# Análise sobre Ataques de Injeção de Códigos Maliciosos: Impactos e Medidas de Proteção

**Deivis F. G. Fagundes**[1], **João P. Righi** [1]

[1]Centro de Tecnologia – Universidade Federal de Santa Maria (UFSM)
Caixa Postal 97105-900 – Santa Maria – RS – Brasil

{dffagundes,jprighi}@inf.ufsm.br

***Abstract.*** *In today's world, numerous applications are developed every day, many of them handling user information. Therefore, it is essential that these systems are extremely secure and do not allow information leaks. This work aims to present and discuss solutions for one of the most well-known and recurring attacks: code injection attacks. This type of attack exploits system vulnerabilities to manipulate its processes and data. Throughout this study, the different types of injection attacks and ways to prevent them are presented. As a result, it is expected that new systems can be developed with greater security, using this work as a reference.*

***Resumo.*** *No contexto atual, diversas aplicações são desenvolvidas diariamente, muitas das quais manipulam informações sensíveis de usuários. Por isso, é essencial que esses sistemas apresentem altos níveis de segurança para evitar vazamento de informações. Este trabalho tem como objetivo analisar e discutir soluções para um dos ataques mais recorrentes: a injeção de código. Esse tipo de ataque explora vulnerabilidades no sistema para manipular seus processos e dados. Ao longo deste estudo, são apresentadas a taxonomia dos ataques de injeção, trabalhos relacionados, e abordagens de mitigação utilizando frameworks modernos. Como resultado, espera-se fornecer um guia prático e teórico para o desenvolvimento de sistemas mais seguros.*

## 1. Introdução

Com a crescente digitalização dos serviços online, a quantidade de dados manipulados por aplicações web expandiu-se de forma exponencial. Consequentemente, a necessidade de sistemas de segurança mais robustos também se tornou imprescindível. Diversos tipos de ciberataques são utilizados para a obtenção ilícita de dados ou para causar prejuízos a organizações. Essas ações maliciosas podem ter impactos severos, como a indisponibilidade de sistemas operacionais, o vazamento de dados de clientes e o comprometimento de informações estratégicas.

Este trabalho destaca métodos de proteção contra ataques de injeção de código, uma das vulnerabilidades mais críticas em aplicações web. Segundo a *Open Worldwide Application Security Project* [OWASP Foundation 2021], as falhas de injeção mantêm-se sistematicamente entre os riscos mais frequentes e severos.

A injeção de código ocorre quando um software vulnerável interpreta erroneamente dados fornecidos pelo usuário como instruções executáveis [Clarke 2012]. Dessa

forma, um invasor introduz comandos maliciosos que alteram o fluxo de execução esperado da aplicação. Essas falhas ocorrem frequentemente em consultas a bancos de dados (SQL, NoSQL), interpretadores de comandos de sistemas operacionais (Command Injection) e renderização no lado do cliente (Cross-Site Scripting - XSS) [Halfond et al. 2006].

Um exemplo notório do impacto de falhas estruturais de segurança ocorreu com o LinkedIn em 2012. Devido a vulnerabilidades na arquitetura de autenticação e ausência de sanitização adequada, milhões de hashes de senhas foram expostos e posteriormente decifrados, demonstrando o risco crítico de falhas em camadas de acesso a dados [SentinelOne 2016].

As principais questões que este artigo busca responder são: Quais as particularidades taxonômicas e mecanismos dos ataques de injeção? E como as práticas e frameworks arquiteturais auxiliam na defesa contra essas ameaças? O trabalho está organizado da seguinte forma: A Seção 2 apresenta os trabalhos relacionados e a taxonomia dos ataques. A Seção 3 aborda os requisitos de segurança. A Seção 4 analisa a arquitetura da solução, a comparação de tecnologias e a demonstração prática. Por fim, as Seções 5 e 6 abordam a manutenção e as considerações finais.

## 2. Trabalhos Relacionados e Taxonomia

### 2.1. Trabalhos Relacionados

A literatura acadêmica apresenta diversas pesquisas focadas na identificação e mitigação de vulnerabilidades de injeção. [Halfond et al. 2006] estabeleceram uma das classificações mais consolidadas para ataques de SQL Injection (SQLi) e contramedidas, destacando a necessidade de uma defesa em camadas. [Clarke 2012] ampliou a discussão abordando as vulnerabilidades de aplicações que não sanitizam adequadamente as entradas do usuário, propondo abordagens de análise de código.

Em uma perspectiva de avaliação, [Fonseca et al. 2007] compararam diferentes ferramentas automatizadas de varredura de vulnerabilidades, ressaltando que, embora úteis, os scanners apresentam altas taxas de falsos negativos, exigindo arquiteturas de software projetadas com segurança intrínseca. Diferente dos trabalhos estritamente teóricos, este artigo foca na aproximação da teoria com a prática de desenvolvimento, detalhando exemplos implementados em um framework de mercado atual.

### 2.2. Taxonomia e Mecanismos de Exploração

A injeção ocorre invariavelmente devido a uma falha arquitetural: a não segregação estrita entre o que o sistema considera ”instrução”e o que é ”dado do usuário”. Para compreender as formas de defesa, categorizam-se os principais vetores:

- **SQL Injection (SQLi):** Manipulação de consultas de banco de dados relacional. Pode ser classificada em:
  - *Tautologia:* O invasor injeta operadores lógicos para tornar a declaração verdadeira. Ex: Em uma autenticação, o atacante insere `admin' OR 1=1 --`. O `--` anula a checagem de senha e a lógica obriga o banco a retornar o acesso.
  - *Baseada em União (UNION):* Combina a consulta original com uma consulta arbitrária para extrair tabelas ocultas.

  - *Baseada em Erro:* Força exceções do banco para revelar metadados detalhados sobre a arquitetura do sistema.
- **Cross-Site Scripting (XSS):** Injeção de scripts maliciosos (HTML/JavaScript) executados no navegador de outras vítimas [Hydara et al. 2015]. Pode ser *Refletido* (no parâmetro da URL, refletindo o ataque na resposta HTTP) ou *Armazenado* (gravado no banco de dados e replicado persistente na página).
- **Command Injection:** Injeção de comandos passados diretamente a sub-processos do sistema operacional hospedeiro [Su and Wassermann 2006] (ex: funções `system()` ou `exec()`). O atacante usa metacaracteres delimitadores (`;`, `||`) para abortar a intenção original e iniciar comandos arbitrários.

## 3. Levantamento de Requisitos de Segurança

Para mitigar tais vulnerabilidades, é crucial formalizar requisitos rigorosos de segurança de software [Sommerville 2011].

### 3.1. Requisitos Funcionais

Os requisitos funcionais referem-se aos mecanismos intrínsecos de controle que o sistema deve executar (Tabela 1).

| **Requisitos** | **Descrição** |
|---|---|
| Validação Estrita | Entradas devem ser validadas contra critérios rigorosos (Allowlist), checando tipos e padrões. |
| Sanitização | Neutralização de caracteres especiais (*Entity Encoding*), evitando que o interpretador os reconheça. |
| Autenticação | Controle de acesso baseado em funções (RBAC) rigoroso a cada requisição HTTP. |

**Tabela 1. Principais Requisitos Funcionais de Segurança**

### 3.2. Requisitos Não Funcionais

Esses requisitos ditam restrições arquiteturais sem comprometer a usabilidade da aplicação (Tabela 2).

| **Requisitos** | **Descrição** |
|---|---|
| Desempenho | A sanitização não deve introduzir latência (overhead de processamento mínimo). |
| Manutenibilidade | Separação de interesses na arquitetura (como no MVC) para garantir rotinas organizadas. |

**Tabela 2. Principais Requisitos Não Funcionais de Segurança**

## 4. Arquitetura da Solução e Abordagens de Defesa

A eficácia sistêmica demanda o princípio de *Defesa em Profundidade*. Para isso, explora-se a base arquitetural da aplicação, compara-se as tecnologias de mitigação, avança-se para a infraestrutura complementar e demonstra-se a aplicação prática no código-fonte.

### 4.1. O Padrão MVC e o Framework Laravel

Na concepção de sistemas seguros, a adoção de padrões arquiteturais maduros é essencial. Analisa-se aqui o framework PHP Laravel [Laravel 2024], estruturado sob o padrão MVC (*Model, View, Controller*), conforme ilustrado na Figura 1.

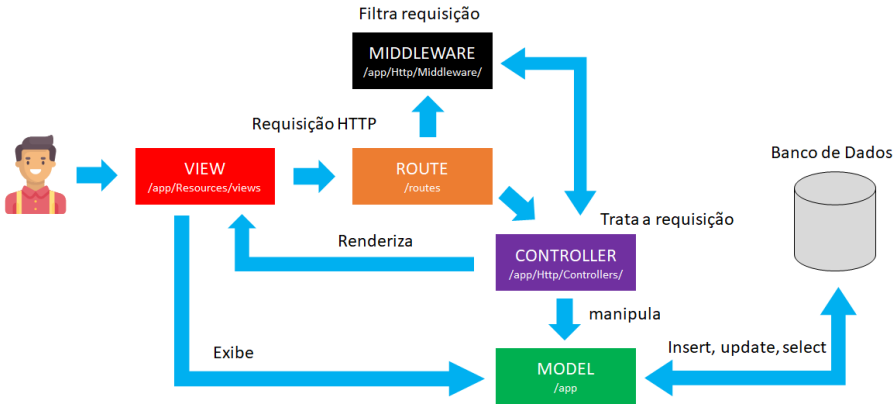


**Figura 1. Padrão MVC e Integração com Banco de Dados. Fonte [Gregório 2021]**

O *Model* (Eloquent ORM) gerencia o banco blindando contra SQLi, a *View* (Blade) foca na apresentação segura aplicando *escape* automático contra XSS, e o *Controller* aplica lógicas e validações estritas por meio de *middlewares* antes de qualquer manipulação de dados sensíveis.

### 4.2. Comparação das Tecnologias de Mitigação

A Tabela 3 contrasta estratégias reativas e proativas de mitigação em nível de aplicação e banco de dados.

### 4.3. Defesa em Profundidade na Infraestrutura (DevSecOps)

Mesmo com a aplicação protegida em nível de código (via arquitetura MVC e abstração de ORM), a abordagem de Defesa em Profundidade estipula que a infraestrutura subjacente seja independentemente resiliente. A adoção de práticas de *DevSecOps* introduz camadas físicas e lógicas de isolamento que delimitam e contêm o raio de ação de um ataque caso o interpretador web seja comprometido.

O isolamento por meio de contêineres, como o ecossistema **Docker**, atua como uma contenção de privilégios primária contra vetores de Injeção de Comandos [Docker Inc. 2026]. Caso um invasor consiga injetar e executar comandos no sistema operacional por meio de uma vulnerabilidade residual (um *bypass* de sanitização em uma funcionalidade de envio de arquivos, por exemplo), sua execução ficará restrita ao *namespace* delimitado do contêiner. Quando configurado com rigor, rodando em modo sem privilégios administrativos (*rootless*) e mapeando o sistema de arquivos base como estritamente leitura (*read-only*), a arquitetura impede ativamente que o invasor modifique

| Tecnologia | Mecanismo de Defesa | Vantagens e Limitações |
|---|---|---|
| **WAF** | Defesa periférica de rede. Inspeciona o tráfego HTTP. | **Vantagem:** Aplicação sem alterar código legado. **Limitação:** Falha contra ofuscação avançada. |
| **Prepared Statements** | Defesa em nível de driver. A sintaxe é compilada separada dos dados. | **Vantagem:** Impossibilita manipulação da *parse tree*. **Limitação:** Requer refatoração manual do código. |
| **ORM** | Defesa arquitetural. Mapeia dados em objetos. | **Vantagem:** Emprega *Prepared Statements* transparentemente. **Limitação:** Possível *overhead* em queries complexas. |

**Tabela 3. Comparação de Tecnologias Contra Injeção**

o ambiente, instale *backdoors* de persistência ou acesse variáveis de sistema críticas do servidor hospedeiro.

Ademais, em ambientes modernos de alta disponibilidade, a orquestração desses contêineres fornece pontos de estrangulamento ideais para a filtragem ativa de anomalias. Ao implantar a aplicação orquestrada via **Kubernetes** [The Kubernetes Authors 2026], a segurança transita das mãos exclusivas do desenvolvedor para a borda da rede (*Edge Computing*). Configurando controladores de entrada (*Ingress Controllers*) alimentados por soluções de proxy reverso e balanceamento de carga de alto desempenho, como o **HAProxy** [HAProxy Technologies 2026], a organização consolida uma inspeção centralizada. O proxy avalia os pacotes da camada de aplicação (Camada 7) e impõe políticas de recusa (WAF) que filtram cargas ativas de XSS e SQLi milissegundos antes que o tráfego atinja o ambiente PHP interno (Laravel). Esta sinergia entre código parametrizado e infraestrutura imutável aniquila a principal premissa dos ciberataques modernos: a capacidade de progressão lateral na rede local.

## 4.4. Demonstração Prática: Defesa Contra SQLi

Para evidenciar o impacto estrutural na camada de software, compara-se um código PHP legado vulnerável a SQLi com a solução proposta via Laravel MVC.

### O Cenário Vulnerável (Spaghetti Code):

```
$id = $_GET['id'];
// VULNERAVEL: Entrada do usuario concatenada diretamente na query
$sql = "SELECT * FROM produtos WHERE id = " . $id;
$result = $conn->query($sql);
```

**Listing 1. Abordagem Vulnerável (Concatenação Direta)**

### O Cenário Seguro (Framework MVC Laravel):

```
public function exibirProduto(Request $request) {
    // 1. Validacao Estrita: rejeita imediato valores maliciosos
    $dadosValidados = $request->validate([
```

```
        'id' => 'required|integer|min:1'
    ]);

    // 2. Defesa via ORM: Uso transparente de Prepared Statements
    $produto = Produto::find($dadosValidados['id']);

    // 3. O framework encaminha para a View
    return view('produto.show', ['produto' => $produto]);
}
```

**Listing 2. Abordagem Segura (Mapeamento ORM e Validação)**

Neste código, a validação exige explicitamente um número inteiro e o método `find()` emprega internamente *Prepared Statements* da biblioteca PDO, isolando a string injetada da execução lógica do SGBD.

### 4.5. Demonstração Prática: Mitigação de XSS na Camada de Visão

Além da proteção no banco de dados, o framework atua na resposta HTTP enviada ao cliente. No Laravel, o motor de templates *Blade* converte automaticamente caracteres especiais em entidades HTML utilizando a função nativa `htmlspecialchars` do PHP, barrando ataques de Cross-Site Scripting.

```
// Uso SEGURO (Escape automatico padrao do Laravel):
<h1>{{ $produto->nome }}</h1>

// Uso VULNERAVEL (Interpretacao direta, usado para renderizar raw HTML
    ):
<h1>{!! $produto->nome !!}</h1>
```

**Listing 3. Prevenção Automática de XSS em Views (Blade)**

O uso da sintaxe de chaves duplas impede que cargas armazenadas no banco de dados reflitam scripts ativos no cliente final. Para fortalecer ainda mais a defesa contra XSS baseado em DOM, é recomendada a implementação arquitetural de cabeçalhos *Content Security Policy* (CSP).

## 5. Plano de Manutenção, CI/CD e SecDevOps

O desenvolvimento seguro é apenas a base do ciclo de vida do software. Como os cenários de ameaças são dinâmicos e novas vulnerabilidades (zero-days) são descobertas de forma contínua, o plano de manutenção deve transcender a simples revisão manual de código e adotar o paradigma *SecDevOps*, integrando a segurança de forma automatizada no pipeline de Integração Contínua e Entrega Contínua (CI/CD) [Vehent 2018].

### 5.1. Testes Estáticos e Dinâmicos de Segurança

A implementação de uma esteira de entrega automatizada deve incluir obrigatoriamente duas abordagens complementares de verificação contínua:

- **SAST (Static Application Security Testing):** Trata-se da análise de código orientada a ”caixa branca”(White-box). Ferramentas de SAST varrem o código-fonte ou os binários sem executá-los. O objetivo primário é mapear a Árvore Sintática Abstrata (AST) do software em busca de padrões de codificação inseguros, como a ausência de sanitização antes de chamadas diretas ao banco de dados ou variáveis mal encapsuladas.

- **DAST (Dynamic Application Security Testing):** Configura uma análise de ”caixa preta”(Black-box). O processo DAST interage com a aplicação real em tempo de execução, enviando massivamente cargas automatizadas de *Fuzzing* (sequências extensas de payloads contendo assinaturas de SQLi e XSS) para as interfaces HTTP da API e formulários web. Ferramentas padrão da indústria, como o OWASP ZAP (Zed Attack Proxy) [OWASP Foundation 2026], avaliam as respostas do servidor em busca de anomalias, latências não naturais ou exceções não tratadas que confirmem uma vulnerabilidade explorável.

### 5.2. Análise de Composição de Software (SCA)

Uma parcela significativa das vulnerabilidades de injeção em sistemas contemporâneos não se origina no código nativo escrito pela equipe interna, mas sim em pacotes de terceiros e bibliotecas *open-source* (dependências diretas e transitivas). O conceito de *Software Composition Analysis* (SCA) envolve a varredura contínua e sistêmica da árvore de dependências do projeto.

Mecanismos integrados de SCA cruzam as versões de todos os pacotes instalados com bancos de dados globais de inteligência de ameaças, como a base do CVE (*Common Vulnerabilities and Exposures*). Caso uma biblioteca de manipulação de imagem ou de interpretação XML sofra a divulgação pública de uma falha de Command Injection, o pipeline de integração contínua (CI/CD) abortará preventivamente a construção (build) e implantação da aplicação até que a equipe providencie o patch (atualização) correspondente.

## 6. Conclusão

Este trabalho apresentou uma análise estruturada sobre os ataques de injeção de código, mapeando desde seus mecanismos de exploração até a infraestrutura sistêmica de defesa. Conclui-se que arquiteturas ancoradas exclusivamente em bloqueios reativos periféricos (como WAFs) não garantem blindagem autossuficiente contra invasores determinados utilizando cargas ofuscadas.

A defesa efetiva demanda a implementação direta de boas práticas no ciclo primário de Engenharia de Software e de Infraestrutura. O emprego metódico de Validação Estrita integrado à solidez de abstrações relacionais como o ORM e motores de renderização, amparados por ecossistemas orquestrados de contêineres e políticas ativas de CI/CD, formam a barreira mais eficiente. O desenvolvimento orientado a padrões arquiteturais modernos reduz sensivelmente a margem de erro humana e propicia uma escalabilidade confiável, atacando o cerne das vulnerabilidades através de um paradigma *Secure-by-Design*.

**Nota de Transparência Institucional:** Durante a etapa final de formatação deste manuscrito, os autores utilizaram o modelo de inteligência artificial generativa Google Gemini de forma estritamente ética, exclusivamente para revisão gramatical, aprimoramento da fluidez semântica e consolidação estrutural dos argumentos. Os autores atestam que a idealização, desenvolvimento prático (código-fonte), embasamento teórico e escolhas metodológicas apresentadas são de autoria unicamente humana, isentando a ferramenta

de qualquer coautoria e assumindo responsabilidade total e irrestrita pela veracidade dos dados finais submetidos.

## Referências


Clarke, J. (2012). *SQL Injection Attacks and Defense*. Syngress Publishing / Elsevier, 2nd edition.

Docker Inc. (2026). Docker security. `https://docs.docker.com/engine/security/`. Acesso em: 22 de agosto de 2026.

Fonseca, J., Vieira, M., and Madeira, H. (2007). Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 365–372. IEEE.

Gregório, J. L. (2021). Introdução ao laravel framework – parte 03. `https://www.jlgregorio.com.br/2021/08/27/introducao-ao-laravel-framework-parte-03/`. Acesso em 13 de julho de 2024.

Halfond, W. G. J., Viegas, J., and Orso, A. (2006). A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*. IEEE.

HAProxy Technologies (2026). Haproxy configuration manual. `https://docs.haproxy.org/`. Acesso em: 22 de agosto de 2026.

Hydara, I., Sultan, A. B. M., Zulzalil, H., and Admodisastro, N. (2015). Current state of research on cross-site scripting (xss) – a systematic literature review. *Information and Software Technology*, 58:170–186.

Laravel (2024). Laravel documentation - security. `https://laravel.com/docs/`. Acesso em: 13 de julho de 2024.

OWASP Foundation (2021). Owasp top 10 - 2021. `https://owasp.org/Top10/`. Acesso em: 13 de julho de 2024.

OWASP Foundation (2026). Owasp zed attack proxy (zap). `https://www.zaproxy.org/`. Acesso em: 22 de agosto de 2026.

SentinelOne (2016). Blast from the past: 2012 linkedin security breach dumps more than 100m additional records. `https://www.sentinelone.com/blog/blast-past-2012-linkedin-breach-dumps-100m-additional-records/`. Acesso em: 13 de julho de 2024.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Prentice Hall, 9th edition.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 372–382, New York, NY, USA. Association for Computing Machinery.

The Kubernetes Authors (2026). Kubernetes security concepts. `https://kubernetes.io/docs/concepts/security/`. Acesso em: 22 de agosto de 2026.

Vehent, J. (2018). *Securing DevOps: Security in the Cloud*. Manning Publications.