Programowanie Defensywne

Podatności XSS i Path Traversal

Arkadiusz Chrobot

Katedra Systemów Informatycznych

21 marca 2024

Plan

1. Podatność XSS
2. Podatność Path Traversal

Podatność XSS

Załóżmy, że chcemy zbudować prosty serwer WWW (taki „bieda serwer” ☺), który będzie wysyłał stronę z pytaniem o imię użytkownika i jeśli je otrzyma, to na kolejnej stronie umieści odpowiedni napis powitalny. Wydaje się to prostym zadaniem, ale tkwi w nim „drobna” pułapka. Nasz serwer będzie obsługiwał tylko metodę GET, ale problem dotyczy także POST. Metody w języku Java, które odpowiadają za obsługę żądań HTTP w serwerze przedstawiono na kolejnych slajdach.

Podatność XSS

```
@Override
public void handle(HttpExchange exchange) throws IOException
 ↪  {
    if("GET".equals(exchange.getRequestMethod())) {
        String requestParameters =
 ↪  exchange.getRequestURI().toString();
        String page;
        if(!requestParameters.contains("name"))
            page = prepareMainResponse();
        else
            page =
 ↪  prepareGreetingsResponse(requestParameters);
        exchange.sendResponseHeaders(200,page.length());
        OutputStream stream = exchange.getResponseBody();
        stream.write(page.getBytes());
        stream.close();
    }
}
```

Podatność XSS

```
private String prepareMainResponse() {
    StringBuilder response = new StringBuilder();
    response
            .append("<html>")
            .append("<head></head>").append("<body>")
            .append("<form action=\"\" method=\"GET\">")
            .append("<label for=\"name\">Input Your name:
 ↪  </label>")
            .append("<input name=\"name\" id=\"name\">")
            .append( "<button>Submit</button>")
            .append("</form>").append("</body>")
            .append( "</html>");
    return response.toString();
}
```

Podatność XSS

```
private String prepareGreetingsResponse(String
↪   requestParameters) {
    StringBuilder response = new StringBuilder();
    String name =
↪   java.net.URLDecoder.decode(requestParameters.split("=")[1],
↪   StandardCharsets.UTF_8);
    response
            .append("<html>")
            .append("<head></head>")
            .append("<body>")
            .append("Hello, ")
            .append(name)
            .append("!")
            .append("</body>")
            .append("</html>");
    return response.toString();
}
```

Podatność XSS

Zwróćmy uwagę, że parametr `name` po zdekodowaniu z formatu URL jest bezpośrednio umieszczany w kodzie strony z powitaniem. Oznacza to, że jeśli np. użytkownik wprowadzi następujący łańcuch znaków: `<script>alert(1)</script>`, to przeglądarka go potraktuje jako fragment kodu HTML, który zawiera kod JavaScript i oba wykona. W tym wypadku pokaże się w oknie przeglądarki okno modalne z komunikatem „1”, ale taki skrypt może być bardziej skomplikowany i np. wykraść ciasteczka lub inne dane pozwalające przejąć sesję legalnego użytkownika, albo wymusić wykonanie w jego imieniu jakiejś operacji.

Podatność XSS

Warianty

Podatność XSS, czyli Cross-Site Scripting polega na wstrzyknięciu kodu JavaScript[1] do aplikacji webowej i występuje w trzech odmianach: odbitego XSS (ang. *Reflected XSS*), które polega na tym, ze kod wstrzyknięty w zapytaniu jest przesyłany w odpowiedzi, trwałego XSS (ang. *Persistent or Stored XSS*), gdzie kod jest zapisywany na stałe po stronie serwera i uruchamiany za każdym razem, gdy użytkownik odwiedza stronę, gdzie został wbudowany i w końcu XSS bazujący na DOM (ang. *DOM-Based XSS*), polega na zmianie przez wstrzyknięty skrypt DOMu strony, z którego którego korzysta inny skrypt, aby doprowadzić do jego błędnego działania.

[1] W szerszym znaczeniu może to być także kod HTML, CSS lub dowolnego języka skryptowego, który może być zinterpretowany i wykonany przez atakowaną aplikację internetową.

Podatność XSS

Ochrona

Podobnie jak w przypadku wstrzykiwania SQL, ochrona przez XSS polega na walidacji danych wejściowych. Niestety, nie jest ona prosta, bo jej charakter zależy od miejsca w stronie, gdzie ten kod JavaScript może być wstrzyknięty. W przypadku tej podatności OWASP także ma listę ▸ zaleceń, które warto zastosować. Należą do nich między innymi:

- zakodowanie niebezpiecznych znaków w postaci encji HTML (nie zawsze wystarcza),
- użycie systemów szablonów, które automatycznie kodują niebezpieczne znaki (np. Jinja2),
- użycie frameworków, które posiadają wbudowaną ochronę (np. Angular),
- unikanie funkcji `eval()`,
- filtrowanie danych wejściowych zarówno w wariancie akceptacji (ang. *white list*), jak i odrzucenia (ang. *black list*),
- użycie nagłówków CSP (Content Security Policy).

Podatność XSS

Ochrona

Warto zauważyć, że ochrona przeciw XSS powinna być stosowana zarówno po stronie serwera, jak i klienta, bo to przede wszystkim ta druga jest narażona na konsekwencje ataków z wykorzystaniem tej podatności.
W przypadku naszego serwera można zastosować wyrażenia regularne, aby uchronić się przed atakiem, którego scenariusz został przedstawiony wcześniej. To rozwiązanie nie uwzględnia jednak innych form ataku z wykorzystaniem XSS. Polega ono na wykryciu, czy w przekazanym przez użytkownika łańcuchu znajdują się „zakazane znaki” i jego odrzuceniu, jeśli tak faktycznie jest. Odpowiedni fragment kodu, który należy dodać do `prepareGreetingsResponse()` został umieszczony w kolejnym slajdzie.

Podatność XSS

Ochrona

```
String name = java.net.URLDecoder.decode(
        requestParameters.split("=")[1],
 ↪  StandardCharsets.UTF_8);
Pattern pattern =
 ↪  Pattern.compile("<[a-z]*>[a-z]*\\(.*\\)<.*>");
if(pattern.matcher(name).find())
    name = "You are doing something nasty";
else
    name = "Hello, " + name;
```

Podatność XSS

Ochrona

Zaprezentowane na slajdzie 11 rozwiązanie działa na zasadzie „czarnej listy” (ang. *black list*), czyli stara się wykryć niepożądane kombinacje znaków w danych wejściowych. To podejście może być użyteczne, ale trudno jest je prawidłowo zastosować, bo takich niebezpiecznych przypadków może być wiele. Wystarczy sprawdzić kilka przykładów z ▸ listy, aby się przekonać, jak bardzo jest zawodne.
Następny slajd zawiera podobny fragment kodu, jak poprzedni, ale stosujący podejście „białej listy”, czyli określa jakie znaki *mogą* się znajdować w danych wejściowych, a jakie nie powinny. Ponieważ aplikacja spodziewa się, że użytkownik wprowadzi swoje imię, to akceptowane są tylko duże i małe litery. Jest to prostsze rozwiązania, a zarazem bardziej skuteczne.

Podatność XSS

Ochrona

```
String name =
 ↪  java.net.URLDecoder.decode(requestParameters.split("=")[1],
 ↪  StandardCharsets.UTF_8);
Pattern pattern = Pattern.compile("[^A-Za-z]");
if(pattern.matcher(name).find())
    name = "You are doing something nasty";
else
    name = "Hello, " + name;
```

Path Traversal

Przyjmijmy, że w wyniku problemów z pierwotną wersją opisanego wcześniej serwera doszliśmy do wniosku, że lepiej będzie, jeśli zmodyfikujemy go tak, aby przesyłał pliki HTML ze statyczną zawartością z ustalonego katalogu. Nazwy tych plików będą pobierane z URI uzyskiwanego z otrzymanego żądania. Kod metody obsługującej to żądanie znajduje się na następnym slajdzie.

Path Traversal

```
public void handle(HttpExchange exchange) throws IOException
 ↪  {
    if("GET".equals(exchange.getRequestMethod())) {
        String requestURL =
 ↪  exchange.getRequestURI().toString();
        String filePath =
 ↪  URLDecoder.decode(requestURL.substring(1),
 ↪  StandardCharsets.UTF_8);
        String fileContent = getFileContent(filePath);

 ↪  exchange.sendResponseHeaders(200,fileContent.length());
        OutputStream stream = exchange.getResponseBody();
        stream.write(fileContent.getBytes());
        stream.close();
    }
}
```

Path Traversal

Plik, którego zawartość ma być wyświetlona przez przeglądarkę, jest odczytywany przez metodę `getFileContent()`, która dodaje do jego nazwy otrzymanej z URI ścieżkę do katalogu i stosuje standardowe strumienie Javy, aby go odczytać. Wydaje się, że nic złego nie może się tutaj stać. Niestety, ale to nie jest prawda. W URL można poprzedzić nazwę pliku np. ciągiem `../../`, który nakaże metodzie `getFileContent()` opuścić „bezpieczny” katalog i rozpocząć odczyt pliku poza nim. Dwie kropki oznaczają katalog nadrzędny, natomiast `/` jest separatorem nazw katalogów (lub katalogiem głównym) w systemach kompatybilnych z Uniksem. W systemach Windows należy go zastąpić znakiem `\`. Aby atak się powiódł trzeba jeszcze ścieżkę umieszczoną w URI zakodować w formacie URL. Narzędzia do wykonania tej operacji są dostępne w wielu miejscach np. ▸ tutaj.

Path Traversal

Ochrona

Usunięcie tej podatności wymaga filtrowania ścieżek do plików w aplikacjach internetowych i usuwania z nich zbędnych elementów. Ewentualnie można ograniczyć możliwości manipulowania ścieżką do plików w metodach związanych z ich wgrywaniem. Należy też zwrócić uwagę na zewnętrzne komponenty i zbadać, czy są one odpowiednio zabezpieczone. Filtrowanie może być trudne, z uwagi na możliwość wielorakiego kodowania „niebezpiecznych" znaków. Dodatkowym zabezpieczeniem są odpowiednie uprawnienia do plików (np. serwer nie powinien pracować na prawach użytkownika uprzywilejowanego, najlepiej żeby znajdował się w bezpiecznym kontenerze lub na maszynie wirtualnej).

W przypadku opisywanego serwera wystarczy zastosować rozwiązanie polegające na usuwaniu z URI wszystkich znaków, poza nazwą pliku, przy założeniu, że ta nazwa znajduje się za ostatnim znakiem `/`. Odpowiednią modyfikację metody `handle()` zaprezentowano na następnym slajdzie.

Path Traversal

Ochrona

```
public void handle(HttpExchange exchange) throws IOException
 ↪  {
    if("GET".equals(exchange.getRequestMethod())) {
        String requestURL =
 ↪  exchange.getRequestURI().toString();
        String filePath =
 ↪  URLDecoder.decode(requestURL,StandardCharsets.UTF_8);
        filePath =
 ↪  filePath.substring(filePath.lastIndexOf("/"));
        String fileContent = getFileContent(filePath);

 ↪  exchange.sendResponseHeaders(200,fileContent.length());
        OutputStream stream = exchange.getResponseBody();
        stream.write(fileContent.getBytes());
        stream.close();
    }
}
```

Pytania

?

Koniec

Dziękuję Państwu za uwagę!