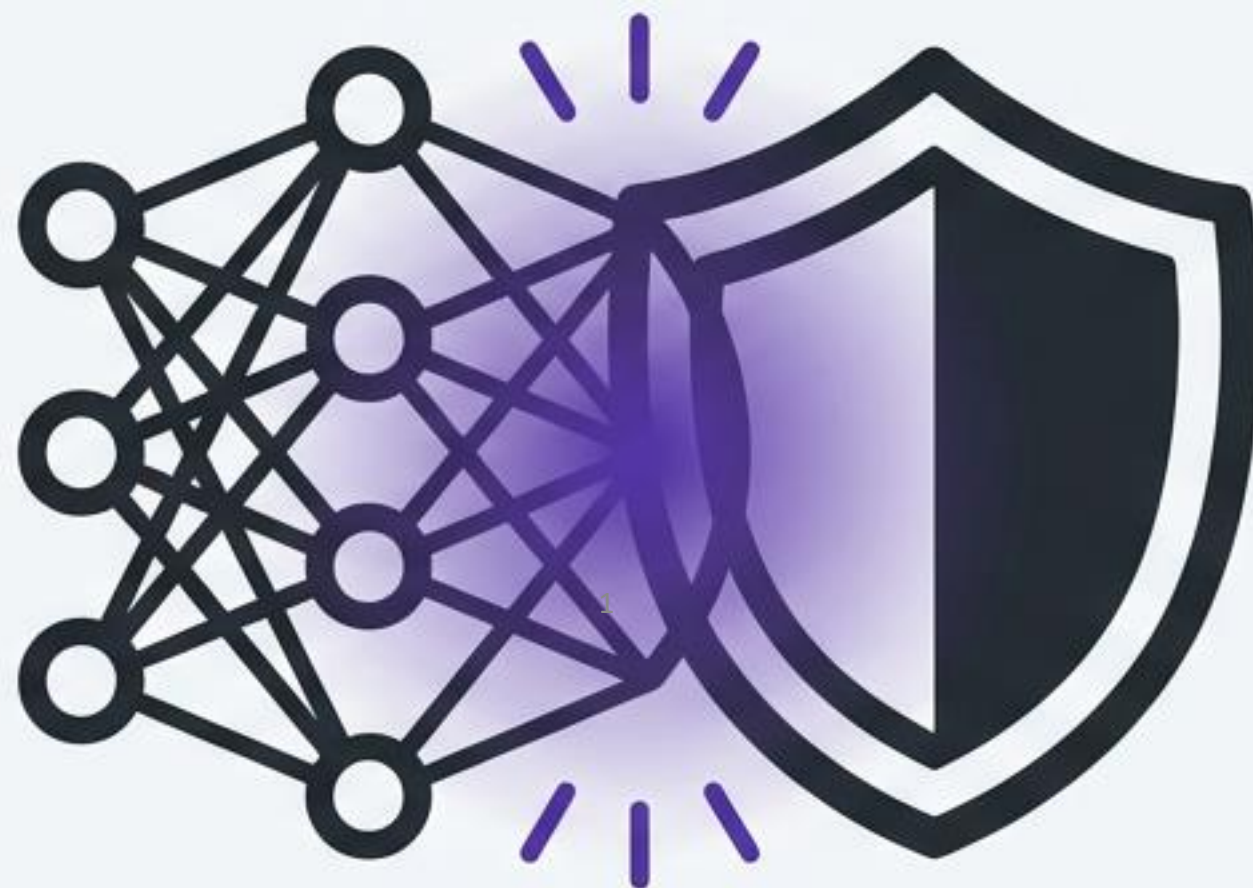


Can Large Language Models Be Taught to Fix Security Flaws?

Introducing vulDeRAG: A System for Enhancing LLMs in Automated Vulnerability Repair.

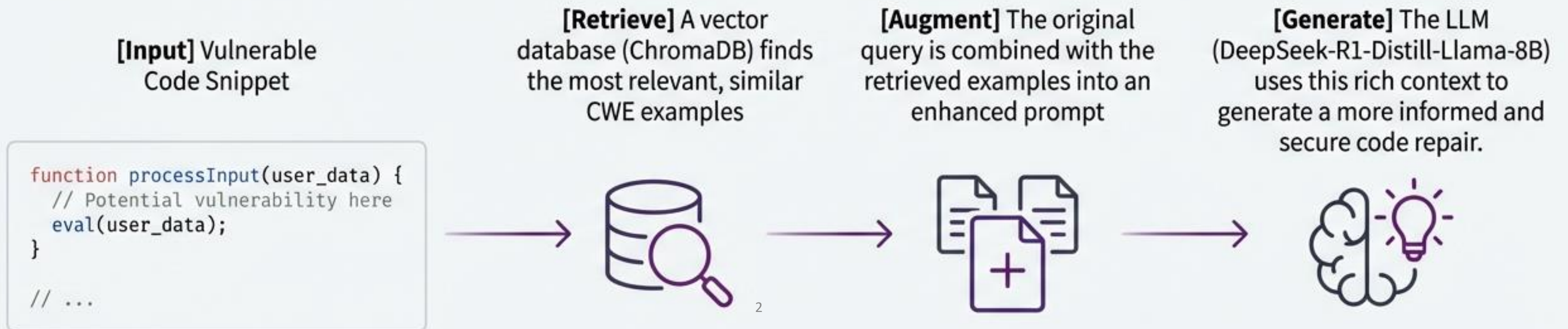
Large Language Models (LLMs) excel at generating code, but fixing security vulnerabilities is a different challenge. It demands deep, specialized knowledge of attack patterns, secure coding practices, and vulnerability classes (CWEs). This project investigates if we can elevate an LLM from a generalist coder to a security specialist.



vulDeRAG

The Solution: Augmenting Intelligence with Expert Knowledge

Our approach is Retrieval-Augmented Generation (RAG). Instead of relying solely on its internal training, the LLM is given access to a curated knowledge base of real-world vulnerability examples and explanations from the Common Weakness Enumeration (CWE) database.



RAG turns the LLM from a student taking a closed-book exam to an expert with an entire security library at its disposal.

RAG Database Building-1

▼ Demonstrative Examples

Example 1

This code snippet deserializes an object from a file and uses it as a UI button:

Example Language: **Java**

(bad code)

```
try {  
    File file = new File("object.obj");  
    ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));  
    javax.swing.JButton button = (javax.swing.JButton) in.readObject();  
    in.close();  
}
```

This code does not attempt to verify the source or contents of the file before deserializing it. An attacker may be able to replace the intended file with a file that contains arbitrary malicious code which will be executed when the button is pressed.

To mitigate this, explicitly define final readObject() to prevent deserialization. An example of this is:

Example Language: **Java**

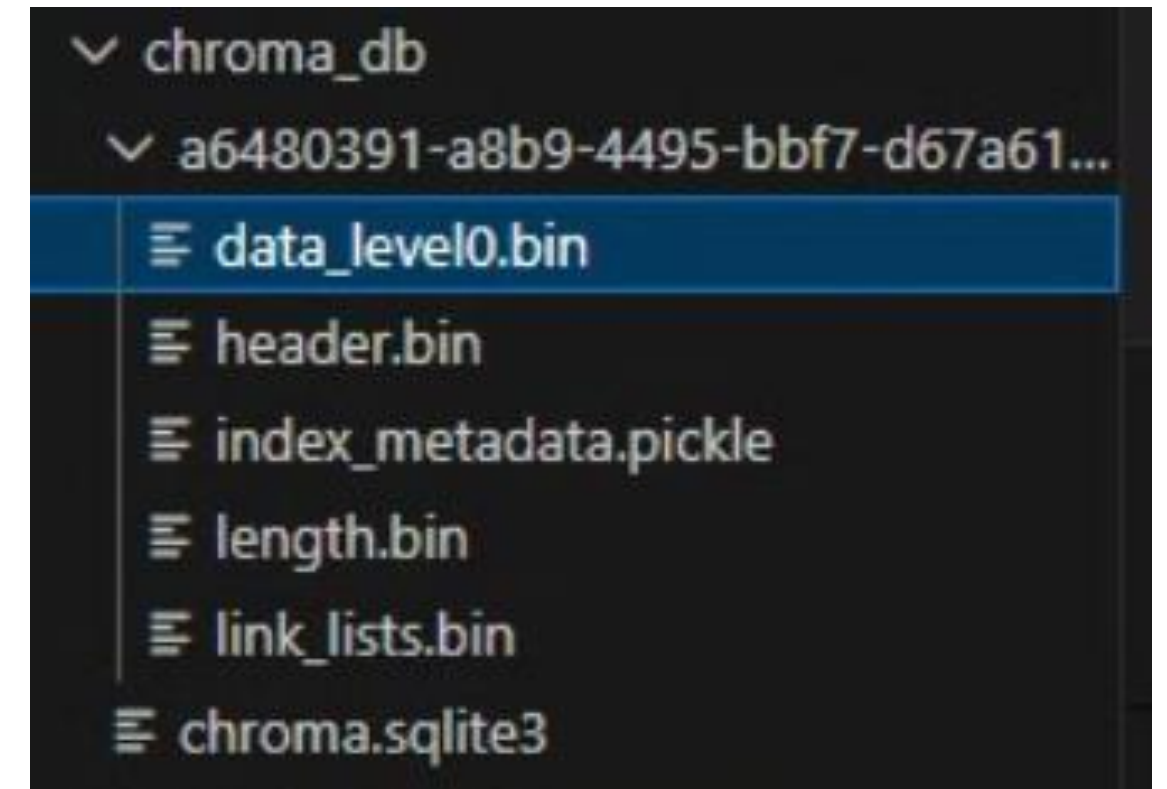
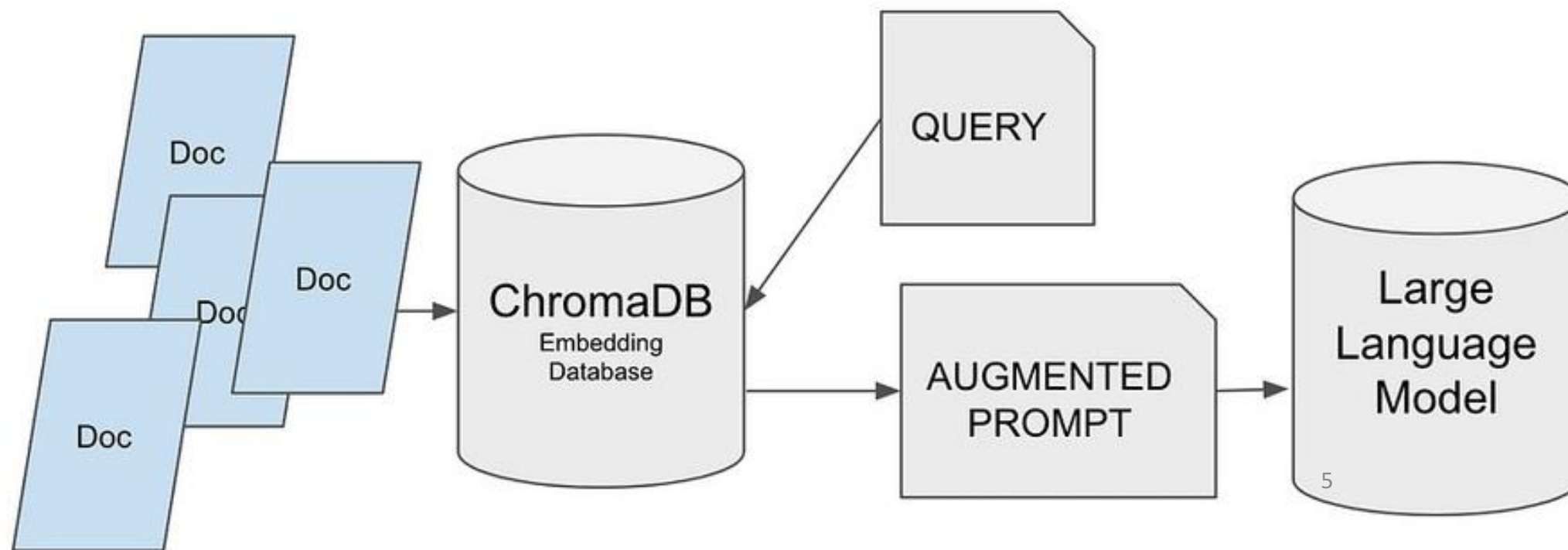
(good code)

```
private final void readObject(ObjectInputStream in) throws java.io.IOException {  
    throw new java.io.IOException("Cannot be deserialized"); }  
}
```


RAG Database Building-2

```
cwe_examples.json X
C: > Users > rock8 > Downloads > vulDeRAG-Final-main (1) > vulDeRAG-Final-main > {} cwe_examples.json > {} cwe_examples > {} 203 > {} 9
1  {
535  "cwe_examples": {
7660  },
7661  {
7662    "title": "Example 9",
7663    "code": "Suppose an algorithm for implementing an encryption routine works fine per se, but the time taken to output the result of the encryption routine depend
7664    "source": "ExampleCode",
7665    "background_color": "#FFCCCC",
7666    "is_fixed": false,
7667    "computed_styles": {
7668      "background-color": "rgb(255, 204, 204)",
7669      "display": "block",
7670      "visibility": "visible"
7671    }
7672  },
7673  {
7674    "title": "Fixed Example 10",
7675    "code": "Artificial delays may be added to ensure that all calculations take equal time to execute.",
7676    "source": "ExampleCode",
7677    "background_color": "#CCCCFF",
7678    "is_fixed": true,
7679    "computed_styles": {
7680      "background-color": "rgb(204, 204, 255)",
7681      "display": "block",
7682      "visibility": "visible"
7683    }
7684  }
7685 ],
7686 "204": [
7687   {
7688     "title": "Example 1",
7689     "code": "my $username=param('username');\nmy $password=param('password');\n\nif (IsValidUsername($username) == 1)\n{\n  if (IsValidPassword($username, $password)
7690     "source": "selenium_top_div",
7691     "background_color": "#FFCCCC",
7692     "is_fixed": false,
7693     "computed_styles": {
7694       "background-color": "rgb(255, 204, 204)",
7695       "display": "block",
7696       "visibility": "visible"
7697     }
7698   },
7699 ]
```

RAG Database Building-3



The Bake-Off: Pitting Three Approaches Head-to-Head

To measure the true impact of RAG, we compared its performance against two common prompting strategies across a series of vulnerability repair tasks.

Baseline (Zero-Shot)



The raw LLM with a simple instruction to fix the code. No examples are provided.

Few-Shot Prompting



The LLM is given a few generic examples of vulnerability analysis to guide its reasoning, but these are not specific to the code being fixed.

vulDeRAG (RAG-Enhanced)



The LLM's prompt is augmented with highly relevant examples of the *specific* vulnerability type retrieved from our expert knowledge base.

Case Study 1: The Classic Stack Buffer Overflow (CWE-121)

Vulnerable Code (`cwe121.c`)	Analysis
<pre>void host_lookup(char *user_supplied_addr){ struct hostent *hp; in_addr_t *addr; char hostname[64]; in_addr_t inet_addr(const char *cp); validate_addr_form(user_supplied_addr); addr = inet_addr(user_supplied_addr); hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET); strcpy(hostname, hp->h_name); // Vulnerable line }</pre>	<ul style="list-style-type: none">• The Flaw: The function uses <code>strcpy()</code> to copy a hostname into a fixed-size buffer (<code>hostname[64]</code>).• The Risk: If the hostname returned by <code>gethostbyaddr</code> (<code>hp->h_name</code>) is longer than 63 characters, <code>strcpy</code> will write past the buffer's boundary, causing a stack-based buffer overflow.• The Goal: A successful repair must replace the unsafe <code>strcpy</code> with a bounds-checked alternative and ideally handle potential null pointers.

Attempt 1: The "Without RAG" Fix (Few-Shot)

Model's Internal Monologue

"The problem here is that if `hp->h_name` is longer than 64 characters, the `strcpy` will cause a buffer overflow... To fix this, I should replace the `strcpy` with a safer function like `strncpy`..."

So the corrected line would be:
`strncpy(hostname, hp->h_name, sizeof(hostname));`

Generated Code & Analysis

```
// ...  
strncpy(hostname, hp->h_name, sizeof(hostname));  
// ...
```

Evaluation

- **Correct?** Partially. It correctly identifies the `'strcpy'` issue and replaces it with `'strncpy'`.
- **Robust?** No. It fails to consider two critical edge cases:
 - 1. What if `'gethostbyaddr'` fails and `'hp'` is `'NULL'`? This would cause a crash.
 - 2. `'strncpy'` may not null-terminate the string if the source is larger than the destination buffer.

Attempt 2: The vulDeRAG Solution (With RAG)

Giving the LLM Expert Context

=== Similar CWE Examples from Vector Database ===

[Example 1] CWE-119 - Example 1

```
```c
void host_lookup(char *user_supplied_addr)
){....strcpy(hostname, hp->h_name);}...
```
```

[Example 2] CWE-119 - Example 6...

[Example 3] CWE-121 - Example 2...

Model's Informed Monologue

The code contains a buffer overflow vulnerability... If `hp->h_name` is longer than 64 characters, `strcpy` will attempt to write beyond the array's bounds... Replace `strcpy` with `strncpy` and added length checks... Added error handling for `hp` being `NULL`...

Generated Code & Analysis

```
if (hp != NULL && sizeof(hostname) > 0) {
    size_t hostname_len = strlen(hp->h_name);
    if (hostname_len <= sizeof(hostname)) {
        strncpy(hostname, hp->h_name, hostname_len);
    } else {
        memcpy(hostname, hp->h_name, sizeof(hostname));
        hostname[sizeof(hostname)-1] = '\0'; // Manual
null-termination
    }
}
```

Correct? Yes.

Robust? Yes. The retrieved context prompted the LLM to consider edge cases, resulting in a much safer implementation that handles `NULL` pointers and ensures null-termination.

Case Study 2: The Subtle SQL Injection (CWE-200/CWE-89)

Vulnerable Code (cwe200.txt)

```
public BankAccount getUserBankAccount(String username, String accountNumber) {  
    //...  
    try {  
        if (isAuthorizedUser(username)) {  
            // VULNERABLE LINE  
            query = "SELECT * FROM accounts WHERE owner = "  
                + username + " AND accountID = " + accountNumber;  
            //...  
            Statement stmt = conn.createStatement();  
            ResultSet queryResult = stmt.executeQuery(query);  
            //...  
        }  
    } catch (SQLException ex) { //... }  
    return userAccount;  
}
```

Analysis

- **The Flaw:** The SQL query is constructed by directly concatenating user-controlled strings (`username`, `accountNumber`).
- **The Risk:** An attacker can inject malicious SQL fragments into the `username` or `accountNumber` fields to alter the query's logic, bypass authentication, exfiltrate data, or even drop tables.
- **The Goal:** The fix must replace string concatenation with parameterized queries (Prepared Statements) to neutralize the injection vector.

Attempt 1: The "Without RAG" Fix (Few-Shot)

Model's Internal Monologue

"The method constructs a SQL query by directly concatenating...

This is risky because if either of these inputs contain special characters... they can break the SQL syntax...

The vulnerabilities here are:

1. SQL Injection...
2. Insufficient Input Validation...

To fix the SQL injection, the code should use parameterized statements..."

Generated Code & Analysis

```
// ...  
// Validate accountNumber is numeric  
try { Long.parseLong(accountNumber); } catch  
    (NumberFormatException e) { //...  
}  
  
// Use prepared statement to prevent SQL injection  
PreparedStatement stmt = conn.prepareStatement(  
    "SELECT * FROM accounts WHERE owner = ? AND accountID = ?");  
stmt.setString(1, username);  
stmt.setString(2, accountNumber);  
ResultSet queryResult = stmt.executeQuery();  
// ...
```

Correct? Yes, it correctly uses `PreparedStatement`.

Comprehensive? Good, but not perfect. It adds numeric validation, which is a good practice but slightly secondary to the primary SQLi fix. The reasoning focuses purely on what it sees, without broader context.

Attempt 2: The vulDeRAG Solution (With RAG)

Giving the LLM Expert Context

=== CWE Vulnerability Summary ===

1. CWE-209 - Example 4 (Generation of Error Message Containing Sensitive Information)
2. CWE-209 - Example 8
3. CWE-200 - Example 4 (Exposure of Sensitive Information to an Unauthorized Actor)

Model's Informed Monologue

“Step-by-Step Explanation: 1. Identify the Vulnerability Type: CWE-209 (SQL Injection)... **CWE-200 (Resource Leak): The database connection and statement aren't properly closed...** Recommended Repair Strategy: Use prepared statements with parameterized queries to prevent SQL injection. Properly close database resources after use to avoid leaks.”

Generated Code & Analysis

```
// The reasoning implies a fix that uses PreparedStatement
// and also proper resource management (e.g., try-with-resources),
// addressing both CWE-89 and CWE-200.
try (Connection conn = dbManager.getConnection();
    PreparedStatement stmt = conn.prepareStatement("SELECT * FROM
        accounts WHERE owner = ? AND accountID = ?")) {
    // ... set parameters and execute
    stmt.setString(1, username);
    stmt.setString(2, accountNumber);
    try (ResultSet queryResult = stmt.executeQuery()) {
        // ... process results
    }
} catch (SQLException ex) {
    // ... handle exception
}
```

Evaluation

Correct? Yes.

Comprehensive? Yes. By retrieving context on related vulnerabilities ('CWE-200'), the RAG-enhanced model identified not only the primary SQLi flaw but also the secondary resource leak issue, leading to a more complete and professional recommendation.

The Results Are In: RAG Consistently Delivers More Robust Repairs

| Vulnerability | Method | Reasoning Quality | Fix Quality & Robustness |
|----------------------------|--------------------|--|---|
| CWE-121
Buffer Overflow | Few-Shot (No RAG) | Identified `strcpy` issue. | Partial. Used `strncpy` but missed `NULL` check and null-termination edge case. |
| CWE-121
Buffer Overflow | vuDeRAG (With RAG) | Identified `strcpy` and used CWE context to consider edge cases. | Excellent. Handled `NULL` pointers, ensured proper string length checking, and guaranteed null-termination. |
| CWE-200
SQL Injection | Few-Shot (No RAG) | Correctly identified SQLi via string concatenation. | Good. Correctly used `PreparedStatement`. Added input validation, which is a positive side-effect. |
| CWE-200
SQL Injection | vuDeRAG (With RAG) | Identified primary SQLi (CWE-89) and secondary resource leak (CWE-200) from context. | Excellent. Recommended `PreparedStatement` and proper resource management, providing a more holistic security fix. |

13

Across different languages and vulnerability types, providing the LLM with relevant, expert context via RAG resulted in qualitatively superior analysis and more secure, production-ready code

Beyond Code Generation: RAG is Key to Creating Specialized AI Experts

The vulDeRAG experiment demonstrates a clear conclusion: for complex, knowledge-intensive tasks like vulnerability repair, an LLM's raw intelligence is not enough.

Retrieval-Augmented Generation (RAG) acts as a force multiplier. It grounds the model in a verifiable, external body of expert knowledge, transforming it from a talented generalist into a focused specialist.

This approach is critical for responsibly deploying AI in high-stakes fields where accuracy, safety, and deep domain expertise are non-negotiable—from cybersecurity and law to medicine and engineering.

