Trusted Path Execution for the Linux 2.6 Kernel as a Linux Security Module

Niki A. Rahimi
IBM Corporation

Abstract

The prevention of damage caused to a system via malicious executables is a significant issue in the current state of security on Linux operating systems. Several approaches are available to solve such a problem at the application level of a system but very few are actually implemented into the kernel. The Linux Security Module project was aimed at applying security to the Linux kernel without imposing on the system. It performs this task by creating modules that could be loaded and unloaded onto the system on the fly and according to how the administrator would like to lock down their system. The Trusted Path Execution (TPE) project was ported to the Linux kernel as a Linux Security Module (LSM) to create a barrier against such security issues from occurring. This paper will attempt to explain how Trusted Path Execution is implemented in the Linux kernel as an LSM. It will also describe how TPE can prevent the running of malicious code on a Linux system via a strategically placed hook in the kernel. The usage of a pseudo-filesystem approach to creating an access control list for users on the system will also be discussed. The paper will further explain how TPE is designed and implemented in the kernel. This paper will show how the access control list is utilized by the module to place checks on the execution of code on the system along with a check of the path the code is being run in. Further, the origins of the "Trusted Path" concept and its origination in the OpenBSD operating system will be discussed along with how TPE was introduced to the Linux security community. The paper will conclude with a synopsis of the contents and future paths and goals of the project.

1 Introduction

The running of malicious code on a Unix system either via an active attacker or unsuspecting local user is one of the greatest threats to computer security we have these days. There are and have been several solutions to this problem. Many of the solutions only solve the problem at one layer of the system such as the network. The problem can and should be approached by applying security at various layers of the system. Much of the scenarios involving malicious code can be attributed to malevolent users placing damaging executables in an unprotected system and running them either remotely or via an unsuspecting local user. Other scenarios involving these unsuspecting local users occur when said user has written potentially damaging code themselves but are unaware that this is the situation. The innocent user attempts to run this buggy code and finds that they have caused major damage to their own system. The Trusted Path Execution LSM attempts to prevent such occurrences from happening.

The problems that malicious executables can cause are varied but for the most part the biggest issue is malicious code being placed on the system either intentionally or accidentally. There are several scenarios of how this can be bad for the system. If you can think of a good way to hose a system, it could probably be done with a malicious executable. The problem is greatly enhanced by systems that are unprepared for such attacks. Of course, where the executable resides and who is running it will be a major factor in how much damage it can do. Malicious code can be especially problematic in unsafe directories of a system where the parent directory of the malicious executable in question is world and/or group writeable. The malicious user is able to run code that when given the privileges of write can potentially overwrite or damage proper code on the system.

The Trusted Path Execution module attempts to prevent the problems that malicious code attacks create. It does so by protecting the system at the point from which the execution of a file takes place on a given system. The module patches the Linux kernel in such a way as to make a quick check of the current user's credentials and verifies that the executable is not being run in a vulnerable path on the system. If a situation is found where the module does not accept the security of the given situation, a failure will occur within the module and an error -EACCES will be returned.

The module utilizes a kernel hook within the Linux Security Module (LSM) framework in which it makes a check at exactly the point of execution on a file in the system. This check will verify whether the path is, in TPE's sense, "trusted" and whether the current user attempting to execute the file is also "trusted". If a situation is found where both the user and path are considered untrusted, then execution fails.
All other scenarios will result in the execution being allowed.

The determination of whether the path and user are trusted is also made by the module. It will check whether the parent directory of the file attempting to be executed is root owned and whether it is group and other writeable. If it is root owned and neither group nor world writeable, the path is considered trusted. If the current user attempting to run the code is either root or is listed in a trusted user access control list, they are considered trusted. The combination of checking the user and path for trustworthiness will determine whether the executable will be allowed to run. If neither are trusted, execution will be denied by the TPE module.

The remainder of this paper will attempt to further explain the problems associated with running malicious code on a vulnerable system. It will also show how TPE proposes to solve this problem and will be concluded with a wrap up of the paper in whole and future thoughts on the subject.

2 Malicious Executables in Untrusted Paths on a Vulnerable System

Preventing the execution of malicious code is a fundamental component of ensuring a Linux system's security. The problem seems quite evident but the scenarios will vary and are rather difficult to anticipate. This is indeed one of the most important problems a system administrator will face when contemplating the security of their system at a local level. The point at which security must be applied to a system is another issue to consider. Protecting at the network layer does not guarantee those that are on the inside, that is to say normal users, won't do something to also jeopardize the system. Bad code is bad code, no mater where it comes from. Similarly, protecting against specific types of well known malicious code won't protect against newly created ones. Of course, having these protections are important. The more points of vulnerability protected the better.

What is malicious code? It is a very general description of any kind of software that can cause damage to a system. This software includes viruses, worms, backdoors, Trojan horses, etc... What exactly can occur? One of several things, but in general buffer overflows from faulty code, exploit programs that can override root access, erasure of core system files, overwrite of system files and so on. In many cases the system will be rendered useless. There are several ways malicious code can be placed in a system. Thus, given the scope of this definition, the problems created by malicious code are immense and should be of high concern to any system administrator.

Take for example, a computer virus. A computer virus is generally defined as a program or piece of code that is loaded onto a computer without the knowledge of the owner/admin and is run against said owner/admin's permission. This is a very broad definition and accordingly the problems associated with computer viruses are also very broad. The computer virus is thus one of the most important security issues these days. Once a computer virus is detected, the damage is normally complete and the system administrator may only have hindsight to their benefit. The administrator is now capable of preventing the specific virus from occurring again, but is still vulnerable to other viruses that have yet to be applied to the system. To make the problem even more difficult to solve, sometimes the case where malicious code is applied to the system is performed by a local user. In fact, many situations of harmful code being run on a system are initiated by a local user with absolutely no idea that their program is about to wreak havoc on their own system. In this scenario, the local user has written code that is defective.

What can make this malicious code even more effective and/or malicious is the location in which it resides on a system. If the directory under which the code resides in is group or other writeable, we have allowed for further situations where the code could potentially overwrite other programs/executables on the system. When the directory allows for group and/or other writeable permissions in particular, this is leaving the code not only especially vulnerable. Unless otherwise given user-only access, the file will default to receiving permissions that allow just about anyone else write access.

Similarly, if all users on the system are given equal access to running the code we open the system up to further attacks. The greater the number of users who are able to run code, the greater the chance of faulty or malicious code getting executed. Why should everyone be given such access, if only a few users actually need to run executables?

Thus, the problem of malicious code is far-reaching. It is a problem that is highly difficult to solve. There are many solutions to preventing such code from entering a system via the network, but once it gets in the system and there are no protections for the system itself. The code is now able to be run and cause the damage. The only surefire approach to preventing any kind of code to get into the system is to simply unplug it from the network. And what about the problem with local users? If the system is simply unplugged from the network, there is still the problem of the innocent local user with buggy code. Must the administrator also remove the users? At this point, an unplugged systems with absolutely no users except administrator serves very little purpose except for being really, really secure. This is where security solutions like TPE can come in and help.

3 How the TPE blocks malicious code from running on the system

Solving the problems associated with malicious code is quite difficult. The question of where and how the code gets implemented is important. Answering this question is a great challenge. Simply containing a system from the network will not solve this problem, either. Common users that either want to cause problems or have no clue of their code's malicious intent will equally break a system. An approach that is both comprehensive and isolated in impact on a system must be taken. This approach must take in to account the various scenarios without making the system obsolete to normal users. It must also accept the fact that code that can harm the system will somehow make its way in. Indeed, it should know that the “bad” code is already in the system, just waiting to be run. With this in mind, the solution must find a way to minimize the chances of this “bad” code from being run without preventing the “good” code from execution. TPE aims to be exactly this type of solution.

TPE attempts to prevent users from accidentally executing malicious code by ensuring that only code installed by trusted users is permitted to execute. TPE is a Linux Security Module which enhances the security of the Linux kernel by monitoring the running of executables in “trusted paths” on a system by particular users. TPE accomplishes this by manipulating a strategically placed hook in the kernel that monitors the execution of files. It performs a check of the path in which the executable resides and the user who is attempting to execute the program. The check of the path will determine whether it is trusted or not. A “trusted path”, in the TPE sense of things, is one in which the parent directory of a file is owned by root and is neither group nor other writable. The component of the trusted path that allows for root owned directories is a convenience to the system administrator as they should be able to actually run system critical code. As a result, unless otherwise altered, base Unix directories like /bin and /usr/bin are considered trusted, but /tmp is not (Phrack 53-08 and 54-06)[2,5].

Why do we trust a “trusted path”? We will consider such directories as already protected by certain features in which the environment the executable resides in does not allow for code to cause major damage. In what way? Well, we'd like to be able to execute code as root and hope that root did not allow Joe user either ownership or execution rights to such code as would be found in /usr/bin. Thus, /usr/bin is a “safe” environment for code to be executed. On the other hand, if Joe user happens to have code X in /joedir where either he has world or group write access, he will be able to run code X. If code X is faulty, either purposely or not, Joe has the potential to cause major damage to the system. As a result, we will consider /joedir an “untrusted path”.

TPE makes use of a Trusted Users access control list to define “trusted users”. Users placed on the list are considered trusted and will be able to run executables they normally have access to run on the system without intervention from TPE. Upon attempt to execute, TPE will check whether the current user attempting to run an executable is trusted or not. If Joe decides to create buggy and/or malicious code X in an untrusted path he does have access to, he will find that he cannot run the code due to the presence of TPE on the kernel. In this case, if Joe is a legitimate user on the system, he can request to be added to the trusted user list, which allows him access no matter where he runs the code. Thus root has choices in whom to allow privileged execution rights. This also minimizes the amount of users who will be able to run executables at a given time. Root must actively add users to the trusted list (root is already in the list upon module instantiation). If Joe-user only needs to login to the system and check his email, root can choose not to put him on the list. In this case, root is able to add those users who absolutely need to run code on the system and thus keep the trusted list to a minimum.

The administrator will pick only those users that critically need to run code. In addition, the administrator could choose to allow certain users access at certain times when, perhaps, the administrator has been able to review the code that needs to be executed before it is run. He/she may also decide to revoke execution access to those users who have jeopardized the system before. This gives the super user much more flexibility in controlling the actions on the system.

There are only four scenarios that can be evaluated, of course; trusted user/trusted path, trusted user/untrusted path, untrusted user/trusted path and finally, untrusted user/untrusted path. The first three scenarios will be allowed execution. In the scenario where an untrusted user in an untrusted path attempts to run an executable on a TPE-ified kernel, the operation will be prevented from occurring. In this case, TPE has successfully accomplished its goal of preventing the potential malicious code from doing its damage. The following table describes the scenarios:

User**Path**	**Trusted**	**Untrusted**
Trusted	Execution Allowed	Execution Allowed
Untrusted	Execution Allowed	Execution not allowed

If malicious code is presented to the system from a remote machine, there is no way in which TPE can prevent this. Similarly, if an innocent user on the system accidentally writes buggy code, the module is not going to be able to do anything about it. What TPE takes into account the fact that malicious code can get into the system very easily but once it's on the system, it does not allow for it to cause the type of damage it would like.

TPE provides a line of defense to the system. It takes into account that malicious code will be on the system in some form or other. It minimizes the entry points for this code to be run and thus minimizes the amount of scenarios that could cause an attack from such programs. The TPE module does not prevent such situations as a malicious user acquiring the root password and/or utilizing a suid attack. The module will not run a firewall or audit the system. TPE is one of several approaches to containing the system and making it more secure. It should be used alongside a good firewall and other beneficial forms of security. It should not be the only method of hardening a Linux system.

4 How TPE is implemented

The Trusted Path Execution is implemented into the kernel as a Linux Security Module [12]. The Linux Security Module framework [10] is made up of a set of hooks into the kernel that can be manipulated in various ways. Of course, the main purpose of the framework is to manipulate the kernel into becoming more secure but several of the hooks can be utilized for other utilitarian purposes as the module needs. The module must also make use of a sysfs pseudo-filesystem [6] that allows for user space to system space interaction. This is a replacement to the common use of command line/system call user to system interface and will be explained further below. The module had been previously created utilizing the sys_security call and had to be migrated to the new method. Until a new system call can be made available to the LSM project, the module will continue to use the pseudo-filesystem approach.

The name of the sysfs pseudo-filesystem that TPE utilizes is "tpefs". It represents a file system interface that presents a file that lists trusted users. A user is considered "trusted" if their uid is on the list. The TPE-ified system will verify whether a user is trusted by reading this list. For convenience, the trusted list can be manipulated by utilizing userspace write actions, such as "echo", to add and remove users from the list. It can also be utilized to show the list to userspace via userspace read commands such as "more". The list is created in memory upon instantiation of the module.

The core code of the module, tpe.c, relies on two checks upon the running of an executable in the system. Within the module this is accomplished by utilizing the tpe_bprm_set_security hook, which is always called upon file execution. The two checks are performed by two functions, TRUSTED_PATH(current path, current uid) and TRUSTED_USER(current uid). The TRUSTED_PATH() function verifies whether the path is root owned and whether it is either group or world writeable. The TRUSTED_USER() function verifies whether the user running the executable is listed in the tpe_acl trusted user list. The functions are called within the module and performed in the tpe header file, tpe.h.

The Access Control list and Pseudo-File system "tpefs"

The tpe_acl trusted user list is created upon initialization of the module from a call to the tpe_init() function. In order to modify the tpe_acl list a sysfs pseudo file system [11] called "tpefs" is also created by the module. Two files are created in the "tpefs" filesystem which when written to, actually edit the tpe_acl list in memory. Thus, the filesystem is really not a file system but a method by which adminstrator can send information from user-space to kernel-space. This is why we call it a "pseudo" filesystem.

By default, root or uid 0 is added to the tpe_acl list upon initialization of the module. Root is protected from deletion from the list by a check in the code. Other users must be added utilizing the "tpefs" file system. Similarly, removing users from the list is performed with a "del" file. Both files, "add" and "del" are created for the filesystem by default from within the module code. The two file approach was utilized rather than a single file in order to keep the code and administration of the module simple for both the kernel and the user.

It should be noted that the usage of the sysfs pseudo-filesystem approach as opposed to a normal system call and command line method was due to the recent drop of the sys_security system call. There were a few other modules that were affected by this, including DTE[1] and SELinux[14]. Both projects are now utilizing the pseudo-filesystem method, as well. This seems to be the standard method by which the modules will be accessing system space from user space.

In order to instantiate the tpefs filesystem, the kernel must be compiled to include the LSM patch and the tpe.c module must be chosen to be installed as a module. Once both actions are taken, a partition must be mounted as type sysfs, as follows: mount -t sysfs sysfs /<mountpoint>. Next, the module must be inserted into the kernel via the insmod command: insmod tpe.o. At this point, a subdirectory under the sysfs mount point, <mountpoint>, created above, will be created under the name tpefs. There will be two files created by the module, namely add and del. In order to add a user, one simply needs to perform a write operation on the "add" file in the following manner: echo <uid> / <mountpoint>/tpefs/add
In a similar, deleting a user will involve a write to the "del" file is performed with the following command:
echo <uid> > /<mountpoint>/tpefs/del. Notice that utilizing the "echo" command is just a suggestion. Any other write command will work in a similar manner. Using "echo" is probably the simplest choice and is the one preferred by our team.

The "add" file may also be utilized to show the list and a description of this is given further on in this section. This is performed by performing a read action on the “add” file. This will instantiate a copy to be made of the list in to a user space buffer. This buffer information is then sent to stdout as a list of uids. The code is implemented from within the module by the trustedlistadd_read_file function(struct file *file, char *buf, size_t count, loff_t *offset) . Upon issuing a command such as cat <mountpoint>/tpefs/add, the list of uids will be presented to stdout. Currently, the maximum number of users that can be added to the TPE access control list is limited to 80. As more work is performed on the module, this will be altered to allow for a greater number of users and/or be made dynamic.

The code only allows for numeric user ids to be added on the command line at this point. It does return an error if any non-numeric values are entered. It also checks whether any other odd combinations are given as uids and returns an error. If any duplicate ids are attempted to be added , the module will also return with an error. This is similar for the removal of user ids. Any errors will be logged in the kernel info log for the system administrators to be able view. This adds a sort of auditing feature of the module and is found throughout the TPE codebase.

In addition to the tpe.c file, the module includes a header file called tpe.h. This is the only other non-document file associated with the module. It contains various macro definitions and a few sub-routine functions for the module. Most importantly, the TRUSTED_PATH() and TRUSTED_USER() functionality. The tpe_init function is also highly important:

Some the lesser involved macros in the tpe.h file include the TPE_ACL_SIZE value, which sets the size of the tpe_acl array to 80and the ACK, NACK and DUP macros which specify return values for the subroutines. Two important functions to the tpe_acl list are defined in the file, tpe_verify and tpe_search. The tpe_verify function will search for the uid that is currently being attempted to be added to the tpe_acl by the administrator. The result of this function will tell the TRUSTED_USER() function whether the uid is valid or not. The tpe_search is utilized by tpe_verify to determine if the uid is already on the list. If the uid is on the list, an error is returned so that repeat uids are not added to the list. These macros are all highly important to the module and are useful in making it more refined and user-friendly.

A documentation file for tpe called tpe.txt is also available on the LSM patch. The document is installed along with the module in the / Documentation/lsm directory. This document will give the user information about the module, how to install it and contains guidance for utilizing the tpefs pseudo-filesystem.

5 Evaluation

The actual effects the Trusted Path execution module makes to the system have been shown to be unintrusive, safe and effective. The module does in fact secure the system in the way it says it should. It is compatible with the Linux kernel and causes no performance issues. To be more accurate, the module has only been run on 2.5/2.6 Linux kernels and therefore this information is only pertinent to those particular kernels.

Security

The module has been thoroughly tested for functionality. It does do what it says it should and that is to enhance security by blocking execution of files. The four basic scenarios of trustworthiness were applied to a system and the results were evaluated. These four scenarios were: trusted user/trusted path, trusted user/untrusted path, untrusted user/trusted path and untrusted user/untrusted path. The results of these tests concluded that execution was allowed to be performed only in the first three scenarios listed above. It was also shown that when an untrusted user in an untrusted path attempted to execute a file, it was denied. This was the crucial point that TPE was indeed applying its security policy to the kernel.

As an example, let's say user Joe would like to run executable “buggy” in the /tmp directory. Prior to loading TPE on the system, so long as Joe has execute permissions to the file, he will be able to execute “buggy” from within the /tmp directory. Joe then goes on to wreak havoc on the system, eventually bringing it down. The administrator is now left with an obsolete system and must reinstall. If the administrator could have turned back the clock and installed the Linux Security Framework on his kernel along with enabling the TPE module, he/she would have saved themselves a headache.

If TPE had been installed on this system and the trusted list was created without Joe's userid added, Joe would have attempted to run “buggy” and found that execution would have been denied. Upon Joe running “./buggy”, TPE's hook in the kernel, tpe_bprm_set_security hook, is called in. No matter what form of execution takes place on the system, TPE will always be called and do the check of the path and user. TPE has detected that Joe is not a trusted user and that /tmp is not root owned and neither group or other writeable. In other words, TPE is checking whether the user and/or path are “trusted”. Since both user and path or not trusted, in this case, execution is denied by the module and -EPERM is returned to stdout. At this point “buggy” was not run and the system is still up. TPE has saved the day.

Suppose the system administrator wanted Joe to run “buggy” for some sadistic reason. In order to allow Joe permission to run his “buggy” executable, the administrator need only add Joe's userid to the trusted list. Once Joe is on the list, his attempts to run “buggy” will be allowed. Perhaps the adminstrator needed to crash the system.

Consider the scenario of an innocent user seeking code from outside the system. The TPE module will not be able to block a user from downloading code from an external resource, such as an ftp download web site or outside host. If this code is malicious/buggy, the system has no way of finding this out. But this does not indicate TPE has failed. Once the code is on the system, it is not able to perform its malicious intent because TPE has blocked it from being executed.

It has been shown that via various testing models, the above scenarios indeed occur with and without the Trusted Path Execution LSM. These tests would check kernel kernel log messages for the appropriate logs from the TPE module. These were all found to be cleanly applied to the kernel event log and were appropriate to the actions taking place during the testing scenario. The tests would also verify that no errors were being logged by the kernel that were not expected at the given action of the TPE module within the system.

Compatibility

The code has been thoroughly reviewed by both the IBM and LSM communities. As a result of this review several enhancements have been made to make the code to make it more effective. We have also modified the code to make it less capable of becoming buggy. One enhancement that came out of the code review was adding kernel spin locking capabilities to make the module smp safe. Another important upgrade for the module was moving from the pcihpfs pseudo-filesystem approach to the sysfs pseudo-filesystem. The move to the new sysfs filesystem made for a cleaner code base. The code was also greatly reduced in size and much easier to debug as a result. This migration also reduced the amount potential vulnerable spots in the code. Thus, once again the code is much less capable of becoming harmful to the kernel and/or operating system.

TPE has been implemented as a Linux Security Module and accepted by the LSM community. Therefore, it abides by the framework the LSM community utilizes to attach to the Linux kernel. It has been written according to guidelines set by this community. The LSM community in turn works closely with the Linux kernel development community. As a result, the module also adheres to coding rules and styles as set by the Linux kernel community. The LSM framework is scrutinized by the Linux kernel core development team as well as the lkml mailing list and anyone else interested in the kernel. Thus, the project has several eyes scrutinizing it. Most importantly, it is scrutinized by those especially knowledgeable with the Linux kernel and system security.

The Trusted Path Execution LSM is still considered new and experimental and thus should be thoroughly reviewed as it progresses as a module. It is indeed simple and small enough that it is not anticipated to be a great effort to test its value to the kernel. The module was written with security coding standards being a high priority and will continue to be modified with this consideration in mind. The module was run through several tests including system and functional verification. These are described in more detailed below.

The TPE LSM has been thoroughly system tested. Basic testing was performed to verify that the module's implementation in the kernel does not break other programs and is cleanly applied to memory. As mentioned in the Security section above, kernel logs were verified so that no inappropriate events occurred during the loading, unloading and run-time actions of the modules. No testing was performed while other security modules were loaded, as TPE was not created to be a stackable module. If at some point in time, this is not the case, stackability will also be tested.

The trusted user list that is created in memory was thoroughly tested. Users were added and removed from the list while the actual list was monitored. It was shown that the list was effectively manipulated and accurately reflected the desires of the system administrator as to who should be listed. Invalid values, non-uids, were attempted to be passed to the list for addition and deletion. The module appropriately denied addition of these invalid parameter values. In addition, uids that were not on the list were attempted to be removed. The module was able to recognize that these uids were not on the list and acted appropriately with an error return. Overall, manipulation of the tpe_acl trusted list was deemed accurate and clean of problems/bugs.

Memory tests were also conducted on the tpe_acl list to verify that no overwrite of memory would take place during actions on the list. Several of the tests were attempts upon overloading the tpe_acl array with too large or too many uid values. Checks in the code prevented this from occurring. As this was the only parameter the module creates in to memory, it was the only object that needed this sort of testing.

The LSM project is also constantly under test and scrutiny. In fact, a fellow IBMer, Trent Jaeger, has created a set of projects to verify the LSM project [9]. Trent and his team are working towards verifying that the LSM framework appropriately implements the security actions of each module. They have used the CQUAL static analysis tool to make sure every security relevant operation could be controlled by an LSM hook. They also created a runtime analysis tool, Vali[3], that finds inconsistencies in operations authorized by the modules. The discoveries that Trent and his team found utilizing their analysis tools have led to improvements of the LSM along with validation that the framework can indeed be utilized to secure the system.

Once TPE is instantiated on the system, there are some limitations to keep in mind. The system is locked down and simple execution is controlled, thus ordinary users may find some annoyances to deal with when wanting to run executables but they don't have access rights to do so because of the module's presence on the system. A TPEified system will essentially become more of a “governed state”, where greater interaction must occur between the end user and system administrator(s).

Although we have performed several tests on this module, it should be noted that this is no guarantee that all potential bugs/defects have been removed from the code. It should also be noted that this is still an experimental project and more time and effort will improve the validity of the code. Further testing is always necessary and shall be performed in the future as new kernel levels are introduced and features are enhanced on the module itself.

Performance

No specific performance testing was completed on the Trusted Path Execution module. Upon instantiation of the module, no visible performance impact was made and thus benchmarking was not deemed necessary. The module makes use of only one kernel hook and thus is quite small on its impact into the system. The usage of the “tpefs” sysfs filesystem is created upon instantiation of the module, as are several other smaller components of the module. Given that the “tpefs” codebase is the largest portion of the module, once we are past insmod'ing the module, there is very little code that actually augments the kernel. If performance benchmarking is deemed necessary in the future for TPE, it will be performed at that time.

6 Related work

The meaning of the “trusted path” has been previously defined [13] by a much older concept in security.
According to the “Secure Programming for Linux and Unix HOWTO (see url below), “A trusted path is simply some mechanism that provides confidence that the user is communicating with what the user intended to communicate with, ensuring that attackers can't intercept or modify whatever information is being communicated.” This definition applies to a context where security is to be tested on a particular system and is not quite associated with the concept of a “trusted path” as defined in the Trusted Path Execution Loadable Security Module project. The original idea presented a similar concept in that the original ensured that the login prompt was legitimate. This is similarly to how the Trusted Path module ensures that all programs an untrusted user executes are legitimately put there by root. For further information see http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/trusted-path.html.

The second definition of a “trusted path” was defined in a project that the TPE LSM was based on. Trusted Path Execution project was originally created for OpenBSD 2.4 as a direct patch to the kernel by Mike Schiffman. It was described in Phrack 52-06 and later modified by the Stephanie project [15] for OpenBSD 2.8 and 2.9. The patch was distributed under the two clause BSD license. The usefulness of this project as an enhancement to Unix security was recognized by the Linux Security Module community and subsequently suggested as a potential module. There are several differences between the original BSD patch and the LSM version. The most notable difference is, of course, in how they are implemented into the BSD and Linux kernels, respectively. The Stephanie project also brought in a few more checks into the BSD kernel, such as restricting symbolic links via the Openwall project from Linux. The Stephanie project also uses an actual system call to implement a tpe_adm command to modify the TPE trusted user access control list, whereas the LSM utilizes the sysfs pseudo-filesystem. Despite these major differences in code, the core concept of the Trusted Path Execution kernel check is the same in all of the projects mentioned above.

There have been a handful of other Linux Security Modules that have been implemented into the Linux kernel prior to the Trusted Path Execution module. The SELinux LSM is one that is most notable. It makes use of the Linux Security Module framework to implement the Flask mandatory access control architecture model into the kernel. It is an example of a much larger implementation of the LSM framework. The project is led by Stephen Smalley and Peter Loscocco of the National Security Agency. The Domain and Type Enforcement (DTE)[7] LSM is another module of note. Created by fellow IBMER, Serge Hallyn[4], it makes use of a mandatory access control model that assigns types to files and domains to processes and furthers this idea by associating which domains can access which types. Interestingly enough, both module implementations, SELinux and DTE, were initially direct Unix kernel patches prior to becoming LSMs, much like the Trusted Path Execution module. Both DTE and SELinux are currently available via the LSM patch. The SELinux module is also available directly on the 2.6 kernel along with the LSM patch.

The Linux Security Module project [8] is continuing its efforts to create a comprehensive set of security modules for the Linux kernel. It is maintained regularly and current patches are available for download off of their main web site. According to it's site maintainers, LSM “provides a lightweight, general purpose framework for access control”(see lsm.immunix.org). The project was created as a means of providing security to the Linux kernel without adding the overhead of direct patches to the kernel. The user may choose which module to implement and thus have greater control of the security of the kernel. The usage of the module approach makes the task of securing the system much more flexible and dynamic. There are several modules to choose from at this point and a couple more have been added since the introduction of the TPE LSM. Since the project is fairly new to the kernel, many of the latest modules are still considered experimental. There are several well established modules, including SELinux, DTE and owlsm. These projects are all current and actively maintained.

7 Conclusion

The Trusted Path Execution LSM has been designed to enhance the security of the Linux 2.6 kernel. In its ability to prevent the running of malicious executables, the module shows one way that the kernel can be manipulated in order to protect a system from potential damage. It is blind to whether the malicious code was intentionally created or not, and thus covers both scenarios. By performing a check into the kernel at the point of file execution, the module is able to monitor whether the path the executable resides in is “trusted “and whether the user is considered trusted. If both the path and user are “untrusted”, the module will prevent execution from occurring.

TPE was accepted by the LSM community in May of 2003. The module was submitted under a dual BSD/GPL license. TPE was integrated into the mainline LSM BK tree immediately and placed on their official patch to the 2.5.70 kernel. The current version of the TPE module has been available on all LSM kernel patches since then. It is anticipated to be placed in the official 2.7 kernel once development commences on that project. TPE is a fairly new addition to the LSM lineup.

Further enhancements to the project will be added in the future to increase the value-add the module can bring to a system. This may include increased administrative capabilities and further checks into the filesystem. It is anticipated that the module will be of great use to many system administrators seeking to improve the security on Linux.

Bibliography

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, Sheila A. Haghighat. Trusted Information Systems, Inc. A Domain and Type Enforcement UNIX Prototype. 5th USENIX Security Symposium. June 1995. Salt Lake City, Utah.
[2] Krzysztof G. Baranowski. Linux Trusted Path Execution Redux. Phrack 53-08. July 1998.
[3] Antony Edwards, Trent Jaeger, Xiaolan Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. ACM Computer and Communications Security. November 2002. Washington, D.C.
[4] Serge Hallyn, Domain and Type Enforcement for Linux. http://www.cs.wm.edu/~hallyn/dte.
[5] route| daemon9. Hardening the Linux Kernel (series 2.0.x).Phrack 52-06. January 1998.
[6] William von Hagen. Migrating to Linux kernel 2.6. LinuxDevices. Com article. February 2004
[7] 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). 6th USENIX Security Symposium. June 1996. San Jose, California.
[8] Chris Wright and Crispin Cowan. Linux Security Modules: General Security Support for the Linux Kernel. USENIX Security Conference. August 2002. Ottawa, Ontario.
[9] Linux Security Analysis Tools. http://www.research.ibm.com/vali/
[10] Linux Security Modules. http://lsm.immunix.org
[11] LWN article. Avoiding sysfs surprises. June 2003.
[12] Security modules begin to appear. LWN Article on TPE. May 2003. http://lwn.net/Articles/31571
[13] Secure Programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/trusted-path.html
[14] SELinux Documentation http://www.nsa.gov/selinux/info/docs.cfm