Code Testing at OpenVision's Security Business Unit, Cambridge Office

Jonathan Kamens

April 6, 1994

Outline

- Why I'm the one giving this talk
- Reasons to do code testing
- Kinds of code testing
- Available code testing tools
- Case study – how we do things

Q&A

Question-and-answer period at the end, but questions during the talk are welcomed.

Why I'm the one giving this talk

Worked for Athena doing bug tracking, which I actually enjoy.

Went to work for OpenVision (formerly Geer Zolot Associates) after leaving Athena. Was appointed our QA person (bug tracking, testing, etc.) because of my experience at Athena.

Designed most of the tests we currently use, implemented many of them.

Talk about 6.170

Reasons to do code testing

To squash bugs

More specifically:

- Detect coding errors

- Verify that code behaves as documented, as required by standards, or as required by product specifications

- Make porting easier

- Verify that “fixes” do what they’re supposed to

- Minimize adverse effects of environment changes (e.g., a new compiler)

Benefits of testing

- Better testing means fewer bugs.
- Fewer bugs means happier customers.
- Happier customers means a better reputation and a more successful company.

Kinds of code testing

Testing categories

Description	Nickname
“Hacker testing”	
Memory error detection	mem
Boundary error detection	bound
Stress testing	stress
Code coverage	cover
Code complexity	complex
Code ”linting”	lint
Code efficiency/analysis	analysis
GUI testing	gui
UI testing	ui
Portability	porting
Standards	standards

Related problems

Interactive debugging	debug
Test driving	drive

These areas overlap; for example, interactive debuggers such as CodeCenter do MEM, BOUND, and LINT as well.

DRIVE is very important – tests are *much* more effective in the long term if they are automated.

Available code testing tools

CITE (stress, cover, gui, ui, drive, analysis)

CODE CHECK (complex, analysis, standards, porting, lint)

CodeCenter (mem, bound, debug, lint)

dbmalloc (mem)

dbx (debug)

DejaGnu (ui, drive)

Expect (ui, drive)

GCT - Generic Coverage Tool (bound, cover)

gdb (debug)

Lint (lint)

McCabe code complexity (complex)

mprof (mem)

Purify2 (mem, bound)

Quantify (analysis)

Sentinel (mem)

TestCenter (mem, bound, cover)

TET - Test Environment Toolkit (drive)

XRunner (gui, drive)

How we do things

Our overall approach

1. Design and implement unit tests for individual components.
2. Run component unit tests on all platforms whenever a component is changed.
3. Use unit tests in conjunction with { mem, bound, cover } tools during development.
4. Run all unit tests for all components as "system test." Run system test automatically every night when close to a code milestone.
5. Require that system test passes completely before shipping a product.

Tools we use (most-used to least-used)

- DejaGnu
- XRunner
- Purify2
- gdb
- CodeCenter
- dbmalloc
- mprof
- GCT

We also use TCL wrappers for testing APIs.

Spend some time here talking about how DejaGnu and XRunner work.

How we design unit tests

- We do *not* require complete component isolation.
- Tests are usually black-box, based on the functional specification for the component.
- If possible, the test designer is someone other than the author of the code.
- Each test has a unique identifier that is referred to in the test implementation.

Walking through an example

Function prototype:

```
ovsec_kadm_ret_t
 ovsec_kadm_init(char *client_name,
                 char *pass,
                 char *service_name,
                 char *realm)
```

One test for this function:

Number: 1

Reason: An empty string realm is rejected.

Status: Implemented

The implementation of the test:

```
one_line_fail_test_nochk \
 {ovsec_kadm_init admin admin \
  $OVSEC_KADM_ADMIN_SERVICE ""}
```

Following it through:

```
proc one_line_fail_test_nochk {command} {
    global prompt
    global spawn_id
    global test

    send "$command\n"
    expect {
        -re "\nERROR .*$prompt$"
                { pass "$test:"; return 1 }
        -re "\nOK .*$prompt$"
                { fail "$test: bad success"; return 0 }
        "wrong # args"
                { error "$test: wrong number args"; return 0 }
        timeout
                { fail "$test: timeout"; return 0 }
        eof
                { fail "$test: eof"; api_exit; api_start; return 0 }
    }
}
```

But what is it sending to? Here's the function that starts the client:

```
proc api_start {} {
	global API
	global env
	global spawn_id
	global prompt

	spawn $API
	expect {
		-re "$prompt$" {}
		eof { error "EOF starting API" }
		timeout { error "Timeout starting API" }
	}
	if {! [info exists env(TCLUTIL)]} {
		error "TCLUTIL environment variable isn't set"
	}
	send "source $env(TCLUTIL)\n"
	expect {
		-re "$prompt$" {}
		eof { error "EOF starting API" }
		timeout { error "Timeout starting API" }
	}
#	puts stdout "Finishing api_start (spawn_id $spawn_id)."
}
```