những hệ thống trung gian này và thường không liên quan đến chức năng được yêu cầu.

**X-Forwarded-Host:** Chứa giá trị ban đầu của Host header từ request của client

Khi có X-Forwarded-Host, sẽ ưu tiên header này hơn

Có thể sử dụng X-Forwarded-Host để đưa đầu vào độc hại vào trong khi phá vỡ bất kỳ xác thực nào trên chính Host header



Một số header tương tự



**C. How to exploit the HTTP Host header**

**1) Password reset poisoning**

Kẻ tấn công thao túng một trang web dễ bị tổn thương để tạo liên kết đặt lại mật khẩu trỏ đến một miền dưới sự kiểm soát của chúng. Hành vi này có thể được tận dụng để đánh cắp mã thông báo bí mật cần thiết để đặt lại mật khẩu của người dùng tùy ý và cuối cùng là xâm phạm tài khoản của họ.

**a) Quá trình đặt lại mật khẩu hoạt động thế nào?**

Một trong những cách tiếp cận phổ biến:

- Người dùng nhập tên người dùng hoặc địa chỉ email của họ và gửi yêu cầu đặt lại mật khẩu.

- Trang web kiểm tra xem người dùng này có tồn tại hay không và sau đó tạo mã thông báo tạm thời, duy nhất, có entropy cao, mã này liên kết với tài khoản của người dùng ở back-end

- Trang web gửi email đến người dùng có chứa liên kết để đặt lại mật khẩu của họ. Token đặt lại duy nhất của người dùng được bao gồm dưới dạng tham số truy vấn trong URL tương ứng: 

- Khi người dùng truy cập URL này, trang web sẽ kiểm tra xem token được cung cấp có hợp lệ hay không và sử dụng nó để xác định tài khoản nào đang được đặt lại. Nếu mọi thứ đúng như mong đợi, người dùng được cung cấp tùy chọn nhập mật khẩu mới. Cuối cùng, token bị phá hủy.

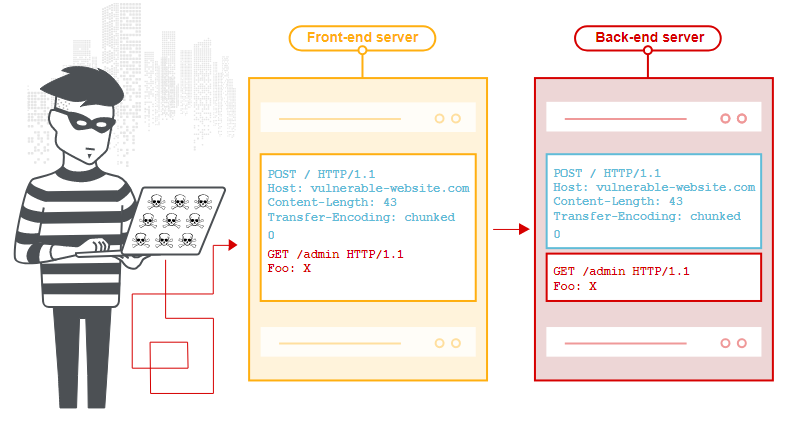
**b) Cách xây dựng cuộc tấn công password reset poisoning**

- Kẻ tấn công thay mặt user gửi request đặt lại mật khẩu. Khi gửi form, họ sẽ chặn yêu cầu HTTP kết quả và sửa đổi Host header để nó trỏ đến một miền mà họ kiểm soát: **evil-user.net**

------------------------------------------------------------------------------------------------------------------------------------------

**WEB15: HTTP request smuggling**

**A. Tìm hiểu HTTP request smuggling là gì?**



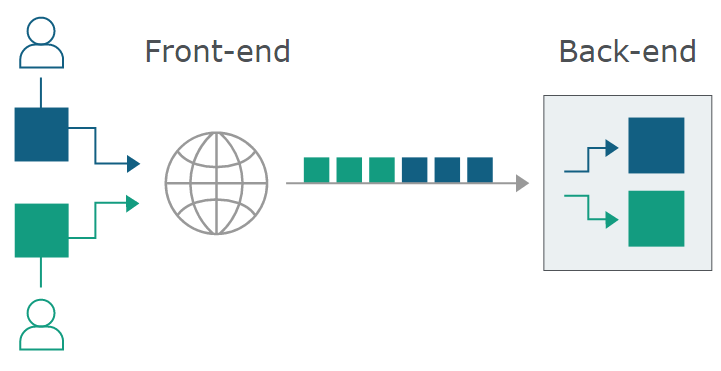
**1) HTTP Request Smuggling là gì?**

- Là một kỹ thuật khai thác sự bất đồng bộ xử lý gói tin HTTP giữa front-end proxies (reverse proxy/balancer/...) với máy chủ web/backend.

- Lỗ hổng xảy ra khi sự bất đồng bộ giữa front-end proxies và backend-server cho phép kẻ tấn công gửi một HTTP request được hiểu là một request duy nhất ở front-end proxies nhưng là 2 request ở back-end server. Cho phép kẻ tấn công thay đổi request tiếp theo tới back-end server.

**2) Chuyện gì xảy ra trong cuộc tấn công HTTP request smuggling?**

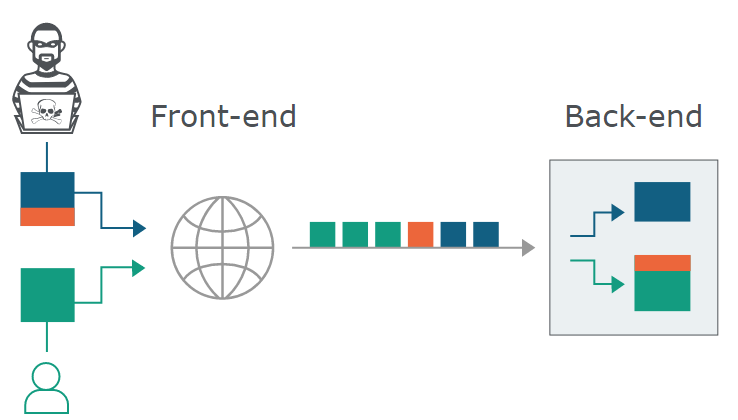
- Các ứng dụng web ngày nay thường sử dụng chuỗi máy chủ HTTP giữa người dùng và backend. Người dùng gửi yêu cầu đến máy chủ **front-end** (thường là các máy chủ cân bằng tải hoặc là **reverse proxy**), sau đó mới gửi tới máy chủ **back-end**.



- Khi front-end server chuyển tiếp các request đến back-end server, nó thường gửi một số request qua cùng một kết nối back-end network.

- Các HTTP request được gửi liên tiếp nhau, và máy chủ nhận (front-end proxies) sẽ phân tích các tiêu đề yêu cầu HTTP để xác định nơi một yêu cầu kết thúc (back-end server) và yêu cầu tiếp theo bắt đầu. Điều quan trọng là hệ thống front-end và back-end phải thống nhất về ranh giới giữa các yêu cầu

- Kẻ tấn công có thể gửi một request không rõ ràng mà front-end proxies và back-end server hiểu khác nhau



Kẻ tấn công khiến 1 phần của front-end request được back-end server hiểu là bắt đầu của request tiếp theo (half-request) => request ngay phía sau sẽ bị đính vào làm 1 phần của half-request => can thiếp vào cách ứng dụng xử lý yêu cầu đó.

**3) Lỗ hổng HTTP request smuggling phát sinh thế nào?**

**-** Hầu hết phát sinh do 2 header: **Content-Length** và **Transfer-Encoding**

**- Content-Length:** kích thước của body tính theo byte.

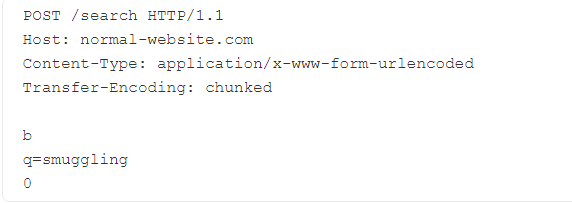


**- Transfer-Encoding:** Cho biết phương thức mã hóa dữ liệu ở phần body. Ở đây đề cập đến kiểu **chunked.** Nội dung body chứa một hoặc nhiều khối dữ liệu



**-** Khi dữ liệu body được **chunked**, nó sẽ có dạng: ký tự b đầu tiên chính là kích thước của đoạn **chunked** theo byte(được biểu thị dưới dạng hex), tiếp đến nội dung **chunked,** và kết thúc là một đoạn **chunked** có kích thước bằng 0. (2 chunked).

**q=smuggling có độ dài là 11. Theo bảng mã ascii thì giá trị hex của 11 là b**



Một message có thể sử dụng cả **Content-Length** và **Transfer-Encoding** sao cho chúng xung đột với nhau. Nếu có cả hai header thì **Content-Length** bị bỏ qua. Điều này có thể tránh sự mơ hồ khi chỉ có một server, nếu có nhiều hơn một server liên kết với nhau thì các vấn đề có thể phát sinh vì hai lý do:

* Một số server không hỗ trợ *Transfer-Encoding* header trong request
* Một số server có hỗ trợ *Transfer-Encoding* header có thể không xử lý nó nếu *Transfer-Encoding* header bị xáo trộn theo một cách nào đấy

-> Chốt lại: nếu FE server và BE server có hoạt động khác nhau đối với TE header, thì có thể không đồng nhất về ranh giới giữa 2 request, dẫn đến lỗ hổng smuggling.

**4) Cách thực hiện cuộc tấn công HTTP Request Smuggling**

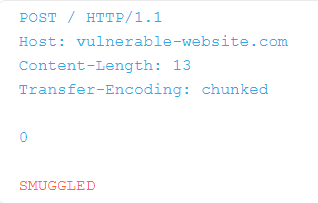
Tấn công **HTTP Request Smuggling** xoay quanh đến hai header là **Content-Length** và **Transfer-Encoding** trên cùng một gói tin HTTP để máy chủ **front-end** và **back-end** xử lý yêu cầu theo cách khác nhau. Một số dạng thường gặp:

- **CL.TE:** máy chủ **front-end** sử dụng header **Content-Length** và máy chủ **back-end** sử dụng header **Transfer-Encoding**.

**- TE.CL:** máy chủ **front-end** sử dụng header **Transfer-Encoding** và máy chủ **back-end** sử dụng header **Content-Length.**

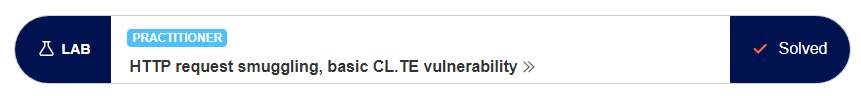
**- TE.TE:** máy chủ **front-end** và **back-end** đều hỗ trợ header **Transfer-Encoding**, nhưng một trong hai loại máy chủ không xử ý được header này, do gói tin HTTP đã bị làm xáo trộn header theo một cách nào đó.

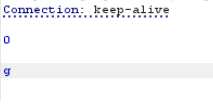
**a) Lỗ hổng CL.TE**



Front-end server sử dụng **Content-Lenght** header và xác định độ dài của body là 13 bytes.

Back-end server sử dụng **Transfer-Encoding** header. Nó xử lý chunked đầu tiên, có độ dài bằng 0 và do đó được coi là kết thúc request. Các byte sau, **SMUGGLED** không được xử lý và back-end server sẽ coi các byte này là phần bắt đầu của request tiếp theo trong chuỗi.

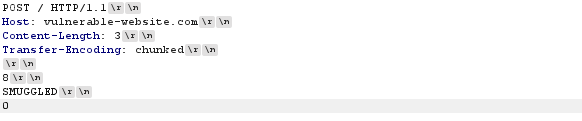


****

Do khi gửi request lần đầu tiên, back-end server xử lý chunked đầu tiên có độ dài bằng 0 nên coi như là kết thúc request đó. Khi gửi request lần 2 thì những byte không được xử lý ở request trước đó sẽ được dính vào phần bắt đầu của request thứ 2 nên mới có phương thức là GPOST.

**Connection: keep-alive**: Duy trì kết nối của client và server sau khi gửi response, cho phép client và server tiếp tục giao tiếp với nhau mà không cần mở kết nối mới.

**b) Lỗ hổng TE.CL**



Note:

+ Bỏ chọn **Update Content-Length**

+ Bao gồm *\r\n\r\n* sau số 0 (đề kết thúc request)

Front-end server xử lý chunked đầu tiên **SMUGGLED** sau đó xử lý tiếp chunked thứ 2 có độ dài là 0 => kết thúc request và gửi đến back-end server

Back-end server xử lý **Content-Length** header và xác định rằng body có kích thước là 3 byte (8, /r, /n). Các bytes sau bắt đầu bằng **SMUGGLED**, không được xử lý và back-end server sẽ coi các byte này là phần bắt đầu của yêu cầu tiếp theo trong chuỗi

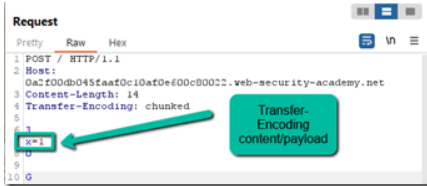
Cách tính kích thước đối với **Transfer-Encoding: chunked** và **Content-Length:** Không tính **/r/n** cuối cùng.

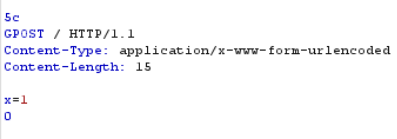
**Các ký tự chuẩn để kết thúc một dòng (/r/n) trong yêu cầu HTTP được tính vào độ dài nội dung**



Nếu payload giống trường hợp CL.TE thì method sẽ là **G0POST** chứ không phải **GPOST**

Để kết quả là **G0POST** thì trong content/payload của **Transfer-Encoding** ta viết luôn hẳn một request có method là **GPOST**



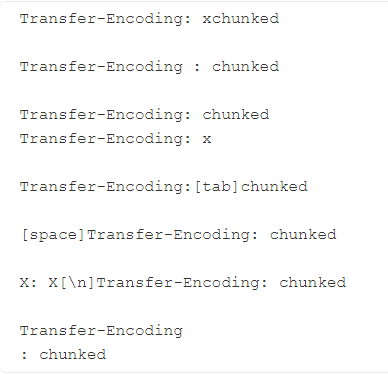


Nếu **Content-Length: 10** thì **bài lab vẫn sẽ được giải quyết** nhưng sẽ luôn nhận được 200 OK và response bình thường ở phía client. Bời vì back-end sẽ xử lý request này **ngay** vì **Content-Length** đã đúng. Không có hàng đợi xảy ra. Để kích hoạt hàng đợi, cần làm cho back-end đợi ít nhất một ký tự nữa trước khi giải phóng yêu cầu GPOST. Bằng cách chỉ định **Content-Length** dài hơn payload được cung cấp, back-end sẽ đợi thêm dữ liệu trước khi xử lý và phát hành => **Content-Length > 10 là được. Nếu để bằng 10: Bài lab sẽ giải quyết nhưng sẽ không thấy GPOST được gửi lại cho phía client.**

**c) TE.TE: làm xáo trộn tiêu đề TE**

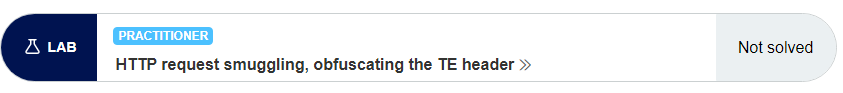
FE-server và BE-server đều hỗ trợ **Transfer-Encoding** header, nhưng một trong các máy chủ có thể được yêu cầu không xử lý nó bằng cách xáo trộn header theo một cách nào đó

Có vô số cách làm xáo trộn Transfer-Encoding header



Để phát hiện một lỗ hổng TE.TE, cần tìm một biến thể của tiêu đề Transfer-Encoding sao cho chỉ có một trong hai máy chủ front-end hoặc back-end xử lý nó, trong khi máy chủ còn lại bỏ qua nó.

Phụ thuộc vào việc FE hoặc BE server không xử lý Transfer-Encoding header, phần còn lại của cuộc tấn công sẽ có hình thức giống như đối với lỗ hổng CL.TE hoặc TE.CL



Bỏ chọn “Update Content-Length”.

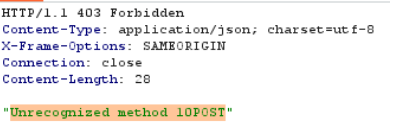
Kiểm tra xem tồn tại lỗ hổng CL.TE và TE.CL: Đặt Content-Length thành một giá trị không chính xác (quá dài hoặc quá ngắn) luôn mang lại phản hồi 200 => FE và BE đều hỗ trợ **Transfer-Encoding**

Test xem có bỏ qua Transfer-Encoding để quay về Content-Length không: Thêm một header Transfer-Encoding nữa lúc này sẽ có 2 cái.

Nếu sửa header một thành giá trị xáo trộn thì nhận 500 => FE hoặc BE server sẽ xét header Transfer-Encoding đầu tiên.

Thử đổi lại thứ tự và vẫn giữ giá trị xáo trộn





=> Có vẻ như FE server đang xử lý yêu cầu theo tiêu đề Trasfer-Encoding đầu tiên và BE server đang cố xử lý yêu cầu theo tiêu đề Transfer-Encoding thứ hai, tiêu đề này có giá trị không hợp lệ, do đó, BE server sau đó sẽ xử lý yêu cầu theo tiêu đề Content-Length. Với tiêu đề **TE bị xáo trộn**, lỗ hổng trở thành **lỗ hổng TE.CL**

**Tóm lại:** Thêm một header Transfer-Encoding có giá trị sai vào và xem response để xác định xem là lỗ hổng CL.TE hay TE.CL

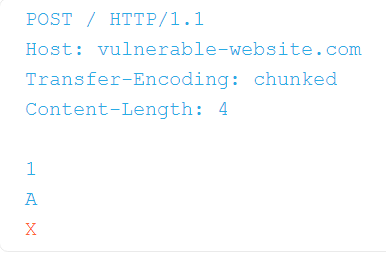
**B. Finding HTTP request smuggling vulnerabilities**

**1) Finding HTTP request smuggling vulnerabilities using timing techniques**

Cách hiệu quả nhất để phát hiện HTTP request smuggling là gửi các request sẽ gây ra delay trong response nếu có lỗ hổng.

**a) Finding CL.TE vulnerabilities using timing techniques**

Gửi request như sau thường gây ra time delay:

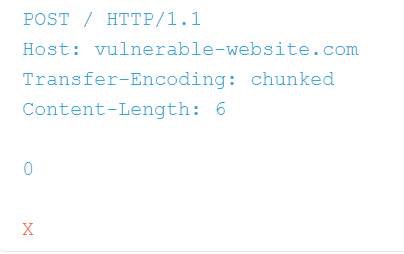


Vì FE sử dụng CL nên nó sẽ chỉ chuyển tiếp một phần của request, bỏ qua dấu X. BE sử dụng TE, xử lý chunked đầu tiên, vì không có chunked size 0 để kết thúc nên BE sẽ đợi chunked tiếp theo đến => gây time delay.

**Time delay do không có chunked size 0 để kết thúc**

**b) Finding TE.CL vulnerabilities using timing techniques**

Gửi request như sau thường gây ra time delay:



Vì FE sử dụng TE nên nó chỉ chuyển một phần của request, bỏ qua dấu X. BE sử dụng CL, mong đợi nhiều nội dung hơn trong body và đợi nội dung còn lại đến => gây time delay

**Time delay do kích thước body nhỏ hơn giá trị Content-Length**

**2) Xác nhận lỗ hổng HTTP request smuggling sử dụng reponse khác nhau**

Khi phát hiện có lỗ hổng, có thể khai thác để kích hoạt sự khác nhau trong response. Điều này liên quan đến việc gửi hai yêu cầu đến ứng dụng liên tiếp nhanh chóng:

* Attack request được thiết kế vào quá trình xử lý request tiếp theo
* Request bình thường

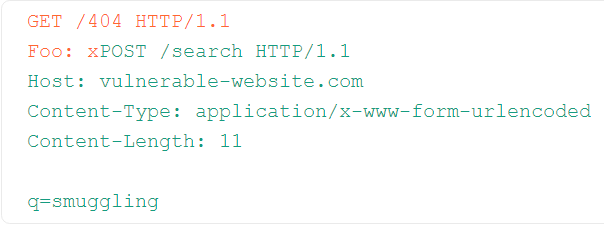
Nếu response với request bình thường chứa sự can thiệp dự kiến => lỗ hổng được xác nhận

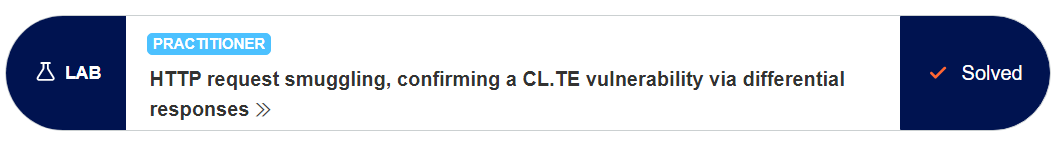
**a) Confirming CL.TE vulnerabilities using differential responses**

Attack request:



Nếu thành công:





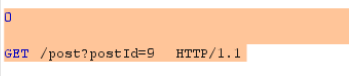
**Detect CL.TE hoặc TE.CL:**

Thêm header Transfer-Encoding, body không phải định dạng chunked. Response trả 500 => FE hoặc BE ưu tiên header TE.

Sửa body thành đúng dạng chunked. Để Burp tự cập nhất CL, nếu gửi request thành công => có tồn tại CL.TE hoặc TE.CL

**Thử exploit CL.TE:**

Thay đổi giá trị header CL (tăng 1 và giảm 1) thì thấy phản hồi 2 **lỗi** khác nhau => có thế suy đoán FE ưu tiên CL và BE ưu tiên TE

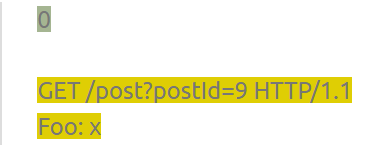




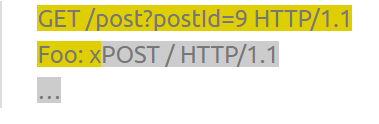
Invalid request vì: Không đúng định dạng request line:



Định dạng đúng: **VERB[space]PATH[space]PROTOCOL**. Để request line có định dạng đúng, thì chỉ cần thêm nội dung vào header sẽ bị bỏ qua bởi cả FE và BE



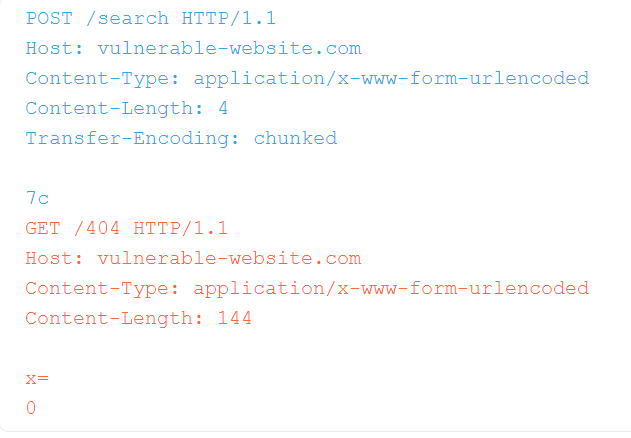
Request sẽ được nối thế này:



Foo không phải header nên không được thực hiện => mất POST.

**b) Confirming TE.CL vulnerabilities using differential responses**

Attack request:



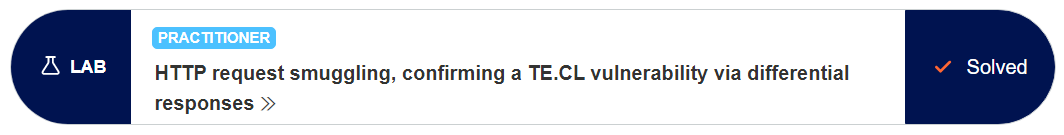
Note:

+ Tắt Update Content-Length

+ Thêm **\r\n\r\n** sau số 0

Nếu tấn công thành công:





Làm giống lab trên để xác nhận có tồn tại lỗ hổng CL.TE hoặc TE.CL

Giảm CL đi 1 => nhận 200 OK

Tăng CL lên 1 => nhận 500

Nếu CL nhỏ hơn hoặc bằng kích thước của payload thực tế thì luôn nhận được 200 còn nếu lớn hơn thì nhận được 500 => Khi gửi với CL được đặt quá dài, rất có thể đang gây ra hàng đợi trên BE-server. BE-server đang chờ nội dung bổ sung cho yêu cầu...không bao giờ hiển thị nên ứng dụng sẽ hết thời gian chờ sau một khoảng thời gian.

**Một số lưu ý khi xác nhận HRS bằng các response khác biệt:**

* Attack request và nomal request phải được gửi bằng các kết nối mạng khác nhau
* 2 request phải sử dụng cùng tên URL và parameter càng nhiều càng tốt để tăng khả năng cả 2 đều được xử lý bởi cùng 1 BE server
* Nếu 2 request được gửi đến 2 BE server thì kiểm tra thất bại, do đó nên gửi nomal request ngay sau attack request
* Cẩn trọng vì có thể ngay sau attack request là 1 request từ người dùng ->ảnh hưởng xấu đến người dùng

**C. Exploiting HTTP request smuggling vulnerabilities**

**a) Using HTTP request smuggling to bypass front-end security controls**

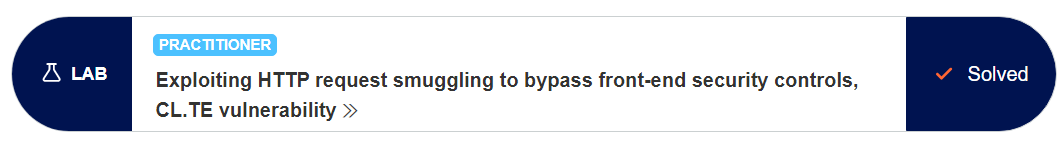
Đôi khi FE server có sử dụng sercurity control, và request sẽ chỉ được chuyển đến BE server khi nó đã thông qua sercurity control của FE server.

Lỗ hổng ở đây là BE server hoàn toàn tin tưởng FE server, cho rằng request đã qua FE server thì không cần kiểm tra gì nữa và tiếp tực xử lý.

Ví dụ: Người dùng được truy cập /home nhưng không được truy cập /admin. Có thể bypass bằng cách sau:

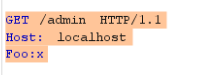


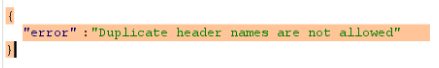
FE server nhìn thấy 2 request đều đến /home do đó các request được chuyển đến BE. Tuy nhiên BE nhìn thấy 1 request đến /home, 1 đến /admin. Nó giả định như mọi khi rằng các request đã đi qua FE controls do đó cấp quyền truy cập vào URL hạn chế



Các bước làm ở phía trên giống với lab xác nhận lỗ hổng CL.TE thông qua response khác nhau

Khi một giao diện chỉ khả dụng cho người dùng cục bộ, điều đó thường có nghĩa là giao diện đó phải được truy cập từ http(s)://localhost, http(s)://127.0.0.1 hoặc một số phương thức chỉ định quyền truy cập cục bộ khác.



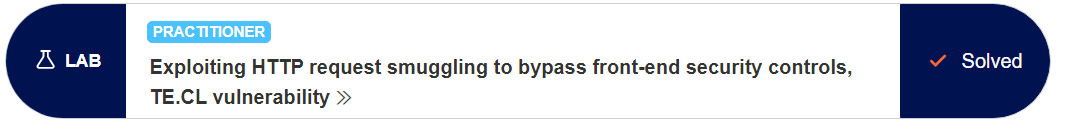


Do có 2 header **Host**

Không có cách nào để thực sự kiểm soát các tiêu đề của yêu cầu thứ hai. Do đó, cách duy nhất để loại bỏ một tiêu đề trùng lặp là đưa nó ra khỏi phần đầu của yêu cầu và di chuyển nó đến body.

Phải xây dựng toàn bộ yêu cầu mà chúng ta muốn gửi đường dẫn /admin và nối phần đầu của yêu cầu thứ hai vào body.





Khi một giao diện chỉ khả dụng cho người dùng cục bộ, điều đó thường có nghĩa là giao diện đó phải được truy cập từ http(s)://localhost, http(s)://127.0.0.1 hoặc một số phương thức chỉ định quyền truy cập cục bộ khác.

Cách làm tương tự như lab xác minh lỗi hổng TE.CL thông qua response khác nhau.

**b) Revealing front-end request rewriting – Để lộ những gì FE viết lại**

Một vài trường hợp, FE server sẽ viết lại request trước khi chúng chuyển đến BE server, thường là thêm vài header VD:

* Chấm dứt kết nối TLS và thêm một số tiêu đề mô tả giao thức và mật mã đã được sử dụng
* Có thể thêm X-Forwarded-For header chứa user IP
* Xác định ID người dùng dựa trên session token và thêm header xác định user
* thêm 1 só thông tin nhạy cảm có lợi cho các cuộc tấn công khác

Nếu trong smuggle request thiếu một vài header thường được FE server thêm vào thì BE server có thể k xử lý theo cách thông thường (ta cần BE server xử lý bình thường thì mới biết được là HRS có thành công không)

Một cách đơn giản để tiết lộ chính xác cách FE server đang viết lại các request. Thực hiện các bước sau:

* Tìm POST request mà nó reflect giá trị của parameter trong response
* Xáo trộn các parameter để reflected parameter xuất hiện cuối cùng trong message body
* Gửi request này đến BE, trực tiếp theo sau là một request thông thường có dạng viết lại mà muốn tiết lộ

Request:



Kết quả response có chứa:



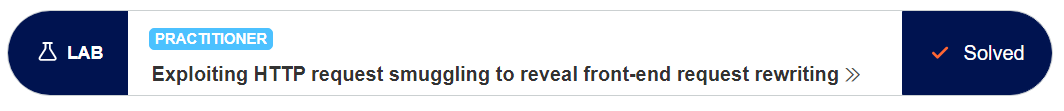
Attack request:



Request sẽ được FE server viết lại để bổ sung các header, sau đó BE sẽ xử lý smuggeled request và coi request thứ 2 được viết lại là giá trị của parameter email. Sau đó nó sẽ reflec giá trị này trong response cho request thứ hai

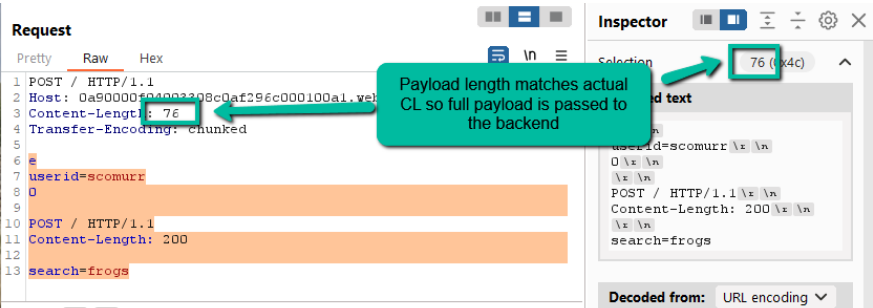


Vì request cuối cùng (request bị dính) đang được viết lại, nên sẽ không biết nó kết thúc trong bao lâu. Giá trị **Content-Length trong smuggled request** sẽ quyết định xem kích thước request bị dính là bao nhiêu. Nếu mà dính ít quá thì không tìm được header bổ sung, nhiều quá thì time out.



Detect được sử dụng: CL.TE

Giá trị tìm kiếm được phản ánh lại trong response => tận dụng để phản ảnh lại những header được bổ sung trở lại phía client.

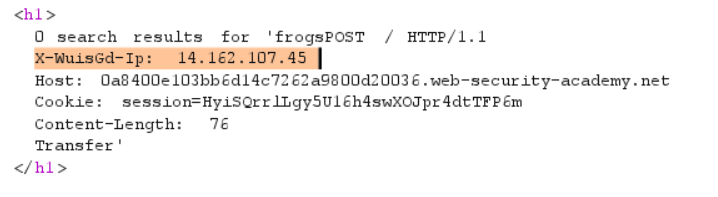


Phần request được lưu vào hàng đợi là:

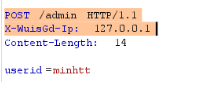


Vì giá trị CL lớn hơn kích thước payload, nên nó sẽ phải đợi request tiếp theo đến nối vào để đủ 200. Đến khi đủ 200 thì BE sẽ reponse về => Sẽ biết được những header được FE bổ sung là gì.

Có thể thấy rằng giao diện người dùng đang thêm một tiêu đề đặc biệt.



Dùng header này để bypass security controls

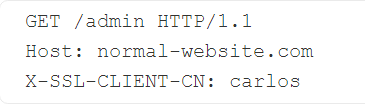


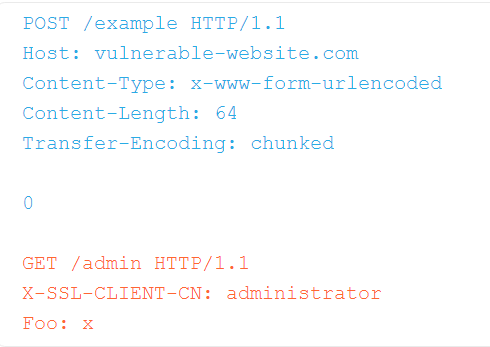
**c) Bypassing client authentication**

Trong TLS handshake, server tự xác thực với client bằng cách cung cấp chứng chỉ. Chứng chỉ này chứa Common Name (CN), mà phải khớp với hostname đã dăng ký của họ. Client sẽ sử dụng nó để xác minh server có hợp pháp hay k.

Phiên bản nâng cấp của điều này là triển khai xác thực lần nhau, tức là client cũng phải gửi chứng chỉ cho server. Trong trường hợp này Common Name thường là username hoặc cái gì đó tương tự. Và FE server thường thêm header chứa Common Name của client trước khi đến BE server. Đương nhiên BE server cũng tin tưởng tuyệt đối.

Nếu chỉ gửi request đơn thuần có header chứa Common Name thì sẽ bị FE server ghi đè ngay -> Smuggle request là sự lựa chọn hoàn hảo để bypass





**d) Capturing other users' requests**

Nếu ứng dụng chứa bất kỳ chức năng nào cho phép lưu trữ và sau đó truy xuất dữ liệu văn bản, thì có thể sử dụng chức năng này để nắm bắt nội dung yêu cầu của những người dùng khác.

Có thể bao gồm session token hoặc dữ liệu nhạy cảm do người dùng gửi. Ví dụ một số chức năng: comment, email, mô tả hồ sơ,…

Để thực hiện cuộc tấn công, gửi một smuggled request gửi dữ liệu đến chức năng lưu trữ, với parameter chứa data để lưu ở vị trí cuối cùng của request.

Ví dụ: Comment trên blog

**POST /post/comment HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Type: application/x-www-form-urlencoded**

**Content-Length: 154**

**Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO**

**csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&comment=My+comment&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net**

Điều gì sẽ xảy ra nếu smuggle một request tương đương với CL quá dài và parameter comment ở cuối cùng

GET / HTTP/1.1

Host: vulnerable-website.com

Transfer-Encoding: chunked

Content-Length: 330

0

POST /post/comment HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 400

Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=

CL = 400 mà body = 144 => đợi 256 => Muốn đủ 256 thì phải lấy từ request sau

POST /post/comment HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 400

Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=GET / HTTP/1.1

Host: vulnerable-website.com

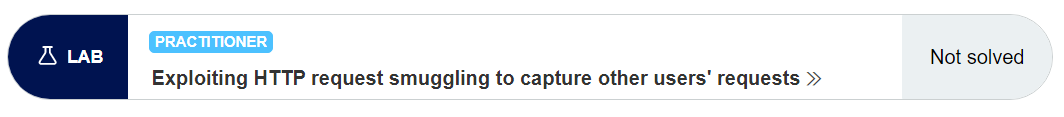
Cookie: session=jJNLJs2RKpbg9EQ7iWrcfzwaTvMw81Rj

...

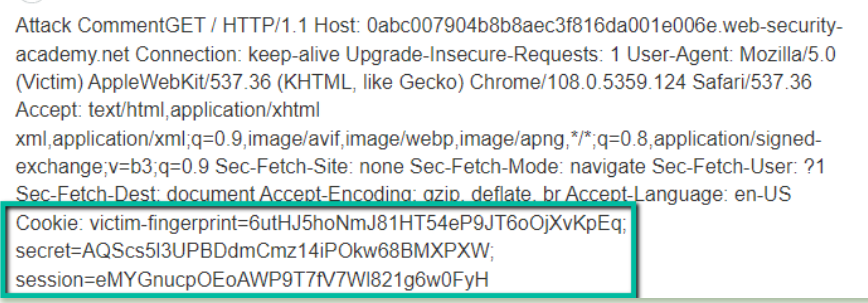
Request của nạn nhân được chứa trong parameter comment, sẽ được đăng dưới dạng nhận xét trên blog => chỉ cần truy cập vào blog tương ứng để đọc

Muốn đọc thêm từ tăng giá trị CL lên. Time out thì lại giảm CL xuống

Hạn chế đối với kỹ thuật này là chỉ thu thập dữ liệu đến khi gặp dấu “&” nghĩa là hết parameter đấy



Các bước làm như lý thuyết. Lấy được cookie

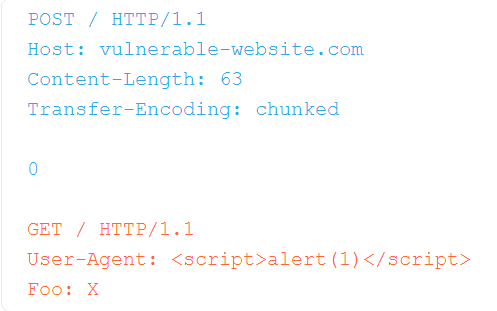


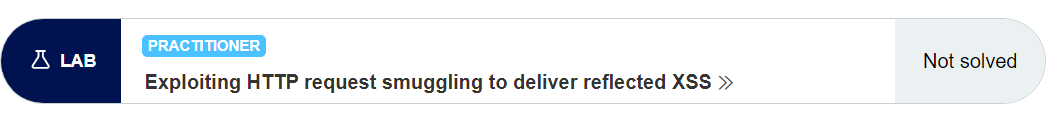
**e) Using HTTP request smuggling to exploit reflected XSS**

Cách tiếp cận này vượt trội hơn so với khai thác reflected XSS thông thường theo hai cách:

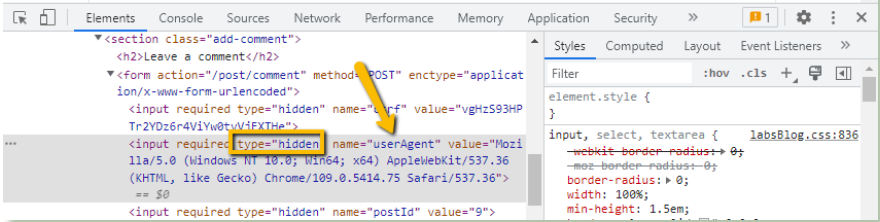
* Không cần user tương tác: gửi 1 phát là request kế của user dính ngay
* Khai thác XSS tại những chỗ mà XSS thông thường không khai thác được( ví dụ là header user-agent)

Ví dụ: Lỗ hổng reflected XSS trong header *User-Agent*. Attack request

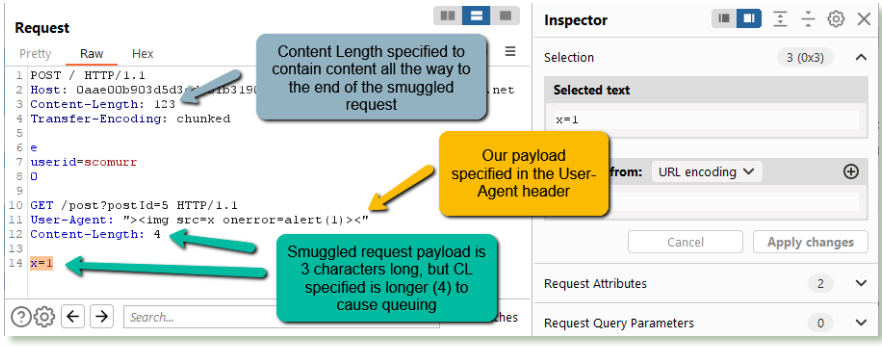




Detect: Sử dụng CL.TE



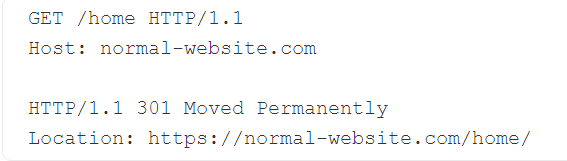
Ứng dụng lấy giá trị User-Agent từ GET request sau đó nhúng nó vào HTML form.



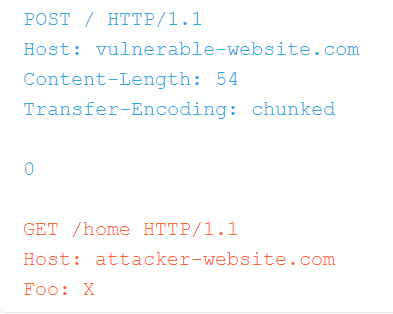
Gửi smuggled request đến postId=5 với giá trị User-Agent gây reflected XSS. Khi một người dùng khác truy cập vào trang chủ thì bị alert().

**f) Using HTTP request smuggling to turn an on-site redirect into an open redirect (biến chuyển hướng tại chỗ thành chuyển hướng mở)**

Nhiều ứng dụng thực hiện on-site redirects từ URL này sang URL khác và đặt hostname từ **Host** header vào URL chuyển hướng



Có thể bị khai thác để chuyển hướng người dùng khác đến một miền bên ngoài

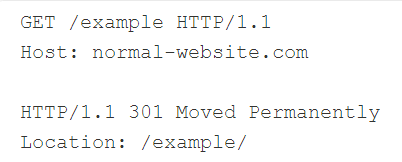


Kích hoạt chuyển hướng đến trang web của kẻ tấn công, ảnh hưởng đến người dùng sau



**g) Turning root-relative redirects into open redirects**

Server-level redirects sử dụng path để tạo root-relative URL cho header **Location**

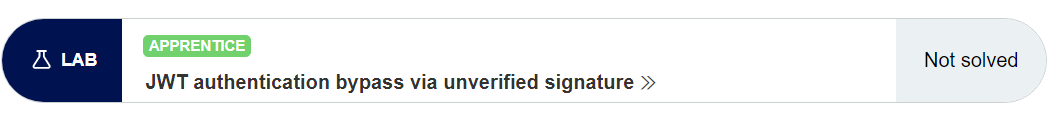


Điều này có khả năng vẫn có thể được sử dụng cho chuyển hướng mở nếu máy chủ cho phép bạn sử dụng URL liên quan đến giao thức trong đường dẫn



------------------------------------------------------------------------------------------------------------------------------------------

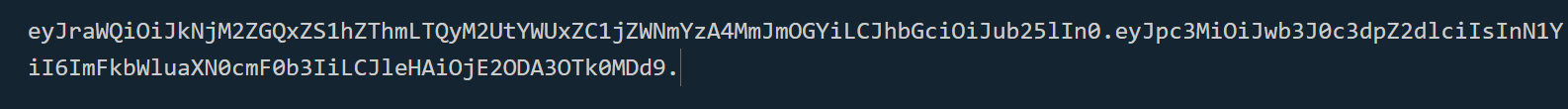
**WEB16: JWT Attacks**

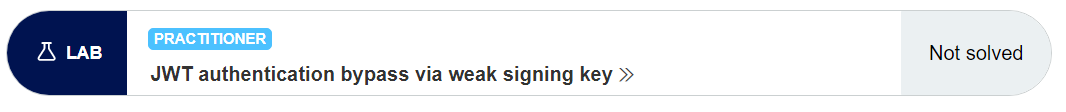


Server không xác thực signature của bất kỳ JWT nào mà nó nhận được => Có thể thay đổi user đã login thành **administrator** để hoàn thành bài lab.

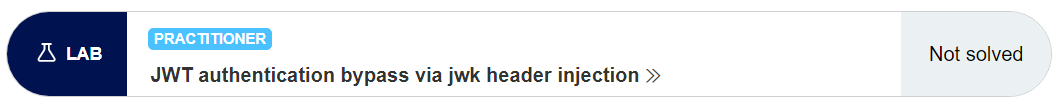


Server cấu hình không an toàn để chấp nhận unsigned JWTSs => Có thể thay đổi giá trị thuật toán **alg** thành **none** thì lúc này sẽ không còn signature trong JWT nữa.





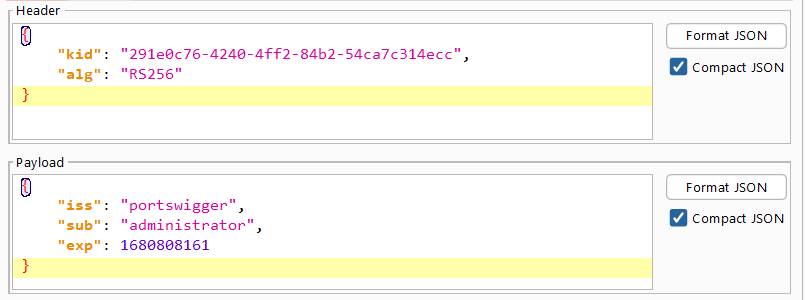
Sử dụng một khóa bí mật cực yếu để sign và verify token. Có thể brute-forced để tìm được (Sử dụng hashcat). Sau khi tìm được khóa bí mật thì sẽ sử dụng khóa này để sign token.



Server hỗ trợ *jwk* parameter trong JWT header. Điều này đôi khi được sử dụng để **nhúng khóa** xác thực chính xác **trực tiếp vào token**. Tuy nhiên, không thể kiểm tra xem **khóa** được cung cấp có đến từ một nguồn tin cậy hay không.

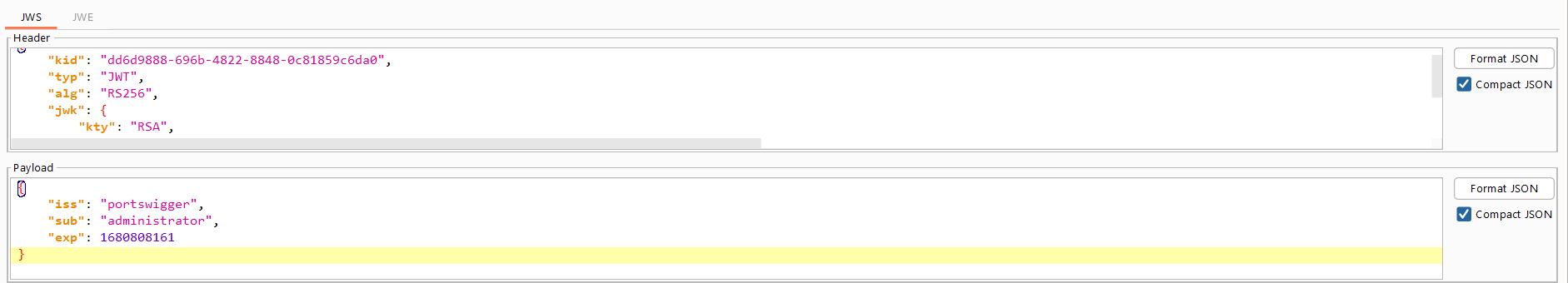
Sử dụng tool **JWT Editor Keys ->** Trong request chọn **Attack** -> **Embedded JWK.**

**TH1: Không sử dụng jwk parameter**



Tạo ra chuỗi JWT một cách bình thường

**TH2: Có sử dụng jwt parameter**

****

Lúc này backend sẽ sử dụng **thuật toán mới được cung cấp** trong token thay vì sử dụng **RS256 => Tạo ra một chuỗi JWT khác.**

Do khi sử dụng **jwt parameter** thì không thể kiểm tra xem **khóa** có đến từ nguồn tin cậy hay không nên khi thay đổi giá trị **sub** thành **administrator** thì không kiểm tra được nên bị by pass.

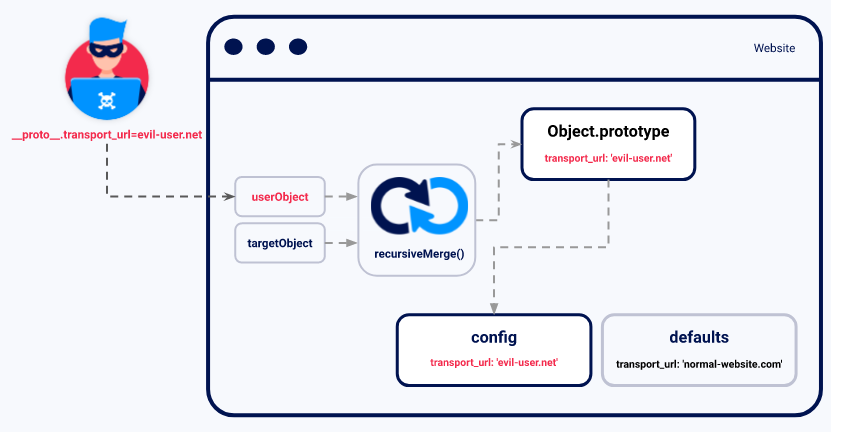
**2 TH SẼ TẠO RA HAI CHUÔI JWT KHÁC NHAU.**

------------------------------------------------------------------------------------------------------------------------------------------

**WEB17: Prototype Pollution**

**1. Prototype Pollution là gì?**

**Prototype Pollution:** Là lỗ hổng JS cho phép kẻ tấn công thêm các thuộc tính (properties) tùy ý vào global object prototypes, mà sau đó có thể được kế thừa bởi các object do người dùng định nghĩa



Prototype pollution cho phép kẻ tấn công kiểm soát các thuộc tính của các đối tượng mà bình thường không truy cập được. Sau đó, ứng dụng xử lý thuộc tính do kẻ tấn công kiểm soát theo cách không an toàn thì có khả năng bị xâu chuỗi với các lỗ hổng khác.

Client-side JS: thường dẫn đến **DOM XSS**

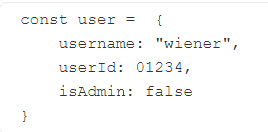
Server-side JS: có thể dẫn đến **RCE**

**2. JavaScript prototypes and inheritance**

JS sử dụng mô hình kế thừa nguyên mẫu (Prototypal inheritance).

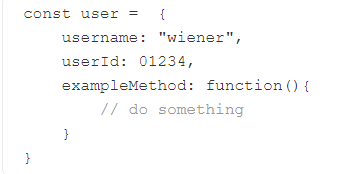
**a) Object trong JS**

Một object JS cơ bản là một tập hợp các cặp *key:value* được gọi là **properties**



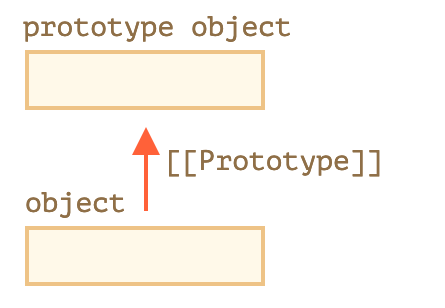


Cũng như data, các thuộc tính có thể chứa các hàm thực thi, gọi là **method**



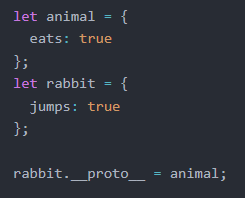
**b) Prototype trong JS là gì?**

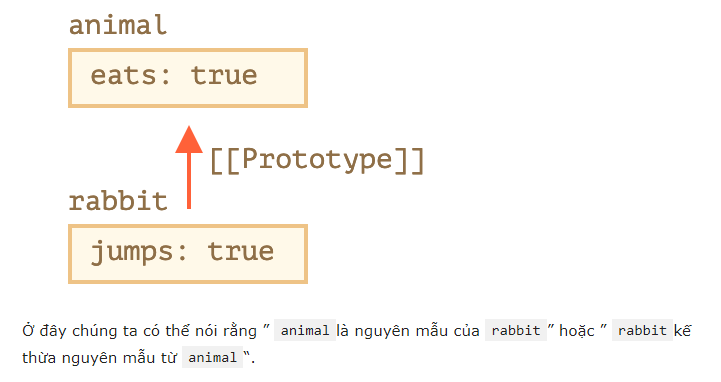
Mọi object trong JS được liên kết với một object khác, được gọi là **prototype** của nó



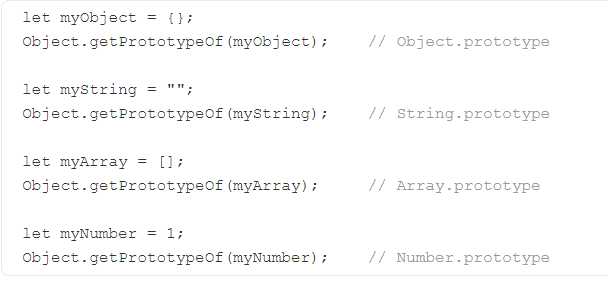
Tất cả các object trong javascript đều có một prototype, và các object này kế thừa các thuộc tính (properties) cũng như phương thức (methods) từ prototype của mình, trừ khi trùng key. Bản thân prototype là một object trong JS, được gọi là prototype object (đối tượng prototype).

Thuộc tính **[[Prototype]]** là nội bộ(private) và ẩn. Cách thiết lập: Sử dụng **\_\_proto\_\_.**

****

****

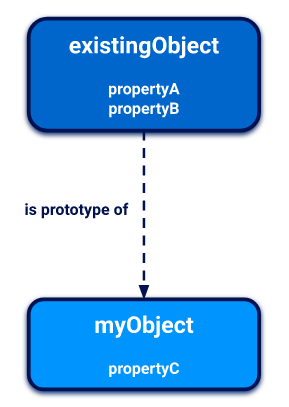
Theo mặc định, JavaScript sẽ tự động gán cho các đối tượng mới một trong các prototype tích hợp sẵn của nó.



Các prototype dựng sẵn cung cấp các thuộc tính và phương thức hữu ích để làm việc với các kiểu dữ liệu cơ bản. Ví dụ: Object *String.prototype* có method *toLowerCase().*

**c) Kế thừa object trong JS hoạt động như thế nào?**

Khi tham chiếu đến một thuộc tính của đối tượng, JS sẽ truy cập thuộc tính này trên chính đối tượng đó. Nếu đối tượng không có thuộc tính phù hợp thì JS sẽ tìm thuộc tính đó trên prototype của đối tượng để thay thế.

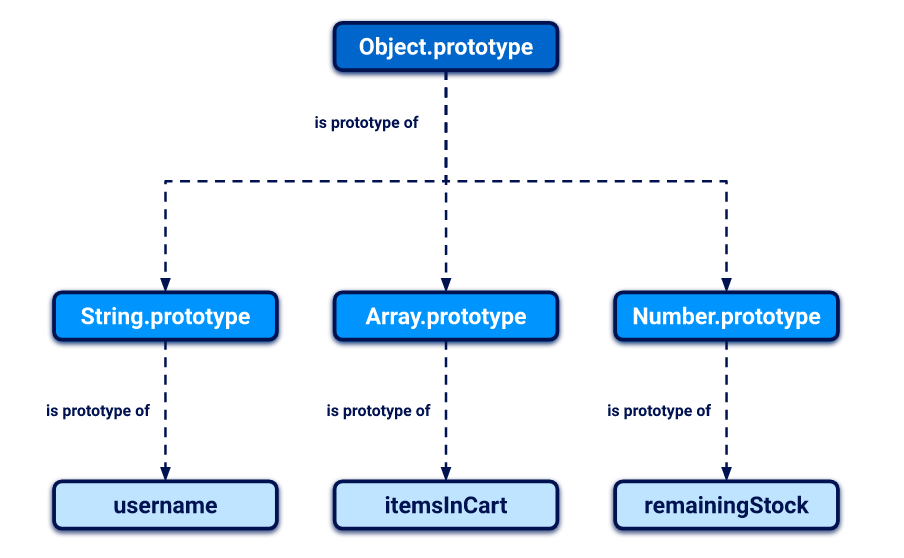


Ví dụ: Cho phép tham chiếu đến *myObject.propertyA*

**d) Prototype chain**

Prototype của một object là một object khác và nó cũng có một protype riêng.

Vì hầu như mọi thứ trong JS đều là đối tượng ngầm, prototype chain cuối cùng sẽ trở lại **Object.prototype** cấp cao nhất, prototype của nó là *null*



Các object không chỉ kế thừa các thuộc tính từ prototype trực tiếp của nó mà còn từ tất cả các object phía trên chúng trong prototype chain. Ví dụ: *username* có quyền truy cập vào các thuộc tính và phương thức của *String.prototype* và *Object.prototype.*

**e) Truy cập prototype của object sử dụng \_\_proto\_\_**

Thuộc tính **\_\_proto\_\_** đóng vai trò vừa là getter và setter cho prototype của object. Có thể sử dụng nó để đọc prototype và các thuộc tính của nó, có thể gán lại chúng nếu cần.

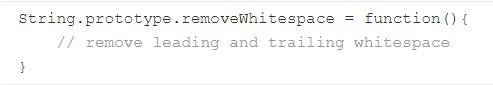
Truy cập \_\_proto\_\_ :





**f) Sửa đổi prototypes**

Tự viết method. Ví dụ: **removeWhitespace()** thay vì sử dụng **trim()** có sẵn



**3. Lỗ hổng prototype pollution phát sinh như thế nào?**

Xảy ra khi hàm JS merge một object chứa các thuộc tính do người dùng kiểm soát vào một object đã tồn tại mà không kiểm tra các keys (trong *key:value*). Cho phép kẻ tấn công tiêm một thuộc tính có key ví dụ như **proto** cùng với các thuộc tính lồng nhau tùy ý.

Merge có thể gán các thuộc tính lồng nhau cho prototype của object thay vì chính object đích.

Thường xảy ra với global Object.prototype được tĩnh hợp sẵn.

Để khai thác thành công prototype pollution, cần một số yêu cầu sau:

* **Prototype pollution source -** Bất kỳ đầu vào nào cho phép đầu độc prototype object bằng các thuộc tính tùy ý
* **Sink -** một hàm JavaScript hoặc phần tử DOM cho phép thực thi mã tùy ý
* **Một tiện ích có thể khai thác -** Bất kỳ thuộc tính nào được chuyển vào sink mà không được lọc

**4.** **Prototype pollution sources**

Là bất kỳ đầu vào nào do người dùng kiểm soát cho phép thêm các thuộc tính tùy ý vào các prototype object

**a) Prototype pollution via URL**



Nếu các khóa và giá trị này sau đó được hợp nhất vào một đối tượng hiện có dưới dạng thuộc tính.



\_\_proto\_\_ đóng vai trò là getter cho prototype. Nếu *targetObject* sử dụng *Object.prototype* mặc định, tất cả các đối tượng trong thời gian chạy sẽ kế thừa *evilProperty*

Kẻ tấn công có thể sử dụng cùng một kỹ thuật để prototype pollution bằng các thuộc tính được sử dụng bởi ứng dụng hoặc bất kỳ thư viện đã nhập nào.

**b) Prototype pollution via JSON input**