Npgsql Security Advisory

Findings Overview

Finding 1 - SQL Injection via Protocol Message Size Overflow **3**
Observations 3
Exploitation 4
Impact 5
Recommendations 5

Context

Affected version(s): 8.0.2

Credits

Paul Gerste, Sonar (https://sonarsource.com)

Disclosure Policy

At SonarSource, we are equally driven by studying and understanding real-world vulnerabilities and helping the open-source community secure their projects. We believe disclosing vulnerabilities to affected projects reduces the cost of in-the-wild attacks and allows everybody to benefit from our research.

While demanding for all the parties involved, we believe enforcing disclosure deadlines is necessary to proceed with clear expectations with the common goal of protecting users of the vulnerable software.

All our reports are subject to a 90-day disclosure deadline: after 90 days elapse or a patch is released, whichever comes first, technical details of our security advisory will be made available to the public.

The 90-day count starts as soon as we send the technical details of our findings for the first time (this is day 1). If our initial contact attempt with the applicable security contact is unsuccessful, we try to reach out via other means such as GitHub Issues, Twitter, and different email recipients to ensure the affected project is made aware.

After acknowledging our security advisory, we will provide you with reasonable help and assistance to enable you to address the issues by providing all the information in our possession, reviewing the patches, and coordinating the public disclosure. We ask you not to fix these findings silently (e.g., without any mention in the release notes) and will only consider embargo requests on a case-by-case basis.

We reserve the right to reduce the 90-day delay and immediately release details if the security advisory is unacknowledged within 30 days or rejected to help users take defensive measures as soon as possible. We may introduce an exceptional 30-day grace period if a patch can be shared with us but could not be deployed publicly.

We apply these rules to all our vulnerability disclosures: all projects and vendors are treated equally.

Please reach out to vulnerability.research@sonarsource.com if you have any questions about this policy.

Finding 1 - SQL Injection via Protocol Message Size Overflow

Observations

Npgsql implements the Postgres protocol to talk to a Postgres database server over the network. The protocol consists of messages sent between the application and the database, and each message has a standard header consisting of a 1-byte message type and a 4-byte message length.

When a client wants to send a message to the database, it has to ensure that the message they are trying to send is not larger than what is representable with a 4-byte integer. This maximum is 2^{32}-1 bytes, or 0xffffffff in hexadecimal, or 4294967295 in decimal.

To write the message size, Npgsql first adds all parameters' lengths and stores the sum in `paramsLength`. This value is then used to calculate the `messageLength`, together with other lengths. The data type of `paramsLength` and `messageLength` is `int`, which can overflow when adding too large values:

src/Npgsql/Internal/NpgsqlConnector.FrontendMessages.cs:

```
Unset
internal async Task WriteBind(/* ... */)
{
    // ...
    var formatCodesSum = 0;
    var paramsLength = 0;
    for (var paramIndex = 0; paramIndex < parameters.Count; paramIndex++)
    {
        var param = parameters[paramIndex];
        param.Bind(out var format, out var size);
        paramsLength += size.Value > 0 ? size.Value : 0;
        formatCodesSum += format.ToFormatCode();
    }

    var formatCodeListLength = formatCodesSum == 0 ? 0 : formatCodesSum ==
parameters.Count ? 1 : parameters.Count;

    var messageLength = headerLength +
                        sizeof(short) * formatCodeListLength +
                        sizeof(short +
                        sizeof(int) * parameters.Count +
                        paramsLength +
                        sizeof(short) +
```

```
                          sizeof(short) * (unknownResultTypeList?.Length ?? 1);

    writeBuffer.WriteByte(FrontendMessageCode.Bind);
    writeBuffer.WriteInt32(messageLength - 1);
    // ...
}
```

If the sum of parameter sizes becomes too large, the `paramsLength` overflows and becomes negative since `int` is a signed data type. If the overflow is big enough, `paramsLength` can become a small positive integer. When this happens, the resulting `messageLength` is much smaller than the actual data written to the buffer.

When the database receives such a message, it will only read the specified number of bytes for this message and interpret any following bytes as a new message. Since the message size field was overflown and now only contains a small number, the database will only read this small amount of bytes and interpret it as a message.

After that, the database will continue reading the next message from the connection. Since it only consumed the start of the sent message, the bytes that it now reads and interprets as a new message were parts of the old message. Hence, the application and the database now have a different understanding of which bytes sent through the connection belong to which message.

When this happens, the integrity of the connection cannot be guaranteed anymore, which can lead to unexpected and potentially malicious behavior. In the next section, we will show how an attacker could use this to inject arbitrary SQL statements.

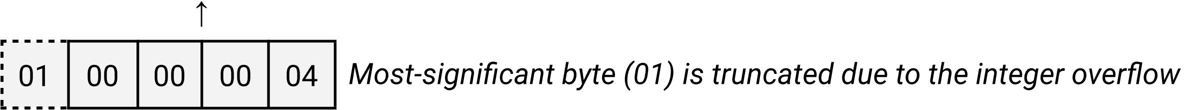
Exploitation

As an example, let's imagine the client wants to send the following query string to the database:

```python
"\x51\x00\x00\x00\x11" + "DROP TABLE users;" + "A" * 0xffffffee
```

The total length of the query string is $2^{32}+4$, causing the client to create an overlong message like the following. Note that the message length includes the field's size, so the shortest valid message length is 4. The overflown length field is marked in red:

Message 1											
Type	Length				Query string						0xffffffee more bytes
51	00	00	00	04	51	00	00	00	11	"DROP TABLE users;"	



Most-significant byte (01) is truncated due to the integer overflow

But the database will parse it as multiple messages:

Message 1					Message 2						Message 3
Type	Length				Type	Length				Query string	0xffffffee more bytes
51	00	00	00	04	51	00	00	00	11	"DROP TABLE users;"	

As a result, the database parses message 1 and ignores it because its query string is empty. After that, the database parses message 2 and executes its SQL statement, dropping the `users` table. The remaining large amount of data is parsed and interpreted as more messages.

In a real-world application, attackers cannot control a full query that the application sends to the database as-is. However, controlling a string parameter of a parameterized query can be enough to perform the same attack. It will require more fine-tuning and message crafting, but the result is the same as in the example above: executing arbitrary, attacker-controlled SQL statements.

Due to string size limitations, the attack will not work when passing the large payload as a single string. An attack can still succeed if at least 8 string parameters of a single query are user-controlled.

We will not describe how to perform an attack in a real-world scenario, as it would go beyond the scope of this advisory. We attached a proof of concept for demonstration purposes that performs an attack against a more realistic query, simulating a real-world scenario. To run it, please follow these steps:

1. Unzip it
2. Adjust the connection options in `connString`
3. Run via: `dotnet run Program.cs`

Impact

The message size overflow allows an attacker to inject arbitrary Postgres protocol messages into an already-established connection between the application and the database. Since

authentication happens at the beginning of a connection (if at all), the injection occurs when the connection is already authenticated.

The final impact of such a vulnerability depends on the application that uses the library, what data it stores in the Postgres database, and so on. Attackers must also find a way to feed 4 gigabytes of data into the application without causing a validation error or any other exception before the data is used to build and send a query.

To determine the real-world impact of this kind of vulnerability, we tested multiple open-source applications that use Npgsql or similar libraries that suffer from the same issue. We found several of them to be vulnerable, some of them even in their default configuration. Attackers can fully take over those vulnerable applications by sending a crafted request.

Recommendations

We recommend checking for integer overflows before writing any integer field of the Postgres protocol. This includes every message's message size field, parameter counts, string lengths, etc.

To perform such checks, a larger data type is required for `paramsLength`, `messageLength`, and similar variables. Otherwise, a size check on a 32-bit integer cannot detect a potential overflow because it already happened. Instead, use a data type that can hold larger numbers than 2^{32}-1, such as `ulong`.

Note that Postgres defines a maximum message length of `0x3fffffff` for the message size field in particular. It is best to restrict messages to this size instead of the potential maximum of 2^{32}-1. Other fields might have different restrictions, which you can find in the Postgres documentation.