CITS3007 Secure Coding
Secure software development

Unit coordinator: Arran Stewart

Highlights

- Threat modelling with STRIDE
- Testing basics

Threat modelling with STRIDE

Four questions

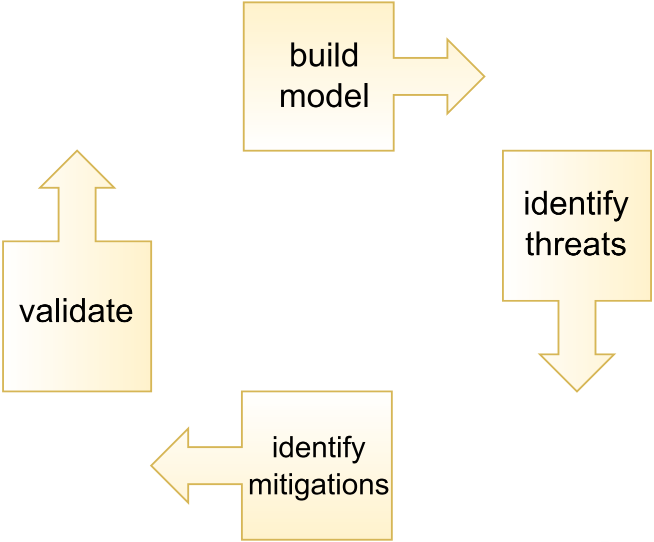
Approach from Adam Shostack at Microsoft:

1. What are we building?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good job?

Four questions – activites

1. What are we building?
   - Outcome: a *model* or diagram of the system (and identified assets)
2. What can go wrong?
   - Outcome: prioritized list of threats
3. What are we going to do about it?
   - Outcome: prioritized list of mitigations or countermeasures
4. Did we do a good job?
   - Outcome: validation of prior steps; tests; gaps identified; improvements to process

Iterable


build
model
identify
threats
identify
mitigations
validate

Scope

“Small and often” is better than “comprehensive, but never finished”.

- *Full* inventory of all potential threats for a large, complex system could be huge
- But it’s better to do *something than nothing*, and it’s better to identify the *most critical* threats than to aim for completeness

First pass = focus on biggest, most likely threats, to high-value assets

- Other assets and threats can be dealt with later; scope can be increased

What are we building?

The aim is to produce a *model* or high-level description of the system, including assets (valuable data and resources) that need protection.

- Traditionally:
  - data flow diagram (DFD), or
  - Unified Modelling Language (UML)

But *any* sort of model will do.

Could be a design document or a box-and-arrows whiteboard sketch.

Level of detail

No model is perfect – it is a useful *simplification* of reality.

- Needs enough granularity that we can analyse it, identify assets and threats
- Always possible to iterate the process later with more detailed models if necessary
- Too little detail $\Rightarrow$ details will be missed
  Too much detail $\Rightarrow$ the work will take too long

Iterating a model

We can always note down spots where a model could be improved later.

Phrases to watch out for: “sometimes”, “also”.

- “*sometimes* this data store is used for X”, “this component is *also* used for Y” $\Rightarrow$ more detail could be useful

Identify assets

These are things we want to protect.

Usually data.

But could also be:

- hardware
- information technology resources (like bandwidth, computational capacity)
- physical resources (electricity)

Can you think of threats targeting these?

Identify assets

Assets should be *prioritized* – which are most important?

- We could try to hide *everything* about our server, for example
  - But is the best use of our time?
- Compare server details with (e.g.) financial data, password hashes, cryptographic keys

Identify assets

Don't try and put a dollar value on assets. Avoid superfluous and unrealistic granularity.

- One idea: categorize with "T-shirt sizes"
- "Large" (major assets), "Medium" (valuable assets, but less critical), "Small" (minor consequence).
  - ... maybe your project needs "Extra-large" (super-critical)?

Remember other parties/stakeholders' viewpoints – something *you* think is of "minor consequence" could be much more important to (e.g.) a customer, CEO, finance, etc.

What can go wrong? – Identify threats

Methodically go through the model, component by component, flow by flow, looking for possible threats.

Identify

- *attack surfaces* (places an attack could originate)
  - Points where an attacker could interpose themselves
- *trust boundaries* (the borders between more-trusted and less-trusted parts of the system)
  - These will intersect data flows
- threats in each of the possible STRIDE categories.

Tip: threats often lurk at trust boundaries.

Identify attack surfaces

These are an attacker's "points of entry", or opportunities for attack. (For example: communication over a network.)

- When we look at mitigations – try to completely remove, or at least reduce, opportunities for attack

Identify attack surfaces

Physical example: we have a building we want to secure.

- What's better – many exits and entries?
- Or: just a single exit and entry, which we can monitor carefully, and have (e.g.) security screening, metal detectors at.

Identify threats

For each of the STRIDE categories – e.g. tampering – we ask, What advantages could an attacker gain if they did/subverted X?

A suggested approach: brainstorm first – come up with ideas quickly, without critiquing or judging them yet

Analyse, understand and prioritize threats

For each identified threat:

- flesh out the details
- try to assess the chance of them happening
- assess what the impacts would be

Analyse, understand and prioritize threats

- For probability and impact – no need for exact numbers – just use a point/level system (e.g. 1 to 3, 1 to 5)
  - Give your levels labels – “likely”, “unlikely”; “minor impact”, “showstopping / enterprise-destroying”
- Be cautious of unrealistic levels of granularity – can you *really* distinguish “5%” versus “7.5%” probability, or “3/10” from “4/10”?

Ranking threats

Microsoft "DREAD" model:

- Damage: How great would the damage be if the attack succeeded?
- Reproducibility: How easy is it to reproduce an attack?
- Exploitability: How much time, effort, and expertise is needed to exploit the threat?
- Affected users: If a threat were exploited, what percentage of users would be affected?
- Discoverability: How easy is it for an attacker to discover this threat?

What are we going to do about it? – mitigations

- Propose ways of dealing with each threat – usually called “mitigation” or “countermeasures”.
- But in full: either mitigate, remove, transfer, or accept.

Mitigations and other approaches

- *Mitigate* risk by redesigning or adding defenses.
  - The aim is either to reduce the chance of the risk occurring, or lower degree of harm to an acceptable level
- *Remove* a threatened asset if it is not actually needed
  - Or if removal is not possible – seek to reduce the exposure of the asset, or limit optional features of your system that increase the threat.

Mitigations and other approaches

- *Transfer* the risk – offload responsibility to a third party.
  - Example: Insurance is a common type of risk transfer
  - Example: Outsource responsibility for e.g. processing payments, or processing sensitive data, to an enterprise with expertise in the area.
- *Accept* the risk (once it's well understood) as being reasonable to incur.

Mitigations – questions to ask

- Can we make the attack less likely to work?
- Can we make the harm less severe – perhaps only some of the data is accessible?
- Can we make it possible to undo the harm – e.g. backups?
- Can we make it more obvious when harm has occurred – e.g. by ensuring we have comprehensive logging and monitoring?

Did we do a good job? – validation, review and testing

- Validate previous steps, act upon them, look for gaps missed
- Test the efficacy of mitigations, from most to least critical

Validation

- For a model – does it match what has actually been implemented?
- For threats – have we describe them properly? missed any?
  - do they: describe the attack, the context, the impact?
- Other stakeholders – have testing/quality assurance staff reviewed the model?
- Mitigations – is each threat mitigated (or otherwise dealt with)
- Are the mitigations done correctly? Have they been tested?

Testing basics

Overview

- What is the purpose of testing?

What is testing? Why test?

- Testing is a systematic attempt to find faults in a software system in a planned way.
  - "Faults", or "defects", are anything in the system that causes it to behave in a way different from its specification.
- We test because it's much cheaper (monetarily, and in cost to an organization's reputation) to find faults *early*, before software is released, than after

Testing functions

- How do we know what a function is supposed to do?
  $\Rightarrow$ Refer to its documentation.
- Could be
  - a man page (e.g. for `strlen`)
  - extracted API documentation

Testing functions

strlen

```
NAME
       strlen — calculate the length of a string

SYNOPSIS
       #include <string.h>

       size_t strlen(const char *s);

DESCRIPTION
       The strlen() function calculates the length of the string pointed
       to by s, excluding the terminating null byte ('\0').
```

- Are there any implicit requirements here?

API documentation systems

- An API comprises all functions, variables, and macros that are publicly available and documented as such
  - If a function says – “this is not part of the API” – you rely on it at your own risk
- Many languages come with API documentation generation tools
  - e.g. Javadoc for Java, Pydoc and Sphinx for Python, Godoc for Go
- C does not
  - Common tools used to extract C API information include Doxygen and cldoc (based on the Clang compiler)

Doxygen

Doxygen extracts API information from specially marked up comments – **documentation blocks**.

```
/** @brief Sets the position of the cursor to the
 *         position (row, col).
 *
 *  Subsequent calls to putbytes should cause the console output to
 *  begin at the new position. If the cursor is currently hidden, a call
 *  to set_cursor() must not show the cursor.
 *
 *  @param row The new row for the cursor.
 *  @param col The new column for the cursor.
 */
void set_cursor(int row, int col);
```

Documentation blocks vs comments

- In C, API documentation is normally embedded in C comments
- But don't think of them as "comments" – they serve an entirely different purpose
- *Comments* are for maintainers and implementers of the code
  - Explain why and how something is implemented
  - Should be brief, and not clutter the code
- *Documentation* is for users of the code
  - Explains what the callers must do, and what they can expect
  - Can be as extensive as needed

API contents

What should go in the API documentation?

- The *preconditions* – any conditions which should be satisfied by the caller when the function is called.
- The *postconditions* – the return value of the function, and any *changes* the function makes to the system state ("side effects")

The specification for a function is like a **contract** between the caller of the function and the implementer:

> *"If you, the caller, satisfy the preconditions, then I, the implementer, promise the postconditions will be true afterwards."*

If the preconditions are *not* satisfied, the behaviour of the function is undefined.

API contents – bsearch

bsearch - binary search of a sorted array

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Description

The bsearch() function searches an array of nmemb objects, the initial member of which is pointed to by base, for a member that matches the object pointed to by key. The size of each member of the array is specified by size.

The contents of the array should be in ascending sorted order according to the comparison function referenced by compar. The compar routine is expected to have two arguments which point to the key object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the key object is found, respectively, to be less than, to match, or be greater than the array member.

API contents – bsearch

bsearch - binary search of a sorted array

Return value

The bsearch() function returns a pointer to a matching member of the array, or NULL if no match is found. If there are multiple elements that match the key, the element returned is unspecified.

Tests in C

What do tests look like in C?

```
// test of strlen
const char * s = "somestring";
size_t expected_result = 10;
size_t actual_result = strlen(s);
if (expected_result != actual_result) {
  fprintf(stderr,
          "%s:%d: expected len to be %zd, but got %zd\n",
          __FILE__,
          __LINE__,
          expected_result,
          actual_result
  );
  exit(EXIT_FAILURE);
}
```

"Arrange, Act, Assert"

When writing tests, it's useful to follow the "Arrange, Act, Assert" pattern:

Arrange prepare any needed resources (variables, data structures, files, external programs, etc.)

Act invoke the behavior we want to test. (In C: calling a function.)

Assert Look at the resulting state of the system and see if it is what we expected.

Testing frameworks

- The disadvantage of the `strlen` test we saw before is that the program exits once a test is failed – annoying, if multiple tests need to be run
- **Testing frameworks** may handle tasks including
  - identifying and running a user-selected set of the tests
  - helping ensure the system is in a known state before a test is run
  - providing developers with utility functions so they can write the "Arrange" and "Assert" parts of a test
  - providing "mocks" for expensive or hard-to-use parts of the environment
  - reporting tests results in a useful format (either human- or machine-readable)

check

In labs, we will use the Check testing framework.

Install with:

```
sudo apt-get install check
```

There are many others, but Check has a number of advantages:

- Doesn't require any special build tools – `gcc` (and Make, if desired) are enough
- Protects the address space of the program under test using `fork`

Address space problems

In C, testing is more difficult, because the testing framework runs in the same address space as the function being tested.

If the function being tested corrupts memory, it could prevent the testing routines from working.

Check addresses this by `fork`-ing off a separate copy of the program for each test –

$\Rightarrow$ every test has its own address space.

Testing using check

- Easiest way: write tests in a test suite (“.ts”) file, and use the checkmk program to convert them into full .c code

mytests.ts

```
#suite adjust_score_tests

#tcase arithmetic_testcase

#test arithmetic_works
    int m = 3;
    int n = 4;
    int expected = 7;
    int actual = m + n;
    ck_assert_int_eq(actual, expected);
```

Compiling and running tests

We usually want to enable protective features and sanitizers

$\Rightarrow$ if a function fails, it fails as early and obviously as possible.

```
# compile
$ gcc -g -std=c11 -pedantic -Wall -Wextra -Wconversion \
    -fno-omit-frame-pointer \
    -fstack-protector-strong \
    -fsanitize=address,undefined,leak \
    -c -o myprog.o myprog.c

$ gcc -g -std=c11 -pedantic -Wall -Wextra -pthread \
    -fno-omit-frame-pointer \
    -fstack-protector-strong \
    -fsanitize=address,undefined,leak \
    -c -o mytests.o mytests.c
```

Compiling and running tests

```
# link
$ gcc -o mytests mytests.o myprog.o \
    -lcheck_pic -pthread -lrt -lm -lsubunit \
    -fsanitize=address,undefined,leak

# run
$ ./mytests
Running suite(s): adjust_score_tests
100%: Checks: 2, Failures: 0, Errors: 0
mytests.ts:65:P:arithmetic_testcase:arithmetic_works:0: Passed
mytests.ts:91:P:filesize_works_testcase:filesize_small_works:0: Passed
```

Mocks

When writing a test, we often want

- the test to run as fast as possible
- the test results to rely only on the function under test – not other extraneous systems

So what if we're testing a function that reads information from a file or database?

⇒ This is very slow, and adds a dependency on the filesystem or DBMS
⇒ If a test failure was reported, was it due to our function or the DB?

Mocks and test doubles

Mocks or **test doubles** (more general term)

- Actors use doubles to replace them during certain scenes
  - Dangerous or athletic scenes
  - Skills the actor doesn't have, like dancing or singing
- Test doubles replace software components that cannot be used during testing

Reasons for test doubles

- Component has not been written
- The real component does something destructive that we want to avoid during testing (unrecoverable actions)
- The real component interacts with an unreliable resource
- The real component runs very slowly
- The real component creates a test cycle
    - A depends on B, B depends on C, C depends on A

C support for test doubles

- To mock *files*, we can use `memfd_create` – provides an "in-memory" file
- To mock *functions* – tricky but various solutions
  - gcc provides the "weak" attribute for functions – allows for library functions which can be *overridden/replaced* by user code