A Framework for Program Recovery Using Checkpointing in UNIX

Kamal R. Prasad
Kip Macy
kamalp@acm.org

Abstract

Checkpointing is a means to save a copy of a temporal copy of a process, so that it can be resumed at a later point of time. The C language is a loosely typed language which makes it possible via pointers to corrupt the process address space. In addition, programs like network daemons are exposed to various types of denial of service attacks -which exploit loopholes in the standards to which the daemons adhere. User-directed checkpointing can help the program to recover itself in case of a crash and possibly use information logged during execution to prevent a similar crash. We discuss an implementation of checkpointing on DragonFlyBSD of one such framework.

1. Introduction

A typical C program has an initialization state, one or more states wherein it waits for input and transitions to other states based on input and finally a termination state. In between these states, the process can be said to be in transition. The initialization state is started by invoking main() from libcrt0.a and the termination state is reached by calling _exit(). In case of a malicious pointer activity which could be either a result of overflow or unexpected input, the process space gets corrupted but no signal is sent immediately on account of C being a loosely typed language. By the time the signal is sent, the process space is corrupted beyond any remedy. In effect, the least expensive remedy is to restore the process to one of the numerous states.
We describe the ideas that have gone into
the implementation of such a checkpoint and recovery mechanism in DragonFlyBSD which is a fork of the FreeBSD 4.4 distribution.

2. Implementation

The implementation consists of a few system calls which the program requiring a recovery would have to use. The basic system call is sys_checkpoint() to save and restore the process image. For convenience, we have defined two macros:-

```
#define tsetjmp() sys_checkpoint
(CHKPT_FREEZE)
#define tlongjmp(x) sys_checkpoint
(CHKPT_THAW, x)
```

tsetjmp() returns a checkpoint descriptor and sets errno on failure.
tlongjmp() returns 0 on sucess and -1 on failure in addition to setting errno.
tlongjmp() takes a checkpoint descriptor as argument. An argument of 0 will result in the last checkpoint to be restored. Checkpoint descriptors are enumerated from 1 onwards and tlongjmp(1) will result in the first checkpoint being restored.

The routine sys_checkpoint() in case of a freeze, opens a file and writes the process image to disk using the same routine as that for doing a coredump. It creates the target file as /tmp/<programname>.<pid>.<checkpoint descriptor>. The global and private shared memory mappings as well as the open file descriptors are saved in the image. Unlike a normal coredump, we save the list of open file descriptors too. In case of a thaw, it opens the associated file as specified by the checkpoint descriptor and overlays the process with the checkpointed data. If the program were to acquire a global resource eg:- a semaphore and do a tsetjmp() after the resource is acquired and free it up and then do a tlongjmp(), it will result in the program not functioning correctly. This is because theoretically, the system cannot assume a tlongjmp()'s occurrence and even if it did, holding back the resource would break the functionality of releasing the global resource and/or preventing other processes from going forward. The signal mask for the process is also saved and restored.

Often, a programmer wants to retain some state information across a tlongjmp(). For this, we have provided a system call persist() which takes a useraddress and the size of of user memory that must persist across a tlongjmp(). It is defined as:-
int persist(caddr_t loc, int size).
Once the user addr is made persistent, all changes to the useraddress are reflected by doing a copyin() before overlaying the old process image and a copyout() after bringing in the new process image. Should a program error cause corruption of persistent memory, the same corruption will reflect after a tlongjmp() and cause the same error to occur.

To overcome this problem, we have defined another system call:-
int cache(int dir, caddr_t loc, int size);

dir can be IN or OUT. The other two parameters are same as that for persist. Unlike persist, cache() copies into kernel memory and out from kernel to usermemory as and when the system call is executed. The user program can perform range checking and any other tests to ensure the program space is sane before caching the data. This comes at a cost because for every update to the user location, the programmer has to make one system call. So, it is advised to make variables that perform an iteration persistent and cross check after a longjmp() is the values are sane.
In addition, we have defined another system call chkpt_status() which reflects the status of the checkpoint/recovery activity in the current process.

Int chkpt_status(int type).
Type = CHKPT_RECOVERED returns 1 if a recovery has been made.
Type = CKPT_FROM_STATE returns the state from which a recovery has been done. It returns EINVAL if no recovery has been made.
Type=CKPT_TO_STATE returns the state to which a recovery was done. If no recovery was done, it returns EINVA.
Type = CKPT_SIGTYPE returns the last signal that was sent to the process. Most likely, this will be the same signal that caused a recovery (depending on how the signal handler was implemented).
Type = CKPT_NUMCHKPTS returns the number of checkpoints (tsetjmp()) that have been set by the process.

When a program starts, it can be assumed to be in state 0. A tsetjmp() returns the checkpoint descriptor which we can refer to as the identifier for that state. So, as the program keeps jumping back and forth to various checkpointed states -we can look at it as a state diagram involving movement across contexts. If the program jumps recursively to the same state as a result of the same signal type (jump histories can be cached) -the programmer can idenitfy that as a loop caused by the same problem and have an algorithm in

Kamal R. Prasad
Kip Macy
kamalp@acm.org

place to filter out that input. That would be one way to build in immunity from a denial of service attack.

3. Experiments

The pseudo-code below shows how the facility is used.

```
Void my_handler(int signum)
{
  if (signum == SIGSEGV)
      tlongjmp(0);
  else
   tlongjmp(1);
}

main()
{
   signal(SIGSEGV, my_handler);
   if (tsetjmp() <0)
     perror(”tsetjmp”);
   printf(“do you want to corrupt the
program\n”);
   if (getchar()\\'y')
        /* perform an operation that
causes segmentation fault */
  /* continue program execution */
}
```

Saving and restoring the program state for a modular program is staright forward. We have put in place code to ensure that program can re-use pipes after a longjmp(). It should be possible to use sockets and many other types of descriptors because their state is not affected by userspace corruption. A sample code to recover from a problem in processing network information would be as follows:-

----

```
if (tsetjmp() <0)
  perror(“tsetjmp”);
while (len = read(sockfd, buff, 1024))
 buff+= len;
```

We are in the process of making this framework a part of commonly used programs.

4. Future work

The present mechanism is rather inefficient that it does not do a diff between checkpoint states to reduce the amount of information being saved. Further, it does not perform any compression to reduce filesystem usage. Putting in these optimizations can help reduce overhead associated with checkpointing. The current implementation goes through the filesystem to serialize the core to disk. But in case of an embedded, diskless device like a networking router -this results in a lot of overhead as the data is being saved to a ramdisk. It is possible to shunt out the vnodes associated with the process image and that will speed up checkpointing in a system that is usually strapped of computing power. The process image once saved could be combined with a migration mechanism to be moved to another processor in a NUMA architecture -which has access to the corefile via NFS. This in turn can help in load balancing on a rack mounted cluster of workstations.

References

[1]. Jim Plank, Micah Beck, Gerry Kingsley, Kai Li, Libckpt: Transparent Checkpointing under Unix, USENIX Winter 1995 Technical Conference
[2]. M. Litzkow and M. Solomon, Supporting Checkpointing and Process Migration outside the UNIX kernel, In Conference Proceedings, USENIX Winter 1992
[3]. W. Richard Stevens, Advanced Programming in the UNIX Environment, Addison Wesley Reading, Mass 1992
[4]. Marshall Kirk McKusick, The Design and Implementation of FreeBSD 4.4.,

Addison Wesley
[5]. E. N. Elnozahy, D.B. Johnson and W. Zwaenepoel, The performance of consistent checkpointing. In 11th Symposium on Reliable Distributed Systems
[6]. S.I. Feldman and C.B. Brown Igor: A system for program debugging via reversible execution: ACM Workshop Notices, Workshop on Parallel and Distributed Debugging.