
DEFCON
RUSTY
PEARLS
VARONIS



## TAL PELEG

- Cloud security research team lead at Varonis
- Full stack security hacker
- Currently specialize in IaaS research
- Excited about breaking things and fixing them again, music, and astrophysics



## COBY ABRAMS

- Cloud security researcher at Varonis
- Experience in various cybersecurity fields
- Currently specialize in IaaS research
- Passion for teaching cybersecurity



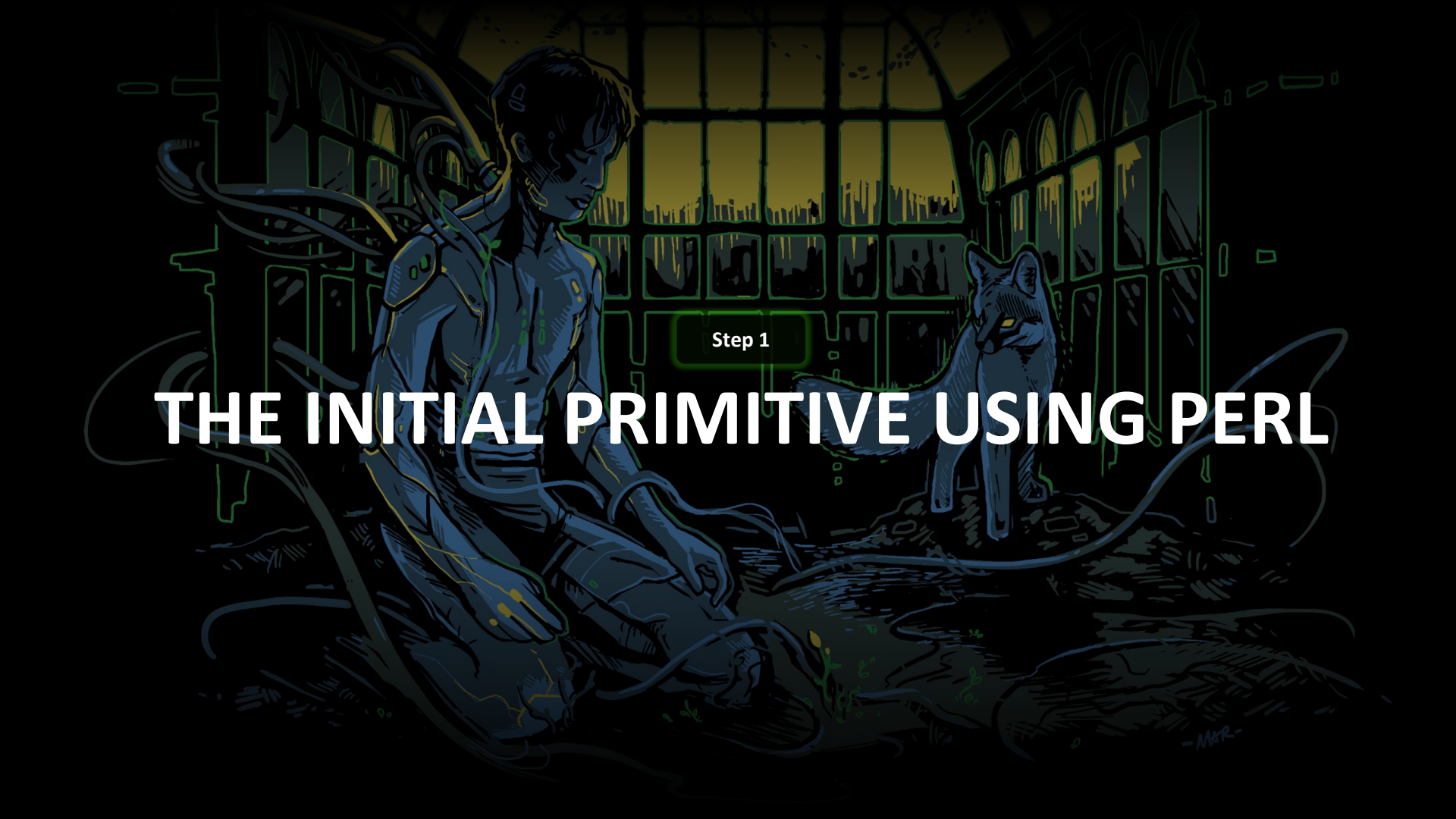

# What's About to Happen

1. We are going to talk about hacking PostgreSQL (CVE-2024-10979) and a cloud hosted PostgreSQL!
2. How we went about the research and why it is interesting
3. Hacking Takeaways for the cybersecurity community

# Background

## PostgreSQL

A popular robust and open-source relational database used in a variety of applications including cloud environments.


Step 1
THE INITIAL PRIMITIVE USING PERL

# Postgres Language Extensions

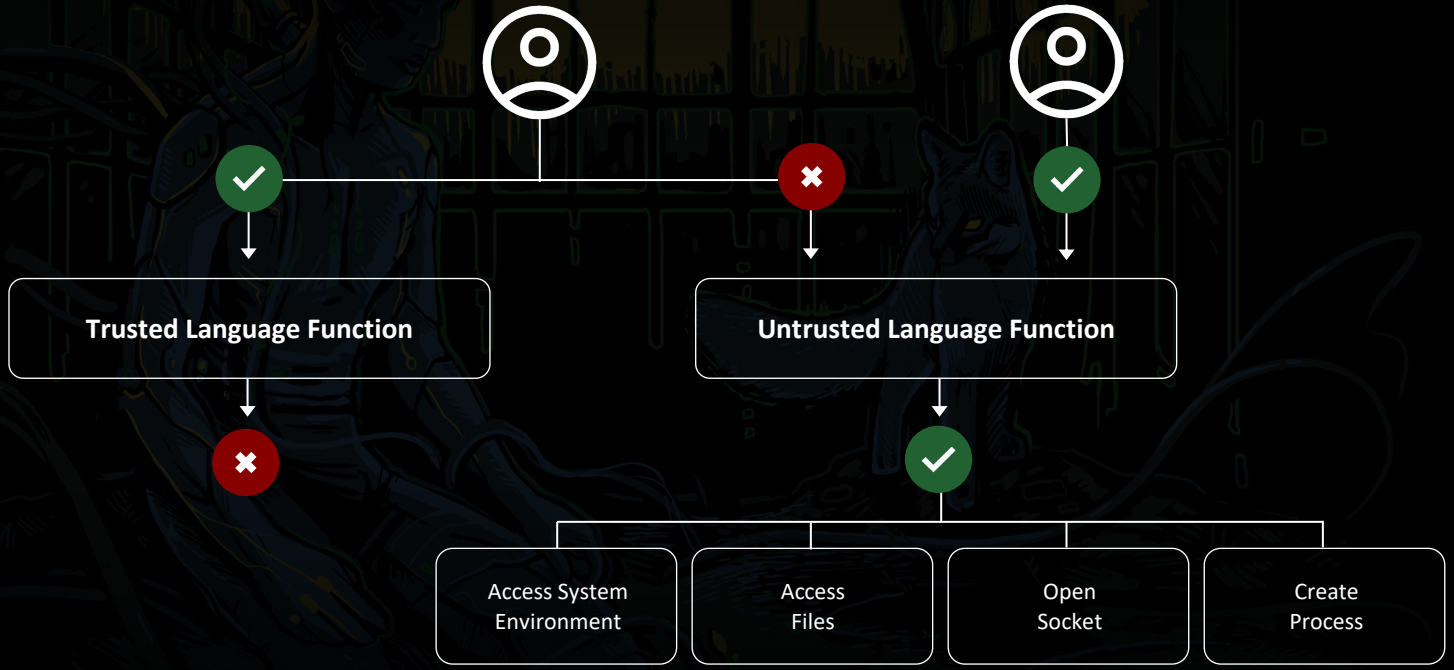
Postgres allows writing functions in various languages

- C (built-in)
- Perl (built in)
- Python
- Rust

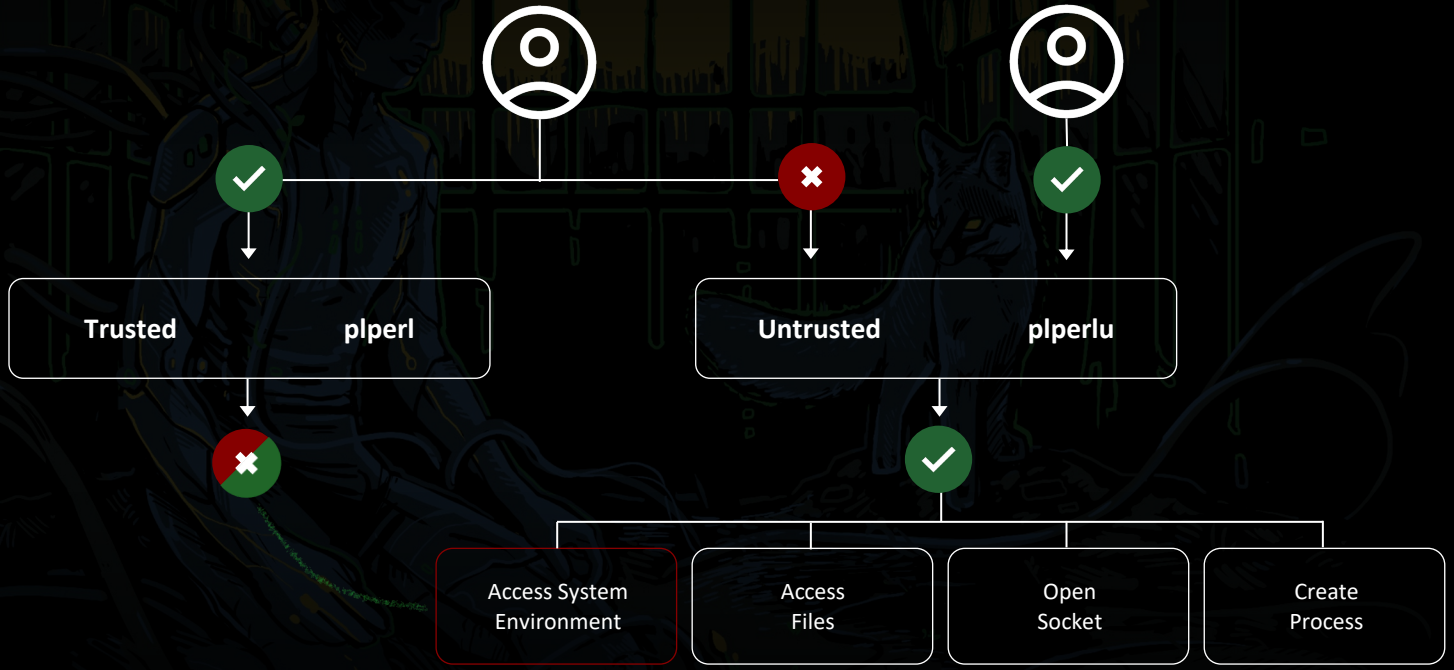
Can be used in two modes

- Trusted
- Untrusted

# Trusted vs. Untrusted Functions



# Trusted vs. Untrusted Functions



# PL/Perl Primitive

- Perl implements “magic” variables tied to special functionality
- %ENV – used to access and set environment variables
  - *$ENV{PATH} = ‘/bin’;*
- The implementation for these is deep within the Perl language
- PL/Perl did not override this behavior in trusted mode

```
Perl_magic_setenv(pTHX_ SV *sv, MAGIC *mg)
{
    STRLEN len = 0, klen;

    const char *key;
    const char *s = "";

    SV *keysv = MgSV(mg);

    if (keysv == NULL) {
        key = mg->mg_ptr;
        klen = mg->mg_len;
    }
    else {
        if (!sv_utf8_downgrade(keysv, /* fail_ok */ TRUE)) {
            ck_warner_d(packWARN(WARN_UTF8), "Wide character in %s", "setenv key
        }

        key = SvPV_const(keysv,klen);
    }

    PERL_ARGS_ASSERT_MAGIC_SETENV;

    SvGETMAGIC(sv);
    if (SvOK(sv)) {
        /* defined environment variables are byte strings; unfortunately
           there is no SvPVbyte_force_nomg(), so we must do this piecewise */
        (void)SvPV_force_nomg_nolen(sv);
        (void)sv_utf8_downgrade(sv, /* fail_ok */ TRUE);
        if (SvUTF8(sv)) {
            ck_warner_d(packWARN(WARN_UTF8), "Wide character in %s", "setenv");
            SvUTF8_off(sv);
        }
        s = SvPVX(sv);
        len = SvCUR(sv);
    }
    my_setenv(key, s); /* does the deed */
```

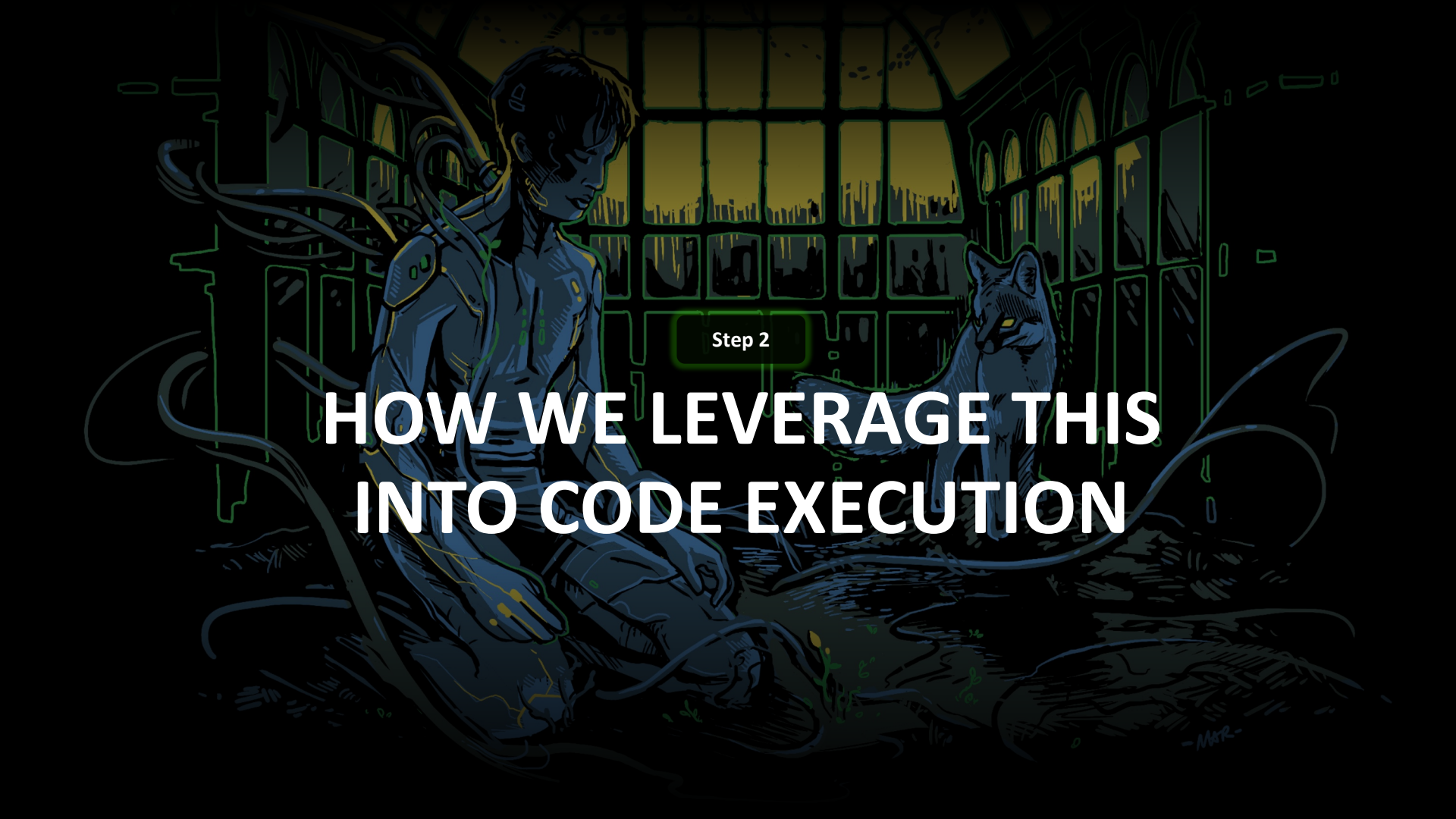
# Modifying Environment Variables in PL/Perl

```
postgres=# CREATE FUNCTION set_env(text, text)
RETURNS void AS
$tag$
$ENV{$_[0]} = $_[1];
$tag$ language plperl;
CREATE FUNCTION
```

```
postgres=# select set_env('PATH', '/tmp/mypath');
 set_env
---------

(1 row)
```




Step 2
HOW WE LEVERAGE THIS INTO CODE EXECUTION

# Another Language Extension: PL/Rust

- Now we need something to read our edited environment variables and execute them
- PL/Rust, a compiled language, and also a trusted language, gave us an answer

```rust
/// Builds a `Command::new("cargo")` with necessary environment variables pre-configured
pub(crate) fn cargo(
    cargo_target_dir: &Path,
    cross_compilation_target: Option<CrossCompilationTarget>,
) -> eyre::Result<Command> {
    let mut command = Command::new("cargo");

    configure_path(&mut command)?;
    configure_rustc(&mut command);
    configure_pg_config(&mut command, cross_compilation_target);
    sanitize_env(&mut command);

    command.env("CARGO_TARGET_DIR", &cargo_target_dir);
    if cfg!(target_os = "macos") {
        command.env("RUSTFLAGS", "-Clink-args=-Wl,-undefined,dynamic_lookup");
    } else {
        // Don't use `env_remove` to avoid inheriting rustflags via the normal
        // search.
        command.env("RUSTFLAGS", "");
    }

    Ok(command)
}
```

```rust
fn sanitize_env(command: &mut Command) {
    command.env_remove("DOCS_RS"); // we'll never be building user function on https://docs.rs
    command.env_remove("PGRX_BUILD_VERBOSE"); // unnecessary to ever build a user function in verbose mode
    command.env_remove("PGRX_PG_SYS_GENERATE_BINDINGS_FOR_RELEASE"); // while an interesting idea, PL/Rust user functions
    command.env_remove("CARGO_MANIFEST_DIR"); // we are in the manifest directory b/c of `command.current_dir()` above
    command.env_remove("OUT_DIR"); // rust's default decision for OUT_DIR is perfectly acceptable to PL/Rust
    command.env_remove("RUSTC_WRAPPER"); // plrustc doesn't like being invoked with RUSTC_WRAPPER set.
    command.env_remove("RUSTC_WORKSPACE_WRAPPER"); // ditto.
}
```

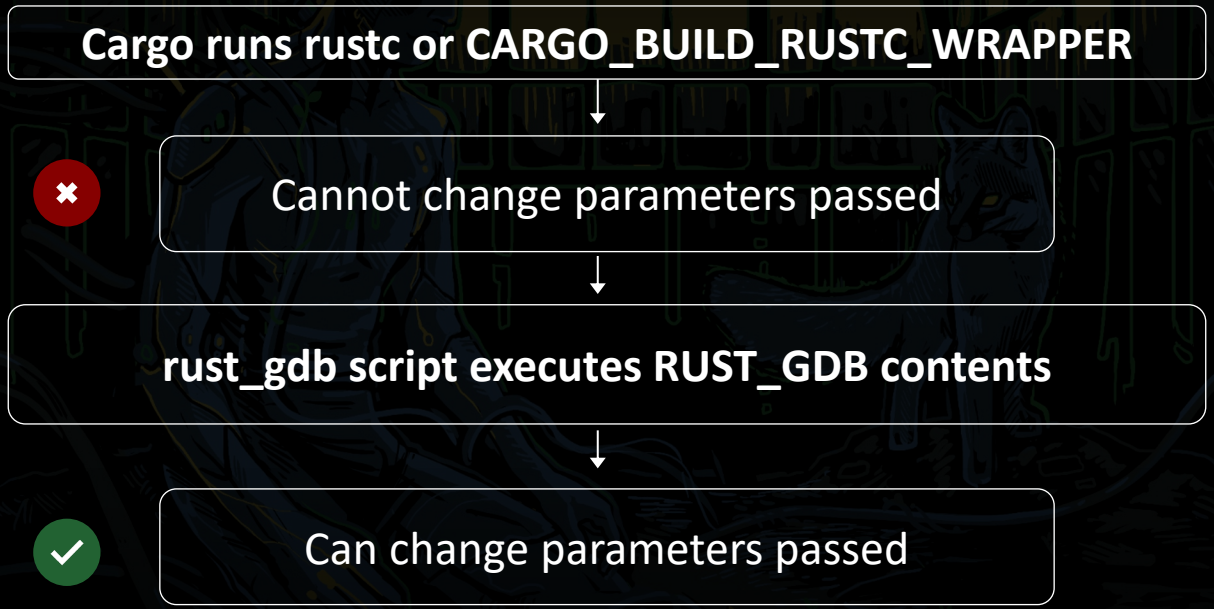
- RUSTC_WRAPPER is removed before invocation
- CARGO_BUILD_RUSTC_WRAPPER, however, is not removed
- rust_gdb script executes the RUST_GDB parameter

`build.rustc-wrapper`

- Type: string (program path)
- Default: none
- Environment: `CARGO_BUILD_RUSTC_WRAPPER` or `RUSTC_WRAPPER`

Sets a wrapper to execute instead of `rustc`. The first argument passed to the wrapper is the path to the actual executable to use (i.e., `build.rustc`, if that is set, or `"rustc"` otherwise).

# Flow


Cargo runs rustc or CARGO_BUILD_RUSTC_WRAPPER
Cannot change parameters passed
rust_gdb script executes RUST_GDB contents
Can change parameters passed


Demo
LET’S SEE THIS IN ACTION!

# So What's Going On?

- We know you probably have some questions…
- How did we find this vulnerability?
- Why did we choose to exploit PL/Perl and PL/Rust?
- You promised us a cloud RCE!

# Choosing the Objective

- AWS RDS came up during a different research project
- I was looking for a PE
- What better PE than… RCE?
- And so, the end-goal was set – RCE on an RDS instance

# Acquiring a Target

- PL/Perl is in the official Postgres repo and is enabled on RDS
- Perl is from the 80’s
- Perl is a little weird (no offence, we like weird, weird is good)
- Perl and PL/Perl are opensource

# Getting It Over the Line

- Postgres provides some challenges – Background Workers
- After finding the primitive, we looked at opensource extensions, specifically those allowed in AWS RDS
- GitHub's search/Copilot to the rescue
  - *Getenv*
  - *System*
  - *Execv*
  - *And in rust, Command*



What postgres extensions run external binaries?

# Yes, We Briefly Ran Code On RDS

- Why attack our own cloud database?
- What did we find?
  - *A very limited environment*
  - *The effectiveness of AWS incident response*
- Bug disclosed to PostgreSQL and AWS on October 10th 2024 and was patched November 11th 2024



# End Users

- Update your databases
- Least privileges
- Limit extensions




Incorrect control of environment variables in PostgreSQL PL/Perl allows an unprivileged database user to change sensitive process environment variables (e.g. `PATH`). That often suffices to enable arbitrary code execution, even if the attacker lacks a database server operating system user. Versions before PostgreSQL 17.1, 16.5, 15.9, 14.14, 13.17, and 12.21 are affected.

The PostgreSQL project thanks Coby Abrams for reporting this problem.

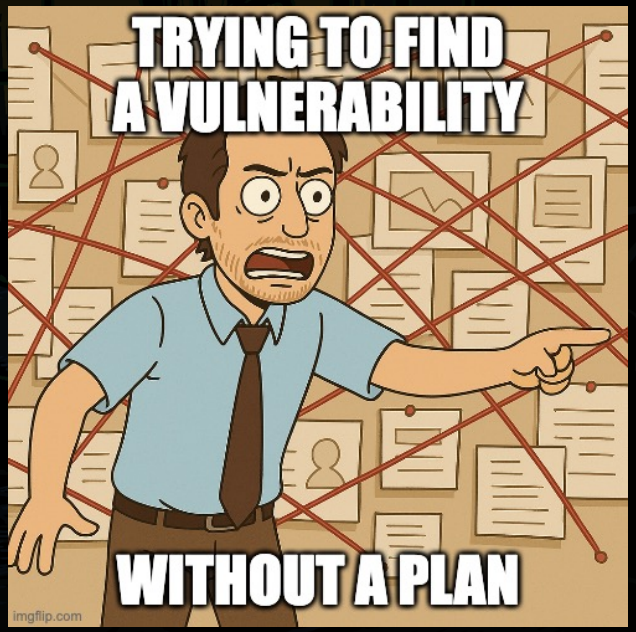
# Cloud Providers

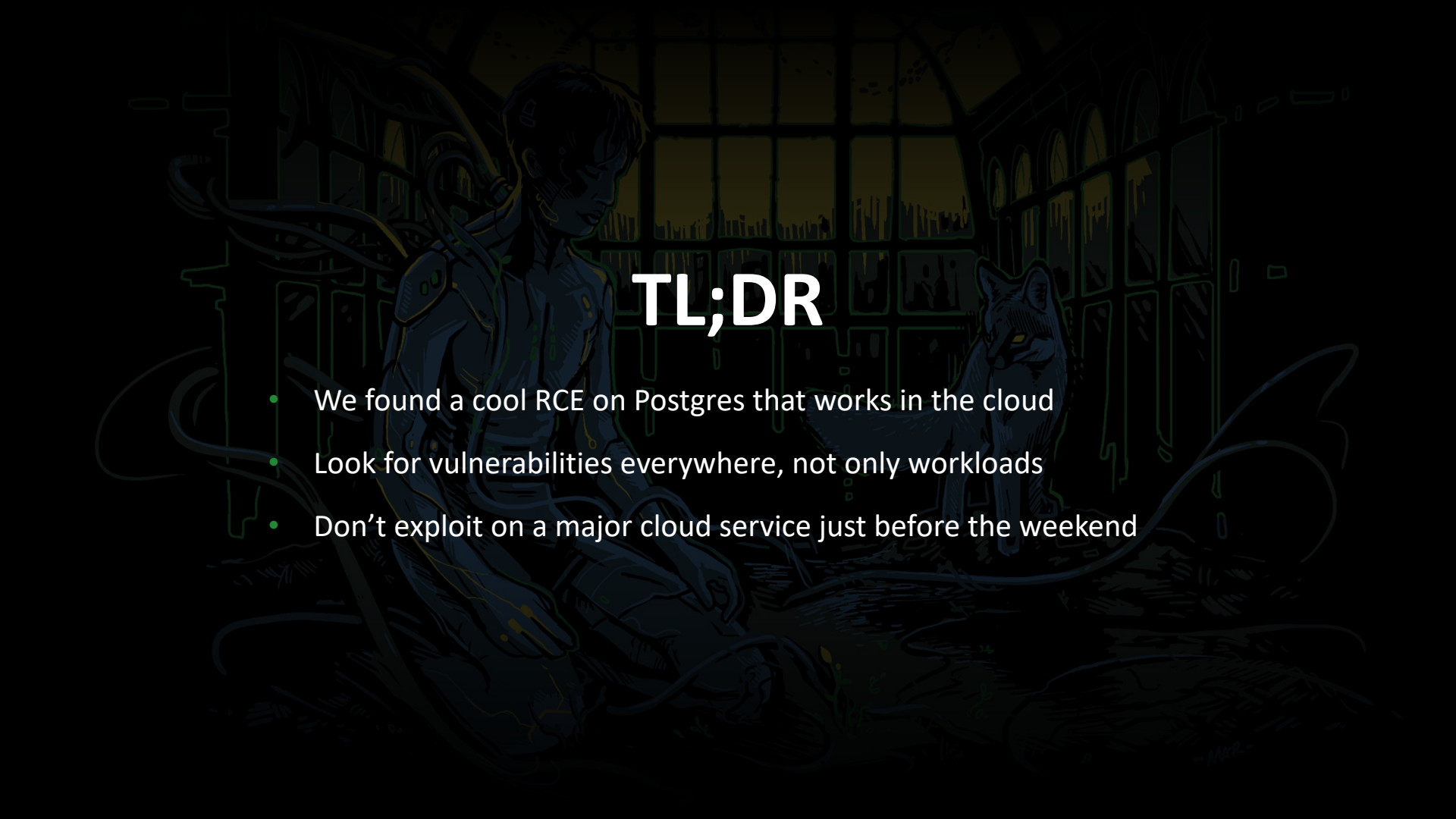
- Managed database backends are prone to these vulnerabilities
  - *The Speckle Umbrella story - Imre Rad*
  - *Azure Cosmos for postgres*
- Responsible community research
  - *Container escapes*
  - *Credentials*
  - *Network*
  - *Cross-account*



# Researchers

- Environment variables are fun
- Prepare before running sensitive payloads
- Deny lists are usually a (good?) bad sign
- Looking for obscure built-in features generally pays off
- Set your goals before researching!





# TL;DR

- We found a cool RCE on Postgres that works in the cloud
- Look for vulnerabilities everywhere, not only workloads
- Don’t exploit on a major cloud service just before the weekend


Thank You!
Scan to visit the Varonis Threat Labs blog
VARONIS