Security Enhanced Virtual Machines
An Introduction and Recipe

Manoj Srivastava

April 6, 2006

Abstract

This paper, and the corresponding workshop, focusses on one of the major problem areas that any organization that is active on the Internet has to solve in order to conduct business in an increasingly hostile environment. The Discretionary Access Controls (DACs) that are the predominant Operating System (OS) techniques in mainstream OS's for managing security make them highly vulnerable to cyber-attacks, since they lack the ability to introduce and enforce strong, system-wide, security policy based, system defenses. This paper details the need for Mandatory Access Control (MAC), the benefits of virtualized server platforms and strong compartmentalization, and walks through a step by step process of implementing such a security architecture on a modern Debian system. This walk through would entail configuring and compiling an virtual machine (the example is an User Mode Linux [UML] image, but the same mechanism can be adopted for Xen VMs as well), creating a base root file system for the UML image to run, and briefly touches on the networking configuration required to connect the virtual machine to the network.

1 Introduction

In this increasingly connected information age, the utility of a computer is negligible unless it is connected to the internet. Also a truism in this age, that any machine connected to the Internet increasingly comes under attack. The trend is a growing spate of remote and local attacks on computer systems.

In face of this increasingly hostile environment, it is difficult for organizations to meet common security goals, including, but not restricted to:

Authentication This ensures that the actors initiating the actions taken by the system are correctly identified, and an assurance that the identity is not false.

Authorization This ensures that the actor is authorized to perform the action under consideration, or has access to resources on the system.

Confidentiality It ensures that the information on a computer system or transmitted over a network is only accessible to authorized entities. This applies to simply revealing the existence of the information, object, or asset.

Integrity It means information (or other assets) can only be modified or deleted by authorized entities, using authorized mechanisms, without bypassing auditing, for example. It is also required that there is some assurance that data has not been tampered with.

Domain Separation For information assurance, it is desirable to compartmentalize information into separate domain, with varying levels of secrecy and security. It is imperative that cross-domain information flow be properly scrubbed, and there be no mechanisms to bypass such scrubbing. Absent these cross-domain pipelines, leakage between domains must be prevented.

Availability It means that the assets are accessible to the authorized parties in a timely manner. Failure to meet this goal results in denial of service.

In this paper we concentrate on AUTHORIZATION, CONFIDENTIALITY, INTEGRITY and DOMAIN SEPARATION. We also touch on some mitigating stances that systems can take to partially counter denial of service attacks.

2 Motivation

Why do we need security and information assurance now more than we ever did before? Because there is an increasing frequency of active attacks based on software bugs, bypassing perimeter security using Trojans, man-in-the-middle attacks, back doors, spy-ware, malware, distributed denial of service attacks and bot nets. Who are the attackers? Some are "script kiddies," while others are spammers and extortionists.

Another trend is for the attackers being increasingly motivated by the profit motif, and the attacks are no longer *hacks* with mischievous intent. Financial, and identity data, is increasingly coming under pressure, and we are only a short distance away from entities being engaged in a network warfare scenario for economic, political, and reasons of sheer spite. In this case, the situation is even more dire, with dedicated teams of opponents engaged specifically to destroy, disrupt, degrade, deny, delay, corrupt or usurp information resident in or transiting through mission critical networks. Have no doubt, this is indeed warfare, and most systems are vulnerable.

A typical scenario involves an attack vector that exploits software flaws in Internet facing services, and subsequently corrupts data residing in the service, or otherwise disrupts nominal operation. For example, suppose the target machine runs SSHD, which has a software flaw. The attackers send a carefully crafted, long string to SSHD, which fails to check length of input stream. The input buffer overflows into the stack. As a result, the attacker gets their code executed by SSHD, which runs privileged in most mainstream operating systems.The machine is now compromised, as is everything that trusts it. Once discovered, such a penetration costs many man-hours to correct. Attacker may, of course, be the authorized user of the machine, thus trying to get unauthorized privilege.

Some of the negative effects of these attacks can easily be mitigated by providing highly granular privilege separation in the service. However, very often this can not be done in the discretionary access control (DAC) model utilized by most current operating systems, since the service often owns the data objects whose integrity needs to be protected.

Another common attack vector results in cases where software flaws in system services are exploited remotely to gain privileges on the target platform often resulting in the attacker taking over the computer system in question. Such an escalation of privileges by the attacker can be hard to prevent in complex pieces of software executing on conventional COTS operating systems. Sand boxing applications and services and protecting them from each other and the underlying system would do a lot to mitigate such attacks.

Yet another family of attacks succeeds by getting an authorized user to run an infected program; and then this "Trojan" has access to any data available to the victim running it, as well as authorization to take any action that the user may legitimately take (like sending email). Given these privileges, the "Trojan" can corrupt data, send it back to the attacker, or erase it. In any case, data integrity and confidentiality are compromised. Lack of fine grained privilege separation leaves the victims open to such attacks. The damage from a Trojan is magnified manifold if it further exploits a local privilege escalation flaw (root exploit) and takes over the machine. In simple cases, the malware tries and infects other computers in the domain, and often these secondary attacks succeed since they are coming from a "trusted" computer in the local domain. In recent years, several worms have exploded over the Internet with wildly exponential infection rates by exploiting just such flawed software that was widely deployed.

In a more extreme but not uncommon scenario, a computer which is taken over may exhibit no immediate symptoms, but may wait for instructions from remote attackers to mount a coordinated attack (sometimes "phoning home" for instructions from the remote attacker) at a particularly critical moment.

The strategies employed by most operating systems to protect against such vulnerabilities (deploying anti-virus scanners and filtering firewalls) are reactive, and are ineffectual in the face of zero day exploits and the gap between an exploit being deployed and the security mechanism being upgraded to detect and disinfect the virus. Furthermore, rapidly mutating viruses present an even greater challenge to reactive, as opposed to pro-active techniques.

Complicating the situation even more in the distributed systems context, remote attacks, including distributed denial of service attacks, are devastating if not intercepted at the perimeter, and can bring application servers to their knees rapidly. Therefor the ability to distinguish and serve legitimate traffic becomes additionally critical for network defense.

In warfare (and modern business), information superiority has long been acknowledged as being critical to success. This implicitly dictates that we should be able to trust the integrity and provenance of the information that we have, and act upon. We must also ensure that the information we have is confidential, and if we operate in multiple security domains, with differing confidentiality requirements, that information flow from the secure to a less secure domain is appropriately scrubbed. This again implies that the information flow through the network must meet processing path guarantees, and must also meet security requirements for information flows, in a manner that can be audited and where we have some assurance that the security mechanisms and scrubbing procedures have not been bypassed.

2.1 Vulnerable programs characteristics

So what kind of programs are the targets of most of these attacks, and thus are high priority assets to defend? All these programs have some common characteristics. These characteristics include:

Privilege changes For example, any `setuid` or `setgid` executable. These programs normally have privileges not available to the user executing them. Any exploitation of weaknesses in the program code can lead to privilege escalation, and may compromise the machine.

Assumption of Atomicity This is specifically exploitable is there is a window between a security check, and performing actions based on that check – for example, checking access permission and opening a file, or deleting a symbolic link – which could have been re-targeted in the interim. In any case, this could lead to bypassing the security check by an attacker properly timing their attack vector.

Trusting the execution environment For example, programs that assume that they are loaded as compiled, or that their plug-ins are to be inherently trusted.

Trusting user input The sshd example above is a classic case of this kind of vulnerability. Programs need to be especially careful when dealing with user input, which could be maliciously formulated, or inadvertently not meet expectations. Even if no unknown users have access, the program is still vulnerable to insider attacks.

Executing mobile code This is becomming common with the growing popularity of interpreted languages, where code is squirted to a server over the network and executed.

Using shared resources This by itself is not a vulnerability, but a compromise of any program accessing shared resources may infect all other users.

3 Solutions

Given the attack vectors and exploitable vulnerabilities detailed above, what are the ways we can counter the threat? Any solution needs to provide as many of the following system characteristics as possible:

Privilege separation This helps contain any compromise of a subsystem from spreading, and moderates attack vectors that target programs performing privilege changes.

Fine grained access control This allows for tighter control of the security of the system without impacting ability of subsystems to perform the required tasks. This also prevents programs run by a user from compromising all the assets accessible to the user, if a proper security policy is in place. Properly deployed, this may obviate the need for privilege changes entirely for some programs.

Role Based access control This allows a single human to wear several hats, and lower the security vulnerability and privilege when working on tasks where it is not required.

Least Privilege No process of user should be given more privileges than they actually need. This can work hand in hand with the finer grained access controls and privilege sets lock down the security of a system. If a program or user does not have a privilege, then it can not be exploited to compromise the machine in the first place.

Limitation of error propagation A compromise of a single application server or subsystem should not lead to the compromise of the machine as a whole.

Resistance to privilege escalation This prevents some of the common exploits immediately – even if a program is compromised.

Assurance Confidence in the completeness and correctness of security policy in use

Protected Paths This provides secure communication guarantees of confidential and unmodified messaging across a mutually authenticated channel

Not be bypassed No security solution is worth anything if an attacker can just bypass the security mechanisms.

Most of these properties require operating system control of program execution, capabilities, and of all system information flow paths to minimize leakage of secrets, prevent insertion of malicious programs, and protect the integrity of system processes.

One of the strongest defenses possible is Security Enhanced Linux from the NSA, with its mandatory access controls, and policy based security (which is added to the discretionary access controls common to UNIX derivatives). It provides all the characteristics of a successful security mechanisms mentioned above. It has no concept of a "superuser", and does not share the shortcomings of traditional UNIX security mechanisms (like depending on coarse grained access control and `setuid` or `setgid` binaries).

Properly written, a mandatory access policy can be used to set up a sandbox for any program (including any and all Internet facing services), such that they can not access resources and information unless expressly permitted. SELinux policies allow for sand-boxing applications to minimize the effect of a successful compromise.

A compromise of a single application server of subsystem may affect data integrity of that application, but does not pose a threat to other subsystems or the system as a whole. Using a static information flow analysis of the security policy, it is feasible to determine what domains would be affected by the compromise of any system, and steps can be taken to either tighten security policies, or to address recovery of the downstream domains in case of a compromise.

3.1 Why Mandatory Access Control?

In view of the failure of conventional discretionary access control mechanisms to provide attestable levels of security for computer systems and networks, mandatory access control (MAC), based on operating system level capabilities, is foundational. In order to provide system security, end systems need to be able to enforce the separation of information based on confidentiality and integrity requirements. This is not possible without active support from the operating system, since otherwise, any security mechanism built on top of operating systems lacking this ability can be bypassed. Without MAC, application security mechanisms are vulnerable to tampering and bypassing, and malicious or flawed applications can easily cause failures in system security. Without MAC, preferably leveraged upon hardware based trusted computing mechanisms, it is impossible to provide the needed data integrity, confidentiality, and domain separation with any level of assurance.

Standard discretionary access controls provided in most operating systems base access decisions on coarse grained user identity and ownership. To ensure a cohesive chain of trust, it must be possible to consider additional security-relevant criteria such as the role of the user, the function and trustworthiness of programs, and the sensitivity or integrity of the data. Under DAC, files are owned by a user and that user has full control over them, including the ability to grant access permissions to other users. The root account has full control over every file on the entire system. An attacker who penetrates an account can do anything with the files owned by that user – and if the user is `root`, has full control over the system. As a corrolary, there is no way to protect against a malicious super user.

Protection against malicious code is not possible using existing DAC mechanisms because every program executed by the user inherits all of the privileges associated with that user. Malicious programs are free to change the permissions associated with all of the user's objects, as well as disclose or alter the objects themselves. This problem is exacerbated by the fact that only two categories of users are supported, completely trusted administrators and completely untrusted ordinary users. Many system services and privileged programs must run with coarse-grained privileges that far exceed their requirements. A flaw in any one of these programs can be exploited to obtain complete system access.

As long as users have complete discretion over objects, it will not be possible to control data flows or enforce a system-wide security policy. Type enforcement, a critical capability provided by MAC enabled operating systems,

allows static analysis and enforcement of data flow, facilitates privilege separation, guards against privilege escalation, and provides all the support mechanisms required to assure data integrity and confidentiality.

Policy based control is yet another critical feature provided by MAC enabled systems. When properly implemented, a detailed security policy enables a system to adequately defend itself and offers critical support for application security by protecting secured applications from being tampered with and being bypassed. It allows critical processing pipelines to be established and guaranteed. A fine grained, tightly configured security policy also enables strong separation of application privileges that permits the safe execution of untrustworthy applications in an isolated, secure sandbox. Its ability to limit the privileges associated with executing processes limits the damage that can result from the exploitation of vulnerabilities in applications and system services.

A MAC security policy, with well formed domain transitions, can also prevent privilege escalations from succeeding. Even a compromised application, overcome by, say, a buffer overflow exploit, would not allow the remote attacker to gain control of the machine, irrespective of whether the exploited application was running as a user process or some privileged system process. MAC enables information to be protected from legitimate users with limited authorization as well as from authorized users who have unwittingly executed malicious applications. Access to sensitive information can be tightly controlled, and tamper proof audit trails can be kept of all access to data. This enforcement of role based access control (RBAC) allows for flexibility without compromising security. The ability for systems to provide capabilities such as these is essential for the design and implementation of secure systems. For example, separating security officer and systems administrator roles makes attacks perpetrated by rogue insiders harder to carry out.

Deploying SELinux results in security policies that are amenable to static analysis, and information flow assurances that provide validation that there are no leaks from one security domain to another. Thus, in the event of a breach in security, the scope of the breach can be readily assessed, and damage limiting mechanisms can be deployed.

Expanding on the above capabilities, SELinux also adds network path protection, where the concept of firewalls is extended to processes on a MAC enabled node. Network access policies will allow a process in a certain security context on one machine to be assured of a connection from another process in a known security context on a remote machine, as opposed to blocking packets based on IP addresses and port tuples as conventional firewall based approaches do. Even with asymmetric security (with MAC based labeling only on one end of the connection), a MAC enabled system can enforce policies based on roles and process domain policies, as opposed to the crude host-to-host policies of a firewall based solution. Once network paths and the packets traveling over them are labeled, it is possible to distinguish legitimate traffic from potentially dangerous traffic, and block it before it reaches application code. This also helps mitigate denial of service attacks, by blocking unauthorized traffic at the perimeter, and not exposing services to such attack vectors.

The role-based access control component defines an extensible set of roles. Each process has an associated role. This ensures that system processes and those used for system administration can be separated from those of ordinary users. The configuration files specify the set of domains that may be entered by each role. Each user role has an initial domain that is associated with the user's login shell. As users execute programs, transitions to other domains may, according to the policy configuration, automatically occur to support changes in privilege.

3.1.1 Non-by-passable processing pipelines

The need to transfer data between security domains over a network is a common requirement today. Firewalls, email gateways, and other edge-of-the-network protection devices are some examples of such cross domain devices. A cross-domain solution (also called guards) that connects different security domains with differing levels of confidentiality and integrity requirements needs a high degree of confidence in its implementation.

A critical requirement of such a solution is that the processing pipeline of filters and scrubbers not be by-passable, and that data be transmitted across the device in a controlled way.

SELinux and MAC policies allow us to define a data flow path which can only happen through the required processing stages.

3.1.2 Eamples of functionality enabled by MAC

With MAC, one can ensure that a mail user agent run by an user only has access to files related to stored mail – and not all files owned by that user. MAC in effect provides each application with a virtual sandbox that only allows the application to perform the tasks it is designed for and explicitly allowed in the security policy to perform. For example, the webserver process may only be able to read web published files and serve them on a specified network

port. An attacker penetrating it will not be able to perform any activities not expressly permitted to the process by the security policy, even if the process is running as the root user. Files are assigned a security context that determines what specific processes can do with them, and the allowable actions are much more finely grained than the standard Unix read/write/execute controls. For example, a web served file would have a context allowing the apache process to read it but not execute or make changes to it, while the log files would be appendable but not readable or otherwise changeable by apache. Network ports are also assigned a context, which can prevent penetrated applications from using ports not permitted to them by security policy. Standard Unix permissions are still present on the system, and will be consulted before the SELinux policy when access attempts are made. If the standard permissions would deny access, access is simply denied and SELinux is not consulted at all. If the standard file permissions would allow access, the SELinux policy is consulted and access is either allowed or denied based on the security contexts of the source process and the targeted object.

3.1.3 In conclusion

Role based access controls, type enforcement, auditable, enforced security policies, strong support for security domains, and a grounds up security policy designed around the principles of privilege separation and least privilege, can move any network operation into a highly defensible security stance. Such a system is pro-active, fail safe, proof against zero-day exploits of program flaws, provides strong separation of security domains, ensures application, and data integrity, ability to limit program privileges, can provide processing pipeline guarantees allows applications to be effectively sand boxed so that a compromised application does not affect other applications and services on the system, and provides an ability to strongly protect audit logs from unauthorized access and from tampering. It can implement authorization limits for legitimate users.

Further, it is undergoing common criteria security evalutaion at various levels, sponsored by various commercial distributions of Linux, so the security offered by SELinux and the reference policies has been certified by domain experts.

The contrast between this approach and the approach of most security products in the anti-virus and intrusion prevention and detection markets could not be more stark. Anti-virus and IDS/IPS systems based on signatures are reactive, operating only on known threats, which is why zero-day exploits are so prized by malware authors. You can compare these products to firewalls with a default “allow any” rule, and many specific “deny” rules. This is a losing battle, as the quantity of malware keeps increasing at an exponential rate and vendors and their customers fight a losing battle to keep up. Any newly discovered security flaw will have a window of vulnerability between the exploit’s release and the signature being added and propagated to the end user.

3.2 Why Virtualization?

Virtualization offers security benefits in its own right; it provides strong data isolation, and can be used to setup multiple services on the same hardware in strict isolation from each other, which helps contain infection. So a compromise of a service on one virtual machine can be quarantined, and the virtual machine can be rebooted from known good data without affecting other services running in other virtual machines.

The best defense against external threats is not to let them in in the first place. The physical separation, or “air gap” defense, has become standard in high assurance computing environments, where separate networks are disconnected from each other. Thus, users needing physical access to multiple security domains must employ a separate CPU and monitor for each domain.

Virtual machines allow the administrator to easily set up multiple security zones on the same hardware, with total domain isolation between all virtual machines and the host machine, and implement different security policies as appropriate for the security zone (think if DMZ and internal servers on the same physical box).

In situations where attestable data separation is important, the ability to show data cannot flow between networks or between one VM to another is an important property – and can be achieved if the host machine also implements mandatory access controls.

Additionally, in conjunction with strong MAC security policies, it allows the administrator to finely tailor security policies for each specific virtual machine, and the services that machine is running.

Virtual machines with copy-on-write virtual file systems can be rolled back to a known good state fairly easily, which is always good if a particular application server was exploited.

3.2.1 Need for small severs and migration

Given the advantages of mandatory access control, and the serious flaws that it mitigates, why has the solution not become more popular and implemented in mainstream operating systems? MAC has been implemented in research operating systems for the best part of a decade, with clear and significant benefits. The stumbling block is that while the security advantages are impressive, the system is notoriously hard to configure, the major obstacle being in the difficulty in implementing a coherent security policy. Though current implementation of modular policy modules promises to reduce complexity and make it easier to incrementally evolve security policy, it is still a high skill task with a steep learning curve to develop policy from scratch. Setting up SELinux is not a task for the faint of heart, and the security polices currently extant are far from complete, making it almost impossible for most folks to convert a working machine to a secure box, and raises the bar for people who just want to casually try out SELinux. Anything to automate this process would help in increasing the security all around. This paper sets out to address these deficiencies.

One possible solution is to utilize virtualization, and instead of trying to convert a full featured, working desktop into a secure platform (quite hard, in advance of Security Enhanced X), and instead create a User Mode Linux virtual server running in strict mode. One of the advantages of running a UML is that we can create a read only root file system, and use copy on write file systems to ensure that any changes can be quickly reverted, even if someone can discover a flaw in the security policy, and exploit it. Also, with UML's, the monitoring mechanisms are out of the ken of the virtual machine, since they can run on the host machine, making it far harder to suborn them.

4 Methodology

In this paper, I am concentrating on a walk through for a UML virtual machine. However, most of the steps taken can be adapted for Xen, with minor changes.

There are a number of problems that novice users face with trying to use a virtual UML instance, firstly, the user-mode-linux package in Debian is showing signs of neglect, and, secondly, is not generally patched to support SELinux. Then there is the issue determining a compatible set of sources, patches, and sources of the patches (though as more and more patches get accepted into the mainstream kernels this is less of a problem now than it used to be).

Even when one has a proper /usr/bin/linux binary, there is the issue of finding a proper root file system to run the UML on. The root file system creation tools in Sid also show signs of neglect, and even then, one would need to install SELinux on these root file systems, which is often a frightening task by itself.

4.1 Compiling the host system

There is little to be done here, except to apply the SKAS patches to the host kernel. These patches allow the UML kernel to run in an entirely different host address space[1] from its processes. This solves the security and honey pot fingerprinting problems by making the UML kernel totally inaccessible to UML processes. Their address spaces are identical to what they would be on the host. A given version of UML guest will look for a specific SKAS patch in the host and fall back to thread tracing mode if it's not there. The boot-up message will tell you which version it's looking for in case you're not sure. The steps are as follows:

1. Download the original kernel sources from kernel.org - selecting the latest version that seems to work well with UML, which is, at the time of writing, 2.6.16.1.

```
% cd /usr/local/src/kernel
% wget ftp://ftp.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.16.1.tar.bz2
% wget ftp://ftp.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.16.1.tar.bz2.
sign
% gpg --verify linux-2.6.16.1.tar.bz2.sign linux-2.6.16.1.tar.bz2
```

2. Untar the sources somewhere (`/usr/local/src/kernel/` is what I use). It is really immaterial where the kernel is unpacked, but I tend to avoid /usr/src since I do not want to be root when compiling the kernels, and so the working directory I use has write permissions for an unprivileged user.

[1] `http://user-mode-linux.sourceforge.net/skas.html`

```
% tar jvvfx linux-2.6.16.1.tar.bz2
```

3. Create a dir for compiling the host kernel (`2.6.16.1, for example`). It is nice to compile the kernel in a separate directory tree from the place it has been unpacked, using a symbolic link farm, since it allows one to apply patches to the build tree while retaining the pristine source tree. Once can then use incremental patches when the next upstream release of the kernel comes out. Additionally, this allows us to build a host and a guest kernel (which needs different patch sets) without having to duplicate all the kernel source tree .

```
% cp -lr linux-2.6.16.1 2.6.16.1
```

4. Get the SKAS host patch. You can find it on the download page[2] for user mode linux. It is also a good idea to check out the download page of the author[3], which has updates. At the time of writing, I got the skas-2.6.16-v9-pre9.patch.bz2 [4] file.

```
% cd ..
% wget http://www.user-mode-linux.org/~blaisorblade/patches/skas3-2.6/
skas-2.6.16-v9-pre9/skas-2.6.16-v9-pre9.patch.bz2
```

5. Apply the SKAS patch.

```
% cd 2.6.16.1
% bzcat ../skas-2.6.12-rc4-v9-pre4.patch.bz2 | patch -p1 dry-run
% bzcat ../skas-2.6.12-rc4-v9-pre4.patch.bz2 | patch -p1
```

6. Configure the host kernel (without ARCH=um, this is a host kernel), enable `/proc/mm` under "Processor type and features" menu if needed, **save the new configuration** and build it.

```
 % make xconfig
% make-kpkg --rootcmd fakeroot kernel-image
```

7. Install the resulting package, tweak your boot loader as needed, and you are good to go.

```
 % dpkg -i../kernel-image-2.6.16.1-skas3-v9-pre9_2.6.16.1_i386.deb
```

If we were trying to compile a Xen virtual machine here, we would be compiling the domain 0 kernel image, which would mean a slightly different `.config` file, and not bothering with the SKAS patch, but not very different.

4.2 A recipe for compiling the UML image

The good news is that the uml patche is now incorporated into the mainline kernel. The mainline SELinux support is also there, for the most part, with occasional patches once in a while. The SELinux kernel patch for 2.6.16 includes a few minor changes pending merge in the next kernel release.

1. First, get the latest SELinux patches from NSA's download page[5].

```
 % wget http://www.nsa.gov/selinux/patches/2.6.16-rc6-selinux1.patch.gz
```

2. Create a dir for compiling the UML kernel (`uml-2.6.16.1`, for example), and populate it with symbolic links.

```
 % cp -la linux-2.6.16.1 uml-2.6.16.1
```

[2] http://user-mode-linux.sourceforge.net/dl-sf.html
[3] http://www.user-mode-linux.org/~blaisorblade/
[4] http://www.user-mode-linux.org/~blaisorblade/patches/skas3-2.6/skas-2.6.16-v9-pre9/skas-2.6.16-v9-pre9.patch.bz2
[5] http://www.nsa.gov/selinux/code/download5.cfm

3. Apply the SELinux patch. Note that there shall be a small failure, for Makefile, since the SELinux patch was against 2.6.16-rc6, and we are actually using 2.6.16.1. This is harmless.

```
 % zcat ../2.6.12-selinux1.patch.gz | patch -p1 --dry-run
% zcat ../2.6.12-selinux1.patch.gz | patch -p1
```

4. Please note that if you configure something as a module, an extra step would be required to install the modules into the UML

   Configure the kernel (don't forget ARCH=um). Since we have patched the host kernel with SKAS, we may turn of the thread tracing mode. Also, hostfs is a nice option to have, unless you are very concerned about security and leaking information from the host system into the UML. Configure the character, block, and network devices to include all the features you shall need in the UML. I strongly suggest using the honey-pot proc pseudo file-system, as well as logging. **DO NOT** turn on SMP and `highmem` support; that is known to be broken for these versions. (Oh, don't forget to turn on SELinux – which also needs AUDIT to be turned on, amongst other things). **Save the new configuration** and build it.

```
 % make ARCH=um xconfig
```

5. Recent versions of *kernel-package* have the functionality to build linux-uml packages natively, so, instead of doing `make ARCH=um linux`, we can build a linux-uml debian package instead.

```
 % make-kpkg --arch=um --rootcmd=fakeroot kernel-image
```

6. This results in a `../kernel-uml-2.6.16.1-selinux1_10Custom_i386.deb`, for example, which can be installed anywhere using `dpkg`.

```
 % dpkg -i ../kernel-uml-2.6.16.1-selinux1_10Custom_i386.deb
```

4.3 Preparing to network the UML's

The networking page[6] at the UML home defines a number of ways to network the resulting UML's. Since the kernels used are way beyond 2.2.X, ethertap was out. Multicast, slip, slirp, and pcap were also out – one should be able to use the UML to provide real world services. That leaves TUN/TAP and the Daemon protocols. The root_fs created below primarily supports the Daemon (since that makes the root_fs more portable, however, tuntap can also be used by just editing `/etc/network/interfaces` on the root_fs and uncommenting the tuntap stanza.

4.3.1 Using TUN/TAP

If you go the tun/tap route, you may either setup a fixed tap device as detailed below, or you may use the `uml_net` command. To do that, make sure that the user that runs the UML is in the group `uml-net`.

```
 # adduser srivasta uml-net
```

Next, make sure `/usr/lib/uml` is in the path for the user who runs the UML

```
 # export PATH=''$PATH:/usr/lib/uml''
```

After this, you just need to specify that the `eth0` interface in the UML needs to use the tun/tap transport, set up `/etc/network/interfaces` inside the UML, and then sit back and let uml_net do the rest (including proxy arp and all). An example is Appendix A on page 13.

```
 # eth0=tuntap,,,<IP of Host Machine>
```

[6] `http://user-mode-linux.sourceforge.net/networking.html`

4.3.2 Using the Daemon transport

If you just want a bunch of UML's to talk to each other, then you are all set – `uml_switch` comes set up out of the box for you. However, if you choose to have the UML communicate to the external world, you need to set up a tap device for the uml_switch network to talk to. All you have to do is add something like this to `/etc/network/interfaces` (I chose to use tap2 since I sometimes use a vpnc client that likes to take up tap0; and *192.168.3.X* is close enough to my internal network that I would have to make minimal changes to firewall rules).

```
 iface tap2 inet static
  address 192.168.3.2
  netmask 255.255.255.0
  tunctl_user uml-net
```

Next, make sure that the tap2 interface comes up, tell uml_switch to listen to tap2, and restart uml_switch.

```
 # ifup tap2
# perl -pli.bak -e \
> 's/^#\s*UML_SWITCH_OPTIONS=.*/UML_SWITCH_OPTIONS=\``-tap tap2\''/'
> /etc/default/uml-utilities
# /etc/init.d/uml-utilities restart
```

Again, you may set up a static network interface inside the UML, or you can set up a DHCP server on the host, and setup your UML to be a DHCP client. The advantage of the latter is that I can then share root file systems across machines, since there is nothing that is site-specific inside the root_fs. An example of the static interface setup is in Appendix A on page 13.

```
 auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
  address 192.168.1.13
  netmask 255.255.255.0
  network 192.168.1.0
  broadcast 255.255.255.255
  gateway 192.168.1.10
```

To set up DHCP, first one needs to setup DHCP; Appendix B on page 13 contains an example you may use to set up dhcp on the host. You may need to edit `/etc/default/dhcp` in order to tell dhcp to listen on tap2, by adding the following line (if you already have an interfaces line, add tap2 to it).

```
 INTERFACES=``tap2''
```

Then just restart dhcpd to have it start listening on the tap2 line:

```
 % /etc/init.d/dhcp restart
```

Then mount the UML root_fs, and change the `/etc/network/interfaces` to the following, and you should be more or less good to go.(In my case, I also had to add tap2 to `/etc/shorewall/interfaces`, and also add tap2 to `/etc/shorewall/masq`, as well as allowing the IP address 192.168.3.X to access my local bind daemon).

```
 auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

At this point, the possibilited expand. You can set up VPN's inside the UML instances, or use `brctl` and create a bridege to a virtual network of UML instances. You can post forward ports on the host system to ports on the virtual machine, and run all network facinf deamons inside the UML.

4.4 Creating a root file system

The first thing we need to be running a UML on a machine is to install the `uml-utilities package`.

```
# aptitude install uml-utilities
```

Though there are already mechanisms in Debian for creating a user mode linux root_fs system (notably, rootstrap[7] they did not work for me. For example, rootstrap fails currently on a 2.6.x host, since rootstrap tries to build the root_fs inside a UML that uses hostfs to mount the current / as the initial file system - and proceeds to fall flat on its face, since libc assumes that it can use the native posix thread library (since the UML kernel version is 2.6.16.1), and `/lib/tls` exists – but UML has not yet ported NPTL over, so things fail.

I decided to write a simple shell script based around debootstrap[8], which also happens to be a simple shell script with very few dependencies, and hence fewer things that can go wrong. This shell script sets up a root_fs, based on a few variables at the top (you may also drop these variables in `/.creatfsrc`), and sets it up with a simple uml_net based networking.

First, select a dir on a file system that has at least a GB of space available, and run the the creatfs.sh script (after suitable modification).

```
% cd /scratch/sandbox
% /path/to/creatfs.sh
```

At this point, you should have a root file system that can bootup, and while it is not yet running SELinux, it is ready to be taken there. There is a script written into `/root/post-install.sh` that needs to be run inside the UML to complete the process.

If you had configured anything in the UML as a module, this is the time to install them. If not, you may skip this step.

```
% cd /scratch/sandbox
% mount -o loop root_fs mounted
% cd /usr/local/src/kernel/uml-2.6.16.1
% make INSTALL_MOD_PATH=/scratch/sandbox/mounted \
>  ARCH=um modules_install
% umount /scratch/sandbox/mounted
```

Russel Coker has created a very useful site[9] that has tweaks required to make a system work properly with SELinux; creatfs.sh tries very hard to incorporate all the relevant tweaks in the root_fs created. You should be able to fire up your UML like so if you are using the TUN/TAP interface and not the persistent tap2 device (just tweak the IP address of the host):

```
% /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

If, on the other hand, you are using the daemon transport, use:

```
% /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

The root file system created in this step would also work with a Xen virtual machine, using a loopback mount. Networking with the Xen instance is different in detail, but conceptually identical.

[7] `http://packages.debian.org/rootstrap`
[8] `http://packages.debian.org/debootstrap`
[9] `http://www.coker.com.au/selinux/tweaks.html`

4.5 Working on SELinux

As mentioned before, please look at the tweaks[10] page and make any changes that `creatfs.sh` may have missed. The next step would be to fire up the UML, and install the SELinux packages. To finish the process, do the following:

1. Fire up the UML. This shall be running in permissive mode, and as yet, there is no policy installed. Expect to see a lot of warnings in this run; but after we reboot the UML, it should be running with no warnings. So, as before, if using TUN/TAP transports, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

   Or else, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

2. Next, login as root, and run the final step inside the UML. Please note that installing selinux-policy-default fails initially, and generates error messages, but the script should clean all that up at the end.

```
 % /bin/bash /root/post-install.sh
```

3. Now, halt the UML

```
 % shutdown -h now
```

4. Fire up the UML a last time. This time around, there should be no warnings as you boot into an SELinux enabled UML. Enjoy.

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=tuntap,,,192.168.1.10
```

   Or else, use:

```
 % /usr/bin/linux mem=256M \
>  con=xterm con0=fd:0,fd:1 con1=xterm devfs=nomount \
>  tty_log_fd=3 3>tty_log_file \
>  eth0=daemon,,unix,/var/run/uml-utilities/uml_switch.ctl
```

5 Conclusion

At this point, you should have a mostly working SELinux virtual machine, barring major bugs in the SELinux packages. It should be easy to byuld up upon the `createfs.sh` script to create specialized root file systems, for isntance, creating a root file system that additionally has postfix installed, or the bind daemon, and mostly automate the creation of small, tightly controlled, server applications.

This is a work in progress, I'll be updating the recipe on the web site `http://www.golden-gryphon.com/software/security/selinux-uml.xhtml` to work with Xen virtual machines.

The area which needs the msot attention at the moment is the referencxe SELinux policy, it has to be tuned for Debian, and we need modules to cater to the huge number of packages that we have but Fedora does not.

[10] `http://www.coker.com.au/selinux/tweaks.html`

A Static network interface

```
auto lo
iface lo inet loopback

# The first network card
# This entry was created during the Debian installation
auto eth0
iface eth0 inet static
    address 192.168.1.13
    netmask 255.255.255.0
    network 192.168.1.0
    broadcast 255.255.255.255
    gateway 192.168.1.10
```

B DHCP configuration file

```
# dhcpd.conf
#
# Sample configuration file for ISC dhcpd
#

# option definitions common to all supported networks...
option domain-name "internal.golden-gryphon.com";
option domain-name-servers glaurung.internal.golden-gryphon.com;
option time-servers 132.236.56.250, 130.203.1.10, 198.82.162.213;

option subnet-mask 255.255.255.0;
default-lease-time 6000;
max-lease-time 72000;

subnet 192.168.1.0 netmask 255.255.255.0 {
  # range dynamic-bootp 204.254.239.33 204.254.239.40;
  range 192.168.1.100 192.168.1.120;
  option broadcast-address 192.168.1.254;
  option subnet-mask 255.255.255.0;
  option routers tiamat.internal.golden-gryphon.com;
}

subnet 192.168.3.0 netmask 255.255.255.0 {
  range 192.168.3.10 192.168.3.63;
  option broadcast-address 192.168.3.254;
  option subnet-mask 255.255.255.0;
  option routers 192.168.3.2;
}

# Fixed IP addresses can also be specified for hosts.   These addresses
# should not also be listed as being available for dynamic assignment.
# Hosts for which fixed IP addresses have been specified can boot using
# BOOTP or DHCP.   Hosts for which no fixed address is specified can only
# be booted with DHCP, unless there is an address range on the subnet
# to which a BOOTP client is connected which has the dynamic-bootp flag
# set.
```

```
group {
  use-host-decl-names on;

  host cinder {
    option ip-forwarding on;
    hardware ethernet 00:01:02:9C:DB:8E;
    fixed-address cinder.internal.golden-gryphon.com;
  }

  host ember {
    hardware ethernet 00:10:A4:E3:F1:9F;
    fixed-address ember.internal.golden-gryphon.com;
    option routers tiamat.internal.golden-gryphon.com;
  }
}
```

References

[AEPO90] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. Generalized framework for access control: An informal description. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 135–143, October 1990.

[Bid05] H. Bidgoli, editor. *Introduction to Multilevel Security*, volume Volume 3 of *Handbook of Information Security*, chapter Threats, Vulnerabilities, Prevention, Detection and Management. John Wiley, 2005. ISBN 0-471-64832-9.

[BP73] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244,, The MITRE Corporation, The MITRE Corporation, Bedford, MA, May 1973. The basis of multi level security.

[BSS+95] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for unix. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77. IEEE, May 1995.

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, October 1992.

[HK00] S. Hallyn and P. Kearns. Domain and type enforcement for linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[LS00] P. A. Loscocco and S. D. Smalley. ntegrating flexible support for security policies into the linux operating system. Technical report, NSA and NAI Labs,, October 2000.

[LS01] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium.*, 2001.

[LSM+98] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference,*, pages 303–314, October 1998.

[Mor04] J. Morris. Networking in nsa security-enhanced linux. *The Linux Journal*, December 2004.

[SSL+99] R. Spencer, S. D. Smalley, P. A. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139. USENIX, August 1999.

[WSB+96] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, California,, 1996. USENIX.