Process-Based Access Control (PBAC)

Søren Nøhr Christensen, Emmanuel Fleury,
Kristian Sørensen, Michel Thrysøe

Aalborg University – Computer Science Dept. & CISS,
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark.
{snc,fleury,ks,mthrysoe}@cs.aau.dk

Abstract

Access control mechanisms can be seen as the last line of defense of an operating system when a flaw occurred. Yet, Unix systems usually follow a Discretionary Access Control (DAC) model whereas the security of this model seems to reach its limits nowadays.

This article presents a new access control model called Process-Based Access Control (PBAC) for embedded Unix systems, using the process-hierarchy and embedded restrictions on the executables to grant or deny access to objects. This scheme has been coupled with a public/private signing mechanism of executable files in order to ensure the authenticity and integrity of the restrictions. We believe PBAC to be both easier to configure than Mandatory Access Control (MAC) model and with a more adequate granularity to fight most of the usual security threats encountered in real-life.

The Umbrella Project[1] is an implementation of the full PBAC scheme as an open-source patch for the Linux kernel 2.6 which is mainly targeting consumer electronics ranging from mobile phones to settop boxes.

Keywords: Access Control, Discretionary Access Control, Mandatory Access Control, Digitally signed binaries, Process-Based Access Control, Consumer electronics, Security framework, Security, LSM.

1 Introduction

Access control [22, 20] is a central point in the security of an operating system. Indeed, if for any reason the system is flawed, the access control mechanism is the last way to contain the attacker. Until now Unix systems have been using the Discretionary Access Control (DAC) model, where the user ID of the owner is used to mediate access to objects. But, as the number of process hijacking attacks has increased (buffer-overflow, format bugs, race conditions, etc.) this model seems to reach its limits nowadays. In replacement of this mechanism, many others have been proposed. The most famous is obviously Mandatory Access Control (MAC, Bell-LaPadula [6]), where the access rights are based on the sensitivity of each object to the system and are managed by the administrator. Or, also, the Role-Based Access Control (RBAC) [10, 21] mechanism which restrict access based on the *role* of each user.

This article introduce a new access control mechanism for embedded systems, namely Process-Based Access Control (PBAC), that we believe both easier to configure compared to MAC and with a more adequate granularity to fight most of the usual security threats encountered in real-life. PBAC is a model using the process-hierarchy and restrictions on executables to decide whether the access to the object is granted or not. It can be combined with DAC and can implement MAC, but it is also more powerful as it provides a *restricted fork* system call which allow to change on demand the restrictions within the execution of a program. As we will see in the following, the PBAC model is based on the belief that programmers know better than any administrator the needs of their softwares and can embed restrictions for a PBAC system in their binaries. This mean that, on the contrary to MAC, most of the restriction is already coded and do not need a very in-depth configuration.

At this point, you should note that it is *crucial* that the origin of the executable could not be faked nor the restrictions embedded by the programmer in the binary be modified by an attacker. In some specifically hostile environments, we have to ensure that such a requisite is met. Therefore, we introduced the possibility to sign a binary with the private key of the programmer. At execution time, the signature of the binary is checked in kernel-space against the public key of the programmer. Once a binary has passed such test, it is either *trusted* or declared *untrusted* and discarded or sandboxed.

Finally, the Umbrella project is a full implementation

[1]This work has been done with partial support of CISS (Center for Embedded Software Systems) and TDC (Largest Danish Telecommunications provider).

of the PBAC scheme as an open-source Linux kernel 2.6 patch. The Umbrella Project is used to demonstrate the capabilities and the power of such a scheme in different consumer electronics domains.

The article is organized as follows, section 3 provides a more in depth description of the PBAC model, Section 4 comes with a detailed view of the digitally signed binaries mechanism, section 5 describes the implementation of the PBAC scheme in Umbrella, section 6 provides digitally signed binaries implementation details. Section 8 gives some benchmarks of the implementation and section 9 discuss two case study which are namely a PDA and an alarm box. Section 10 finally concludes this articles and gives further work and possible improvement of the scheme and the implementation.

2 Related Work

Other projects have been exploring different access control mechanisms. For the example, the Medusa DS9 project implements MAC by using a virtual space model, where a virtual space grants access to a resource and access to the virtual space is enforced by a access vector placed on objects and subjects in the system, as described in [29].

The LOMAC project provides a two-level MAC implementation with compile time policy, described in [12, 11]. LOMAC divides the system into two integrity levels; high and low. The high level contains critical system components. The low level contains client and server processes that read from the network, local user processes and their files. Once the integrity level is assigned to a file it is never changed, but high-level processes can be demoted on run-time, if they read low-integrity data. It is however not possible to increase the integrity level of a process once it has been demoted.

The Linux Intrusion Detection System (LIDS) is a kernel patch and administration tool for strengthening the security in Linux. LIDS includes a reference monitor and MAC and has furthermore a port scan detector, a file protection system and a mechanism for protection of processes; this is described in [28, 14].

The RSBAC project use the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula [1]. All security relevant system calls are extended by security enforcement code. This code calls the central decision component, which in turn calls all active decision modules and generates a combined decision [2]. RSBAC includes a Mandatory Access Control module, a Malware Scanner that can detect Linux viruses and a Process Jail that is a enhanced version of the chroot feature.

The general security framework, Linux Security Modules (LSM), is described in [27]. By itself, the LSM does not provide any additional security. It adds void security fields to kernel objects and framework of security hooks to mediate access to these objects. It is the base of many security systems for Linux including Security-Enhanced Linux, DigSig and the Umbrella implementation described in this article.

Security-Enhanced Linux (SELinux) is developed by NSA and is native in the vanilla Linux kernel source; the system implements several different MAC schemes and has the ability to exchange the security decision-making code to implement another scheme. SELinux is based on LSM and is described in [24, 25, 23].

Immunix has made a LSM based security mechanism called SubDomain, presented in [7]. SubDomain provides a least privilege mechanism that is based on programs. The focus on programs as opposed to users minimizes performance, administration and implementation costs.

In a similar manner, the digitally signed binaries have also been implemented in projects such as the BSign project which provide means of inserting a GPG [5] signed SHA1 checksum [9] of a ELF binary file into the ELF header; the project is hosted at the Debian Linux web site.

Or, the DigSig project includes a LSM based kernel module that can verify BSign signed files [26, 4]. Verification of binaries is a solution to the problem of malicious executables like virus and other malware. Some of the code for authentication in the Umbrella project is inspired from the DigSig project.

3 Process-Based Access Control

The concept of *process-based access control* is a new access control scheme where the security policy is enforced on individual process. Processes are the lowest entity on which it makes sense to perform access control. Taking the access control one level deeper, to threads, would not make sense, since threads share memory space. When threads share memory space, access control cannot be used to protect threads against each other. The level of granularity obtained by enforcing access control on a process level is very fine, and has excellent possibilities. A good example of this is an email client, where the process that handles attachments can be sandboxed, with the result, that execution of an email borne virus also would be sandboxed. Software that handles insecure or untrusted data, can use this scheme to handle unsafe data in a safe manner.

The design is based on restrictions as opposed to permissions. A restriction is a rule on a process that denies the process access to the a given resource. On the contrary to many other security scheme the PBAC scheme contains only *one* fixed global security policy.

Global Policy 1 *Given that p_1 and p_2 are nodes in the process tree P and p_1 has the restriction set r_1 and p_2 has the restriction set r_2. r_1 and r_2 are sub-sets of R which is the set of all possible restrictions.*

If p_1 is a descendant of p_2 then r_1 is a superset of r_2.

This policy implies that all processes are always at least as restricted as their parent and it ensures that the ownership of processes does not affect the process-based restrictions. It ensures that an attacker does not gain access to the entire system, if he hijacks a process owned by a superuser. The global policy ensures that the least privilege principle is enforced on the system.

For example, on the figure 1 we can see the three ways of starting a new process and the impact of the PBAC in each case. First, the *parent process* is starting an *email client* via an `exec()` system call. As a consequence of the PBAC scheme, the resulting process is inheriting of the restrictions of the parent (access denied on `/boot`) and the restrictions attached to the executable (access denied on `/etc`). The *email client* can then either `fork` and clone the restrictions in the resulting process or perform a *restricted fork* and restrict further the child. For example, executing an evil mail attachment can be sandboxed by denying the access to the address book and the network (see figure 1).

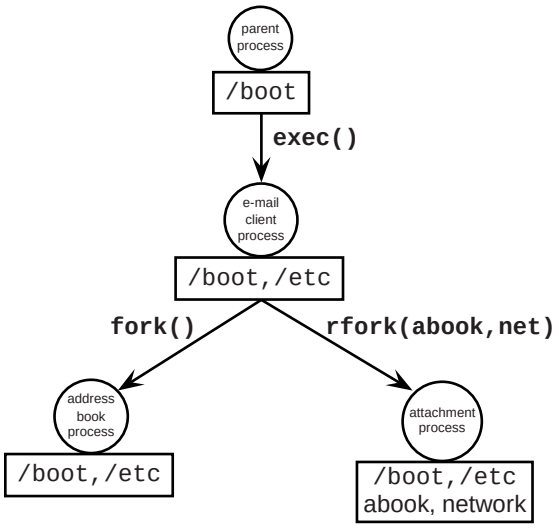

Figure 1: Process Execution and Restricted Fork.

PBAC works in combination with DAC instead of replacing it. PBAC does not differentiate between read, write and execute restriction, but provides access granted or access denied. If PBAC grants access to an object, DAC is used to decide what kind of access is permitted, possibly none. That is, if either DAC or PBAC denies access to a given object, access is denied. DAC does not provide any defense against process hijacking, where a hijacker gains the access rights of the attacked processes owner. PBAC can contain such attacks by restricting individual processes to least privilege. A webserver often runs with the privileges of the superuser. To contain possible attacks, several techniques are used, some processes change owner and are run using chroot. If the webserver is properly restricted using PBAC, all of these different measures could be avoided and still the attacker would not have access to anything besides the resources that the individual webserver processes may access.

3.1 Restricting a Process

Examples of restrictions are:

- *no access to network*,
- *no access to /foo/bar*,
- *not permitted to create processes*,
- ...

There are tree ways for setting restrictions on a new process exists; namely inheritance of restrictions, execute restrictions and a restricted fork.

Inheritance Enforce the Global Policy. This policy ensure that the security policy of a program is not circumvented by forking new processes, a hijacked process cannot raise its privileges. It is **not** possible to change the restrictions of a running process, as this could be used to violate the Global Rule.

Execute restrictions A set of restrictions attached to a binary file. When the binary is executed the execute restrictions are applied to the resulting process. This option provides transparency, in that all programs can contain their on security policy as a set of execute restrictions.

Restricted fork Is a wrapper for the normal fork system call, which allows the parent process to set additional restrictions for the child process. This enables a very fine granularity of the security policy *within* the program, where a child process handling dangerous data can be sandboxed. Sub-processes which are intended for a very specific task, e.g. rending a document, could be easily sandboxed to least privilege.

Process-based access control provides a transparent access control mechanism with a granularity fine enough to design security policy for individual processes.

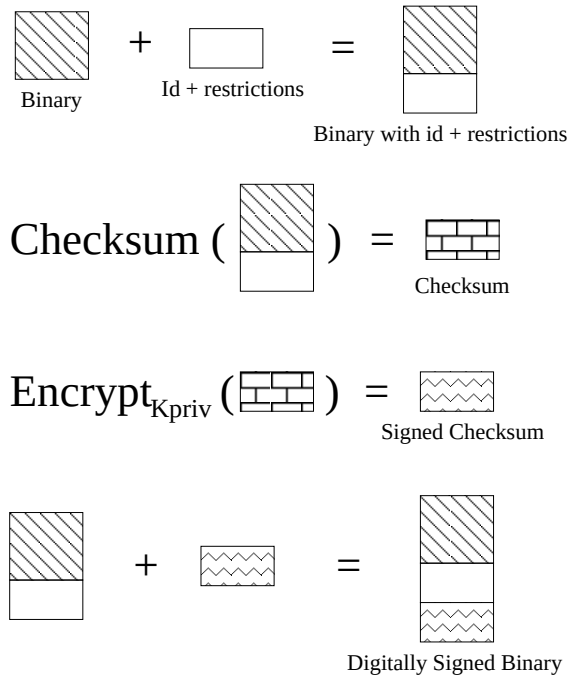

Figure 2: Creating a DSB.

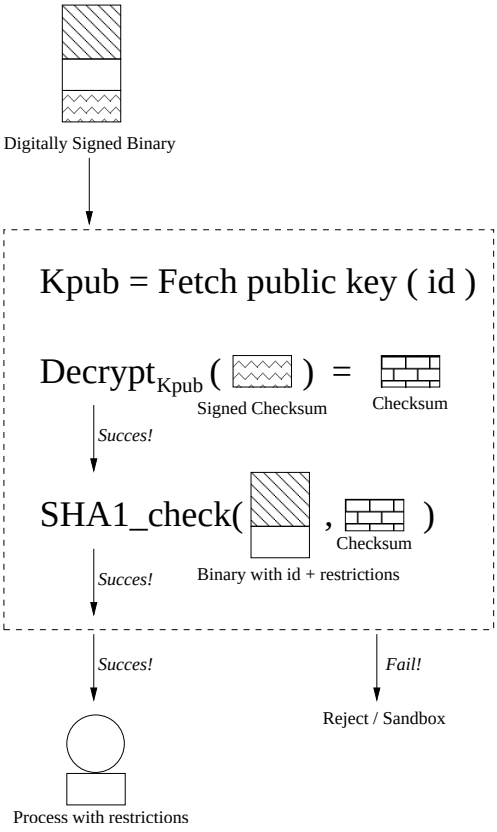

Figure 3: Authenticating and executing a DSB.

4 Digitally Signed Binaries

Digitally signed binaries (DSB) provides two features, namely authentication and integrity. Authentication of binaries to verify the origin of binaries on a system and integrity to prevents execution of binaries that have been tampered with.

Authentication of binaries is a way to control what is executed on a system and without it the user is able to execute software that is a threat to the system. Authentication can be obtained using public key cryptography [18]. The vendor of a given piece of software computes a signature from the software, using his private key. When executed, the vendor's public key can be used in performing the authentication.

Ensuring the integrity of binaries is important to prevent execution of software that has been tampered with and it is done using a checksum. This does not prevent that malicious code is injected into software, but it prevents the execution of such software. The procedure for signing a binary is seen in Figure 2.

Using authentication and integrity checking allow to separate binaries in to two categories, namely *trusted* and *untrusted*. Trusted binaries are binaries that can be positively checked for authenticity and integrity. Untrusted binaries are binaries with an invalid or no signature at all, binaries with no corresponding public key on the system or authenticated binaries with invalid checksum. Figure 3 shows how a trusted binary is executed, while an untrusted is denied execution or sandboxed.

Using the DSB scheme jointly with the PBAC scheme, provides the possibility for distributing the access control policy *with* the binaries in a secure way, which is also seen in Figure 3. Tampering with the set of execute restrictions can be effectively prevented using the integrity check. Together the two concepts can provide tamper free software with a secure embedded security policy, and which origin can be authenticated.

4.1 Key Management

The DSB scheme requires that only public keys from trusted vendors are available in the kernel. Two approaches to this are possible.

One way of handling this is to compile the public keys of trusted vendors into the kernel itself, making it impossible to tamper with them nor add new keys. This approach is feasible if the provider of the device knows what vendors should deliver software to the device at time of roll out. This approach is very easy to implement, but also very inflexible. Another issue is that if the private key of one or more vendors is exposed, it will be possible for an attacker to become trusted, and keys can only be changed by changing the entire kernel.

On the system public keys most be protected from tampering and substitution. This is done by storing them in a kernel ring protected by the kernel. The kernel writes the key ring to disk, to ensure persistent storage. To protect the public keys from unauthorized access on disk, *all* processes will be restricted from this resource. A powerful feature of inheriting restrictions from parent processes to children, now shows its strength: It is enough to restrict the top level process from accessing the key ring on disk, and this will automatically be inherited by all other processes.

From Linux 2.6.10 an option for including digital keys in the kernel was added. This feature can be used to build a keyring by a searchable sequence of keys. By default each process is equipped with access to five standard key rings: UID-specific, GID-specific, session, process and thread. It is believed that this new feature can be used to implement the keyring in a Linux implementation.

The key management scheme is future work and this section described ideas for the future design.

5 PBAC Implementation

Two data structures are used to store restrictions in Umbrella. A bit-vector for the capability restrictions and a tree structure for the file system restrictions. Both structures are stored in a `security_struct` bound to every process.

```
struct security_struct {
  bitvector cap_res;
  bitvector child_cap_res;
  struct fsr *file_system_res;
  char *child_file_system_res[];
};
```

`child_cap_res` and `child_file_system_res` are two fields used by the `rfork()` wrapper to temporary store the restrictions for the next child process. This partition of the design in two parts, gives a necessary combination of performance and flexibility.

5.1 Capability Restrictions

The capability restrictions are implemented in a bit-vector of 32 bit, where each bit represents a capability restriction. The bit-vector library used in Umbrella has been made specifically this project and it is implemented with simplicity and performance as goals. The simple interface is as follows.

```
bitvector bv_create(void);
void bv_destroy(bitvector bv);

void bv_bit_on(bitvector bv, int index);
void bv_or(bitvector result, bitvector a,
           bitvector b);
void bv_reset(bitvector bv);

int bv_testbit(bitvector bv, int index);
```

Restriction	Enforcing hook
Ability to send signals	`task_kill()`
All networking	`socket_create()`
Fork new processes	`task_create()`

Table 1: Capability restrictions in Umbrella and the LSM hooks used for enforcing them.

The functions `bv_create()` and `bv_destroy()` are memory handling functions. The `bv_bit_on()` function is used to set a bit, to denote that the restriction associated with the given bit is set for the process in question. Checking if a restriction is set is done through `bv_testbit()`. When a new process has been forked using `rfork()` the parent process' `child_*_restriction` fields must be reset and `bv_reset()` is used to do this. For combining restrictions from inheritance, `rfork()` and binaries the function `bv_or()` is used to quickly create the new child's capability bit-vector.

The list of capability restrictions implemented in Umbrella is shown in Table 1 together with the LSM hook used to enforce this restriction. The hooks are defined in `include/linux/security.h` and an example is the `socket_create` hook.

```
int (*socket_create) (int family, int type,
                      int protocol, int kern);
```

The nature of the LSM framework ensures, that this hook is called with four parameters to check permissions prior the creation of a new socket. The `family` contain one of the currently 32 supported protocol families defined in `include/linux/socket.h`. The `type` variable holds the requested communications type, `protocol` contains the requested protocol and `kern` defines if its a kernel socket which is being created. Below is the actual implementation from Umbrella.

```
struct security_struct *s = current->security;

switch (family) {
  case 2:
    decision = bv_testbit(s->cap_res, NOIP);
    break;

  case 23:
    decision = bv_testbit(s->cap_res, NOIRDA);
    break;

  case 31:
    decision = bv_testbit(s->cap_res,
                          NOBLUETOOTH);
    break;
}

if (decision != 0)
  decision = -EPERM;

return decision;
```

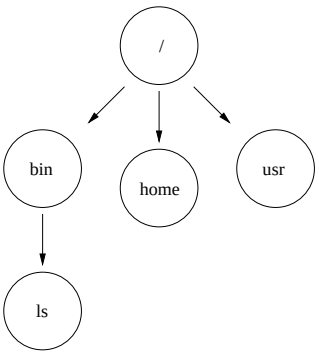

Figure 4: An example of a tree of `fsr` structures, where a particular process is restricted from file `/bin/ls` and the directories `/home/`, `/usr/bin/` and `/usr/lib/` and everything below them.

In the Umbrella implementation, `socket_create` supports setting restrictions on access to IP, infrared and bluetooth address families. First the security structure from the `current` process is fetched and stored in the variable `security`. Then the `family` variable is checked and if a supported family is found, the function `bv_testbit` is used to check, if a restriction is set for that family on the current process. If needed the granularity of Umbrella can be increased by implementing more of the socket families or adding a check of the protocol used.

The code for the signal restrictions is build after the same scheme and implemented in the `task_kill` hook. A `signal` variable is used to see what signal is sent.

The hook `task_create` is called before any new process is forked, and can therefore be used to mediate the creation of new processes. The implementation is very simple, it checks the corresponding bit in the current process using `bv_testbit`.

Other possible capability restrictions include hard and soft linking and creation of device nodes.

5.2 File System Restrictions

The file system restrictions are implemented in a tree structure. This data structure ensures fast restriction checking and it is conceptually simple to understand. Figure 4 shows an example of a file system restriction tree.

The file system restriction trees are created from instances of this simple structure.

```
1 struct fsr {
2   char *name;
3   struct fsr **successors;
4 };
```

Below is the interface to the file system restriction trees.

```
1 struct fsr *fsr_create(void);
2 void fsr_destroy(struct fsr *target);
3
4 int fsr_insert(struct fsr *root, char **path);
5 struct fsr *fsr_copy(struct fsr *source);
6
7 int fsr_check(struct fsr *root, char **path);
```

Functions `fsr_create()` and `fsr_destroy()` are for memory handling. In the case of inheritance, the parent's file system restriction tree must be copied to the child, and the function `fsr_copy()` is implemented to this purpose. Since each "layer" in the tree is implemented as an array, copying a tree can be done efficient using `memcpy()`.

The most important and interesting functions in the file system restrictions interface are `fsr_insert()` and `fsr_check()`. Especially the `fsr_check()` function most be efficient since it is called every time an inode is accessed. The `fsr_insert()` function must prune the tree if a wider covering restriction is inserted. If the restriction `/bin` is inserted in the tree in Figure 4, the node with the name `ls` must be freed. Below is the algorithm that is implemented in the function `fsr_check()`.

```
1  while (path is not empty) do {
2    if (successors != NULL) {
3      extract the first name_i and remove it from
           the path;
4      compare name_i to each the successors;
5
6      if (none of the successor match name_i)
7        return allowed;
8      }
9      else
10       return denied;
11   }
12 }
13 return allowed;
```

The algorithm iterates over `path`, which is the path to the accessed resource. Each individual directory or file of the path is compared to the corresponding level in the file system restriction tree. The access is denied if a match is found for a node with no successors. In the worst case, the algorithm runs in linear time, over the length of the path.

The LSM hooks used in Umbrella for protecting the file system are `inode_permission`, `inode_link`, `inode_unlink`, `inode_rename`, `inode_mkdir`, `inode_setattr` and `inode_create`.

All the LSM hooks which are intercepting calls to the file-system are implemented in one simple and generic `file_hook_wrapper`, which is listed below. The call that makes a lookup in the processes FSR tree is done in line 6 using the algorithm outlined above.

```
1 static inline int file_hook_wrapper(struct
    dentry *dentry) {
```

```
2  if (parse_path(dentry, path) == -EOVERFLOW)
3    ss_decision = -EOVERFLOW;
4
5  else {
6    ss_decision = fsr_check(cur_security->fsr,
         path);
7    if (ss_decision)
8      ss_decision = -EPERM;
9  }
10 return ss_decision;
```

The purpose of this function is to find the right `dentry` structure. The `dentry` structure (Directory ENTRY) is defined in `include/linux/dcache.h`. Once the `dentry` is found, the file-system path is extracted via the `parse_path()` function. The file system path is checked against the current process' fsr tree using `fsr_check` to see if the current process is restricted from this particular path.

5.3 Setting Restrictions

Each of the three ways for setting restrictions on a new process, in the Umbrella implementation, are closely connected to a system call. POSIX is an international standard with an exact definition and a set of assertions which can be used to verify compliance [15].

The POSIX `exec()` family of calls is used to replace the current process image, essentially executing a binary. PBAC introduces execute restrictions, and these must be set on the process that is a result of the `exec()` call. This is done by implementation of the LSM hook `bprm_check_security()`, in which also handles issues regarding digitally signed binaries and this is described in detail in Section 6.

The POSIX `fork()` call is used to create a new process, that is a copy of the parent except that it has its own process id. The inheritance of restrictions introduced by PBAC does nothing to change this, since the child gets a copy of the parents restrictions. However, due to the implementation of LSM, Umbrella most actively copy the restrictions, and this is implemented in the LSM hook `task_alloc_security()`, called whenever a new process is created.

The Umbrella implementation adds `rfork()`, which is a user space wrapper for the `fork()` system call. The purpose of `rfork()` is to give developers the possibility to sandbox certain processes in a program. It makes it possible to handle insecure data in a secure manner by restricting to least privilege. For testing purposes mainly, a proc file system interface have been developed for this, it is elaborated in Section 7.

6 DSB Implementation

Digitally signed binaries in Umbrella have the layout seen in Figure 5. They carry a vendor id to ensure a fast lookup of the vendors public key in the kernel. A checksum of the binary is signed using the vendors private key to ensure integrity and authentication. If the DSB scheme is used together with the PBAC scheme, the digitally signed binaries also include the security policy for the binary, stored as a set of restrictions.

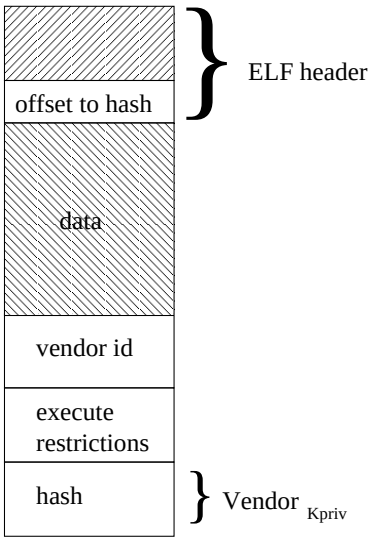

Figure 5: Layout of a digitally signed binary.

6.1 Signing a Binary

Creating a signed binary is elaborated in section 7.2. The checksum (SHA1 [9]) is computed from the binary data, the vendor id and the restrictions if any. The checksum is the signed using the vendors private key, ensuring authentication. This is done using the RSA algorithm [18]. Signing the binary data prevents that an attacker can hide malicious code in a legitimate binary. If existing, the restriction set is signed as well, to prevent that an attacker lowers the security by removing or tampering with the restrictions on a binary. Signing the checksum is very much faster than signing the entire binary, yet still providing the security needed.

6.2 Executing a Signed Binary

Mediation of binaries execution is used to perform two security related tasks; integrity checking of the executable and the transfer of execute restrictions from the executable to the new process. The LSM hook `bprm_check_security` is called whenever a file is executed and is therefore used for this purpose. More precisely this hook is called whenever the `execve` system call is searching for a binary handler. If the security checks succeeds, the function `load_elf_binary()` is called to load the binary into memory, see Figure 6.

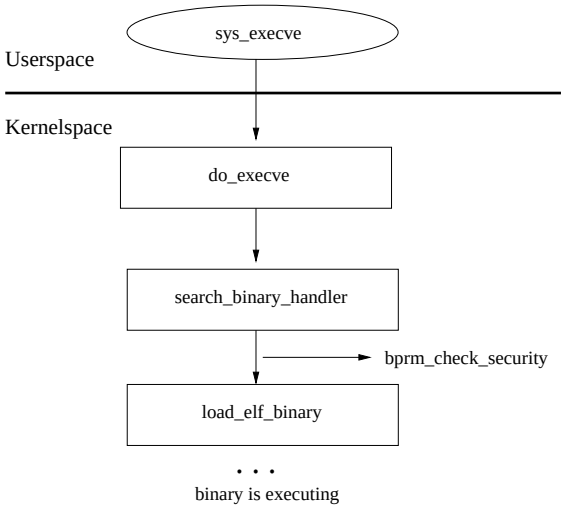

Figure 6: Control flow of a binary execution.

```
int umb_bprm_check_security (struct
    linux_binprm * bprm) {
  return umb_handle_signature(bprm);
}
```

The LSM hook `bprm_check_security` is calling `umb_handle_signature` to handle authentication and the assignment of execute restrictions. As the function is the entry point for the signature part of Umbrella, all other functions in this part, will only be evoked as a result of statements in this function.

```
int umb_handle_signature(
                 struct linux_binprm *bprm) {
...
elf_hdr = umb_read_elf_header(file);
...
elf_shdata =
  umb_read_section_header(file, size,
                          elf_hdr->e_shoff);
file_sig =
  umb_find_signature(elf_hdr, elf_shdata,
                     file, &signature_offset);
...
retval =
  umb_check_signature(elf_shdata, file_sig,
                      file, signature_offset);
...
if ( retval == 0) {
  parsed_usig =
    umb_parse_usig(bprm,file_sig->usig);
...
  set_execute_res(parsed_usig);
...
}
...
return retval;
```

In the first part of the function the ELF data is read into memory. The function `umb_read_elf_header()` returns an `elfhdr` struct containing the actual ELF header stored in the variable `elf_hdr`. Then, the function `umb_read_section_header()` is called to transfer all entries in the section header table of the ELF to `elf_shdata`. If `umb_read_section_header` fails, the ELF is invalid and `-EPERM` is returned to deny execution or sandbox the resulting process.

The function `umb_find_signature()` is called to extract the signature. The function takes four parameters; `elf_hdr` is an ELF header, `elf_shdata` holds all entries in the section header table of the ELF, `file` is the file handler of the binary and `sh_offset` is the offset of the signature section in the ELF binary. The return type is a `dsig` struct, which holds the BSign and Umbrella signature.

```
struct dsig {
  char *bsig;
  char *usig;
};
```

The function `umb_check_signature()` checks the BSign signature using the appropriate public key and fetches the public key if not already loaded. When the key is loaded the actual authentication is initiated using `umb_verify_signature()` and the return value is passed to `umb_handle_signature()`. If the verification of the BSign signature is successful, the Umbrella specific signature parts (ID and execute restrictions) are extracted using the function `umb_parse_signature()` which returns a `parsed_sig` struct containing the Umbrella signature.

If the signature was parsed successfully and the current process has a security field the execute restrictions found in the signature are transferred to the current processes' child `capability` and `fsr` fields using the function `set_execute_res()`. Currently the implementation does not include the caching of restrictions in the LSM security fields on inode structures.

6.3 RSA vs. Elliptic Curves

Two schemes were considered for the public key algorithms, RSA Public key Cryptosystem [18] and Elliptic Curve Cryptosystem (ECC) [16, 17]. Elliptic curve cryptography can provide the same level of security with smaller keys, compared to the RSA scheme and other conventional discrete algorithms. According to [13] and [8] RSA is faster at verifying a signature but slower at signing. To the scheme of digitally signed binaries, verifying the signature is more important. As the Umbrella implementation is aimed at handhelds and embedded systems space is also a consideration, which is why ECC is interesting. For the Umbrella implementation, however, performance was chosen over space considerations, which is why RSA was used.

6.4 Optimization Issues

In the scheme presented the digital signature must be verified every time a binary is executed. This is a point for major optimization. This can be solved by using the LSM security field associated with each inode. This field is protected by the kernel and therefore secure but is not persistent and thus perfectly suited for the task. The idea is to attach a structure like the one below, to every inode.

```
1 struct inode_security_struct {
2   int trusted;
3   bitvector cap_res;
4   struct fsr *file_system_res;
5 };
```

All files are *untrusted* at first and must be reset to this whenever they are modified and this can be done using the all ready implemented inode mediation hooks. When a binary is executed, the `trusted` field is checked and if set, the binary is executed without further. If the `trusted` field is not set, the DSB scheme is followed as in Figure 3 and the `trusted` field is set if verification was successful.

If the PBAC scheme is in use as well, the `cap_res` and `file_system_res` fields are used to cache the restrictions associated with the binary.

After a reboot the security fields are reset and the procedure for first time execution must be performed again. This scheme is especially efficient for systems that are seldom rebooted.

Currently the implementation does not include the caching of restrictions in the LSM security fields on inode structures, this is a major pending optimization.

7 Using Umbrella

In the following, two examples of how to use Umbrella in practice. First we will explore the use of the restricted fork and then see how to sign binaries and how this process may be incorporated into e.g. a Make file.

7.1 Restricting Processes

The Umbrella kernel patch is accompanied by a library `libumbrella`, which is a small library that implements the `rfork` wrapper for the `fork` system call.

The code snippet below illustrates use of the restricted fork. We import `umbrella.h` in line 1, we define the non filesystem restrictions in line 5 and the filesystem restrictions in line 6. Instead of making a normal `fork` in the `switch` statement at line 8 we call `rfork`.

The child process created by the restricted fork is now restricted as specified. When it breaks out of the `switch` structure it executes a terminal in line 17, which inherit the restrictions.

```
1  #include<umbrella.h>
2
3  int main() {
4    int pid;
5    int  cr[] = {IPNET, BLUETOOTH};
6    char *fsr[] = {"/boot", "/foo", NULL};
7
8    switch (pid = rfork(cr, fsr)) {
9      case 0: /* child */
10       break;
11     case -1:
12       printf("rfork ERROR\n");
13       return -1;
14     default: /* parent */
15       exit (0);
16   }
17   system("/usr/bin/xterm");
18   return 0;
19 }
```

This example demonstrates the ease and simplicity of adapting programs to utilize the possibilities Umbrella. The only effort required by developers is to consider and set appropriate restrictions before forking children. If attachments is allowed executed from an email client, this attachment could be denied access to e.g. network and address book. This would limit the spreading of worms, that we have seen on personal computers through the last years.

The compiled Umbrella library is also of the ELF format, which enables signing. Thus, if an attacker exchanges the binary umbrella library, and this is not signed with a key present within the kernel keyring, the library will not be loaded into memory during runtime, and thus the program calling the umbrella will fail.

7.1.1 Using the proc Interface

An interface to set restrictions on new processes have been implemented through the proc file system. This is mainly for testing and developing purposes, and it offers a limited interface compared to using the `rfork()` wrapper. Note, the current implementation of the procfs interface only offers the possibility for setting the capability restrictions. The following Python example shows how to make use of it.

```
1 >>> import os, struct
2 >>> pfd=os.open('/proc/umbrella', os.O_WRONLY)
3 >>> deny_ipnet = struct.pack('iii', 1, 8, -10)
4 >>> os.write(pfd, deny_ipnet)
5 12
6 >>> command = os.popen('ping localhost')
7 Operation not permitted
```

In line 3 the binary data are prepared to be written to the proc filesystem. All commands have the same basic layout, namely the command ID followed by the arguments. The command ID is a 32 bit integer and the arguments naturally depend on the command. Here, `pack()` comes with the command 1 (setting capability restrictions) and with the restrictions 8 (IPNET) and -10

(list termination). There are two commands available, namely for setting the capability restrictions (command 1) and debugging by printing the current process' security structure to the kernel log (command 4). Rely totally on the procfs interface for setting restrictions has been discussed in Section 5.3.

7.2 Signing Binaries

The process of signing binaries is depicted in Figure 2 on page 4. To automate this process Umbrella provides a small Ruby script. As arguments, it takes the vendor ID, which is an ID for finding the appropriate public key in the kernel, the capability- and file system restrictions. From this, a string is created and appended directly to the end of the binary file. Now, to make the SHA1 hash, sign it and place it in the ELF header, the tool BSign (developed by Debian Linux) is used. BSign requires that GNU Privacy Guard is present together with the private key of the user signing the file.

The process can be incorporated into Make files, like the following example shows.

```
1  LDFLAGS=-lumbrella
2
3  all: compile sign
4
5  compile: program_to_be_signed
6
7  sign:
8      sign_file.rb --id=UmbrellaInc \
9                   --cr=BLUETOOTH:IPNET \
10                  --fsr=/etc/shadow:/var/keys \
11                  --file=program_to_be_signed
```

When the above Make file is initiated, the program is compiled as normal and further the script for signing the file is called.

Running an Umbrella signed binary on a system without the Umbrella patch applied, would not be noticeable as long as the Umbrella library is present (the program links to this). The kernel simply ignore unknown system calls (rfork) and also the extra ELF header and the appended restrictions is ignored.

8 Umbrella Benchmarks

The Umbrella implementation has been benchmarked for performance. One requirement of an access control mechanism is that it does not introduce an unacceptable slowdown on the system. The following presents some benchmarks that has been done to investigate the overhead of Umbrella. The benchmarking is run on a machine with the following specifications.

- Intel Pentium 4 1.8 GHz
- CPU cache 512 KB
- 512 MB RAM
- Red Hat Linux 9
- Linux-2.6.9

8.1 Benchmark Details

Four different benchmarks has been performed, each ran five times, on a system running an Umbrella patched Linux and one that ran a clean Linux.

1. Process creation – 40k processes
2. File system access – unpacking Linux-2.6.9
3. Interactive simulation – compilation of Linux-2.6.9
4. DSB benchmark – compilation with signed and unsigned tools

The process creation benchmark (1) will stress the functions involved in assigning security information, namely allocation of memory for security structure and inheritance of restrictions. The benchmark is done by timing the execution of a script that creates 40.000 processes. This is done for different sets of restrictions. The results can be found in Table 2.

The file system access benchmark (2) is aimed at the performance of the functions involved in checking file system restrictions. This benchmark will also be performed for different settings of restrictions. The benchmark is performed by unpacking a Linux kernel tree, which will create approximately 19.000 files. The results can be found in Table 3.

The interactive simulation benchmark (3) is a combination of benchmark 1 and 2. This benchmark will compile the Linux 2.6.9 kernel, which will create a large number of processes and access a large part of the 19.000 files in the kernel tree. This benchmark simulates interactive behavior, where files, process and I/O wait is involved. The results can be found in Table 4.

The benchmark of the DSB part (4) of the implementation will show the overhead of using signed binaries as opposed to unsigned binaries. The benchmarks is done by comparing the compilation of a Linux kernel on a clean system, with a compilation on a system with Umbrella, where the tools `gcc`,`as`, `ld`, `bash` and `make` are signed. The results can be found in Table 5.

8.2 Discussion

The Umbrella implementation is done, but not yet optimized, which makes us believe that the above results can be improved on a final system. The preliminary results show that Umbrella will not suffer from major performance issues.

System	Time	Overhead
Clean	15.8s	N/A
No restrictions	16.2s	2.4%
Only cap. restr.	16.1s	2.1%
5 fsr restr.	16.4s	3.7%
10 fsr restr.	16.5s	4.4%
20 fsr restr.	16.6s	4.9%
40 fsr restr.	16.8s	6.3%

Table 2: Results of process creation benchmark (1).

System	Time	Overhead
Clean	45.8s	N/A
No restrictions	46.3s	1.1%
Only cap. restr.	45.9s	0.1%
5 fsr restr.	46.3s	1.1%
10 fsr restr.	45.4s	-0.8%
20 fsr restr.	46.1s	0.5%
40 fsr restr.	46.0s	0.3%
3 levels deep	59.0s	28.8%
5 levels deep	58.7s	27.9%
10 levels deep	61.2s	33.2%

Table 3: Results of unpacking benchmark (2).

System	Time	Overhead
Clean	386.0s	N/A
No restrictions	385.4s	-0.1%
Only cap. restr.	390.3s	1.1%
5 fsr restr.	390.8s	1.2%
10 fsr restr.	391.1s	1.3%
20 fsr restr.	391.1s	1.3%
40 fsr restr.	391.5.0s	1.4%

Table 4: Results of compilation benchmark (3).

Unsigned	Signed	% Overhead
396s	408s	3%

Table 5: Compiling with unsigned and signed tools.

Instead of going through the results from one end to the other, the most interesting results will be discussed in the following.

In the process creation benchmark the performance of the inheritance of restrictions is put to the test. Creating such a large number of processes that does nothing, stresses the algorithms for inheritance and inserting restrictions. The results in Table 2 clearly shows the overhead imposed by these algorithms. As the number of restrictions is enlarged the overhead rises to more than 6%. This overhead is somewhat large, but with normal system use, this amount of processes will never be created without any intermediate computation or I/O-wait.

The overhead introduced by the access control functions is investigated in the second benchmark. As Table 3 shows the overhead does not rise with the number of restrictions. It does, however, rise when the depth of the restrictions is increased. This overhead rises to more than 30%. This is a suitable place for optimization.

In the third benchmark the overhead of Umbrella is minimal. The main reason for this, is that once the compiler is working, no files are accessed and no further processes are forked. We believe that this benchmark is a good approximation to ordinary use of the system, since both process creation, file system access and a plain computation is performed.

The benchmark of the DSB implementation reveals a rather large overhead, as expected. Every time a signed file is executed the signature is decrypted and the checksum is calculated. When the Linux kernel is compiled using a set of signed tools, the overhead of is 3%. This is believed to be an acceptable overhead that can be further improved by implementing caching on the LSM security fields, as described in 6.

9 Case Studies

The concepts of PBAC and DSB are very well suited for embedded systems and handheld devices like PDAs and mobile phones. In this section two *proof of concept* cases are presented, showing the versatility of the concepts. Umbrella has been adapted for both cases.

9.1 TDC Alarmbox

The largest telecommunications provider in Denmark, TDC[2], has requested a proof of concept implementation of Umbrella for a Linux based alarmbox. The alarmbox is used to fire alarms, transmission of technical data, danish army NATO POL system, banks and other high risk intrusion targets. It is based on an Intel compatible AMD ELAN SC520 processor and is supplied by the Danish

[2] www.tdc.com

company Linux In A Box (LIAB). The alarmbox has a potential market of 65,000 units in Denmark alone.

Following is the main requirements to a security solution for TDC's alarmbox.

- The box should only boot TDC's own kernel
- The box should only be able to execute software provided by TDC
- The security system should be able to function without interaction from a security administrator

The second and the third requirement can be meet by the scheme of digitally signed binaries. The first requirement can be meet by deployment of a system like those presented in [3, 19] and implemented in e.g. Trusted Computing Groups Trusted Platform Module. The combination of a trusted boot and Umbrella to protect the running system, would yield a very powerful security system.

One of the most likely attack scenarios on the alarmbox, is a situation where a customer with bad intentions installs malicious software on an alarmbox and thereafter returns it to TDC. In this case there is a possibility that the compromised box will be installed at another customer and where the malicious software may be able to deactivate the alarm systems of e.g. a bank. Umbrella can prevent the execution of software that is not signed by TDC or signed software that has been tampered with. This would effectively prevent the execution of software not provided by TDC and software that a customer has tampered with.

The alarmbox case shows that the concepts of digitally signed binaries can be used to prevent attacks from a malicious user trying to run his own software on the device. It is also an example of a system, where the absence of a security administrator rises a demand for software that has its security policy embedded – and digitally signed binaries meet this demand.

9.2 Handheld Devices

Umbrella has been implemented on a HP iPAQ 5550 running Linux. The security requirements in this case are different to the requirements of the alarmbox.

The platform is a handheld device with communications features like bluetooth, infrared, GPRS and wireless networking. These communication features combined with the flexibility of this type of devices make exploits, like the ones found on desktop computer, migrate to this new platform. Prime examples are the new worms and trojans for mobile phones, like Skulls[3], Mosquitos[4] and Cabir [5], which all have emerged recently.

Mobile phones are, as most mobile devices, designed to be single user systems. Typically, a mobile phone would be used by the owner exclusively. The recent addition of the ability to use the Internet and display multimedia contents like games, music and movie clips have greatly increased the number of possible threats to such devices.

The Mosquitos trojan, which is disguised as a cracked version of the game "Mosquitos", is an example of the kind of threats that modern mobile phones face. If executed, the trojan attempts to send expensive SMS text messages to premium rate number. This behavior resembles the dialer schemes found on ordinary desktop systems where a malicious dialer program is installed and automatically calls expensive premium services in other countries. The malicious program is only installed after the user has seen several warnings about possible dangers of installing unsigned applications. However, users are not always to be trusted. The Mosquitos trojan shows an example of how vulnerable mobile phones are to the viruses, spyware and adware, that plagues desktop systems today. This is mainly because no way of controlling access to system resources currently exist in the operating systems used on the majority of handheld devices.

Another example is the Cabir worm. Once the worm is installed it runs every time the device is booted and constantly attempts to send itself to other bluetooth enabled devices found in the proximity of the infected mobile phone. The original Cabir worm does nothing malicious but already more harmful descendants like the Lasco worm has emerged. This is an important reminder that a worm with a malicious payload spread easily.

To counter these new threats, a number of requirements must be met.

- Determine the origin of binaries
- Verify the correctness of binaries
- Limit the privileges of unknown binaries
- Protection of critical system areas

Handheld devices in general are characterized by the fact the user is normally also the security administrator. However, as countless examples show, the users cannot be expected to be proficient in system administration and the security decisions should not be left to the user.

The first three requirements are met by the digitally signed binaries in combination with the PBAC scheme. The signature of a binary is verified every time the binary is executed, which is accomplished by verifying the signature against the keys of the "trusted vendors",

[3] www.sophos.com/virusinfo/analyses/trojskullsa.html

[4] www.sophos.com/virusinfo/articles/mosqit.html

[5] www.sophos.com/virusinfo/articles/cabirhi.html

which are present in the kernel keyring. If the origin of the binary is unknown, it can be denied execution or the resulting process can be placed in a sandboxed environment. The second requirement is satisfied by the (signed) checksum in the binary ELF header, which is used to detect any tampering with the binary or the embedded restrictions.

The combination of authentication and verification would have prevented the Mosquitos trojan or Cabir worm from carrying out their primary purposes, namely the sending of premium rate SMS messages and spreading via bluetooth, simply by restricting from accessing the GSM and bluetooth network.

10 Conclusion

This paper presented PBAC, which is a novel approach to access control, that works on processes and uses the process hierarchy to maintain the global security policy that child processes are at least as restricted as their parent. By using security policies embedded in binary files and a restricted fork to set the policy for individual processes within a program, this scheme becomes transparent to the user and requires no security administrator. To ensure the validity of the embedded restrictions and the authenticity of the executable files the scheme have been combined with public key cryptography mechanism in the form of DSB. The result is a scheme that is easier to configure than the MAC model and with a more adequate granularity than DAC to fight most of the current security threats.

The scheme is used in combination with DAC. The PBAC scheme is used to ensure system integrity by protection vital areas from tampering, while normal DAC rules apply if access is granted. This combination is transparent to the user, because DAC is an integrated part of UNIX and any PBAC policies are embedded within the binary itself, thereby eliminating the need for additional configuration.

The access matrix, we believe, is the weak point for current MAC implementations, because adding a new object or subject, would require that a policy for *all* other objects and subjects is specified. This is a very demanding task even though some other MAC schemes support some degree of automation of this.

One of the foundations of PBAC is the belief that programmers are more aware of the needs of their programs than an administrator or an ordinary user. This allows the programmer to set additional restrictions, using restricted fork, in the software, whenever the program is handling input which could potentially be dangerous. An example this could be the rendering of an image, where a malformed image could possibly trigger a buffer overflow, a number of restrictions could be set to prevent the rendering thread from accessing any vital parts of the system. The result of this that unlike MAC, the restrictions are already present within the program itself and do therefore not need an in-depth configuration.

The Umbrella Project is developed as an open source project hosted on SourceForge.net. The Umbrella web site have had more than 50,000 visits since the public launch in February 2004, which is an indication of the level of interest in the subject.

The completed parts of the implementation covers restrictions on processes, which are implemented together with the intended functionality, i.e. inheritance, the restricted fork and setting restrictions from digitally signed binaries. Authentication of digitally signed binaries is in the final implementation phase.

The Umbrella implementation has been benchmarked for performance and the results are very promising. The overhead introduced by Umbrella is currently acceptable and pending optimizations is expected to improve these results further.

The future work on the Umbrella implementation includes the design and implementation of a kernel keyring that can be accessed in a secure manner from user space. This includes a scheme for deciding and controlling who should be able to manipulate the keyring. Furthermore the current implementation need to be stabilized and optimized for practical use.

The combination of process-based access control and digitally signed binaries yields a completely new security scheme for consumer electronics, which we believe is a step in the right direction toward stopping an increasing rain of attacks on CE devices.

11 Acknowledgments

The authors would like to thank the following people and organizations.

TDC for sponsoring four months of development of the Umbrella project and especially Jakob Bjerggaard, Frank Larsen, Arne Larsen and Per Lydholm for input and practical support when working with the alarmbox implementation.

CISS at Aalborg University, especially Kim G. Larsen for practical support and for sponsoring traveling. Arne Skou and Mikkel Christiansen for initial arrangements and contact to TDC.

Magnus Therning from Phillips, Eindhoven and Katherine Guo, I.P. Park and Steven Johnson from Panasonic, N.J. USA, for ideas, feedback and great discussions on the topic.

CE Linux Forum for showing great interest in the project and for interesting discussions during the security working group meeting in October 2004 and CELF plenary meeting January 2005.

References

[1] M. D. Abrams, L. J. LaPadula, K. W. Eggers, and I. M. Olson. A Generalized Framework for Access Control: An Informal Description. In *Proc. of the 13th National Computer Security Conference*, pages 135–143, October 1990.

[2] M. D. Abrams, L. J. LaPadula, and I. M. Olson. Building Generalized Access Control on UNIX. In *Proc. of the 2nd USENIX Security Workshop*, pages 65–70, Portland, August 1990.

[3] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM White Paper*, July 2004.

[4] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. DigSig: Runtime authentication of binaries at kernel level. In *Proc. of the 18th USENIX Large Installation System Administration Conference (LISA'04)*, 18th, pages 59–66, Atlanta, November 2004.

[5] M. Ashley. The GNU Privacy Handbook. http://www.gnupg.org/gph/en/manual.html, May 2004.

[6] D. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical report, MITRE Corp., Bedford, Mass., March 1976.

[7] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proc. of the 14th USENIX Large Installation System Administration Conference (LISA'00)*, pages 355–367, New-Orleans, December 2000.

[8] E. Cronin, S. Jamin, T. Malkin, and P. McDaniel. On the performance, feasibility, and use of forward-secure signatures. In *Proc. of the 10th ACM conference on Computer and communications security (CCS'03)*, pages 131–144. ACM Press, 2003.

[9] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, Internet Society (ISOC), September 2001.

[10] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proc. of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, USA, October 1992.

[11] T. Fraser. LOMAC: Low water-mark integrity protection for cots environments. In *Proc. of the 2000 IEEE Symposium on Security and Privacy (SP'00)*, page 230. IEEE Computer Society, 2000.

[12] T. Fraser. LOMAC: Mac you can live with. In *Proc. of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 1–13. USENIX Association, 2001.

[13] V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for ssl. In *Proc. of the ACM workshop on Wireless security (WiSE'02)*, pages 87–94. ACM Press, 2002.

[14] B. Hatch. An Overview of LIDS. http://www.securityfocus.com/infocus/1496, November 2003.

[15] L. Johnson, B. Needham, C. Severence, L. Ambuel, and C. Schaufler. POSIX 1.e. In *IEEE Standard 1003.1e*, 1999.

[16] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computations*, 48:203–209, 1987.

[17] V. Miller. Uses of Elliptic Curves in Cryptography. *Advances in Cryptology - Crypto '85*, 218:417–426, 1986.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[19] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. Technical report, IBM Research, 2004.

[20] R. S. Sandhu. Access Control: The Neglected Frontier. In *1st Australian Conference on Information Security and Privacy*, volume 1172 of *Lecture Notes in Computer Science*, pages 219–227, Wollong, Australia, 1996. Springer-Verlag.

[21] R. S. Sandhu, J. E. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[22] R. S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, September 1994.

[23] S. Smalley. Configuring the SELinux Policy. Technical report, NSA, February 2002.

[24] S. Smalley, C. Vance, and W Salamon. Implementing SELinux as a Linux Security Module. Technical report, NAI Labs, May 2002.

[25] R. Spencer, P. Loscocco, S. Smalley, M. Hilbler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. Technical report, Secure Computing Corporation and NSA and University of Utah, 1998.

[26] The DSI Team. DSI: secure carrier-class linux. *Linux J.*, 2002(99):6, 2002.

[27] C. Wright, C Cowan, S Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. of the 11th USENIX Security Symposium*, pages 17–31. USENIX Association, 2002.

[28] H. Xie, P. Biondi, Y. Wilajati Purna, and S. Klein. Linux Intrusion Detection System.

[29] M. Zelem and M. Pikula. ZP Security Framework. Technical report, Faculty of Electrical Engineering and Information Technology Slovak University of Technology in Bratislava, 2000.