A Tale of Two Clients

Plot summary. In which a client Chuck (C) is having a pleasant exchange of requests and responses with server Sam (S), when Mary (M) suddenly appears and attacks Sam.

The scenario unfolds exactly as in the lab/project p2. Sam's server runs the "BuggyServer" code you studied for p2, ready to receive connections on port 9000. All processes involved are single-threaded. All participants run "classic Unix". All processes use the various Unix system calls as discussed in class.

The scenarios involve two simple and familiar Unix utility programs. The echo program simply writes its arguments to stdout. The nc program is netcat, which opens a socket and then uses read and write system calls to transfer data from its stdin to the socket and from the socket to its stdout.

Instructions. Answer all questions. Please attempt to confine your answers to the boxes provided. The small boxes are worth 10 points. The large boxes are worth 20 points.

Many questions on this exam ask you to explain process states of Chuck, Mary, and Sam at various stages in the narrative. For those questions, the state of a process is the state of its thread: either Running, Exited, or Blocked. If a process is Blocked, name the system call in which it is blocked and the object it is blocked on. Please assume that each process has had sufficient time to execute to reach a stable state.

As always: “Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., linked lists: don’t write code for those.” If you don’t know the answer to a question, then just say something else relevant for partial credit.

There are no “April Fools” jokes on this exam, with the possible exception of this statement.

A quick note about the answers. Many students said that the state of some process (thread) was Running. But in general a process (thread) will always block or exit if you give it a millisecond or two to quiesce. The only processes that are ever in the Running state for longer than that are performing some involved computation, or they are stuck in some kind of endless loop or livelock. But all of the processes at issue here are just managing I/O. So the answer was always Blocked (with an Exited or two); the important question was: blocked on what? Note also that ports and sockets do not block. Rather, threads block on ports and sockets.
Part 1. In which Chuck sends a request. (50 points)

Chuck types the following command to his shell to issue a valid HTTP request to Sam:

/bin/echo "GET /file HTTP" | nc sam 9000

(a) What is the state of Sam’s server process before the request is sent?

(b) Which of Chuck’s processes creates a pipe? How does another process gain access to the pipe?

(c) What is the state of each of Chuck’s processes after the request is sent, but before Sam’s reply is received?

(d) List the key system calls that the nc process issues before arriving at the state in (c), in the order it issues them.

(e) What causes Chuck’s nc process to (eventually) exit?

**Block in accept() on the web socket** (bound to port 9000).

The shell process interprets the command string, sees the | symbol, and creates the pipe object with a pipe() syscall. It then forks two child processes to run echo and nc. The children inherit both ends of the pipe from the parent. The left child closes the read end and dupes the write end over its stdout. The right child closes the write end and dupes the read end onto its stdin.

shell: blocked in wait() for children to exit.

echo: exited

nc: blocked in read() awaiting reply from Sam on socket (blocked in connect/write OK).

Forgetting the shell in 1(c) and 1(d) costs “about 5 points” total. It should have been more.

dup*, close, exec* (while running shell program)

socket, connect, write, read (while running nc)

For true completeness, we also have to read() from the echo pipe at some point (0 points).

Sam sends a reply and **closes the server socket** when it is done sending the reply. That causes nc to learn the reply is complete (e.g., by reading an EOF from its socket). Note that reading an EOF from the echo pipe is not enough: nc passes all data from the socket to its stdout before it exits, so it needs an EOF on the socket too.
Part 2. In which Mary attacks Sam. (60 points)

Mary types the following command to her shell to issue a valid HTTP request to Sam. The request contains a malicious attack string with an embedded shellcode to inject into Sam's server. Sam's server receives the attack string and copies it into a local buffer variable of its handle() procedure, overflowing the buffer.

/bin/echo -e "GET <attack string> HTTP" | nc sam 9000

(a) Why is it necessary for the copy to overflow the buffer in order for the attack to be successful?

(b) Mary must guess the location of the buffer in the server's virtual address space, and include a stack address within the attack string. Why is this necessary in order for the attack to be successful?

(c) Draw a picture of Sam’s server stack after the handle() procedure is called but before the copy operation occurs. Be sure to illustrate the key elements that are relevant to the attack.

(d) Draw a picture of Sam’s server stack after the copy operation but before handle() returns. Be sure to show the key regions of the attack string and their locations relative to the elements in (c).

It is necessary to **overwrite the return address** (RA) in order to transfer control into the shellcode. The RA is stored separately from the buffer, just past the end of the buffer.

It is necessary to **replace the saved RA with an address of the shellcode** or a preceding NOP slide, wherever they land on the stack.

Stuff that must be present:
- buffer
- saved RA
- which way is “up”

You should offer some additional detail:
- saved params, and/or
- saved frame pointer, and/or
- other locals, and/or
- env/argv at the base of the stack
(5 points off for offering nothing)

For completeness, you really should show the whole stack, and not just the handle() frame. 5 points off for no indication that there are other frames lower in the stack.

Stuff that must be present:
- RA value to overwrite RA
- NOP slide
- shellcode region
- New RA points into NOP slide

These should be ordered right. Code executes toward higher addresses.

Note: the shellcode could fit within the handle() frame (as advocated by “Smashing the Stack”) but take care: stack operations performed by the shellcode may overwrite the code.
Part 3. In which Sam executes Mary’s shellcode. (40 points)

The shellcode is a sequence of machine instructions with the usual behavior: create a fresh socket, bind the socket to a port number (the shell port), listen/accept a connect request from Mary on the shell port, and use the execve system call to run a shell program that receives commands from Mary and sends their output back to her.

(a) One possible outcome is that the server process generates a fault when it executes its first instruction from Mary’s attack string. (Let’s say the instruction is a NOP.) What is the most likely cause of the fault?

(b) If the attack succeeds in running a shell, what user identity does the attack shell run with? How does Sam’s kernel determine what user identity to place in the security label of the shell’s process?

(c) The nc process that Mary used to launch the attack (in Part 2) does not exit after the attack. Why not? How is it different from Chuck’s nc process in Part 1?

(d) What is the state of Sam’s server process after Mary connects to the shell port, but before she sends a shell command?

---

**NX**

By default a modern server is configured so that the stack region is not executable. Any attempt to execute an instruction on the stack raises a protection fault or segmentation fault. We disabled this protection for p2. Note: since the instruction is a NOP, it cannot raise any other kind of fault.

The attack does not change the userID: the attack shell runs in the same process with the same userID that the server process was running with before. That could be root (for a poorly administered server) or more likely it is an ordinary user ID, say, Sam. The kernel sets the userID in the label by inheriting the value from the parent process, unless the value is changed with a setuid system call or execution of a program with the setuid bit set. In any case, the attacker has access to any files that the server had access to before the attack.

Once the server’s control flow is forced into the shellcode, it does not return normally from the handle() routine, and therefore the server does not send a reply. Per 1(c), the client’s nc process is blocked awaiting a reply. After receiving a complete reply it exits, per 1(e).

**Blocked in read() on stdin.** Once Mary connects to the shell port, the shellcode (as given above) returns from accept() and then uses exec* to overlay the server process with a completely different program: the attack shell. All server state is gone, and now there is an ordinary shell (not shellcode) reading from the socket as its stdin, blocked awaiting a command. Note: the shellcode does not fork/wait: it could, but need not and does not as described above.
Part 4. In which Chuck is out of luck. (30 points)

Chuck repeats the command of Part 1 after Mary initiates the attack, but before Mary disconnects from her attack shell.

/bin/echo "GET /file HTTP" | nc sam 9000

Chuck types the command, hits the return key, and....nothing happens.

(a) What is the state of Sam's server process at this point?

This was unintentionally the same as 3(d). Nonetheless I got some different answers. The point is that the server process has been hijacked. It does not accept() connections on 9000 as it would have before, because its stack was smashed: it will never return from handle() and therefore will never execute that accept() code ever again. Moreover, once Mary connects to the shell port (not 9000), the process does an exec* and runs her shell (not shellcode). Instead, it reads commands from Mary from her connection/socket, executes them, and does nothing else.

(b) What is the state of each of Chuck's processes at this point?

This is the same as 1(c). Nonetheless I got some different answers.

Note: the echo process does not receive or wait for any input from the pipe. Pipes are unidirectional. Also, the nc process does not fail to connect to port 9000: port 9000 is still open and in a listen state in Sam’s process, but the process is now running a shell that does not know or care anything about port 9000 and so does not accept() or respond to queued connections there.

(c) What is the state of Sam's server process after Mary disconnects from the attack shell by exiting her client program?

Once Mary closes her socket (i.e., on exit), the server process (running the attack shell) reads any remaining data and then receives an EOF on its stdin, which causes the shell to exit the process.

I saw some strange ideas suggesting that the port had been “blocked” or “occupied by a connection”, and is now “free” again after Mary disconnects. That is not how it works. A server socket may be bound to a single port (e.g., 9000, or the shell port opened by the shell code), but once bound, the server can accept() many connections on that port/socket concurrently. This is why it is so important to understand accept(). Each accept() call returns a new socket for a new client connection that has arrived on the server’s port, and is established and sitting in the listen queue. Again, the reason Chuck’s connection attempt “hangs” is not that Sam’s port is “blocked” or “occupied”, but that the server process no longer calls accept() because it is off doing something else instead. At bottom it is a concurrency problem: the server does only one thing at a time.
Part 5. In which Mary gets smarter and avoids detection. (20 points)

One weakness of Mary’s shellcode is that Chuck might report his problem to Sam, who might then discover Mary’s attack.

If Mary is busted, she could forfeit her property and be imprisoned for 20 years under US Federal Penal Code 18 § 1030 (CFAA, the Computer Fraud and Abuse Act), and regret this sorry episode for the rest of her miserable life.

Propose a solution for Mary. Of course, Mary should not mount the attack at all, but if she is committed to this path of darkness, how could she modify her shellcode to hide her attack from Chuck? How could she avoid disrupting the functioning of Sam’s webserver? Discuss any practical difficulties of your idea and any assumptions needed to overcome them.

I expected a few good answers for these last few points, and I got a few. Thanks!

Mary must (1) run her attack shell as a separate process, so she can (2) leave the web server intact and serving requests as it was before. (She might be able to do something cute with threads, but we stipulated at the start that all processes are single-threaded.)

The first objective is easy: modify the shellcode to fork() a child to bind to the shell port. The shellcode (running in the child) then continues as before: block in accept() for a connection from Mary, then dup the socket for Mary’s connection to its (the child’s) stdin/stdout/stderr, and exec a shell in the child. Mary can now send her attack commands to the shell running in the child, while the parent goes on its merry way. 10 points for that fork. (With a minus for not recognizing the challenge of the second objective.)

The second objective is harder. After the fork, the shellcode running in the parent can do anything: it does not have to wait() like a parent (or at least a shell) typically would. But what to do? The stack is smashed, the old return address is (probably) unknown, and any locals and other state in the calling frame are (probably) toast. There is no way for the attacker to save that information before overwriting it, since the attacker cannot execute any of its code in the process until after that data is overwritten. So: there is no going back.

The clever answer is for the shellcode running in the parent after the fork to simply exec* the webserver program again. A close inspection of BuggyServer reveals that it does such an exec* automatically to “reboot” itself to recover from a fault signal. A successful exec* restores the server process and its address space to their pristine initial state before the attack (or fault). The trick is to know what arguments to pass to exec*. The program name and arguments are in argv[], which is typically at the base of the stack. So we need to know where the base of the stack is, but we needed to know that anyway in order to mount any attack to begin with (no ASLR!). And of course this trick fails if the attack string was large enough to overwrite the argv[] array. We can find the environment variables the same way, but we’ll have to get lucky. And perhaps they are not needed: BuggyServer doesn’t pass them on reboot.

It just might work. Otherwise it’s 20 years in the clink to think about shoulda coulda woulda.
Results. The distribution looks like it usually does.

The results here are consistent with the exam being of slightly narrower focus than usual. The median is a little lower than usual: 145 rather than 150. The top performers stand out a little more: there are 10 exams at 180 or above. The mean is higher because I was a little bit more generous at the low end.