Coverage Based Fault Injection

Siddharth Shah (sas2387), Akshay Nagpal (an2756), Kunal Baweja (kb2896)

Abstract

Code coverage is one of the most important parameters for achieving robust software testing and design, however achieving the same is almost impossible given the limitations in testing scenarios. One such scenario is simulating environment faults for the application under test. In this report we present a prototype implementation of *Coverage Based Fault Injection* (CBFI)[7], to test programs for errors caused due to faulty or malicious underlying hardware, operating system and other related components. In this approach CBFI[7] is implemented as a coverage based fuzzer with dynamically linked library that wraps around existing GNU libc calls and permutes fault injections to fail a set of libc calls and test the program robustness as well as susceptibility to vulnerabilities due to unhandled exceptions or crashes at runtime. Our work is inspired and extends the previous work on fault injections, specifically that presented in Vulnerability Testing of Software System Using Fault Injection[1] and Libfaultinj[2].

Key Words

Fault Injection, CBFI (Coverage Based Fault Injection), LD_PRELOAD ,GNU libc, GNU Coreutils, pygdbmi, GDB

1. Introduction

Taking inspiration from the paper *Vulnerability Testing of Software System Using Fault Injection*[1] and Libfaultinj[2], we decided to create a coverage based fault injection tool. Our tool, CBFI[7], allows users to test for vulnerabilities arising due to errors in execution environment, such as I/O errors, network failures etc. based on code coverage in program. Such errors are difficult to simulate and lack in most testing tools because system components are highly reliable in a normal execution environment.

The CBFI[7] prototype is a coverage-based fault injecting fuzzer and shared object library that can be preloaded to wrap around the GNU libc calls to control the execution of various libc calls, used by most programs for interaction with underlying system and execution environment. This version of CBFI[7] contains wrappers for a selected set of GNU libc calls, tested to work on x86_64 architecture for the scope of the current project.

We studied Libfaultinj[2] that uses the LD_PRELOAD flag along with environment variables to fail certain libc calls used by the program under test. This approach blindly fails all of the occurrences of the specified libc call within a program. For example, if the user wants to test for failure of *open()* with EBADF error code, Libfaultinj[2] will return errors from all *open()* calls made within the program. In this scenario, if a fault injection causes program crash then user can't check for coverage or for other fault points beyond the first point where error occured. CBFI[7] extends this approach by introducing line coverage as a heuristic to permute fault injection points within the program, helping with rigorous testing.

In section 2 we describe in detail the architecture, design and our approach towards Coverage Based Fault Injection testing. Section 3 contains the implementation details of CBFI[7] prototype. In sections 4 and 5 we describe the experimental setup for testing our library on sample test programs and GNU coreutils suite and discuss the results in details. Section 6 describes the challenges faced and some known issues with the current implementation of CBFI[7] and possible solutions. Additionally we provide a sample run of the program in Appendix and the source code can be accessed on CBFI github repository [7].

2. Architecture

The CBFI fuzzer[7] links the shared object wrapper library at run time with the program under test and fuzzes the program in terms of libc calls to fail during the program execution to simulate fault injection scenarios such as I/O error, network errors etc. This is achieved in several disjoint steps combined together:

Running Example

In order to better explain the architecture of our shared library and fuzzer we use the example of *open()* libc call in this section, but we would like to emphasize that it is not limited to just one libc call, rather the same concept applies to all of the libc calls that we have overridden using our dynamic linked library, described in section 2.1.

For reference, FUNCTION_COUNTER refers to a counter maintained within the wrapper library for keeping a count of the number of time a given libc FUNCTION is invoked by the program under test. In addition, FUNCTION_FAIL represents the special flag that needs to be passed as a comma separated list through environment variables to indicate the specific occurrences of FUNCTION within the program that our library must fail for simulating environment errors.

A combination of FUNCTION_FAIL flags for different functions is permuted based on increments in coverage by our fuzzer to test for fault injections. This is described in detail in section 2.2 and section 3.

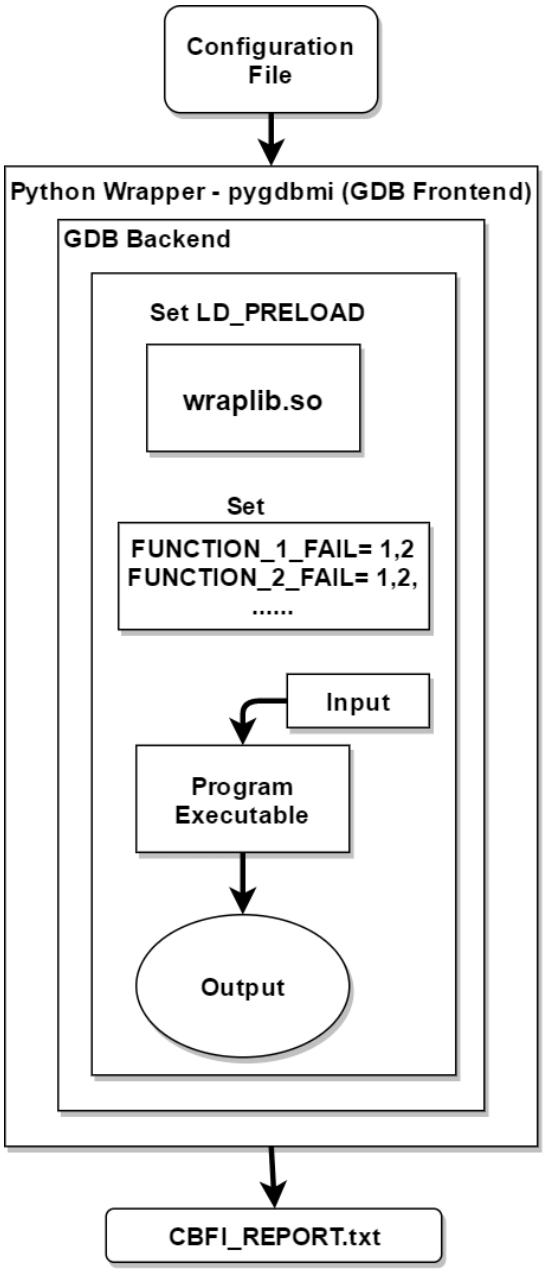

Figure 1: CBFI Tool Architecture

2.1 Share Object Linked Library

The key component of CBFI[7] is the dynamically linked library that is loaded at runtime to override the default behaviour of libc calls used by programs under test. At run time we need to ensure that dynamic linker loads our shared object file prior to loading the symbols from GNU libc library so that all calls pass through our library where we can override the default libc function or simply execute it normally. The LD_PRELOAD flag is used for this, which is checked by the dynamic linker *ld* on UNIX systems for shared libraries At present our shared object library supports programs for x86_64 systems.

Within each wrapper function for a libc call implemented by our shared library *wraplib.so* we maintain a FUNCTION_COUNTER for the number of times the libc function has been called by the test program. Let's say we have a libc function *open(),* used by programs to open file descriptor for reading an input stream. At start time OPEN_COUNTER is set to 0 and each time the program under test calls this function, OPEN_COUNTER is incremented by 1.

Now, we need to indicate to our shared library the specific occurrences of *open()* call within a program to be failed (simulate I/O error). This is achieved by passing an additional environment variable OPEN_FAIL as a list of comma separated numbers. Within the shared library implementation of *open()* OPEN_FAIL list is checked for existence of the current value of OPEN_COUNTER, each time *open()* is called. If not found, the *open()* libc call parameters are passed to the original libc call for normal execution, otherwise the wrapper fails the libc call by setting a randomly chosen *errno* such as EBADF, EACCES from the list of valid errno for the particular function and returns an error code -1 (or NULL or EOF as per the libc function specifications).

2.2 Fuzzing Program

The above described shared library sets the stage for fault injection in programs by intercepting the libc function calls. But our aim extends beyond this to test a program for as many fault injection vulnerabilities as possible which can be achieved by implementing a coverage based fault injection program. For this we implemented a simple fuzzer that automates the loading of our shared library, fault injection by setting the appropriate environment variables and then running the program. Further our fuzzer observes the coverage achieved after each run and based on that generates the next permutation of FUNCTION_FAIL variables to be passed

to the program under test for use in shared library, as described in the previous section. Simultaneously our fuzzer also generates a human readable report indicating the crash points in the program that occur due to fault injection. We describe more on implementation of fuzzer in section 3 and appendix.

3. Implementation

The CBFI[7] tool comprises of two basic components, a shared library, *wraplib.so* to intercept libc function calls made by a program and a python based fuzzer, described in the section 2 on architecture. Here we provide the comprehensive implementation details including all of the sub-components that go into the implementation of CBFI Fuzzer.

3. 1 Program Instrumentation

To perform CBFI[7] testing using our tool the first step that the user needs to do is instrument the program with code coverage instruction along with debugging flags which are required for running the program along with coverage information, described further in section 3.3.

User/programmer should include the *"cbfi.h"* header file in their C or C++ code. This file contains a simple function to invoke __gcov_flush() while executing the program to record coverage of program during test. Additionally the user must compile their program using the following flags with GCC compiler:

a) **--coverage:** Compile and link the test program with instrumentation for coverage analysis
b) **-O0:** Turn off all compile time optimization to ensure any libc calls used by the developer are not replaced by any other call
c) **-rdynamic:** Instruct the linker to add all the symbols to the dynamic symbol table of program
d) **-g:** add GDB debugging symbols for analysis at runtime in the fuzzer
d) **-fno-builtin:** Turn off GCC optimization for built-in C library functions. This helps ensure the default expected behaviour as per C99 standards.

3.2 Shared Library Implementation

The dynamically linked library described previously has been implemented as a shared object file, *wraplib.so* that can be loaded at run time to inject faults and simulate environment errors in the test program. The complete list of libc calls that we have implemented is mentioned in the Appendix A section, some of the examples being open(), close(), write(), chdir(), chown() etc.

The library has been implemented in C and tested on x86_64 system on Linux platform. To ensure that *wraplib.so* gets loaded prior to the default implementation of libc LD_PRELOAD flag is set in environment variables to point to system path of *wraplib.so* shared object file. The dynamic loader checks the LD_PRELOAD flag for user specified shared libraries before linking shared libraries to the executable. This ensures that function wrappers, implemented in *wrapperlib.so* with the same function signatures as corresponding libc function calls get loaded into program address space before the libc library and successfully intercept the calls to libc functions.

3. 3 Python Fuzzer

The fuzzer is implemented as a python program that runs the program under test in a GDB sandbox, acting as GDB frontend that interacts with GDB backend through *pygdbmi*[6] module. The fuzzer takes as input a json configuration file with following contents:

```
{
"SOURCE_FILE_PATH":"path/source/file.c",
  "EXECUTABLE":"path/to/executable",
  "ARGUMENTS":"command line args",
  "FAIL_CALLS":["OPEN", "READ", "CLOSE"],
  "LIBRARY_PATH":"path/to/wraplib.so",
  "PLAYGROUND": "path/to/test/directory"
}
```

Next, the CBFI[7] fuzzer starts the program execution within GDB sandbox and sets the LD_PRELOAD environment variable within GDB to the path of shared object file, *wraplib.so*. Now the fuzzer starts execution with a seed configuration of environment variables FUNCTION_FAIL for each of the libc calls specified in FAIL_CALLS in the input json. Once the program finishes execution, code coverage of the test program is observed and based on that a new permutation of FUNCTION_FAIL variables is generated and set within the GDB for next round of execution. Simultaneously fuzzer also maintains a queue of the FUNCTION_FAIL configurations to be passed next to the executable being run within GDB. This cycle of permutation of libc call fail

configurations and observing line coverage continues until the fuzzer is unable to increment the coverage and has exhausted all of the queued configurations generated in fuzzing. At this point, fuzzer terminates by dumping all of the observed coverages along with configurations in a text based readable report. It also reports the line numbers from the source file where crashes were observed as a result of fault injection.

3.4 GNU Libc Call Fail Permutations

Permuting the combination of libc functions to fail during program execution and the call numbers of each function to be failed is a crucial part of our tool. We describe it here with an example.

Let us consider that the json configuration file specifies FOPEN and FGETC in the list of calls the user or programmer wants to test for fault injection possibilities. The CBFI[7] fuzzing tool maintains a queue called FAILQ, initially empty. The fail call configurations generated during fuzzing are stored and maintained within this queue throughout the testing.

To generate a new configuration for the function calls to fail and the individual instance of those calls, our tool first attempts to simply increment the count of FOPEN_FAIL to FGETC_FAIL environment variables (or equivalents) from the previous execution and appends it to FAILQ in case of observed increase in coverage. Next time the configuration is retrieved from the queue and executed, it is checked for increase in coverage. If there is no increase in coverage on two successive permutations resulting from a given configuration then that configuration and the associated execution path is ignored further.

Fig 2 shows the changes in states of FAILQ at different steps through the fault injection fuzzing using CBFI[7] tool. Initially the tools starts execution with an empty queue. As a seeding condition, it starts by attempting to fail single individual instances of FOPEN and FGETC. If the program has not handled error cases of these calls then most likely it will result in a crash that our tool detects and records as a crash vulnerability. Based on this feedback from the program the CBFI[7] fuzzer checks for coverage information in the gcda file associated with the executable program and uses this information to generate the next permutation of libc functions to fail. Here FOPENX and FGETCX denote Xth instances of FOPEN and FGETC respectively that are specified for fault injection via FOPEN_FAIL and FGETC_FAIL environment variables (part of permutations that are generated).

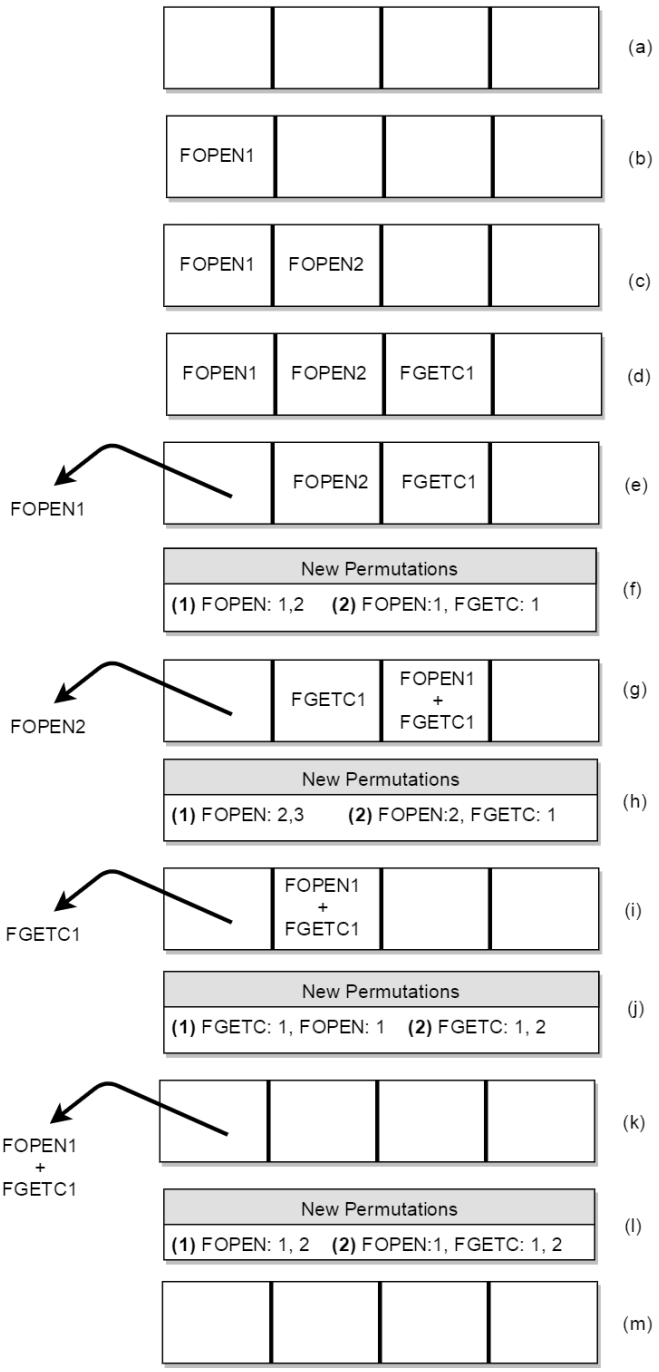

Figure 2: Fail Queue

In Fig 2, the program execution with FOPEN1 results in some new coverage lines and gets added to FAILQ. Next FOPEN_FAIL is incremented to 2 and executed. This also results in increased coverage and hence added to the FAILQ for permutation in further iterations. Next it

tries to fail FOPEN3 but does not observes any coverage increase so that is ignored. The process described so far for FOPEN is also repeated for FGETC and hence FGETC1 and FGETC2 are appended to the FAILQ.

Once the queue is stabilised after trying all individual instances of each function call the fuzzer starts removing items from the queue and further permutes them with combinations of 2 function calls from the list of functions to be failed, for example {FOPEN1, FGETC1}. Internally it is represented within the code as following environment variables:

```
FOPEN_FAIL=1    FGETC_FAIL=2
```

This method of generating new permutations based on previous code coverage attained is repeated until no more increase in coverage is observed and the FAILQ has been emptied. While adding the new generated permutations it is taken care that some of the permutations might be repetitive that is either already tried out and rejected due to no increase in coverage or maybe they are already present in the FAILQ as a result of generation from some previous configuration. Such cases are ignored to avoid repition and provide an upper bound on the number of test cases tried for testing the program.

4. Experiments

For building this tool we did rigorous experimentation and testing with our self written sample programs and *cat* and *ls* programs from coreutils test suite. The self written test cases were written deliberately without handling possible errors to test the CBFI[7] tool performance against presence of crashes due to unhandled errors.

4.1 Metrics

We use line coverage measure for coverage measurement and additionally detect any program crashes as part of the testing process for measuring program robustness. The less is the number of crashes, the more robust a given program is. Line coverage plays a two way role in helping us ensure complete coverage of program and also allows us to determine that CBFI[7] tool has looked at as much portion of code as possible to test for crashes due to fault injections.

4.2 Experimental Setup

Our sample program that we used for experiments is available on the link: check-pc.c which is a small random program combination of libc calls we use for testing coverage. The code is also shown in Appendix B below. The results for this program's coverage and fault injection are summarized in Table 1.

4.3 Comparison with Libfaultinj

In comparison with *Libfaultinj*[2] that we have based our work on, the CBFI[7] tool with fault injection based on line coverage is a significant improvement in terms of robust software testing. However pitting Libfaultinj[2] against CBFI[7] tool might be unfair because Libfaultinj[2] as it is a very basic framework to inject one fault at a time that does not provide any functionality to cherry pick which call occurrences.

4.4 Observations

Lines with errors: 32, 57, 59, 43, 39		
# Iterations	90	**NOTE:** Please refer to source code file link and full length report generated at below links: 1. Check-pc.c 2. CBFI-REPORT 3. Config File
Calls Failed	OPEN CLOSE GETC PUTC PUTS READ WRITE	
Observed Coverage	32 - 92.59 %	

Table 1: CBFI Analysis (Sample Program 1)

We also tested *ls* and *cat* commands from coreutils package shown. Coreutils proved to be a very robust code and we could not find crashes in the same, however our tool did measure some coverage and premutation that we summarize in the table below:

Lines with errors: None		
# Iterations	75	**NOTE:** Please refer to full length report and BCFI configuration file at below links: LS-REPORT LS-CONFIG
Calls Failed	CHDIR CHMOD GETC GETS PUTC PUTS OPEN CLOSE READ WRITE	

Observed Coverage	9.55 %	

Table 2: CBFI Analysis (ls coreutils)

Table 1 confirms that our approach is helpful in testing programs for unexpected environment faults such as I/O errors based on coverage. Table 2 further confirms that our tool generates fault injection permutations for a non-trivial number of given libc functions.

Lines with errors: 16, 17, 19, 20		
# Iterations	12	**NOTE:** Please refer to source code file link and full length report generated at below links: 4. dir-test.c 5. DIR_REPORT 6. Config File
Calls Failed	RMDIR MKDIR OPENDIR	
Observed Coverage	50 - 83 %	

Table 3: CBFI Analysis (Sample Program 2)

4.5 Test Suite and Source Code

The above mentioned experiments are only two out of the many sample tests that we carried out during development of this CBFI[7] tool. We encourage the reader to take a look at the complete source code repository: https://github.com/akshaynagpal/cbfi. Specifically the tests subfolder contains all of the unit tests for each libc function that our tool is capable of testing.

5. Results and Conclusion

In this project based study, we have attempted to create a Coverage Based Fault Injection Tool that rigorously tests a given program for unexpected errors in the execution environment such as I/O errors, network errors, permissions errors etc.

It is seen that many times, developers miss out safety checks while writing code that interacts with execution environment like reading / writing files, accessing memory, accessing networks etc because almost always these interactions are taken for granted due to the high level of reliability and availability in most cases. However these errors are still likely, especially in a compromised execution environment and developers must ensure that complete error checking at each possible step is done to prevent programs from crashing abruptly. While we can agree that modern machines are robust and fault-tolerant, but using such programs which may crash on environment faults for sensitive mission critical scenarios such as aeronautics, military, medical / surgery, nuclear science, governance etc, turning into disastrous results. Using test tools like CBFI[7] for rigorous fault tolerance analysis allows developers to prevent disastrous crashes due to aberrations that may arise in very rare scenarios and write code that can gracefully handle such environment anomalies.

Our tool in its current form runs perfectly on C/C++ programs on x86_64 architectures to test for errors caused due to fault injection in execution environment of the program. The results shown in section 4 confirm our hypothesis that coverage based fault injection is a promising approach towards software testing, especially towards unexpected errors from execution environment which might be tough to simulate in a normal scenario but nonetheless likely to occur.

6. Implementation Challenges and Learnings

Initially we started with writing a custom wrapper for printf, as it is a very commonly used function. After a lot of tries we noticed that our custom printf method was not being called when we used newline character at the end of string. The reason behind this was the internal optimization done by GCC at compile time that had replaced *printf()* with *puts()*. Hence we added -O0 and -fno-builtin flags for compiling C programs to be tested under our tool.

Every libc function wrapper that we wrote introduced us to different intricacies about the GNU libc interaction with underlying system and the programs that use the libc calls. Implementing some of these calls was fun as well as challenging that we enjoyed. To ensure robustness we wrote unit tests for each of the 44 libc call wrappers that we implemented. The source code for this CBFI[7] tool is available on Github CBFI repository.

Another challenging task that we faced was the use of some core libc functions like *open(), close(), write()* within the *__gcov_flush()* function that we use at each step in our fuzzing execution to record the program coverage information. By analyzing the *ltrace* of the program we figured out that some methods were called even without their use in our source program. To overcome this we needed a check to identify if the libc function wrappers were called by the __gcov_flush() call or the test program. To solve this we used the *backtrace()* function,

which partially solved our problem but also limited us to go without overwriting malloc() call, thus restricting us from implementing a crucial memory fault injection.

Our current implementation of CBFI[7] is closely linked with gcov and gdb for testing programs for coverage based fault injections. Since these programs themselves use some of the coreutils functions it was beyond our current scope to filter out all possible libc calls made by these programs from our implementation, thus we had to forgo some of those. Specifically, we faced issues in implementation of *malloc()* which is an essential memory allocation function used by almost all programs and functions, including many libc functions themselves

A major performance challenge in this tool is that unless a program executes normally gcda coverage files are not generated. Although this is a default behaviour as per coverage instrumentation, it is a limiting factor for our CBFI[7] tool. We couldn't find coverage in case of program crashes, which was the whole motive of this tool. Hence we used GDB and gcov_flush along with a python fuzzer to drive the program execution by single-stepping through the program execution. This allows us to execute gcov flush after each step in the GDB execution thus recording coverage information even in case of program crashes (till the crash point). However, as a tradeoff the speed is also reduced significantly which is a major challenge at this point of time, even for modestly sized programs.

Additionally some of the dependencies also fork out child processes using the *exec()* calls that we were not able to implement at this point. Another issue was using the C builtin function *backtrace()* to get the sequence of methods and their calling function names from the stack. It became a recursive loop as it uses *malloc()* internally causing our library to disfunction for permutations that caused malloc() to fail. A possible solution to the problem is using a non-GNU library called unwind_backtrace. Due to time constraints, we were not able to implement the solution in this iteration. We plan to integrate it in the next update of the tool.

References

[1] Vulnerability Testing of Software System Using Fault Injection https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/98-02.pdf

[2] Libfaultinj (Fault injection library) - https://github.com/androm3da/libfaultinj

[3] How to wrap a system call (libc function) in Linux http://samanbarghi.com/blog/2014/09/05/how-to-wrap-a-system-call-libc-function-in-linux

[4] GCC Printf: http://www.ciselant.de/projects/gcc_printf/gcc_printf.html

[5] GDB/MI: https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html

[6] pygdbmi: https://github.com/simark/pygdbmi

[7] CBFI Source: https://github.com/akshaynagpal/cbfi

Appendix

A. Wrapper methods in shared object wraplib.so

1. chdir
2. chmod
3. chown
4. close
5. connect
6. dup
7. dup2
8. getnetbyname
9. getnetbyaddr
10. getnetent
11. fclose
12. fgetc
13. fgets
14. fopen
15. fileno
16. fork
17. fprintf
18. fputc
19. fputs
20. fread
21. fseek
22. fwrite
23. getc
24. gets
25. gethostbyname
26. gethostbyaddr
27. getpid
28. getwd
29. getumask
30. kill
31. lseek
32. memcpy
33. memcmp
34. mkdir

35. open
36. opendir
37. printf
38. putc
39. puts
40. read
41. rmdir
42. scanf
43. system
44. write

B. check-pc.c

```
1. // test python wrapper
2.
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <unistd.h>
6. #include <assert.h>
7. #include <string.h>
8. #include <errno.h>
9. #include <sys/types.h>
10.#include <sys/stat.h>
11.#include <fcntl.h>
12.#include "cbfi.h"
13.
14.int main(int argc, char **argv) {
15.int fd;
16.int err, i;
17.size_t len;
18.char buffer[100];
19.FILE *fp;
20.char r = '\0';
21.
22.char              *fname              =
   "/home/kunal/Documents/secure-4995/cb
   fi/python-wrapper/playground/sample.t
   xt";
23.char *content = "Hello World";
24.len = strlen(content) * sizeof(char);
25.
26.// open for writing
27.fd  =  open(fname,  O_WRONLY|O_CREAT,
   0640);
28.
29.
30.// write
31.err = write(fd, content, len);
32.assert(err != -1);
33.
34.// close file
35.err = close(fd);
36.
37.// open for reading
38.fd = open(fname, O_RDONLY);
39.assert(fd > -1);
40.
41.// read file
42.err = read(fd, buffer, len);
43.assert(err == len);
44.
45.// close fd
46.close(fd);
47.
48.for (i = 0; i < strlen(content); ++i)
49.puts(content);
50.
51.// putc
52.fp = fopen("sample.txt", "a+");
53.
54.fseek(fp, SEEK_SET, 0);
55.for (i = 0; i < 5; ++i) {
56.r = getc(fp);
57.assert(r != EOF);
58.err = putc('a', fp);
59.assert(err != EOF);
60.}
61.
62.fclose(fp);
63.
64.return errno;
65.}
```

C. dir-test.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <assert.h>
5. #include <errno.h>
6. #include <sys/stat.h>
7. #include <sys/types.h>
8. #include <dirent.h>
9. #include "cbfi.h"
10.
11.int main(){
12.int x,y;
```

```
13.x=mkdir("/home/akshay/Desktop/playg
   round/testmakedir",777);
14.y =
   mkdir("/home/akshay/Desktop/playgro
   und/testmakedir2",777);
15.if(x==0 && y==0){
16.opendir("/home/akshay/Desktop/playg
   round/testmakedir");
17.opendir("/home/akshay/Desktop/playg
   round/testmakedir2");
18.if(errno==0){
19.rmdir("/home/akshay/Desktop/playgro
   und/testmakedir");
20.rmdir("/home/akshay/Desktop/playgro
   und/testmakedir");
21.}
22.else{
23.printf("%d\n",errno);
24.}
25.}
26.else{
27.printf("%d\n",errno);
28.}
29.return 0;
30.}
```