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

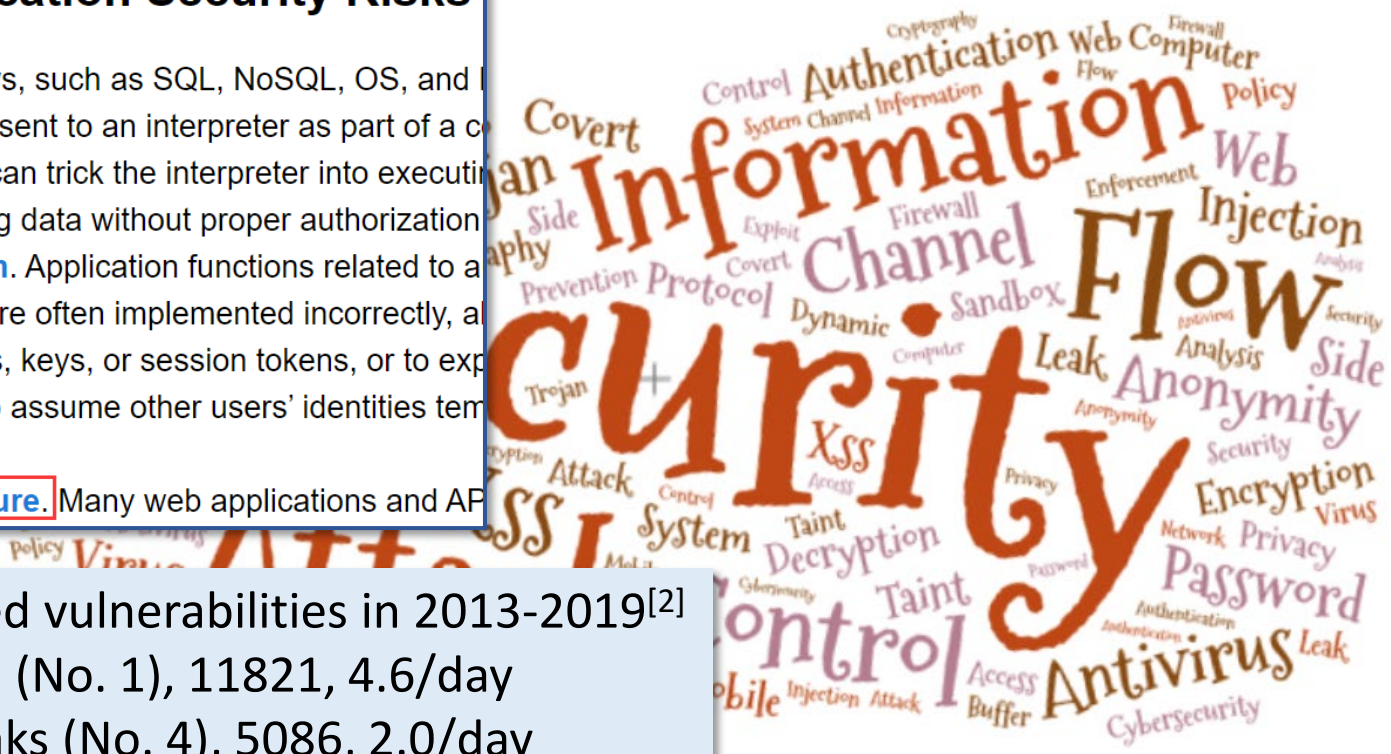
Becomes increasingly important nowadays

Computer Security

Becomes increasingly important nowadays

Top 10 Web Application Security Risks [1]

1. **Injection**. Injection flaws, such as SQL, NoSQL, OS, and l when untrusted data is sent to an interpreter as part of a c attacker's hostile data can trick the interpreter into executi commands or accessing data without proper authorization
2. **Broken Authentication**. Application functions related to a session management are often implemented incorrectly, al compromise passwords, keys, or session tokens, or to exp implementation flaws to assume other users' identities tem permanently.
3. **Sensitive Data Exposure**. Many web applications and AP



Causes of exploited vulnerabilities in 2013-2019[2]

- Injection errors (No. 1), 11821, 4.6/day
- Information leaks (No. 4), 5086, 2.0/day

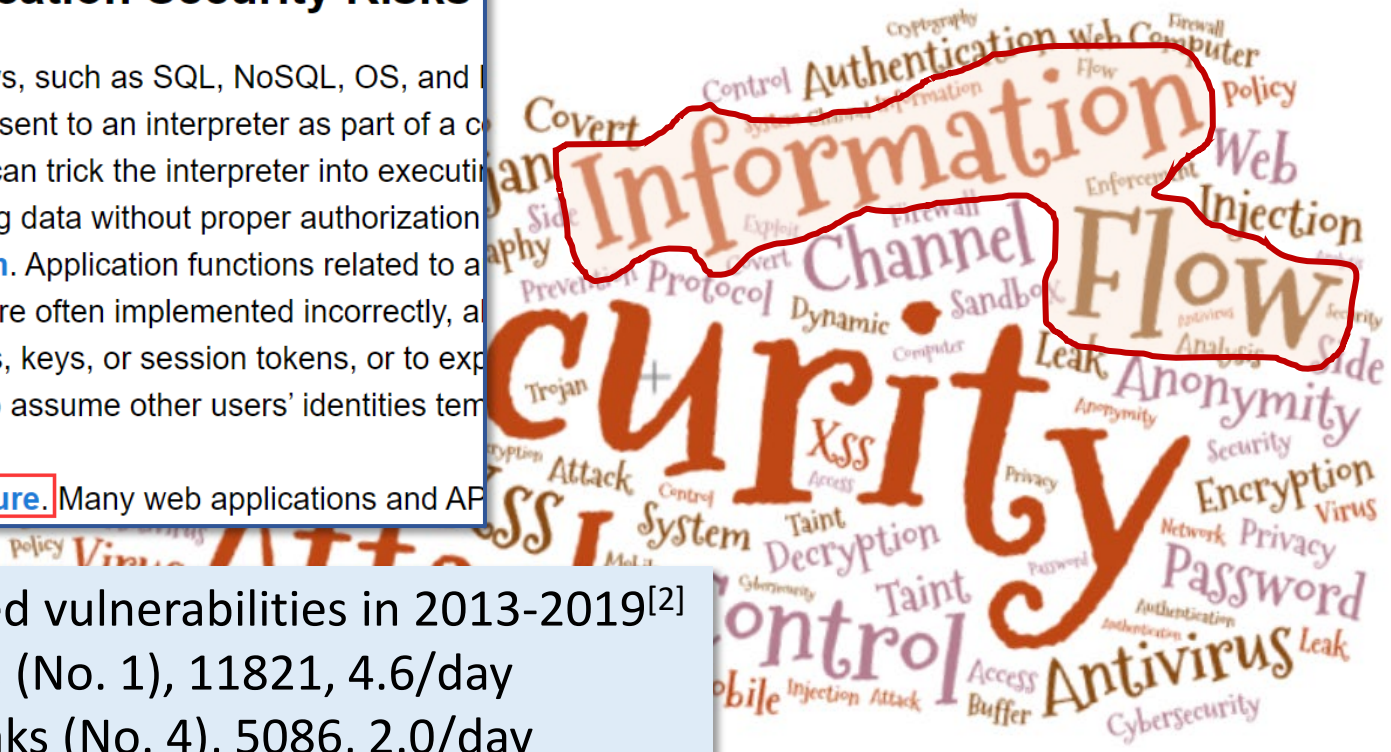
[1] The Open Web Application Security Project, https://owasp.org/
[2] National Vulnerability Database, https://nvd.nist.gov/

Computer Security

Becomes increasingly important nowadays

Top 10 Web Application Security Risks [1]

1. **Injection**. Injection flaws, such as SQL, NoSQL, OS, and l when untrusted data is sent to an interpreter as part of a c attacker's hostile data can trick the interpreter into executi commands or accessing data without proper authorization
2. **Broken Authentication**. Application functions related to a session management are often implemented incorrectly, al compromise passwords, keys, or session tokens, or to exp implementation flaws to assume other users' identities tem permanently.
3. **Sensitive Data Exposure**. Many web applications and AP

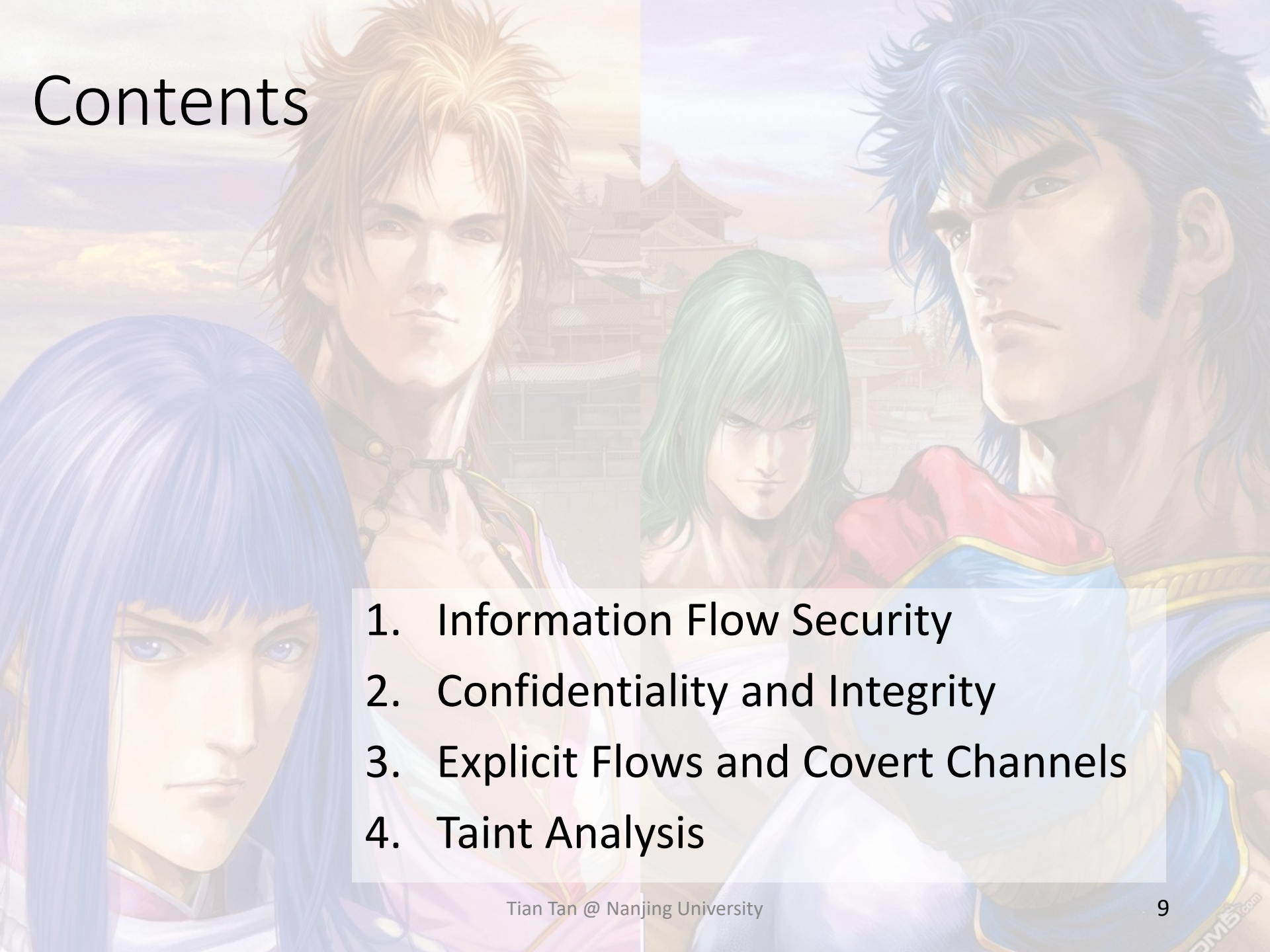


Causes of exploited vulnerabilities in 2013-2019[2]

- Injection errors (No. 1), 11821, 4.6/day
- Information leaks (No. 4), 5086, 2.0/day

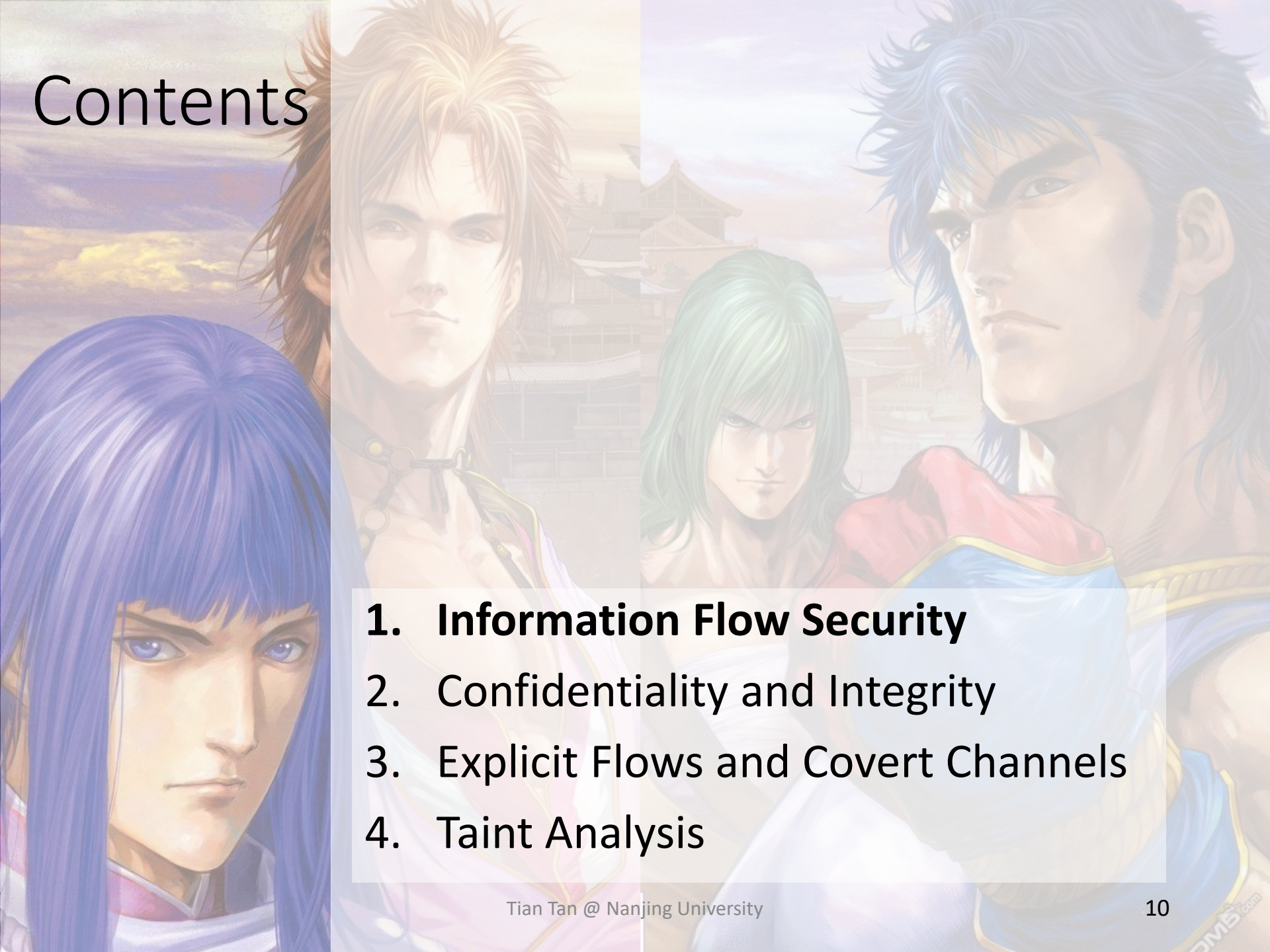
[1] The Open Web Application Security Project, https://owasp.org/
[2] National Vulnerability Database, https://nvd.nist.gov/

Contents



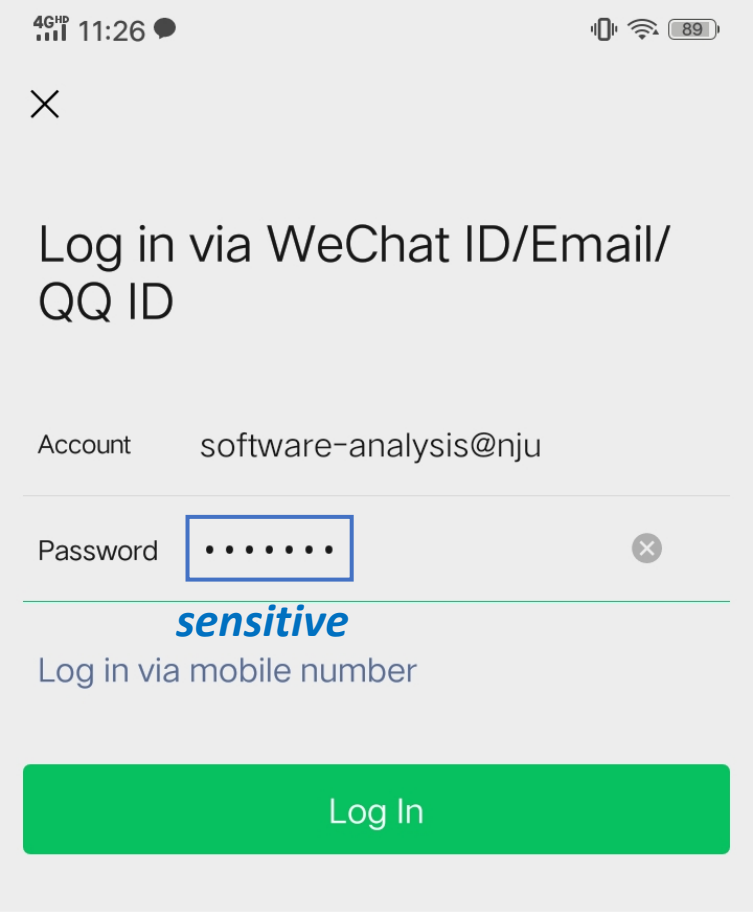
1. Information Flow Security
2. Confidentiality and Integrity
3. Explicit Flows and Covert Channels
4. Taint Analysis

Contents

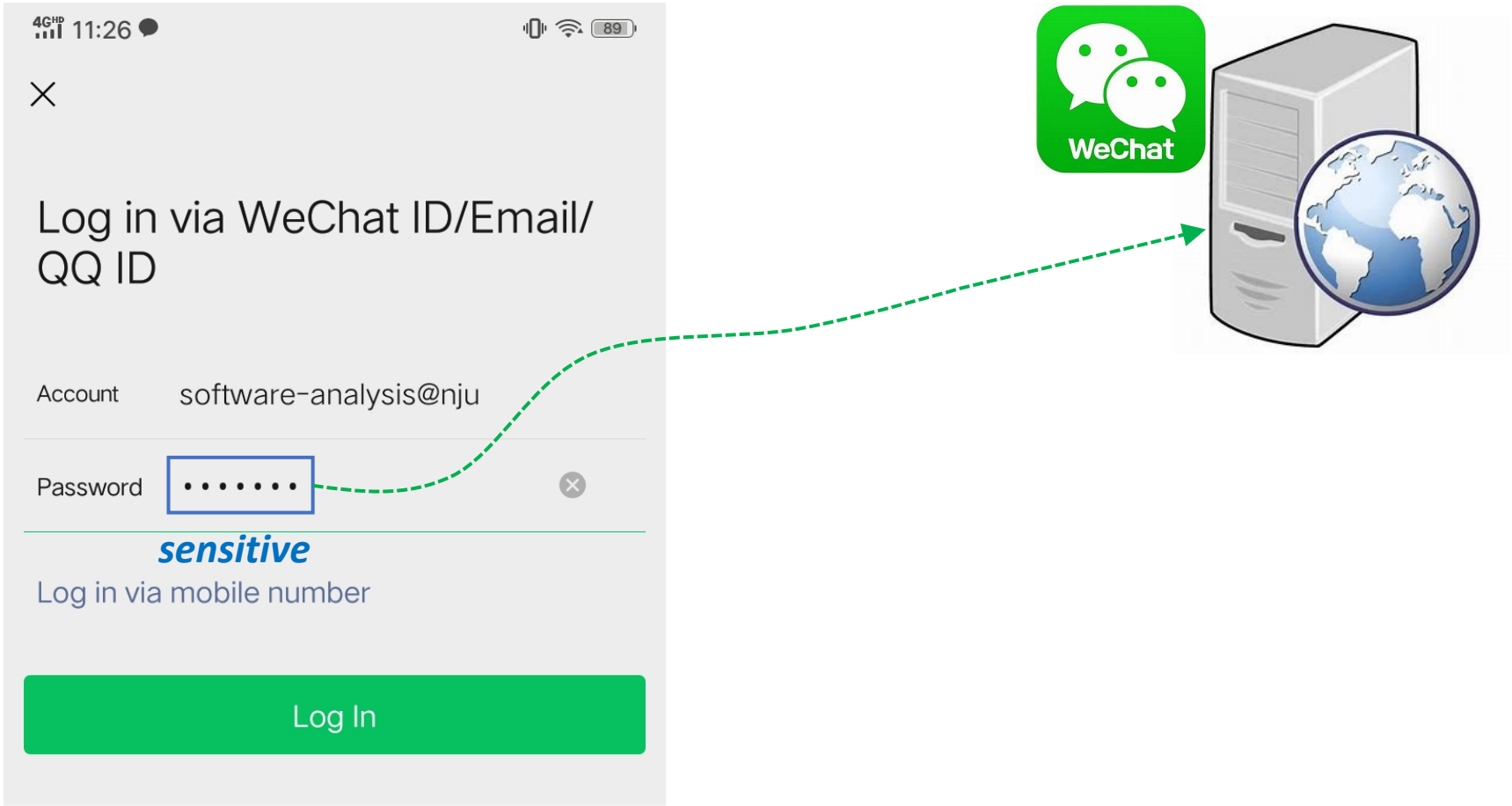


1. **Information Flow Security**
2. Confidentiality and Integrity
3. Explicit Flows and Covert Channels
4. Taint Analysis

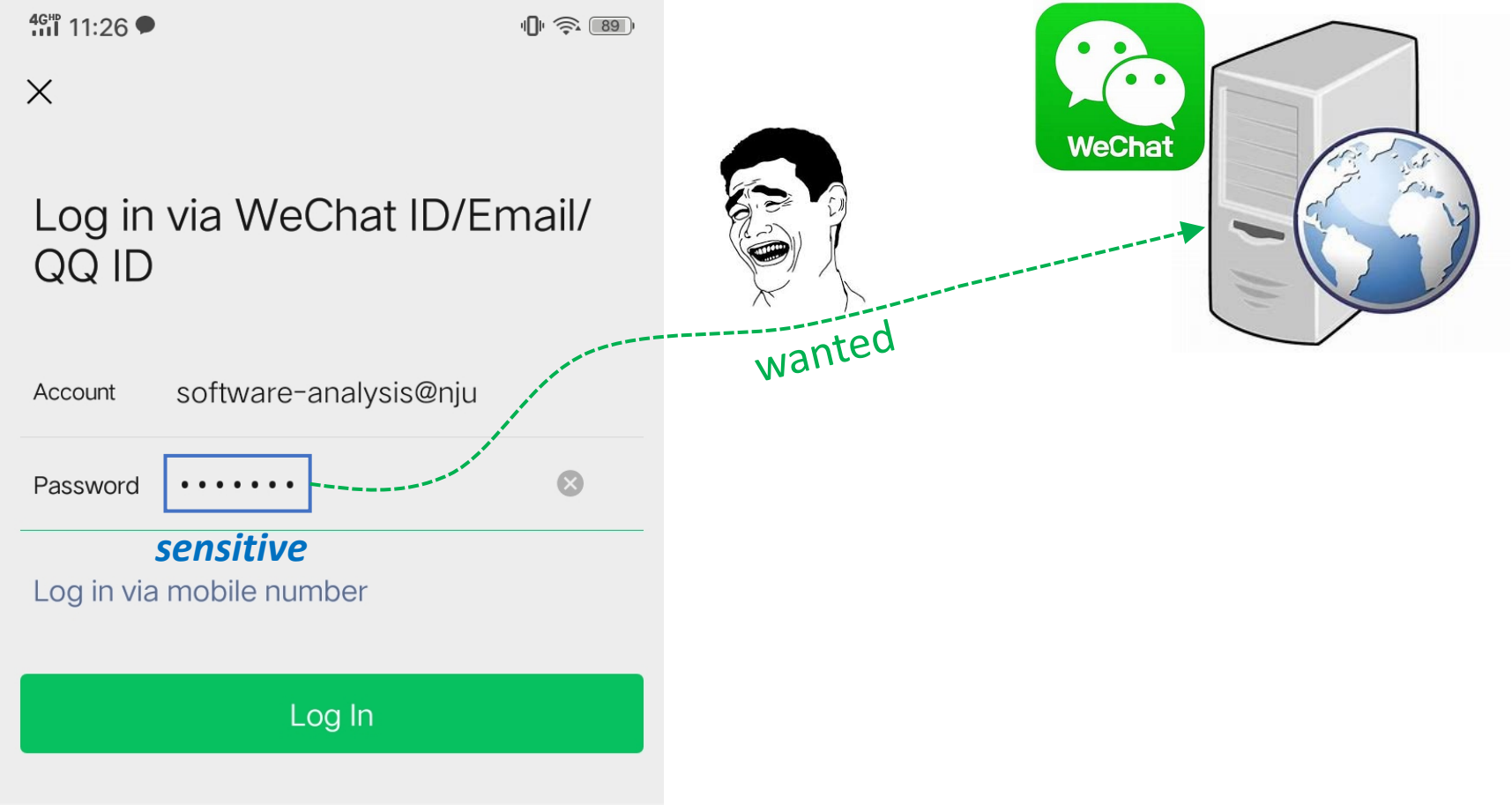
Information Flow: An Example



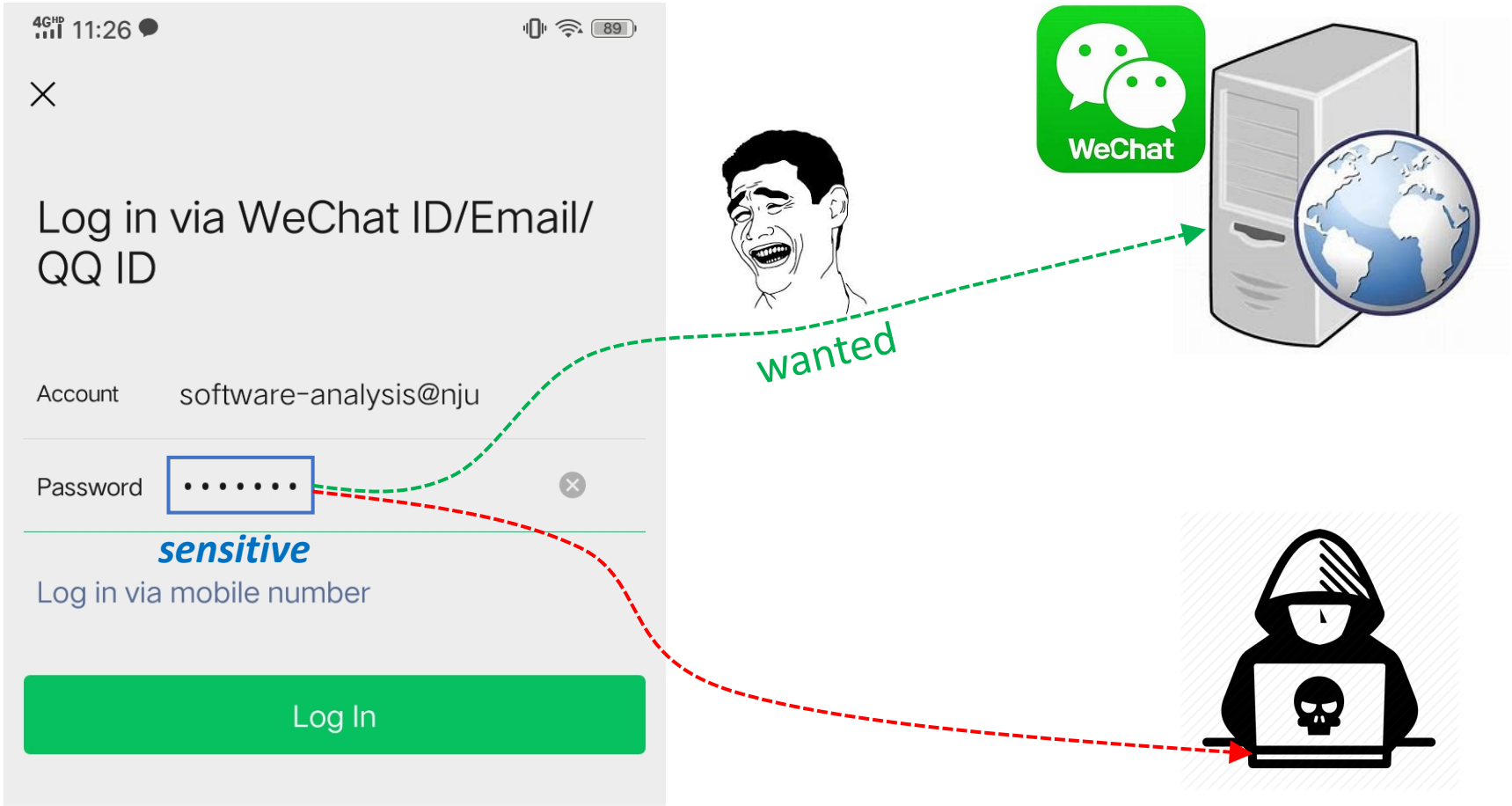
Information Flow: An Example



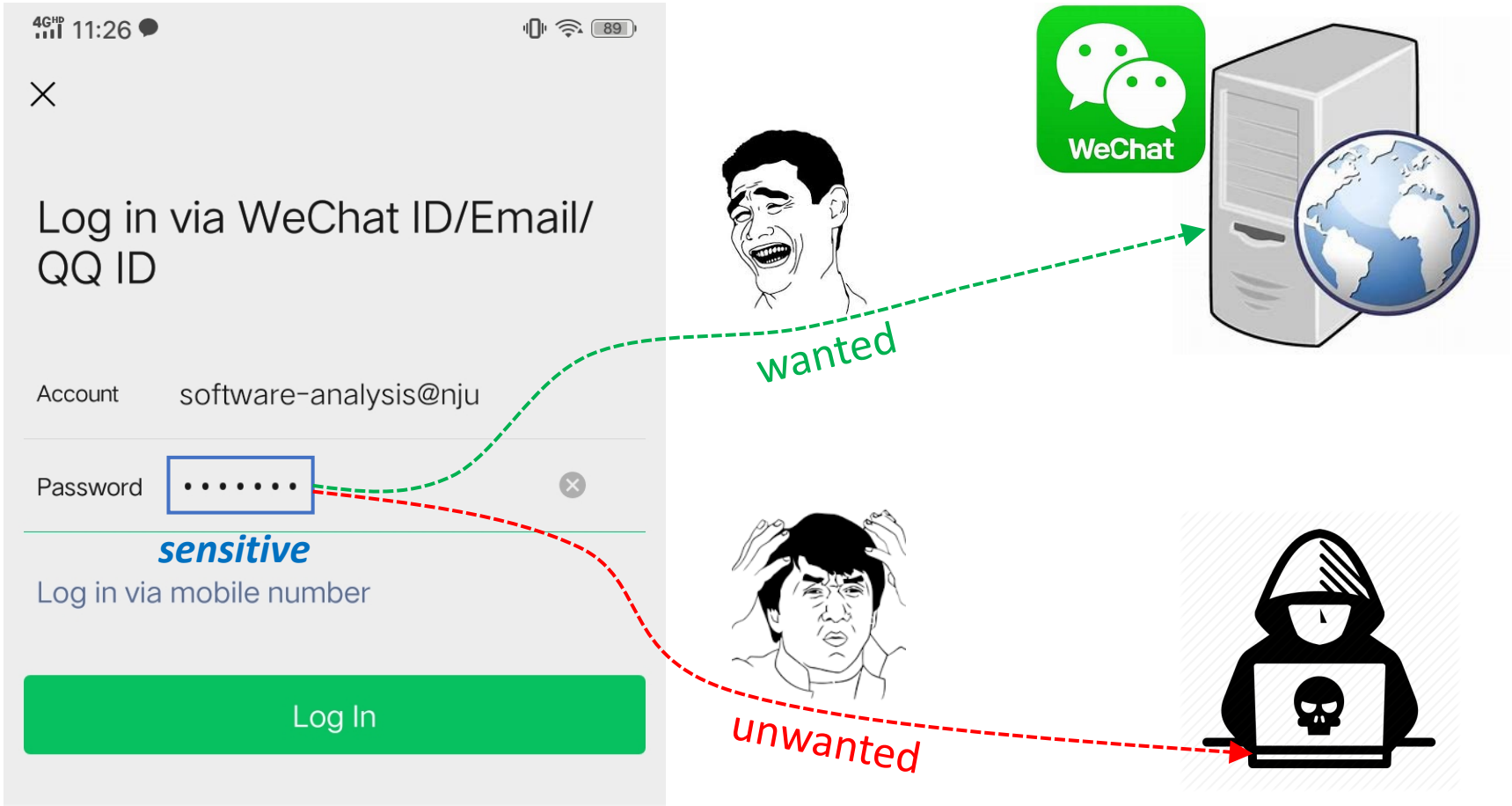
Information Flow: An Example



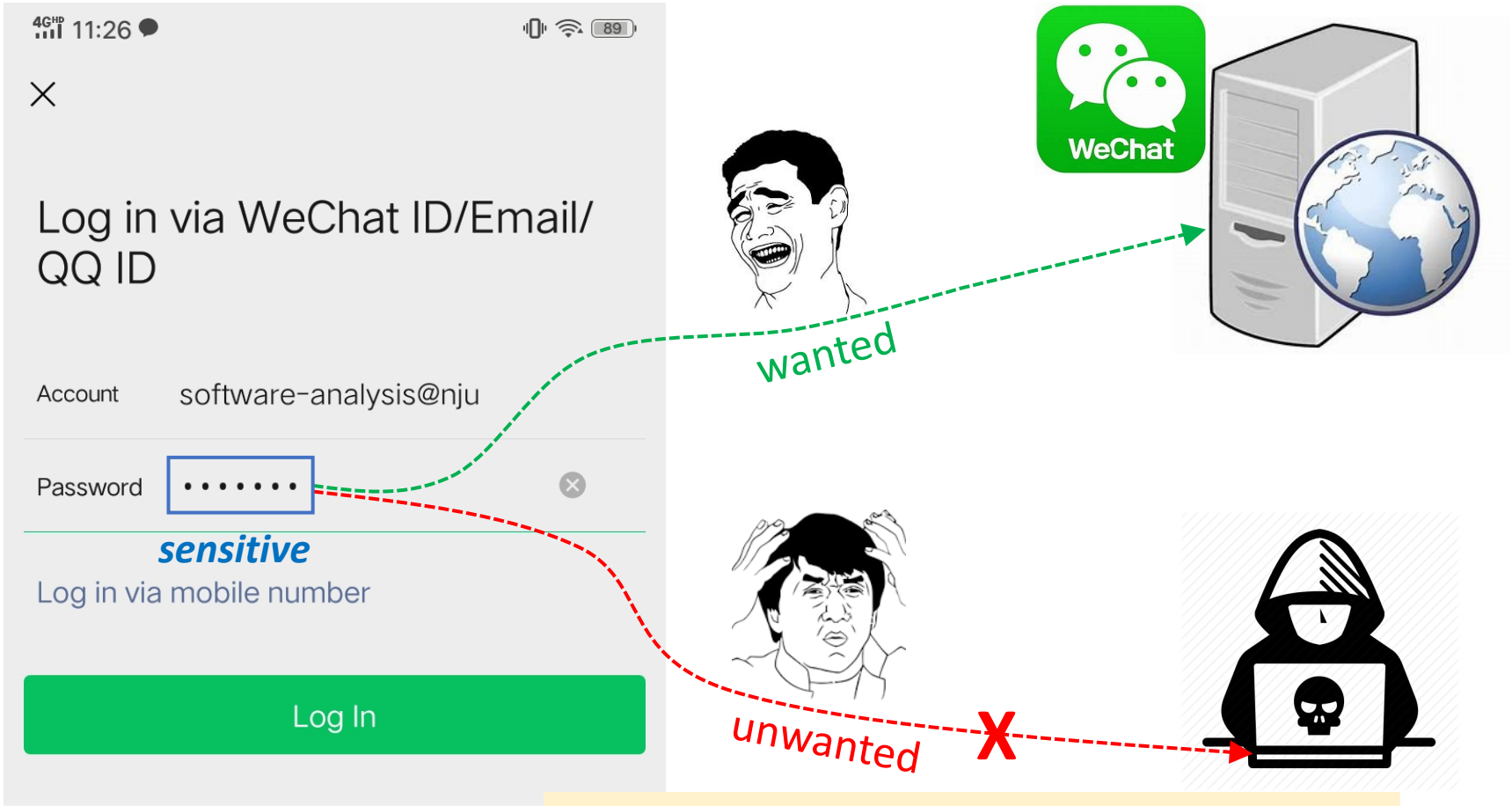
Information Flow: An Example



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 $x \longrightarrow y$

- 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*

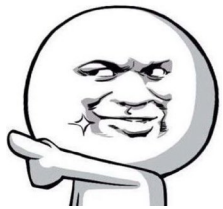
- Information flow: if the information in variable x is transferred to variable y, then there is information flow $x \longrightarrow y$

- Examples


```
y = x;
```

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

Looks like pointer analysis?
You 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

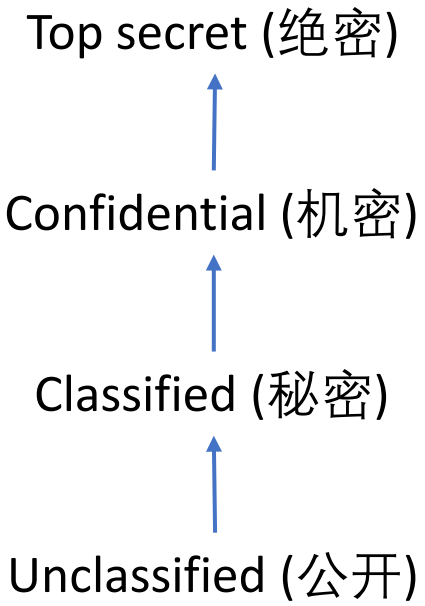
```
h = getPassword(); // h is high security
broadcast(l);      // l is low security
```

- Security levels can be modeled as *lattice**
  - L ≤ H

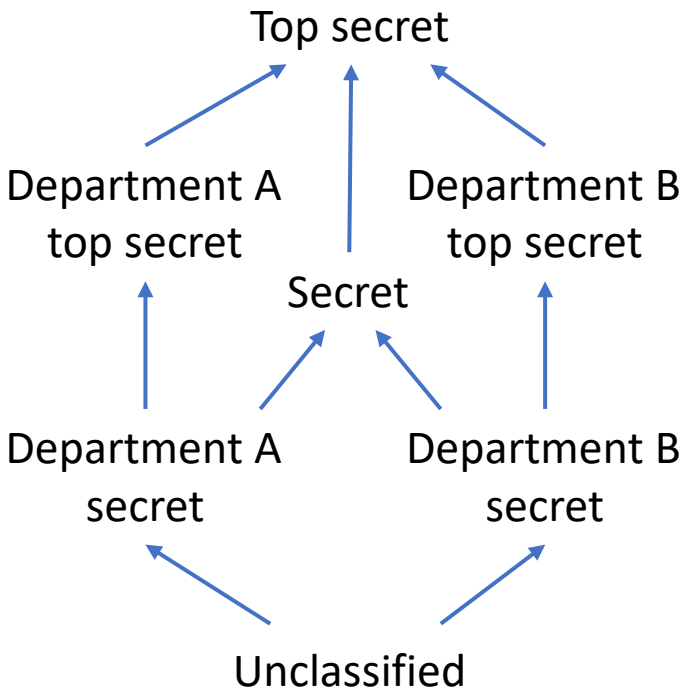
*Dorothy E. Denning, "*A Lattice Model of Secure Information Flow*". CACM 1976.

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

✓ • $x_H = y_H$

✓ • $x_L = y_L$

✗ • $x_L = y_H$

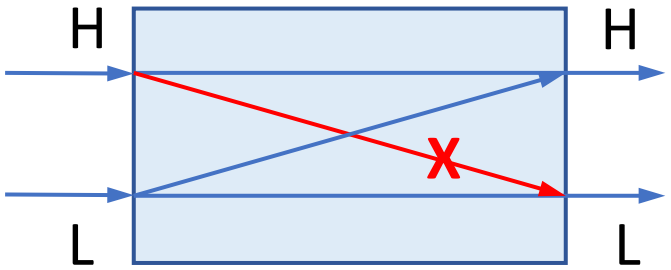
✓ • $x_H = y_L$

✗ • $x_L = y_L + z_H$

Noninterference

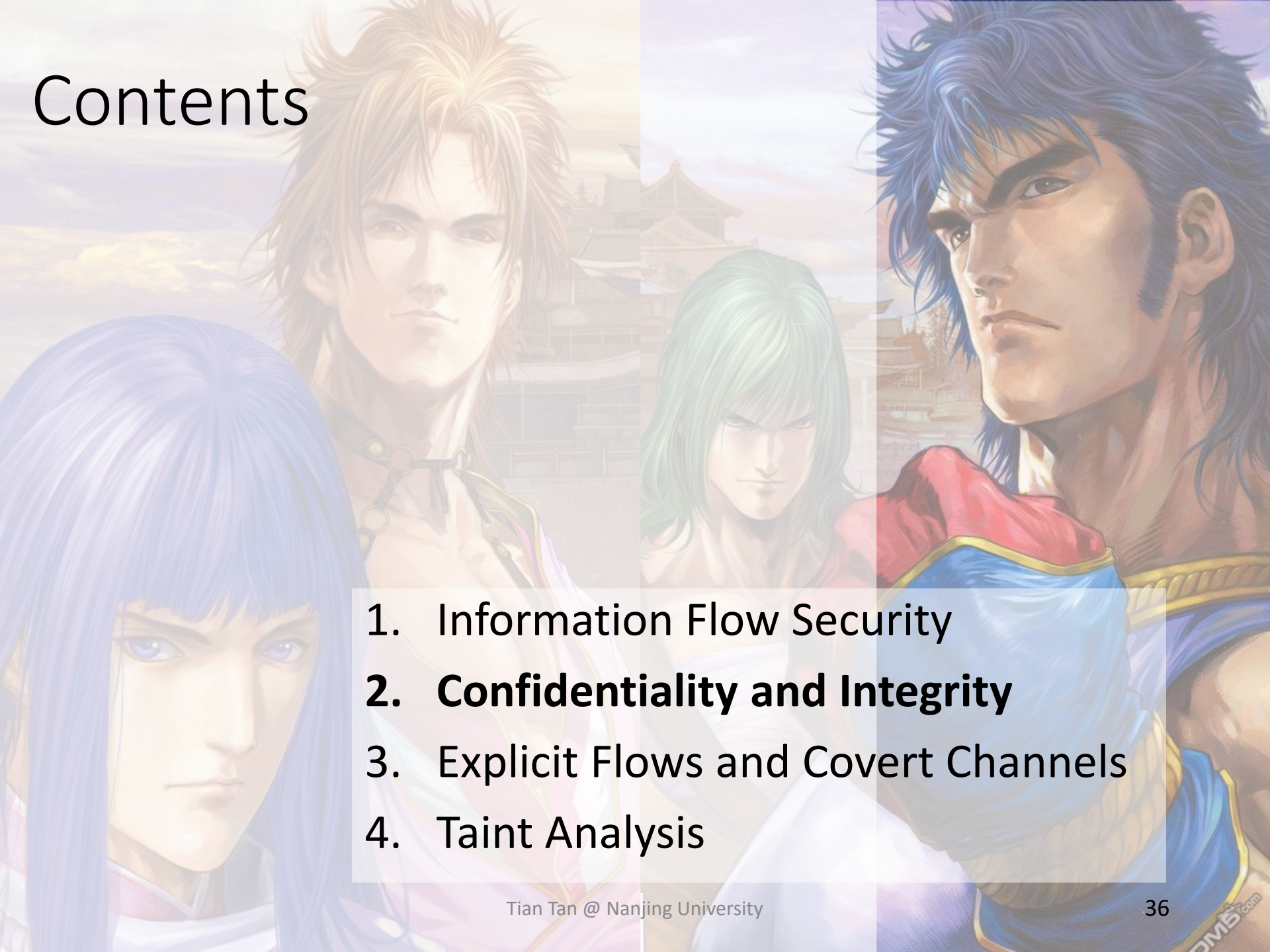
- Requires the information of high variables have no effect on (i.e., should not interfere with) the information of low variables

✓ • $x_H = y_H$
✓ • $x_L = y_L$
✗ • $x_L = y_H$
✓ • $x_H = y_L$
✗ • $x_L = y_L + z_H$



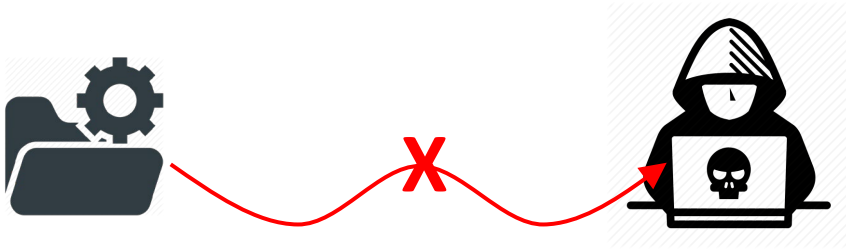
Ensures that information flows only upwards in the lattice

Contents

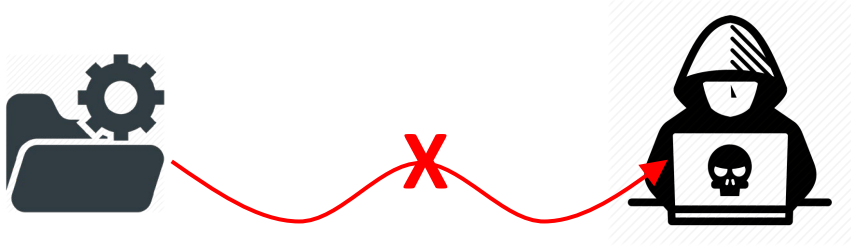


1. Information Flow Security
2. **Confidentiality and Integrity**
3. Explicit Flows and Covert Channels
4. Taint Analysis

- Confidentiality
  - Prevent secret information from being leaked

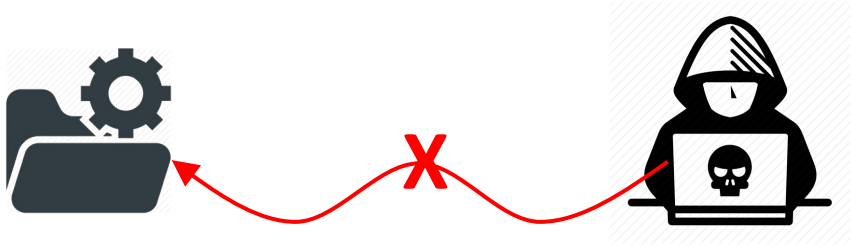


- Confidentiality
  - Prevent secret information from being leaked



Information flow security from another perspective

- Integrity
  - Prevent untrusted information from corrupting (trusted) critical information



Integrity

- Prevent untrusted information from corrupting (trusted) critical information[1]

```
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, Bed-ford, MA, 1977.

Integrity

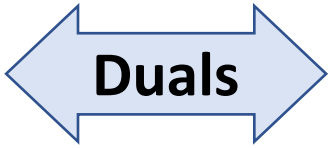
- Prevent untrusted information from corrupting (trusted) critical information[1]

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

- Injection errors (#1 cause of vulnerabilities in 2013-2019[2])
  - Command injection
  - SQL injection
  - XSS attacks
  - …

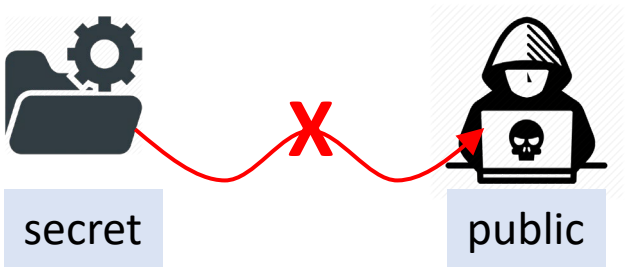
1. Ken Biba, "*Integrity Considerations for Secure Computer Systems*". Technical Report, ESD-TR-76-372, USAF Electronic Systems Division, Bed-ford, MA, 1977.
2. National Vulnerability Database, https://nvd.nist.gov/

Confidentiality and Integrity

Confidentiality  **Integrity**

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

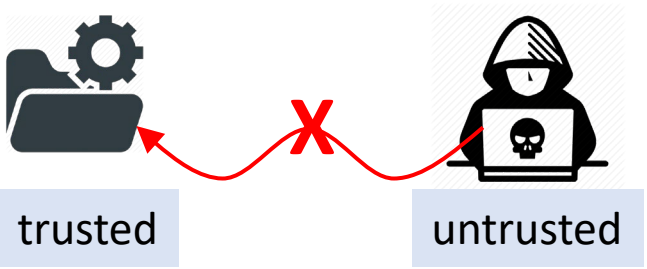
H
↑
L



- Read protection

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

L
↑
H



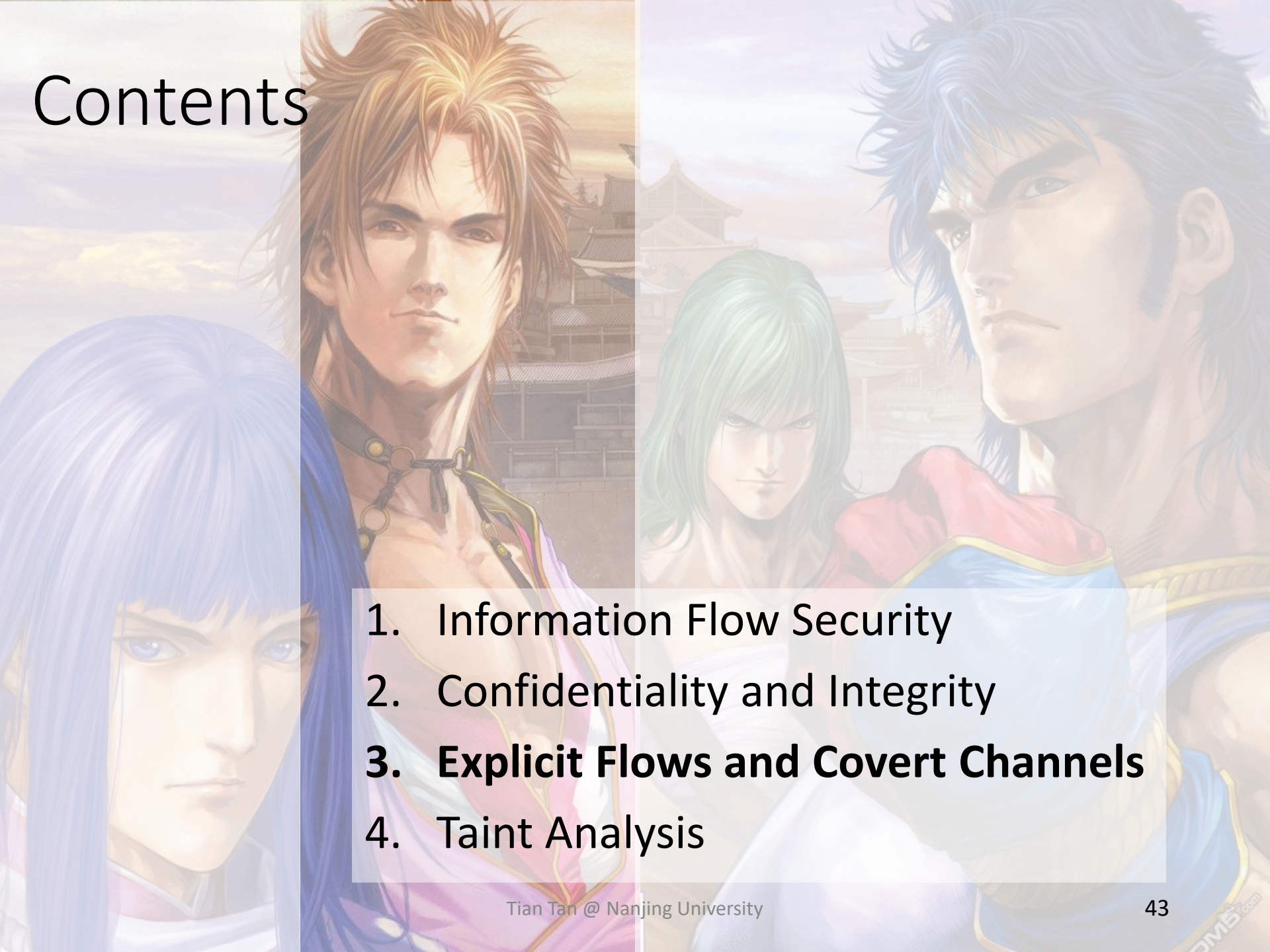
- Write protection

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?

```
secret_H = getSecret();
if (secret_H < 0)
    publik_L = 1;
else
    publik_L = 0;
```

Does Secret Information Leak?

```
secret_H = getSecret();
if (secret_H < 0)
    publik_L = 1;
else
    publik_L = 0;
```

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

Implicit Flows

```
secret_H = getSecret();
if (secret_H < 0)
    publik_L = 1;
else
    publik_L = 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

Are there *any other* kinds of information flows?

```
secret_H = getSecret();
if (secret_H < 0)
    publik_L = 1;
else
    publik_L = 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.

Does Secret Information Leak?

? `while (secret_H < 0) { … };`

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

?

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

```
if (secret_H < 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 (secret_H < 0) { … };
```

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

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```

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

?

```
if (secret_H < 0)
  throw new Exception("…");
```

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```

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

```
if (secret_H < 0)
  throw new Exception("…");
```

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

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```

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

```
if (secret_H < 0)
  throw new Exception("…");
```

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

?

```
int sa_H[] = getSecretArray();
sa_H[secret_H] = 0;
```

Does Secret Information Leak?

```
while (secret_H < 0) { … };
```

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

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```

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

```
if (secret_H < 0)
  throw new Exception("…");
```

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

```
int sa_H[] = getSecretArray();
sa_H[secret_H] = 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 (secret_H < 0) p_L = 1; else p_L = 0;
```

  signal information through the control structure of a program

- Termination channels

```
while (secret_H < 0) { … };
```

  signal information through the (non)termination of a computation

- Timing channels

```
if (secret_H < 0) for (…) { … };
```

  signal information through the computation time

- Exceptions

```
if (secret_H < 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 secret_H = getSecret();
int publik_L = secret_H;
```

Explicit flow:
transmits 32 bits of information

```
int secret_H = getSecret();
if (secret_H % 2 == 0)
    publik_L = 1;
else
    publik_L = 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 secret_H = getSecret();
int publik_L = secret_H;
```

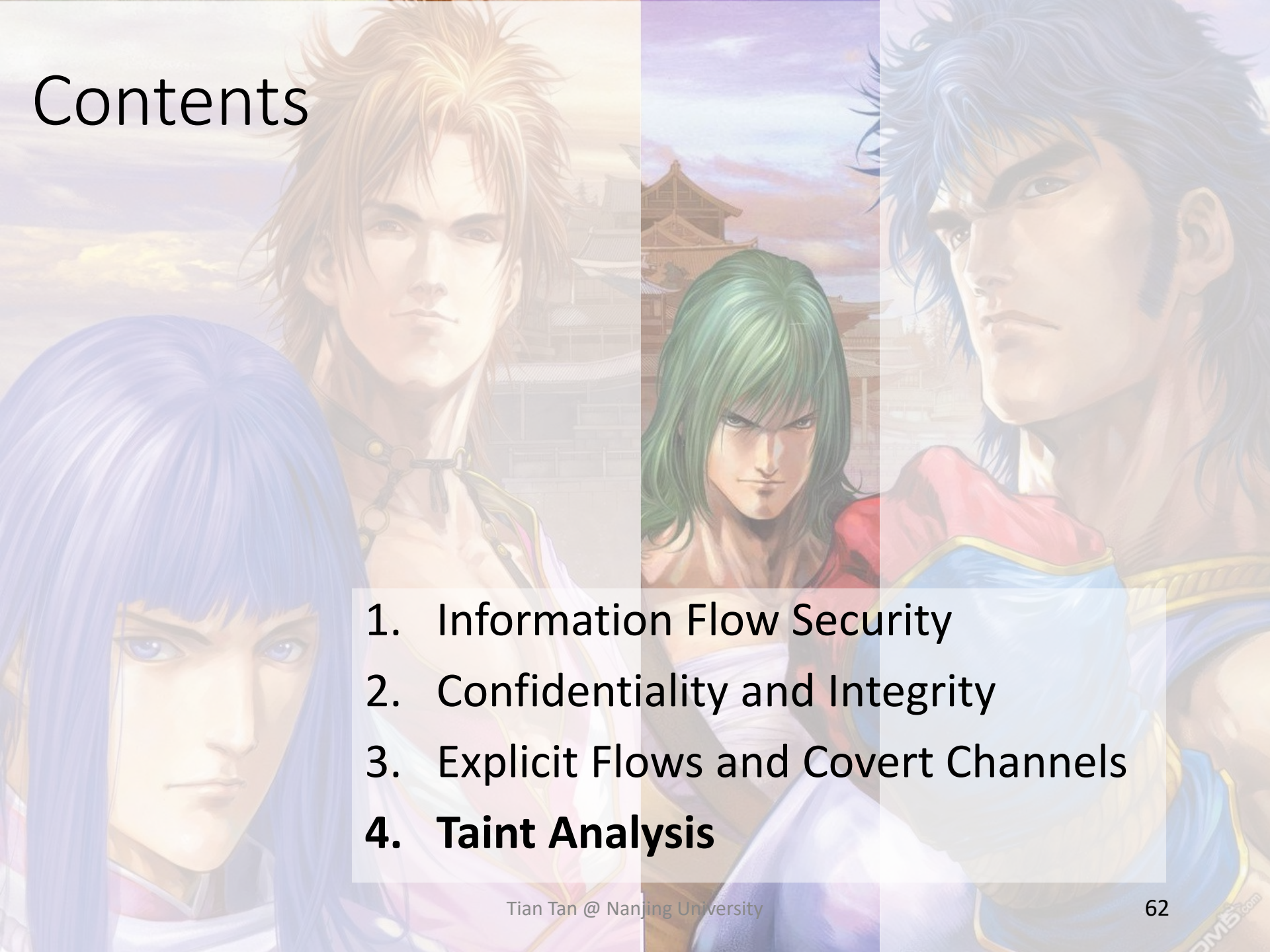
Explicit flow:
transmits 32 bits of information

```
int secret_H = getSecret();
if (secret_H % 2 == 0)
    publik_L = 1;
else
    publik_L = 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:	$x, y \in \mathrm{V}$
Fields:	$f, g \in \mathrm{F}$
Objects:	$o_i, o_j \in \mathrm{O}$
Tainted data:	$t_i, t_j \in \mathrm{T} \subset \mathrm{O}$
Instance fields:	$o_i.f, o_j.g \in \mathrm{O} \times \mathrm{F}$
Pointers:	$\mathrm{Pointer} = \mathrm{V} \cup (\mathrm{O} \times \mathrm{F})$
Points-to relations:	$pt : \mathrm{Pointer} \rightarrow \mathcal{P}(\mathrm{O})$

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

Domains and Notations

Variables: $x, y \in \mathrm{V}$

Fields: $f, g \in \mathrm{F}$

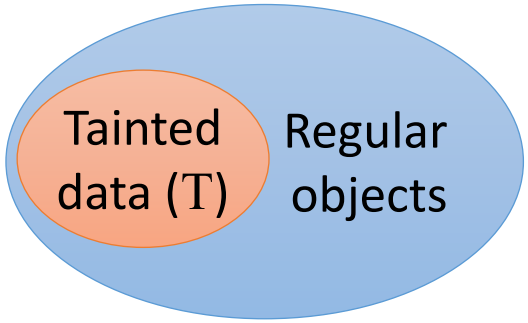
Objects: $o_i, o_j \in \mathrm{O}$

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

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

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

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



- t_i denotes the tainted data from call site i
- $\mathcal{P}(\mathrm{O})$ 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., $\langle t_i, m \rangle \in TaintFlows$ 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(a1,…,an)`	$\frac{l \to m \in CG \quad m \in Sources}{t_l \in pt(r)}$

Rules (Same as Pointer Analysis)

Kind	Statement	Rule
New	`i: x = new T()`	$\dfrac{}{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$
Call	`l: r = x.k(a1,…,an)`	$\dfrac{\begin{array}{c} o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k}) \\ o_u \in pt(aj), 1 \le j \le n \\ o_v \in pt(m_{ret}) \\ o_i \in pt(m_{this}) \\ o_u \in pt(m_{pj}), 1 \le j \le n \end{array}}{o_v \in pt(r)}$

Propagate objects and tainted data

Rules: Call

- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{l \to m \in CG \quad m \in Sources}{t_l \in pt(r)}$

- Handles sinks (generates taint flow information)

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{l \to m \in CG \quad m \in Sinks \quad \exists i, 1 \le i \le n : t_j \in pt(ai)}{\langle t_j, m \rangle \in TaintFlows}$

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(a1,…,an)	$\dfrac{l \rightarrow m \in CG \quad m \in Sources}{t_l \in pt(r)}$

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
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

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

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(a1,…,an)	$\dfrac{l \to m \in CG \quad m \in Sinks \quad \exists i, 1 \le i \le n: t_j \in pt(ai)}{\langle t_j, m\rangle \in TaintFlows}$

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

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

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

李樾 谭添