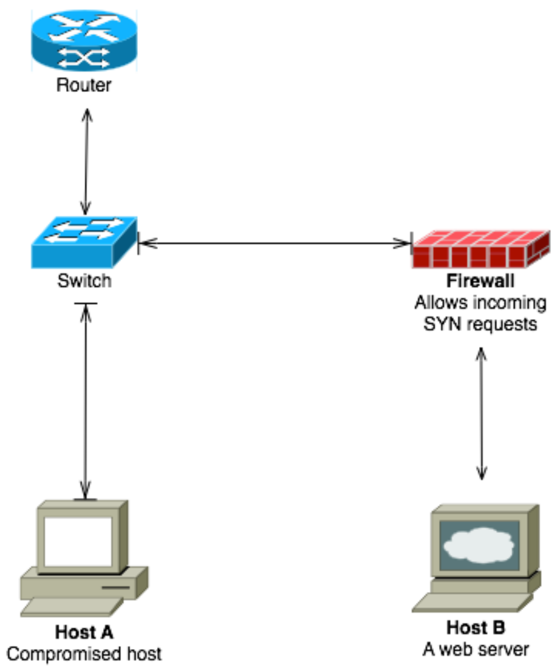
Backdoor Design and Testing

COMP 8505 – Final Assignment
June 2013

Table of Contents

Application Design

Our backdoor consists of two components, each described in detail below. Our backdoor is designed to bypass network security appliances. The following diagrams shows a typical enterprise environment and will be used to explain how our backdoor functions.



Host A is a standard network host, likely a PC. It is connected directly into a switch (or wireless access point). **Host B** is a also a host on the network (in this case a compromised web server). It is behind a firewall. It is important to note that Host A need not be a compromised host and simply needs to be on the same network as Host B. Our application shall be connectionless and rely only on TCP SYN and FIN packets. At no point is a 3-way TCP handshake completed.

Server

The backdoor server application is simple. Listen for traffic from the client on an incoming port. In this case, we would listen to traffic on a known port, such as 80 or 8080. The server shall only read SYN packets and ignore everything else. This will allow the packets to get through a firewall. When traffic comes in, read data from a hidden TCP field (TCP Window in this case) and byte-by-byte read commands. The will be either to receive a command, receive a file, or send a file. If a command is received, it will be executed and the output sent back to the client. The server's listening port shall be dynamically configurable so the server can operate on a variety of compromised hosts (not just web servers).

Client

The client will send SYN requests the compromised server. The client will be able to send a file, receive a file or send a command. It shall receive the output of the command once it's executed on the server. The client will be able to connect to send packets to multiple ports. The user will be able to specify the rate to send packets in order to operate in a more promiscuous mode. The client will also be able to spoof the source IP (but will not get a response back then).

Challenges

Bypass host and network firewalls. Bypass Firewall Appliances

Our client and server applications utilize *libpcap*, a Linux raw socket library for packet crafting. The unique thing about our backdoor application is that it reads packets on the network card level. Firewalls operate on the application layer and by the time they drop our packets, our application has already received (and parsed) them. That takes care of host firewalls. Network firewalls are a little more challenging, but this challenge is overcome by the covert communication (below).

Communicate over a covert-channel and go undetected by an IDS or TCPDump traces.

To overcome this challenge, we'll need a solid understanding of both the TCP protocol, as well as how IDS systems work. It's important to clarify that when we say "covert channel" we are not referring to encryption. Encryption is the opposite of covert as your messages are *obviously* hidden. We want to communicate in a way where somebody looking at our traffic will not only not know what we are communicating, but not know we are communicating in the first place. In order to hide our messages we need to disguise our traffic as "common" traffic. Step 1: get rid of the payload field. IDSs almost always analyze payloads as most exploits utilize it. Step 2: disguise our traffic. How do we communicate without using the TCP payload/data field? Well, we need to use a different field. There are many TCP fields that are irrelevant to packets arriving. Our application hides the payload in the TCP Window field and sends data one byte at a time until we are done communicating (when we send a FIN packet). We set the SYN flag on our packets to mimic an incoming connection request. To an IDS or naked eye looking at captures, our communication will merely appear to be normal incoming connections request. On a web server receiving hundreds or thousands of SYN packets, it becomes next to impossible to separate our backdoor traffic from legitimate traffic and because no firewall in front a web server would *ever* drop SYN requests to port 80, we have effectively bypassed a network firewall. Challenge 1 overcome.

Hide our backdoor on the compromised system

Any half-decent system admin will notice a process running on a system called "backdoor." How do we get past this? Simple. Linux has hundreds of start-up processes that are automatically running (and vital to core functionality). Our server.rb application simply spawns a process, re-names it to something like */sbin/init* and we're in business. Which process you choose largely depends on the context you're using this application in. To avoid detection, applications like these should be designed on a per-environment, per-use basis. Yes, that's a lot of work. Yes, it's the only way to not get caught. For a lazy or impatient hacker, it's only a matter of time before they get caught.

Psuedo-Code

Note that all “sending “ below is done over a covert channel. Packets are sent with the SYN flag set, the payload in the TCP Window field (one byte at a time) and there is no data payload on the packets. They are also encrypted using a basic encryption algorithm (which also makes the TCP window look more realistic).

Server

```
• Listen for incoming packets on a specified port
• Until a FIN packet comes in on the port, grab all packets to the
  port and concatenate the TCP Window Field
• When a FIN packet comes in, split the command on a space
  character. We will have: Array(‘command’, ‘arg)
• If the first element is GET - get(argument)
• If the first element is PUT – put(argument)
• Else try to execute the command and return the output to the
  sender.

• get(arg) #client issued GET. arg is the file to retrieve.

    o If the file does not exist, return # symbol (to indicate
      error).
    o If the file does exist, load it and send it back byte by
      byte.


• put(arg) # client issued a put command. Arg is the filename
    o Listen for packets again and save TCP Window until a FIN
      comes through.
    o Write filename and data to the disk.
```

Client

```
• Wait at prompt() for users input
• Split the input on a space. We will have: Array(‘command’, ‘arg’)
• If command is GET
    o Tell server we are retrieving a file.
    o Listen for that file
• If command is PUT – put()
    o Tell server we are sending a file
    o Open the file, send the filename and the file.
• Else
    o send the input to the server for remote execution.
    o Listen for output from server
• prompt()
```

Tests

No	Name	Description	Expected Result	Actual Result	Pass
1	Covert Channel	Testing covert transport.	Data is sent covertly from the client and parsed correctly by the server	Data is sent covertly from the client and parsed correctly by the server	√
2	File Transfer	Testing file transfer over covert channel. Assumes pass on test #1	File is succesfuly transferred from node a to node b	File is succesfuly transferred from node a to node b	√
3	Remote Control	Tests command execution from node a to node b	Node a sends a command to node b. Node b executes and sends output back to node a.	Node a sends a command to node b. Node b executes and sends output back to node a.	√

Evidence

Test 1: Covert Channel

Below is a capture of the client sending the following file:

Filename: secret_message.txt
Content: “My message is secret.”

Tcpdump was run on the server to capture incoming packets from 10.0.1.29 on port 8202. For readability, some verbose data was surpressed.

TOS 0x01 = End of segment character. Tells server to drop out of current loop. For readability, after each packet, the ascii character is bolded.

Packets received on node b (from node a)

```
16:30:03.244524 IP (tos 0x73, ttl 64, id 423, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:03.301021 IP (tos 0x65, ttl 64, id 838, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:03.341688 IP (tos 0x63, ttl 64, id 922, offset 0, flags [S.],
proto TCP (6), length 20) c
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]
```

```
16:30:03.347878 IP (tos 0x72, ttl 64, id 152, offset 0, flags [S.],
proto TCP (6), length 20) r
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:03.35989 IP (tos 0x65, ttl 64, id 956, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:03.361032 IP (tos 0x74, ttl 64, id 421, offset 0, flags [S.],
proto TCP (6), length 20) t
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:03.421071 IP (tos 0x5f, ttl 64, id 713, offset 0, flags [S.],
proto TCP (6), length 20) _
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:03.433024 IP (tos 0x6d, ttl 64, id 833, offset 0, flags [S.],
proto TCP (6), length 20) m
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:03.991021 IP (tos 0x65, ttl 64, id 630, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:04.101002 IP (tos 0x73, ttl 64, id 911, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:04.235828 IP (tos 0x73, ttl 64, id 857, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:04.301021 IP (tos 0x61, ttl 64, id 167, offset 0, flags [S.],
proto TCP (6), length 20) a
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:04.512322 IP (tos 0x67, ttl 64, id 337, offset 0, flags [S.],
proto TCP (6), length 20) g
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]



16:30:04.811020 IP (tos 0x65, ttl 64, id 148, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:04.931317 IP (tos 0x2E, ttl 64, id 894, offset 0, flags [S.],
proto TCP (6), length 20) .
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]
```

```
16:30:05.001475 IP (tos 0x74, ttl 64, id 768, offset 0, flags [S.],
proto TCP (6), length 20) t
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:05.335322 IP (tos 0x78, ttl 64, id 772, offset 0, flags [S.],
proto TCP (6), length 20) x
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:05.441496 IP (tos 0x74, ttl 64, id 153, offset 0, flags [S.],
proto TCP (6), length 20) t
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:05.546941 IP (tos 0x01, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) [EOS]
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.127410 IP (tos 0x4D, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) M
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.130015 IP (tos 0x79, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) y
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.149937 IP (tos 0x32, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) ' '
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.291155 IP (tos 0x6D, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) M
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:06.322841 IP (tos 0x65, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:06.391488 IP (tos 0x73, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.410010 IP (tos 0x73, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.516551 IP (tos 0x61, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) a
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.552861 IP (tos 0x67, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) g
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.857222 IP (tos 0x65, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) e
```

```
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.890336 IP (tos 0x32, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) ‘ ’
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:06.997511 IP (tos 0x69, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) i
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:07.004128 IP (tos 0x73, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:07.092333 IP (tos 0x32, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) ‘ ’
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]

16:30:07.214268 IP (tos 0x73, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) s
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:07.319585 IP (tos 0x65, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) e
10.0.1.29.8202 >10.0.1.31.8202 Flags [S]


16:30:07.346941 IP (tos 0x63, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) c

16:30:07.426941 IP (tos 0x72, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) r

16:30:07.440153 IP (tos 0x65, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) e

16:30:07.486941 IP (tos 0x74, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) t

16:30:07.49789 IP (tos 0x01, ttl 64, id 218, offset 0, flags [S.],
proto TCP (6), length 20) [EOS]
```

Test 2: File Transfer

Continuing from the above test, we examine the files sent by the client, and in turn, written to the server.

Client Side

```
Vm1-fedora1:Assignment1 atmaledy$ stat secret_message.txt

16777218 3003852 -rw-r--r-- 1 atmaledy staff 0 20 "Apr 29 16:28:24
2013" " Apr 29 16:28:24 2013" "Apr 29 16:28:24 2013" "Apr 29 17:24:20
2013" 4096 8 0x40 secret_message.txt
```

Server side

```
Vm2-fedora:Assignment1 atmaledy$ stat secret_message.txt

16777218 3003852 -rw-r--r-- 1 atmaledy staff 0 20 " Apr 29 16:30:6
2013" "Apr 29 16:30:06 2013" "Apr 29 16:30:07 2013" "Apr 29 16:30:07
2013" 4096 8 0x40 secret_message.txt
```

Here you can see the date created/modified matches up with the packet capture. The file size from the client and server also match accordingly.

Test 3: Remote Control

Server (folder contents italicized)

```
    Vm2-fedora1:Assignment2 ls
    Top_secret_file.txt Passwords Missile_launch_codes

    Vm2-fedora1:Assignment2 atmaledy$ ruby src/server.rb
```

Client (server output italicized)

```
    Vm1-fedora1:Assignment2 atmaledy$ ruby src/client.rb Enter a
    command > lsRemote Server Says:
    Top_secret_file.txt
    Passwords
    Missile_launch_codes
```

Above you can clearly see that the server has sent the data returned by the client-issued-command back. We can see that the client has displayed it accordingly.