Testing

How to break software

Acknowledgements: Michael Huth
Objectives

• To appreciate basic terminology of software testing
• To study an effective fault model in detail and apply it to widely used applications
• To familiarize yourself with a basic runtime fault injection tool
• To develop a tester’s mindset when testing your own applications in projects etc.
Why do we test software?

For different purposes:

- *Acceptance testing*: should we buy?
- *Conformance testing*: meets implementation standard?
- *Usability testing*: easy to use?
- *Performance, reliability, robustness, system integration testing* etc.
Commonalities of all testing

• Testers work from description of software’s behaviour (e.g. source code, binaries)
• Product is executable in real or simulated environment (e.g. run-time environment or code review)
• Product’s functionality explored in a methodical manner
• Know whether a test has negative or positive outcome.
Our focus

Demonstrate how to explore software functionality in intelligent manner regardless of how little documentation we have.

Emphasis on

• *Black-box testing*: no access to internals of software

• *Grey-box testing*: inferring internal structure from the software’s interfaces.
Our fault model

Definition: A *fault model* is a model for how and why faults in code manifest themselves as faulty behaviour at run-time.

A simple but effective fault model:
- Software runs in *specific environment*; need to understand it and test accordingly
- All software has *inherent capabilities* that we can use to infer and test its behaviour.
Challenge of testing software

Too many inputs/outputs.

Too many input/output values.

Too many input combinations.

Too many internal states of stored data.
Possible solutions

High-cost solutions: theorem-proving, model checking, and other formal methods.

Cost-effective solutions: experienced testers with good fault model.

Note: our fault model is independent of choice of programming language or paradigm.
Four user groups of software

- Operating system kernel: e.g. memory, file pointers, time services
- File system: permanent storage in binary, hexadecimal etc
- Human interface: a set of GUIs receiving input from the keyboard, mouse etc
- Other software: return values of API calls, e.g. databases, library functions etc.
Human-user abstraction

Human input typically triggers activities at all four levels.

Example: changing the font in PowerPoint requires ten calls to two kernel functions.

- Input through GUIs, menus, and command line
- Input through APIs conceptually similar to Human input.
File system user

For testers, files are/should be users, their contents is input.

• Software works if files are in specified format?
• Software works if files are corrupted?
• E.g. what if file-access privileges change dynamically?
Operating system user

E.g. Windows 2000 supplies 1000+ kernel functions, each may return an “error” value.
Does the software under test cope with exceptions raised by the kernel?

Problem: how to test behaviour safely/realistically under severe memory or network constraints?

Solution: use a tool that simulates these effects.
Software’s capabilities

Our fault model is rooted in software’s capabilities, software can

1. Accept input from its environment
2. Produce output and transmit it to environment
3. Store data internally and react depending on that internal state
4. Perform computation based on input and stored data.
Testing strategies

• 1. or 2. black-box testing: no knowledge of internals; we will use that

• 3. or 4. white-box testing: some knowledge of internals; we turn that into grey-box testing: inferring software’s internals

• Link to a page on “unit testing,” a white-box testing, in Java:
  www.doc.ic.ac.uk/~jnm/se_third_yr/testing-notes.pdf
Four capabilities, four tests

1. Testing input: prevents acceptance of valid input? Invalid combination breaks software?
2. Right format or quantity? May require domain knowledge. (e.g., Feb. 29, ’01)
3. Data stored, retrieved, and modified in anticipated way? Default values in right places?
4. Right feature interaction and safe computation result?
Summary

Test software for many reasons, but always

- Establish the environment in which software runs
- Test its capabilities within that environment.

Environment has four user types: operating system, file system, human user, and other software.

Software has four types of capabilities: input, output, date (state), and computation.
Outlook

In the next lectures we will try to answer directly:

• How does one study systematically an application domain’s environment?
• How does one establish the capabilities an application has?

Indirectly:

• What makes a good test good and a bad test bad?
• How much testing is enough?
Exercises

1. List what kind of testing you have already done on what kind of software.
2. Discuss the pros and cons of developers testing their own software.
3. Given a spectrum of threat rankings “very high, high, medium, moderate, low, very low” give examples of bugs in applications for each ranking and justify your ranking.
User-interface attacks: input & output
“Attacks”

“Attack” effective metaphor for testing mindset.

Four categories of attack, applied one at a time:

1. Accepting input
2. Producing output
3. Storing data
4. Performing computation.
Tester’s mindset

Purpose of software typically unclear (no requirements document).

Expected behaviour of software unclear (no written specifications).

Strategy: don’t apply ad-hoc input; act like a user who “gets real work done.”

Example: for a word processor, create and edit documents etc.
Attack #1

Apply input that forces all *programmed* error messages to occur.

Rationale: error-handling code is hard to get right; its development often deferred or done sloppily; error states are very complex (e.g. all files may still be open).
How to attack

Developers may use

- Input filters (prevent invalid entry)
- Input checking (e.g. “switch” statements)
- Exception handling (last resort; clean up mess).

Do you see each intended error message?

Vary input type/length, use boundary values.

**Example:** -2,147,483,649 for a 4-byte signed integer.
Demo #1

Attack #1 at work on Microsoft Word 2000:

- Launch Word,
- Choose “Insert” menu,
- Select “Index and Tables” tab
- Change “Column” field to “5” and press enter key.

⇒ Error message appears, but twice!
Attack #2

Apply inputs that force the software to establish default values.

Rationale: check whether developers enforced the right order “variable declared $\rightarrow$ variable assigned $\rightarrow$ variable used;” default values may get established late in development process. Satisfactory choice of defaults for users?

(Some instances can be checked at compile-time.)
Testing oracles

General problem:

• How to obtain a good *oracle* that provides correct program behaviour?

• Without good oracle, don’t know *actual* outcome of a test.

➤ You may see an error without realizing that it is one!
How to attack

• Look for option buttons, configuration panels, setup screens etc
• Build intuition of how entered data is used
• Consult data declaration sections of code
• Enter null value and/or delete displayed values
• Change default value and then re-change it to original default value
• Enter valid value and change it back to default value.
Demo #2

Attack #2 at work on Microsoft Word:
• Launch Word and select “Insert” menu,
• Choose “Index and Tables” item,
• Click “Table of Contents” tab,
• Click “Options” button, and press enter.
➡ Part of the static display disappears, part of the dynamic view changes.
Attack #3

Explore allowable character sets and data types.

Rationale: if strings are valid input, which characters are treated in special way?

Examples: “&” in C, reserved names for kernel functions, control characters in ASCII etc.
How to attack

Consult reference tables for relevant programming languages, operating systems, and character sets. Then attack.

Attack may result in software that hangs or renders wrong string output.

Example: Microsoft Internet Explorer 5.5; type “file://c:AUX” in URL field, resulting in “cannot find server, not responding.”
Attack #4

Overflow buffer inputs.

Rationale: string-based attack; developer may not have foreseen handling of very long strings. Can be security concern.

Probability of overflow in user context of software?

Successful attack typically crashes software under test.
How to attack

Attack by exploring string boundaries of input fields.

Then use boundary values.

Then exceed those values and input them.
Demo #4

Attack #4 at work on Microsoft Word 2000:
• Launch Word and select the “Edit” menu,
• Click on the “Replace” tab,
• Enter short string next to “Find what,”
• Enter 200+ characters long string under “Replace with”
➡ Forces the application under test to terminate with unspecified errors.
Attack #5

Find inputs that may interact and test combinations of their input values.

Rationale: developers may test each input for validity but combination of valid inputs may be invalid and not filtered or handled.

Attack may hang or crash software, or corrupt data and resulting output.
How to attack

Difficult in general as interaction of input is very complex.

Example: number of columns 1 – 63, number of rows 1 – 32,767; and so 2,064,321 many interactions possible.

Aggressive abstraction techniques on input domains.
Demo #5

Attack #5 at work on Microsoft Word 2000:

• Launch Word,
• Select “Insert” option from “Table” menu,
• Verify that input fields cannot be overflowed,
• Enter e.g. 52 and 32,000 for number of columns and rows (respectively); each is valid input
  ➔ The software hangs or may crash.
Attack #6

Repeat the same input or series of inputs numerous times.

Rationale: software may fail after repeated input when same input is being re-applied, as this may consume resources and cause data initialization problems.

Attack can result in unpredictable behaviour (stresses memory usage).
How to attack

Repeat inputs that are likely to be repeated by typical user:

• Download several files,
• Nest parentheses in mathematics package
• Apply text formatting command on each paragraph of text etc.
Example

Microsoft Equation 3.0 spots boundary level of 10+ nesting of parentheses:
• e.g. enter ((((((((((((3+5))))))))))))
• clear resulting error message/notification

⇒ equation “disappears” and cannot be recovered.
Attack #7

Force different outputs to be generated for each input.

Rationale: some inputs have rich contexts and may require context-dependent output.

General point about output-based attacks: harder to do as they require an “intuition” about “inverse” computation.
How to attack

Identify inputs in which context is likely to be important.

Try to determine all legal outputs for that input and try to generate them all.

Example: switching software; picking up the phone in context in which
• That phone rings
• That phone doesn’t ring and user wishes to make a call.
Attack #8

Force invalid outputs to be generated.

Rationale: testers may have intimate knowledge of problem domain so they can tell legal outputs from illegal ones. Useful tool to think about worrisome combinations of input.

Attack will not show anything unusual, but output values are “semantically” wrong.
Example

Bug in old Windows NT system, fixed in service pack 5:

- display February 2000 in system calendar,
- then increment year by one

➔ still see 29 days in February
Attack #9

Force properties of an output to change.
Rationale: some outputs are editable, changeable, or may somehow come back to haunt the software under test. Helps testers think about persistent output. Code is often changed in initialization method but not in update method. Related to attack on stored default values.
How to attack

Attack involves painstaking screen verification.

“Output size” is a prime candidate for persistent output properties to attack. Example: clock display; changes from 9:59 to 10:00; or from 12:59 to 1:00 for a US clock.
Demo #9

Attack #9 at work on Microsoft PowerPoint:

- in PowerPoint open WordArt feature,
- enter long string, e.g. “Now is the time for all good students to test the robustness of their software,”
- Press “OK” and obverse the effect,
- Change string to “N” only, press “OK” and study new effect,
- Change string to a rather long paragraph of text, press “OK” and study the effect.
Attack #10

Force the screen to refresh.

Rationale: applications often need to write objects to a window and display it to a user. Overlays, moves, and resizing requires frequent refreshing. Helps identify problems with screen output.

Developers walk on a thin line: refresh too often → application slows down; refresh too little → annoyance or prevention of use.
How to attack

Vary distance of object moves.
Overlay objects to touch on edges of objects of varying types.
Force one object to leave its object container.

Etc.
Demo #10

Attack #10 at work on MS PowerPoint:
• Launch PowerPoint,
• Insert a text box on blank page,
• Type “abc” and the superscript “123”
• Change font size of superscript to 30 or more and observe the effect,
• Move the text box around to see more effects.
Commonalities of attacks

All attacks studied so far are

• Goal-oriented,
• Increase trust (or mistrust) in reliability of software under test.
• Help you develop an understanding of the software’s behaviour.
Summary of lecture on User Interface Attacks

Input attacks:
• Ensure all error messages occur when needed
• Force software to assign default values accessible through user interface
• For all input fields, enter wrong type or strings with special default meaning
• For all input fields, enter maximal or more number of characters allowed
• Determine legal inputs and test its combinations
• Apply the same input over and over.
Summary (continued)

Output attacks:

• Focus on context-rich input, generate output for each context
• Force illegal output through combination of legal input
• Apply input that changes “persistent” properties of output
• Determine when and how software is refreshing screen: too often? Too little?
Exercises

Pick any application of the MS Office 2000 suite as software under test.

1. Apply each input attack to your application. Document the attacks you applied, the features that worked as specified, and the bugs you found.

2. Repeat the activities of previous item for all output attacks.
User-interface attacks: data & computation
Methodological remarks

These attacks traditionally in “white-box testing:” access to source or byte code.

Here: grey-box testing of data & computation by “looking through the interface.”

Examples: if information entered once and displayed in second field or can be queried on demand, it is likely to be stored.

Testers with programming skills may “reverse engineer” internal data structures and control.
Rationale for grey-box testing

Source code or executable code is often not available.

Exploration of user interface often allows inferences about which data are stored, or whether and how they are manipulated by the software under test.
Attack #11

Apply inputs using a variety of initial conditions.

Rationale: attack may find situations in which internal data is incompatible with certain input or output sequences. Correct handling of these incompatibilities is complex and error prone.
How to attack

Ask yourself: “What preconditions of stored data are likely to make this input not work?”

Example: “save” of a file, when open, open and modified, new etc.

Example: “group / ungroup” feature in Microsoft Word where group members are of the same type, different type (the latter is buggy).
Attack #12

Force a data structure to store too few or too many values.

Rationale: checks that boundaries on arrays and other data structures are enforced. Unchecked, this could result in underflow or overflow and corrupt data.

(This is a prime target for unit testing on source code!)
How to attack

Try to add element if data structure is “full.”

Try to remove element from “empty” data structure e.g.

• Queues
• Stacks.
Demo #12

Attack #12 at work on Microsoft Word 2000:
• Launch Word,
• Open the “Table” menu and launch “Insert”
• Choose small number of columns,
• Choose “32,767” for number of rows (can you increase that?) and observe effects.
Attack #13

Investigate alternative ways to modify internal constraints of software.

Rationale: More general version of #12; tests whether error code is present at creation, edits, and modifications of data structure.

May be difficult to detect, but consequences can be catastrophic.

Example: limit “undo” to nesting of 20; years may have (semantic) range 1980 – 2095 only etc.
How to attack

Ask yourself:
• “What is the allowable range of values?”
• “Under what constraints does the data operate?”
• “Are there contexts in which data can exist?”

Then initialize target data, modify them, and check integrity with respect to constraints.
Demo #13

Attack #13 at work on MS PowerPoint:
• Launch PowerPoint,
• Choose “Table” from the “Insert” menu,
• Specify “25” for number of columns and rows and press “OK”
• Some versions offer a tool box in which you could increase number of rows to 26+ crashing the program.

Next attacks explore computation and feature interaction…
Attack #14

Experiment with invalid operand and operator combinations.

Rationale: verifies that internal computation does not proceed with incorrect data values; testing invalid cases increases our trust in reliability.

Almost all operators have invalid operands. Invalid applications typically result in a complete crash or require careful exception handling.
How to attack

Identify places in which computation occurs.

Get general idea of what data is being used.

Vary that data to achieve invalid operands.
Demo #14

Attack #14 at work on the MS calculator:

- Launch shipped calculator,
- Compute e.g. “2 SQRT MS * MR * MR * MR,” resulting in display of “4”
- Then type “- 4” resulting in display of “5.068967601371654149798031145533e-37”

Stored and displayed value out of sync. User gets false sense of security.
Attack #15

Force a method to call itself recursively.

Rationale: It may be that an object can interact only with *other* objects and developers have not written code to enforce that.

Domain knowledge often vital for this attack.

**Example**: displayed web page that automatically executes a script which reloads its host page etc.

Successful attack typically $\rightarrow$ heap overflow.
Demo #15

Attack #15 at work on Microsoft Word 2000:

• Launch Word and select “Footnote” from the “Insert” menu,

• Write something into that footnote, then create another footnote within that footnote,

• Observe the *unintended* effects

⇒ Recursive call not anticipated, strange change of state.
Attack #16

Force computation results to be too large or too small.

Rationale: this attack will often overflow the storage set aside for the result of a computation.

Usually requires you to force a computation to occur over and over or to occur on very large or small data values.
Example #16

```c
const count 2;
main() {
    int sum, value[count];
    sum = 0;
    for (i = 0; i < count; ++i) {
        sum = sum + value[i];
    }
}
Crashes if value[0] = 32700, value[1] = 70 \(\Rightarrow\) result > 32,767; or if count = 33000, value[i] = 1, all i.
Attack #17

Find features that share data, interact poorly.

Rationale: this attack helps identify common characteristics of features and tests that these cannot be broken during interaction.

Example: a word processor can produce footnotes and two columns; correct interaction?
How to attack

Search for features with overlapping input or output domain.

Ask if one feature’s computation can obstruct the computation of some other feature.

In general it is difficult to make this attack effective.
Summary

Data attacks:
• Test context-rich input for all possible and some impossible initial contexts
• “look through the interface” to infer internal computation and data structures
• Apply too few or too many values to data structures
• Determine constraints on data-structure creation and usage; find other ways to break such constraints.
Summary

Computation attacks:

• Find “places” of computation and force operators to be placed with invalid operands
• Force an internal function to call itself
• Force a computation that overflows storage for computation result
• Pick a feature and think of other features that may interact with it in a bad way.
Exercises

1. Repeat the first exercise for previous lecture with all data and computation attacks of this lecture.

2. Choose any programming language and make a partial list of operators supported by that language. For each listed operator specify which operands are valid/invalid. For which attack is such a list of use?
Testing from the file interface
General rationale

Testers commonly assume that files are not the source of problems with applications. But: mishandling files may result in permanent data corruption. Corrupted files are similar to corrupted user input with similar grave consequences.

Media-based attacks: simulate problems with storage media such as a hard disk.

File-based attacks: simulate corrupt files etc.
Attack #18

Fill the system to its capacity.
Rationale: despite ever increasing and cheaper mass storage, disk space often gets full. A simulation guarantees that the software under test performs gracefully under those circumstances.

Obvious reasons for preferring a simulation environment for this attack ➔ simulated run-time fault injection.
How to attack

We use Canned HEAT 3.0 (CH 3.0) “Hostile Environment Application Tester,” downloadable at

http://www.se.fit.edu/CannedHEAT/Downloads.htm

Simulate full disk, then make application open, create, save, modify, delete files etc.

Please do not install this software on any department machine!
Demo #18

Attack #18 at work on MS Internet Explorer:

1. Launch CH 3.0, choose “Open an executable,”
2. browse to and open IEXPLORE.EXE,
3. Click “Application” and “Run,”
4. Under “Disk Storage” select “Insufficient Disk Space,”
5. Try to save the displayed web page

⇒ Netscape 4.7: “error while closing file.”
Attack #19

Force the media to be busy or unavailable.

Rationale: this attack ensures that developers write and enforce appropriate error codes in case that media is busy or unavailable. Otherwise, the application will fail. Excessive delays should generate error codes. (Small delays should not be interpreted as bugs.)
Demo #19

Attack #19 at work on MS Internet Explorer:

• First three steps as in Demo #18
• Under “Disk Storage” select “Too many files open,”
• Try to save the currently displayed web page and observe effects

⇒ Netscape 4.7 crashes and disappears.
Attack #20

Damage the media (under no circumstances to be applied within the college!).

Rationale: application may be safety critical and tests with damaged media may be prescribed by law, executive directive or some government agency.

We now move to file-based attacks…
Attack #21

Assign an invalid file name.

Rationale: try to identify whether the software under test has states in which it enforces no constraints or the wrong constraints on file names. Otherwise, the application may crash or files will be corrupted or hard to find.
How to attack

Depending on operating system at hand
• Choose invalid file names for files
• Use software under test to open, create … files under such a name.

You will see either error messages or the software is likely to crash.
Demo #21

Attack #21 at work on Microsoft Word 2000:
• Launch Word,
• Select “New” under the “File” menu,
• Type some text into edit panel,
• Select “Save as” under “File” menu,
• Enter “startrecks:starwars-8.1.2001” under file name and observe whether saved file is recognized as Word document.
Attack #22

Dynamically vary file access permissions.

Rationale: can identify subtle bugs when a file used by an application is manipulated outside the control of that application.

Example (if you have those privileges): open a file with low permissions, upgrade those permissions, and then save that file again to see what happens next.
Attack #23

Varied or corrupt file contents.
Rationale: attack shields against intentional (e.g. malicious) or incidental (e.g. transmission error) modification of a file.
Application may crash or may produce corrupted data.

Example: edit HEX file, or use CH 3.0.
Summary of lecture 10

Media-based attacks:
• Can your software handle a full storage medium?
• Can your software gracefully deal with a busy file system?
• Try forcing your software through file operations with a damaged (simulated) medium.
Summary of lecture 10

File-based attacks:

- Assign invalid file name to the software’s data files, temporary files, or read-only files
- Change access permissions of files
- Determine whether your software can handle corrupt data in a file.

Testing the file-system user often overlooked but just as often cause of faulty or crashing application. Such tests are possible with runtime fault injection tools.
Exercises for lecture 10

1. Choose any MS Office 2000 application, make list of features and identify features that make application to go to disk or access file system.
2. Install Canned HEAT on your personal machine.
3. Apply each of the file-based attacks to your application and document your findings.
4. Apply each of the media-based attacks to your application. Document your findings.
Testing from the Software/OS Interface
General rationale

Software typically makes “API calls” to other software, e.g. network drivers or the operation system.

User-interface attacks focus on “normal” software-to-software interaction.

But such interaction could be “abnormal,” e.g. a buggy memory-allocation routine.

Record-and-simulate attacks ask: What faults should be injected at run-time?
Three kinds of answers

1. *Developer-centric*: inject faults that cause all error-handling code to be executed and all exceptions to be tripped
2. *Tester-centric*: inject faults that can be readily staged in the testing lab
3. *User-centric*: inject faults that might realistically occur in the “field.”

Simulation tools allow for a user-centric approach.
Methodology

1. Use memory monitor to determine “memory hogs” of your application.
2. Test memory-intense functions to determine at what memory-size they are still doing OK.
3. Test these functions under randomly varying memory conditions.
4. Then test them for specific fault injections.
Demo

Software under test: MS Internet Explorer

1. Downloading pages obvious memory hog
2. Threshold around 50% on my machine
3. Varying memory size ➔ “memory could not be written”
4. Specific faults: copes well with “random failures” of network on my laptop.
Evaluation

Canned HEAT: course-grained fault injection
Holodeck 1.1.0 lite (very limited demo version):
www.sisecure.com/holodeck/

More fine-grained, surgical approach. Want ability to fail a specific call in specific context.

Observe-and-fail attacks: able to see which API calls are made, intervene on call-by-call basis.
Demo (with commercial version)

1. LoadLibraryExW causes external-code libraries to be loaded in MS IE.

2. Holodeck: MSRATING.DLL provides services to that application: prompts for password if rating of site meets password-protection criteria.

3. Instruct Holodeck to simulate that MSRATING.DLL cannot be opened.

   Causes protection feature to be disabled, allowing access to entire web.
Summary

• Simulation tools for run-time fault injection allow a user-centric stance for software testing
• First subject software to various memory failures, focussing on memory-intense features
• Then inject specific failure
• Record-and-simulate attacks specify failure specific to call context of applications, requires fined-grained simulation tools.
Exercises

Launch Canned HEAT and use the methodology outlined in this lecture to test your favourite web browser application.

1. List functions that are memory hogs, apply all fault scenarios under the “Memory” tab and document your findings. For which memory threshold is your application still working?

2. Repeat the previous exercise with fault scenarios under the “Network” tab. At which minimum bandwidth is your application still working?
Conclusion of part 3

• Bulk of MS software is *test code* and we still could find bugs after plenty of (in-house) white-box testing

• Freeware applications, if subjected to these attacks, show similar effects

• Testing may also happen on requirements or formal, executable specifications

• Choice of attacks may be informed by risk model, which attacks are realistic/likely in the use context of the software under test.