CS 536 Fall 2019

Lab 4: Basic Network Security and Symmetric Client/Server Apps [250 pts]

Due: 10/29/2019 (Tue), 11:59 PM

Objective

In this lab, we will investigate basic network security vulnerabilities using the remote command client/server app of Problem 1, lab2, as an example. We will consider cryptographic primitives treated as abstract modules which serve as building blocks of network security protocols on the Internet. We will implement a simplified version for performing network authentication. We will also look at the design of symmetric client/server applications where all communicating parties are homogenous. We will implement a canonical symmetric client/server, an Internet chat app.

Reading

Read chapter 3 from Peterson & Davie (textbook).

Problem 1 [100 pts]

1.1 Buffer overflow vulnerability

The remote command client/server of Problem 1, lab2, has potential security vulnerabilities given the nature of the service provided: ability to execute commands on a shared x86 Linux machine. One of the most obvious ones is buffer overflow where a long string (supposedly representing a command) is transmitted that overflows the server's char buffer. When performing any form of input (e.g., through recvfrom(), read()) it is imperative that the amount of data read never exceed the memory allocated to hold the data. This is especially true when using user space library functions that do not provide an interface through which length checking can be exercised. If this is the case, such library functions should not be used to code network applications. It is recommended not to rely on hardware, operating system, and compiler support to mitigate the buffer overflow problem. Instead, implement your application so that it is self-reliant to the extent feasible. In Problem 1, lab2, the third argument of recvfrom() allows the length of the user buffer (second argument) to be specified. Given the variable length of remote commands transmitted by a client, check your code to determine what happens when a transmitted string exceeds the length specified in the third argument. Do not rely solely on man page specification but test and verify. Enforce a policy that a command cannot exceed 30 bytes. Modify request format and parsing so that a request does not end with NUL. Any command that exceeds 30 bytes is dropped (i.e., not executed). Update your client request reading/parsing code so that it is not vulnerable to buffer overflow.

1.2 Client authentication: cryptographic primitives

Another clear and present concern is client authentication since we want a select subset of authorized users to be given capability to execute remote commands. Authentication is facilitated by basic cryptographic primitives which can also be used for implementing confidentiality (i.e., encryption) and integrity (i.e., message has not been modified by an attacker). The underlying mechanisms are one and the same. Confidentiality has a second,

and the only provably secure, foundation which we will consider separately. Cryptographic primitives used in network protocols rely on a message encoding function E, a decoding function D, and two distinct keys, e and d, called public and private keys, respectively. These systems are referred to as asymmetric or public key encryption/cryptographic systems. Given a message m (a bit string), called plaintext, that Alice wishes to send Bob, confidentiality that protects m from eavesdroppers works as follows:

Confidentiality:
(a) Alice computes encrypted message, s = E(m,e), using m and Bob's public key e. B's public key, as the name indicates, is assumed known to all parties who wish to send secret messages to B.
(b) Bob receives s (e.g., Ethernet frame, UDP or TCP packet), then computes m = D(s,d) which results in the original unencrypted message m.
(c) It is assumed that without knowing B's private key d, computing m from s is difficult.

D can be viewed as an inverse of E, and D is a "one-way function" in the sense that encrypting -- going in one direction -- is computationally easy but decrypting (i.e., inverting E) without knowing B's private key (which we assume he safeguards) is computationally hard. That is, unless P = NP. For authentication, the same primitives are employed but in reverse. That is, Bob wishes to send a message (called certificate) s to Alice whereby Alice, upon receiving s, can determine that Bob is the originator of s.

Authentication:
(a) Bob computes certificate, s = D(m,d), where m is a plaintext message that says "I am Bob, born around 1905, my favorite color is gray, gettimeofday() is ... and I want you to run the remote command ls -l" using his private key d.
(b) Alice, upon receiving s, computes m = E(s,e), which yields the original plaintext message which follows a strict format (e.g., name, birthday, favorite color, time stamp, etc.).
(c) It is assumed that only B can generate a certificate s using his private key d, that when encrypted via E using B's public key e results in a meaningful plaintext message following a specific format.

Hence the two functions E and D commute in the sense that either order of function composition yields the original (plaintext) message since they act as inverses of each other. The popular RSA public key cryptosystem that relies on the assumption that factoring large primes is computationally hard yields such E and D.

1.3 Client authentication: public key infrastructure

A central issue of using a public key cryptosystem, in practice, for authentication and confidentiality is bootstrapping, since Alice knowing Bob's public key is easier said than done. For example, if an impostor C somehow convinces A that a key that C created, say e*, is B's public key, all bets are off. So how do we bootstrap a distributed system where the true public keys of all relevant parties are accessible? Unfortunately, there is no solution to the bootstrapping problem but for brute-force system initialization where designated "trusted" parties, called certificate authority (CA), serve as arbiters of verifying public keys. Operating systems are distributed with hardcoded public keys of CAs so that a user of the operating system can make use of CA services. For example, if a secure communication session over UDP or TCP (i.e., user data sent as UDP or TCP payload is encrypted before transmission) to a server is desired, the server's address -- the symbolic domain name in place of IP address (e.g., www.purdue.edu in place of 128.210.7.200) -- is transmitted to the CA (encrypted with the CA's public key). The CA then responds with the public key of the server as a signed certificate (i.e., step (a) of Authentication) which the client can verify for authenticity by applying E with the CA's public key (step (b) of Authentication). To reduce overhead, servers may obtain pre-signed CA certificates which are communicated to clients directly.

When engaging in lengthy data exchanges (e.g., file server responding to client requests), a public key infrastructure (PKI) is only used to establish mutual authentication and share a secret private key, after which symmetric encryption using the shared private key is used to achieve data confidentiality. This is due to symmetric encryption incurring less overhead. A popular symmetric encryption method is one-time pad which XORs data bits with a random sequence (obtained from private key). One-time pad is the only provably secure

encryption method as long as the random sequence is indeed random and the private key is known to the communicating parties only. In the real world, pseudo-random sequences take the place of random sequences, hence one-time pads are only as secure as the randomness of pseudo-random sequences generated from private keys. As John von Neumann noted: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

1.4 Client authentication: implementation

When building network applications using socket programming, TLS (Transport Layer Security) and its precursor SSL (Secure Sockets Layer) using the OpenSSL API is the default way to implement SSL crytographic protocols. Unlike TCP/UDP/IP socket programming which is narrow in scope and supported by select operating system system calls, SSL is more complex due to its generality stemming from the diversity and richness of context dependent security protocols. SSL programming is the subject of a course in network security and outside the scope of our course. Instead, we will implement a limited version of application layer interaction over UDP sockets to achieve client authentication in the remote command execution server app.

We will assume that our remote command server maintains an access control list (ACL) of IP addresses (dotted decimal format) and associated public keys. A client sends a request in the form of a certificate signed using function D and its private key. The server, upon receiving a request, applies E with the client's public key assuming its IP address is contained in the ACL. If the result matches a certain format, the requested command is executed. Otherwise the request is dropped. From a network programming perspective, the internals of E and D are not relevant since they can be treated as black boxes. In place of emulating RSA (or other cryptographic algorithms), we will define simplified black boxes E and D as follows.

Encoding function E:
The encoding function, char myencode(char x, unsigned int pubkey), takes a char value x and public key pubkey as input and returns an encoded byte value. It does so by performing bit-wise XOR of the 8 bits of x and the first 8-bits of pubkey. The second time myencode() is called the 8 bits of its first argument are XOR'ed with the second 8 bits of pubkey (i.e., second byte). The next time the third byte of pubkey, and the fourth time XOR with the most significant byte of pubkey. Then repeat the process with the first byte of pubkey.

Decoding (i.e., signing) function D:
The decoding function, char mydecode(char x, unsigned int prikey), takes a char value x and private key prikey as input and returns a decoded byte value. It does so by behaving exactly the same way as myencode(). Since XOR is its own inverse, E and D will commute and satisfy our needs.

myencode() and mydecode(), if used with the same key (i.e., pubkey = prikey) implement a circular one-time pad. Since the pads are re-used, they violate the one-time property. Security of this method can be enhanced by increasing key length or using keys to generate long pseudo-random sequences. Since our focus is on networking and not algorithmic aspects of security, the present scheme will suffice for implementing and testing client and server protocol interaction in the remote command server app for achieving authentication. The format of the plaintext before it becomes a certificate by applying mydecode() is 4 bytes of IP address followed by up to 30 bytes of char representing the command (note that by 1.1 the NUL character has been removed). In addition to machine-to-machine authentication, a second layer comprised of user password authentication is common. The latter can be further strengthened by applying two factor authentication as Purdue has recently instituted. Note that myencode() and mydecode() are easily hacked and subject to replay attack (i.e., an attacker captures a certificate packet and re-uses it in the future to gain access to the remote command server). Therefore, as in Problem 1 of lab2, terminate the server after testing is complete.

1.5 Testing

Test and verify that your remote command client/server app works correctly. Implement the ACL as a 2-column file, acl.txt, at the server that is read when the server is started. At the client, remote-command-cli, provide the

private key as the third command-line argument following the port number (and preceding the command to be executed). Check that when invalid requests are submitted, they are appropriately dropped by the server. As noted in 1.1, make sure that your code does not suffer under a buffer overflow problem. If it does, the consequences will be severe. Submit your code, with adequate annotation and Makefile, in v1/.

Problem 2 [150 pts]

The network chat application, talk, popular in the 1980s and 1990s on university campuses and research labs well before the advent of myriad messaging applications, allowed users on UNIX machines connected via an IP network to exchange messages in real-time. Unlike the client/server apps studied in preceding labs where there was a clear distinction between server and client -- there was an asymmetry in their roles -- talk was a prototypical symmetric client/server app, also referred to as peer-to-peer (P2P) app. In talk, there is no designated server to whom clients send service requests and who responds after servicing the request. Instead, every host running talk is both a server and client.

2.1 Establishing a chat session

Implement a UDP-based talk app, call it terve, where the app is invoked with a single command-line argument

% terve port-number

which specifies the port number that terve will use to communicate using UDP. When terve runs, it will bind itself to port-number, print "#ready: ", and wait for stdin input from the user. The user may initiate a chat session by entering the IP address (in dotted decimal notation) and port number of the other party. For example,

#ready: 128.10.25.208 55555

followed by ENTER/RET (i.e., '\n'). terve transmits a control message in the payload of a UDP packet to the destination IP address and port number (128.10.25.208:55555 in the above example) containing 5 bytes: the first byte contains unsigned value 5 specifying "let's talk" (i.e., session initiation). The next four bytes contain an unsigned integer specifying a random number. Before transmitting the message, terve sets a timer for 5 seconds using alarm(). If a response does not arrive before SIGALRM is raised, terve retransmits the message. It gives up after retransmitting twice and prints to stdout a message indicating that establishing a session has been unsuccessful followed by a fresh "ready" prompt. For example,

#failure: 128.10.25.208 55555
#ready:

If terve receives a response from the specified party containing a 5-byte payload where the first byte is the unsigned value 6 specifying "ok let's talk" (i.e., session establishment) and the next four bytes contain the previously transmitted random number, it prints to stdout that a session has been established followed by a prompt to start typing a message. For example,

#success: 128.10.25.208 55555
#your msg:

If the first byte contains a value different from 6, or the 4-byte random number does not match, terve prints the same message to stdout as when no response was received after 10 seconds. If during the session initiation phase a session request from another party is received, the third party's IP address and port number are output to stdout but no response is sent in return. For example, if a request from 128.10.112.202:44444 is received

#session request from: 128.10.112.202 44444

is output and terve continues to wait for a response from 128.10.25.208:55555. When terve is first executed and no input is forthcoming from the user, terve keeps waiting for user input at the "ready" prompt or for a session initiation request. If a session initiation request arrives, terve prints the other party's coordinate followed by the "ready" prompt. For example,

#session request from: 128.10.112.202 44444
#ready:

If the user enters ASCII character 'y' on stdin then it means to accept session initiation request, and terve responds to the sender with a UDP packet containing value 6 in the first byte followed by the 4-byte unsigned number received. If the user enters 'n' then terve responds with the unsigned value 7 in the first byte followed by the 4-byte unsigned number received. The value 7 signifies that session initiation is rejected. If any character other than 'y' or 'n' is entered, terve prints a fresh "ready" prompt and waits for valid (i.e., 'y' or 'n') user input.

2.2 Interaction during established chat session

During an established chat session after the 2-way handshake, three events must be concurrently handled. First, the user typing a message on stdin that is sent via UDP to the other party. Second, a message arriving from the other party that is output to stdout. Third, the user terminating the chat session. The third case is described in 2.3 below. terve, after establishing a chat session, blocks on read() waiting to read up to 50 bytes of input (for simplicity we impose a limit on maximum message size) terminated by '\n' (i.e., ENTER/RETURN key). If during typing (which for humans can take arbitrarily long) no message arrives from the other party, terve sends the message (up to 50 byte long not including '\n') via UDP. The first byte of UDP's payload is the unsigned value 8 specifying that the following bytes are chat data bytes. The next 4 bytes are the session's random number, followed by the actual chat message. After transmitting the message, terve prints "#your msg: " on a new line and goes back to blocking on user input.

In the second case, while the user is in the midst of typing a message or waiting for the other party to send a message, a message from the other party arrives. This raises the SIGIO (equivalently SIGPOLL) signal which terve handles asynchronously using a SIGIO handler, int terve_msg_receive(int), that is registered using signal() when terve is executed. terve_msg_receive() reads up to 50 bytes using recvfrom(), and assuming the message is from the session's other party with control byte value 8 and correct 4-byte random number, the data bytes of the message are output on stdout on a separate line followed by a fresh "your msg" prompt. For example,

#your msg: How about lunch at the new sushi place at noo
#received msg: lunch @mcdonald @12:15
#your msg: n? Ok.

Since UDP packet arrivals signaled via SIGIO are asynchronous, interleaving of output to stdout may occur as indicated in the example. In the original talk app, the curses library is used to split the terminal in two (upper region and lower region) so that messages being typed and received are visually separated. Since our focus is on networking, we will sacrifice making things look nice (super important in production apps) for the sake of simplicity. If the received message is a new session initiation request, a session request message is output to stdout but otherwise ignored (no response sent and no 'y' or 'n' input expected from the user). For example,

#your msg: How about lunch at the new sushi place at noo
#session request from: 128.10.112.203 33333
#your msg: n?

The control byte of the message received from the other party of the established chat session may contain unsigned value 9 which specifies "let's terminate the chat session" which is discussed below.

2.3 Synchronous and asynchronous chat session termination

For one reason or another, the user may want to terminate an established chat session and quit the application which corresponds to synchronous termination. During a chat session, synchronous termination is accomplished via the user typing CTRL-\ (i.e., control key and backslash) which raises the SIGQUIT signal. terve registers a SIGQUIT signal handler, int terve_quit(int), that transmits a message to the other party containing the first byte of value 9 followed by the 4-byte random number (no other UDP payload). After transmitting the "let's terminate the chat session" control message, the SIGQUIT handler calls exit(0) to terminate the application. If terve has not yet established a chat session, SIGQUIT causes the app process to be terminated by calling exit(0).

Asynchronous chat session termination occurs if the SIGIO handler receives a message from the other party of the established chat session with control byte containing unsigned value 9. terve_msg_receive() outputs "#session termination received" and terminates terve by calling exit(0). In this version of terve, chat session termination also leads to app process termination.

2.4 Testing

Test your implementation of terve by verifying its correct behavior under (a) pairwise interaction and termination where terve is run on two different lab machines and one contacts the other, they establish connection, interact, and terminate through synchronous/asynchronous termination, (b) a third party tries to establish connection while pairwise chat session has already been established or is on-going, (c) session initiation race condition, (d) irregular session termination. In the case of (c), two hosts may concurrently initiate a chat session which is not specifically addressed in 2.1. Describe your solution to this issue in Lab4Answers.pdf. Case (d) refers to terve running on a host having established a chat session, not responding to the other party for whatever reasons. For example, the app process was accidentally killed, the host has gone down, the user is too tired and fell asleep. Discuss your approach to handling the irregular termination issue in the context of the terve chat application. Place your code along with Makefile in v2/.

Bonus Problem [20 pts]

There are two options for extra 20 bonus points (but not both). In option A, change the synchronous/asynchronous termination behavior of terve so that SIGQUIT (synchronous) or SIGPOLL raised by a message with control byte 9 does not terminate the process. Instead, the established session is terminated as before, but terve returns to printing the "ready" prompt and waiting for new chat session establishment. At the "ready" prompt, the user can now type 'Q' which will terminate the app by calling exit(0). Alternatively, the user may continue to engage in new chat sessions. Test that your revised app works correctly and place your code in v3/.

Option B, only recommended for folks already familiar with GUI programming using C, addresses the interleaving problem of the shared stdout file descriptor which can produce messy output exceeding that of the examples illustrated in 2.2. One method is to use the curses library and put terminal input (i.e., reading from keyboard) in raw mode. By default, Linux puts interaction with keyboard input in cooked mode which buffers character input until '\n' is emitted. In the interleaving example shown in 2.2, an incremental approach that reduces confusion stemming from interleaved output is to read keyboard character input as soon as each character is entered without buffering. Hence typing the first character 'H' of "How" in 2.2 makes system call read() return immediately (without buffer flushing through '\n') so that 'H' can be stored in terve's user space buffer. The same goes for subsequent characters 'o', 'w', and so forth. After typing 'n' (intending to type "noon?"), the SIGIO handler intrudes and its output to stdout causes potentially messy interleaving. In the raw mode version, since the preceding characters typed by the user, "How about lunch at the new sushi place at noo", have been stored in terve's user space buffer, terve's SIGIO handler -- after printing the received message to stdout -- can output to stdout the previously typed message "#your msg: How about lunch at the new sushi place at noo" with the cursor blinking after 'o' in "noo". This makes it easier for the user to continue typing before being interrupted by the received message.

One caveat about using curses is that system programming in raw mode is error prone and not recommended without prior exposure. For those with prior experience, it can be a way to program a user friendly version of terve with minimal graphics support. A popular GUI based C programming option is GTK. There are a number of other options including Tcl/Tk (via interpreted code embedded in C) or using the X Library (with or without common widgets). If you have never done GUI programming in C, my recommendation is to consider option A. If you are comfortable with GUI programming, then option B may be a more interesting way to enhance the UI component of the terve app. If choosing option B, please include a README file that describes your UI. Submit your code in v3/.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab4/ where you deposited the submissions and type the command

turnin -v -c cs536 -p lab4 lab4

This lab is an individual effort. Please note the assignment submission policy specified on the course home page.

Back to the CS 536 web page