Programowanie Defensywne

Podatność XXE, niebezpieczna deserializacja

Arkadiusz Chrobot

Katedra Systemów Informatycznych

2 kwietnia 2024

Plan

1. Podatność XXE
2. Niebezpieczna deserializacja

Podatność XXE

Opis

Podatność XXE (ang. *XML External Entity*) jest, co sugeruje jej nazwa, związana z przetwarzaniem formatu danych XML. Ten format używany jest w plikach, ale także niektórych protokołach (np. SOAP, XMPP/Jabber). Przyjrzymy się tej podatności, na przykładzie pliku zawierającego dane zapisane w XML.
Przyjmijmy, że otrzymaliśmy zadanie napisania komponentu, który wyświetli w formie czytelniej dla przeciętnego człowieka dane o książkach zapisane w pliku. Otrzymaliśmy przykładowy plik, którego zawartość jest przedstawiona na następnym slajdzie.

Podatność XXE

Opis

```
<?xml version="1.0" encoding="utf-8"?>
<books>
        <book genre="science fiction">
                <author>Andy Weir</author>
                <title>Marsjanin</title>
                <publisher>Akurat</publisher>
                <year>2014</year>
        </book>
</books>
```

Podatność XXE

Opis

Jak łatwo zauważyć, struktura takiego pliku jest bardzo prosta, więc tworzymy prosty prototyp jego czytnika w języku Java. Konstrukcja tego komponentu wydaje się na tyle prosta, że nie może w niej być żadnej podatności.

Podatność XXE

Opis

```
public void printXMLFile() {
    DocumentBuilderFactory factory =
↪   DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder documentBuilder =
↪   factory.newDocumentBuilder();
        Document document =
↪   documentBuilder.parse(xmlFile);
        parseXMLDocument(document);
    } catch(ParserConfigurationException | SAXException
↪   | IOException e) {
        e.printStackTrace();
    }

}
```

Podatność XXE

Opis

```
private void parseXMLDocument(Document document) {
 NodeList list = document.getElementsByTagName("book");
 for(int i=0; i< list.getLength(); i++) {
  Node node = list.item(i);
  if(node.getNodeType() == Node.ELEMENT_NODE) {
   Element element = (Element) node;
   System.out.println("Gatunek: "+element.getAttribute("genre"));
   System.out.println("Autor: " +
↪   element.getElementsByTagName("author").item(0).getTextContent());
   System.out.println("Tytuł: " +
↪   element.getElementsByTagName("title").item(0).getTextContent());
   System.out.println("Wydawca: " +
↪   element.getElementsByTagName("publisher").item(0).getTextContent());
   System.out.println("Rok wydania: " +
↪   element.getElementsByTagName("year").item(0).getTextContent());
  }
 }
}
```

Podatność XXE

Opis

Niestety, nie uwzględniliśmy w naszym prototypie tego, że korzysta on z implementacji parsera XML dostępnego w języku Java. Ten parser jest zgodny ze standardem XML, który pozwala na definiowanie *encji*, obsługiwanych przy pomocy mechanizmu, który działa podobnie do preprocesora w języku C. Zatem, jeśli nasz komponent zostanie użyty np. do realizacji zdalnej usługi przetwarzania pliku XML, to intruz może wykorzystać go do uzyskania dostępu do treści plików znajdujących się na serwerze, o ile tylko będzie znał ich lokalizację. W tym celu musi jedynie odpowiednio spreparować plik XML, dodając do niego sekcję DTD (ang. *Document Type Definition*) i definiując w niej odpowiednią encję, do której odwołanie następnie umieści w jedynym z odczytywanych znaczników. Przykład takiego pliku XML znajduje się na następnej stronie.

Podatność XXE

Opis

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE books [
 <!ENTITY trap SYSTEM "../java/pl/kielce/tu/XXEDemo.java">

 ]>
<books>
        <book genre="science fiction">
                <author>Andy Weir</author>
                <title>Marsjanin &trap;</title>
                <publisher>Akurat</publisher>
                <year>2014</year>
        </book>
</books>
```

Podatność XXE

Opis

Z możliwości definiowania encji korzysta także atak „billion laughs”, typu DoS. Choć jest „spokrewniony” z podatnością XXE, to jednak wykorzystuje encje w innym celu. W przypadku takiego ataku intruz tworzy encje, które rekurencyjnie się do siebie odwołują. Choć rozmiar pliku, w którym są one zapisane nie przekracza 1*KiB*, to jego przetwarzanie wymaga od parsera użycia nawet kilkudziesięciu *GiB* pamięci RAM. Przykład takiego pliku znajduje się na następnym slajdzie. Twórcy niektórych parserów XML domyślnie ograniczają głębokość rekursji encji w pliku, co powoduje, że atak będzie miał ograniczony zasięg — spowoduje tylko awarię parsera i być może usługi. Takie zabezpieczenie ma wbudowany parser XML w najnowszych edycjach języka Java, ale nie ma jej parser języka Python, dlatego na kolejnych slajdach znajduje się implementacja czytnika pliku w tym właśnie języku.

Podatność XXE

Opis

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2
↪  "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3
↪  "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4
↪  "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5
↪  "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6
↪  "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7
↪  "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8
↪  "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9
↪  "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Podatność XXE

Opis

```
#!/usr/bin/env python3
import sys
from xml.dom.minidom import parse

def parseXML(fileName):
    dom = parse(fileName)
    book = dom.getElementsByTagName("book")[0]
    print("Gatunek: "+book.getAttribute("genre"))
    print("Autor:
     ↪  "+dom.getElementsByTagName("author")[0].firstChild.data)
    print("Tytuł:
     ↪  "+dom.getElementsByTagName("title")[0].firstChild.data)
    print("Wydawca:
     ↪  "+dom.getElementsByTagName("publisher")[0].firstChild.data)
    print("Rok wydania:
     ↪  "+dom.getElementsByTagName("year")[0].firstChild.data)
```

Podatność XXE

Opis

```
if __name__ == "__main__":
    if len(sys.argv) == 2:
        parseXML(sys.argv[1])
```

Podatność XXE

Ochrona

Ochrona przez podatnością XXE, jak również innymi problemami związanymi z przetwarzaniem danych w formacie XML polega na:

- wyłączeniu w konfiguracji parsera przetwarzania encji zewnętrznych,
- ograniczeniu zagnieżdżeń encji,
- ograniczeniu rozmiaru przetwarzanego pliku XML,
- narzuceniu limitu czasu przetwarzania pliku.

Fundacja OWASP udostępnia na swojej stronie artykuł o zabezpieczeniach przeciwko XXE. W nowszych edycjach języka Java (od wersji 6) można ustawić flagę `FEATURE_SECURE_PROCESSING`, która nakazuje implementacji parsera bezpieczne przetwarzanie formatu XML. Przykład jej zastosowania zamieszczono na kolejnym slajdzie. Z kolei w przypadku języka Python zamiast pakietu `xml.dom.minidom` można zastosować pakiet `defusedxml.minidom`, który między innymi wyłącza przetwarzanie zewnętrznych encji. Można go zainstalować przy użyciu narzędzia `pip`.

Podatność XXE

Ochrona

```
public void printXMLFile() {
  DocumentBuilderFactory factory =
↪   DocumentBuilderFactory.newInstance();
  try {
    factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,true);
    DocumentBuilder documentBuilder = factory.newDocumentBuilder();
    Document document = documentBuilder.parse(xmlFile);
    parseXMLDocument(document);
  } catch(ParserConfigurationException | SAXException | IOException
↪   e) {
    e.printStackTrace();
  }

}
```

Podatność XXE

Ochrona

```
#!/usr/bin/env python3
import sys
from defusedxml.minidom import parse

def parseXML(fileName):
    dom = parse(fileName)
    book = dom.getElementsByTagName("book")[0]
    print("Gatunek: "+book.getAttribute("genre"))
    print("Autor:
     ↪  "+dom.getElementsByTagName("author")[0].firstChild.data)
    print("Tytuł:
     ↪  "+dom.getElementsByTagName("title")[0].firstChild.data)
    print("Wydawca:
     ↪  "+dom.getElementsByTagName("publisher")[0].firstChild.data)
    print("Rok wydania:
     ↪  "+dom.getElementsByTagName("year")[0].firstChild.data)
```

Podatność XXE

Ochrona

```
if __name__ == "__main__":
    if len(sys.argv) == 2:
        parseXML(sys.argv[1])
```

Niebezpieczna deserializacja

Wprowadzenie

Choć XXE jest umieszczana w zestawieniach OWASP Top 10 i CWE Top 25 jako osobna podatność, to ma ona dużo wspólnego z problemem niezabezpieczonej deserializacji.
Serializacja to konwersja struktury danych, najczęściej obiektu, do formatu, który ułatwia jej zapisanie w pliku, bazie danych lub przesłanie jej przez sieć. Format ten może mieć postać tekstową (np.: XML, JSON), binarną lub hybrydową. Mechanizmy serializacji dostępne są w większości współczesnych języków programowania, takich jak Java, Python, Ruby, PHP, a także w platformie .NET [1].
Deserializacja jest działaniem odwrotnym do serializacji. Podatności wiążące się z tym działaniem wynikają głównie z tego, że domyślnie deserializacja wykonywana jest na nieuwierzytelnionych danych. Przyjrzymy się im na przykładzie języka Python.

Niebezpieczna deserializacja

Python

W języku Python najpopularniejszym mechanizmem serializacji jest pakiet `pickle`. Dostarcza on metod `dump()` i `dumps()`, które zapisują obiekt w postaci zserializowanej. Pierwsza z ich pozwala tak przekształcony obiekt zapisać do pliku, a druga go zwraca. Za deserializację odpowiedzialne są z kolei metody `load()` i `loads()`. Zserializowany obiekt jest zapisany w formacie hybrydowym, który jest językiem maszyny wirtualnej o organizacji stosowej. Szczegóły tego formatu, nazywanego przez twórców języka Python *protokołem* można poznać studiując dokumenty ▸ PEP 307, ▸ PEP 3154 i ▸ PEP 574.
Następne dwa slajdy przedstawiają skrypt w języku Python, w którym zdefiniowano klasę `Dog`, która, oprócz konstruktora, posiada metodę `bark()` i atrybut `name`. W skrypcie tworzony jest obiekt tej klasy, uruchamiana jest jego jedyna metoda, a następnie jest on serializowany i zapisywany do pliku, przy pomocy metody `save()`.

Niebezpieczna deserializacja

Python — Serialize.py

```python
#!/usr/bin/env python3
import sys, pickle

class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("{0}: {1}".format(self.name, "Hau!"))

def save(file_name, dog):
    with open(file_name, "wb") as file:
        pickle.dump(dog, file)
```

Niebezpieczna deserializacja

Python — Serialize.py

```
if __name__ == "__main__":
    dog = Dog("Reks")
    dog.bark()
    if len(sys.argv) == 2:
        save(sys.argv[1], dog)
```

Niebezpieczna deserializacja

Python

Plik stworzony przez przedstawiony na poprzednich slajdach skrypt jest odczytywany przez inny skrypt, zawarty na dwóch kolejnych slajdach, który wykonuje deserializację obiektu klasy `Dog` i wykonuje jego metodę `bark()`. Efekt jest taki sam, jak w przypadku skryptu serializującego.

Niebezpieczna deserializacja

Python — Deserialize.py

```
#!/usr/bin/env python3
import sys, pickle

class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("{0}: {1}".format(self.name, "Hau!"))

def restore(file_name):
    with open(file_name, "rb") as file:
        return pickle.load(file)
```

Niebezpieczna deserializacja

Python — Deserialize.py

```
if __name__ == "__main__":
    if len(sys.argv) == 2:
        dog = restore(sys.argv[1])
        dog.bark()
```

Niebezpieczna deserializacja

Python

Niestety, ten plik można zmodyfikować dodając do niego złośliwy kod. Robi to skrypt `Intruder.py` przedstawiony na dwóch kolejnych slajdach. Dodaje on na początku pliku (choć mógłby zastępować nim całą treść piku) ciąg bajtów:

```
b'\x80\x03cos\nsystem\nS"echo \"Aqq\""\n\x85R.'
```

Jest to kod, który powoduje wykonanie polecenia systemowego `echo "Aqq"`. Treść tego kodu w formie bardziej czytelniej dla człowieka można uzyskać za pomocą metody `dis()` z pakietu `pickletools`:

```
    0: \x80 PROTO      3
    2: c    GLOBAL     'os system'
   13: S    STRING     'echo "Aqq"'
   27: \x85 TUPLE1
   28: R    REDUCE
   29: .    STOP
highest protocol among opcodes = 2
```

Niebezpieczna deserializacja

Python

Pierwsza instrukcja określa numer wersji protokołu (w tym przypadku 3). Druga odkłada na stos referencję do funkcji `os.system`. Trzecia instrukcja odkłada na stos ciąg znaków `echo "Aqq"`. Czwarta instrukcja zdejmuje ten łańcuch i przekształca go w krotkę, którą ponownie umieszcza na stosie. Piąta instrukcja pobiera ze stosu krotkę stanowiącą argumenty i referencje do funkcji, a następnie tę funkcję wykonuje, po czym umieszcza na stosie wynik tego wykonania. Ostatnia instrukcja oznacza zakończenie przetwarzania.
Jeśli teraz skrypt `Deserialize.py` odczyta taki plik, to na ekranie pojawi się napis `Aqq`, a potem informacja o wyjątku spowodowana tym, że zdeserializowany obiekt nie posiada metody `bark()`.

Niebezpieczna deserializacja

Python — Inturder.py

```python
#!/usr/bin/env python3
import sys, pickletools

def print_pickle(file_name):
    with open(file_name, "rb") as file:
        pickle = file.read()
        print(pickle)
        pickletools.dis(pickle)

def read_pickle(file_name):
    with open(file_name, "rb") as file:
        return file.read()
```

Niebezpieczna deserializacja

Python — Inturder.py

```
def inject_pickle(file_name):
    command = b'\x80\x03cos\nsystem\nS"echo \"Aqq\""\n\x85R.'
    pickle = read_pickle(file_name)
    content = command+pickle
    pickletools.dis(content)
    with open(file_name, "wb") as file:
        file.write(content)

if __name__ == "__main__":
    if len(sys.argv) == 2:
        print_pickle(sys.argv[1])
        inject_pickle(sys.argv[1])
```

Niebezpieczna deserializacja

Python — obrona

Sposoby wykorzystania tej podatności przez intruza są jeszcze inne. W najprostszym przypadku może on wykorzystać ją do wprowadzenia fałszywych danych do aplikacji, ale może również doprowadzić do zdalnego wykonania kodu.
Najprostszą formą obrony jest nieużywanie w ogóle serializacji. Niestety, nie zawsze jest to możliwe, bo aplikacja może korzystać z komponentów, które ją przeprowadzają. W takim wypadku dokumentacja zaleca stworzenie specjalnej klasy do deserializacji, która będzie dopuszczała tylko użycie w protokole bezpiecznych metod. Można też zastosować serializację tekstową, ale ta też niesie ze sobą niebezpieczeństwa. O formacie XXE wspominaliśmy wcześniej, natomiast JSON nie określa typów danych, które zawiera i to intruz może wykorzystać.

Niebezpieczna deserializacja

Python — obrona

Zalecenia odnośnie ochrony przed niebezpieczną deserializacją przedstawia OWASP. Jeśli aplikacja deserializuje tylko dane, które sama zserializowała, lub które pochodzą ze znanego źródła, to można tę operację zabezpieczyć poprzez zastosowanie podpisów kryptograficznych. W przypadku języka Python takie sygnatury zapewnia pakiet `hmac`. Dostarcza on metody `new()`, która konfiguruje i zwraca obiekt `hmac`. Przyjmuje ona trzy argumenty: klucz (ciąg bajtów), wiadomość do podpisania i nazwę funkcji skrótu. Do wygenerowania sygnatury, w postaci ciągu bajtów, służy metoda `digest()` obiektu `hmac`.

Niebezpieczna deserializacja

Python — obrona

Ten ostatni sposób wykorzystano w skrypcie `SecureSerialize.py`. W funkcji `secure_save()` obiekt klasy `Dog` jest najpierw serializowany do ciągu bajtów za pomocą metody `dumps()` i zapisywany w zmiennej `serialized`. Następnie jest on podpisywany przy użyciu `hmac` (jest wiadomością dla `new()`). Uzyskana sygnatura jest zapisywana do zmiennej `signature` (ma rozmiar 32 bajtów). W końcu do sygnatury jest dodawany zserializowany obiekt i obie informacje, jako ciąg bajtów, są zapisywane do pliku.

Niebezpieczna deserializacja

Python — SecureSerialize.py

```
#!/usr/bin/env python3
import sys, pickle, hmac, hashlib

class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("{0}: {1}".format(self.name, "Hau!"))

def secure_save(file_name, dog):
    serialized = pickle.dumps(dog)
    signature = hmac.new(b"Ais4lee%qu", serialized,
     ↪  hashlib.sha256).digest()
    with open(file_name, "wb") as file:
        file.write(signature+serialized)
```

Niebezpieczna deserializacja

Python — SecureSerialize.py

```
if __name__ == "__main__":
    dog = Dog("Reks")
    dog.bark()
    if len(sys.argv) == 2:
        secure_save(sys.argv[1], dog)
```

Niebezpieczna deserializacja

Python — obrona

Aby to zabezpieczenie zadziałało i w ogóle możliwe było poprawne zinterpretowanie pliku, to należy także wprowadzić odpowiednie modyfikacje w skrypcie deserializującym, który będzie się nazywał teraz `SecureDeserialize.py`. Musi on także posiadać ten sam klucz, co skrypt serializujący. Program deserializujący odczytuje zawartość pliku przy pomocy zwykłej metody `read()`. Następnie oddziela z odczytanej treści pierwsze 32 bajty, a resztę, czyli zserializowany obiekt przekazuje do podpisania. Następnie za pomocą metody `compare_digest()`, porównuje sygnaturę przez siebie wyliczoną z tą odczytaną z pliku. Jeśli są zgodne, to wykonuje deserializację obiektu przy pomocy metody `loads()`. Jeśli nie, to tworzy bezpieczny obiekt klasy `Dog`, aby można było bez zagrożeń uruchomić jego metodę `bark()`.

Niebezpieczna deserializacja

Python — SecureDeserialize.py

```python
#!/usr/bin/env python3
import sys, pickle, hmac, hashlib

class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("{0}: {1}".format(self.name, "Hau!"))
```

Niebezpieczna deserializacja

Python — SecureDeserialize.py

```
def secure_restore(file_name):
    with open(file_name, "rb") as file:
        content = file.read()
        file_signature = content[:32]
        serialized = content[32:]
        signature = hmac.new(b"Ais4lee%qu", serialized,
        ↪  hashlib.sha256).digest()
        if hmac.compare_digest(file_signature, signature):
            return pickle.loads(serialized)
        else:
            return Dog("Puszek")

if __name__ == "__main__":
    if len(sys.argv) == 2:
        dog = secure_restore(sys.argv[1])
        dog.bark()
```


Insecure Deserialization. 2023. URL: https://portswigger.net/web-security/deserialization.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!