CITS3007 Secure Coding
Analysis and testing

Unit coordinator: Arran Stewart

Highlights

- Avoiding vulnerabilities
- (Automatic) program analysis
- Static analysis techniques
- Dynamic analysis techniques
- Fuzzing

Avoiding vulnerabilities

- In lectures and labs, we've now seen a few example of things you should do (e.g. sanitizing inputs) and things to be wary of (e.g. wraparounds, overflows) when implementing software
  - Some vulnerabilities (buffer overflows) might seem only relevant to C – but many other languages are implemented in or use C
  - Others (sanitization problems, integer wraparounds) are relevant to nearly all languages.
- Are there any general tools or approaches for ensuring we do the good things, and avoid the bad things?

Avoiding vulnerabilities

- In lectures and labs, we've now seen a few example of things you should do (e.g. sanitizing inputs) and things to be wary of (e.g. wraparounds, overflows) when implementing software
  - Some vulnerabilities (buffer overflows) might seem only relevant to C – but many other languages are implemented in or use C
  - Others (sanitization problems, integer wraparounds) are relevant to nearly all languages.
- Are there any general tools or approaches for ensuring we do the good things, and avoid the bad things?
- "Ensuring"? Sadly, no. But there are many things that can help.

Avoiding vulnerabilities

There are things we can do during

- analysis & design
- implementation
- testing, and
- maintenance

to reduce the chance that vulnerabilities will occur in our software (and ameliorate the effects if they do).

In general, security is not something you can just "add in" at, say, the implementation or testing phase – it has to be considered at all phases of the software development lifecycle (SDLC).

Avoiding vulnerabilities

In this unit, we mostly look at the implementation and testing phases.

But in future lectures, we will briefly look at secure development methodologies, which cover all phases of the SDLC.

Analysis & design

- Security *requirements* can be developed in tandem with *threat modelling* – where we identify potential security threats.
- Threat modelling asks the question, "What could go wrong?"
- It helps us identify threats, and also mitigations we'll put in place – we'll do that during later phases of the SDLC.
- Problems with *design* can often be found by human review of the design.

Implementation and testing

During implementation and testing, we'll implement mitigation strategies identified earlier.

We also can use static and dynamic *analysis* techniques to help identify potential problems in our code, and thorough testing to identify breaches of our security requirements.

Program analysis

What is it?

- the process of using automated tools to analyse the behaviour of computer programs for particular properties
  - e.g. correctness, security, speed, termination

Areas where it's used:

- Compiler development
  - Compilers have to analyse code, and turn it into executable binaries or bytecode
- Verification
  - We may want to verify that a program is *correct* (implements its specification properly)
- Security
  - We want to find code that can lead to vulnerabilities

Program analysis approaches

- Static analysis: performed without executing the program.
  - Uses the program's *static* artifacts (usually, source code, but sometimes binary executables)
- Dynamic analysis: performed at runtime.
  - Actually runs the program (or part of it)
- Hybrid: a mix of the previous two.

Related techniques

If a *human* analyses the code, we don't usually call that "program analysis".

- In the static case, we just call it "reading the code", or "code review"
- In the dynamic case, we call it "debugging" or "manually running tests"

The borderline can be fuzzy, though. Program verification isn't *completely* automated, for instance – it requires a great deal of human input.

Static analysis examples

Static analysis: analysis performed without executing the program.

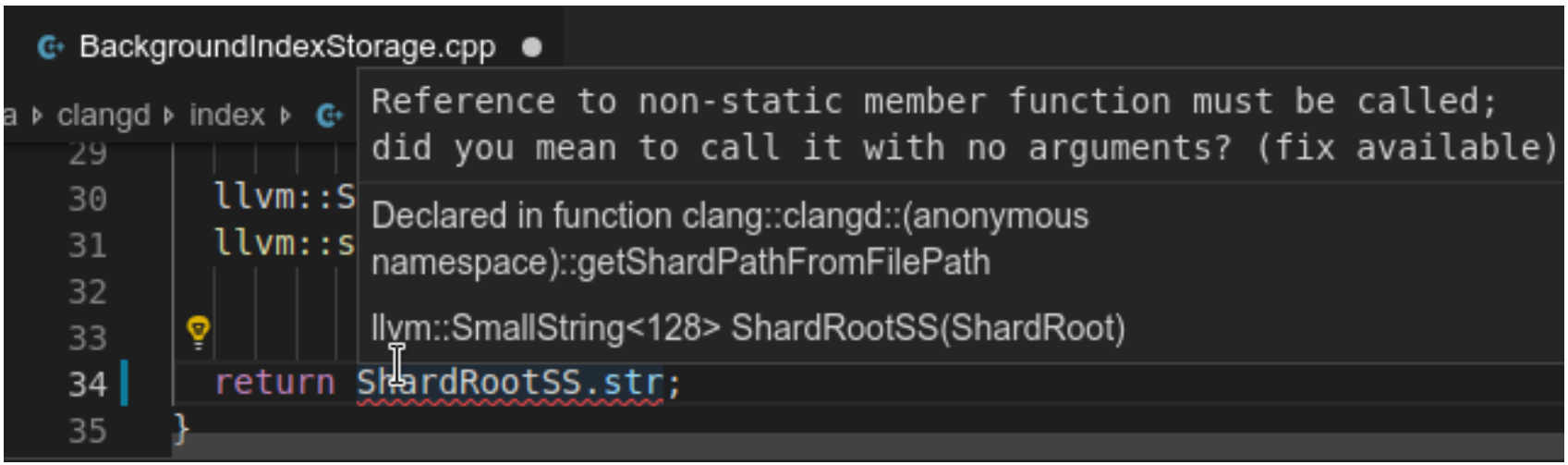
Some examples:

- Flawfinder
  - Developed by David A. Wheeler, director of security at the Linux Foundation
- clang-tidy
  - Created by the developers of the Clang compiler

We use both of these in lab 6 ("static analysis") – they warn about problematic code constructs.

Static analysers are also sometimes called "linters" or "bugfinders" (depending on their focus).

Static analysis examples



Many IDEs and editors provide ways of integrating warnings from static analysers into your development environment

- e.g. They may show warnings as red underlines in your code, with "hoverable" details

Static analysis examples

Static analysis does cover a very wide range of techniques.

From very very simple – e.g. `grep`ping code for functions like `strcpy`, known to be unsafe – to very complex.

Targets of static analysis

Many static analysis programs operate on the source code for a program, but some instead analyse compiled binaries.

For example, the Ghidra framework can be used to analyse binary code.

We won't use any binary analysis techniques in this unit, but they can come in handy when doing (for instance) penetration testing.

Dynamic analysis

Analyses which require the program to be *run* in order to work.

A common approach is to inject extra instructions into the code when it is compiled, to answer questions like "Do any array accesses go out of bounds when we run this program?"

Dynamic analysis examples

Valgrind:

- Actually a suite of analyses – the most common is `memcheck` – plus a framework for developing new ones.
- Simulates the effect of every instruction in the original program (fairly slow)
- Doesn't require you to re-compile your code with instrumentation: uses the original binary, and inserts extra instructions (e.g. for out of bounds writes)

Dynamic analysis examples

Sanitizers, developed by Google

- code is actually part of LLVM, a compiler infrastructure framework
- the sanitizers:
  - AddressSanitizer (ASan) – detects buffer overflows and other, similar memory errors
  - ThreadSanitizer (TSan) – detects data races and deadlocks)
  - UndefinedBehaviorSanitizer (UBSan) – detects various sorts of undefined behaviour
  - plus many others
- incorporated into both the `clang` and `gcc` compilers
- require you to recompile your program and link to special libraries, so you code can be properly *instrumented* (i.e., have additional code inserted)

Dynamic versus static analysis

Dynamic analysis can be very fast and precise.

- e.g. Google sanitizers will give very precise reports on where (say) undefined behaviour or out-of-bounds memory access occurs, and usually only add about 10-15% to runtime
- However, you're limited to only the code that was actually executed – you may have very limited *coverage*.
  - If you don't run your program with the right inputs, you may never discover a particular vulnerability
  - One solution to this: instrumented fuzzing (more on this later)

Dynamic versus static analysis

Static analysis can have perfect coverage (since the whole source code is available)

Doesn't need to run the program – may even be able to detect problems "as-you-type" (ALE in `vim` will do this)

However, answering some questions (e.g. "Will array accesses ever be made out of bounds?") are intractable for static analyses.

Static analysis concepts

Dynamic versus static analysis

Static analysis cannot be as precise as dynamic.

```
if halts(f):
  reveal_all_the_secrets()
```

Instead, static analyses *approximate* the behaviour of the program: they provide either false positives or false negatives.

False positive Reporting a program has some property (e.g. a vulnerability) when it does not

False negative Reporting a program does not have some property when it does

Soundness and completeness

Capabilities of static analyses are often described in terms of **soundness** and **completeness**.[1]

An analyser can be sound in relation to some runtime property of a program.

sound If the analyser asserts that a program has property P, then the program *does* have property P. The analyser is never wrong about this.

[1] The terms comes from logic. "sound" $\approx$ "says true things", "complete" $\approx$ "all true things are said")

Soundness

sound If the analyser asserts that a program has property P, then the program *does* have property P. The analyser is never wrong about this.

But:

- It might be "over-scrupulous" – sometimes say that programs which *do* have property P, actually do not.
- An analyser could be trivially sound, by simply *never* saying a program has property P – it would never be wrong...

Soundness example

- Suppose a static analyser, *DivDetector*, analyses programs and tries to assess whether they have property *DivGood*: "At runtime, this program will never encounter a division-by-zero exception".
- When fed a program, DivDetector may say "Yes, it has property DivGood" or "No, it does not".
  (Some analysers might also say "I don't know", or remain silent.)

✓ If DivDetector never lies when it says "Yes, this program has property DivGood", then it is *sound*.

✘ But it might say "No, this program does *not* have property DivGood", and be wrong.

Soundness example

If we frame the problem as "identify programs with property DivGood", then DivDetector can suffer from false negatives – it sometime might say a program *doesn't* have property DivGood, when it does.

Completeness

complete If a program has property P, then the analyser always detects it.
(An analyser could be trivially complete, by saying *everything*. e.g. It says "This program is correct", *AND* "This program is incorrect".)

But:

- It may be "over-eager" – might sometimes say that programs which *don't* have property P, actually do.
- It could be trivially complete, by saying *every* program has property P – it would never "miss" a program.

Compromises

What we would *like* is for an analysis to be both sound and complete – it says exactly all the true things.

But it is impossible for any algorithm to determine (for all programs) any non-trivial semantic property of a program – this is Rice's theorem.

So instead, analysers will have to compromise on one or the other (or both)

- Sometimes they say a program has some property P, but it doesn't
- Sometimes a program will *have* property P, but the analyser won't report it

Compromises

They could also be sound or complete "up to certain assumptions".

e.g. A static analyser might be sound, when detecting some property of Java programs, *as long the program doesn't use reflection*.
(If it does, the analyser might give wrong answers, and "let in" some programs which don't actually have the property.)

Phrasing

What sort of error you think an analyser is making depends on how you phrase the question.

If your analyser is looking for property P ("...never divides by zero"), but sometimes classifies programs as dividing by zero when they don't, then it's incomplete.

But if your analyser is looking for property $\neg P$ ("...will divide by zero at least once"), but sometimes classifies programs as dividing by zero when they don't, then it's unsound.

Usually when we say something is "sound", we tend to be talking about desirable properties.

So it'd be more typical to say your analyser is sound, but incomplete.

Example – compilers

For statically typed languages, the compiler guarantees that type errors won't occur at runtime.

But it does so by forbidding many programs in which type errors never would have occurred at runtime.

```
// java method
void do_a_thing() {
  int n = 1;
  if (1 + 1 == 3) {
    n == "hello";
  }
}
```

Example – compilers

```
// java method
void do_a_thing() {
  int n = 1;
  if (1 + 1 == 3) {
    n == "hello";
  }
}
```

Compiler writers would probably say the question being asked is, "Is the program well-typed (i.e., no type errors occur at runtime)?"

The compiler is *sound* (if it says a program is well-typed, it is), but incomplete (sometimes a program would actually be well-typed, but the compiler says it is not).

Example – compilers

But it's just as valid to flip the question, and ask "Will the program exhibit a type error at runtime?".

Then the compiler is *complete* – when type errors occur, they are always reported – but it is not *sound* (sometimes the compiler says a type error would occur, when it actually won't).

Reporting (e.g. security) errors

Does a program have property *GoodProperty* (e.g. *BadThing* never happens)?

	complete	incomplete
sound	Reports all occurrences (If *GoodProperty* is there, analyser says it exists) No false alarms (If analyser says *GoodProperty* exists, it does exist) **Undecidable**	May miss occurrences (If *GoodProperty* is there, analyser may or may not say so) No false alarms (If analyser says *GoodProperty* exists, it does exist) **Decidable**
unsound	Reports all occurrences (If *GoodProperty* is there, analyser says it exists) Possible false alarms (Analyser reports property, but it may not exist) **Decidable**	May miss occurrences (If *GoodProperty* is there, analyser may or may not say so) Possible false alarms (Analyser reports property, but it may not exist) **Decidable**

Decidability

"Undecidable" means no *algorithm* (guaranteed to terminate) exists.

But if we wanted, instead of compromising on soundness or completeness, we could compromise on terminability. (Sometimes, the algorithm might run forever).

(Example: some type checker algorithms *are*, in fact, not guaranteed to terminate; but it turns out that for all "normal" programs written by and of interest to humans, they do end up terminating.)

Static analysis in practice

Since perfect solutions aren't possible

- analysers give *approximate* results (compromise on soundness or completeness), or
- require manual assistance, or
- have timeouts (analysis could in theory run forever)

Static analysis in practice

In general, programmers dislike false positives:

- analyser reports many "problems", most of which are false alarms

We will see an example of this in the lab with Flawfinder and clang-tidy.

Many of the signed-to-unsigned conversions reported are probably harmless, but all the reports obscure bigger problems.

Sorts of static analysis

Static analysers for many different tasks:

- Type checking
- Style checking
- Property checking (ensuring some property holds – e.g. no deadlocks, bad behaviour of some sort)
- Program verification (ensuring correct behaviour w.r.t some specification)
- Bug finding (detecting likely errors)

Sorts of static analysis

- Type checking
- Style checking
- Property checking (ensuring some property holds – e.g. no deadlocks, race conditions, bad behaviour of some sort)
- Program verification (ensuring correct behaviour w.r.t some specification)
- Bug finding (detecting likely errors)

All are useful.

- Type checking prevents runtime errors
- Style checking makes code easier for humans to review
- Property checking can avoid some security bugs
- Program verification (doesn't ensure our *design* or use of e.g. crypto functions is correct though – just correctness w.r.t spec)
- Bug finding (use for secure coding is obvious)

Type systems

Powerful type systems can provide very strong guarantees about program behaviour.

- Memory-safe languages (Java, ML, Haskell, Rust): memory corruption (if the language is properly implemented, and programmer restricted to safe parts of language) impossible

But

- All these languages do, in practice, provide "escape hatches" (if nothing else, they all allow C routines to be called which aren't themselves type-checked)
- Programmers may regard type system as annoying and overly restrictive

Type systems

Statically checked type systems are *modular*

- small pieces can be checked
- the pieces are put together, and the interfaces are checked

One "holy grail" of type system + security research: are there type systems which would provide compositional guarantees for security properties?

Style checking

Type systems are *part* of the language, but style checking covers *good practice*.

Usually covers

- coding standards (layout, bracketing)
- naming conventions (e.g. `snake_case`, `camelCase`, `SCREAMING_SNAKE_CASE`)
- checking for dubious code constructs (e.g. in Python, use of `eval()`)

Example tools:

- `clang-format`, `clang-tidy` (C and C++)
- `pylint`, `black` (Python)
- `checkstyle`, `findbugs`, `PMD` (Java)
- `ShellCheck` (Bash)

Style checking

Most style checkers allow you to customize what rules are applied, and to exclude particular files, functions or lines from being checked

e.g. for pylint:

```
pylint: disable=some-message
```

Language-based security

Language-based security

The idea here is to use language features to check for application-level attacks

One example: taint tracking.

Can be static or dynamic

We add security labels to data inputs (sources) and data outputs (sinks).

tainted Data from unsafe sources (e.g. user input)

Or data derived from or influenced by tainted data

untainted Data we can safely output or use

Taint tracking

To switch something from tainted to untainted, it has to go through particular *sanitization* functions.

Examples:

- Perl provides a "taint mode" (https://perldoc.perl.org/perlsec)
- Taintdroid – modified Android runtime, tracks data flows at runtime (https://github.com/TaintDroid)

Type-checking information flow

Idea: define a type system which tracks security levels of variables in the program, and adding levels to sources and sinks. Security levels may be:

High

- Sensitive information, e.g., personal details
- Any other data that
  - is computed directly from high data
  - occurs in a high context (high test in if)

Low

- Public information, e.g., obtained from user input

Fuzzing

Dynamic analysis – fuzzing

- Many test cases are input into the target application
- Application is monitored for errors (e.g. crashes, hangs)

Types of fuzzing

- Mutation based (“dumb”)
- Generation based (“smart”)
- Evolutionary

Mutation Based Fuzzing

- Little or no knowledge of the structure of inputs
  - Not good for inputs with checksums, challenge responses, etc.
- New inputs may be completely random or follow some
- Very easy to set up
- Depends on inputs provided

Generation Based Fuzzing

- Test cases generated from some description, rules or grammar
- Knowledge of input structure usually gives better results than mutation
- Usually trickier to set up

Evolutionary Fuzzing

- Attempts to generate inputs based on the response of the program