Vulnerability Report

Cryptography and Network Security I
CSCI 4971 – Final Project

Jared Candelaria **Shawn Denbow** **Jeremy Pope**

Vulnerability 1 – Attacking Crypto

The crypto initially appeared to be impossible to break given the limitations of the project--while it used a symmetric key (meaning one compromised ATM would bring down the entire scheme), however we were not allowed to extract keys from the binaries. There is, however, a slight problem: the nonce is always chosen by the bank. This means that while replaying the ATM's messages to a bank isn't possible (the nonces wouldn't match the ones chosen by the bank), one can replay the bank's messages to the ATM. Since the bank is expected the choose the nonce, and the ATMs don't keep track of nonces already used, the ATM has no idea that the "bank" is really just a set of pre-recorded messages. We implemented a proof of concept that allows a withdrawal of $100 by alice without the bank ever receiving any connection. This can be repeated to withdraw any amount of money. See *data.h* for an example capture packet of withdrawing 100 as alice and *proxypwn.cpp* to actually replay the attack.

Vulberability 2 – Bad Code

The code also contains a buffer overflow due to the use of a signed integer for the incoming packet length (present in the example code), however it appears to not be exploitable. The only useful thing it overflows into is the scratch area for the random number generator, which we believe is no longer in use. Also, the code was compiled with the stack protector, which means that any overflows up to the frame pointer/return address result in the program being killed before the overflow can cause any damage. Since the program is killed, though, it can still be used as a denial of service attack.

How to overflow the bank:
* Send length = 0xFFFFFFFF
* Send packet with true length of about 2048

(python -c 'print “\xff\xff\xff\xff” + “A”*2048') | nc localhost 4444

Vulnerability 3 – Theoretical Race Condition

In our search for vulnerabilities we noted that the banking program was multithread and did not use any mutexes. This piqued our interest as this means that, theoretically, two threads can modify the same chunk of memory. A successful exploit of these so-called race conditions would allow us to withdraw more money than would be in an account, disconnect authenticated users, and other nasty things. Sadly for us, these attacks proved to be rather difficult to pull off. In the following paragraphs, however, we shall detail the vulnerabilities that were discovered though not necessarily exploited.

Infinite Money

--

Since the list of users is a global variable, any thread can access, and modify, its contents. If a malicious user were to initiate two, or more, connections in a precisely timed manner he would be able to login multiple times concurrently. This race-condition coupled with yet another race-condition exploit of withdrawing money would allow for, in the worst case, a malicious user to withdraw more money than his account would have. How it works is that there is a small amount of time between the checking of a condition and the action such a condition requires. If one were to perform multiple checks simultaneously, their required actions would also be performed simultaneously, and the results of those actions would only be reflected after each check was performed. Due to the unpredictable nature of multiple threads modifying the same block of memory, this could corrupt memory in an arbitrary way. For example, instead of one thread winning out and subtracting its withdrawn amount from the original, BOTH threads could modify the memory simultaneously resulting in a randomly number. This may create an account with the maximum possible representable money, or with no money. One fix would be to employ mutexes to ensure data is only ever read/written to by one thread at a time.

Malicious Session Hangups

--

Another global variable was the count of active threads. Modifying this variable would not result in stealing money but it would allow an attacker to disconnect established sessions without crashing the bank. If a malicious user were to create MAX_THREADS concurrent sessions in a precisely timed way, a legitimate user may be disconnected. The vulnerability appears on line 308 of bank.cpp (in our copy of the source). Basically, it checks if there are too many threads and breaks out of the infinite loop if there are. A fix would be to move the check outside the loop to the top of the function so as not to execute it every iteration of the loop. Again, since the effects of multiple memory writes is absolutely undefined, we could end up setting it to 0, INT_MAX, or any other value.

This list is by no means exhaustive. These were just two that seemed of any interest.