Program analysis for security
Two main classes

• Static:
  • Operates on source or binary at rest

• Dynamic:
  • Operates at runtime

• Also hybrids of the two
Static: Examples

• Code review
• Grep
• Taint analysis
• Symbolic execution
• Templates/specifications (metacompilation)
Dynamic: Examples

• Testing
• Debugging
• Log-tracing
• Fuzzing
Static: Pros and Cons

• Analyze everything in the program
  • Not just what runs during this execution

• Don’t need running environment (e.g. comms)
  • Can analyze incomplete programs (libraries)
    • If you have the source code

• Everything could be a lot of stuff!
  • Scalability
    • Code that never runs in practice (or dead)

• No side effects

• Only find what you are looking for
Dynamic: Pros and Cons

- Concrete failure proves an issue
  - May aid fix

- Computationally scalable

- Coverage?

- Resources/environment?
Static Analysis

Some material from Dave Levin, Mike Hicks, Dawson Engler, Lujo Bauer

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html
From here we mostly mean automated: in a sense, ask a computer to do your code review
High-level idea

• Model program properties abstractly
• Set some rules/constraints and then check them
• Tools from program analysis:
  • Type inference
  • Theorem proving
  • etc.
• What kinds of properties are checkable this way?
• What guarantees can we have? (FP/FN)
• Resources/scalability?
The Halting Problem

• Can we write an analyzer that can prove, for any program $P$ and inputs to it, $P$ will terminate?
  • Doing so is called the **halting problem**
  • Unfortunately, this is **undecidable**: any analyzer will fail to produce an answer for at least some programs and/or inputs

Some material inspired by work of Matt Might: [http://matt.might.net/articles/intro-static-analysis/](http://matt.might.net/articles/intro-static-analysis/)
Check other properties instead?

- Perhaps security-related properties are feasible
  - E.g., that all accesses $a[i]$ are in bounds

- But these properties can be converted into the halting problem by transforming the program
  - A perfect array bounds checker could solve the halting problem, which is impossible!

- Other undecidable properties (Rice’s theorem)
  - Does this string come from a tainted source?
  - Is this pointer used after its memory is freed?
  - Do any variables experience data races?
So is static analysis impossible?

- **Perfect** static analysis is **not possible**

- **Useful** static analysis is **perfectly possible**, despite
  1. **Nontermination** - analyzer never terminates, or
  2. **False alarms** - claimed errors are not really errors, or
  3. **Missed errors** - no error reports ≠ error free

- Nonterminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors
**Soundness**
If analysis says that X is safe, then X is safe.

**Completeness**
If X is safe, then analysis says X is safe.

- **Trivially Sound**: Say nothing is safe
- **Trivially Complete**: Say everything is safe

**Sound and Complete**: *Say exactly the set of true things*
• **Soundness**: No error found = no error exists
  • Alarms may be false errors

• **Completeness**: Any error found = real error
  • Silence does not guarantee no errors

• Basically any useful analysis
  • is neither **sound** nor **complete** (def. not **both**)  
  • … usually *leans* one way or the other
The Art of Static Analysis

- **Precision**: Carefully model program, minimize false positives/negatives
- **Scalability**: Successfully analyze large programs
- **Understandability**: Actionable reports
• Observation: **Code style is important**
  
  • Aim to be precise for “good” programs
  
  • OK to forbid yucky code in the name of safety
  
  • Code that is more understandable to the analysis is more understandable to humans
Adding some depth: Dataflow (taint) analysis
Tainted Flow Analysis

- Cause of many attacks is **trusting unvalidated input**
  - Input from the user (network, file) is **tainted**
  - Various data is used, assuming it is **untainted**

- Examples expecting untainted data
  - source string of `strcpy (≤ target buffer size)`
  - format string of `printf` (contains no format specifiers)
  - form field used in constructed SQL query (contains no SQL commands)
Recall: Format String Attack

• Adversary-controlled format string

```c
char *name = fgets(..., network_fd);
printf(name);  // Oops
```
The problem, in types

• Specify our requirement as a type qualifier

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);
```

• **tainted** = possibly controlled by attacker

• **untainted** = **must not be** controlled by attacker

```c
tainted char *name = fgets(..., network_fd);
printf(name);    // **FAIL**: untainted <- tainted
```
Analyzing taint flows

• **Goal**: For all possible inputs, prove tainted data will never be used where untainted data is expected
  - **untainted** annotation: indicates a trusted sink
  - **tainted** annotation: an untrusted source
  - no annotation means: not specified (analysis must figure it out)

• Solution requires inferring **flows** in the program
  - What **sources can reach what sinks**
  - If any flows are *illegal*, i.e., whether a **tainted** source may flow to an **untainted** sink

• We will aim to develop a (mostly) **sound** analysis
Legal Flow

```c
void f(tainted int);
untainted int a = ...;
f(a);
```

*f* accepts *tainted* or *untainted* data

Illegal Flow

```c
void g(untainted int);
tainted int b = ...;
g(b);
```

*g* accepts *only untainted* data

Define allowed flow as a constraint:

untainted < tainted

At each program step, **test** whether inputs ≤ policy

(Read as: input less tainted (or equal) than policy)
Analysis Approach

• If no qualifier is present, we must infer it

• Steps:
  • Create a name for each missing qualifier (e.g., $\alpha$, $\beta$)
  • For each program statement, generate constraints
    • Statement $x = y$ generates constraint $q_y \leq q_x$
  • Solve the constraints to produce solutions for $\alpha$, $\beta$, etc.
    • A solution is a substitution of qualifiers (like tainted or untainted) for names (like $\alpha$ and $\beta$) such that all of the constraints are legal flows

• If there is no solution, we (may) have an illegal flow
Example Analysis

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);

char *name = fgets(..., network_fd);
char *x = name;
printf(x);
```

First constraint requires $\alpha = \text{tainted}$
To satisfy the second constraint implies $\beta = \text{tainted}$
But then the third constraint is illegal: $\text{tainted} \leq \text{untainted}$

No possible solution for $\alpha$ and $\beta$
Taint Analysis: Adding Sensitivity
But what about?

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);

→
char *name = fgets(..., network_fd);
t char *x;
x = name;
x = "hello!";
printf(x);
```

```
tainted ≤ α
α ≤ β
untainted ≤ β
β ≤ untainted
```

No constraint solution. Bug? **False Alarm!**
Flow Sensitivity

• Our analysis is **flow insensitive**
  - Each variable has **one qualifier**
  - Conflates the taintedness of all values it ever contains

• **Flow-sensitive analysis** accounts for variables whose contents change
  - Allow each assigned use of a variable to have a different qualifier
    - E.g., $\alpha_1$ is x’s qualifier at line 1, but $\alpha_2$ is the qualifier at line 2, where $\alpha_1$ and $\alpha_2$ can differ
  - Could implement this by transforming the program to assign to a variable at most once
Reworked Example

int printf(untainted char *fmt, ...);
tainted char *fgets(...);

→ char *name = fgets(..., network_fd);
  char β *x1, γ *x2;
  x1 = name;
  x2 = "%s";
  printf(x2);

tainted ≤ α
α ≤ β
untainted ≤ γ
γ ≤ untainted

No Alarm

Good solution exists:
γ = untainted
α = β = tainted
Handling conditionals

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);

char *name = fgets(..., network_fd);
char *x;
if (...)  x = name;
else    x = "hello!";
printf(x);
```

Constraints still unsolvable

Illegal flow
Multiple Conditionals

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);

void f(int x) {
    char *y;
    if (x) y = "hello!";
    else y = fgets(..., network_fd);
    if (x) printf(y);
}
```

\[ \alpha \leq \text{untainted} \leq \alpha \]

- \text{untainted} \leq \alpha
- \text{tainted} \leq \alpha
- \alpha \leq \text{untainted}

No solution for \( \alpha \). Bug? \textbf{False Alarm!}

(and flow sensitivity won’t help)
Path Sensitivity

• Consider path feasibility. E.g., $f(x)$ can execute path
  • 1-2-4-5-6 when $x \neq 0$, or
  • 1-3-4-6 when $x == 0$. But,
  • path 1-3-4-5-6 infeasible

A path sensitive analysis checks feasibility, e.g., by qualifying each constraint with a path condition

\[
\begin{align*}
&x \neq 0 \Rightarrow \text{untainted} \leq \alpha \quad \text{(segment 1-2)} \\
&x = 0 \Rightarrow \text{tainted} \leq \alpha \quad \text{(segment 1-3)} \\
&x \neq 0 \Rightarrow \alpha \leq \text{untainted} \quad \text{(segment 4-5)}
\end{align*}
\]
Why *not* use flow/path sensitivity?

- Flow sensitivity **adds precision**, path sensitivity adds more
  - Reduce false positives: less developer effort!

- But both of these **make solving more difficult**
  - Flow sensitivity *increases the number of nodes* in the constraint graph
  - Path sensitivity *requires more general solving procedures* to handle path conditions

- In short: **precision** (often) **trades off scalability**
  - Ultimately, limits the size of programs we can analyze
Implicit flows

```c
void copy(tainted char *src,
          untainted char *dst,
          int len) {
    untainted int i;
    for (i = 0; i<len; i++) {
        dst[i] = src[i]; //illegal
    }
}
```

**Illegal flow:**

`tainted` ≠ `untainted`
void copy(tainted char *src, untainted char *dst, int len) {
    untainted int i, j;
    for (i = 0; i<len; i++) {
        for (j = 0; j<sizeof(char)*256; j++) {
            if (src[i] == (char)j)
                dst[i] = (char)j; //legal?
        }
    }
}

Missed flow!
Implicit flow analysis

- **Implicit flow**: one value *implicitly* influences another

- One way to find these: maintain a scoped **program counter (pc)** label
  - Represents the maximum taint affecting the current pc

- Assignments generate constraints involving the *pc*
  - $x = y$ produces two constraints:
    - $\text{label}(y) \leq \text{label}(x)$ (as usual)
    - $\text{pc} \leq \text{label}(x)$
Implicit flow example

tainted int src;
α int dst;
if (src == 0)
dst = 0;
else
dst = 1;
dst += 0;

Taint on $\alpha$ is identified.
Discover implicit flow!
Why not implicit flow?

• Tracking implicit flows can lead to **false alarms**
  • E.g., ignores values
    ```java
tainted int src;
α int dst;
if (src > 0) dst = 0;
else           dst = 0;
```

• Extra constraints **hurt performance**

• The evil copying example is *pathological*
  • We typically don’t write programs like this*
  • Implicit flows will have little overall influence

• **So:** taint analyses tend to ignore implicit flows

* Exception coming in two slides
Other challenges

• Taint through operations
  • tainted a; untainted b; c=a+b — is c tainted? (yes, probably)

• Function calls and context sensitivity
  • Function pointers: Flow analysis to compute possible targets

• Struct fields
  • Track taint for the whole struct, or each field?
  • Taint per instance, or shared among all of them (or something in between)?
    • Note: objects ≈ structs + function pointers

• Arrays: Track taint per element or across whole array?

No single correct answer!
(Tradeoffs: Soundness, completeness, performance)
Other refinements

• Label *additional* sources and sinks
  • e.g., Array accesses must have untainted index

• Handle *sanitizer functions*
  • Convert tainted data to untainted

• Complementary goal: Leaking confidential data
  • Don’t want *secret sources* to go to *public sinks*
    • Implicit flows more relevant (malicious code)
  • *Dual* of tainting
Static analysis in practice

• Thoroughly check limited but useful properties
  • **Eliminate** some categories of errors
  • Developers can concentrate on **deeper reasoning**

• Encourage **better development practices**
  • Programming models that **avoid mistakes**
  • Teach programmers to **manifest their assumptions**
    • Using **annotations** that improve tool precision

• Seeing **increased commercial adoption**
Fuzzing

Some material from Tal Garfinkel, Dmitry Vyukov

https://reviewsfromtheabyss.files.wordpress.com/2012/07/2007_hot_fuzz_002.jpg
Testing vs. Fuzzing

• Testing: Test many (mostly) normal inputs
  • Goal: Keep user from encountering bugs

• Fuzzing: Test abnormal inputs
  • Goal: Look for exploitable weakness
High-level idea

- Generate many weird inputs
  - Files (.pdf, .wav, .html, etc)
  - Network packets
  - Other?
- Monitor application for errors
  - Crashes \( \equiv \) vulnerabilities?
How to generate inputs?

- Random/brute force (hmm....)
- Mutation: Tweak valid inputs
- Grammar-based
- Using symbolic execution / static analysis (whitebox)
- **Coverage-guided** (greybox)
Coverage-guided fuzzing

• While (true):
  • Select input from corpus
  • Mutate input
  • Run target program, collect code coverage
  • If got new coverage, add input back to corpus
Types of mutations

- Add/remove/swap bytes from one input
- Splice two inputs
- Insert token from dictionary or magic number
- Change semantic token ("123" -> "456", "cat" -> "dog")
- etc.
Detecting a “problem”

• Did it crash?

• Did it freeze?

• Did it give the correct output?
  • Round trip: encode/decode, etc.
  • Compare to reference implementation
How much fuzz is enough?

- Random mutations can take a while to hit
- Even w/ coverage metrics!
  - Can cover it without hitting the bug
  - Lots of code you never reach