Robustness and Security Hardening of COTS Software Libraries

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

Department of Computer Science
Technische Universität Dresden, Germany

37th DSN, 06/26/2007
Motivation: Issues in Safety Critical Systems

- build up on unreliable components
- missing isolation (e.g., in embedded systems)
- large number of components

➔ automatically harden unreliable components
Robustness Benchmark – Ballista [1]

Normalized Failure Rates for 233 POSIX Calls

AIX 4.1
DUNIX 3.2
DUNIX 4.0D
FreeBSD 2.2.5
HPUX 9.05
HPUX 10.20
Irix 5.3
Irix 6.2
Linux
Lynx
NetBSD
QNX 4.22
QNX 4.24
SunOS 4.13
SunOS 5.5
Containment Wrappers – HEALERS [2]

\[ \text{strcpy (dest, src);} \]

- unsafe arguments
- safe arguments
Containment Wrappers – HEALERS [2]

```c
strcpy (dest, src);
```

- unsafe arguments
- safe arguments

```

- `WRITEONLY`
- `NULL`
- `READONLY`

- `NULL_WRITE`  
- `NULL_READ`  

- `READ_WRITE`

```
Containment Wrappers – HEALERS [2]

```c
strcpy (dest, src);
```

- unsafe arguments
- safe arguments

- NULL_READ_WRITE
- WRITEONLY
- NULL
- READONLY
- NULL_WRITE
- NULL_READ
- READ_WRITE
Containment Wrappers – HEALERS [2]

```c
strcpy (dest, src);
```

- unsafe arguments
- safe arguments

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEONLY</td>
<td>READONLY</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL_WRITE</td>
</tr>
<tr>
<td>READONLY</td>
<td>NULL_WRITE</td>
</tr>
<tr>
<td>WRITEONLY</td>
<td>READONLY</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL_WRITE</td>
</tr>
<tr>
<td>READONLY</td>
<td>NULL_WRITE</td>
</tr>
</tbody>
</table>
```

37th DSN, 07/26/2007
Robustness and Security Hardening of COTS Software Libraries
Slide 7
Shortcommings of previous approaches

- difficult to apply to non-standard libraries
  - not extensible
    - disjoint sets of test values for test types
  - not flexible
    - test values for fault injection coupled to safe argument type computation
Our Approach

✔ decoupled
  - fault injection and robust argument type computation
✔ extensible
  - fault injection
  - robust argument type computation
Decoupling

- $\text{strcpy}(\text{char}* \text{ dest}, \text{ const char}* \text{ src})$
- $\text{check}_1 = \text{string}?(\text{src})$
- $\text{check}_2 = \text{buf\_write}?(\text{dest, strlen (src)} + 1)$

<table>
<thead>
<tr>
<th>check$_1$</th>
<th>check$_2$</th>
<th>robust?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- minimize truth-table: $\text{check}_1 \text{ and check}_2$
Extensibility

- easily add
  - new checks
  - new test cases

<table>
<thead>
<tr>
<th>test case</th>
<th>check_1</th>
<th>check_2</th>
<th>...</th>
<th>check_n</th>
<th>robust?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

1. extensible dependability testing
   • generate test types with static analysis
   • generate checks with static analysis
2. containment wrappers
3. evaluation
Extensible Test Type System

- root
- pointer
- string
- filename
- format string
- handle
- Ballista type
- meta type
- generated type
- meta_string
- meta type
Security with Feedback Types

• use feedback to adjust test values
• can detect buffer overflows

1\textsuperscript{st} run

buffer
Security with Feedback Types

- use feedback to adjust test values
- can detect buffer overflows
Security with Feedback Types

- use feedback to adjust test values
- can detect buffer overflows
Security with Feedback Types

- use feedback to adjust test values
- can detect buffer overflows
Generated Test Types

- handles
  - transparent references to hidden state
  - experiments on Apache Portable Runtime Lib:
    - found 13 handle types
    - one is used by 44 functions
- data structures
  - specialized feedback type
- mount generated types with meta types
Map C Types to Test Types

- static analysis on public sources ("header files")
  - pointer?
  - sizeof
  - converts_to_int?
  - signed?
  - content_size
  - ishandle?

- fixed mapping to static and generated Test Types
## Type Mapping Example

<table>
<thead>
<tr>
<th></th>
<th>short</th>
<th>char*</th>
<th>apr_socket_t*</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sizeof</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>converts_to_int?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>signed?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>content_size</td>
<td>n/a</td>
<td>1</td>
<td>68</td>
</tr>
<tr>
<td>ishandle?</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Test Type</td>
<td>wchar + short</td>
<td>string</td>
<td>meta_apr_socket_t</td>
</tr>
</tbody>
</table>
## Type Mapping Example

<table>
<thead>
<tr>
<th></th>
<th>short</th>
<th>char*</th>
<th>apr_socket_t*</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sizeof</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>converts_to_int?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>signed?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>content_size</td>
<td>n/a</td>
<td>1</td>
<td>68</td>
</tr>
<tr>
<td>ishandle?</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Test Type</strong></td>
<td>wchar + short</td>
<td>string</td>
<td>meta_apr_socket_t</td>
</tr>
</tbody>
</table>

- meta_apr_socket_t
- handle_apr_socket_t
- feedback_apr_socket_t
- meta_generic_ptr

**Generated Type**

**Static Type**
Other Sources of Test Inputs

- record arguments from running applications
- bit-flips on recorded arguments
Checks

- two uses
  - classify test values to build truth table
  - check arguments in containment wrappers

- three types of checks
  - about a single function argument
  - about two functions arguments
  - generates checks

- mapping from C types like to Test Types
Basic Checks

- checks are bound to one specific argument
- example:
  ```c
  apr_socket_create (apr_socket_t**   a1,
                    int              a2,
                    int              a3,
                    apr_pool_t*      a4);
  ```

  some checks:
  - `is_NULL?(a1)`
  - `is_zero?(a2)`
  - `is_positive?(a3)`
  - ...
Compound Checks

- combine two function arguments
- example:

  ```c
  apr_socket_create (apr_socket_t** a1, int a2, int a3, apr_pool_t* a4);
  ```

  some checks:
  - `writeable_fixed_sized_buffer?(a4, a2)`
  - `readable_fixed_sized_buffer?(a4, a3)`
  - ...
Generated Checks

• generated from C type and feedback
• example:
  
  \texttt{apr\_socket\_create (apr\_socket\_t** \ a1, \ 
  \texttt{\int \ a2, \ 
  \texttt{\int \ a3, \ 
  \texttt{apr\_pool\_t* \ a4); \ 

some checks:
  
  – \texttt{writable\_buffer\_size4?(a1) \ 
  – \texttt{readable\_buffer\_size68?(a4) \ 
  – \texttt{is\_handle\_apr\_pool\_t?(a4) \ 
  – ...}
## A Function's Protection Hypothesis

<table>
<thead>
<tr>
<th>check_1</th>
<th>check_2</th>
<th>...</th>
<th>check_n</th>
<th>robust?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- double rows ➔ ignore
- missing rows ➔ treat as robust
- contradicting rows ➔ treat as robust

\[
\text{hypothesis}_{\text{strcpy}} = \text{check1 and check2}
\]
Containment Wrapper

• prevent function's execution if its hypothesis is not fulfilled
  
  ```c
  containment_wrapper_strncpy (...) {
    if (!hypothesis_strncpy)
      return error_code;
    <call original function>
  }
  ```

• `error_code` derives from functions return type
Evaluation

• tested in Apache Portable Runtime (APR)
  – no specific adjustments for APR
• at most 10,000 tests per function
• performed ~1,000 test per minute
• run containment wrappers with Apache webserver
Robustness per Function
Containment Wrappers

- run with Apache
  - wrapped 77 functions
  - without faults
  - no false positives
- bit-flips in function arguments
  - prevented 56.81% of all crashes
Containment Wrapper under Bit Flips

- Prevented crashes: 39.6%
- Unprevented crashes: 30.1%
- Correctly predicted no crash: 28.6%
- Falsely predicted crash: 1.7%
False Positives

• tolerated
  – by application's error handling
• can be eliminated/reduced with test values from:
  – golden runs
  – bit-flips
Conclusion

- flexible, extensible, general dependability benchmark
  - instantiate various Test Types and checks
  - generate truth table over checks
- containment hypothesis
  - minimize truth table
  - expression build from checks
- prevent more 56% of crashes
References


[3] Süßkraut, M. & Fetzer, C. Automatically Finding and Patching Bad Error Handling; Sixth European Dependable Computing Conference (EDCC'06); October 2006