**Insecure deserialization**

**Khái niệm**

Serialization là quá trình chuyển đổi cấu trúc dữ liệu phức tạp như các đối tượng và các trường của chúng sang 1 định dạng “phẳng hơn” có thể được gửi và nhận như 1 luồng byte tuần tự. Serializing dữ liệu giúp:

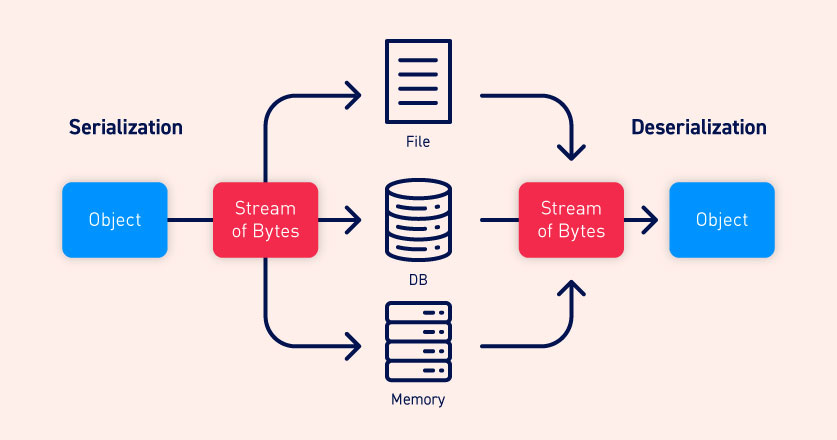
Ghi dữ liệu phức tạp vào bộ nhớ, tệp hay cơ sở dữ liệu.

Gửi dữ liệu phức tạp qua mạng giữa các thành phần khác nhau của 1 ứng dụng hay gọi API.

Quan trọng hơn, khi serialize 1 đối tượng, trạng thái của nó vẫn được duy trì. Nói cách khác, thuộc tính của đối tượng được giữ nguyên cùng với các giá trị được gán cho chúng.

**Serialization vs deserialization**

Deserialization là quá trình khôi phục luồng byte này thành 1 bản sao đầy đủ chức năng của đối tượng ban đầu. Logic của trang web sau đó có thể tương tác với đối tượng đã được deserialize này.



Rất nhiều ngôn ngữ lập trình hỗ trợ serialization. Một vài ngôn ngữ serialize đối tượng thành định dạng nhị phân, trong khi số khác sử dụng các định dạng chuỗi khác nhau với mức độ dễ đọc khác nhau với con người. Lưu ý là tất cả thuộc tính của đối tượng ban đầu được lưu trữ trong luồng dữ liệu đã được serialize, bao gồm miền private. Để ngăn chặn 1 miền không bị serialize, nó phải được gán “transient” trong khai báo class.

Hãy cẩn thận khi làm việc với nhiều ngôn ngữ lập trình khác nhau, serialization có thể được gọi là marshalling(Ruby) hay pickling (Python).

**Insecure deserialization**

Insecure deserialization là khi dữ liệu do người dùng kiểm soát được deserialize bởi 1 trang web. Điều này cho phép kẻ tấn công tận dụng các đối tượng đã được serialize để mà gửi đi dữ liệu độc hại trong code của ứng dụng.

Thậm chí có thể thay thế 1 đối tượng đã serialize với 1 đối tượng của 1 lớp khác. Nguy hiểm hơn, các đối tượng của 1 lớp bất kì có thể được trang web deserialize và khởi tạo, bất kể lớp nào được mong đợi. Với lí do đó, insecure deserialization thường biết đến là lỗ hổng object injection.

Một đối tượng của 1 lớp không được mong đợi có thể gây ra ngoại lệ. Rất nhiều cuộc tấn công deserialization được hoàn thành trước khi quá trình deserialization xảy ra. Điều này có nghĩa là chính quá trình deserialization có thể khởi tạo 1 cuộc tấn công, thậm chí chức năng của trang web không trực tiếp tương tác với đối tượng độc hại. Với lí do trên, các trang web có logic tương đối mạnh cũng có thể bị dính lỗ hổng với kĩ thuật này.

**Lỗ hổng insecure deserialization sinh ra như thế nào?**

Insecure deserialization sinh ra vì có sự thiếu hiểu biết về mức độ nguy hiểm của deserializing dữ liệu do người dùng kiểm soát. Tốt nhất là đầu vào của người dùng không nên được deserialize hết.

Tuy nhiên, nhà phát triển trang web nghĩ họ an toàn vì họ thực thi một vài filter với dữ liệu đã được deserialize. Cách tiếp cận này không hiệu quả vì không thể thực thi xác thực hay sàng lọc để tính đến mọi tình huống. Những bộ lọc này về cơ bản cũng bị lỗi vì chúng dựa vào việc kiểm tra dữ liệu sau khi nó đã được deserialize, trong rất nhiều trường hợp sẽ là quá trễ để ngăn chặn cuộc tấn công.

Lỗ hổng này sinh ra vì các đối tượng đã được deserialize được giả sử là rất đáng tin cậy. Đặc biệt khi dùng các ngôn ngữ với định dạng serialize là nhị phân, các nhà phát triển có thể nghĩ rằng người dùng không thể đọc hay thao tác với dữ liệu một cách hiệu quả. Tuy nhiên, trong khi nó có thể đòi hỏi nhiều nỗ lực hơn thì kẻ tấn công cũng có thể khai thác các đối tượng đã serialize ở dạng nhị phân như khai thác các định dạng dựa trên chuỗi.

Các cuộc tấn công dựa trên deserialization cũng có thể thực hiện được do rất nhiều dependencies còn tồn tại trong trang web hiện đại. Một trang web có thể thực thi rất nhiều thư viện, mà mỗi cái có những dependencies của chúng. Điều này tạo ra một nhóm lớn các lớp và phương thức khó quản lí một cách an toàn. Vì kẻ tấn công có thể tạo ra nhiều exploit của bất kì lớp nào, rất khó dự đoán các phương thức có thể được gọi trong dữ liệu độc hại. Điều này đặc biệt đúng khi kẻ tấn công có thể kết hợp 1 chuỗi dài các phương thức không mong đợi, gửi dữ liệu qua 1 sink mà không liên quan đến source ban đầu. Do đó, gần như không thể lường trước được luồng dữ liệu độc hại và bịt mọi lỗ hổng tiềm ẩn.

Tóm lại, không thể deserialize dữ liệu không đáng tin cậy một cách an toàn.

**Ảnh hưởng của insecure deserialization**

Ảnh hưởng của insecure deserialization có thể rất nghiêm trọng vì nó cung cấp 1 điểm vào để tăng thêm phạm vi tấn công. Nó cho phép 1 kẻ tấn công sử dụng lại mã ứng dụng hiện có theo cách độc hại, dẫn đến vô số lỗ hổng khác, thường là RCE.

Thậm chí nếu RCE không khả thi, insecure deserialization có thể dẫn đến leo thang đặc quyền, truy cập tệp bất kì và các cuộc tấn công DOS.

**Khai thác các lỗ hổng insecure deserialization**

**Làm sao để xác định insecure deserialization?**

Việc xác định insecure deserialization khá đơn giản bất kể bạn đang kiểm thử blackbox hay whitebox.

Trong quá trình kiểm thử, bạn nên xem tất cả dữ liệu đang gửi trong trang web và cố gắng xác định bất cứ thứ gì trông giống dữ liệu serialized. Dữ liệu đã serialize có thể được xác định dễ dàng nếu bạn biết định dạng mà các ngôn ngữ dùng.

**Định dạng PHP serialize**

PHP dùng 1 định dạng chuỗi có thể đọc được với các chữ cái đại diện cho kiểu dữ liệu và số đại diện cho độ dài của mỗi điểm vào. Ví dụ xem xét 1 đối tượng User với các thuộc tính:

$user->name = "carlos";

$user->isLoggedIn = true;

Khi serialize, đối tượng này có thể trông giống như:

O:4:"User":2:{s:4:"name";s:6:"carlos"; s:10:"isLoggedIn";b:1;}

Nó có thể được thông dịch như sau:

* O:4:"User" - An object with the 4-character class name "User"
* 2 - the object has 2 attributes
* s:4:"name" - The key of the first attribute is the 4-character string "name"
* s:6:"carlos" - The value of the first attribute is the 6-character string "carlos"
* s:10:"isLoggedIn" - The key of the second attribute is the 10-character string "isLoggedIn"
* b:1 - The value of the second attribute is the boolean value true

Các phương thức để PHP serialization là serialize() và unserialize(). Nếu bạn có thể truy cập mã nguồn, bạn nên bắt đầu bằng cách tìm kiểm hàm unserialize() ở bất cứ đâu trong code và điều tra sâu hơn.

**Định dạng Java serialization**

Java dùng định dạng nhị phân để serialize . Nó khó đọc hơn nhưng bạn vẫn có thể xác định dữ liệu được serialize nếu bạn nhận ra một vài dấu hiệu quen thuộc. Ví dụ, các đối tượng serialized thường bắt đầu với các bytes giống nhau được mã hóa như ac ed ở hexa và rO0 ở Base64.

Bất cứ lớp nào thực thi java.io.Serializable có thể được serialize hay deserialize. Nếu bạn có thể truy cập mã nguồn, lưu ý bất kỳ mã nào sử dụng phương thức readObject(), được sử dụng để đọc và deserialize dữ liệu từ InputStream.

**Tận dụng các đối tượng đã serialize:**

Khai thác một vài lỗ hổng deserialization có thể rất dễ như thay đổi 1 thuộc tính trong 1 đối tượng đã serialize. Vì trạng thái của đối tượng được duy trì, bạn có thể nghiên cứu dữ liệu đã serialize để xác định và chỉnh sửa các thuộc tính thú vị. Sau đó gửi 1 đối tượng độc hại trong trang web qua quá trình deserialize. Đây là bước đầu để khai thác deserialization.

Nói chung, có hai cách tiếp cận bạn có thể làm để lạm dụng các đối tượng đã serialize. Bạn có thể chỉnh sửa đối tượng trực tiếp trong định dạng byte stream của nó hay bạn có thể ghi 1 tập lệnh nhỏ tương ứng với ngôn ngữ đó để tạo và serialize chính đối tượng mới. Cách tiếp cận thứ hai thường dễ dàng hơn khi làm việc với các định dạng serialization nhị phân.

**Điều chỉnh các thuộc tính của đối tượng**

Khi giả mạo dữ liệu, miễn là kẻ tấn công giữ cho đối tượng đã serialize được hợp lệ, quá trình deserialization sẽ tạo 1 đối tượng ở phía máy chủ với thuộc tính đã được chỉnh sửa.

Giả sử 1 trang web sử dụng đối tượng User đã serialize để lưu trữ dữ liệu về phiên của người dùng trong 1 cookie. Nếu kẻ tấn công phát hiện đối tượng đó trong 1 yêu cầu http, họ có thể giải mã nó để tìm thấy luồng byte sau:

O:4:"User":2:{s:8:"username";s:6:"carlos";s:7:"isAdmin";b:0;}

Thuộc tính isAdmin là 1 điểm nguy hiểm nhất. Kẻ tấn công có thể thay đổi giá trị bool là 1, mã hóa lại đối tượng và ghi đè cookie với giá trị đó. Bình thường nó không có tác dụng. Tuy nhiên, nếu trang web dùng cookie này để kiểm tra xem người dùng hiện tại có được phép truy cập chức năng quản trị không:

$user = unserialize($\_COOKIE);

if ($user->isAdmin === true) {

// allow access to admin interface

}

Lỗ hổng này sẽ khởi tạo 1 đối tượng User dựa trên dữ liệu từ cookie, bao gồm thuộc tính isAdmin. Ứng dụng không kiểm tra tính xác thực của đối tượng đã serialize. Dữ liệu này sau đó được đưa vào câu lệnh điều kiện, sẽ cho phép leo thang đặc quyền.

Ví dụ này không phổ biến trong thực tế. Tuy nhiên, chỉnh sửa thuộc tính theo cách này đang mô tả bước đầu tiên để mở rộng phạm vi tấn công do lỗi insecure deserialization sinh ra.

**Điều chỉnh kiểu dữ liệu**

Logic của PHP dễ bị dính lỗ hổng với cách này vì hành vi của toán tử loose comparison khi so sánh các kiểu dữ liệu khác nhau. Ví dụ, nếu bạn so sánh giữa số và chuỗi, php sẽ cố gắng chuyển chuỗi về số, nghĩa là 5 == "5" sẽ trả về đúng.

Điều này cũng hoạt động đối với bất kì chuỗi nào bắt đầu với 1 số. Trong trường hợp đó, PHP sẽ chuyển toàn bộ chuỗi sang integer dựa vào số ban đầu. Phần còn lại của chuỗi bị bỏ qua hoàn toàn. Do đó, 5 == "5 of something" ở trong thực tế là 5 == 5.

Nó trở nên kì lạ hơn khi so sánh 1 chuỗi với int(0):

0 == "Example string" // true

Tại sao? Vì không có số, nghĩa là 0 chữ số trong chuỗi. Php xử lí toàn bộ chuỗi này như số 0.

Xem xét 1 trường hợp mà toán tử == được sử dụng kết hợp với dữ liệu do người dùng kiểm soát từ 1 đối tượng đã deserialize. Điều này có thể dẫn đến lỗi logic nguy hiểm:

$login = unserialize($\_COOKIE)

if ($login['password'] == $password) {

// log in successfully

}

Kẻ tấn công chỉnh sửa thuộc tính password để mà nó chứa số 0 thay vì 1 chuỗi mong muốn. Khi mật khẩu được lưu trữ không bắt đầu với 1 số, điều kiện sẽ luôn trả về true 🡪 bỏ qua xác thực. Điều này chỉ khả thi khi deserialization duy trì kiểu dữ liệu. Nếu code lấy mật khẩu từ yêu cầu, số 0 sẽ chuyển sang chuỗi thì điều kiện sẽ trả về false.

Khi điều chỉnh kiểu dữ liệu của 1 đối tượng đã serialize ở bất cứ định dạng nào, nhớ chỉnh sửa các type lables và độ dài trong dữ liệu đó luôn. Ngược lại, đối tượng đã serialize sẽ bị hỏng và không deserialize được.

Khi làm việc trực tiếp với các định dạng nhị phân, chúng tôi yêu cầu dùng Hackvector extension, có sẵn trong BApp store. Với HackVector, bạn có thể chỉnh sửa dữ liệu đã serialize như 1 chuỗi và nó sẽ tự động cập nhập dữ liệu nhị phânvà điều chỉnh độ lệch cho phù hợp. Điều này có thể giúp bạn tiết kiệm rất nhiều nỗ lực thủ công.

**Sử dụng chức năng của ứng dụng**

Cũng như chỉnh sửa thuộc tính, chức năng ứng dụng có thể thực hiện những hành động nguy hiểm với dữ liệu từ 1 đối tượng đã deserialization. Trong trường hợp đó, bạn có thể dùng insecure deserialization để gửi dữ liệu không mong muốn và tận dụng chức năng đó để tấn công.

Ví dụ, chức năng Delete user của trang web, ảnh hồ sơ của người dùng bị xóa bằng cách truy suất đường dẫn trong thuộc tính $user->image\_location. Nếu người dùng đó được tạo từ 1 đối tượng đã serialize, kẻ tấn công có thể khai thác nó bằng cách gửi 1 đối tượng đã chỉnh sửa với image\_location là 1 đường dẫn tệp bất kỳ. Sau đó, xóa tài khoản người dùng của chính họ cũng sẽ xóa tệp tùy ý này.

Ví dụ này dựa vào việc kẻ tấn công gọi phương thức nguy hiểm qua chức năng người dùng có thể truy cập. Tuy nhiên, insecure deserialization trở nên nguy hiểm hơn khi bạn tạo ra các khai thác mà gửi dữ liệu trong các phương thức nguy hiểm 1 cách tự động. Nó được thực thi bằng cách dùng magic methods.

**Magic methods**

Magic methods là 1 tập hợp con các methods đặc biệt mà bạn không phải gọi rõ ràng. Thay vào đó, chúng sẽ được gọi tự động khi 1 sự kiện xảy ra. Magic methods là 1 tính năng phổ biến của lập trình hướng đối tượng. Đôi khi chúng được biểu thị bằng tiền tố hay bao quanh tên phương thức bằng dấu gạch dưới kép.

Các nhà phát triển có thể thêm magic methods đến 1 lớp để xác định trước code nào sẽ được thực thi khi sự kiện xảy ra. Chính xác khi nào và tại sao 1 magic method được gọi sẽ khác nhau giữa các phương thức. Một trong những ví dụ phổ biến nhất trong PHP là \_\_construct(), được gọi khi 1 đối tượng của 1 lớp được khởi tạo, tương tự như \_\_init\_\_ của Python. Nhìn chung, constructor magic methods chứa code để khởi tạo các thuộc tính của đối tượng. Tuy nhiên, magic methods có thể được điều chỉnh bởi các nhà phát triển để thực thi bất cứ mã nào họ muốn.

Magic methods được dùng khá rộng rãi và bản thân chúng không đại diện cho 1 lỗ hổng. Nhưng chúng có thể trở nên nguy hiểm khi code mà chúng thực thi xử lí dữ liệu do người dùng kiểm soát, ví dụ từ 1 đối tượng đã deserialization. Điều này có thể bị khai thác bởi kẻ tấn công để tự động gọi các phương thức trên dữ liệu đã deserialization khi điều kiện được thỏa.

Quan trọng nhất trong bối cảnh này, một vài ngôn ngữ có magic methods được gọi tự động trong quá trình deserialization. Ví dụ, phương thức unserialize() của PHP tìm kiếm và gọi magic method \_\_wakeup() của 1 đối tượng.

Trong Java deserialization, việc tương tự cũng áp dụng với phương thức ObjectInputStream.readObject(), được dùng để đọc dữ liệu từ luồng byte ban đầu và về cơ bản hoạt động như 1 constructor để khởi tạo lại 1 đối tượng đã serialize. Tuy nhiên, các lớp Serializable cũng có thể khai báo các phương thức readObject() riêng:

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException

{

// implementation

}

Phương thức readObject() được khai báo theo cách này hoạt động như magic method được gọi trong quá trình deserialization. Điều này cho phép lớp kiểm soát deserialization của các trường riêng của nó chặt chẽ hơn.

Bạn nên chú ý đến bất cứ lớp nào có chứa magic methods. Chúng cho phép bạn gửi 1 đối tượng đã serialize vào trong code của trang web trước khi đối tượng được deserialize 1 cách đầy đủ. Đây là điểm đầu tiên để tạo các khai thác nâng cao hơn.

**Tiêm các đối tượng tùy ý**

Như chúng ta đã thấy, rất ít khả năng khai thác lỗ hổng insecure deserialization bằng cách chỉnh sửa đối tượng được cung cấp bởi trang web. Tuy nhiên, tiêm các loại đối tượng tùy ý có thể mở ra nhiều cơ hội hơn.

Trong lập trình hướng đối tượng, các phương thức chỉ khả thi với đối tượng khi được khai báo trong lớp của nó. Do đó, nếu kẻ tấn công có thể thao túng được lớp của đối tượng được gửi trong dữ liệu đã serialize, họ có thể ảnh hưởng đến cách code được thực thi sau đó, thậm chí trong quá trình deserialization.

Các phương thức deserialization thường không kiểm tra chúng đang deserialize cái gì. Điều này có nghĩa là bạn có thể gửi các đối tượng của 1 lớp seriable bất kì mà khả thi với trang web và đối tượng sẽ được deserialize. Điều này cho phép kẻ tấn công tạo các biến thể của các lớp tùy ý. Sự thật là đối tượng đó không thuộc lớp mong đợi cũng không vấn đề gì. Loại đối tượng không mong đợi có thể gây ra ngoại lệ trong logic ứng dụng, nhưng đối tượng độc hại sẽ được khởi tạo ngay sau đó.

Nếu kẻ tấn công truy cập được mã nguồn, họ có thể nghiên cứu tất cả lớp chi tiết. Để thực hiện 1 khai thác đơn giản, họ sẽ tìm kiếm các lớp chứa magic methods deserialization, sau đó kiểm tra xem chúng có thực hiện các hành vi nguy hiểm với dữ liệu có thể kiểm soát không. Kẻ tấn công sau đó có thể gửi 1 đối tượng đã serialized của lớp đó để dùng magic methods cho 1 khai thác.

Các lớp chứa magic methods deserialization có thể được dùng để khởi tạo các cuộc tấn công phức tạp hơn liên quan đến 1 chuỗi lời gọi phương thức được biết đến là gadget chain.

**Gadget chains**

Gadget là 1 đoạn mã tồn tại trong ứng dụng mà có thể giúp kẻ tấn công đạt được mục đích. Một gadget riêng không thể trực tiếp gây hại với đầu vào của người dùng. Tuy nhiên, mục đích của kẻ tấn công chỉ đơn giản là gọi 1 phương thức sẽ gửi đầu vào của họ trong gadget khác. Bằng việc nối nhiều gadgets với nhau, kẻ tấn công có thể gửi đầu vào của họ trong 1 sink gadget nguy hiểm, nơi nó có thể gây thiệt hại lớn.

Quan trọng là không như một số loại khai thác khác, 1 gadget chain không phải là payload của việc nối chuỗi các phương thức do kẻ tấn công xây dựng. Tất cả mã đã có sẵn trên trang web. Điều duy nhất kẻ tấn công kiểm soát là dữ liệu được gửi trong gadget chain. Điều này thường được làm bằng magic method được gọi trong quá trình deserialization, thường biết đến là “kick-off gadget”.

Trong thực tế, rất nhiều lỗ hổng insecure deserialization chỉ có thể khai thác qua việc sử dụng gadget chains. Điều này đôi khi có thể là 1 hay hai bước chain, nhưng xây dựng các cuộc tấn công rất nghiêm trọng có thể sẽ yêu cầu 1 chuỗi phức tạp hơn các khởi tạo đối tượng và gọi phương thức. Do đó, có thể xây dựng gadget chains là 1 trong những chìa khóa quan trọng để khai thác thành công insecure deserialization.

**Làm việc với gadget chains được dựng sẵn**

Xác định gadget chains thủ công có thể là 1 một quá trình khó khăn và không khả thi khi không truy cập được mã nguồn. May mắn thay, có một vài lựa chọn để làm việc với gadget chains được dựng sẵn mà bạn có thể thử trước.

Có một vài công cụ cung cấp các chains đã được khám phá trước đó mà khai thác thành công trên những trang web khác. Thậm chí nếu bạn không truy cập được mã nguồn, bạn có thể dùng những công cụ này để xác định và khai thác lỗ hổng insecure deserialization với ít nỗ lực hơn. Cách tiếp cận này khả thi vì các thư viện mà chứa gadget chains có thể khai thác được sử dụng lan rộng. Ví dụ, nếu 1 gadget chain trong thư viện Apache Commons Collections của Java có thể được khai thác trên 1 trang web, bất cứ trang web nào dùng thư viện đó cũng có thể khai thác bằng chain tương tự.

**ysoserial**

Một công cụ cho Java deserialization là “ysoserial”. Nó cho phép bạn chọn một trong những gadget chains cho 1 thư viện mà bạn nghĩ ứng dụng đích đang dùng, sau đó gửi 1 lệnh mà bạn muốn thực thi. Nó sẽ tạo 1 đối tượng đã serialized thích hợp dựa trên chain đã chọn. Điều này vẫn liên quan đến việc thử và sai nhất định, nhưng nó ít tốn công sức hơn so với việc xây dựng gadget chains riêng của bạn theo cách thủ công.

Lưu ý là không phải tất cả gadget chains trong ysoserial cho phép bạn thực thi code. Thay vào đó, nó có thể rất hữu dụng cho nhiều mục đích khác. Ví dụ, bạn có thể dùng những cái sau để giúp bạn nhanh chóng phát hiện insecure deserialization trên hầu hết các máy chủ:

* URLDNS chain thực thi tra cứu DNS cho Url được cung cấp. Quan trọng nhất, nó không dựa trên ứng dụng đích dùng 1 thư viện có lỗ hổng và làm việc trên phiên bản Java nào. Điều này làm cho nó trở thành gadget chain phổ biến nhất cho mục đích phát hiện. Nếu bạn phát hiện 1 đối tượng đã serialize trong lưu lượng truy cập, bạn có thể dùng gadget chain này để tạo 1 đối tượng thực thi DNS tương tác với máy chủ Burp Collaborator. Nếu nó làm vậy, bạn có thể chắc chắn rằng deserialization đã xảy ra trên trang web.
* JRMPClient là 1 chain phổ biến khác mà bạn có thể dùng cho phát hiện ban đầu. Nó làm cho máy chủ thử thiết lập 1 kết nối TCP đến địa chỉ IP được cung cấp. Lưu ý là bạn cần cung cấp 1 địa chỉ IP thô hơn là tên 1 máy chủ. Chain này có thể rất hữu dụng trong các môi trường mà tất cả lưu lượng truy cập ra bị tường lửa chặn, bao gồm các tra cứu DNS. Bạn có thể thử tạo các payloads với 2 địa chỉ IP khác nhau: 1 cái cục bộ và 1 cái bị tường lửa chặn, cái bên ngoài. Nếu ứng dụng phản hồi ngay lập tức với payload ở địa chỉ cục bộ, nhưng bị treo với địa chỉ bên ngoài, dẫn đến sự trì hoãn trong phản hồi, điều này chỉ ra rằng gadget chain đã hoạt động vì máy chủ đã cố gắng kết nối đến địa chỉ bị chặn. Trong trường hợp đó, sự khác nhau về thời gian trong phản hồi có thể giúp bạn phát hiện deserialization có xảy ra trên máy chủ không, thậm chí trong trường hợp bị blind.

**PHP Generic Gadget Chains**

Hầu hết các ngôn ngữ thường gặp phải các lỗ hổng insecure deserialization đều có các công cụ POC tương ứng. Ví dụ, với những trang viết bằng PHP bạn có thể dùng “PHP Generic Gadget Chains” (PHPGGC)

Điều quan trọng phải nhớ là lỗ hổng do quá trình deserialize dữ liệu có thể kiểm soát, không liên quan đến 1 gadget chain trong code của trang web hay thư viện chúng dùng. Gadget chain chỉ là 1 phương tiện để thao túng luồng dữ liệu có hại khi chúng được tiêm vào. Điều này cũng áp dụng cho các lỗ hổng memory corruption dựa vào deserialization dữ liệu không đáng tin cậy. Nói cách khác, 1 trang web vẫn có thể bị dính lỗ hổng ngay cả khi nó đã quản lí điểm vào của mỗi gadget chain.

**Làm việc với gadget chains trong tài liệu**

Không phải lúc nào cũng có công cụ chuyên dụng để khai thác các gadget chains đã biết trong thư viện được dùng bởi ứng dụng. Trong trường hợp đó, bạn nên tìm kiếm trực tuyến để xem có bất kỳ tài liệu nào mà khai thác bạn có thể áp dụng 1 cách thủ công. Tinh chỉnh mã có thể yêu cầu một số hiểu biết cơ bản về ngôn ngữ và thư viện, và bạn phải tự serialize đối tượng, nhưng cách tiếp cận này thường tốn ít nỗ lực hơn việc xây dựng 1 exploit ngay từ đầu.

Ngay cả khi bạn không thể tìm thấy 1 gadget chain dùng được, bạn vẫn nhận được những kiến thức quý giá giúp bạn tạo ra 1 khai thác của riêng mình.

**Tạo ra khai thác của riêng mình**

Khi các gadget chains có sẵn và tài liệu khai thác không thành công, bạn vẫn cần tạo 1 khai thác của riêng mình.

Để thành công dựng được gadget chain, bạn phải truy cập được mã nguồn. Bước đầu tiên là học mã nguồn đó để xác định 1 lớp chứa magic method được gọi trong quá trình deserialization. Truy cập hàm mà magic method này thực thi để xem nó có thực hiện hành vi nguy hiểm với các thuộc tính do người dùng kiểm soát không. Điều này luôn đáng kiểm tra trong mọi trường hợp.

Nếu magic method không thể tự khai thác được, nó vẫn có thể là 1 gadget ban đầu cho gadget chain. Nghiên cứu bất cứ method nào mà gadget ban đầu gọi đến. Trong đó có cái nào thực hiện hành vi nguy hiểm với dữ liệu mà bạn kiểm soát không? Nếu không, xem xét kỹ hơn từng phương thức mà sau đó chúng được gọi..

Lặp lại quá trình này, theo dõi những giá trị bạn có quyền truy cập cho đến khi bạn chạm đến 1 ngõ cụt hay xác định 1 gadget sink nguy hiểm mà dữ liệu độc hại của bạn được chuyển vào.

Khi bạn tìm ra cách xây dựng 1 gadget chain thành công trong code của ứng dụng, bước tiếp thoe là tạo 1 đối tượng đã serialize chứa payload đó. Đây chỉ đơn giản là 1 trường hợp khai báo lớp trong mã nguồn và tạo 1 đối tượng đã serialize 1 cách hợp lệ với giá trị thích hợp cho khai thác của bạn. Như ở ví dụ trước, điều này khá đơn giản khi làm việc với định dạng serialization ở dạng chuỗi.

Làm việc với định dạng nhị phân, như khi xây dựng 1 Java deserialization exploit thường có thể cồng kềnh. Khi thực hiện những thay đổi nhỏ với 1 đối tượng đang có, bạn có thể thấy thoải mái khi làm việc trực tiếp với các bytes. Tuy nhiên, khi tạo ra sự thay đổi lớn hơn, như gửi 1 đối tượng hoàn toàn mới, điều này nhanh chóng trở nên phi thực tế. Việc viết mã của riêng bạn bằng ngôn ngữ đích để tạo và serialize đối tượng sẽ trở nên dễ dàng hơn.

Khi tạo 1 gadget chain của riêng bạn, tìm kiếm các cơ hội để dùng chúng mở rộng phạm vi tấn công, thực thi 1 lỗ hổng thứ hai.

Bằng việc nghiên cứu mã nguồn cẩn thận, bạn có thể khám phá các gadget chains dài hơn cho phép bạn xây dựng các cuộc tấn công rất nghiêm trọng, thường là RCE.