Global Information Assurance Certification Paper

Understanding the Security Features of SELinux

Research on Topics in Information Security
GSEC Certification Practical
Research Topic Option 2
V 1.4C

Author: John Mayes
20 January 2005

Abstract

With the vulnerabilities inherent to complex Unix systems, there is an increased need for more stringent security controls within an operating system. While system hardening and firewalls can inhibit the curious, a determined attacker can uncover weaknesses in the system. Such attacks can be initiated from entities external to the machine as well as authorized users of the machine. In both cases, such attacks prey upon flaws in the system software or weaknesses in the system configuration. It is a well-known fact that few software products are completely free from defects. Even the most talented programmers can inadvertently leave some small piece of code in an application that makes it vulnerable to such attacks as buffer overflows. Similarly, as systems become more complex, it can be easy for a system administrator to overlook a configuration element that creates a weakness in the security of the system. With such a high rate of attacks and compromises, operating systems have begun to require a much more fine-grained control over what applications can and can't do. By incorporating security features such as Mandatory Access Control, Role Based Access Control and fine-grained auditing, the risk of system compromise can be greatly reduced. In this paper, we describe the security aspects of Security Enhanced Linux (SELinux) and demonstrate how an enforced security policy can protect the system.

1. Introduction

Since its public release in December of 2000, SELinux has experienced increased acceptance in the open source community. It has been incorporated in to the Linux kernel source and has found its way into many of the popular Linux distributions. In fact, many of the current distributions install SELinux as the default operating system configuration.

One might ask why SELinux is working its way into the main stream. As shown in Table 1, the number of instances of compromises and attacks as reported by CERT has doubled with each passing year. With the prevalence of attacks, it has become necessary to employ tighter security controls within an operating system. Until recently, such controls were only available on high end and special purpose systems. With SELinux, such security controls have become available for use smaller, general-purpose systems. Additionally, SELinux has enjoyed considerable development efforts. Over the last couple of years it has reached a state of maturity and stability. As a result, it has become easier to install, configure and maintain.

Year	1990	1991	1992	1993	1994	1995	1996
Incidents	252	406	773	1,334	2,340	2,412	2,573
Year	1997	1998	1999	2000	2001	2002	2003
Incidents	2,134	3,734	9,859	21,756	52,658	82,094	137,529

Table 1: CERT Incidences [1]

In this paper we discuss the features of SELinux. In section two, we provide a brief overview of the security architecture and show how these features can create a system that is difficult to compromise. Additionally, we describe the use of the various utilities that aid in the creation of a security policy. In section three, we demonstrate how these tools are used to create a security policy for an insecure application. We show how SELinux can be configured such that applications are restricted to only those operations that are defined by the security policy. Finally, we conclude the paper and provide a direction for current development efforts.

2. SELinux

Development of SELinux began at the National Security Agency (NSA) [2]. The NSA, working in conjunction with Secure Computing Corporation and researchers at the University of Utah, developed a security model and implemented working prototype in a research operating system called Fluke. The security architecture was given the name Flask[3] (Fluke Advanced Security Kernel). Following these efforts, the NSA worked with Network Associates and Mitre to incorporate the Flask architecture into the Linux kernel and released SELinux to the public [4]. While SELinux has been available since the end of 2000, it has taken substantial development efforts to bring SELinux to the main stream.

SELinux does not differ greatly from a standard Linux system. The security features of a regular Linux system, user authentication, firewalls and Discretionary Access Controls (DAC), are still in place. SELinux incorporates an additional security model in addition to the traditional security model. This security model implements Mandatory Access Controls (MAC) [5] and Role-Based Access Controls [6] within the operating system and supplies access controls through a security policy. With DAC, users have a substantial amount of control over the files and directories that they create. With MAC, security controls are implemented by the system and can be changed only through a privileged operation available only to system administrators based upon their role.

Among the features of SELinux is the ability to run the system with SELinux Development support compiled into the kernel. By enabling development support, the system can be set to enforce or not enforce the security policy. System administrators can set the kernel to enforcing mode or permissive mode during system boot or at runtime. When set for enforcing mode, applications

that violate the security policy are immediately terminated and the operation that caused the violation is logged. In permissive mode, applications are allowed to execute to their entirety. Each operation that does not pass the security policy is logged. By permitting a permissive mode, it allows the developer of the security policy greater ease in constructing the policy for an application. In fact, the utilities for SELinux includes a script that examines the generated logs and constructs a set of rules based upon the denied operations. When the security policy has been thoroughly tested, this feature can be completely removed by installing a new kernel that does not have contain development support.

2.1 Security Architecture

In simple terms, the security architecture for SELinux boils down to subjects, objects and allowed actions [7]. Subjects are the user, or more precisely, processes acting on behalf of a user. Objects are the entities that the subject needs to work with. In the system, these objects are files, directories, devices, sockets and even other processes. Lastly, actions are the operations that are performed by the subject on the object. Typical operations include read, write, and execute to name a few.

Each subject and object on the system is given a security context. This context is composed of three elements, namely a user, role and domain. The user can either be a class of users, such as system administrators, or specific user of the system that matches an entry in */etc/passwd*, such as root. For objects, the user attribute signifies the owner of the object. Note that this is a security policy ownership and it is distinct from the regular file system object ownership given by the ls command. Each user of the system is assigned to one or more roles. Roles are assigned a set of permissions to act on objects. Objects are assigned to a dummy role of “object”. Finally, domains define the relationship between subjects and objects and divide them into related groups. Unless explicitly allowed, a user operating at a particular role in one domain cannot access objects that are defined with the same user and role but lie in a different domain. By defining different domains, one can effectively isolate applications or a set of tasks to working with only those objects that are defined in the domain.

Sitting between subjects and objects are the three components of the Flask architecture, namely a security server, object broker and access vector cache (AVC) [3]. These components act as arbiters between subjects and objects and either allow an operation or deny any operation that is not explicitly defined. They are implemented directly in the Linux kernel and cannot be circumvented when the policy is enforced. The security server maintains the security policy defined for the system. The security policy is a set of rules that define the actions that are allowed. For example, one such rule may be to allow the root user the ability to read and write to the network host database */etc/hosts*. The object broker is the enforcer of the policy. It attempts to retrieve a security context that specifies a given subject has the capability to perform an action on an object based on the security context of both the user and object. Because

computers don't work efficiently with strings, each security context is mapped to a unique security identifier or SID. Between the security server and the object broker is the access vector cache (AVC). Each SID is allocated a set of bitmap vectors that define the allowed operations taken by a subject on an object. The AVC stores these vectors in a lookup table so that subsequent lookups for an identical SID do not have to query the security server. This effectively reduces the performance overhead of maintaining and enforcing the security policy.

The security context association of user, role and domain comprises the Mandatory Access Control (MAC) label. As stated earlier, each subject and object on the system is assigned a security context, which is its MAC label. These labels can be either be persistent or transient [7]. Persistent objects encompass those objects that exist in permanent storage. The Linux file system has been adapted to store the MAC label of persistent objects with the object. Transient objects are those objects that exist only for a short period of time and exist in the memory of the system. To understand a transient object, consider the life cycle of a process.

In SELinux, the security context of the user generally differs from that of the program. When a user executes a program, the created process must transition from the user's security context to a new, dynamically generated, context under a different domain. Such transitions must be explicitly allowed by program's policy to allow the user to make the transition. The new security context keeps the originating user and role but assumes the domain of the program. Similarly, if a running program must act upon an object that lies in a different domain, a new security context is generated that joins the program's context with that of the required object. The new contexts that are generated exist only for that instance of the of the process and can be dismissed when the process is complete.

2.2 Policy Structure

Upon first inspection, it might seem like a daunting task to understand the structure of SELinux. However, by spending a little "quality time" with the operating system, one finds the security policy structure fairly intuitive and easy to manipulate. We'll begin by examining the files that the security source uses to create a policy.

SELinux maintains the security policies in the directory */etc/selinux/src*. A security policy is split between two files, namely a file context and a type enforcement [8]. These are located in the file*_contexts/program* and *domains/program* directories within the SELinux source directory, respectively. The file context has the file extension .fc and describes the MAC labels that will be set on persistent objects that are specific only to a particular application. The type enforcement file has the file extension .te and defines the set of rules that an application is allowed to perform. In general, it is preferable to group applications according to a single domain. While it is possible to define multiple

domains within a file context/type enforcement pair, the size and complexity of the policy may make it difficult to create and maintain.

The file context is a simple file that describes the security context associations given to objects within a domain. Each line in the file has the form "<target> <label>". The target object can either be a single file, directory or a set of files and directories. As described in Section 3, the target description can utilize a regular expression to match all files under a directory tree. The label is composed of the three elements of a security context and has the form "*user:role:domain*". In almost all instances, the user assigned to an object is the system user, *system_u*, and the role is the generic dummy role, *object_r*. The domain varies for the type of object. Executable files, log files, and configuration files are generally given their own unique identifiers as to their purpose in the domain. For example, an executable in the domain "*abc*" would be given the domain label of "*abc_exec_t*".

The type enforcement file defines the domain and the set of allowable operations that can be performed within the domain. To simplify the generation of rules, the SELinux policy source defines a set of macros that expand to a set of declarations in the policy. For example, the macro "*tmp_domain(abc)"* expands to the set of allowable rules used by the domain "*abc*" for creating and working with files in the */tmp* directory. It is left to the curious reader to examine the files in the *macros* directory under the SELinux policy source to determine the purpose and use of these macros. What remains in the file are *type* and *allow* definitions. Types define the domain and other domain specific labels. As stated earlier, an executable file within the domain would be given a label resembling "*abc_exec_t*". A type declaration would be specified in the type enforcement file that binds the domain label to an executable type within the domain. The *allow* entries define the allowable operations that subjects in the domain can perform on objects in the domain. The list of allowable actions is quite lengthy, and goes beyond the scope of this paper. However, in Section 3, we show the use of type and allow definitions as they pertain to the development of a security policy.

2.3 SELinux Utilities

Bundled with the SELinux core utilities package are a set of applications that used to configure and maintain the system as well as for policy development. Provided here is a list of the more common utilities that one can use.

- **newrole** – starts a new shell with a new role, type or role and type combination.
- **chcon** – changes the security context of a file, directory or an object.
- **checkpolicy** – checks the security policy and compiles it into a binary form suitable for loading into the kernel.
- **load_policy** – loads the compiled policy into the running kernel.

- **getenforce** – gets the current status of the policy enforcement and returns either permissive or enforcing. If the kernel has been compiled without SELinux development support, this utility always returns enforcing.
- **setenforce** – sets the kernel to enforcing (1) or permissive mode (0).
- **setfiles** – sets the MAC labels on directories and files as defined in the system security policy. If the supplied target is a directory, all files under that directory will be labeled.

For the most part, a policy developer will use a makefile that is defined in the SELinux policy source directory. This makefile includes several targets that build and load the policy into the kernel. Instead of calling the *checkpolicy* and *load_policy* utilities individually, one can simple navigate to the policy source directory and type "make load". This will make the appropriate calls to compile the policy and load it into the running kernel.

3. Securing Services with SELinux

In this section we describe the steps necessary to generate a security policy for a simple server application. In addition, we show how the security policy can be used to protect the system from unauthorized access. While there are certainly many options that can be used to configure a policy for a program, our objective is to demonstrate the basic policy construction and use of the various tools.

3.1 Naive Server

To demonstrate the use of an SELinux application policy, a simple network client and server application named naive has been created. The application's purpose is to mimic the "get" functionality of an ftp or http server. A client connects, requests a file that exists on the server and is presented with the contents of the file. As shown in Appendix 1, the program source for the simple server uses many of the operations found in traditional network server applications. The functions performed include network socket creation, network input and output, file operations and child process instantiation.

The application does however have one glaring security hole. It makes no attempt to verify that the client or the server has permission to get the requested file. If no security policy for the server exists on the system, it has access to any file on the system that the executing user has permission to read. When executed by the root user, the server has access to all files on the system. As we shall see, a security policy can be constructed such that the server application is restricted to its own, isolated, execution domain and has access to only those files and capabilities that are defined for its domain.

3.2 Security Policy

As stated in Section 2, a security policy is defined by two properties, namely a

file context and a type enforcement. The file context defines the association of files, directories and executables to a particular security domain identified by its security label. The type enforcement defines the privileges, read, write, execute, etc., that a user or application may perform within the security domain.

To construct the security policy for the naive server application, we begin with the file context. As the root user, using the *sysadm_r* role, we create the file */etc/selinux/src/file_contexts/program/naive.fc*. This file contains the following entries:

```
/opt/bin/naive          -- system_u:object_r:naive_exec_t
/opt/naive_files(/.*)?     system_u:object_r:naive_file_t
```

The first line defines the application executable /opt/bin/naive to have the label *system_u:object_r:naive_exec_t* label. The security context defined by this label specifies that it belongs to the system user, is assigned to the object role, and has a unique type enforcement of *naive_exec_t*. The double dashes signify htat the object is a single file. Similarly, the second line defines the file objects that are contained within the naive domain. Note that it is possible to supply a regular expression to the file definition. Therefore, the second line matches any file contained under the */opt/naive_files* directory and signifies that the directory and all files under that directory will be labeled as belonging to the naive domain.

The next step in constructing the security policy is to create the type enforcement file. Because this file specifies the various system level operations required by the application, the type enforcement file is more complex. Additionally, it is difficult to know the full set of privileges that an application will require when creating the type enforcement file. Luckily, the SELinux policy utilities define an application, *audit2allow*, which aids in the creation of rules. This utility examines the logs generated from operations that are denied by the access vector cache and outputs a set of rules that will allow the denied operations. It is worthwhile to note that this application should not be used indiscriminately. The log file can contain denied operations from other applications. It is therefore necessary to evaluate each rule generated by *audit2allow* to determine if the rule is appropriate.

To construct the type enforcement file, create the file */etc/selinux/src/domain/program/naive.te* as the root user, using the *sysadm_r* role. We start with a skeleton domain that contains no allowable privileges. At this stage the file contains the following entries:

```
type naive_t, domain;
type naive_exec_t, file_type, sysadmfile, exec_type;
type naive_file_t, file_type, sysadmfile;
domain_auto_trans(staff_t,naive_exec_t,naive_t)
```

The first line defines *naive_t* as a domain. The second and third lines define the file types for the domain as referenced by the file context and associate them as accessible by the system administrator. The fourth line authorizes the application to transition from the *staff_t* domain to the *naive_t* domain. When executed the application will begin in the domain of the parent shell, which in our case is root, running at a label of *root:staff_r:staff_t*. This transition is necessary for the application to transition to its own isolated domain. Also, note that the fourth line differs from the preceding lines. SELinux defines several macros that help in the creation of policy rules. These macros are defined in the directory */etc/selinux/src/macros* and are fairly easy to understand.

Once these two files are created, it is then necessary to generate the policy and load it into the running kernel. This is achieved by running the "make policy" and "make load" commands within the directory */etc/selinux/src* as root, running at the *sysadm_t* role. Next, we run the command

```
#setfiles /etc/selinux/src/file_contexts/filecontext /opt
```

to relabel the files in /opt that contains our naive server binary and related files. Now that the policy is installed and the files are labeled, what remains is to generate the set of allowed privileges and add them to the type enforcement file. For this exercise, it is assumed that the kernel is running in permissive mode as described in Section 2 and that the application is allowed to complete. While it is possible to create the privilege set in enforcing mode, it is substantially easier to generate the rules from a permissive mode execution.

In order to generate the allowed privileges, it is necessary to reload the policy into the running kernel by issuing the "make reload" command within */etc/selinux/src*. This marks the log file so that *audit2allow* will generate rules that pertain to the execution of the naive server. Next, we perform a complete execution of the application. The resulting access vector cache denials will be logged to kernel log. We then run the command

```
#audit2allow –l –i /var/log/kern.log > /temp/naive.te.rules
```

to generate the set of allow rules. This command will produce the following set of allow rules:

```
allow naive_t etc_t:dir search;
allow naive_t ld_so_cache_t:file { getattr read };
allow naive_t lib_t:dir search;
allow naive_t lib_t:lnk_file read;
allow naive_t naive_file_t:dir search;
allow naive_t naive_file_t:file { getattr read };
allow naive_t self:process { fork sigchld };
allow naive_t self:tcp_socket { accept bind create listen
        read setopt write };
```

```
allow naive_t netif_eth0_t:netif { tcp_recv tcp_send };
allow naive_t node_inaddr_any_t:tcp_socket node_bind;
allow naive_t node_t:node { tcp_recv tcp_send };
allow naive_t port_t:tcp_socket { name_bind recv_msg
        send_msg };
allow naive_t root_t:dir search;
allow naive_t shlib_t:file { execute getattr read };
allow naive_t staff_devpts_t:chr_file { getattr read write };
allow naive_t usr_t:dir search;
allow staff_t naive_file_t:dir search;
allow sysadm_t ld_so_cache_t:file execute;
```

Each of these rules must be carefully scrutinized to ensure that they are appropriate for the application. When the rules have been judged as being correct, they can then be added to the file */etc/selinux/src/domains/program/naive.te*. Please refer to Appendix 2 for a listing of the compiled, annotated type enforcement file.

Once again, we reload the policy into the running kernel by performing a "make reload" in */etc/selinux/src* and test the application. Upon inspection of the logs, we find the follow entry:

```
audit(1105305557.105:0): security_compute_sid:  invalid context root:staff_r:naive_t for
    scontext=root:staff_r:staff_t tcontext=system_u:object_r:naive_exec_t tclass=process
```

The invalid context error specifies that we have not given the *staff_t* domain access to the *naive_t* domain. To enable this privilege we add the following line to the naive type enforcement file, naive.te

```
role staff_t types naive_t;
```

This specifies that a role in the *staff_t* domain has the right to access the *naive_t* domain. As a final check, we reload the policy, execute the naive server and verify that no logs are generated.

Now that the policy has been created and installed in the kernel, we can now verify that the system behaves as intended, allowing access to files defined in the domain while denying access to all other files. To demonstrate that the system does indeed restrict access, we first enable the kernel to enforce the policy by issuing the command "*setenforce 1*". We then run the naive server and attempt to get a file within its domain, *testfile*, and a file in a different domain, */etc/passwd*. When, retrieving a file within its own domain, the server correctly reads the file and sends the file content to the client as shown by the output

```
# ./client selinux
    get /opt/naive_files/testfile
    ================================================
```

```
this is a test file
```

This corresponds to the contents of the requested test file. However, when attempting to access a file that lies outside the *naive_t* domain, say */etc/passwd*, the client responds with the following output:

```
# ./client selinux
get /etc/passwd
================================================
Error retrieving file Permission denied
```

Additionally, the SELinux server generates the log entry:

```
audit(1105333950.229:0): avc:  denied  { getattr } for  pid=5726 exe=/opt/bin/naive
    path=/etc/passwd dev=sda3 ino=338061 scontext=root:staff_r:naive_t
    tcontext=system_u:object_r:etc_t tclass=file
```

Upon inspection of the log entry, we see that the naive server is running with the label *root:staff_r:naive_t* as defined by the *scontext* in the above log entry. In order for it to access */etc/passwd*, it must have access to the *etc_t* domain as shown by the *tcontext*. Because the type enforcement definition does not explicitly allow access to */etc/passwd* or allow access to the *etc_t* domain, the security policy will forbid the naive server from accessing the file. The same is true for any other file on the system that is not within the *naive_t* domain. By creating a specific domain for the naive server, it effectively jails the application to working with only those resources that are defined within the domain.

In this section, we have demonstrated the ease with which a security policy can be generated. While this example is a little contrived, it does show how a policy can effectively protect the system from wayward applications. If we extend this philosophy to include vulnerabilities due to software flaws or misconfiguration, we see that the objects at risk are those that are defined as accessible by the security policy.

5. Conclusion

In this paper, we have introduced SELinux. We have presented some of the features of the security controls present in the operating system and shown how they can be used to protect the system from vulnerabilities. SELinux is by no means a magic bullet. It is still the responsibility of the system administrators to exercise due diligence in maintaining the security of the system. While the use of SELinux does place a burden upon system administrators to learn an additional security model, we have shown that the security features and policy generation are fairly easy for one to understand.

At the time of this writing, support for Multi-Label Security (MLS) is still very experimental. While the SELinux kernel source does provide support for MLS, utilities for using and configuring MLS are still under development. In

applications where data confidentiality and integrity are paramount, support for MLS will be an absolute necessity.

Appendix 1 : Naive Server Source

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
#include<errno.h>

#define PORTNUMBER 12345

void error_out(int ns, char *buf) {
  int msglen;

  msglen = strlen(buf);
  send(ns, &msglen, 4, 0);
  send(ns, buf, msglen, 0);
  close(ns);
  exit(0);
}

int main(int argc, char **argv){
  char inbuf[1024];
  char outbuf[1025];
  char *filename;
  int n, s, ns, len, status, msglen, on=1;
  pid_t pid;
  struct sockaddr_in name;
  struct stat filestat;
  FILE *file;

  if((s=socket(AF_INET, SOCK_STREAM, 0)) < 0){
    perror("socket");
    exit(1);
  }
  memset(&name, 0, sizeof(struct sockaddr_in));
  name.sin_family = AF_INET;
  name.sin_port = htons(PORTNUMBER);
  len = sizeof(struct sockaddr_in);

  n = INADDR_ANY;
  memcpy(&name.sin_addr, &n, sizeof(long));

  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  if(bind(s, (struct sockaddr *) &name, len) < 0){
    perror("bind");
    exit(1);
  }

  if(listen(s, 5) < 0){
```

```
    perror("listen");
    exit(1);
  }

  for(;;){
    if((ns = accept(s, (struct sockaddr *) &name, &len)) < 0){
      perror("accept");
      exit(1);
    }

    pid = fork();

    if( pid == -1 )
    {
      perror("fork");
      exit(1);
    }
    else if( pid == 0 ) /* child */
    {
      if((n = recv(ns, inbuf, 1024, 0)) <= 0){
        perror("recv");
        exit(1);
      }
      inbuf[n] = '\0';

      if(strncmp(inbuf, "get", 3) == 0)
      {
        filename = inbuf+4;

        if(stat(filename, &filestat) == -1) {
          sprintf(outbuf, "Error retrieving file %s",
            strerror(errno));
          error_out(ns, outbuf);
        }
        if(!S_ISREG(filestat.st_mode)) {
          sprintf(outbuf, "Error retrieving file. Not a file");
          error_out(ns, outbuf);
        }

        if( (file = fopen(filename, "r")) == NULL) {
          sprintf(outbuf, "Error retrieving file %s",
            strerror(errno));
          error_out(ns, outbuf);
        }

        printf("filename %s size %d\n", filename,
          filestat.st_size);
        sprintf(outbuf, "Got comannd get file %s", filename);
        msglen = filestat.st_size;
        send(ns, &msglen, 4, 0);

        while( (n = fread(outbuf, 1, 1024, file)) > 0) {
          send(ns, outbuf, n, 0);
        }
```

```
            fclose(file);
          }
          else {
            sprintf(outbuf,"Unknown command %s", inbuf);
            error_out(ns, outbuf);
          }

          close(ns);
          exit(0);
        }
        else { /* parent */
          wait(status);
        }
      }

      close(ns);
      close(s);
      exit(0);
}
```

Appendix 2 : Type Enforcement File

```
# create the domain
type naive_t, domain;

# create the types
type naive_exec_t, file_type, sysadmfile, exec_type;
type naive_file_t, file_type, sysadmfile;

# make the domain to transition from staff_t to naive_t
domain_auto_trans(staff_t,naive_exec_t,naive_t)

# allow the root user in the staff_t domain the ability to search the
# directory
allow staff_t naive_file_t:dir search;

# allow the staff_r role access to the type/domain
role staff_r types naive_t;

# allow the app to run in a terminal
allow naive_t staff_tty_device_t:chr_file { getattr ioctl read write };
allow naive_t getty_t:fd use;

# allow the process to traverse directories
allow naive_t root_t:dir search;
allow naive_t etc_t:dir search;

# allow the process to search the directory in its domain
allow naive_t naive_file_t:dir search;

# allow the process to work with libraries
allow naive_t ld_so_cache_t:file { getattr read };
```

```
allow naive_t lib_t:dir search;
allow naive_t lib_t:lnk_file read;
allow naive_t shlib_t:file { execute getattr read };

# allow network operations
allow naive_t self:tcp_socket { accept bind create listen setopt };
allow naive_t port_t:tcp_socket name_bind;
allow naive_t node_inaddr_any_t:tcp_socket node_bind;

# allow the process to receive and send thru eth0
allow naive_t netif_eth0_t:netif { tcp_recv tcp_send };

# allow the process to fork and process signals
allow naive_t self:process { fork sigchld };

# allow the process to read and write to the socket
allow naive_t self:tcp_socket { read write };
allow naive_t node_t:node { tcp_recv tcp_send };
allow naive_t port_t:tcp_socket { recv_msg send_msg };

#allow naive_t usr_t:dir search;

# allow the process to read a file in it's domain
allow naive_t naive_file_t:file { getattr read };
```

References

[1] CERT/CC Statistics 1988 – 2004, <http://www.cert.org/stats/cert_stats.html#incidents>

[2] SELinux Background. <http://www.nsa.gov/selinux/info/index.htm>

[3] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, J. Lepreau. Flask Security Architecture: System Support for Diverse Security Policies. <http://www.cs.utah.edu/flux/papers/flask-usenixsec99-abs.html>, 1999

[4] Peter Loscocco, Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. <http://www.nsa.gov/selinux/papers/freenix01.pdf>, 2001

[5] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, J. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. <http://www.nsa.gov/selinux/papers/inevitablility.pdf>, 1998

[6] D. Ferraiolo and R. Kuhn. Role-Based Access Control. Proceedings of the 15th National Computer Security Conference, 1992.

[7] B. McCarty. SELinux: NSA's Open Source Security Enhanced Linux. O'Reilly Media, 2005

[8] P. A. Loscocco and S. D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. <http://www.nsa.gov/selinux/papers/ottawa01.pdf>, 2001