Multilevel Windows on a Single-level Terminal†

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Outboard from the IX system described in a companion paper, ‘‘Multilevel security with fewer fetters,’’ are ‘‘intelligent’’ terminals that contain a local operating system to support multiple windows and downloaded programs, all without benefit of memory management hardware. A program in the host mediates between (multiple) shell sessions and the terminal. To run multilevel windows, the host program needs to run as a privileged program, keep track of labels, and monitor the trustedness of the terminal. Very small changes in the terminal program enforce mandatory security policy.

Mux, the manager of *layers,* or windows, for Teletype 5620 and related terminals,[1, 2] poses difficult security problems. In principle, each layer on a terminal behaves as a separate virtual terminal serving its own shell and associated process group. To get the most out of the terminal, we wish to run each layer with a separate label. Hence data transfers among layers must obey the formal security policy. This is difficult because layers are mutually accessible. It is possible to copy data between layers. Worse, it is possible to download arbitrary programs into layers, and layers enjoy no hardware protection.

Mux is implemented by a pair of programs, a host part that multiplexes data transfers to the terminal and a downloaded terminal part—a multiprocess operating system in its own right. The host part multiplexes bidirectional traffic to all layers. Since it must deal with layers that have different labels, the host part must be trusted, with capability *Tnochk*. The host part deals with the process group of a layer through a pipe, which the process group sees as a terminal. The pipe obeys the same labeling discipline as would a terminal; its label is marked rigid and can be changed only by trusted processes with T_{mount} capability. To detect label changes the host process enables signal `SIGLAB`. It accepts all changes, secure in the knowledge they they must have been made by trusted processes. Each change is relayed to the appropriate layer. Upon a downward label change, the layer is reset to expunge all extant data; in effect the process group gets a new layer.

The terminal part knows only enough about labels to prevent leaks. It does not implement the full dynamic label scheme. Labels are checked on every attempt to cut and paste data between layers; attempts to copy downward are ignored. As an extra precaution in the face of a shared address space, the terminal part erases all storage as it becomes free, including screen bitmaps, downloaded programs, and displayed text. No logging of actions at the terminal is provided, however.

Programs to be downloaded into layers run in the native hardware of the 5620 and have access to the entire address space of the terminal. Hence code run in the presence of multiple labels must be trusted. An untrusted program may be countenanced only if its label is identical to the label of all data in the terminal; otherwise it could write down to or read down from other processes. Moreover an untrusted program cannot be loaded into the terminal while any private path reaches the terminal. Since nothing can prevent untrusted code from modifying the terminal part of *mux* itself, the terminal, which becomes untrusted as

---

† An earlier version of this paper appeared in *Proceedings UNIX Security Workshop,* Usenix Association, Portland, August 1988, 24-31.

soon as untrusted code is downloaded into it, must remain untrusted forever – or until rebooted. The labels stick at the untrusted value. In practice we are more stringent and require all labels in an untrusted terminal to have the same value as the the initial label of the terminal.

To separate concerns, *mux* was designed so that downloading would be done through it, not by it. Yet, to trust the terminal, *mux* must assess the trustability of downloaded code. Downloads into layers are handled by a trusted specialist program, *32ld,* that passes data through *mux. Mux* ascertains the trustedness of the downloader program and its connection to *mux.* The downloader is expected in turn to determine the trustability of downloaded code. Since the main pipe between *32ld* and *mux* is shared by a shell and perhaps by other processes, we protect communications over that pipe by using pex to prevent other processes from using the pipe until its status reverts. The downloader is deemed trustworthy if it has capability T_{mount}, which it relinquishes when downloading an untrusted file. *Mux* observes the trustedness by examining indicia of privilege that come with pex. Unless the downloader is trusted, *mux* marks the terminal untrusted.

A legitimate trusted download ends with a special coda from *32ld* that must be received while the pipe is marked for exclusive use. If an imposter kills *32ld* in mid-download the download pipe becomes unusable: the *mux* end is marked for exclusive use, while the other end reverts to permissive use. This state prevents all IO activity, in particular, attempts by the imposter to forge a download or to reassert process-exclusive access. *Mux* detects the change in state with a failed *read* system call and deletes the layer. The terminal remains trusted.

The standard version of *mux* depends on *32ld* to download the terminal part before the host part begins. We have abandoned this arrangement, which made it difficult to assure the host part that it is indeed talking to the correct terminal part. Instead, we let *mux* do that download itself, again protecting its access to the terminal with pex. In fact, process-exclusive access is retained through the whole *mux* session. This curious division of labor, wherein downloading into the raw terminal and into layers is done by different agents, is marginally justifiable: existing programs that need to load into layers already know about *32ld,* and the protocols differ, one being in hardware and one in software.

Mux also uses pex to provide a private path between user terminal and application program. While the terminal is trusted, a trusted user process running in a layer may issue a pex call on its pipe to *mux.* As in the special case of *32ld* sketched above, *mux* detects the new process-exclusive status of the pipe together with indicia of the privilege of the process that issued the call. It notifies the terminal part, which in turn gives the layer a distinctive visual mark. The user then knows that new data typed in that layer are not accessible by eavesdropping programs. This private path mechanism can be used for confidential negotiations, such as password collection, that should not pass through an untrusted terminal.

Various flaws remain.

First, the 5620 itself might be subverted by downloading a terminal simulator that could receive and corrupt a later download of *mux.* This is impossible to do without giving a visual indication on the screen, but it could happen while nobody was looking. One way to detect such a gambit would be to download a program that fills memory with hard-to-compute numbers, then answers inquiries about the contents of randomly chosen locations. If it doesn't answer fast enough, the memory may be suspected of harboring other, unwanted, code. Another countermeasure would be to modify the hardware to provide an out-of-band channel from the host to the native boot program. We have taken neither precaution, counting instead on users avoiding the trap by booting the terminal afresh just before starting *mux.* Aside from its dependence on human behavior, this procedure is sound: if a phony *mux* were downloaded instead, it would not be trusted, so multilevel data could not be accessed.

Second, under *mux,* the terminal practically becomes an extension of the operating system. For genuine safety, the physical security arrangements for the host computer should be extended to every *mux* terminal, and especially to the terminal's ROM.

Third, the private path for session-authorizing negotiations outlined above, will not work with an untrusted terminal. This is only a special case of a more general question: how to perform an authorizing negotiation over any external medium, the trustedness of which has not been established, and may be unnecessary for the session that follows. Challenge boxes, which accomplish confidential negotiations in the presence of eavesdroppers, are the preferred solution.

REFERENCES

[1] Pike, R., ‘‘The Blit: a multiplexed graphics terminal,’’ *Bell Laboratories Tech. J.* **63**, pp. 1607-1631 (1984).

[2] AT&T Bell Laboratories Computing Science Research Center, *UNIX Research System Programmer’s Manual*, Vol. 1, Saunders, Philadelphia (1990).