Recap

We have looked at:
- examples of vulnerabilities and exploits
- particular programming failure patterns
- security engineering

Now it’s time to look at some:
- principles and tools
  for ensuring software security.

Code review and architectural analysis

Remember the secure software development process “touchpoints”, in priority order:
1. Code review and repair
2. Architectural risk analysis
3. Penetration testing
4. Risk-based security testing
5. Abuse cases
6. Security requirements
7. Security operations

This lecture examines static analysis as a set of techniques to help with code review and repair.
Some advanced static analysis techniques may help with architectural (design) understanding too.

Vulnerabilities in design

Design flaws are best found through architectural analysis. They may be generic or context-specific.

**Generic flaws**
- Bad behaviour that any system may have
- e.g., revealing sensitive information

**Context-specific flaws**
- Particular to security requirements of system
- e.g., key length too short for long term

Vulnerabilities in code

Programming bugs (and sometimes more serious flaws) are best found through static code analysis.

**Generic defects**
- Independent of what the code does
- May occur in any program
- May be language specific
- e.g., buffer overflow in C or C++

**Context-specific defects**
- Depend on particular meaning of the code
- Even when requirements may be general
- Language agnostic. AKA logic errors.
- e.g., PCI-CSS rules for CC number display violated

Testing is also useful, of course, but has failed spectacularly in many cases, including some very embarrassing cases recently.

<table>
<thead>
<tr>
<th>Seen in Code</th>
<th>Only Seen in Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static analysis sweet spot, built-in rules make it easy for tools to find these without programmer guidance.</td>
<td>Most likely to be found through architectural analysis.</td>
</tr>
<tr>
<td>Example: buffer overflow.</td>
<td>Example: the program executes code downloaded as an email attachment.</td>
</tr>
</tbody>
</table>

Possible to find with static analysis, but customization may be required.
- Example: mishandling of credit card information.

Requires both understanding of general security principles along with domain-specific expertise.
- Example: cryptographic keys kept in use for an unsafe duration.

Matrix from Secure Programming with Static Analysis, Chess and West, 2007
Common Weakness Enumeration

Recall (from Lecture 7):

- **Weaknesses** classify **Vulnerabilities**
- A **CWE** is an identifier such as CWE-287
- CWEs are organised into a hierarchy
- The hierarchy (perhaps confusingly) allows:
  - multiple appearances of same CWE
  - different types of links
- This allows multiple views
  - different ways to structure the same things
  - also given CWE numbers

See [https://cwe.mitre.org](https://cwe.mitre.org)

CWE hierarchy

Figure 1: Section of CWE Hierarchy

7 Pernicious Kingdoms

One developer-oriented classification was introduced by Tsimpnyuk, Chess, and McGraw in 2005.

1. Input validation and representation
2. API abuse
3. Security features
4. Time and state
5. Error handling
6. Code quality
7. Encapsulation
8. Environment ("a separate realm")

This appears as the view CWE 700.

**Exercise.** Browse the CWE hierarchy to understand representative weaknesses in each category.

CWE 700 at Mitre

Visualisation of CWEs at cvevis.org

Static analysis

A **white box** technique. Takes as input

- source code, usually
- binary code, sometimes (Q. Why?)

As output, provide a report listing either

- assurance of **good behaviour** ("no bugs!") or
- evidence of **bad behaviour**, ideally proposed fixes

40 years of research, growing range of techniques and tools. Some standalone, some inside compilers, IDEs.

Complexity ranges from simple scanners (linear in code size) to much more expensive, deep code analysis, exploring possible states in program execution.
Static analysis for security

In principle a perfect fit for security because:
▶ it examines every code path
▶ it considers every possible input

And only a single path/input is needed for a security breach. Dynamic testing only reaches paths determined by test cases and only uses input data given in test suites.

Other advantages:
▶ often finds root cause of a problem
▶ can run before code complete, even as-you-type

But also some disadvantages/challenges...

Solving an impossible task

Perfect static security analysis is of course impossible.

```
if halts(f) then
call expose_all_mysecrets
```

Rice’s Theorem (informal)
For any non-trivial property of partial functions, there is no general and effective method to decide whether an algorithm computes a partial function with that property.

Static analysis in practice

▶ Correctness undecidable in general
  ▶ focus on decidable (approximate) solution
  ▶ or semi-decidable + manual assistance/timeouts
▶ State-space explosion
  ▶ must design/derive abstractions
  ▶ data: restricted domains (abstract interpretation)
  ▶ code: approximate calling contexts
▶ Environment is unknown
  ▶ program takes input from outside
  ▶ other factors, e.g., scheduling of multiple threads
  ▶ again, use abstractions
▶ Complex behaviours difficult to specify
  ▶ use generic specifications

Space of programs

Results of a static analysis tool

False positives (false alarms)

Because the security or correctness question must be approximated, tools cannot be perfectly precise. They may raise false alarms, or may miss genuine vulnerabilities.

The false positive problem is hated by users:
▶ too many potential problems raised by tool
▶ programmers have to wade through long lists to weed out
▶ true defects may be lost, buried in details

So tools compete on false positive rate for usability.
False negatives (missing defects)

In practice, tools trade-off false positives with missing defects.

Risky for security:
- one missed bug enough for an attacker to get in!

Academic research concentrates on sound techniques (if a problem exists, the algorithm will identify it), which have no false negatives.

But strong assumptions are needed for soundness. In practice, tools must accept missing defects.

How are imprecise tools measured and compared? Difficult. The US NIST SAMATE project is working on static analysis benchmarks.

False positives in type checking

```
short s = 0;
int i = s;
short r = i;
```

```
[dice]da: javac ShortLong.java
ShortLong.java:5: error: possible loss of precision
    short r = i;
        ^
required: short
found: int
1 error
```

False positives in type checking

```
int i;
if (3 > 4) {
    i = i + "hello";
}
```

```
int i;
if (3 > 4) {
    i = i + "hello";
}
```

False positives in type checking

```
int i;
if (3 > 4) {
    i = i + "hello";
}
```

Static analysis jobs

There is a wide range of jobs performed by static analysis tools and techniques:

- **Type checking**: part of language
- **Style checking**: ensuring good practice
- **Program understanding**: inferring meaning
- **Property checking**: ensuring no bad behaviour
- **Program verification**: ensuring correct behaviour
- **Bug finding**: detecting likely errors

General tools in each category may be useful for security. Dedicated **static security analysis tools** also exist. Examples are HP Fortify and Coverity.

Type systems: a discipline for programming

- Proper type systems provide **strong guarantees**
  - Java, ML, Haskell: memory corruption impossible
  - These are **strongly typed** languages
- Sometimes frustrating: seen as a hurdle
  - old joke: *when your Haskell program finally type-checks, it must be right!*
- Do programmers accept type systems?
  - yes: type errors are necessary, not “false”
  - no: they’re overly restrictive, complicated
  - …likely influence on rise of scripting languages

**Type systems: a discipline for programming**

- Proper type systems provide **strong guarantees**
  - Java, ML, Haskell: memory corruption impossible
  - These are **strongly typed** languages
- Sometimes frustrating: seen as a hurdle
  - old joke: *when your Haskell program finally type-checks, it must be right!*
- Do programmers accept type systems?
  - yes: type errors are necessary, not “false”
  - no: they’re overly restrictive, complicated
  - …likely influence on rise of scripting languages
Type systems: intrinsic part of the language
In a statically type language, programs that can’t be
type-checked don’t even have a meaning.

- Compiler will not produce code
- So code for ill-typed programs cannot be executed
- Programming language specifications (formal
  semantics or plain English): may give no meaning,
  or a special meaning.

Robin Milner captured the intuition “Well-typed programs can’t go
wrong” as a theorem about denotational semantics. Adding a
number to a string gives a special denotational value “wrong.”

Type systems: flexible part of the language
In practice, programmers and IDEs do give meaning
(sometimes even execute) partially typed programs.

Recent research: *gradual typing* (and related work) to
make this more precise:

- start with untyped scripting language
- infer types in parts of code where possible
- manually add type annotations elsewhere
- ... so compiler recovers safety in some form
- A good example is TypeScript

Sometimes even strongly-typed languages have escape routes, e.g.,
via C-library calls or abominations like unsafePerformIO.

Type systems: motivating new languages
High-level languages arrived with strong type systems
early on (inspired from mathematical ideas in
functional languages, e.g., Standard ML, Haskell).
Language designers asked if static typing can be
provided for systems programming languages, without
impacting performance *too much*.
Two prominent young examples:

- Go (2007-)
- Rust (2009-)
both are conceived as type safe low-level languages
with built-in concurrency support.

**Question.** Why add concurrency support? Are there
benefits for secure programming?

Type systems: massive advantage
By design, provide modularity
- write programs in separate pieces
- type check the pieces
- put the types together: the whole is type-checked

This property extends to the basic parts of the language:
we find the type of an expression from the type of its
parts.
Programming language researchers call this
compositionality.

**Research question:** can we find type systems that
provide compositional guarantees for security?
Style checking for good practice

Informally, comparing with natural language (intuition)
- type system: becomes part of syntax of language
- style checking: a bit like grammar checking in NL

Style checking traditionally covers good practice
- syntactic coding standards (layout, bracketing etc)
- naming conventions (e.g., UPPERCASE constants)
- lint-like checking for dubious/non-portable code
  - modern languages are stricter than old C
  - (or have fewer implementations)
  - style checking becoming part of compiler/IDE
  - but also dedicated tools with 1,000s rules

Example tools: PMD, Parasoft.

Legal in language, type checks and compiles fine:
```
[dice] da: gcc enum.c
```
But with warnings:
```
[dice] da: gcc -Wall enum.c
```
```
enum.c: In function 'showWarning':
enum.c:7:3: warning: enumeration value 'GREEN' not handled in switch [-Wswitch]
  switch (c) {
    ^
```

Question. Why have some languages decided that omitted cases should not be allowed?

CodePro Analytix

A nice Java program analysis tool acquired by Google and made freely available for a while:

Un fortunately it now seems to be no longer available: Google hoped but had no time to make it open-source. Their current developer tools include a range of app testing mechanisms.

We'll look at some other tools next time.

Review Questions

Static versus dynamic analysis
- Static analysis requires access to source (sometimes binary) code. What advantages does that enable?
- Why do practical static analysis tools both miss problems and report false problems?

Types of static analysis tool
- Apart from type and style checking, describe three other jobs a static analysis tool may perform.

References and credits

Some of this lecture (and the next) is based Chapters 1-4 of
- Secure Programming With Static Analysis by Brian Chess and Jacob West, Addison-Wesley 2007.

Recommended reading: