UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO

Bernardo Melo

De objetos à problemas: Uma análise sobre a desserialização insegura de objetos na linguagem Java

Recife

2025

Bernardo Melo

De objetos à problemas: Uma análise sobre a desserialização insegura de objetos na linguagem Java

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação

Área de Concentração: Segurança da Informação

Orientador (a): Cleber Zanchettin

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Melo, Bernardo Sarinho Galvão Aureliano de.
De objetos à problemas: Uma análise sobre a desserialização insegura de objetos na linguagem Java / Bernardo Sarinho Galvão Aureliano de Melo. - Recife, 2025.
36 p.

Orientador(a): Cleber Zanchettin
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.

1. Segurança da Informação. 2. Desserialização Insegura. 3. Aplicações Web. 4. Vulnerabilidades em software. I. Zanchettin, Cleber. (Orientação). II. Título.

000 CDD (22.ed.)

Bernardo Melo

De objetos à problemas: Uma análise sobre a desserialização insegura de objetos na linguagem Java

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação

Aprovado em: 11/08/2025

BANCA EXAMINADORA

_____

Prof. Dr. Cleber Zanchettin (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Marcio Lopes Cornelio (Examinador Interno)

Universidade Federal de Pernambuco

AGRADECIMENTOS

Primeiramente gostaria de agradecer à minha família, por todo o apoio e suporte que sempre recebi. Sem eles eu com certeza não teria chegado até aqui, e por isso serei eternamente grato.

Gostaria também de agradecer aos professores do Centro de Informática que, sempre solícitos, me guiaram e me ajudaram a conhecer um pouco mais sobre os computadores que sempre me cercavam. Não posso deixar de mencionar meu orientador, Cleber Zanchettin, que teve a boa vontade e a paciência de me ajudar na criação deste trabalho.

Agradeço ao pessoal da Tempest Security Intelligence, onde aprendi praticamente tudo que sei sobre segurança da informação. Agradeço também aos amigos que fiz ao longo da minha jornada de graduação, dentro e fora do Centro de Informática. Em especial, agradeço a Rafael, Ricardo e Luis, que sempre me encorajaram a terminar este trabalho, a Lucas, que nunca me deixou esquecer o que era mais importante durante os momentos onde não conseguia enxergar com clareza.

Por último, agradeço aos que me fizeram companhia durante meu tempo no CIn. Os nomes são muitos e não conseguirei lembrar de todos, mas tenho muito carinho por cada um de vocês.

RESUMO

Não é exagero afirmar que o mundo atual é mediado por *software*. De aplicações financeiras que movimentam quantias bilionárias diariamente a dispositivos IoT e aparelhos celulares, o *software* está em toda parte. Devido a isso, falhas de segurança descobertas nos mais variados tipos de *software* podem afetar milhões de indivíduos diferentes, impactando fortemente suas vidas. Dentre as falhas de segurança existentes, um grupo de vulnerabilidades particularmente críticas são os problemas de desserialização insegura, que por natureza tendem a causar alto impacto, muitas vezes possibilitando eventuais atacantes a executar código arbitrário dentro do contexto do sistema comprometido. Este trabalho tem como objetivo abordar o funcionamento das vulnerabilidades de desserialização insegura que ocorrem na linguagem Java, com um enfoque particular no contexto de aplicações *web*. Dessa forma, através de uma análise teórica e prática, este projeto não somente detalha com profundidade conceitos relacionados ao funcionamento do processo de serialização e desserialização de objetos dentro da linguagem, mas também aborda como esta vulnerabilidade tende a ocorrer, incluindo cenários reais onde ela foi identificada ou explorada, bem como boas práticas e medidas preventivas que devem ser tomadas para impedir a ocorrência e exploração desta vulnerabilidade.

Palavras-chaves: segurança da Informação; desserialização insegura; Java; aplicações web; vulnerabilidades em software.

ABSTRACT

It is not an exaggeration to state that modern society is mediated by software. From financial applications handling billions of dollars daily to IoT devices and mobile phones, software permeates virtually every aspect of contemporary life. Consequently, security vulnerabilities discovered in various types of software can potentially affect millions of individuals, exerting significant impacts on their lives. Among these vulnerabilities, insecure deserialization represents a particularly critical class, as it inherently tends to have high severity, often enabling attackers to execute arbitrary code within the context of a compromised system. This thesis aims to examine the nature and functioning of insecure deserialization vulnerabilities within the Java programming language, with particular emphasis on their implications in the context of web applications. Through both theoretical and practical analysis, this work offers a comprehensive exploration of the serialization and deserialization processes in Java, elucidates the conditions under which such vulnerabilities arise, presents real-world exploitation scenarios, and outlines best practices and preventive measures necessary to mitigate associated risks.

Keywords: information security; insecure deserialization; Java; web applications; software vulnerabilities.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

super

SUMÁRIO

1 INTRODUÇÃO

Este capítulo tem como objetivo dar uma visão geral sobre o problema a ser abordado, bem como enumerar os objetivos que serão abordados ao longo do trabalho.

1.1 CONTEXTO E MOTIVAÇÃO

Como consequência do avanço tecnológico e do aumento da demanda e complexidade geral dos *softwares* desenvolvidos nos dias de hoje, nota-se uma rápida extensão na superfície de ataque de tais aplicações, que muitas vezes são compostas de uma combinação entre trechos de código proprietário, código aberto e uma variedade de dependências.

Dentre os tipos de vulnerabilidades existentes, problemas ligados ao tratamento de dados de entrada estão entre os mais comuns, e são altamente visados por eventuais atacantes. Dentro desse contexto, as vulnerabilidades de desserialização insegura requerem atenção especial, pois, embora já sejam amplamente conhecidas nos dias de hoje, destacam-se pela sua natureza crítica e dificuldade de correção em muitos casos.

Tais vulnerabilidades são especialmente perigosas pelo motivo de, em sua maioria, possibilitarem a execução remota de código quando exploradas, através da injeção de um objeto malicioso que, ao ser desserializado, executará comandos arbitrários no servidor vulnerável, resultando em uma das formas mais severas de comprometimento de um sistema.

Devido à sua criticidade, falhas de desserialização são amplamente reconhecidas no contexto de aplicações *web*, inclusive por organizações como a OWASP (*Open Worldwide Application Security Project*), que menciona a desserialização insegura como uma das principais vulnerabilidades existentes nas aplicações desta categoria, incluindo-a como uma das vulnerabilidades abordadas no OWASP Top 10 desde a sua edição de 2017 (OWASP, 2017)

Outro fator bastante preocupante é a dificuldade de detecção da vulnerabilidade durante as etapas de desenvolvimento de um dado sistema, já que o comportamento malicioso geralmente depende de condições específicas e cadeias complexas de objetos sendo interpretadas em tempo de execução para ser reproduzido. Devido a isso, ataques explorando falhas desse tipo já foram identificados em diversos sistemas corporativos, levando a consequências sérias, como a perda e vazamento de dados e o comprometimento de servidores.

1.2 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo geral analisar e demonstrar o funcionamento, os riscos e o impacto das vulnerabilidades de desserialização insegura, com enfoque principal nos casos relacionados à linguagem Java, uma das mais utilizadas na construção de aplicações *web*. Com o intuito de atingir tal objetivo, o trabalho em questão conta com os seguintes objetivos específicos:

- Apresentar os conceitos fundamentais de serialização e desserialização de objetos, explicando seus propósitos, seu funcionamento interno e o papel que desempenham em aplicações;
- Explicar a vulnerabilidade de desserialização insegura, destacando sua natureza, as condições em que ocorre e os cenários típicos de exploração realizados por atacantes no mundo real;
- Analisar, mostrando detalhes de implementação, o funcionamento dos processos de serialização e desserialização em Java, além das formas como problemas de desserialização insegura geralmente ocorrem na linguagem. Esta etapa inclui ainda exemplos reais de ataques registrados no ecossistema Java, ilustrando os potenciais impactos causados pela exploração da vulnerabilidade;
- Demonstrar, dentro de um ambiente controlado, como a vulnerabilidade é explorada, considerando tanto pesquisas acadêmicas quanto experimentos conduzidos por profissionais da área de segurança;
- Apresentar e discutir estratégias de mitigação do problema, abordando métodos e estratégias de correção comumente aplicados para aumentar a robustez dos ambientes e aplicações em relação à vulnerabilidade.

Com essa abordagem, o trabalho busca não somente esclarecer os aspectos técnicos da desserialização insegura, mas também pretende contribuir com a conscientização sobre sua relevância e discutir práticas para prevenção, conhecimento este extremamente útil tanto para desenvolvedores quanto para profissionais da área de segurança da informação.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo principal introduzir o leitor aos conceitos de serialização e desserialização de objetos, bem como explicar, de maneira geral, como a vulnerabilidade de desserialização insegura funciona. Isto é necessário para que, no capítulo seguinte, seja possível abordar com mais detalhes como as ideias discutidas aqui se aplicam dentro do contexto da linguagem Java.

2.1 SERIALIZAÇÃO E DESSERIALIZAÇÃO DE OBJETOS

Para a compreensão da vulnerabilidade de desserialização insegura, é necessário entender dois conceitos fundamentais: a serialização e a desserialização de objetos. A serialização é o processo de conversão de um objeto (que por sua vez existe em um dado programa em tempo de execução) em um formato que possibilite a este ser armazenado ou transmitido, como uma sequência de *bytes* (também chamada de *stream*). Esse processo é fundamental para a persistência de dados e para a comunicação entre diferentes componentes de um sistema, especialmente em arquiteturas distribuídas. Uma vez serializado, o objeto tem seu estado salvo e pode ser, por exemplo, gravado em disco ou enviado através da rede.

Já a desserialização, por sua vez, é o processo inverso: consiste em reconstruir, a partir da sequência de *bytes* gerada durante a serialização, uma instância do objeto original na memória, isto é, dentro de um programa. Esse mecanismo permite restaurar o estado de um objeto previamente serializado e é amplamente utilizado em *frameworks* de comunicação remota, sistemas de mensagens, persistência de dados, entre outros. A figura 1 ilustra, de maneira simplificada, a mudança de estados de um dado objeto ao passar pelos processos de serialização e desserialização:

Figura 1 – Esquema simplificado representando o processo de serialização e desserialização de um objeto

Ambos os processos mencionados acima são amplamente utilizados em linguagens de programação modernas, como PHP, Python e Java. Cada linguagem oferece mecanismos nativos

ou bibliotecas específicas para serializar objetos em formatos binários ou textuais. Fora disso, padrões de serialização independentes da linguagem, como JSON ou XML também existem, e permitem que diferentes tecnologias se comuniquem com facilidade. Este formato é bastante utilizado em cenários como arquiteturas baseadas em microsserviços e APIs RESTful, além da troca de mensagens de maneira assíncrona.

Contudo, embora estas sejam práticas consolidadas e essenciais em vários contextos, existem riscos associados aos processos de serialização e desserialização. Quando a entrada a ser desserializada provém de fontes externas ou não confiáveis, o processo de desserialização pode abrir brechas de segurança consideráveis, especialmente no caso em que os mecanismos de validação e controle não estejam devidamente implementados. Em tais cenários, a confiança implícita nos dados recebidos pode ser explorada por atacantes para introduzir objetos maliciosos no sistema receptor. Essa condição leva ao que se conhece como desserialização insegura, uma vulnerabilidade explorada em diversos ambientes e aplicações, e que muitas vezes possui potencial para causar consequências catastróficas.

2.2 A VULNERABILIDADE DE DESSERIALIZAÇÃO INSEGURA

Embora o processo de desserialização tenha papel essencial na transmissão de dados em aplicações nos dias de hoje, como mencionado previamente, seu uso incorreto pode representar sérios riscos à segurança de sistemas. A desserialização insegura ocorre quando dados originados de uma fonte externa são convertidos diretamente em objetos em memória na aplicação, sem a realização de devida verificação prévia. Nesses casos, um atacante pode construir manualmente uma estrutura de dados aparentemente legítima que, ao ser desserializada, desencadeia comportamentos que podem acabar por comprometer a integridade, confidencialidade ou a disponibilidade do sistema.

A vulnerabilidade está associada ao fato de que, durante a desserialização, a aplicação muitas vezes executa automaticamente certos métodos internos dos objetos a serem desserializados, como `readObject()` no caso de Java ou `__reduce__()` em Python. É possível, então, fazer uso de um objeto especialmente construído para se aproveitar deste comportamento e executar código arbitrário ou manipular partes da lógica de execução da aplicação.

Um dos mecanismos comuns de ataque é o uso de *payloads* de desserialização, também conhecidos como *gadget chains*: cadeias de objetos serializados que foram criadas com a intenção explícita de explorar a aplicação que irá recebê-las. Esses *payloads* podem incluir

classes da própria linguagem ou de bibliotecas de terceiros que, quando instanciadas, executam código perigoso. Exemplos seriam comandos no sistema operacional, conexões de rede ou manipulações de arquivos. Ferramentas como o ysoserial (FROHOFF, 2014) (para cenários em Java) automatizam a geração de tais *payloads* com base em bibliotecas vulneráveis já conhecidas, o que facilita ainda mais a exploração da falha.

Por se tratar de uma vulnerabilidade de alto impacto, casos reais de exploração de falhas de desserialização insegura tendem a ser conhecidos no mercado. Por exemplo, em 2015, a biblioteca *Apache Commons Collections*, amplamente utilizada no ecossistema Java, foi explorada em diversos ataques de execução remota de código, culminando no registro de vulnerabilidades como o CVE-2015-4852 (NIST, 2015). A exploração desta falha permitia que atacantes fossem capazes de executar comandos arbitrários sem autenticação em servidores vulneráveis, através do envio de um objeto serializado malicioso. Na época, a repercussão desse incidente foi tão grande que levou à criação de diversas ferramentas de ataque e defesa voltadas especificamente à análise de fluxos de serialização.

Dessa forma, a desserialização insegura representa uma ameaça séria atualmente, pois explora uma *feature* legítima presente em uma variedade de linguagens de programação existentes, aproveitando-se da confiança implícita que muitas vezes é depositada nos dados a serem desserializados. Este risco ainda é amplificado caso consideremos a dificuldade de detecção desse tipo de falha, que, em muitos casos, permite a exploração sem a necessidade de autenticação prévia.

3 DESSERIALIZAÇÃO INSEGURA EM JAVA

Este capítulo tem como objetivo expandir os conceitos introduzidos no capítulo anterior, aplicando-os dentro do contexto da linguagem de programação Java. Além de detalhar como os processos de serialização e desserialização em Java funcionam, o capítulo também se aprofunda no tópico central do trabalho: a desserialização insegura de objetos em Java.

3.1 SERIALIZAÇÃO E DESSERIALIZAÇÃO EM JAVA

No ecossistema Java, a serialização é utilizada com frequência, e possui aplicações diferentes a depender do cenário onde é empregada. Dentre as aplicações comuns, destacam-se o uso na transferência de dados via requisições HTTP (em parâmetros, *cookies*, etc.), o uso na comunicação entre processos (via tecnologias como o Java *Remote Method Invocation*), e o armazenamento em bancos de dados e *file systems*.

A linguagem provê suporte nativo à serialização através de duas interfaces:

- **java.io.Serializable**: Interface mais comumente utilizada, é o mecanismo padrão de serialização do Java. Ela também não possui métodos ou constantes (e por isso é considerada uma interface de marcação).

Código Fonte 1 – Assinatura da interface Serializable

```
package java.io;

public interface Serializable {
}
```

- **java.io.Externalizable**: Uma alternativa mais manual e explícita, onde o desenvolvedor assume o controle da lógica de desserialização. Sua assinatura contém dois métodos que precisam ser implementados e serão chamados durante o processo: `readExternal()` e `writeExternal()`.

Código Fonte 2 – Assinatura da interface Externalizable

```
package java.io;

public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
    void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException;
}
```

O processo de serialização e desserialização em si ocorre através de chamadas a dois métodos, que agem como o mecanismo padrão para serializar e desserializar objetos:

Código Fonte 3 – Assinatura do método ObjectOutputStream.writeObject()

```
java.io.ObjectOutputStream.writeObject(Objeto):

public final void writeObject(Object obj) throws IOException
```

Código Fonte 4 – Assinatura do método ObjectInputStream.readObject()

```
 java.io.ObjectInputStream.readObject():

public final Object readObject() throws IOException, ClassNotFoundException
```

Quando uma classe implementa a interface *Serializable* e não define seus próprios métodos `writeObject()` e `readObject()`, o processo de serialização e desserizaliação ocorrem de maneira automática. Contudo, caso desejado, é possível inserir lógica adicional nestes processos ao implementar os métodos `writeObject()` e `readObject()` nas classes a serem serializadas, utilizando as assinaturas abaixo:

Código Fonte 5 – Assinatura do método writeObject()

```
private void writeObject(ObjectOutputStream out) throws IOException
```

Código Fonte 6 – Assinatura do método readObject()

```
private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException
```

Os métodos dos trechos de código 5 e 6 são muitas vezes chamados de *Magic Methods*, e fazem parte de um conjunto de métodos que são invocados dentro das chamadas feitas aos

métodos `java.io.ObjectOutputStream.writeObject()` e `java.io.ObjectInputStream.readObject()` quando um objeto é serializado ou restaurado, permitindo com que desenvolvedores sejam capazes de sobrescrever o comportamento padrão dos processos. Vale ressaltar que esses métodos, embora tenham o mesmo nome, tratam-se de métodos distintos, como é observável a partir de suas assinaturas.

3.2 SERIALIZAÇÃO DE DADOS EM JAVA: UM EXEMPLO PRÁTICO

Esta seção tem como objetivo exemplificar o processo de serialização, mostrando como um objeto é serializado, sua representação enquanto uma sequência de *bytes*, e como este é restaurado posteriormente.

3.2.1 Serializando um objeto de uma Classe

Tomemos como exemplo a classe `Pessoa`, que implementa a interface `Serializable`, como mostrado no trecho de código 7:

Código Fonte 7 – Classe Pessoa

```
public class Pessoa implements Serializable {

    private static final long serialVersionUID = 1;

    String nomeCompleto;
    int idade;
    String CPF;


}
```

Supõe-se então um cenário onde um objeto da classe acima assume os valores mostrados abaixo:

```
nomeCompleto = “João Silva”;
idade = 30;
CPF = “01234567890”
```

Caso exista a necessidade de serializar este objeto, isto deverá ser realizado através de uma chamada ao método `java.io.ObjectOutputStream.writeObject()`, como discutido previamente. Para isso, é necessário antes instanciar um novo objeto do tipo `ObjectOutputStream` a partir de uma `FileOutputStream`, como mostrado no trecho de código 8:

Código Fonte 8 – Instanciando e serializando um objeto da classe Pessoa

```
//

//Objeto da classe Pessoa sendo instanciado
Pessoa pessoa1 = new Pessoa();
        pessoa1.nomeCompleto = "Joao Silva";
        pessoa1.idade = 30;
        pessoa1.CPF = "01234567890";



        FileOutputStream fileStream  = new FileOutputStream("/tmp/
            output_serialization");
        ObjectOutputStream outputStream = new ObjectOutputStream(fileStream);
        outputStream.writeObject(pessoa1);

//
```

3.2.2 Analisando a Stream de bytes resultante

Como resultado do processo mostrado, o arquivo indicado pelo `FileOutputStream` contém o objeto `pessoa1` serializado. A figura 2 ilustra a *stream* de dados resultante do processo:

```
00000000  ac ed 00 05 73 72 00 1c  65 78 65 6d 70 6c 6f 5f  |....sr..exemplo_|
00000010  73 65 72 69 61 6c 69 7a  61 74 69 6f 6e 2e 50 65  |serialization.Pe|
00000020  73 73 6f 61 00 00 00 00  00 00 00 01 02 00 03 49  |ssoa...........I|
00000030  00 05 69 64 61 64 65 4c  00 03 43 50 46 74 00 12  |..idadeL..CPFt..|
00000040  4c 6a 61 76 61 2f 6c 61  6e 67 2f 53 74 72 69 6e  |Ljava/lang/Strin|
00000050  67 3b 4c 00 0c 6e 6f 6d  65 43 6f 6d 70 6c 65 74  |g;L..nomeComplet|
00000060  6f 71 00 7e 00 01 78 70  00 00 00 1e 74 00 0b 30  |oq.~..xp....t..0|
00000070  31 32 33 34 35 36 37 38  39 30 74 00 0b 4a 6f c3  |1234567890t..Jo.|
00000080  a3 6f 20 53 69 6c 76 61                           |.o Silva|
00000088
```

Figura 2 – Stream resultante do processo de serialização

Nota-se que, por se tratar de um protocolo binário, a *stream* de *bytes* resultante da serialização não é muito legível em um primeiro momento. Contudo, uma vez que esta segue a especificação de serialização de objetos do Java (ORACLE, 2017b), é possível entender sua estrutura através da análise da gramática que rege o formato das *streams* de dados serializados via os mecanismos nativos da linguagem (ORACLE, 2017a). As figuras 3 e 4 detalham a mesma *stream* mostrada anteriormente, dessa vez destacando cada um dos campos que a compõe:

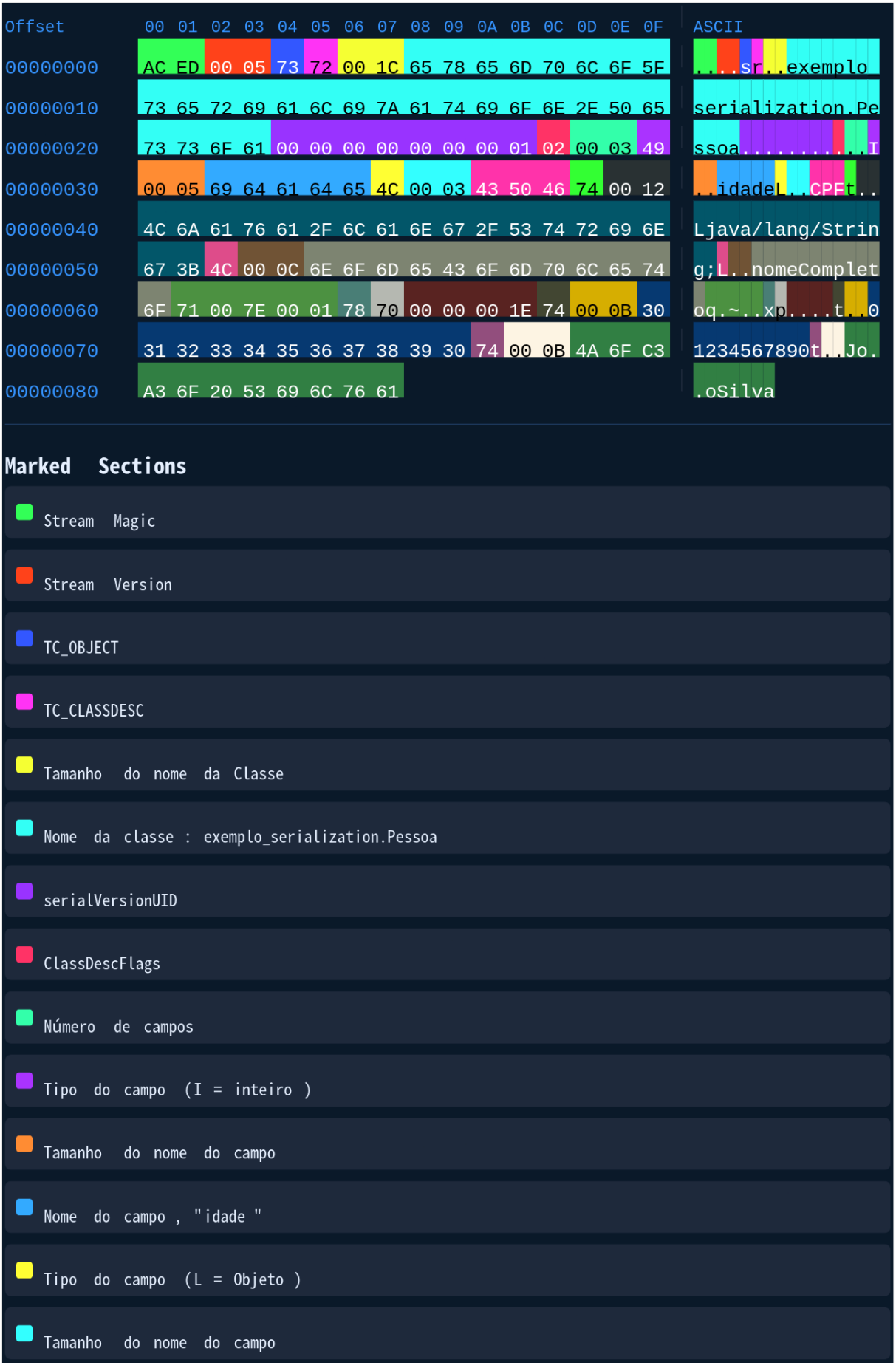

Figura 3 – Stream de dados detalhada byte-a-byte

Figura 4 – Stream de dados detalhada byte-a-byte (continuação)

A *stream* de *bytes* resultante poderia então, por exemplo, ser codificada em base64 para posteriormente ser enviada através de um parâmetro de uma requisição HTTP, como mostra o formato abaixo:

rO0ABXNyABxleGVtcGxvX3NlcmlhbGl6YXRpb24uUGVzc29hAAAAAAAAAAECAANJAAVpZGFkZUwA
A0NQRnQAEkxqYXZhL2xhbmcvU3RyaW5nO0wADG5vbWVDb21wbGV0b3EAfgABeHAAAAAedAALMDEy
MzQ1Njc4OTB0AAtKb8OjbyBTaWx2YQ==

3.2.3 Reconstruindo um objeto serializado através da desserialização

Por último, para reconstruir um objeto serializado previamente, é necessário realizar o caminho inverso. Para isto, seguindo o exemplo mostrado, após decodificar a *stream* de bytes mostrada acima e salvá-la em um arquivo, utiliza-se o método `java.io.ObjectInputStream.readObject()`, responsável por instanciar o novo objeto a partir de uma *stream* de *bytes*. O exemplo de código 9 ilustra a desserialização do objeto `pessoa1` mostrado previamente, a partir de uma instância da classe `ObjectInputStream`, que por sua vez foi instanciada a partir de uma `FileInputStream` apontando para o arquivo que contém a *stream* decodificada após sua transmissão:

Código Fonte 9 – Objeto sendo reconstruído a partir da stream de bytes obtida anteriormente

```
//dados recebidos via rede salvos no arquivo output_serialization

FileInputStream fileStream  = new FileInputStream("/tmp/output_serialization");
        ObjectInputStream inputStream = new ObjectInputStream(fileStream);
        exemplo_serialization.Pessoa pessoa2 = (exemplo_serialization.Pessoa)
            inputStream.readObject();


//
```

Após a execução do trecho de código 9, o novo objeto `pessoa2` irá conter todas as características do objeto `pessoa1` previamente serializado, finalizando assim o processo de desserialização.

3.3 DESSERIALIZAÇÃO INSEGURA EM JAVA: O CERNE DO PROBLEMA

Dado o cenário exibido na sessão anterior, agora é possível entender com mais clareza o funcionamento dos processos de serialização e desserialização no contexto da linguagem Java. Contudo, embora a serialização em Java seja uma funcionalidade poderosa e amplamente usada, seu mecanismo padrão de desserialização apresenta sérios riscos de segurança.

Isto se dá porque, como uma de suas características principais, o processo de desserialização

em Java por padrão não realiza nenhuma validação quanto à origem ou integridade do conteúdo contido na *stream* de *bytes* recebida. A API nativa do Java assume que os dados que chegam à ela são confiáveis, o que abre caminho para ataques quando essa entrada é manipulada de forma maliciosa. A partir disso surge o problema central abordado neste trabalho: a desserialização insegura de dados.

3.3.1 Desserialização: Classes utilizadas X classes disponíveis

Durante a desserialização, a instância do `ObjectInputStream` utilizada tenta automaticamente reconstruir os objetos da *stream* que está lendo, instanciando suas respectivas classes e populando seus campos. Contudo, um detalhe importante é que qualquer classe presente no *classpath* da aplicação que implemente a interface `Serializable`, mesmo que não tenha sido projetada para esse propósito específico, poderá ser instanciada caso apareça na *stream*. Isso implica que até mesmo classes que não são utilizadas em momento algum pela aplicação (como porções não usadas de bibliotecas importadas no projeto) podem ser instanciadas. Esse comportamento decorre da forma como a JVM (*Java Virtual Machine*) localiza e carrega classes, não restringindo a desserialização às classes estritamente utilizadas pela aplicação, mas sim a qualquer classe disponível e compatível no ambiente de execução.

Esse nível de permissividade implica que, caso um atacante envie uma *stream* contendo um objeto de uma classe explorável (isto é, que execute algum tipo de código considerado perigoso dentro das implementações dos *magic methods* discutidos anteriormente, por exemplo), o Java irá carregar e instanciar essa classe automaticamente, sem nenhum mecanismo interno que limite tal comportamento por padrão. Essa lacuna cria um vetor de ataque poderoso, sobretudo em aplicações que aceitam objetos serializados oriundos da rede, como servidores de aplicações *web* e sistemas distribuídos. Além disso, como muitos dos *frameworks* em Java utilizam da serialização para a comunicação interna, o impacto dessa vulnerabilidade pode ser ampliado ainda mais, chegando a possivelmente afetar não apenas o sistema local, mas sim todo o ecossistema de serviços interconectados.

3.3.2 Tipos de Falhas de desserialização insegura

As falhas de desserialização insegura podem ser divididas em dois grupos principais, que variam de acordo com a superfície de ataque da aplicação e com a forma com que os dados

serializados são manipulados. A depender desses fatores, geralmente é possível classificar as falhas entre problemas de manipulação direta dos dados serializados, também conhecido como data *tampering*, e execução de cadeias de métodos maliciosos, comumente chamadas de *gadget chains*.

No primeiro grupo, o atacante modifica diretamente o conteúdo de um objeto serializado antes que ele seja desserializado pela aplicação. Devido a ausência de validação por padrão durante a reconstrução do objeto, qualquer modificação nos valores dos atributos pode alterar logicamente o comportamento do programa. Tomando como exemplo uma instância da classe `Pessoa`, mostrada previamente, a alteração manual do número de CPF contido no objeto pode refletir no acesso ou modificação indevida de dados de terceiros, por exemplo. Vale ressaltar que o impacto de falhas incluídas neste grupo está fortemente relacionado às regras de negócio da aplicação.

Mais grave que a alteração de atributos do primeiro grupo, o segundo grupo ilustra um cenário onde um atacante consegue forçar a aplicação a executar comandos arbitrários durante o processo de desserialização. Isso ocorre através da construção de uma cadeia de objetos aninhados que, ao ser desserializada, provoca uma sequência de invocações de métodos que acabam por atingir o objetivo malicioso plantado pelo atacante. Tais cadeias são comumente conhecidas como *gadget chains*.

3.3.3 Gadgets e Gadget Chains

De maneira geral, *gadgets* são classes legítimas que já existem no *classpath* da aplicação, podendo ser partes da própria aplicação ou de suas dependências externas. Estas classes, por sua vez, possuem métodos específicos (como o *magic method* `readObject()`, por exemplo) que são responsáveis por executar operações automaticamente durante o processo de desserialização. Tais classes, isoladamente, não representam uma falha de segurança. No entanto, ao serem encadeadas de maneira específica, elas podem gerar efeitos colaterais devastadores.

Atacantes podem construir um objeto serializado contendo uma cadeia de *gadgets* que, quando interpretada pelo mecanismo de desserialização, percorre essa sequência e termina executando trechos de código arbitrários. Esse tipo de ataque, além de ser extremamente perigoso, geralmente não exige qualquer tipo de interação de usuários para ser executado. Na realidade, muitas vezes ataques deste tipo não exigem sequer que o processo de desserialização seja finalizado com sucesso, o que permitem ser executados mesmo em casos onde a

instanciação de um objeto resulta em um erro.

3.3.4 Exemplo de uma Gadget Chain

Esta seção tem como objetivo demonstrar como a desserialização de um objeto pode levar à execução remota de comandos em uma aplicação, e cria um cenário fictício onde um objeto serializado malicioso é desserializado por uma aplicação Java. Dado que a aplicação confia (erroneamente) no objeto recebido, isto é, não realiza nenhum tipo de checagem de segurança sobre este, através deste exemplo, é possível ter uma ideia do tipo de impacto que pode ser causado pela vulnerabilidade.

Uma dada aplicação Java contém a seguinte classe em seu *classpath*. Tal classe foi desenvolvida especificamente para a aplicação, e implementa a interface *Serializable* devido à necessidade de atender a regras de negócio existentes:

Código Fonte 10 – Classe ClasseUsadaPelaApp

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class ClasseUsadaPelaApp implements Serializable{

    private static final long serialVersionUID = 1L;

    public String log_request;
    public ClasseVulneravel objVulneravel;



    private void readObject(ObjectInputStream in) throws ClassNotFoundException,
        IOException {

        in.defaultReadObject();
        objVulneravel.log(log_request);
    }
}
```

Tal classe, por sua vez, possui como um de seus atributos um objeto da classe `ClasseVulneravel` . Além disso, como mostrado no trecho de código 10, ela também implementa o *magic method*

`readObject()`, devido à necessidade de implementar lógica adicional relacionada ao registro de *logs* com base na variável `log_request`.

A implementação da classe `ClasseVulneravel` pode ser vista no trecho de código 11.

Código Fonte 11 – Classe ClasseVulneravel

```
import java.io.IOException;
import java.io.Serializable;

public class ClasseVulneravel implements Serializable{

    private static final long serialVersionUID = 2L;

    private String log_details;



    public void log(String str) throws IOException {

        //registro de logs realizado

        Runtime.getRuntime().exec(str);
    }

}
```

Em condições normais, ao receber uma *stream* de *bytes* contendo um objeto serializado da classe `ClasseUsadaPelaApp`, a aplicação iria desserializá-la com sucesso, executando a lógica adicional de registro de *logs* incluída no *magic method* `readObject()`. O trecho de código da função `main()` responsável por iniciar o processo de desserialização a partir do arquivo que contém a *stream* de *bytes* pode ser visualizado no código 12:

Código Fonte 12 – Trecho responsável por desserializar o objeto

```
          FileInputStream fileStream1 = new FileInputStream("/tmp/gadget_chain");
        ObjectInputStream inputStream = new ObjectInputStream(fileStream1);
        exemplo_serialization.ClasseUsadaPelaApp cap = (exemplo_serialization.
            ClasseUsadaPelaApp) inputStream.readObject();
```

Contudo, dado que a aplicação desserializa a *stream* de *bytes* recebida de uma fonte não confiável sem realizar checagens de segurança, modificações realizadas a esta são capazes de influenciar o curso do processo de desserialização, pois uma vez que um agente externo controla

o valor da variável `log_request`, é possível invocar o método `ClasseVulneravel.log()` passando um valor arbitrário como argumento. Como consequência, este comportamento implica que agentes externos são capazes de executar comandos arbitrários através do abuso da confiança implícita depositada na *stream* de *bytes* a ser desserializada.

A sequência de invocações de métodos (*gadget chain* resultante) invocados ao longo do processo é listada abaixo, com o último método (`Runtime.exec()`) sendo responsável por executar um comando a partir de um argumento controlado pelo atacante (`log_request`):

```
ObjectInputStream.readObject()
ClasseUsadaPelaApp.readObject()
ClasseVulneravel.log()
Runtime.getRuntime()
Runtime.exec(log_request)
```

4 CASOS MEMORÁVEIS DE INCIDENTES ENVOLVENDO FALHAS DE DESSERIALIZAÇÃO INSEGURA EM JAVA

Embora muitas vezes seja considerada um problema que ocorre dentro de um contexto altamente específico, as falhas de desserialização insegura já se mostraram ser uma ameaça concreta, que tem capacidade de gerar consequências graves em sistemas reais. Incidentes envolvendo grandes corporações demonstram que até mesmo aplicações empresariais em produção já foram suscetíveis a ataques sofisticados que exploram esse vetor de ameaça, muitas vezes resultando em comprometimentos severos de segurança, o que muitas vezes se traduz em impactos financeiros ou danos à imagem de tais empresas. Este capítulo tem como objetivo realizar uma breve análise de casos reais de exploração da vulnerabilidade, e serve não apenas como comprovação empírica da gravidade desse tipo de falha, mas também como alerta para o potencial impacto que a falta de realização de validações de segurança sobre o que é aceito como entrada na desserialização de objetos pode causar.

4.1 APACHE COMMONS COLLECTIONS (2015)

Um dos episódios mais marcantes relacionados à desserialização insegura ocorreu em 2015, quando uma falha crítica na biblioteca *Apache Commons Collections*, amplamente utilizada por uma série de *frameworks* Java, foi descoberta. O problema residia em uma cadeia de métodos da classe `InvokerTransformer`, que podia ser explorada durante a desserialização para executar código arbitrário. A esta falha foi atribuído o CVE-2015-4852 (NIST, 2015), e evidenciou-se que aplicações que utilizavam servidores como *WebLogic*, *JBoss* e *Jenkins*, todas possuíam o *Apache Commons Collections* em seu *classpath* (BREEN, 2015). A vulnerabilidade foi amplamente explorada por meio de objetos especialmente construídos, que desencadeavam uma *gadget chain* ao serem desserializados. O impacto foi global, com muitos ambientes corporativos sofrendo invasões silenciosas, e uma série de *proof-of-concepts* (PoCs) sendo publicados dias após a divulgação do problema. Esse incidente levou ao surgimento da famosa ferramenta de ataque, a ysoserial (FROHOFF, 2014), que é utilizada para gerar *payloads* com cadeias de *gadgets* de bibliotecas vulneráveis até os dias de hoje.

4.2 JENKINS CI (2017)

Outro caso notório ocorreu com o servidor de integração contínua (CI) *Jenkins*, amplamente utilizado no mercado. Em 2017, uma falha de desserialização insegura permitia que usuários autenticados (ou mesmo não autenticados, a depender da configuração) enviassem objetos maliciosos serializados ao *endpoint* de comunicação remota do *Jenkins*. A vulnerabilidade foi catalogada como CVE-2017-1000353 (NIST, 2017) e permitia a atacantes executar código remotamente no host executando a instância do *Jenkins*. A falha teve alto impacto, afetando milhares de instâncias públicas do *Jenkins* em ambientes corporativos, e culminou na necessidade de uma revisão no modelo de serialização utilizado internamente pelo servidor.

4.3 ORACLE WEBLOGIC (2018–2020)

O servidor de aplicações *Oracle WebLogic* foi afetado por diversas vulnerabilidades relacionadas à desserialização entre os anos de 2018 e 2020 que, como agravante, teve a maioria delas exploradas em larga escala através da internet. Um dos casos mais graves foi o CVE-2019-2725 (NIST, 2019), que permitia a execução remota de código de forma não autenticada via *payloads* XML enviados através do protocolo SOAP para *endpoints* específicos do servidor. Essa falha foi considerada crítica (com a altíssima pontuação de CVSS: 9.8) e teve exploração ativa em *botnets* e *ransomware* dias após sua divulgação (CADIEUX COLIN GRADY, 2019). A Oracle publicou atualizações emergenciais, mas servidores vulneráveis continuaram sendo atacados por meses. Estima-se que centenas de servidores tenham sido comprometidos, sendo utilizados para finalidades como criação de *backdoors* persistentes e ataques de movimentação lateral em redes corporativas.

5 TÉCNICAS DE MITIGAÇÃO E PREVENÇÃO

Falhas de desserialização insegura são, por natureza, vulnerabilidades difíceis de mitigar completamente. Isto é especialmente verdade quando a arquitetura da aplicação depende fortemente da serialização para funcionar, e também é influenciado pelo tamanho total da aplicação, refletido no número de classes existente em seu *classpath*. No entanto, diversas estratégias e boas práticas podem ser adotadas para reduzir significativamente o risco de exploração. Este capítulo tem como objetivo abordar algumas delas.

5.1 SOLUÇÃO IDEAL: NÃO DESSERIALIZAR DADOS NÃO CONFIÁVEIS

Naturalmente, a forma mais eficaz de prevenção é muito simples: basta apenas não desserializar dados recebidos de fontes que não sejam confiáveis, como é o caso de fontes externas, solicitações HTTP, dados em *cookies* ou *headers*. Idealmente, a desserialização deve ser tratada como uma operação sensível, e seu uso deve ser cuidadosamente avaliado em qualquer sistema que esteja exposto a entradas externas. Contudo, dado que muitas aplicações dependem dos mecanismos de serialização para funcionarem corretamente, tal abordagem nem sempre é possível.

5.2 VALIDAÇÃO E RESTRIÇÃO DE TIPOS

Em casos onde a desserialização é inevitável, uma das abordagens válidas é restringir os tipos de classes que podem ser instanciadas durante o processo. Isso pode ser realizado através da criação de dois tipos de listas, que têm como objetivo limitar as classes que podem de fato ser desserializadas. São elas:

- **Allowlists**: permite apenas classes explicitamente autorizadas a serem desserializadas. Essa abordagem é mais restrita e segura, mas necessita que todas as classes passíveis de desserialização sejam determinadas previamente, o que pode acabar impactando questões de compatibilidade como a integração entre sistemas diferentes, por exemplo.
- **Denylists**: funciona através do bloqueio de classes já conhecidas por serem perigosas (como classes de bibliotecas vulneráveis). Essa abordagem é um pouco mais frágil,

pois exige conhecimento extenso da potencial superfície de ataque, além de atualização constante.

Do Java 9 em diante, a Oracle introduziu melhorias no mecanismo de segurança através do `ObjectInputFilter` (ORACLE, 2015), que permite definir políticas de filtragem de classes durante a desserialização, como suporte à utilização de filtros, por exemplo. Essa funcionalidade é fundamental para evitar que classes inesperadas sejam carregadas a partir de uma *stream* manipulada.

5.3 LOOK-AHEAD DESERIALIZATION

Outra técnica de mitigação adicional é a chamada *Look-Ahead Deserialization*, que consiste em analisar os metadados da *stream* serializada antes de processar totalmente a desserialização dos objetos nela contidos. Isso permite verificar, em tempo de execução, quais classes estão sendo dinamicamente carregadas, bem como aplicar políticas de segurança apropriadas antes de efetuar de fato a desserialização do objeto.

Embora o Java padrão não ofereça suporte direto a esse tipo de análise preemptiva, bibliotecas e *frameworks* de segurança podem realizar inspeções baseadas em assinaturas de classes ou em políticas específicas antes da execução do método `readObject()`. Um exemplo de ferramenta que realiza tal validação é o *SerialKiller* (IKKISOFT, 2015)

5.4 GADGET HUNTING: MOTIVOS PARA EVITAR

Uma armadilha bastante comum encontrada por equipes na hora de mitigar vulnerabilidades de desserialização insegura é tentar mapear manualmente todos os *gadgets* presentes no *classpath* da aplicação. Tal abordagem não é muito recomendável pois, além de ser praticamente impossível prever todas as combinações de classes e métodos que podem compor uma potencial *gadget chain*, nada impede que tais *gadgets* existam em bibliotecas internas ou dependências essenciais para o funcionamento da aplicação. Além disso, novas dependências ou atualizações podem introduzir novos *gadgets* a qualquer momento, sem aviso prévio.

5.5 ESTRATÉGIAS COMPLEMENTARES

Por último, além das práticas principais citadas acima, outras técnicas adicionais podem fortalecer a segurança de aplicações que precisem realizar a desserialização de objetos. Estas estão listadas abaixo:

- Executar o código de desserialização em ambientes de baixo privilégio, de forma a reduzir o impacto de eventuais *payloads* maliciosos.
- Fazer uso da *keyword* `transient` em campos sensíveis, pois estes campos não são incluídos na serialização padrão, evitando que informações críticas sejam manipuladas ou expostas.
- Verificar a integridade de objetos serializados, aplicando técnicas como a checagem de assinaturas digitais sobre o conteúdo da *stream*.
- Fazer uso de soluções de análise de código, com o objetivo de certificar que as políticas de segurança envolvendo a serialização e a desserialização de objetos (como políticas de validação da entrada dos usuários, por exemplo) estejam sendo corretamente impostas.

Vale também mencionar que embora os mecanismos nativos de serialização do Java representem uma grande parte da superfície de ataque existente, é importante relembrar que a desserialização insegura não se limita apenas a esse contexto. Além dele, uma outra parte significativa dos riscos encontrados em aplicações modernas decorre do uso de *frameworks* e bibliotecas de serialização alternativas como *Jackson* ou *XStream*, que embora por padrão restrinjam quais classes possam ser desserializadas, podem acabar criando mais problemas quando seu uso acompanha configurações potencialmente perigosas ou permissivas, como a habilitação de polimorfismo. Além disso, mecanismos de comunicação remota, como o previamente mencionado Java Remote Method Invocation ou o Java Message Service frequentemente utilizam internamente processos de serialização e desserialização, o que amplia ainda mais a superfície de ataque de aplicações.

É imprescindível ter consciência de que, a medida que a *codebase* da aplicação cresce, a sua superfície de ataque também se expande. Tal correlação deve ser compreendida pelos desenvolvedores e responsáveis pela aplicação, de forma a sempre ajustar e revisar as medidas de segurança a serem tomadas para garantir robustez do ponto de vista de segurança.

De forma a facilitar a visualização e a distinção entre as estratégias apresentadas, a tabela abaixo exibe um comparativo entre as medidas de mitigação discutidas:

Tabela 1 – Comparativo entre as estratégias de mitigação apresentadas

Estratégia de mitigação	Descrição
Impedir desserialização de dados não confiáveis	Impede o problema. Contudo, não é viável caso a aplicação precise aceitar objetos serializados originários de fontes externas.
Validar e restringir tipos	Razoavelmente efetiva, mas tem suas limitações: É necessário determinar previamente todas as classes que poderiam ser desserializadas (o que a depender de como a aplicação for estruturada pode não ser viável) ou todas as classes que não possam ser desserializadas (impossível, pois não há como saber todas as classes que podem ser potencialmente perigosas)
Look-Ahead Deserialization	Efetiva, mas possui os mesmo problemas da validação de tipos (isto é, é necessário saber previamente que classes são malicisas ou não).
Gadget Hunting	Abordagem altamente ineficiente, não configura uma boa estratégia de mitigação.
Executar desserialização em ambiente de baixo privilégio	Especialmente útil como parte de uma abordagem de segurança em camadas, ajuda na mitigação de maior impacto em um primeiro momento, mas não deve ser usado como única medida.
Utilização da *keyword* transient	Impede a manipulação de campos que contenham informações potencialmente críticas, mas não atua diretamente sobre vetores como a exploração de *gadget chains*.
Verificar a integridade dos objetos serializados	Ajuda a garantir que os objetos não foram manipulados antes da desserialização.
Fazer uso de soluções de análise de código	Auxilia a garantir que as devidas políticas de segurança, como validações de entrada, estão sendo corretamente aplicadas.

6 CONCLUSÃO

Este trabalho apresentou uma análise aprofundada da vulnerabilidade de desserialização insegura no contexto da linguagem Java, abordando desde a fundamentação conceitual sobre os mecanismos de serialização e desserialização de objetos até as técnicas de mitigação existentes. Foi evidenciado que, apesar de essenciais para a persistência e transmissão de dados em sistemas distribuídos, essas operações, quando realizadas sem as devidas validações de segurança, introduzem vetores de ataque com elevado potencial de comprometimento da aplicação.

O trabalho demonstrou que o processo padrão de desserialização em Java não impõe restrições quanto às classes que podem ser instanciadas a partir da *stream* de *bytes* serializada, permitindo a reconstrução de qualquer objeto cujo tipo esteja presente no classpath e implemente a interface `Serializable`. Essa característica, somada à possibilidade de manipulação da estrutura serializada quando esta é proveniente de uma fonte externa, abre espaço para ataques que exploram tanto a modificação de atributos já existentes na *stream*, quanto a execução de sequências de invocações de métodos que culminam em impactos mais severos, através de estruturas conhecidas como *gadget chains*.

A partir da análise de casos reais e da construção de exemplos práticos, observou-se o impacto real da vulnerabilidade: ela tem sido historicamente explorada contra sistemas de grande porte, ocasionando brechas significativas de segurança. Por fim, a pesquisa chamou atenção às medidas de mitigação do problema, e destacou a importância de tomar cuidado quando é necessário executar a desserialização de objetos oriundos de fontes não confiáveis. Discutiu-se também a eficácia de abordagens de mitigação que podem ser intuitivamente consideradas eficientes em um primeiro momento, mas que provam-se não muito efetivas, como a identificação manual de *gadgets*. Por último, enfatizou-se também a necessidade de mitigações robustas quando a desserialização de objetos vindos de fontes externas é inevitável, como a aplicação de listas de controle e o emprego de técnicas como *Look-Ahead Serialization*, bem como a adoção de estratégias complementares, como a execução da lógica de desserialização em ambientes restritos de privilégio.

Os riscos associados à desserialização insegura evidenciam a importância da ênfase no aspecto de segurança da engenharia de *software* desde os fundamentos de uma aplicação, e destacam como, muitas vezes, vulnerabilidades de natureza crítica podem estar escondidas em

suas camadas profundas, apenas esperando serem encontradas. Por último, como sugestão de trabalhos a serem realizados futuramente, destacam-se a exploração de ferramentas automatizadas para realização do mapeamento de *gadgets* no código fonte de aplicações, bem como o estudo mais aprofundado de incidentes de segurança envolvendo falhas de desserialização insegura.

REFERÊNCIAS

BREEN, S. *What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability.* 2015. Https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/. Acesso em: 25 jul. 2025.

CADIEUX COLIN GRADY, J. S. M. V. P. *Sodinokibi ransomware exploits WebLogic Server vulnerability*. 2019. Disponível em: <https://blog.talosintelligence.com/sodinokibi-ransomware-exploits-weblogic/>. Acesso em: 19 jul. 2025.

FROHOFF, G. *Ysoserial*. 2014. Disponível em: <https://github.com/frohoff/ysoserial>. Acesso em: 15 jun. 2025.

IKKISOFT. *SerialKiller*. 2015. Disponível em: <https://github.com/ikkisoft/SerialKiller>. Acesso em: 19 jul. 2025.

NIST. *CVE-2015-4852*. 2015. Disponível em: <https://nvd.nist.gov/vuln/detail/cve-2015-4852>. Acesso em: 22 jun. 2025.

NIST. *CVE-2017-1000353*. 2017. Disponível em: <https://nvd.nist.gov/vuln/detail/CVE-2017-1000353>. Acesso em: 11 jul. 2025.

NIST. *CVE-2019-2725*. 2019. Disponível em: <https://nvd.nist.gov/vuln/detail/cve-2019-2725>. Acesso em: 11 jul. 2025.

ORACLE. *Interface ObjectInputFilter*. 2015. Disponível em: <https://docs.oracle.com/javase/9/docs/api/java/io/ObjectInputFilter.html>. Acesso em: 25 jul. 2025.

ORACLE. *Grammar for the Stream Format*. 2017. Disponível em: <https://docs.oracle.com/en/java/javase/11/docs/specs/serialization/protocol.html#grammar-for-the-stream-format>. Acesso em: 26 jun. 2025.

ORACLE. *Java Object Serialization Specification: Contents*. 2017. Disponível em: <https://docs.oracle.com/en/java/javase/11/docs/specs/serialization/index.html>. Acesso em: 26 jun. 2025.

OWASP. *OWASP Top Ten 2017*. 2017. Disponível em: <https://owasp.org/www-project-top-ten/2017/>. Acesso em: 15 jun. 2025.