LLDB Reproducers
"The debugger doesn't work"

– Somebody on the internet
LLDB Bugs

$ lldb ./a.out
(target) target create "a.out"
(target) b main.cpp:12
...
(target) run
...
(target) expr @import Foo
(target) expr Bar
*cannot materialize variable*
LLDB Bug Reports

Bob

"Hey this doesn't work..."

Alice
LLDB Bug Reports

"Can you attach the expr log?"

Bob

Alice
LLDB Bug Reports

Expression log

Bob

Alice
LLDB Bug Reports

"Can you attach the type log?"

Bob

Alice
LLDB Bug Reports

Bob

Type log

Alice
LLDB Bug Reports

"How do I reproduce?"

Bob ➔ Alice
LLDB Bug Reports

Steps to reproduce

Bob

Alice
LLDB Bug Reports

"It doesn't reproduce..."

Bob

Alice
Reproducers

- Automate the process
- Everything needed to reproduce
- Inspired by clang
Reproducers

```
$ lldb ./a.out --capture
(lldb) target create "a.out"
(lldb) b main.cpp:12
...
(lldb) run
...
(lldb) expr @import Foo
(lldb) expr Bar
cannot materialize variable
(lldb) reproducer generate
```

```
$ lldb --replay reproducer
(lldb) target create "a.out"
(lldb) b main.cpp:12
...
(lldb) run
...
(lldb) expr @import Foo
(lldb) expr Bar
cannot materialize variable
```
LLDB Reproducers
Reconstruct the debugger's state

- How we get there is more important than the final result
- Capture data
- Debug during replay
Information

User interaction

• Commands typed in the command line interpreter
• Use of the public API

System interaction

• Data from the (file) system
• Data from the process being debugged
Minimize impact

• Don't hide or introduce bugs
• Reuse existing infrastructure
• Transparency and abstraction
Components
User Interaction

Command Line Interpreter

Public API (Scripting Bridge)

Files

GDB Remote Protocol
$ lldb ./a.out
(lldb) target create "a.out"
(lldb) b main.cpp:12
...
(lldb) run
...
(lldb) expr @import Foo
(lldb) expr Bar
User Interaction

Command Line Interpreter

Public API (Scripting Bridge)

Files

GDB Remote Protocol
Stable C++ API

• Accessible through Python wrappers
• Used by IDEs such as Xcode, Eclipse, Visual Studio Code
• How the command line driver is implemented
Python Example

```python
import lldb

debugger = lldb.SBDebugger.Create()
target = debugger.CreateTarget("/path/to/a.out")
target.BreakpointCreateByName("foo")
process = target.LaunchSimple(...)```
Capture and replay API calls

- Capture the call and its argument values
- Capture the return value
API Boundary

Python

SBDebugger

SBTarget

CreateTarget()

GetByteOrder()
Detecting API Boundaries

- RAII object consisting of two booleans
- Static boolean toggles when crossing the API boundary
- Non-static boolean tracks if boundary needs to be reset
Capturing Calls

- Toggles the API boundary
- Captures the call and its arguments
- More than 2000 instances

```cpp
lldb::SBThread SBValue::GetThread() {
    LLDB_RECORD_METHOD_NO_ARGS(lldb::SBThread, SBValue, GetThread);
    ...
    return LLDB_RECORD_RESULT(sb_thread);
}
```
Capturing Calls

- Maps functions to unique identifier
- Type safe
- Synthesizes deserialization logic

```c
LLDB_REGISTER_METHOD(void, SBDebugger, SetAsync, (bool));
LLDB_REGISTER_METHOD(bool, SBDebugger, GetAsync, ());
LLDB_REGISTER_METHOD(void, SBDebugger, SkipAppInitFiles, (bool));
LLDB_REGISTER_METHOD(void, SBDebugger, SkipAppInitFiles,(bool));
```
lldb-instr

- Utility on top of libTooling
- Traverses the clang AST
- Inserts the record and register macros
Capturing Arguments

• Stream values to file
• Look at underlying value for pointers and references
Capturing Objects

- Index based on object address
- Table keeps mapping between object and index

```cpp
SBDebugger debugger = SBDebugger::Create();
SBTarget target = debugger.createTarget();
SBLaunchInfo info("--arguments");
target.Launch(info);
```

<table>
<thead>
<tr>
<th>Object</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>debugger</td>
<td>1</td>
</tr>
<tr>
<td>target</td>
<td>2</td>
</tr>
<tr>
<td>info</td>
<td>3</td>
</tr>
</tbody>
</table>
while (deserializer.HasData(1)) {
    unsigned id = deserializer.Deserialize<unsigned>();
    GetReplayer(id)->operator()(deserializer);
}
Components
System Interaction

Command Line Interpreter
Public API (Scripting Bridge)

Files
GDB Remote Protocol
$ lldb ./a.out
(lldb) target create "a.out" (binary)
(lldb) b main.cpp:12 (debug information)
...
(lldb) run (shared libraries)
...
(lldb) expr @import Foo (headers)
(lldb) expr Bar
Virtual File System

- Use files from the reproducer
- YAML mapping between virtual and real paths
- Lifted from clang to LLVM
- Devirtualize to support FILE* and file descriptors
Filesystem Class

- Wrapper around the VFS
- All file system access must go through this class
- `FileCollector` used for recording files used by LLDB & clang
Components
System Interaction

Command Line Interpreter
Public API (Scripting Bridge)

Files
GDB Remote Protocol
GDB Remote Protocol

- Simple command and response protocol
- Read and write memory, registers and to start/stop the process
- Designed for remote debugging but also used locally
Capture

LLDB

GDB Remote Protocol

debugserver

Implementation Defined

a.out

Reply #1

Reply #2

Reply #3

Reply #4

Reply #5

...
Replay

- Responds with recorded reply (in order)
- Fully transparent to the debugger
- Replay remote debug sessions
Limitations and future work
API Arguments

- Function pointers
- Void & data pointers

```c
void Foo(char* data, size_t length);
```
Memory Management

- No lifetime tracking for now
- Pointer addresses can be reused
- Objects created during replay are never deallocated
Swift

- Virtual File System
- FileCollector callback
Reproducer Size

- Large files
- Many files
- Do we need all of them?
Crashes

- No guarantees in the signal handler
- Do something smart like clang
Privacy

- Reproducers contain a lot of potentially sensitive information
- Need to be clear and upfront about this to the user
Please try it out!

bugs.llvm.org
Questions?