Gotcha Documentation

Release 0.1.0

David Poliakoff, Matt Legendre

May 31, 2018

Contents

Version 0.1.0

CHAPTER 1

About

Gotcha is an API that provides function wrapping, which is the concept of injecting a new wrapper function between another function and its callsites. Many tools rely function wrapping as a fundamental building block. A performance analysis tool may, for example, put function wrappers around “read()” and “write()” that trigger a stopwatch around the original calls, which allows the tools to measure time an application spends in IO.

Tools traditionally implemented function wrapping with the LD_PRELOAD environment variable on glibc-based systems. This environment variable allowed the tool to inject a tool library into the target application. Any functions that the tool library exports, such as a “read()” or “write()” function, will intercept calls to the matching function names from the original application. While powerful, the LD_PRELOAD approach had several limitations:

- Tool libraries can have challenges matching ABI-compatibility with the application.
- Multiple tools cannot wrap the same function.
- The set of wrapped functions are determined at tool build-time and cannot be changed in response to application behavior.

Gotcha addresses these limitations by providing an API for function wrapping. Tool libraries make wrapping requests to Gotcha that say, for example, “wrap all calls to the read() function with my tool_read() function, and give me a function pointer to the original read().” Gotcha’s API allows tool wrapping decisions to be made at runtime, and it handles cases of multiple tools wrapping the same function. It does not, however, provide any new mechanisms for injecting the tool library into an application—gotcha-based tools should be added to the application at link-time or injected with LD_PRELOAD.

Gotcha works by rewriting the Global Offset Table (GOT) that links inter-library callsites and variable references to their targets. Because of this Gotcha cannot wrap intra-library calls (such as a call to a static function in C) or calls in statically-linked binaries. Binary rewriting technology such as DyninstAPI (https://github.com/dyninst/dyninst) is more appropriate for these use cases.

CHAPTER 2

Building

Gotcha is hosted on Github at https://github.com/llnl/gotcha. It uses CMake to build, and details can be found in the README file in the Gotcha root directory.

Gotcha builds to a shared library, libgotcha.so, and accompanying C header files. Tools may choose to use CMake to build a static .a archive for Gotcha, but there should not be multiple instances of the gotcha library in process (which could happen if multiple tools use the archive option).

CHAPTER 3

Developer's Guide

This section describes how to use the Gotcha API. There are several general concepts that are worth understanding:

- The gotcha API deals in multiple types of function pointers. **Wrappers** are tool-side functions that are injected into the application. Wrapper function pointers are typically passed from the tool into gotcha. **Wrappees** are application-side functions that are wrapped by wrappers. Wrappee function pointers are typically passed from gotcha back to the tool.
- Gotcha is configured through **tool names**. Tool names are C strings that set scope for runtime configuration options. If multiple tools share a gotcha instance, then the tool name differentiates each tool's configuration. One tool may, for example, set its stacking priority by calling

  gotcha_set_priority("tool_name_A", 50), while another tool may set different priorities with a gotcha_set_priority("tool_name_B", 25). Configurations can be hierarchical—the forward slash character '/' separates tool name hierarchies. For example, a tool named "tool_A/subcomponent" will inherit and potentially overwrite configuration options from "tool_A".
- Multiple tools adding wrappers to the same function will produce a **wrapper stack**, which call each tool's wrapper in a consistent order (based on the tool priority). Each tool wrapper must call the next wrapper, which can be obtained through the gotcha_get_wrappee() function.
- Unlike LD_PRELOAD, gotcha wrapping actions only start after a call to gotcha_wrap. If an application calls a wrappee before the gotcha has turned on wrappings, then no wrapper will intercept the call. It is therefore recommended to enable gotcha wrappings early in the application's start-up, perhaps at the top of main or in a library constructor.

3.1 Example

This example shows how to use gotcha to wrap the open and fopen libc calls. This example is self-contained, though in typical gotcha workflows the gotcha calls would be in a separate library from the application.

The example logs the parameters and return result of every open and fopen call to stderr.

```c
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5
6  #include "gotcha/gotcha.h"
7
8  typedef int (*open_fptr)(const char *pathname, int flags, mode_t mode);
9  typedef FILE* (*fopen_fptr)(const char *pathname, const char *mode);
10
11 static gotcha_wrappee_handle_t open_handle;
12 static gotcha_wrappee_handle_t fopen_handle;
13
14 static int open_wrapper(const char *pathname, int flags, mode_t mode) {
15   open_fptr open_wrappee = (open_fptr) gotcha_get_wrappee(open_handle);
16   int result = open_wrappee(pathname, flags, mode);
17   fprintf(stderr, "open(%s, %d, %u) = %d\n",
18           pathname, flags, (unsigned int) mode, result);
19   return result;
20 }
21
22 static FILE *fopen_wrapper(const char *path, const char *mode) {
23   fopen_fptr fopen_wrappee = (fopen_fptr) gotcha_get_wrappee(fopen_handle);
24   FILE *result = fopen_wrappee(path, mode);
25   fprintf(stderr, "fopen(%s, %s) = %p\n",
26           path, mode, result);
27   return result;
28 }
29
30 static gotcha_binding_t bindings[] = {
31   { "open", open_wrapper, &open_handle },
32   { "fopen", fopen_wrapper, &fopen_handle }
33 };
34
35 int main(int argc, char *argv[]) {
36   gotcha_wrap(bindings, 2, "demotool");
37
38   open("/dev/null", O_RDONLY);
39   open("/dev/null", O_WRONLY | O_CREAT | O_EXCL);
40   fopen("/dev/random", "r");
41   fopen("/does/not/exist", "w");
42
43   return 0;
44 }
```

The fundamental data structure in the Gotcha API is the gotcha_binding_t table, which is shown in lines 29-32. This table is stating that calls to open should be rerouted to instead call open_wrapper, and similarly for fopen and fopen_wrapper. The original open and fopen functions will still be accessible via the open_handle and fopen_handle handles.

The binding table is passed to Gotcha on line 36, which specifies there are two entries in the table and that these are part of the “demotool” tool. The open_handle and fopen_handle variables are updated by this call to gotcha_wrap and can now be used to look up function pointers to the original open and fopen calls.

The subsequent callsites to open and fopen on lines 37-40 are redirected to respectively call open_wrapper and fopen_wrapper on lines 14-20 and 22-27. Each of these functions looks up the original open and fopen functions using the gotcha_get_wrappee API call and the open_handle and fopen_handle on lines 15 and 23.

The wrappers call then call the underlying functions open and fopen functions on lines 16 and 24. The print the parameters and results of these calls on lines 17 and 25 and return.

Note that this example skips proper error handling for brevity. The call to gotcha_wrap could have failed to find instances of fopen and open in the process, which would have led to an error return. The calls to fprintf on lines 17 and 25 are stomping on the value of errno, which could be set in the open and fopen calls on lines 16 and 24.

3.2 API Reference

This section describes the API calls and semantics in Gotcha. It includes information about the types and function calls.

3.2.1 Types

gotcha_wrappee_handle_t

```
typedef void* gotcha_wrappee_handle_t;
```

The gotcha_wrappee_handle_t is an opaque type that is a handle to a function being wrapped by Gotcha—a wrappee. It is typically filled-in by a call to gotcha_wrap, and then translated into a function pointer by a call to gotcha_get_wrappee.

This type is accessible by including gotcha.h.

gotcha_binding_t

```
typedef struct {
  const char *name;
  void *wrapper_pointer;
  gotcha_wrappee_handle_t *function_handle;
} gotcha_binding_t;
```

This type describes a function wrapping action that gotcha should perform. A table of multiple gotcha_binding_t objects is typically used to describe all the wrapping actions that a tool would like gotcha to perform.

The name field is the name of a function to wrap. The wrapper_pointer is a function pointer (but cast to a void*) to the wrapper function. The function_handle is a handle that can be used to get a function pointer to the wrappee.

This type is accessible by including gotcha.h.

gotcha_error_t

```
typedef enum {
  GOTCHA_SUCCESS = 0,
  GOTCHA_FUNCTION_NOT_FOUND,
  GOTCHA_INTERNAL,
  GOTCHA_INVALID_TOOL
} gotcha_error_t;
```

This enum is used to indicate successful gotcha calls or return error values.

This type is accessible by including gotcha.h.

3.2.2 Functions

gotcha_wrap

```
enum gotcha_error_t gotcha_wrap(gotcha_binding_t *bindings,
                                int num_actions,
                                const char *tool_name);
```

The gotcha_wrap function enables a set of function wrappings for a tool. The bindings parameter is a table of gotcha_binding_ts of size num_actions. For each entry in the gotcha_binding_t table, Gotcha will wrap the function whose symbol name matches the table entry's name field, with the function pointed to by wrapper_pointer. On a successful wrapping, Gotcha will set *function_handle to an opaque value that can be used with gotcha_get_wrappee to get a function pointer to the wrappee.

The tool_name parameter is a string of the tool's name. See the introduction section of this for information about tool names.

If a specific bindings' entry contains the name of a function that does not exist in the process, then Gotcha will set that entry's *function_handle to NULL. It will continue applying wrappers around other functions from the same table.

gotcha_wrap will return one of the following values:

- GOTCHA_SUCCESS if all wrappings were successfully applied.
- GOTCHA_FUNCTION_NOT_FOUND if at least one function referenced by bindings was not found.
- GOTCHA_INVALID_TOOL if an invalid tool name was specified
- GOTCHA_INTERNAL if an internal error occurred in Gotcha.

This function is accessible by including gotcha.h.

gotcha_get_wrappee

```
void *gotcha_get_wrappee(gotcha_wrappee_handle_t handle);
```

This function returns a function pointer to a wrappee function, given a specific wrappee handle. The handle is typically returned as an output parameter from a call to gotcha_wrap.

If multiple tools are using Gotcha to wrap the same function, then it's possible for gotcha_get_wrappee to return a function pointer to the next tool's wrapper. In this case it is the responsibility of the next tool to eventually call the actual wrappee.

This function is accessible by including gotcha.h.

gotcha_set_priority

```
enum gotcha_error_t gotcha_set_priority(const char *tool_name,
                                        int priority);
```

gotcha_set_priority sets the priority of a tool to an integer value, which is used to determine wrapper order when multiple tools wrap the same function. Tools with smaller priority values are wrapped innermost, compared to tools with larger priority values.

tool_name can refer to a previously unknown tool. Gotcha will create a new tool when it sees a new tool name.

gotcha_set_priority will return one of the following values:

- GOTCHA_SUCCESS if the priority was successfully set.
- GOTCHA_INVALID_TOOL if an invalid tool name was specified.
- GOTCHA_INTERNAL if an internal error occurred in Gotcha.

This function is accessible by including gotcha.h.

gotcha_get_priority

```
enum gotcha_error_t gotcha_get_priority(const char *tool_name,
                                        int *priority);
```

gotcha_get_priority returns the priority of the tool name tool_name in the priority field.

gotcha_get_priority will return one of the following values:

- GOTCHA_SUCCESS if the priority was successfully set.
- GOTCHA_INVALID_TOOL if an invalid tool name was specified.
- GOTCHA_INTERNAL if an internal error occurred in Gotcha.

This function is accessible by including gotcha.h.