CS 5750 Programming Assignment #2: `getd`

Due Date: *Monday, March 18, 2019 @ 8am*

In this project, you will implementing the functionality of `get` in a daemon. Rather than using an SUID binary, `getd` will run as a daemon that listens on a port for a request to get a file. However, for this assignment, you may ignore the access control lists and use only one user.

Requirements

There are two subjects: `get` and `getd`. `getd` is a subject that will read the file. `get` is a subject that wishes to access `getd`'s files. Your task is to write `getd`. Here is a scenario illustrating how this system works.

1. The daemon `getd` listens for a connection from a program supplied by the instructor called `get`. The purpose of `getd` is to read a file and send the contents of that file to `get` via a socket. Your `getd` should use `nanomsg` on a one-to-one (`NN_PAIR`) socket for interprocess communication.
2. The program `get` is run just like in Program 1. However, it uses the protocol given below to communicate with `getd` to receive the file and write its contents to a new file.

The binary `get` communicates with `getd` through sockets . The daemon `getd` only communicates with `get` and sends the contents of a file via the socket back to `get`.

Protocol.

Attempt to get a file by executing the command

```
get <source> <destination>
```

`get` requests a connection to `getd` on the port `ipc:///tmp/getd.ipc`. Once the connection is established, `get` sends a request for a file and waits for a response. If this request is successful `getd` sends the contents of the file to `get` and then ends the communication. You may assume that there will only be one request to `getd` at a time. Below is a list of message types involved in the communication protocol between `get` and `getd`.

Type 0: Request a session.

Type 1: Send a session id.

Type 2: Indicate error encountered.

Type 3: Request the contents of a file.

Type 4: The contents of a file.

Type 5: Terminate a session.

Type 6: Acknowledge receipt.

Protocol Header Format. The messages in this protocol are variable length depending upon the type. Each message begins with a header that contains a 1-byte message type identifier followed by a 4-byte integer that gives the length of the rest of the message. The C definition for a header is

```
typedef struct _header {
   unsigned char messageType;
   unsigned int messageLength;
} Header;
```

Type 0 Message Format. A **Type 0** message is sent from `get` to `getd` in order to establish a session. The message contains a header, and 4-byte integer that gives the length of a user id name and a 33-byte field containing the user id name (max 32 characters) of the real user id of `get` (with one byte for the an end of string character). The C definition for a **Type 0** message is

```
typedef struct _type0 {
   Header header;
   unsigned int dnLength;
   char distinguishedName[33];
} MessageType0;
```

Type 1 Message Format. A **Type 1** message is sent from `getd` to `get` in order to establish a session. The message contains a header, a 4-byte integer containing the session id length and a 129-byte field containing a 128-character unique session id (with one byte for the an end of string character) that identifies the session and must be used by `get` to request a file. The C definition for a **Type 1** message is

```
typedef struct _type1 {
   Header header;
   unsigned int sidLength;
   char sessionId[129];
} MessageType1;
```

Type 2 Message Format. A **Type 2** message is sent to indicate an error. The message contains a header, a 4-byte integer containing the length of the error message and a 257-byte field for a maximum 256-character error message. The C definition for a **Type 2** message is

```
typedef struct _type2 {
   Header header;
   unsigned int msgLength;
   char errorMessage[257];
} MessageType2;
```

Type 3 Message Format. A **Type 3** message is sent from `get` to `getd` in order to request a file. The message contains a header, a 4-byte integer indicating the session id length, a 4-byte integer indicating the length of the file path,a 129-byte field containing a 128-character unique session id, and 4097-byte field containing a max 4096-character full path name of the file requested. The C definition for a **Type 3** message is

```
typedef struct _type3 {
   Header header;
   unsigned int sidLength;
   unsigned int pathLength;
   char sessionId[129];
   char pathName[4097];
} MessageType3;
```

Type 4 Message Format. A **Type 4** message is sent from `getd` to `get` to send the contents of a file. The message contains a header, a 4-byte integer indicating the session id length, a 4-byte integer indicating the length of the file contents,a 129-byte field containing a 128-character unique session id, and 4096-byte field containing a max 4096-character contents of the file requested (There is no end of string character since this field is a set of bytes, not characters). The C definition for a **Type 4** message is

```
typedef struct _type4 {
   Header header;
   unsigned int sidLength;
   unsigned int contentLength;
   char sessionId[129];
   char contentBuffer[4096];
} MessageType4;
```

Type 5 Message Format. A **Type 5** message is sent from `getd` to `get` to terminate a session. The message contains a header, a 4-byte integer containing the session id length and a 129-byte field containing a 128-character unique session id. The C definition for a **Type 5** message is

```
typedef struct _type5 {
   Header header;
   unsigned int sidLength;
   char sessionId[129];
} MessageType5;
```

Type 6 Message Format. A **Type 6** message is sent from `get` to `getd` to acknowledge the receipt of a **Type 4** message. The message contains a header, a 4-byte integer containing the session id length and a 129-byte field containing a 128-character unique session id. The C definition for a **Type 6** message is

```
typedef struct _type6 {
   Header header;
   unsigned int sidLength;
   char sessionId[129];
} MessageType6;
```

Function of `get`.

`get` will be provided by the instructor and has the same command-line syntax as in Program 1:

```
   get <source> <destination>
```

This process uses `nanomsg` on a one-to-one (`NN_PAIR`) port named `ipc:///tmp/getd.ipc` for inter-process communication. This process will behave as follows:

1. Send a **Type 0** message to `getd` to request a connection. Wait for a reply from `getd`.
2. Expect a **Type 1** message and save the session id.
3. Send a **Type 3** message to `getd` to request the contents of `<source>`. Wait for a reply from `getd`.
4. If a `Type 5` message is received, send a `Type 6` message and exit.
5. Otherwise, expect a `Type 4` message. Send a `Type 6` message. Write the contents to `<destination>`. Wait for a new message. Goto 4.

If at any point a `Type 2` message is received, the error message is printed and `get` exits. If an unexpected message type is received, this is reported and `get` exits.

Function of `getd`.

`getd` is a daemon that must use `nanomsg` to listen on a one-to-one (`NN_PAIR`) port. The URL for the port is `ipc:///tmp/getd.ipc`. `getd` must have the following behavior.

1. Expect a **Type 0 Message**. Store the user id.
2. Generate a session id. Send a **Type 1** message to `get` containing the session id.
3. Expect a **Type 3** message. Store the file path.
4. Read up to 4096 bytes from the file and send it in a **Type 4** message to `get`.
5. Expect **Type 6** message.
6. If the file has not been sent completely, goto 4.
7. Send a **Type 5** message.
8. Expect a **Type 6** message.

9. Goto 1.

If at any point a `Type 2` message is received, the error message is printed and `getd` jumps to Step 1. If an unexpected message type is received, this is reported, a `Type 2` message is sent to `get` and `getd` jumps to Step 1.

Benign `get`. A version of `get` that is benign (follows the protocol faithfully) is available for download from eLearning in the zipped tar file `Program2.tgz`. This `tar` file contains a working version of `get` in `get/get` and the header file for the message types in `util/message.h` and `util/general.h`. Use these files in your code so that your message structures match those of `get`. `get` will send well-formed messages in the proper order. You will not be expected to catch the errors from Program 1. Only valid file requests will be sent.

Evil `get`. Your code will be tested against a second version of `get`. The second version, `evil_get`, will not follow the protocol. In fact, it will purposely send messages in the wrong order and with many invalid formats. You must defend against `evil_get` without ever having seen it or used it.

Miscellaneous. You need not worry about file locking for this assignment. In addition, you do not need to secure the messages in transit via encryption. That will be the next assignment. This project must be coded in C and will be tested under Ubuntu 18.04 LTS.

Collaboration Rules

This project may be performed in pairs or by a single person. Of course, there are no restrictions on interactions between members of the same pair. However, each pair must work independently. A pair may neither show any other its code nor look at the code of another pair. (This policy extends to any external resource, including code found on the web or individuals who are not enrolled in the course.) Individual work will be allowed, but no allowance will be made in the due date or other submission requirements if this option is freely chosen.

Linux

Use Ubuntu 18.04 LTS for this assignment. You will need to install `cmake` and `nanomsg`. To install `cmake`, use the command

```
sudo apt install cmake
```

Next, you'll need to dowload and install `nanomsg 1.1.5` from `https://github.com/nanomsg/nanomsg/archive/1.1.5.tar.gz`. Untar the file and follow the instructions in `nanomsg-1.1.5/README.md` to install `nanomsg`.

Submissions

You must prepare a `makefile` and all necessary source files so that I can simply do a `make` and build `getd`. Each pair (or individual) must send me their names by Friday, February 8, 2019 at 5:00pm. With your submission, include a README file that gives the names of each person who worked on the submission and an outline of each person's contribution to the completed program. Your code should be well commented. In addition to normal documentation, include comments in your code at points related to the security requirements. Also, in the README file, give an overview of your implementation and identify and defend any security-related decisions you had to make during the implementation. **You will be graded on how well you follow the secure coding guidelines discussed in class.**