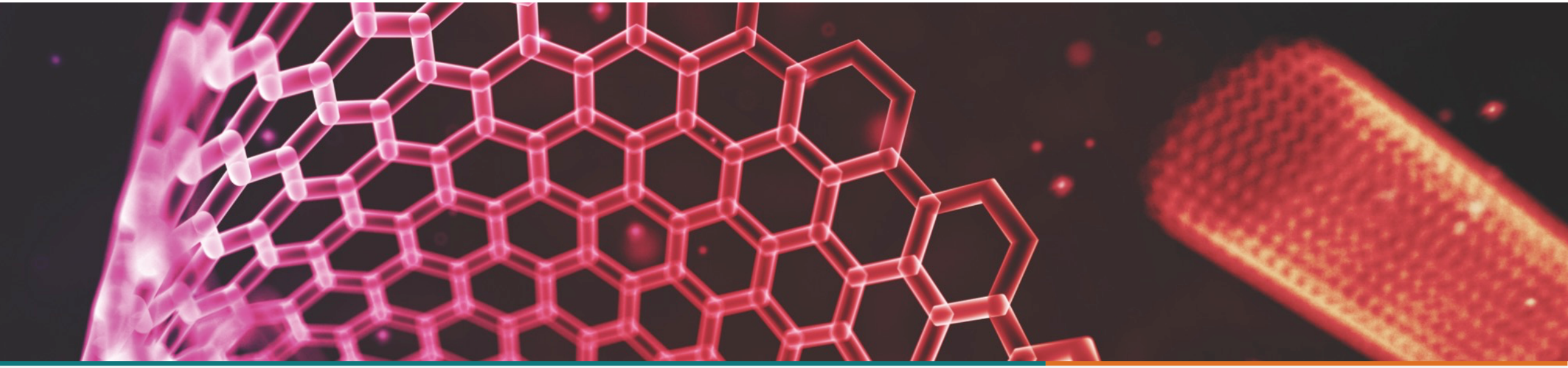
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

Security Concerns and Defenses



STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

stevens.edu

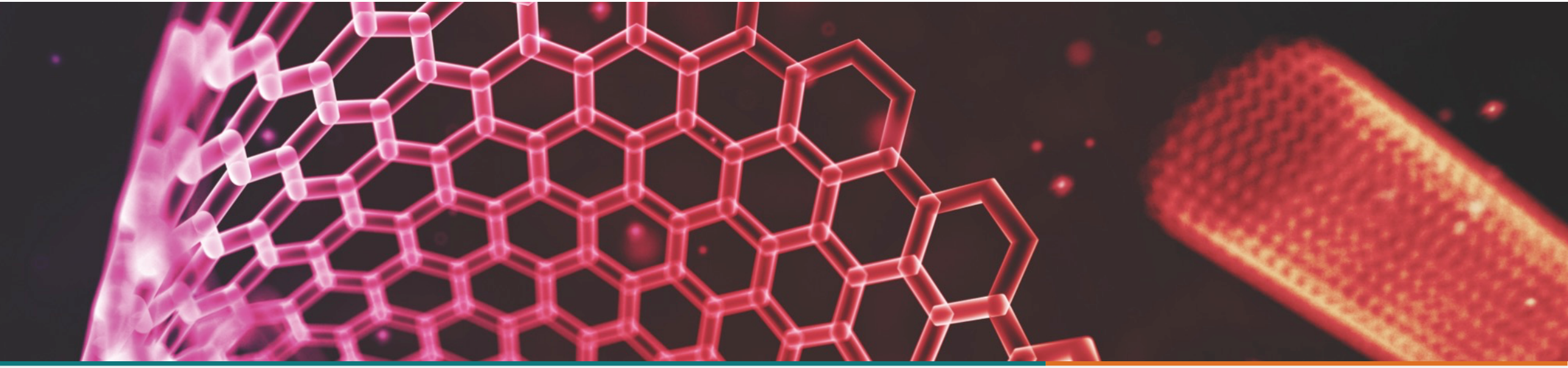
Patrick Hill

Adjunct Professor

Computer Science Department

Patrick.Hill@stevens.edu

General Notes on Security



What is Security?

Security is the protection of a system from a form of intrusion.
There are many threats to security

- Many types of attacks exist to damage a system
- Many types of attacks exist to gain access to resources that a user should not be able to access

There are many reasons for a system to be attacked

- Sensitive data is valuable to know, and valuable to sell
- Information in general is always valuable to have
- Take down a system due to personal views against the system
- Inconvenience users

How Secure Do We Need to Be?

Security is a sliding scale. Generally, you should always be secure enough that:

- User credentials and personal information are protected and not easily obtainable.

However, you routinely need to revisit your security because:

- More attacks are created daily
- As computers become more powerful, it becomes easier to break though common encryption style defenses

Overall, the more valuable the information you work with is, the more security you need.

- At some point, something is secure enough that it’s not worth attacking
- On the other extreme, sometimes data is unimportant enough that it’s not worth securing

Terminology and Actors

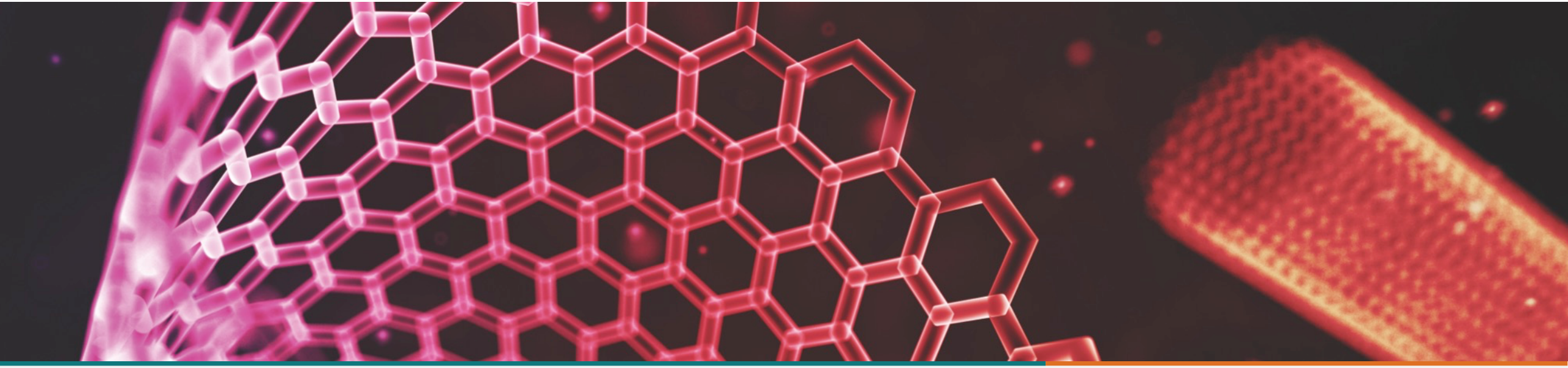
Terms:

- **Attacker**: a malicious user that is attempting to find and abuse a vulnerability in your system
- **User**: everyone else that is using your system

Actors:

- **You**: You! The system administrator, programmer, and all the other roles of technology!
- **Schmidt**: An attacker!
- **Jess**: Another attacker!
- **Nick**: A user, prone to being attacked.
- **Winston**: Another user who generally uses passwords such as *cat*
- **Cece**: Another user, whom is quite security conscious

Attacks and Defenses



XSS Attacks: Concept

An XSS Attack is an injection attack, where a malicious user manages to inject content(typically, JavaScript) into your website.

In an XSS Attack, code is executed on a user's browser; this allows an attacker full access to the page and all associated data

- Once you get access to one page, you can execute any amount of JavaScript to simulate a full, normal, experience while stealing all sorts of information. You can even trick the browser into making the user believe they are changing pages, but really keep the malicious JS requesting new pages and constantly scraping for new information to steal.
- Due to AJAX requests, they can send all sorts of sensitive data (usernames, passwords, credit card numbers, etc) to their remote servers the second users input the data.

XSS Attacks: Example Attack

Nick has decided to load up a local community website in order to look for a couch for sale. This websites has account information, private messages, personal preferences stored, and some personal information to allow online payment for local shopping.

Our first attacker, *Jess*, has made several posts in the local listings sections, one of which is for couches. These pages seem fine at first, however, her listings contained JavaScript embedded alongside her post description that execute code to do the following:

- When hitting the *inquire about this item* button, a user login modal will now appear asking for the user to input their username and password
- After the user inputs their username and password in this malicious login modal, it will be sent via AJAX to *Jess'* server, where she will harvest the information to login to *Nick's* account tonight and purchase the couch for $800.

Unfortunately, *Nick* falls for the attack and has a bad week.

- *Jess* has now stolen *Nick's* money and reads his private messages; she ruins his bank account, and then reveals embarrassing details to his girlfriend from his private messages to ruin their relationship. She also uses his personal information stored for billing details to steal his identity a few months later.
- She also uses that username and password on any account she can, including his email!

XSS Attacks: Defense

While *Jess* was the malicious actor here, *you* are the reason this attack was successful.

You can prevent XSS attacks by never displaying raw input that any user on your site may submit; you must always sanitize it and strip HTML from it.

- It's always safer to deny all HTML except for a whitelisted set of tags and attributes, rather than reject tags and attributes that you think should not be allowed.
- You can use the *XSS* package to help protect yourself
- https://www.npmjs.com/package/xss

CSRF Attacks: Concept

Cross-Site Request Forgery (CSRF) is the process of leveraging a browser's native process of making requests to load data on a page in order to have the user unintentionally trigger an action on another application.

These are generally mitigated with randomized tokens to be required to make anything useful happen at all.

CSRF Attacks: Example Attack

Jess is currently running a website that posts funny pictures of dogs each morning gets hundreds of thousands of hits each day. As a malicious attacker, however, she has other plans.

Through some amount of analytics and clever guessing, Jess changes the behavior of the *like* button for a picture for users that she thinks uses *mybank, LLC* and makes it occasionally submit a form request to change their username and password on the *mybank, LLC* website to something that she can later access!

The browser will submit this form and will have all the user's cookie and session data available while making the request; if the user logged into *mybank, LLC* before liking the photo, there is a large chance that their account credentials were just changed!

CSRF Attacks: Defense

CSRF attacks are easily circumvented with the concept of single-use tokens that are generated to be used on each form submission for a particular user's session. These tokens are sent back and forth as form data or querystring parameters so that they have to purposefully be included and cannot be scraped; the attacker's server does not have a way to scrape this data without already knowing the username, password, etc of the user.

SQL Injection Attacks: Concept

An SQL Injection is when you allow for a user to sneak their own SQL statements into SQL you're sending to the server. This allows them to attain unauthorized access to your database.

Much like other injection attacks, this is often caused by a combination of string concatenation and user input creating a string that does more than you, the programmer, intended to be allowed.

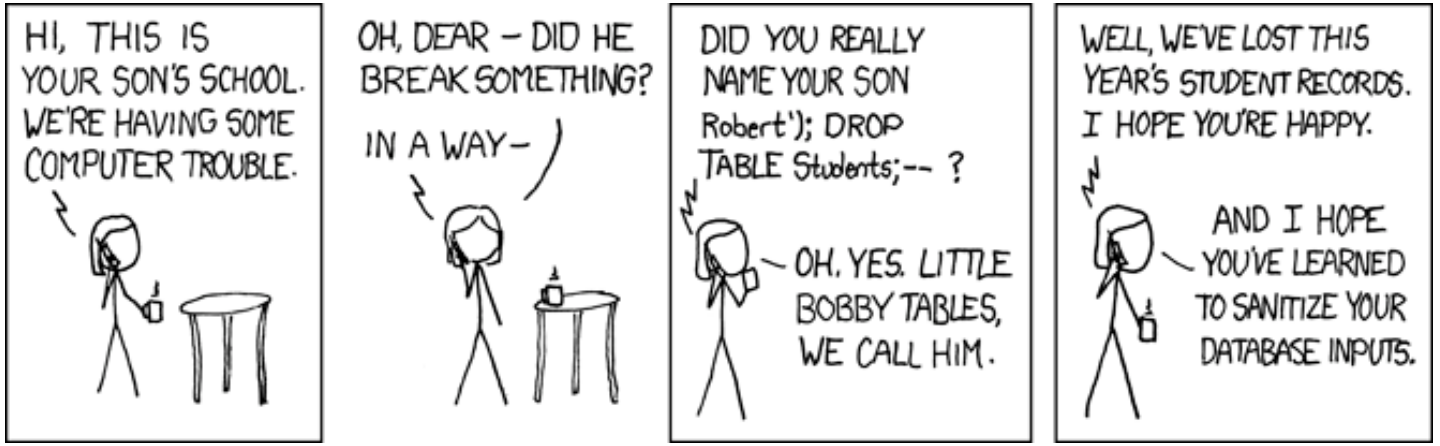
You can prevent this by:

- Sanitizing all strings used as input
- **Using prepared statements when interacting with SQL databases**

SQL Injection Attacks: Example

No example can come close to the classic XKCD comic about Little Bobby Tables

https://xkcd.com/327/



SQL Injection Attacks: Defense

By using a prepared statement, we can setup our database to receive a template of what a SQL command will be like and submit data through the use of submitting variables to populate that template.

- See: https://en.wikipedia.org/wiki/Prepared_statement for more information on the concept
- See your respective database documentation for specific implementation details.

File Inclusion Vulnerabilities: Concept

When you allow users access to resources that are stored in files, you may be tempted to give them urls that have paths to the files stored in the GET string, like so:

- http://localhost:3000/view_story?file=my_story.txt

This leaves you open to a Local File Inclusion!

- You can change my_story.txt to be ../../database_credentials.txt or something similar to access local files, if you do not allow users to select from a limited set of filenames

You're also vulnerable to a Remote File Inclusion!

- You can change my_file.png to be http://my_hacker_site.com/my_attack_vector.txt in order to execute your own scripts inside their page, which would allow you to gain access to their system!

These file inclusion vulnerabilities are very similar to the remote code execution vulnerability listed later.

File Inclusion Vulnerabilities: Example Attack

Imagine the following situation:

- You have created a website that uses a configuration file to stored database credentials.
  - Stored at: ***/app/config/db.config***
- Your website's purpose is to display recipes, which are stored in files
- You reference these files based on URL parameters and open them from the web server.
  - Ie: **http://myrecipes.com/recipes/view/jalapeno-poppers** will open ***/app/server/uploads/jalapeno-poppers***

Now, *Jess* decides to mess with the URL until she prints out the database credentials:

- She tries: **http://myrecipes.com/recipes/view/../app.js**
- She tries: **http://myrecipes.com/recipes/view/../../app.js** and is met with a page that prints the app file, which references a dbconfig!
- She tries: **http://myrecipes.com/recipes/view/../../config/db.config** and prints out the database config!

File Inclusion Vulnerabilities: Defense

Preventing File Inclusion Vulnerabilities are actually quite simple:

- Never reference files by name, always look them up by a randomly generated UUID of some sort.
- Always lookup files from a trusted location, and do not allow directory traversal.
- Offload your files far, far away from your application and try not to perform any file system operations on your own system.

Brute Forcing: Concept

A Brute Force Attack is when a user attempts to find information about your system/resources in your system (such as users) by providing a constant stream of values.

Essentially, a brute force is when anything guessable is constantly guessed at, in an attempt to access something that an attacker should not have access to.

These resources are often easily scriptable, such as scanning through thousands of pages with query parameters for data and scraping the content or submitting a form many times.

Brute Forcing: Example Attack

Jess is attempting to gain access to *Cece's* social media profile.

Jess is aware of *Cece's* username due to its public nature but needs to guess her password in order to gain access.

Jess can simply create a script that iterates through the hundred thousand most common passwords and try one guess at a time until one of them works.

Fortunately, Cece uses a password manager which generated a 16-digit randomized string for a password, which renders it effectively unguessable.

Had Jess been attempting to hack *Winston*, who's password is *ilovecats*, she would have gotten his password in a very short amount of time.

Brute Forcing: Defense

Brute Forcing's primary defense is rate limiting and throttling the amount of times that a user can request data. You can mitigate the ability to brute force a website by:

- Tracking the IP of every form submission and limiting based on that
- Lock accounts after some number of failed attempts to access.
- On password login attempts, you can purposefully stall your responses so that the attacker has to wait longer and longer, reducing the effectiveness of their attack.
- Require CAPTCHA style systems

DDOS Attack: Concept

A DDOS attack is a Distributed Denial of Service attack.

In general, a DOS is when some form of attack renders a server unable to respond in a timely manner.

A DDOS attack is often caused by many machines being used simultaneously to access a website

/ server, causing it to fall under such heavy load that it cannot keep up with any of the requests. This renders the server unusable; all requests will timeout.

DDOS: Example Attack

Winston runs a local cat boarding business out of his home. As his web admin, you have set him up with a small server with minimal bandwidth due to the relative low traffic his website generates.

Schmidt has been hired by another budding cat boarding business in the same area as Winston. Winston has announced a holiday discount for boarding cats more than five days in a row and is taking in a majority of the business due to his rapport in the area.

In order to cripple Winston, Schmidt has enabled a botnet across the world to all make thousands of requests towards Winston's website so that it appears as though his company has been crumpled.

Winston's competitor will now scoop up all the business in the area, leaving Winston unable to pay his bills. Winston closes up shop two weeks later, and the botnet is disabled; he decides to become a bard.

DDOS: Defense

Defending against DDOS attacks is inherently tricky, as there's very little way to determine whether or not traffic is real or from an attack.

- Who's to say he wasn't featured on the local news station as *Small Business of the Week* and now has thousands of requests an hour to get cats boarded? This could be his big break!

Despite that, there are ways that you can save Winston's business:

- When detecting a high amount of traffic, you can place a CDN in front of your website so that it renders static versions of all the content that are cached by highly powerful servers
- You can scale out to multiple web servers to handle more requests

Insecure Direct Object References: Concept

An IDOR is when some form of identifier (such as the primary key for your data) is visible to the end user. This itself is not a security hole, however may indicate a hole in your system.

At times, your system may expose information to the user, such as an order number for a shipment, that gives them a path to have the system query and display sensitive / private information.

When this path is exposed, some users may recognize the ability to tweak certain values (such as URL Params) in order to gain access to sensitive data that they may not have access to.

Insecure Direct Object References: Example Attack

While *Nick* puts his life back together, *Jess* is once again looking for vulnerabilities in the local community website.

Jess notices that when viewing a particular private message, her URL is:

- **http://mylocalwebsite.com/messages.php/?message=129**

Guessing that this message parameter leads to rendering a private message, she changes *129* to *291*, and the website displays a private message conversation between *Nick* and *Winston and* learns Nick's mother's maiden name and Winston's date of birth.

- Both very sensitive pieces of info!

Insecure Direct Object References : Defense

Preventing this IDOR is simple:

- You must implement an access control for all sensitive data.

Using the example before, the only two people who should be able to read that private message are *Winston* and *Nick*; when querying, the system should check if the current logged in user is either of those two parties. If not, it should take them to an error page.

Remote Code Execution: Concept

Remote Code Executions are when an attacker manages to have your system execute some amount of code that they supply, rather than that comes from your system.

This will allow them to gain some amount of access over your data, and potentially access to your entire system.

Remote Code Execution: Example Attack

Schmidt is perusing a popular fanfiction website that allows you to upload *txt* files, which will then be rendered as the text for each chapter of a submitter's story.

Schmidt deduces that the server is PHP, and guesses that they are using PHP's *include* keyword to include the file uploaded and print its content.

Schmidt creates an account and updates a txt file that has a great deal of PHP code in it that will print out all sorts of system environmental variables. Upon deducing that it is an Ubuntu server, he uploads another txt file that will issue shell instructions to make an account that he can SSH into and wreak havoc accordingly.

Remote Code Execution: Defense

Mitigating remote code execution attacks are actually quite simple:

- Identify any area of your application that dynamically includes code.
- Ensure that the code to be run comes from a trusted source, such as your internal system.
- When including user uploaded content, print it in such a way that code is not run but rather text is printed.

Username Enumeration: Concept

Username Enumeration is a case of trying to make a helpful prompt for the user, and in turn giving away sensitive information.

Username Enumeration is a vulnerability where a system will inform, on a failed login attempt, whether or not the username requested existed.

This will allow attackers to extract a list of usernames that are valid in the system.

Username Enumeration: Example Attack

Schmidt has decided to begin a long-planned attack on a small credit union for a town with a population of 15,000 people.

Schmidt has accumulated a list of several thousand usernames from the following online sources:

- Local newspaper websites from the town
- Local community forums from that town
- Local blogs from that town

He has also scraped the newspaper for first names, last names, and other personal details.

With this data, Schmidt creates a crawler that generates potential username combinations for each resident and tests each of them, accumulating a list of six hundred valid usernames.

He can then brute force many of their passwords, gaining access to their account.

Username Enumeration: Defense

The simplest way to prevent username enumeration related attacks is to **only** inform users that either their username **or** password is valid.

- **Never specify which field is invalid! Only specify that the combination was invalid.**

Buffer Overflow: Concept

A buffer overflow is a complex attack where the attacker manages to fill locations in memory with data (such as executable code) that allow attacker to manipulate the system.

- Allows them to change code execution
- Allows them to overwrite other data for their benefit.

Buffer Overflow: Example Attack

Schmidt and *Jess* decide to work together to take down a local newspaper website after discovering the ability in the comment section of the web application to overflow a buffer.

In order to execute a buffer overflow, they have had to determine what system the newspaper is running on and had to find a buffer vulnerability in the actual server code. The two of them work together for several months to force buffer overflows on the server technology the newspaper uses.

After finding a vulnerability that is triggered via form data manipulation, Jess submits a maximum size comment through form data, followed by a sequence of binary representing instructions to shut down the newspaper system.

Buffer Overflow: Defense

In order to prevent a buffer overflow, you have several options:

- Always check if you are setting more than the maximum amount of data that you are allowed to set when dealing with arrays and buffers!
- Do not use programming languages that allow you to manually manipulate memory.

LDAP Injection: Concept

LDAP injection leverages the LDAP protocol for programs running on a machine to access user data.

- https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

LDAP injection is conceptually similar to SQL injection, where string manipulation leads to unwarranted access to the system.

LDAP Injection: Example Attack

Jess is scanning a portal website in a company's public wifi and finds a directory of employees. She notices that the querystring parameter changes to reflect what appears to be a username.

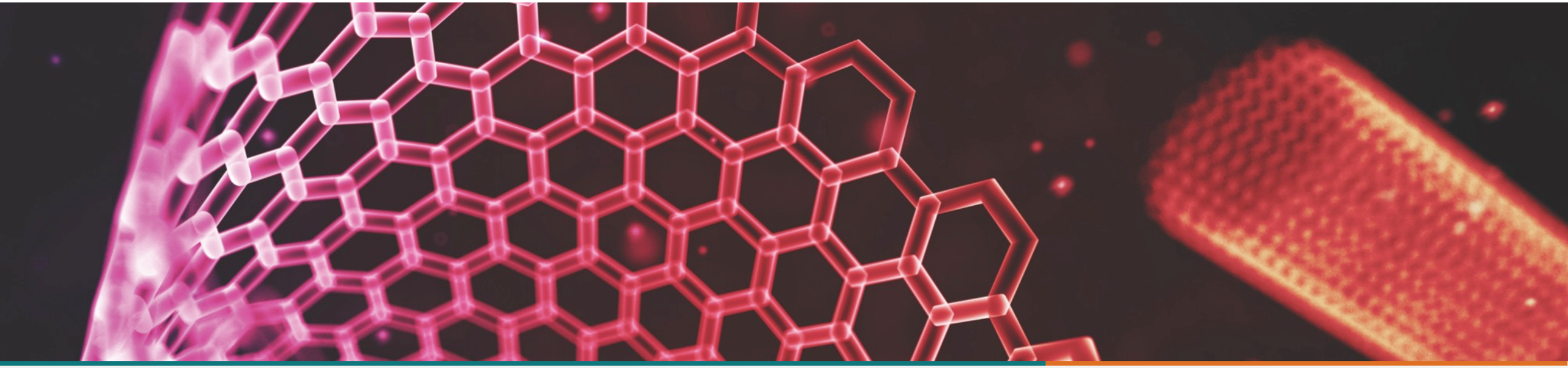
She also notices that an employee login screen is accessible on the site and guesses that they are using LDAP protocol to pull this profile information as well as to login to the corporate website.

Jess logs in with a username of *, which is an LDAP wildcard, and matches a password of
ilovecats, causing her to log in as *Winston*.

LDAP Injection: Defense

Much like SQL injections, the primary manner of defense against LDAP injections are to sanitize user input.

Other Web-Related Security Concepts



What Else is There?

The world of security is extremely massive, particularly due to the public nature of the internet and the extreme amount of sensitive data there is.

There are many concepts you should familiarize yourself with that are not necessarily attacks but are still pertaining to good security practices.

Social Engineering

The weakest point of security is the human beings. Machines are not easily manipulated by the following:

- Someone crying over a telephone call about how if you don't help them reset the password their child will forever starve
- Someone yelling at them that they *will* speak to your manager and report you
- Someone imitating a user, knowing 90% of what the person would know about themselves, and convincing the machine that they are definitely the user despite not knowing the password, backup key, security answer, etc.

Humans, unfortunately, often fall prey to social engineering to reveal sensitive information

- https://en.wikipedia.org/wiki/Social_engineering_(security)

Rate Limiting With Captcha

The internet is easily crawled through by malicious scripts

- Many scripts run through the internet constantly trying to break into Wordpress websites, for example!

One of the easiest ways to validate a real human is submitting data as opposed to a robot is to use a CAPTCHA on your forms.

CAPTCHA are utilities that detect whether or not a real human is filling out the form by making people perform some task that machines cannot do, such as reading a warped piece of text and writing it back; the text is selected because OCR software was not able to detect the letters, ergo it is generally machine-proof.

HTTPS

HTTPS is a protocol for secure communication across the networks such as the internet. HTTPS allows for end-to-end encryption of communication.

Setting up HTTPS is relatively straightforward (albeit a long process), however involves the acquisition of an SSL Certificate, which is generally purchased.

- Nowadays, *Let's Encrypt* is a free service that allows for that!
- https://letsencrypt.org/

A guide for setting up HTTPs on a Digital Ocean instance gives a good overview of how the process works

- https://www.digitalocean.com/community/tutorials/how-to-install-an-ssl-certificate-from-a-commercial-certificate-authority

Encrypting Passwords

Encrypting passwords is mandatory at all times. You should never, ever, ever store plaintext passwords.

- If your database is compromised, your attackers gain access to the passwords
- If you accidentally leak data (ie, sending JSON of the entire user object) , the plaintext password will be exposed.

Not-encrypting your passwords in a hard-to-break encryption technique results in points taken off your final project.

- You may use the popular ***bcrypt*** function to accomplish this
- https://en.wikipedia.org/wiki/Bcrypt
- https://www.npmjs.com/package/bcrypt

HIPAA Concerns

In software development, you may end up working with healthcare related data, meaning you will have to follow a series of requirements to ensure that your data is HIPAA compliant. There are many aspects of HIPAA Compliance:

- **Transport Encryption**: Is always encrypted as it is transmitted over the Internet
- **Backup**: Is never lost, i.e. should be backed up and can be recovered
- **Authorization**: Is only accessible by authorized personnel using unique, audited access controls
- **Integrity**: Is not tampered with or altered
- **Storage Encryption**: Should be encrypted when it is being stored or archived
- **Disposal**: Can be permanently disposed of when no longer needed
- **Omnibus/HITECH**: Is located on the web servers of a company with whom you have a HIPAA Business Associate Agreement (or it is hosted in house and those servers are properly secured per the HIPAA security

See LuxSci article for a full explanation of HIPAA concerns and ways to address them

- https://luxsci.com/blog/what-makes-a-web-site-hipaa-secure.html

Questions?