Understanding Covert Backdoor Implementation

Cole Rees - A00741578

David Tran - A00801942

COMP 8505 Set 7D

British Columbia Institute of Technology

Aman Abdulla

Monday, December 8 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.

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.

## Building on our Proof of Concept

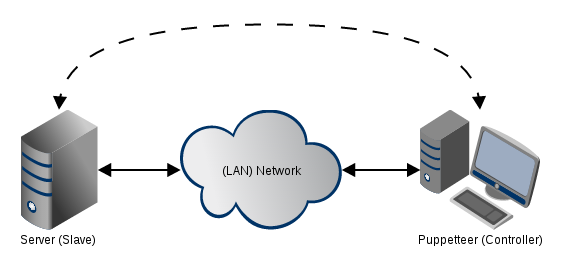
In our previous discussion, we have seen how covert channels work and how commands can be executed remotely using a client and server backdoor. Our final project will be to elaborate on our proof of concept, demonstrating that backdoors can be much more feature rich than to just receive and execute commands. Our goal is to be able to provide directory and file monitoring, alerting our client any changes in files or folders. The reasoning behind this is to illustrate how skilled attackers can use this concept in order to determine key files.

In addition to our previously working program of the Slave and Controller, we will add more features such as file and directory monitoring, and being able to covertly send back the key files and such to our headquarters. We will also implement port knocking, and the use of configuration files to our programs. All of these features will work on top of our previous program capabilities.

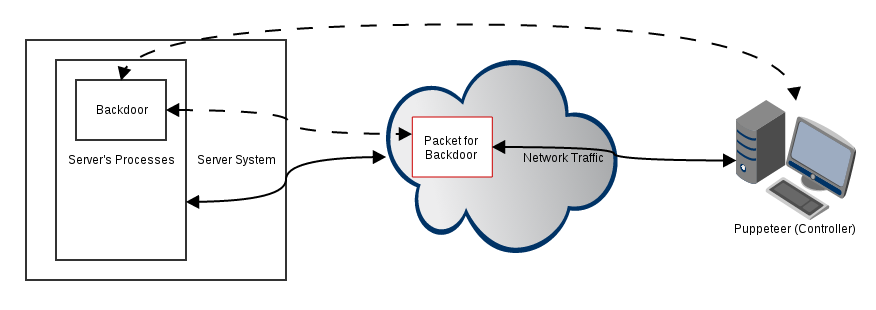
# High-Level Design Work

|  |  |
| --- | --- |
| **Design Features** | **Description** |
| Covert Channel Usage | The user shall be able to specify one of the following protocols to use as their covert channel: TCP or UDP. However, the user shall not be able to specify which headers will be used. The protocol selected must be in agreement between both Client and Server. See **Project Limitations and Future Implementations** for details on this design plan. |
| Client Modes | The user shall be able to explicitly specify if the Client will run in “listening” mode or “commanding” mode. “Listening” mode will allow the client to be in a server-like state, listening for the backdoor to alert of document or folder changes. “Commanding” mode will allow the client to send and receive simple commands. Client modes will be specified in command line arguments. |
| Port Knocking Feature | ~~Client and~~ Server will ~~both~~ implement the ~~same~~ port knocking sequence. We have decided to only implement the sequential component of port knocking. The sequence of ports can be determined by the user in the backdoor’s knocking script as well as the firewall script on the controller’s machine. |
| Encryption | Both Client and Server shall have the ability to encrypt and decrypt data. We will be reusing XOR encryption method in Assignment 2 of this course. |
| Server Monitoring & Exfiltration | The Client will be able to send the backdoor a series of files to monitor or transfer back to headquarters. These files will be explicitly specified in the configuration file by the user. Once exfiltration is done, any ports used by the Client or backdoor will be closed. |
| ~~Server Configuration File~~ | ~~The server configuration file will contain all the necessary user settings to allow the client to communicate covertly. These will be determined throughout development.~~ |
| Client Configuration File | The server configuration file will contain all the necessary user settings to allow the client to communicate covertly. These will be determined throughout development. Some examples are: which files to monitor, which directories to watch, etc. |

### High Level Diagram



### Logical Diagram



### Pseudo Code

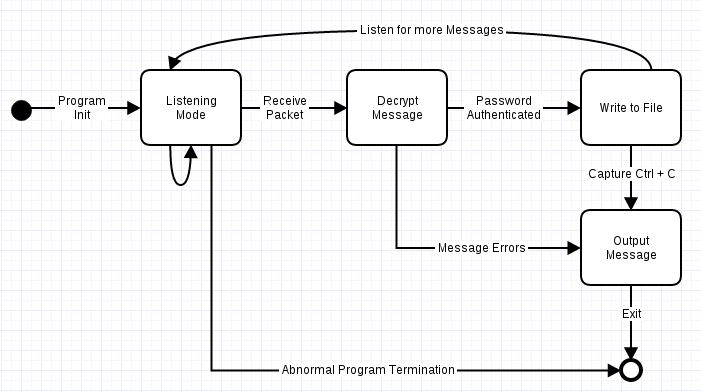
|  |
| --- |
| **Backdoor (Pseudo) Code** |
| /\* Some Global Variables \*/  Puppeteer’s IP = 192.168.0.XX  Puppeteer’s Port = // some random port  Password = // some random password  KNOCKING\_ports = // some predetermined pattern  FILE\_to\_read = // the file to monitor  DIRECTORY\_to\_monitor = //the directory to monitor for changes  /\* Our Main Function \*/  void main (...)  {  var unscrambled  create a false process name  read(config\_file)  // we will always listen for commands  int listenID = pthread\_create(listener\_mode(“command”))    if (FILE\_to\_read is not null) // we want to watch a file  int fileID = pthread\_create(watching\_mode(“file”))  if (DIRECTORY\_to\_monitor is not null) // we want to watch a directory  int directoryID = pthread\_create(watching\_mode(“directory”))  join threads  Catch Ctrl + C (or some Termination Command from Client)  kill all threads  exit  }  listener\_mode(“command”) {  while true  {  listens for packets from certain ip and port  if packets match our signature  {  unscrambled = decrypt (password)  }  if unscrambled == Password  {  // do some error checking to make sure commands work  run “command”  send (command outputs in packet to client)  }  }  }  watching\_mode([file | directory])  {  while true  {  check if file or directory has changed  if the file or directory has changed  make note of the change or copy the file  send (the note to client or send the copied file)  }  }  send ([file | console\_output])  {  knocker(KNOCKING\_ports) // do our port knocking first before we send our data  XOR(package up our packet)  send the packet on its way to client  }  knocker(KNOCKING\_ports)  {  if KNOCKING\_ports != null // there is a knocking pattern, otherwise do nothing  {  temp.array = KNOCKING\_ports.split(“,”) // split up the ports  foreach port in temp.array  {  send a packet to port  do not wait for a response; sleep (1);  }  }  }  read(config\_file)  {  while (there is a line to read)  {  read each line of the file,  split using a delimiter (probably “=”) into key, value pairs  switch statement (key)  {  case “FILE”  set FILE\_to\_read variable with its value pair  break  case “KNOCKING”  set KNOCKING\_ports to its value pair  break  case “DIRECTORY”  set DIRECTORY\_to\_monitor to its value pair  break  ... (other cases to be determined)  }  }  }  /\* Our Encryption/Decryption Method \*/  XOR(item) // using XOR  {  put item into algorithm to scramble/unscramble  return scrambled/unscrambled data  } |

### 

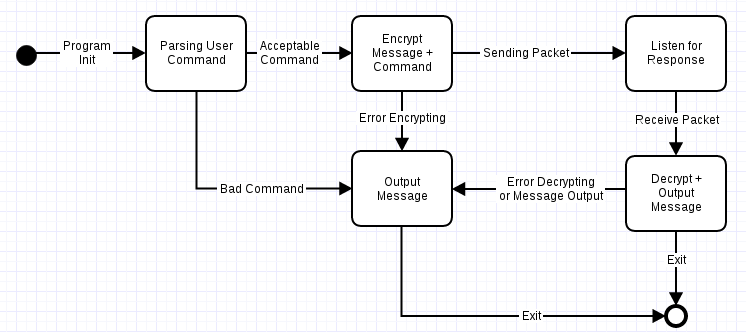
|  |
| --- |
| **Puppeteer (Controller Pseudo) Code** |
| /\* Some Global Variables \*/  Password = // some password  KNOCKING\_ports = // some predetermined pattern  FILE\_to\_output = // the file path to export changes  DIRECTORY\_to\_output = //the directory path to export changes  /\* Our Main Function \*/  void main ( ... )  {  read arguments  read config file    // we will always listen for responses  int listenID = pthread\_create(listener\_mode(“response”))  int commandID = pthread\_create(commander\_mode())  join threads  Catch Ctrl + C  kill all threads  exit  }  listener\_mode(“response”)  {  while true  {  listens for packets from certain ip and port  if packets match our signature  unscrambled = decrypt (password)  if unscrambled == Password  if last packet, kill  if packet contains file contents  export content to text file\_%DATE-RECEIVED  if packet contains directory changes  export content to text directory\_%DATE-RECEIVED  }  }  commander\_mode()  {  while true  {  listens for packets from certain ip and port  if packets match our signature  unscrambled = decrypt (password)  if unscrambled == Password  {  if last packet, kill    if packets are validation packets  {  knocker(KNOCKING\_ports) // do our port knocking first before send  prepare packets  XOR(packets with injected command)  send the packets  }    if packets are command results  {  parse results  print results to console  }  }  }  }  knocker(KNOCKING\_ports)  {  if KNOCKING\_ports != null // there is a knocking pattern, otherwise do nothing  {  temp.array = KNOCKING\_ports.split(“,”) // split up the ports  foreach port in temp.array  {  send a packet to port  do not wait for a response; sleep (1);  }  }  }  read(config\_file)  {  while (there is a line to read)  {  read each line of the file,  split using a delimiter (probably “=”) into key, value pairs  switch statement (key)  {  case “FILE”  set FILE\_to\_read variable with its value pair  break  case “KNOCKING”  set KNOCKING\_ports to its value pair  break  case “DIRECTORY”  set DIRECTORY\_to\_monitor to its value pair  break  ... (other cases to be determined)  }  }  }  /\* Our Encryption/Decryption Method \*/  XOR(item) // using XOR  {  put item into algorithm to scramble/unscramble  return scrambled/unscrambled data  } |

## State Chart Diagrams

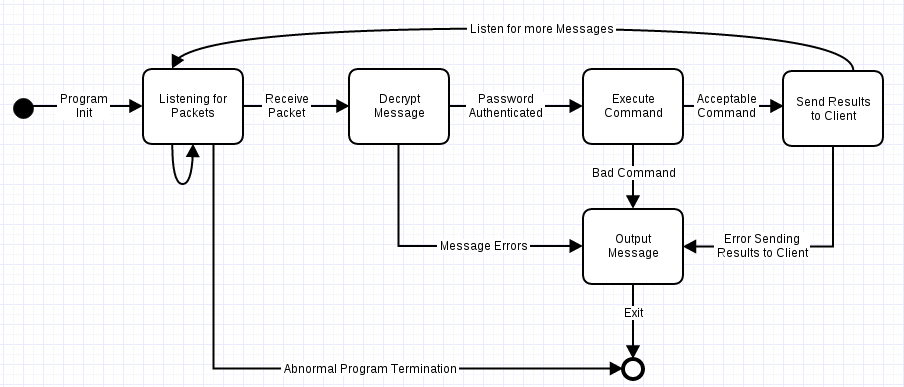
### Client - Listening Mode



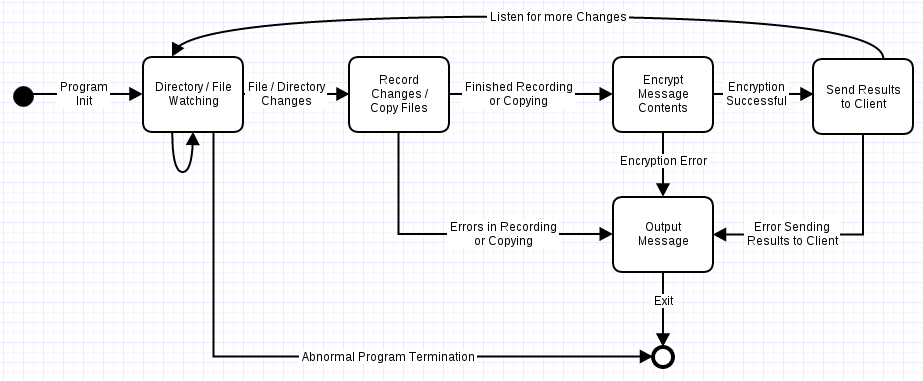
### Client - Commanding Mode



### Backdoor - Running Commands



### Backdoor - File & Directory Changes



# Scheduled Timeline, Tasks & Milestones

INSERT GANTT CHART HERE

# 

# 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 | * Valgrind | * htop |

# Testing, Evidence & Observations

## Names & Aliases

|  |  |  |
| --- | --- | --- |
| **IP Addresses** | **Send / Receive Port** | **Alias** |
| 192.168.0.21, 192.168.0.16 | 10022 | Controller |
| 192.168.0.22, 192.168.0.17 | 10022 | Slave |

## Test Cases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| A | Programs do not have any memory leaks | Valgrind | No memory leaks. | PASSED. See results for details. |
| B | Programs are not intensive for machines to run | htop | There are no significant slowdowns; they do not exhaust any CPU processing power | FAILED. See results for explanation. |
| C | Basic firewall implementations on Controller | iptables | All external packets are dropped, except for those on port knocking | PASSED. See results for details. |
| I | Backdoor can mask its process name | ps aux | The Backdoor application’s name is not Backdoor | PASSED. See Appendices for details. |
| II | 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. |
| III | 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. |
| IV | 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. |
| V | 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. |
| 1 | Controller can read config file | Terminal | Console will explicitly output each parameter in config file | PASSED. See results for details. |
| 2 | Slave will be monitoring a file if ~~a FILE\_to\_read is specified~~ Controller sets mode to “file” | Terminal | Console will output the data regarding the file, read from Controller | PASSED. See results for details. |
| 3 | When there is a change in the file to be monitored, send the file back to the Controller using TCP | Wireshark, Terminal | Slave will initiate a port knocking sequence; upon success, Slave will take the file, package it in a packet, and send it back to the Controller | PASSED. See results for details. |
| 4 | When there is a change in the file to be monitored, send the file back to the Controller using UDP | Wireshark, Terminal | Slave will initiate a port knocking sequence; upon success, Slave will take the file, package it in a packet, and send it back to the Controller | PASSED. See results for details. |
| 5 | Controller will be exporting changes of file if a FILE\_to\_output is specified | Terminal | Results will be shown on Terminal Console | PASSED. See results for details. |
| 6 | Slave can execute commands from Controller via TCP | Wireshark, Terminal | Over a period of time, the Slave will output results to console and send via TCP to Controller | PASSED. See results for details. |
| 7 | Controller can receive command results from Slave after execution through TCP | Wireshark, Terminal | Console will output the commands executed by the Slave. | PASSED. See results for details. |
| 8 | When a connection is requested, the Controller is able to recognize the port knocking sequence | Wireshark, Terminal, iptables | If port knocking sequence is valid, the connection will be granted. | PASSED. See results for details. |

## 

## 

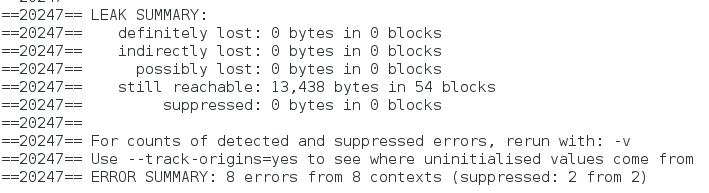
## Evidence & Observations

### Programs do not have any memory leaks

We ran the following Valgrind command to check for memory leaks in the backdoor application:

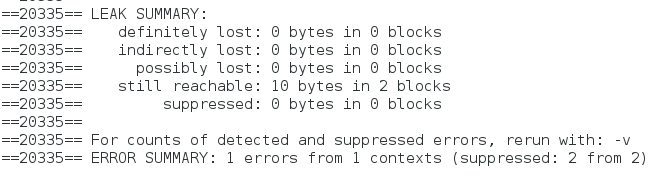
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./[application name]

This was the result from Valgrind with respect to the backdoor:



Note that “still reachable” has 13,500 bytes of memory lost. However, this is acceptable because the memory “loss” is due to libpcap and inotify libraries, which we are assuming to have their own memory allocation management.

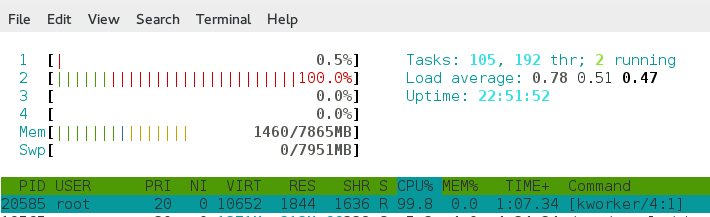
Similar results are shown from the Controller application when we ran it against Valgrind:



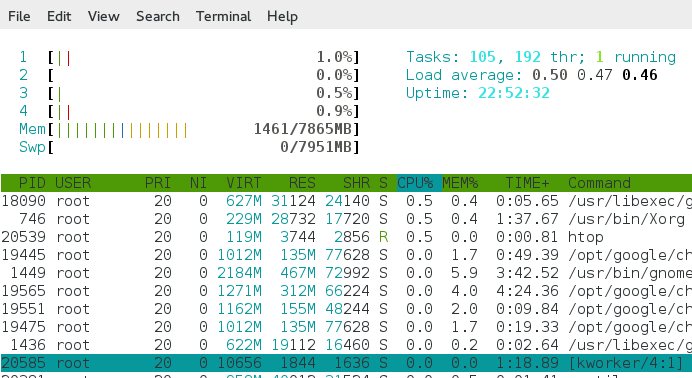
This confirms that our applications are sufficiently managing memory allocation and that there are no significant memory leaks.

### Programs are not intensive for machines to run

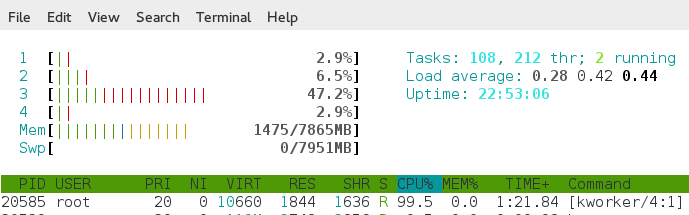
This is a screenshot of the backdoor application running on the Slave machine with htop running. Note how the CPU usage on the second core is at 100%. This is due to the fact that the backdoor is intently listening for any communication from the Controller.



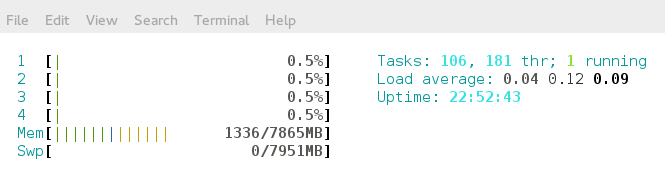
However, once it receives a file monitoring command or a command line argument from the Controller, the CPU usage drops drastically as shown in the next screenshot:



Regardless, once it finishes its job and goes back into “listening” mode, the CPU usage is back up nearly 50% on the third core and our “kworker” process is back to 99.5% CPU usage:



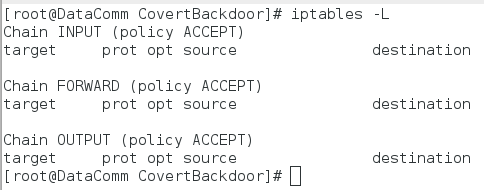
On the Controller application, there are a few fluctuations in the CPU usage, but nothing too drastic to warrant a full pin on the core:



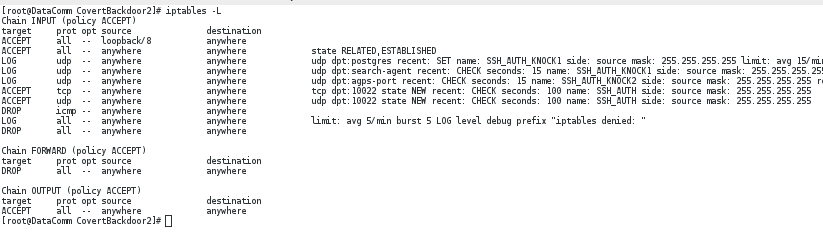
Unfortunately, our program fails in the area of minimal CPU usage. We have addressed this issue and discussed some remedies for it in the **Project Limitations and Future Implementations** section of this report.

### Basic firewall implementations on Controller

In the following screenshot, the current firewall settings are set to allow all traffic.



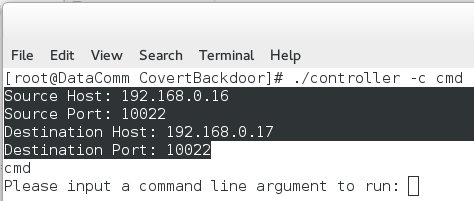
After we run the following command, we configure the firewall to accept essential traffic, such as DNS, HTTP and ICMP, while dropping other traffic. Refer to the following screenshot:



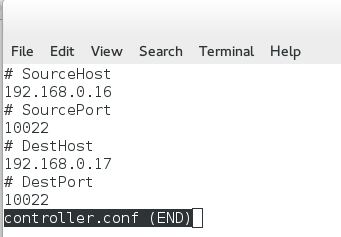
The Controller’s firewall is properly configured with timers for TCP and UDP access, and is ready for backdoor simulation.

### Controller can read config file

In the following screenshot, upon initializing the Controller, the output of the program to console regarding its configuration file is as follows:



To confirm that it is reading the configuration file successfully, a screenshot of the controller.conf configuration file is shown below:

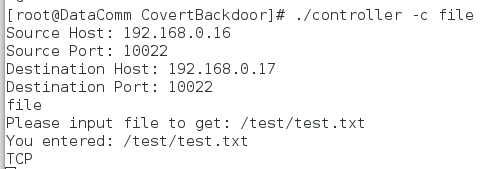


There are no outstanding differences between program and configuration file.

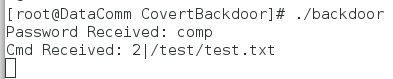
### Slave will be monitoring a file if Controller sets mode to “file”

NOTE: Prior to running this test, a directory called “test” is created on the root directory of the Slave machine. Then, a file called “test.txt” is created within the newly created test directory. We will use this file as our file to monitor.

While the backdoor is running on the Slave machine, the following command is executed from the Controller. This command tells the backdoor application on the Slave machine to monitor the specified file:

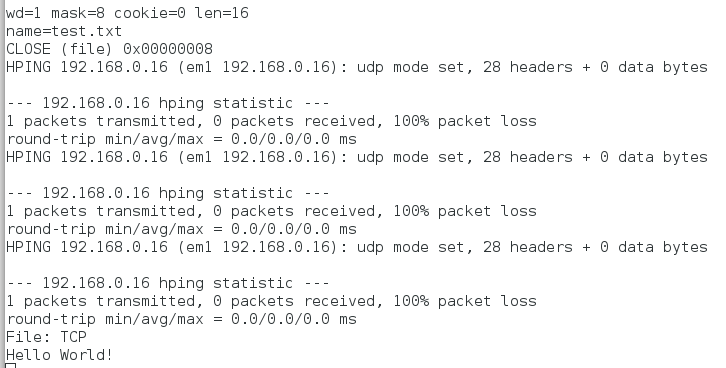


On the Slave machine, we receive the following messages from the backdoor:

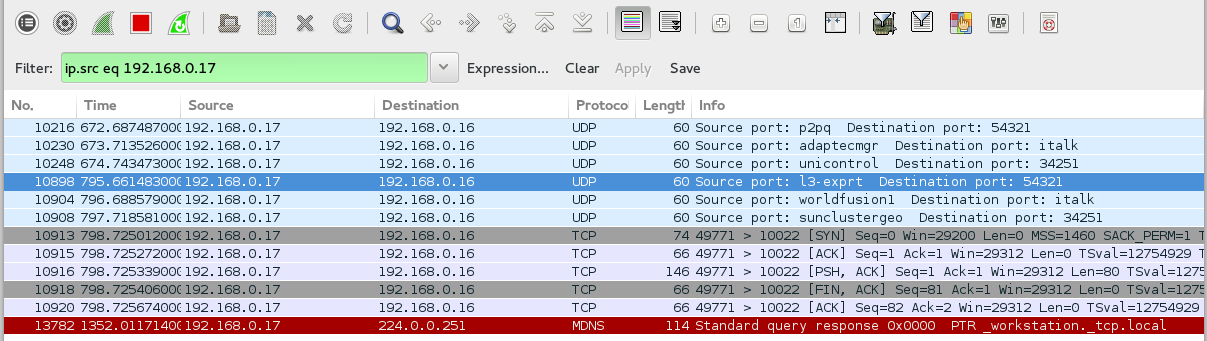


### When there are changes in file to monitor, send results to Controller via TCP

We modify the file’s contents. Once we save the file, we receive the following messages from the backdoor; note the port knocks from the backdoor:



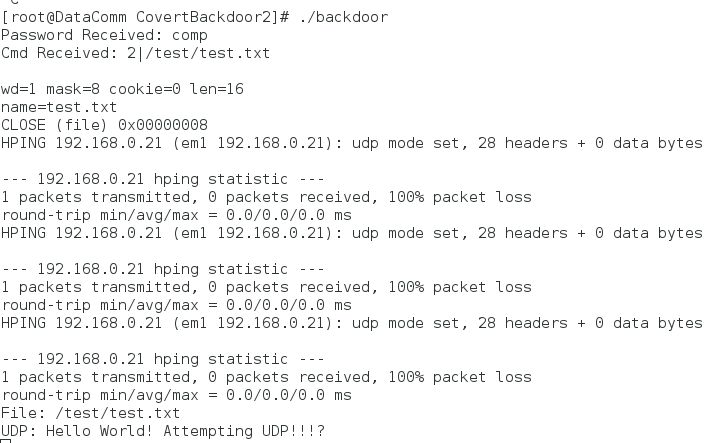
These screenshots confirm that our program is capable of monitoring files. To check the network, Wireshark was running during testing. After filtering the capture, the following screenshot confirms our communication:



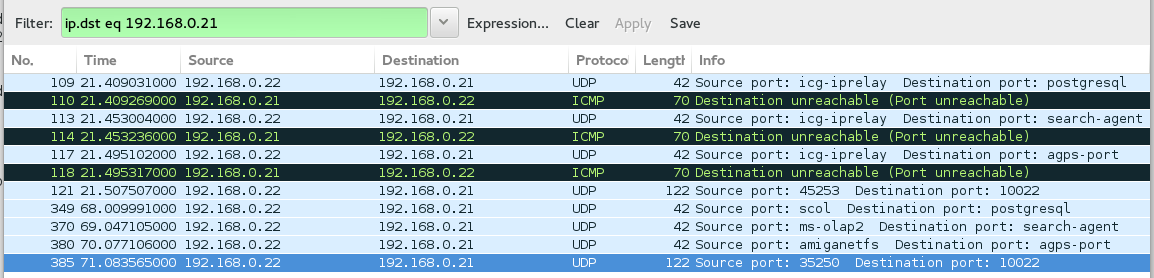
Of these packets, there are no evidence to suggest that “Hello World!” exists as plaintext within the payload. Therefore, we were able to exfiltrate the data using covert channels.

### When there are changes in file to monitor, send results to Controller via UDP

We modify the file’s contents. Once we save the file, we receive the following messages from the backdoor; note the port knocks from the backdoor:



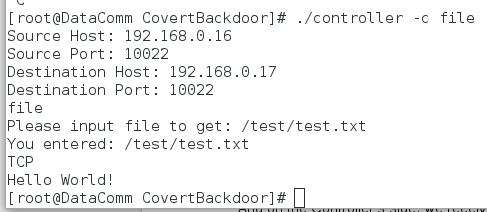
These screenshots confirm that our program is capable of monitoring files. To check the network, Wireshark was running during testing. After filtering the capture, the following screenshot confirms our communication:



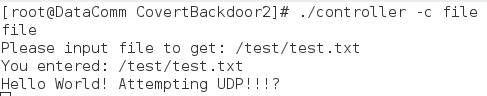
Of these packets, there are no evidence to suggest that “Hello World!” exists as plaintext within the payload. Therefore, we were able to exfiltrate the data using covert channels.

**Controller will be exporting changes if file to monitor is specified**

On the Controller’s side, we receive the changed values from the file we were monitoring outputted to console when the protocol was using TCP:



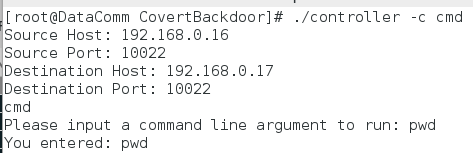
Likewise, the output for UDP component of the exfiltration yields similar results as seen in the following screenshot.



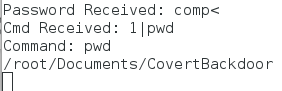
This confirms that the Controller is able to listen in modes TCP and UDP protocols.

### Slave can execute commands from Controller

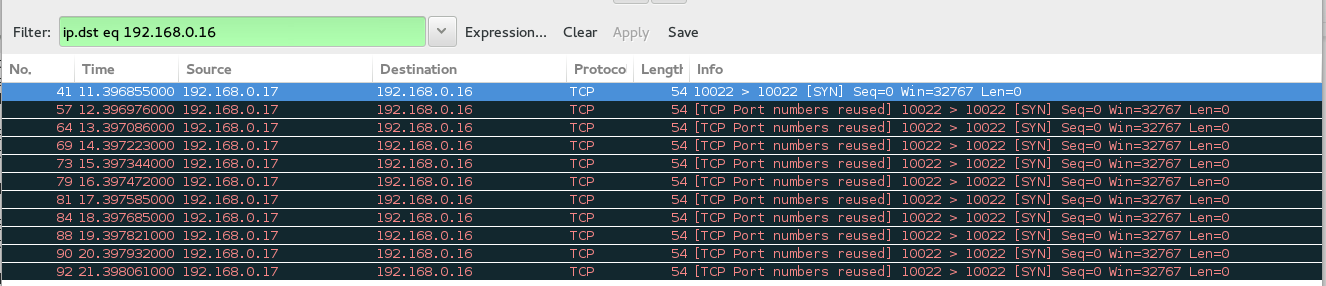
While the backdoor is running on the Slave, the Controller executes the following command and sends it to the backdoor application over TCP:



The backdoor parses this command, and attempts to execute it. Upon completion of execution, as seen in the below screenshot, the backdoor sends the output back to the Controller.



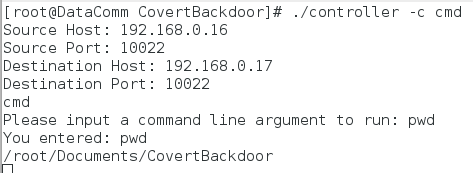
This screenshot from Wireshark running on the Slave machine shows the traffic being sent to the Controller:



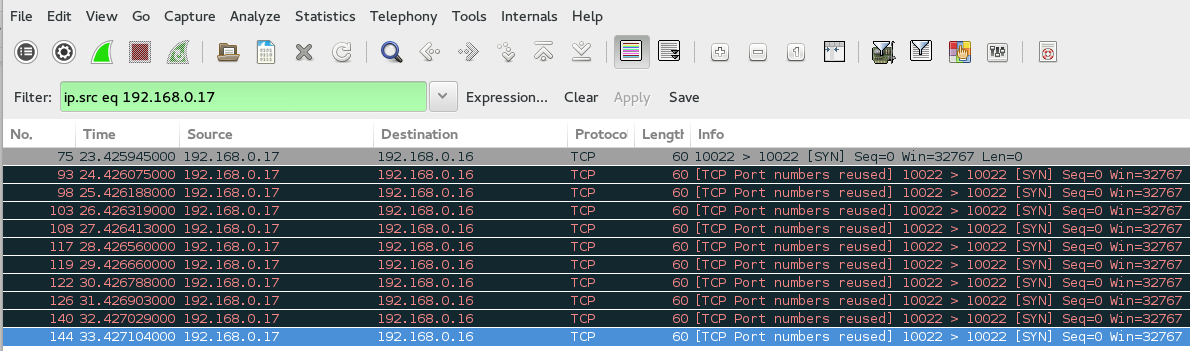
This confirms that our backdoor is able to communicate back to the Controller.

### Controller can receive command results from Slave after execution

Once we were able to send our command to the backdoor, the Controller waits until it receives a response from the backdoor. When it does, the application outputs response values like so:



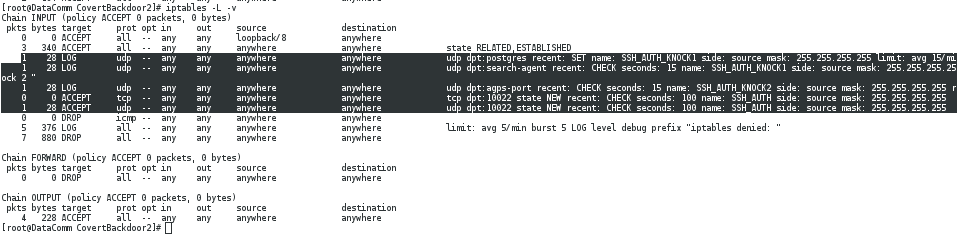
While its not possible to see the time between each line of output, the following screenshot from Wireshark can help highlight the spread of these payloads:



Basically, over a set of 11 packets, each packet was delayed by 1 second. For a small line of text, this is quite long. However, to maintain covertness, it is imperative to have the traffic spread out. This confirms that the Controller can receive responses from the backdoor.

### When connection is requested, Controller recognizes port knocking sequence

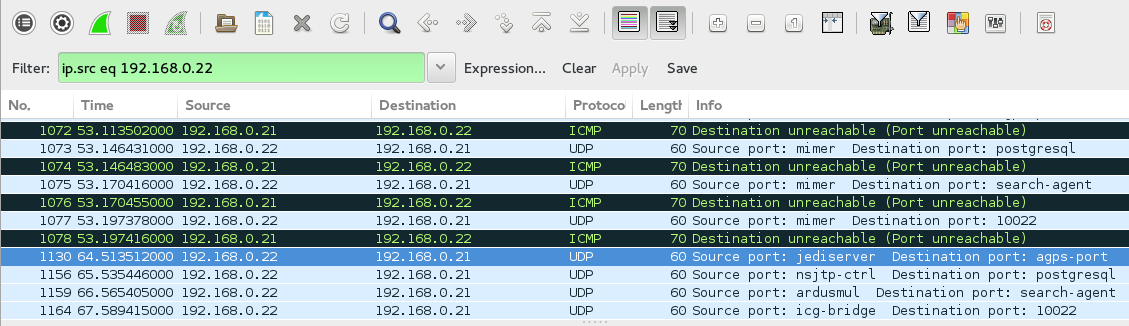
Once the firewall configuration is applied, a port knocking script is tested against the system. The script is executed by the backdoor to use hping3 to craft packets specific to our port knocking signature. Here are the firewall results after sending a packet to each port STATE:



In the above screenshot, 3 packets were fired to the following ports: 5432, 1234, and 3425 in that particular order before firing a packet to 10022, which is our listening port for both applications. Once these ports were knocked upon by our script, the firewall allowed our UDP packet to pass through. However, once we change the order to ports 3425, 5432 and 1234 respectively…



The firewall disallows the packet to port agps-port (3425) in the first packet, but allows postgres (5432) and search-agent (1234) because they are configured in first and second order. Here is a screenshot of the Wireshark capture:



The Controller’s firewall is properly configured and is capable for enabling port knocking features.

# Project Limitations & Future Implementation

During the development phase of our program, using Linux’s System Monitor, we found that our backdoor pinned down one of the four cores in our victim machine. This is due to the fact that the backdoor uses pcap loop to listen for our controller to send commands. Unfortunately, this is susceptible to detection when an adept user realizes that their machine is losing performance. To remedy this, we would suggest implementing Epoll and Signals to alleviate the stress placed on one core.

One of the few awkward things about our program was how it was not able to handle specified protocols in the Controller’s configuration file. Initially, we found that it would be most intuitive if the protocol is specified by the user within the configuration file. However, upon compilation, our program continues to cause Segmentation Faults and they were due to the way the Controller handled and parsed the protocol we specified. As a workaround, and although it is cumbersome, we hardcoded the option into the program headers. For future implementation, we would like to tackle migrating this configuration issue back to the configuration file instead of specifying it in the header.

We also intended on adding threads to this project. Unfortunately, due to time constraints and awkward problems that we encountered, we decided to focus on the functional components of the project. With that in mind, it would be wise to allow the backdoor to listen on more than one file of interest. This can be accomplished using multithreading or epoll systems.

Finally, our project is currently only capable of monitoring one monitor for one change, per client session. With the inclusion of multithreading in the future, this design feature can become more robust for continual monitoring, and multiple file monitoring. Some other roadblocks occurred during development. One peculiar problem was a file corruption on either the Controller or Backdoor application. This resulted in a delay in the development and testing component of the project, and thus had to omit some design features.

# Prevention & Detection

Throughout this project, we learned how sophisticated backdoors can be. In fact, the most robust backdoor is quite rare, but once it is embedded within the victim’s machine, there is an infinite amount of possibilities the attacker could do. At that point, the only way to stop the victim machine from being used by the attacker is to prohibit any network access to it.

Some measures to detect covert activity is to implement a logging system for the firewall. Because the program may use a port knocking feature, by monitoring the firewall for any suspicious or unauthorized actions of “lowering shields”, then that could very well mean that there is a backdoor.

In terms of prevention, frankly speaking, is much more difficult to handle because the means of implementing a backdoor often uses an unsuspecting user. Humans are the weakest link in any security chain, and once an attacker is able to convince such users that they are harmless, that is when users are taken advantage of, and machines become compromised.

The best course of action is, as a network administrator who specializes in network security, to educate the users as much as we can with regards to these dangerous type of attacks. That begs the question of what kind of education do these users require? To start, users should begin to follow corporate policies on stronger passwords. Other suggestions include being aware of social engineering tactics, phishing emails, malicious documents, and “free” storage media.

# Conclusion

To close the discussion, as network administrators, we have done our due diligence to protect the network as much as we can when we educate our frail and sometimes technologically illiterate users. Even so, it is a never ending battle between hackers and administrators.

Sun Tzu says that “If you know the enemy and know yourself, you need not fear the result of a hundred battles.” However, “If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.” The purpose of all the assignments and projects thus far in our courses are meant to help us understand the enemy.

By understanding the enemy, his intentions and his attack strategies, we can tailor our defenses to counter them. We, as beginner network administrators, will have a better grasp of these attacks and while we may not win every battle, we will likely not lose all the battles compared to the rest of our competition. For us, it will take time and experience to fully understand the enemy before we can counter their every move.

# Appendices

## Appendix I - Files on Disk

The following are files that are located on-disk:

## 

## Appendix I - Test Evidence & Observations (from previous implementation)

## Test Cases

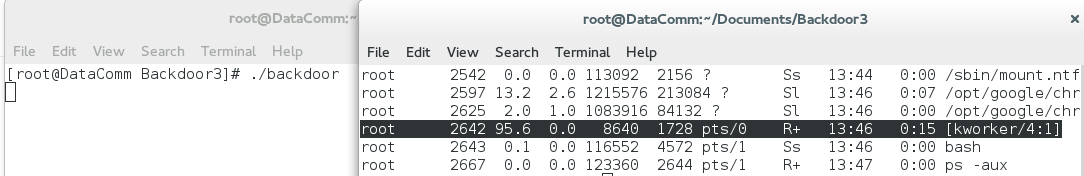
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1a | The Slave’s process is masked | ps | The process name is explicitly defined in the Slave’s code; cross referencing the ps command, we see that it exists | PASSED. 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. |
| 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. |

### 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:

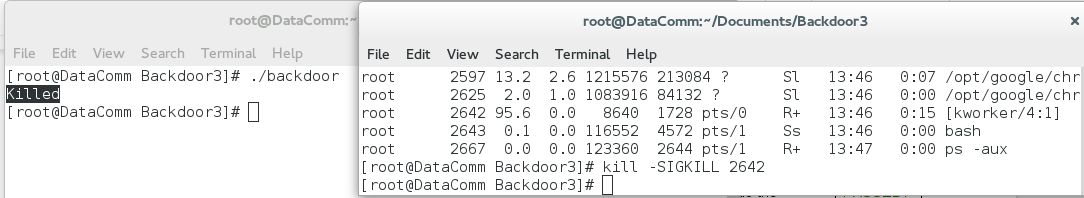


After compiling and running the program, we can see that it exists as a process:



### 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:



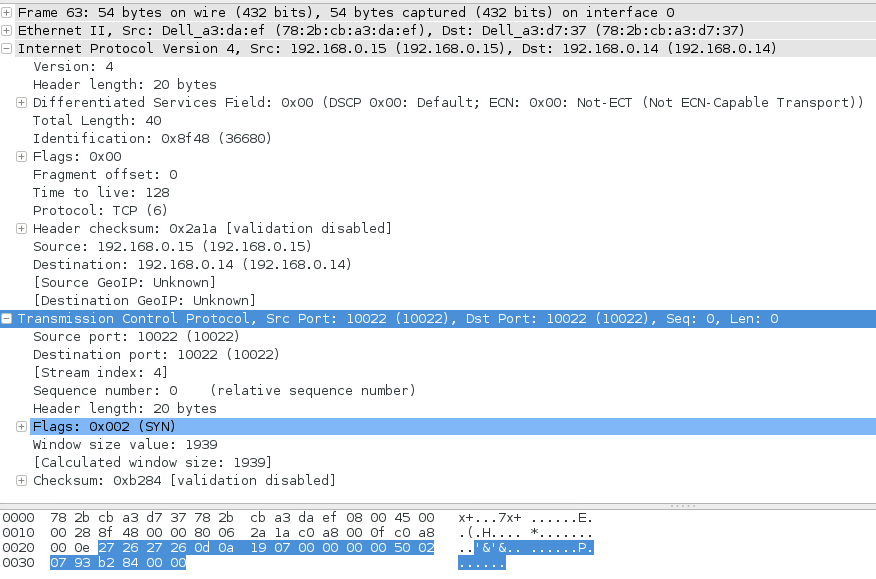
Note on the left terminal: The program has explicitly told us that it has been killed. Therefore the “[kworker/4:1]” with the process of 2642 was the same as our backdoor.

### 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

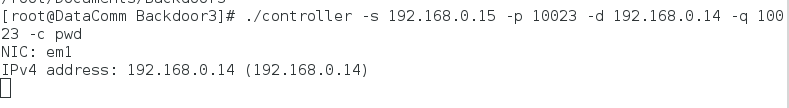
Note the “FILTER\_PORT” is 10022, which is our listening port.



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:



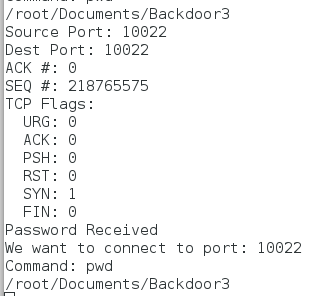
And the application simply hangs there. Unfortunately, the controller is unable to communicate and must be manually killed. However, because they are not on the same ports, it allows the Slave to specifically listen on a port and nowhere else.

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

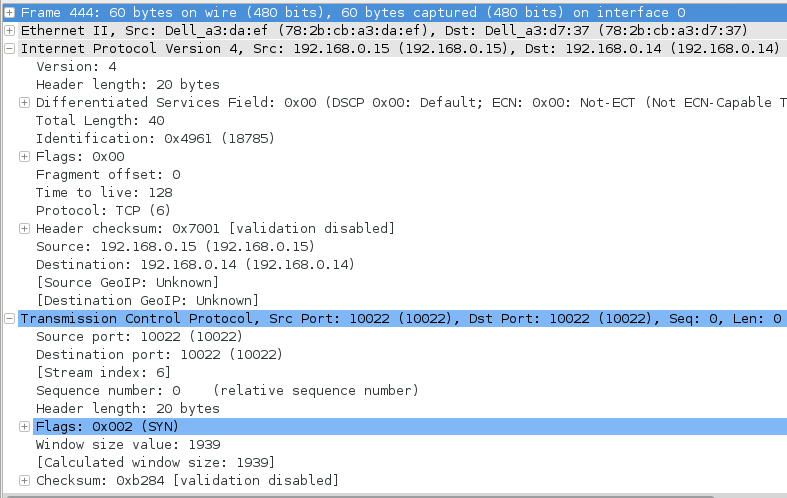
On our controller, we executed the following command:

./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:



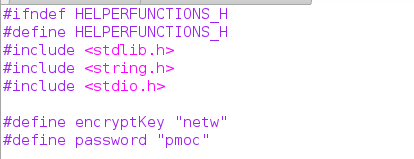
On our Slave machine, with Wireshark running, we were able to intercept the incoming packet:



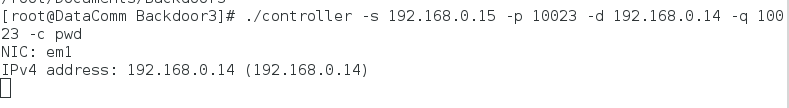
We can see that the source ports and destination ports are what they should be, as well as the proper source and destination addresses.

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

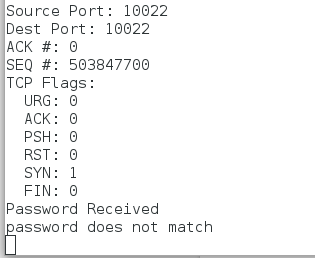
Firstly we will purposely change the password to an invalid one from “comp” to “pmoc” on our controller:



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



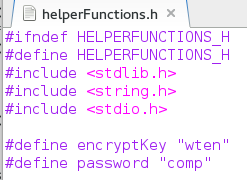
Unfortunately, it hangs trying to listen for a response from the Slave. Because we know that we placed an improper password, the Slave will reject it. The following screenshot depicts exactly that:



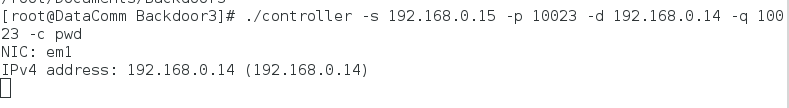
Note the “password does not match”; this is exactly what we would expect from an invalid password from the Controller!

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

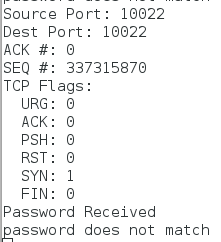
Now let’s revert the invalid password to a correct one, but this time, change the key from “netw” to an invalid “wten”:



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

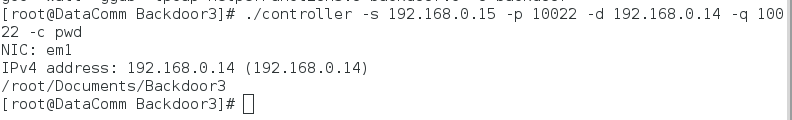


While the Controller hangs, this is a symptom of what we would expect from an invalid password. However, although we entered a proper password, the encryption key with “wten” outputs an unexpected value that the Slave cannot decrypt. In turn, it becomes an invalid password, as expected. Here’s the output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

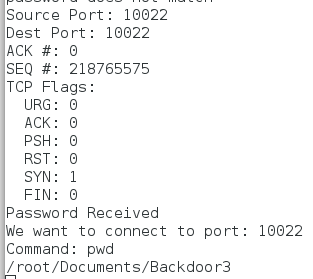


### 4c. Slave is able to decrypt the password and authenticate the Controller

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:

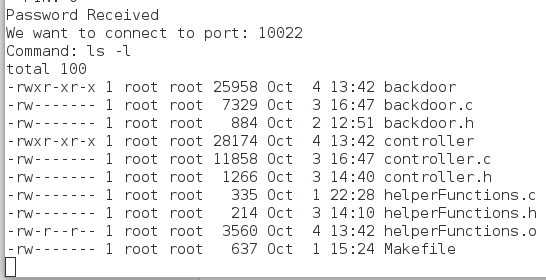


### 5. Once authenticated and connected, the Slave will execute the command specified in [2]

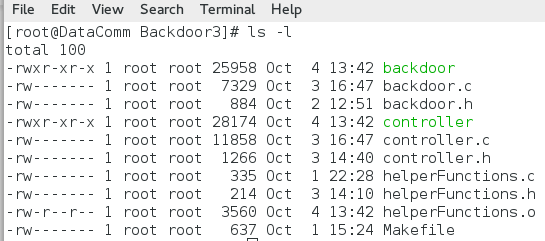
Once again, here is the command that we have been using thus far:

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

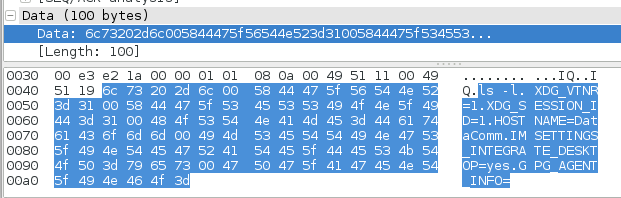
We will change the command switch from “pwd” to “ls -l”. Currently on our Slave, we are located within our “Backdoor3” directory which has the contents of our program, both Controller and Slave. We are expecting a few items. First from our Slave:



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

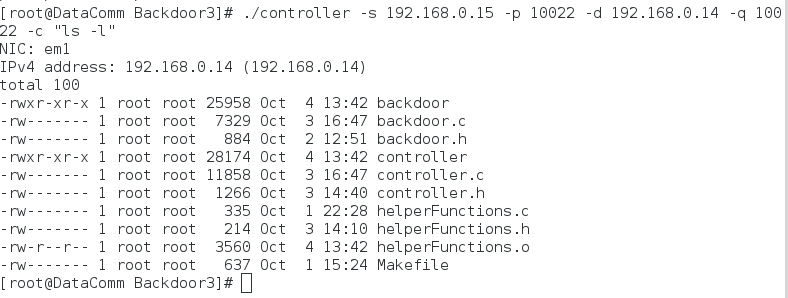


And thanks to a Wireshark capture from the Slave’s machine, we can intercept the packet with the command in plain sight:

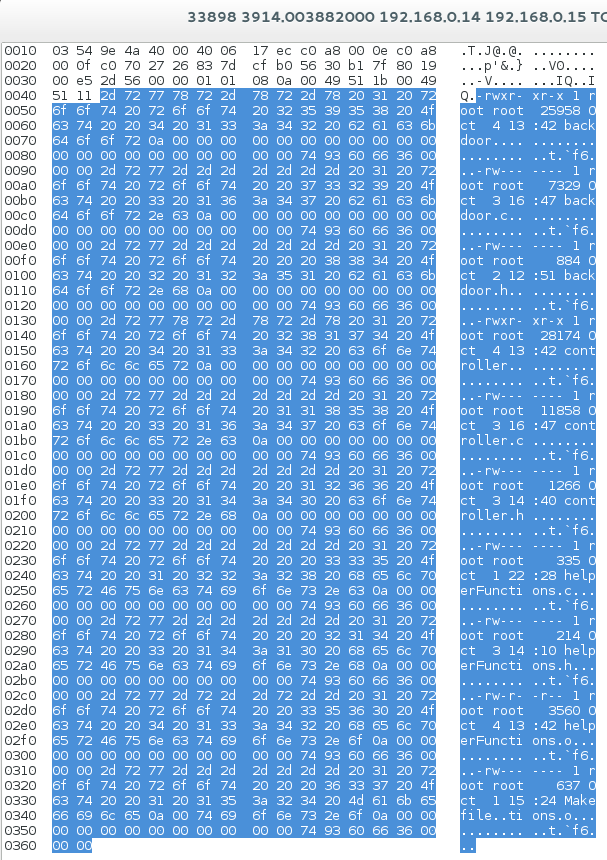


### 6. Controller received content from the Slave

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:



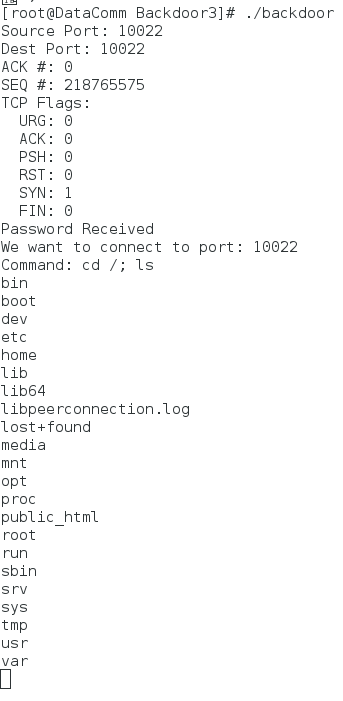
### 7. Once packet is sent, Slave will listen for another packet with password, commands, etc.

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”

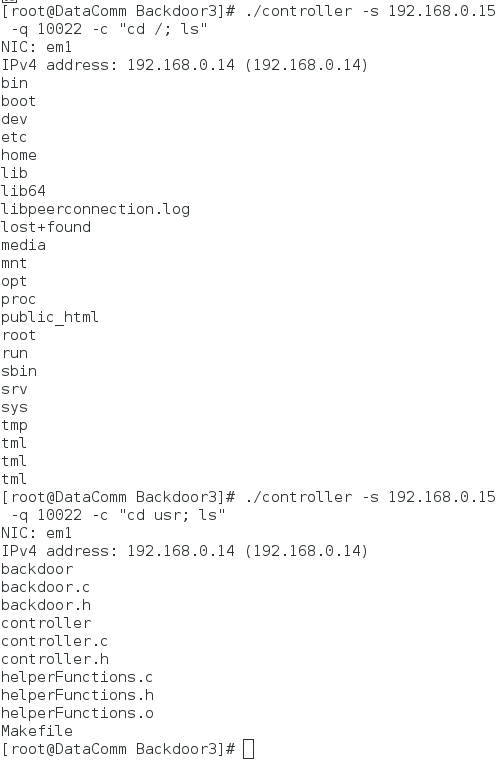
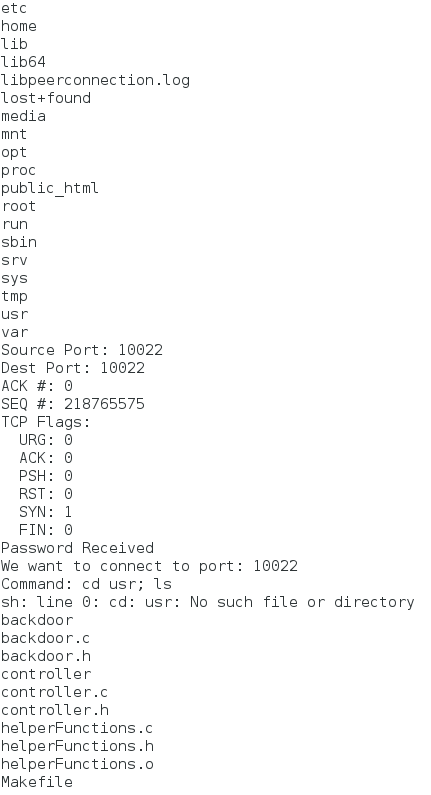
Results from the Slave (left) and Controller (right):



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.