Harry Langford
hjel2@cam.ac.uk

Part I

Seed Labs

1 TCP Attacks Lab

1.1 SYN Flooding Attack

I launched the attack through Python, while I noticed an increase in response time from the client, I was still able to telnet into it. When I retried the attack with C, the increased response time made the system unusable. However, I was still able to telnet into it until I decreased the queue size to 80. After enabling SYN cookies, the client worked perfectly and I did not observe any impacts of the attack.

1.2 TCP TST Attacks on `telnet` Connections

I setup a telnet connection between two docker images and was able to use wireshark to find the source port, destination port, source IP, destination IP and sequence number. I was able to successfully close the connection.

1.3 TCP Session Hijacking

I setup a telnet connection between two docker images and was able to use wireshark to find the source port, destination port, source IP, destination IP and sequence number. I was able to send packets which got acknowledged and accepted but did not find commands which actually *did* anything.

I got the attack vector working but couldn't find a payload

Part II

Exam Questions

2 2006 Paper 3 Question 9

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2006p3q9.pdf

(a) Name *three* types of software vulnerability; give an example for each and a brief description of how each could be exploited:

- Metacharacter Vulnerability

  SQL Injection, CSRF and XSS are examples of vulnerabilities which can arise as a result of Metacharacter Vulnerability. Metacharacter Vulnerability is defined as when a user is allowed to input characters which have special meanings and are not interpreted as data. In SQL Injection, the user can close the quote, add a comment and write their own query.

- Buffer Overflow

  Return to LibC attacks in a setuid program can yield a root shell. In Return to LibC, the attacker overwrites the buffer, sets the return address of the buffer to be a library function and passes it an argument which causes it to act maliciously. For example, jumping to `system()` with the argument being a pointer to an environment variable storing `/bin/sh` will open a shell.

- Race Condition

  A race condition is where two operations happen concurrently; and the one which finishes first will succeed. An example of this is a TCP RST attack – the attacker eavesdrops the sequence number and infers the next sequence number. They then spoof a TCP RST message. If this RST message arrives before the senders next message, then the connection will be forcibly closed.

(b) Alice wants to attack Bob's computer via the Internet, by sending IP packets to it, directly from her own computer. she does not want Bob to find out the IP address of her computer.

(i) Is this easier for Alice with TCP or UDP based application protocols? Explain why.

It's easier for Alice with UDP. UDP is connectionless and only uses destination port and IP address to multiplex messages – UDP also does not have a handshaking protocol. If Alice sends UDP messages (with random source IP addresses) then they are treated exactly the same as messages from any other device. If they are sent at a sufficiently fast rate then Bob will not have enough time to process the real UDP messages.

TCP has a handshaking protocol which establishes a connection. Alice would not receive the response from Bob (since she has to use a fake IP address). Alice would then be unable to setup the connection to send messages. Alice is also unable to send messages to Bob at a sufficiently fast rate since only a small number of messages would be accepted (due to the window size).

(ii) For the more difficult protocol, explain *one* technique that Alice could try to overcome this obstacle and *one* countermeasure that Bob could implement in his computer.

Alice could use SYN flooding instead. Rather than attempting to overwhelm Bob with normal messages, Alice could flood him with SYN requests, filling up his TCB and making him unable to process any legitimate requests.

The first stage of handshaking is for the client to send a SYN to the server "I'd like to send data to you". The server then allocates some memory storing this (known as a TCB) and sends its own SYN + ACK "I hear you and would like to send data to you". The server will then resend its SYN + ACK if they go missing and eventually give up. However, during this process, the TCB is still allocated – using up memory.

The attack is for Alice to send many SYN requests from random IP addresses. Bob will then store TCBs and eventually run out of memory and be unable to accept any new communications.

This can be combated by using a SYN cookie as the sequence number. A SYN cookie is a MAC (Message Authentication Code); a hash of the servers private key, the time (at low precision i.e 64 seconds), the clients IP address, port number and the sequence number they sent. On receiving any ACK, Bob will use the data in the ACK to reverse engineer the SYN cookie he would have made for the SYN – if the sequence number of the ACK is exactly one larger than this then Bob accepts the connection, else he rejects it. Bob will not allocate any state for a connection until it is established. Connections can only be established if the client *actually receives the SYN + ACK response*. An additional protection is to store a list of "known" clients and allocate a certain proportion (i.e a quarter) of TCBs for their exclusive use. Usage of a SYN cookie has a very low probability of failure (due to storing a timestamp). It also adds some overhead in the form of hashing. It's therefore common practice to dynamically enable these protections when Bob detects he is under a SYN flooding attack.

(iii) Name *three* functions that Alice's Internet Service provider could implement to make it more difficult or Alice to achieve her goal.

- Alice's ISP could drop any outgoing packets with a source address which is not found on the subnet.
- Alice's ISP could limit the number of open flows which can be made through a single home router.
- Alice's ISP could limit the rate at which Alice can make SYN requests.

(c) In what way are TCP/UDP port numbers below 1024 special?

Ports below 1024 are "privileged" and can only be used by root.

3 2018 Paper 4 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2018p4q7.pdf

(a) An application process receives information via a UDP packet over a wired Ethernet LAN connection. If the packet carries a source port number below 1024, under which conditions can information be trusted.

I was unable to find a convincing answer to this. I will therefore state conditions under which this is guaranteed to be trustworthy:

- The source address is from a node on the LAN and the LAN is not connected to the wider internet or the router / switch filters packets such that no adversary can spoof packets with IP addresses on the subnet; AND we trust no adversary on the LAN has root access.
- The UDP packet contains information which authorises it via a secure protocol. For example, a private key signature of: a challenge sent to the node, the content of the data and the metadata.

(b) What is a *UDP-based amplification attack* and why are similar attacks far less practical via TCP?

A *UDP-based amplification attack* is an attack where the attacker sends a small packet to a server with the return address of the victim. The server then responds to this request by sending a large amount of data to the victim. The actual attacker can cause the victim to receive hundreds of times more data than they actually sent. UDP is connectionless and does not verify source addresses – so this attack is trivial.

TCP is not connectionless, for the attacker to get the server to send data to the victim, they would have to establish a connection with the server as the victim. This would be challenging. Furthermore, on receiving the first packet, the victim would send RST to the server closing the connection. So the attacker would have to re-establish the connection repeatedly.

(c) Name and briefly explain *four* techniques that the designers of C compilers and operating system kernels can implement to reduce the risk of stack-based buffer-overflow attacks.

- Address Space Layout Randomisation (ASLR)

  ALSR randomises the virtual addresses at which different parts of the stack are placed. This means the absolute addresses cannot be determined with certainty and reduces the chance of a successful attack. This can be implemented by Operating System designers.

- Non-executable Stack

Marking the stack as non-executable prevents the attacker from inputting the bytecode which they wish to execute. However, the machine is still vulnerable to a return-to-libc attack. This can be mitigated by using shells which immediately revert to the real user ID – so the worst that a user can do is execute a shell with the permissions they already have. This approach is taken by default in Ubuntu.

- Shadow Stack

  The program can keep a second copy of the stack (known as a shadow stack) which only contains the return address and other basic information (no local variables). When a function is called, the shadow stack is updated and when a function returns, the value on the real stack is compared against the value on the shadow stack. If they are inconsistent then the program terminates. This must be implemented by C compiler designers.

- Stack Canary

  A Stack Canary is a word which comes immediately below the return address of each function. It is a random sequence of bytes (of which a copy is stored elsewhere, off the stack). On returning from a function, the program will check that the stack canary matches the copy. If it does not, the program will terminate, having detected "stack smashing".

(d) How can an implementation of the C function `strcmp()` cause a vulnerability to a side-channel attack, and how can this be mitigated.

If the C function `strcmp()` takes time proportional to the length of the input which are the same (i.e returns once failure is known) then we can determine the value of the string we are comparing against by using a timing attack.

A timing attack on a non-constant length function is as follows: try a set of random strings. The one which took the longest time had the longest matching prefix. Work forwards, adjusting characters until changing a character makes the comparison no faster. This is the first character which is incorrect. Alter it until the comparison is slower. Repeat for subsequent characters.

This can be mitigated by making `strcmp()` take time which is only dependent on the *length* of the input. For example (assuming s1 is the input string):