KM : Effective Data Tracing In Distributed Systems

Your Name Here
Your Institution

Second Name
Second Institution

Abstract

Modern Internet services often involve large and complex distributed systems with data scattered around. As the number of services grows and data access scenarios get varied, important data might get leaked or stolen with no knowledge of the system administrators. In this paper we propose KM, a system which is used to trace data access and transfer in distributed systems. We present the model for tracing how data are accessed and transferred, showing that this model can succinctly capture users behaviors and thus can be used to replay scenarios where data are leaked or stolen. We also present a efficient implementation of KM and show that with proper sampling strategies our implementation can have negligible impact on existing services.

1 Introduction

Data access control and monitoring in complex distributed systems have been a real world concern [**?**] but have not yet gathered enough attention. Large internet companies which provide heterogeneous services often have their data stored in different data centers but the technologies they use to process these data, such as Mapreduce [1] and Spark [**?**], are built with no security in mind, making it hard to precisely grant data access permission to relevant users and, in case of data being stolen, trace how the data are transferred. In this paper we try to mitigate this situation by using a model to capture user program behaviors and design a system called *KM* to efficiently collect information of interest such that we can trace how a piece of data is accessed and transferred within or across machines and then probably stolen (i.e., copy into a USB stick or send to some remote machine).

The goal of KM is to track data access and transfer. The term *event* is used to describe any activity related to a piece of data. An *event* is modeled with a triad *(Subject, Action, Object)*. To succinctly model program behaviors, *event*s are classified into three categories: *file event*s, *network event*s and *process event*s. Examples of *file event*s include opening, reading and writing files; examples of *network event*s are binding a socket, connecting to remote machines and accepting incoming connections; examples of *process event*s are forking subprocesses and executing executables via the *execve()* system call. As will be shown below, these three classes of events is sufficient to model program behaviors.

The whole system is designed to be configurable. Strategies can be add or delete at anytime using a *KMctl* program or with the corresponding API. Therefore one can set up a central configuration service to manipulate strategies used by every single machine. These strategies include turning the tracing system on and off, controlling sampling rate and adding a file to be tracked. Note that our system is *data-oriented*, which mean that instead of collecting all events information, it will only collect information related to the data of interest. If during file transfering some intermediate file is not in our list (*e.g.*, process A read file *a* and write to file *b*. Process B read from file *b*. We have file *a* in our list, but the intermediate file *b* is not in our list), we will add the intermediate file to the global configuration at runtime, which can then be seen by the whole system.

Besides, we have implement a utility called *hotpatcher* which can be used to insert a dynamic shared object (DSO) into a user program, so that one can deploy this tracing system without having to reboot the machine and affecting existing services. This utility can also be used to *partially* deploy the tracing system, which is very helpful at the time of testing.

Under the hood we have interposition in libc, using the LD_PRELOAD technique [**?**] or using the *hotpatcher* utility program described above. Whenever a program try to operate on the data of interest (e.g, reading or writing), a piece of information, which is necessary for data tracing afterward, will be collected by our interposition code and then sent to a message queue in a piece of shared memory. The shared memory is set up by a daemon called *KMagent* when it is started (usually at boot time) and is shared among all processes at that single machine. *KMagent* is the consumer here, continuously polling the shared message queues and collecting data, whereas all the other processes are producers. The message queue in shared memory is wait-free and operations on it are guided by a protocol to make them efficient and safe, affecting no existing services. As will be demonstrated below, with the current implementation, the overhead within a single event is about $1us$ on our[– TODO machine specification –]. With proper sampling strategies, we show that our implementation have only negligible impact on file events, 10% overhead on process event and 10% overhead on network event.

Periodically a central service try to pull from all machines to collect newly generated data. We use a *bulk mode* [**?**] to make this operation cheap. Besides, our system is *data-oriented*, which makes the volume of collected data small, thus placing only low burden on the system.

There has already been lots of works in using tracing techniques to gather information about distributed system behaviors [**?**, **?**]. However, those works almost all focus on performance analysis and trouble-shooting. Most of them are targeted to some specified program (*e.g.*, web server) [**?**] and require help from the operation system kernel. Compared to them, our attention is on tracking how data is accessed and transferred, and then determine whether those operations are legitimate. Our system can affect *every* process on a machine with sufficiently low overhead. With the implementation in userspace, the whole system is also easy to test and customize. We draw some ideas from [**?**, **?**]. Our model can capture user behaviors precisely. Along the way, we design an simple-yet-efficient system which impose negligible impact on existing services.

To summarize, our contributions are:

1. a model that can precisely capture program behaviors and track how a piece of data are accessed and transferred.

2. a system with efficient implementation which impose negligible impact on existing services.

3. the ability to query and control strategies used in the system and the ability to deploy the whole system while requiring no reboot.

The structure of this paper is as follows: in section 2 we present the model we use to capture user behaviors and tracing data transfer. We then show details of our implementation in section 3, along with its strategies. In section 4, we present some experiments carried out on a small cluster and show that it has negligible impact on existing services. We discuss related work and future work in section 5 and section 6 respectively and then conclude in section 7.

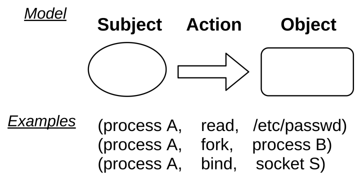
Model
Subject
Action
Object
Examples
(process A, read, /etc/passwd)
(process A, fork, process B)
(process A, bind, socket S)

Figure 1: The Subject-Action-Object Model

2 Model

In this section we present the model used by KM. In KM, any activity related to a piece of data is called an *event*, which is modeled with a triad *(Subject, Action, Object)*, as shown in Figure 1, with examples. All events are classified into three categories: *file event*s, *network event*s and *process event*s. If process A read a file */etc/passwd*, this "read" operation will be recorded by our system which then emits a triad *(process A, read, /etc/passwd)*. Similarly, if process A calls *fork(2)* (and probably communicate with its child process afterward), this "fork" operation will also be recorded and finally a triad *(process A, fork, process B)* will be emitted by the system. Same for the network event when a process A bind to socket S.

Since most Unix-like systems (*e.g.*, Linux) treat most things as files, if we can record all the relevant file events, we can then capture most of the key operations related to some particular piece of data, such as reading a file and sending it over a socket, thus being confident in knowing how a file is accessed and transferred. If we also have processes events at hand, we can know how processes interact with each other and build more confidence in file transfer within a single machine. Moreover, with network events, we can know how a file is transferred across machines. Therefore, with information of these three types of events, we can capture user behaviors on some piece of protected data and trace its transfer route in a distributed system.

Trace on The Same Host On a single machine, if a file is under protection, then any access of it by any process will be recorded by our system. Subsequent file operations by that particular process (*e.g.*, writing to a file in */tmp* or send it over a socket) will also be recorded.

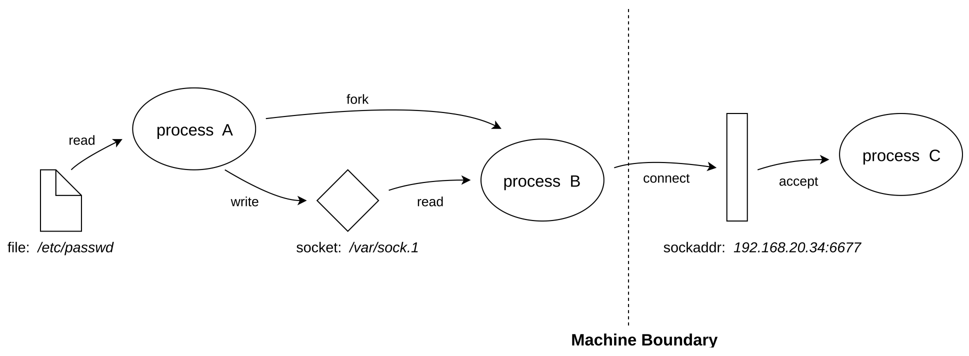

Figure 2: left: file access and transfer in a single machine. right: distributed system

These records are all emitted as file events. Because processes mostly interact with their child processes or parent processes, we also record function calls such as *fork(2)*, *clone(2)* and *execve(2)* and emit those records as process events. Combining file events and process events, we can build a clear picture of how a protected file are accessed and transferred on a single machine, as shown at the left of Figure 2.

Trace Across Hosts In a distributed system, processes in different machines usually communicate with each other through network operation such as *connect(2)* and *accept(2)*. These operations are all recorded by KM which then emits them in the form of network events. These network events bridge the gap of file transfer between different machines, as shown at the right of Figure 2.

TODO probability model

3 Implementation

3.1 System Overview

KM is implemented in userspace and relies on the LD_PRELOAD mechanism [**?**] of the dynamic linker. In short, whenever a program starts, the dynamic linker will load the shared libraries listed in the LD_PRELOAD environment variable before any other libraries that the program depends on, including libc.so. After the dynamic linker is done with symbols relocation, symbols exported by shared libraries in LD_PRELOAD will "shadow" those with the same names in libc.so. Because libc is the common runtime which provides system call wrappers and common routines for application to interact with the operating system and most programs depends on libc.so, we can use this feature to add wrappers around libc's and insert some interposition code between user program code and libc. This is illustrated in Figure 3.

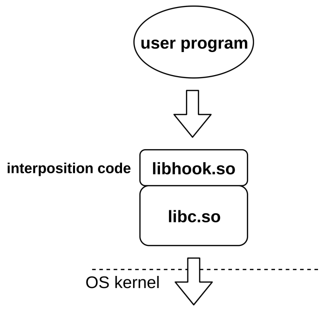

Figure 3: The LD_PRELOAD Mechanism

This schema is common for libc interposition [**?**]. However, to make the implementation safe and efficient is not easy. To make it safe, one have to adhere strictly to the semantic of every API and every piece of code should be multithread-safe and async-signal-safe. To make it efficient, only a small piece of code should be added, because many system calls (*e.g.*, *accept(2)*) will be invoked very frequently. In order to achieve these, our interposition code in every function wrappers contain only a few memory operations. This requires transforming most operations to memory operations. Therefore, a piece of shared memory is set up by the *KM agent* at the very beginning and attached by every hooked user program at program startup. After that, all operations, such as sending function invocation record, can be done through this shared memory. Also, there are configuration rules in the shared memory, which can be referenced by interposition code in every user program.

At program startup, the constructor of `libhook.so` will be invoked, which then allocates a piece of memory in the process space of the current process. This piece of memory is used to hold relevant information of the current process that may be needed afterward. The relevant information will be gathered by reading the `/proc` filesystem (*e.g.*, to get process starttime) or invoking proper system call (*e.g.*, getpid()).

In the interposition code, all our wrappers will perform configuration checking, gather information about this particular function invocation and then send those information to the in-shared-memory message queues, which will be described in section 3.2. All these operations involve only some memory operations. Below is a simplified wrapper for *read(2)*:

```
int read_wrapper(int fd, void *buf,
                 size_t count) {

    int fd = read_original(fd, buf, count);
    if(fd < 0)
        return fd;
    if(!read_specified())
        return fd;
    if(!check_conf())
        return fd;
    send_read_info();
    return fd;
}
```

Basically it invokes the original *read(2)* in libc first, and then check configurations in shared memory to see whether or not this file is under monitor and thus operations upon it should be recorded. Because the configuration checking is done through the shared memory, which is attached at program startup, and we employ some hashing techniques (described in section 3.4) for fast searching, it should be fast and safe. After that a traid of file event, as described above in section 2, will be send to message queues in shared memory. Note that we do not have to make explicit effort to obtain information of the current process, because we already have that information stored at some place at the very beginning, as described above. In another word, we "defer" the process of information collection to program startup to speed up the interposition code. The `read_specified()` is specified to this particular wrapper. It is used to filter out some useless wrapper, for optimization. It will be discussed in section 3.4.

On the other side, there is a *KM agent* which keeps periodically polling the message queues in shared memory to collect function invocation records produced by the interposition code in `libhook.so`. There is a protocal between the producers (*i.e.*, the interposition code) and the consumer (*i.e.*, agent) to make this message transfer efficient. We describe that in detail in section 3.2.

All these collected function invocation records are then sent to central service for analysis.

3.2 Efficient Message Passing

One key component in this system is the in-shared-memory message queues which is used to pass message from producers (*i.e.*, the interposition code) to consumers (*i.e.*, *KM agent*)). It is designed as such that message transfer are

1. **Non-blocking**. Because a blocking implementation will not only slow down the interposition code, but also probably change the semantic of some functions that we are going to hook;

2. **Atomic**. By atomic it means message from on process will not get interleaved with messages from any other processes, even though they share the same message queue.

3. **Fast**. In order reduce over, the implementation should be as fast as possible such that it has negligible impact on existing services.

As of this writing our implementation meets all the three requirements above, with a overall overhead of about 1 *us* each function wrapper on our [–machine specification –].

In the shared memory, there are several message queues, the number of which is equal to the number of CPU. Upon program startup, each process will attach to the message queue belonging to the CPU where the process run. Since most processes will spend most of their lifetime running on the same CPU [**?**], assigning each process a "local" message queue will greatly reducer false-sharing, thus boosting performance of the whole system.

Each message queue is essentially a ring buffer [**?**] with a *index* that can be advanced by both the consumer and producers:

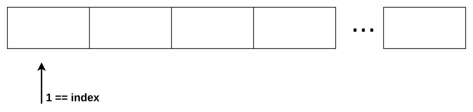

Figure 4a: A single message queue

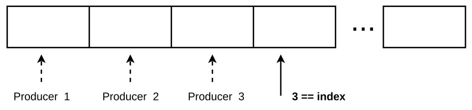

Figure 4b: After three fetch-and-add operations

There are only one consumer (*i.e.*, *KM agent*) but many producers (*i.e.*, the hooked processes). Upon sending message, producers will advance the *index* using the fetch-and-add atomic operation such that every producer get its own slot. This process is illustrated in figure 4b. Every producer simply take the module of the result of the fetch-and-add operation on the size of the queue to avoid index overflow. To collect messages from producers, the *KM agent* periodically polls all the message queues and read the slots one by one.

Synchronization Between A Consumer and A Producer. In each slot, a flag variable is used to synchronize between a consumer and any one of the producers. Before collecting information in any slot, the consumer will check whether that flag is set to SLOTFULL. If so, it will go ahead reading the slot, after which the flag is set to SLOTEMPTY.Before writing to any slot, the producer will check whether whether that flag is set to SLOTEMPTY. If so, it will go ahead writing information in that slot, after which the flag is set to SLOTFULL. Because there is only one consumer, there is no need to use any atomic operation on that flag. A ordinal read and write will suffices. Therefore, synchronization between the consumer and producers is cheep and fast.

Synchronization Among Producers. Synchronization among producers is more tricky because simply taking the modulo of the result of the fetch-and-add operation on the size of the queue (to avoid index overflow) will probably result in two process collict in the same slot. To avoid this, an internal lock is setup and atomic operations are used to operation on it. This internal lock is a 8 bit integer so operation on it should be fast. The advantages of using this kind of home-brew lock with atomic operation is not only its speed and simplicity, but also that it can be turned into a robust lock so that any process which accidentally dies while holding this lock will not prevent the corresponding slot being used by other process. To achieve this robustness, we use fetch-and-add atomic operation for locking and take advantage of the automatic wrap-around effect of an unsigned integer. When a process exits without unlocking this lock, other processes which try to lock this lock with fetch-and-add operation will eventually overflow the underlying unsigned integer to zero, effectively unlocking it. Because we use a single byte as the underlying lock, 255 times of fetch-and-add would suffice. And because the message queue is long enough, the overflow will not happen too fast. The overall algorithm for this process are shown in Algorithm 1.

Algorithm 1 Synchronization Among Producers

```
1: procedure SYNCPRODUCER
2:     if $flag = SLOTFULL$ then
3:         return false
4:     if $0 \neq fetch-and-add(lock)$ then
5:         return false
6:     copy message to slot
7:     unset flag and lock altogether
8:     return true
```

3.3 Forward Tracing

3.4 Optimization

3.5 DSO Runtime Injection

4 Experiments and Evaluation

5 Related work

6 Future work

7 Acknowledgments

A polite author always includes acknowledgments. Thank everyone, especially those who funded the work.

8 Availability

It's great when this section says that MyWonderfulApp is free software, available via anonymous FTP from

`ftp.site.dom/pub/myname/Wonderful`

Also, it's even greater when you can write that information is also available on the Wonderful homepage at

`http://www.site.dom/~myname/SWIG`

Now we get serious and fill in those references. Remember you will have to run latex twice on the document in order to resolve those cite tags you met earlier. This is where they get resolved. We've preserved some real ones in addition to the template-speak. After the bibliography you are DONE.

References

[1] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.