Secure IX Network†

Jim Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper sketches a design for a network of computers running the McIlroy-Reeds IX system. The emphasis is on modularity and decentralization; security does not rely much on central key distribution. It assumes that there are multiple overlapping domains of authority, and relies only loosely on an ultimate common organizational loyalty. This work is speculative. It is heavily influenced by the networking arrangements in the Research 10th Edition UNIX® system.

1. Single Machine IX*

When we began working on IX M. D. McIlroy and I simply wanted to add to the usual UNIX system model a stricter file access policy, a variant of the military security classification system. We believed that a no-frills implementation of this part of the Orange Book[DOD] requirements for a secure computer would satisfy most real security needs of most users, even if it might NOT satisfy the Orange Book people themselves. We also believed that our new access restrictions could be kept largely orthogonal to the existing scheme of `rwx` bits, which we left essentially intact. The new access policy could be airtight and draconian in its preservation of security labels (and the intellectual property rights they represent), even if on the same machine, at the same time, the usual UNIX system concepts of userid, setuid, root accounts, and `rwx` file permission bits were subverted.

After three years of part-time but intense work we have not changed those beliefs, but we have expanded our notion of what needs to go into a ‘‘no-frills implementation.’’ Our resulting system, like all other attempts at secure operating systems, ends up with a two-tier structure. There is the generality of user programs, including all programs written by ordinary users and most of the programs in the public program directories. And then there are the privileged programs used to administer the security system, covering functions like logging users in, changing user’s security clearances, and so on.

The distinguishing property is that the privileged programs must break the usual security rules, or viewed another way, that the kernel must be assured by a privileged program that some particular violation of the usual rules is not in fact a breach of security. Files, for instance, are not usually allowed to drop in security classification level. Terminal ports are files in the UNIX system, so when a terminal port used in a classified login session becomes free at the end of the session, before it may be used for a new, unclassified, login session it must be declassified by a privileged program. Ordinary disk files might be declassified from time to time for the usual reasons; this again needs a privileged program. The kernel cannot itself know when such actions are acceptable: it relies on the privileged programs to tell when to deviate from its worst-case application of the usual rules.

† Reprinted from DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 2, *Distributed Computing and Cryptography,* J. Feigenbaum and M. Merritt, editors, pp. 235-244, by permission of the American Mathematical Society.

* Some details of single-machine IX given below are, for exposition’s sake, simplified. The true description is in the accompanying papers. IX is *not* pronounced like the German word *ich.*

Trusted Computing Base

The presence of these privileged programs complicates things enormously. Obviously questions like ‘‘how do I know that a baddie hasn’t tampered with a privileged program,’’ or ‘‘how do programs become privileged,’’ or ‘‘how do new releases of privileged programs get installed on the system’’ are relevant. The easy answers (essentially the same in all secure operating systems) are: Giving or removing privileged status to files can only be done by a privileged program. A privileged program can not be removed or changed, other than to have its privileged status removed. And the program that first runs at boot time, *init*, is privileged. The privileged programs thus form a self-nominating closed ‘‘committee’’ which, together with the kernel as an *ex-officio* member, runs the computer. The jargon for this committee is TCB: trusted computing base.

This in turn leads to the hard questions. How does the TCB know which information from the outside world it should act on, and which is spurious? How does the computer user know he is talking to the TCB and not an imposter? How does one privileged program communicate with another, in such a way that each recognizes the other’s privilege? IX’s answers to these questions are based on two mechanisms, one in the kernel, the other coded into the privileged programs, both of which seem easily extendible over a network. Hence the TCB’s iron reign can be extended over a network, and hence so can IX security services.

PEX

The first of these wonder mechanisms is a form of mandatory exclusive file locking, which we call PEX for ‘‘process exclusive.’’ Any file may be marked for exclusive use by a process in which it is open, so long as no other process has done so first. The exclusivity lasts until the process explicitly drops it or until the file is closed or the process exits. A special twist occurs in pipes, each end of which may be independently PEXed. A pipe will not carry data unless both ends are PEXed or neither end is PEXed: an attempt to read or write on a half-PEXed pipe result in an error. When a pipe end is PEXed the other end is given indicia of privilege of the PEXing process. All this is done synchronously and independently of the usual stream[R] discipline. Thus a privileged process communicating on a pipe can tell if it has a unique partner process at the other end, and whether the partner process is privileged. A privileged program, for example, can refuse to collect a password unless it can PEX the standard input. If talking to an IO device unPEXed to any other process, the PEX call succeeds and no interloping process that also has the device open can steal the password. If talking to a pipe (say, to a window on a windowing terminal) the PEX bid will be accepted or rejected by the process at the other end of the pipe. If accepted, the privileged program can now tell if the other end is also in the TCB (as a trusted terminal multiplexor running secure windows would have to be), in which case the password dialogue can go forward, and so on. Using these means parts of the TCB can recognize each other, can know when they are talking to users, and —although the details are clumsy— can prove to users that they are indeed talking to the TCB.

This can be thought of as a generalization of the Orange Book’s ‘‘trusted path’’ mechanism, and as an alternative to Biba-style ‘‘inverse labels’’. (Biba inverse labels are described in pages 69-71 of Gasser[G]; the labels track trustability or provanance of data rather than secrecy.)

The notion of a PEXed pipe is nothing other than that of a secure communications channel between identified parties, an end goal of both classical and modern cryptology and of secure network design. The novelty is in making this service available to user processes, protecting themselves from eavesdropping, forging or spoofing by other processes. In its current single-machine form, the writ of PEX ends at the machine boundary, and cryptography is not needed to implement this new operating system feature. (New to the UNIX operating system, at least.) For PEX to go over a network to another machine would require at least a special protocol for the kernels to use in discussing the PEXity of their ends of the cross-machine pipe, and also possibly cryptography to identify and authenticate the end machines and to protect the data in transit.

Privilege Server

The second special mechanism for privileged process control in IX is a *privilege server* embodying a particular design for denoting, recording, and exercising users' rights. The main idea is that although the TCB is all-powerful, it doesn't have any security interests of its own, save self-protection and economic control of the local hardware resources. The TCB is a custodian of users' rights, which might originate off-machine, possibly created under an authority separate from the management or ownership of the local host computer. In IX these fiduciary responsibilities of the TCB are managed by the privilege server.

The privilege server centralizes *authorization* calculations, much the way an authentication server centralizes authentication services. Authorization calculations often take a stylized form: a resource's owner passes a credential to a user, who later presents the credential together with a particular resource usage request to the custodian of the resource. In IX the privilege server is the custodian of most users' rights (other than routine file access rights implied by user id and process label). For each request, the privilege server must check that the credential indeed authorizes the request, that the privilege server itself is authorized to act on the request (which includes checking whether the resource in question has been entrusted to the privilege server), as well as authentication information, such as whether the issuer of the certificate is the owner of the resource, and so on.

Requests for privileged services are made to the privilege server as text strings, which are also statements in a trivial computer command language, *nosh*. (This is a restricted, feature-starved ''secure'' shell with fewer possibilities for latent subversive side effects than found in the usual interactive shells *sh*, *csh*, or *ksh*.) If the requester has a right warranting the command string in question then the privilege server executes the command. Corresponding to each right, then, is the set of *nosh* command strings it warrants.

More precisely, a ''right'' is a regular language. Rights are recorded in a privileges file as pairs (R, Q), where R is a regular expression* (specifying the set of *nosh* commands warranted by the right) and where Q is a predicate applied to: the user, the execution environment of the current invocation of the privilege server, the status of the user's connection to the server, and (optionally) a protocol execution status. When a request s is presented to the server, it is executed only if a pair (R, Q) can be found such that s is an element of R and such that the predicate Q evaluates true. Thus ''anyone'' who satisfies Q may exercise R.

For ease of subdivision and delegation, we organize our privilege file as a rooted tree. Node (R_1, Q_1) may be above (closer to the root than) node (R_2, Q_2) only if language R_2 is contained in R_1. Thus sub-nodes correspond to sub-rights, and if you can exercise a right at a given node you automatically may do anything you could do by exercising rights at sub-nodes.

Our command language has statements to edit the privileges file, so it is possible to formulate rights to change rights. In particular, it is easy to formulate a right to edit only a sub-tree of the tree of rights. Allowed edit commands are: diminish the set of rights at a node, delete a node or sub-tree, create a sub-node with rights equal to the parent node, and change the Q part of a node. Thus editing can only reparcel or dilute existing rights, but not create new ones. (Since privilege file edits are relatively infrequent there is no practical performance loss in forbidding concurrent edits, or forbidding privilege calculations during edits.)

To delegate a right means simply to create a sub-node in the tree of rights.

The Q predicate might require given login ids, ask for a password, require successful completion of a challenge-response protocol, or insist that the privilege server is being invoked by a particular named program. Q specifies the *authentication* needed to enjoy the rights in question.

A weak point in this scheme lies in the formulation of rights as regular expressions: extreme care must be taken to ensure that the *nosh* command language statements accurately catch the real meaning of the rights in question. In particular, the manual describing the TCB and the *nosh* language must be accurate, and worse, the meaning of command-line arguments and options to privileged programs may never be lightly changed, lest established rights be overthrown or inadvertently widened.

---

*Abuse of notation to follow: I make no notational distinction between a regular expression R and its corresponding language, so sometimes R really means $L(R)$.

An Example

Here is how the privilege server helps administer a security classification compartment. Suppose some users of an IX machine start a new project and want to create a security compartment to guard their work on the computer. This new security compartment shows up outside the computer as a new classification keyword stamped on documents and inside the computer as a new bit field in the security labels carried by files and processes. Say project leader Alpha wants to create a compartment `BETA` for his project; what steps does it take to teach the TCB about this new compartment?

To begin with the TCB's only interest in `BETA` is in issues of operating system resource exhaustion: is the keyword `BETA` already in use for some other security compartment and is there an unused label bit field on the local machine to represent `BETA`? The usual solutions to these problems apply: categories might have hierarchical names (keywords like `ADONIS.PICCOLO.BETA` are less likely to be preempted), and only certain users may consume label bit fields by creating new security classifications. So among the rights already stored in the privileges file is an entry (R_{mkcat}, Q_{mkcat}) giving project leaders the right to run the privileged *mkcategory* program:

$$R_{\text{mkcat}} = \texttt{mkcategory .*}$$

$$Q_{\text{mkcat}} = login\_id \in \{Alpha,\ Rocky,\ Boris,\ Natasha \cdots \}.$$

So Alpha presents the request

```
mkcategory BETA
```

to the privilege server, which runs the *mkcategory* program in such a way that *mkcategory* knows this invocation is an authorized invocation. Note that so far the privilege server has only been guarding the local machine's operators' interest in preventing resource exhaustion, and not any security interest *per se*.

Mkcategory has two functions. One is to allot a bit field in the machine's internal representation of security labels and bind the name `BETA` to it, registering this in an official list. The other function is to register Alpha as the ''owner'' of `BETA` by placing a new entry $\beta = (R_\beta, Q_\beta)$ in the privileges file. (The *mkcategory* program itself has the right to edit part of the tree of rights, according to a $Q_{\text{mkcat2}} = invokerprog \equiv \texttt{mkcategory}$.) To begin with *mkcategory* finds out from Alpha what formula to use for Q_β, that is, how future commands from Alpha about `BETA` will be recognized. Alpha may provide any formula for Q_β he wishes: he may simply specify a password, or may specify that possession of Alpha's *login_id* suffices, may present the public key by which future commands may be verified, *a la* RSA, and so on. The *R* part of the new right is always the same for a newly created category, and expresses the union of these rights:

> Exerciser may access data marked `BETA` by logging on with the appropriate bit set in his process label.
>
> Exerciser may declassify category `BETA` from files, by executing a privileged command of form `downgrade BETA .*` which clears the `BETA` bit from the named file.
>
> Exerciser may specify network addresses that may receive `BETA` data.
>
> Exerciser may edit this node β in the tree of rights.

Now project leader Alpha has a private security classification label all of his own. He can make classified login sessions, he can create secret files readable by no one else, he can declassify them.

Soon however Alpha wishes his assistants Mr. Gamma and Ms. Delta could access his secret files, too. He invokes his right to edit node β (by presenting whatever credentials he earlier spelled out in Q_β) and creates a sub-node, call it β*/access*, whose *R* part consists solely of the exerciser's right to read and write data marked `BETA`. The *Q* part is again at Alpha's discretion, who (say) chooses the formula

$$Q_{\beta/\text{access}} = login\_id \in \{Gamma,\ Delta\}\ \&\ shows\_password(\texttt{cyto97plasm}).$$

(Alpha need not add himself to the list in $Q_{\beta/\text{access}}$ because he already may exercise the superior right β, but might want to do so because for simple file access $Q_{\beta/\text{access}}$ might be easier to use than Q_β.) Note that Alpha has extended only his access rights to Gamma and Delta: as desired, Gamma and Delta can read and write `BETA` secrets, but they cannot change the list of people so cleared.

The project prospers, and when new assistants Eps through Lamb show up boss-man Alpha tires of editing β*/access*. So he appoints Ms. Delta as his BETA-clearance officer, by creating a new node β*/clearance*. $Q_{\beta/\mathrm{clearance}}$ only lets Delta in. $R_{\beta/\mathrm{clearance}}$ only allows the exerciser to edit the predicate $Q_{\beta/\mathrm{access}}$.

At appropriation time some good press coverage is needed, so Alpha appoints Gamma his publicity manager, whose job of course includes shrewdly sequenced leaks of BETA data, which means a new node β*/declass* is formed, with a $Q_{\beta/\mathrm{declass}}$ for Mr. Gamma alone, and with $R_{\beta/\mathrm{declass}}$ allowing BETA-downgrade rights to the exerciser.

What would it take to extend these activities to another computer? Obviously at set-up time the remote file servers need to reconcile their differing internal representations of file security labels: it is almost guaranteed that the bit slot allotted to represent category BETA will differ on all machines. But before BETA files may be traded the two machines should really check to make sure that what each knows as category BETA is really Alpha's BETA, by trading and verifying credentials signed by Alpha, according to the recipe spelled out in Q_{β}.

2. Networked IX

The general shape of a network of IX machines is clear. If a pair of machines trust their network connection, and trust each other's TCBs to enforce the same basic security policies, then they may extend PEX service and remote file system service as sketched above. A minimal version might consist of the following ingredients:

- A LAN offers secure networking within the confines of a single department.
- The individual IX machines are run by the same systems staff, who run identical software on the machines, so any of the machines can trust any other's TCB as much as it trusts its own
- Terminals are at known network locations at known physical locations

Within such a network different machines can carry varying mixes of secret data: some label categories may exist on all machines, other categories may exist on single machines only, other categories yet may exist on other subsets of machines. The user communities may be heterogeneous: not all users may have accounts on all machines, nor need all users of a given machine have access to the same security categories. Such a setup might be appropriate in a small department, or in a rigidly run branch of the government.

A more ambitious network might include subnets of the above description but would also have insecure network links, a variety of terminals in unknown or uncontrolled locations, a variety of computers with differing software, some of which are in isolated secure physical locations. Such a network would need a measure of cryptographic boost to give untappable connections, or connections to known endpoints. While it is unreasonable to hope that the computers all run exactly the same system software there is still a chance that their security software is the same. If two machines each believe the other's TCB there is a possibility that they can trade security services. To prove that partner machines have correct TCBs some appeal to authority is needed. Depending on details, machine A might trust B if B is at a known network address, or if B has a document attesting B's honesty, signed by an authority A trusts, or (what is really a variation of this) if B can communicate at all with A when A uses cryptographic keys it was provided with by a trustworthy authority.

In an ambitious network user terminals and user authentication pose a real problem: there is no general way for a network or a computer to tell the difference between a terminal and a work station or heftier computer. Any command purporting to come from a user at a terminal might really come from a hostile computer, which can play the IX protocols for dishonest purposes. Terminals on the network are thus treated as computers: unless they are at certain special secure network addresses on a trusted network, they must prove their *bona fides* before they are given PEX service, and hence before they may be used to get favors from the privilege server.

Little special software is needed for networked IX given single-machine IX and given regular UNIX system networking facilities. As hinted earlier, a protocol for the TCBs to manage cross-machine PEX would be needed, possibly as a stream[R] module. And to handle remote file systems, for example, a form of ''power of attorney'' protocol is needed, by which one machine can prove to another that it has authority from a third party known to both. To be usable there must be user software to make it convenient to create (say) RSA keys on demand, and so on.

3. Special Terminals and User Authentication Hardware

In a network of IX machines, terminals at unknown or uncontrolled network locations must be treated as potentially hostile computers, which of course makes logging in, authenticating users, and reading secret data over a terminal difficult. Here is a possible solution to these problems, relying on special hardware, prompted by our experience with windowing terminals in single-machine IX

The main idea is to use special purpose secure computers as terminals, together with smart-card-like user identification tokens which are also special purpose secure computers.

The terminal and screen software is part of the TCB. The TCB is unprogrammable (in ROM, say) and contains secret cryptographic keys in an uninspectable store; the whole packaged in a tamper-proof container. The TCB can enforce a primitive form of IX file access policy (each multiplexed terminal process and associated windows, say, has a label, with label inequalities obeyed on mouse-initiated cross-window copies or ‘‘snarfs’’). Also, a form of PEX is available: if a screen layer is accessible by exactly one terminal process, and if the keyboard is accessible only by that terminal process, then a distinctive unforgeable visual mark may be placed on the screen layer (a flashing border, say).

The ‘‘user authentication tokens’’ are small special purpose computers with their own TCBs and tamper-proof memory. A token can be plugged into a special terminal, and has a light visible to the user when the token is plugged into the terminal.

This hardware is used to help several authentications. There are as many as four parties with differing security interests: user, user token, terminal, and host computer. The user token must be convinced that its legitimate user is present. The terminal and host must like each other’s TCBs enough to set up cross-machine PEX. The host must communicate through the terminal to the user token, but the terminal will not play PEX unless it trusts the token, and so on.

Most of these security interests are satisfied by multiple applications of, say, the Fiat-Shamir[FFS] protocol, where the terminal, host, and token successively play differing roles. The FS protocol has two roles: the ‘‘prover’’, who has a certificate issued by an authority, and a ‘‘verifier’’ who checks the certificate without—and this is the big trick—actually seeing it. The authority knows the factorization of a modulus N. The modulus N, but not its factorization, is known ahead of time by the verifier. What the verifier knows at the end of the protocol is that whoever prepared the certificate knew the factorization of N.

The terminals and tokens carry certificates from their manufacturers, signed according to a modulus N_{term}, characteristic of the brand name of the terminal. This modulus N_{term} is known by all the host computers. This prior distribution of public authenticating key is not a burden because the number of distinct brands of secure terminals will remain small.

User identification tokens also carry certificates attesting to the identity of their owners. These certificates are signed with a modulus N_{dom} characteristic of the security domain in which the user lives. There is nothing wrong with a user identification token carrying several such certificates valid in different domains. Computers also have one or more certificates signed by the same moduli N_{dom}. The assumption is that the same authorities that certify users are able to certify computers. Roughly, if you have an account on a machine, there is at least one N_{dom} you and the machine have in common.

Here is how to set up a login session with such equipment. First, the token and terminal authenticate each other with respect to modulus N_{term}. They tell the user that they like each other by visual signals: the light on the token is lit, a special icon or message is displayed on the terminal. Then the token has a chance to demand a password from the user, typed via the keyboard of the now-trusted terminal. Then the token tells its N_{dom} values to the terminal. Terminal and host computer negotiate to find a common N_{dom}, then the computer proves its *bona fides* to the terminal using modulus N_{dom} and the terminal proves to the computer that it is a special terminal with modulus N_{term}. The computer and terminal now set up the cross-machine PEX protocol on their link, and the user proves his identity to the computer either by typing a classical password or by letting his token prove the user’s credentials using N_{dom}.

A simplification is possible in the case where there is only one security domain which knows about all trustable computers. Then the user token is not needed: the single modulus N_{term} can suffice for all applications of Fiat Shamir.

Conclusion

The forgoing describes a general architecture for a network of secure computers, offering far more security than is commonly found in the UNIX system, offering users a measure of distributed security services with a minimal overhead of centralized bureaucracy.

How large an organization could such a net serve, before the methods of user authentication, delegation of rights and authorities, and system administration sketched above becomes too cumbersome? How much work would it take to build such a net, given the existing pieces?

I have had chats with Baldwin, Coutinho, Feigenbaum, Fernandez, Fraser, Grampp, Kurshan, McIlroy, Merritt, Pike, Presotto, Ritchie, Thompson, Wilson, Zempol, among others, to whom I am grateful for ideas and helpful criticism.

References

[DOD] Department of Defense Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria.* US Department of Defense, Fort Meade, MD, 15 August 1983.

[FFS] U. Feige, A. Fiat, and A. Shamir. ‘‘Zero-Knowledge Proofs of Identity’’, *Journal of Cryptology,* **1**:77-94, 1988.

[G] M. Gasser. *Building a secure computer system.* New York, Van Nostrand Reinhold, 1988.

[MR2] M. D. McIlroy and J. A. Reeds. ‘‘Multilevel Security with Fewer Fetters’’, in *UNIX Around the World: Proceedings of the Spring 1988 EUUG Conference.* European UNIX Users’ Group, London, 1988.

[MR3] M. D. McIlroy and J. A. Reeds. ‘‘Multilevel Windows on a Single-level Terminal’’ in *Proceedings, UNIX Security Workshop, August 29-30, 1988. Also in the present collection.* USENIX, Portland, OR.

[R] D. M. Ritchie. ‘‘A Stream Input-Output System’’, *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984.