**Server Side Template Injection**

**Khái niệm:**

**Server-side template injection là khi 1 kẻ tấn công có thể dùng cú pháp của template để tiêm 1 payload độc hại trong 1 template, sau đó thực thi bên phía máy chủ.**

**Các công cụ template được thiết kế để tạo các trang web bằng việc kết hợp các mẫu cố định với dữ liệu dễ thay đổi. SSTI có thể xảy ra khi đầu vào của người dùng được nối trực tiếp trong 1 template hơn là truyền như 1 dữ liệu. Điều này cho phép kẻ tấn công tiêm các chỉ thị template bất kỳ để mà tận dụng được công cụ template, thường cho phép họ kiểm soát hoàn toàn máy chủ. Các payload của lỗ hổng SSTI được vận chuyển và thực thi ở phía máy chủ làm cho chúng nguy hiểm hơn nhiều so với client side template injection.**

**Ảnh hưởng của lỗ hổng SSTI:**

**Lỗ hổng SSTI có thể khiến các trang web gặp nhiều cuộc tấn công khác nhau tùy thuộc vào công cụ template và cách ứng dụng dùng nó. Tronng một số trường hợp hiếm hoi, những lỗ hổng này không gây ra rủi ro bảo mật thật sự. Tuy nhiên, đa số SSTI gây ra ảnh hưởng nghiêm trọng.**

**Ở mức độ nghiêm trọng cuối cùng của thang đo, kẻ tấn công có thể thực thi mã từ xa, giành quyền kiểm soát toàn máy chủ và thực hiện những cuộc tấn công khác trong cơ sở hạ tầng bên trong.**

**Nếu thực thi mã từ xa không khả thi, kẻ tấn công có thể dùng SSTI như 1 cơ sở cho nhiều cuộc tấn công khác, như giành quyền truy cập vào dữ liệu nhạy cảm trên máy chủ.**

**SSTI sinh ra như thế nào?**

**SSTI sinh ra khi đầu vào của người dùng được nối vào các template thay vì truyền vào như dữ liệu.**

**Những template tĩnh cung cấp placeholders cho dynamic content được kết xuất nhìn chung sẽ không bị dính lỗ hổng SSTI.**

**Ví dụ kinh điển là 1 email chào mọi người bằng các tên của họ được trích xuất từ Twig template:**

**$output = $twig->render("Dear {first\_name},", array("first\_name" => $user.first\_name) );**

**Nó không bị dính lỗ hổng SSTI vì first name của người dùng được truyền vào template như dữ liệu.**

**Tuy nhiên, vì template là các chuỗi đơn giản, các nhà phát triển web thường nối đầu vào của người dùng vào trong các template trước để kết xuất. Cũng là ví dụ trước nhưng lần này người dùng có thể điều chỉnh các phần của email trước khi gửi. Ví dụ, họ có thể chọn tên:**

**$output = $twig->render("Dear " . $\_GET['name']);**

**Trong ví dụ này, thay vì 1 giá trị tĩnh được truyền vào template, 1 phần của template được tạo linh hoạt bằng tham số name. Vì cú pháp của template được xem xét ở phía máy chủ, điều này cho phép kẻ tấn công đặt 1 payload SSTI trong tham số name như sau:**

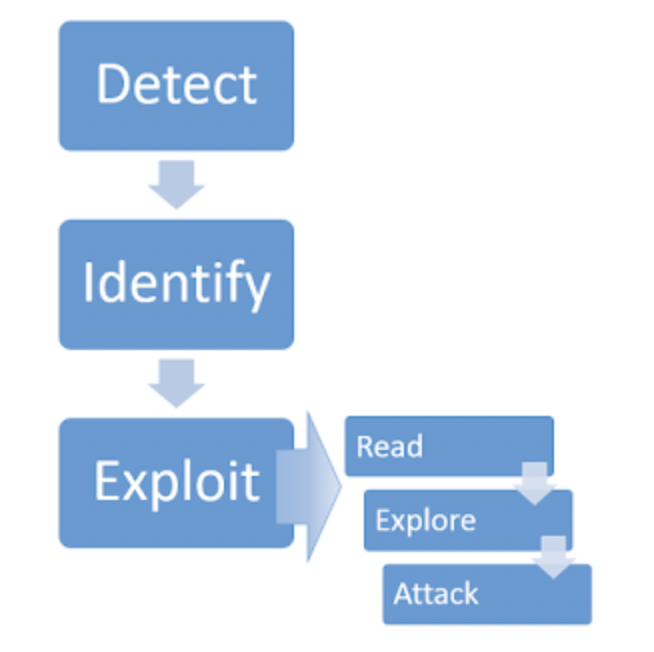
**http://vulnerable-website.com/?name={{bad-stuff-here}}**

**Các lỗ hổng như vậy thường được sinh ra bởi những template được thiết kế kém bởi những người không ham hiểu về bảo mật. Giống như ví dụ trên, bạn có thể thấy nhiều thành phần khác: template chứa đầu vào của người dùng, được nối chuỗi và nhúng trong 1 template.**

**Tuy nhiên, hành vi này đôi khi được thực hiện có chủ ý. Ví dụ, một vài trang web cho phép người dùng có đặc quyền như nhà xuất bản nội dung được chỉnh sửa hoặc gửi những template tùy chỉnh. Điều này rõ ràng đặt ra 1 rủi ro lớn nếu kẻ tấn công có thể thâm nhập 1 tài khoản có đặc quyền như vậy.**

**Xây dựng 1 cuộc tấn công SSTI:**

**Xác định lỗ hổng SSTI và tấn công bao gồm 1 quá trình sau:**

****

**Phát hiện:**

**Nếu bạn có thể tìm thấy lỗ hổng SSTI, nó có thể rất dễ khai thác. Điều này đặc biệt đúng với các môi trường unsandboxed.**

**Giống như các lỗ hổng khác, bước đầu tiên của quá trình khai thác là tìm thấy nó. Bước đầu tiên là thử fuzz template bằng cách tiêm 1 chuỗi các kí tự đặc biệt thường được dùng trong template expressions như ${{<%[%'"}}%\ . Nếu sinh ra exception, điều đó chỉ ra injected template đã được thông dịch bởi máy chủ 🡪SSTI.**

**Các lỗ hổng SSTI xảy ra ở hai bối cảnh khác nhau, mỗi cái yêu cầu 1 phương pháp riêng để phát hiện. Bất kể kết quả của fuzzing là gì, bạn cũng cần xác định bối cảnh để khai thác.**

**Plaintext context**

**Đa số template cho phép bạn nhập nội dung bằng thẻ html hoặc cú pháp của template, sẽ được kết xuất sang html ở phía máy chủ trước khi phản hồi http được gửi đi. Ví dụ, trong Freemarker, dòng render('Hello ' + username) sẽ kết xuất là Hello Carlos.**

**Điều này có thể khai thác bằng XSS, trong thực tế thường nhầm lẫn với lỗ hổng XSS đơn giản. Tuy nhiên, bằng cách đặt 1 phép toán là giá trị của tham số, chúng ta có thể kiểm tra xem đây có phải là entry point cho SSTI hay không.**

**Xem xét 1 template chứa đoạn mã sau:**

**render(‘Hello ‘ + username)**

**Trong quá trình kiểm thử, chúng ta có thể test SSTI bằng cách yêu cầu url sau:**

**http://vulnerable-website.com/?username=${7\*7}**

**Nếu kết quả là Hello 49, điều này cho thấy phép toán đã được thực hiện ở phía máy chủ 🡪SSTI**

**Lưu ý: cú pháp để thực hiện thành công phép toán sẽ dựa vào template đang được dùng.**

**Code context**

**Trong trường hợp khác, lỗ hổng bị phát hiện do đầu vào của người dùng được đặt trong 1 template expression. Điều này có thể ở dạng tên biến do người dùng kiểm soát được đặt bên trong tham số như sau:**

**greeting = getQueryParameter('greeting')**

**engine.render("Hello {{"+greeting+"}}", data)**

**Trên trang web, Url sẽ giống như sau:**

**http://vulnerable-website.com/?greeting=data.username**

**Điều này sẽ được kết xuất thành Hello Carlos.**

**Bối cảnh này rất dễ bị bỏ qua trong quá trình kiểm thử vì nó không dẫn đến 1 lỗ hổng XSS rõ ràng và hoàn toàn khác với 1 hashmap lookup. Một phương pháp để kiểm tra lỗ hổng SSTI trong bối cảnh này là thiết lập tham số không chứa lỗ hổng XSS bằng cách tiêm 1 thẻ html :**

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

**Nếu không có XSS, nó sẽ dẫn đến 1 điểm vào trống (chỉ có Hello mà không có username), thẻ đã được mã hóa hay 1 thông báo lỗi. Bước tiếp theo là kết thúc câu lệnh và cố gắng tiêm 1 thẻ html:**

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

**Nếu điều này dẫn đến lỗi hay đầu ra trống, bạn đã dùng sai template hoặc nếu không có cú pháp nào trông có vẻ hợp lệ, ssti là không khả thi. Thay vào đó, nếu đầu ra được kết xuất đúng, cùng với thẻ html, đây là chìa khóa quan trọng để biết ssti tồn tại:**

**Hello Carlos<tag>**

**Identify**

**Khi bạn đã phát hiện template injection, bước tiếp theo là xác định template engine.**

**Mặc dù có rất nhiều ngôn ngữ template, đa số là sử dụng chung cú pháp để tránh xung đột với các ký tự html. Vì vậy, rất dễ tạo 1 payload để thăm dò template engine đang dùng.**

**Gửi sai cú pháp thường là đủ vì thông báo lỗi sẽ cho bạn biết template engine là gì, thậm chí là phiên bản của nó. Ví dụ biểu thức <%=foobar%> kích hoạt phản hồi sau từ ERB engine (Ruby):**

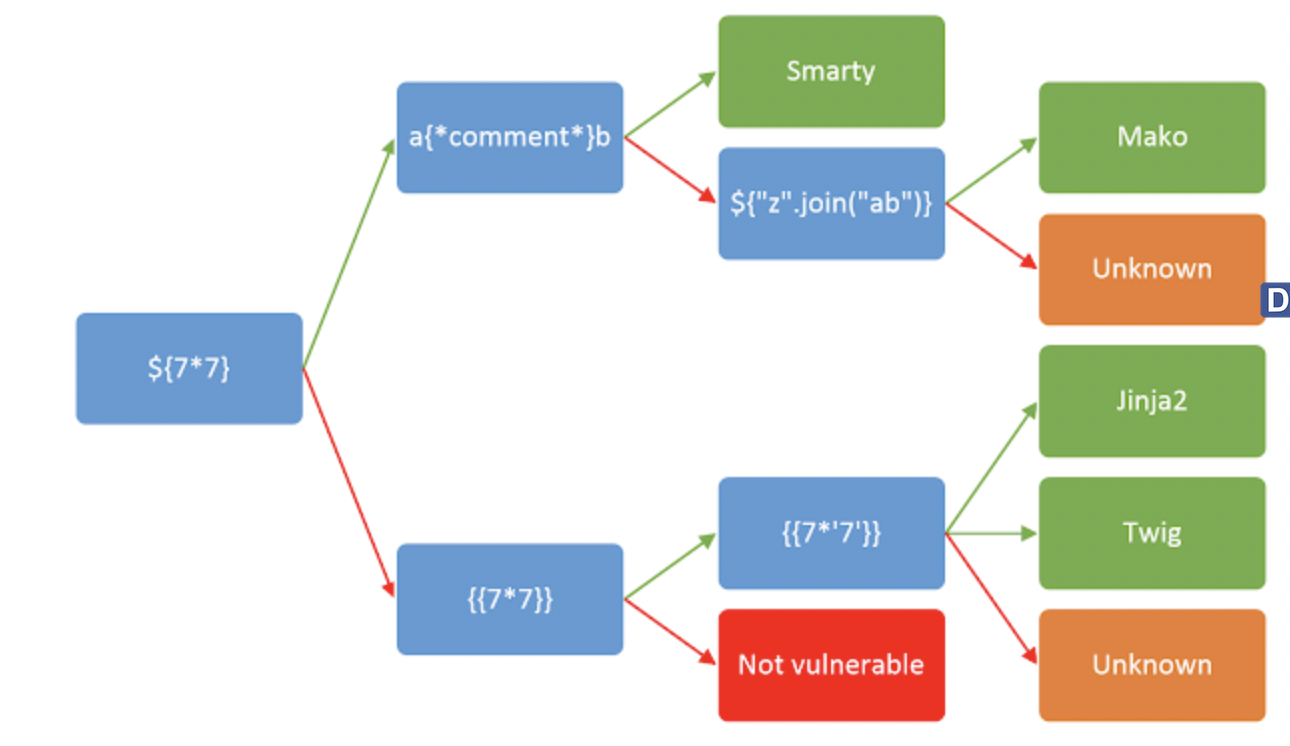
**(erb):1:in `<main>': undefined local variable or method `foobar' for main:Object (NameError)**

**from /usr/lib/ruby/2.5.0/erb.rb:876:in `eval'**

**from /usr/lib/ruby/2.5.0/erb.rb:876:in `result'**

**from -e:4:in `<main>'**

**Nếu không thì, bạn sẽ cần thử những payload của ngôn ngữ khác và học cách chúng được thông dịch bởi template engine. Bằng cách loại bỏ dựa trên cú pháp trông có vẻ hợp lí hay không hợp lí, bạn có thể thu hẹp những lựa chọn mà bạn nghĩ đến. Một cách phổ biến để làm điều này là tiêm 1 phép toán bằng cú pháp của các template engine khác nhau. Sau đó bạn có thể quan sát xem chúng có được đánh giá thành công hay không. Để trợ giúp cho quá trình này, bạn có thể dùng cây như sau:**

****

**Bạn nên nhớ là 1 payload có thể được các template engine khác nhau trả về 1 phản hồi thành công. Ví dụ, payload: {{7\*'7'}} trả về 49 ở Twig và 7777777 ở Jinja2. Do đó, điều quan trọng là không vội kết luận chỉ dựa vào 1 phản hồi thành công.**

**Khai thác lỗ hổng SSTI**

**Khi bạn đã khám phá được lỗ hổng SSTI và xác định được template engine đang dùng, khai thác thành công liên quan đến quá trình sau:**

* **Read**
  + **Template syntax**
  + **Security documentation**
  + **Documented exploits**
* **Explore the environment**
* **Create a custom attack**

**Read**

**Nếu bạn không biết template engine đang ở trong, đọc tài liệu của nó là bước đầu tiên. Trong khi điều này có thể không phải là điều thú vị nhất để bạn mất thời gian, nhưng không được đánh giá thấp sự hữu ích của tài liệu mang lại.**

**Học cú pháp template cơ bản**

**Học cú pháp cơ bản rất quan trọng, cùng với key functions và xử lí các biến. Ngay cả những việc đơn giản như học cách nhúng đoạn mã ban đầu trong template đôi khi cũng có thể nhanh chóng dẫn đến việc khai thác. Ví dụ, khi bạn biết template engine Mako của Python được dùng, thực thi code từ xa có thể đơn giản như sau:**

**<%**

**import os**

**x=os.popen('id').read()**

**%>**

**${x}**

**Trong môi trường unsandbox, thực thi code từ xa và dùng nó để đọc, chỉnh sửa và xóa tệp bất kì cũng đơn giản với đa số template engines.**

**Đọc những thực thi bảo mật**

**Ngoài việc cung cấp các nguyên tắc cơ bản về cách tạo và sử dụng template, tài liệu cũng cung cấp thêm mục bảo mật. Tên của mục này sẽ khác nhau nhưng nó thường sẽ phác thảo tất cả những điều nguy hiểm mà mọi người nên tránh làm với template. Đây là 1 mã nguồn vô giá, 1 cheatsheet về những hành vi bạn nên tìm kiếm khi kiểm thử và cách để khai thác chúng.**

**Thậm chí nếu không có mục bảo mật, nếu 1 đối tượng hay hàm được dựng sẵn có thể gây ra rủi ro bảo mật, luôn có những cảnh báo trong tài liệu. Cảnh báo có thể không cung cấp nhiều thông tin nhưng ít nhất giúp chúng ta mở rộng thêm phạm vi điều tra.**

**Ví dụ, trong ERB, tài liệu tiết lộ rằng bạn có thể liệt kê tất cả thư mục và sau đó đọc tập tin bất kì như sau:**

**<%= Dir.entries('/') %>**

**<%= File.open('/example/arbitrary-file').read %>**

**Tìm kiếm những khai thác có sẵn**

**Một khía cạnh quan trọng khác của việc khai thác lỗ hổng SSTI là tìm kiếm thêm những mã nguồn online. Khi bạn có thể xác định được template engine đang dùng, bạn nên duyệt web để tìm kiếm những lỗ hổng đã được khám phá. Vì một số template được dùng khá phổ biến nên dễ tìm thấy những khai thác đã có sẵn mà bạn có thể điều chỉnh để khai thác trên chính trang web của mình.**

**Khám phá**

**Đến đây, bạn có thể phát hiện 1 cách khai thác khả thi bằng tài liệu. Nếu không, bước tiếp theo là khám phá môi trường và tìm các đối tượng bạn có thể truy cập.**

**Rất nhiều template engines dùng đối tượng self hay environment: giống như 1 namespace chứa tất cả đối tượng, phương thức và thuộc tính được hỗ trợ bởi template engine. Nếu đối tượng đó tồn tại, bạn có thể dùng nó để tạo 1 danh sách các đối tượng trong cùng phạm vi. Ví dụ, trong template của Java, bạn có thể liệt kê tất cả đối tượng trong môi trường bằng lệnh sau:**

**${T(java.lang.System).getenv()}**

**Đây là nền tảng để tạo 1 danh sách ngắn các đối tượng và phương thức nguy hiểm để điều tra thêm.**

**Các đối tượng được cung cấp bởi người dùng**

**Điều quan trọng phải nhớ là trang web sẽ chứa các đối tượng được dựng sẵn bởi template và các đối tượng được cung cấp bởi nhà phát triển web. Bạn nên đặc biệt chú ý đến những đối tượng không chuẩn này vì chúng có thể chứa thông tin nhạy cảm hay phương thức khai thác. Vì các đối tượng có thể rất khác nhau giữa các template trong cùng 1 trang web, bạn có thể cần nghiên cứu hành vi của đối tượng trong bối cảnh của mỗi template trước khi tìm cách khai thác nó.**

**Trong khi SSTI có thể dẫn đến thực thi code từ xa và giành quyền kiểm soát toàn máy chủ, trong thực tế không dễ làm được điều đó. Tuy nhiên, vì bạn đã loại bỏ khả năng thực thi code, không có nghĩa là không có 1 loại khai thác khác. Bạn có thể tận dụng lỗ hổng SSTI cho những khai thác nghiêm trọng hơn như directory traversal để truy cập được dữ liệu nhạy cảm.**