ProFTPD Vulnerability POCs

Proof-of-concept demonstrations for CVE-2026-42167, a SQL injection vulnerability in ProFTPD's `mod_sql` logging pipeline that allows an unauthenticated attacker to execute arbitrary SQL — and to inject backdoor users into the FTP authentication database or achieve remote code execution on the database host.

All POCs are PostgreSQL-specific, but backdoor user ones work with MySQL and sqlite backends with some modifications to the injected query (which we're leaving as an exercise for the reader).

- This vulnerability was discovered by ZeroPath Research. Our technical blog contains more details.

Vulnerability

`is_escaped_text()` bypass in mod_sql SQLLog (CVE-2026-42167, CWE-89)

ProFTPD's `mod_sql` logs every FTP command through the `SQLLog` / `SQLNamedQuery` mechanism. When resolving format variables like `%U` (original username) or `%{basename}` (filename component), the framework calls `is_escaped_text()` to decide whether escaping is needed — and skips escaping entirely for any value that starts and ends with a single quote and contains no internal single quotes (e.g. `'|| (SELECT 1) ||'`). The check is purely syntactic and runs on raw, attacker-controlled input from the FTP session, so it cannot distinguish "pre-escaped by trusted code" from "crafted by an attacker to look pre-escaped."

The standard documented pattern wraps format variables in single quotes for SQL safety:

```
SQLNamedQuery log_activity INSERT "'%U', '%r', '%m'" activity_log
SQLLog        *           log_activity
SQLLog        ERR_*       log_activity
```

When an attacker supplies a value of the form `'<payload>'`, the substitution produces `''<payload>''` in the final SQL — the empty string literals close the surrounding quotes and the payload runs as raw SQL. With PostgreSQL (`PQexec`) or SQLite (`sqlite3_exec`) the stacked-query support means the payload can be a full `INSERT`, `UPDATE`, `CREATE TABLE`, or `COPY TO PROGRAM` statement.

When is a server exploitable?

The vulnerable code is in `contrib/mod_sql.c` (the shared SQL framework), not in any specific backend, so all SQL backends are affected — but the attack surface depends on how the admin has configured logging. A server is exploitable when **both** of the following are true:

1. The admin has defined a `SQLNamedQuery INSERT` (or `UPDATE`) whose format string interpolates one of these attacker-controllable variables wrapped in single quotes — e.g. `"'%U', '%m'"`. The variables that come from attacker input are:

Var	Meaning
`%A`	anonymous-login password string
`%J`	command parameters (everything after the verb)
`%S`	response message string (may include attacker input echoed back in errors)
`%U`	original username from `USER` (set before auth, available even on failed login)
`%d`	directory name (last path component)
`%l`	RFC 1413 ident response (attacker-controlled if they run identd)
`%m`	FTP method/verb (attacker chooses which command to send)
`%r`	full FTP command (verb + args)
`%u`	authenticated username

Var	Meaning
`%{basename}`	filename component of the path argument, no directory prefix

`%f`, `%F`, and `%D` look attacker-controlled but always resolve to absolute paths beginning with `/`, so they cannot match the `is_escaped_text()` start-with-`'` requirement.

2. The admin has bound that `SQLNamedQuery` to a `SQLLog` directive for an FTP command (or command class) the attacker can reach. The widely documented `SQLLog *` and `SQLLog ERR_*` wildcards are the broadest case and what makes the `%U` path pre-auth: `ERR_*` fires on a failed `USER`, so no credentials are needed. Per-command directives like `SQLLog STOR` cover any authenticated user.

What can an attacker do?

The bypass turns the logging path into an arbitrary-SQL primitive on the backend. The two highest-impact scenarios:

- **Inject a backdoor user with arbitrary privileges (auth bypass).** ProFTPD's SQL auth backend reads usernames, password hashes, uid, gid, homedir, and shell from the same `users` table the logging INSERT can now write to. A stacked `INSERT INTO users` plants an attacker-chosen account — `uid=0`, `homedir=/`, plaintext password — and the attacker then logs in normally with full filesystem access through the FTP daemon. Reachable pre-auth via the `%U` + `SQLLog ERR_*` path, or post-auth via any attacker-controlled variable bound to a command the attacker can issue (e.g. `%{basename}` + `SQLLog STOR`). Works on PostgreSQL and SQLite (both support stacked queries).

- **Remote code execution on the database host via `COPY TO PROGRAM`.** PostgreSQL's `COPY (SELECT …) TO PROGRAM '<cmd>'` runs `<cmd>` through a shell on the database server. A stacked-query injection that issues `COPY TO PROGRAM` gives arbitrary OS command execution as the postgres OS user, which is enough for credential exfiltration, lateral movement, and persistence. Reachable through the same trigger paths as the backdoor case (pre-auth via `%U` or post-auth via `%{basename}` etc.). PostgreSQL-specific, and requires the ProFTPD DB role to have superuser privileges (the prerequisite for `COPY TO PROGRAM`) — common in single- tenant deployments where the role is the DB owner.

POC choices

The POCs in this repository target **PostgreSQL** specifically — the strongest case, since `PQexec()` supports stacked queries and `COPY TO PROGRAM` provides direct OS command execution. The bypass itself is in the shared SQL framework and affects all backends:

- **PostgreSQL** (covered here): stacked queries via `PQexec`, RCE via `COPY TO PROGRAM`.
- **SQLite**: stacked queries via `sqlite3_exec`, runs under `PRIVS_ROOT` in the proftpd worker — backdoor injection works the same way; RCE primitive differs (no `COPY TO PROGRAM` equivalent, but writable `users` table on a root-process SQL backend is plenty).

- **MySQL**: the bypass fires the same way, but reaching the `users` table or OS execution from inside the single logging INSERT is harder. Single-statement manipulation (subquery exfil in a VALUES slot, time-based blind, error-based blind) is straightforward; turning that into a write to a *different* table requires getting around several obstacles:
  - `mod_sql_mysql` calls `mysql_real_query()` without `CLIENT_MULTI_STATEMENTS`, so a trailing `; INSERT INTO users …` is rejected by the connection.
  - An attacker would have to figure out how to work around this limitation to write to the `users` table or a file on disk.

Within the PostgreSQL setup, the POCs demonstrate two representative trigger paths. Other combinations from the table above are equivalent in principle:

- **Pre-auth via `%U` + `USER`.** `SQLLog ERR_*` makes this fully unauthenticated.
- **Post-auth via `%{basename}` + `STOR`.** Requires any FTP credentials but no privileges beyond the ability to upload a file.

Repository Contents

- **`setup/`** — Automated environment setup. Clones the ProFTPD source tree pinned to the commit these findings were reported against, builds the daemon with `mod_sql` + `mod_sql_postgres`, and stands up a Docker Compose cluster (ProFTPD + PostgreSQL) with seed data and the vulnerable SQLLog configuration.

- **`pocs/`** — Five exploit scripts. All are self-contained Python scripts requiring only the standard library.

  - `preauth_user_backdoor.py` — Pre-auth `%U` trigger → backdoor user (uid=0, homedir=`/`) injected into the auth DB. No credentials, no DB superuser required. This POC is PostgreSQL-specific, but the issue can be exploited with either mysql or sqlite backend as well.
  - `preauth_user_rce.py` — Pre-auth `%U` trigger → RCE on the PostgreSQL host via `COPY TO PROGRAM`. No credentials. Requires the ProFTPD DB role to be a PostgreSQL superuser.
  - `postauth_stor_backdoor.py` — Post-auth `%{basename}` trigger → backdoor user injected. Requires any authenticated FTP user. This POC is PostgreSQL-specific, but the issue can be exploited with either mysql or sqlite backend as well.
  - `postauth_stor_rce.py` — Post-auth `%{basename}` trigger → RCE on the PostgreSQL host. Requires any authenticated FTP user and a PostgreSQL superuser DB role.
  - `postgres_blind_dump.py` — Pre-auth `%U` trigger → time-based blind extraction of the auth `users` table. Doesn't use stacked queries, so works on deployments where the proftpd DB role has only minimal privileges (just `INSERT` on the log table) and the backdoor / RCE POCs above would fail closed. Pulls every byte of every column — including the `passwd` column — by binary-searching one bit at a time via `pg_sleep()`. Mirrors what sqlmap would automate, hand-rolled with no third-party dependencies.

Instructions

Prerequisites: Docker, Git, Python 3.10+, and uv.

```
cd setup
./setup.sh
```

Releases

No releases published

Packages

No packages published

Contributors 1

johnatzeropath

Languages

Python 86.9% **Shell** 13.1%