SQL injection
countermeasures
The underlying issue

- This one string combines the code and the data
- Similar to buffer overflows

When the boundary between code and data blurs, we open ourselves up to vulnerabilities

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```
The underlying issue

$result = mysql_query("select * from Users where(name='\$user' and password='\$pass');");

Should be \textit{data}, not \textit{code}
Prevention: Input validation

- We require input of a certain form, but we cannot guarantee it has that form, so we must **validate it**
  - Just like we do to avoid buffer overflows

- Making input trustworthy
  - **Check** it has the expected form, reject it if not
  - **Sanitize** by modifying it or using it such that the result is correctly formed
Sanitization: Blacklisting

- **Delete** the characters you don’t want
- **Downside**: “Lupita Nyong’o”
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?
- **Downside**: How to know you’ve ID’d all bad chars?
Sanitization: Escaping

- **Replace** problematic characters with safe ones
  - Change ‘ to \\
  - Change ; to ;
  - Change – to −
  - Change \ to \ \

- Hard by hand, there are many libs & methods
  - `magic_quotes_gpc = On`
  - `mysql_real_escape_string()`

- **Downside**: Sometimes you want these in your SQL!
  - And escaping still may not be enough
Checking: Whitelisting

- Check that the user input is **known to be safe**
  - E.g., integer within the right range

- Rationale: Given invalid input, **safer to reject than fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - Principle of fail-safe defaults

- **Downside**: Hard for rich input!
  - How to whitelist usernames? First names?
Sanitization via escaping, whitelisting, blacklisting is HARD.

Can we do better?
Sanitization: Prepared statements

- Treat user data according to its *type*
- Decouple the code and the data

```php
$db = new mysql("localhost", "user", "pass", "DB");

$statement = $db->prepare("select * from Users
where(name=? and password=?);");

$statement->bind_param("ss", $user, $pass);
$statement->execute();
```

```
$result = mysql_query("select * from Users
    where(name='$user' and password='$pass');");
```

### Bind variables

Bind variables are typed
Using prepared statements

```php
$statement = $db->prepare("select * from Users
 where(name=? and password=?);"※");
$stmt->bind_param("ss", $user, $pass);
```

Binding is only applied to the leaves, so the structure of the tree is fixed
Takeaways: Verify before trust

- Improperly validated input causes many attacks
- Common to solutions: check or sanitize all data
  - Whitelisting: More secure than blacklisting
  - Checking: More secure than sanitization
    - Proper sanitization is hard
- All data: Are you sure you found all inputs?
- Don’t roll your own: libraries, frameworks, etc.
Static Analysis

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

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/](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
  - That a certain line of code is reachable

- 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 SQL 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
**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
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` (≤ 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*

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

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

  ```
  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
Example Analysis

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

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

**Illegal flow!**
No possible solution for α and β

1. tainted ≤ α
2. α ≤ β
3. β ≤ untainted

First constraint requires α = tainted
To satisfy the second constraint implies β = tainted
But then the third constraint is illegal: tainted ≤ untainted
Taint Analysis: Adding Sensitivity
But what about?

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

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

\[
tainted \leq α \\
α \leq β \\
untainted \leq β \\
β \leq untainted
\]

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 β *x1, γ *x2;
x1 = name;
x2 = "hello!";
printf(x2);

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

\text{No Alarm}

\text{Good solution exists:}
\gamma = \text{untainted}
\alpha = \beta = \text{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);
```

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

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);
}
```

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

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
  • 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

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

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

  • $x \neq 0 \Rightarrow \text{untainted} \leq \alpha$ (segment 1-2)
  • $x = 0 \Rightarrow \text{tainted} \leq \alpha$ (segment 1-3)
  • $x \neq 0 \Rightarrow \alpha \leq \text{untainted}$ (segment 4-5)
Static analysis in practice

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