Your name: _____

Sign for your honor:_____

Midterm Exam
CPS 210: Operating Systems
Fall 2012

Grading the Instructor:
20 Questionable Answers to Questions You Might Have Asked

I am running short on time to grade your exam papers, so I ask you to grade mine instead. Grade the following statements on a scale of 0 (hopelessly false or garbled) to 10 (true, correct, and complete), assigning partial credit as appropriate. Write your score in the left hand margin next to the statement number. You may justify your score in the space provided below the statement, e.g., by briefly noting any exceptions or errors in the statement, or any assumptions that the statement depends on. If you do not justify your score then I might take exception with it, even if your score is justifiable. All 20 statements are equally weighted (200 points total).

In these statements I make no distinction between various versions of Unix. In particular, Linux is just a flavor of Unix. When I say Unix I am referring generally to the Unix environment as described in the reading, discussed in class, and used in your project work. When I say Android I am referring to the Android framework built above the Linux kernel.

1. In Unix systems every process is created with the fork() system call and executes with the same protection label (user ID) as its parent.

Yes but. The process tree needs a root: the kernel handcrafts the init process at startup. (1 pt) All other processes are created by a fork syscall, or some variant thereof (e.g. clone, vfork). So if you said no then you lost points unless you said why (nobody mentioned fork or clone). Also: the child does not always run with the parent's user ID: it might change it with a setuid syscall (e.g., as the login program does) or by exec of a program whose setuid bit is set. (2 pts)

2. In both Unix and Android, Applications run in processes with separate virtual address spaces. They use the same process model implemented by the operating system kernel, but there is an important difference: in Unix, different applications running on behalf of the same user execute with the same userID, while in an Android system they run with different userIDs. That is, in Android the userID for an installed application is a property of the application, regardless of which user runs it, while in Unix it is a property of the user that runs the application, and not of the application itself: in Unix an installed application can run with different user IDs at different times, whereas in an Android system it always runs with the same userID.

Yes but. Unix has a mechanism to make the userID a property of the application: the setuid bit. Also, it might be possible for Android apps to run with the same userID if they are from the same app provider. It wasn't necessary to mention that about Android, but it was OK if you did.

In general, if you mentioned setuid attribute of a program for either Q1 or Q2 then you got the points for it for both questions (2 pts each).

3. Processes that pass data through a pipe or pipeline are children of a common parent process. They are members of the same process group, which is a different process group from the parent. The parent sets up the pipeline. Each pipe transfers data in only one direction: it links the standard output of one process with the standard input of its successor. The standard input of the process group leader receives input from the terminal. The standard output of the last process in the pipeline posts output to the terminal.

This is generally true for pipelines created by a shell. What I was looking for here was that redirects may be used to read/write on a file instead of the terminal, at the head and/or tail of the pipeline. (2 pts) It is OK to say that the children have the same process group as the parent by default, but it is wrong to say they are always in the same process group: any shell will put its children in a separate process group to protect itself from certain signals sent to its children.

4. In Unix, one program (an initiator) can launch another (the target) only if the initiator knows the pathname of the target's executable program file. In Android, an initiator can launch a target by initiating communication with one of the target's components declared in its application manifest. It must know the name of the component, but not necessarily the name of the file(s) containing the component's code. Android finds the code and uses Unix system calls to launch the target application and set up the communication channel between the initiator and target.

People looked for all kinds of reasons why this isn't true, but it is. Mostly the answers I got were non-sequiturs, which are harmless as long as you don't garble any details. As with all statements, if you marked the statement as partly false but your answer did not say why, then I knocked off a point just out of annoyance of having to read all your scrawled text searching for the missing justification. And I took off more points if your answer made other statements that were wrong.

It is true that Android apps need permissions to launch another app activity via an intent, though it wasn't necessary to point it out.

5. After an attacker succeeds in mounting a trojan horse attack on a Unix user, the attacker's malware then has access to any of the user's files. But on Android a trojan horse malware program cannot access a user's data even if the user is tricked into launching the program.

You can add various qualifications to this statement, but it was enough to just say that it is true. Most of the qualifications don't amount to much. For example, Android allows a Trojan app to access any of the data that the malware itself creates, which could be called the "user's data". And if the app requests permissions at install time and the user grants them, then that might expose more of the user's data to the malware. Also, other Android apps can choose to reveal their data to everyone, including any malware. And Unix users can try to protect their files from programs running with their own identity (including malware) by fiddling with the access mode bits (e.g., 0444), but the program can always override those protections: the malware can access any data that the user can access.

6. A Unix system call occurs when a stub procedure in the standard library executes a special machine instruction that posts a trap event to the kernel, and the machine delivers it by invoking a handler routine registered by the kernel. The handler routine executes in protected kernel mode and has access to internal kernel data structures. It uses a kernel stack for the process that initiated the trap, and its execution may be interleaved with the execution of other processes on the system. When it has completed the request it returns control back to the stub that initiated the trap, executing in user mode on a stack in the process virtual memory.

Y*es but. Some system calls do not return, e.g., exec* and exit. (2pts) Some people said that interrupts and faults are also system calls, but they're not. Some people do use the terms loosely, but we define the terms more precisely in this class to reflect important distinctions.*

7. The Unix wait() system call and variants such as waitpid() block the calling process: the process is placed in a sleep state, suspending its execution until a child process stops or exits.

Yes but. Wait does not necessarily block the caller. (2 pts) For example, if a child has already changed state (stop or exit) then wait may return immediately with the information. Modern systems have options that do not block the caller in any case. I expected you to know about that because your dsh uses them to implement the "jobs" builtin command as we have defined it.

8. If an attacker succeeds in compromising a Web Certifying Authority (CA), and obtaining the CA's private key, then it can create a fake version of any website. The browser HTTPS/SSL security mechanisms are unable to detect that the website is fake.

True, with optional qualifications: to mount a successful attack, the adversary must be able to interpose itself to intercept/hijack traffic to the DNS name or IP address for the "real" website. Also, the browser must be configured to trust certificates issued by the compromised CA. Today's browsers come pre-configured to trust dozens or hundreds or even up to 1000 CAs, including many whose trustworthiness is suspect. It is also true that various projects are attempting to improve security by adding new mechanisms. It was not necessary to add these qualifications, but it was OK if you did. Some people could not believe that a browser would take any CA's word that a public key is bound to a given DNS name, even if the owner of the DNS name (e.g., google.com) does not use that CA to issue its certs. It was OK to express skepticism, but if you justified your skepticism with a false statement then you lost some points.

9. Secure Sockets Layer (SSL, as used in HTTPS) is based on symmetric encryption/decryption because it is faster than asymmetric encryption/decryption. This choice requires an extra step during the protocol to 'change ciphers'. Data transferred on the channel is encrypted with a key chosen by the parties for use within that channel. The key is shared by both parties, but is not known to anyone else.

It should be added that SSL/HTTPS uses asymmetric encryption/decryption as well as symmetric. Specifically, it uses asymmetric crypto during the handshake phase at connection setup to exchange the secret key (the session key) without the risk of having it stolen. Once the parties are in possession of the session key they change ciphers to symmetric crypto using the session key, because it is much faster than asymmetric crypto. Mentioning this in some form was worth 3pts.

10. A well-chosen password protects against "fake ID" attacks in which the attacker logs into a system with the identity of a victim user. The password should not be "letmein" and it should not be written down on a piece of paper next to your keyboard.

All of this is true, except that there are plenty of other ways to steal a password, so choosing the password well is not sufficient to protect it. Mentioning phishing (or social attack) was worth 2 pts, since it is the most common way to steal a password. It might be good enough to say generally that the password must be chosen well AND guarded carefully. No points for mentioning the risk of "dictionary attacks", since a well-chosen password is a strong defense against a dictionary attack. I let it go if you also said that the password should be changed frequently to further reduce the risk of dictionary attacks.

11. If Alice knows Bob's public key, and Bob signs all of his messages with his private key, then Alice can verify that a received message originated with Bob and has not been tampered. If Bob includes a nonce in each message, such as the time at which Bob sent the message, then Alice can verify that each message is fresh, i.e., that it is not an old message replayed by an attacker.

This is true, but it makes the crucial assumption that Bob alone possesses Bob's private key, i.e., Bob can keep a secret. In practice, systems have no alternative but to make this assumption,

and they are expected to make it, so failing to point it out was only worth 2 pts. Even so, it is a fundamental assumption of cryptosystems. They are useless if the assumption is violated.

12. It is not necessary for a heap manager to zero out the memory blocks that it returns (e.g., from malloc()): the kernel zeros all virtual memory in the heap (e.g., from sbrk()).

"Zero out" (clear) means to assign zero values into all bytes of a block of memory. A system might clear blocks either to conceal what was there before (security and isolation) or as a convenience for a program that uses the memory. But zeroing memory is expensive, so systems generally do not do it just as a convenience: they allow the program to decide whether it is necessary. And since a heap is for use within a single process/program, it is not necessary for the system to conceal that program's old data from itself.

Therefore, programs should not assume that heap blocks returned by malloc are zeroed: it is the caller's responsibility to initialize the memory for its intended use by the program. (2 pts) Failing to initialize values in heap-allocated structures is a common source of errors. This is one of the first things to know about malloc().

It is true that sbrk() does return zeroed memory: the kernel clears memory frames that it grants to a process in order to avoid leaking data across process boundaries, since the frame may have been used previously by some other process. It was not necessary to know that or say it, but denying it cost a couple of points. In any case, it is irrelevant because getting clear memory from sbrk is not sufficient to ensure that data returned by malloc is clear, because heap blocks are frequently reused/recycled. If you said that malloc should return clear memory and the sbrk property was indeed sufficient to ensure that, then you lost more points.

13. A bug in an application program could cause a heap manager (Lab 1) to fail or fault, e.g., a bug in an application program could cause the heap manager code to reference a pointer that is null or wild. That could happen even if the heap manager itself is perfect. Thus the heap manager is not part of the system's Trusted Computing Base. In contrast, a bug in an application program can never cause the kernel to fail if the kernel is implemented correctly.

All true. In particular, a wild program can arbitrarily overwrite the heap manager's metadata. A surprisingly large handful of students said that the Lab1 heap manager is part of the kernel and so is protected. That was a big mistake.

14. The execve() system call can pass one or more argument strings to the program that it executes, but the kernel must assist in transferring the argument data because it crosses an address space boundary for delivery into the child process. Similarly, an executed program can return a short result (an exit status) via the exit() system call, but the kernel must assist to deliver the result data across an address space boundary into the parent process.

This is all true, except that the exec or execve does not pass the strings across an address space boundary or pass data into a process (called the "child" process in the question, as it would be in shell scenarios). The child process calls exec to overlay portions of its address space with a new program to execute. Exec arranges for the argument strings to be preserved and made available to the executed program, e.g., by copying them to some safe place in the address space and passing a pointer into the program's main(). Pointing this out in some way was worth 2 pts.*

15. A program must be careful in how it handles any data that it reads from a socket, to avoid a cross-channel attack such as a buffer/stack overflow. Support for secure sockets (e.g., SSL/HTTPS) is an important defense against such attacks.

The point to note here is that although SSL/HTTPS may authenticate the channel and does keep it private and tamper-evident, it does not prevent one side from mounting a buffer overflow attack

across the channel. So it is wrong to say that SSL/HTTPS is a defense against such attacks. (3 pts)

16. A file system implementation is vulnerable to external fragmentation, but it can compact data stored on disk to reduce the problem.

We have not discussed file systems in any depth, and file system implementations vary, but the simple model presented in class (and in the Unix reading) gives a basis to answer this question. File systems represent data and metadata in blocks on storage ("disk"), generally using fixed block sizes. Files are variable-length objects, but they can be assembled from blocks that need not be contiguous. The file is represented by a block map (such as a Unix inode), which is linked into other metadata structures, e.g., the directory tree. The maps and other metadata structures contain "pointers" referencing logical block offsets on the disk. The file system software can traverse the metadata structure from a root (e.g., "superblock" or root directory) and find all of the maps and all of the "pointers" they contain.

There are a few points to make about the statement based on this information. Fragmentation can be a problem for any allocation of variable-length objects from any storage: a surprising number of students suggested that it is specific to heap managers. But fragmentation in file systems is generally internal rather than external (2 pts) because the block map approach reduces external fragmentation (2 pts). Even so, file systems can indeed compact the data by changing the block locations and "rewiring the pointers" (cf. Google Chrome comic), since they know where all of the "pointers" are. Compaction is distinct from compression. It reorganizes data/metadata blocks to be closer together, and reorganizes free regions into larger free regions (logical "coalescing").

Some students seemed to rely on their prior knowledge of "defragmenting" programs for file systems. These programs perform file system compaction. Some archaic file systems (such as MS-DOS/FAT) are vulnerable to external fragmentation, and do require compaction to defragment them. It is also true that, in general, file system compaction can improve performance by reducing the distance that a disk arm must travel to serve the request stream, even if external fragmentation is not a problem. But you were not presumed to know these things since we have not covered them.

17. Cryptographic hash functions (also called secure hashing or SHA) are useful even if the result digest (also called a hash or fingerprint) is not encrypted, as it is with digital signatures. For example, if Alice knows a secret, and passes Bob a digest of the secret, then Bob can determine if another party also knows the secret, even without knowing the secret himself.

This question implicitly assumed that "another party" would demonstrate its knowledge of the secret by passing its hash/digest/fingerprint to Bob. It wasn't necessary to point this out, but it was OK if you did.

It is true that Bob can verify that another party knows the hash without knowing the secret himself. The problem here is that knowledge of the hash is not sufficient to demonstrate knowledge of the secret. (2 pts) For example, Bob knows the hash but does not know the secret.

Some students suggested that encrypting the hash would help, or made other ill-considered suggestions that might have cost a couple of points. Some presumed that the hash was encrypted, even though the statement says that it is not.

Some students took the statement as a reference to servers that store user passwords as hashes as a defense against theft of the server's password file. In that scenario a user authenticates to the server (Bob) by revealing the actual secret (the user's password): Bob can verify that a given value is in fact the correct value of the secret by hashing it and comparing it to the stored hash,

and then forgetting the value. This was OK if you were explicit enough about how you understood the question and in what sense Bob "does not know the secret" in this scenario (i.e., in this case the secret itself is revealed to Bob, but Bob chooses not to store it or remember it).

18. Standard Unix file names are 'hard links', and they have a number of useful properties: a file can have multiple names/links, and the file named by a given link does not change, even if someone destroys another link/name for the file (e.g., using the unlink system call or rm command), and then reuses the same name to link to some different file with different contents. Symbolic links are different: a user can modify a symbolic link to switch it from an 'old' file to a 'new' one, and any process accessing the file through that symbolic link sees the new file rather than the old one. However, a symbolic link can be a 'dangling reference' that names no file at all, whereas a hard link always names a unique file.

This is all true and correct. Of course, the contents of a linked file can change, but even so the identity of the file does not change. (It was not necessary to point this out.) Some students added some detail here and there. Some students preferred the term "soft" link to "symbolic" link, but they mean the same thing. Each hard link names a unique file (i.e., the name mapping is a function), even though the name of the file is not necessarily unique, i.e., a file may have more than one link (a many-to-one function), as the statement says.

This statement also has an unintended ambiguity: the "modify a symbolic link" is intended to mean "modify the target of a symbolic link". One student pointed that the way the statement is worded it does not truly justify the claim that symbolic links are "different" (the antecedent to the colon), even if the next sentence does indicate a clear difference. (Yikes.) This student received full credit and is recommended to apply to the Law School.

19. The Garlan/Shaw paper refers to event abstractions as 'implicit invocation' because they enable the producer of an event to cause invocation of a handler procedure/method in some other component, without naming the receiving component or even having to know its name, and without invoking the method explicitly, e.g., by binding to an advertised service API and invoking it with RPC (e.g., binder) calls. The producer merely generates the event and continues executing while the event system invokes any handlers registered to receive the event. Android supports a 'broadcast' event abstraction through the 'receiver' component type.

This is all true and correct. Some students added that Android uses the term "intent" to include the Garlan/Shaw event abstraction. Android supports two flavors of the event abstraction: "normal" (parallel) and "ordered" broadcast intents. Some students offered more detail about how components are named (package name for app, as given in the manifest, + component class name).

20. Machine-level event mechanisms are important for implementing native virtual machine hypervisors. If a program generates an exception event such as a trap or fault, or if a device generates an interrupt, the program or device does not know or need to know how the event is handled, i.e., whether the OS kernel handles it or (alternatively) if it is intercepted by a hypervisor handler 'below' the kernel.

All true. I should note that the term "machine" can refer either to a virtual machine or a physical machine. The statement uses it in both ways, which may have been unintentionally confusing.