

A NOTE REGARDING COVERT CHANNELS

Timothy E. Levin and Paul C. Clark

Naval Postgraduate School

Abstract: This note presents an overview of some abstract concepts regarding covert channels. It discusses primary means of synchronization and illicit interference between subjects in a multilevel computing environment, and it describes a detailed laboratory exercise utilizing these abstractions.

Key words: security, covert channel, interference, multilevel, mandatory

1. INTRODUCTION

In a multilevel computing environment, a security policy is enforced which requires that low-sensitivity subjects (e.g., a process or task) should not observe high-sensitivity information (e.g., data, code, or activities of high-sensitivity subjects). The most intuitive interpretation of such a policy is a confidentiality policy, in which for example, subjects with a low clearance are not allowed access to highly classified data¹.

A multilevel system may enforce such a policy on all subjects under its control and all of the objects that it exports to those subjects (viz., objects to which an explicit reference is possible via a system interface). Such an enforcement mechanism is said to enforce *mandatory access control* (MAC) with respect to the *exported objects*.

Despite the successful enforcement of MAC, a *covert channel* exists in such a system when information can be passed from a high sensitivity *sender* subject to a low sensitivity *receiver* subject via an *internal* object (i.e., one that is not an exported object). This reflects a processing model in which all interactions between subjects occur through objects of some type, such as buffers, messages, registers and files.

Covert channels are normally conceived as a medium for a *series* of transmissions from high to low. Thus, for each transmission, the receiver has to know when to read. This is done through a *synchronization* mechanism. There also needs to be something – the internal object – that the sender can modify and the receiver can observe: this forms the *interference* mechanism of the channel, as shown in the Figure 1.

¹ A similar interpretation applies to *integrity* policies, e.g., wherein subjects with high integrity should not execute or observe low-integrity data or code.

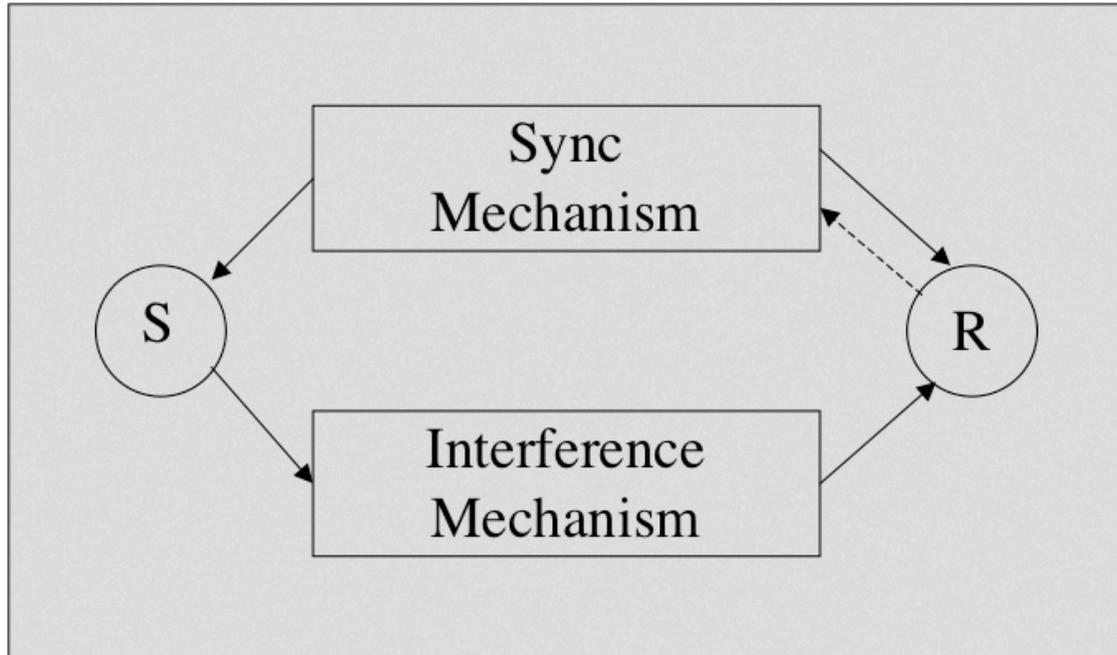


Figure 1. Abstract Covert Channel

2. SYNCHRONIZATION

An important distinction for a multilevel security system is that the sender (high) cannot send synchronization signals to the receiver (low) except through covert channels (if a receiver can *directly* read an external object that a sender writes to, it is considered a system flaw rather than a covert channel). The receiver, on the other hand, can send “I am ready” synchronization signals to the sender, in which case, the synchronization mechanism is a legitimate “channel” from low to high. Alternatively, they can synchronize via out-of-band (temporal) agreements relative to a “clock” mechanism that they both can read, such as, “you write on the odd time periods, and I will read on the even time periods. Synchronization can also be achieved through a mutually exclusive scheduling mechanism such as is commonly provided by an operating system.

In many descriptions of theoretical covert storage and timing channels, the sender and receiver synchronize by reading a clock or by depending on a scheduler to alternately activate them, because polling and the use of additional covert channels for synchronization may be less efficient for the purposes of optimal channel capacity estimation.

3. CLOCKS

If there exists a fluctuating variable, not modified by the sender, which the receiver consults to interpret or decode the data received, then that variable is effectively a “clock.” A clock can have regular or irregular progression; and it can have different possible values, including a binary range. Any modulated (e.g., fluctuating or monotonically increasing or decreasing) variable can be used as a “clock.”

4. INTERFERENCE

The sender *interferes* with the receiver through what we call a “control structure,” such as a file lock or disk-full indicator for a *storage channel*, and a scheduler or disk arm location for a *timing channel*. In most storage channels, the receiver indirectly reads the interference information via an error message; for timing channels, the receiver *interprets* the interference (e.g., how long was my delay) via a clock. This is shown in the Table 1.

	Synchronization Mechanism	Interference Mechanism	Interpretation Mechanism
Storage Channel	Clock/Scheduler	Control Structure	Error Message
Timing Channel	Clock/Scheduler	Control Structure	Clock

Table 1. Covert Storage and Timing Channels

The sender’s activity in a timing channel can be viewed abstractly as interfering with the receiver such that the clock variable, when read by the receiver, will be at the value that the sender wishes the receiver to perceive. This is true regardless of whether or not the clock is monotonically increasing by regular values. It is only required that the sender knows what effect its actions will have on the clock value perceived by the receiver.

For a priority scheduler based timing channel, the sender interferes by consuming a smaller or bigger time-slice. When the receiver is next scheduled (completing its wait for the CPU), it reads the “clock” to interpret the bit or bits transmitted – the clock is not the channel, as the transmission of information has already occurred in the *delay*. In another timing channel example, the control mechanism is an internal data structure (e.g., shared table or disk arm location), which is modified, added to, or deleted from by the sender to interfere with the receiver; the receiver indirectly accesses the control structure through (e.g.) a synchronous operation; after completing its wait for the operation, the receiver interprets the bits transmitted by reading the clock.

5. COVERT STORAGE CHANNEL EXERCISE

In this section we present a detailed description of a covert storage channel that is present in certain file system implementations. File system directories “contain” files, as well as a description of those files. When a file is created in a directory, the process creating the file abstractly modifies that directory. In a multilevel environment, then, normal secrecy rules require that the secrecy level of the process creating the file must be *equal* to the secrecy level of the directory. This presents some practical problems, such as maintaining publicly writable directories, like /tmp in Unix: how can processes at every secrecy level write to a directory if they all have to be at the level of the directory? To remedy this problem in Trusted Solaris², it supports the notion of a Multi Label Directory (MLD), sometimes referred to as a *deflection directory* in other multilevel operating systems.

An MLD is assigned the secrecy level of the process that creates it, but files may be created in the MLD by any process whose secrecy level is *greater than or equal to* the level of the MLD. The OS does this by creating hidden subdirectories at the level of the creating process, and creating the file there. For example, a process at the UNCLASS level can create *file1* in an MLD, and a process at the SECRET level can create *file2* in the same MLD. When the directory is

² It should be noted that this behavior exists in an older version of Trusted Solaris. The existence of this behavior has not been verified on the newer releases.

viewed from the UNCLASS level, only *file1* can be seen, and when viewed from the SECRET level, only *file2* can be seen. As desirable as this functionality is, it creates a covert channel.

The cause of the covert channel is traced to another desirable aspect of file system directories: we should not be able to delete a directory that still has files in it. A process will receive an error message if it attempts to delete a directory that is not empty. Given a low-level MLD containing a high-level file, a low-level process cannot see the high-level file, but if it tries to delete the MLD, it will receive an error message, and thus know that one or more high-level files exist in that directory. In other words, the covert channel exists because a high-level process can modify the state of the MLD, interfering with the ability of the low-level process to delete the directory.

At the Naval Postgraduate School, one of the lab exercises assigned in the *Introduction to Computer Security* course has the students take advantage of this covert channel to transfer data from a high secrecy process to a low secrecy process. In this exercise, the channel is initialized when a low-level process creates an MLD known by a high-level process. The two processes must also agree on the meaning of the signal. It is assumed that if the low-level process can delete the directory, then a bit value of 0 is being passed from high to low. If the low-level process cannot delete the directory, then a bit value of 1 is being passed. The high-level process puts something into the agreed upon MLD, or not, depending on the value of the bit being passed. There must also be some synchronization mechanism for knowing when the low-level process is ready to receive information, when it should try to delete the directory, and some mechanism for knowing when the full message has been transferred. Conveniently, this communication can also be arranged with other MLDs. In addition, it is possible to transfer more than one bit at a time by making use of multiple MLDs at a time.

Putting this all together, the following gives the steps that both sender and receiver use to transfer the contents of a file, eight bits (one byte) at a time, from high to low. We start by assuming the sender and receiver know the location of the “.backdoor” directory used to stage the activities.

5.1 Low-Level Receiver Process

1. Create all the MLDs used to transfer each byte of information, i.e., `.backdoor/bit0`, `.backdoor/bit1`...`.backdoor/bit7`.
2. Create the MLD used by the high-level process to signal when all the data has been transferred: `.backdoor/quit`. Once this directory is created, it also signals the high-level process that the low-level process is ready to receive data.
3. Repeat the following:
 - a. Create the `.backdoor/signal` MLD. When the high-level process detects that this directory exists, it is a signal that the low-level process is ready to receive another byte of information. The high-level process puts a file in this directory after it detects its presence as a signal to the low-level process that it is still transferring information.
 - b. Sleep for two seconds to give the high-level process time to wake up and create a file in the signal directory.
 - c. Try to delete the signal directory. Keep trying until successful, waiting one second between each attempt. Once it can be deleted, the high-level process is done transferring a byte of information.
 - d. Try to delete the `.backdoor/quit` directory. If successful, the high-level process is communicating that there is no more data to send, and the loop must be exited.
 - e. Extract the byte of information by trying to delete each bit directory. If a directory can be deleted, then the associated bit is a "0", otherwise it is a "1". Build a byte from this information.
 - f. Recreate the bit directories that were deleted in the previous step.
 - g. Save the byte.

4. Delete all the bit directories.

5.2 High-Level Sender Process

1. Keep trying to create a file in the .backdoor/quit MLD directory until successful. The .backdoor/quit directory does not exist until the low-level process creates it. This becomes a signal to the high-level process that it has initialized the MLD directories required to transfer a byte, and is otherwise ready to receive information.
2. Do the following until the contents of the file have been transferred:
 - a. Delete any files placed in the eight bit directories.
 - b. Request information about the .backdoor/signal directory until no error is returned. If an error is returned, go to sleep for one second. When the high-level process is able to obtain information about the directory, it means that the low-level process has created it, which means that the low-level process is ready to receive a byte of information.
 - c. Create a file in the .backdoor/signal directory. This prevents the low-level process from deleting the directory, which is a signal to the low-level process that the high-level process is not done transferring information.
 - d. Get the next byte in the file to transfer and determine which bits are 1's.
 - e. Create files in the associated MLD directories for those bits that are 1's. For example, if the 6th bit is a 1, then a dummy file is created in the .backdoor/bit6 directory. This prevents the low-level process from deleting this directory.
 - f. Delete the file created in the .backdoor/signal directory. This allows the low-level process to delete the directory, signaling that the high-level process has finished transferring a byte of information.
 - g. Sleep for two seconds. This allows the low-level process to wake up and delete the .backdoor/signal directory, signaling the high-level process that it is not done retrieving the information.
3. Delete any dummy files placed in the eight bit directories.
4. Delete the file in the .backdoor/quit directory. This allows the low-level process to delete the directory, signaling that there is no more data to transmit.

6. SUMMARY

Covert channel exploitation scenarios can vary widely, yet, most utilize the same abstract mechanisms to illicitly transfer information from a high sensitivity subject to a low sensitivity subject: synchronization and interference. We have provided concise descriptions of these abstractions, and used them to differentiate covert timing and storage channels, as well as to describe a concrete and detailed laboratory exercise.