TRON:
Process–Specific File Protection for the UNIX Operating System

Andrew Berman, Virgil Bourassa and Erik Selberg
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The file protection mechanism provided in UNIX is insufficient for current computing environments. While the UNIX file protection system attempts to protect users from attacks by other users, it does not directly address the agents of destruction—executing processes. As computing environments become more interconnected and interdependent, there is increasing pressure and opportunity for users to acquire and test non–secure, and possibly malicious, software.

We introduce TRON, a process–level discretionary access control system for UNIX. TRON allows users to specify capabilities for a process' access to individual files, directories, and directory trees. These capabilities are enforced by system call wrappers compiled into the operating system kernel. No privileged system calls, special files, system administrator intervention, or changes to the file system are required. Existing UNIX programs can be run without recompilation under TRON–enhanced UNIX. Thus, TRON improves UNIX security while maintaining current standards of flexibility and openness.

1. Introduction

This paper describes the design and implementation of TRON[1], a process–specific file protection mechanism for the UNIX[2] operating system. TRON puts flexible, easy–to–use discretionary access control in the hands of the user, giving him the power to monitor and prevent "Trojan Horse" attacks and other undesirable file accesses. No special accounts, magic files, or actions by the system administrator are required.

TRON works in conjunction with the existing UNIX file protection mechanism. No recompilation of binary executables is required, and processes not protected by TRON incur no significant performance penalty. TRON facilitates protected cross–domain procedure calls by allowing client processes to temporarily grant their rights to server processes. TRON also accommodates various policies for handling access violations.

TRON allows users to express their intended process accesses in a natural, succinct fashion. Rights can be specified for files, directories and directory trees, to correspond to the structure of the UNIX file system. Invocation of access restrictions is separated from invocation of processes, providing reuse and flexibility.

2. Motivation

The user–oriented, access control list file protection mechanism provided in UNIX attempts to protect user's files from other users. But this is an incomplete view of file security. Files are not affected by users directly, but indirectly through the execution of programs.

As a result, UNIX files are susceptible to attack by the user on himself. Once a user logs in, his processes have all rights afforded to the user, and thus incorrect or malicious programs (e.g. Trojan Horses, computer virus-

---

1. Named after the heroic fictional protection program from the 1982 Walt Disney movie, *TRON*.
2. UNIX is a registered trademark of X/Open Company Limited.

es, etc.[5]) as well as unintentional mistakes (e.g., typing "`rm -fr *`" in the wrong directory) executed by the user can perform actions the user did not intend nor foresee.

This susceptibility is exacerbated by the necessary and common method of granting temporary privileges in UNIX, namely, `suid` (set user id) and `sgid` (set group id). These commands allow a program's user to temporarily gain the rights of the program's owner. Such privileges have successfully been exploited to gain undesired access to user accounts and files by other users[12].

A common security technique to enhance UNIX file protection for programs such as `ftp` is to use the privileged command, `chroot`, to "change the root," causing the visible directory hierarchy to be replaced by a safe subdirectory. For `ftp`, this makes utilities such as `ls` unreachable, so copies must be maintained in the safe subdirectory. An alternative requiring less maintenance would be to use the existing file structure while restricting the rights given the `ftp` daemon process—namely, full rights to the safe subdirectory combined with read and execute rights for necessary utilities. The utilities, when executed, would likewise require limited rights.

Although an individual user may have access rights to a large variety of files, restricting the rights granted to individual processes to those believed to be germaine can prevent abuse. We wish to provide UNIX users with a service that limits file access on a per–process basis, while still allowing the temporary extension of rights as necessary.

3. Design

TRON provides process–specific file protection to complement the existing UNIX user–specific file protection mechanism. TRON provides a means of executing processes within protected *domains*, in which the process has a restricted set of rights to files and directories, specified by a set of *capabilities*. Enforcement of rights is performed in the kernel to ensure security.

When a user first logs on, he has his full, normal UNIX file permissions by default[1]. At any time, the user may elect to execute processes in a more restrictive TRON domain. As a complementary protection mechanism, TRON does not allow a process to perform an action that would violate normal UNIX file permissions. Conversely, even processes with super–user privileges are restricted when executing in a protected TRON domain.

3.1. Capabilities

For protection, an agent requires access rights to a resource. A list of access rights associated with a resource is called an *access control list*[17]. Alternatively, a list of access rights associated with an agent is called a *capability list*[6]. The design of TRON is patterned after capability–list systems in that the access rights are associated with the agents of access, processes.

Unfortunately, UNIX does not lend itself to a traditional capability–based approach[15], in which each resource is identified uniquely. In UNIX, files and directories can have many names because of directory links and mounted file systems. For this reason, we have chosen to make TRON a "character string" protection service, meaning TRON capabilities are expressed as rights associated with a character string representing an object. An object may have several such character strings naming it, but TRON is unaware of the connection between them.

The character string representing a file or directory object is its *canonical* pathname. The canonical pathname is the character string representing the absolute, rooted path of the file or directory name. When not specified directly in canonical form, the pathname of a file or directory is determined relative to the current working directory.

TRON currently understands two types of objects—files and directories. The rights associated with an object are dependent upon the object's type, and have meaning only in the context of accesses related to that type. This generality may allow the concept of TRON

1. Although system administrators are free to configure either or both of the login process and initial console shell to execute within restricted TRON domains as well.

capabilities to be extended to resources beyond files and directories, e.g., sockets.

Rights that can be specified for file access include: *read*, *write* (does not imply the right to delete the file), *execute*, *delete*, and *modify* UNIX permissions (e.g., with `chmod`)[1]. To aid in specifying rights for several files at once, directory capabilities perform dual duty—they specify rights for a directory and the files contained within it. In addition to the file–specific rights, which apply to all files within the directory, directory rights include: *create* new files in the directory and create *link*s to files in other directories. *Link* also requires full rights to the file being linked to. A final directory right, *subtree*, extends the rights to the directory's subdirectory tree. The *subtree* right conveys the other directory rights to all objects with canonicalized names beginning with the name of the directory.

The determination of whether an object is a file or a directory is made automatically at the time of creating the capability. We require that the named object exist at this time.

3.2. Domains

TRON protection domains provide a protection environment for every process. A TRON domain consists of a set of processes, a set of capabilities, and a violation handler. Each process executes in exactly one TRON domain. A normal `fork` operation causes a newly created process to inherit its parent's TRON domain. A modification of the `fork` operation, *`tron_fork`*, creates a new TRON domain containing a subset of the parent's TRON domain capabilities, in which to run the new process.

We initially considered providing more flexible TRON domain creation by using a separate call, *`tron_restrict_domain`*, to be used immediately following the `fork` command to create and enter a restricted domain, but concluded that combining the operations into the single `tron_fork` call provided overriding benefits. Because `tron_fork` first verifies that the newly created TRON domain will contain a strict subset of rights prior to creating the child process, failure of the domain creation does not result in a new child process with more rights than the parent process intended, nor does the parent process have to rely on this child process for notification of the failure. Additionally, combining the domain creation with the process creation ensures that each process exists within a single, invariant TRON domain throughout its lifetime, facilitating both simple garbage collection of TRON domains as well as the ability to loan capabilities to other processes (see 3.4 below).

We likewise considered another alternative, `tron_create_domain`, to be called by the parent process prior to performing the `fork`. However, conveying the identity of the newly created TRON domain to the `fork` operation requires either a change to the existing interface, breaking all existing code making use of `fork`, or introducing a new global state variable to handle what is essentially a special case, adding to the process state clutter. In addition, a simple reference counting method for automatic garbage collection of TRON domains does not suffice for this alternative. Thus, we decided that the `tron_fork` functionality was the most appropriate for our design.

A special TRON domain, called the *full* TRON domain, gives full rights to all files and directories, effectively limiting access to the user's existing UNIX file permissions. This is the TRON domain of the first processes entered into the process table, namely `init`, `pager`, `swapper`, and `idleproc`. By default, progeny of these processes continue to inherit the full TRON domain until a `tron_fork` is performed. Thus, until activated, TRON does nothing beyond the normal functioning of the UNIX file protection mechanism. Therefore, all processes executed up to and including the first process performing a `tron_fork` are run with the full set of default UNIX rights.

3.3. Enforcement

TRON enforces domain–specific access rights from within the UNIX kernel. Placing TRON access enforcement within the kernel both ensures the security of the enforcement and obviates the need for recompiling existing executables. TRON access enforcement is localized to the *syscall* table, from which all system calls are dispatched upon entry into

1. Our set of rights is inspired by those of the Andrew File System[11].

the kernel. TRON *wrappers* are placed around all pertinent system call operations (e.g. `open`, `chmod`, etc.). The syscall table is modified to vector control to the appropriate TRON wrapper for each system call. These wrappers perform the general algorithm given in Figure 1.

The enforcement algorithm directly invokes the desired kernel service for processes in full TRON domains. Limited TRON domains are examined for the necessary access rights to the desired service routine, invoking the service if found. Otherwise, a violation handler is called instead of the desired service routine. For processes not employing the TRON protection mechanism (i.e., in the full TRON domain), the performance impact of the additional if–test and procedure call is not significant relative to the cost of the kernel trap itself.

Note that it is possible for a file or directory to be interpreted with one canonicalized pathname at the time a capability is created, and a different canonicalized pathname at the time of enforcement, if either of both of the specifications are relative to the current working directory (due to links, mount points, etc.). When the names don't agree, the access attempt fails. This is a conservative solution to the name ambiguity problem inherent in the UNIX file system.

We envision the violation handlers to take on varied forms depending on the requirements of the computing site. Potential handler actions include soft responses, such as setting the offending process' global `errno` value to `EACCES`, or hard responses, such as killing the offending processes with a specific exit value. Logging and user intervention are other actions that may be taken. Because violation handlers are invoked from within the kernel, they must be built into the kernel itself, with policies established for their interaction with user–level processes.

3.4. Granting Capabilities

Our mechanism for temporarily granting access to another process consists of two system calls executed by the grantor of the capabilities, *`tron_grant`* and *`tron_revoke`*. This mechanism allows protected cross–domain procedure calls that are transparent to the serving process. The `tron_grant` call from a client process instructs the kernel to extend a given server process' TRON domain to include a subset of the client's capabilities. The extended capabilities are marked as revocable by the granting process. Once received, the server process, as well as any other process within the server process' TRON domain, has the extended access. In no case does the granting of capabilities provide file access to the server beyond the existing UNIX permission system. The extended TRON domain then allows servers to fork off copies of themselves to perform the desired service and still retain the necessary capabilities. By judicious creation of the server's TRON domain, it's straightforward to ensure that unintended processes do not share in the granted rights.

An additional system call, *`tron_get_cap_list`*, is available for determining the current domain capabilities. Processes are free to grant any or all of their

```
tron_foo_wrapper( foo_arguments )
{
   if ( tron_domain[ process ] ≠ FULL_TRON_DOMAIN ) {
      determine required_rights for foo( foo_arguments );
      if ( required_rights ⊄ capabilities[ tron_domain[ process ]] ) {
      invoke violation_handler[ tron_domain[ process ]];
         return from kernel;
      }
   }
   invoke foo( foo_arguments );
   return from kernel;
}
```

Figure 1. General System Call Wrapper Algorithm.

capabilities to other processes. Multiple capabilities for the same object granted to the same process are each individually added to the receiving process' TRON domain—any one of these providing the necessary rights suffices for access to the object.

The `tron_revoke` call removes all capabilities from a TRON domain that are marked as revocable by the calling process. This function serves more as a cleanup mechanism than as a security mechanism. We take the stance that the granting of a capability to another agent implies trusting that agent to be responsible with it. We use the `tron_revoke` call merely to keep servers' TRON domains from accumulating an indefinitely large number of capabilities.

The `tron_revoke` call is not transitive, e.g., if process A grants capability α to process B, which subsequently grants α to process C, then process A revoking its granted rights from process B does not imply revocation of α from process C. It is incumbent upon each direct grantor to explicitly revoke their granted rights prior to terminating. Thus, using `tron_grant` without a corresponding `tron_revoke` provides a capability hand–off mechanism.

A user–level command, *`tron_loan`*, provides for transparent capability granting and subsequent revocation on behalf of existing TRON–unaware client executables. The `tron_loan` command is executed from within an existing TRON domain. It first grants a specified subset of the TRON domain capabilities to an executing server process, then forks and executes the desired client program within the original domain. Upon completion of the client process, `tron_loan` revokes the granted capabilities from the server process. In other words, the `tron_loan` command acts as a proxy for a new client process to temporarily loan needed rights to a server process.

Although the `tron_loan` mechanism has some shortcomings for TRON–unaware executables, such as for servers acting as clients of other servers, it does not preclude the transparent use of TRON in the general case. Rather the `tron_loan` command provides an extension of the normal TRON protection scheme for common cases by allowing server processes to maintain a minimal set of capabilities, while acquiring others as needed. We hope that with experience more general mechanisms will come to light.

4. Implementation

We have successfully implemented a prototype of the TRON service in ULTRIX V4.2A, running on a DECstation 5000/200[1]. The impact on the existing system call code path is minimal when the protection is not in force and reasonably inexpensive when it is in force (on the order of the number of capabilities in the process' TRON domain). Modification of the existing ULTRIX source code is limited to small changes to a handful of files. Less than 2000 lines of new code are introduced into the kernel, while roughly 800 lines of code are introduced at the user level in the form of library calls and shell commands.

TRON functionality extends from user–level commands to internal kernel procedures. User commands consist of `tron` and `tron_loan`; system calls consist of `tron_fork`, `tron_grant`, `tron_revoke`, and `tron_get_cap_list`; and internal kernel procedures consist of the various TRON wrappers, `tron_canonical`, `tron_verify`, and `tron_soft_violation_handler`.

4.1. User Commands

The user commands introduced for TRON are summarized in Figure 2.

The `tron` command creates a TRON domain in which the given command is executed with the rights speci-

1. ULTRIX and DECstation are trademarks of Digital Equipment Corporation.

```
% tron [-p <rights_spec> [<file>|<dir>]...]... [-c <command_line>]
% tron_loan <pid> [-p <rights_spec> [<file>|<dir>]...]... [-c <command_line>]
% alias stdtron 'tron -p rx $path ~ / -p rs /etc \!*'
```

Figure 2. TRON User Command Summary

fied. If no <command_line> is given, TRON invokes a shell (using the SHELL environment variable or /bin/sh if not found) in the new TRON domain. Note that the TRON domain specified must contain the rights to read and execute the given command. File and directory names are put into canonical form, verified to exist, and typed as a file or directory automatically. A <rights_spec> applies to all file and directory names encountered before the next -p or -c flag. The <rights_spec> is any combination of the following letters, which convey the associated rights:

r: read	m: modify UNIX permissions
w: write	c: create
x: execute	l: link
d: delete	s: subtree

In practice, we have found the (c–shell) alias given in Figure 2, stdtron, to be helpful for common uses of tron. It conveys, at a minimum, read and execute rights to files on the search path, home, and root directories, and read rights to the /etc subtree.

The tron_loan command does not create a new TRON domain, but rather grants a subset of capabilities of the current TRON domain to the <pid> process' TRON domain, then forks the specified command within the current TRON domain. The capabilities are verified to be a subset of the current capabilities and, when granted, are marked as revocable by the tron_loan process. Like tron, when no <command_line> is given, tron_loan invokes a shell process. When the command or shell terminates, the capabilities are revoked from the <pid> process' TRON domain.

4.2. System Calls

The additional system calls introduced for TRON are summarized in Figure 3.

The tron_fork system call verifies that the capabilities presented are a subset of the parent's TRON domain capabilities. If so, it creates a new TRON domain containing these capabilities and forks a new process within the new TRON domain. If the capabilities are incorrect, an error code is returned to the parent process and no fork is performed.

Like tron_fork, the tron_grant system call verifies that the capabilities presented are a subset of the process' TRON domain capabilities. If so, it appends the capabilities to the TRON domain containing the pid process, marking them as revocable by the current process.

The tron_revoke system call removes all capabilities marked as revocable by the current process from the TRON domain containing the pid process. If the pid process and the current process are identical no action is taken.

The tron_get_cap_list system call returns up to buf_size of the process' current capabilities to its cap_list_buffer, and returns the total number of capabilities in the TRON domain in num_caps.

4.3. Kernel Modifications

The modifications to the original kernel source are simple and straightforward. A TRON domain index is added to the process structure[1], fork is modified to copy this value, and exit is modified to clean up the TRON domain memory when no longer in use. Some initialization code is added to init_main.c. Finally, as alluded to earlier, syscall.h, syscalls.c, and init_sysent.c are modified to vector the appropriate system calls to the correspond-

1. The addition of a field to the process structure was done for convenience in the prototype. A separate array of TRON domain indices to shadow the process table would allow existing system programs that depend on proc.h to run without recompilation.

```
tron_fork( cap_list, num_caps );
tron_grant( pid, cap_list, num_caps );
tron_revoke( pid );
tron_get_cap_list( cap_list_buffer, buf_size, num_caps );
```

Figure 3. TRON System Call Summary

```
tron_canonical( filename );
tron_verify( object_name, required_rights );
tron_soft_violation_handler( object_name, required_rights, trap_num,
                             error_status, return_value );
```

Figure 4. TRON Kernel Function Summary

ing TRON wrappers.

The TRON wrapper procedures are straightforward applications of the algorithm described in section 3.3. In addition to the TRON wrappers, the support functions summarized in Figure 4 have also been added to the kernel.

The `tron_canonical` procedure is used to normalize file or directory names passed as arguments to system calls. It returns the canonical pathname of the given `filename`. The current working directory determination exactly mirrors the effective functionality of the user command, `pwd`. Local and remote mount points, hard and soft links, etc., are handled as they would be by the individual system calls made by `pwd`.

The `tron_verify` procedure is used to determine if the required rights are permitted by some capability in the current process' TRON domain. It searches through capabilities of the current process' TRON domain for any that convey the `required_rights` (a bit field with the corresponding bits for the required rights set to one). This includes capabilities for parent directories with the subtree right.

The `tron_soft_violation_handler` procedure is our prototype access violation handler. This procedure sets the current process' global `errno` value to `error_status` (generally set to `EACCES`) and the system call return value to `return_value` (typically `-1`). It does no reporting of the access violation.

5. Examples

TRON usage examples are given in Figure 5. Example A runs a normal shell, but prevents deletion or writing of any files.

Example B runs a shell that allows deletion and writing in your current directory and `/tmp`. Note that rights are given to the current directory at the time the command is given, so the rights don't "float" if the current directory changes.

Example C allows use of `emacs` in the background to edit files only in the current directory (assuming all `emacs` support files are in the `/usr/local/emacs` subdirectory).

Example D creates a TRON domain to hold the finger daemon, permitting normal execution of its functions. This would have prevented the Internet Worm from taking advantage of the bug in `fingerd` to infiltrate systems[18]. Similar TRON domains can be created for each daemon in `/etc/inetd.conf`.

If users wished to have their .plan and .project files accessible to the finger daemon only when they are logged in and are alerted of their being "fingered," this

```
A) % tron -p rxmcls /
B) % tron -p rxmcls / -p wd . /tmp
C) % stdtron -p rwcds /usr/local/emacs -p rw . -c emacs myfile.tex &
D) # tron -p rs / -p x /usr/ucb/finger /usr/etc/fingerd -p w /usr/adm/daemon.log
   -c fingerd &
E) # tron -p rx /usr/ucb/finger /usr/etc/fingerd -p rw /usr/adm/daemon.log -c
   fingerd &
F) % ps -ax | egrep fingerd | egrep -v egrep
   20960 ?  I      0:04 fingerd
   % tron_loan 20960 -p r ~/.{plan,project}
```

Figure 5. TRON Usage Examples

could be facilitated by tron_loan. As shown in Example E, the super–user initializes the finger daemon without the pervasive read rights given in the previous example.

Upon logging in, the user can temporarily grant rights to read their .plan and .project to the finger daemon using `tron_loan` as depicted in Example F. In this example, we have created a normal shell that, while it exists, grants the rights to read the .plan and .project files to the finger daemon. (Obviously, these steps could be automated in the user's .login using, for example, `awk`.) Another user, attempting to `finger` a user who is not logged in is denied permission to read their .plan and .project files.

6. Performance Impact

Two benchmarks were run to determine the impact of the TRON domains to existing UNIX performance. The first benchmark determines the time to fork a null thread, the second to open a file, write 1000 bytes, and close it. The results are summarized in Table 1. The table shows average times, in microseconds, for the *forktest* and *filetest* benchmarks, tested without TRON (i.e., standard ULTRIX distribution), with TRON using the full TRON domain, using the stdtron TRON domain with 15 capabilities, using a TRON domain with 50 capabilities, and, finally, using a TRON domain with 80 capabilities. The timing values are averaged over 10,000 trials. In the restricted domain tests, the required capability is found at the end of a linear search.

The results are fairly telling. The cost to verify a capability, even when using an inefficient linear search mechanism, has negligible impact upon system performance in comparison to the larger costs of the system call and I/O.

7. Related Work

Capabilities and access control lists are well–known topics in operating systems design[6][17]. There have been several papers on the subject of strengthening UNIX security by adding capabilities or access control lists. The most commonly suggested method is to grant users access control over files[8][16][19]. Access control lists of this type have several drawbacks. Since users are persistent, the lists must be persistent and stored on disk in protected files. If one user runs a program owned by another user, functionality and security requirements force both users to have appropriate access control list entries in all relevant files, and possibly delete entries after each use. Users are thus required to perform a lot of maintenance on their access control lists. Furthermore, these systems offer no protection for the user against programs owned by the user, which may contain errors, Trojan Horses, or viruses.

Lampson [14] points out that users must be protected against activities by their own programs as much as by other users. Wichers, et.al. [20], propose an access control system, PACLs, in which rights are granted to programs rather than users. This method protects somewhat against viruses and Trojan Horses, since they must have the correct program name to work their effects. However, their system requires extensive effort by the user. Among other things, special files must be maintained, and the user must enter a password to perform actions such as removing a file or editing the control lists. To save maintenance time, they allow users to create files that are not constrained by PACLs, or to temporarily disable PACLs. The danger is that coupling extensive maintenance requirements with the ability to completely opt out of the security mechanisms can lead to reduced use of these mechanisms.

test	without TRON	full rights	stdtron (15 capabilities)	50 capabilities	80 capabilities
forktest	3388	3438	3625	3699	3719
filetest	33726	33648	33730	33786	33717

Table 1: Performance Benchmarks

Lai & Gray [13] propose a system that is similar to TRON. In their system, processes are granted rights to files. Processes reside in Untrusted Process Families (UPFs), which contain lists of capabilities. Child processes inherit their parents' UPF, or can be placed in a new UPF that contains a subset of the capabilities of the parent's UPF. This is the paradigm used in TRON. Unlike files and users, processes are temporary. Thus, process capabilities are temporary, reducing maintenance on the part of the users. Furthermore, a process is a more fine–grain construct than a program or a user. A single user can simultaneously have more non–interfering "sets" of capabilities at the process level than at the program or user level.

However, the Lai & Gray approach suffers from an inflexible and often incorrect capability determination mechanism, which strongly restricts the types of programs that can be protected. The system grants capabilities to processes based on their argument lists and to temporary files they create. The actions of processes on files named as arguments are unrestricted, which may not be the user's intent. Also, the common use of permanent files other than those passed as arguments is disallowed. Furthermore, program arguments are often not filenames, but in this system must be treated as such. Finally, the approach has no means of indicating that an argument specifies a directory tree, making it impossible to protect programs, such as `ls`, `find` and `rm,` that recursively access directories[7].

Programs that do not cooperate with the capability determination scheme must opt out of running in a protected domain. To compensate, Lai & Gray propose a cumbersome mechanism for specifying that such programs are "trustable," involving intervention by a system administrator to validate the program, then mark a bit in the program's inode structure. Ultimately, this system suffers from the same problem as PACLs in combining a difficult to use and intrusive protection facility with a method of completely opting out of the system.

8. Contributions

Increasing UNIX security while maintaining current standards of flexibility and openness has been a long–standing research and industry goal. We feel that the integration of TRON moves UNIX a long way towards meeting this goal.

By using process–based capabilities we restrict the activities on files at a finer granularity than user or program–based capabilities. As Lai and Gray point out in [13], the goals of computer users are effected through process invocation, and therefore it is the user's intended *process* accesses that should be delineated and enforced. Unlike Lai & Gray's system, however, we leave it to the user to directly convey their intended process accesses. Because we create new protection domains as part of the UNIX forking mechanism (`tron_fork`), rather than as part of the UNIX `exec` mechanism, we don't run into their necessity to guess the required capabilities at program execution time.

TRON enables process–based discretionary access control with practically no extra maintenance on the part of the user or system administrator. No changes to the file system or special files are required. While the kernel must be recompiled, all existing user–level programs are binary compatible[1]. The new TRON user commands and system calls are available to all users, without requiring super–user privileges.

We feel that our system of naming file capabilities is powerful and correct for directory–tree–based file systems such as UNIX. TRON employs directory capabilities that apply to all files within a directory, and an optional *subtree* right allowing the ready extension of capabilities to entire directory subtrees. This scheme is flexible enough to specify large portions of the file system with just a few capabilities.

Perhaps our most important contribution is that we have created a system that is powerful yet easy to operate and understand. Users can take advantage of the TRON service with a minimal amount of effort. And ultimately, the success of any computer security system is dependant on the willing participation of the system's users.

---

1. System programs that include `proc.h`, such as `ps`, must be also be recompiled in our prototype implementation, but this can be avoided as discussed in the footnote of section 4.3.

9. Conclusion

We have argued that a process–specific, expressive, and easy–to–use service is an effective means of providing protection from Trojan Horses, computer viruses, etc. We have presented the design of TRON as an extension to the UNIX operating system that provides such a service in a safe, transparent and flexible manner. We have successfully integrated this service into an existing UNIX implementation and demonstrated its usefulness. Finally, we have shown how our service relates to previous efforts to strengthen UNIX security and provides additional benefits beyond those efforts.

10. Acknowledgments

We'd like to thank Hank Levy, Jeff Chase, Alex Klaiber, Dylan McNamee and Alec Wolman for their guidance and advice in the development of TRON. Thanks also to Debbie Berman for editing advice. And additional thanks to Paul Barton–Davis for introducing us to the lore of malicious programs.

11. References

[1] Bershad, B. N. and Pinkerton, C. B., "Watchdogs—Extending the Unix File System," *Computing Systems*, vol. 1, no. 2, pp. 169-88, Spring 1988.

[2] Candia, T., "Unix Security Myths and Truths," *EDPACS*, vol. 20, no. 6, pp. 8–13, December 1992.

[3] Carson, M. E. and Wen, Der Jiang, "New Ideas in Discretionary Access Control," *Proceedings: UNIX Security Workshop*, Portland, OR, USA, pp. 35–7, USENIX, August 29–30 1988.

[4] Davida, G. I. and Matt, B. J., *Unix Guardians: Delegating Security to the User*, USENIX, Berkeley, CA, USA, 1988.

[5] Denning, Dorothy E., *Cryptography and Data Security*, Addison–Wesley Publishing Company, Reading, MA, 1982.

[6] Dennis, Jack B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, no. 3, pp. 143–155, March, 1966.

[7] Duff, Tom, "Viral Attacks On UNIX System Security," Proceedings, 1989 Winter USENIX Technical Conference, San Diego, CA, pp. 165–171, USENIX, January 30–February 3, 1989.

[8] Fernandez, G. and Allen, L., "Extending the UNIX Protection Model with Access Control Lists," *Proceedings* of *USENIX*, San Francisco, CA, USA, USENIX Assoc., Berkeley, CA, USA, Summer 1988, pp 119–132.

[9] Hardy, N., "The Confused Deputy (or Why Capabilities Might Have Been Invented)," *Operating Systems Review*, vol. 22, no. 4, pp. 36–8, October 1988.

[10] Heydon, A., and Tygar, J. D., "Specifying and Checking UNIX Security Constraints," *Computing Systems*, vol. 7, no. 1, pp. 91–112, Winter 1994.

[11] Howard, John H., Kazar, Michael L., Menees, Sherri G., Nichols, David A., Satyanarayanan, M., Sidebotham, Robert N., West, Michael J., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51-88, February 1988.

[12] Kaplan, R., "SUID and SGID Based Attacks on UNIX: a Look at One Form of the Use and Abuse of Privileges," *Computer Security Journal*, vol. 9, no. 1, pp. 73–7, Spring 1993.

[13] Lai, Nick, and Gray, Terence E., "Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses," *Proceedings* of *USENIX*, San Francisco, CA, USA, USENIX Assoc., Berkeley, CA, USA, Summer 1988, pp. 275–86.

[14] Lampson, Butler W., "Protection," *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437–443, reprinted in *Operating Systems Review*, vol. 8, no. 1, January 1974, pp. 18–24

[15] Levy, Hank, *Capability Based Computer Systems*, Digital Press, 1984.

[16] Low, Marie Rose, and Christianson, Bruce, "Fine Grained Object Protection in Unix," *Operating Systems Review*, vol. 27, no. 1, pp. 33–50, January 1993.

[17] Organick, E.I., "The Multics System: An Examination of Its Structure. MIT Press, Cambridge MA, USA 1972

[18] Seeley, Don, “A Tour Of the Internet Worm,” Conference Proceedings 1989 USENIX Technical Conference, USENIX Assoc., Berkeley, CA, USA, pp. 287–304.

[19] Strack, H., “Extended Access Control in UNIX System V—ACLs and Context,” *USENIX Workshop Proceedings on UNIX Security II*, Portland, OR, USA, pp. 87–101, USENIX Assoc., Berkeley, CA, USA, August 27–28 1990.

[20] Wichers, D. R., Cook, D. M., Olsson, R. A., Crossley, J., Kerchen P., Levitt, K. N., and Lo, R., *PACL's: An Access Control List Approach to Anti–Viral Security*, USENIX Assoc., Berkeley, CA, USA.

[21] Wobber, E., Abadi, M., Burrows, M., and Lampson, B., “Authentication in the TAOS Operating System,” *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 5-8, 1993.

Andrew Berman (aberman@cs.washington.edu) is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Washington, Seattle, Washington. His research interests include algorithm design, data structures, and computer security. He received his A.B. in computer science from Princeton University in 1988.

Virgil E. Bourassa (virgil@cs.washington.edu) is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Washington, Seattle, Washington. His research interests include computer operating systems and architectures. He joined Boeing in 1988 and now works as a scientist in the Computer Science organization of the Computer Services Division in Bellevue, Washington. He received his BS in electrical engineering from Arizona State University, Tempe, Arizona, in 1987 and his MS in electrical engineering from the University of Washington, Seattle, Washington in 1990.

Erik W. Selberg (selberg@cs.washington.edu, http://www.cs.washington.edu/homes/speed) is pursuing his Ph.D. in computer science from the University of Washington, Seattle, Washington. His research interests include systems issues concerning security, integrity, and authentication, as well as artificial intelligence issues involving multi–agent coordination and planning. Currently he is investigating methods for conducting commerce on the Internet using software agents. He graduated from Carnegie Mellon University in 1993 with a double major in computer science and logic, and received the first Allen Newell Award for Excellence in Undergraduate Research.