

Static Program Analysis

Yue Li and Tian Tan



2020 Spring

Static Program Analysis

Static Analysis for Security

Nanjing University

Tian Tan

2020

Security

Achieving some goals in the presence of **adversaries**

Security

Achieving some goals in the presence of **adversaries**

Physical World

- Goals
 - Personal safety
 - Property safety
 - ...
- Adversaries
 - Thieves
 - Criminals
 - ...

Security

Achieving some goals in the presence of **adversaries**

Physical World

- Goals
 - Personal safety
 - Property safety
 - ...
- Adversaries
 - Thieves
 - Criminals
 - ...

Cyber World

- Goals
 - Dependability
 - Data safety
 - ...
- Adversaries
 - Crackers
 - Cyber attackers
 - ...

Computer Security

Becomes increasingly important nowadays

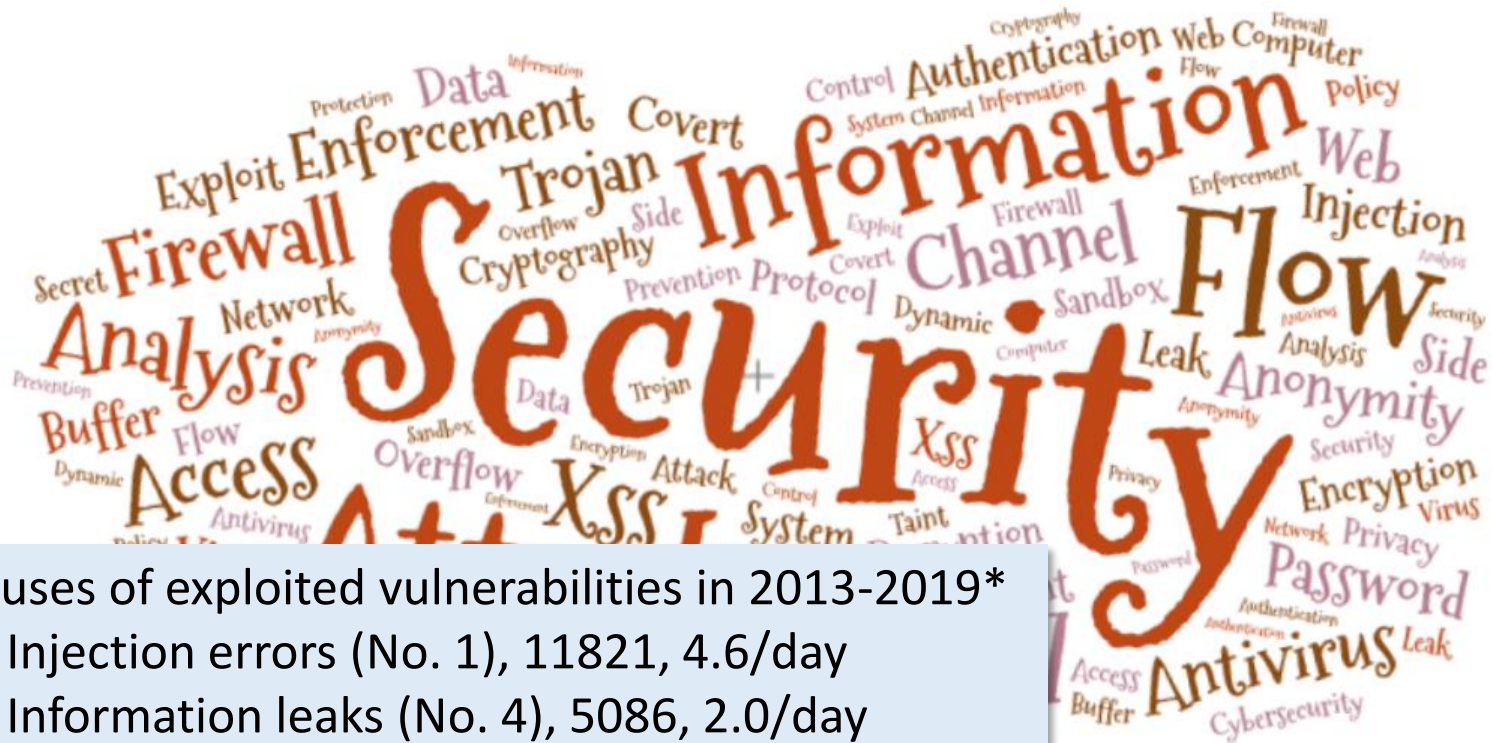
Computer Security

Becomes increasingly important nowadays



Computer Security

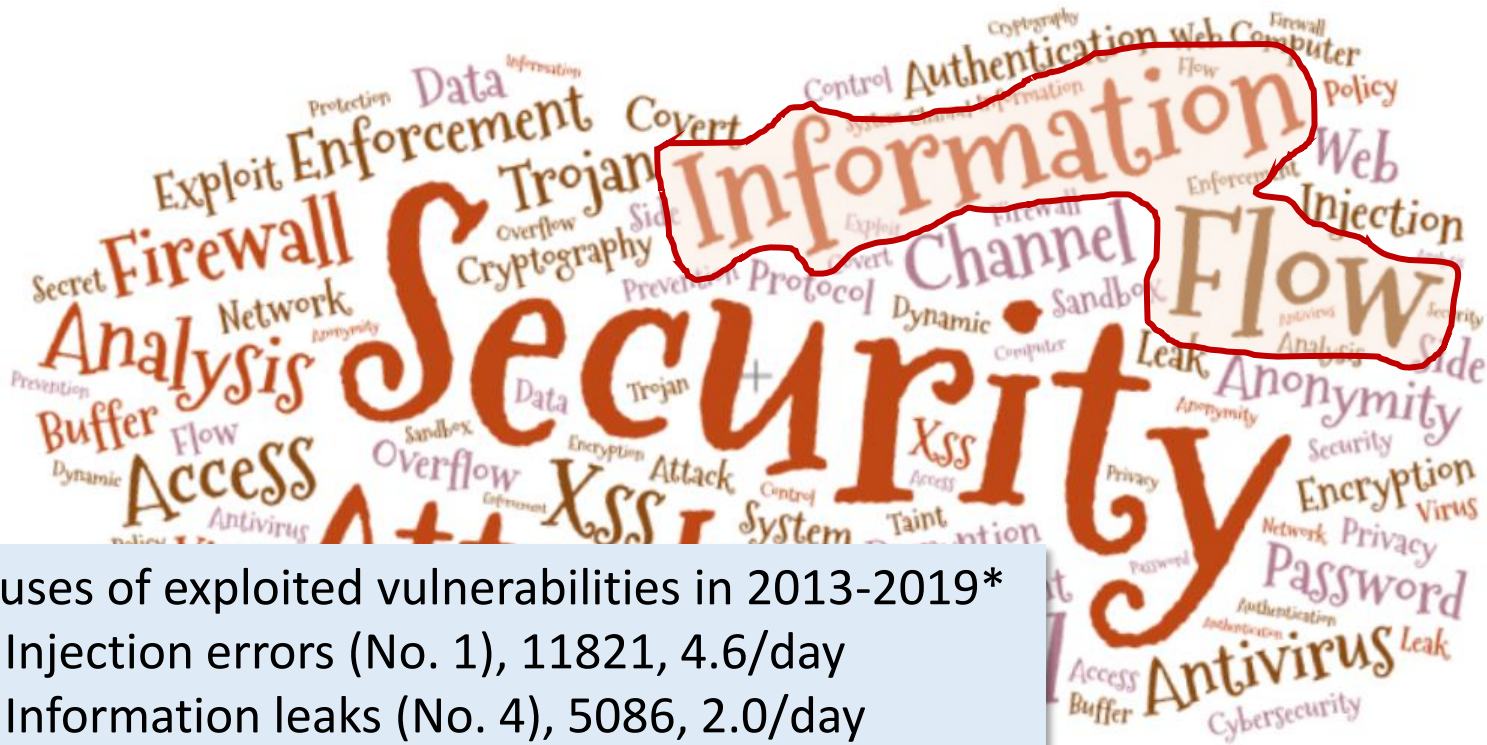
Becomes increasingly important nowadays



*National Vulnerability Database, <https://nvd.nist.gov/>

Computer Security

Becomes increasingly important nowadays



*National Vulnerability Database, <https://nvd.nist.gov/>

Contents



1. Information Flow Security
2. Confidentiality and Integrity
3. Explicit Flows and Covert Channels
4. Taint Analysis

Contents



- 1. Information Flow Security**
2. Confidentiality and Integrity
3. Explicit Flows and Covert Channels
4. Taint Analysis

Information Flow: An Example

4G^{HD} 11:26

✕

Log in via WeChat ID/Email/
QQ ID

Account software-analysis@nju

Password ✕

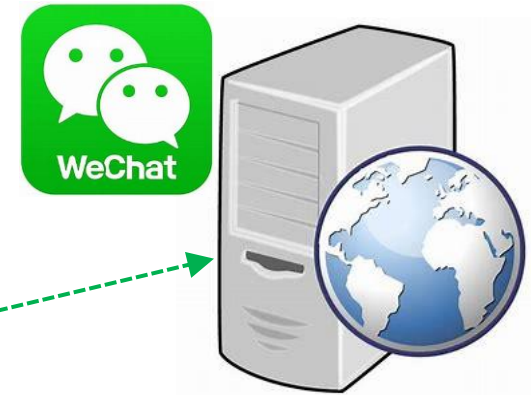
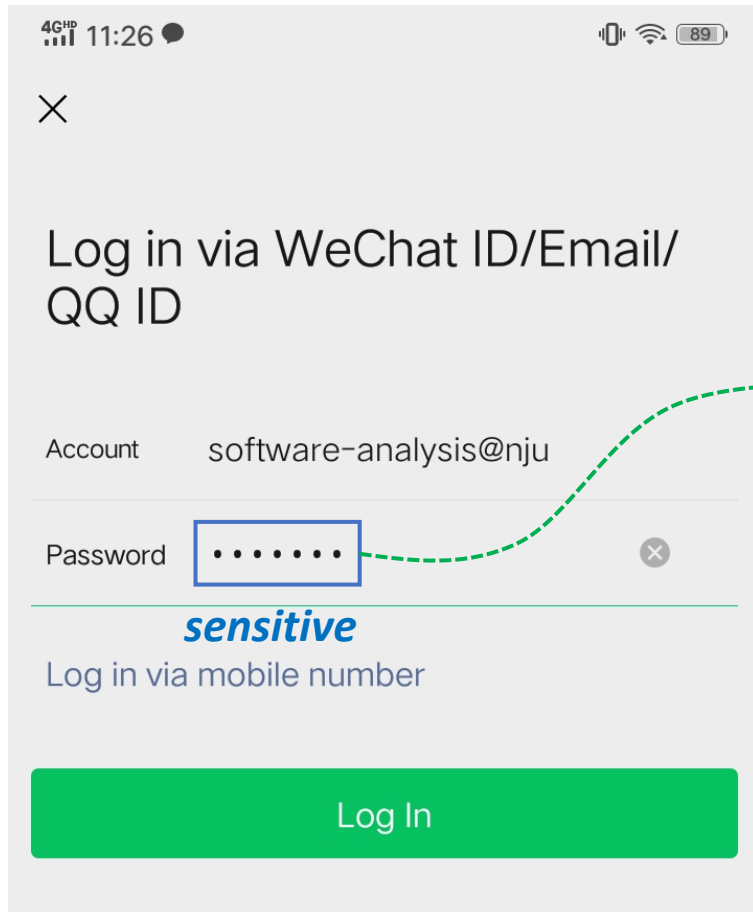
sensitive

Log in via mobile number

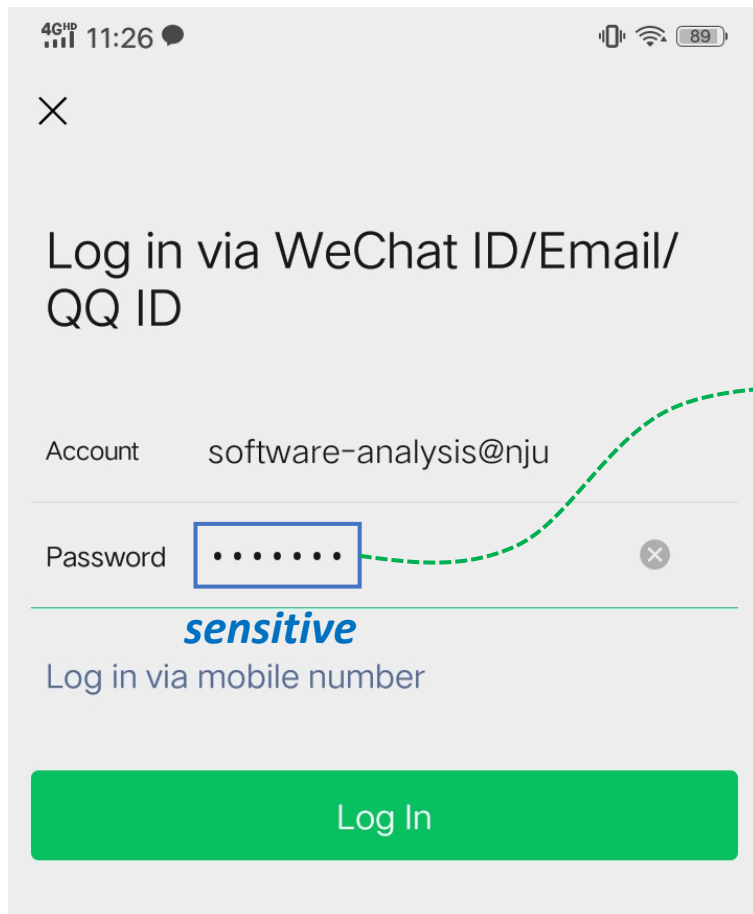
Log In

The image shows a mobile application login screen. At the top, there is a status bar with '4G^{HD}' and '11:26'. Below that is a close button (✕). The main heading is 'Log in via WeChat ID/Email/ QQ ID'. There is an 'Account' field with the value 'software-analysis@nju'. Below that is a 'Password' field containing seven dots, which is highlighted with a blue border. To the right of the password field is a clear button (✕). Below the password field, the word 'sensitive' is written in blue, italicized font. Underneath that is the text 'Log in via mobile number'. At the bottom is a large green button labeled 'Log In'.

Information Flow: An Example



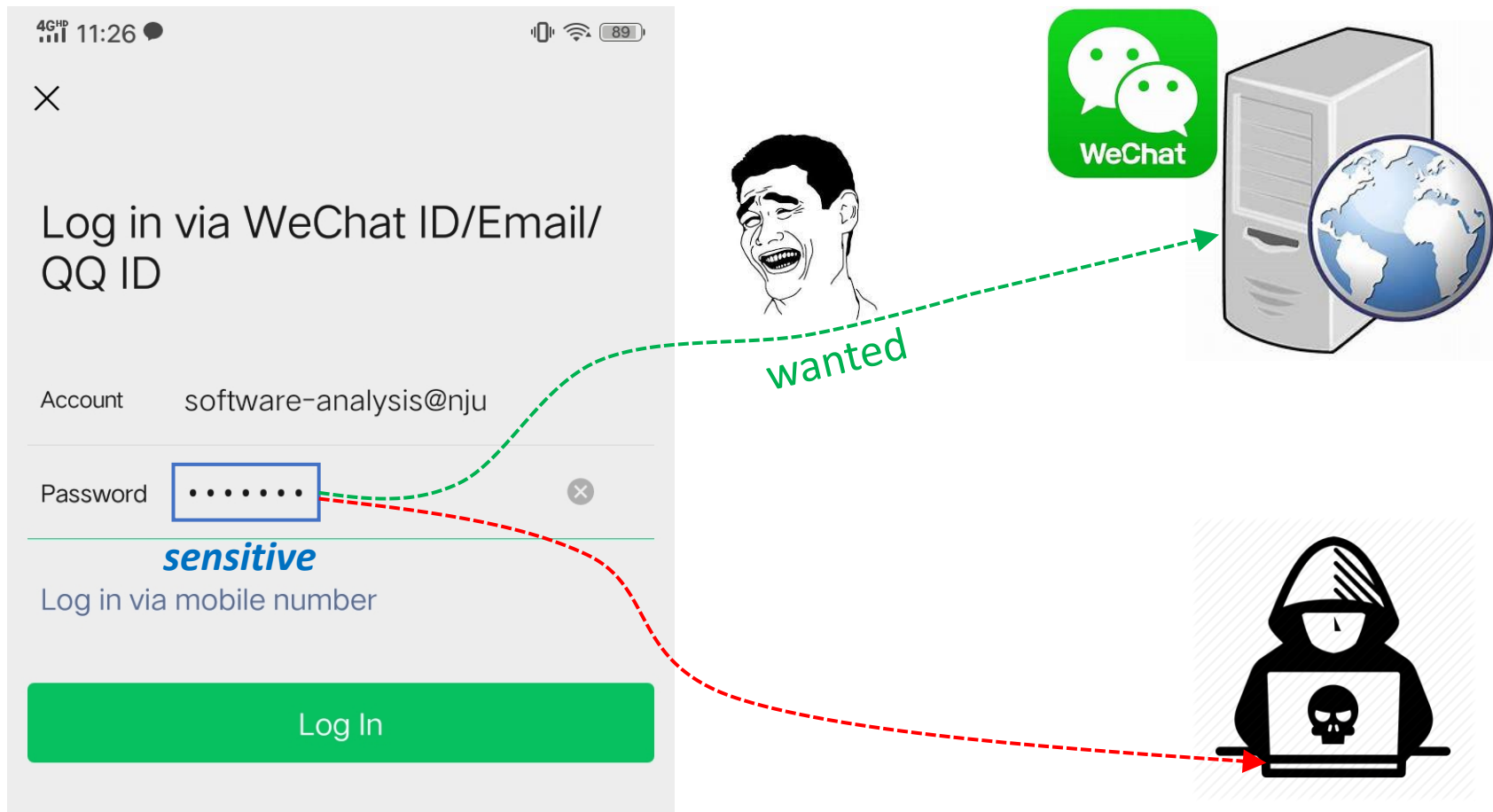
Information Flow: An Example



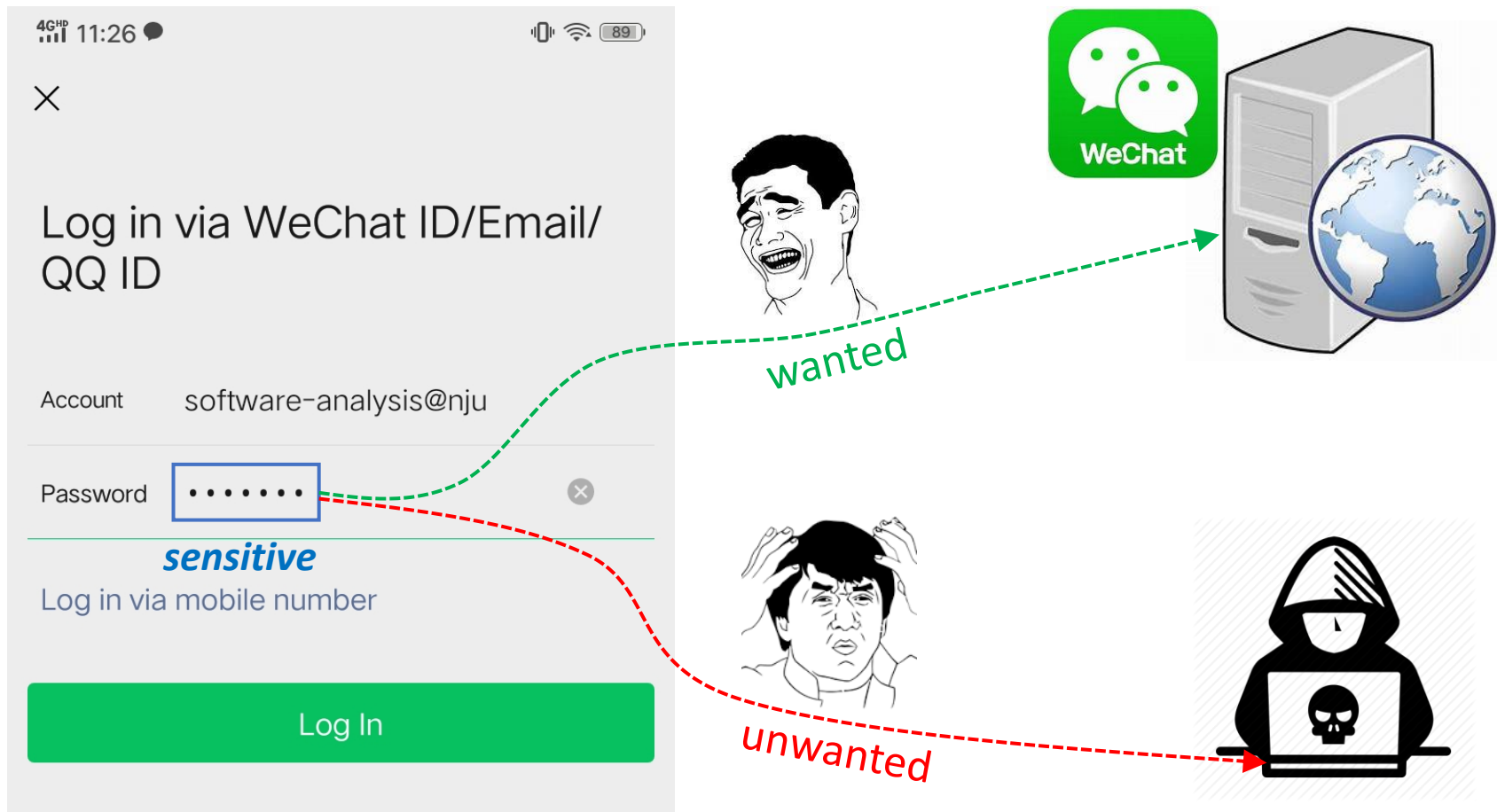
wanted



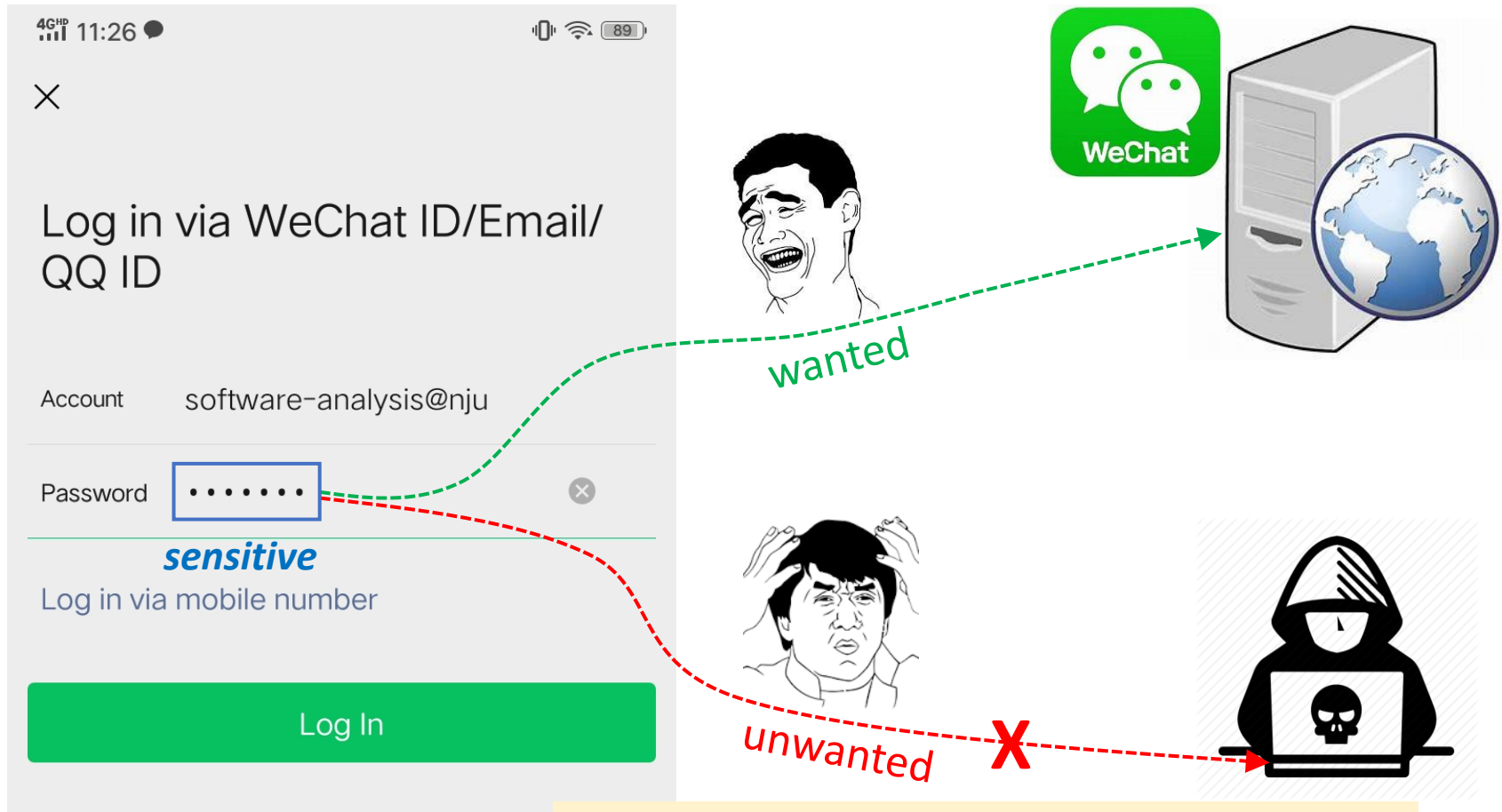
Information Flow: An Example



Information Flow: An Example



Information Flow Security: Motivation



Prevent unwanted information flow
Protect information security

Access Control vs. Information Flow Security

- Access control (a standard way to protect sensitive data)
 - Checks if the program has the rights/permissions to access certain information
 - Concerns how information is **accessed**

Access Control vs. Information Flow Security

- Access control (a standard way to protect sensitive data)
 - Checks if the program has the rights/permissions to access certain information
 - Concerns how information is **accessed**

What happens after that?

Access Control vs. Information Flow Security

- Access control (a standard way to protect sensitive data)
 - Checks if the program has the rights/permissions to access certain information
 - Concerns how information is **accessed**

What happens after that?

- Information flow security (end-to-end)
 - Tracks how **information flows** through the program to make sure that the program handles the information **securely**
 - Concerns how information is **propagated**

Access Control vs. Information Flow Security

- Access control (a standard way to protect sensitive data)
 - Checks if the program has the rights/permissions to access certain information
 - Concerns how information is **accessed**

What happens after that?

- Information flow security (end-to-end)
 - Tracks how **information flows** through the program to make sure that the program handles the information **securely**
 - Concerns how information is **propagated**

*"A practical system needs both **access** and **flow control** to satisfy all security requirements."*

— D. Denning, 1976

Information Flow*

- Information flow: if the information in variable x is transferred to variable y , then there is information flow $x \rightarrow y$

*Dorothy E. Denning and Peter J. Denning, “*Certification of Programs for Secure Information Flow*”. CACM 1977.

Information Flow*

- Information flow: if the information in variable x is transferred to variable y , then there is information flow $x \rightarrow y$

- Examples



```
y = x;
```

```
a = x;  
b.f = a;  
y = b.f;
```

* Dorothy E. Denning and Peter J. Denning, “*Certification of Programs for Secure Information Flow*”. CACM 1977.

Information Flow*

- Information flow: if the information in variable x is transferred to variable y , then there is information flow $x \rightarrow y$

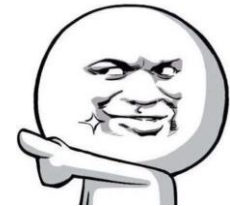
- Examples



```
y = x;
```

```
a = x;  
b.f = a;  
y = b.f;
```

Looks like pointer analysis?
We will see ...



* Dorothy E. Denning and Peter J. Denning, “*Certification of Programs for Secure Information Flow*”. CACM 1977.

Information Flow Security

Connects information flow to security


- Classifies program variables into different **security levels**
- Specifies permissible flows between these levels, i.e., information flow policy

Security Levels (Classes)

- The most basic model is two-level policy, i.e., a variable is classified into one of two security levels:
 1. H, meaning *high* security, secret information
 2. L, meaning *low* security, public observable information
- `h = getPassword(); // h is high security`
- `broadcast(l); // l is low security`

Security Levels (Classes)

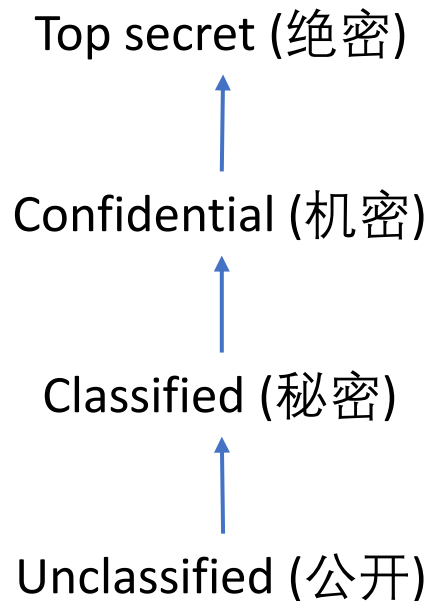
- The most basic model is two-level policy, i.e., a variable is classified into one of two security levels:
 1. H, meaning *high* security, secret information
 2. L, meaning *low* security, public observable information
 - `h = getPassword(); // h is high security`
 - `broadcast(l); // l is low security`

- Security levels can be modeled as *lattice**
 - $L \leq H$

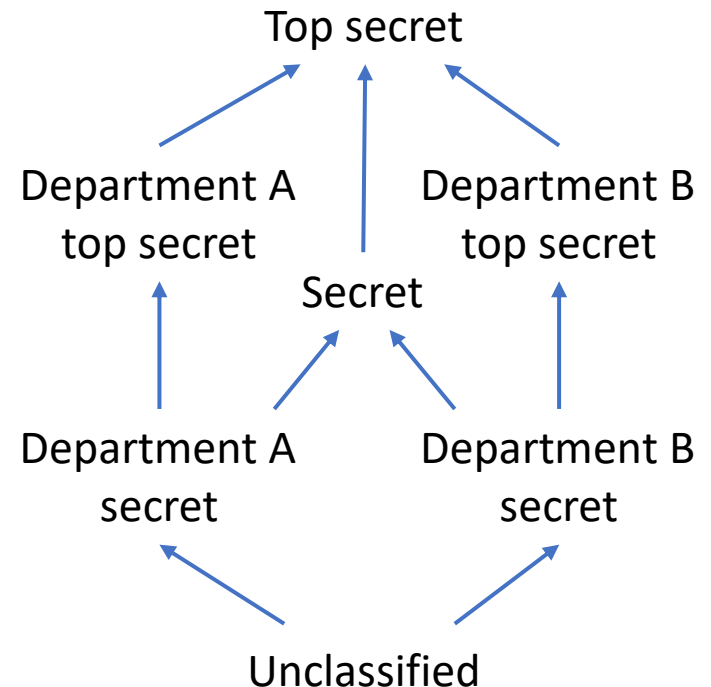
*Dorothy E. Denning, “*A Lattice Model of Secure Information Flow*”. CACM 1976.

More Complicated Security Levels

- China classification



- A (possible) business classification



Information Flow Policy

- Restricts how information flows between different security levels

Information Flow Policy

- Restricts how information flows between different security levels
- Noninterference policy*
 - Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables
 - Intuitively, you should not be able to conclude anything about high information by observing low variables



* J. A. Goguen and J. Meseguer, "[Security policies and security models](#)". S&P 1982.

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$

✓ • $x_L = y_L$

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$

✓ • $x_L = y_L$

? • $x_L = y_H$

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$

✓ • $x_L = y_L$

✗ • $x_L = y_H$

? • $x_H = y_L$

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$

✓ • $x_L = y_L$

✗ • $x_L = y_H$

✓ • $x_H = y_L$

? • $x_L = y_L + z_H$

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$

✓ • $x_L = y_L$

✗ • $x_L = y_H$

✓ • $x_H = y_L$

✗ • $x_L = y_L + z_H$

Noninterference

- Requires the information of high variables **have no effect** on (i.e., should not interfere with) the information of low variables

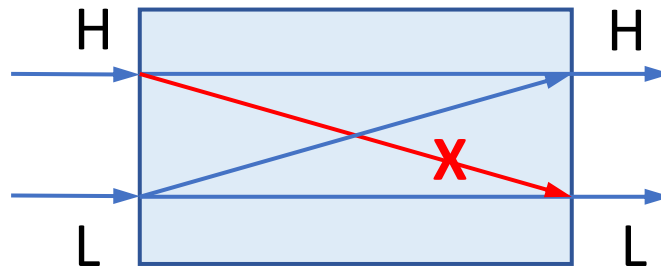
✓ • $x_H = y_H$

✓ • $x_L = y_L$

✗ • $x_L = y_H$

✓ • $x_H = y_L$

✗ • $x_L = y_L + z_H$



Ensures that information flows only **upwards** in the lattice

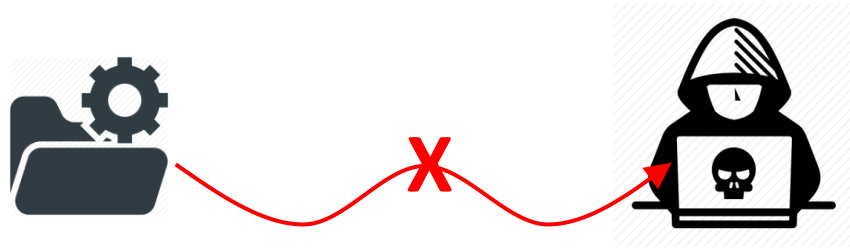


Contents



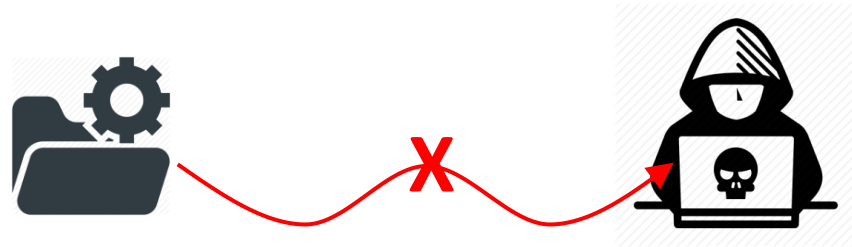
1. Information Flow Security
2. **Confidentiality and Integrity**
3. Explicit Flows and Covert Channels
4. Taint Analysis

- Confidentiality
 - Prevent secret information from being leaked



- Confidentiality

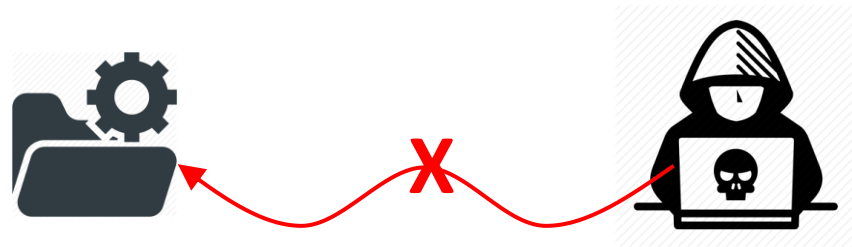
- Prevent secret information from being leaked



Information flow security from another perspective

- Integrity

- Prevent untrusted information from corrupting (trusted) critical information



Integrity

- Prevent untrusted information from corrupting (trusted) critical information¹

```
x = readInput(); // untrusted
cmd = "... " + x;
execute(cmd); // critical (trusted)
```

1. Ken Biba, “*Integrity Considerations for Secure Computer Systems*”. Technical Report, ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.

Integrity

- Prevent untrusted information from corrupting (trusted) critical information¹

```
x = readInput(); // untrusted
cmd = "... " + x;
execute(cmd); // critical (trusted)
```


- Injection errors (#1 cause of vulnerabilities in 2013-2019²)
 - Command injection
 - SQL injection
 - XSS attacks
 - ...

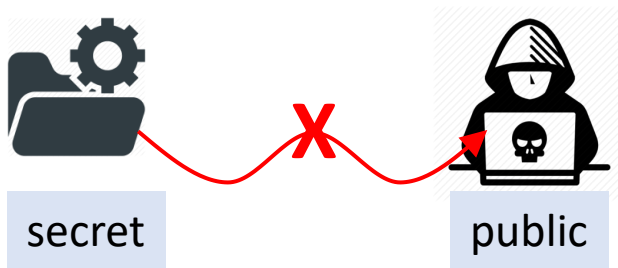
1. Ken Biba, “*Integrity Considerations for Secure Computer Systems*”. Technical Report, ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.
2. National Vulnerability Database, <https://nvd.nist.gov/>

Confidentiality and Integrity




- Security classification

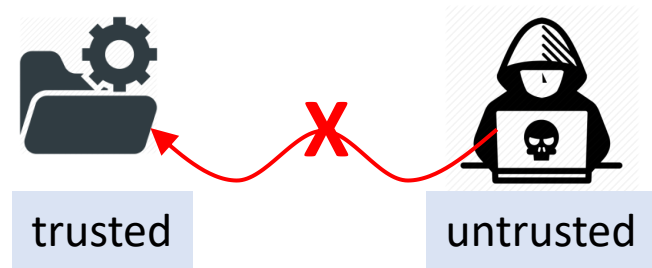
- Secret (**H**igh secret) **H**
 - Public (**L**ow secret) **L**
- 



- Read protection

- Security classification

- Trusted (**H**igh integrity) **L**
 - Untrusted (**L**ow integrity) **H**
- 



- Write protection

Integrity, Broad Definition

- To ensure the correctness, completeness, and consistency of data
- Correctness
 - E.g., for information flow integrity, the (trusted) critical data should not be corrupted by untrusted data
- Completeness
 - E.g., a database system should store all data completely
- Consistency
 - E.g., a file transfer system should ensure that the file contents of both ends (sender and receiver) are identical

Contents



1. Information Flow Security
2. Confidentiality and Integrity
- 3. Explicit Flows and Covert Channels**
4. Taint Analysis

How Does Information Flow

- $x_H = y_H$
- $x_L = y_H$
- $x_L = y_L + z_H$

We have seen how information flows through direct copying. This is called **explicit flow**.

How Does Information Flow

- $x_H = y_H$
- $x_L = y_H$
- $x_L = y_L + z_H$

We have seen how information flows through direct copying. This is called **explicit flow**.

Is this the only way of information flow?

Does Secret Information Leak?

```
secretH = getSecret();  
if (secretH < 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Does Secret Information Leak?

```
secretH = getSecret();  
if (secretH < 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Leak, we can conclude if secret is negative or not by observing publik

Implicit Flows

```
secretH = getSecret();  
if (secretH < 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Leak, we can conclude if secret is negative or not by observing publik

- This kind of information flow is called **implicit flow**, which may arise when the control flow is affected by secret information.
- Any differences in side effects under **secret control** encode information about the control, which may be **publicly observable** and leak secret information.

Implicit Flows

```
secretH = getSecret();  
if (secretH < 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Leak, we can conclude if secret is negative or not by observing publik

Are there *any other* kinds of information flows?

- This kind of information flow is called **implicit flow**, which may arise when the control flow is affected by secret information.
- Any differences in side effects under **secret control** encode information about the control, which may be **publicly observable** and leak secret information.

Does Secret Information Leak?

? `while (secretH < 0) { ... };`

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

?

```
if (secretH < 0)  
    for (int i = 0; i < 1000000; ++i) { ... };
```

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

```
if (secretH < 0)  
    for (int i = 0; i < 1000000; ++i) { ... };
```

Leak, we can conclude that secret is negative if the program execution spends more time

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

```
if (secretH < 0)  
    for (int i = 0; i < 1000000; ++i) { ... };
```

Leak, we can conclude that secret is negative if the program execution spends more time



```
if (secretH < 0)  
    throw new Exception("...");
```

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

```
if (secretH < 0)  
  for (int i = 0; i < 1000000; ++i) { ... };
```

Leak, we can conclude that secret is negative if the program execution spends more time

```
if (secretH < 0)  
  throw new Exception("...");
```

Leak, we can conclude that secret is negative if we observe the exception

Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

```
if (secretH < 0)  
    for (int i = 0; i < 1000000; ++i) { ... };
```

Leak, we can conclude that secret is negative if the program execution spends more time

```
if (secretH < 0)  
    throw new Exception("...");
```

Leak, we can conclude that secret is negative if we observe the exception

```
int saH[] = getSecretArray();  
saH[secretH] = 0;
```



Does Secret Information Leak?

```
while (secretH < 0) { ... };
```

Leak, we can conclude that secret is negative if the program does not terminate

```
if (secretH < 0)
    for (int i = 0; i < 1000000; ++i) { ... };
```

Leak, we can conclude that secret is negative if the program execution spends more time

```
if (secretH < 0)
    throw new Exception("...");
```

Leak, we can conclude that secret is negative if we observe the exception

```
int saH[] = getSecretArray();
saH[secretH] = 0;
```

Leak, exception may reveal that secret is negative

Covert/Hidden Channels

- Mechanisms for signalling information through a computing system are known as ***channels***.
- Channels that exploit a mechanism whose primary purpose is not information transfer are called ***covert channels****.

*Butler W. Lampson, “*A Note on the Confinement Problem*”. CACM 1973.

Covert/Hidden Channels

- Mechanisms for signalling information through a computing system are known as **channels**.
- Channels that exploit a mechanism whose primary purpose is not information transfer are called **covert channels***

- Implicit flows

```
if (secretH < 0) pL = 1; else pL = 0;
```

signal information through the control structure of a program

- Termination channels

```
while (secretH < 0) { ... };
```

signal information through the (non)termination of a computation

- Timing channels

```
if (secretH < 0) for (...) { ... };
```

signal information through the computation time

- Exceptions

```
if (secretH < 0) throw new Exception("...");
```

signal information through the exceptions

- ...

*Butler W. Lampson, “[A Note on the Confinement Problem](#)”. CACM 1973.

Explicit Flows and Covert Channels

- Explicit flows generally carry more information than covert channels, so we focus on **explicit flows**

```
int secretH = getSecret();  
int publikL = secretH;
```

Explicit flow:
transmits **32** bits of information

```
int secretH = getSecret();  
if (secretH % 2 == 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Implicit flow:
transmits **1** bit of information

Explicit Flows and Covert Channels

- Explicit flows generally carry more information than covert channels, so we focus on **explicit flows**

```
int secretH = getSecret();  
int publikL = secretH;
```

Explicit flow:
transmits **32** bits of information

```
int secretH = getSecret();  
if (secretH % 2 == 0)  
    publikL = 1;  
else  
    publikL = 0;
```

Implicit flow:
transmits **1** bit of information

How to **prevent** unwanted information flows, i.e.,
enforce information flow policies?

Contents



1. Information Flow Security
2. Confidentiality and Integrity
3. Explicit Flows and Covert Channels
4. **Taint Analysis**

Taint Analysis

- Taint analysis is the most common information flow analysis. It classifies program data into two kinds:
 - Data of interest, some kinds of labels are associated with the data, called **tainted data**
 - Other data, called untainted data

Taint Analysis

- Taint analysis is the most common information flow analysis. It classifies program data into two kinds:
 - Data of interest, some kinds of labels are associated with the data, called **tainted data**
 - Other data, called untainted data
- Sources of tainted data is called **sources**. In practice, tainted data usually come from the return values of some methods (regarded as sources).

Taint Analysis

- Taint analysis is the most common information flow analysis. It classifies program data into two kinds:
 - Data of interest, some kinds of labels are associated with the data, called **tainted data**
 - Other data, called untainted data
- Sources of tainted data is called **sources**. In practice, tainted data usually come from the return values of some methods (regarded as sources).
- Taint analysis tracks how tainted data flow through the program and observes if they can flow to locations of interest (called **sinks**). In practice, sinks are usually some sensitive methods.



Taint Analysis: Two Applications

- Confidentiality

- Source: source of secret data
- Sink: leakage
- Information leaks

```
x = getPassword(); // source  
y = x;  
log(y); // sink
```

- Integrity

- Source: source of untrusted data
- Sink: critical computation
- Injection errors

```
x = readInput(); // source  
cmd = "... " + x;  
execute(cmd); // sink
```

Taint analysis can detect **both** unwanted information flows

Taint Analysis

- “Can tainted data flow to a sink?”

Taint Analysis

- “Can tainted data flow to a sink?”

Or, in another way

- “Which tainted data a pointer (at a sink) can point to?”

Taint and Pointer Analysis, Together*

The essence of **taint analysis**/**pointer analysis** is to track how **tainted data**/**abstract objects** flow through the program

- Treats tainted data as (artificial) objects
- Treats sources as allocation sites (of tainted data)
- Leverages pointer analysis to propagate tainted data

*Neville Grech and Yannis Smaragdakis, “*P/Taint: Unified Points-to and Taint Analysis*”. OOPSLA 2017.

Domains and Notations

Variables: $x, y \in V$

Fields: $f, g \in F$

Objects: $o_i, o_j \in O$

Tainted data: $t_i, t_j \in T \subset O$

Instance fields: $o_i.f, o_j.g \in O \times F$

Pointers: $\text{Pointer} = V \cup (O \times F)$

Points-to relations: $pt : \text{Pointer} \rightarrow \mathcal{P}(O)$

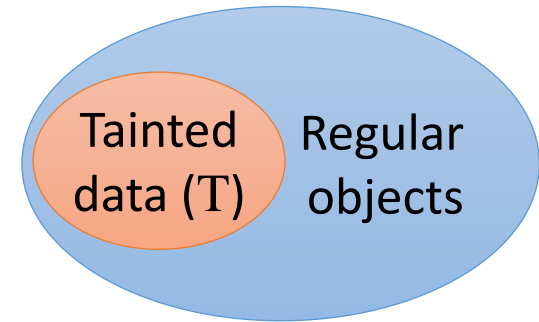
- t_i denotes the tainted data from call site i
- $\mathcal{P}(O)$ denotes the powerset of O
- $pt(p)$ denotes the points-to set of p

Domains and Notations

Variables: $x, y \in V$

Fields: $f, g \in F$

Objects: $o_i, o_j \in O$



Tainted data: $t_i, t_j \in T \subset O$ All objects (O)

Instance fields: $o_i.f, o_j.g \in O \times F$

Pointers: $\text{Pointer} = V \cup (O \times F)$

Points-to relations: $pt : \text{Pointer} \rightarrow \mathcal{P}(O)$

- t_i denotes the tainted data from call site i
- $\mathcal{P}(O)$ denotes the powerset of O
- $pt(p)$ denotes the points-to set of p

Taint Analysis: Inputs & Outputs

- Inputs

- *Sources*: a set of source methods (the calls to these methods return tainted data)
- *Sinks*: a set of sink methods (that tainted data flow to these methods violates security polices)

- Outputs

- *TaintFlows*: a set of pairs of tainted data and sink methods
E.g., $\langle t_i, m \rangle \in \textit{TaintFlows}$ denotes that the tainted data from call site i (which calls a source method) may flow to sink method m

Rules: Call

- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	$L: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG}{m \in \text{Sources}}$ $t_l \in pt(r)$

Rules (Same as Pointer Analysis)

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\overline{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x), o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x), o_j \in pt(o_i.f)}{o_j \in pt(y)}$
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{\begin{array}{l} o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\ o_u \in pt(a_j), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{o_i \in pt(m_{this})}$ $\frac{\phantom{\frac{\begin{array}{l} o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\ o_u \in pt(a_j), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{o_i \in pt(m_{this})}}}{o_u \in pt(m_{pj}), 1 \leq j \leq n}$ $o_v \in pt(r)$

Propagate objects and tainted data

Rules: Call

- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	$L: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \mathbf{Sources}}{t_l \in pt(r)}$

- Handles sinks (generates taint flow information)

Kind	Statement	Rule
Call	$L: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \mathbf{Sinks} \quad \exists i, 1 \leq i \leq n: t_j \in pt(a_i)}{\langle t_j, m \rangle \in \mathbf{TaintFlows}}$

Taint Analysis: An Example

```
1 void main() {
2     A x = new A();
3     String pw = getPassword();
4     A y = x;
5     x.f = pw;
6     String s = y.f;
7     log(s);
8 }
9 String getPassword() {
10    ...
11    return new String(...);
12 }
13 class A {
14     String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Taint Analysis: An Example

```
1 void main() {  
2   A x = new A();  
3   String pw = getPassword();  
4   A y = x;  
5   x.f = pw;  
6   String s = y.f;  
7   log(s);  
8 }  
9 String getPassword() {  
10  ...  
11  return new String(...);  
12 }  
13 class A {  
14   String f;  
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
x	O_2

Taint Analysis: An Example

```
1 void main() {
2   A x = new A();
3   String pw = getPassword();
4   A y = x;
5   x.f = pw;
6   String s = y.f;
7   log(s);
8 }
9 String getPassword() {
10  ...
11  return new String(...);
12 }
13 class A {
14   String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
x	o_2
pw	o_{11}

Taint Analysis: An Example

```

1 void main() {
2     A x = new A();
3     String pw = getPassword();
4     A y = x;
5     x.f = pw;
6     String s = y.f;
7     log(s);
8 }
9 String getPassword() {
10    ...
11    return new String(...);
12 }

```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
x	o_2
pw	o_{11}, t_3

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \text{Sources}}{t_l \in pt(r)}$

Taint Analysis: An Example

```
1 void main() {
2     A x = new A();
3     String pw = getPassword();
4     A y = x;
5     x.f = pw;
6     String s = y.f;
7     log(s);
8 }
9 String getPassword() {
10    ...
11    return new String(...);
12 }
13 class A {
14     String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
x	o_2
pw	o_{11}, t_3
y	o_2

Taint Analysis: An Example

```
1 void main() {
2     A x = new A();
3     String pw = getPassword();
4     A y = x;
5     x.f = pw;
6     String s = y.f;
7     log(s);
8 }
9 String getPassword() {
10    ...
11    return new String(...);
12 }
13 class A {
14     String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
x	o_2
pw	o_{11}, t_3
y	o_2
Field	Object
$o_2.f$	o_{11}, t_3

Taint Analysis: An Example

```
1 void main() {
2     A x = new A();
3     String pw = getPassword();
4     A y = x;
5     x.f = pw;
6     String s = y.f;
7     log(s);
8 }
9 String getPassword() {
10    ...
11    return new String(...);
12 }
13 class A {
14     String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

Variable	Object
<i>x</i>	o_2
<i>pw</i>	o_{11}, t_3
<i>y</i>	o_2
<i>s</i>	o_{11}, t_3
Field	Object
$o_2.f$	o_{11}, t_3

Taint Analysis: An Example

```

1 void main() {
2   A x = new A();
3   String pw = getPassword();
4   A y = x;
5   x.f = pw;
6   String s = y.f;
7   log(s);
8 }
9 String getPassword() {
10  ...
11  return new String(...);
12 }

```

Sources: { getPassword() }

Sinks: { log(String) }

TaintFlows: { $\langle t_3, \text{log(String)} \rangle$ }

Variable	Object
x	o_2
pw	o_{11}, t_3
y	o_2
s	o_{11}, t_3
Field	Object
$o_2.f$	o_{11}, t_3

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \text{Sinks} \quad \exists i, 1 \leq i \leq n: t_j \in pt(a_i)}{\langle t_j, m \rangle \in \text{TaintFlows}}$

Taint Analysis: An Example

```
1 void main() {
2   A x = new A();
3   String pw = getPassword();
4   A y = x;
5   x.f = pw;
6   String s = y.f;
7   log(s);
8 }
9 String getPassword() {
10  ...
11  return new String(...);
12 }
13 class A {
14   String f;
15 }
```

Sources: { getPassword() }

Sinks: { log(String) }

TaintFlows: { $\langle t_3, \text{log(String)} \rangle$ }

Variable	Object
x	o_2
pw	o_{11}, t_3
y	o_2
s	o_{11}, t_3
Field	Object
$o_2.f$	o_{11}, t_3

The X You Need To Understand in This Lecture

- Concept of information flow security
- Confidentiality and integrity
- Explicit flows and covert channels
- Use taint analysis to detect unwanted information flow

注意注意!
划重点了!



软件分析

南京大学

计算机科学与技术系

程序设计语言与

静态分析研究组

李棣
谭添