Application Monitoring Using libc Call Interception

Harish Thuwal[1] and Utkarsh Vashishtha[2]

[1] Department of Computer Science and Engineering, IIT, Delhi
`hthuwal@gmail.com`
[2] Department of Computer Science and Information Systems, BITS, Pilani
`f2015130@alumni.bits-pilani.ac.in`

Abstract. Generally Available Application Monitoring solutions comprise basic aggregated metrics. This paper collects such metrics in the form of raw data. We have worked out a very primitive piece written in `Golang`, which manifests results from `HTTP/HTTPS` calls from/to the system being monitored. This process can be easily extended and applied to various architectures that use the `libc` library to make low-level system calls to interact with sockets or in a more basic sense, files. The experimentation and the proof of concept were carried out on various languages viz `C, Java, Python`, and `Golang` itself using docker and some classic open-source compilers.

Keywords: Process level monitoring, `ld_preload`, gccgo, go lang monitoring, language agnostic monitoring.

1 Introduction

The central problem started with trying to develop a `Golang` monitoring system which worked agnostic of the underlying architecture and was a stable solution but ultimately ended with the **development of a language-agnostic, platform-agnostic solution to collect data at the process level**, allowing us to essentially collect data from sockets at the Application layer.

The study falls under Application Monitoring but instead of highly sophisticated techniques available today, it provides for a crude but novel technique to be taken and transformed into production-ready software. This mitigates challenges that one faces while trying to study their application which gets mostly fixed at compile-time, instead of being flexible enough to be mutated at run-time. To cite an example, the Go language with a compiler that uses the `gc tool-chain` resolves its `bind()` (the socket bind operation) method at compile-time, the memory location of this particular method gets ingrained in the binary(.o) file and the corresponding symbol in the generated symbol table gets resolved.

The work performed as part of this paper not only shows that we can perform monitoring of applications programmed in compiled languages but also opens up various possible use cases. The ability to intercept low-level calls allows us to non-invasively monitor, verify, restrict, and even modify a program's behavior.

When it comes to application monitoring, we show that any application can be monitored irrespective of the language used to build it, whereas traditional monitoring systems usually require different solutions for different programming languages. Any I/O operation, be it a network call or a file system access, occurs via low-level calls which when intercepted gives us access to the data being read and written to the network or the file system. This allows us to monitor the I/O performance of an application and also reinforce the security of the system by discarding suspicious or malicious data.

A simple illustration of restricting a program's behavior is limiting its ability to allocate/deallocate memory to a specific number of bytes by intercepting the `malloc/free` system calls. This will prevent any program from using more than a definitive amount of memory. Another example is preventing sensitive and special files from being accessed irrespective of the user access level. This can be achieved by tapping in the `fopen` system call and discarding it if the path of the file being accessed matches a sensitive location.

The following sections of the paper discuss the core idea of low-level call interception and its application to perform basic monitoring of a Golang server but it can easily be extended to languages like python and java.

2 Literature Review

Mandar Sahasrabudhe et. al in [6] demonstrates application monitoring methodologies. These involve the deployment of an Application Performance monitoring tool, generally referred to as the agent, alongside the application being monitored in the production environment. It collects the adequate metrics and thresholds and sends these over to a data store which creates and evaluates various important KPIs. Solutions based on the aforementioned methodology usually don't have a single universal tool but rather have different solutions for every programming language. The work in the paper as will be described ahead provides a way to overcome this constraint for many cases.

Yasushi Saito (2005) [2] in his work proposed an execution record/replay tool for debugging Linux programs. The tool can record the invocation of system calls and CPU instructions and later replay them in a deterministic manner. This approach is limited to instances where the link step of the application can be modified, as the symbols of interest need to be specified at link time. At its core the tool consists of a compiled shared object `libjockey.so` which is used to intercept every system call in `libc`. This is done by pre-loading the shared object before the target application. During recording the shared object intercepts and logs the values generated by the system calls which are then returned using the replay. To ensure that the target sees the same set of environment variables and command-line parameters during both record and replay the tool also creates checkpoints.

Zhan Shi et al. [4] (2010) show that the same principle of intercepting/hooking the system calls can be used to create a Lightweight File System Management Framework. The main idea is to hook and modify file requests sent by applications from management service running in user-space. One thing that they do differently from the record/replay tool is that they use LD_PRELOAD in two stages, once to hook the file system management framework to a shell program such as bash to get access to all processes being created and then to pre-load a dynamic library based on the type of process. Former interception targets all exec() functions which are the entry point for all user process creation while the later targets process-specific calls (specified by a Rule lib) depending on the use case.

Hyeong-chan Lee et al. [5] (2011) in their paper show that system call and standard library interception attacks are possible on the ARM-based Linux system. They pre-load a libhook.so object and demonstrate that they can modify the behavior of getuid() function. Though the goal here is to demonstrate interception attacks such as rootkit.

Ronny Brendel et al. [8] demonstrate a simple and robust way to record performance data on library function calls in *C/C++*. To demonstrate this they eventually need to create wrappers around the functions they aim to benchmark. They use --wrap option of the GNU to do this statically. But the static approach is limited to instances where the link step of the application can be modified, as the symbols of interest need to be specified at link time.

All the aforementioned techniques, from various sources, end up utilizing LD_PRELOAD to either inject a shared object into the main application or for wrapping functions dynamically.

3 Tools Used

The design and working code was developed for a small scale application using specific docker images running on a couple of machines and consequently excludes the performance aspects of this solution for very high-intensity processes. Some type of degradation in performance corresponding to the operations involved in call interception and generation of any custom metrics, if any, is bound to happen, the intensity of which may vary depending upon the use-case and how efficient the solution pans out taken forward from its current crude form. We primarily used the following tools for our study :

3.1 LD_PRELOAD

Before the execution of any program, the Linux dynamic loader locates and loads all the necessary shared libraries to perform symbol resolution and prepare the program for execution.

For this study, we made extensive use of the LD_PRELOAD [10] environment variable which provides one with an option to communicate with the loader to load custom logic in the form of shared objects before any other library. This is generally referred to as pre-loading a library that allows the pre-loaded library methods to be used before others of the same name in any other library being loaded later on. This provides us with the capability to intercept commonly used methods and replace them allowing for a subtle modification without recompilation of the user code. The following listings demonstrate this through an example of a C program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
  srand(time(NULL));
  int i = 10;
  while(i--) printf("%d\n",rand()%100);
  return 0;
}
```

Listing 1.1. random_num.c [9]

```c
int rand(){
    return 42; //the most random number in the universe
}
```

Listing 1.2. unrandom.c Overriding rand [9]

The main application `random_num.c` is programmed to print 10 random numbers on each execution. But when the compiled shared object of `unrandom.c` is pre-loaded before its execution using

```
LD_PRELOAD=unrandom.so ./random_nums
```

the injected implementation of the `rand` is used and ergo the output contains the number 42 printed 10 times. We leverage this technique to inject our monitoring code into the main application.

3.2 Gccgo

The creators/maintainers of Golang support two different types of compilers: `gc` and `gccgo` [3] which use two separate `toolchains` namely, the `gc toolchain` and the `gcc toolchain`. Go is a compiled language and needs to be assembled to a binary format before it is executed. This binary contains assembly code from either of those `toolchains`, depending upon the compiler you choose.

`gccgo` is a `Go` front-end connected to the `GCC` back-end. Compared to `gc`, `gccgo` compiled code contains optimizations implemented in `GCC` over the years and while it may be slower to compile, it supports more powerful optimizations and consequently in many cases may run faster [12].

Due to it's long and successful history GCC, and hence the `gccgo` compiled code, supports many more processors and architectures than `gc`. This may further be utilized to create a better architecture-agnostic framework supporting even `Solaris`.

The primary consequence to be drawn out from this study is how to build a multi-purpose monitoring/ data-collection system and put it into a binary that can be easily used. As aforementioned as part of this study, we would be using docker images to create a Go server and multiple users in multiple programming languages, basic Linux tools to figure out various `libc` calls being made during server-user interaction, an ingestion service, using a back-end DB for storing any data/metrics that would be created.

4 Experimentation and Implementation

After developing the call interception mechanism using `LD_PRELOAD` 3.1, we used the `strace`[11] and `ptrace`[1] system calls to figure out which `libc` calls to intercept to get the required information, essentially the socket level data, which amounts to reading from a simple file descriptor. We realized during this exercise that, for writing the interception class, we would need to override all the basic calls related to file interactions used while working with a socket. For instance, the call used to accept an incoming connection, which is an `accept()` `libc` call, is overloaded into many types such as the `accept4()` method. Different programming languages use different forms of the `accept()` method. We would also need to maintain access to the original `libc` handles so that we can perform the actual task after we are done gathering data. This was done by using `dl GoLang` package [7], a runtime dynamic library loader.

This multi-call interception class was then compiled into a shared object(.so) file that would be further injected using the aforementioned `LD\_PRELOAD` technique.

```
//export accept4
func accept4(s C.int, a unsafe.Pointer, st C.size_t, flags C.
    int) *C.int {
  t := (*syscall.RawSockaddr)(a)
  lib, _ := dl.Open("libc", 0)
  defer lib.Close()

  var old_accept func(s C.int, a *syscall.RawSockaddr, st C.
    size_t, fl C.int) *C.int
  lib.Sym("accept", &old_accept)
  oa := old_accept(s, t, st, flags)
  return oa
}
```

Listing 1.3. Intercepting accept4 libc call in Golang

We also realized, using the `nm` command, that the symbol table created when compiling a Go program using the go toolchain generates already resolved symbols for the basic system calls that, we initially intended to override for the interception. Next we stumbled upon the gccgo compiler that, uses the gccgo toolchain explained in Section 3.2 which renders these symbols, for instance the `bind()`, `listen()` etc., unresolved. We needed to now ensure that the target application uses gccgo for compilation.

Using `LD\_PRELOAD, GccGo`, and `libc` interception we were able to successfully override `libc` system calls comprehensive enough to encompass most of the mainstream programming languages. Our final goal was to intercept socket interaction calls that were related to any of the HTTP verbs. We wanted to capture the data being put into/read by a socket, specifically a file descriptor(FD), since at a particular time a process would be allocated a socket, and thereby an FD, creating somewhat of a session identifier that could be useful to study the pattern of data being sent from a particular IP.

```
//export read
func read(fd C.int, buf unsafe.Pointer, count C.size_t) *C.
    size_t {
  // Similar to accept, read the data using the libC read
  // into strdata

  // Perform monitoring/reporting tasks only for network
    traffic
  if socket != nil && strings.Contains(strdata, "HTTP") {
    // It is a response of earlier outgoing request
    if !strings.Contains(strdata, "Host:") {
      socketstr := getPortFromSocket(socket)
      if guid, ok := outgoingcalls[socketstr]; ok {
        // do whatever you want here
      }
    } else { // New Incoming request that contains
    singularity header
      if strings.Contains(strdata, "singularity") {
        // do whatever you want to do.
      }
    }
  }
  return oa
}
```

Listing 1.4. Intercepting read libc call in Golang

We published this (IP, FD, socket data) tuple into a back-end database (Figure 1) which could be queried according to the type of data/ IP to get kind of cumulative information for any further analysis.

5 Results and Future Work

Figure 1 depicts the overall architecture of the final setup that we implemented.

Element descriptions are as follows:

- **Source Application**: The target application written in `Golang` that will ultimately be monitored. The application can both receive and send HTTP traffic.

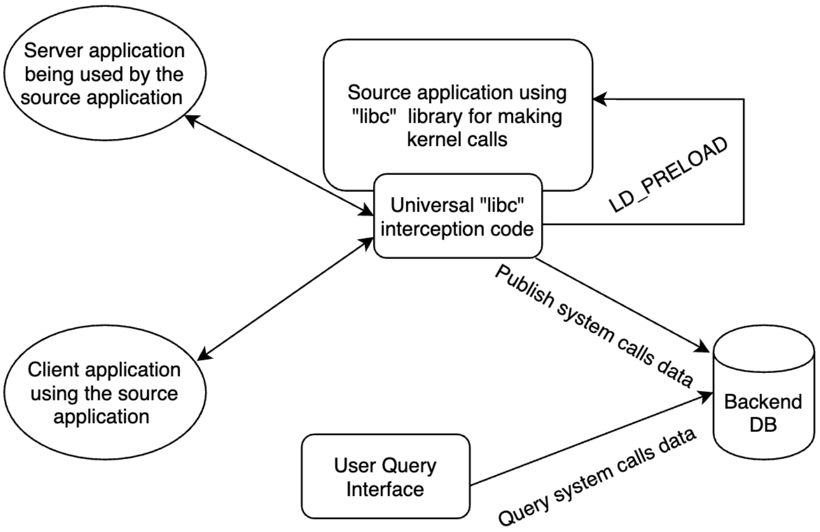
Server application being used by the source application
Source application using "libc" library for making kernel calls
Universal "libc" interception code
LD_PRELOAD
Client application using the source application
Publish system calls data
Backend DB
User Query Interface
Query system calls data

Fig. 1. Overview

- **Interception Code**: The shared object code that contains the `libc` call interception, filtering, and monitoring logic.
- **Server Application**: The application that the source application uses for its services.
- **Client Application**: The application that uses source application for its services.
- **Backend DB**: The database to populate the collected data from the source application
- **User Query Interface**: The framework to query Backend DB.

The source application pre-loaded with our interception code receives the `HTTP` request from the client application and sends an `HTTP` request to a web service. The incoming connections from client applications are monitored by intercepting the `read libc` call while the `write libc` call interception is used to monitor the outgoing requests to the web-server. The monitoring data and statistics collected by the interception code are periodically published to the back-end database which can be later queried to analyze the source application's behavior.

Note that for demonstration purposes, to simulate rather simple real-time systems, we used docker images to create the aforementioned source/client/server applications across multiple machines and captured metrics/ socket data on a separate one.

```
"_source" : {
  "data1" : {
    "Port": 37474,
    "Addr": [172,17,0,2],
    "Content": "HTTP/1.1 200 OK\nContent-Encoding: gzip\
   nAccept-Ranges: bytes\nCache-Control: max-age=604800\
   nEtag: \"3147526947+gzip\"\nExpires: Thu, 30 Jan 2020
   19:56:53 GMT\nServer: ECS (phd/FD69).....more data"
  },
  "data2" : {
    "Port": 37500,
    "Addr": [172,17,0,2],
    "Content": "HTTP/1.1 200 OK\nContent-Encoding: gzip\
   nAccept-Ranges: bytes\nCache-Control: max-age=604800\
   nEtag: \"3147526947+gzip\"\nExpires: Thu, 30 Jan 2020
   19:56:54 GMT\nServer: ECS (phd/FD69).....more data"
  },
  "data3" : {
    "Port": 37488,
    "Addr": [172,17,0,2],
    "Content": "HTTP/1.1 200 OK\nContent-Encoding: gzip\
   nAccept-Ranges: bytes\nCache-Control: max-age=604800\
   nEtag: \"3147526947+gzip\"\nExpires: Thu, 30 Jan 2020
   19:56:58 GMT\nServer: ECS (phd/FD69).....more data"
    }
}
```

Listing 1.5. Sample monitoring data

Listing 1.5 depicts sample data collected from the source application, this is the raw data obtained from a socket's file descriptor. The application as of now only captures the raw data but can be further developed to create a full-fledged socket monitoring application.

Our implementation mainly focuses on the HTTP traffic but since we can intercept calls associated with opening a file, closing a file, and thereby have access to every byte being written to or read from the associated file descriptor one can easily extend the same idea to perform File System Monitoring.

Apart from the `libc` call interception approach we also pondered over the idea of ELF file manipulation which if successful might be more performant. `Golang` applications generally use the net/http library to send and receive HTTP requests. Similar to `libc` call interception this approach also involves pre-loading a shared object. The shared object will contain a modified implementation of the net/http library's `httpServeAndListen` method which is essentially used by all `goLang` written code to serve http calls. The task will then be to modify the target ELF in such a way that the `httpServeAndListen` calls now go to the memory address of our modified implementation. This path can be explored as part of further research.

6 Conclusions

Through this work we demonstrated that by intercepting `libc` calls we can non-invasively monitor an application irrespective of the programming language used, given it makes use of those for basic open/read/write operations. The potential implications open up future work to easily create in-house monitoring systems that could span across multiple tiers from files to cross-service communications.

The exponential pace at every sector of the world is becoming digital has made application monitoring the need of the hour. But since modern applications span across different platforms and programming languages the task of monitoring all of them comprehensively becomes difficult. The ability to perform language-agnostic monitoring simplifies this task by requiring just one system that can monitor all.

7 Acknowledgements

This work was done as performed as part of a hackathon organized by Appdynamics, Cisco. We would like to thank and acknowledge Appdynamics for providing us with the opportunity to take part in the hackathon which enabled us to research, design and, implement the proposed work.

References

1. Padala, P. Playing with ptrace, Part I (2002).
2. Saito, Y. *Jockey: a user-space library for record-replay debugging* in *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (2005), 69–76.
3. Lance Taylor, I. *The Go frontend for GCC* in *Proceedings of the GCC Developers' Summit* (2010), 115–128.
4. Shi, Z., Feng, D., Zhao, H. & Zeng, L. *USP: A Lightweight File System Management Framework* in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage* (2010), 250–256.
5. Lee, H.-c., Kim, C. H. & Yi, J. H. *Experimenting with system and Libc call interception attacks on ARM-based Linux kernel* in *Proceedings of the 2011 ACM Symposium on Applied Computing* (2011), 631–632.
6. Sahasrabudhe, M., Panwar, M. & Chaudhari, S. *Application performance monitoring and prediction* in *2013 IEEE International Conference on Signal Processing, Computing and Control (ISPCC)* (2013), 1–6.
7. S.L., R. C. *Runtime dynamic library loader* https://github.com/rainycape/dl. 2015.
8. Brendel, R. *et al.* in *Programming and Performance Visualization Tools* 21–37 (Springer, 2017).
9. *Dynamic linker tricks, in 5.7.1* https://rafalcieslak.wordpress.com/2013/04/02/.

10. *ld.so, ld-linux.so - dynamic linker/loader* http://man7.org/linux/man-pages/man8/ld.so.8.html.
11. *strace - trace system calls and signals* http://man7.org/linux/man-pages/man1/strace.1.html.
12. *The Go Blog, Gccgo in GCC 4.7.1* https://blog.golang.org/gccgo-in-gcc-471.