Automatically Finding and Patching Bad Error Handling

Martin Süßkraut, Christof Fetzer
martin.suesskraut@tu-dresden.de, christof.fetzer@tu-dresden.de

Department of Computer Science
Technische Universität Dresden, Germany

EDCC-6, 10/18/2006
Motivation

- robust software has to be prepared for the worst case
- analogy: crash tests in the automotive industry

Source: Dread Pirate Roberts (at Flickr)
Bad Error Handling

• Cristian [1]:
  – “... often more than two thirds of the code is devoted to detecting and handling exceptions...”
  – “... exception handling code is in general the least documented, tested and understood part.”
  – “telephone switching system ... two third of system failures are due to design failures in exception handling code”
Bad Error Handling

- Westley Weimar and George C. Necula[2]:
  - 1% to 5% are catch and finally blocks
  - 3% to 46% of program text is reachable from catch and finally
- Pong Vo et. al. [3]
  - “... many bugs can be traced to sloppy handling of exceptions from library functions that are both well-defined and documented.”
Finding Bad Error Handling

• search bad error handling with error injection

```c
/* grep-5.2.1 src/search.c:152 */
152 char *mb_properties = malloc(size);
153 ...
157 memset (mb_properties , 0, sizeof (char) * size);
```

"Crash test": return NULL
Patching Bad Error Handling

- generate patches
- mask bad error handling by:
  - preallocation
  - error mapping
- prevent up to 84% of crashes
Overview

- automatically find & patch bad error handling on interface to dynamically linked libraries
- **no** source code
- tested on Linux Coreutils

**Application:**

```c
char *mb_properties = malloc(size);
```

**Library:**

```c
void* malloc(size_t size) { ... }
```
Automatically Patching Bad Error Handling
Static & Dynamic View

- application binary – static view:
  - specific function call instruction: \textit{call site}
  - identified by code address
- application execution – dynamic view:
  - executed function call: \textit{call}
  - identified by call site + counter
Systematic Error Injection

- static view:
  - for each call site:
    • generate error injector wrapper
- dynamic view:
  - intercept all calls from call site:
    • return error code
Bad Error Handling

![Bar chart showing the number of bugs found in different programs.](chart)

- unsafe calls
- sum
- uname
- wc
- df
- grep
- md5sum
- du
- sort
- unzip
- touch

EDCC-6, 10/18/2006  Automatically Patching Bad Error Handling  Slide 11
Patch Patterns

• preallocation
  – do resource reservation before unsafe call
  – need to know arguments
• error code mapping
  – return error code at safe call if unsafe call failed
Getting Arguments for Preallocation

• hard coded within binary: **static arguments**
• otherwise: **dynamic arguments**
  – record samples from typical runs
  – use max/avg of samples for preallocation
  – compensative measurements if actual arguments differ
Preallocation

/* grep-5.2.1 src/search.c:152 */
152 char *mb_properties = malloc(size);
153 ...
157 memset (mb_properties , 0, sizeof (char) * size);

• if preallocation fails:
  – retry until unsafe call is executed
  – return error code at safe calls
Error Code Mapping by Example (1)

/* grep-5.2.1 src/dfa.c:3240 */
static void freelist (char **cpp)
{
    int i;

    if (cpp == NULL) return;
    for (i = 0; cpp[i] != NULL; ++i) {
        free(cpp[i]);
        cpp[i] = NULL;
    }
}

CRASH
if mp[i].in uninitialized
Error Code Mapping by Example (2)

```c
/* grep-5.2.1 src/dfa.c:3423 */
mp[i].in = (char **) malloc(sizeof *mp[i].in);
mp[i].left = malloc(2);
mp[i].right = malloc(2);
mp[i].is = malloc(2);
if (mp[i].in == NULL || mp[i].left == NULL ||
    mp[i].right == NULL || mp[i].is == NULL)
    goto done;

/* line 3623 */
done:

/* line 3636 */
freelist(mp[i].in);
```

Free resources:

Error handling:
Error Code Mapping

- patch call groups:
  - map errors from unsafe calls to safe calls

```c
mp[i].in = (char **) malloc(sizeof *mp[i].in);
mp[i].left = malloc(2);
mp[i].right = malloc(2);
mp[i].is = malloc(2);
```
Patch Call Groups

f1: unsafe, dynamic args.
f2: safe
f3: unsafe, static args.
f4: safe
f5: unsafe, dynamic args.
f6: unsafe, static args.
Analysis Summary

- Error Injection
  - identify unsafe call sites
- Static Analysis:
  - extract static arguments
  - find call groups
- Argument Recorder
  - fallback for dynamic arguments
Patching Summary

• shift execution of unsafe calls forward
  – retry
  – error mapping
• error mapping
  – map error from unsafe call sites to safe call sites
  – especially call groups
Evaluation

- prototype: memory allocation functions (malloc, calloc, and realloc)
- influence on performance: at most 10% slowdown
- tested patches with 4 error pattern
  - systematic
  - knockover
  - resource limit
  - probabilistic
Error Pattern

- systematic error pattern
  - simulate inability to fulfill **one** specific request
  - let all calls to a specific call site fail
- knockover error pattern
  - simulate complete system outage at a specific point in execution
  - let **all** calls fail after a specific call site is reached the first time
Systematic Error Injection
Knockover Pattern

The diagram shows the comparison between the number of crashes without and with the patch applied. The x-axis represents different programs (sum, uname, wc, df, grep, md5sum, du, sort, unzip, touch), and the y-axis represents the number of crashes. The red bars indicate the number of crashes without the patch, and the blue bars indicate the number of crashes with the patch.
Resource Limit Pattern

![Graph showing resource limit pattern for different commands]
Probabilistic Pattern

<table>
<thead>
<tr>
<th>Command</th>
<th>Crashes Without Patch</th>
<th>Crashes With Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>uname</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>wc</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>df</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>md5sum</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>du</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>unzip</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>touch</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Summary

- robust software needs crash tests
- automatic approach to find and patch bad error handling
  - found 10 bugs in Coreutils
  - patch pattern
    - preallocation
    - error code mapping
  - generated patch masked >80% of all crashes
References