Confining Root Programs with Domain and Type Enforcement (DTE)

Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger,
Michael J. Petkac, David L. Shermann, Karen A. Oostendorp

Confining Root Programs with Domain and Type Enforcement (DTE)

June 13, 1996 (2:57pm)

Kenneth M. Walker
Daniel F. Sterne
M. Lee Badger
Michael J. Petkac
David L. Sherman
Karen A. Oostendorp

0. Abstract

The pervasive use of the root privilege is a central problem for UNIX security because an attacker who subverts a single root program gains complete control over a computing system. Domain and type enforcement (DTE) is a strong, configurable operating system access control technology that can minimize the damage root programs can cause if subverted. DTE does this by preventing groups of root programs from accessing critical files in inappropriate access modes. This paper illustrates how a DTE-enhanced UNIX prototype, driven by simple, machine-interpretable DTE policies, can provide strong protection against specific classes of attacks by malicious programs that gain root privilege. We present a sequence of policy components that protect system binaries against Rootkit, a widely-used hacker toolkit, and protect password, system log, user, and device special files against other root-based attacks. Tradeoffs among DTE policy complexity, scope of protection, and other factors are discussed.[1] [2]

1. Introduction

The pervasive use of the all-powerful root privilege is one of the most important sources of UNIX security problems. An attacker who subverts a single root program can gain complete control over a system, including enterprise data, operating system components, cryptographic keys used for secure communications, and privileged network connections to other local hosts. Unfortunately, many daemons and other programs that execute with root privilege are complex and riddled with security vulnerabilities. For these reasons, root programs are extremely attractive targets for attackers. UNIX system lore is replete with examples of root programs being tricked into misusing their privileges for a variety of malicious purposes [6,8]. By providing and relying on a single all-powerful privilege, UNIX systems "put all their eggs in one basket."

Finding and fixing vulnerabilities in root programs is important; but it is unwise to assume that all such vulnerabilities can be found and fixed before attackers can exploit them. Domain and type enforcement (DTE) is an operating system access control technology offering a fundamentally different approach to this problem [2,12]. We assume that any UNIX system will include one or more root programs containing exploitable vulnerabilities. We then attempt to minimize the damage these programs can cause if subverted. Configuring an appropriate DTE access control policy causes many root programs to be executed in restrictive domains that only allow access appropriate to each program's assigned responsibilities, thereby curtailing unnecessary access, especially to security-critical files. In effect, this places the UNIX system eggs in separate baskets.

Previous papers on DTE have focused on the goals, design, and features of our DTE-enhanced UNIX prototype and have illustrated these via hypothetical policy examples [2,12]. This paper illustrates how DTE mechanisms driven by simple DTE policies can provide strong protection against specific classes of root-based attacks on UNIX. In particular, we present the core of a simple, experimentally validated policy that protects UNIX against Rootkit [16], an increasingly popular hacker toolkit that attempts to overwrite system binaries. Policy enhancements are then presented to demonstrate how specific vulnerabilities can be protected. In particular, we will

[1] Approved for Public Release - Distribution Unlimited (#96-S-2589)

[2] This research was supported by ARPA contract DABT63-92-C-0020.

protect various system log files, the password file, and several device special files (i.e., /dev/kmem). Finally, we will provide a policy extension that allows a user to run a web browser in a restricted domain. This domain will restrict the damage that can be caused by a browser (or a helper application) when interpreting malicious code.

2. DTE Background

DTE is an access control technology derived from the earlier work of Bobert and Kain [5,13] that restricts process access according to a site-specific security policy. A DTE system associates a *domain* with each running process and a *type* with each object (e.g., file, packet). As a DTE UNIX system runs, a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type. DTE restricts root as well as non-root processes and operates in addition to traditional UNIX protection bits. Suitably configured, DTE partitions a system according to the principle of *least privilege*, which grants each program only those access rights needed to perform its assigned function. This is a well-established technique for increasing both the security and reliability of computing systems.

A characteristic of DTE that distinguishes it from other access control schemes [3,4,5] is its use of a human-friendly high-level language for specifying security policies. The DTE Language (DTEL) provides four primary constructs for specifying policies:

- The **type** statement declares equivalence classes of data (e.g., personnel, manufacturing, corporate proprietary) that are treated differently by the policy.
- The **domain** statement defines the access modes a process running in that domain is permitted to use when accessing objects of specified types (e.g., read, write, or execute) or interacting with processes in other domains. A process running in domain A may *transition* to another domain B only by executing one of B's entry point programs (via exec()). A process may transition to another domain by explicit request only if its domain includes the *exec* mode of access to the target domain. Alternatively, a process may be automatically transitioned if its domain includes *auto* mode access to the target domain. The auto transition feature allows a policy to unilaterally partition families of existing programs into separate domains without requiring code modifications. A domain may also provide the right to send specified UNIX signals to processes in other domains.
- The **initial_domain** statement determines the domain in which the system's first process (usually /etc/init) starts.
- The **assign** statement binds types of data to specific files or directory hierarchies of files.

DTE and DTEL are described in greater detail in previous papers [2,12]; this paper covers them only as needed to elucidate the policy examples below.

3. Protecting System Binaries from Rootkit

Hacker toolkits are now widely available through many sources, including the Internet. These toolkits allow attackers to break into systems by exploiting security holes. Some toolkits also help the attacker cover their tracks afterward. An increasingly popular toolkit is Rootkit [16]. Once an attacker has penetrated a system and obtained root permission, Rootkit builds a hidden backdoor into the system for future access. Installing Rootkit modifies the standard UNIX login program to recognize special login names, skip normal access checks for those names, and provide a hidden session with root privileges. Rootkit modifies several other UNIX utilities, including ls, netstat, and ps, to hide the presence of the intruder after login.

3.1 Strategy

Our strategy for preventing installation of Rootkit is simple. Using DTE, we create a special administrative domain having write access to system binaries and their containing directories, and we allow transitions into that domain only after performing strong user authentication and authorization checks. All other processes, including root daemons, will run in less powerful domains that lack such write access. If an attacker subverts a root daemon, its accesses will be confined according to the daemon's domain, and it will be unable to replace login, ls, and other programs that constitute Rootkit's hidden backdoor.

The organization of a simple DTE policy that implements this strategy is depicted in Figure 1. The policy partitions all processes into four domains: 1) daemon_d, a domain for system daemons including init;

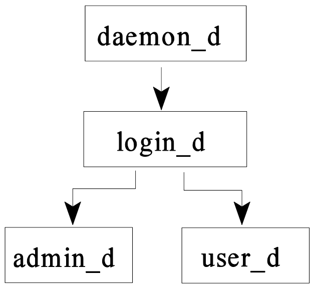

Figure 1. Domain Relationships

2) login_d, a domain for the DTE-enhanced login program; 3) user_d, a domain for ordinary user sessions; and 4) admin_d, a domain for system administrator sessions. The daemon_d domain includes an access right for transitioning to the login_d domain so that the daemon that invokes login (getty) can start login in the proper domain. The login program acts as a gatekeeper, determining which user sessions should be started in the user_d and admin_d domains. Consequently, the login_d domain includes rights that allow transitions to those two domains. Although most system administration tasks will be carried out using root processes in the administrator domain, both root and non-root processes can exist in each of these four domains.

All information in files and other objects is categorized into five types: 1) generic information created by user processes (generic_t); 2) system binaries (binaries_t); 3) UNIX configuration files to which most processes need only read access (readable_t); 4) DTE security metadata (dte_t); and 5) miscellaneous system-generated information that many processes may need to update (writable_t).

3.2 Initial Policy Core

Figure 2 shows the core of a simple DTE policy that implements this strategy. Additional policy elements needed to loosen constraints on functionality or strengthen security are described later.

3.2.1 Daemon_d

The daemon_d domain is the domain in which the first system process, init, runs. This is indicated in the initial_domain statement and by the tuple showing that /sbin/init is the entry point for the daemon_d domain. All descendants of init will run in the daemon_d domain until one of them invokes /usr/bin/login, the entry point for the login_d domain. This causes an automatic transition of the login program into the login_d domain, as indicated by the tuple (auto->login_d). Note that invoking the login program is the *only* way for processes in this domain to transition to any other domain.

Like most domains, daemon_d's definition includes a comma-separated list of expressions of the form (modes->type). DTE modes include read (r), write (w), and execute (x). These are similar to UNIX modes except that execute does not include directory traversal, which is a separate mode (d). The default creation type (c), is an extension of write mode; it identifies the type attribute automatically associated with new objects if the creating process does not explicitly specify a type.

The tuple (rxd->binaries_t) allows a process in the daemon_d domain to read, execute, and search for system binaries but not modify them, even if the process has root privilege; this tuple implements a key aspect of our strategy to thwart Rootkit. In this domain there is no type of file that is both executable and modifiable. As a result, processes in this domain cannot manufacture or import any executable that they can subsequently execute. Similarly, the tuple (rd->generic_t, readable_t) allows daemon processes to read but not modify various user-generated files and UNIX configuration files. The assign statements at the bottom of Figure 2 assign the type readable_t by default to all files in /etc and the type generic_t by default to all files in the file system that are not assigned a type by other assign statements. Additional assign statements can be added to label and protect additional configuration files in any other part of the file hierarchy. Defaults for directories are indicated by the -r (recursive) flag. The -s (strict) flag indicates that the type associated with the pathname cannot be changed at runtime, even if the file is replaced by a different file. When combined with the -r flag, -s indicates that all files within the indicated directory will be labeled with the specified type, no other types will be permitted in the directory.

3.2.2 Login_d

The login_d domain is the only domain that includes the tuple (exec->user_d, admin_d). It is therefore the only domain that can transition into the user_d or admin_d domains and is, hence, non-bypassable. Because the login_d domain is critical to our strategy, it has been designed so that only one binary can execute in it. That binary is the domain's entry point /usr/bin/login, the DTE-enhanced login program. Because this domain lacks execute (x) access to any type, any attempt by the login program to invoke any other binary without first transitioning to another domain (and shedding privilege) will be denied. This increases an attacker's difficulty of "commandeering" a process in this domain, should a means be discovered for penetrating the login program.

The DTE login program itself has been extended and strengthened in several ways. At login time, it obtains a role request from the user. If the user is authorized to assume that role, the login program invokes the user's shell (or other program) in the initial domain associated with that role, in this example, user_d or admin_d. The authentication and role authorization databases are labeled with DTE types (readable_t and dte_t) and access to these type is strictly controlled by the DTE policy to protect these files from unauthorized modification. Furthermore, at session start, the login program establishes a separate, immutable user ID, called the DTE UID, that cannot be modified via setuid programs or system calls. In addition, we will assume that the DTE login program uses stronger authentication mechanisms than ordinary reusable passwords. In fact, we have not yet implemented this modification, though doing so appears straightforward and several such mechanisms are readily available [1, 9, 17]. In a later section, we illustrate how DTE can be used to protect authentication data, using password files as a familiar example. The techniques described, however, are equally applicable to other kinds of authentication data.

```
type generic_t, binaries_t, dte_t, readable_t, writable_t;

domain    daemon_d =(/sbin/init),
                    (crwd->writable_t),
                    (rxd->binaries_t),
                    (rd->generic_t, readable_t),
                    (auto->login_d);

domain    login_d = (/usr/bin/login),
                    (crwd->writable_t),
                    (rd->generic_t, readable_t, dte_t),
                    setauth,
                    (exec->user_d, admin_d);

domain    user_d =  (/usr/bin/{sh, csh, tcsh}),
                    (crwxd->generic_t),
                    (rwd->writable_t),
                    (rxd->binaries_t),
                    (rd->readable_t, dte_t);

domain    admin_d = (/usr/bin/{sh, csh, tcsh}),
                    (crwxd->generic_t),
                    (rwxd->writable_t, binaries_t, readable_t,
                         dte_t);

initial_domain =    daemon_d;

assign    -r        generic_t         /;
assign    -r        writable_t        /usr/var, /dev, /tmp;
assign    -r        readable_t        /etc;
assign    -r -s     dte_t             /dte;
assign    -r -s     binaries_t        /sbin, /bin, /usr/libexec,
                                      /usr/{sbin,bin},
                                      /usr/local/bin;
```

Figure 2. Simple DTE Policy For Protecting Binaries and Configuration Files

3.2.3 User_d

The user_d domain allows read and directory access to all types of information on the system, subject to the additional restrictions imposed by ordinary UNIX mechanisms. In addition, execute access is allowed for system binaries (binaries_t) and other files created in this domain (generic_t). The "c" in the tuple (crwxd->generic_t) indicates that by default, any objects created in this domain will automatically be typed as generic_t. In addition, processes in this domain are permitted to modify existing objects of type writable_t or create new ones by explicit request. As indicated by the tuple (/usr/bin/{sh, csh, tcsh}), several common shells can be used as entry points into this domain. To non-administrative users running sessions in this domain, the system seems to behave like an ordinary UNIX system. Administrative users running in the user_d domain are prevented from carrying out certain administrative functions, even after becoming the superuser. In order to carry out all administrative duties, administrative users must login to the admin_d domain via the login domain.

3.2.4 Admin_d

The admin_d domain allows read, write, execute, and directory access to all types of files on the system. This includes the capability to create and modify binaries, UNIX configuration files, and DTE policy files. This capability is not available in any other domain. DTE files are of type dte_t and are kept in the /dte directory, as indicated by the policy statement "assign -r -s dte_t /dte."

The policy core described above, together with a small amount of standard policy boilerplate, has been validated experimentally on both OSF/1-based and BSD/OS-based DTE prototypes and shown to prevent installation of Rootkit while transparently allowing many normal system activities. We invoked the Rootkit installation script from a root shell in the daemon_d and user_d domains and verified that the script was unable to overwrite the login, ps, ls, and other binaries because of DTE access control checks.

3.3 Extension 1: Controlling Access to Password Files

In the initial policy example, it is not possible for regular users to change their system passwords or login shells. The password files typically reside in /etc, and therefore are of type readable_t. There are several ways this functionality deficiency can be handled. The simplest approach would be to provide the user_d domain with write access to readable_t. This approach is not acceptable because it allows any user who gains root privilege to change the password of any other user. This could allow an attacker in the regular user_d domain to change the password of a user authorized for the admin_d domain, thereby gaining access to a domain with the power to modify system binaries.

In order to fully protect user authentication information from unauthorized modification, access to this information must be strictly controlled. To do this, the password files are assigned a new type to which most domains are granted read-only access. The admin_d domain and a newly created domain (passwd_d) are permitted to write the password files. For this example, regular UNIX authentication mechanisms are employed, so the password programs (i.e., passwd, chpass and chsh) are the entry points to the passwd_d domain. User_d's domain specification forces the domain transition whenever one of these programs is executed.

This insures that only particular programs are permitted access to the authorization information. However, as indicated above, this is not sufficient because root users can change other users' passwords. Since we assume that root access can be obtained illicitly, the UNIX UID cannot be trusted for user identification. The DTE UID which is only set by login could be used, but the password programs would need to be modified to do this.

A less disruptive approach that allows the password programs to remain unmodified involves creating a simple wrapper (dtpasswd) for the programs. The wrapper program checks the DTE UID to ensure that the user is not attempting to change someone else's password information and then executes the requested password program. The wrapper program is designated as an entry point to the domain that has write access to the password files. This new domain is set up as an auto-transition from the user_d domain. If the user attempts to run a password program directly without using the wrapper, the program will run in the user_d domain and will not be allowed to update the password files. However, from the admin_d domain, the programs can execute unconstrained since the admin_d domain has write access to passwd_t. Therefore, the administrator can bypass the wrapper and will be permitted to change other user's passwords. The pertinent policy statements are included below; the full policy is in the appendix.

```
type  passwd_t;
domain passwd_d = (/usr/bin/dtpasswd),
                  (crwd->passwd_t),
                  (rwd->readable_t, writable_t),
                  (rxd->binaries_t),
                  (rd->generic_t);
assign    -s      passwd_t    /etc/{master.passwd, spwd.db, pwd.db,
                              passwd};
```

Figure 3. DTE Policy Extension for Passwd

3.4 Extension 2: Providing Additional Assurance

The above policies protect a system's executables and configuration files from direct manipulation by an attacker. In particular, the above policy protects these resources from direct attack via a toolkit like Rootkit. We now provide an extension that protects these resources from other, more sophisticated attacks by users who obtain root privilege. One such class of attacks attempts to manipulate raw devices via device special files, bypassing normal access paths and their associated DTE constraints. Two types of device special files present particularly inviting targets. The disk device special files allow access to the disk, bypassing the file system controls. The memory device files (/dev/kmem, /dev/mem) allow direct access to system memory. Using these devices, an attacker could change data on disk or in memory and undermine the security policy.

To protect the memory device files, a new type is defined (mem_t), and access to this type is restricted. The memory device files do not need to be written by any process, though they are read by several of the daemons and user processes (e.g. ps, vmstat).

Similarly, the disk device files can be protected by creating a new type for the device files and then restricting access to this type. Access can be limited to a few administrative processes (i.e., fsck, newfs, mount_mfs) by placing these processes in a new domain. This eliminates the need for the daemon_d domain to retain any access to the disk devices.

Another possible attack is to create new device special file that aliases a critical resource, such as the boot disk partition, and then freely manipulate that resource via the new device special file. The DTE prototype prevents these attacks by requiring that all device special files naming the same physical device have the same type and by allowing mknod commands only if the requesting process has read and write access to the type of the device. These mechanisms ensure that only processes in domains authorized by the DTE policy can manipulate devices. Note that this kind of protection can be extended to other kinds of devices (e.g., the console).

File system mount operations provide another potential means of attack. Because the DTE kernel binds type attributes implicitly to files based on directory hierarchies, an attacker might attempt to alter the hierarchies, thereby changing the type bindings in a way that lessens DTE constraints. For example, if the root user unmounts a file system and mounts it at a mount point that is recursively typed "foo_t" , all of the files on the file system would have their types changed to foo_t. To prevent this, DTEL provides "mount

```
type         mem_t, disk_t;
domain       fsck_d =    (/sbin/{fsck, mount_mfs}),
                         (crwd->disk_t),
                         (rwd->writable_t),
                         (rd->generic_t, readable_t);
assign    -s      mem_t       /dev/{kmem, mem, core};
assign    -s      disk_t      /dev/{rsd0a, rsd0b, rsd0g, rsd0h,
                                    sd0a, sd0b, sd0h, sd0g};

mount             (/dev/sd0a,       /);
mount             (/dev/sd0h,       /usr);
mount             (/dev/sd0g,       /usr/home);
```

Figure 4. DTE Policy Extension for Device Special Files

constraints" that restrict where file systems can be mounted. The DTEL mount constraints contain information similar to that in a UNIX fstab file. In addition to ensuring that mount operations conform to the constraints, the DTE prototype also prevents modification (e.g., rename) of device special files that are mount constrained. These mechanisms ensure that rogue root programs cannot disturb the type associations but allow for flexibility to remount disk partitions in cases where the types would not be affected.

By protecting the device special files and mount operations, a potentially serious backdoor through the security of most UNIX systems has been closed. The pertinent policy additions are included below; the full policy is provided in the appendix.

4. Broader Protection

The previous section describes how a simple DTE policy can protect system binaries from Rootkit and other root-based attacks. This section illustrates how small policy additions can extend DTE protection to other important information resources, in particular, system logs, administrative functions, and user data.

4.1 Protecting System Logs

Attackers often attempt to “cover their tracks” by modifying or disabling the attacked host’s audit system. This threat can be countered by another simple policy extension. First, the audit log files are given their own type, syslog_t. Second, a new domain is created for the syslogd daemon. Only this domain and the admin_d domain are permitted to write the syslog_t type. All other domains are given either no access or read-only access to the syslog_t type. The admin_d domain needs write access to this type to allow for controlled cleanup of the log files. Daemon_d will be permitted (in fact, required) to auto-transition into syslog_d when the syslogd program is executed.

As part of the login process, login modifies the files wtmp and lastlog. These files are given a separate type (usr_log_t), and the login_d domain is provided write access to this type. No other domain needs write access to this type. The important policy statements are included below. For the complete policy, see the appendix. This policy protects the system log files from modification by an attacker who has broken a system daemon or obtained access to a regular user account. It also prevents a malicious user from using signals to disrupt the syslog daemon’s operation.

4.2 Providing Multiple Administrative Roles

In many environments, it is prudent to split administrative duties into several parts so that some administrative duties can be performed by individuals whose privileges are limited. As an example of how this can be accomplished using DTE, we will derive from the admin_d domain a limited UNIX administrative domain. For clarity, we will rename the admin_d domain dte_admin_d. These domains will be designated as initial domains for the UNIX administrator and the DTE administrator roles. Users operating in the UNIX administrator role will be allowed to modify system binaries and configuration files but not the DTE configuration or password files. DTE administrators will be permitted to perform all administrative duties. Access to either of these roles is controlled by login and can be constrained with appropriate strong authentication. The UNIX administrator is not permitted to change other users’ passwords because this would allow a UNIX administrator to change a DTE administrator’s password and assume the DTE administrator role. The important changes to the DTE policy files are included below; see the appendix for a complete listing of the policy file.

```
type  syslog_t, usr_log_t;
domain      syslog_d =  (/usr/sbin/syslogd),
                        (crwd->syslog_t),
                        (rwd->writable_t),
                        (rd->readable_t, generic_t);

assign -r   syslog_t    /usr/var/{log, run/{syslog.pid, utmp}};
assign      usr_log_t   /usr/var/log/{wtmp, lastlog};
```

Figure 5. DTE Policy Extension for Syslogd

```
domain      unix_admin_d =    SHELLS,
                              (crwxd->generic_t),
                              (rwxd->binaries_t, writable_t, readable_t,
                                    syslog_t, usr_log_t),
                              (rd->dte_t, passwd_t),
                              (auto->passwd_d);
domain      dte_admin_d =     unix_admin_d,
                              (rwxd->dte_t, passwd_t);
```

Figure 6. DTE Policy Extension for Separation of Duties

4.3 Securing a Web Browser

The World Wide Web has made the Internet accessible to millions. The web was originally used as an easy way to share documents. Gradually, full-featured languages, like Java, have been developed that extend this technology. Using Java, a browser can retrieve code (an applet) from a remote machine and execute the code locally. Conceptually, this is a good way to remove processing bottlenecks; however, the security implications are considerable. A whole new genre of malicious code is now possible, code received from distant, unknown sites, at a user's request (or possibly without the users knowledge) and executed on the local machine with the user's access privileges. Although Java purports to prevent hostile applets from harming the browser's environment, a number of critical security flaws in the interpreter have already been discovered; we expect that additional flaws will be identified in the future. Generally, web browsers do not run with root privilege; however, an attack could certainly be developed that exploits a previously unknown hole in a setuid program to gain root access. Even without root, an unconstrained browser could do significant damage to user data.

Using DTE, users can be forced to run web browsers in a constrained domain that restricts access to the system's critical files. For this paper we constrained the NCSC's Mosaic web browser and Netscape's web browser. The only files that the browsers need to write are: 1) the "hotlist" and other files that the browser must update to capture history between sessions; and 2) other files in a designated scratchpad subdirectory in the user's home directory. The browser may download files into this subdirectory for printing or other purposes. In this domain, the browser cannot delete, modify, or overwrite any other file on the system, even if root privilege is obtainedn addition, all files created in the browser's domain are labeled with a type to which neither the browser, its progeny, nor any other process running in an ordinary user's domain has execute access rights. This means that the browser cannot import or manufacture binaries that it or other processes on the system are able to execute. As a consequence, the browser cannot plant trojan horse binaries that users might execute inadvertently. An auxiliary domain could be built that would allow users to regrade downloaded files to another type for execution. This would require the user to make a conscious decision to execute the code and could not be done by the browser. The pertinent policy statements are included below. Figure 8 shows how all of the above domains relate to each other.

```
type        browser_t;
domain      browser_d =        (/usr/X11R6/bin/{Mosaic, netscape}),
                               (crwd->browser_t),
                               (rwd->writable_t),
                               (rxd->binaries_t),
                               (rd->generic_t, readable_t, passwd_t);
assign    -r      browser_t    /usr/home/ken/{.MCOM-HTTP-cookie-file,
                               .MCOM-preferences, .MCOM-global-history,
                               .MCOM-cache, .MCOM-bookmarks.html};
assign    -r      browser_t    /usr/home/ken/{.mosaic-global-history,
                               .mosaic-hotlist-default, .mosiacpid,
                               .mosaic-personal-annotations);
```

Figure 7: DTE Policy Extension for a World Wide Web Browser

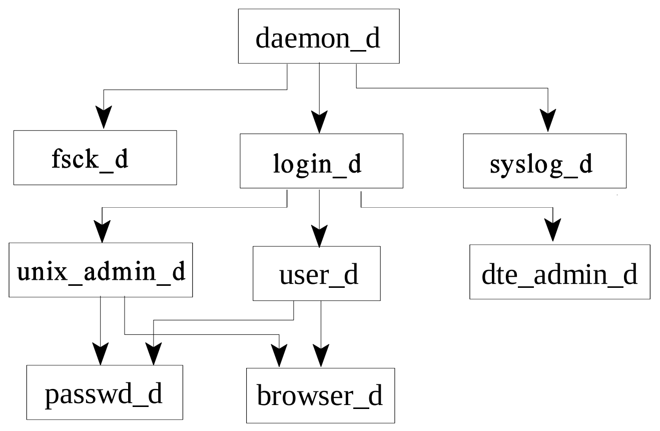

Figure 8. Final Domain Relationships

5. Discussion

5.1 DTE Policies For Operational Systems

The DTE policy components presented above and in the appendix have undergone limited testing to verify that they protect against the kinds of root-based attacks we cite. In addition, we have verified that they allow a significant range of common UNIX programs to operate normally including vi, emacs, ps, rlogin, telnet, and mail.

Day-to-day operational use of DTE systems, however, will likely require further policy extensions and DTE-aware versions of a few standard utilities. For example, a DTE-aware cron daemon may be needed that can start system and user programs in the domain of a requester or any domain into which a requester is authorized to transition. This requires that cron's own domain include the ability to transition into domains of requesters. Another example is preauthenticated rlogin. Providing security for preauthenticated rlogin in the presence of possible root penetrations requires that all inputs into the login_d domain be conveyed over DTE-protected paths; these inputs include file descriptors, command-line arguments, and DTE UID's received from the network. Protecting these input paths requires, in turn, that all getty, rlogin, and other trusted ancestors of login be segregated from other processes and placed into their own domain.

5.2 Security Tradeoffs

Perfect security is an impractical goal. Effective security, on the other hand, requires that security measures be deployed selectively and in a manner that balances security against other competing concerns. One of our research objectives is to build strong, flexible mechanisms that enable organizations to make tradeoffs involving operating system security.

One tradeoff illustrated above is the tradeoff between extent of protection and policy simplicity. Finer-grained control over the behavior of root processes can be gained by partitioning a system into more types and domains, where each domain provides minimal essential access rights for a group of related processes. Increased partitioning, however, results in a DTE policy that is larger, more complex, and less easily understood and maintained. This phenomenon is illustrated by the progression of incremental policy additions above. The initial policy core is simple but protects against a limited class of threats; the composite policy provided in the appendix is more complex but provides significantly broader protection.

Another tradeoff is between policy simplicity and the use of unmodified programs. The examples above attempt to minimize the number of standard UNIX programs that must be modified because of DTE. By modifying more programs, equivalent policies can be made simpler. Consider the domain for syslog described in section 4.1. Syslog needs to update the system log, which is of type syslog_t. Syslog also must be able to send return codes back to clients in other domains. Since the existing syslog program is not a

DTE-aware program, these return codes will also be of type syslog_t. Consequently, the right to read the type syslog_t must be added to the domains of syslog clients, making each of them somewhat more complex. The need to lengthen these domains can be obviated by modifying syslog so that it labels return codes sent to each client with type used by that client when it requests log services.

5.3 Comparison With Conventional Techniques

Like DTE, the UNIX chroot facility can limit damage that can occur if a system program is penetrated or tricked into misusing its privileges. A process running in a chroot'd environment is only able to access the subset of the file hierarchy beneath a designated directory. Other directories and files that are siblings or parents of this directory are inaccessible to the chroot'd process.

In principle, chroot could be used to confine root processes as described above. Doing so, however, confronts significant practical problems.

- Each chroot'd environment must contain its own copy of each file needed by the process(es) that will run in that environment. Setting up numerous chroot'd environments is therefore inconvenient and can waste disk space. Moreover, when updates are necessary, the presence of multiple file copies makes maintaining consistency across environments more difficult.

- There are known attacks through which root-privileged programs can subvert the chroot mechanism.

In contrast, DTE permits files to be made selectively inaccessible to different processes without file copying or the problems associated with links. More important, DTE can provide stronger protection than chroot'd environments because its underlying mechanisms provide no special exemptions for root programs. The primary drawback of DTE as compared with chroot is that DTE requires a modified kernel.

Tripwire and COPS are security configuration checking tools that protect system binaries and other critical files, but in a more limited manner that DTE. When run initially, Tripwire computes and stores cryptographic checksums for system binaries and other security-critical files. When run subsequently, it recomputes these checksums and compares them with the stored values. If a binary has been replaced or modified, the checksums will not match. When Tripwire detects a mismatch, it notifies the security administrator.

COPS compares ownership, permission bits, and contents of security-relevant files to sets of security expectations derived from established administrative practices. For example, it verifies that /bin and /etc are not world writable and that device special files are not world readable. It also detects poorly-chosen passwords. COPS produces reports that point out potential insecurities in the system's current configuration.

In terms of protecting system binaries and other critical files, the most important difference among the DTE prototype, COPS, and Tripwire, is that the former can *prevent* malicious modification while the latter two cannot. Instead, Tripwire detects such modifications after the fact. COPS simply warns about critical files that are unnecessarily exposed to attack by unprivileged programs. COPS provides little help, and can be disabled, if an attacker obtains root privilege.

COPS and Tripwire have been ported to a variety of UNIX systems; neither requires kernel modifications.

6. Related Work

The designers of a number of UNIX-oriented secure operating systems have attempted to eliminate root privilege or reduce its potential for misuse. Sidewinder (TM) [1], an Internet firewall, embodies the approach most similar to ours. Sidewinder is based on a version of BSD-OS UNIX that has been extended to include type enforcement, which it uses to impose additional constraints on root and non-root processes. Sidewinder includes two different kernels. Under the operational kernel, all processes are governed by type enforcement while "security policy checks are bypassed" [1] in the administrative kernel, which is used to install software. The administrative kernel can only be entered by a user who is "physically connected to the Sidewinder" [1] and only after shutting down the operational kernel. Sidewinder includes a fixed, vendor-supplied access control configuration not intended to be modified by customers. This is entirely appropriate since Sidewinder is an embedded turnkey system (i.e., a fixed-function device); moreover it increases the likelihood that the access controls have been and will remain configured properly.

By contrast, the DTE prototype is intended to be a general purpose UNIX system whose security mechanisms are easily configured by user organizations in accordance with their own security objectives, policies, and tradeoff decisions. This paper explains how these mechanisms can be configured effectively. The prototype provides a single kernel for both operational and administrative purposes. This allows an organization to use DTE to define and delimit the rights of administrative personnel as appropriate to the organization's needs. It also allows administrative tasks, including installing software and extending a system's DTE policy, to occur without rebooting.[3] Moreover, administrative tasks for multiple user workstations can be carried out from a single administrator workstation via networking.

Trusted Mach (TM), Trusted XENIX (TM), and HP-UX CMW (TM) use other strategies to mitigate root-related vulnerabilities. Each has been designed to protect classified information from leakage in accordance with the Trusted Computer Systems Evaluation Criteria [2]. Trusted Mach effectively runs a separate UNIX operating system (OS) at each security classification, e.g., unclassified or secret. Each user process also has a classification and can communicate only with the corresponding OS. Strong separation between the OSs and clients at different classifications is enforced by a trusted computing base completely independent of UNIX mechanisms. As a result, an adversary that obtains UNIX root privilege at one classification gains no additional access rights to information at other classifications; hence security, in the sense of preventing leakage, is preserved.

Trusted XENIX is a modified UNIX operating system that supports user processes at multiple security levels concurrently [3]. In Trusted XENIX, root privilege has been replaced by a collection of 36 specific privileges that selectively allow use of privileged system calls and non-privileged system calls with privileged options. For example, a call to audit() succeeds only if the caller possesses the AUDIT privilege. Privileges are encoded in a bit vector associated with each executable file and stored in an extended inode structure. When a process executes a program, the process obtains the program's privileges. Trusted XENIX defines five hierarchical roles for administrative users.

Like Trusted XENIX, HP-UX CMW supports user processes at multiple security levels concurrently [4]. Similarly, HP-UX replaces root privilege with a collection of finer-grained privileges, each of which grants a process the right to invoke a particular action such as setting the system clock. HP-UX CMW provides three predefined administrative roles.

A fundamental difference between the DTE prototype and both Trusted XENIX and HP-UX is that the latter two have a predefined privilege structure that is hard-coded into their kernels. By contrast, the DTE prototype relies only minimally on the notion of predefined privileges. Instead, it allows user organizations to choose the kind and granularity of protection appropriate to their needs, recognizing that these needs may evolve over time and that unnecessary granularity may increase the cost and difficulty of administering and maintaining a system.

7. Conclusion

The pervasive use of the all-powerful root privilege is one of the most important sources of security problems in UNIX systems. This paper has explained how a DTE-enhanced UNIX, driven by simple DTE policies, can substantially improve the security of UNIX by confining the accesses of root programs and preventing them from accessing critical files in inappropriate access modes.

As an illustrative example, we have presented a simple DTE policy that thwarts Rootkit and other root-based attacks that attempt to modify or replace critical system binaries, e.g., login. This policy has been experimentally validated on our BSD-OS-based DTE prototype as preventing Rootkit from installing malicious binaries while allowing a variety of unmodified UNIX programs to be used normally. In addition, we have presented a sequence of small, incremental policy extensions that provide broader protection against other root-based attacks. While these extensions do not address all known attacks, they do provide strong protection for several critical UNIX abstractions including password files, raw disk devices, kernel memory, mount operations, and system audit logs.

The DTE prototype is intended to be a general purpose system whose security mechanisms can be configured by user organizations in accordance with their own security concerns and tradeoff decisions. The policy examples presented here illustrate that very simple DTE policies can provide limited but useful protection

[3] Features to constrain "on-the-fly" policy changes are currently under development.

beyond that provided by ordinary UNIX. Moreover, they illustrate that broader protection can be provided at the expense of policy simplicity. The policies presented here are predicated on minimizing the number of existing UNIX utilities that must be modified. Additional policy simplifications are possible in some cases if minor modifications are made to particular UNIX programs.

The results presented here represent work in progress. We are now beginning to use the above policies to protect the UNIX workstations we depend on for routine, daily computing. We expect to continue improving the security of these workstations against root-based attacks by refining and extending these policies and the DTE prototype.

Bibliography

1. F. Avolio, M. Ranum, "A Network Perimeter with Secure External Access," Proc. Internet Society Symposium on Network and Distributed System Security, San Diego, CA, Feb 1994.

2. L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker, "A Domain and Type Enforcement UNIX Prototype," USENIX Computing Systems, Vol 9, No.1, Winter 1996, pages 47-83.

3. D.E. Bell and L. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," Technical Report No. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976.

4. K.J. Biba, "Integrity Considerations for Secure Computer Systems," USAF Electronic Systems Division, Bedford, MA, ESD-TR-76-372, 1977.

5. W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, p. 18, 1985.

6. W. Cheswick, and S. Bellovin, Firewalls and Internet Security: Repelling the Wily Hacker, Addison-Wesley, 1994.

7. D. Farmer, "The COPS Security Checker System," Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, p. 165.

8. S. Garfinkel, G. Spafford, Practical UNIX Security, O'Reilly and Associates, 1991.

9. N. Haller, "The S/Key One-Time Password System," Proc. Internet Society Symposium on Network and Distributed System Security, San Diego, CA Feb, 1994.

10. G. Kim, E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," Purdue Technical Report CSD-TR-93-071, November 1993.

11. "Secure Web Platform Whitepaper," http://www.sware.com/papers/, SecureWare Inc., February 2, 1996.

12. D.L. Sherman, D. F. Sterne, L. Badger, S.L. Murphy, K.M. Walker, S.A. Haghighat, "Controlling Network Communication With Domain and Type Enforcement," Proc. 18th National Computer Security Conference, pages 211-220, Baltimore, MD, 1995.

13. D. J. Thomsen, "Role-based Application Design and Enforcement," In Proc. of the Fourth IFIP Workshop on Database Security, Halifax, England, September 1990.

14. D. J. Thomsen, "Sidewinder: Combining Type Enforcement and UNIX," Proc. 11th Computer Security Applications Conference, Orlando, FL, December 1995.

15. Trusted Mach Philosophy of Protection, Edoc-0003-93B, Trusted Information Systems, Inc., May, 1993.

16. W. Venema, "Root Kit," Presentation at SURFnet CERT-NL SGG-SEC/SSC Workshop, May, 1995.

17. W. Venema, "Logdaemon," software kit, available from ftp://ftp.win.tue.nl/pub/security/logdaemon-5.0.tar.gz

18. D.J. Thomsen, “Sidewinder: Combining Type Enforcement and UNIX,” Proc. 11th Computer Security Applications Conference, Orlando, FL, December 1995.

19. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

20. *Trusted XENIX Version 3.0 Final Evaluation Report*, CSC-EPL-92/001 National Computer Security Center, Fort Meade, MD, April 8, 1992.

21. “HP-UX CMW Compartmented Mode Workstation Version 10.9,” <http://www.dmo.hp.com/Fed/tac4/CMW.10.6.html>, Hewlett-Packard Company, May 15, 1996.

Appendix

The final DTE policy files are now presented. Some of the policy statement functions were not covered in the paper, a short explaination of these functions are provided in in-line comments.

```
/*
 *    File: /dte/policy/dt_policy
 *
 *    DTE policy file.
 */

/*******************   TYPE SECTION   *********************************/

type
      binaries_t, // System executables
      dte_t,      // DTE configuration files
      generic_t,  // User data
      readable_t, // System configuration files
      writable_t, // System-created data
      syslog_t,   // Log files
      usr_log_t,  // User logfiles
      kmem_t,     // Kernel memory device special files
      disk_t,     // Disk device special files
      passwd_t,   // Files used for user authentication
      browser_t,  // Files written by a web browser
      tcp_t;      // System generated TCP overhead data
                  // only used when no user data is sent

/*******************   END TYPE SECTION   *****************************/

#define     SHELLS            (/bin/{sh,csh}, /usr/contrib/bin/tcsh)

domain      tcp_d  =    (crw->tcp_t);
domain      non_dte_d = (crw->generic_t),
                        (r->binaries_t, dte_t, readable_t, writable_t,
                              tcp_t, browser_t);

/*
 *    User domains.
 */

domain      daemon_d =  (/sbin/init), SHELLS,
                        (crwd->writable_t),
                        (rxd->binaries_t),
                        (rd->generic_t, readable_t, dte_t, syslog_t, kmem_t,
                              passwd_t),
                        (r->disk_t),
                        (auto->login_d, syslog_d, fsck_d);

domain      fsck_d =    (/sbin/{fsck, mount_mfs}),
                        (crwd->disk_t),
                        (rwd->writable_t),
                        (rd->generic_t, readable_t);

domain      syslog_d =  (/usr/sbin/syslogd),
                        (crwd->syslog_t),
                        (rwd->writable_t),
                        (rd->readable_t, generic_t);

domain      login_d =   (/usr/bin/login),
                        (crwd->writable_t),
                        (rwd->usr_log_t),
                        (rd->generic_t, readable_t, dte_t, syslog_d,
                              passwd_t),
                        setauth,
                        (exec->user_d, dte_admin_d, unix_admin_d);

domain      user_d =    SHELLS,
                        (crwxd->generic_t),
                        (rwd->writable_t),
                        (rxd->binaries_t),
                        (rd->syslog_t, usr_log_t, readable_t, dte_t, kmem_t,
                              passwd_t, browser_t),
                        (auto->passwd_d, browser_d);

domain      passwd_d =  (/usr/bin/dtpasswd),
                        (crwd->passwd_t),
                        (rwd->writable_t, readable_t),
                        (rxd->binaries_t),
                        (rd->generic_t);

domain      browser_d = (/usr/X11R6/bin/{Mosaic, netscape}),
                        (crwd->browser_t),
                        (rwd->writable_t),
                        (rxd->binaries_t),
                        (rd->generic_t, readable_t, passwd_t);

domain      unix_admin_d =    SHELLS,
                        (crwxd->generic_t),
                        (rwxd->writable_t, binaries_t, readable_t, syslog_t),
                        (rwxd->usr_log_t, disk_t, kmem_t),
                        (rd->dte_t, passwd_t),
                        (exec->daemon_d), /* for restarting daemons */
                        (auto->passwd_d, browser_t),
                        (sigtstp->daemon_d);    /* System reboot */

domain      dte_admin_d =     unix_admin_d,
                        (rwdx->dte_t, passwd_t),
                        (auto->browser_d),
                        (sigtstp->daemon_d);    /* System reboot */

initial_domain =  daemon_d;

mount           (/dev/sd0a,     /);
mount           (/dev/sd0h,     /usr);
mount           (/dev/sd0g,     /usr/home);

inet_assign non_dte_d   0.0.0.0;

#include dt_assign
```

```
/*
 *    File: /dte/policy/dt_assign
 *
 *    DTE attribute association file.
 */

// Default type for all files.
assign -r         generic_t   /;

// Protect security information
assign -r -s      dte_t       /dte;

// Types for executable areas
assign -r         binaries_t  /bin, /sbin, /usr/{bin, sbin};
assign -r         binaries_t  /usr/contrib/bin, /usr/libexec;
assign -r         binaries_t  /usr/games, /usr/local/etc;
assign            binaries_t  /etc/uucp/{daily, weekly, uuxqt_hook};

// Writable areas
assign -r         writable_t  /usr/var, /dev, /tmp;
assign            writable_t  /dte/dt_diag;

// Read-only areas
assign -r         readable_t  /etc;

// System log areas
assign -r         syslog_t    /usr/var/log;
assign            syslog_t    /usr/var/run/{syslog.pid, utmp};

// User log areas
assign            usr_log_t   /usr/var/log/{wtmp, lastlog};

// Password files
assign    -s      passwd_t    /etc/{master.passwd, spwd.db, pwd.db,
                                    passwd};

// Critical device special files
assign      -s    kmem_t      /dev/{kmem, mem, drum};

assign     -s     disk_t      /dev/{rsd0a, rsd0b, rsd0g, rsd0h,
                                    sd0a, sd0b, sd0g, sd0h};

// Browser writable files
assign -r         browser_t   /usr/home/ken/{.MCOM-HTTP-cookie-file,
                              .MCOM-preferences, .MCOM-bookmarks.html,
                              .MCOM-global-history, .MCOM-cache};
assign -r         browser_t   /usr/home/ken/{.mosaic-global-history,
                              .mosaic-hotlist-default, .mosaicpid,
                              .mosaic-personal-annotations};
```