COMP 4580 Computer Security

Aaron Noyes 7831181

Research Project – SSL based VPN

April 6 2020

seedVPN

SeedVPN is a Virtual Private Network (VPN) that uses Transport Layer Security (TLS) to set up secure connectiond between a server and a number of clients. The VPN offers a variety of features to ensure that messages between clients and the server are delivered securely. The VPN also provides virtual addresses within the virtual network that may be used for communication. SeedVPN is based on the "Virtual Private Network Lab" described by Wenliang Du and other collaborators at Syracuse University (Du, n.d.). The basic workings of the VPN along with design decisions and features are detailed in this report. All information about building and running the VPN can be found in the associated "README.md" file.

TUN/TAP

Linux provides a low-level virtual network interface referred to as either TUN or TAP depending on the functionality needed. A TUN device is used here because it deals directly with IP packets. The TUN interface acts in the same way as a hardware network interface in the sense that packets can be routed to it. TUN differs from typical network interface because packets routed to it are delivered to a userspace program rather than being sent outside of the computer.

The basis for seedVPN is a tunnel program described in the Linux documentation modified by Davide Brini (Brini, 2010). This program is used to set up a tunnel between two devices where a server which waits for a client to connect via a Transmission Control Protocol (TCP). Both the client and server open a TUN (or TAP depending on configuration) by opening "/dev/net/tun" and issuing ioctl() calls to create and configure the device. Brini's program creates this interface using a function named tun_alloc() which allows a caller to also choose a name for this device. Naming is useful for adding routing information, however does not need to be supplied as the TUN/TAP driver will pick a name if none is supplied.

Once this device is created opened, it can be enabled and routing information can be added to it either through the command line programs “ip” and “route” or directly via ioctl() calls. The lab uses the former method, however seedVPN supplies functions to automate this configuration which are described below. When the device is enabled and assigned an address, packets routed to it can be read using a standard read() call. Brini’s program reads the packets sent to this device and relays them directly between the client and server – creating a tunnel for communication.

The TCP connection established here does not lend itself nicely to the problem of a VPN. TCP ensures orderly reliable connections, but User Datagram Protocol (UDP) sockets are better suited for the VPN because packets being sent between clients and the server do not require any state information. Additionally, the efficiency of UDP means that packets are routed quickly and ensure the VPN does not bottleneck information. A TCP connection is useful later in the implementation serving as a control channel between connected peers. This will be addressed in subsequent sections.

Brini’s program serves as a useful starting point because of the TUN device creation function and the application logic for reading packets and sending them over the tunnel. As it is, however, information sent between the client and server in this program is unencrypted and not much different than sending packets normally through a socket. In addition, routing information must be manually entered on the client and server machines. The communication channel via tunnel may be secured to form the basis of the VPN.

Gateway-to-Gateway

The SEED lab also specifies the need to create a tunnel between two gateways. This adds a network structure and allows several hosts to be connected to the network but is not impacted much by the implementation of the VPN or tunnel. This configuration involves connecting the client and server as usual with the addition of a few routing details. These details are the same for both the client and server, as at this point both behave in an identical fashion after the tunnel has been created.

The client and server may both be connected to several hosts in a private network. Using VirtualBox this can be achieved via an “Internal Network” adapter between the client or server and any number of hosts. The client or server act as a gateway by setting “net.ipv4.ip_forward” equal to 1, allowing any incoming packets to be forwarded along the network instead of being read. For example, assume a gateway connected to a peer via the tunnel. The peer’s address is 10.0.2.2 and the virtual interface “tun0”. The peer is connected to a destination device with the address 10.0.6.2. The gateway could use “sudo route add -net 10.0.2.0 netmask 255.255.255.0 dev tun0”. With IP forwarding enabled, the sending host would set the gateway as it’s gateway for addresses 10.0.6.0/24, and any packet it destined for that range would get send to the gateway, then over the tunnel to the peer, then finally to the destination.

These details should be configured statically based on networking needs, but the implementation of the tunnel is not affected by this.

Encryption

The next step in creating seedVPN is to add encryption to ensure that messages passed over the tunnel cannot be snooped and read by other hosts on the network. To achieve this, it is first important to understand how data flows along the tunnel. After the tunnel is established when the client or server will receive packets in the program that opened the TUN interface. This program can then alter the data before it is sent. The peer receives the altered packet over its regular network interface, then it may alter the packet before passing it along to its own TUN interface. This provides a seam for encryption. Before a peer sends a packet along the UDP tunnel, it can encrypt it. When a peer receives a packet over the network, it can decrypt it and then pass it along.

OpenSSL provides the necessary mechanisms for encryption through its EVP library of functions. For seedVPN, “aes.c” implements two functions that interface with OpenSSL: encrypt_aes() and decrypt_aes(). These functions take a session key, initialization vector (IV), and a message, then use the EVP library to encrypt and decrypt as needed. These functions use OpenSSL’s EVP_aes_256_cbc() method for encryption and decryption.

For 256-bit AES, an IV of any size may be used, but they key has to be 256 bits or 32 bytes. For consistency, seedVPN restricts IVs to 16 bytes. This serves as a simple mechanism to ensure some security for client and server supplied IVs since they are stored in buffers in memory. The tunnel peers must also be able to share a key for this to work, this is addressed in a later section.

Integrity

As specified in the SEED lab, seedVPN uses SHA256-based HMACs to ensure messages sent between peers are not altered either intentionally or through error. SeedVPN uses a library written in "hmac.c" which provides two public interfaces: sign_hmac() and verify_hmac(). Similar to the AES functions, these wrap around OpenSSL's EVP library. Another private function is available to the sign and verify methods which creates the necessary EVP_CTX object which contains the HMAC digest method (SHA256) and the secret key used to generate the HMAC. The same secret session key is used for HMACs that is used for AES. This is not necessary but is useful in the scope of this project.

Before a peer sends a message over the tunnel, it computes a SHA256 HMAC using the same session key shared and the sign_hmac() function and adds it before the encrypted message. Internally, the verify_hmac() function receives a message and an HMAC, then uses sign_hmac() to calculate an HMAC from the received message and compares it to the supplied HMAC. This method is not necessary but ensures consistency in how both parties compute HMACs. Comparison between the calculated and received HMACs is done using libc's memcmp() function since the length is known to be 32 bytes. When a peer receives data over the tunnel, it assumes that the first 32 bytes of the message are the HMAC and loads that into memory for comparison. This operation is safe because the HMAC is of known size. The peer then loads the message, decrypts it, and verifies that the HMAC it calculates is the same. If it is, and the message is decrypted, the receiver knows that the sender shares a session key, and that the message has not been altered.

Key Exchange

The VPN needs to be able to share a session key. One way to do this would be to share a private key in a file before setting up the connection, but this is unnecessary overhead for the purpose of this lab. SeedVPN uses a TLS-based control channel between the server and clients to share sensitive information, and later, reconfiguration commands.

First, the server and client need to create a TCP connection that will facilitate this control channel. The server opens a TCP port at a known location and waits for a client to connect. Once the client connects, a TLS connection can be established. The lab recommends using SSL, but this is outdated security-wise. OpenSSL can be used to turn this TCP connection into a secure channel, abstracting lower-level implementation details and using an industry standard public key method.

Before the client and server can form an SSL connection, they need to be able to verify each other's authenticity. In seedVPN, this is done through PEM-encoded certificates generated through OpenSSL's command line. First, a certificate authority (CA) was created with a self-signed x509 certificate. Next, the CA's certificate is used to sign the server's certificate as well as the client's. The client and server also use OpenSSL to generate an RSA private/public key file. Both the client and server have the will request each other's certificate before connecting and maintain the CA's certificate file in order to verify that the peer's certificate is valid. An alternative to this would be to have the server store a username and password for clients and verify this upon connection. This was not chosen because it adds additional security concerns to the program. Further checks could be implemented using the certificate method like adding hostnames to the certificates. For the purposes of this project, all of the certificates for the client, server, and CA are stored in a folder named "ssl".

One issue with this method is that in order for the client and server to load their private/public key files into the SSL context, it first needs to be decrypted. For seedVPN's purposes, the decision was made to hard code these passwords. This is bad practice, but this implementation is beyond the scope of this lab. A better way of doing this would be to require the client and server to store their own copy of the password and read it into seedVPN dynamically to decrypt the keyfile.

SeedVPN provides TLS functionality through two public methods: ssl_init_ctx() and ssl_handsh(). The context initialization function is of notable importance and is shared between the client and the server. This function is responsible for loading in the CA's certificate as a trusted verification certificate using OpenSSL's "SSL_CTX_load_verify_location()". Next, the appropriate private key file and certificate are loaded into the context. This context is initialized by the client and server, and then passed to the ssl_handsh() function which binds the already-established TCP socket to an OpenSSL SSL struct that can be used for communication. An additional argument is supplied to this function that indicates whether the caller is a server or not. If the caller is a server, then it waits using SSL_accept() for a client to connect to it via SSL_connect(). Before these calls are made, OpenSSL's SSL_set_verify() function is used to indicate that both the client and the server should send their certificates for the other to verify. If the call to SSL_connect() or SSL_accept() returns without error, then the handshake is complete and a secure channel is set up.

After the handshake, the SSL struct can be used with SSL_read() and SSL_write() to share information between client and server safely. This is used to share the session key and IV for the encrypted tunnel. For security, the key and IV are generated by the server using getrandom() which internally reads from "/dev/urandom" and ensures non-bloking random data. The server generates these one at a time and sends them to the client which waits for them. After this, both the client and server share the same, randomly generated, key and IV for encrypting traffic over the UDP tunnel.

Routing

SeedVPN adds the automation of routing to the implementation described in the lab. The lab description recommends using the command line both to configure the TUN device by assigning it an IP address and setting it's state to "up". This process is inconvenient, and the timing can interfere with the establishment of a connection between the client and server. SeedVPN uses ioctl() calls to set up the device along with an IP address supplied via a command line argument. This argument specifies the IP address to be used by the device inside of the

VPN and is assigned to the TUN device. The calls to do this configuration are wrapped in a function titled tun_config().

First, a UDP socket is opened as this file descriptor is needed for the ioctl() calls. Next, the device name (ie “tun0”) is added to the request structure. Three ioctl() requests are then made. The first uses the argument “SIOCSIFADDR” to assign the specified IP address to the device. Second, “SIOCSIFNETMASK” is used to set the netmask which is statically defined to be “255.255.255.0”. Finally “SIOCSIFFLAGS” activates the interface. The client and server can both call this to setup their device before any communications are sent.

Next, the client and the server need to know to route packets destined for each other through the TUN device. After the TLS connection is established, the client and server exchange their VPN IP address over the control channel. The API for achieving this is poorly documented. While developing seedVPN, attempts were made to use Linux’s “strace” utility to figure out how the “route” utility achieves this, but to no avail. Instead, seedVPN internally uses the “route” utility to add a route to its peer. Two ways of doing this were considered: system() and execve(). Execve() was chosen because system() internally uses the system shell which opens up additional concerns. Most of the arguments for “route” are held in a static array of character string pointers. The missing arguments are the IP address to add, and the device to assign it to. The device name is supplied via a char pointer argument, but additional computation is required for the IP address. “Route” requires an IP address that is compatible with the subnet mask. In the case of seedVPN addresses, this mask is “255.255.255.0”, so the IP address assigned to the device via “route” needs to end in a 0. Linux’s inet_addr() is used to convert the character string IP address to an in_addr_t or unsigned integer. This integer must then be masked with the subnet mask to clear the least significant bits of the address. An issue that arose was the byte order of the integer returned by inet_addr(), as a result, the netmask had to be defined as 0x00FFFFFF instead of 0xFFFFFF00. With this solved, inet_ntoa() can be used with the masked IP to get a character representation suitable for “route”. After this call is finished, the server and client have now dynamically added eachother’s routing information through the TUN device.

Reconfiguration

The SEED VPN lab suggest implementing the ability to reconfigure the session key and IV from the client end which can be done via the TLS connection between the client and server. The lab also recommends using fork() to create new processes since reading from the SSL connection and standard input are blocking, but this is not needed. David Brini's program already uses the select() system call to avoid this blocking behaviour. The TCP socket and standard input file descriptors can be added to the FD_SET, meaning the main program loop can stay the same. With this, the client and server can wait for a user to type commands on standard input, and route them via the secure TLS channel to the peer.

The first command is closing the connection between the client and server. This can be implemented easily based on the fact that the TCP socket will close when one peer disconnects. If the client or server closes, the other peer sees this via an empty read() call to the TLS connection and can exit accordingly.

The key and IV commands are implemented in a function in "commands.c" named parse_command(). When one peer types receives input on standard input, it writes the information to the TLS connection via a buffer of known length. For security, these read and write calls read a set number of bytes into a character array that is the same size. Both client and server then execute the same parse_command() function. This ensures both peers have the exact same result when calling the function, meaning that if the command is invalid, both fail. The sender of the command also waits for a blank SSL_write() from the receiver so that network latency won't cause the peers to update their key or IV out of sync.

The "key" command followed by a 32-byte character string will update the session key. The "iv" command followed by a 16-byte character string will update the shared IV. Both commands fail if the length of the supplied character string is not correct. The programs then continue as normal.

In addition to the commands specified in the lab, seedVPN implements the ability to reconfigure the key used to sign MACs. The command logic is the same as the key and IV changes. This is useful because if the session key were to be recovered, HMACs could be forged since they use the same key and therefore authenticity would not be ensured.

Multiple Clients

The final step in configuring seedVPN is to allow the server to support multiple connected clients. To support multiple clients the server must be able to route traffic to the appropriate client and encrypt/decrypt using the correct key. To keep each client's connection separate the serve must fork every time it accepts a connection over it's open TCP port. This means that every client should get its own TUN device. This adaptation is easy to implement requiring one change aside from the fork() call. The server has a format string defined as "tun%d" along with an integer representing the number of connected clients. Every time a new client connects, it gets a TUN device named according to its connection number (ie "tun5").

Demonstration

In order to aid with demonstration, one additional function was implemented which prints a specified number of bytes of a given array. This function could be hazardous to production code, since it is used to print data that is shared over the network and may be of arbitrary length and content. To mitigate this, a define statement is used so that the function is only added to the executable if a specific flag (DANGEROUSDEBUG) is supplied to the compiler.

Using this arbitrary print bytes function along with the debug flag while running the program, all key exchange and message passing can be observed. The easiest way to do so is by pinging one device's internal VPN IP address from another peer. Here it is possible to see values before and after encryption, as well as to verify the session key is shared properly after the handshake.

Future

Future implementations should focus on adding more robust security. The use of arrays could be phased out in order to check values on a byte-by-byte basis. Although care was taken to ensure all memory accesses are of a fixed size, this would offer better input sanitization. Additionally, the use of configuration files for peer's private key decryption passwords would fix one of the larger holes in the system. A final addition that would be nice is a command

interface for gateways to dynamically add routing information for their internal networks. Given the existing infrastructure, this could be implemented easily.

Works Cited

Brini, D. (2010, March 26). Retrieved March 2020, from https://backreference.org/2010/03/26/tuntap-interface-tutorial/

Du, W. (n.d.). *Virtual Private Network (VPN) Lab*. Retrieved March 2020, from SEEDLabs: http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/VPN/