Internet Security

CS 144 Web Applications
TA: Zijun Xue, Zhehan Li
03/09/2018

Outline

- Introduction to Cryptography
  - Symmetric Key
  - Asymmetric Key (Public-Key Infrastructure)
- Common Web Vulnerability
- Project 5

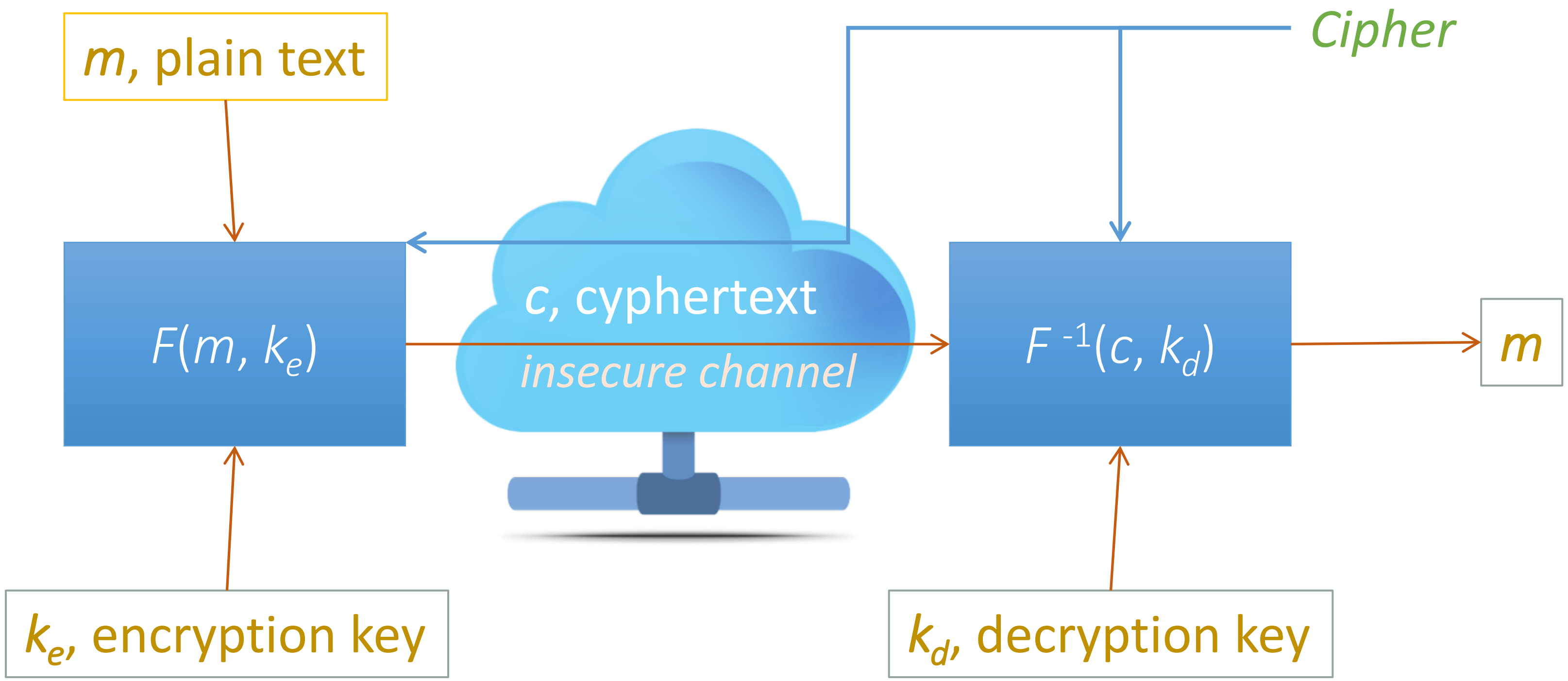
Some common attack techniques

- DDoS (Distributed Denial of service)
- Phishing
  - spoof web site to look like the real one
- Pharming (DNS cache poisoning)
  - e.g. wrong DNS resolution
- Packet sniffing
  - Cache theft
- SQL injection
  - SELECT * FROM users WHERE (name='cs144') and (password='1' OR '1'='1');

Desired Guarantees

- Confidentiality
  - Maintaining the privacy of the “conversation.”
- Message/data integrity
  - No one can modify the content of the messages.
- Authentication
  - Making sure the other party is who he/she claims to be.
- Authorization
  - Managing access to resources upon successful authentication.
    - UNIX User and Group permission mechanism

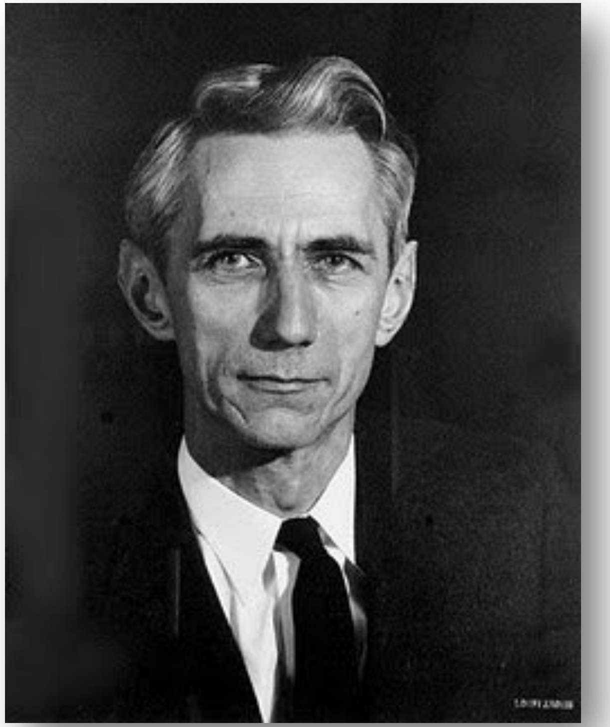
Encryption Algorithm



- Symmetric: $k_e = k_d$
- Asymmetric: $k_e \neq k_d$

Cipher properties

- (*Shannon*) ***Perfect Secrecy***: $P(\ m \mid c\ ) = P(\ m\ )$
  - We should not be able to guess *m* by just looking at the cyphertext *c*.
- *One-Time Pad* (OTP)
  - Guarantees Perfect Secrecy.
  - Use a brand new key to encrypt messages every time.
  - Never use the same key again.
  - Pad value (i.e. key) must be at least as long as the message
  - Very expensive in practical terms.



Symmetric-Key Cryptography

- Same key for encrypting and decrypting: $k_e = k_d = k$
  - XOR is the simplest operator:

m, plain text		*k*, the key		*c*, cyphertext
11110000	$\oplus$	10111001	=	01001001

c, cyphertext		*k*, the key		*m*, plain text
01001001	$\oplus$	10111001	=	11110000

$F = F^{-1}$

Symmetric-Key Cryptography

- **DES**, *Data Encryption Standard*, 64-bit cypher.
- **AES**, *Advanced Encryption Standard*, 128-bit cypher, 128/192/256-bit key.
- How do we ensure ***authentication***?
  - Challenge message (a random number).
- How do we ensure ***confidentiality***?
  - Generate individual key for each conversation.
- How do we ensure ***authorization***?
  - Encrypting distinct data with distinct keys.
- How do we ensure ***integrity***?
  - Encrypting a checksum.
- Any problem with symmetric-key cryptography?
  - Agreeing on key.
  - *n(n-1)/2* keys for *n* parties

Asymmetric-Key Cryptography

- Different keys
  - For encryption: $e = k_e$ (Public Key)
    - $c = F(m, e)$
  - For decryption: $d = k_d$ (Private Key)
    - $m = F^{-1}(c, d)$

- Requirements:
  - $F^{-1}(F(m, e), d) = m$
  - $c = F(m, e) \not\Rightarrow m$ (Perfect Secrecy)
  - $e \not\Rightarrow d$

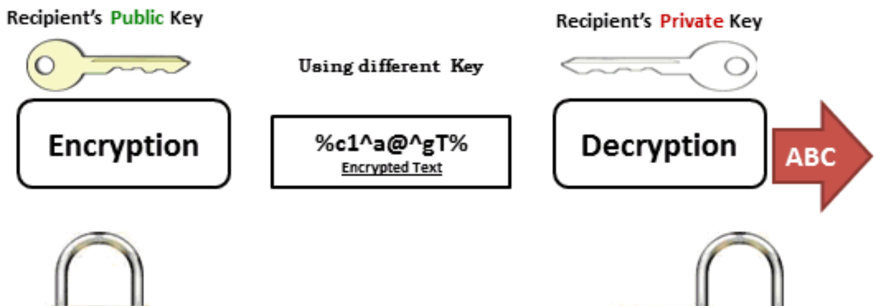


Fig : Asymmetric Encryption

RSA Algorithm

1. Pick two large prime numbers: p and q.
2. Select e such that $1 < e < (p-1)(q-1)$
   - e doesn't have to be random.
   - e is coprime to $(p-1)(q-1)$
3. Solve for d in $de \bmod (p-1)(q-1) = 1$
4. Make $n = pq$ and throw away p and q.
5. The new keys are $k_e = (e, n)$ and $k_d = (d, n)$
   - Encryption $c = F(m, k_e) = m^e \bmod n$
   - Decryption $m = F^{-1}(c, k_d) = c^d \bmod n$

Rivest, Shamir, and Adleman

RSA Properties

- $F^{-1}(F(m, e), d) == m$?
  - $m = m^{ed} \bmod n$
- Can we obtain m from $c = m^e \bmod n$?
  - We know c, e, and n.
  - **RSA problem**
- Can we obtain d from $de \bmod (p - 1)(q - 1) = 1$?
  - We know e, and $n = pq$.
  - **Large-number factorization problem**.

- *RSA is one thousand times slower than symmetric-key cryptography.*

RSA Example

- Let $p = 7$ and $q = 11$
- $e < (6)(10)$
  - $e = 13$, being coprime to p and q
- Solve for d in $13d \bmod 60 = 1$
  - $d = 37$
- $n = (7)(11) = 77$, and throw away p and q
- If $m = 3$, then $c = 3^{13} \bmod 77$
  - $c = 38$
- If $c = 38$, then $m = 38^{37} \bmod 77$
  - $m = 3$

Asymmetric-Key Cryptography Applications

- Confidentiality?
  - Asymmetric-Key Cryptography is very expensive.
  - Use it to agree on a common key, and continue communicating with ***symmetric-key cryptography***.

- Authentication?
  - *A* wants to make sure *B* is not someone else.
  - *A* generates a random number ***r***, and sends ***c*** = $F(r, k_e^B)$ to *B*.
  - *B* decrypts *c*, and sends back ***r'*** = $F(c, k_d^B)$ to *A*.
  - If $r' = r$, then *B* is authentic.

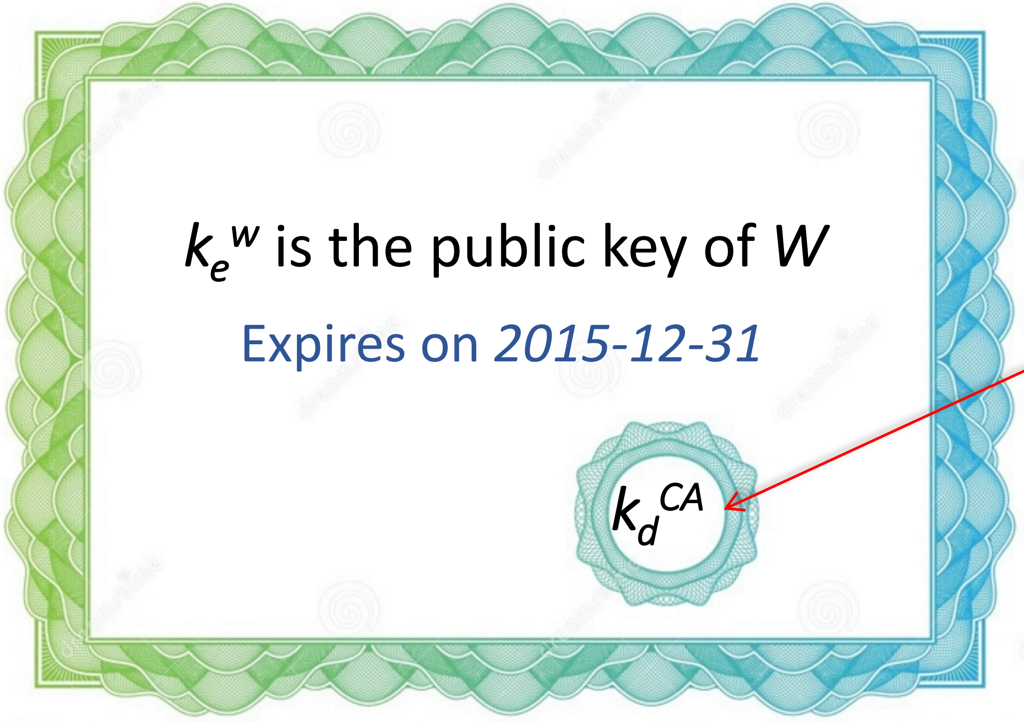
Asymmetric-Key Cryptography Applications

- Integrity?
  - Given a checksum $\boldsymbol{h}$, A signs h with its private key $\boldsymbol{k_d^A}$, and sends $\boldsymbol{h'} = F^{-1}(h, k_d^A)$ to B.
  - B gets h' and obtains the original checksum $\boldsymbol{h}$ by using the public key of A, $\boldsymbol{k_e^A}$. Then $\boldsymbol{h} = F(h', k_e^A)$.
- How could we make sure that the public key is really the public key of the party we want to communicate in the first place?
  - Certificate Authorities

Certificate Authorities

- Allow us to trust the *w* website and its public key k_e^w
- The *CA* emits a ***certificate*** (with the public key k_e^w) for the *w* website, and ***signs*** it with its own private key k_d^{CA}
- Our browsers contain a list of trusted *CAs* with their public keys k_e^{CA}

Website **W** introduces itself with this certificate when we start a secure connection



The certificate has been signed a CA with its private key.

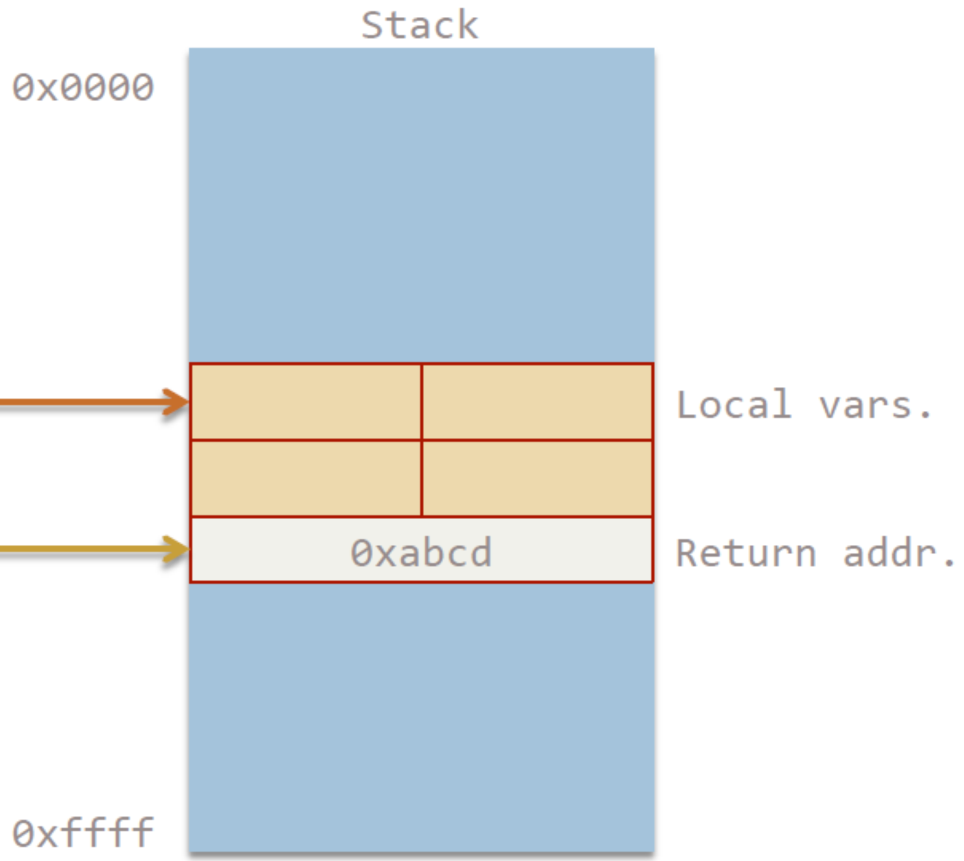
Outline

- Introduction to Cryptography
  - Symmetric Key
  - Asymmetric Key (Public-Key Infrastructure)
- Common Web Vulnerability
- Project 5

Common Vulnerabilities

- Buffer overflow

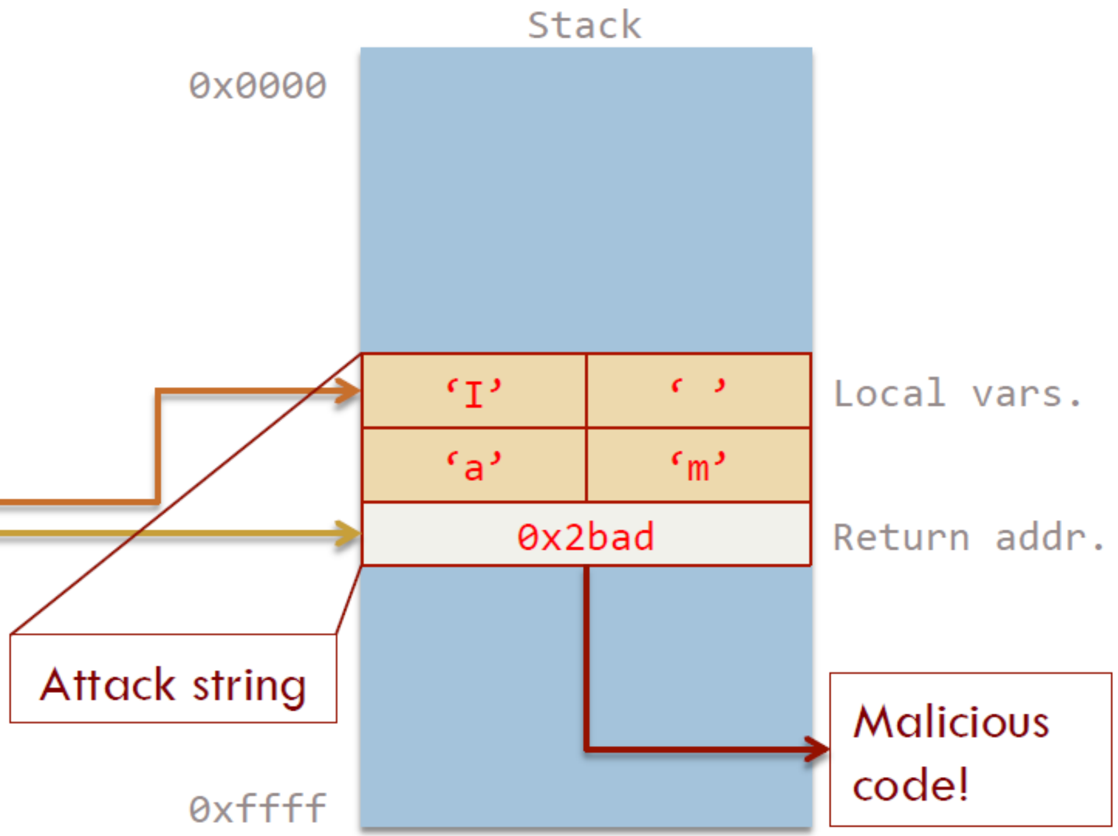
```
int main(){
   if( login() ){
      start_session();
   }
   return 0;
}

int login(){
   char passwd[4];
   gets( passwd );
   return (strcmp(passwd, “mpw”)==0);
}

int start_session() { ... }
```



Common Vulnerabilities

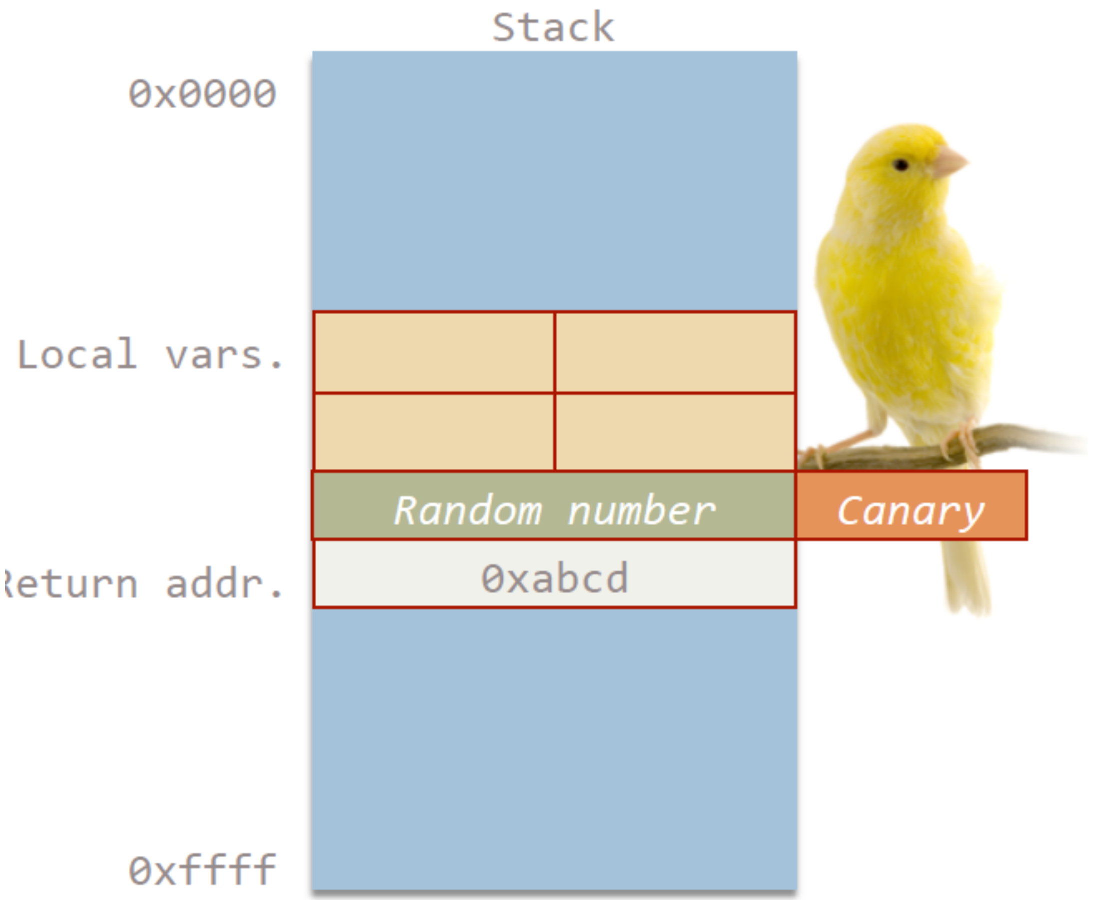
- Buffer overflow

```
int main(){
    if( login() ){
        start_session();
    }
    return 0;
}

int login(){
    char passwd[4];
    gets( passwd );
    return (strcmp(passwd, "mpw")==0);
}

int start_session() { ... }
```



Common Vulnerabilities

- Buffer Overflow (Solution)



- Activate a "Stack Guard" to insert Canary.
- Give minimum privileges to a program
- **Never trust user input**: don't use unsafe string functions in C (like strcpy, gets, strcat, sprintf, etc.)

Common Vulnerabilities

- Client-State Manipulation

- Don't store **sensitive** information in the client (only a Session ID)
- **Encrypt a checksum** (using a **signature** prior storing in the client).
- Attach either an **expiration date** or the Session ID to client's state.

Common Vulnerabilities

- SQL Injection

```
"SELECT * FROM product
WHERE id = " + user_input + ";"
```

```
123; DROP TABLE product;
```

- Create users with **minimum privileges**.
- Use **prepared statements**
- Encrypt sensitive data in DBMS
- Don't trust user input!

Common Vulnerabilities

- SQL Prepared Statement

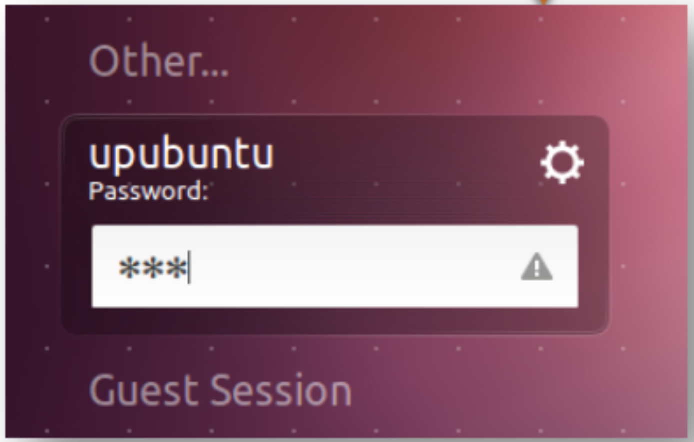
```
PreparedStatement s = db.prepareStatement("SELECT * from
                              Product WHERE id = ?");
s.setInt(1, Integer.parseInt(user_input));
ResultSet rs = s.executeQuery();
```

Common Vulnerabilities

- Command Injection

```
system( “cp f1.dat $user_input ” );
```

f2.dat; rm /etc/passwd



- Don’t use the System command. Use Runtime.exec() instead.
- “**Taint**” variables.
- Give **minimum privileges** to your application.

Common Vulnerabilities

- Cross-Site Scripting (XSS)

```
<body>
   Welcome to $user_input$ website
</body>
```

```
You’re doomed <script>hacked();</script>
```

- Why is it called Cross-Site?
- It’s very difficult to protect against it – we want to allow users insert HTML
- 2 options: white listing, black listing. *Which one is better?*

```
<div fu="alert(’XSS!');" STYLE="background-image: url(javascript:eval(this.fu))">
```

Common Vulnerabilities

- Cross-Site Request Forgery (XSRF)

- Always logout!
- *On server side, use an **action-token**.*

Outline

- Introduction to Cryptography
  - Symmetric Key
  - Asymmetric Key (Public-Key Infrastructure)
- Common Web Vulnerability
- Project 5

Project 5

- Part A - Testing Performance of Server
  - Learn to use Locust
  - Write some test cases for Locust
  - Find the max user number that the Server could handle

- Part B – Apache Spark
  - Learn to use Spark (shell)
  - Learn some basic Scala
  - Write a 10-line code in Scala to solve a problem.