Contents

1 Extending Stackdb **1**

1.1 Code Structure 1

1.2 Target Lifecycle 2

1.3 Overlay Targets 2

1.4 Symbols 2

1.5 Current Drivers 2

1.6 Writing a Driver 2

1.7 Writing a Personality 2

1 Extending Stackdb

Hopefully, you'll have arrived at this section because you want to extend Stackdb's platform support. If you hoped to get advice on how to improve its design, unfortunately this won't be much help to you.

1.1 Code Structure

Here's a quick overview of the source tree:

- include/ – common Stackdb header files: data structures, architecture support, utility functions and structures
- lib/ – implementations of the header files in include/
- dwdebug/ – DWARF debuginfo and ELF support for reading binary files, and extracting and indexing debugging data, including symbols, types, addresses/locations, source file/line information, stack-unwinding data, etc; provides fast lookups and optimized file loading and indexing; supports C/C++ (and its data structures suffice to describe the core of languages like Python, PERL, PHP, and the like – but some constructs of higher-level languages might be harder to "fit" into the dwdebug abstractions, since they were designed with C language features in mind.
- target/ – the core of Stackdb: the target abstraction and its common code (targets, threads, addrspaces, regions, ranges, bsymbols, etc); drivers; personalities
- analysis/ – supports Stackdb analyses (at one point, we envisioned Stackdb-based programs and libraries as *analyses*; this is first-class support for the metadata that describes those programs)
- xml/schema/ – XML schema describing debuginfo and target data structures

- xml/service/ – WSDL and SOAP web services; exports three categories of Stackdb functions (dwdebug; target; and analysis) as web services
- xml/client/ – SOAP client support; Apache Axis 2.x-based Java client libs and example programs; Python sample clients.
- tools/ – several basic tools, written using Stackdb, that can be applied to any target (i.e., dumpdebuginfo, backtrace, dumptarget, dumpthreads, spf, cfi_check, rop_checkret)

1.2 Target Lifecycle

1.3 Overlay Targets

1.4 Symbols

Probably what you're curious about is, why are there three different data structures describing symbols? `struct symbol` and `struct lsymbol` in dwdebug/, and `struct bsymbol` in target/. First, we desired to provide first-class support for looking up symbol *expressions*, not just individual symbols. It's much easier for the user to look up an expression like "init_task.mm.mm_count", instead of the individual members and/or typed pointers that make up the chain – and given this chain (an lsymbol), the user can ask the to lookup a bsymbol (a "bound" symbol – bound to a target region, because the debugfile in which the symbol was found is associated with a specific memory region) – and then to load it. In other words, the target library can load symbol expressions, not just individual symbols. This is quite convenient.

Unfortunately, it complicates the API. The target API (bsymbols) essentially wraps the dwdebug API (symbols and lsymbols) functions.

Finally, symbols are reference counted. This is necessary because of optimizations in the dwdebug/ library that strive to remove duplicate debuginfo data (there can be a lot of it!). Users are never exposed to symbols that might be deleted; the reference counting is primarily used to guard symbols while the loading algorithms are running and conducting space-saving optimizations.

This means when users lookup a symbol, they must release it via bsymbol_release(), lsymbol_release(), or symbol_release(). Unfortunate, but that's how it has to be to enable the optimizations.

1.5 Current Drivers

1.6 Writing a Driver

1.7 Writing a Personality