Static Analysis

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

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html
Previously

- How software bugs can lead to exploits
- Approaches for defense (principled, avoidance)
- Malware
Today

- Finish up malware
- Static analysis
  - Overview
  - Example: Taint/flow analysis
  - Example: Metaccompilation
“Modern” Malware
Stuxnet: Propagation

June 2010

• **Virus:** initially spread by infected USB stick
  • Once inside network, acted as a **worm**, spreading quickly

• Exploited **four** zero-day exploits
  • Zero-day: Known to only the attacker until the attack
  • Typically, one zero-day is enough to profit
  • Four was unprecedented
    - Immense cost and sophistication on behalf of the attacker

• Rootkit: Installed *signed* device drivers
  • Thereby avoiding user alert when installing
  • Signed with *certificates stolen* from two Taiwanese CAs
Stuxnet: Payload

- Do nothing

- Unless attached to particular models of frequency converter drives that operate at 807-1210Hz
  - You know, like those in Iran and Finland
  - ... those ones that are used to operate centrifuges
  - ... for producing enriched uranium for nuclear weapons

- In which case, slowly increase the freq to 1410Hz
  - You know, enough to break the centrifuge
  - ... all the while sending “looks good to me” readings to the user
  - ... then drop back to normal range
Stuxnet: Payload

- Target industrial control systems: overwrite programmable logic boards

- Man-in-the-middle between Windows and Siemens control systems; looked like it was working properly to the operator

- In reality, it sped up and slowed down the motors

- Result: Destroy (or at least decrease the productivity of) nuclear centrifuges
Stuxnet: Fallout

- Iran denied they had been hit by Stuxnet
- Then claimed they were, but had contained it
- Now believed it took out 1k of Iran’s 5k centrifuges
- Security experts believe the U.S. did it (possibly along with Israel) due to its sophistication and cost

- Legitimized cyber warfare
Detecting modern malware

- Connection to known C&C server
  - Counter: Cycle domain and use dynamic DNS
  - Re-counter: Block connections to new domains
- “Custom” TCP and UDP
- Generating direct email (vs. traversing mail server)
- Anomaly detection

Detection, not prevention

All subject to arms race!
Malware summary

- Technological arms race between those who wish to detect and those who wish to evade detection
- Started off (kind of) innocuously
- Became professional, commoditized
  - Economics, cyber warfare, corporate espionage
- Advanced detection: based on behavior, anomalies
  - Must react to attacker responses
Static Analysis

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

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html
Static analysis
Current Practice
for Software Assurance

- **Testing**: Check correctness on set of inputs
- **Benefits**: Concrete failure proves issue, aids fix
- **Drawbacks**: Expensive, difficult, coverage?
  - No guarantees
Current Practice

- **Code audit**: Convince someone your code is correct
- **Benefit**: Humans can generalize
- **Drawbacks**: Expensive, hard, no guarantees
• How can we do better?
Static analysis

• Analyze program’s code without running it
  • In a sense, ask a computer to do code review

• **Benefit:** (much) *higher coverage*
  - Reason about many possible runs of the program
    - Sometimes *all of them*, providing a **guarantee**
    - Reason about incomplete programs (e.g., libraries)

• **Drawbacks:**
  • Can only analyze limited properties
  • May miss some errors, or have false alarms
  • Can be time- and resource-consuming
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/
Check other properties instead?

• Perhaps security-related properties are feasible
  • E.g., that all accesses \texttt{a[i]} are in bounds

• \textit{But} these \textbf{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 \textbf{SQL string} come from a \textbf{tainted source}?
  • Is this \textbf{pointer used after} its memory is freed?
  • Do any variables experience \textbf{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
**Completeness**

If analysis says that X is true, then X is true.

**Soundness**

If X is true, then analysis says X is true.

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

**Sound and Complete**:

*Say exactly the set of true things*
Stepping back

- **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

• Design goals:
  • **Precision**: Carefully model program, minimize false positives/negatives
  • **Scalability**: Successfully analyze large programs
  • **Understandability**: Error reports should be actionable

• 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: Taint (flow) 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` ($\leq$ 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
```

• Attacker sets name = “%s%s%s” to crash program
• Attacker sets name = “…%n…” to write to memory
  • Yields code injection exploits

• These bugs still occur in the wild occasionally
  • Too restrictive to forbid non-constant format strings
The problem, in types

- Specify our requirement as a type qualifier

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

- **tainted** = possibly controlled by adversary
- **untainted** = must not be controlled by adversary

```c
tainted char *name = fgets(...,network_fd);
printf(name);  // FAIL: tainted ≠ untainted
```
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 sure (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 *sound* analysis
Legal Flow

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

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

*untainted* ≤ *tainted*

Define allowed flow as a **lattice**:

```
untainted < tainted
```

Illegal Flow

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

g accepts only **untainted** data

*tainted* ⊈ *untainted*

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

• Finish static analysis
• Start web security
• Project 1 due Friday at midnight
  • Submit server is now available
  • Don’t wait until the last minute to get help
Recall: Information flow

- Label variables to ensure dirty (tainted) inputs cannot reach protected (untainted) sinks
Example Analysis

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

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

1. `tainted \leq \alpha`
2. `\alpha \leq \beta`
3. `\beta \leq \text{untainted}`

Illegal flow!

No possible solution for \( \alpha \) and \( \beta \)

First constraint requires \( \alpha = \text{tainted} \)
To satisfy the second constraint implies \( \beta = \text{tainted} \)
But then the third constraint is illegal: \( \text{tainted} \leq \text{untainted} \)
Taint Analysis: Adding Sensitivity
But what about?

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

→ α char *name = fgets(..., network_fd);
   β char *x;
   x = name;
   x = “hello!”;
   printf(x);

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

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

char *name = fgets(…, network_fd);
char   *x1,  *x2;
x1 = name;
x2 = "%s";
printf(x2);
```

\[ \alpha \] char *name = fgets(…, network_fd);
\[ \beta \] char *x1, \[ \gamma \] char *x2;
x1 = name;
x2 = "%s";
printf(x2);

\[ \text{tainted} \leq \alpha \]
\[ \alpha \leq \beta \]
\[ \text{untainted} \leq \gamma \]
\[ \gamma \leq \text{untainted} \]

No Alarm

Good solution exists:
\[ \gamma = \text{untainted} \]
\[ \alpha = \beta = \text{tainted} \]
Handling conditionals

int printf(\textcolor{green}{untainted} char *fmt, ...);
tagged{tainted} char *fgets(...);

\[
\begin{align*}
\alpha & \text{ char *name = fgets(..., network_fd);} \\
\beta & \text{ char *x;}
\text{if (...)} & \text{ x = name;} \\
\text{else} & \text{ x = "hello!";} \\
\text{printf(x);} \\
\end{align*}
\]

\textcolor{green}{untainted} \leq \alpha
\alpha \leq \beta
untainted \leq \beta
\beta \leq \textcolor{green}{untainted}

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);
}
```

No solution for $\alpha$. Bug?

False Alarm!

(and flow sensitivity won’t help)
Path Sensitivity

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

\begin{verbatim}
void f(int x) {
  char *y;
  if (x)
    y = "hello!";
  else
    y = fgets(…);
  if (x)
    printf(y);
}
\end{verbatim}

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

• $x \neq 0 \implies \texttt{untainted} \leq \alpha$ \hspace{1cm} (segment \texttt{1-2})
• $x = 0 \implies \texttt{tainted} \leq \alpha$ \hspace{1cm} (segment \texttt{1-3})
• $x \neq 0 \implies \alpha \leq \texttt{untainted}$ \hspace{1cm} (segment \texttt{4-5})
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)
    - \( pc \leq \text{label}(x) \)
Implicit flow example

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

Taint on $\alpha$ is identified.

Discovers implicit flow!
Why not implicit flow?

- Tracking implicit flows can lead to **false alarms**
  - E.g., ignores values

- 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
  • \textit{tainted} a; \textit{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 \approx structs + function pointers

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

\textbf{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
Meta-level compilation

(briefly)
Meta-level compilation

(Engler et al., 2000)

• Compilers are good at verifying rules
  • But don’t have domain-specific knowledge

• Developers have domain knowledge
  • But manual inspection is painful, erratic

• Metacompilation: Devs give compiler extra rules
1. Write rules
   - Define legal and error states
   - Specify state transitions

2. Meta-compiler checks rules
   - Flow sensitive, context sensitive (more or less)
   - Scales well to large, real programs
Simple example: Interrupts

After interrupts are disabled, they should be re-enabled.

1. Define states

- s_enabled
- s_disabled
- ERROR
- WARNING
Simple example: Interrupts

After interrupts are disabled, they should be re-enabled.

2. Define transitions

```
s_enabled
  cli()           sti()    end of code

s_disabled

cli()           sti()    sti()    cli()

ERROR          WARNING
```

More interesting example: Free

\[ v = \text{malloc}() \]

- **unknown**
  - \[ v \neq 0: \text{TRUE} \]
  - \[ v = 0: \text{FALSE} \]

- **null**
  - \[ v = 0: \text{TRUE} \]
  - \[ v \neq 0: \text{FALSE} \]

- **non-null**
  - \[ v \neq 0: \text{TRUE} \]
  - \[ v = 0: \text{FALSE} \]

- **freed**

- **ERROR**
  - free(v)

- use *v as operand
- use v as operand
Developer writes state machine
Many rules are transferable to other programs

```c
// No dereferences of null or unknown ptrs.
void null, void unknown: { *(any *)v } =>
    { err("Using ptr illegally!"); }

// Allow free of all non-freed variables.
void unknown, void null, void not_null:
    { free(v); } => void freed;

// Check for double free and use after free.
void freed:
    { free(v) } => { err("Dup free!"); }
    | { v } => { err("Use-after-free!"); }

// Overwriting v’s value kills its state
void all: { v = v1 } => void ok;
```
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**
Static analysis in practice

Caveat: appearance in the above list is not an implicit endorsement, and these are only a sample of available offerings