Sonderdruck aus
JavaSPEKTRUM 6/2021

Aggressives Uhrmacherhandwerk

Deserialisierungsangriffe gegen Java-Anwendungen

Rafael Fedler

Im Allgemeinen ist Java eine vergleichsweise sichere Programmiersprache. Allerdings ist sie für eine Klasse von äußerst gefährlichen Schwachstellen anfällig, die in anderen Sprachen seltener auftauchen: Schwachstellen im Umgang mit serialisierten Objekten, das heißt, Objekte, von deren Speicherbereich ein Abzug zur Ablage oder Übertragung erstellt wurde. Diese Schwachstellenkategorie erlaubt außerdem einen guten Einblick in Hacking-Methoden und -herangehensweisen.

Der Artikel „Die OWASP Web Top 10 in Java-Anwendungen" [Fed21] hat die zehn verbreitetsten Schwachstellenkategorien in Web-Anwendungen vorgestellt, wie diese in Java-Anwendungen auftreten können und wie man durch sichere Entwicklung und Konfiguration seine Anwendungen gegen Sicherheitslücken aus diesen Kategorien schützen kann.

Eine der Kategorien der OWASP Web Top 10 (in der aktuellen Fassung von 2017, [OWT10]) ist *A8: Unsichere Deserialisierung*. Diese Kategorie ist in gewisser Weise ein Sonderfall von *A1: Injection*. Die Kategorie *A1* umfasst Schwachstellen, bei denen ein Angreifer eigenen Code in eine Web-Anwendung injizieren kann. Nicht wenige der Schwachstellen aus Kategorie *A1* gehören zur „Königsklasse" der Sicherheitslücken, den sogenannten Remote Code Execution-Schwachstellen, abgekürzt RCE. Diese Schwachstellen erlauben es dem Angreifer, eigenen Code in die verwundbare Anwendung einzubringen und auszuführen (Execution) – und dies sogar aus der Ferne (Remote), das heißt über das Netzwerk.

RCE-Lücken erlauben in der Konsequenz oft nicht nur die Übernahme der betroffenen Anwendung, sondern meist des gesamten Systems, auf dem diese läuft. Wenn dieses System wiederum in einem Netzwerksegment mit anderen Systemen steht, was oft der Fall ist, kann der Angreifer nun versuchen, sich vom initial angegriffenen System auszubreiten und auf andere Systeme überzuspringen. Dies bezeichnet man als Lateral Movement. Da Angreifer mit RCE-Lücken Systeme aus der Ferne übernehmen können, erhalten diese Sicherheitslücken im Schwachstellenrisikobewertungsschema CVSS (Common Vulnerability Scoring System) meist eine Risikobewertung von „Hoch" oder „Kritisch", den beiden höchsten Stufen.

Gutes und schlechtes Beispiel zugleich

Die Kategorie *A8*, um die es hier geht, ist insofern ein Sonderfall von *A1*, als dass ein Angreifer ebenfalls Code in Zielanwendungen injizieren kann und es fast immer zur RCE kommt. Diese Kategorie ist aber auch insofern besonders, als dass ein Angreifer primär klug konstruierte Java-Objekte in den Speicher seines Ziels injiziert, in denen sich ähnlich einer Matroschka-Figur verkapselt sein Code befindet. Bestimmte Schnittstellen-Methoden dieser Objekte werden in Folge von Spezifikationen der Java-Standard-APIs nun durch die JVM automatisch bei Deserialisierung ausgeführt und durch eine Verkettung mehrerer gekapselter Objekte, von denen Methoden in Reihe aufgerufen werden, so verknüpft, dass irgendwann – am Ende dieser

Serialisierung	Deserialisierung

Person hans = new Person(18, "Hans");
FileOutputStream file =
  new FileOutputStream("hans.ser");
ObjectOutputStream out =
  new ObjectOutputStream(file);
out.writeObject(hans);
out.close();
file.close();
	FileInputStream file =
	  new FileInputStream("hans.ser");
	ObjectInputStream in =
	  new ObjectInputStream(file);
	Person hansFromDisk =
	  (Person)in.readObject();
	in.close();
	file.close();

Tabelle 1: Beispiel für Serialisierung und Deserialisierung in Java

langen Kette von Methodenaufrufen – der Angreifer beliebigen eigenen Code zur Ausführung bringt. Diese Verkettung von Methoden und Verkapselung von Objekten kann mitunter sehr komplex werden.

Deserialisierungsschwachstellen beziehungsweise Angriffe auf diese sind daher illustrativ für viele Aspekte von Hacking-Angriffen, sowohl bezüglich deren Ausführung als auch Schutz gegen selbige. Angriffe sind eine Kombination aus:

- *Kreativität und Out-of-the-Box-Denken:* Angreifer müssen oft Dinge kombinieren oder Probleme lösen, die so noch nicht kombiniert oder gelöst wurden.
- *Fachwissen:* Angreifer müssen die Komplexität, Zusammenhänge und den Unterbau der Technologien, die sie angreifen, verstehen. Sie müssen auch wissen, wie sie Lücken überhaupt entdecken und wie sie diese ausnutzen können.
- *Feinarbeit in Uhrmachermanier:* Angreifer müssen lange Ketten an Objekten und verschränkten Methodenaufrufen konstruieren, um schlussendlich eine Methode zu erreichen, die beliebigen Code auszuführen erlaubt. Dies hat große Ähnlichkeit zu einer Technik aus dem Bereich Binary Exploitation namens „Return-oriented Programming" – ROP beziehungsweise „Jump-oriented Programming" – JOP. In beiden Fällen – Angriffen gegen Deserialisierung und ROP/JOP – spricht man von sogenannten „Gadget Chains".

Illustrativ für die Entwickler- und Verteidigerseite ist, dass Entwickler die von ihnen genutzten Technologien zumindest zu einem gewissen Grad bezüglich der inhärenten Risiken verstehen müssen und dass ein einziger kleiner Fehler im Umgang mit einer riskanten Technologie (in diesem Fall: Deserialisierung von Objekten) zu enormen Risiken führen kann.

```
class Person implements Serializable {
  private int age = 0;
  private String name = "";

  public Person(int age, String name){
    this.age = age;
    this.name = name;
  }
  public int getAge(){
    return this.age;
  }
  public String getName(){
    return this.name;
  }
}
```

Listing 1: Person-Klasse

Rafael Fedler ist seit fast 10 Jahren in der IT-Sicherheit tätig. Seit 2015 prüft und stärkt er als Penetrationstester und Red Teamer die Sicherheit von Unternehmen mit offensiven Methoden. Von 2011 bis 2014 war Rafael Fedler in der IT-Sicherheitsforschung aktiv.
E-Mail: rf@nsideattacklogic.de

Des Weiteren sind Deserialisierungslücken ein Beispiel dafür, dass riskante architektonische Entscheidungen aus der Vergangenheit kaum rückgängig zu machen sind. Sie führen zu Sicherheitsproblemen auf Jahrzehnte. Andere Beispiele hierfür sind die NULL-Terminierung von Strings in C/C++, die Vermengung von Nutzdaten und Code in modernen Rechnerarchitekturen, das inhärente Vertrauen vieler Netzwerkprotokolle, automatische Typecasts bei Loose Comparisons in PHP, das Einführen von automatisch ausgeführten Methoden bei Deserialisierung und viele mehr. Wir Angreifer nennen diese unsicheren Architekturentscheidungen auch „The gift that keeps on giving". Dies ist eine wichtige Lektion für Entwickler und Softwarearchitekten: Sicherheit muss schon in der Architektur-, Konzept- und Designphase bedacht werden. Ansonsten kann man sich Sicherheitsrisiken einhandeln, die man nicht oder nur mit großem Aufwand und Kosten wieder loswird.

Der Autor dieses Artikels hofft, im Folgenden einen Einblick in diese nicht ganz triviale, aber spannende Schwachstellenart geben zu können und gleichzeitig in den „Attacker's Mindset", das heißt, wie Angreifer gedanklich beim Ausnutzen von einigen Schwachstellenklassen und Aufbau von Angriffsketten vorgehen.

(De-)Serialisierung in Java im Schnelldurchlauf

Java ist eine der Sprachen, die bereits von Hause aus einen nativen Mechanismus bereitstellt, um Objekte so, wie sie im Speicher eines Prozesses vorliegen, zu nehmen und in eine Form zu bringen, die eine Speicherung oder Übertragung erlaubt. Dieses Umwandeln von der nativen Speicherform in den „Abschrieb" des Objekts nennt man Serialisierung. Den umgekehrten Prozess, den Abschrieb eines Objekts zu nehmen und ihn zurück in das valide Objekt im Speicher eines laufenden Prozesses zu konstruieren, nennt man Deserialisierung.

Entwickler müssen somit keinen eigenen Code zum Speichern von Objekten auf Festplatte, in der Datenbank (wofür es mittlerweile ORM – Object-Relational Mapping gibt) oder zur Übertragung über das Netzwerk schreiben. Vor allem komplexere Datenstrukturen wieder einzulesen und zu parsen, kann mitunter aufwendig und fehleranfällig sein, weshalb die nativen Java-Methoden eine willkommene Hilfestellung für Entwickler sind.

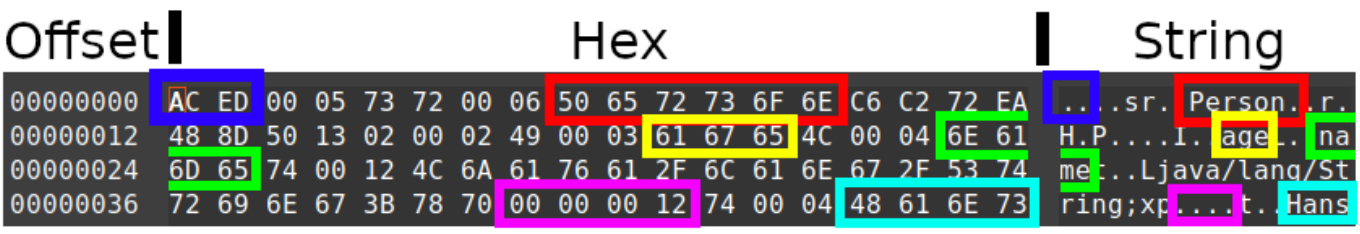

Abb. 1: Serialisiertes Person-Objekt

Nehmen wir als Beispiel für ein serialisiertes Objekt die Klasse `Person` aus Listing 1. Serialisiert nimmt diese Klasse nur einige wenige (72) Byte ein. Dies liegt daran, dass Java beim Serialisieren von Objekten nur ihren *Zustand* speichert, nicht jedoch ihren Code. Es werden somit im Fall einer Instanz der obigen Klasse namens `Person` nur die Werte der Member-Variablen `age` und `name` gespeichert sowie zusätzlich einige Metadaten zur Klasse.

Ein Objekt mit `age = 18` und `name = "Hans"` sieht im Speicher beispielsweise wie in Abbildung 1 aus. Verschiedene Bestandteile sind farblich markiert, sowohl in der Hex- wie auch in der String-Darstellung: Blau (oben links) ist der Header für serialisierte Java-Objekte mit Hex-Wert `ACED`. (Wenn man das Muster `ACED` oder, in Base64 codiert, `rO0` irgendwo findet, ist dies ein guter Indikator für serialisierte Objekte, der einen aufhorchen lassen sollte.) In Rot markiert ist der Klassenname des Objekts (`Person`). Die Markierung für den Namen der ersten Member-Variablen (`age`) ist gelb und die der zweiten (`name`) ist grün. Von letzterer kann man sogar den Objekttyp `java.lang.String` (nicht farblich markiert) erkennen. Der Wert der `age`-Variablen ist in Magenta hervorgehoben (hexadezimal `00 00 00 12` entspricht dezimal dem Wert 18). Zu guter Letzt ist der Wert der `name`-Variable türkis hervorgehoben (`"Hans"`).

Um Objekte zu serialisieren und zu deserialisieren, kann man die Methoden aus Tabelle 1 verwenden. Bei dem Beispielcode möchte ich die Aufmerksamkeit des Lesers vor allem auf zwei Aspekte lenken: Erstens die Methode `ObjectInputStream#readObject()` und zweitens auf die Tatsache, dass diese zuerst aufgerufen wird und der Cast zu einem `Person`-Objekt danach stattfindet.

Angriffsvektor und Sündenfall: Deserialisierungs-Callback-Methoden

Das in Listing 1 erläuterte, triviale Beispiel für ein `Person`-Objekt, welches in Tabelle 1 serialisiert und deserialisiert wird, sieht äußerst ungefährlich aus – und zwar aus gutem Grund, denn das ist es auch. Selbst wenn ein Angreifer in der Lage ist, das serialisierte Objekt vollständig zu manipulieren – zum Beispiel, weil er über das Netzwerk Objekte dieser Klasse schicken oder serialisierte Objekte in der Datenbank oder Festplatte schreiben kann, wo sie von der Anwendung gelesen werden –, wird er mit Objekten der Klasse `Person` keinen Angriff starten können. Erst wenn er es gegen Objekte anderer Klassen austauscht, kann es gefährlich werden.

Gefährlich sind nur Objekte solcher Klassen, die selbst eine `private void readObject(ObjectInputStream stream)`-Methode implementieren und dabei die in `Serializable` definierte Methode überschreiben. Diese wird nämlich wiederum von `ObjectInputStream#readObject()` aufgerufen. Der Nutzen hiervon ist, dass Klassen erlaubt werden soll, eigenes Verhalten oder zusätzliche Schritte durchzuführen, wenn Instanzen ihrer selbst deserialisiert werden. Dies erlaubt Java-Entwicklern bei der Serialisierung und Deserialisierung zwar viele Freiheiten und ist nützlich, um gewisse Automatismen zu implementieren. Gleichzeitig ist dies jedoch auch der Sündenfall der Deserialisierungsschwachstellen: Sobald eine Klasse in der (selbst definierten) `readObject`-Methode Verhalten definiert, welches direkt oder indirekt über Verkettung mit anderen Methoden, die wiederum von der `readObject`-Methode aufgerufen werden, missbraucht werden kann, ist Angreifern Tür und Tor geöffnet.

Leider gilt dies nicht nur für die selbst von Entwicklern in ihrem eigenen Code definierten Klassen, sondern prinzipiell für *alle* in der Anwendung *verfügbaren* Klassen, also auch solche, die in Libraries (z. B. in JARs) definiert wurden. Angreifer können prinzipiell Objekte jeder dieser Klassen konstruieren und einer Schnittstelle, die deserialisiert, übergeben. Warum dies der Fall ist, wird deutlich, wenn man sich das Deserialisierungsbeispiel aus Tabelle 1 noch mal anschaut (zur Erinnerung: `Person hansFromDisk = (Person) in.readObject();`). *Erst* wird `readObject()` aufgerufen, *dann* wird in das Zielobjekt gecastet. Das bedeutet: Die JVM deserialisiert alles, was sie in die Finger kriegt. Der Angreifer muss es nur schaffen, ein serialisiertes Objekt an eine Schnittstelle zu übergeben, die generell deserialisiert. Dann kann er die Schnittstelle mit beliebigen Objekten füttern.

Schritt für Schritt: Verkettung bis zur Arbitrary Code Execution

Doch der Aufruf einer `readObject`-Methode alleine wird dem Angreifer noch nicht helfen. Bei der Deserialisierung von den allermeisten Klassen, die selbst `readObject` implementieren, ist der Aufruf von `readObject` für den Angreifer nutzlos. Stattdessen muss der Angreifer eine Klasse finden und ein Objekt dieser Klasse konstruieren, die selbst die Interface-Methode `readObject` überschreibt *und* in deren eigener Implementierung Methoden anderer Klassen aufgerufen werden. Aber nicht irgendeine Methode einer beliebigen Klas-

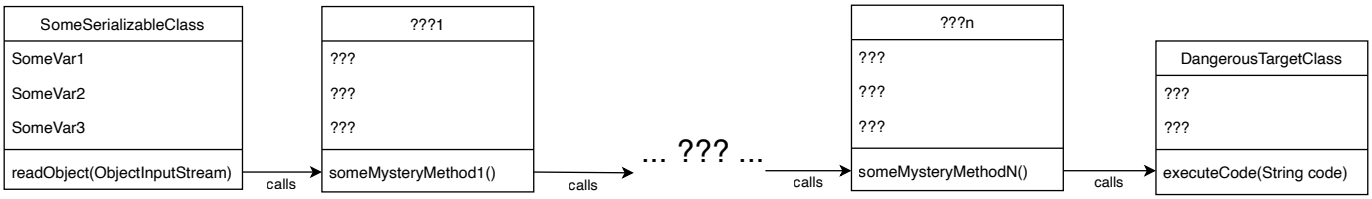

Abb. 2: Der Angreifer muss eine Kette von Klassen finden, um seine „Einstiegsklasse" (links) mit einer Zielklasse, die beliebigen Code ausführen kann (rechts), zu verbinden

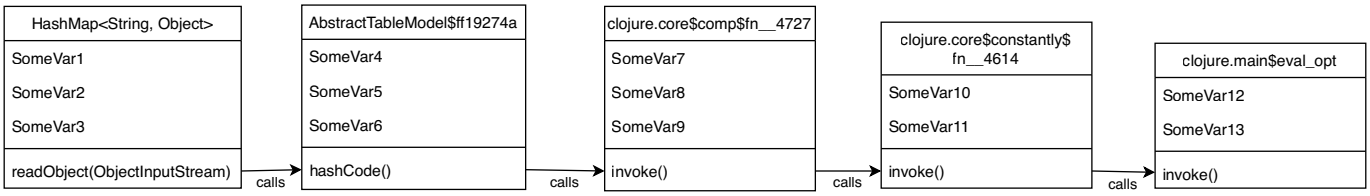

Abb. 3: Gadget Chain, die auf der linken Seite eine HashMap mit einer generellen Clojure-Command-Execution-Funktion auf der rechten Seite über eine Aufrufkette verknüpft, die bei Deserialisierung ausgelöst wird. Quelle: [YSOCLJ]

se, sondern eine Methode, die wiederum eine Methode aufruft, die wiederum eine Methode aufruft … die irgendwann eine Methode aufruft, die es erlaubt, beliebigen Code auszuführen, den der Angreifer in seinem serialisierten Java-Objekt untergebracht hat.

Die Problemstellung, eine solche „Gadget Chain" zu finden, visualisiert Abbildung 2. Wie eingangs erwähnt, hat dies große Ähnlichkeit zu sogenannten ROP Chains, welche beim Angreifen von Pufferüberläufen, Format String-Schwachstellen und anderen Lücken in nativen Binärprogrammen verwendet werden.

Ein Beispiel für eine valide Gadget Chain aus der Praxis zeigt Abbildung 3. Sie macht sich einen Mechanismus in Clojure (`clojure.java.shell`) zunutze, bei dem über einen Code-String wie `"(use '[clojure.java.shell :only [sh]]) (sh %s)"` beliebige Kommandozeilen-Befehle (auszufüllen an Stelle von `%s`) aus Java ausgeführt werden können. Diese Payload wird so in einem Objekt (genau genommen: einer HashMap) untergebracht, dass bei der Deserialisierung der HashMap und der enthaltenen Clojure-Objekte der Befehl über Umwege ausgeführt wird. Diesen Befehl kann der Angreifer nun nutzen, um den Server zu übernehmen, indem er ihn zum Beispiel anweist, einen Trojaner herunterzuladen und auszuführen oder eine Reverse Shell, also eine Kommandozeile, zu öffnen. Bei dieser Gadget Chain handelt es sich übrigens um eine der einfachsten. Viele Gadget Chains bestehen aus 10 bis 15 Objekten und ihren jeweiligen Methoden.

Einige der gefährlichen Objekte mit eigenen Implementierungen von `readObject`, die sich mit anderen Methoden verketten lassen, befinden sich in weit verbreiteten Bibliotheken. Hierzu gehören unter anderem, aber nicht ausschließlich: Apache Commons Collections, Clojure, Groovy, Jython, Spring Core, Hibernate und viele mehr. Auf Grund der weiten Verbreitung dieser Bibliotheken in vielen Anwendungen ist die Wahrscheinlichkeit, dass sich prinzipiell mindestens eine ausnutzbare Klasse finden lässt, relativ groß. Somit führt häufig alleine die Tatsache, dass eine deserialisierende Schnittstelle vorhanden ist, dazu, dass ein Angreifer das entsprechende System übernehmen kann, solange keine Gegenmaßnahmen getroffen wurden.

Angriffsfläche

Um Deserialisierungs-Schwachstellen auszunutzen, muss der Angreifer jedoch in der Lage sein, die von ihm konstruierten, bösartigen serialisierten Java-Objekte irgendwie in die Anwendung hineinzubekommen. Das heißt, er muss es schaffen, dass die Anwendung von irgendwoher sein Objekt liest und danach eine der Java-Deserialisierungsmethoden wie `readObject` aufruft. Üblicherweise gibt es drei Quellen, von denen eine Java-Anwendung serialisierte Objekte entgegennimmt:

- vom Massenspeicher (Festplatte, Netzwerklaufwerk o. Ä.),
- aus der Datenbank (heutzutage sehr selten),
- über das Netzwerk, zum Beispiel als Teil von HTTP-Kommunikation.

Deserialisierungsangriffe über das Netzwerk sind am häufigsten, da es seltener der Fall ist, dass der Angreifer Objekte, die vom Massenspeicher oder aus der Datenbank gelesen werden, komplett in seinem Sinne manipulieren konnte. Dies kann aber auch der Fall sein, vor allem dann, wenn eine lokale Java-Anwendung angegriffen wird statt einer Anwendung, die über das Netzwerk genutzt wird.

Vollendung des Puzzles

Wir haben nun alle Teile beisammen und müssen sie nur noch zusammenfügen. Insgesamt ergeben sich folgende Angriffsschritte:

- Auffinden einer Schnittstelle (meist über das Netzwerk, oft über HTTP), die serialisierte Java-Objekte entgegennimmt.
- Konstruktion einer Gadget Chain (bestehend aus einem Einstiegsobjekt, welches `Serializable` implementiert und `readObject` überschreibt, mehreren Zwischenobjekten und -methoden sowie einer letzten Klasse mit einer Methode, die beliebigen Code ausführt).
- Einbau unserer Payload (z. B. einem Befehl, um einen Trojaner herunterzuladen und auszuführen, oder um uns eine sogenannte Remote Shell, also eine Kommandozeile über das Netzwerk zu verschaffen) für die letzte Methode der Gadget Chain.
- Füttern der deserialisierenden Schnittstelle mit dem von uns konstruierten Objekt, bestehend aus der Gadget Chain und unserer Payload.

Sobald die Ziel-Java-Anwendung unser Objekt zu deserialisieren versucht, kommt unser Code zur Ausführung und wir können das System unter unsere Kontrolle bringen.

Automatisierung der Konstruktion von Angriffspayloads

Wie gezeigt ist das Konstruieren von Gadget Chains und verschachtelten Objekten nicht trivial. Mit dem Tool ysoserial lassen sich öffentlich bekannte Gadget Chains für verschiedene geladene Java-Bibliotheken automatisch konstruieren. Solange die verwundbare Anwendung eine dieser Bibliotheken verwendet, lässt sich ysoserial nutzen, um Payloads zu generieren. Viele der von ysoserial unterstützten Libraries sind weit verbreitet, sodass es nicht unwahrscheinlich ist, dass mindestens einer der Payload-Typen von ysoserial in einer verwundbaren Java-Anwendung funktioniert. Zum aktuellen Zeitpunkt kann ysoserial Payloads in mehr als 30 verschiedene Gadget Chains schnüren. Auch für .NET existiert ein ähnliches Tool mit dem Namen ysoserial.NET.

Weitere Risiken

Bis hierhin haben wir in diesem Artikel Code-Execution-Schwachstellen besprochen, die sich durch die Verkettung von Methodenaufrufen ergeben, welche mit einem initialen Aufruf zu einem Deserialisierungs-Callback starten. Doch auch wenn eine Anwendung sich gegen Code-Execution-Risiken absichert, gibt es noch einige weitere Risiken, die sich bei Verwendung von Serialisierung ergeben können – sofern die Anwendung die serialisierten Objekte an die Außenwelt abgibt und von dieser wieder entgegennimmt. Dies sind im Endeffekt Risiken der Daten- und Kontrollflussmanipulation. Es folgt eine Auswahl von zwei Beispielen aus der Berufspraxis des Autors als Penetrationstester.

Beispiel 1

In einem Penetrationstest eines weit verbreiteten Java-basierten CRM-Systems hat der Autor festgestellt, dass der (ebenfalls Java-basierte) Fat Client desselben CRM-Systems mit dem Server kommuniziert, indem serialisierte Java-Objekte zwischen Client und Server übertragen werden. Das CRM-System sollte unter anderem Datenisolation zwischen verschiedenen Nutzern desselben CRM-Servers umsetzen. Unglücklicherweise wurde diese Isolation allerdings umgesetzt, indem die Isolationsparameter und Primary Keys von Datenbankobjekten (d. h. Zeilen/Einträgen, die wiederum dank ORM als serialisierte Objekte zurückgeschickt wurden), auf die der Client zugreifen wollte, Client-seitig gesetzt wurden und in Request-Objekte verpackt wurden, die serialisiert an den CRM-Server geschickt wurden.

Da der Server somit die Primary Keys und Isolationsparameter für Zugriff auf die Datenbank aus serialisierten Objekten extrahierte, die vom Client kamen, konnte man diese serialisierten Objekte manipulieren: Indem man die Isolationsparameter entfernt hat oder die Primary Keys im serialisierten Objekt auf beliebige Werte gesetzt hat, konnte man auf beliebige Zeilen in der Datenbank zugreifen. Durch Automatisierung des Prozesses war es dem Autor dieses Artikels möglich, im Penetrationstest die gesamte Datenbank der CRM-Installation auszulesen. Da das CRM im Krankenversicherungsbereich eingesetzt werden sollte, wäre der Schaden, falls die Lücke nicht im Vorfeld entdeckt worden wäre, enorm gewesen.

Beispiel 2

In einem Penetrationstest einer Java-basierten Web-Anwendung für einen B2B-Usecase hat die Web-Anwendung als Sitzungscookies serialisierte Java-Objekte verwendet. Diese sahen grob wie folgt aus:

```
class MySessionCookie {
  String sessionId = /* zufälliger String von 32 Byte Länge */;
  boolean isAdmin =  /* true wenn Admin, false für normale Nutzer */;
}
```

Die Lücke ist offensichtlich: Jeder Nutzer der Anwendung konnte das ihm beim Login von der Web-Anwendung ausgestellte Sitzungs-Cookie nehmen und den Boolean-Wert `isAdmin` von `false` auf `true` setzen. Wenn der Nutzer dieses Cookie nun an die Web-Anwendung überträgt, kann der Nutzer Administrator-Funktionalität nutzen und hat somit innerhalb der verwundbaren Anwendung volle und uneingeschränkte Rechte.

Beide genannten Beispiele sind Fälle davon, dass Entwickler Daten vertrauen, die sie über das Netzwerk empfangen – nur, weil sie sie zuvor selbst ausgestellt haben. Sie bedenken nicht, dass diese Daten auch zwischenzeitlich auf Client-Seite manipuliert werden können. Es muss also nicht immer gleich Remote Code Execution sein, um großen Schaden anzurichten.

Weitere betroffene Technologien

Die native Java-Objekt-Deserialisierung ist allerdings nicht die einzige Technologie, die für unsichere Deserialisierung anfällig ist. Ähnliche Probleme betreffen auch verschiedene Bibliotheken für das Serialisieren und Deserialisieren in textbasierte Formate wie JSON, XML und andere. Auch Python Pickles können betroffen sein, genauso wie verschiedene .NET-Methoden. Mehr Informationen hierzu finden sich unter anderem bei der OWASP im Deserialization Cheat Sheet [OWDCS] sowie in den Präsentationen „Friday the 13th JSON Attacks“, gehalten auf der Black Hat USA-Konferenz 2017 [BHU17], und „Marshalling Pickles“, gehalten auf der AppSecCali-Konferenz 2015 [ASC15].

Schutz

Wie bei vielen Technologien, die konzeptionell unsicher sind, ist der beste Schutz, diese einfach nicht einzusetzen. Oder, wenn diese eingesetzt werden sollen, nur so, dass niemals Daten „von außen” entgegengenommen werden, das heißt, dass alle serialisierten Objekte sich nur anwendungsintern oder lokal auf dem System bleiben und sich niemals an Orten befinden, wo Nutzer oder dritte Systeme diese ändern oder selbst kreieren können.

Falls dies nicht möglich sein sollte, kann man Risiken von Deserialisierungsschwachstellen auf anderem Wege entgegenwirken. Hierzu lassen sich verschiedene Java-Methoden überschreiben, wie `ObjectInputStream#resolveClass()`. Noch besser ist jedoch, dies mit Bibliotheken umzusetzen, zum Beispiel SerialKiller [SRLKLR]. Die `ValidatingObjectInputStream`-Klasse der Apache Commons IO-Bibliothek [APCMS] ermöglicht ebenfalls, erlaubte Klassentypen zu spezifizieren; alle anderen lehnt sie ab und deserialisiert diese nicht.

Bei den im vorherigen Abschnitt angesprochenen anderen potenziell anfälligen Technologien wie JSON- und XML-Deserialisierung müssen die Sicherheitshinweise der jeweiligen Bibliotheken beachtet und unsichere Mechanismen wie Polymorphismus oder automatische Typ-Ermittlung vermieden werden. Weitere Informationen zum Schutz finden sich bei der OWASP im Deserialization Cheat Sheet [OWDCS].

Schlusswort

Deserialisierungsschwachstellen sind ein illustratives Beispiel für wichtige Aspekte der technischen IT-Sicherheit: Sowohl für die Probleme der Verteidigerseite wie auch die Anforderungen und Vorgehensweise der Angreiferseite. Sie zeigen, wie wichtig Verständnis für beide Parteien ist: Verständnis der Risiken der verwendeten Technologien und dass man weiß, was man tut. Es zeigt auch, dass IT-Sicherheit ein teils komplexes Problem ist, und dass teils sehr mächtige Angriffe durch sprichwörtliche winzige Nadelöhre ihren Weg in verwundbare Systeme finden und diese vollständig übernehmen können. Ebenfalls zeigt sich, wie Design- und Architekturentscheidungen auf Jahre große Auswirkungen auf die Sicherheit aller abhängigen Komponenten haben. Zuletzt hofft der Autor, dass der Einblick in die Vorgehensweise beim Ausnutzen dieser Schwachstellentypen interessant war und einen Einblick in die Angreiferperspektive geben konnte.

Literatur und Links

[ASC15] Marshalling Pickles, AppSecCali 2015, https://frohoff.github.io/appseccali-marshalling-pickles/

[APCMS] Apache Commons IO: Bibliothek für verschiedene IO-Funktionen, u. a. sichere Deserialisierung, https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/serialization/ValidatingObjectInputStream.html

[BHU17] Friday the 13th JSON Attacks, Black Hat USA 2017: https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-p.pdf

[Fed21] R. Fedler, Die OWASP Top 10 in Java-Anwendungen, in: JavaSPEKTRUM, 5/2021

[OWDCS] OWASP Deserialization Cheat Sheet, https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

[OWT10] OWASP Web Top 10 (2017), https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010-2017%20(en).pdf

[SRLKLR] SerialKiller: Bibliothek zum Verhindern von Angriffen gegen Deserialisierung, https://github.com/ikkisoft/SerialKiller

[YSOCLJ] Clojure Gadget Chain in ysoserial, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure.java

[YSOSRL] ysoserial: Tool zum Bündeln von Angriffs-Payloads in Gadget Chains, https://github.com/frohoff/ysoserial