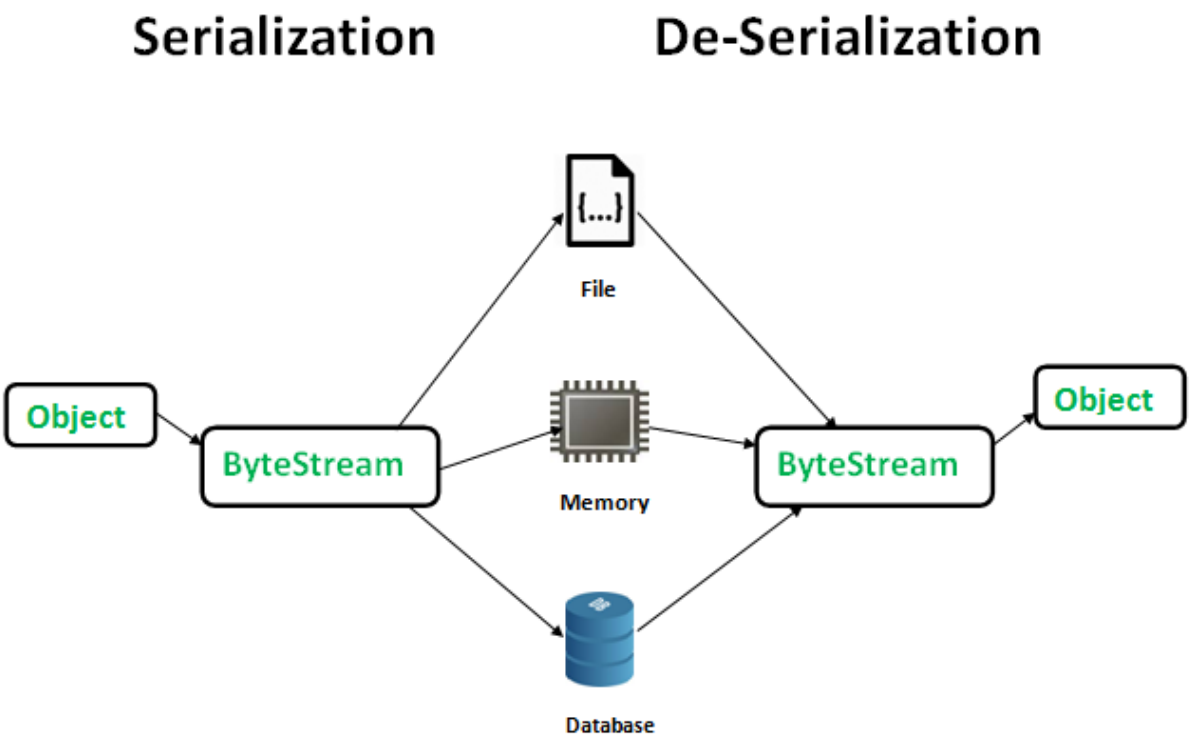
아이템 85 자바 직렬화의 대안을 찾으라

자바 직렬화

- 객체를 **메모리, 데이터베이스** 혹은 **파일**로 옮길때 사용
- Serialization: 객체를 바이트 스트림으로 바꾸는것

  (= 객체에 저장된 데이터를 스트림에 쓰기-write 위해 연속적인-serial 데이터로 변환하는 것)
- 직렬화의 주 목적: 객체를 상태 그대로 저장하고 필요할 때 다시 생성하여 사용하는 것
- 직렬화 하면서 생긴 **바이트 스트림은 플랫폼에 독립적이다**

  ⇒ 직렬화된 객체는 다른 플랫폼에서 역직렬화가 가능하다



https://www.geeksforgeeks.org/serialization-in-java/

- 역직렬화: 네트워크나 영구저장소에서 바이트 스트림을 가져와서 객체가 저장되었던 상태로 변환하는 것
- 직렬화 도입: 97년 자바에서 처음으로 직렬화 도입
    - 처음에는 매력적인 기능같았으나, 보이지 않는 생성자, API와 구현 사이의 모호해진 경계, 잠재적인 정확성 문제, 성능, 보안, 유지보수성 등 **문제점**이 많았다

직렬화의 취약점

직렬화의 근본적인 문제: 공격 범위가 너무 넓고 지속적으로 더 넓어져 방어하기 어렵다

- ex) 샌프란시스코 교통국 랜섬웨어 공격
- **ObjectInputStream 의 readObject** 메서드가 객체 그래프를 **역직렬화** 하면서 공격 범위가 넓어지게 된다.

  readObject 메서드

    - (Serializable를 구현한) classpath 안의 거의 모든 타입의 객체를 만들어 낼 수 있는 `마법같은` 생성자
    - 바이트 스트림을 역직렬화 하는 과정에서 readObject가 그 타입들 안의 모든 코드를 수행할 수 있다 ⇒ 타입들의 코드 전체가 공격 범위가 된다

▼ 직렬화 문제점의 위험

- 자바 표준 라이브러리나 아파치 커먼즈 컬렉션 같은 서드 파티 라이브러리는 물론 애플리케이션 자신의 클래스들도 공격 범위에 포함된다. 관련한 모든 모범 사례를 따르고, 모든 직렬화 가능 클래스들을 공격에 대비하도록 작성해도 여전히 취약할 수 있다.
- 자바의 역직렬화는 명백하고 현존하는 위험이다. 이 기술은 지금도 애플리케이션에서 직접 혹은, 자바 하부 시스템(RMI, JMX, JMS같은)을 통해 간접적으로 쓰이고 있기 때문이다. 신뢰할 수 없는 스트림을 역직렬화하면 원격 코드 실행(RCE), 서비스 거부(Dos) 등의 공격으로 이어질 수 있다. 잘못한게 아무것도 없는 애플리케이션이라도 이런 공격에 취약해 질 수 있다. -CERT 조정센터 기술관리자(로버트 시커드)

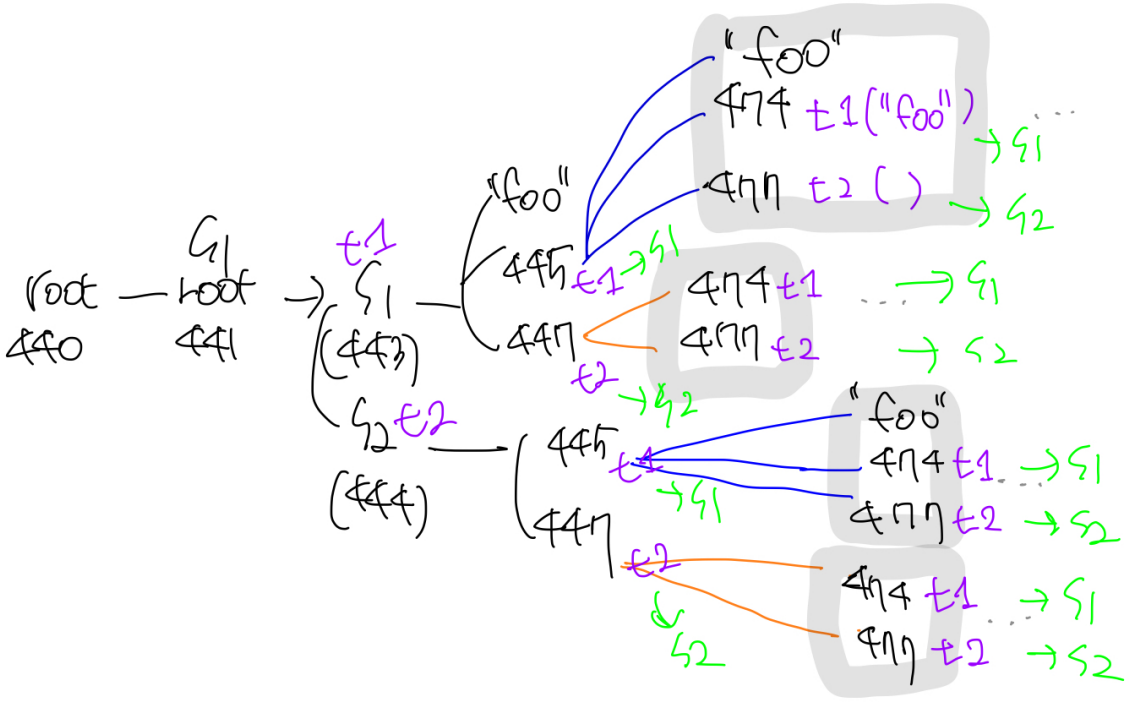
아주 신중하게 제작한 바이트 스트림만 역직렬화 해야한다.

1. 가끔 공격자가 기반 하드웨어의 네이티브 코드를 마음대로 실행할 수 있는 강력한 `가젯 체인` (여럿 가젯을 함께 사용하는 것) 이 발견되기도 하기 때문이다.
    - `가젯` : 역직렬화 과정에서 호출되어 잠재적으로 위험한 동작을 수행하는 메서드들
2. 역직렬화에 시간이 오래 걸리는 짧은 스트림(`역직렬화 폭탄`)을 역직렬화하는 것만으로 서비스 거부 공격에 쉽게 노출될 수 있다.

```
public class DeserializationBomb {
    public static void main(String[] args) throws Exception {
        System.out.println(bomb().length);
        deserialize(bomb());
    }

    static byte[] bomb() {
        Set<Object> root = new HashSet<>();
        Set<Object> s1 = root;
        Set<Object> s2 = new HashSet<>();
        for (int i = 0; i < 100; i++) {
            Set<Object> t1 = new HashSet<>();
            Set<Object> t2 = new HashSet<>();
            t1.add("foo"); //t1을 t2와 다르게 만든다.
            s1.add(t1); // s1: {{"foo"}}
            s1.add(t2); // s1: {{},{"foo"}}
            s2.add(t1); // s2: {{"foo"}}
            s2.add(t2); // s2: {{},{"foo"}}
            s1 = t1; // s1: {"foo"}
            s2 = t2; // s2: {}
        }
        return serialize(root);
    }
}
```

- 문제는 HashSet 인스턴스를 역직렬화하려면 그 원소들의 해시코드를 계산해야 한다는 데 있다.



- 루트 HashSet에 담긴 두 원소는 각각 다른 HashSet 2개씩을 원소로 갖는 HashSet이다.
- 반복문에 의해 이 구조가 깊이 100단계까지 만들어진다. 따라서 이 HashSet을 역직렬화하려면 hashCode 메서드를 2^100번 넘게 호출해야한다. 역직렬화가 영원히 계속된다는 것도 문제지만, 뭔가 잘못됐다는 신호조차 주지 않는다는 것도 큰 문제다. 이코드는 단 몇 개의 객체만 생성해도 스택 깊이 제한에 걸려버린다.

대처법

1. 직렬화 위험을 회피하는 가장 좋은 방법은 **아무것도 역직렬화하지 않는 것**이다.

- 크로스-플랫폼 구조화된 데이터 표현 : 객체와 바이트 시퀀스를 변환해주는 매커니즘.
  자바 직렬화의 여러 위험을 회피하면서 다양한 플랫폼 지원, 우수한 성능, 풍부한 지원 도구, 활발한 커뮤니티와 전문가 집단 등 수많은 이점을 제공한다.
  - 자바 직렬화보다 훨씬 간단하다
  - 임의 객체 그래프를 자동으로 직렬화/역직렬화하지 않는다
  - 속성-값 쌍의집합으로 구성하고 간단하고 구조화된 데이터 객체를 사용한다.
  - 기본 타입 몇개와 배열 타입만 지원한다.
  - 대표적으로 JSON, 프로토콜 버퍼(protobuf)

JSON

- 원래 자바스크립트용, 브라우저와 서버의 통신용으로 설계 됨
- 텍스트 기반이라 사람이 읽을 수 있다
- 데이터를 표현하는데 쓴다

Protobuf

- 원래 C++용, 서버 사이에 데이터를 교환하고 저장하기 위해 설계 됨
- 이진 표현이라 효율이 훨씬 높다
- 문서를 위한 스키마(타입)을 제공하고 올바로 쓰도록 강요한다
- 사람이 읽을 수 있는 텍스트 표현(pbtxt) 지원

1. 자바 직렬화를 완전히 배제할 수 없는 레거시의 경우 → **신뢰할 수 없는 데이터는 절대 역직렬화 하지 않는다**.
2. 직렬화를 피할 수 없고, 역직렬화한 데이터가 안전한지 완전히 확실할 수 없을 때 → **객체 역직렬화 필터링**을 사용하자(자바9 에서 추가됨)

- 데이터 스트림이 역직렬화되기전에 **필터**를 설치하는 기능
- 클래스 단위로 특정 클래스를 수용, 거부 할 수 있다

▼ example

```
public class SerializationFilter {
    public static void main(String[] args) throws Exception {

        //Serialization
        Employee emp = new Employee("Test Name", "Test Title");
        String filename = "employee.ser";
        FileOutputStream filewrite = new FileOutputStream(filename);
        ObjectOutputStream out = new ObjectOutputStream(filewrite);
        out.writeObject(emp);
        out.close();
        filewrite.close();

        //Deserialization
        FileInputStream fileread = new FileInputStream(filename);
        ObjectInputStream in = new ObjectInputStream(fileread);

        // 개별 스트림에 대한 사용장 정의 필터 설정 (람다식 사용)
        in.setObjectInputFilter(info -> ((info.serialClass() != null)
                  && info.serialClass().getName().equals("com.example.Employee"))
              ? ObjectInputFilter.Status.REJECTED : ObjectInputFilter.Status.UNDECIDED);
        Employee empread = (Employee) in.readObject();
```

```
        System.out.println(empread.name); // Test Name
        System.out.println(empread.title); // Test Title

        in.close();
        fileread.close();
    }
}
```

- 기본 수용 모드 : 블랙리스트에 기록된 잠재적으로 위험한 클래스를 거부
- 기본 거부 모드 : 화이트리스트에 기록된 안전한 클래스만 수용

```
ObjectInputFilter filesOnlyFilter = ObjectInputFilter.Config
          .createFilter("example.File;!example.Directory");
```

- 블랙리스트 방식보다 화이트리스트 방식을 추천
  - 블랙리스트 방식은 이미 알려진 위험으로부터만 보호할 수 있기 때문

- SWAT이라는 도구로 화이트 리스트를 자동으로 생성할 수 있다
- 필터링 기능은 메모리를 과하게 사용하거나 객체 그래프가 너무 깊어지는 사태로부터 보호한다.
  - 하지만 직렬화 폭탄은 걸러내지 못한다.

핵심 정리

- 직렬화는 위험하니 피해야 한다.
- 시스템 밑바닥 부터 설계한다면 JSON이나 프로토콜 버퍼 같은 대안을 사용하자.
- 신뢰할 수 없는 데이터는 역직렬화하지 말자.
- 꼭 해야한다면 객체 역직렬화 필터링을 사용하되 공격을 모두 막아줄 수는 없음을 기억해야한다.
- 클래스가 직렬화를 지원하도록 만들지 말고, 꼭 그렇게 만들어야 한다면 정말 신경써서 작성해야 한다.