Building an FTP Guard

Paul D. Sands
Sandia National Laboratories[1]
P. O. Box 5800
Albuquerque, New Mexico 87185-0806
pdsands@sandia.gov

ABSTRACT

This paper shows how an FTP Guard was constructed and implemented without degrading the security of the underlying B3 platform. The paper then shows how the guard can be extended to an FTP proxy server or an HTTP proxy server. The extension is accomplished by allowing the high-side user to select among items that already exist on the low-side. No high-side data can be directly compromised by the extension, but a mechanism must be developed to handle the low-bandwidth covert channel that would be introduced by the application.

KEYWORDS

File Transfer, FTP, guard application, proxy server

INTRODUCTION

Classified designs usually include lesser classified (including unclassified) components. An engineer working on such a design needs access to the various sub-designs at lower classification levels. For simplicity, the problem is presented with only two levels: high and low. The solution presented could be propagated as needed to create a solution for an arbitrary number of classification levels and categories, as shown in Figure 1. The figure shows a separate guard between each pair of enclaves. In practice, all guards going to an enclave (e.g., secret) could be combined.

If the low-classification component designs are stored in the high network, they become inaccessible to persons working on a low network. In order to keep the networks separate, the component designs may be duplicated in all networks, resulting in a synchronization problem. Alternatively, they may be stored in the low network and brought into the high network when needed. The latter solution results in the use of sneaker-net (copying the files from the low system to a tape and carrying the tape to a high system) or a file transfer guard.

Figure 1 shows the direction of data flow in the networks. However, whether by network, telephone, or other means, there is an implied request-flow in the direction opposite the data flow. If the request can be stated in advance, e.g., “get every file in directory D of machine M,” then no additional reverse flow is needed. This is the method used by Phase I (the FTP Guard, as

[1] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

implemented). However, if requests are to occur spontaneously, some method of real-time reverse flow is required, as described in the proposed extensions.

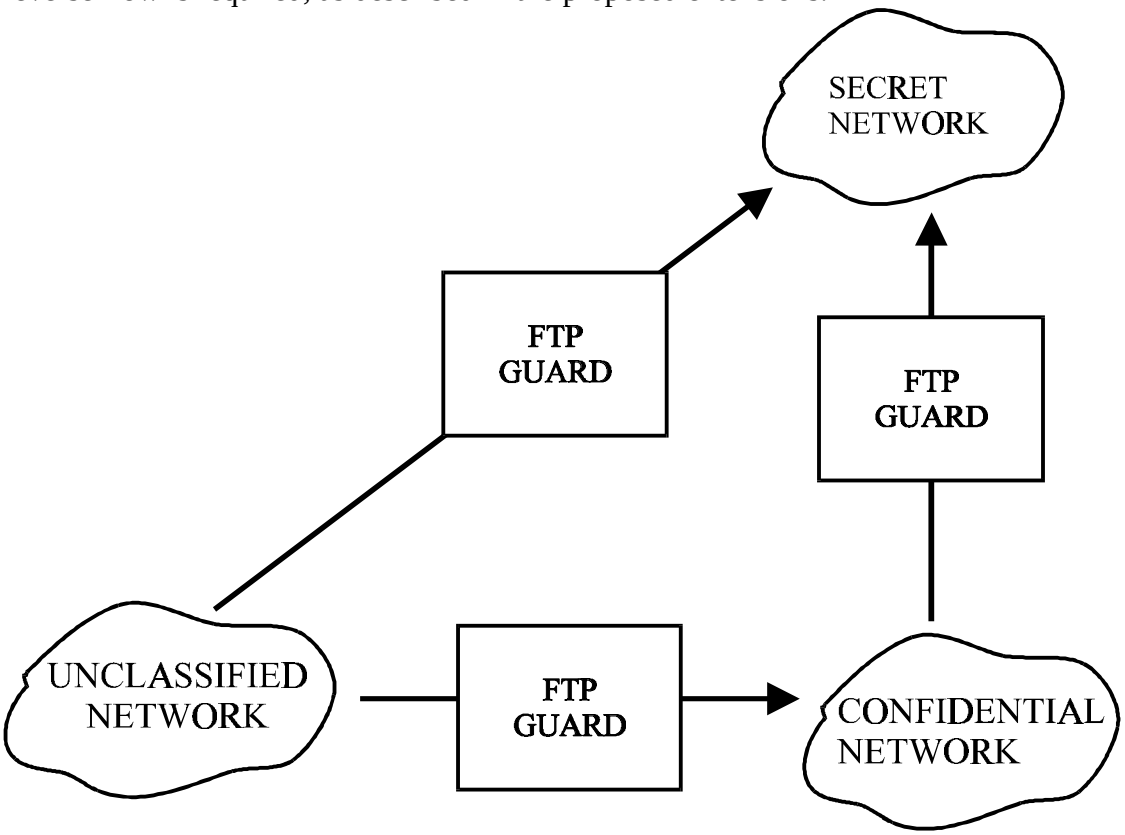


Figure 1. A Generalization Using an FTP Guard between Each Network Pair

DESIGN OF THE GUARD

The guard was designed to meet the following criteria:

- The high-side user can get low-side files using a standard (unmodified) FTP client.
- The high-side user cannot put files, rename files, or make any other change that would be visible to a low-side user.
- The low-side network contains hundreds of servers and millions of files. Files cannot be pulled up to the guard except as needed.
- High-side users are not allowed access to all low-side files. The guard must preserve low-side discretionary access controls.
- (Phase I) No reverse flow is permitted. The FTP software is unprivileged (subject to all restrictions of the security model).
- The guard system does not run any extraneous software.

The base platform for the MLS FTP Guard is a Pentium-based Wang XTS-300 computer running the STOP operating system. Its predecessor, the 486-based XTS-300 was evaluated by the National Computer Security Center and given a rating of B3. The Pentium configuration is being

evaluated under the RAMP program and is expected to be rated soon. The XTS-300 uses separate network cards for the high and low-side networks. Each network card is controlled by its own Internet Protocol (IP) daemon[2]. Network traffic is not forwarded from one network to the other.

The STOP operating system has a UNIX-like appearance, but is a secure system based on a four ring architecture, a mechanism that physically isolates security domains of the operating system. Communication between rings occurs only at designated gates between adjacent rings, preventing untrusted processes from calling security kernel services. The rings are

- Ring 0 - the Security Kernel
- Ring 1 - Trusted System Services
- Ring 2 - Operating System Domain / Some applications
- Ring 3 - Unprivileged Applications

The guard consists of four processes, which are replicated as needed to service users. The high-side process (server) is a modified FTP daemon, a process that listens for network requests and processes them. It allows users to get files that have been spooled by the low-side processes. The low-side processes are minion (a modified FTP client), cronjob (a time-driven job that processes user requests by calling a minion), and gman (the garbage collector). No processes are privileged. The high-side process cannot talk to any of the low-side processes nor delete any files (including spooled files).

The Cronjob Process

The Cronjob process creates a child process for each FTP Guard user. Every person wanting to use the FTP Guard must provide the Guard administrator with the following information:

- the name of a low-side computer from which to get files
- the directories and files to be obtained
- an account name and password to be used to get the files
- how often the files should be obtained (spooled).

The cronjob process starts a minion (FTP client) process to contact the low-side system and get the files. Interprocess communication messages are used to send requests to the minion. This mechanism was used in anticipation of later phases where a high-side process would be permitted to communicate with a minion process.

The Minion Process

The minion process looks like an FTP client to the low-side servers. It logs in as the appropriate user, gets the requested files, and writes them to the FTP Guard spool disk. It creates a directory index file that the server process will use to map low-side file names to spool file names. The directory structure created by minion allows discretionary access to be preserved for spooled files.

[2] A daemon process is one that controls a device, as opposed to one that talks to a user terminal.

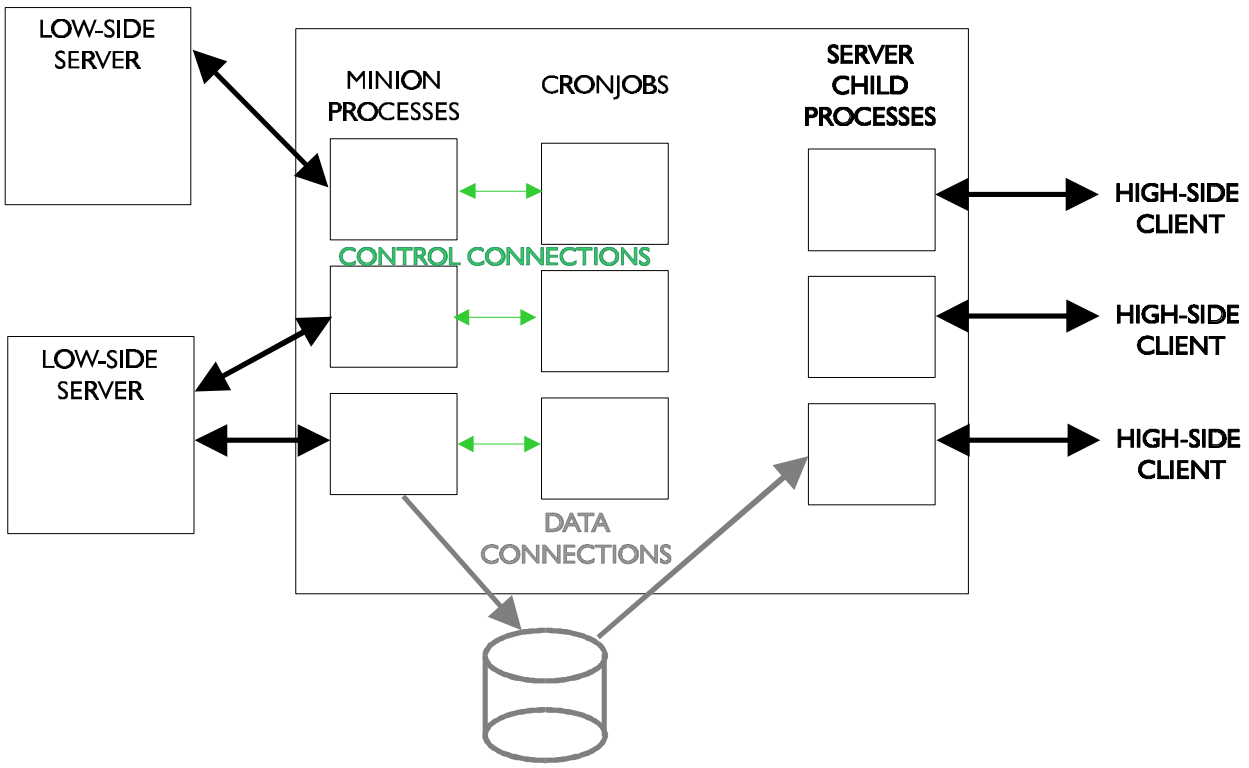


Figure 2. FTP Guard Processes (Phase I)

The Server Process

The server process runs at the security level of the high-side and looks to a high-side user like an FTP server. High-side users must authenticate themselves to gain access. Once connected, the user appears to be looking at the low-side system. The standard FTP commands (CHDIR, LIST, GET, etc.) are available and appear to be acting on the low-side system. In actuality, this process never communicates with the low-side, but reads spool and index files that were created by a minion process.

Since the server is a high-side process and the spooled files are at a lower security level, the server cannot delete spool files when they are no longer needed. Nor can it send any kind of message to a low-side process that it should delete the files. Either of these actions would constitute a write-down, which is not permitted by the security model. Therefore, spool files must be deleted by gman based on their age, irrespective of whether or not they have been retrieved.

The Gman Process

Gman deletes files based on their age. On the Phase I system, all spool files are deleted seven days after they are created. It is the user's responsibility to retrieve them before they go away. The system has a 2 Gb spool disk, and usage is currently light, so this is adequate. Gman could be tuned to have different aging times for different users or sizes of files. It can also be tuned to delete sooner as the spool disk fills up.

ENHANCING THE GUARD

The guard, as described above, contains no privileged processes. Since each component must obey all the rules of the security model, the final product does nothing to nullify the B3 rating of the underlying system. The guard is unfortunately less functional than it would be if we could develop a mechanism for allowing the high-side to request unspooled low-side files spontaneously. This capability would also provide a mechanism for extending the guard to other protocols, such as HTML. The enhanced capability described is not part of the FTP Guard, but is being considered for future implementation.

The remainder of this paper describes a mechanism for allowing requests from the high-side to the low-side. Unlike "dirty-word" filters, that generally catch high-side information accidentally being sent to the low-side, the proposed mechanism absolutely prevents the flow of high-side data to the low side. The selection mechanism is functionally equivalent to allowing a user to select from a low-side menu or pick a card from a deck of cards. No information is passed except a numeric selection code ("I'll take that one."). Any selection mechanism might be used to covertly signal information to the low side. However, the covert signaling mechanism can be made sufficiently difficult to exploit that it does not introduce a significant risk. The enhanced configuration is shown in Figure 3.

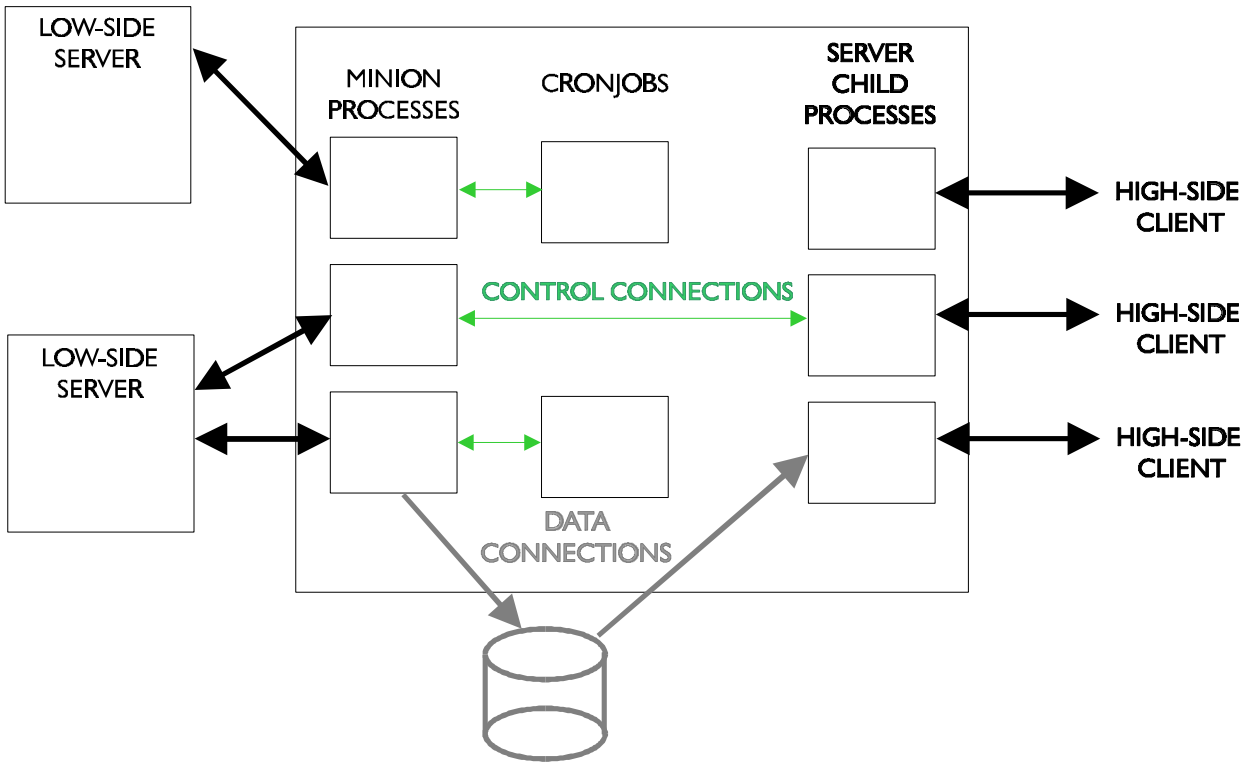


Figure 3. FTP Guard Processes (Enhanced)

Filtering the High-Side Request

High-side requests for low-side data consist of the following items:

- a low-side user name (for logging in to the low-side server)
- a low-side password
- the name of a low-side system
- the name of a low-side directory or file
- standard FTP commands.

Clearly none of these items can be high-side data since all exist on low-side systems. To ensure that no high-side data is accidentally sent out, a high-to-low request contains only a request type and a request item index. The valid request types are the following:

1. OPEN/USER/PASS - connect to a specified low-side system, as a specified username, with a specified password. The system, username, and password are specified by the line on which they occur in the FTP password file. The FTP password file is a file maintained by the Guard for FTP users. It is not the XTS-300 password file.
2. CHDIR - change to another directory.
3. GET - get the nth file from the current directory.
4. QUIT

Each request from the server to a low-side process (minion) will have a single numeric argument that references a low-side list. For example, user "bob" wants to get files from system "servadori" where his password is "IamBob". The high-side server will search the FTP password file for an entry that includes "bob,servadori,IamBob." If the entry is found, e.g., at line n, the server will tell its minion process to connect based on line n. If the entry is not found, the server tells bob his request is invalid. This means that users must be registered on the guard before they can use it. Selection of directories and files is done in the same way, but from a dynamic list. When the guard first connects to a system, it changes directory to the root and lists the directory. The list is written to the spool disk. A high-side user request is processed as follows:

A high-side process (server) accepts an FTP request, such as, "GET /aaa/bbb/file_c." It processes the command as described below. Messages from the high-side server to the low-side minion are restricted to the four commands and numeric arguments described above.

1. Send a CHDIR message to a low-side minion process to change to the root directory, "/". The CHDIR command supports two special arguments (root = -2 and parent = -1) in addition to an index into a table.
2. When a minion process changes to a directory, it always does a listing of the directory and updates the directory description on the spool disk. Whenever a minion completes a request, it sends an interprocess message to its caller, which may be at a higher security level. A high-side process (server) can read the spool data, which is at the low-side level. Read-down and write-up are permitted by the security model.
3. The high-side process will see if "aaa" is a subdirectory in the root directory. If not, it will tell the high-side user "File does not exist."
4. If "aaa" is the nth entry in the directory, the low-side will be told to change directory to the nth entry and do another listing. This will cause the spool area to be updated again.

5. The high-side process will see if "bbb" is a subdirectory in the directory. If not, it will tell the high-side user "File does not exist."
6. If "bbb" is the kth entry in the directory, the low-side will be told to change directory to the kth entry and do another listing.
7. The high-side process will see if "file_c" is a file in the directory. If not, it will tell the high-side user "File does not exist."
8. If "file_c" is the jth entry in the directory, the low-side will be told to get the jth entry. All files are obtained using the binary transfer mode. FTP commands such as MODE are supported between the high-side user and the Guard, but do not affect transfers between the low-side server and the Guard. The minion will store the file on the guard spool disk.
9. The low-side process will notify the high-side that the file is ready and the high-side will send it to the high-side user and delete it from the hard drive.

Since no information is sent to the low-side except what is already on the low-side, this process cannot directly cause high-side information to be sent out.

Extending the Guard to HTML

The process of selecting from low-side choices could also be used to create a Web proxy server. The guard would begin with a list of acceptable URLs[3]. A user moves about the Web by clicking on pictures and text which are linked to other URLs. As each page is brought in for a user, a list would be made of the URLs on the page. Selecting a URL then becomes a choice from a known low-side list. Some Web requests, such as forms, do not fall into this description. Since they allow the user to send arbitrary text, they would not be permitted. In addition, web pages would be filtered to eliminate executable content, such as JAVA bytecode.

Limiting the Ability to Exploit the Signaling Mechanism

A covert data channel could be created by associating one or more bits of information with a request for a specific low-side file. For example, suppose a process in the high-side network requests file file_a on computer server1 to send a 0 and requests file file_b on the same computer to send a 1. If the computer is in the registration database and the file exists, a request would be carried out. A privileged process operating on computer server1 could see that each file was requested, thus receiving the "1" or "0".

Various methods described below could be used to confuse or disrupt the covert channel, but it could not be completely eliminated. The Orange Book recognizes the inevitability of covert channels in multilevel systems:

> "... for many types of covert channels, techniques used to reduce the bandwidth below a certain rate (which depends on the specific channel mechanism and system architecture) also have the effect of degrading system performance provided to legitimate system users. Hence, a trade-off between system performance and covert channel bandwidth must be made."[4]

[3] A URL is a description of a resource on the Web, such as http://www.sandia.gov.
[4] Trusted Computer Systems Evaluation Criteria (Orange Book) DOD 5200.28-STD, p81.

The guard can confuse this data channel in various ways: limiting the number of bits that can be sent, adding bits to the message, omitting bits from the message, and reordering bits in the message. Some methods for counteracting the covert channel are the following:

- The actual commands sent to the low-side computer are not the ones received from the high-side computer. As described earlier, the high-side can only ask for directories or files that exist and are accessible by the username given. The guard never asks for a file that doesn't exist, and will not ask for a file again until it has changed. The channel bandwidth can be further limited by limiting the number of requests per day from each user and specifying a minimum time between requests for the same file, even if changed. Limiting the number of requests would limit the number of bits that could be sent covertly.
- Failed requests do not result in file requests to the low-side computer. The high-side user is just told "File does not exist." Not sending a request to the low-side would leave bits out of the intended covert message.
- The low-side computer is not told which high-side computer is making the file request. All requests come from the FTP guard. The low-side computer does know which user is requesting the file.
- The Guard could randomly send commands that are not requested by any user, e.g., change directory or get files that are not needed. Adding requests would confuse a covert message by adding unintended bits.
- At any given time, more than one process may be requesting files of the same low-side system. This could also result in bits being added to confuse a covert message.

The developers can further obstruct anyone attempting to use the covert channel by determining the kinds of information that would be needed to exploit it and limiting distribution of that information. This topic cannot be discussed in detail here without affecting the sensitivity of this paper.

CONCLUSION

The Phase I FTP guard is an effective, but limited, means to allow a high-side user to get low-side files. Because it is subject to the security model implemented in the B3 system, high-side users cannot directly request low-side files. Instead they must arrange with the administrator of the FTP Guard to have certain directories and files staged on a regular basis. This provides a useful initial capability, but is not an efficient mechanisms for cases involving large numbers of files or need for real-time access.

The system can be significantly enhanced and extended by allowing high-side users to send requests that result in high-to-low request flow. No high-data can be directly lost in this way, but a covert signaling mechanism would be introduced in the guard application. The covert mechanism introduced could not be easily exploited and could be slowed to an acceptable level without seriously degrading the file transfer process. Furthermore, use of the covert data channel can be detected by examining audit logs. If creating a versatile file transfer Guard requires the

addition of a covert signaling mechanism, it becomes the developer's responsibility to make it difficult to exploit.

Building an FTP Guard

National Information Systems Security Conference

October 6, 1998

Paul D. Sands
pdsands@sandia.gov

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy under contract DE-AC04-94AL85000.

Sandia National Laboratories

Building an FTP Guard

- The Need for an FTP Guard
- The Basic Design
- Phased Approach
  - Phase I - Keep it Simple / No Privileged Code
  - Phase II - More Capability
  - Phase III - Extend Concept to Web Proxy Server
- Project Status

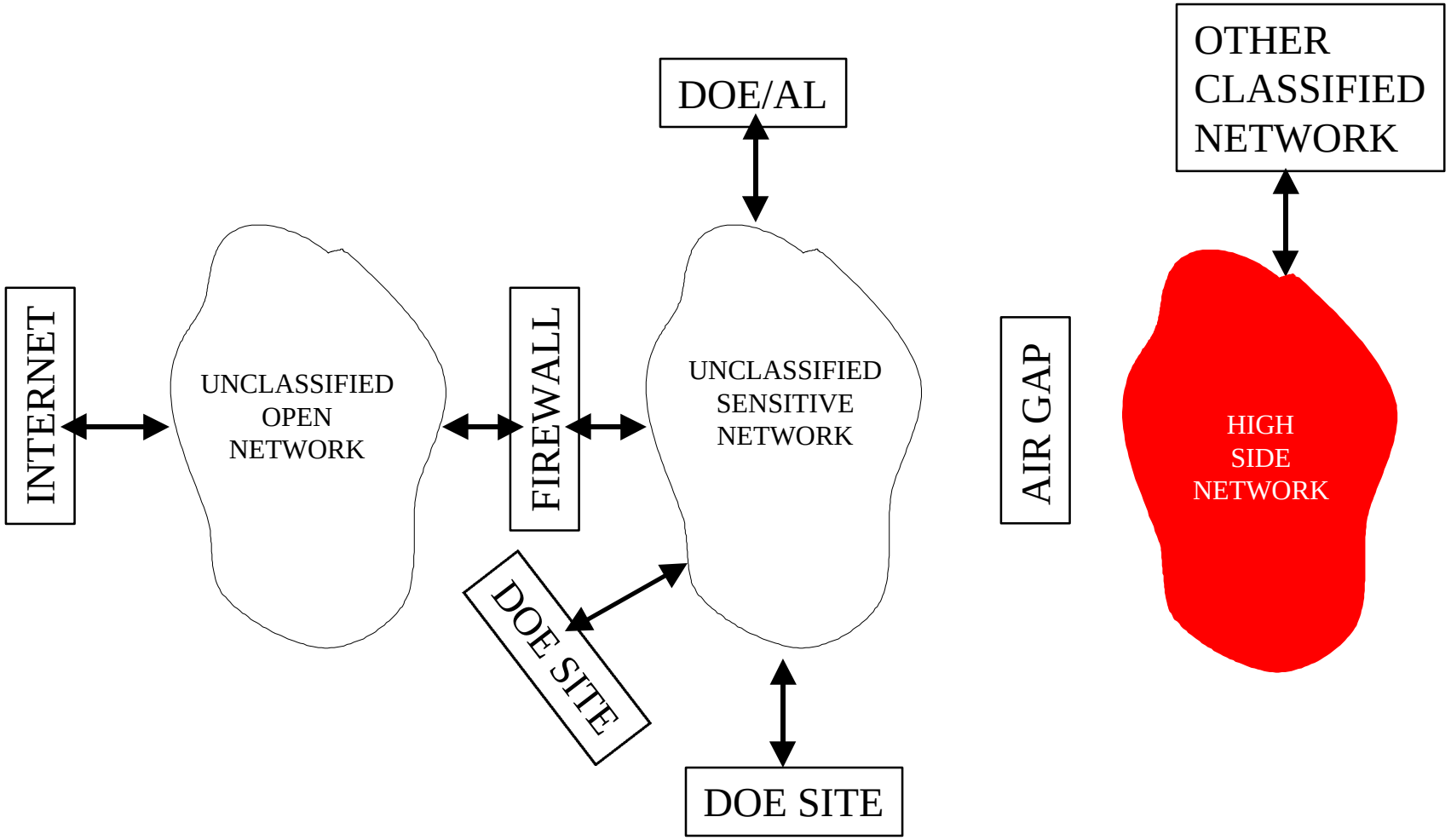
The Need for An FTP Guard

- Classified Designs use Many Unclassified Components
- Unclassified Files are Needed in Both the Classified and Unclassified Environments
- Some of the Choices
  - Duplicate the Files
    - Synchronization Problems
  - Keep them in the Unclassified Network
    - Off-line Transfer or Electronic Connection

The Solution: Electronic Access from Classified Systems to Unclassified Data

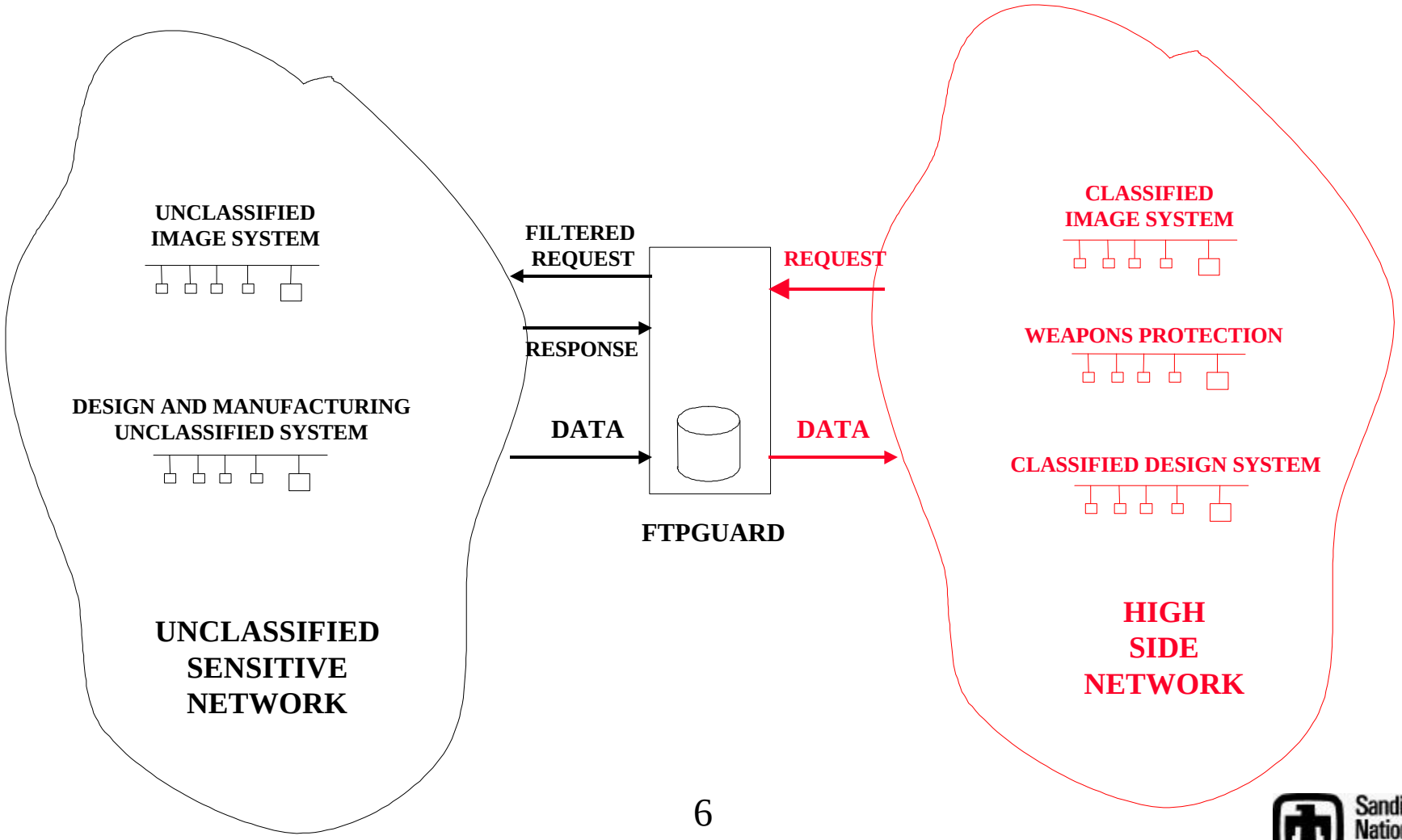
- Allows Master Copy to be Maintained in the Unclassified Environment
- Files can be Easily Moved to Synchronize Classified Copies with Unclassified Master
- Files Move Only from Unclassified to Classified

The Starting Point



Sandia National Laboratories

The Destination



Sandia National Laboratories

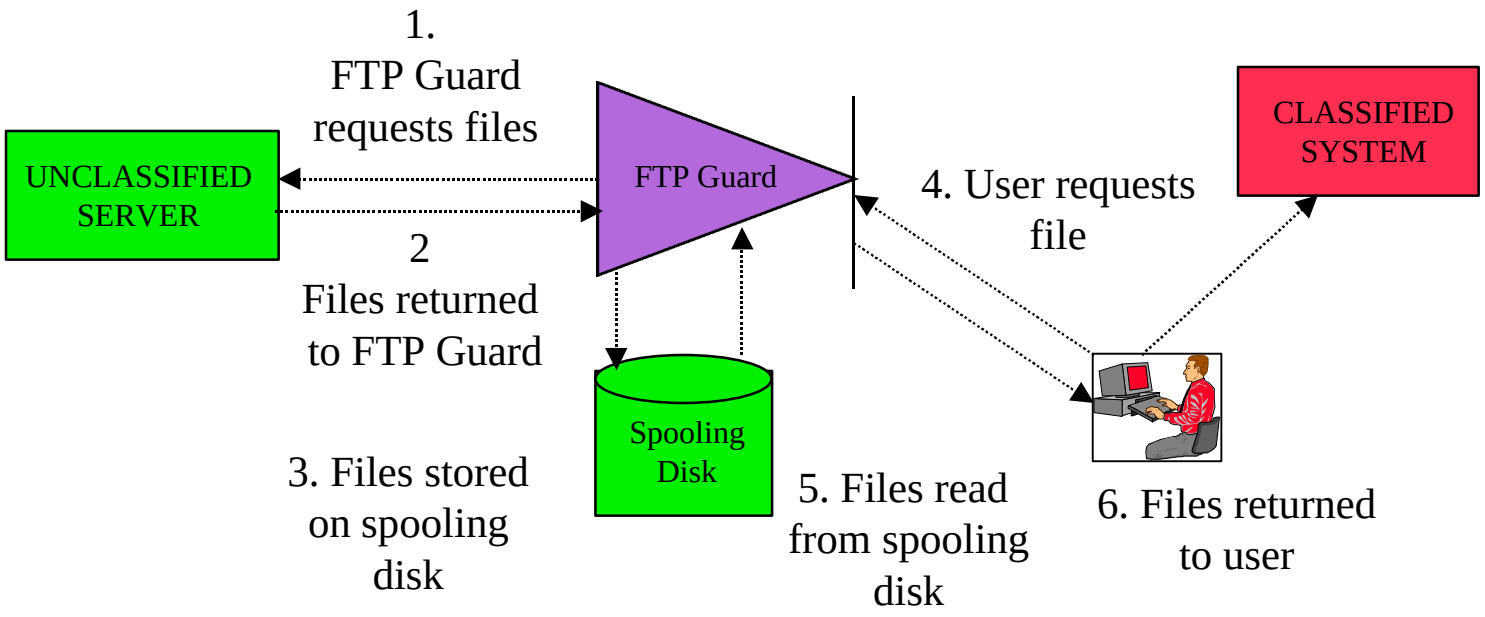
Characteristics of the Guard

- **Works with Standard, Unmodified FTP Client from a Classified (ISN) System**
- **To the Classified User, it Looks Like they are Talking to the Unclassified Server (They never really are)**
- **No Action on the Classified Side Causes a Visible Effect on the Unclassified Side (Phase I)**
  - **(No Covert Channels added in Phase I)**

Phase I: Pre-Spooled Files

- Time-Driven Job on Guard (cronjob) Causes Low_Side Files to be Spooled to the Guard
- High-Side FTP Client Gets Spooled Files
- No privileged processes: Every Process is Bound by the Usual Rules
  - No read up
  - No write down
- Garbage Collector Removes Spooled Files Based on Age (whether retrieved or not)

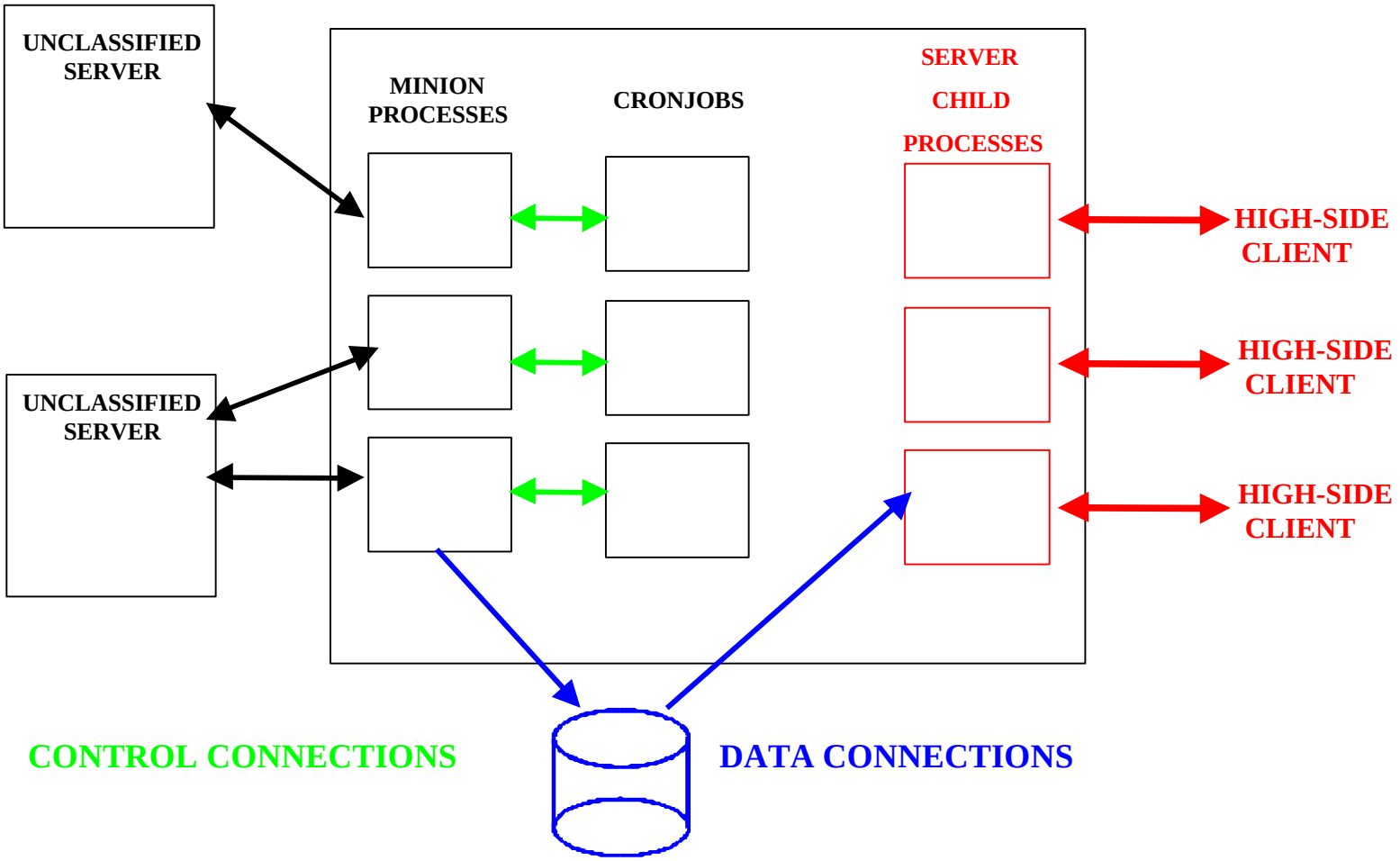
Phase I: Periodic Staging of Selected Files



Phase 1: Files must be staged from unclassified system to FTP Guard spooling disk (by prior arrangement) before becoming accessible to classified user.

Note: Bandwidth is limited by size of the spooling disk

Phase I Processes

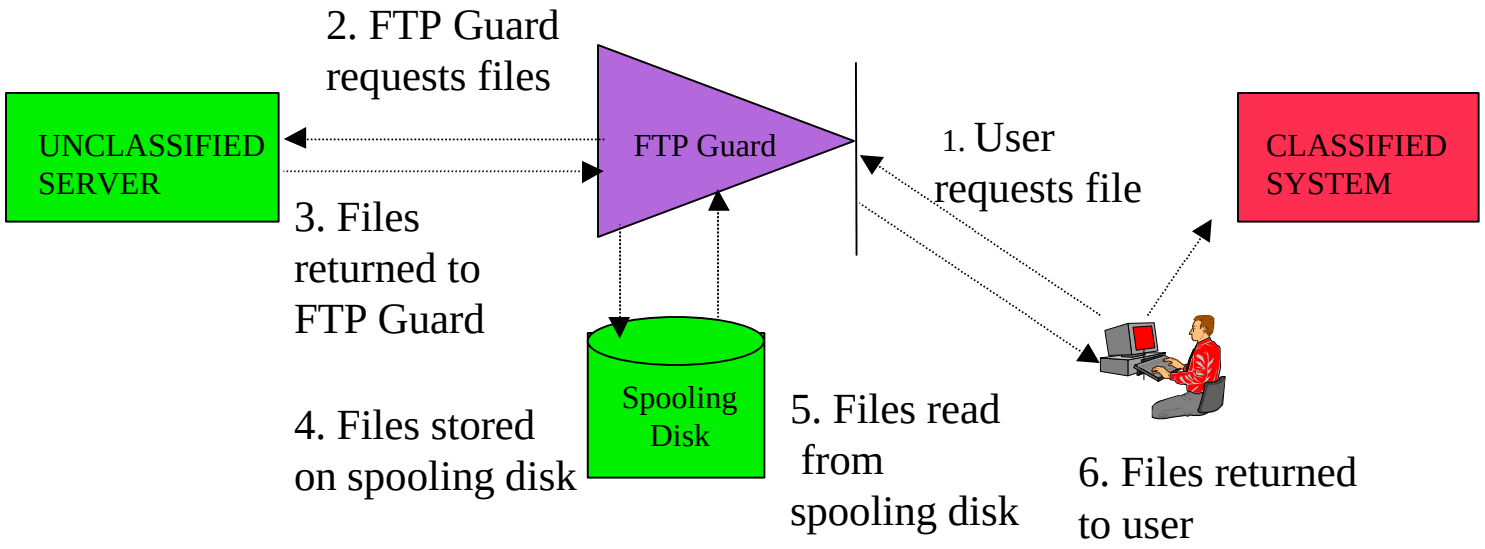


Sandia National Laboratories

Phase II: Directed Transfer

- Everything in Phase I plus
- Classified Users with SecurID Cards can Ask for Unspooled Data
- A High-Side Process must be allowed to Send Messages to a Low-Side Process
- Only Unclassified Information is Contained in Requests
- How do we know?
  - later

Phase II: Access to Files On Demand



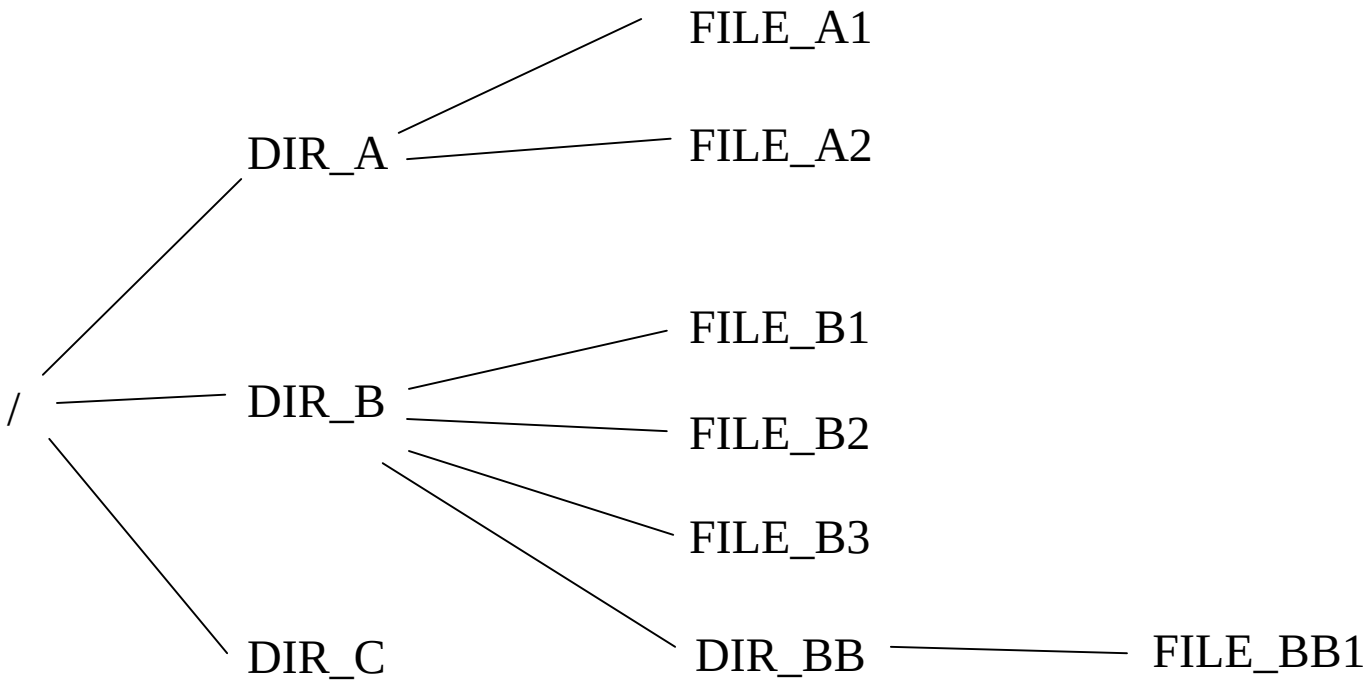
Phase 2: Files staged from unclassified system to FTP Guard spooling disk and returned to classified user on demand.

Phase 3: Web Access will be similar to Phase 2, except the HTTP protocol will be used.

No Classified Data Goes Out

- **High Processes Make Choices from Unclassified Objects**
- **No Process Sends an Arbitrary String from the Classified to the Unclassified Side**
- **Unclassified Data Moves to the Classified Side, but Only by Request (from the Classified Side)**

FILE SYSTEM ON REMOTE (BLACK) SYSTEM SHOWN BELOW.
{NOT YET VISIBLE TO RED USER}

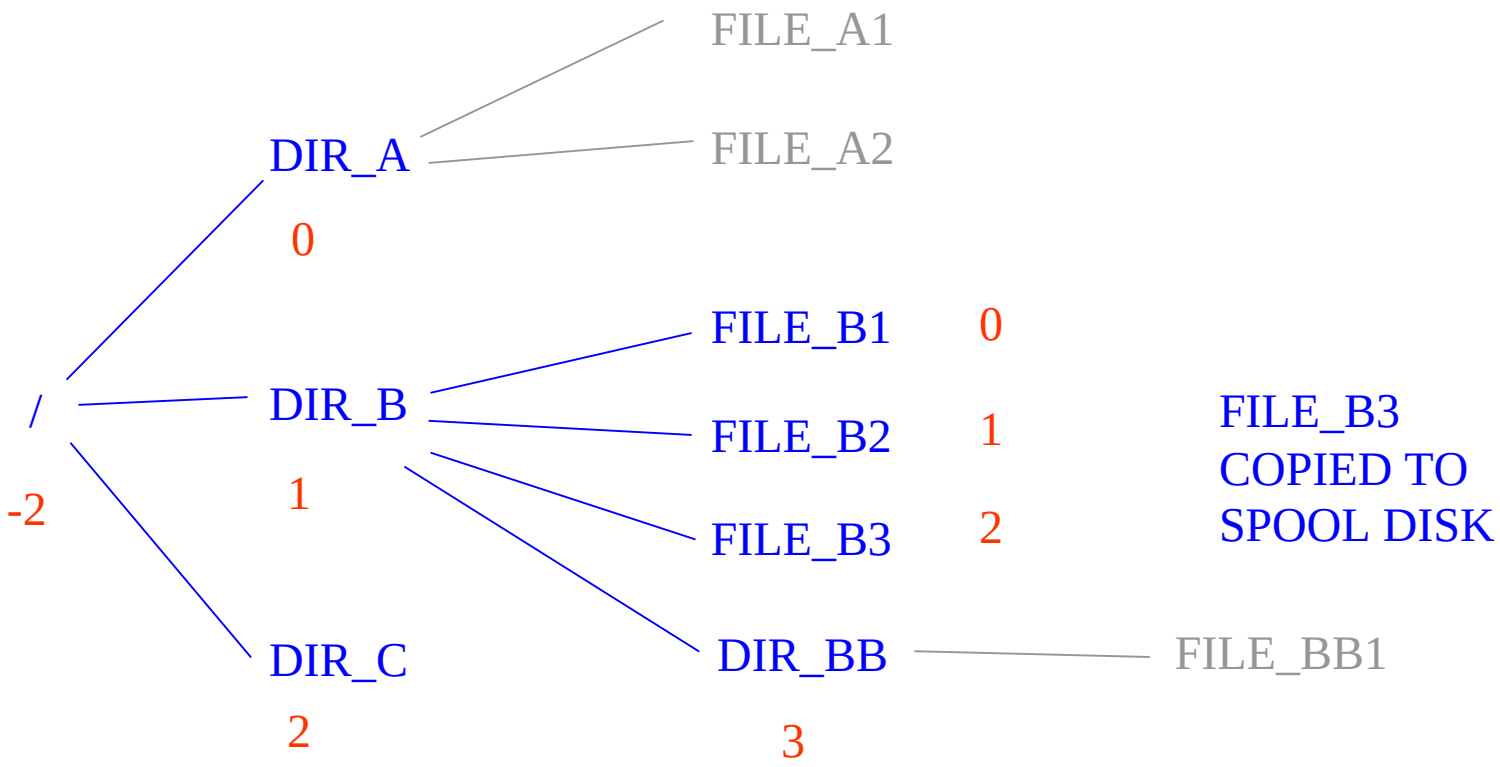


USER COMMAND: GET /DIR_B/FILE_B3

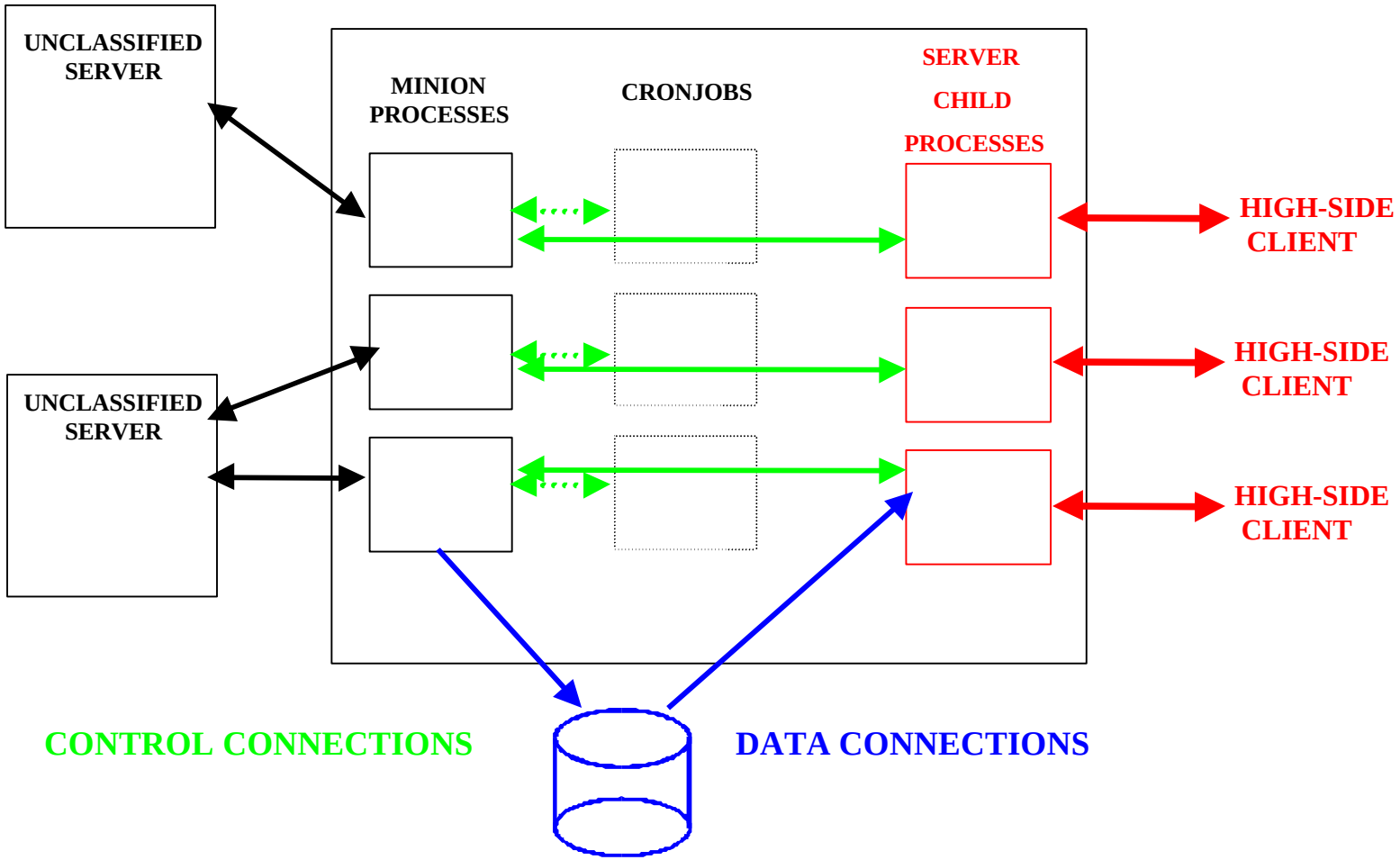
COMMAND SENT: CD -2 {CHANGE TO ROOT & LIST}

CD 1 {CHANGE TO DIR_B & LIST}

GET 2 {GET FILE_B3}



Phase II Processes



Sandia National Laboratories

Possible Phase III

- **WWW Proxy Server between Red and Black**
  - **Guard Begins with a List of Valid URLs (e.g., Certain Home Pages)**
  - **Any URL on a Page Fetched by a User is Added to the List for that User**
  - **Since the Only Strings that can be Sent Out have come from Unclassified Sources, no Chance of Accidentally Sending Classified Info**
  - **Requires Filtering of Incoming Files to Eliminate Executable Content (e.g., Java, Postscript)**

Project Status

- **Phase I**
  - **Programming Complete**
  - **IV&V Favorable, Security and Test Plans Approved**
  - **Tests conducted in November/December 97**
  - **Interim accreditation Dec 97 to Feb 98**
  - **DOE requested NCSC status on XTS-300**
  - **Second Interim Accreditation July 98**
- **Phase II**
  - **Programming to Finish Sep 30, 1998**
  - **Second IV&V Needed**