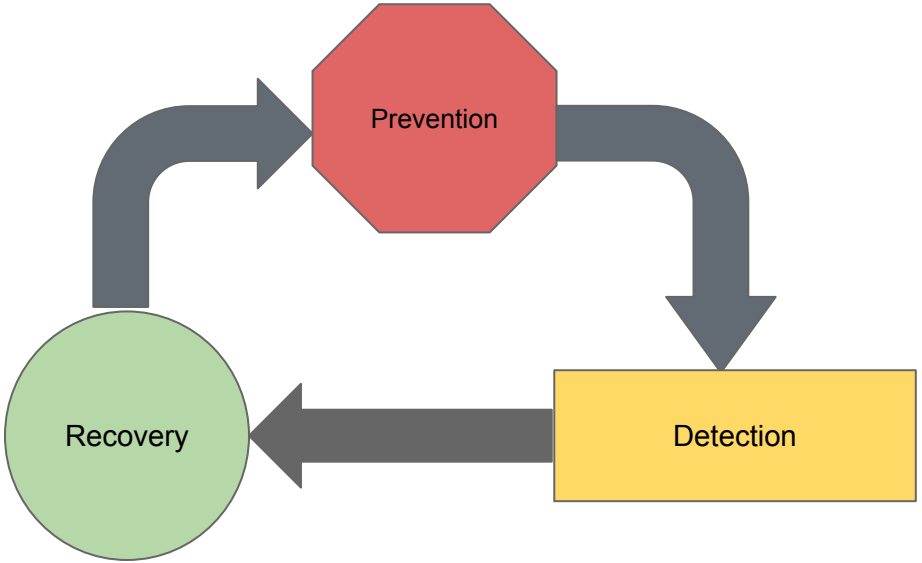
Hacker Hour: Cyber Security

“Alice, Bob, Eve and Trent”

Aditya Geria

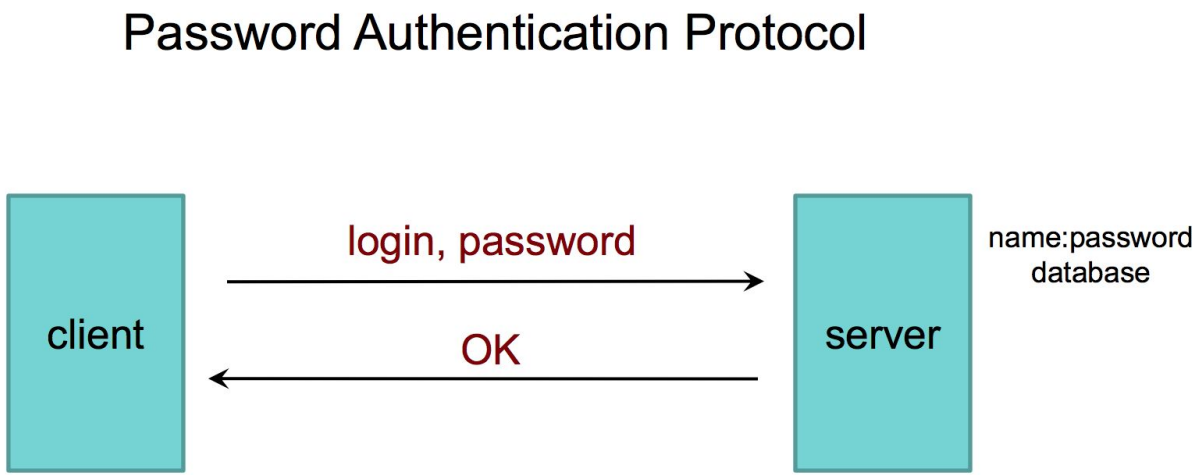
Security Model



What are some ways to combat this?

Authentication

- Authentication vs Identification vs Authorization
  - Identification = “Who are you?” (A user ID, name)
  - Authentication = “Prove it” (A password/shared secret)
  - Authorization = “You can do the following” (permissions)



Building on Passwords

SALT YOUR PASSWORDS!

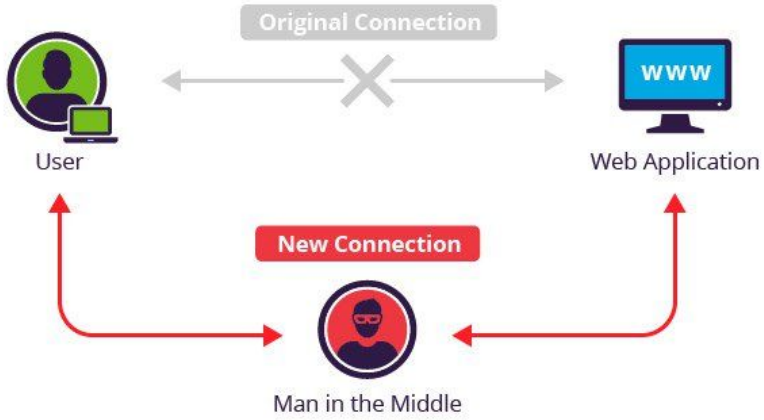
- **Salt** = a random *nonce* that is appended to a user's password and is stored in the clear in the password database
- Database will store: A username, salt, H(salt || password)
- Authentication: $H(\text{password}_{(client)} \,||\, \text{salt}_{(server)}) \stackrel{?}{=} H(\text{salt} \,||\, \text{password})_{(server)}$
- General Salting guidelines: https://crackstation.net/hashing-security.htm

Further reading: MS-CHAP Protocol, Forward Secrecy

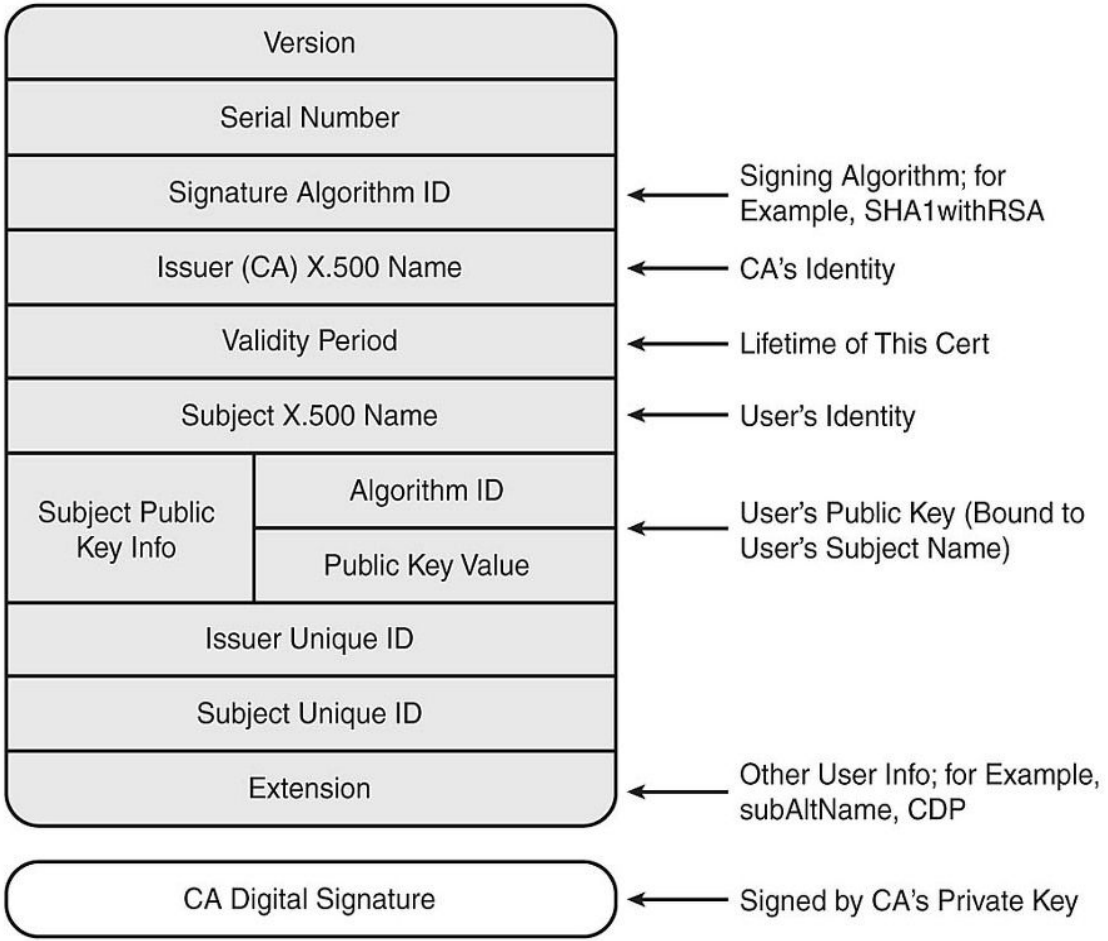
Man in the Middle - why you need Encryption

- Causes:
  - Spoofing a wifi access point
  - Key-reuse
  - Communicating over HTTP or insecure channel

- Defenses
  - Use a signed message: { message, encrypted hash of message }
  - Replay attacks can still happen -> add timestamps/sequence IDs



SSL Certificate



The Modern Internet

- Browsers and Applications are super complex!
  - JavaScript – allows code execution
  - Document Object Model (DOM) – change appearance of page
  - XMLHttpRequest (AJAX) – asynchronously fetch content
  - WebSockets – open interactive communication session between JavaScript on a browser and a server
  - Multimedia support - <audio>, <video>, <track>
  - MediaStream recording (audio and video), speech recognition & synthesis
  - Geolocation
  - NaCl (Chromium Native Client) – run native code inside a browser (sandboxed)

- **Same-origin policy:** A browser permits scripts in one page to access data in a second page only if both pages have the same origin (URI, port, and host)

Cookies and XSRF

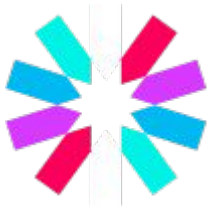
- Websites use cookies to track client progress through a website
- Enables **XSRF: “Cross Site Request Forgery”** Attacks on weak websites
  - Planting a link or forging a cookie in order to allow access to a client’s browsing session
  - Example: http://mybank.com/?action=transfer&amount=100000&to=attacker_account
  - Tom Scott on XSRF: https://www.youtube.com/watch?v=vRBihr41JTo
- **Defenses**: *Referrer* header in a cookie or require unique tokens per session
  - JWT (JSON Web Token) https://jwt.io/introduction/
  - OAuth2

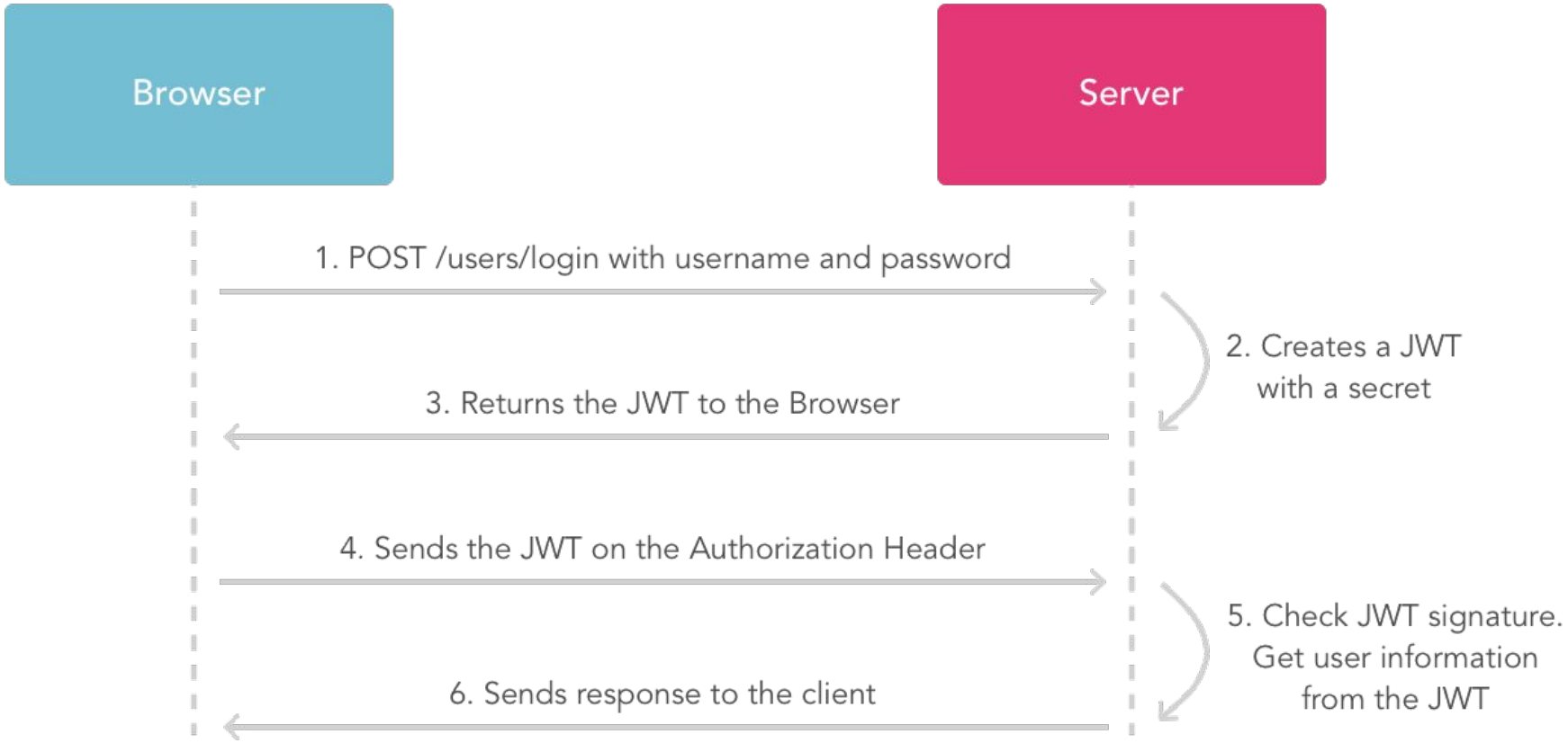
JWT - JSON Web Tokens

*“self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.”*

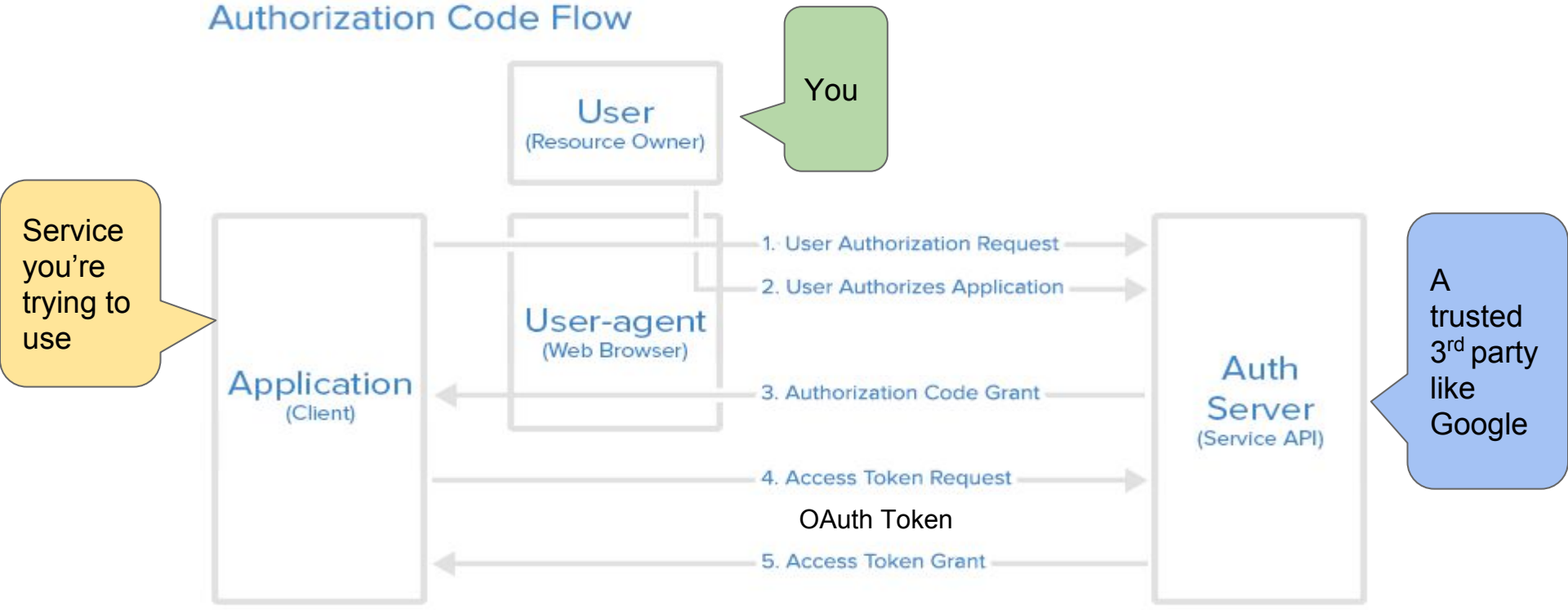
“Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.”

https://jwt.io/introduction/




Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the Browser
4. Sends the JWT on the Authorization Header
5. Check JWT signature. Get user information from the JWT
6. Sends response to the client

OAuth 2.0


Authorization Code Flow
User
(Resource Owner)
You
Service you're trying to use
Application
(Client)
User-agent
(Web Browser)
1. User Authorization Request
2. User Authorizes Application
3. Authorization Code Grant
4. Access Token Request
OAuth Token
5. Access Token Grant
Auth Server
(Service API)
A trusted 3rd party like Google

XSS - Cross Site Scripting

- Code injection attack that allows attacker to execute JavaScript in a user's browser
  - Exploit vulnerability in a website the attacker visits
  - Possible if the website includes user input in its pages
- Example: a weak chatting system (GAH)
- **Possible damage:** Hijack a session, Create arbitrary HTTP requests with arbitrary content via XMLHtttpRequest, Make arbitrary DOM modifications, Install keyloggers, Download malware/miners, run JavaScript ransomware, try phishing by manipulating the DOM and adding a fake login page.

```
28
29  socket.on('chatMessage', function(from, msg){
30    var me = $('#user').val();
31    var color = (from == me) ? 'green' : '#009afd';
32    var from = (from == me) ? 'Me' : from;
33    $('#messages').append('<div class="row"><b style="color:' + color + '">' + from + '</b>: ' + msg + '</div>');
34  });
35
```

Defenses against XSS

Sanitize ALL user input!!

- **OWASP**: https://www.owasp.org/index.php/Injection_Prevention_Cheat_Sheet_in_Java
- **XSS Defense Cheat Sheet:** https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
- **Content Security Policy (CSP):** Allows website owners to identify approved origins of content & types of content
- `String safe = Jsoup.clean(unsafe, Whitelist.basic()); //java`

SQL Injection

- Similar to XSS, but inject SQL Code to mess with a database
- Occurs when **volatile data** is passed in as part of a query
  - `statement = "SELECT * FROM users WHERE name = '" + userName + "';"`
- Or when the Query can be exposed via a URL
  - https://mywebsite.com/important?query=select%20*%20from%20some_table%20where...

- **Blind SQL Injection:** Results of a query are not visible to the attacker
- **Tom Scott on SQL Injections**: https://www.youtube.com/watch?v=_jKylhJtPmI

```
@app.route('/api/v1.0/storeAPI/<item>', methods=['GET'])
def searchAPI(item):
    g.db = connect_db()
    [redacted]
    curs = g.db.execute("SELECT * FROM shop_items WHERE name = '%s'" %item)
    results = [dict(name=row[0], quantity=row[1], price=row[2]) for row in curs.fetchall()]
    g.db.close()
    return jsonify(results)
```

```
@app.route('/api/v1.0/storeAPI/<item>', methods=['GET'])
def searchAPI(item):
    g.db = connect_db()
    #curs = g.db.execute("SELECT * FROM shop_items WHERE name=?", item) #The safe way to actually get data from db
    curs = g.db.execute("SELECT * FROM shop_items WHERE name = '%s'" %item)
    results = [dict(name=row[0], quantity=row[1], price=row[2]) for row in curs.fetchall()]
    g.db.close()
    return jsonify(results)
```

Defenses against SQL Injections

Sanitize your input!!!!

- A query should not be a part of the URL. If it is, use **Database permissions** to disallow any queries except for read queries.
- Escaping input:
  - Every occurrence of a single quote in a parameter must be replaced by two single quotes to form a valid SQL string literal
  - PHP uses `mysqli_real_escape_string();`
- Parameterized Values (placeholder values and prepared statements) : **‘?’**
  - Resilient against SQL injection because values which are transmitted later using a different protocol are not compiled.
  - If the statement template is not derived from external input, SQL injection cannot occur.

Database permissions on MS SQL Server

```
deny select on sys.sysobjects to webdatabaselogon;
deny select on sys.objects to webdatabaselogon;
deny select on sys.tables to webdatabaselogon;
deny select on sys.views to webdatabaselogon;
deny select on sys.packages to webdatabaselogon;
```

Escaping (PHP)

```
$mysqli = new mysqli('hostname', 'db_username', 'db_password', 'db_name');
$query = sprintf("SELECT * FROM `Users` WHERE UserName='%s' AND Password='%s'",
                 $mysqli->real_escape_string($username),
                 $mysqli->real_escape_string($password));
$mysqli->query($query);
```

Prepared statements

1. **Prepare**: At first, the application creates the statement template and send it to the DBMS. Certain values are left unspecified, called *parameters*, *placeholders* or *bind variables* (labelled "?" below):

   ```
   INSERT INTO products (name, price) VALUES (?, ?);
   ```

2. Then, the DBMS compiles (parses, optimizes and translates) the statement template, and stores the result without executing it.
3. **Execute**: At a later time, the application supplies (or *binds*) values for the parameters of the statement template, and the DBMS executes the statement (possibly returning a result). The application may execute the statement as many times as it wants with different values. In the above example, it might supply "bike" for the first parameter and "10900" for the second parameter.

Wrap Up

- Security isn't perfect, and neither is our code :(
- We must do our best to write code that is robust and resilient against malicious hackers
- Must be knowledgeable about the things that **can** go wrong, and be able to act accordingly
- Knowing what can go wrong helps us write code that fails less often, and build systems that are highly available

Further Reading:

- Encryption, how to encrypt data in transit (PyCrypto)
- Key Management (Amazon KMS)
- Buffer Overflow Vulnerabilities
    - Capture the flag challenges (check out: Protostar Exploit Exercises)

Thanks!

github.com/901

adgeria@gmail.com

Aditya Geria