PostgreSQL Auditing

Last updated by | Lisa Liu | Nov 6, 2020 at 10:36 AM PST

PostgreSQL Auditing

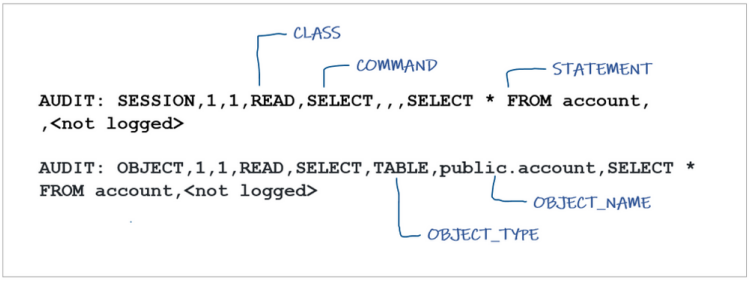
Friday, February 7, 2020
12:12 PM

Auditing: Know what's going on in your Postgres database

Today it's almost unsurprising to see another headline related to data breaches or data privacy. This affects you personally as a customer of those companies - you want someone to take good care of your data. It also affects you professionally; as a person who works with data, you are that 'someone' for your customers. Good audit logging is one important tool in the belt of a security-aware data professional.

There are different kinds of security tools. Some are preventative, like role-based access control and authentication policies. Audit logging, on the other hand, is in the investigative section of the toolbox because a log is a record of a past action. (Although knowing that their actions will be tracked may inspire employees to be more careful). Audit logs help you identify the who, what, and when of data access, and to an extent the where and how. Having audit logs available for review is a key requirement for compliance in certain regulated industries, but also increasingly in other industries as data privacy laws like GDPR take effect. If worst comes to worst and there is a breach, these logs help you trace the extent and ideally the culprits. Knowing these exact details can help you prevent such events in the future, by clearly identifying where vulnerabilities were exploited. On the bright side, with alerting built on audit logs you could identify unusual activity and stop a breach early.

As a Postgres user you may have heard of the PostgreSQL Auditing Extension. It is more commonly known by its shorter name: pgAudit. How does pgAudit help you know the who, what, when of data access? Let's look at the format of pgAudit logs:



You can see that pgAudit is very good at the 'what'. The CLASS categorizes the kind of statement (READ, WRITE, DDL etc) and COMMAND tells you what sub-class it is. The OBJECT_TYPE and OBJECT_NAME identify whether a table, view etc was accessed and which one it was. The STATEMENT field is a key factor that sets pgAudit apart from using Postgres statement logging only. You may have been asking, why can't I just use log_statement=all as my audit logging? It's because pgAudit shows you not only the query that was sent by the client, but the actual statement executed on the back end.

(For a full breakdown of the log format visit the pgAudit documentation).

You've probably noticed that pgAudit alone does not give us the who or the when. pgAudit relies on Postgres's log_line_prefix parameter for this information. Log_line_prefix sets the initial text of each log. Postgres's default prefix is '%m [%p] ' which logs time stamp and process ID. For auditing, it is helpful to enhance this prefix. For example, the setting `t=%m u=%u db=%d pid=[%p]: ` gives you a log like:

t=2019-10-01 01:21:41.377 UTC u=alice db=postgres pid=[96]: LOG: AUDIT: SESSION,1,1,READ,SELECT,,,SELECT 1,<not logged>

Labels like 'u' and 'd' indicate which field is which; you can remove or customize to suit your taste. Now you can see who executed the statement (user), when they did so (timestamp), and in what database.

The process ID can be handy for tracking all events for a particular login. If you see suspicious activity you can use the process ID to trace other statements executed during that process. In fact, the process ID helps us with another important 'w' - where - which is not available with pgAudit alone. By turning on Postgres's log_connections parameter (default is off) and logging process ID, you can trace an unusual statement back to the IP address that initiated the process.

t=2019-10-01 06:25:35.766 UTC u=[unknown] db=[unknown] pid=[108]: LOG: connection received: host=111.107.109.5 port=5432 pid=108
t=2019-10-01 06:25:36.110 UTC u=bob db=postgres pid=[108]: LOG: connection authorized: user=bob database=postgres
t=2019-10-01 08:25:36.110 UTC u=bob db=postgres pid=[108]: LOG: AUDIT: SESSION,5,1,READ,SELECT,,,SELECT * FROM accounts;,<not logged>

As you would imagine, turning on auditing generates a large volume of logs. This is a good thing, but also a tough thing. Tough because it can be hard to act on that volume of information. pgAudit offers some settings that can help control the volume of logs that are generated. There are two types of logging and you can enable one or both: session-level and object-level auditing.

Session auditing logs all statements that are executed by users. You can use the pgaudit.log parameter to scope this down to certain classes of statements: READ, WRITE, FUNCTION, ROLE, DDL, MISC. Select all the classes or the subset that is most pertinent for you.

On the other hand, object auditing records statements for specific relations. For example, you can have an audit trail for a table that stores credit card information, but not have auditing for a table that stores non-personally identifiable data. (Of course, account for your compliance requirements when deciding). Using this finer grained audit option requires some additional setup. You need to register a Postgres role that has permissions on the relation in question. As new relations are created, you need to be sure you are including the relevant ones in the auditor role's permissions. Also, object auditing logs only selects, inserts, updates, and deletes.

Depending on your industry, it is safer to err on the side of logging more. Storage is not expensive these days. On the other hand, not logging something you later realize is pertinent can be an expensive decision.

You can put the logs in long term storage and only come back to them after an incident or when external auditors visit. But there is the potential to do more with your audit logs. Instead of reducing the volume of logs generated, pipe them to a service that offers good search tooling (if you're an Azure user, check out Azure Monitor Logs). Search will help you quickly narrow down the pertinent lines from the thousands of files that are generated. Even better, if the service offers alerting, you can set up notifications for when certain classes of operations or actions on restricted objects take place. That can be the difference between you stopping a breach early and finding out about it months later.

The pgAudit documentation provides step by step guidance on how to set up the extension. If you're interested in pgAudit on Azure Database for PostgreSQL, visit our auditing article. Keep in touch.

From <https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/auditing-know-what-s-going-on-in-your-postgres-database/ba-p/885243>

Created with Microsoft OneNote 2016.

How good have you found this content?