Understanding Backdoor Implementation

Cole Rees - A00741578

David Tran - A00801942

COMP 8505 Set 7D

British Columbia Institute of Technology

Aman Abdulla

Monday, October 6 2014

Table of Contents

# Introduction

One of the more common approaches to compromising a machine is the unknowing implementation of code that secretly allows access to other than the intended users. The reason why it is difficult to find is because the process of the code masks itself as another trusted process. A prime example for Linux operating systems is the “kworker” process, since there are multiple instances of kworkers, and for the inexperienced user, implementing another one under the same name can easily bypass their suspicion.

The purpose of this assignment is to grasp a deeper understanding of the backdoor. We shall be coding the backdoor and then implement it on a target machine. The program will work behind the scenes to capture specific sequences and signatures that are meant for it. Once it does and confirms that these packets match a corresponding pattern, it will execute a set of commands and then send back the results. We shall be executing some commands that may or may not compromise the machine, but the intention is to make a proof of concept. The commands can be anything, as long as the program has all the requirements to do it.

# Tools & Equipment

## Hardware

|  |  |  |
| --- | --- | --- |
| * 8GB RAM | * Intel i5 Quad Core | * 500GB HDD |
| * Controller (Puppeteer) | * Server Host (Slave) |  |

## Software

|  |  |  |
| --- | --- | --- |
| * Fedora Linux 20 64-bit | * C Programming | * Wireshark |
| * Terminal |  |  |

# Testing, Evidence & Observations

## Names & Aliases

|  |  |  |
| --- | --- | --- |
| **IP Addresses** | **Send / Receive Port** | **Alias** |
| 192.168.0.15 | 10022 | Controller |
| 192.168.0.14 | 10022 | Slave |

## Password

For testing purposes, our password will explicitly be the following:

password: “**comp**”

## Test Cases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1a | The Slave’s process is masked | ps | The process name is explicitly defined in the Slave’s code; cross referencing the ps command, we see that it exists | PASSED. See results for details. |
| 1b | The Slave’s process is masked | ps, kill | If we kill the Slave’s masked process name, the Slave should die | PASSED. See results for details. |
| 2 | Packet has the proper information configured in its headers | Wireshark | Wireshark shows that the packet has the proper destination IP and destination Port | PASSED. See results for details. |
| 3a | Slave is listening on the same port as the Controller is sending on | Terminal | If the ports are not the same, we will not connect. | PASSED. See results for details. |
| 3b | Slave is able to receive the packet that was destined for it | Wireshark,  Terminal | Terminal responds with appropriate message as expected; Wireshark displays the same packet as in [2] | PASSED. See results for details. |
| 4a | Slave is able to decrypt the password and authenticate the Controller | Terminal | If the password does not match, there will be no attempt to connect to the Controller | PASSED. See results for details. |
| 4b | Slave is able to decrypt the password and authenticate the Controller | Terminal | If the encrypt/decrypt key does not match, there will be no attempt to connect to the Controller | PASSED. See results for details. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 4c | Slave is able to decrypt the password and authenticate the Controller | Terminal | Terminal responds with appropriate message that the Slave has successfully connected with the Controller | PASSED. See results for details. |
| 5 | Once authenticated and connected, the Slave will execute the command specified in [3] | Wireshark, Terminal | Command is executed; the results are then sent back to the controller; Wireshark packet capture shows content | PASSED. See results for details. |
| 6 | Controller receives content from the Slave | Wireshark,  Terminal | Terminal outputs expected results; Wireshark confirms that the packet contains the same values | PASSED. See results for details. |
| 7 | Once packet is sent, Slave will listen for another packet with password, commands, etc. | Terminal | Attempt to redo the previous tests again to see if the Slave’s responses are identical | PASSED. See results for details. |

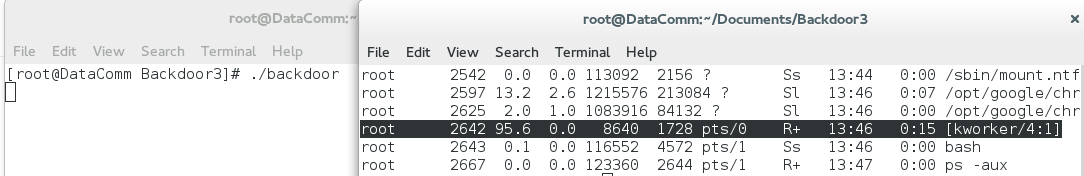
## Evidence & Observations

### 1a. The Slave’s Process Name is Masked

The header file, “backdoor.h”, explicitly tells us that its name will be “[kworker/4:1]” as seen below:

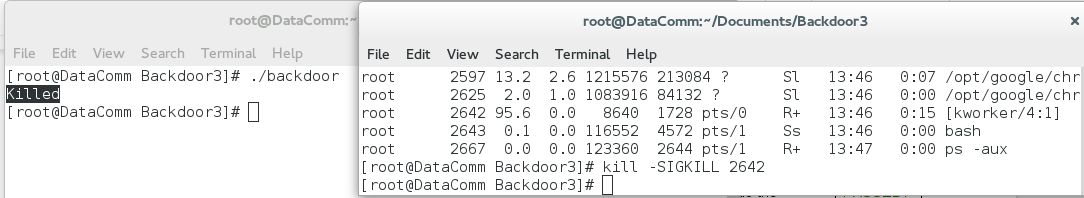


After compiling and running the program, we can see that it exists as a process:



### 1b. The Slave’s Process Name is Masked

To prove that this is the same “[kworker/4:1]” process as our program, we will attempt to kill it using its process id:



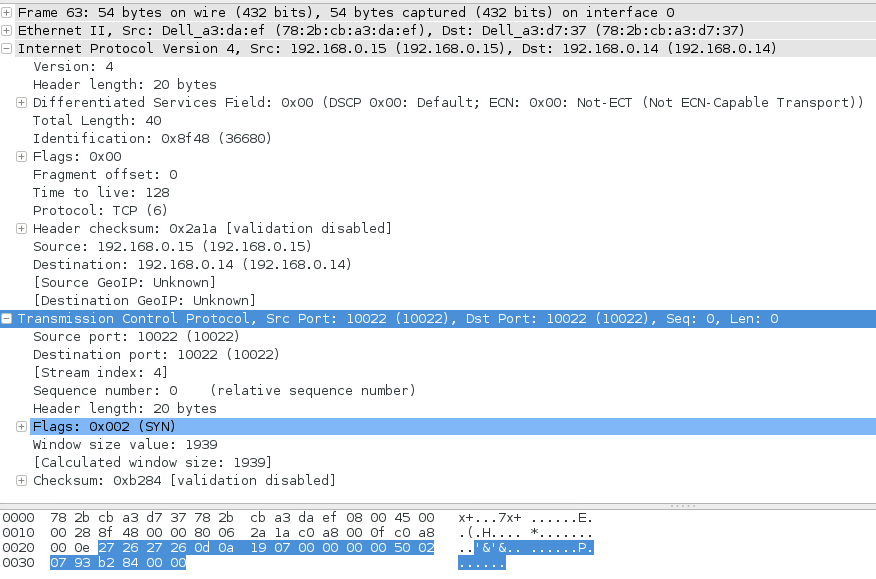
Note on the left terminal: The program has explicitly told us that it has been killed. Therefore the “[kworker/4:1]” with the process of 2642 was the same as our backdoor.

### 2. Packet has the proper information configured in its headers

To use the Controller, we used the following execution:

./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 10022 -c pwd

Here’s is the packet capture from Wireshark:



We can see that the Source and Destination Ports are as expected, as well as the proper source and destination addresses.

### 3a. Slave is listening on the same port as the Controller is sending on

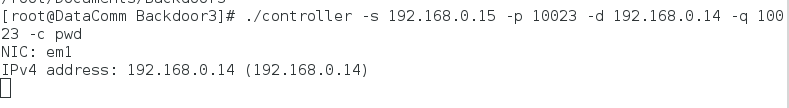
Note the “FILTER\_PORT” is 10022, which is our listening port.



If our controller were to change the destination port, then we would expect to have no communication. The following is our execution command:

./controller -s 192.168.0.15 -p 10023 -d 192.168.0.14 -q 10023 -c pwd

Note that the ports have been purposely changed from 10022 to 10023. Here’s our result:



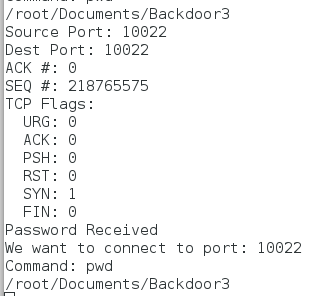
And the application simply hangs there. Unfortunately, the controller is unable to communicate and must be manually killed. However, because they are not on the same ports, it allows the Slave to specifically listen on a port and nowhere else.

### 3b. Slave is able to receive the packet that was destined for it

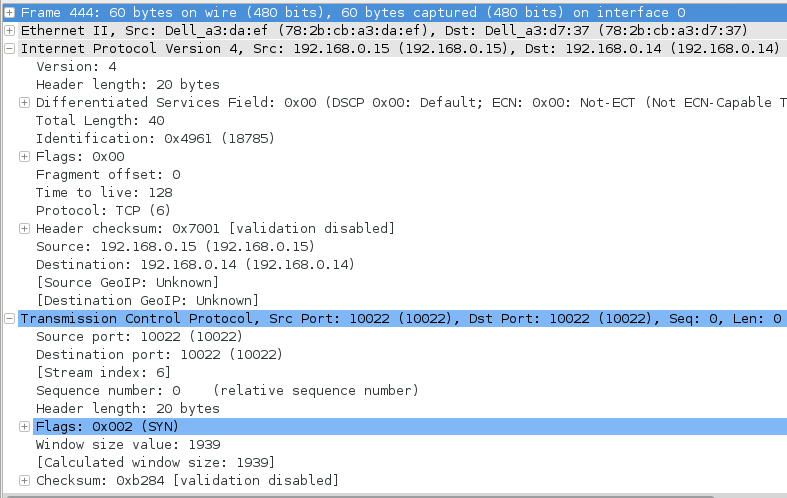
On our controller, we executed the following command:

./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 10022 -c pwd

On the Terminal, here is what we received:



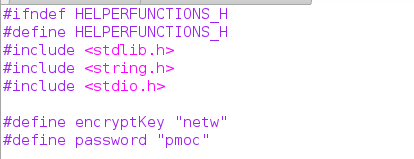
On our Slave machine, with Wireshark running, we were able to intercept the incoming packet:



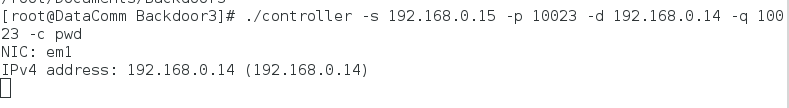
We can see that the source ports and destination ports are what they should be, as well as the proper source and destination addresses.

### 4a. Slave is able to decrypt the password and authenticate the Controller

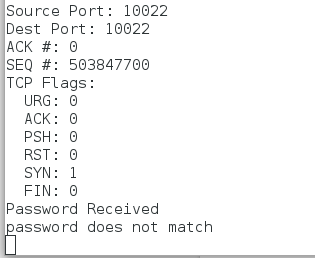
Firstly we will purposely change the password to an invalid one from “comp” to “pmoc” on our controller:



Then we executed our command as we would normally do. The following screenshot shows the results from our Controller:



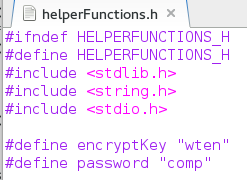
Unfortunately, it hangs trying to listen for a response from the Slave. Because we know that we placed an improper password, the Slave will reject it. The following screenshot depicts exactly that:



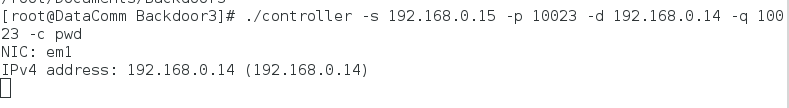
Note the “password does not match”; this is exactly what we would expect from an invalid password from the Controller!

### 4b. Slave is able to decrypt the password and authenticate the Controller

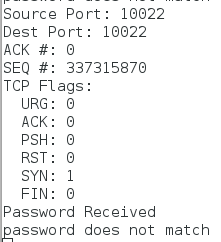
Now let’s revert the invalid password to a correct one, but this time, change the key from “netw” to an invalid “wten”:



We will execute the command again, as we did in [4a].

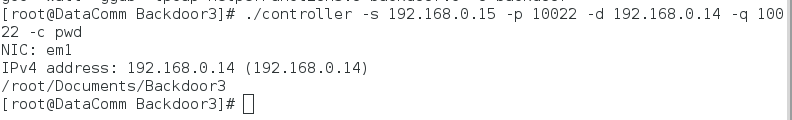


While the Controller hangs, this is a symptom of what we would expect from an invalid password. However, although we entered a proper password, the encryption key with “wten” outputs an unexpected value that the Slave cannot decrypt. In turn, it becomes an invalid password, as expected. Here’s the output from the Slave as proof:

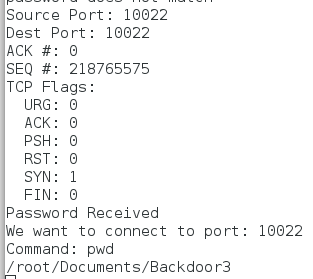


### 4c. Slave is able to decrypt the password and authenticate the Controller

We will now revert both password and key to “comp” and “netw” respectively on the Controller. We would expect that the transaction would continue as normal. The following screenshot is from the Controller:



We see here that the application has terminated, which implies that the password has been accepted. To ensure that it has definitely worked, here’s the screenshot from the Slave:

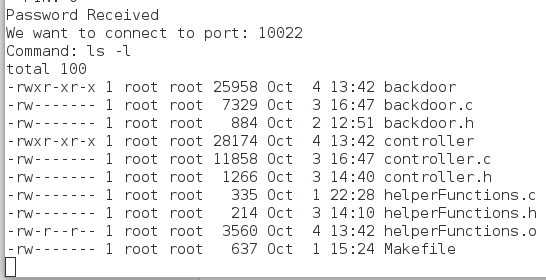


### 5. Once authenticated and connected, the Slave will execute the command specified in [2]

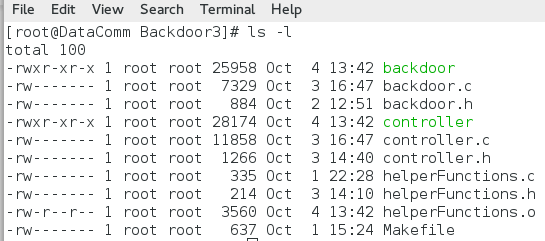
Once again, here is the command that we have been using thus far:

./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 10022 -c pwd

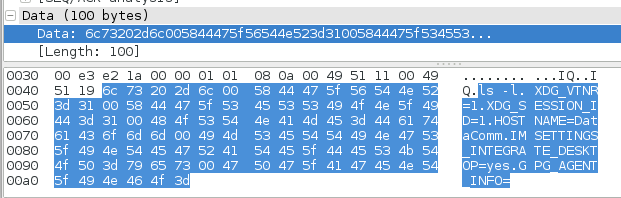
We will change the command switch from “pwd” to “ls -l”. Currently on our Slave, we are located within our “Backdoor3” directory which has the contents of our program, both Controller and Slave. We are expecting a few items. First from our Slave:



And if we manually ran the command in our directory, we can see that they’re exactly the same!

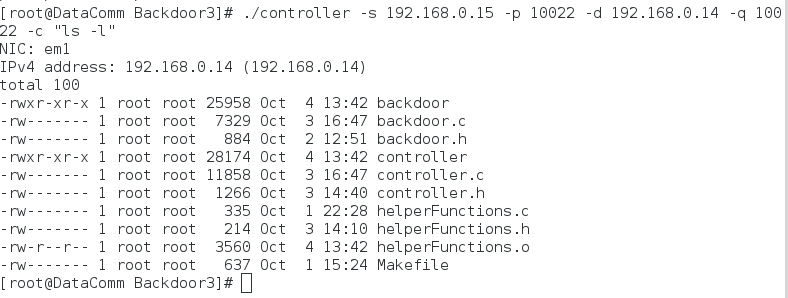


And thanks to a Wireshark capture from the Slave’s machine, we can intercept the packet with the command in plain sight:

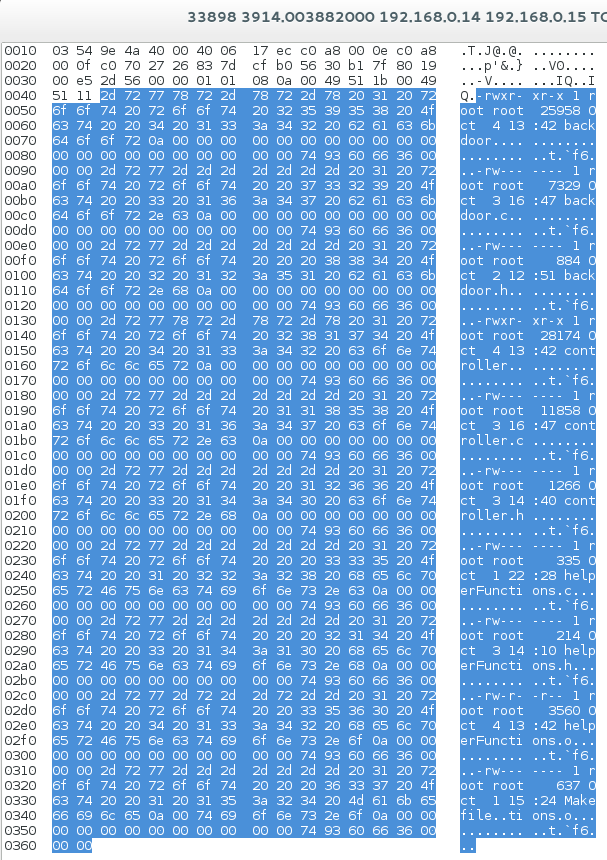


### 6. Controller received content from the Slave

Continuing from our last test case, the screenshot from our Controller:



Which is the same as our Slave’s contents. And again, we can see the data within Wireshark:



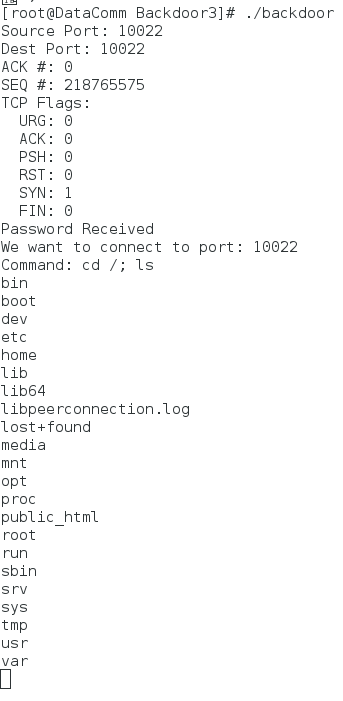
### 7. Once packet is sent, Slave will listen for another packet with password, commands, etc.

Our Slave’s purpose is to continually listen for more connections with the Controller. While the Controller terminates after sending and receiving a packet from the Slave, the Slave continues to loop as a listener. To prove that, we will show two consecutive connections from the Controller with two different commands:

First Command:

./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 10022 -c “cd /; ls”

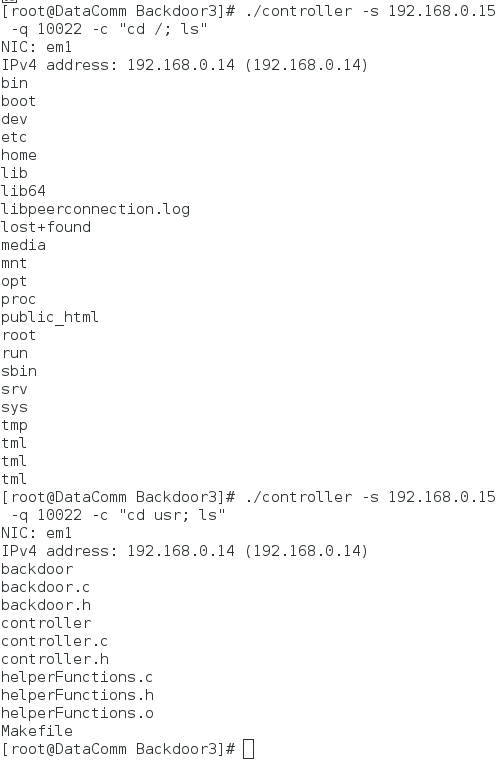
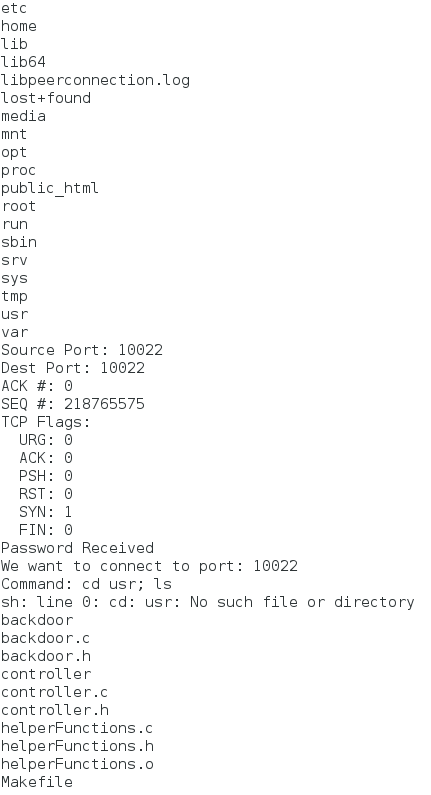
Results from the Slave (left) and Controller (right):



Second Command:

./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 10022 -c “cd usr; ls”

Screenshot from Slave (left) and Controller (right):



As we can see, there are remnants of the previous transaction on our Slave’s terminal, but we see no termination of the application. This ensures that the application continues to run after each transaction. Conversely, our Controller terminates after every transaction. We can see how we had to enter the command twice. Furthermore, note that our second command to traverse into “usr” directory is a relative command, not absolute! This ensures that it is dependent on our previous command and it lists exactly what we would expect.

# Limitations & Improvements

The success of this program, if we were to deploy it in real life, requires the need for the program to be open on a Terminal. An immediate remedy for this flaw is to simply build the program with daemon capabilities. Once the program is no longer reliant on the Terminal, then it would be more stealthy.

Further improvements on this application includes the bypassing of a firewall, port knocking techniques, log file sanitation and delayed returns to bypass any intrusion detection systems.

# Conclusion

It is dangerously apparent that backdoors can exist on our machines. It is also quite difficult for the average person to find these hidden backdoors. This begs the question, if the programmer was experienced enough, how would commercial malware/spyware/virus protection be able to detect these backdoors? It seems like the only person that would know the existence of backdoors are the ones that create them.

Of course, there is the chance of the user uncovering these programs themselves. However, the likelihood of a paranoid user that would monitor network traffic such as in our assignment is slim and the chances of detecting such communication is even slimmer.

So therefore, the only “optimal” way of handling such communications is to use a firewall. The downside, however, is that our application uses raw socket programming, and the listener uses the libcap library, which listens to traffic at the network card level, therefore bypassing the firewall itself. The program cares little if the firewall drops inbound packets to unfamiliar ports, or if they are sent to well known ports. The fact of the matter is the program is able to read it and parse it prior to any filtering is applied.

For network administrators such as ourselves, we can see how such attacks can be difficult to manage and mitigate. It is therefore imperative that we reduce all chances of the installation of these backdoors by increasing our understanding of these attacks. However, even then, we are prone to human stupidity, and a user on the network may be the one that compromises our system without even knowing that they did due to an attractive program or download.

# Appendix

## Appendix I - Files on Disk

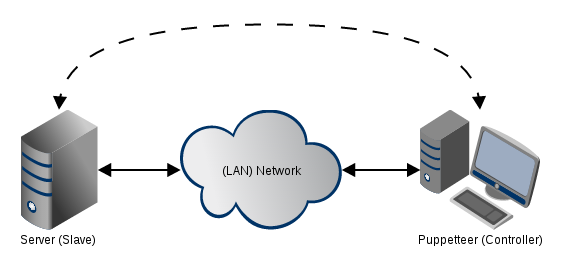
Located on disk are the following files:

* Understanding Backdoor Implementation (.pdf)
* controller.c
* controller.h
* backdoor.c
* backdoor.h
* Makefile
* README.txt
* helperFunctions.c
* helperFunctions.h
* (helperFunctions.o)

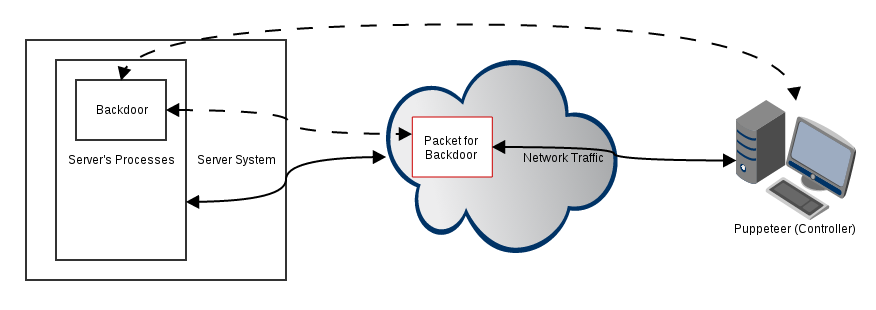
## Appendix II - Network Design & Pseudo Code

The following diagrams and prototypes depicts the setup for this assignment:

### High Level Diagram



### Logical Diagram



### 

### 

### Pseudo Code

|  |
| --- |
| **Backdoor (Pseudo) Code** |
| /\* Some Global Variables \*/  Puppeteer’s IP = 192.168.0.XX  Puppeteer’s Port = // some random port  Password = // some random password  /\* Our Main Function \*/  void main (...)  {  var unscrambled  create a false process name  while true  {  listens for packets from certain ip and port  if packets match our signature  {  unscrambled = decrypt (password)  }  if unscrambled == Password  {  // execute commands  // do some error checking to make sure commands work  // probably store the results in a temp file  send temp file contents as packet to ip and port  // destroy temp file  }  }  }  /\* Our Decryption Method \*/  decrypt (password)  {  put password into algorithm to unscramble  return unscrambled password  } |
| **Puppeteer (Controller Pseudo) Code** |
| /\* Some Global Variables \*/  Password = // some password  /\* Our Main Function \*/  void main ( ... )  {  read arguments  prepare packets  encrypt password and inject into packets  send packets  while true  {  listen for response  when packet is received  {  if last packet, kill    if packets are validation packets  {  prepare commands  prepare packets with commands  send commands  }    if packets are command results  {  parse results  print results to console  }  }  }  }  /\* Our Encryption Method \*/  Encrypt (password)  {  // encrypt password using our algorithm  return encrypted password  } |