CITS3007 Secure Coding
Analysis and testing

Unit coordinator: Arran Stewart

Highlights

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

Vulnerabilities in design

Problems with *design* are usually found by human review of the design – we'll look at this more when we look at secure development methodologies.

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

flawfinder, used in Lab 5

- Runs the gamut from very very simple techniques (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.

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*.
  - 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)

Static analysis concepts

Dynamic versus static analysis

Static analysis cannot be as precise as dynamic.

```
if halts(f):
  expose_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**.

An analyser can be sound or correct in relation to some property (e.g. program correctness)

sound	If the analyser says that X is true, then X is true (An analyser could be trivially sound, by simply never saying anything)
complete	If X is true, then the analyser always says that X is true (An analyser could be trivially complete, by saying *everything*. e.g. It says "This program is correct", *AND* "This program is incorrect".)

(Comes from logic. "sound" = "says true things", "complete" = "all true things are said")

Soundness and completeness

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 X is the case, but it isn't
- Sometimes X will be the case, but the analyser won't report it

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";
  }
}
```

The compiler's analysis is *complete* in relation to reporting type errors (if a type error would occur, then the compiler does indeed report it), but not *sound* (sometimes the compiler says a type error would occur, when it actually won't).

(Though compiler writers like to phrase it as: Is the program free of type errors? The compiler is *sound* if, whenever it reports a program is free of errors, it is indeed free of runtime type errors. But it isn't

Reporting (e.g. security) errors

Does an error exist?

	complete	incomplete
sound	Reports all errors (If an error exists, analyser says it exists) No false alarms (If analyser says error exists, it does exist) **Undecidable**	May not report all errors (If error exists, analyser may or may not say so) No false alarms (If analyser says error exists, it does exist) **Decidable**
unsound	Reports all errors (If an error exists, analyser says it exists) Possible false alarms (If analyser says error exists, it may not exist) **Decidable**	May not report all errors (If error exists, analyser may or may not say so) Possible false alarms (If analyser says error exists, it may not exist) **Decidable**

Decidability

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

But if we wanted, we could compromise on that, instead (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 saw this in the lab – many of the signed-to-unsigned conversions 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