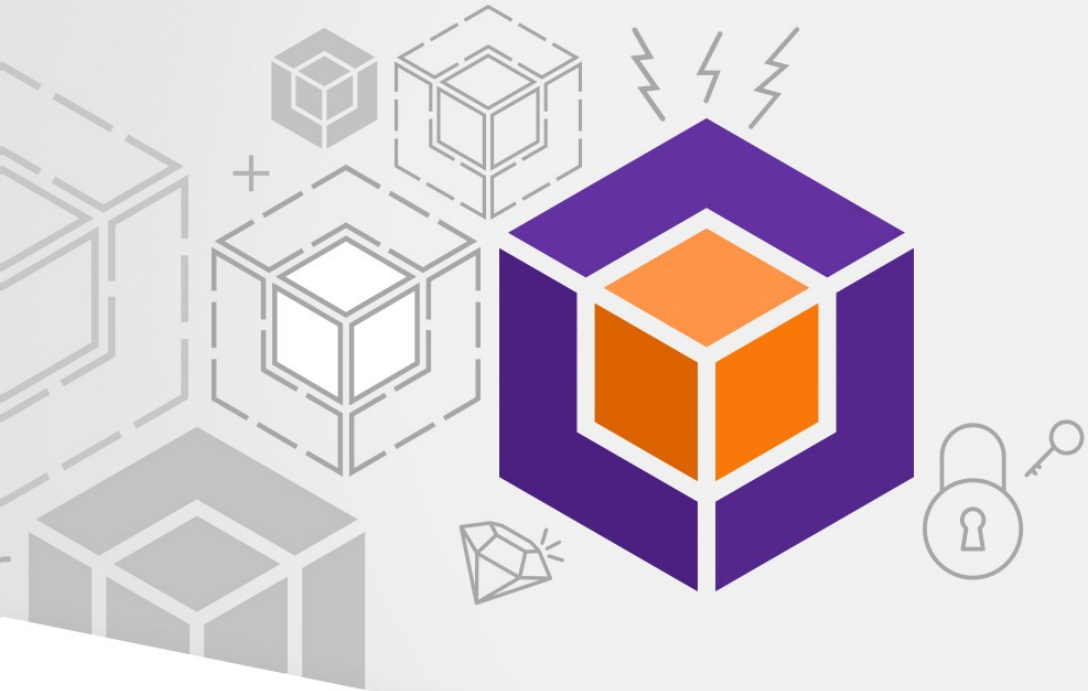
YottaDB

April 27, 2019



Echo [Terminal] 23: Watching Terminals for Fun and Profit

Charles Hathaway

Step 0

- Install YottaDB (https://yottadb.com/product/get-started/):
  - mkdir /tmp/tmp; cd /tmp/tmp
  - wget https://gitlab.com/YottaDB/DB/YDB/raw/master/sr_unix/ydbinstall.sh
  - chmod +x ydbinstall.sh
  - sudo ./ydbinstall.sh --utf8 default --verbose
- Install rust: curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Outline

YottaDB

- The problem
- File descriptors and standard IO
- System calls
- Solution design
- Implement
- Demo
- Q&A

YottaDB

The Problem

Recording terminals for audit and profit

Hackers (1995)

YottaDB

- This kid is hacking “The Gibson”

Hackers (1995)

YottaDB

- This is what hacking looks like, btw

Hackers (1995)

YottaDB

- But he's caught! And this guy calls his boss, who announces:

  "Fear not, I, iz here…. Let's echo 23 and see what's up"

So the problem

YottaDB

- How does one echo terminal 23?
  - echo “terminal 23”
  - cat /dev/pts/23
  - cat /proc/fd/0
  - cat /dev/tty0
- None of these work :’(

YottaDB

Echo [Terminal] 23

How do we echo a terminal?

Understanding file descriptors

YottaDB

- A file descriptor is a number unique to an application which represents a file handle
  - Defined by the POSIX API
  - Files for everything; network connections, standard input, standard output, files

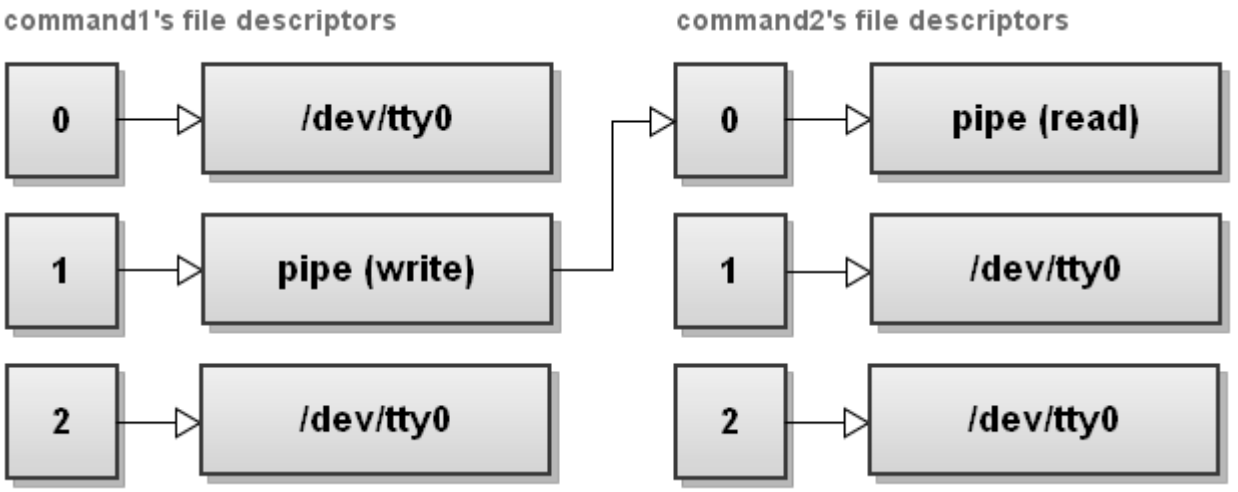
Understanding file descriptors

YottaDB

- Most applications have these 3 file descriptors:
  - FD 0: standard input
    - Remember writing your first “Hello <name>” application? This is where you read input from
  - FD 1: standard output
    - This is where all the “Hello worlds” go
  - FD 2: standard error
    - This where all the “Oh noo!”s go

Understanding file descriptors

- File descriptors can point to anything; even other file descriptors



http://www.catonmat.net/blog/bash-one-liners-explained-part-three/

Spawning Programs

YottaDB

- We could spawn a program, then redirect it's STDERR and/or STDOUT
  - popen (man 3 popen) will let you get stdout
  - Or, use fork and exec* after setting pipes in parent program
- But this would only work on a new terminal; we can't attach to terminal 23

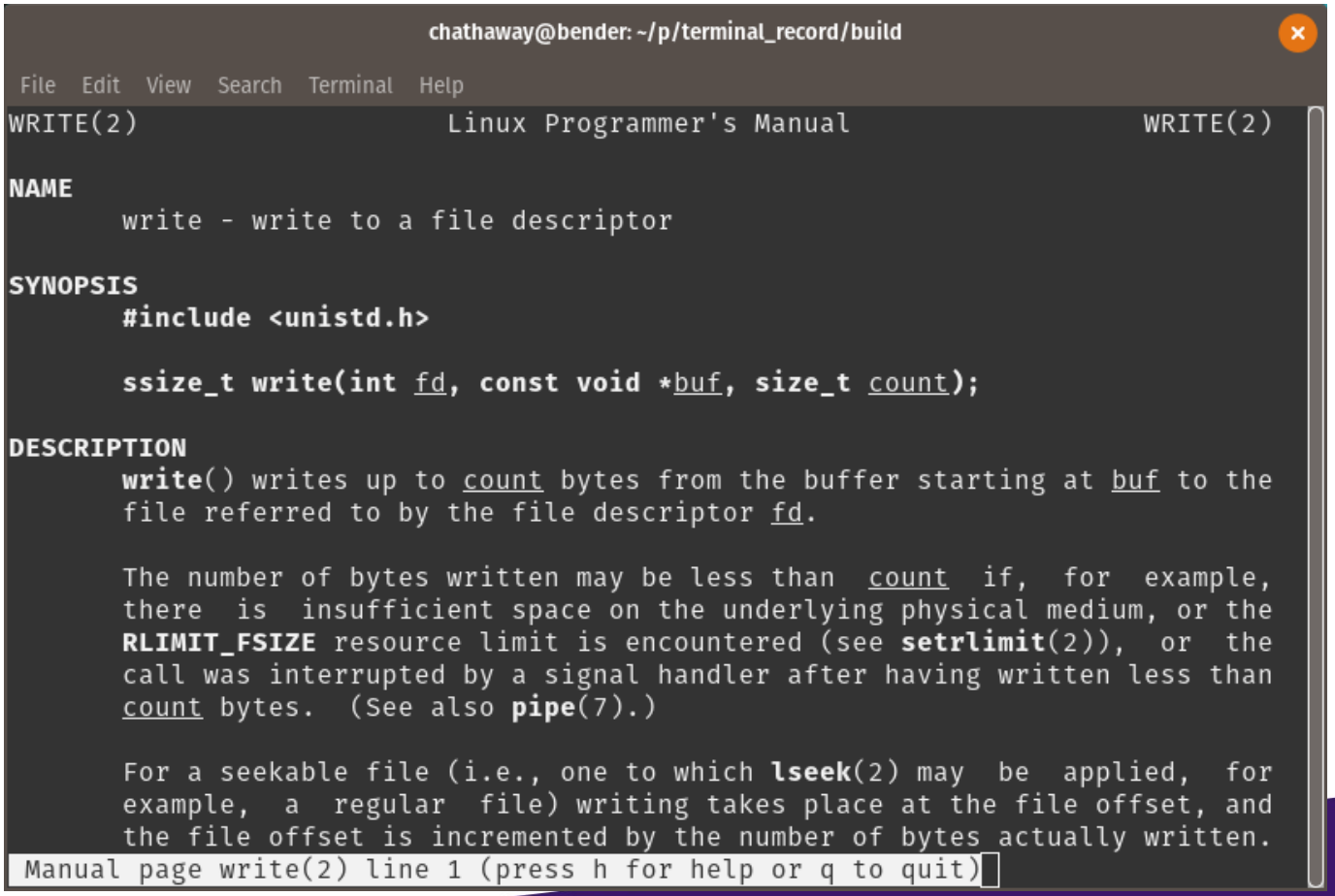
Writing to a file descriptor

YottaDB

- When a process wants to write, it calls something like:
  - printf, fprintf, dprintf
  - And the above call: write (man 2 write)

Writing to a file descriptor

YottaDB



man man

- 1 Executable programs or shell commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files (usually found in /dev)
- 5 File formats and conventions eg /etc/passwd
- 6 Games
- 7 Miscellaneous (including macro packages and conventions), e.g.
- man(7), groff(7)
- 8 System administration commands (usually only for root)
- 9 Kernel routines [Non standard]

man 2 syscalls

YottaDB

"The system call is the fundamental interface between an application and the Linux kernel."

- To do anything related which requires work from the kernel, a system call is required
- This mean all output to the terminal must go through write, and therefore, must go through a system call
- Can we watch for system calls?

man 1 strace

strace - trace system calls and signals

- Very cool program which will tell you exactly what system calls are being placed by a program
- Can attach to an existing process
- Output is often hard to parse, and there is a lot of it

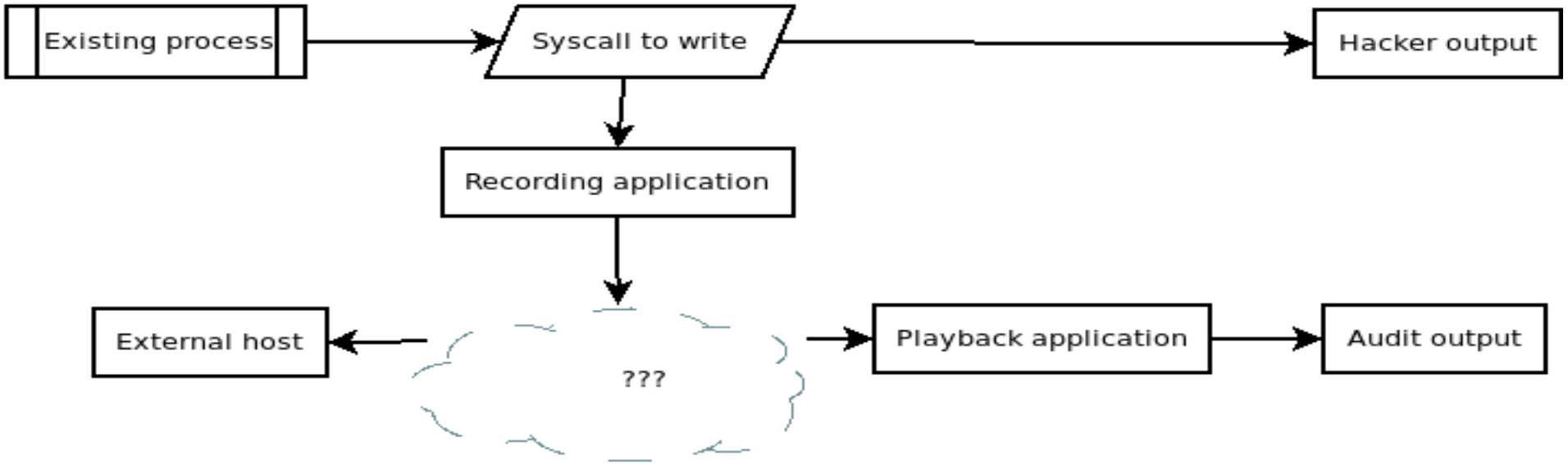
YottaDB

Solution design

Watching syscalls

Overall design

YottaDB



Parsing strace output

YottaDB

- We will pipe a command like:
  - strace -p 5679 -e write -x -s 8192
- Into the terminal recording (termrec) program
  - -p [pid] -e [catch write] -x -s [limit to 8192 chars per message]
  - -x means always output hex escaped characters

Quick note about terminals

YottaDB

- Escape characters are everywhere
  - Terminal colors? Escape sequences
  - Newline? Escape sequence
  - Backspace? Escape sequence
  - These are invisible characters we catch via strace
  - If we echo them exactly, we get the same output

Language to write our tool in

YottaDB

- We're going to be working at a pretty low-level; aside from parsing strace, we may want to record terminal size, encoding, etc.
- Most of this data is available via system calls
- Low-level language with high-level abstractions?
  - Rust

What is Rust?

YottaDB

- A system programming language with zero-cost abstractions
- Strongly typed with a trait system for abstracting behaviors
- Strict borrowing system to hide concurrency complexity and memory management

Where to persist data?

YottaDB

- YottaDB, of course!
- YottaDB is daemon-less, has a straight-forward Rust API, and provides features to safely and securely replicate data off of a host if it is compromised
  - Great at storing time-series data
  - High performance, minimal overhead

Hierarchical Database

YottaDB

- A database which stores data in a hierarchy
  - For example, the key “person” can have subscripts like “Hathaway” and “Charles”, to give us the key [“person”, “Hathaway”, “Charles] which contains a value
  - Convenient way to store data where “time” can be considered part of the key

How to persist data?

- We're going to store the output of the strace command to YottaDB using a schema like:
  - ["^termrec", "<session-id>",

    "<millisecond>",<unique number>]= <value>

Implementation

- Sample usage:
  - termrec record -p <pid>
  - termrec list
  - termrec play <session id>

YottaDB

Implement

Implementation – Parsing arguments

- Rust library Clap; use factory to generate usage

```
let matches = App::new("termrec")
 .version("1.0")
 .author("Charles Hathaway <charles@yottadb.com>")
 .about("Attaches to an interactive shell and records the terminal output")
 .subcommand(SubCommand::with_name("list")
            .about("Lists record session ID's"))
 .subcommand(SubCommand::with_name("record")
            .about("Records a session")
            .arg(Arg::with_name("pid").help("PID to record").required(true).index(1)))
 .subcommand(SubCommand::with_name("play")
            .about("Play a session")
            .arg(Arg::with_name("session_id").help("Session to replay").required(true)
                .index(1)))
 .get_matches();
```

Implementation – Parsing strace

YottaDB

- Input looks something like:
  - <pid> write(2, "<hex-escaped string>", <num>) = <num>
- Parsing can be tricky; we will use the nom Rust library to parse things

Implementation – Parsing strace

YottaDB

- First, reading the function name and integers

```
named!(function_name, do_parse!(val: take_while!( is_char ) >> (val)));
named!(read_usize<&[u8], Result<usize, ParseIntError>>,
    map!(nom::digit, |val: &[u8]| String::from_utf8_lossy(val).parse::<usize>()));
```

Implementation – Parsing strace

- Reading a hex-escaped string

```
named!(escaped_string <Vec<u8>>, map!(escaped_transform!
(take_until_either1!("\"\\"), '\\', alt!(
                tag!("\\") => { |_| &b"\\"[..] }
              | tag!("\"") => { |_| &b"\""[..] }
              | tag!("n") => { |_| &[10][..] }
              | tag!("x") => { |_| &b"\\"[..] }
          )), replace_hex_chars));
```

Implementation – Parsing strace

YottaDB

- Putting it all together

```
named!(syscall_record<&[u8], SyscallRecord>,
    ws!(do_parse!(
            opt!(tag!("[pid")) >> opt!(read_usize) >> opt!(tag!("]")) >>
            function: function_name >>
            tag!("(") >> fd: read_usize >> tag!(",") >> tag!("\"") >>
            val: escaped_string >> tag!("\"") >>
            (SyscallRecord{function: String::from_utf8_lossy(function).into_owned(),
                fd: fd.unwrap(), val: Vec::from(val)})
        )
    ));
```

Implementation – Storing a record

YottaDB

- Database is in memory, but context stores metadata

```
fn record(val: &[u8], session: &str, ctx: &Context) -> Result<(), Box<Error>> {
    let time = SystemTime::now().duration_since(UNIX_EPOCH)?;
    let mut k = make_ckey!(ctx, "^termrec", session, time.as_millis().to_string());
    let unique_id = k.increment(None)?;
    k.push(unique_id);
    k.set(&Vec::from(val))?;
    Ok(())
}
```

Implementation – Recording

YottaDB

```
fn handle_record(matches: &ArgMatches) -> Result<(), Box<Error>> {
    let ctx = Context::new();
    let mut session_key = make_ckey!(ctx, "^termrec");
    session_key.increment(None)?;
    let session = session_key.get()?;
    let session = String::from_utf8_lossy(&session);
    println!("Recording session {}", session);
    let stdin = io::stdin();
    for line in stdin.lock().lines() {
        let rec = line?; let rec = rec.as_bytes(); let rec = syscall_record(&rec);
        if rec.is_err() { continue; }
        let rec = rec.unwrap().1;
        record(&rec.val, &session, &ctx)?;
    }
    println!("Done!");
    Ok(())
}
```

Implementation – Playback

YottaDB

```
fn handle_playback(ctx: &Context, matches: &ArgMatches) -> Result<(), Box<Error>> {
    let session_id = matches.value_of("session_id").unwrap();
    println!("Playing back session {}", session_id);
    let mut session_key = make_ckey!(ctx, "^termrec", session_id, "0");
    let mut start_time = 0;
    let stdout = io::stdout();
    for v in session_key.iter_key_subs() {
        let mut k = v?;
        start_time = handle_sleep(&k[2], start_time)?;
        k.push(Vec::from(""));
        for v in k.iter_values() {
            let val = v?;
            Stdout.lock().write(&val)?;
        }
    }
    Ok(())
}
```

Implementation – List

YottaDB

```
fn handle_list(ctx: &Context, matches: &ArgMatches) -> Result<(), Box<Error>> {
    println!("Listing sessions");
    let mut sessions_key = make_ckey!(ctx, "^termrec", "0");
    let stdout = io::stdout();
    for v in sessions_key.iter_key_subs() {
        let k = v?;
        let session = String::from_utf8_lossy(&k[1]);
        println!("Session: {}", session);
    }
    Ok(())
}
```

Implementation – Pulling it all together

YottaDB

```
let ctx = Context::new();
if let Some(matches) = matches.subcommand_matches("record") {
    handle_record(&ctx, matches)?;
}
if let Some(matches) = matches.subcommand_matches("play") {
    handle_playback(&ctx, matches)?;
}
if let Some(matches) = matches.subcommand_matches("list") {
    handle_list(&ctx, matches)?;
}

Ok(())
```

YottaDB

Demo

YottaDB

Thank You!

yottadb.com