Chris Cohen
CS354 Spring 2020
Lab 4

3 - Blocking send() IPC

3.1 → bsend.c, bsend.h

a) In **include.queue.h**, I changed NQENT from (NPROC + 4 + NSEM + NSEM) to **(NPROC + 4 + NSEM + NSEM + 2*NPROC).** I added the extra 2*NPROC at the end because we needed another queue for each process to have a receiving buffer for all queued up messages. Each queue will need a head and a tail, so each queue needs to allocated spaces. NPROC is the max number of processes total, so we need another 2*NPROC entries to accommodate all possible new processes.

3.2 → receive.c, receive.h

3.3 → testing

a) To gauge correctness, I created one receiver process and 5 sender processes. The receiver process calls receive(), followed by a 200ms delay, in an infinite loop. Each sender process sends a message to the receiver process. In bsend(), receive(), and the sending processes themselves, I printed when a message was sent, queued, dequeued, and received. Initially, I expect the receiver to attempt to receive(), find nothing, then delay for 200ms. In this 200ms, all 5 senders will queue up. The first sender will send normally, using send(). Senders 2-5 will queue up and block until their request is received. The receiver will incrementally receive one process at a time, 1-5, in numerical order (since they were queued that way).

4 - Hijacking a process by modifying its run-time stack

4.2

a) For attackerA, we must find the address at which the return address is located on the process's stack which we are attacking. To determine this address, I first had to know that the stack is set up like this:

EIP
old EBP
EFLAGS
EAX
ECX

EDX
EBX
ESP
EBP
ESI
EDI **(ESP points here)**

In order to get to EIP, I had to do some pointer arithmetic to get up to EIP. To skip over the General Purpose Registers (GPRs), I had to look at how many register 'pushal' pushes. It pushes EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI - a total of 8 registers. EGLAGS and old EBP both take up 1 space, so I needed to go back 10 spaces to get to EIP.

4.3

a) First, in **attackerB.c**, I make sure that quietmalware() functions as a detour before going to the original EIP. To do this, I save the original EIP in a local variable, then overwrite it in the stack with the address of quietmalware() (in the same fashion as in 4.2 - ***(prstkptr + 10)**). After that, I put the original EIP above the overwritten value in the stack - ***(prstkptr + 11)**. The final product looks like this:

original EIP - victimA
overwritten EIP - quietmalware
old EBP **(current location of EBP as well)**
EFLAGS
EAX
ECX
EDX
EBX
ESP
EBP
ESI

EDI **(ESP points to the top of this entry)**

This makes ctxsw() return to quietmalware(), then quietmalware() returns back to the original EIP, which is victimA, like no detour ever happened.

The next challenge was changing the value of x to 9. I had to find where local variables are stored in the stack first. Since there's only one local variable, I didn't have to worry about iterating through that local variable area in the stack to find 'x'. The stack layout, starting from EBP is below:

STACKMAGIC **(EBP points here)**
INITRET **(function that kills a process)**
old EBP
0 - padding from gcc
EFLAGS
local variable(s)

This means the local variable (x) that I need to modify is at *(EBP - 5), since EBP points to the top of this stack frame. These values were pushed when create() was called to make it look like it was halfway through ctxsw. STACKMAGIC is the starting point for the stack used to detect stack overflow (not important right now). The next value pushed on is INITRET, which is the address of the function that kills the process. Next is the old EBP, which is used so that we can jump back to the previous stack frame by restoring it to EBP. Next, pushed after old EBP, is padding from gcc that is ignored during runtime. Next is EFLAGS for restoring the executing state of the processor, namely restoring interrupts (0x200). The local variables are pushed on below this point, and since there is only one, it has to be the space directly below EFLAGS in the stack. This means *(EBP - 5) is the local variable 'x'. Directly changing *(EBP - 5) from 5 to 9 gives the desired output.

BONUS

a) In kill.c, as the current process is being killed, it should check if there are any queued processes in prblockedsenders (do this at the VERY end). If there are, dequeue prblockedsenders until it's empty. For every process dequeued, ready it up. Once readied, the newly readied process will continue bsend(). At the very end of bsend(), the readied processes will check if the receiver's state is PR_FREE. If that is the case, the receiver process has been terminated, so return SYSERR.