Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Porting NSA Security Enhanced Linux to Hand-held devices

Russell Coker
`russell@coker.com.au`
`http://www.coker.com.au/`

Abstract

In the first part of this paper I will describe how I ported SE Linux to User-Mode-Linux and to the ARM CPU. I will focus on providing information that is useful to people who are porting to other platforms as well. In the second part I will describe the changes necessary to applications and security policy to run on small devices. This will be focussed on hand-held devices but can also be used for embedded applications such as router or firewall type devices, and any machine that has limited memory and storage.

1 Introduction

SE Linux offers significant benefits for security. It accomplishes this by adding another layer of security in addition to the default Unix permissions model. This is achieved by firstly assigning a *type* to every file, device, network socket, etc. Then every process has a *domain*, and the level of access permitted to a type is determined by the domain of the process that is attempting the access (in addition to the usual Unix permission checks). Domains may only be changed at process execution time. The domain may automatically be changed when a process is executed based on the type of the executable program file and the domain of the process that is executing it, or a privileged process may specify the new domain for the child process.

In addition to the use of domains and types for access control SE Linux tracks the *identity* of the user (which will be *system_u* for processes that are part of the operating system or the Unix user-name) and the role. Each *identity* will have a list of roles that it is permitted to assume, and each *role* will have a list of domains that it may use. This gives a high level of control over the actions of a user which is tracked through the system. When the user runs SUID or SGID programs the original identity will still be tracked and their privileges in the SE security scheme will not change. This is very different to the standard Unix permissions where after a SUID program runs another SUID program it's impossible to determine who ran the original process. Also of note is the fact that operations that are denied by the security policy [1] have the *identity* of the process in question logged.

I often run SE Linux demonstration machines on the Internet which provide root access to the world and an invitation to try and break the security. [2]

For a detailed description of how SE Linux works I recommend reading the paper Peter Loscocco presented at OLS in 2001 [3].

SE Linux has been shown to provide significant security benefits for little overhead on servers, desktop workstations, and laptops.

However it has not had much use in embedded devices yet.

Some people believe that SE Linux is only needed for server systems. I think that is incorrect, and I believe that in many situations laptops and hand-held devices need more protection than servers. A server will usually have a firewall protecting it, with a small number of running applications which are well maintained and easy to upgrade. Portable computers are often used in hostile environments that servers do not experience, they have no firewalls to protect them, and often they are connected to routers operated by potentially negligent or hostile organizations.

But there are two main factors that cause an increased need for security on portable devices. One is that it is usually extremely difficult and expensive to upgrade them if a new security fix is needed. This means that in commercial use portable computers tend to never have security fixes applied. Another factor is that often the person in posession of a hand-held computer is not authorised to access all the data it contains, and may even be hostile to the owner of the machine.

Naturally for a full security solution for portable computers a strong encryption system will need to be used for all persistent file systems. There are various methods of doing this, but all aspects of such encryption are outside the scope of this project and can be implemented independently.

2 Kernel Porting

The current stable series of SE Linux is based on the 2.4.x kernels and uses the Linux Security Modules (LSM) [4] interface. The current LSM interface has a single sys_security() system call that is used to multiplex all the system calls for all of the security modules. SE Linux uses 52 different system calls through this interface. Due to problems in porting the kernel code to some platforms (particularly those that have a mixed 32 and 64bit memory model) the decision was made to change the LSM interface for kernel 2.6.0. The new interface will make the code fully portable and remove the painful porting work that is currently required. However I needed to have SE Linux working with the 2.4.x kernels so I couldn't wait for kernel 2.6.0.

The main difficulty in porting the code is the system call execve_secure() which is used to specify the security context for the new process. This calls the kernel funtion do_exec() to perform the execution, and do_exec() needs a pointer to the stack, thus requiring architecture specific code in the sys_execve_secure() function. The sys_security_selinux_worker() function (which determines which SE Linux system call is desired and passes the appropriate parameters to it) calls sys_execve_secure() and therefore also needs architecture specific code, and so does the main system call sys_security_selinux().

My first port of SE Linux was to User-Mode Linux [5]. This was a practice effort for the main porting work. It is quite easy to debug kernel code under UML, and as it uses the i386 system call interface I could port the kernel code without any need to port application code.

The main architecture dependent code is in the source file `security/selinux/arch/i386/wrapper.c`, which has code to look on the stack for the contents of particular registers. This needs to be changed for platforms with different register names, and for UML which does not permit such direct access of registers.

The solution in the case of UML was to not have a wrapper function, as the *current* structure had a pointer to the stack anyway that could be used inside the sys_execve_secure() function. So I renamed the sys_security_selinux_worker() function to sys_security_selinux() for the UML port and entirely removed all reference to the wrapper. Then I moved the implementation of sys_execve_secure() into the platform specific directory and implemented a different version for each port.

This was essentially all that was required to complete the port, the core code of SE Linux was all cleanly written and could just be compiled. The only other work involved getting the Makefile's correctly configured, and adding a hook to sys_ptrace().

One thing I did differently with my port to the ARM architecture was that I removed the code to replace the system call entry. When the SE Linux kernel code loads on UML and i386 it replaces the system call with a direct call to the SE Linux code (rather than using the option for LSM to multiplex between different modules). As there is currently no support for having SE Linux be a loadable module there seems to be no benefit in this, and it seems that on ARM there will be more overhead for adding an extra level of indirection for this. So I made the SE Linux patch hard-code the SE system call into the sys-call table.

3 iPaQ Design Constraints

The CompaQ/HP iPaQ [6] computers are small hand-held devices. The most powerful iPaQ machines on sale have a 400MHz ARM based CPU that is of comparable speed to a 300MHz Intel Celeron CPU, with 64M of RAM and 48M of flash storage.

An iPaQ is not designed for memory upgrades. There are some companies that perform such upgrades, but they don't support all models, and this will void your warantee. Therefore you are stuck with a memory limit of 64M.

The flash storage in an iPaQ can only be written a limited number of times, this combined with the small amount of storage makes it impossible to use a swap space for virtual memory unless you purchase a special *sleeve* for using an external hard drive. Attaching an external hard drive such as the IBM/Hitachi *Micro Drive* is expensive and bulky. Therefore if you have a limited budget then storage expansion (for increased file storage or swap space) is not an option.

For storing files, the 32M file system can contain quite a lot. The Familiar distribution is optimised for low overheads (no documentation or man pages) and all programs are optimised for size not speed. Also the JFFS2 [7] file system used by Familiar supports several compression algorithms including the Lempel-Ziv algorithm implemented in zlib, so more than 32M of files can fit in storage.

For a system such as SE Linux to be viable on an iPaQ it has to take up a small portion of the 32M of flash storage and 64M of RAM, and not require any long CPU intensive operations.

Finally the screen of an iPaQ only has a resolution of 240x320 and the default input device is a keyboard displayed on the screen. This makes an iPaQ unsuitable for interactive tasks that involve security contexts as it takes too much typing to enter them and too much screen space to display them. As a strictly end-user device this does not cause any problems.

4 CPU Requirements

Benchmarks that were performed on SE Linux operational overheads in the past show that trivial system calls (reading from /dev/zero and writing to /dev/null) can take up to 33% longer to complete when SE Linux is running, but that the overhead on complex operations such as compiles is so small as to be negligible [8]. The machines that were used for such tests had similar CPU power to a modern iPaQ.

One time consuming operation related to SE Linux installation is compiling the policy (which can take over a minute depending on the size of the policy and the speed of the CPU). This however is not an issue for an iPaQ as the policy takes over a megabyte of permanent storage and 5 megs of temporary file storage, as well as requiring many tools that are not normally installed (make, m4, the SE Linux policy compilation program checkpolicy, etc). The storage requirements make it impractical to compile policy on the iPaQ, and the typical use involves configuration being developed on other machines for deployment on iPaQ. So the time taken to compile the policy database is not relevant.

The only SE Linux operation which can take a lot of time that must be performed on an iPaQ is labeling the file system. The file system must be relabeled when SE Linux is first installed, and after an upgrade. On my iPaQ (H3900 with 400MHz X-Scale CPU) it takes 29.7 seconds of CPU time to label the root file system which contains 2421 files. For an operation that is only performed at installation or upgrade time 29.7 seconds is not going to cause any problems. Also the *setfiles* program that is used to label the file system could be optimised to reduce that time if it was considered to be a problem.

I conclude that for typical use of a handheld machine SE Linux only requires the CPU power of an iPaQ. In fact the CPU use is small enough that even the older iPaQ machines (which had half the CPU power) should deliver more than adequate performance.

5 Kernel Resource Use

To compare the amounts of disk space and memory I compiled three kernels. One was 2.4.19-rmk6-pxa1-hh13 with the default config for the H3900 iPaQ. One was a SE Linux version of the same kernel with the options CONFIG_SECURITY, CONFIG_SECURITY_CAPABILITIES, and CONFIG_SECURITY_SELINUX. Another was the same SE Linux kernel with development mode enabled (which slightly increases the size and memory).

For this project I have no need for the multi-level-security (MLS) functionality of SE Linux or the options for labelled networking and extended socket calls. This optional functionality would increase the kernel size. I am focussing on evaluating the choice of whether or not to use SE Linux for specific applications, once you have decided to use SE Linux you would then need to decide whether the optional functionality provides useful benefits to your use to justify the extra disk space and memory use.

The kernel binaries are 658648 bytes for a non-SE kernel, 704708 bytes for the base SE Linux kernel, and 705560 bytes for the development mode kernel. The difference between the kernel with development mode enabled and the regular one is that the development kernel allows booting without policy loaded, and booting in permissive mode (with the policy decisions not being enforced). For most development work a kernel with development mode enabled will be used, also for this test it allowed me to determine the resource consumption of SE Linux without a policy loaded.

To test the memory use of the different kernels I configured an iPaQ to not load any kernel modules. My test method was to boot the machine, login at the serial console, wait 30 seconds to make sure that all daemons have started, and run *free* to see the amount of memory that is free. This is not entirely accurate as random factors may result in different amounts of memory usage, however this is not as significant on the Familiar distribution due to the use of *devfs* for device nodes and *tmpfs* for /var and /tmp which means that in the normal mode of operation almost nothing is written to the root file system, so two boots will be working on almost the same data.

From the results I looked at the *total* field in the results (which gives the amount of RAM that is available for user processes after the kernel has used memory in the early stages of the boot process), and the *used* field which shows how much of that has been used. The kernel message log gives a break-down of RAM that is used by the kernel for code and data in the early stages of boot, however that is not of relevance to this study only the total amount that is used matters.

The *total* memory available was reported as 63412k for the non-SE kernel, 63308k for the SE Linux kernel, and 63300k for the development mode kernel. So SE Linux takes 104k of kernel memory early in the boot process and 112k if you use the development mode option.

The memory reported as *used* varied slightly with each boot. For the vanilla kernel the value 18256k was reported in two out of four tests, with values of 18252k and 18260k also being reported. I am taking the value 18256k as the working value which I consider accurate to within 8k.

For a standard SE Linux kernel the amount reported as *used* was 19516k in three out of six tests with the values of 19532k, 19520k, and 19524k also being returned. So I consider 19516k as the working value and the accuracy to be within 16k.

For the SE Linux kernel with development mode enabled the memory *used* was 19516k in three out of four tests, and the other test was 19524k. So the difference between the development mode kernel and the regular SE Linux kernel is only 8K of kernel memory in the early stages of the boot process.

Finally I did a test of a development mode kernel with no policy loaded. The purpose of this test was to determine how much memory is used on a SE Linux kernel if the SE Linux code is not loading the policy. For this the memory reported as *used* was 18292k in three out of five tests, with the values of 18296k and 18300k also being returned.

Kernel	memory used
non-SE	18256k
SE no policy	18292k
SE with policy	19516k

So an SE Linux kernel without policy loaded uses approximately 36K more memory after boot than a non-SE kernel in addition to the 104k or 112k used in the early stages of boot.

With a small policy loaded (360 types and 23,386 rules for a policy file that is 583771 bytes in size) the memory used by the kernel is about 1224k for the policy and other SE Linux data structures. The policy could be reduced in size as there are many rules which would only apply to other systems (the sample policy is quite generic and was quickly ported to the iPaQ), although there may be other areas of functionality that are desired which would use any saved space.

So it seems that when using SE Linux the memory cost is 104k when the kernel is loaded, and a further 1260k for SE Linux memory structures and policy when the boot process is complete. The total is 1364k of non-swappable kernel memory out of 64M of total RAM in an iPaQ, this is about 2% of RAM.

All tests were done with GCC 3.2.3, a modified Linux 2.4.19, and an X-scale CPU. Different hardware, kernel version, and GCC version will give different results.

6 Porting Utilities

The main login program used on the Familiar [9] distribution is *gpe-login*, which is an *xdm* type program for a GUI login. This program had to be patched to check a configuration file and the security policy to determine the correct security context for the user and to launch their login shell in that context. The patch for this functionality made the binary take 4556 bytes more disk space in my build (29988 bytes for the non-SE build compared to 34544 bytes for the version with SE Linux support).

The largest porting task was to provide SE Linux support in Busybox [10]. Busybox provides a large number of essential utility programs that are linked into one program. Linking several programs into one reduces disk space consumption by spreading the overhead for process startup and termination code across many programs. On arm it seems that the minimum size of an executable generated by GCC 3.2.3 is 2536 bytes. In the default configuration of Familiar Busybox is used for 115 commonly used utilities, having them in one program means that the 2.5K overhead is only used once not 115 times. So approximately 285K of uncompressed disk space is saved by using busybox if the only saving is from this overhead. The amount of disk space used for initialisation and termination code would probably increase the space used by more than 80% if all the applets were compiled separately (my build of Busybox for the iPaQ is 337028 bytes).

The programs that are of most immediate note in busybox are *ls*, *ps*, *id*, and *login*. *ls* needs the ability to show the security contexts of the files, *ps* needs to show the security contexts of the running processes, and *id* needs to show the context of the current process. Also the */bin/login* applet had to be modified in the same manner as the *gpe-login* program. These changes resulted in the binary being 5600 bytes larger (337028 bytes for a non-SE version and 342628 bytes for the version with SE Linux support.

7 Busybox Wrappers for Domain Transition

In SE Linux different programs run in different security *domains*. A domain change can be brought about by using the *execve_secure()* system call, or it can come from an automatic domain transition. An example of an automatic domain transition is when the *init* process (running in the *init_t* domain) runs */sbin/getty* which has the type *getty_exec_t*, which causes an automatic transition to the domain *getty_t*. Another example is when getty runs */bin/login* which has the type *login_exec_t* and causes an automatic transition to the domain *local_login_t*. This works well for a typical Linux machine where */sbin/getty* and */bin/login* are separate programs.

When using Busybox the getty and login programs will both be sym-links to */bin/busybox* and the type of the file as used for domain transitions will be the type of */bin/busybox*, which is *bin_t*. SE Linux does not perform domain transitions based on the type of the sym-link, and it assignes security types to the Inodes not file names (so a file with multiple hard links will only have one type). This means that we can't have a single Busybox program automatically transitioning into the different domains.

There are several possible solutions to this problem, one possible partial solution would be to have Busybox use *execve_secure()* to run copies of itself in the appropriate domain. Busybox already has similar code for determining when to change UID so that some of the Busybox applets can be effectively SETUID while others aren't. The SETUID management of Busybox requires that it be SETUID root, and involves some risk (any bug in busybox can potentially be exploited to provide root access). Providing a similar mechanism for transitioning between SE Linux security domains would have the same security problems whereby if you crack one of the Busybox applets you could then gain full access to any domain that it could transition to. This does not provide adequate security. Also it would only work for transitions between privileged domains (it would not work for transitions from unprivileged domains). I did not even bother writing a test program for this case as it is not worth considering due to a lack of security and functionality.

A better option is to split the Busybox program into smaller programs so transitions can work in the regular manner. With the current range of applets that would require one program for *getty*, one for *login*, one for *klogd*, one for *syslogd*, one for *mount* and *umount*, one for *insmod*, *rmmod*, and *modprobe*, one for *ifconfig*, one for *hwclock*, one for all the fsck type programs, one for *su*, and one for *ping*. Of course there would also be one final build of busybox with all the utility programs (ls, ps, etc) which run with no special privilege. To test how this would work I compiled Busybox with all the usual options apart from modutils, and I did a separate build with only support for modutils. The non-modutils build was 323236 bytes and the build with only modutils was 37764 bytes. This gave a total of 361000 bytes compared to 342628 bytes for a single image, so an extra 18372 bytes of disk space was required for doing such a split.

Splitting the binary in such a simple fashion would likely cost 18K for each of the eleven extra programs. If we changed the policy to have syslogd and klogd run in the same domain (and thus the same program) and have hwclock run with no special privs (IE the domain that runs it needs to have access to */dev/rtc*) then there would only be nine extra programs for a cost of approximately 162K of disk space. This disk space use could be reduced by further optimisation of some of the applets, for example in the case of *ifconfig* the code to check *argv[0]* to determine the applet name could be removed. A simple split in this manner would also make it more difficult for an attacker to make the program perform unauthorized actions. When a single program has */bin/login* functionality as well as */bin/sh* then there is potential for a buffer overflow in the login code to trigger a jump to the shell code under control of the attacker! When the shell is a separate program that can only be entered through a domain transition it is much more difficult to use an attack on the login program to gain further access to the system.

Finally if we have a single Busybox program that includes applets running in different domains we need to make some significant changes to the policy. The default policy has *assert* rules to prevent compilation of a policy that contains mistakes which may lead to security holes. For the domains *getty_t*, *klogd_t*, and *syslogd_t* there are assertions to prevent them from executing other programs without a domain transition, and to prevent those domains being entered through executing files of types other than the matching executable type (this requires that each of those domains have a separate executable type, IE they are not all the same program). Adding policy which requires removing these assertions weakens the security of the base domains and also makes the policy tree different from the default tree which has been audited by many people.

Another way of doing this which uses less disk space is to have a wrapper program such as the following:

```
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv,
         char **envp) {
  /* ptr is the basename of the
   executable that is being run */
  char *ptr = strrchr(argv[0],
                      '/');
  if(!ptr)
    ptr = argv[0];
  else
    ptr++;

  /* basename must match one of
     the allowed applets,
     otherwise it's a hacking
     attempt and we exit   */
  if(strcmp(ptr, "insmod")
  && strcmp(ptr, "modprobe")
  && strcmp(ptr, "rmmod"))
    return 1;
  return execve("/bin/busybox",
                argv, envp);
}
```

This program takes 2912 bytes of disk space. The idea would be to have a copy of it named */sbin/insmod* with type *insmod_exec_t* which has symlinks */sbin/rmmod* and *modprobe* pointing to it. Then when *insmod*, *rmmod*, or *modprobe* is executed an automatic domain transition to the *insmod_t* domain will take place, and then the Busybox program will be executed in the correct context for that applet.

This option is easy to implement, one advantage is that there is no need to change the Busybox program. The fact that the entire Busybox code base is available in privileged domains is a minor weakness. Implementing this takes about 2900 bytes of disk space for each of the nine domains (or seven domains depending on whether you have separate domains for klogd and syslogd and whether you have a domain for hwclock). It will take less than 33K or 27K of disk space (depending on the number of domains). This saves about 130K over the option of having separate binaries for implementing the functionality.

A final option is to have a single program to act as a wrapper and change domains appropriately. Such a program would run in its own domain with an automatic domain transition rule to allow it to be run from all source somains. Then it would look at its parent domain and the type of the symlink to determine the domain of the child process. For example I want to have *insmod* run in domain *insmod_t* when run from *sysadm_t*. So I have an automatic transition rule to transition from *sysadm_t* to the domain for my wrapper (*bbwrap_t*). Then the wrapper determines that its parent domain is *sysadm_t*, determines that the type of the symlink for its *argv[0]* is *insmod_exec_t* and asks the kernel what domain should be entered when a process in *sysadm_t* executes a program of type *insmod_exec_t*, and the answer is *insmod_t*. So the wrapper then uses the *execve_secure()* system call to execute Busybox in the *insmod_t* domain and tell it to run the insmod applet.

I implemented a prototype program for this. For my prototype I used a configuration file to specify the domain transitions instead of asking the kernel. The resulting program was 6K in size (saving 27K of disk space over the multiple-wrapper method, and 156K of disk space over the separate programs method), although it did require some new SE Linux policy to be written which takes a small amount of disk space and kernel memory.

One problem with this method is that it allows security decisions to be made by an application instead of the kernel. It is preferrable that only the minimum number of applications can make such security decisions. In a typical configuration of SE Linux the only such applications will be *login*, an X login program (in this case *gpe-login*), *cron* (which is not installed in Familiar), and *newrole* (the SE Linux utility for changing the security context which operates in a similar manner to *su*).

The single Busybox wrapper is more of a risk than most of these other programs. The login programs are only executed by the system and can not be run by the user with any elevated privileges which makes them less vulnerable to attack. *Newrole* is well audited and the domains it can transition to are limited by kernel to only include domains that might be used for a login process (dangerous domains such as *login_t* are not permitted).

Due to the risks involved with a single busybox wrapper, and the fact that the benefits of using 6K on disk instead of 33K are very small (and are further reduced by an increase in kernel memory for the larger policy) I conclude that it is a bad idea.

I conclude that the only viable methods of using Busybox on a SE Linux system are having separate wrapper programs for each domain to be entered (taking 33K of extra disk space and requiring minor policy changes), or having entirely separate programs compiled from the Busybox source for each domain (taking approximately 162K of extra disk space with no other problems). Also with some careful optimisation the 162K of overhead could be reduced for the option of splitting the Busybox program. If 162K of disk space can be spared (which should not be a problem with a 32M file system) then splitting Busybox is the right solution.

8 Removed Functionality

A hand-held distribution doesn't require all the features that are needed on bigger machines such as servers, desktop workstations, and laptops. Therefore we can reduce the size of the SE Linux policy and the number of support programs to save disk space and memory.

For a full SE Linux installation there are wrappers for the commands *useradd*, *userdel*, *usermod*, *groupadd*, *groupdel*, *groupmod*, *chfn*, *chsh*, and *vipw*. These can possibly be removed as there is less need for adding, deleting, or modifying users or groups on a hand-held device in the field. These programs would take 27K of disk space if they were included.

A default installation of Familiar does not include support for */etc/shadow*, and therefore there is no need for the wrapper programs for the administrator to modify users' accounts. However I think that the right solution here is to add */etc/shadow* support to Familiar rather than removing functionality from SE Linux. This will slightly increase the size of the login programs.

In a full install of SE Linux there are programs *chsid* and *chcon* to allow changing the security type of files. These are of less importance for a small device. There will be fewer types available, and the effort of typing in long names of security contexts will be unbearable on a touch-screen input device. A hand-held device has to be configured to not require changing the contexts of files, and therefore these programs can be removed.

In the Debian distribution there is support for installing packages on a live server and having the security contexts automatically assigned to the files. As iPaQ's are used in a different environment I believe that there is less need for such upgrades and such support could optionally be removed to save disk space. I have not written the code for this yet, but I estimate it to be about 100K.

The default policy for SE Linux has separate domains for loading policy and for policy compilation. On the iPaQ we can't compile policy due to not having tools such as *m4* and *make*, so we can skip the compilation program and its policy. Also the policy for a special domain for loading new policy is not needed as the system administration domain *sysadm_t* can be used for this purpose. It is possible to even save 3500 bytes of disk space by not including the program to load the policy (a reboot will cause the new policy to take affect).

A server configuration of SE Linux (or a full workstation configuration) includes the *run_init* program to start daemons in the correct security context. On a typical install of Familiar there are only three daemons, a program to manage X logins, a daemon to manage bluetooth connections, and the PCMCIA cardmgr daemon. For restarting these daemons it should be acceptable to reboot the iPaQ, so *run_init* is not needed.

9 Disk Space and RAM Use

In the section on kernel resource usage I determined that the kernel was using 1364K of RAM for SE Linux with a 583771 byte policy comprising 23,386 rules loaded. Since the time that I performed those tests I reduced the policy to 455,422 bytes and 18,141 rules which would reduce the kernel memory use. I did not do any further tests as it is likely that I will add new functionality which uses the memory I have freed. So I can expect that 1.3M of kernel memory is taken by SE Linux.

The SE Linux policy that is loaded by the kernel takes 67K on disk when compressed. The *file_contexts* file (which specifies the security contexts of files for the initial installation and for upgrades) takes 24K. The kernel binary takes 64K more disk space for the SE Linux kernel. So the kernel code and SE Linux configuration data takes 156K of disk space (most of which is compressed data).

The program *setfiles* is needed to apply the *file_contexts* data to the file system. *Setfiles* takes 20K of disk space. The *file_contexts* file could be reduced in size to 1K if necessary to save extra disk space, but in my current implementation it can not be removed entirely. In Familiar a large number of important system directories (such as */var*) on Familiar are on a *ramfs* file system. I am using *setfiles* to label */mnt/ramfs*. So far it has not seemed beneficial to have a small *file_contexts* file for booting the system and an optional larger one for use when installing new packages or upgrading, but this is an option to save 23K. Another option would be to write a separate program that hard-codes the security contexts for the *ramfs*. It would be smaller than setfiles and not require a emphfile_contexts file, thus saving 30K or more of disk space. Currently this has not seemed worth implementing as I am still in a prototype phase, but it would not be a difficult task. Also if such a program was written then the next step would be to use a *jffs2* loop-back mount to label the root file system on a server before installation to the iPaQ (so that *setfiles* never needs to run on the iPaQ.

The patches for the *gpe-login* and *busybox* programs to provide SE Linux login support and modified *ls*, *ps*, and *id* programs cause the binaries to take a total of 10K extra disk space.

Splitting Busybox into separate programs for each domain will take an estimated 162K of disk space.

The total of this is approximately 348K of additional disk space for a minimal installation of SE Linux on an iPaQ. Adding support for */etc/shadow* and other desirable features may increase that to as much as 450K depending on the features chosen. However if you use multiple Busybox wrappers instead of splitting Busybox then the disk space for SE Linux could be reduced to less than 213K. If you then replaced *setfiles* for the system boot labeling of the *ramfs* then it could be reduced to 190K.

10 Conclusion

Security Enhanced Linux on a hand-held device can consume less than 1.3M of RAM and less than 400K of disk space (or less than 200K if you really squeeze things). While the memory use is larger than I had hoped it is within a bearable range, and it could potentially be reduced by changing the kernel code to optimise for reduced memory use. The disk space usage is trivial and I don't think it is a concern.

I believe that the benefits of reducing repair and maintenance problems with hand-held devices that are deployed in the field through better security outweigh the disadvantage of increased memory use for many applications.

All source code and security policy code releated to this article will be on my web site [11].

References

[1] *Configuring the SELinux Policy*. Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/policy2-abs.html`

[2] *Details of SE Linux test machine*, `http://www.coker.com.au/selinux/play.html`

[3] *Meeting Critical Security Objectives with Security-Enhanced Linux*. Peter A. Loscocco, NSA; Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/ottawa01-abs.html`

[4] *Linux Security Modules*, `http://lsm.immunix.org/`

[5] *User-Mode Linux*, `http://sourceforge.net/projects/user-mode-linux/`

[6] *HP Site for iPaQ Information*, `http://whp-sp-orig.extweb.hp.com/country/us/eng/prodserv/handheld.html/`

[7] *Journalled Flash File System 2*, `http://sources.redhat.com/jffs2/`

[8] *Integrating Flexible Support for Security Policies into the Linux Operating System*. Peter A. Loscocco, NSA; Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/freenix01-abs.html`

[9] *Familiar Linux distribution for hand-held devices*, `http://familiar.handhelds.org/`

[10] *Busybox - Swiss Army Knife of Embedded Linux*, `http://busybox.net/`

[11] *My SE Linux Web Pages*, `http://www.coker.com.au/selinux/`