**SQL injection**

**SQL injection là gì? (\*)**

- Là lỗ hổng bảo mật cho phép kẻ tấn công can thiệp vào các truy vấn của ứng dụng với CSDL. Điều này cho phép kẻ tấn công xem dữ liệu mà thường không thể truy xuất. Hoặc có thể sửa đổi hoặc xóa dữ liệu.

- Trong 1 số trường hợp, kẻ tấn công có thể leo thang tấn công SQL để xâm phạm máy chủ cơ sở hoặc cơ sở hạ tầng phụ trợ khác hoặc có thể thực hiện tấn công DOS.

**Tác động của 1 cuộc tấn công SQL injection? (\*)**

- Có thể dẫn đến truy cập trái phép vào dữ liệu nhạy cảm:

* + Mật khẩu
  + Chi tiết thẻ tín dụng
  + Thông tin cá nhân của user

- Trong 1 số trường hợp, kẻ tấn công có thể có được 1 cửa hậu liên tục vào hệ thống của 1 tổ chức, dẫn đến xâm phạm lâu dài.

**Cách phát hiện lỗ hổng SQL injection (\*)**

- Có thể thực hiện thủ công bằng sử dụng 1 tập hợp các bài kiểm tra có hệ thống với mọi điểm vào trong ứng dụng:

* + Ký tự dấu nháy đơn 'và tìm kiếm lỗi hoặc các bất thường khác.
  + Một số cú pháp SQL cụ thể đánh giá theo giá trị cơ sở (gốc) của điểm vào và một giá trị khác, đồng thời tìm kiếm sự khác biệt có hệ thống trong phản hồi của ứng dụng.
  + Các điều kiện Boolean như OR 1=1và OR 1=2, và tìm kiếm sự khác biệt trong phản hồi của ứng dụng.
  + Tải trọng được thiết kế để kích hoạt độ trễ thời gian khi được thực hiện trong truy vấn SQL và tìm kiếm sự khác biệt về thời gian phản hồi.
  + Tải trọng OAST được thiết kế để kích hoạt tương tác mạng ngoài băng tần khi được thực hiện trong truy vấn SQL và giám sát mọi tương tác phát sinh.

**Tiêm SQL vào các phần khác nhau của truy vấn (\*)**

- Hầu hết các lỗ hổng xảy ra trong mệnh đề WHERE của truy vấn SELECT. Tuy nhiên, lỗ hổng có thể xảy ra ở bất kỳ vị trí nào trong truy vấn và trong các loại truy vấn , ví dụ:

* Trong các câu lệnh UPDATE, trong các giá trị được cập nhật hoặc mệnh đề WHERE.
* Trong các câu lệnh INSERT , bên trong các giá trị được chèn vào.
* Trong các câu lệnh SELECT, bên trong tên bảng hoặc cột.
* Trong các câu lệnh SELECT, trong mệnh đề ORDER BY.

**Ví dụ về tiêm SQL (\*)**

- Một số ví dụ phổ biến về SQL injection:

* Truy xuất dữ liệu ẩn , nơi bạn có thể sửa đổi truy vấn SQL để trả về kết quả bổ sung.
* Phá hoại logic ứng dụng , nơi bạn có thể thay đổi truy vấn để can thiệp vào logic của ứng dụng.
* Tấn công UNION , nơi bạn có thể lấy dữ liệu từ các bảng cơ sở dữ liệu khác nhau.
* Tiêm SQL mù , trong đó kết quả của truy vấn bạn kiểm soát không được trả về trong phản hồi của ứng dụng.

**Lấy dữ liệu ẩn (\*)**

- Hãy tưởng tượng một ứng dụng mua sắm hiển thị sản phẩm theo các danh mục khác nhau. Khi người dùng nhấp vào danh mục Quà tặng , trình duyệt của họ yêu cầu URL:

<https://insecure-website.com/products?category=Gifts>

- Điều này khiến ứng dụng thực hiện truy vấn SQL để lấy thông tin chi tiết về các sản phẩm có liên quan từ cơ sở dữ liệu:

SELECT \* FROM products WHERE category = 'Gifts' AND released = 1

Truy vấn SQL này yêu cầu cơ sở dữ liệu trả về:

tất cả các chi tiết ( \*)

từ productsbảng

nơi categorylàGifts

và releasedlà 1.

Hạn chế này released = 1được sử dụng để ẩn các sản phẩm chưa được phát hành. Chúng ta có thể cho rằng đối với các sản phẩm chưa được phát hành, released = 0.

Ứng dụng không triển khai bất kỳ biện pháp phòng thủ nào chống lại các cuộc tấn công SQL injection. Điều này có nghĩa là kẻ tấn công có thể xây dựng các cuộc tấn công sau, ví dụ:

<https://insecure-website.com/products?category=Gifts'-->

Điều này dẫn đến truy vấn SQL:

SELECT \* FROM products WHERE category = 'Gifts'--' AND released = 1

Quan trọng, lưu ý rằng đó --là một chỉ báo chú thích trong SQL. Điều này có nghĩa là phần còn lại của truy vấn được hiểu là một chú thích, về cơ bản là xóa nó. Trong ví dụ này, điều này có nghĩa là truy vấn không còn bao gồm AND released = 1. Kết quả là, tất cả các sản phẩm đều được hiển thị, bao gồm cả những sản phẩm chưa được phát hành.

Bạn có thể sử dụng một cuộc tấn công tương tự để khiến ứng dụng hiển thị tất cả các sản phẩm trong bất kỳ danh mục nào, bao gồm cả những danh mục mà ứng dụng không biết:

<https://insecure-website.com/products?category=Gifts'+OR+1=1-->

Điều này dẫn đến truy vấn SQL:

SELECT \* FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1

Truy vấn đã sửa đổi trả về tất cả các mục có giá trị là category, Giftshoặc 1 bằng 1. Như 1=1 luôn đúng, truy vấn trả về tất cả các mục.

**Phá vở logic ứng dụng (\*)**

- Hãy tưởng tượng một ứng dụng cho phép người dùng đăng nhập bằng tên người dùng và mật khẩu. Nếu người dùng gửi tên người dùng wienervà mật khẩu bluecheese, ứng dụng sẽ kiểm tra thông tin xác thực bằng cách thực hiện truy vấn SQL sau:

SELECT \* FROM users WHERE username = 'wiener' AND password = 'bluecheese'

- Nếu truy vấn trả về thông tin chi tiết của người dùng thì đăng nhập thành công. Nếu không, đăng nhập sẽ bị từ chối.

- Trong trường hợp này, kẻ tấn công có thể đăng nhập với tư cách là bất kỳ người dùng nào mà không cần mật khẩu. Họ có thể thực hiện việc này bằng cách sử dụng chuỗi chú thích SQL --để xóa kiểm tra mật khẩu khỏi WHERE mệnh đề của truy vấn. Ví dụ, gửi tên người dùng administrator'-- và mật khẩu trống sẽ dẫn đến truy vấn sau:

SELECT \* FROM users WHERE username = 'administrator'--' AND password = ''

usernameTruy vấn này trả về tên người dùng administratorvà ghi nhận thành công kẻ tấn công vào tài khoản người dùng đó.

**Lấy dữ liệu từ các bảng cơ sở dữ liệu khác (\*)**

- Trong trường hợp ứng dụng phản hồi với kết quả của truy vấn SQL, kẻ tấn công có thể sử dụng lỗ hổng SQL injection để lấy dữ liệu từ các bảng khác trong cơ sở dữ liệu. Bạn có thể sử dụng UNIONtừ khóa để thực hiện truy vấn bổ sung SELECTvà thêm kết quả vào truy vấn gốc.

Ví dụ, nếu một ứng dụng thực hiện truy vấn sau có chứa thông tin đầu vào của người dùng Gifts:

SELECT name, description FROM products WHERE category = 'Gifts'

Kẻ tấn công có thể gửi thông tin đầu vào:

' UNION SELECT username, password FROM users—

Điều này khiến ứng dụng trả về tất cả tên người dùng và mật khẩu cùng với tên và mô tả sản phẩm.

**Lỗ hổng tiêm SQL mù (\*)**

- Nhiều trường hợp tiêm SQL là lỗ hổng ẩn. Điều này có nghĩa là ứng dụng không trả về kết quả của truy vấn SQL hoặc thông tin chi tiết về bất kỳ lỗi cơ sở dữ liệu nào trong phản hồi của ứng dụng. Lỗ hổng ẩn vẫn có thể bị khai thác để truy cập dữ liệu trái phép, nhưng các kỹ thuật liên quan thường phức tạp hơn và khó thực hiện hơn.

- Các kỹ thuật sau đây có thể được sử dụng để khai thác lỗ hổng tiêm SQL mù, tùy thuộc vào bản chất của lỗ hổng và cơ sở dữ liệu liên quan:

* Bạn có thể thay đổi logic của truy vấn để kích hoạt sự khác biệt có thể phát hiện được trong phản hồi của ứng dụng tùy thuộc vào tính đúng đắn của một điều kiện duy nhất. Điều này có thể bao gồm việc đưa một điều kiện mới vào một số logic Boolean hoặc kích hoạt có điều kiện một lỗi như chia cho số không.
* Bạn có thể kích hoạt có điều kiện thời gian trễ trong quá trình xử lý truy vấn. Điều này cho phép bạn suy ra sự thật của điều kiện dựa trên thời gian ứng dụng phản hồi.
* Bạn có thể kích hoạt tương tác mạng ngoài băng tần, sử dụng các kỹ thuật OAST. Kỹ thuật này cực kỳ mạnh mẽ và hoạt động trong những tình huống mà các kỹ thuật khác không làm được. Thông thường, bạn có thể trực tiếp trích xuất dữ liệu qua kênh ngoài băng tần. Ví dụ, bạn có thể đặt dữ liệu vào tra cứu DNS cho một tên miền mà bạn kiểm soát.

**Tiêm SQL bậc 2 (\*)**

- Tấn công SQL cấp một xảy ra khi ứng dụng xử lý dữ liệu đầu vào của người dùng từ yêu cầu HTTP và kết hợp dữ liệu đầu vào đó vào truy vấn SQL theo cách không an toàn.

- Tiêm SQL bậc hai xảy ra khi ứng dụng lấy dữ liệu đầu vào của người dùng từ một yêu cầu HTTP và lưu trữ để sử dụng trong tương lai. Điều này thường được thực hiện bằng cách đưa dữ liệu đầu vào vào cơ sở dữ liệu, nhưng không có lỗ hổng nào xảy ra tại điểm dữ liệu được lưu trữ. Sau đó, khi xử lý một yêu cầu HTTP khác, ứng dụng sẽ truy xuất dữ liệu đã lưu trữ và kết hợp dữ liệu đó vào truy vấn SQL theo cách không an toàn. Vì lý do này, tiêm SQL bậc hai còn được gọi là tiêm SQL được lưu trữ.

- Tiêm SQL bậc hai thường xảy ra trong các tình huống mà các nhà phát triển biết về lỗ hổng tiêm SQL và do đó xử lý an toàn việc đặt đầu vào ban đầu vào cơ sở dữ liệu. Khi dữ liệu được xử lý sau đó, nó được coi là an toàn, vì trước đó nó đã được đặt vào cơ sở dữ liệu một cách an toàn. Tại thời điểm này, dữ liệu được xử lý theo cách không an toàn, vì nhà phát triển đã nhầm tưởng rằng nó đáng tin cậy.

**Kiểm tra CSDL (\*)**

- Một số tính năng cốt lõi của ngôn ngữ SQL được triển khai theo cùng một cách trên các nền tảng cơ sở dữ liệu phổ biến và rất nhiều cách phát hiện và khai thác lỗ hổng SQL injection hoạt động giống hệt nhau trên các loại cơ sở dữ liệu khác nhau.

- Tuy nhiên, cũng có nhiều điểm khác biệt giữa các cơ sở dữ liệu phổ biến. Điều này có nghĩa là một số kỹ thuật phát hiện và khai thác SQL injection hoạt động khác nhau trên các nền tảng khác nhau. Ví dụ:

* + Cú pháp nối chuỗi.
  + Bình luận.
  + Truy vấn theo đợt (hoặc xếp chồng).
  + API dành riêng cho nền tảng.
  + Thông báo lỗi.

- Sau khi bạn xác định được lỗ hổng SQL injection, việc thu thập thông tin về cơ sở dữ liệu thường hữu ích. Thông tin này có thể giúp bạn khai thác lỗ hổng.

Bạn có thể truy vấn thông tin chi tiết về phiên bản cho cơ sở dữ liệu. Các phương pháp khác nhau có hiệu quả với các loại cơ sở dữ liệu khác nhau. Điều này có nghĩa là nếu bạn tìm thấy một phương pháp cụ thể có hiệu quả, bạn có thể suy ra loại cơ sở dữ liệu. Ví dụ, trên Oracle, bạn có thể thực hiện:

SELECT \* FROM v$version

Bạn cũng có thể xác định những bảng cơ sở dữ liệu nào hiện có và các cột chúng chứa. Ví dụ, trên hầu hết các cơ sở dữ liệu, bạn có thể thực hiện truy vấn sau để liệt kê các bảng:

SELECT \* FROM information\_schema.tables

**Tiêm SQL trong các bối cảnh khác nhau (\*)**

Trong các bài thực hành trước, bạn đã sử dụng chuỗi truy vấn để chèn tải trọng SQL độc hại của mình. Tuy nhiên, bạn có thể thực hiện các cuộc tấn công tiêm SQL bằng bất kỳ đầu vào nào có thể kiểm soát được được ứng dụng xử lý dưới dạng truy vấn SQL. Ví dụ, một số trang web lấy đầu vào ở định dạng JSON hoặc XML và sử dụng định dạng này để truy vấn cơ sở dữ liệu.

Các định dạng khác nhau này có thể cung cấp cho bạn nhiều cách khác nhau để che giấu các cuộc tấn công vốn bị chặn do WAF và các cơ chế phòng thủ khác. Các triển khai yếu thường tìm kiếm các từ khóa tiêm SQL phổ biến trong yêu cầu, do đó bạn có thể bỏ qua các bộ lọc này bằng cách mã hóa hoặc thoát các ký tự trong các từ khóa bị cấm. Ví dụ: tiêm SQL dựa trên XML sau đây sử dụng chuỗi thoát XML để mã hóa Ský tự trong SELECT:

<stockCheck>

<productId>123</productId>

<storeId>999 &#x53;ELECT \* FROM information\_schema.tables</storeId>

</stockCheck>

Nội dung này sẽ được giải mã ở phía máy chủ trước khi chuyển đến trình thông dịch SQL.

**Làm thế nào để ngăn chặn SQL injection (\*)**

- Bạn có thể ngăn chặn hầu hết các trường hợp tiêm SQL bằng cách sử dụng các truy vấn tham số hóa thay vì nối chuỗi trong truy vấn. Các truy vấn tham số hóa này cũng được gọi là "câu lệnh đã chuẩn bị".

Đoạn mã sau đây dễ bị tấn công SQL injection vì dữ liệu đầu vào của người dùng được nối trực tiếp vào truy vấn:

String query = "SELECT \* FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);

- Bạn có thể viết lại mã này theo cách ngăn chặn dữ liệu đầu vào của người dùng can thiệp vào cấu trúc truy vấn:

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();

- Bạn có thể sử dụng các truy vấn tham số hóa cho bất kỳ tình huống nào mà dữ liệu đầu vào không đáng tin cậy xuất hiện dưới dạng dữ liệu trong truy vấn, bao gồm WHEREmệnh đề và giá trị trong câu lệnh INSERTor UPDATE. Chúng không thể được sử dụng để xử lý dữ liệu đầu vào không đáng tin cậy trong các phần khác của truy vấn, chẳng hạn như tên bảng hoặc tên cột hoặc ORDER BYmệnh đề. Chức năng ứng dụng đặt dữ liệu không đáng tin cậy vào các phần này của truy vấn cần áp dụng một cách tiếp cận khác, chẳng hạn như:

* Danh sách trắng các giá trị đầu vào được phép.
* Sử dụng logic khác nhau để đưa ra hành vi cần thiết.

Để truy vấn tham số hóa có hiệu quả trong việc ngăn chặn SQL injection, chuỗi được sử dụng trong truy vấn phải luôn là hằng số được mã hóa cứng. Nó không bao giờ được chứa bất kỳ dữ liệu biến nào từ bất kỳ nguồn nào. Không nên cố gắng quyết định từng trường hợp xem một mục dữ liệu có đáng tin cậy hay không và tiếp tục sử dụng phép nối chuỗi trong truy vấn cho các trường hợp được coi là an toàn. Rất dễ mắc lỗi về nguồn gốc có thể có của dữ liệu hoặc thay đổi mã khác để làm hỏng dữ liệu đáng tin cậy.