

BEACON LABS

CYBERSECURITY EDUCATION LABS

Off-Path TCP Exploits Lab

Module: Network Security



BOISE STATE UNIVERSITY

COLLEGE OF ENGINEERING

Department of Computer Science

Table of Contents

1. Overview	3
2. Virtual Machine Setup	4
3. Lab Task Set 1: Clock Synchronization	6
4. Lab Task Set 2: Four-Tuple Inference	9
5. Lab Task Set 3: Sequence Number Inference and Nash Equilibrium	10
6. Lab Task Set 4: Acknowledgement Number Inference	12
7. Submission	13
References	14

1 Overview

This attack is executed on Linux Kernel 3.13. In Linux Kernel 3.6 RFC 5961 was faithfully implemented to stop blind in-window attacks, but it also created a new vulnerability. RFC 5961 proposed a window outside of the correct ACK-window where the server would respond with a challenge-ACK to packets that did not have the correct sequence or ACK number. Figure 1 demonstrates this below. To prevent excess resources from being used on these challenge-ACKs, a limit of 100 per second was implemented, known as the Global Rate Limit. This limit is where the vulnerability lies.

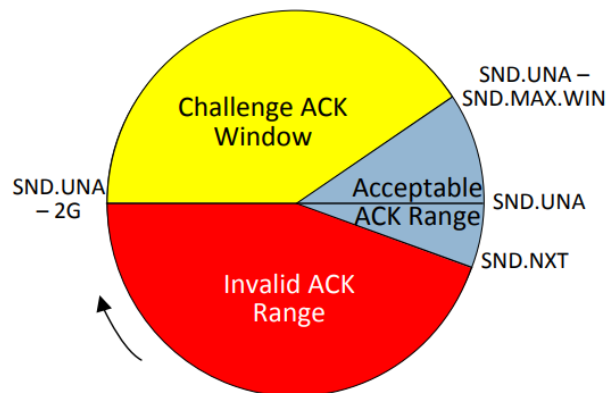


Figure 1: ACK Window Illustration

The Global Rate Limit can be exploited to show if a TCP connection is present, and then subsequently be used to infer the four-tuple of the client and server, infer the next acceptable sequence number, then finally infer the acknowledgment number. After this is completed it is trivial to inject spoofed packets.

Readings and Videos:

Video explaining how the attack works:

<https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao>

Paper that the attack is based upon:

https://www.cs.ucr.edu/~zhiyunq/pub/sec16_TCP_pure_offpath.pdf

The objective of this lab is to have students understand how this attack is carried out and write some of the code necessary to execute it.

2 Virtual Machine Setup

In this lab we will have three virtual machines: the client, server, and attacker. There are two virtual machines to download as the client and attacker use the same image. Through a python script to create a simple messaging service between the client and server, we create the TCP connection that will be hijacked.

The server is running a version of Ubuntu 14.04 that has not been updated. The security updates for this operating system patch the vulnerability that we are exploiting, so we must be sure not to update it.

The client and the attacker are running Ubuntu 16.04. The attacker also has the libtins libraries installed. They are the libraries that we use to spoof and send packets, so they are necessary to have on the virtual machine.

By interacting with the server as seen in figure 2, we can complete all steps of the attack. After this is completed we can then inject packets imitating the server to the client, as seen in figure 3.

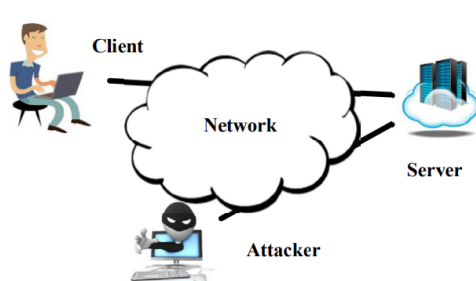


Figure 2: Exploiting Server

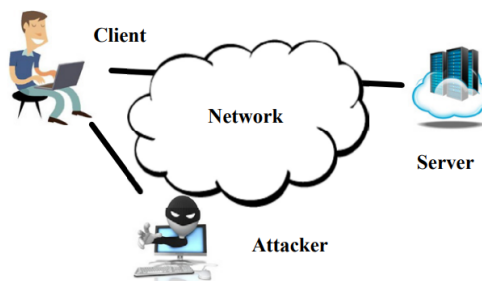


Figure 3: Imitating the Server

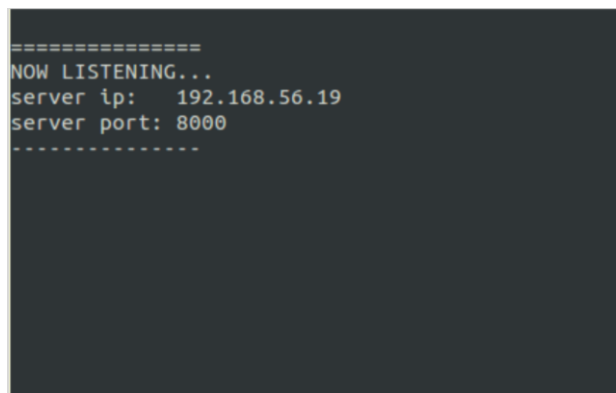
Make sure all virtual machines only have a Host-Only Adapter in their network adapters.

2.1 Connecting Client and Server

On the desktop of the server and client there is a file labeled `tcp_server.py` and `tcp_client.py` respectively. The code requires the window to be larger than the standard size, so put the terminal in full screen before executing commands. Through the terminal navigate to the desktop on the server first and run the code via the command:

```
$ sudo python3 tcp_server.py <server_ip> <server_port>
// Find your server ip and use it as an argument
// Choose a port to host the connection on. Make sure no other connection are using that port
```

Once this is complete your server terminal should look like figure 4.



```
=====
NOW LISTENING...
server ip: 192.168.56.19
server port: 8000
-----
```

Figure 4: Server Waiting for Connection

Once that is complete, we can connect the client to the server. Navigate to the desktop again and run `tcp_client.py` using the same arguments:

```
$ python3 tcp_client.py <server_ip> <server_port>
```

Your client terminal should look like figure 5, and your server terminal should look like figure 6.

2.2 Setting up the Attacker

The attacker code is saved in the folder `attacker_cpp` in documents. When you alter the code and want to compile it again, remove the file named `exploit`, then run the command `make` in the terminal whilst in the folder.

The attack can then be run from the terminal by typing:

```

=====
CLIENT CONNECTED...
ip: 192.168.56.19
port: 8000
-----
[Empty box]
(Enter to send)(Ctrl+H to backspace)

```

Figure 5: Client Terminal after Connection

```

=====
CLIENT CONNECTED
server ip: 192.168.56.19    client ip: 192.168.56.7
server port: 8000         client port: 50106
-----
ip:port == 192.168.56.7:50106
[Empty box]
(Enter to send)(Ctrl+H to backspace)

```

Figure 6: Server Terminal after Connection

```
$ sudo ./exploit <client_ip> <server_ip> <server_port>
```

The server port will be the same as the one used to set up the tcp connection. The attack will not work yet though as it is missing code that will be added by you. After attempting the attack on the tcp connection, further attacks on the same connection may be slow or unsuccessful. Restarting the tcp connection of the server and client on a new port will mitigate this.

3 Lab Task Set 1: Clock Synchronization

To make sure all of our packets arrive within the same time interval, we must first synchronize our clock with the server. This is done by first initiating a legitimate TCP connection. In `synchronize_clock.cpp` (Figure 7) you must choose what flags should be sent to create this connection.

Write what flag should be set to first initialize the TCP connection on line 25, then disable this flag and set another one that will be used after receiving the response from the server on lines 47 and 48.

```

16
17  /* The attacker will use a random port between 4k and 9k */
18  std::random_device rd;
19  std::mt19937 mt(rd());
20  std::uniform_int_distribution<> dist(4000, 9000);
21  int attacker_legit_port = dist(mt);
22  IP legit_IP = IP(SERVER_IP) / TCP(SERVER_PORT, attacker_legit_port);
23  TCP &legit_TCP = legit_IP.rfind_pdu<TCP>();
24
25  legit_TCP.set_flag(TCP::Flags::/* Delete this comment write code here */, 1);
26
27  std::unique_ptr <PDU> response(sender.send_rcv(legit_IP));
28
29  if (!response) {
30      std::cout << "No response!" << std::endl;
31  }
32
33  TCP &legit_TCP_response = response->rfind_pdu<TCP>();
34  if (!(legit_TCP_response.get_flag(TCP::SYN) &&
35      legit_TCP_response.get_flag(TCP::ACK))) {
36      std::cout << "Didn't respond with SYN ACK!\n";
37      std::cout << "Check given server PORT...\n";
38      exit(-1);
39  } else {
40      std::cout << "connection established\n";
41  }
42  uint32_t seq_response = legit_TCP_response.seq();
43  //increment ack seq
44  legit_TCP.ack_seq(seq_response + 1);
45  legit_TCP.seq(1);
46
47  legit_TCP.set_flag(TCP::Flags::/* Delete this comment write code here */, 0);
48  legit_TCP.set_flag(TCP::Flags::/* Delete this comment write code here */, 1);
49

```

Figure 7: *synchronize_clock.cpp* Step 1

After altering the code, compile it by navigating to the `attacker_cpp` folder, removing the `exploit.exe` file, and then entering the `make` command.

Once we have our legitimate connection established we must now send 200 packets equally spaced in the span of a second to see how many challenges this prompts. Since only 100 challenges can be sent within a second, we know that our clock is synchronized when we only receive 100 challenges.

There is a chance that the clock will already be synchronized, but it is far more likely that we will end up receiving more than 100 challenges. We try this twice and adjust the time between the first and second round. The third round uses the calculated time delay necessary to receive only 100 challenges. Your task is to write code to equally space out the 200 packets in the span of a second so that each packet has an equal amount of time between the one before and the one after and use the correct time delay for round one, two, and three.

The way the code is currently, all packets will be sent as fast as the computer can. Your task is to create a delay between each packet being sent

out so that all 200 are equally spaced within the span of a second, having the same amount of time between the packet before it and the one after. Make sure to add the variable "start" to your time delay. Write your code on line 75 for round one as seen in figure 8.

```
71
72     for (int i = 1; i <= 200; ++i) {
73         sender.send(eth);
74
75         // WRITE CODE HERE:
76 |
77
78     }
79
```

Figure 8: *synchronize_clock.cpp* Step 2

In the second round we want to add an additional three seconds to the time delay to make sure all packets from the last round were responded to and the global rate limit has been reset. We also want to add another 5 milliseconds to the time delay so that a different number of challenges are received and we can calculate what the final adjustment of time should be. Don't forget to add the start to the time delay again. Write your code on line 98 for round two as seen in figure 9.

```
94     // Round 2
95     for (int i = 1; i <= 200; ++i) {
96         sender.send(eth);
97
98         // WRITE CODE HERE:
99
100
101     }
102 |
```

Figure 9: *synchronize_clock.cpp* Step 3

For the final round we want to send another group of equally spaced packets with the delay of 6 seconds (3 from the first round, and an additional 3

from the second). For this final round though we want to add the variable of `n_result` to our time delay to make sure our calculation was correct and only 100 packets are received in a second. A margin of error of one packet (101 received) is acceptable. Write your code on line 127 for round three as seen in figure 10.

```
123     for (int i = 1; i <= 200; ++i) {  
124         sender.send(eth);  
125  
126  
127         // WRITE CODE HERE:  
128  
129     }
```

Figure 10: `synchronize_clock.cpp` Step 4

After altering the code, compile it by navigating to the `attacker_cpp` folder, removing the `exploit.exe` file, and then entering the `make` command. There is a redundancy in the code to try the time synchronization again in case there was any packet loss. If the code is written correctly by the student it should only take one round to complete but there is a possibility it will need a second or third.

4 Lab Task Set 2: Four-Tuple Inference

Since there is a tcp connection between the client and server, a SYN-ACK packet sent with that four-tuple will prompt a challenge-ACK. If the SYN-ACK packet is sent to a port that does not have a tcp connection taking place, the server will send an RST packet in response. We use this response to find the correct client port.

Over the course of a second, we send out SYN-ACK packets to the server with the four-tuples of `<srcIP = clientIP, dstIP = serverIP, srcPort = X, dstPort = serverPort>`. The source port is labeled as X as it is the part that we need to figure out. The default linux port range is 32768 to 61000. By halving this and sending packets from 45000-61000, we can determine which part of this range holds the correct port number.

Once we have sent out SYN-ACK packets in the first half, we can send 100 rst packets during the same second. If 100 challenge-ACKs are observed, we know it is in the first half of the search space. If 99 are observed, we know one was sent to the client, and thus the port is in the first half. Write how we use the amount observed to find if the correct client port is in the range we searched and how we should adjust the right or left port accordingly. Write your code on line 67 as seen in figure 11.

```
62     }
63     // waiting to make sure all packets can be recieved
64     std::this_thread::sleep_until(clock);
65     |
66
67     // WRITE CODE BELOW:
68
69
70
71
```

Figure 11: 4tuple_inference.cpp Step 1

After altering the code, compile it by navigating to the attacker_cpp folder, removing the exploit.exe file, and then entering the make command. Once the client port is found we can move on to finding the sequence number.

5 Lab Task Set 3: Sequence Number Inference and Nash Equilibrium

Challenge-ACKs are prompted for RST packets that are received if the sequence number is in the window, but does not exactly match the next expected sequence number (RCV.NXT). Knowing this, we can break the sequence number space into blocks of the received window size and send packets to each block.

To find the correct block, we can either go through the search space by increasing our block_id, or decreasing it. Below is a game theory table (Table 1 representing the advantages and disadvantages of both.

The defender has the option of starting the sequence number on a high value or a low value. Starting on a high value costs more resources, thus

		Defender	
		Start Low	Start High
Attacker	Search Low	(2, -2)	(2, -2)
	Search High	(1, -1)	(3, -3)

Table 1: Game Theory

it gives the attacker 2 and the defender -2. If the defender starts it on a low value, then it's 1 and -1 instead. If the attacker chooses correctly to search high or low first, then another 1 and -1 is given to the attacker and defender respectively.

Analyze this game theory table and make a decision of what would be more advantageous as an attacker: Searching high or searching low. Once you have made your decision, write the for loop on line 61 to reflect it. Give an explanation for your decision in your documentation.

The definitions of the variables are as follows. SEQ_MAX is the maximum value a 32 bit int can be as that is the largest sequence number WINDOW_SIZE is 12,600 as that is the default window size for linux. chunk_size is 8000 as that is the area we send a packet to each round. SEQ_MAX/chunk_size/WINDOW_SIZE is the maximum chunk_id we would search to if counting up from 0.

Three steps must be taken to find the correct sequence number. First by approximating the range in which the number lies (above), second finding the correct block, and finally finding the sequence number. In step one, you must also write code to send packets on the legitimate connection to exhaust the global rate limit on line 73, and an if statement to check if the correct block is in the current search space on line 91 as seen in figure 12.

After altering the code, compile it by navigating to the attacker_cpp folder, removing the exploit.exe file, and then entering the make command.

In step 2, we narrow down the search space to a single block. This is done in a similar method to the 4-tuple inference where we progressively narrow down our search space by seeing how many packets are received after sending them to one half. You do not have to write any code for this

```

65
66 // Sending spoofed packets to test where the sequence number is
67 for (uint32_t block_i = 0; block_i < chunk_size; block_i++) {
68     spoofed_TCP.seq((block_i * WINDOW_SIZE) + chunk_id * chunk_size * WINDOW_SIZE);
69     sender.send(spoofed_IP);
70 }
71
72 // WRITE CODE HERE:
73
74
75
76
77
78 auto finish = std::chrono::high_resolution_clock::now();
79 clock += std::chrono::seconds{2};
80 std::this_thread::sleep_until(clock);
81 std::chrono::duration<double> elapsed = finish - start;
82 std::cout << ".\t\t\t\t\t" << chunk_id * chunk_size * WINDOW_SIZE
83     << " and " << (chunk_id + 1) * chunk_size * WINDOW_SIZE << "\t\t\t\t\t"
84     << "\t\t\t\t\t" << elapsed.count() << "s\t\t\t\t\t"
85     << "packet_count: " << packet_count << "\n";
86 if (elapsed.count() > 0.95) {
87     std::cout << "Unexpected delay detected! Redoing chunk!";
88     --chunk_id;
89 }
90 //WRITE CODE HERE
91
92

```

Figure 12: *sequence_inference.cpp* Step 1

step.

In step 3 we iterate through sequence numbers until we find the correct one. Once we have found it we can proceed to inferring the Acknowledgement Number. You do not have to write any code for this step.

6 Lab Task Set 4: Acknowledgement Number Inference

To find the next expected acknowledgement number of the server, we use a multi-bin search. The maximum window size for acknowledgement numbers is 2^{30} . We define this in the code as `one_G`. By sending one packet to the 0 sequence number, two to `one_G - 1`, four to `two_G - 1`, and eight to `three_G - 1`, we can then count the amount of packets received to find the area in which the correct number lies. For instance, if 94 challenges are received, then we know `one_G` and `two_G` have triggered challenges, thus the range is in between the two of them. If 96 challenges are observed, then we know only `two_G` has triggered challenges, thus the range is inside of `two_G`. Your task is to write how the code will handle missing four packets on line 89, eight packets on line 100, and 12 packets on line 11 as seen in figure 13.

```
86      case (ACK_LIMIT - 4):
87
88          // WRITE CODE HERE:
89
90
91      break;
92      case (ACK_LIMIT - 6):
93          left_ack = 0;
94          right_ack = 2 * one_G - 1;
95          break;
96      case (ACK_LIMIT - 8):
97
98          // WRITE CODE HERE:
99
100
101      break;
102      case (ACK_LIMIT - 9):
103          left_ack = 2 * one_G - 1;
104          right_ack = 4 * (one_G - 1);
105          break;
106      case (ACK_LIMIT - 12):
107          |
108
109          // WRITE CODE HERE:
110
111
112      break;
```

Figure 13: *ack_inference.cpp*

After altering the code, compile it by navigating to the `attacker_cpp` folder through the terminal, removing the `exploit.exe` file, and then entering the `make` command.

Once this is completed we can find the left boundary of the acceptable ACK range, and then create spoofed packets that can be sent to the client. If all code is working correctly, you should be able to write messages into the console that will show up as the server in the client's messaging terminal.

7 Submission

Submit a lab report using the provided [lab report template](#). Create a lab report that consists of text and images from each task in the lab. Describe the process you went through to solve each task. Be sure to include how you solved the game theory table. With your submission include the `synchronize_clock.cpp`, `4tuple_inference.cpp`, `sequence_inference.cpp`, and

ack_inference.cpp files.

References