Domain and Type Enforcement for Linux

Serge E. Hallyn
College of William and Mary
hallyn@cs.wm.edu, http://www.cs.wm.edu/˜hallyn
Phil Kearns
College of William and Mary
kearns@cs.wm.edu, http://www.cs.wm.edu/˜kearns

Abstract

Access control in Linux is currently very limited. This paper details the implementation of Domain and Type Enforcement (DTE) in Linux, which gives the system administrator a significant advantage in securing his systems. We control access from domains to types, domain transitions, and signal access between domains, based on a policy which is read at boot time.

1 Introduction

Access control in Linux currently consists of traditional Unix permissions and POSIX capabilities[Caps-faq]. Domain and Type Enforcement (DTE) has been presented [DTE95, DTE96] as a useful method for enhancing access control. DTE groups processes into domains, and files into types, and restricts access from domains to types as well as from domains to other domains. Type access can be any of read, write, execute, create, and directory descend. Domain access refers the right to send signals as well as that to transition to a new domain. A process belongs to exactly one domain at any particular time. A process transitions to a new domain by executing a file which has been defined as an *entry point* to that domain. The three types of domain transitions are *auto*, *exec*, or none. If Domain A has auto access to domain B, and a process in domain A executes an entry point for domain B, then the process will be automatically switched to domain B. If domain A has exec access to domain B, then a process running under domain A can choose whether to switch to domain B on execution of one of B's entry points.

DTE can be considered an abbreviated form of classical capabilities[Dennis66]. In a system based upon classical capabilities, a process carries with itself a set of access rights to particular objects. At any point, a process can give up, or reclaim (if permitted) some of its capabilities. POSIX capabilities work similarly, but these capabilities are limited to a predefined subset of superuser access rights such as the ability to nice a process, boot the system or open a privileged ($<$ 1024) port. In DTE, a process carries with itself only an indicator of the domain in which it runs, and this determines the process' access rights. A process can enter a new domain (and hence change its access rights) only upon file execution.

Trusted Information Systems has used DTE in its proprietary firewalls, but details of its implementation were not publicly available, and TIS appears to have stopped using DTE altogether. A group at SAIC has recently begun a DTE for Linux implementation[SAIC-DTE]. Jonathon Tidswell and John Potter[Tidswell97] submitted theoretical work on extending DTE to allow safe dynamic policy changes, but have attempted no implementation.

Presented here is our prototype implementation of DTE for Linux version 2.3.

2 Implementation

We have implemented a DTE prototype in the `2.3.28` Linux kernel. Our implementation of DTE attaches type information to VFS inodes and domain information to process descriptors (task structs). A DTE policy is read at boot time from the text file `/etc/dte.conf`.

Traditional UNIX permissions are still enforced. There are several reason for this, such as user and system administrator familiarity with traditional UNIX protection. Most importantly, however, DTE is designed to provide mandatory access control to protect a system from subverted superuser processes. A DTE policy to replace traditional UNIX access control would be very large and complex. However, one could completely void traditional access control by simply giving all users full access to all files. Similarly, one can bypass DTE by creating a DTE policy with only one type and one domain, and full access from the sole domain to the sole type.

2.1 Data Management

At boot time, we build a structure for each domain as specified in the DTE policy file. This structure contains information regarding permitted access to types, permitted transitions and signal access to other domains, and entry points. Every process' task structure will contain a pointer to the structure for the domain to which it currently belongs.

At this time we also create an array containing the names of all types. Types are then compared by the offset of the type name in this array. Every inode contains three pointers which either are NULL or point into this array. The three pointers represent the *etype*, *rtype* and *utype* values. The *etype* value is the type of this particular file or directory. The *rtype* represents the value of this directory and its children, whereas the *utype* represents only the type of its children.

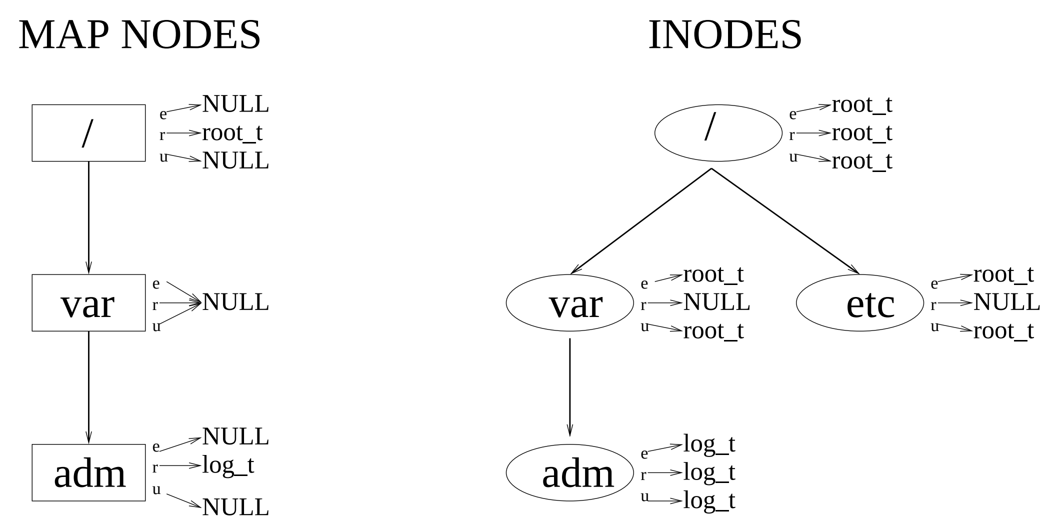
The type of a file is determined in one of three ways. First, if we have previously determined the type, then the inode's *etype* will be set and we simply use it. If this is the first time we are looking up this file, and a rule exists assigning it a type, then that rule is used. Finally, if a rule does not exist assigning a type to the file, then the values are inherited from the parent's *utype*[1]. The *etype* of an inode is always set. If there is no rule specifying the *etype* for the file, then the *etype* is set to the parent's *utype*. The *utype* of an inode must therefore also exist. It is set, in order of preference, to the assigned *utype*, the assigned *rtype*, or the parent's *utype*. The *rtype* of an inode is set only if a rule assigns an *rtype* to the inode's path.

Since type information always comes from either the DTE policy or from an inode's ancestors in the filesystem tree, no information needs to be added to the filesystem on disk. The type assignment rules are represented in memory by a tree of *map nodes* which is constructed at boot time from the type assignment rules in the policy. A sample tree for a particular set of rules is shown in Figure 1, along with the type information in corresponding inodes. The map nodes are only used to determine whether a rule exists binding a path to a type. Once an inode's type is set, subsequent lookups will not cause us to check the map nodes again.

2.2 Type Access Enforcement

When a process performs an *open* system call, the modified kernel checks for DTE permission before checking the standard UNIX permissions. We use the domain structure pointed to by the current task structure to check whether the current domain has the requested access to the type to which the file being opened belongs. If this access is granted, then we proceed to perform the normal UNIX checks. A check for DTE execute permission is delayed until the actual call to *execve*. If the execution causes an allowed domain transition, then this transition should occur before the check for execute access, since the new domain may be the only one allowed to execute the entry point.

[1]The type of the root of the filesystem is set explicitly in the DTE policy.



```
# extract from the sample policy:
default_rtype    root_t
assign   −r   /var/adm    log_t
```

Figure 1: Sample DTE assign rules and corresponding map nodes

Domain to type access information is kept in a set of hash tables. Each domain structure has a hash table keyed by the type name, and each entry lists the domain's access rights to the particular type. A type access check, therefore, consists of simply calculating the hash value of the type name to find the appropriate domain to type access entry for the current domain, and comparing the requested access to the permitted access. This is done regardless of username, so that the superuser is not exempt from the DTE policy.

2.3 Domain Access Enforcement

Our DTE implementation enforces restrictions on signals between processes in different domains. Each domain structure contains a linked list of *dte_signal_access* structures, which contain the signal number and a pointer to the domain to which it may be sent. One of these structures exists for every signal which may be sent to another domain. However, for the sake of abbreviation, setting the domain to null allows the specified signal to be sent to all domains, and setting the signal to 0 allows all signals to be sent to the specified domain.

2.4 Domain Transition Enforcement

Three types of domain transition are possible each time a file is executed. The first is an *auto*, or mandatory, domain transition. Since this must be automatic, it means that each time a process calls `execve`, we must check whether the file being executed is an entry point into a domain to which the current domain has *auto* access. The second type of transition is an *exec*, or user-requested, transition. This is facilitated by a new system call, `sys_dte_execve`, which takes an additional argument over `execve` containing the name of the requested domain. The third and default type of transition is the NULL transition, wherein the domain is not changed.

Domain transition information is kept in two types of structures, both linked from the domain structure. Since *auto* transitions must be checked for on every `execve` system call, the search for a particular pathname must be very quick. Therefore, each domain structure contains a hash table of the pathnames whose execution lead to *auto* domain transitions, along with the domain to be switched to.

The domain structure also has a linked list of structures representing allowed *exec* transitions. Since a `dte_exec` is a relatively rare, and user-requested, event, efficiency is not so critical, and we can elect for a more memory-efficient representation. Therefore we do not keep a hash table of every file which may cause an *exec* transition, but simply point to the domains to which a voluntary (*exec*) transition is allowed. To check for *exec* access to a domain, we must first check for an *exec* entry for the desired domain, then check whether the file being executed is an entry point for that domain.

3 Administration

Administering DTE consists of editing the policy, which is defined in the file `/etc/dte.conf`. The system must be rebooted to effect the changes[2].

The DTE policy file consists of several sections. We first enumerate the types and domains. Next we specify the default type for the filesystem root (`"/"`) and its children, and the domain in which to run the first process (init). Following is the detailed definition of all domains. For each domain we specify the entry points, permitted type access, permitted domain transitions and permitted signals to processes in other domains. Finally we list the type assignment rules.

A sample policy file is in Figure 2. First we specify that there will be two types, *root_t* and *log_t*, and two domains, *common_d* and *log_d*. We set the default root rtype, hence the default type for the entire filesystem, to *root_t*. Next we set the type of the first process to *common_d*. We specify that the *log_d* domain will have one entry point, `/sbin/syslogd`, and should have read, execute and directory descend access to files of type *root_t* and read, write, execute, create and directory descend access to files of type *log_t*. For the domain *common_d*, we specify read, write, execute, create and directory descend access to files of type *root_t*, but only read access to files of type *log_t*. This domain also receives *auto* transition access to domain *log_t*, meaning that, on execution of `/sbin/syslogd`, a process in domain *common_d* will automatically be switched to domain *log_d*. Finally, the last statement assigns the type *log_t* to the directory `/var/adm/log` and all files thereunder.

4 The DTE API

Three additional system calls are provided to allow software to interact with DTE. The `sys_dte_exec` call was discussed earlier. A user may invoke `sys_dte_gettype` to learn the type associated with a file. Similarly, `sys_dte_getdomain` may be called to learn the domain associated with a process.

5 Performance

We measured the performance of both a DTE-enabled and a DTE-free `2.3.28` kernel for the `execve` and `lookup_dentry` system calls, the overhead imposed by the DTE-specific `sys_dte_exec` and `dte_auto_switch` system calls, and a full kernel compile. The following tests were run on a 400Mhz Pentium II (397.31 bogomips) with 512K L2 cache and 384M ram. Each test was run on a kernel compiled without DTE and one with DTE using the simple policy shown in Figure 3. We used the Pentium cycle clock for timing. All confidence

[2]Allowing the safe run-time changing of access control rules is a topic of some ongoing research[Tidswell97]

```
# this is a comment
types root_t log_t      # enumerate the types
domains common_d log_d  # enumerate the domains

default_rtype root_t    # default type for /
default_domain common_d # domain for process 0

# A domain is specified in n parts:
# spec_domain <domain_name> (entry points) (type access) (domain access) \\
#                  (signal access)
spec_domain log_d (/sbin/syslogd) (rdx->root_t rwxcd->log_t) () ()
#             ^        ^                ^                        ^
#          (name) (entry point)   (type access)             signal access
spec_domain common_d () (rwxcd->root_t r->log_t) (auto->log_d) ()
#                                                     ^
#                                               domain access

assign -r /var/adm/log log_t   # assign type log_t to /var/adm/log
                               # and all files there-under
```

Figure 2: A sample DTE configuration file

intervals are 95%.

5.1 Permission

The `fs/namei.c:permission` kernel function is used before any file operations to check whether the user is authorized to perform the requested action. The code to check for domain to type access rights is located at the top of this function, so that DTE permissions are checked before standard UNIX permissions. As mentioned above, each domain has a hash table, keyed by type name, listing the domain's access rights to types. The DTE permission check is therefore very quick and constant time with respect to the number of types. Of course, it is linear with respect to the length of the pathname, as we need to first find the pathname and then hash it.

The first time it is called on a particular file or directory, however, the *etype* may not yet have been set. In this case, we must check for a type assignment rule or, if such a rule does not exist, set the type from the parent directory. Furthermore, if the parent directory does not exist then we must first do the same for it, and so on until a directory is associated with a type assignment rules or has its type set.[3]. The DTE type assignment rules are kept in a tree format analogous to the filesystem tree, as shown in Figure 1. The children are currently not sorted, so that a large number of assignment rules for files under a single parent directory could impact performance. However, for normal cases this lookup should be reasonably quick.

We timed the upper part of the kernel function `fs/namei.c:permission`, where the DTE code is located. Over the course of a boot sequence, several repeats of the lookup test above, some general milling around, and a shutdown, the DTE code added 1578 ± 400 clock cycles to each `permission` call.

5.2 lookup_dentry

The time required to look up a given pathname greatly affects the subjective performance of the system. The function `fs/namei.c:lookup_dentry`, which performs this task in the Linux kernel, is affected by DTE in two places. First, for each subdirectory in a pathname, `lookup_dentry` calls `per-`

[3]As must eventually be true since the filesystem root has a defined type

```
types root_t login_t user_t test_t spool_t tripwire_t
domains root_d login_d user_d test_d tripwire_d
default_d root_d
default_et root_t
default_ut root_t
default_rt root_t
spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t rwxcd-
>spool_t \
      rwcdx->user_t) (auto->login_d auto->tripwire_d)
spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd-
>spool_t) \
      (exec->root_d exec->user_d exec->test_d)
spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxd->root_t rwxcd-
>spool_t) \
      (exec->root_d exec->test_d)
spec_domain test_d (/bin/bash) (rwxcd->test_t rdx->user_t rwdx-
>root_t rwxcd->spool_t) \
      ()
spec_domain tripwire_d (/bin/tripwire) (rwxcd->tripwire_t rxd->user_t rxd-
>spool_t \
      rxd->root_t) ()

assign -r /etc/tripwire tripwire_t
assign -r /var/spool/tripwire tripwire_t
assign -u /home user_t
assign -u /tmp spool_t
assign -u /var spool_t
assign -u /dev spool_t
assign -u /scratch user_t
assign -r /dte_test_dir test_t
# next one is a test - user_d should *not* see it since no ’d’ to /dte_test_dir
assign -e /dte_test_dir/aha user_t
```

Figure 3: DTE policy used for performance tests

`mission` to check for execute access. Second, if the types for the deepest path element being looked up have not been set, then we must set them, using the same function we use above in `permission`.

We timed `lookup_dentry` on a set of pathnames ranging in depth from 1 to 9 components, both for fully existing and fully nonexistent pathnames. For the first execution, each component of each pathname was uncached. On subsequent executions, all path components and their corresponding DTE type information were (naturally) cached.

The results can be seen in figures 4, 5, 6 and 7. For the case of a lookup for uncached filenames, results appear to be rather unpredictable. If this appears to be more true for existing file lookup than for nonexistent files, this is a result we should have predicted by our method of testing. We tested the lookup for each set of pathnames, then rebooted, and repeated the test, eleven times in all. However, for the nonexistent filename lookup, a part of the pathname was legitimate. This piece was looked up uncached only for our first test after reboot, which was for the first table entry in figure 6. Since the DTE kernel was faster than the plain kernel more often than it was slower, it appears safe to say that disk i/o completely overshadows any time spent setting DTE types from map rules and parents.

For cached lookups, the DTE kernel appears to do slightly better than twice as long as the plain kernel.

5.3 *auto* Domain Transitions

Upon file execution, we must check whether the requested execution should cause a mandatory domain switch. This is done using `kernel/dte.c:dte_auto_switch`. As previously mentioned, this function must be fast as it is called with every file execution. Therefore, it simply hashes the name of the executable to check for an entry in a table of gateways, or executables which cause an automatic domain switch.

We compiled a kernel which timed the execution of `dte_auto_switch`. If a particular domain has no gateways, then `dte_auto_switch` does not bother to hash the typename, so that the `dte_auto_switch` call during `execve` takes 308 ± 6 clock cycles. If there are gateways, then we must search the hash tables. While we tested using domains with a variable number of permitted auto switches,[4] this number does not affect the running time of `dte_auto_switch`, which is 6655 ± 166 clock cycles. Since this function does not exist in the plain Linux kernel, its running time must be considered pure overhead to file execution.

5.4 *exec* Domain Transitions

The least efficient of all the code added with DTE certainly sits in `kernel/dte.c:sys_dte_exec`. First, the user provides the name of a domain to switch to. Since domains are currently not kept hashed or in any order, the lookup for the corresponding domain structure is $O(d \times m)$, where d is the number of defined domains and m is the maximum length of any domain name. Next, we search another unsorted list, containing the domains to which the current domain may voluntarily switch, to check whether the domain switch is legal. Then we search a third list, containing valid entry points for the destination domain.

In order to measure the amount of time required to check for an *exec* domain switch, we set up 12 domains, with entry points numbering $2, 4, 6, 8, 10, 12, 14, 16, 18, 20$ and 30, where two domains had 30 entry points. Then we performed an *exec* domain switch into each of these domains, and measured the time between the start of `sys_dte_exec` and its call to `sys_execve`. Since entry points are stored unsorted, the 12th domain's list of entry points contained the entry point which we actually executed last, whereas all others listed it first. The results for 10 trials (excluding the first of eleven since the entry

[4] Mainly to test for a bad implementation of poor hash function.

# Path Elements	plain	DTE
1	14239 ± 115	3715106 ± 3764708
2	12982865 ± 740602	9443087 ± 3764708
3	12328686 ± 3305948	17481580 ± 18583745
4	16894558 ± 1323098	15491659 ± 3111030
5	11514025 ± 1124946	11145553 ± 2406925
6	14939328 ± 837802	13684836 ± 2800215
7	21312794 ± 17210467	12670629 ± 3303167
8	6362366 ± 2955509	4912447 ± 2608638
9	33647759 ± 20269841	19853653 ± 4192949

Figure 4: existing file lookup, first runs

# Path Elements	plain	DTE	% increase
1	4450 ± 56	8222 ± 71	85
2	5085 ± 67	8925 ± 81	76
3	5223 ± 75	10798 ± 105	107
4	6475 ± 102	12076 ± 151	87
5	7097 ± 101	13463 ± 124	90
6	7747 ± 126	14605 ± 171	89
7	8374 ± 155	15918 ± 167	90
8	9194 ± 99	17415 ± 144	89
9	9867 ± 254	18602 ± 186	89

Figure 5: existing file lookup, cached runs

# Path Elements	plain	DTE
1	11437289 ± 911243	7667678 ± 3621479
2	10045 ± 1270	12319 ± 239
3	9421 ± 345	12260 ± 322
4	9343 ± 378	12482 ± 281
5	9911 ± 1171	35990 ± 43904
6	9934 ± 1298	12036 ± 371
7	9299 ± 287	12789 ± 1147
8	9956 ± 1151	12929 ± 1231
9	9236 ± 275	12336 ± 543

Figure 6: non-existent file lookup, first runs

# Path Elements	plain	DTE	% increase
1	4508 ± 90	8221 ± 57	82
2	4414 ± 136	8270 ± 120	87
3	4334 ± 120	8240 ± 103	90
4	4330 ± 90	8270 ± 133	91
5	4267 ± 177	8236 ± 123	93
6	4247 ± 184	8231 ± 133	94
7	4326 ± 63	8206 ± 137	90
8	4416 ± 79	8267 ± 99	87
9	4415 ± 106	8240 ± 154	87

Figure 7: non-existent file lookup, cached runs

point needed to be read from disk) are shown in Figure 8.

The first 11 domains each executed the first file in the respective domains' linked list of entry points. The difference in performance is due to the analogous problem with the list of allowed *exec* transitions.

For domains with what we believe to be a realistic number of entry points (1-8), `sys_dte_exec` takes about 4 times as long as `dte_auto_switch`. Clearly, performance will be greatly improved when we store entry points, domains, and allowed *exec* transitions in a data structure which allows quicker lookups. This will be a simple but low priority improvement, since a policy must be quite large for the effects to become noticeable, and a `sys_dte_exec` call is a rare event.

5.5 execve

To time the kernel function `fs/exec.c:do_execve`, we wrappered it and took a timestamp before and after the real function call. In this way we measure the full time for file execution including such details as the time to load library files. For more fine-grained measurements of specific parts of this process, we later measure the time to check for the *auto* domain switch, a user-requested domain switch and filename lookup.

The command

```
/bin/echo -n .
```

was executed 500 times. Execution time for the first run was an order of magnitude larger than for subsequent runs, both with and without DTE. This is to be expected since some library files as well as executable `/bin/echo` may not yet have been loaded from disk. The same thing occurs for later performance tests. Since this is independent of the DTE code and serves only to hide the performance impact of DTE, we will, in all subsequent tests, ignore the first execution after boot.

The DTE code introduces a 10% overhead. The table in Figure 9 shows the timing results.

5.6 *make bzImage*

Finally we turned off all micro-performance measurements and used `/usr/bin/time` to determine the performance on a kernel make on both DTE and non-DTE enabled kernels. The plain `2.3.28` kernel took 5 minutes and 55 seconds for the first compile, and 5:35 $\pm$ 0.384387 for 14 subsequent compiles, while the DTE-enabled kernel required 5 minutes and 56 seconds for the first compile and 5:36 $\pm$ 0.205464 for subsequent compiles.

Clearly a new access control system cannot be added without affecting performance. The above sections, in testing specifically the areas of the kernel where code was added, might make the performance impact of DTE seem more significant than it really is. This result shows that, when amortized over the course of a realistic activity, which includes heavy file opening, creation and execution as well as heavy computation and file i/o, the amount of overhead, one second for every six minutes, is negligible.

6 Real Attacks

To show the effectiveness of our DTE implementation, we picked a recent, high-profile vulnerability, the buffer overflow in *wu-ftpd*[CERT-ftpd], and showed how our implementation of DTE can prevent an attacker from obtaining a root shell. Our goal was to show that we could protect the system from the *wu-ftpd* vulnerability (the posted exploits as well as future or hand-crafted ones) without modifying the binary. In order for ftp to retain its full functionality, it would need to be made DTE-aware so that it could, like login, allow ftp

# entry points	Clock cycles for `sys_dte_exec`
2	22692 ± 184
4	22777 ± 133
6	23186 ± 269
8	23432 ± 256
10	24099 ± 336
12	23946 ± 123
14	24367 ± 238
16	24503 ± 120
18	24841 ± 125
20	24978 ± 133
30	25282 ± 142
30 (entry point last)	32427 ± 259

Figure 8: Dependence of `sys_dte_exec` performance on number of entry points.

	Non-DTE	DTE
First Execution	4221290 ± 911041	4545504 ± 730168
Subsequent executions	195591 ± 3919	215549 ± 4624

Figure 9: Time in clock cycles to run *echo*.

to transition into the domain associated with a user being authenticated[5]. We did not do this, but set protections such that users can retrieve files from, if not deposit files onto, the server. Anonymous ftp is fully functional.

The policy shown in Figure 10 prevents domain *ftpd_d* from executing any system binaries other than `/usr/sbin/in.ftpd` and binaries located under `~ftp/bin/` (lines 19-21). These files are defined to be of the type `ftpd_xt` (lines 29 and 30), which the domain *ftpd_d* may execute but not write (line 20). Only *ftpd_d* may execute this type (lines 9-21), and *root_d* automatically switches to *ftpd_d* on execution of `/usr/sbin/in.ftpd` (line 12), since that is an entry point to *ftpd_d* (line 19). The exploits to be found on the internet to take advantage of this vulnerability will therefore fail, as they expect to be allowed to run `/bin/sh`. Nor can a script be written to upload and run a Trojan horse, since the only types which *ftpd_d* is allowed to write may not be executed by anyone.

The script which we tested was `wuftpd2600`, which can be found at `http://www.securityfocus.com`. It connected to our test machine, and exploited the buffer overflow. However, the DTE-enabled kernel refused to allow the *ftpd_d* domain to execute `/bin/sh`. The script therefore hung, and the system was not compromised. The error messages in Figure 11 were sent to *syslog*. In contrast, the plain `2.3.28` kernel happily provided a root shell.

[5]Of course, to do this on a system which we are attempting to make secure, we would begin by using a version of ftp which does not send plaintext passwords.

7 Status and Future Work

Our implementation of DTE for Linux is functional. It reads a policy file at boot time and enforces domain to type access as well as domain transitions. We have not implemented DTE for networking.

7.1 Administration

First, we must extend our policy parser to allow easier and abbreviated entry of more

```
01 # ftpd protection policy
02 types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
03       config_t ftpd_t ftpd_xt w_t
04 domains root_d login_d user_d ftpd_d
05 default_d root_d
06 default_et root_t
07 default_ut root_t
08 default_rt root_t
09 spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t rwxcd-
>spool_t \
10       rwcdx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t rwxcd-
>passwd_t \
11       rxwcd->shadow_t rwxcd->dev_t rwxcd->config_t rwxcd->w_t) (auto-
>login_d \
12       auto->ftpd_d) (0->0)
13 spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd-
>spool_t \
14       rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t rwxcd-
>dev_t \
15       rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17-
>0)
16 spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxd->root_t \
17       rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd-
>shadow_t \
18       rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17-
>0)
19 spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwcd->ftpd_t rd->user_t rd-
>root_t \
20       rxd->lib_t r->passwd_t r->shadow_t rwcd->dev_t rd->config_t rdx-
>ftpd_xt \
21       rwcd->w_t d->spool_t) () (14->root_d 17->root_d)
22 assign -u /home user_t
23 assign -u /tmp spool_t
24 assign -u /var spool_t
25 assign -u /dev dev_t
26 assign -u /scratch user_t
27 assign -r /usr/src/linux user_t
28 assign -u /usr/sbin binary_t
29 assign -e /usr/sbin/in.ftpd ftpd_xt
30 assign -r /home/ftp/bin ftpd_xt
31 assign -e /var/run/ftp.pids-all ftpd_t
32 assign -r /home/ftp ftpd_t
33 assign -e /var/log/xferlog ftpd_t
34 assign -r /lib lib_t
35 assign -e /etc/passwd passwd_t
36 assign -e /etc/shadow shadow_t
37 assign -e /var/log/wtmp w_t
38 assign -e /var/run/utmp w_t
39 assign -u /etc config_t
```

Figure 10: A DTE policy to protect from *wu-ftpd*, with line numbers added.

```
Aug  4 13:12:03 wicked kernel: do_exec: d_t_check_x re-
turned 1(exec denied).
Aug  4 13:12:03 wicked kernel: do_exec: domain ftpd_d type root_t.
```

Figure 11: Error messages resulting from attempted *wu-ftpd* exploit.

complicated policies. Next, we plan to create tools to help a system administrator graphically create and view DTE policies and detect possible security risks. Examples of such risks might include a domain which is permitted to enter another domain as well as write one of the other domain's entry points, or a domain which has *auto* access to two domains which share an entry point.

7.2 Rename

Badger et al.[DTE95] suggest dynamically changing the DTE policy as certain events occur. For example, a particular file (`/var/adm/topsecretlog`) might be tightly protected by a particular type. If this file is then moved to `/tmp`, then Badger et al. suggest that a rule should be added to keep the file under its original type. Alternatively, they suggest that renames across type boundaries could be forbidden.

We currently go the lazy route. If a domain has permission to two types, and a process running in that domain chooses to move a file from a directory belonging to one type to that belonging to another, then the file's type simply changes. Since the person (or process) moving the file had the permission to do so, we trust it to understand the implications.

The more dangerous problem lies with hard links. Since hard links provide no notion of one name being superior to another, the type of an inode with multiple corresponding filenames is currently determined based upon the name first looked up.[6] We can prevent creation of hard links across type boundaries, however a change in policy can thwart this defense quite easily. We will, in future versions, allow the system to add its own type assignment rules (which will be necessary for several other desirable features), and plan to use this capability to implement a better resolution of the problem with hard links.

[6]Note there is no analogous problem with soft links, since these do provide a notion of a single correct pathname.

7.3 Filesystem-Defined Policies

When a partition is mounted into the filesystem tree, it fits into the tree defined by type rules depending on where it is mounted. For example, mounting a partition under `/tmp` instead of `/mnt` might completely change access permissions to the partition. It seems helpful to allow the filesystem on a partition to specify certain type assignment rules which apply only to the partition. Badger et al[DTE96] also added a DTE configuration section allowing a DTE administrator to limit mount points for partitions, which should be trivial for us to add as well.

8 Availability

DTE for Linux is freely available as a patch to `2.3.28` at `http://www.cs.wm.edu/~hallyn/dte`.

References

[AC-sum] R. Sandhu, *Access Control: The Neglected Frontier*, First Australasian Conference on Information Security and Privacy, 1996.

[Caps-faq] Alexander Kjeldaas, *Linux Capability FAQ v0.1*, `http://www.uwsg.indiana.edu/hypermail/linux/kernel/9808.1/0178.html`, (1998).

[CERT-ftpd] *Cert Advisory CA-2000-13: Two Input Validation Problems in FTPD*, `http://www.cert.org/advisories/CA-2000-13.html`.

[Dennis66] Jack B. Dennis and Earl C. Van Horn, *Programming Semantics for Multiprogrammed Computations*, Communications of the ACM, March 1966, pp. 143-155.

[DTE95] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker and Sheila

A. Haghighat, *A Domain and Type Enforcement UNIX Prototype*, Fifth USENIX UNIX Security Symposium Proceedings, Salt Lake City, Utah, June 1995.

[DTE96] 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)*, Sixth USENIX UNIX Security Symposium, 1996.

[SAIC-DTE] *Domain and Type Enforcement*, `http://research-cistw.saic.com/cace/dte.html/`.

[Tidswell97] Jonathon Tidswell and John Potter, *An Approach to Dynamic Domain and Type Enforcement*, 2nd Australasian Conference on Information Security and Privacy, 1997.