**Code Correctness: Constructor Invokes Overridable Function Development Mitigation SOP**

Code correctness vulnerabilities occur when an Object API is not used properly or as intended. Code correctness vulnerabilities can be caused by a constructor when it can be overridden by an attacker. When a constructor calls an overridable function, it may allow an attacker to access the this reference before the object is fully initialized, which can in turn lead to a vulnerability.

**Defense Against Code Correctness: Constructor Invokes Overridable Function**

Constructors should not call functions that can be overridden, either by specifying them as final, or specifying the class as final. If this code is only ever needed in the constructor, you could instead use the private access specifier or logic could be placed directly into the constructor of the superclass.

**Examples**

**General Example**

…

class User {

private String username;

private boolean valid;

public User(String username, String password) {

this.username = username;

this.valid = validateUser(username, password);

}

public boolean validateUser(String username, String

password) {

//validate user is real and can authenticate

}

public final boolean isValid() {

return valid;

}

}

…

…

class Attacker extends User {

public Attacker(String username, String password){

super(username, password);

}

public boolean validateUser(String username, String

password) {

return true;

}

}

…

class MainClass {

public static void main(String[] args) {

User hacker = new Attacker(“Evil”, “Hacker”);

if(hacker.isValid()) {

System.out.println(“Attack successful”);

} else {

System.out.println(“Attack failed”);

}

}

}

**Explanation**

Because the function validateUser and the class are not final, it means they can be overridden, and initializing a variable to the subclass that overrides this function would allow the validateUser function to be bypassed. The code above will print “Attack successful” because the Attacker class overrides the validateUser() function that is called from the constructor of the superclass User, and Java will first look in the subclass for the functions called from the constructor.

**Recommendation**

The code below specifies the User class as final, so that it cannot be subclassed, and changes the validateUser() function to private since it is not needed elsewhere in this application. This is programming defensively. If it is decided later that the User class needs to be subclassed, it could result in this vulnerability reappearing if the validateUser() function was not set to private.

…

final class User {

private String username;

private boolean valid;

public User(String username, String password) {

this.username = username;

this.valid = validateUser(username, password);

}

private boolean validateUser(String username, String

password) {

//validate user is real and can authenticate

}

public final boolean isValid() {

return valid;

}

}

…

**Resources**

1. [HP Enterprise Security – Code Correctness: Byte Array to String Conversion](https://vulncat.fortify.com/en/detail?id=desc.semantic.java.code_correctness_byte_array_to_string_conversion#Java%2fJSP)