

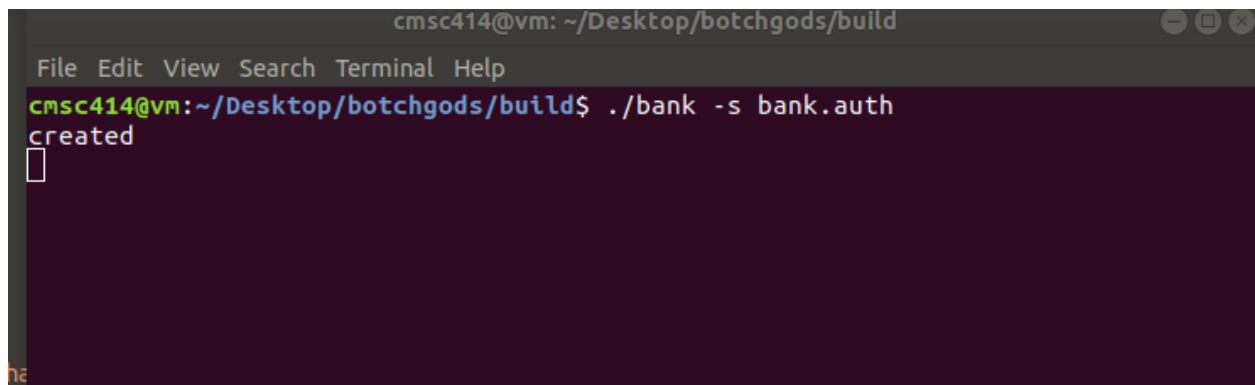
Design Document

Team BotchGods

Dylan Bourgerie, Brent Freedman, Branden Tsang

- Describe your overall protocol in sufficient detail for a reader to understand the security mechanisms you put into place, without having to look at the source code. This must explain your source code and why you structured the modules in a particular way.

In this project, we implemented 2 programs, atm and bank. The atm acts as the client and the bank as the server. Therefore, while the bank program is running, many requests from atm programs can be made and the bank must keep track of its current state (number of accounts, balance of each count). The first challenge with a client-server interaction like this is allowing the 2 programs to communicate. In this simulation of a bank and atms, the bank listens on a certain port (provided as a command line argument) and the instances of an atm program send requests to the bank through a port that is also provided in the command line. Therefore, it is important for the bank and atm to both know that they are actually communicating with who they think they are. To accomplish this, the bank creates an auth file when being created. Then, when an atm attempts to connect to a bank, it first ensures that its auth file is the same as the auth file that the bank created (assuming that the file was sent securely between the two). After verifying this, the atm can attempt to perform its desired operation.

A screenshot of a terminal window with a dark background. The title bar at the top reads 'cmssc414@vm: ~/Desktop/botchgods/build'. Below the title bar is a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows a command prompt 'cmssc414@vm:~/Desktop/botchgods/build\$' followed by the command './bank -s bank.auth'. The output of the command is 'created' on the next line, with a cursor on the line following it.

```
cmssc414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help
cmssc414@vm:~/Desktop/botchgods/build$ ./bank -s bank.auth
created
█
```

(when the bank is initially opened)

```
cm414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help
cm414@vm:~/Desktop/botchgods/build$ ./bank -s bank.auth
created
error on accept call: Resource temporarily unavailable
cm414@vm:~/Desktop/botchgods/build$
```

(the bank should time out after 100 seconds)

The atm has 4 possible modes: n (new account), d (deposit), w (withdraw), and g (get balance), which represent the 4 operations it can perform. The mode of operation is given as a command line argument to the program. Aside from the mode, there are other command line arguments possible (-a <account>, -s <auth-file>, -i <ip-address>, -p <port>, and -c <card-file>). To handle all of the possible command line options and associated values, we use the getopt function included in the C header file unistd.h. This function allows us to provide the command line arguments in any order and also supports POSIX compliance (e.g “./atm -ga dylan” is a valid way to run the atm to get the balance of the user “dylan”). This function also allows us to detect any extraneous arguments that are not valid, and exit with an appropriate error. We also use getopt to handle the same issues with the bank’s command line arguments (-p <port> and -s <auth_file>). The only required command line arguments in either program is the account of the atm and the mode of operation of the atm (any one of the 4). The rest of them have default values that are used if another value is not provided on the command line (e.g default port of 3000 for the atm and bank). We take care of the required arguments by remembering which command line options we were provided. If we are missing the account name, are not provided a mode, or are provided more than one mode, we can then exit the program with the proper status.

The last step to verifying that the user input (provided at the command line) is valid is ensuring that the various arguments are of the right format for the particular option. For example, file names can only include hyphens, dots, digits, and lowercase alphabetical characters, must be between 1 and 127 characters long, and cannot be the special file names “.” and “..”. To handle these required formats, we use regular expressions. If any of the provided command line arguments are not of the proper format, we exit the program with the proper exit value. Not only do these regular expressions ensure that the arguments are the right format, but they also prevent code injection. In particular, the bank and atm both output the result of the operations in a JSON format (which includes the account name and possibly the deposit/withdrawal

amount provided by the user). Therefore, if the user provides an account name and/or amount with a “”, “.”, or “,”, they could trick the bank into thinking they deposited more than they actually did because their input would be treated as JSON code. The regular expressions prevent this since only valid characters are allowed (filters out any potential malicious characters).

After knowing that it has all of the necessary arguments and they are all of valid format, the atm has to communicate its request to the bank. To do so, we generate a message with all of the necessary information separated by a comma (since we know none of the user input can include a comma). For example, to create a new account, the atm generates the message “n,<account name>,<starting balance>”. Since the messages generated by the atm include user input, we make sure to dynamically allocate the buffers for them, ensuring the buffer has enough space. This prevents a potential buffer overflow attack where the user may input very large values at the command line. After generating the message, the atm then sends it to the bank using atm_send.

cmssc414@vm: ~/Desktop/botchgods/build	cmssc414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help	File Edit View Search Terminal Help
cmssc414@vm:~/Desktop/botchgods/build\$./bank -s bank.auth	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -n 23.20
created	{"account":"brent","initial_balance":23.20}
{"account":"brent","initial_balance":23.20}	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -g
{"account":"brent","balance":23.20}	{"account":"brent","balance":23.20}
{"account":"brent","withdraw":10.00}	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -w 10.00
{"account":"brent","balance":13.20}	{"account":"brent","withdraw":10.00}
{"account":"brent","deposit":20.00}	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -g
{"account":"brent","balance":33.20}	{"account":"brent","balance":13.20}
{"account":"dylan","initial_balance":100.00}	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -d 20.00
{"account":"dylan","deposit":300.00}	{"account":"brent","deposit":20.00}
{"account":"dylan","balance":400.00}	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -g
{"account":"brent","balance":33.20}	{"account":"brent","balance":33.20}
	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a dylan -n 100.00
	{"account":"dylan","initial_balance":100.00}
	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a dylan -d 300.00
	{"account":"dylan","deposit":300.00}
	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a dylan -g
	{"account":"dylan","balance":400.00}
	cmssc414@vm:~/Desktop/botchgods/build\$./atm -a brent -g
	{"account":"brent","balance":33.20}
	cmssc414@vm:~/Desktop/botchgods/build\$

(bank commands and outputs for correct inputs)

```
cm414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help
cm414@vm:~/Desktop/botchgods/build$ ./bank -s bank.auth created
error on accept call: Resource temporarily unavailable
cm414@vm:~/Desktop/botchgods/build$

cm414@vm:~/Desktop/botchgods/build$ ./atm -a Dylan321 -d 300.00
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$ ./atm -a dylan -n 4294967296.00
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$ ./atm -a dylan -q 12.12
./atm: invalid option -- 'q'
cm414@vm:~/Desktop/botchgods/build$ ./atm -a dylan -n 50
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$ ./atm -a dylan -n 50.00 -p 65536
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$
```

(invalid inputs)

(error 1: capital letter in an account name) exit 255

(error 2: create an account with an amount greater than maximum allowed) exit 255

(error 3: -q is not a command) exit 255

(error 4: must include numbers with 2 digits after the decimal) exit 255

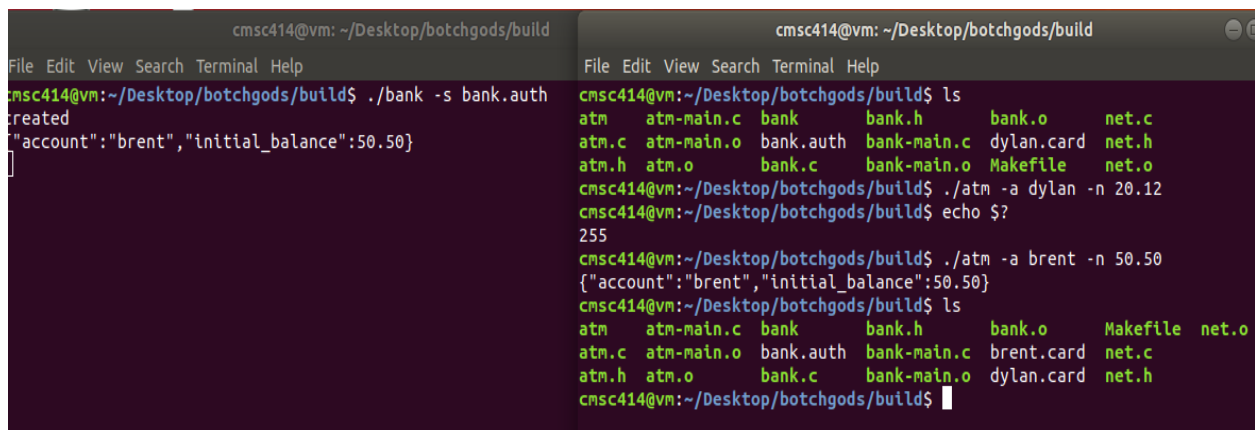
(error 5: port is not within the allowed limit 1024-65535) exit 255

(bank times out after 100 seconds)

After receiving the message sent from the atm (with bank_recv), the bank processes the request as necessary. The bank looks at the first character of the message to determine what the mode of operation of the atm was. For example, when the bank sees “n” as the first character of the message, it knows that the atm wants to create a new account. The bank then extracts the other necessary information from the message (account name and initial balance in the case of “n” mode), using the commas to know where the beginning and end of each value is. The bank stores user accounts in a linked list data structure. Therefore, when attempting to make an account, the bank first searches the linked list to make sure there is no other account with that name. Since the atm must generate a card file (essentially a “pin code” for the account) when an account is created, the bank generates a random string of length 32 to store in the

user's account locally for verification of later requests and also sends it back to the atm upon account creation. The bank reports to the atm whether the account creation (or other operation) succeeded or failed so that the atm can either exit with an error or print the results of the operation in JSON format. For account creation, upon getting a success message (which includes the random 32 character string), the atm will write the card file to disk (with the name given on the command line or <account>.card as default), writing the random string as its contents.

This card file is used as verification for all other modes of operation (deposit, withdraw, and get balance) of the atm. When running in any of these modes of operation, the atm will read the card file with the name provided on the command line (or the default of <account>.card). Then, it will send the "pin" contained in the card file along with all of the other necessary information to the bank. After the bank receives the message, it will then look for the user's account in the linked list (using the account name). If the account is not found, then an error is reported back to atm. If it is found, then the bank verifies whether the pin it is storing in the user's account locally in the linked list is the same pin that was sent in the message from the atm. If these match, then the bank has successfully verified that the card file is the same file that was generated by atm when the account was created (the odds of being able to guess an exact length 32 string of random lowercase/uppercase letters and digits 0-9 is effectively 0). Therefore, the bank can complete the requested operation (either deposit the given amount, withdraw the given amount, or get the current balance) and return the result to the atm.



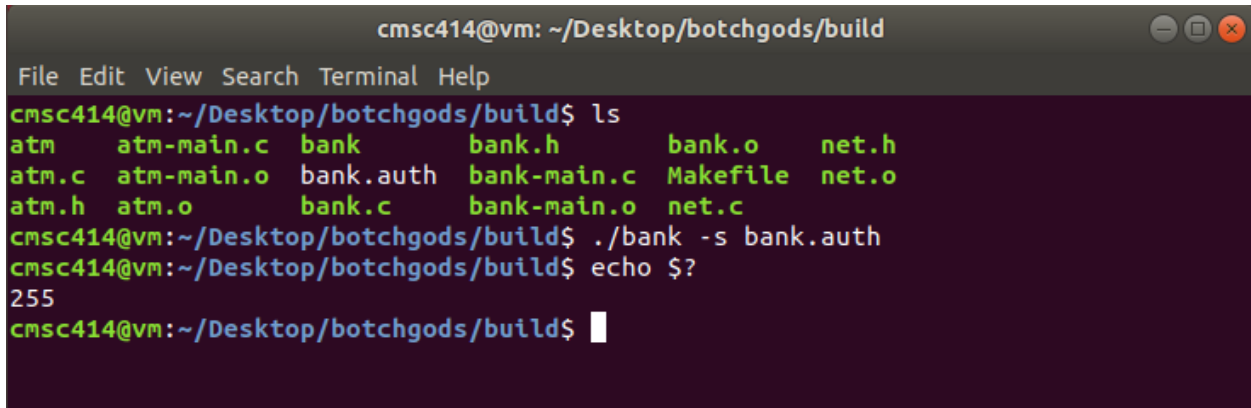
```
cm414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help
cm414@vm:~/Desktop/botchgods/build$ ./bank -s bank.auth
created
{"account":"brent","initial_balance":50.50}

cm414@vm:~/Desktop/botchgods/build$ ls
atm  atm-main.c  bank  bank.h  bank.o  net.c
atm.c  atm-main.o  bank.auth  bank-main.c  dylan.card  net.h
atm.h  atm.o  bank.c  bank-main.o  Makefile  net.o
cm414@vm:~/Desktop/botchgods/build$ ./atm -a dylan -n 20.12
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$ ./atm -a brent -n 50.50
{"account":"brent","initial_balance":50.50}
cm414@vm:~/Desktop/botchgods/build$ ls
atm  atm-main.c  bank  bank.h  bank.o  Makefile  net.o
atm.c  atm-main.o  bank.auth  bank-main.c  brent.card  net.c
atm.h  atm.o  bank.c  bank-main.o  dylan.card  net.h
cm414@vm:~/Desktop/botchgods/build$
```

(if a card already exists for an account, an error is thrown with exit code 255)

(can still create more users as long as the bank is still running)

Besides the card file, the bank also uses an auth file as verification to authenticate the bank. When initializing the bank, the bank should take a <auth_name>.auth file or a default bank.auth file as it's authentication file. After verifying that this file does not exist already, the bank will run. If the file already exists, the bank should exit with return code 255.



```
cm414@vm: ~/Desktop/botchgods/build
File Edit View Search Terminal Help
cm414@vm:~/Desktop/botchgods/build$ ls
atm      atm-main.c  bank      bank.h      bank.o      net.h
atm.c    atm-main.o  bank.auth bank-main.c Makefile    net.o
atm.h    atm.o       bank.c    bank-main.o net.c
cm414@vm:~/Desktop/botchgods/build$ ./bank -s bank.auth
cm414@vm:~/Desktop/botchgods/build$ echo $?
255
cm414@vm:~/Desktop/botchgods/build$
```

(bank.auth already exists)

In all instances of an error (either internally or reported back from the bank), the atm exits with a status of 255. After every successful operation, both the bank and atm print the results of the operation to stdout in JSON format.

- List, one by one, the specific attacks that you have considered and describe how your protocol counters these threats. This is critical for how we will be grading this part of the project (see Grading below).

Buffer overflow attacks: Because C is one of the few languages that allow direct access to memory, buffer overflow attacks will always be inevitably a problem. One way we decided to counter it is by using “strncpy” instead of “strcpy” when we needed to copy any strings in order to append a null terminating byte to that string. Another fix we implemented was dynamically allocating memory for any buffers that take in user input. With the buffers being allocated based on input, the user can not input more than the buffer can hold.

Integer Overflow Attack: While looking over the description, we encountered a problem with the maximum deposits and maximum balance. Because the maximum

deposit was so large and any person can deposit as much money as he wants, he could technically accrue an infinite balance which would be impossible to contain. Therefore, we decided to store our balance in a long double which would make it almost impossible for someone to try to store a balance larger than the maximum amount. Additionally, due to storing the balance in a variable with a maximum limit we have to be wary of an integer overflow where the balance could potentially wrap and become less when attempting to add a value to it. We added a check to make sure that after adding a value, the balance increases and does not become smaller.

Command line Attacks: Using regular expressions, we sanitize the input from the users which prevents malicious attacks. Regexes help prevent attacks that malicious users use when trying to make command line inputs to be interpreted as code. Malicious users who would want to exploit JSON by inserting colons or quotes in order to make their balance an absurd amount will not be able to because our regex sanitizes the input. Users who may wish to use the command line to potentially steal information and data from the atm or bank will also be prevented due to the constraint imposed on them.

The bank generates a random 32 byte key on a card file when a new user account is created. Then whenever the ATM makes a request the bank sends this key with the information.

- You can also mention threats that you chose to ignore because they were unimportant, as well as threats that were important but you were unable to address.

Man in the middle attacks: Due to the nature of the ATM and the Bank communicating with one another over a network, a perpetrator can intercept the data communication and alter its contents undercover. We knew this was a serious problem as it is a threat that can happen undetected. One method to reduce a man in the middle attack is to use public key pair authentication such as with RSA. Authenticated encryption ensures that the message was not modified when sent from the atm to the receiver. Using public key authenticated encryption will give a better sense that the communication is coming from the right sources as they are authenticating with the key pair. Another simple way to

make man in the middle attacks harder is by encrypting the wireless access points. With this encryption it should be harder for attackers to join the network as they have to get through the encryption first.

- List the specific contributions of each team member.

Branden Tsang: Assisted Dylan in coding atm-main and bank-main. Worked on implementing security mechanisms and thought of ways attackers would be able to hack our code. Worked on setting up the Gitlab and getting the code to build on Bibifi.

Brent Freedman: Worked on the design document and thought of ways that attackers would be able to hack our code.

Dylan Bourgerie: Primary coder and worked on atm-main and bank-main. Did a majority of the parsing for commands as well as processing each individual account and making sure the atm and bank work. Analyzed the code and described our overall protocol and security mechanisms in detail.

Overall, all the group members met up in person and we worked together to evenly contribute to the project. We all contributed to the planning, the code, and the text document together. To keep everyone contributing, while one member was working on the coding portion of the project, the other two members were either working on the text document or assisting the coder with ideas.

Test input cases used in oracle:

```
[ {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-n", "10.30"],"base64":false},  
 {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-d", "5.00"]},  
 {"input":["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-g"]},
```

FAILED? WHAT IS THIS INPUT {"input":["LXA=", "JVBPULQI", "LWK=", "JULQJQ==",
"LWE=", "dGVk", "LWc="],"base64":true}}

[**FAILED? WORKS LOCALLY** {"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "chris",
"-d", "4294967295.99"]},

{"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-d", "0.00"]},

{"input": ["-p", "%PORT%", "-i", "%IP%", "-a",
"aaa
aaa", "-n", "10.01"]},

{"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "chrisasdf", "-d", "0.00"]},

FAILED? WORKS LOCALLY {"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "ted", "-g"]},

{"input": ["-a", "ted", "-n", "10.00", "-p", "%PORT%", "-i", "%IP%"]},

{"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "joe", "-g", "-g"]},

{"input": ["-p", "%PORT%", "-i", "%IP%", "-a", "alice", "-c", "ted.card", "-d", "21.0"]}]