Resin: runtime-enforced information flow control

•••

Kinan Bab

Plan

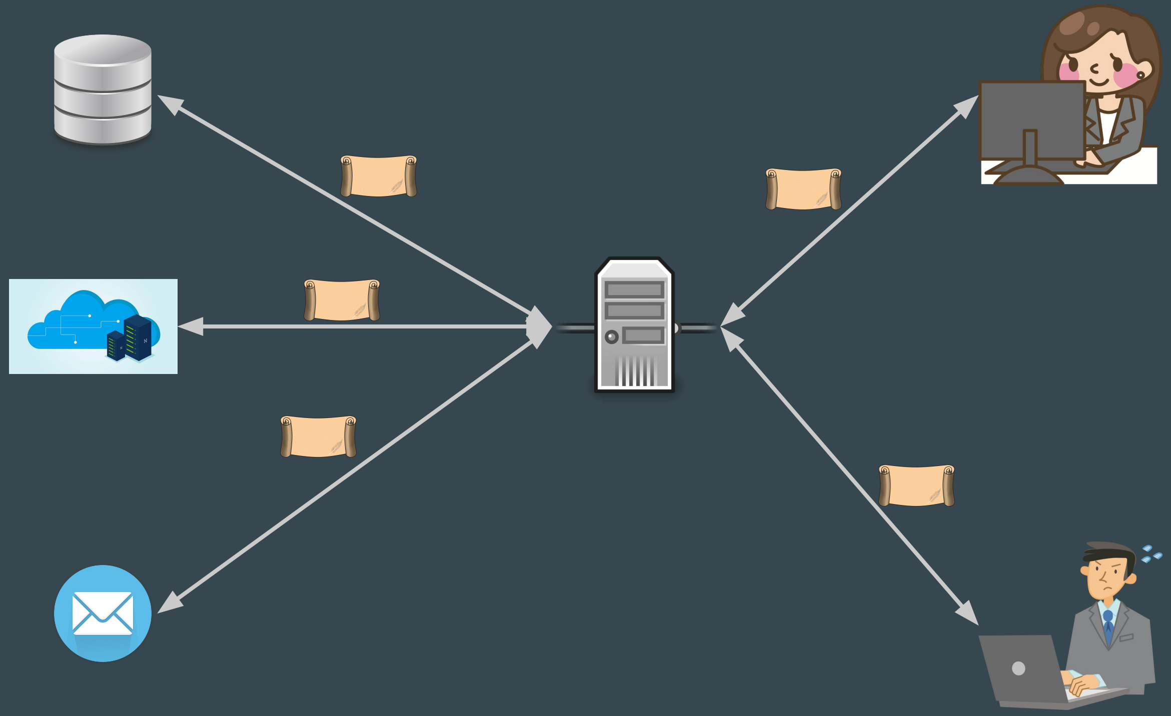
1. What is information flow control? What can it be used for?
2. How does Resin enforce information flow control?
3. Are there other approaches to information flow control?
4. Discussion!

What is [Information Flow] [Control]?

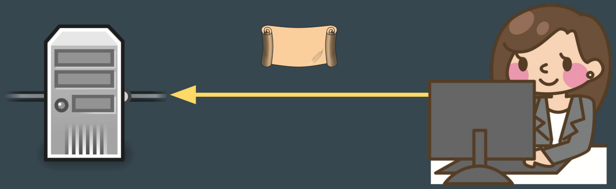
Web applications are about moving data around between users and components of the application!

Moving data ⇒ Information Flow

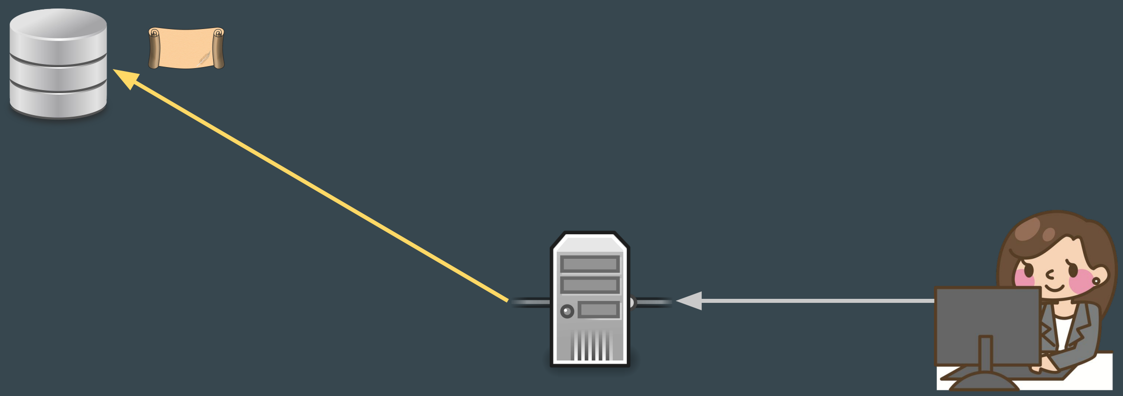
Information Flow



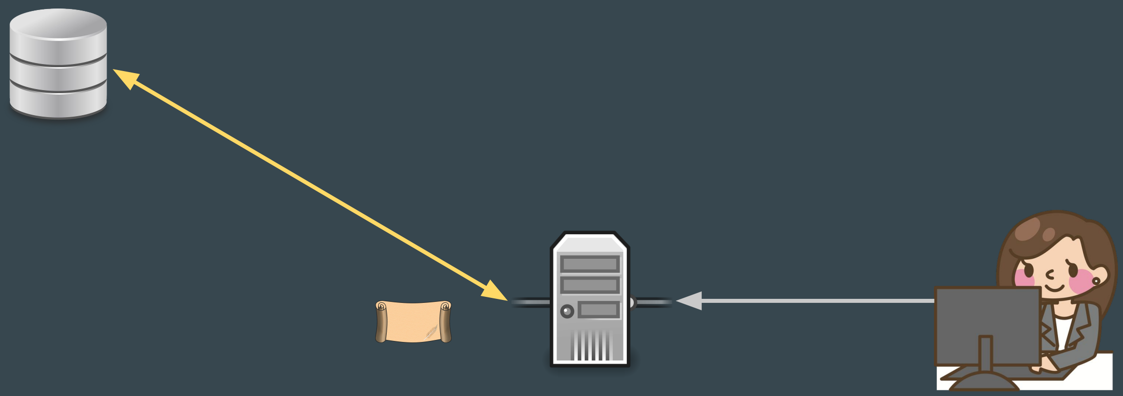
Example Flow - Password Reset (1)



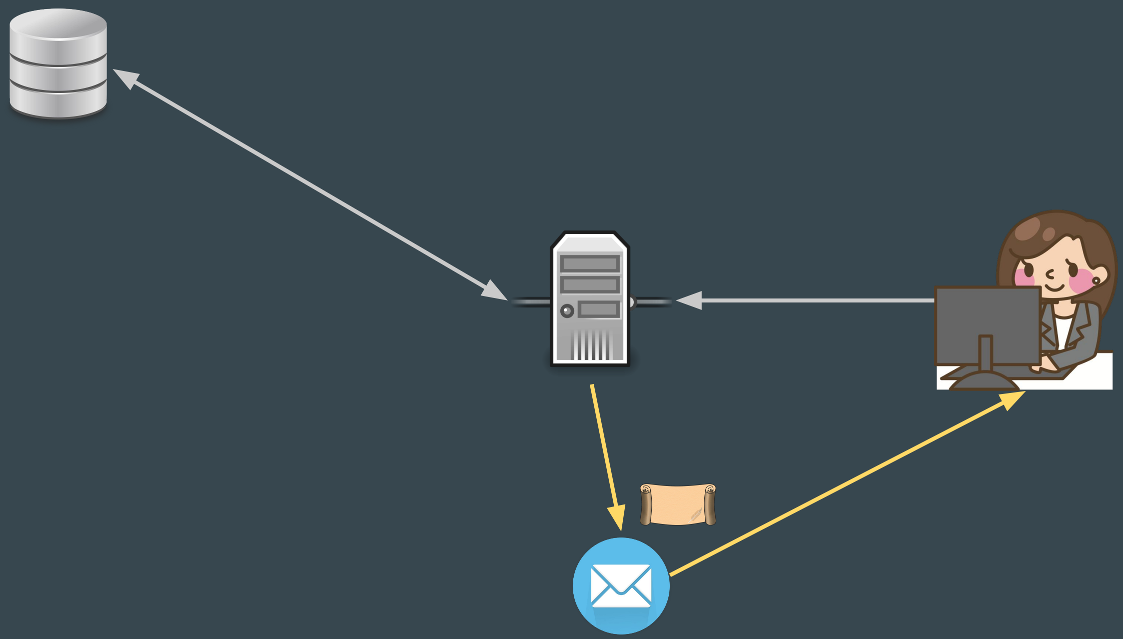
Example Flow - Password Reset (2)



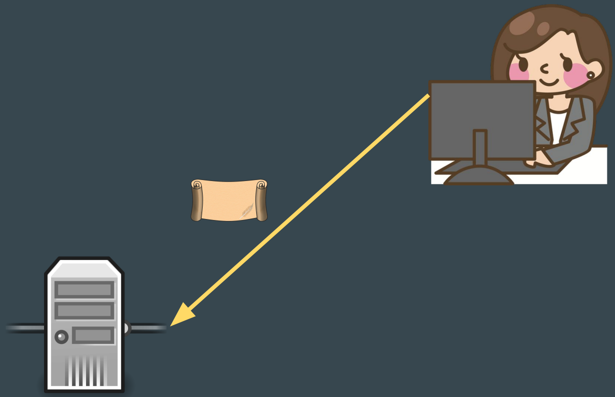
Example Flow - Password Reset (3)



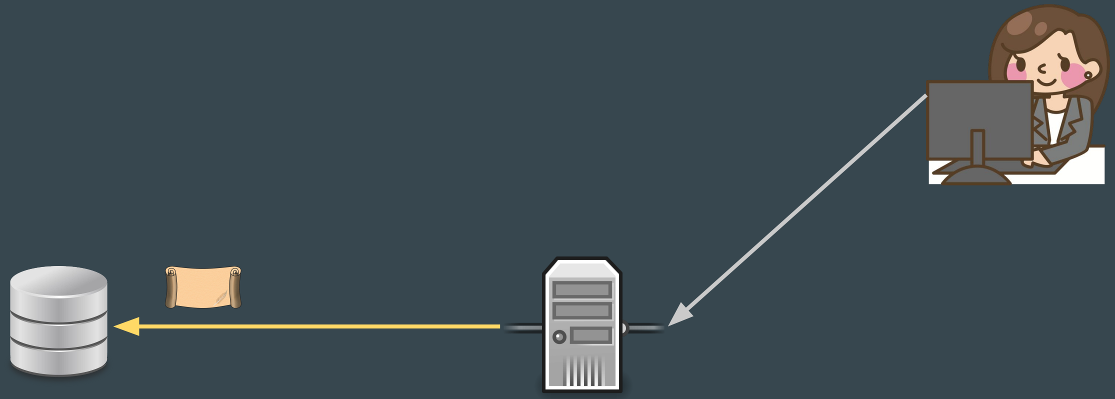
Example Flow - Password Reset (4)



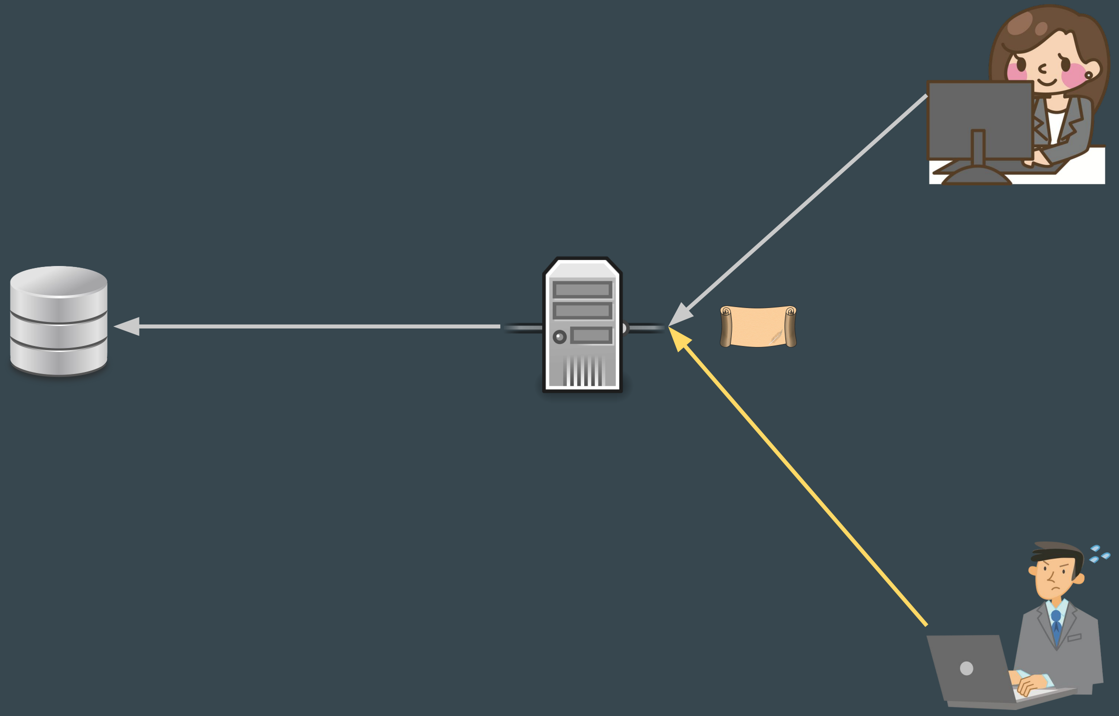
Example Flow - Discussion Board (1)



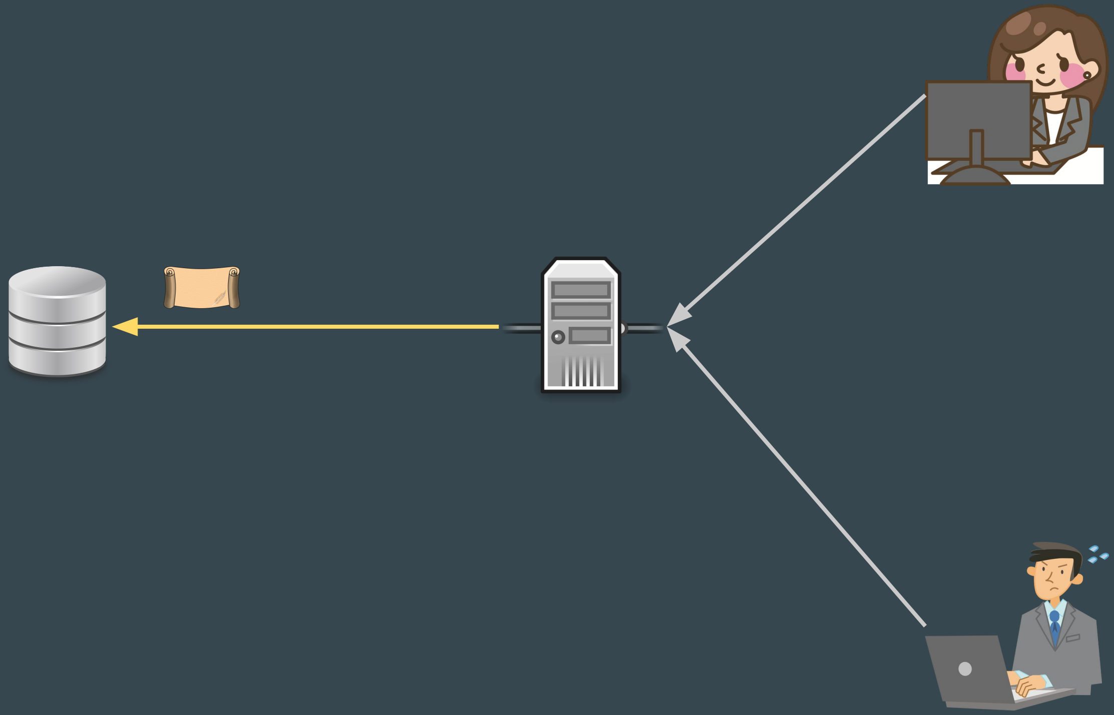
Example Flow - Discussion Board (2)



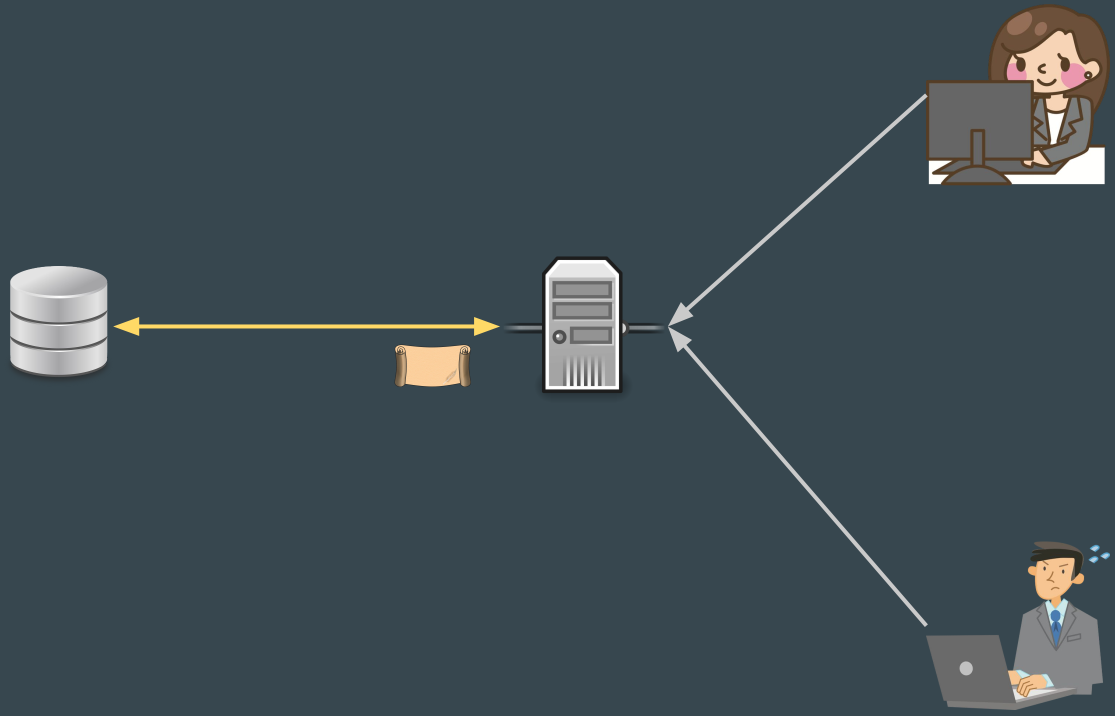
Example Flow - Discussion Board (3)



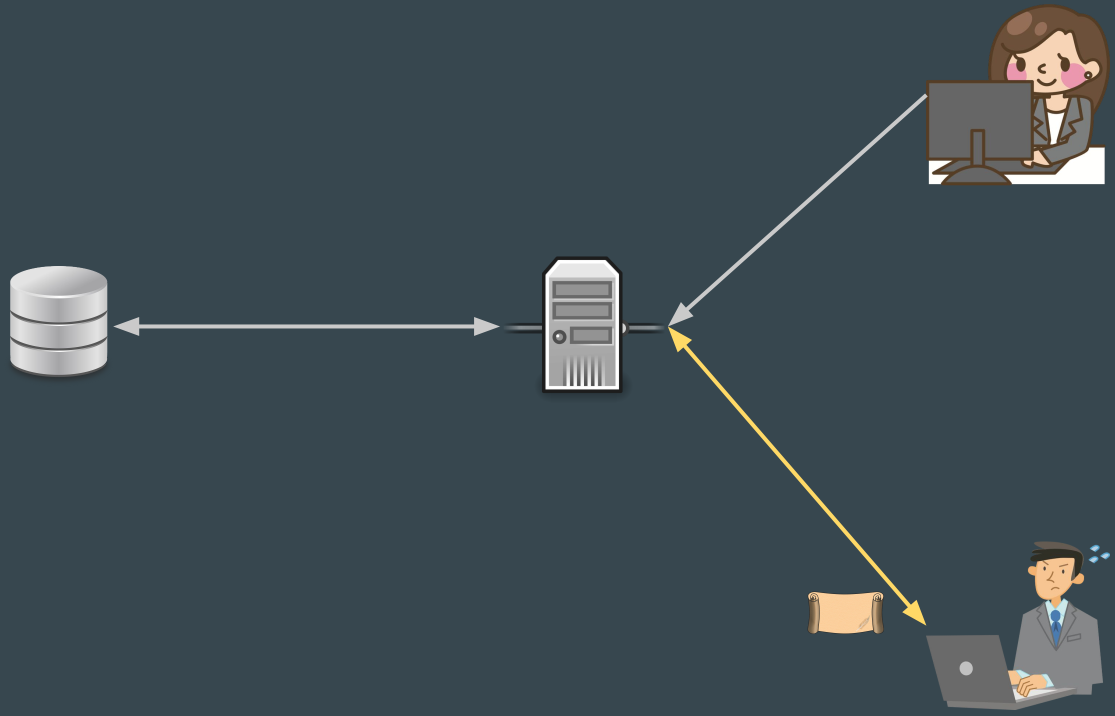
Example Flow - Discussion Board (4)



Example Flow - Discussion Board (5)



Example Flow - Discussion Board (6)



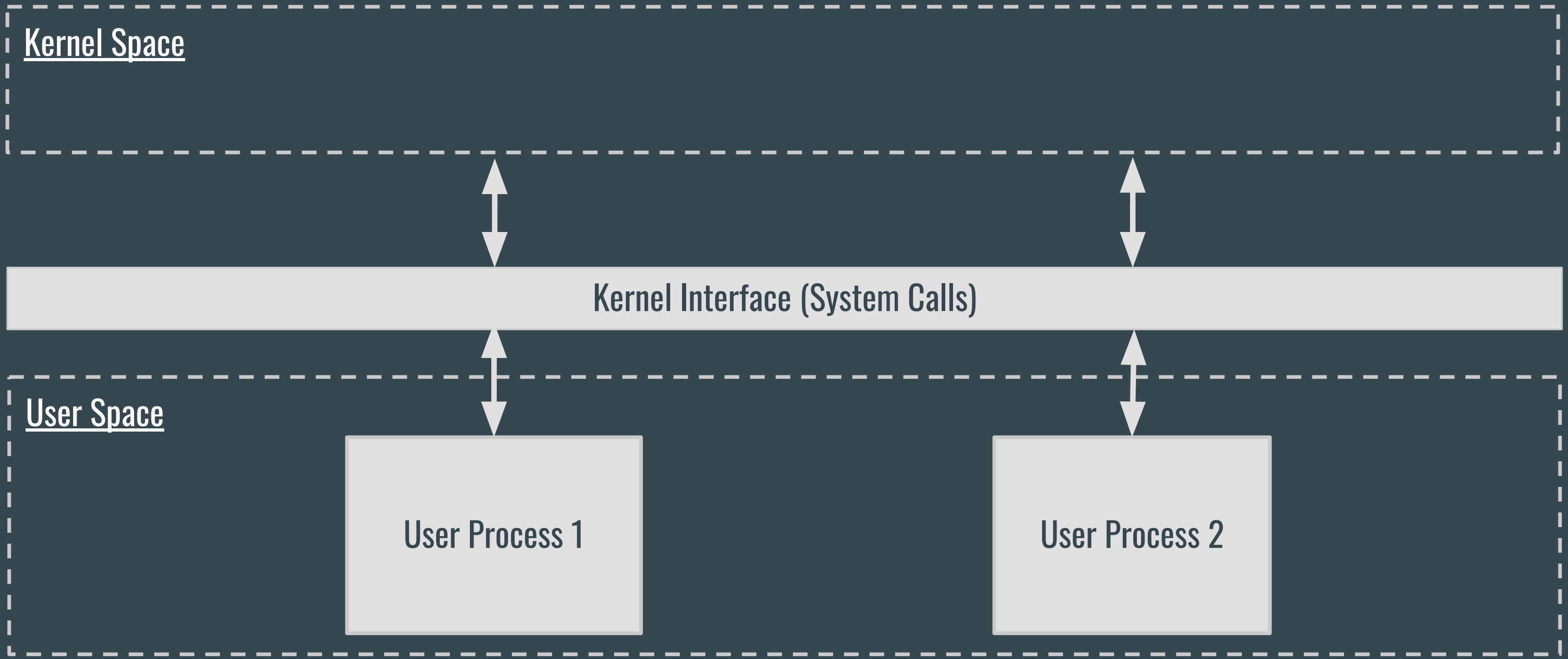
Information Flow Control:
Must ensure that the “correct” information flows
from/to the “correct” ***entities***!

Information flow *entities* are broader than you think...

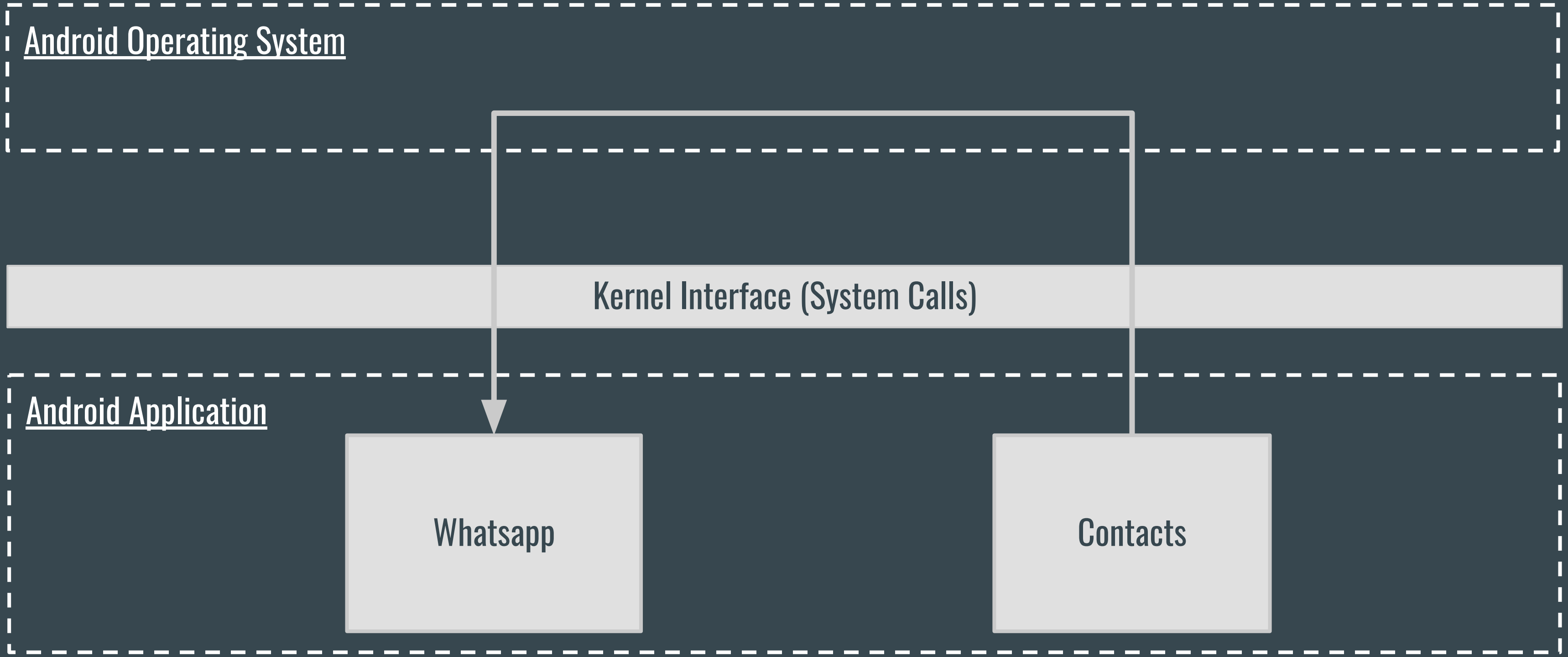
- Physically separated: Users / parties / servers

- Logically separated:
  - Trusted vs Untrusted code bases
  - Different threads or processes
  - Program/code components (classes, functions, etc)

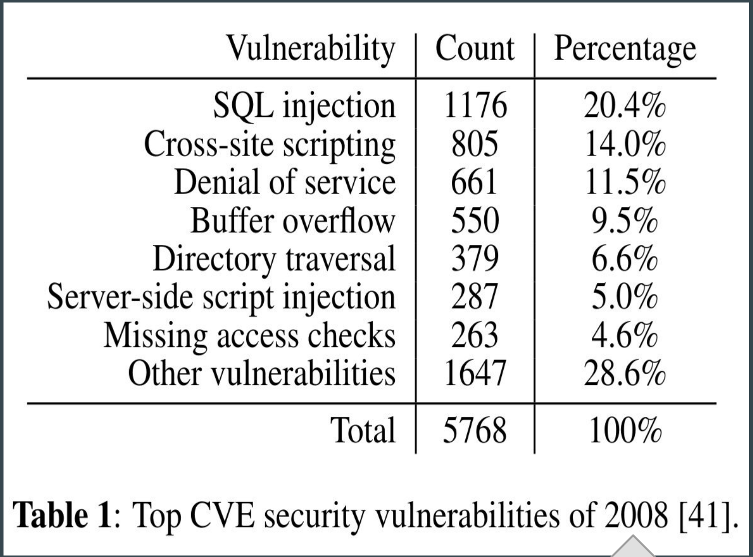
Information flow *entities* are broader than you think...



Information flow *entities* are broader than you think...

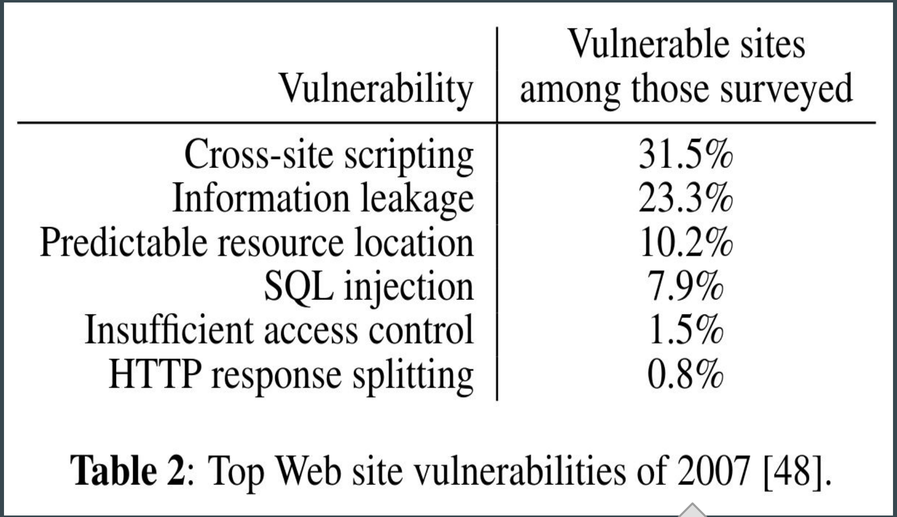


More common than you think!



Vulnerability	Count	Percentage
SQL injection	1176	20.4%
Cross-site scripting	805	14.0%
Denial of service	661	11.5%
Buffer overflow	550	9.5%
Directory traversal	379	6.6%
Server-side script injection	287	5.0%
Missing access checks	263	4.6%
Other vulnerabilities	1647	28.6%
Total	5768	100%

Table 1: Top CVE security vulnerabilities of 2008 [41].



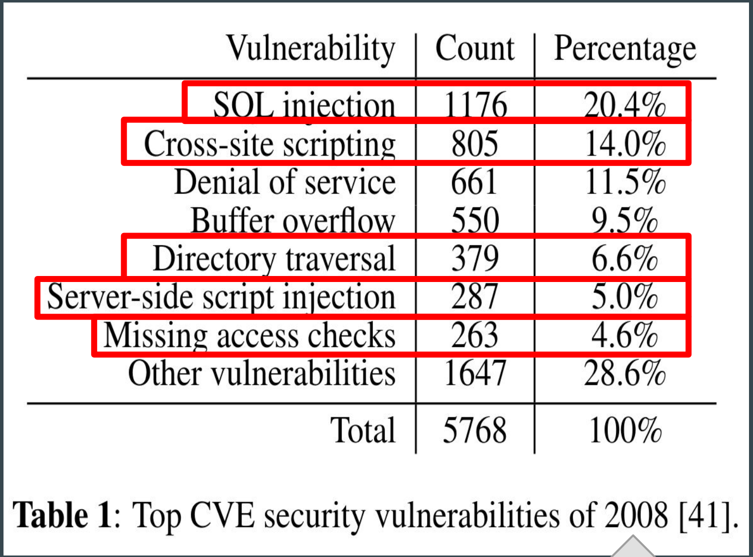
Vulnerability	Vulnerable sites among those surveyed
Cross-site scripting	31.5%
Information leakage	23.3%
Predictable resource location	10.2%
SQL injection	7.9%
Insufficient access control	1.5%
HTTP response splitting	0.8%

Table 2: Top Web site vulnerabilities of 2007 [48].

Percentage out of all reported vulnerabilities

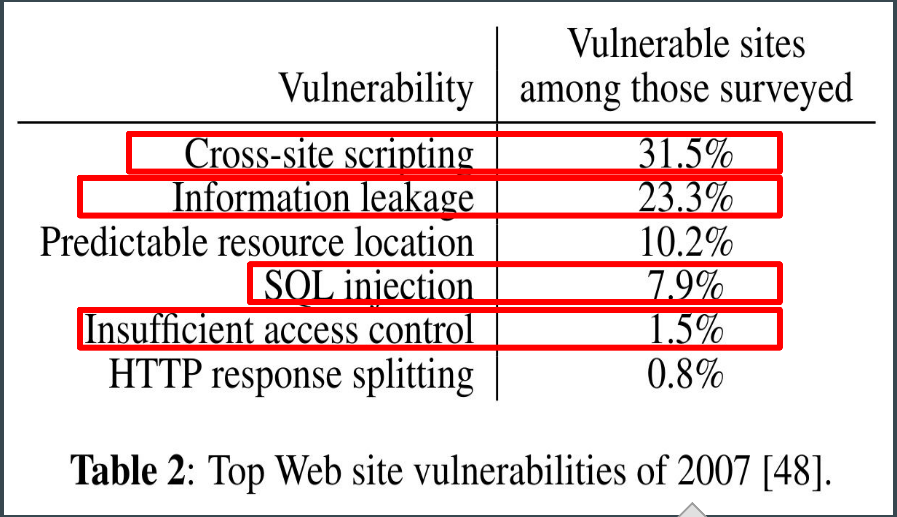
Percentage of websites suffering from this vulnerability

More common than you think!



Vulnerability	Count	Percentage
SQL injection	1176	20.4%
Cross-site scripting	805	14.0%
Denial of service	661	11.5%
Buffer overflow	550	9.5%
Directory traversal	379	6.6%
Server-side script injection	287	5.0%
Missing access checks	263	4.6%
Other vulnerabilities	1647	28.6%
Total	5768	100%

Table 1: Top CVE security vulnerabilities of 2008 [41].



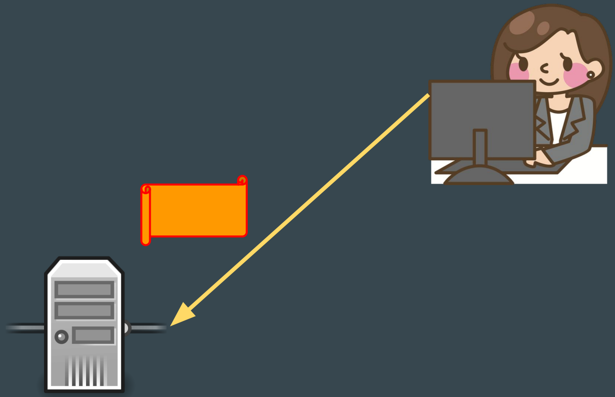
Vulnerability	Vulnerable sites among those surveyed
Cross-site scripting	31.5%
Information leakage	23.3%
Predictable resource location	10.2%
SQL injection	7.9%
Insufficient access control	1.5%
HTTP response splitting	0.8%

Table 2: Top Web site vulnerabilities of 2007 [48].

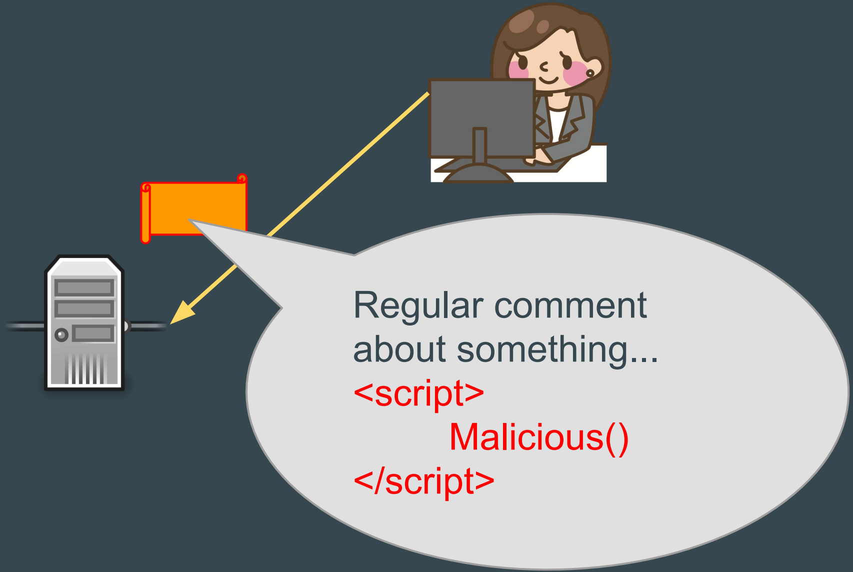
Percentage out of all reported vulnerabilities

Percentage of websites suffering from this vulnerability

Bad Flow - Cross-Site Scripting (1)

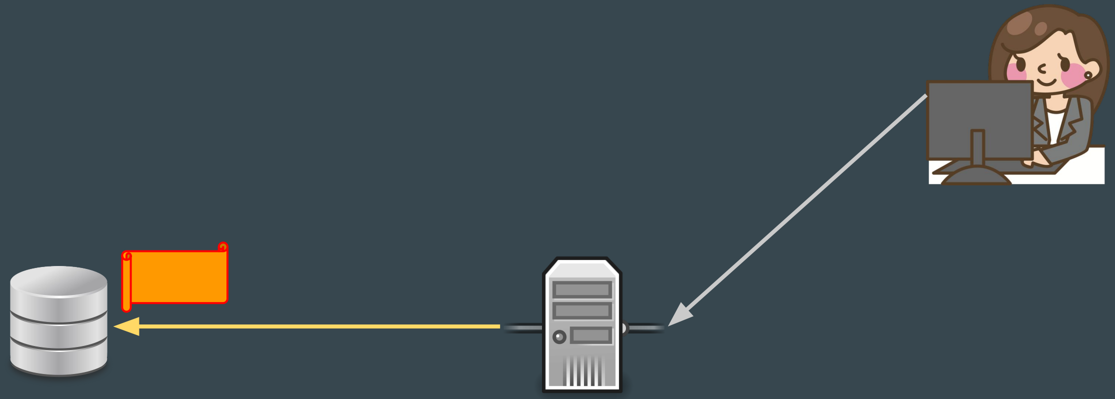


Bad Flow - Cross-Site Scripting (2)

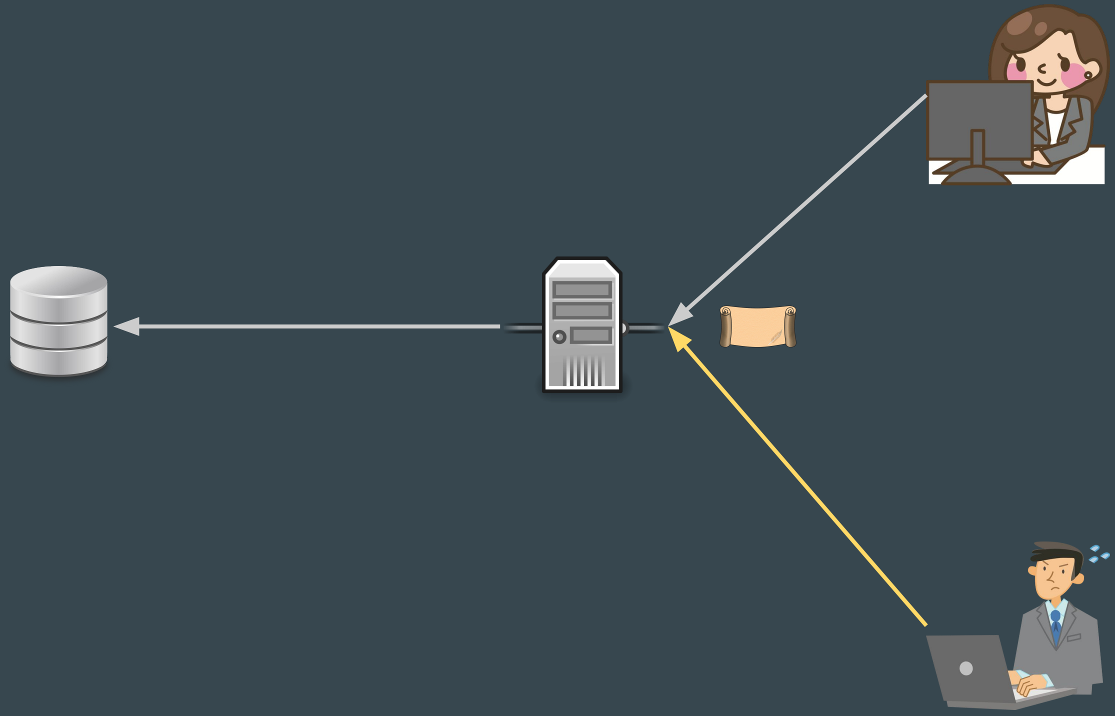


```
<script>
        Malicious()
</script>
```

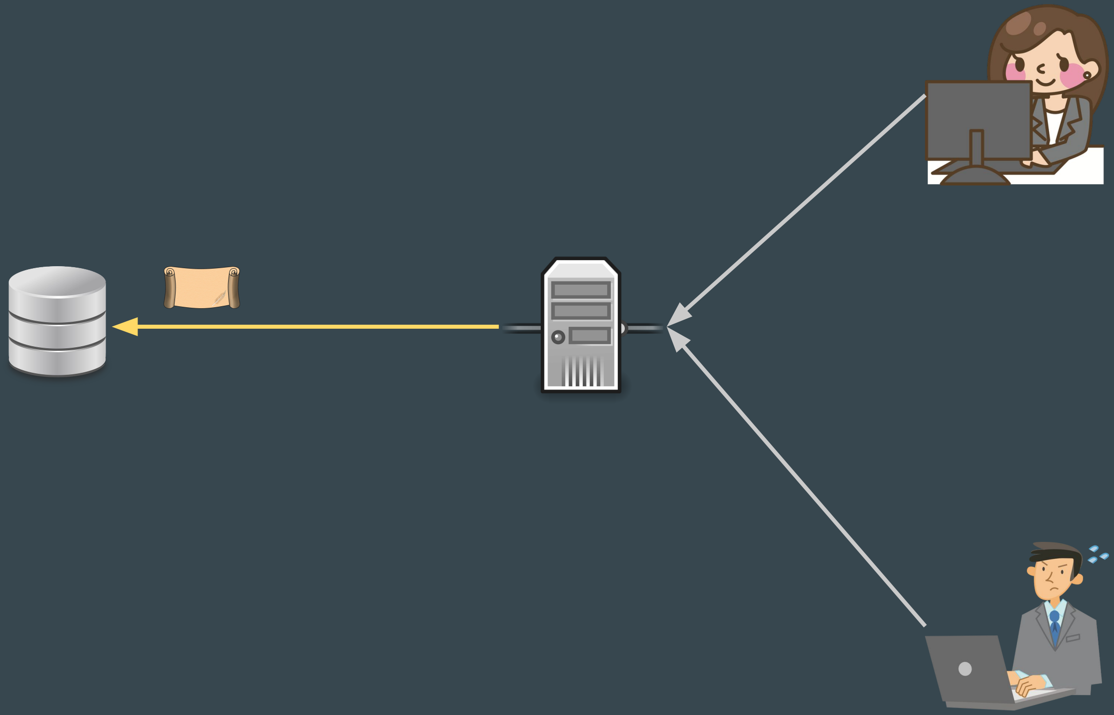
Bad Flow - Cross-Site Scripting (3)



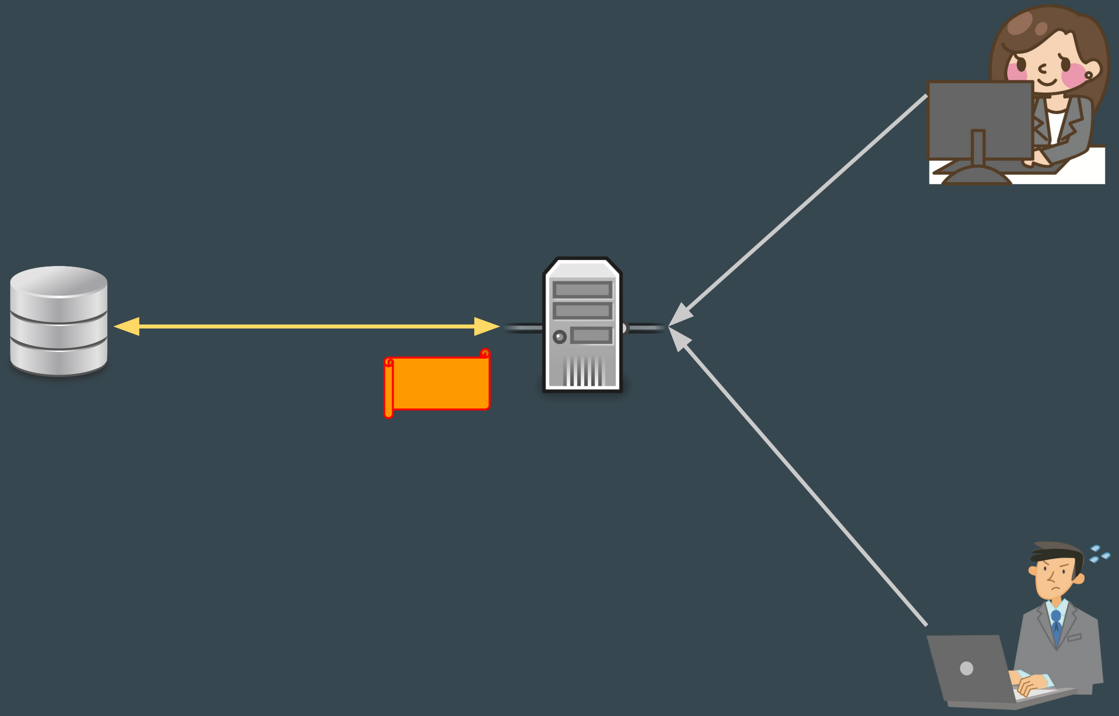
Bad Flow - Cross-Site Scripting (4)



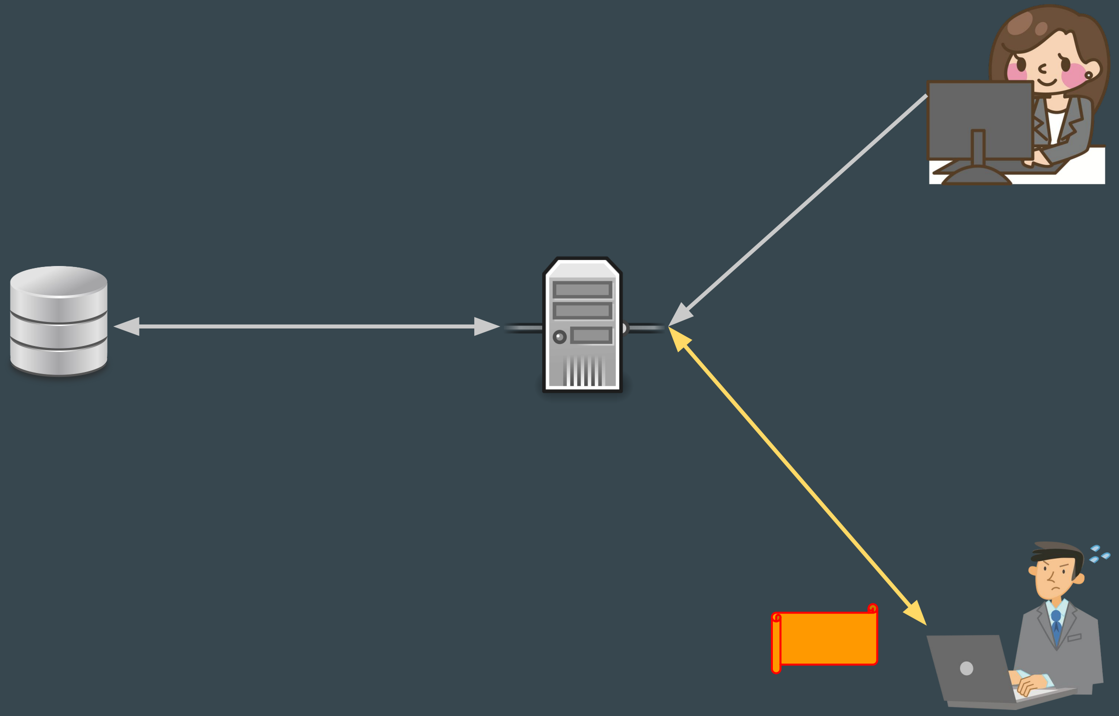
Bad Flow - Cross-Site Scripting (5)



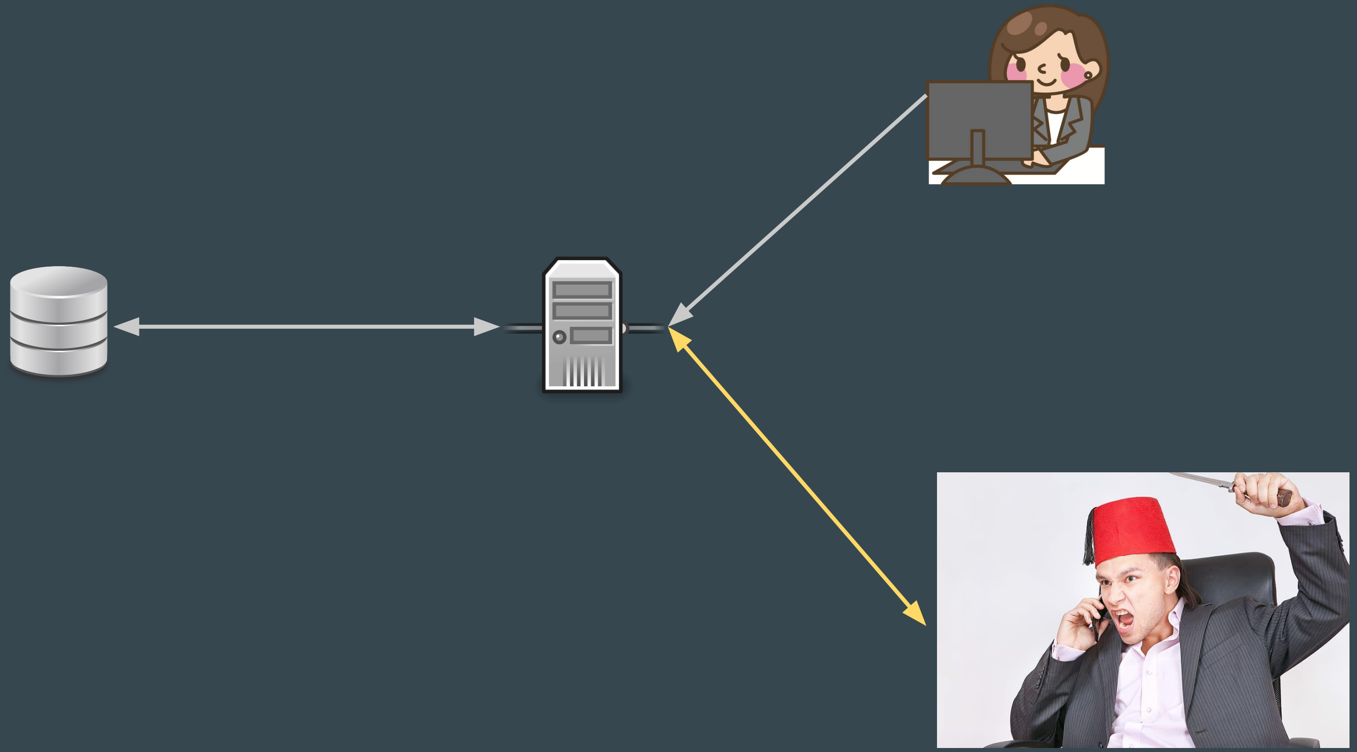
Bad Flow - Cross-Site Scripting (6)



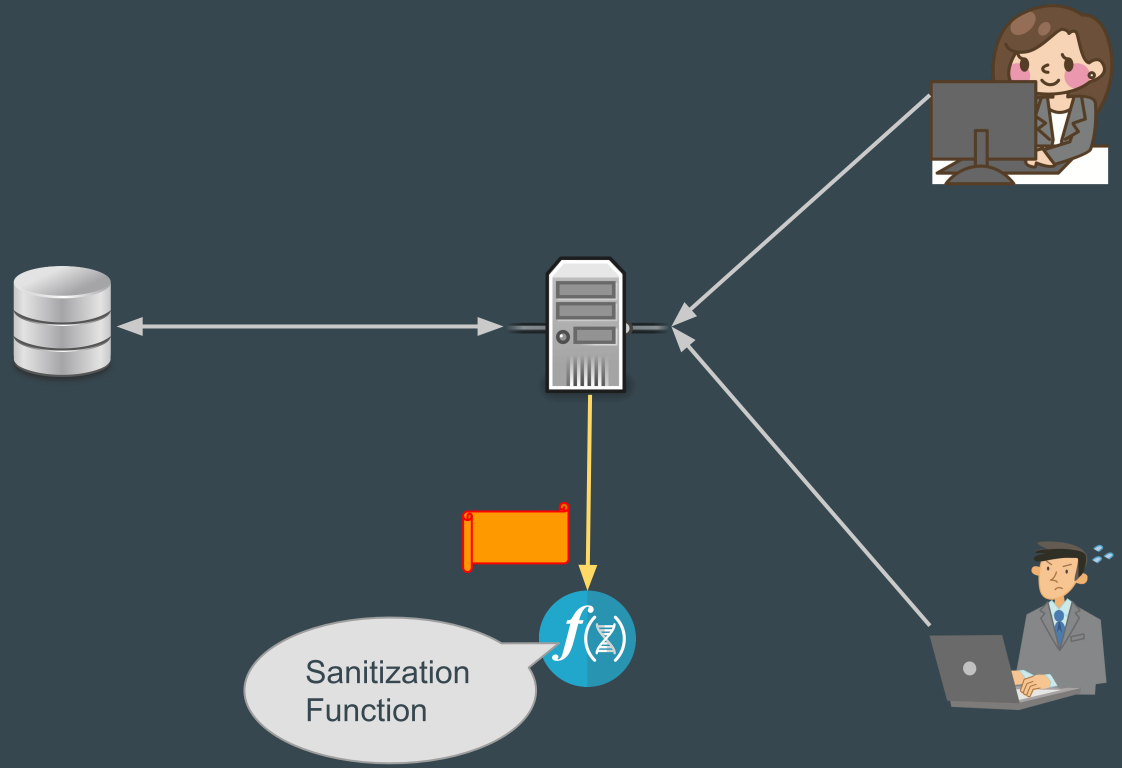
Bad Flow - Cross-Site Scripting (7)



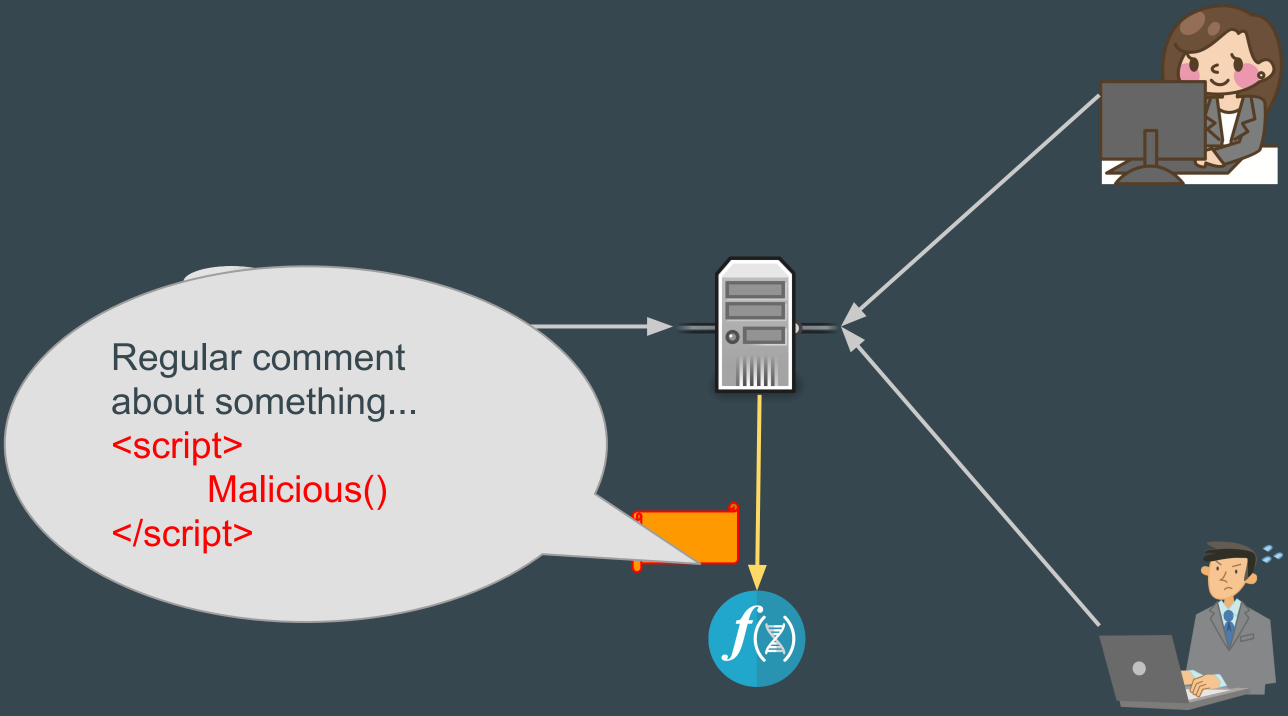
Bad Flow - Cross-Site Scripting (7)



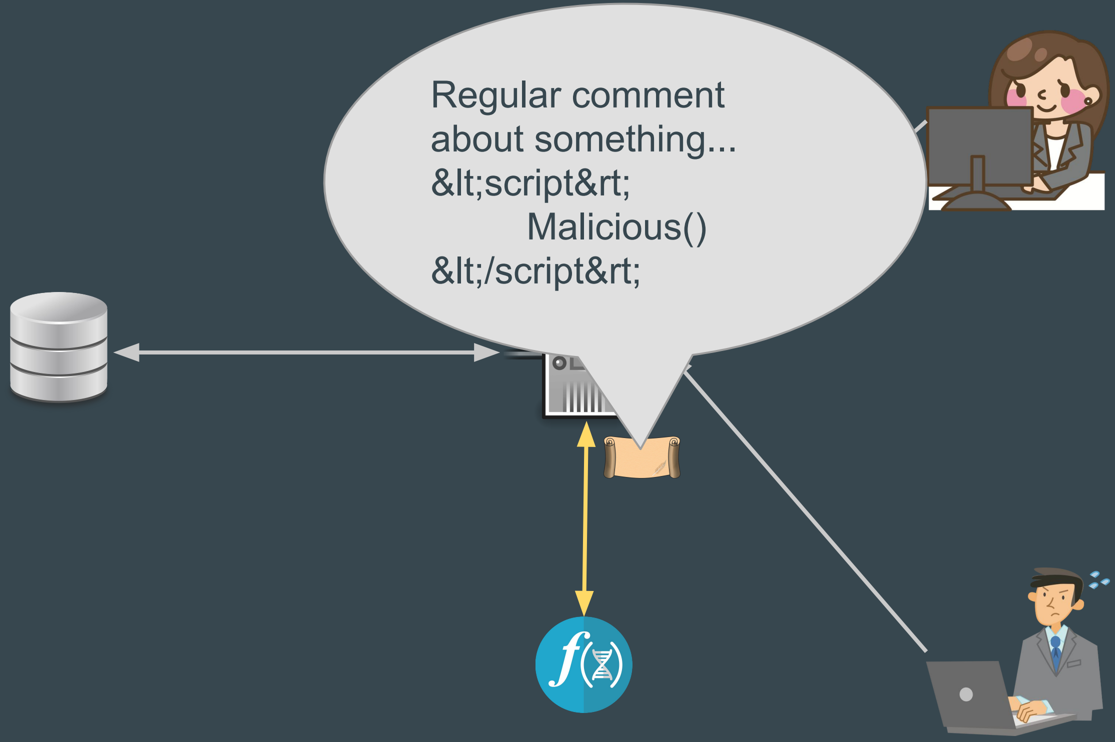
Good Flow



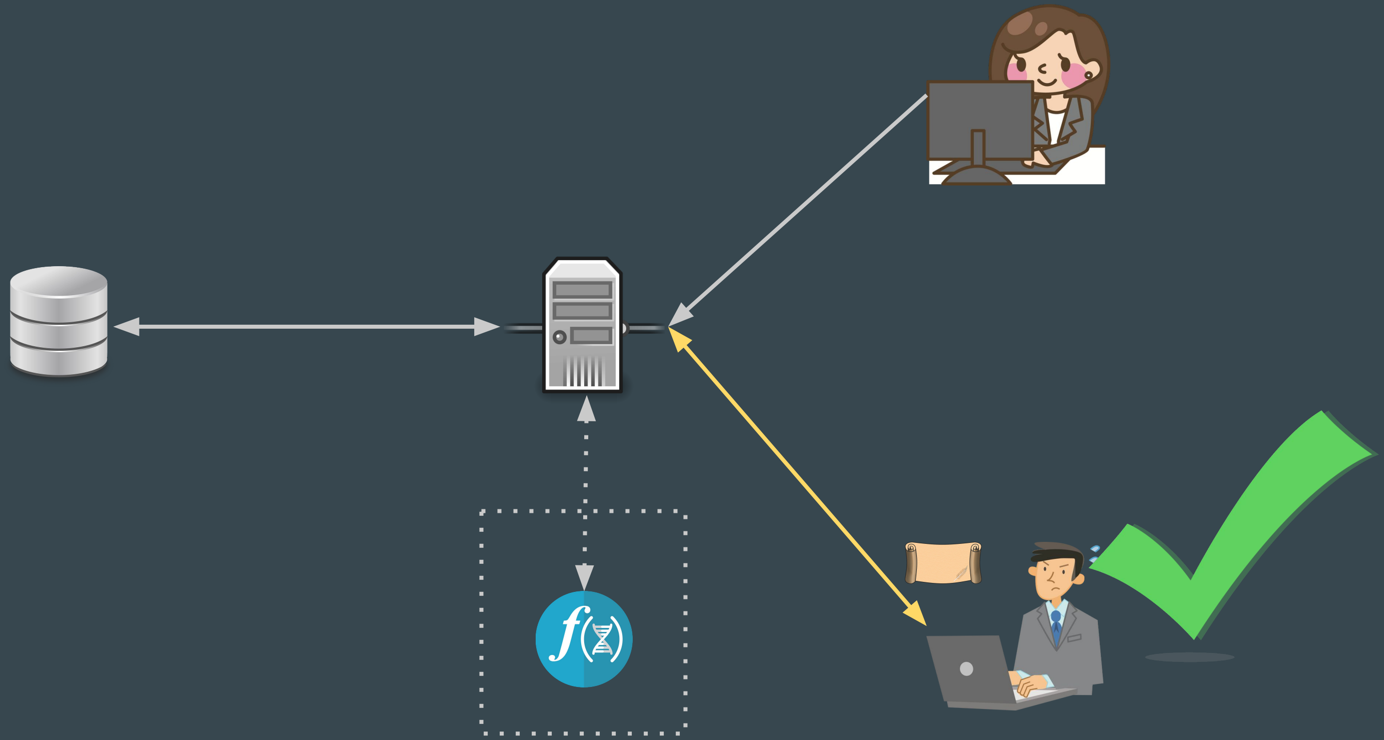
Good Flow



Good Flow


Regular comment
about something...
&lt;script&rt;
Malicious()
&lt;/script&rt;

Good Flow



Goal of Resin

- Help programmers avoid information flow mistakes
    - Omitting checks
- API for explicitly defining information flow assertions
- Automatic enforcement of these assertions
    - Runtime enforcement

How does Resin work?

Retracing Resin's Design (1)

```
def insert_post(request, response):
  post_content = request.post_content
  # Insert the new post into database
  Database.insert(post_content)
  # Signal success to user
  response.send_to_user("success")
```

Retracing Resin's Design (2)

```
def get_post(request, response):
  post_id = request.post_id
  # Look up post content from database
  post_content = Database.lookup(post_id)
  # Send post content to requesting user
  response.send_to_user(post_content)
```

Retracing Resin’s Design (3)

```
def get_post(request, response):
  post_id = request.post_id
  # Look up post content from d
  post_content = Databa
  # Send post co
  response.send_to
```

What’s Wrong!?

Retracing Resin's Design (4)

```
def insert_post(request, response):
  post_content = request.post_content
  # Insert the new post into database
  Database.insert(post_content)
  # Signal success to user
  response.send_to_user("success")
```

Comes from a user!
(Policy Object)

Retracing Resin's Design (5)

```
def get_post(request, response):
  post_id = request.post_id
  # Look up post content from database
  post_content = Database.lookup(post_id)
  # Send post content to requesting user
  response.send_to_user(post_content)
```

Potentially came from a user!
(data tracking)

Something that was potentially came from a user is sent to a user!
(filter objects)

Resin Design

1. Programmers explicitly annotate data with policy objects
2. Programmers use filter objects to define boundaries
    a. Filter is like a channel
    b. Filter checks that the data going through this channel is annotated with appropriate policy.
3. Resin <u>automatically</u> tracks annotations as its associated data moves around.

Resin Design

1. Programmers explicitly annotate data with policy objects
2. Programmers use filter objects to define boundaries
   a. Filter is like a channel
   b. Filter checks that the data going through this channel is annotated with appropriate policy.
3. Resin automatically tracks annotations as its associated data moves around.

How would you implement something like this?

Simple Implementation (1)

```
def insert_post(request, response):
  post_content_and_policy =
    (request.post_content, {comes_from_user: True})
  # Insert the new post into database
  Database.insert(post_content_and_policy)
  # Signal success to user
  response.send_to_user("success")
```

Manual policy annotation

Taint

Policy is Serialized with data to persistent storage.

Simple Implementation (2)

```
def get_post(request, response):
  post_id = request.post_id
  # Look up post content from database
  post_content_and_policy = Database.lookup(post_id)
  # Send post content to requesting user
  user_filter(response.send_to_user, post_content_and_policy)
```

(Automatic) data tracking

Simple Implementation (3)

```
def user_filter(channel, data_and_policy):
  data, policy = data_and_policy
  if policy.comes_from_user:
    raise Error("Unsafe!")
  channel(data)
```

Programmer-defined filter

Design Evaluation (1)

Advantages:

Design Evaluation (2)

Advantages:

1. Simplicity!

Disadvantages:

Design Evaluation (3)

Advantages:

1. Simplicity!

Disadvantages:

1. Enforcement at runtime adds overhead (both space and time!)
   a. Resin has 33% runtime overhead
2. (for sample implementation, but not for Resin) Data tracking may be inaccurate

Taint Laundering

```
string_data, _ = string_data_and_policy

do_unsafe_things(string_data)



new_string = library_function(string_data_and_policy)

do_unsafe_things(new_string)
```

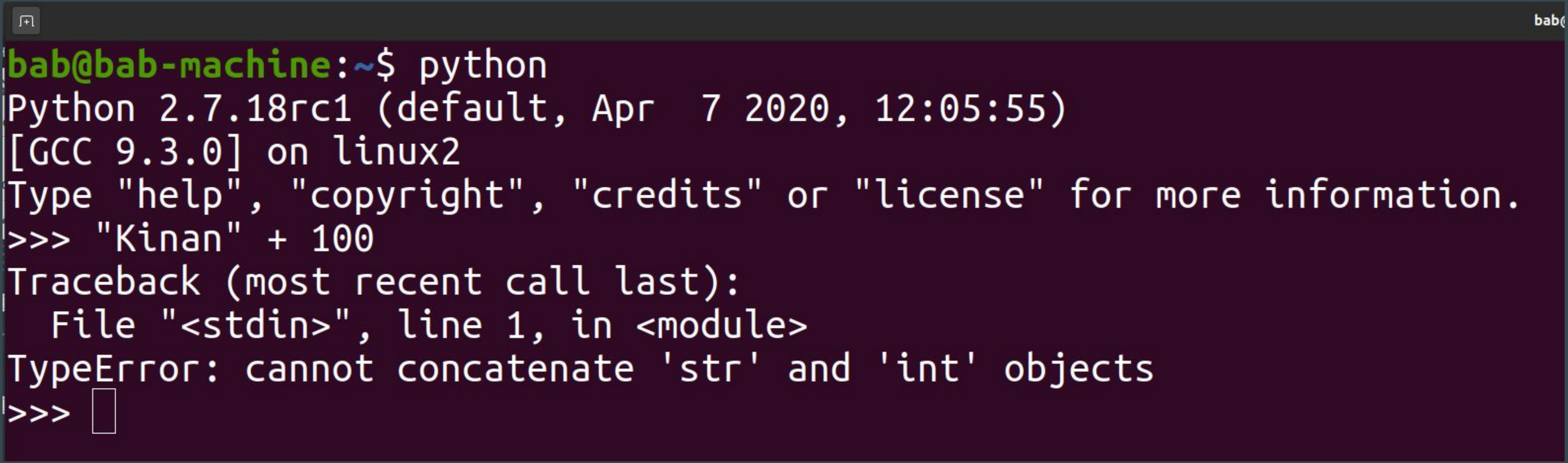
Other Issues

```
# what should be concat's policy!?

concat = string_with_policy1 + string_with_policy2



# what should be sum's policy?

sum = int_with_policy1 + int_with_policy2
```

Resin Design (1)

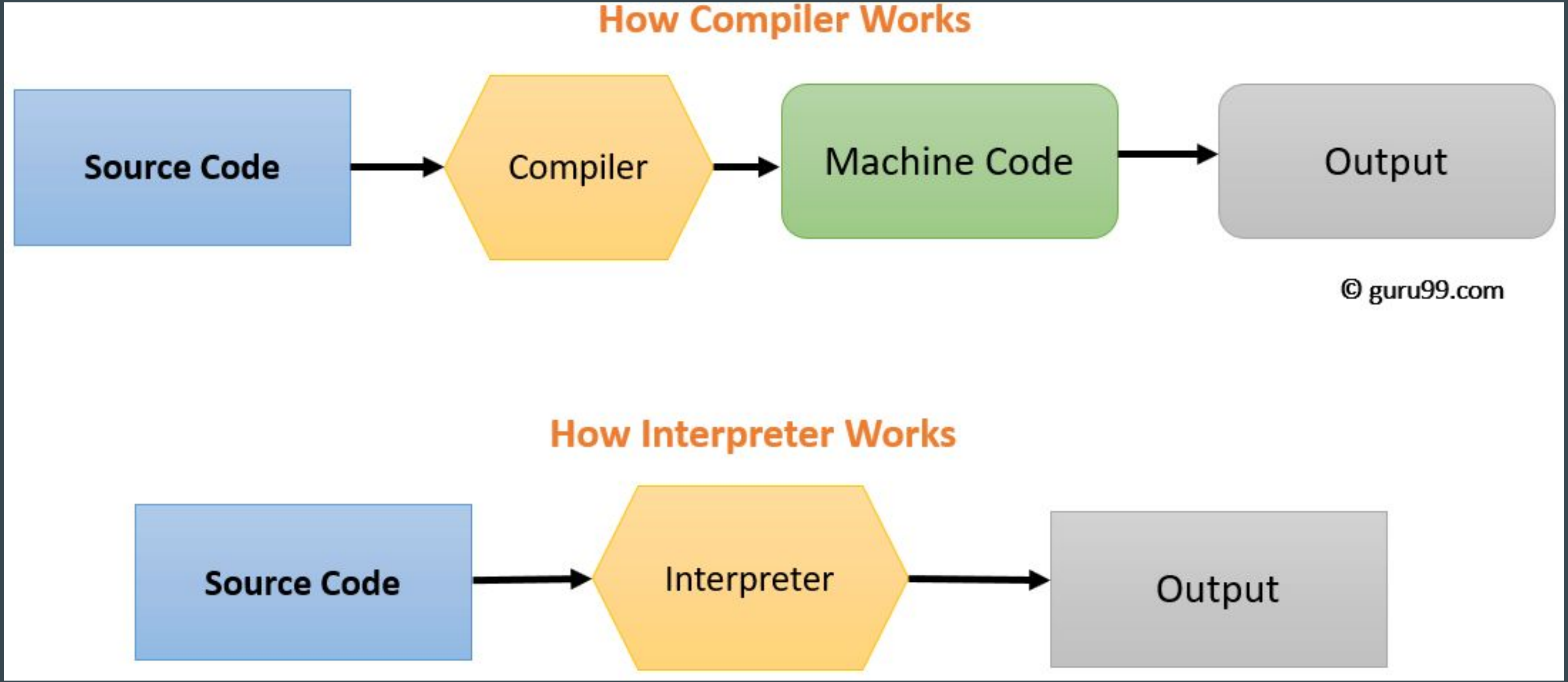
- Web languages are interpreted (python, php, nodejs, etc..)
  - Modify the runtime of the language so that the taints are stored within the runtime
  - "Make the taint part of the language"
  - Whenever the runtime interprets an operation, it can track the taint!
  - Similar to what happens in python if you add a string to an int!

Resin Design (2)



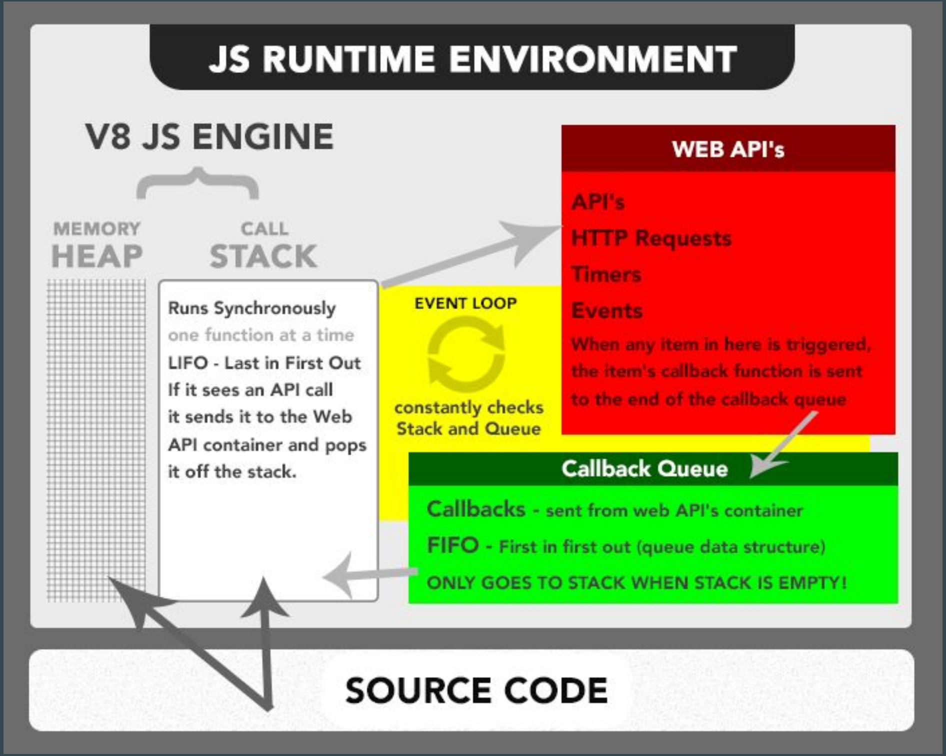
```
bab@bab-machine:~$ python
Python 2.7.18rc1 (default, Apr  7 2020, 12:05:55)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "Kinan" + 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> 
```

Detour (1) - Compiled vs Interpreted Languages



Credit: https://www.guru99.com/difference-compiler-vs-interpreter.html

Detour (2) - Language Runtime



Credit: https://medium.com/@olinations/the-javascript-runtime-environment-d58fa2e60dd0

Resin Design (3)

- Strings policy are set with respect to a range
  - Substrings can have different policies

- Operations on data with different policies combines the policies.

Resin Design (4)

- Resin does not protect against malicious developers:
  - They can intentionally miss-use policy objects and filters

- Resin does not protect against malicious or compromised code bases:
  - Resin does not protect against non-information flow based attacks (e.g. buffer overflow)
  - These attacks can be used to disable Resin's runtime protections or corrupt taints

Resin Design (5)

- Resin does not track implicit flow:
    - Challenging to track and discover implicit flow
    - Unclear what the policy should be
    - Developers should transform implicit flow to explicit ones

```
if condition(var_with_policy):
    # Information flows from var_with_policy to var_without policy implicitly
    var_without_policy = some_value

# Information flow from index to value implicitly
var_without_policy = array[index_with_policy]
```

Can we enforce information control flow differently?

Related Work (1) - Static Information Flow Control

- “Prove” that all the information flow in the program satisfy our information flow assertions/requirements!
  - Compiler
  - Strong type systems: type contains static taint (jif)
  - Static program analysis, theorem proving (nickel)

- May be automatic, user-assisted/interactive, or manual

Related Work (1) - Static Information Flow Control

- "Prove" that all the information flow in the program satisfy our information flow assertions/requirements!
  - Compiler
  - Strong type systems: type contains static taint (jif)
  - Static program analysis, theorem proving (nickel)

```
UserString s = request.post_content;

String<UserPolicy> s =
  request.post_content;
```

- May be automatic, user-assisted/interactive, or manual

Related Work (2) - Dynamic Information Flow Control

- Without modifying the runtime: requires languages with a strong type system (e.g. Haskel)
    - Type contains a dynamic taint (lio)
- Operating System level (HiStar, Dstar)
- Control flow integrity (Microsoft Control Flow Guard)

Related Work (2) - Dynamic Information Flow Control

- Without modifying the runtime: requires languages with a strong type system (e.g. Haskel)
    - Type contains a dynamic taint (lio)

```
TaintedString s =
 new TaintedString(request.post_content,userPolicy);
```

- Operating System level (HiStar, Dstar)

- Control flow integrity (Microsoft Control Flow Guard)

Discussion

1. Would you use Resin in your application?
   a. What requirements would you use to determine if you would use Resin or a similar system?
2. How can Resin/Information Flow Control help us guarantee better privacy (e.g. GDPR compliance)?
3. Can you think of other examples of Information Flow assertions or applications?
4. How would you evaluate a system like Resin? Did the paper have adequate evaluation?
5. What do you think about persistent policies?