Return Value Testing of Linux Applications

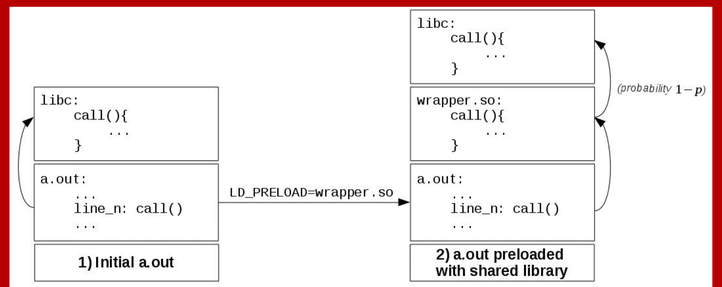
Keith Funkhouser, Malcolm Reid Jr., and Colin Samplawski

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Abstract

We used fuzz testing methods to investigate the robustness of various Linux applications. We used the `LD_PRELOAD` environment variable to perform library interposition for interception of system and library calls. Erroneous return values were injected into the calling applications probabilistically. In a suite of 88 small-scale utilities and large-scale programs, crashes (unintentional core dumps or hangs) were observed for at least one of the 19 intercepted calls in 31 of the 88 applications tested (35.2%). We found a greater incidence of crashes in large-scale applications as compared to small-scale utilities. Memory allocation library calls accounted for the majority of crashes, and small-scale applications crashed exclusively by core dump. Failure to check return values continues to lead to unexpected program behavior, even in some of the most popular open source projects (e.g. Firefox, VLC, and gcc).

Mechanisms



We used the `LD_PRELOAD` environment variable to do library interposition, as shown above.

1. Program makes call to function
2. Call is intercepted by a shared library wrapper for that call
3. With probability p, an error value is returned. Otherwise, the actual call is made, allowing the application to continue as normal

We created a wrapper for each individual call. Within this wrapper, we get the handle of the actual function using the `dlsym` function. `dlsym` scans through the dynamically loaded libraries for a function that has the same name as the argument passed to it.

We used static analysis to auto-generate wrappers. First, we programmatically parsed the abstract syntax tree to discover method name, return type, and argument types. Second, we manually inspected the `man` page for error return values and `errno` (if applicable). This process is illustrated below:

Header file (`stdlib.h`)

```
extern void *malloc (size_t __size);
```

Man Page (`malloc(3)`)

```
RETURN VALUE
...On error, these functions return NULL...
```

Generated Wrapper

```
# define _GNU_SOURCE // needed to compile as PIC
# include <dlfcn.h>  // dlsym
# include <stdlib.h>

// function pointer for real malloc
static void *(*real_malloc)(size_t size) = NULL ;

// malloc wrapper
void *malloc(size_t size) {
  real_malloc = dlsym(RTLD_NEXT,"malloc"); // get handle of real malloc
  if(rand() > 0.5) {
    return real_malloc(size);              // call real malloc
  } else {
    return NULL;
  }
}
```

Results

We used our library interposition method to test the robustness of small utilities and large-scale applications. We primarily focused on finding crashes, which we defined as unintentional core dumps or hangs. The tables below show the calls and applications we tested and which applications crashed.

Calls Tested

`calloc`	`execvp`	`malloc`	`pipe`	`read`
`close`	`fopen`	`mmap`	`poll`	`realloc`
`creat`	`fork`	`open`	`pthread_create`	`write`
`execv`	`fstat`	`opendir`	`pthread_mutex_init`	

GNU Core Utilities (13/51 = 25.5%)

`base64`	`dirname`	**`logname`** (m)	`sort`	`uniq`
`basename`	**`du`** (cm)	`ls`	**`stat`** (m)	`unlink`
`cat`	`echo`	`md5sum`	`sum`	`uptime`
`chmod`	**`expr`** (m)	`mkdir`	`touch`	`users`
`cksum`	**`factor`** (mr)	`mktemp`	`tr`	**`wc`** (m)
`comm`	`fold`	**`mv`** (c)	`true`	**`who`** (m)
`cp` (c)	`groups`	`nproc`	`truncate`	**`whoami`** (m)
`cut`	`head`	`printenv`	`tsort`	
`date`	**`hostid`** (m)	`pwd`	`tty`	
`df` (r)	**`id`** (m)	`seq`	`uname`	
`dircolors`	`link`	`shred`	`unexpand`	

c = crashed with `calloc`, m = `malloc`, r = `realloc`

net-tools (4/11 = 36.4%)

`dnsdomainname` (m)
`domainname`
`hostname`
`ifconfig` (m)
`ipmaddr` (m)
`mii-tool`
`netstat` (m)
`nisdomainname`
`rarp`
`route`
`ypdomainname`

Other Utilities (5/16 = 31.3%)

`ctags`	`man`
`finger` (m)	`sdiff`
`grep`	**`ssh`** (m)
`gunzip`	`tar`
`gzip`	`unzip`
`ip` (m)	**`w`** (m)
`last`	`zip`
`make` (COR, c)	`zipinfo`

C = crashed with `close`, O = `open`, R = `read`

Large-scale Applications (9/10 = 90%)

	Chrome	gvim	Thunderbird	Firefox	VLC	LibreOffice	VirtualBox	gcc	javac	Eclipse
`malloc`	H				C			C	C	C
`open`										
`read`	H									
`write`	H	H	H	H	H		N/A			H
`close`	H		H			H				
`mmap`		C	CH	C	H	C				
`pthread_create`	H			C	CH	H	N/A	N/A		C
`pthread_mutex_init`	H		H	H	H	C	N/A	N/A	H	
`fork`								N/A		
`pipe`	H							N/A		
`fopen`		C	C		C					

H = hang, C = core dump, CH = hang and core dump (in separate runs), N/A = not called, empty = called, but no crash or hang

Crash Analysis

```
+    pn = (struct add_r *) malloc(sizeof(struct addr));
+    if (pn == NULL) {
+      perror("netstat");
+      return (-1);
+    }
     pn->addr = *sin;
     pn->next = INET_nn;
     pn->host = host;
     pn->name = (char *) malloc(strlen(name) + 1);
+    if (pn->name == NULL) {
+      perror("netstat");
+      return (-1);
+    }
     strcpy(pn->name, name);
```

`netstat`: two calls to `malloc` were unchecked (lines beginning with '+' indicate our changes)

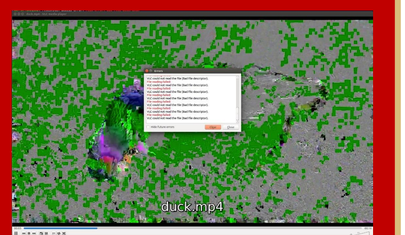
```
    for (ns = 0; ns < statp->nscount; ns++) {
        EXT(statp).nssocks[ns] = -1;
        if (statp->nsaddr_list[ns].sin_family == 0)
            continue;
        if (EXT(statp).nsaddrs[ns] == NULL)
            EXT(statp).nsaddrs[ns] =
                malloc(sizeof (struct sockaddr_in6));
+       if(EXT(statp).nsaddrs[ns] != NULL) {
+           EXT(statp).nscount++;
+       }
        if (EXT(statp).nsaddrs[ns] != NULL)
            memset (memcpy(EXT(statp).nsaddrs[ns],
                           &statp->nsaddr_list[ns],
                           sizeof (struct sockaddr_in)),
                    '\0',
                    sizeof (struct sockaddr_in6)
                    - sizeof (struct sockaddr_in));
    }
-   EXT(statp).nscount = statp->nscount;
}
```

`hostid`: one call to `malloc` was unchecked

Anecdotes

Firefox tab bar with `open`

gvim with `mmap`



VLC with `read`

Conclusions

- We found that large-scale applications were more prone to crashes
  - Large-scale applications accounted for 11.4% of our applications, but 29% of crashes
- Memory allocation functions (`calloc`, `malloc`, `realloc`) accounted for the most crashes, especially in the smaller utilities
- We found that utilities have become more robust since 1995 [1]
- Even the most mature codebases fail to check return values

References

- Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.