Secure coding in Java

Administration

- Hours 8:30am to 4:30pm
- Breaks
  - Short breaks throughout the day
  - Lunch 12:00pm to 1:15pm

Agenda

- System security
- Security Terms
- Threat modeling
- OWASP
  - Top 10
  - Application Security Verification Standard (ASVS)
- Java Security Architecture
  - JDK – Java Development Kit
  - Core

Agenda

- Java Secure Coding
- Java Cryptography
- REST API
- JAAS - Java Authentication and Authorization Service
- Security configuration
- Spring Security

Introduce yourself

- Name
- What is your role in the organization
- Indicate exposure to secure programing

Development environment

- This course does not assume any development environment other than at least Java 11 and maven.
- You are welcome to use the IDE of your choice
  - https://code.visualstudio.com/
  - https://www.eclipse.org/ide/

Start Early

- Security considerations should not start at implementation or a customer bug report. For effective security, you need to play effective offense and not just have a defensive strategy. Security should be an early consideration. How early? Even before design, when the features of the product are being determined.

Team Responsibility

- Everyone within an organization is responsible for security, not just developers and architects.
  - This includes:
    - Executive team
    - Management
    - Designers
    - Operations
    - DevOps
    - Testers
- A competent security strategy spans the entire software development life cycle. For that reason, only a collaborative and iterative approach towards security is likely to be successful.

System Security

- Security happens in layers, starting with the hardware all the way up to the application.

Security layers

- Hardware
- Operating System (OS)
- Java
  - Virtual Machine (VM)
  - Bytecode
  - Verifier
  - Language
- Code
- Application

Hardware

- Hardware can be attacked, and security is built at this level to try and prevent it.
- An example is the rowhammer attack.
- In this attack, a program is executed repeatedly on a row of transistors on a memory chip. The row is hammered until electricity leaks into another row and flips some bits. This can be used to execute privilege escalation exploits.

OS

- Operating systems provide process isolation.
  - Memory protection
    - Prevent one process from accessing the memory of another process
    - Often done by providing virtual address
    - Requires 2 modes of CPU execution: privileged and unprivileged
  - Inter-processes communication
    - Important to restrict communication between processes of different users
  - Boot Security
  - Access to raw devices

Writing safer code

- Java was created with security and safety in mind.
    - Uses a virtual machine that interprets bytecode
    - Bytecode is run through a verifier
    - No raw pointers
    - Access modifiers to protect methods
    - Strict typing
    - Garbage collection
    - Sandboxing of code
        - SecurityManager

Application

- Authentication
  - Who are you?
- Authorization
  - What can you do?
- Rate limiting
  - How much can you do?

What is secure coding?

- Design and develop software by applying industry best practices, and security standards to avoid weaknesses that lead to security-related vulnerabilities.

Why security is important

- Preventing cybersecurity incidents starts with the code itself.
- Weaknesses in code open vulnerabilities that can be exploited.
- It is important for developers to
  - think securely when developing
  - understand common weaknesses
  - how to proactively avoid common weaknesses

Security as an afterthought

- Reasons why security is only considered after the system is built.
  - Functional release prioritized over security aspects.
  - Program complexity.
  - Not understanding of the specifics of the development language.
  - Knowledge of Security Coding Guidelines not transmitted to the team.

Implement Security by Design and Default

- It is important that team is aware of possible vulnerabilities and risks of the software.
    - E.g., If developing a health care application, the top risk would be theft of personal health data.

Develop a Secure Coding standard

- Team needs to be aware of it and compliance must be checked.
- A few examples of default items in a secure design.
  - Authentication needs to be implemented at every layer.
  - Authentication tokens communicated only over encrypted channels.
  - Properly store and protect all keys, passwords, and certificates.
  - Implement encryption on files, databases, and data elements.

Security terms

- CWE (Common Weakness Enumeration)
- Threat vector
- Attacks
- Threat modeling

CWE

- A community-developed list of software and hardware weakness types. It serves as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.
- http://cwe.mitre.org/
- https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

Threat

- A program which has the potential to cause serious damage to the system
- A malicious act that seeks to damage data, steal data, or disrupt digital life in general
- Computer viruses, data breaches, Distributed Denial of Service (DDoS), and other attack vectors

Attack Terms

- Attack -
  - An attempt to break security and make unauthorized use of an asset
  - Any form of malicious activity that targets IT systems, or the people using them, to gain unauthorized access to the systems and the data
- Attack Vector
  - Method of gaining unauthorized access
- Attack Surface
  - Total number of attack vectors an attacker can use

Vulnerability

- Weakness that can be exploited to gain unauthorized access to a computer system

Data Breaches

- An unauthorized access where sensitive information is copied, transmitted, viewed, stolen
- Common ways data breaches happen
  - Exploitation system vulnerabilities
  - Injection
  - Spyware
  - Phishing
  - Insecure Passwords
  - Broken or Misconfigured Access Controls
  - Third-Party Vendor Breaches

Distributed Denial of Service (DDoS)

- Overloading a server with high amounts of fake traffic to make a service unavailable
- Usually uses a botnet, a collection of devices with malware installed which allows them to be remotely controlled
- Types of DDoS attacks
  - Volume-Based
  - Protocol or Network-Layer

Adversarial

- For security modeling, understanding the adversarial perspective is the best approach. Apply this approach to all aspects of threat modeling:
    - Attack surface
    - Vulnerabilities
    - Threats
- Thinking like an adversary may not be natural but is important in order to secure systems!

Threat model

- The threat model is not a plan or security solution itself:
  - Provides guidance for security features
  - Prioritizes security threats
  - Highlights vulnerabilities
  - Identifies threat assets
  - Reveals possible mitigations
- The ingredients of a threat model can vary and may include Data Flow diagrams (DFD), attack trees, threat list, vulnerabilities, and whatever is helpful for securing a product.

Threat modeling

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good enough job?

STRIDE

- Common threat modeling approach
- Has six areas of focus:
  - Spoofing
    - Faking credentials
  - Tampering
    - Changing data
  - Repudiation
    - Covering tracks (Audit logs counter this)
  - Information disclosure
    - Getting unauthorized data
  - Denial of Service
  - Elevation of Privilege

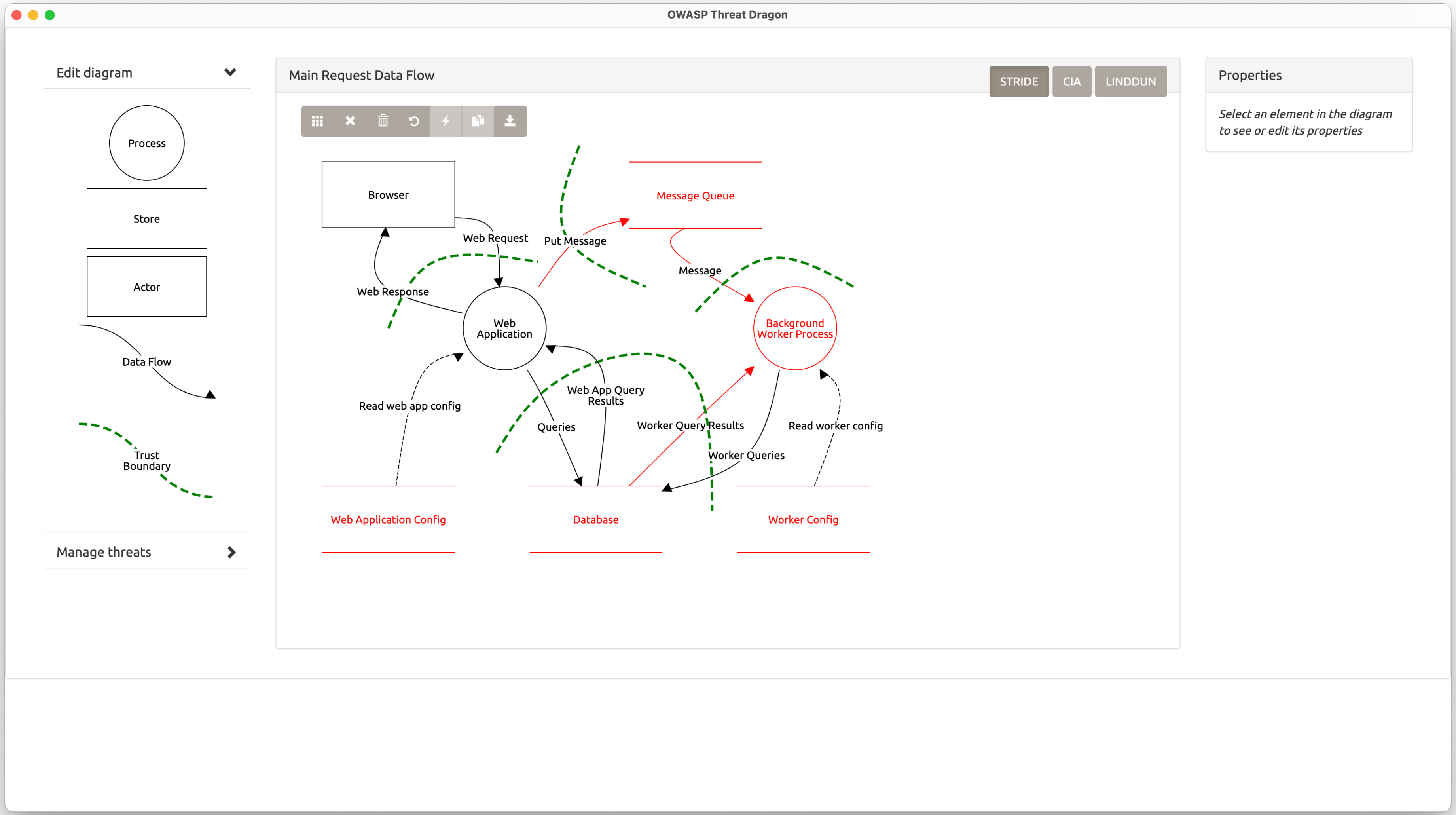
STRIDE

- To follow STRIDE, you decompose your system into relevant components, analyze each component for susceptibility to the threats, and mitigate the threats. Then you repeat the process until you are comfortable with any remaining threats.

Threat modeling automation

- There are several tools available to aid in your threat modeling
  - OWASP Threat Dragon
    - https://owasp.org/www-project-threat-dragon/
  - Cairis
    - https://cairis.org/
  - IriusRisk
    - https://www.iriusrisk.com/
  - Kenna.VM
    - https://www.kennasecurity.com/products/vm/
  - Microsoft Threat Modeling Tool
    - https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

OWASP Threat Dragon



Select an element in the diagram to see or edit its properties

Threat maps

- Proactive strategy against zero-day attacks
- Threat maps
  - https://www.digitalattackmap.com
  - https://cybermap.kaspersky.com/
  - https://threatmap.checkpoint.com/

Honey pots

- Cybersecurity is not just about strengthening your computer systems.
- Providing bait - applications, data, and systems that look real can be used to deflect cybercriminals from a legitimate target or to gain information on how the cybercriminals operate.
- A honey pot is this bait.
- Threat maps use honey pots as data source.
- www.honeynet.org

Open Web Application Security Project (OWASP)

- Nonprofit foundation that works to improve the security of software.
- Founded 2001
- Projects, tools, documents, forums, and chapters are free and open to anyone interested in improving application security.
- A great place so start your security journey.
- https://owasp.org/

OWASP Top Ten (2021)

- **A new list is released every few years and provides a list of the most common vulnerabilities found in production code.**
  - **A01:2021-Broken Access Control**
  - **A02:2021-Cryptographic Failures**
  - **A03:2021-Injection**
  - **A04:2021-Insecure Design**
  - **A05:2021-Security Misconfiguration**
  - **A06:2021-Vulnerable and Outdated Components**
  - **A07:2021-Identification and Authentication Failures**
  - **A08:2021-Software and Data Integrity Failures**
  - **A09:2021-Security Logging and Monitoring Failures**
  - **A10:2021-Server-Side Request Forgery**

Broken Access control

- Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

Injection

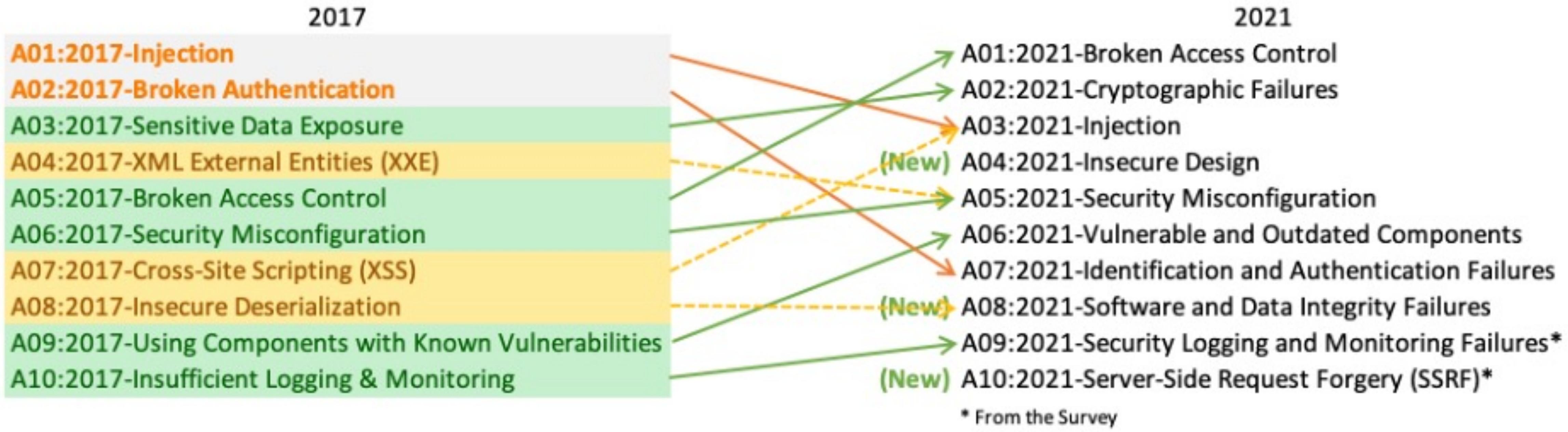
- XSS (Cross-site scripting)
- SQL Injection
- Command and Code injection

Input validation

- Assume all input is malicious.
  - Reject any inputs that do not conform to specifications.
- Consider all potentially relevant properties.
  - Length, type of input, full range of acceptable values, syntax, etc.
  - Semantic; e.g., “house” is syntactically correct but only colors expected.
- Do not rely exclusively on deny lists.
  - Useful for potential attack detection or if malformed request should be rejected outright.

OWASP Top Ten 2017 to 2021

- Things change over the years
  - Developers get better at preventing some weaknesses and new one arise.


2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

List of categories

- The OWASP Top Ten is a list of categories
  - Each category includes several CWEs
    - A01:2021-Broken Access Control
      - 34 CWEs
    - A02:2021-Cryptographic Failures
      - 29 CWEs
    - A03:2021-Injection
      - 33 CWEs
    - Etc.

Application Security Verification Standard (ASVS)

- Provides a list of requirement developers can use for secure development and testing of application security controls.
- Designed to be used as:
  - Metric – assess degree of trust in application
  - Guidance – what to build into security controls
  - Procurement – specifying security verification requirements in contracts

Principle of Least Privilege (PoLP)

- Practice of limiting access rights to only those needed to perform the operation.
- Deny by default (Zero trust)

Why PoLP

- Personal information of 100 million Target customers was stolen because a third-party contractor (HVAC maintainer) was given privileged access to Target's network.

JDK – Java Development Kit

- JDK – strong emphasis on security
  - Java language
    - Type-safe
    - Garbage collection
    - Secure class loading and verification
  - Many security API's

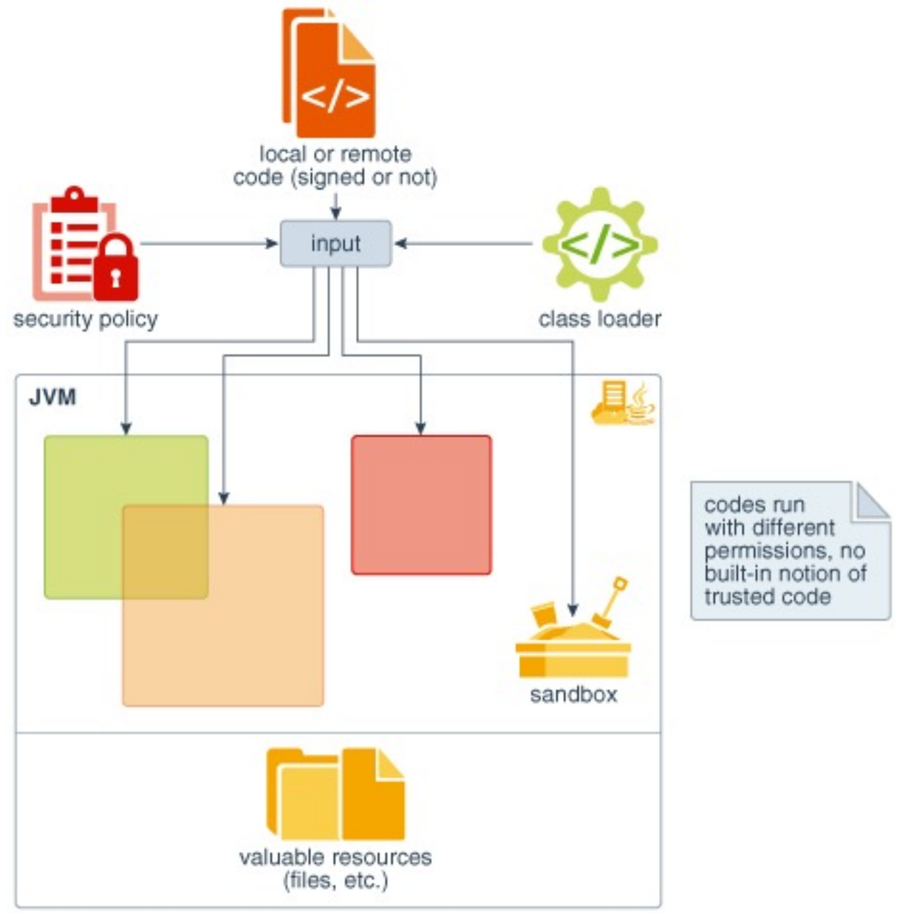
Java SE Platform Security Architecture

Overview

- Sandbox model
- Protection mechanisms
- Permissions and policy
- Access control
- Security management

Sandbox model

- Features
  - Fine-grained access control.
  - Easily configurable security policy.
  - Easily extensible access control structure.
  - Security checks apply to all Java programs
    - applications as well as applets.


local or remote code (signed or not)
security policy
input
class loader
JVM
sandbox
codes run with different permissions, no built-in notion of trusted code
valuable resources (files, etc.)

Terms

- Principal
  - an entity in the computer system to which permissions (and as a result, accountability) are granted
- Protection domain
  - Includes the set of objects that are currently directly accessible by a principal
- Two main categories of protection domain
  - system
  - application

Protection domains

- Protected external resources, such as the file system, the networking facility, and the screen and keyboard should only be accessible via system domains.
- Objects in the application domain interact with the system domain to access such a resources.
- It is important that a less "powerful" domain cannot gain additional permissions as a result of calling or being called by a more powerful domain.

AccessControler.doPrivileged

- Java uses the doPrivileged method to temporarily enable access for application domain objects to protected resources
- Invoked by resource-handling code directly or indirectly

Policy and Permissions

- Java security is policy based
- The default `SecurityManager` reads a security.policy file to determine permissions
- `java.security.Permission` is an abstract class that is subclassed for specific permissions

java.security.Permission

- Common permissions
  - AllPermission
  - BasicPermission
  - FilePermission
  - SocketPermission
  - UnresolvedPermission
  - URLPermission

Policy File

- Default system policy *<java-home>*/conf/security/java.policy
  - Grants default permissions

```
...
// default permissions granted to all domains
grant {
    // allows anyone to listen on dynamic ports
    permission java.net.SocketPermission "localhost:0", "listen";

    // "standard" properies that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
...
```

Policy File location

- Location of the policy file can be set in
  - *<java-home>*/conf/security/java.security

```
...
# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
#

policy.url.1=file:${java.home}/conf/security/java.policy
policy.url.2=file:${user.home}/.java.policy
...
```

Policy File at runtime

- You can also specify an additional policy file at runtime

```
java -Djava.security.manager -Djava.security.policy=someURL SomeApp
```

- Or completely replace the policy

```
java -Djava.security.manager -Djava.security.policy==someURL SomeApp
```

Policy File Syntax

- Contains zero or more "grant" entries and maybe a "keystore" entry

```
keystore .keystore;
keystorePasswordURL "file:/usr/local/";

grant codeBase "file:/usr/java/com/greetings/-" {
    permission java.util.PropertyPermission "foo", "read";
};
```

Grant Entries

- All code in Java is considered to come from a code source
- Each grant entry includes one or more permission entries and optionally a codeBase and signedby

```
grant signedBy "signer_names", codeBase "URL",
        principal principal_class_name "principal_name",
      ... {

      permission permission_class_name "target_name", "action",
          signedBy "signer_names";
      permission permission_class_name "target_name", "action",
          signedBy "signer_names";
      ...
  };
```

Omit to allow any principal

Permission Entry

- Begins with `permission`
  - followed by permission class name, target, and optional action

```
grant {
      permission java.lang.RuntimePermission "setFactory";
      permission java.io.FilePermission "example.txt", "read";
  };
```

JEP 411

- JEP 411: Deprecate the Security Manager for Removal
  - Deprecate the Security Manager for removal in a future release. The Security Manager dates from Java 1.0. It has not been the primary means of securing client-side Java code for many years, and it has rarely been used to secure server-side code. To move Java forward, we intend to deprecate the Security Manager for removal in concert with the legacy Applet API (JEP 398).
- Version: Java 17
- https://openjdk.java.net/jeps/411
- Still code that uses it, but new development should avoid.

Lab

1. Create a simple java application that takes two parameters
   a. file name
   b. message
2. Open the file and write the message to the file
3. Run your program and validate you are writing to the file

Lab

4. Run the program now with –Djava.security.manager
5. File write should fail now
6. Create a policy file that would grant your application the permission to write the file
7. Specify your policy file when you run the application with the security manager.

Java Secure Coding Guidelines

Overview

- Oracle provides a "Secure Coding Guidelines for Java SE" document for guidance on making your code more secure.
  - https://www.oracle.com/java/technologies/javase/seccodeguide.html

Isn't Java already secure?

- Java has a lot of built-in security.
- It was designed to provide a restricted environment for executing code with different permission levels.
- The runtime provides automatic memory management and bounds-checking on arrays.
  - Making it highly resistant to the stack-smashing and buffer overflow attacks
- Checks are in place so that unexpected conditions yield predictable behavior
- Etc.

Things Java security cannot do

- Defend against implementation bugs that occur in trusted code
- Therefore, it is recommended that developers adhere to coding guidelines

Guidelines

- The document is divided up into ten major areas of concern
  - Fundamentals
  - Denial of Service
  - Confidential Information
  - Injection and Inclusion
  - Accessibility and Extensibility
  - Input Validation
  - Mutability
  - Object Construction
  - Serialization and Deserialization
  - Access Control
- We will look at a few of the guidelines but you are encouraged to read through the document

Fundamentals

- These are general guidelines that can apply to any programming language.
  - Prefer to have obviously no flaws rather than no obvious flaws
    - Meaning make your code easy to read and understand
  - Design APIs to avoid security concerns
    - Think about security up front
  - Avoid duplication
    - Don't copy and paste (the original form of reuse but better to avoid)
  - Restrict privileges
    - The less access code has, the less damage an attacker can do if the code is compromised

Fundamentals

- Establish trust boundaries
- Minimize the number of permission checks
  - Limit the number of calls to the SecurityManager
  - Return a capability to be used instead
- Encapsulate
- Document security-related information
- Secure third-party code
  - Make sure you stay on top of security patches and such

Denial of Service

- Input should be checked to make sure it will not consume excessive resources
  - Beware of activities that may use disproportionate resources
    - "Zip bombs" - set limits on the decompressed data size
    - Many keys to be inserted into a hash table with the same hash code
    - Ensure that each iteration of a loop makes some progress

Denial of Service

- Release resources in all cases
  - Use try-with-resources

```
public void readFileBuffered(InputStreamHandler handler)
 throws IOException {
     try (final InputStream in = Files.newInputStream(path)) {
         handler.handle(new BufferedInputStream(in));
     }
}
```

Denial of Service

- Resource limit checks should not suffer from integer overflow
  - `current + extra` could overflow yielding a negative number

```
private void checkGrowBy(long extra) {
    if (extra < 0 || current + extra > max) {
        throw new IllegalArgumentException();
    }
}
```

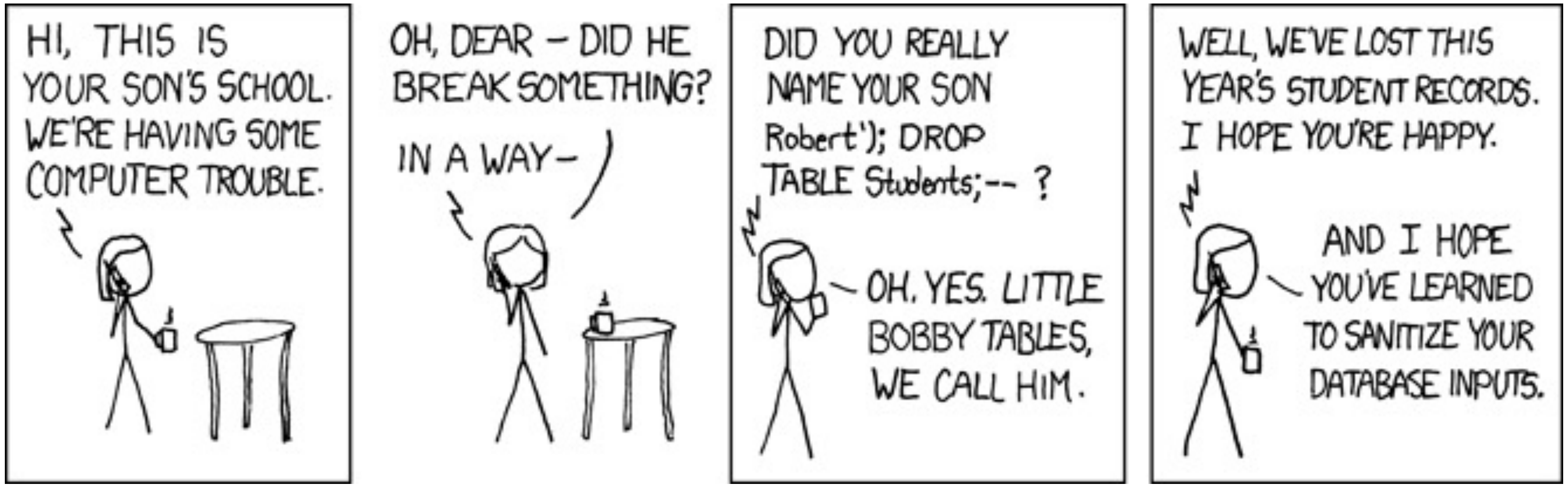
```
private void checkGrowBy(long extra) {
    if (extra < 0 || current > max - extra) {
        throw new IllegalArgumentException();
    }
}
```

Confidential Information

- Purge sensitive information from exceptions
- Do not log highly sensitive information

Injection and Inclusion

- Cause an unanticipated change of control through carefully crafted data passed to the application
  - https://xkcd.com/327


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Injection and Inclusion

- Avoid dynamic SQL

```
String sql = "SELECT * FROM User WHERE userId = ?";
PreparedStatement stmt = con.prepareStatement(sql);
stmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
```

Accessibility and Extensibility

- Limit the accessibility of classes, interfaces, methods, and fields
  - Declare classes and interfaces package-private unless part of a public API
  - Declare members private or package-private unless part of a public API
- Limit the extensibility of classes and methods
  - Declare classes final unless designed for inheritance
  - Prefer composition to inheritance

Input Validation

- Define wrappers around native methods
  - Declare native methods private and expose the functionality through a public Java wrapper method

Mutability

- Prefer immutability for value types
- Create copies of mutable output values

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

Mutability

- Treat passing input to untrusted object as output

```
private final byte[] data;

public void writeTo(OutputStream out) throws IOException {
    // Copy (clone) private mutable data before sending.
    out.write(data.clone());
}
```

Mutability

- Treat output from untrusted object as input

```
private final Date start;
private Date end;

public void endWith(Event event) throws IOException {
    Date end = new Date(event.getDate().getTime());
    if (end.before(start)) {
        throw new IllegalArgumentException("...");
    }
    this.end = end;
}
```

Object Construction

- Avoid exposing constructors of sensitive classes
  - Use static factories instead
    - Allows the return of objects without making their classes public

```
Date d = Date.from(instance);
BigInteger p = BigInteger.valueOf(Integer.MAX_VALUE);
```

Serialization and Deserialization

- There are six guidelines in this section, but the main takeaway is:
  - Deserialization of untrusted data is inherently dangerous and should be avoided

Access Control

- The guidelines in this section deal with elevate or restrict permissions though use of the SecurityManager
  - Since Java has deprecated the SecurityManager we will not review this section

Summary

- Java has many features that make it a good choice for building secure systems
- However, preventing flaws from being introduced by developers cannot be done by the platform
- Coding guidelines need to be reviewed and used

Java Security APIs

- The JDK provides many security APIs
  - Cryptography
  - Authentication
  - Secure communication
  - Access control

Java Cryptography Architecture (JCA)

What is JCA

- Set of APIs that implements
  - Digital signatures
  - Message digests
  - Certificates

Why JCA

- API's provided in JCA allow developers to integrate security into their application without having to work directly with complicated security algorithms.

Design principles

- Implementation independence
- Implementation interoperability
- Algorithm extensibility

Implementation independence

- Security providers support a standard interface and are plugged into the JDK.
- Applications request security services from the JDK.
- Multiple independent providers can be used by an application.

Implementation interoperability

- Applications need not be bound to specific provider.

Algorithm extensibility

- JDK provides a basic set of security services.
- Applications however may rely on proprietary services or emerging standards.

java.security.Provider

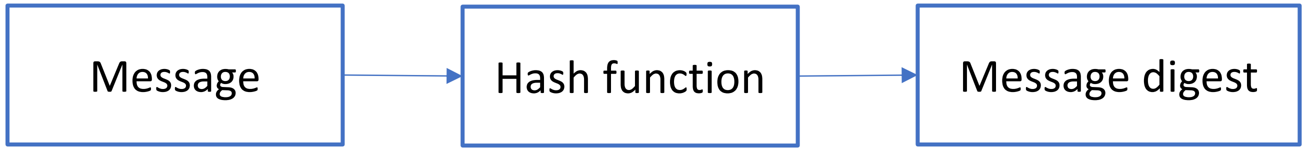
- Classes that return providers
    - CertificateFactory
    - KeyFactory
    - KeyPairGenerator
    - KeyStore
    - MessageDigest
    - Policy
    - SecureRandom
    - Signature

JCA - Terms

- Cryptography
  - Mathematical algorithms
  - Securing of digital data
- Cryptanalysis
  - Breaking the ciphertext
  - Used to test cryptographic strengths
- Cryptography Primitives
  - Encryption
  - Hash functions
  - Message Authentication codes (MAC)
  - Digital Signatures

Hash functions

- A hash function takes an arbitrarily long numeric input and converts it to a fixed-size compressed numeric value.
- The new value is known as a "message digest" or "hash value".



java.security.MessageDigest

- JCA provides a MessageDigest class that can be used with different algorithms to produce message digests (hash values)

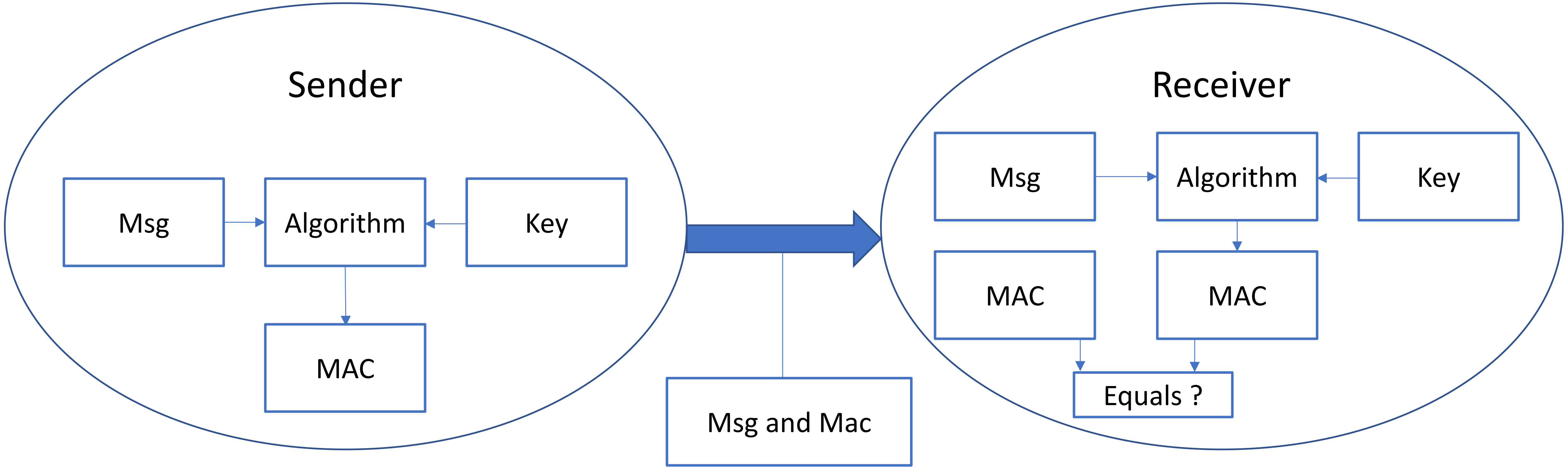
```
String message = ...;

// Creating the MessageDigest object
MessageDigest md = MessageDigest.getInstance("SHA-256");

// Passing data to the created MessageDigest Object
md.update(message.getBytes());

// Compute the message digest
byte[] digest = md.digest();
System.out.println(digest);
```

Message Authentication Code – (MAC)

- Message authentication can be provided by using a symmetric cryptographic technique known as a MAC algorithm
- The MAC is generated by hashing the message using a one-way hash function with a shared key.
- The MAC is transmitted along with the message.
- In essence the MAC is an encrypted checksum on the message.
- The receiver of the message also generates the MAC and compares it with the MAC sent.

MAC usage

- Both sender and receiver have the same key.
  - Sender generates the MAC and sends it with the message
  - Receiver also generates the MAC and compares to the MAC sent



MAC – Key generation

- Developers can use javax.crypto.KeyGenerator to generate a key to use for generating MACs.
- Keys are generated using a random seed.
    - Make sure and use java.security.SecureRandom to get a cryptographically strong random number.

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
SecureRandom secRandom = new SecureRandom();

keyGen.init(secRandom);

Key key = keyGen.generateKey();
```

MAC – Key generation

- A javax.crypto.Mac can then be used to generate a MAC with the provided key.

```
Mac mac = Mac.getInstance("HmacSHA256");

mac.init(key);

String msg = new String("Hi how are you");

byte[] bytes = msg.getBytes();

byte[] macResult = mac.doFinal(bytes);
```

Cipher system

- Infrastructure to provide information security services using cryptographic techniques.
- Components:
    - Plaintext
    - Encryption Algorithm
    - Ciphertext
    - Decryption Algorithm
    - Encryption Key
    - Decryption Key

Key Encryption

- Types:
  - Symmetric
    - One key shared
  - Asymmetric
    - Two keys public and private

Symmetric Key Generation

- javax.crypto.KeyGenerator is used to create a symmetric key.

```
var keyGen = KeyGenerator.getInstance("AES");

keyGen.init(new SecureRandom());

var key = keyGen.generateKey();
```

Asymmetric Key Generation

- java.security.KeyPairGenerator is used to create an asymmetric key pair.

```
var keyGen = KeyPairGenerator.getInstance("RSA");

keyGen.initialize(2048);

var pair = keyGen.generateKeyPair();
```

Encryption

- Symmetric or Asymmetric keys can be used along with javax.crypto.Cipher to encrypt data.

```
byte[] encrypt(String msg, Key key, Cipher cipher) throws Exception {
    cipher.init(Cipher.ENCRYPT_MODE, key);

    byte[] input = msg.getBytes();

    return cipher.doFinal(input);
  }
```

Decryption

- Symmetric or Asymmetric keys can be used along with javax.crypto.Cipher to decrypt data.

```
byte[] decrypt(byte[] cipherText, Key key, Cipher cipher) throws Exception {
   cipher.init(Cipher.DECRYPT_MODE, key);

   return cipher.doFinal(cipherText);
 }
```

Symmetric Key Encryption/Decryption

- Cipher instance must match algorithm.

```
var cipherText = encrypt(
     "Goodbye to you!",
     key,
     Cipher.getInstance("AES/ECB/PKCS5Padding")
   );

   var decipheredText = decrypt(
     cipherText,
     key,
     Cipher.getInstance("AES/ECB/PKCS5Padding")
   );
```

Symmetric Key Encryption/Decryption

- Cipher instance must match algorithm.

```
var cipherText = encrypt(
     "Goodbye to you!",
     key,
     Cipher.getInstance("DES/ECB/PKCS5Padding")
   );

   var decipheredText = decrypt(
     cipherText,
     key,
     Cipher.getInstance("DES/ECB/PKCS5Padding")
   );
```

Asymmetric Key Encryption/Decryption

- Cipher instance must match algorithm.

```
var cipherText = encrypt(
     "Hello there!",
     pair.getPrivate(),
     Cipher.getInstance("RSA/ECB/PKCS1Padding")
   );

   var decipheredText = decrypt(
     cipherText,
     pair.getPublic(),
     Cipher.getInstance("RSA/ECB/PKCS1Padding")
   );
```

KeyStore

- Java uses a KeyStore to hold private keys, certificates, and public keys
- The key store can be accessed in code or through the keytool

Keytool

- The keytool can be used to generate private/public key pairs, that can be used to sign code

```
keytool -genkey -alias wile_e_coyote -keystore acme.store -keyalg RSA
```

Sign a JAR

- You can sign a jar file with your private key from the keystore

```
jarsigner -keystore acme.store -signedjar sig_acme.jar acme.jar wile_e_coyote
```

Export certificate

- Use the keytool to export your public certificate to give to your client

```
keytool -export -keystore acme.store -alias wile_e_coyote -file ACME.cer
```

Import certificate

- Client will import the certificate into their keystore

```
keytool -import -alias wile -file ACME.cer -keystore roadrunner.store
```

Policy file

- Create a policy file to use the keystore with the certificate and provide permissions to the signed codebase

```
keystore "roadrunner.store";
keystorePasswordURL "filewithpass";

grant signedBy "wile", codebase "file:sig_acme.jar" {
    permission java.util.PropertyPermission "java.home", "read";
  };
```

Use the acme.jar

- Because the jar is signed, we are confident that is comes from ACME
- We can also now run it under a SecurityManger to restrict its access

```
java -Djava.security.manager -Djava.security.policy=roadrunner.policy
-cp acme.jar acme.Message
```

Lab – Crypto

1. Create an application that takes a message as a command line param
2. Encrypts that argument with a symmetric algorithm
3. Outputs the cipher text
4. Decrypts the message
5. Outputs the deciphered text
6. Change the program to use an asymmetric algorithm

Java Authentication and Authorization Service (JAAS)

What is JAAS?

- As the name indicates JAAS serves two purposes
  - Authentication – reliably determine who is executing Java code in applications, applets, beans, or servlets.
  - Authorization – ensure users have access rights to perform requested actions.

Authentication

- JAAS works in a pluggable fashion so that applications can remain independent of underlying authentication technologies.
- Developers instantiate a `LoginContext` with a `Configuration` which is used to determine authentication technologies and correct `LoginModule` to use.

LoginContext

- Instantiate a LoginContext
    - Specify the Configuration to use
    - CallbackHandler provided to allow underlying security services the ability to interact with a calling application

```
LoginContext lc = null;

lc = new LoginContext("Sample", new MyCallbackHandler());
```

login

- Call login to authenticate user

```
try
{
  lc.login();
  break; // no exception, authentication succeeded

} catch (LoginException le) {
  System.err.println("Authentication failed:");
  ...
}
```

CallbackHandler

- Implement a CallbackHandler to interact with the LoginModule

```
class MyCallbackHandler implements CallbackHandler {
  ...
}
```

Callback

- Several Callback types
    - Each represents different communication with application.
        - NameCallback
        - PasswordCallback
        - TextOutputCallback
        - Others

Login Prompts

- LoginModule calls your CallbackHandler with an array of Callbacks for you to handle.

```
for (int i = 0; i < callbacks.length; i++) {
  if (callbacks[i] instanceof TextOutputCallback) {
    // display the message according to the specified type
  } else if (callbacks[i] instanceof NameCallback) {
    // prompt the user for a username
  } else if (callbacks[i] instanceof PasswordCallback) {
    // prompt the user for sensitive information
  } else {
        throw new UnsupportedCallbackException(...);
  }
}
```

NameCallback

- Gather the login name from the user and provide it back to the security services though the NameCallback

```
 } else if (callbacks[i] instanceof NameCallback) {
     // prompt the user for a username
     NameCallback nc = (NameCallback) callbacks[i];

     System.err.print(nc.getPrompt());
     System.err.flush();
     nc.setName(
       (new BufferedReader(new InputStreamReader(System.in))).readLine()
     );
}
```

PasswordCallback

- Gather the password from the user and provide it back to the security services through the PasswordCallback

```
} else if (callbacks[i] instanceof PasswordCallback) {
     // prompt the user for sensitive information
     PasswordCallback pc = (PasswordCallback) callbacks[i];
     System.err.print(pc.getPrompt());
     System.err.flush();
     pc.setPassword(System.console().readPassword());
}
```

LoginModule

- Interface implemented by security service

```
public interface LoginModule {
  void initialize(Subject subject, CallbackHandler callbackHandler,
                Map<String,?> sharedState,
                Map<String,?> options);
  boolean login() throws LoginException;
  boolean commit() throws LoginException;
  boolean abort() throws LoginException;
  boolean logout() throws LoginException;
}
```

SampleLoginModule

- Example of a simple login method

```
public class SampleLoginModule implements LoginModule {
  public boolean login() throws LoginException {
   ...
   // instantiate callback array to get info from application
   Callback[] callbacks = new Callback[2];
   callbacks[0] = new NameCallback("user name: ");
   callbacks[1] = new PasswordCallback("password: ", false);

   // send callbacks to callback handler
   callbackHandler.handle(callbacks);
   username = ((NameCallback)callbacks[0]).getName();
   char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();

   // compare returned values with known username/password
   ...
```

Authorization

- JAAS can also be used for authorization
    - existing security policy-based Java security architecture is extended

Adding principals

- The `LoginModule` adds principals to the `Subject` at authentication

```
public class SampleLoginModule implements LoginModule {
  public boolean commit() throws LoginException {
    ...
    // add a Principal (authenticated identity) to the Subject
    // assume the user we authenticated is the SamplePrincipal
    userPrincipal = new SamplePrincipal(username);
    if (!subject.getPrincipals().contains(userPrincipal))
        subject.getPrincipals().add(userPrincipal);
   ...
```

Execute a privileged action

- Get the `Subject` from the `LoginContext` and execute a privileged action with the `Subject`

```
Subject mySubject = lc.getSubject();

var action = new SampleAction();
Subject.doAsPrivileged(mySubject, action, null);
```

PrivilegedAction

- Implement `PrivilegedAction`

```
public class SampleAction implements PrivilegedAction {
  public Object run() {
    System.out.println(
      "\nYour java.home property: " + System.getProperty("java.home")
    );
    ...
  }
}
```

Add grants to security.policy

- A policy file will need to be provided
  - allow application
    - to create a `LoginContext`
    - read/write file descriptors
    - invoke `doAsPrivileged`
  - allow the `LoginManager` to modify principals
  - provide authorization to the authenticated user

Application grants

- Grants needed by the application

```
grant codebase "file:./Application.jar" {
   permission javax.security.auth.AuthPermission "createLoginContext.Sample";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
   permission java.lang.RuntimePermission "writeFileDescriptor";
   permission java.lang.RuntimePermission "readFileDescriptor";
};
```

LoginManager grants

- Grants needed by the `LoginManager`

```
grant codebase "file:./MyLoginManager.jar" {
    permission javax.security.auth.AuthPermission "modifyPrincipals";
};
```

User grants

- Grants needed by the Subject in order to perform privileged action

```
grant codebase "file:./SampleAction.jar", Principal
sample.principal.SamplePrincipal "testUser" {
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.util.PropertyPermission "user.dir", "read";
    permission java.io.FilePermission "config.txt", "read";
};
```

SecurityManager

- The application will then need to be run with a `SecurityManager`

```
java -classpath ...
  -Djava.security.manager
  -Djava.security.policy==custom_security.policy
  -Djava.security.auth.login.config==custom_jaas.config
   sample.Application
```

Summary

- JAAS is a set of APIs that work in a pluggable way to allow applications to remain independent of authentication technologies they use.
- JAAS also provides a link into the policy based Java security architecture.

JAAS Demo

- Demo of JAAS code

JSON Web Token (JWT)

What is JWT?

- A way to securely transmit information between parties as a JSON object that is compact and self-contained.

Where is JWT used?

- The most common scenario for using JWT is authorization.
  - Once the user is logged in a JWT will be passed with each request allowing the user to access resources permitted by the token.
  - JWT is widely used with Single Sign On.

What do JWTs look like?

- An encoded representation of a JSON object

JSON

```
{
  "header":{
    "alg":"HS256"
    "typ":"JWT"
  }
  "payload":{
    "sub":"9876543210"
    "name":"Wile E Coyote"
    "iat":1516239022
   }
}
```

JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiI5ODc2NTQ
zMjEwIiwibmFtZSI6IldpbGUgRSB
Db3lvdGUiLCJpYXQiOjE1MTYyMzk
wMjJ9.1BCa8CfrDFXMsTxpm1VSVj
Kt5fkLGzN1P2IFvcTSIpc
```

What is the structure of a JWT?

- JWT is made up of three parts separated by '.'
    - xxxxxxxxxx.yyyyyyyyyy.zzzzzzzzzz
    - Header
    - Payload
    - Signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiI5ODc2NTQzMjEwIiwibmFtZSI6Ild
pbGUgRSBDb3lvdGUiLCJpYXQiOjE1MTYyMzkwMj
J9.
1BCa8CfrDFXMsTxpm1VSVjKt5fkLGzN1P2IFvcT
SIpc
```

JWT Header

- The header is made up of two values
  - Algorithm
    - The algorithm used for signing
  - Type
    - JWT
- The JSON is then Base64Url encoded

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
eyJhbGciOiJIUzI1NiIsI
nR5cCI6IkpXVCJ9
```

JWT Payload

- The payload contains the claims

```
{
  "sub":"9876543210"
  "name":"Wile E Coyote"
  "iat":1516239022
}
```

- The JSON is then Base64Url encoded

```
eyJzdWIiOiI5ODc2NTQzM
jEwIiwibmFtZSI6IldpbG
UgRSBDb3lvdGUiLCJpYXQ
iOjE1MTYyMzkwMjJ9
```

Standard JWT claims

- iss (issuer): Issuer of the JWT
- sub (subject): Subject of the JWT (the user)
- aud (audience): Recipient for which the JWT is intended
- exp (expiration time): Time after which the JWT expires
- nbf (not before time): Time before which the JWT must not be accepted for processing
- iat (issued at time): Time at which the JWT was issued; can be used to determine age of the JWT
- jti (JWT ID): Unique identifier; can be used to prevent the JWT from being replayed (allows a token to be used only once)

Public and Private claims

- Public claims
  - Should be defined with https://www.iana.org/assignments/jwt/jwt.xhtml
  - Or use a URI with collision resistant namespace
- Private claims
  - Anything agreed on between parties

JWT Signature

- The signature is created but using the specified algorithm on the Base64Url encoded header and playload, and a secret.

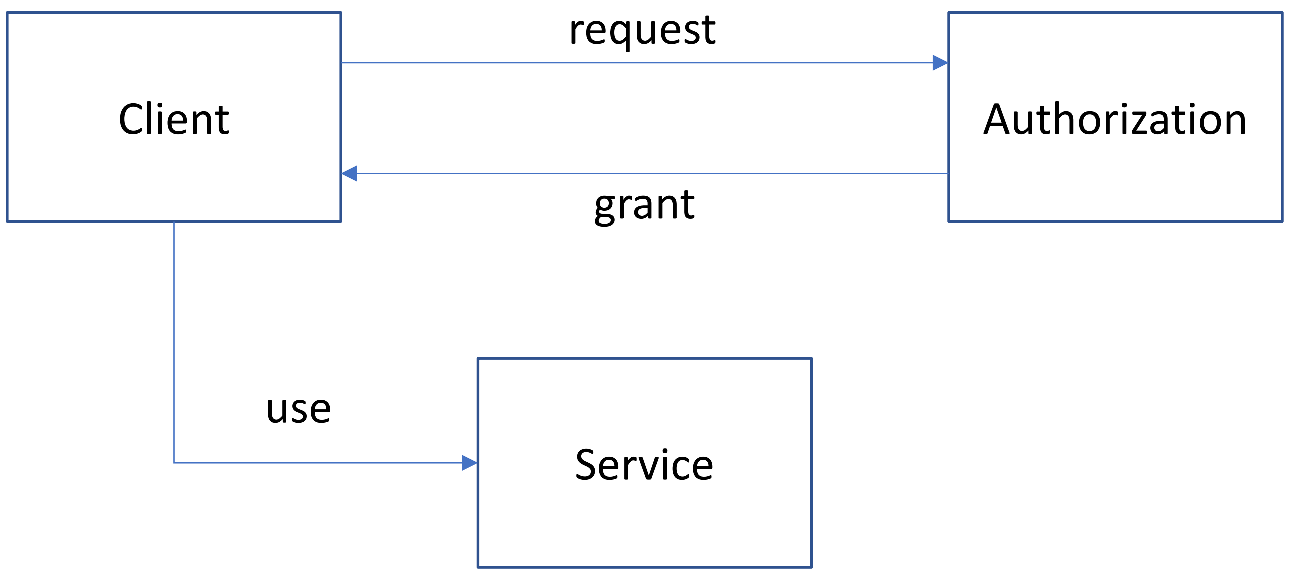
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  theSecret)
```

The signature is then Base64Url encoded

```
1BCa8CfrDFXMsTxpm1VSVjKt5fkLGzN1P2IFvcTSIpc
```

Using JWT

- Typical use is to allow clients to access resources from a service through an API



Java JWT

- Library for creating and verifying JSON Web Tokens
  - https://github.com/jwtk/jjwt

Build a JWT

- JJWT uses fluent interfaces for ease of use.

```
var key = Keys.secretKeyFor(SignatureAlgorithm.HS256);

String jws = Jwts.builder()
               .setSubject("Joe")
               .signWith(key)
               .compact();
```

Read a JWT

- Claims can be extracted easily

```
Jws<Claims> jws;

try {
    jws = Jwts.parserBuilder()  // (1)
    .setSigningKey(key)         // (2)
    .build()                    // (3)
    .parseClaimsJws(jwsString); // (4)

    // trust the JWT

catch (JwtException ex) {       // (5)

    // we *cannot* trust the JWT
}
```

Resources

- https:///jwt.io
- https://github.com/jwtk/jjwt
- https://stormpath.com/blog/jwt-java-create-verify

Lab - JWT

- In this lab we will create a basic app with maven, include the JJWT dependances and create a JWT token

1. Create your app using maven

```
mvn archetype:generate \
 -DgroupId=com.example \
 -DartifactId=jwtapp \
 -DarchetypeArtifactId=maven-archetype-quickstart \
 -DinteractiveMode=false
```

Lab - JWT

2. Make sure you have the compiler set to correct java version in pom.xml

```
<properties>
  <java.version>16</java.version>
  <maven.compiler.source>16</maven.compiler.source>
  <maven.compiler.target>16</maven.compiler.target>
...
```

Lab - JWT

3. Add the JWT dependencies

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.2</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.2</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.2</version>
  <scope>runtime</scope>
</dependency>
```

Lab - JWT

4. Generate a token using JJWT and print it out
5. Run your App

```
mvn clean install
mvn exec:java -Dexec.mainClass=com.example.App
```

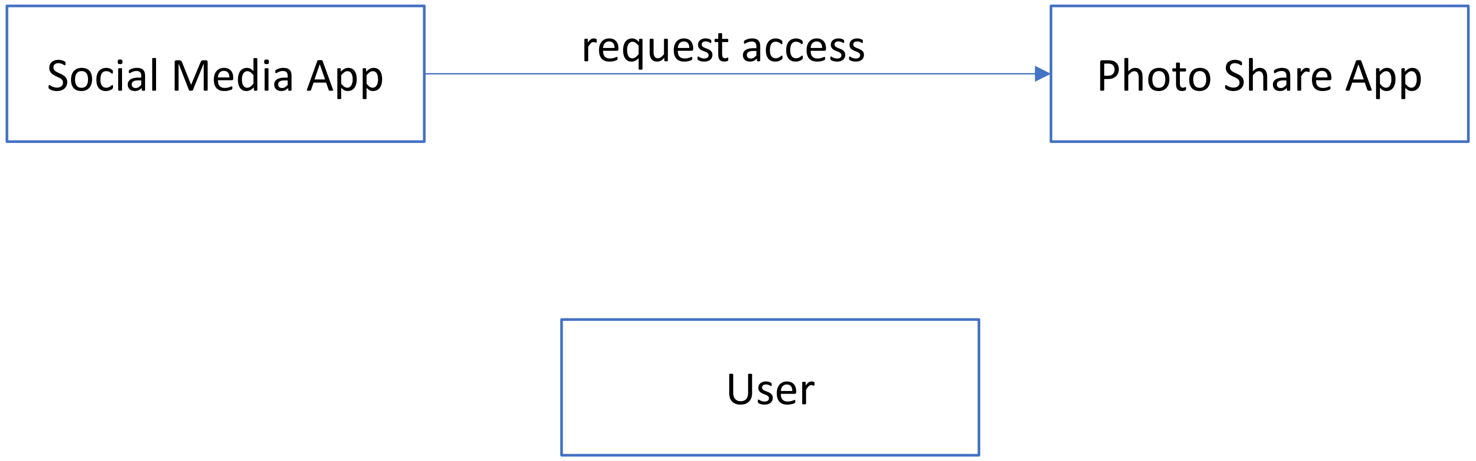
OAuth

What is OAuth?

- A protocol that allows services to acquire authorization information from another service.
- Most widely used version is 2.0
- Often referred to as OAuth2
- The term OAuth is used to mean OAuth2 in this lecture
- Example:
  - You want to have your social media site update your timeline with photos from your photo sharing site
  - The social media site will need authorization to access your photos.

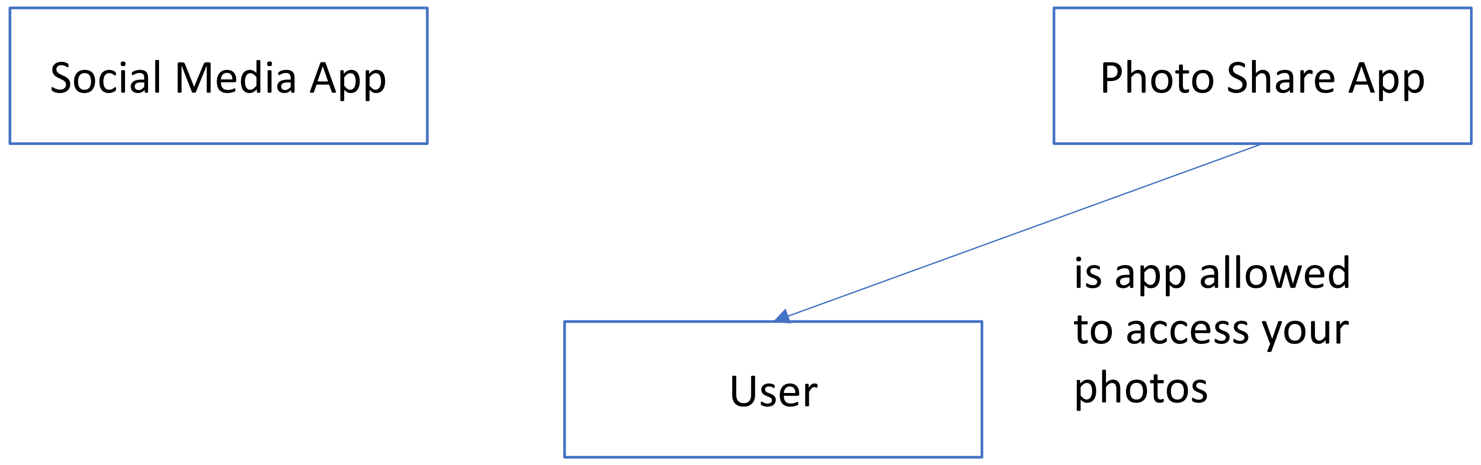
Sample flow

- Social Media App requests authorization from Photo Share App



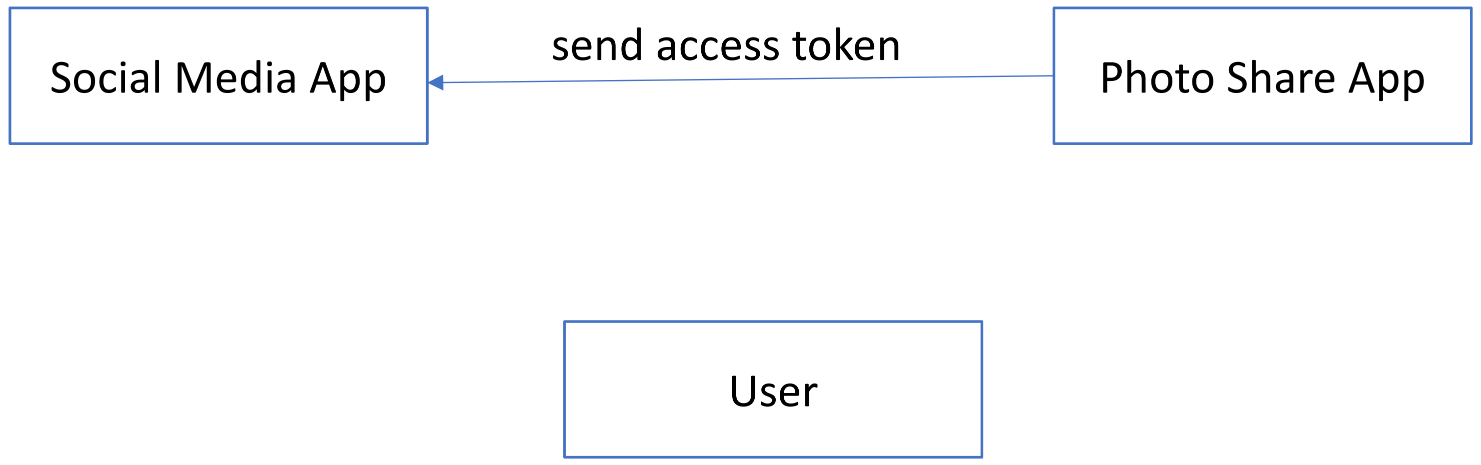
Sample flow

- Photo Share App asks user to authorize Social Media App



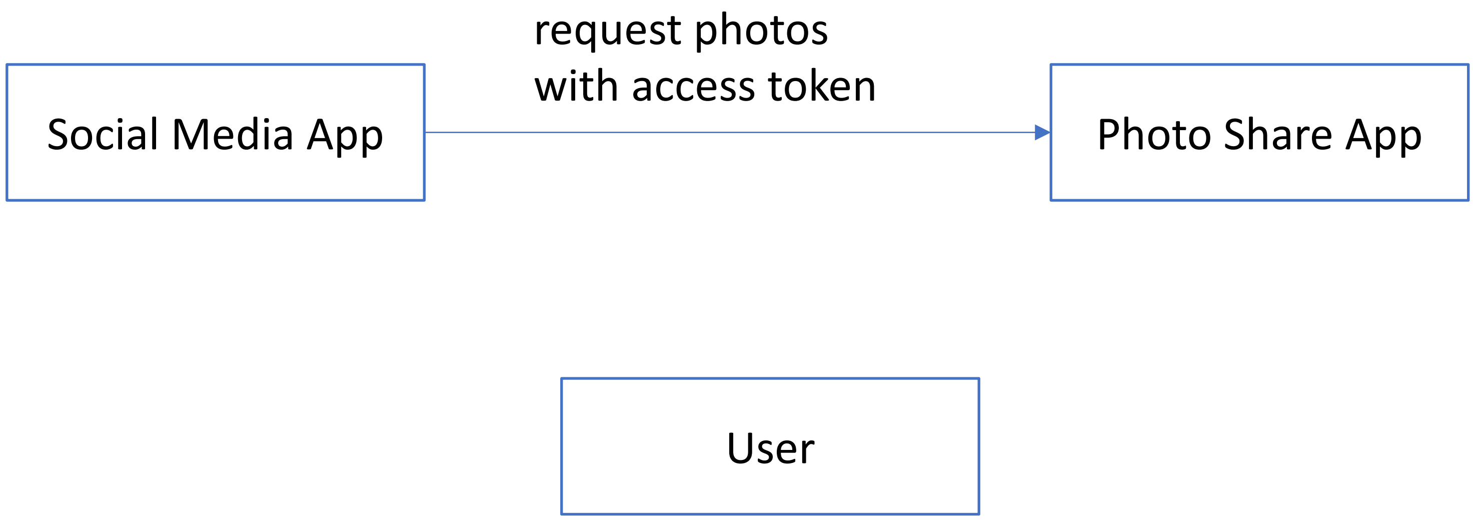
Sample flow

- If user says yes, then Photo Share App issues Social Media App an authorization token it can use to request photos on behalf of the user



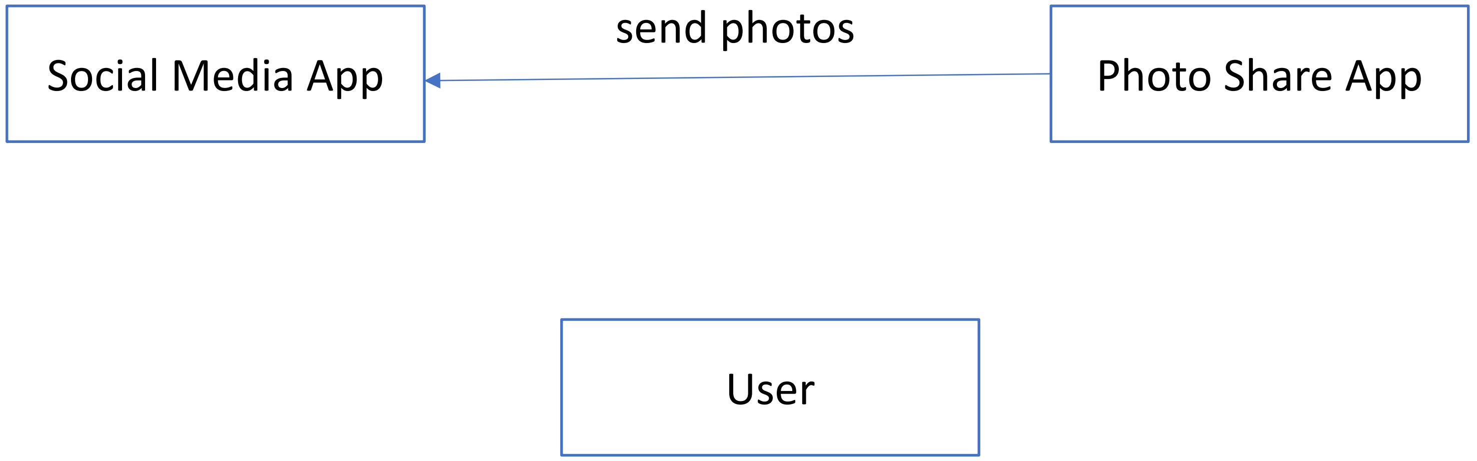
Sample flow

- Social Media App uses token to request photos



Sample flow

- Photo Share App returns photos to Social Media App

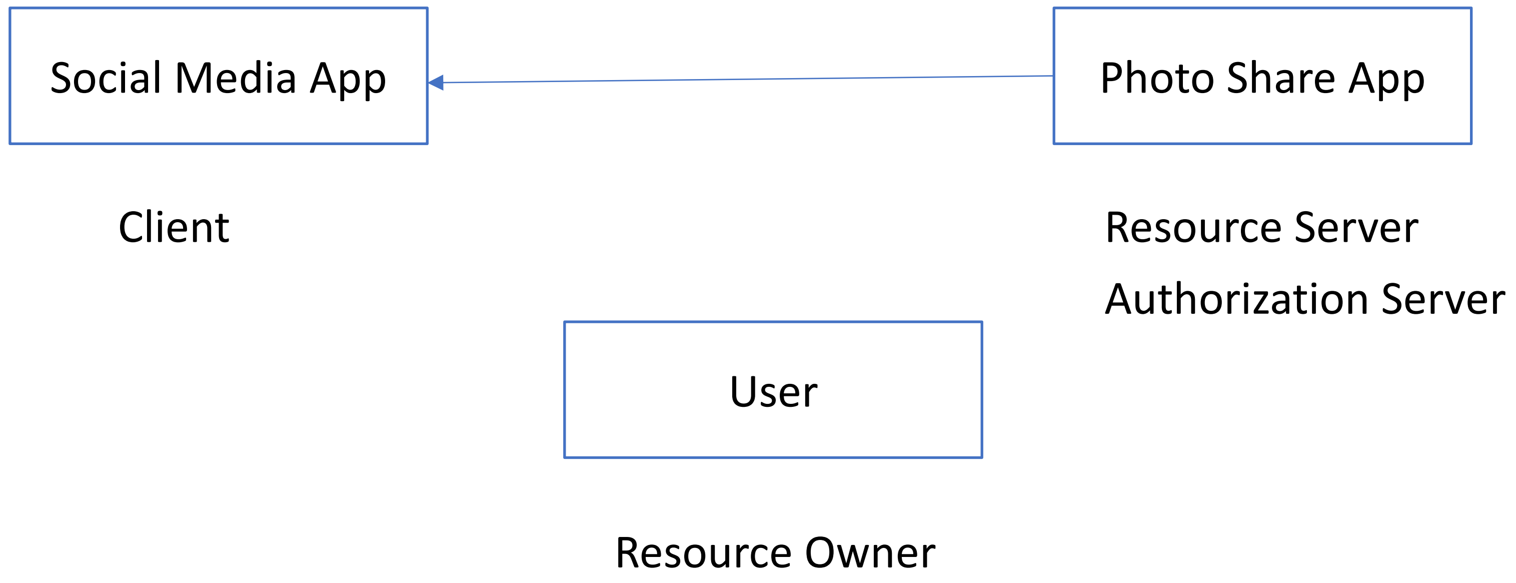


OAuth terms

- Resource (aka Protected Resource)
- Resource Owner
  - an entity capable of granting access to protected resource
- Resource Server
  - server hosting the protected resource
- Authorization Server
  - server that issues access tokens to client
- Client
  - application making protected resource requests of the resource server on behalf of resource owner and with its authorization

Sample flow

- Photo Share App returns photos to Social Media App



OAuth Flows

- Authorization Code Flow
- Implicit Flow
- Client Credentials Flow

Authorization Code Flow

1. Resource owner requests client to access resource from authorization server
2. Client asks authorization server for access to resource
3. Authorization server asks resource owner if it is ok
4. Resource owner approves
5. Authorization server sends a short lived auth token to client
6. Client uses auth token to request an access token from the authorization server
7. Authorization server sends client access token
8. Client uses access token to request resource of resource server
9. Resource server sends resource

Implicit Flow

1. Resource owner requests client to access resource from authorization server
2. Client asks authorization server for access to resource
3. Authorization server asks resource owner if it is ok
4. Resource owner approves
5. Authorization server sends client access token
6. Client uses access token to request resource of resource server
7. Resource server sends resource

Implicit Flow

- Not as secure as Authorization Code Flow
- Short lived tokens often used
- Used with JavaScript apps

Client Credentials Flow

- Used to authorize trusted clients
  1. Known client request access token from authorization server
  2. Authorization server sends access token
  3. Client uses access token to request resource from resource server
  4. Resource server sends resource to client

OAuth for authentication

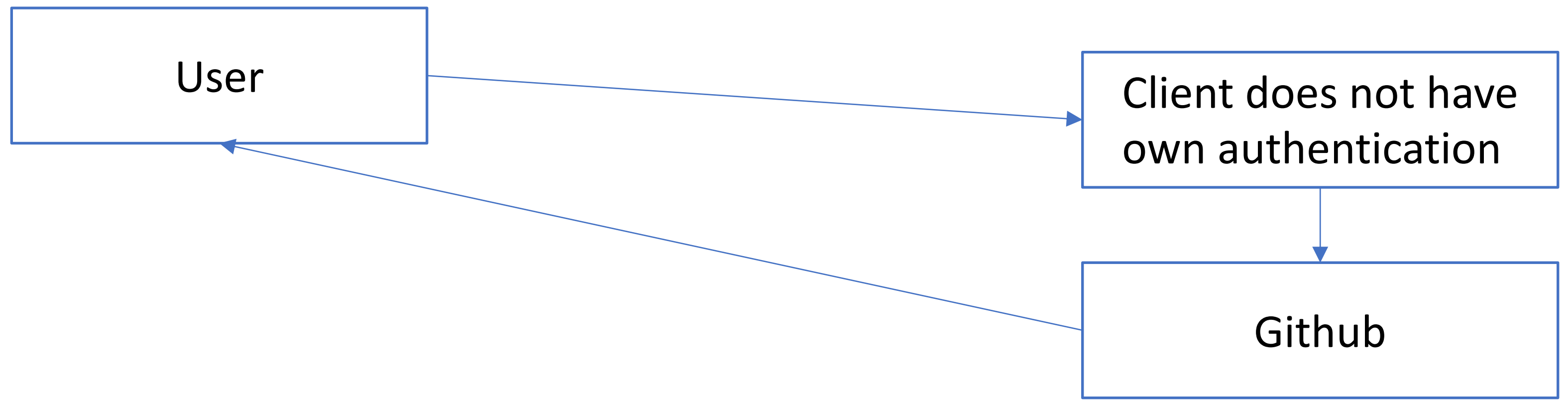
- Although OAuth is designed to provide authorization between services, it is also often used to provide authentication for a service.

Authentication flow

- Resource is user profile information
- Client requests authorization from service like GitHub
- Client does not allow access to itself unless able to get authorization token from GitHub

Authentication flow

1. User tries to access client
2. Client requests authorization token from GitHub
3. GitHub request authorization from user
4. GitHub returns successful authorization to client
5. Client allows access to user



Summary

- OAuth protocol that allows services to acquire authorization information from another service.
- We looked at three authorization flows that are part of the specification.
  - Authorization Code Flow
  - Implicit Flow
  - Client Credentials Flow
- OAuth can also be for authentication by restricting access unless the user can authorize access to an authorization service like GitHub.

Fuzzing

What is fuzzing?

- The automatic generation and execution of tests with the goal of finding security vulnerabilities

Why fuzzing?

- It has proven to be an effective way to find security bugs in software.
- If your app processes untrusted inputs, you should use fuzzing.
- If you deal with large, complex data parsers, fuzzing is very effective.
- Vulnerabilities often missed by static program analysis and code inspection can be found with fuzzing.

Type of fuzzing

- Three main types of fuzzing
  - Blackbox random fuzzing
  - Grammar-based fuzzing
  - Whitebox fuzzing

Blackbox random fuzzing

- Randomly mutates well-formed program inputs
- Runs program with mutated inputs to try and trigger bugs
- Effective in finding bugs in programs that have not been fuzzed before

Grammar-based fuzzing

- Fuzzer allows an input grammar to be supplied by the user
- Generates many new inputs, each satisfying the constraints supplied
- Allows user’s creativity and expertise to guide fuzzing

Whitebox fuzzing

- Pioneered by Microsoft Research
  - https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/
- Consists of symbolically executing the program, gathering constraints on inputs by looking at conditional branches encountered, and negating the constraints to exercise different execution paths

Resources

- https://github.com/CodeIntelligenceTesting/jazzer
- https://github.com/microsoft/onefuzz
- https://google.github.io/oss-fuzz/

Lab

- Install Jazzer and run the examples
- Go to https://github.com/CodeIntelligenceTesting/jazzer
- Following the instructions to install Jazzer on your system and run the examples

API vulnerabilities

- https://owasp.org/www-project-api-security/
- By 2021 APIs will provide a larger surface area for attacks than the UI for 90% of web applications. Gartner.

Microservices

- Modern web applications are often made up of several smaller applications or micro services.
- These micro services will provide an API to communicate with the front-end application.

REpresentational State Transfer (REST)

- Common way for microservices to expose an API
- Creating REST services in Java
    - JAX-RS
    - SpringREST

Application security

- Authentication
- Authorization
- Principals
- Grants (granted authorizations)
- Roles (groups)

Spring Security

What is Spring Security?

- Spring Security is application-level security, it allows you to add security to your Spring web application or API
- Things like
  - User/password authentication
  - SSO
  - App level authorization
  - Intra app authorization (OAuth)
  - Microservice security (tokens, JWT)

Concepts

- Authentication
- Authorization
- Principal
- Granted Authority
- Roles

Authentication

- Authentication answers the question "Who are you?"
- Once authenticated the application trusts you are who you say you are

Authorization

- Authorization answers the question "What are you allowed to do?"
- Once authorized the application will allow you to access only those parts of the application you have been authorized for.

Principal

- Represents the logged in user in the application

Granted Authority

- Permissions granted to the user
  - fine grain control e.g., read or write access to a resource

Role

- Coarse grained access granted to user
  - e.g., USER or ADMIN

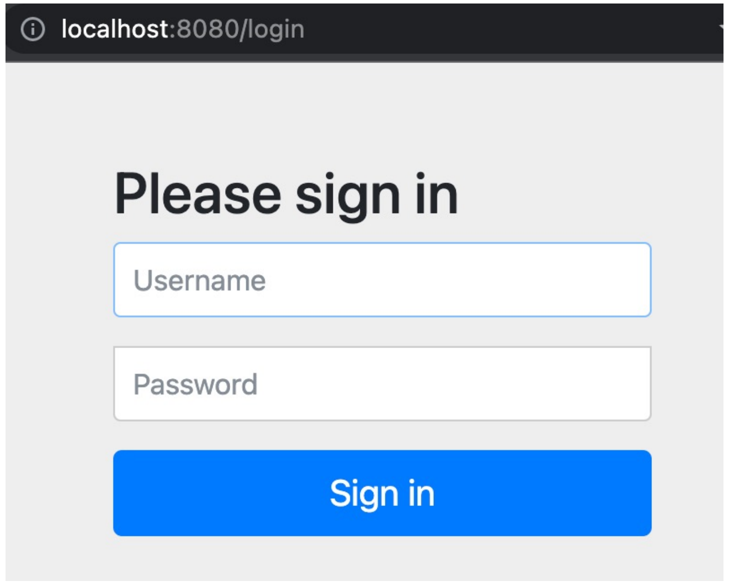
Using Spring Security

- Adding Spring Security to your application is as easy as adding a dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Deny by default

- Spring Security uses deny by default, so once you add the dependency all access to your web app will be restricted
  - If you try to access any endpoint you will receive the following



Default login

- Spring provides a default login
  - username = user
  - password = <generated with each start>
    - e.g., Using generated security password: 0f1ff3c9-dd6d-437f-b62f-d0d1c654ac9b

Configure authorization manager

- Provide a class that extends `WebSecurityConfigurerAdapter`
    - org.springframework.security.config.annotation.web.configuration
- Override `configure`

```
@EnableWebSecurity
public class SecurityConfigurer extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
      .inMemoryAuthentication()
      .withUser("bradley")
      .password("thepass")
      .roles("USER");
  }
}
```

Configure authorization manager

- You will also need to provide a `PasswordEncoder` Bean

```
@EnableWebSecurity
public class SecurityConfigurer extends WebSecurityConfigurerAdapter {
  ...
  @Bean
  public PasswordEncoder getPasswordEncoder() {
      return NoOpPasswordEncoder.getInstance();
  }
}
```

Configure authentication

- Override another configure method in your SecurityConfigurer class
  - `configure(HttpSecurity http)`
  - Also uses fluent interface

```
@EnableWebSecurity
public class SecurityConfigurer extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
      http
        .authorizeRequests()
        .antMatchers("/**")
        .hasRole("ADMIN")
        .and()
        .formLogin();
  }
}
```

Configurating authentication with OAuth

- Spring also makes it easy to use sites like Github for your apps authentication through OAuth.

Add the OAuth dependency

- You need to add the oauth2-client dependency to your pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

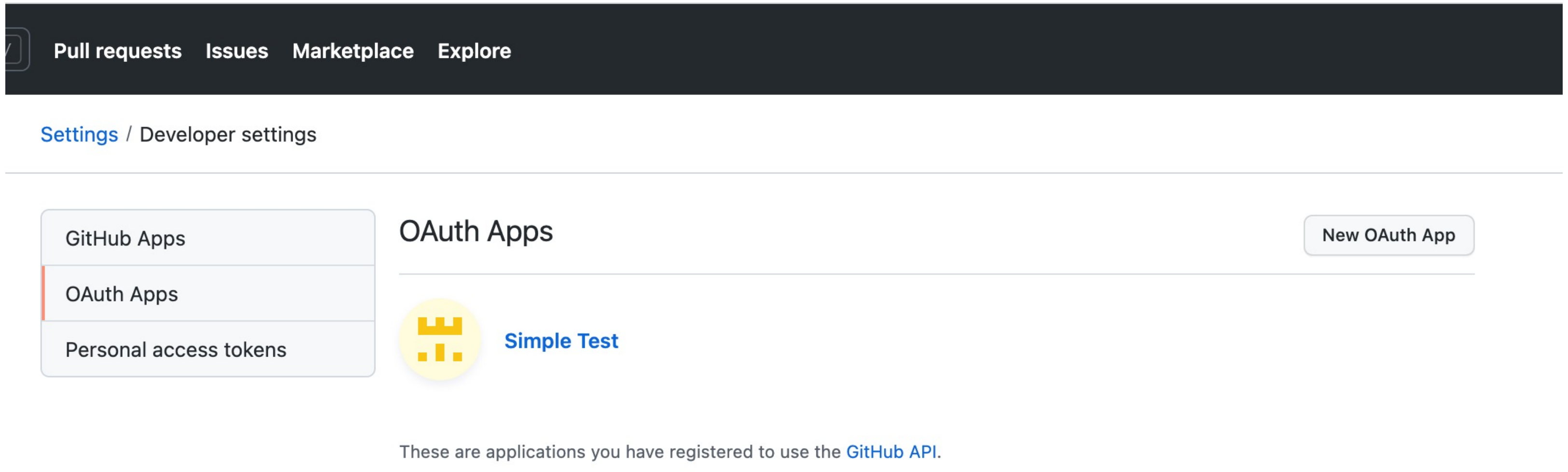
Add the OAuth2 client to application.properties

- Using a yaml file because it is easier to work with the nested properties

```
spring:
 security:
  oauth2:
   client:
    registration:
       github:
        clientId: 274**************
        clientSecret: 3ebc***********************
# ...
```

Where does the client info come from?

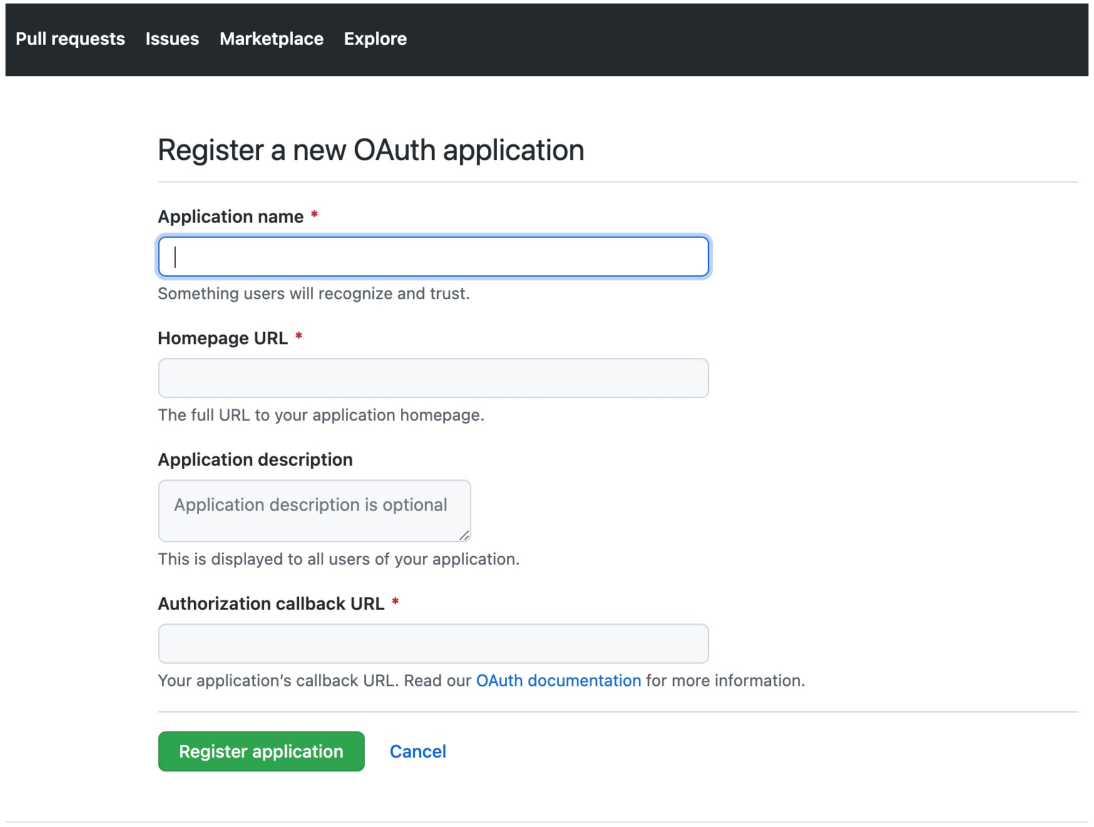
- You register an application with the provider
- For GitHub it is under Settings->Developer Settings



These are applications you have registered to use the GitHub API.

Create a new OAuth app

- Click on "New OAuth App"



Register a new OAuth application

Application name *

Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

Application description is optional

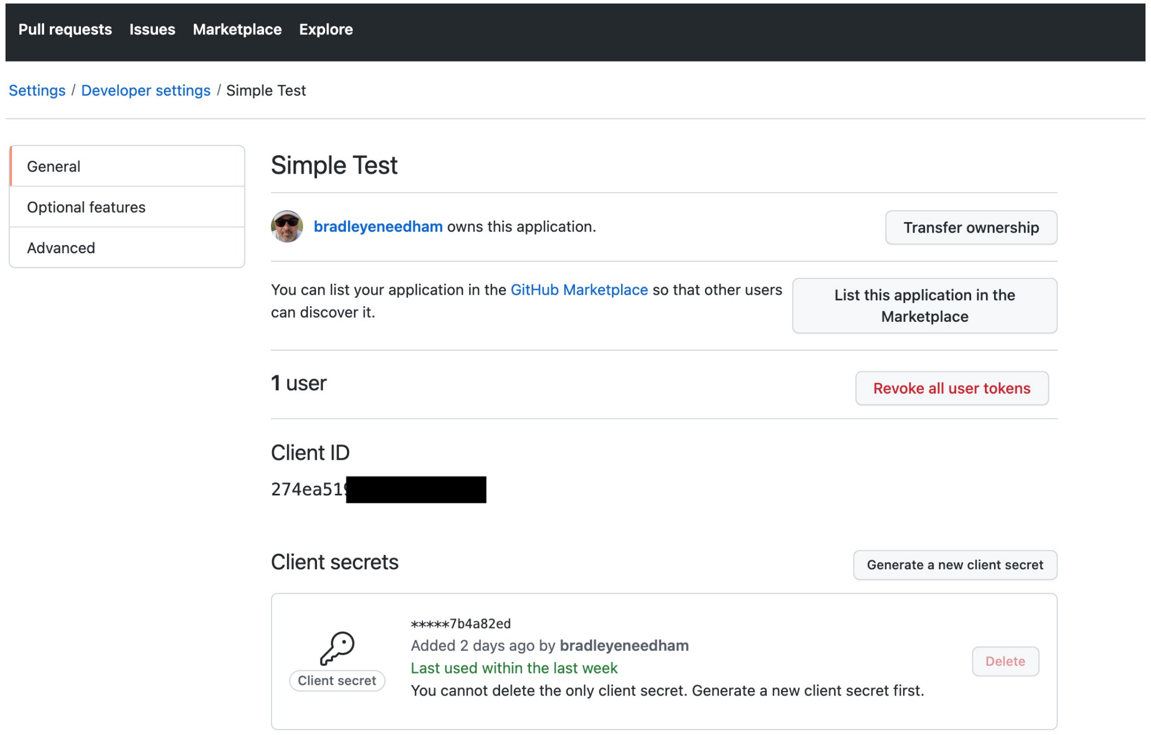
This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our OAuth documentation for more information.

Client id and secret

- The client id and secret will be displayed
- Copy the information into your app.



Simple Test

bradleyeneedham owns this application.

You can list your application in the GitHub Marketplace so that other users can discover it.

1 user

Client ID

274ea519[redacted]

Client secrets

*****7b4a82ed
Added 2 days ago by bradleyeneedham
Last used within the last week
You cannot delete the only client secret. Generate a new client secret first.

Summary

- Spring Security provides an easily configurable mechanism to accomplish application-level security

Lab

- Create a new Spring Boot app
  - https://start.spring.io/
- Create a new GitHub OAuth App
- Add the correct dependencies and properties to your app
- Test logging on to your site

Course complete

Resources

- https://shiro.apache.org/
- https://www.oracle.com/java/technologies/javase/seccodeguide.html
- https://openjdk.java.net/jeps/411
- https://www.upguard.com/blog
- https://docs.oracle.com/en/java/javase/17/security/java-security-overview1.html
- https://docs.oracle.com/en/java/javase/17/security/java-cryptography-architecture-jca-reference-guide.html

Resources

- https://docs.oracle.com/en/java/javase/11/security/jaas-authorization-tutorial.html
- https://docs.spring.io/spring-security/servlet/authentication/jaas.html
- https://stormpath.com/blog/jwt-java-create-verify
- https://github.com/jwtk/jjwt#jws-key
- https://jwt.io/
- https://auth0.com/docs/security/tokens/json-web-tokens/

Resources

- https://www.jstoolset.com/jwt
- https://www.youtube.com/playlist?list=PLqq-6Pq4lTTYTEooakHchTGglSvkZAjnE
  - Spring Security Tutorial
- https://google.github.io/oss-fuzz/
- https://www.code-intelligence.com/blog
- https://github.com/CodeIntelligenceTesting/jazzer
- https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

Resources

- https://github.com/microsoft/onefuzz/
- https://patricegodefroid.github.io/public_psfiles/Fuzzing-101-CACM2020.pdf
- https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/
- https://www.youtube.com/watch?v=Ai3wnnSFC-8
  - Using jazzer
- https://spring.io/guides/tutorials/spring-boot-oauth2/
- https://developer.okta.com/blog/2019/10/30/java-oauth2

Resources

- http://redhat-crypto.gitlab.io/defensive-coding-guide/#chap-Defensive_Coding-Java
- https://www.softwaretestinghelp.com/guidelines-for-secure-coding/
- https://money.cnn.com/2014/02/06/technology/security/target-breach-hvac/index.html
- https://www.upguard.com/blog/principle-of-least-privilege
- https://cwe.mitre.org/data/definitions/660.html
- https://owasp.org/Top10/
- https://www.threatmodelingmanifesto.org/

Resources

- https://cheatsheetseries.owasp.org/
- https://www.fuzzcon.eu/2021/
- https://www.scala-sbt.org/1.x/docs/Faq.html
- https://security.googleblog.com/2021/03/fuzzing-java-in-oss-fuzz.html
- https://www.akana.com/blog/what-is-jwt
- https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf

Resources

- https://owasp.org/www-project-api-security/
- https://www.cve.org/