The Ethos Development Manual

W. Michael Petullo and Wenyuan Fei and Jon A. Solworth

May 19, 2017

Contents

Introduction

This manual contains the core documentation for the Ethos project. Ethos developers—both kernel- and user-space—should be familiar with this manual. Its companion, *Go on Ethos: A Tutorial* provides additional material that is critical for successful user-space development. There are many other TEXNotes documenting various pieces of Ethos, but *The Ethos Development Manual* and *Go on Ethos: A Tutorial* provide a necessary foundation.

Chapter 1

Installing Fedora

1.1 Introduction

This chapter describes how you can install Fedora with Xen and a Dom0 Linux kernel on a computer so that the computer can then run Ethos.[1] In particular, this document describes how to set up two types of configurations:

(1) VMware-hosted Xen, Fedora & Ethos and

(2) Xen, Fedora & Ethos on bare metal.

1.2 Obtain Fedora Installation Media

The Fedora network install image, `Fedora-Workstation-netinst-x86_64-24-1.2.iso`, can be found at

`https://download.fedoraproject.org/pub/fedora/linux/releases/24/Workstation/x86_64/iso/Fedora-Workstation-netinst-x86_64-24-1.2.iso`.

Download this image and transfer it to a writable CD or thumb drive.

1.3 *Cheating* with Kickstart

Warning: the kickstart process will overwrite *all* of the data on your disk (or virtual machine's disk) with the Fedora install. If you simply want to get a Dom0 Fedora installation running quickly and are not interested in the details of the process, then you can use a Kickstart script. We have written a Kickstart script that automates the steps enumerated in this document. This script is available at

`https://www.ethos-os.org/ethosInstall/Fedora-24/Fedora-24-x86_64-ethos.ks`.

To use it, boot the Fedora installation media. When prompted by the initial boot menu, press `Tab` to edit the installation boot parameters. Add:

```
ks=https:
//www.ethos-os.org/ethosInstall/Fedora-24/Fedora-24-x86_64-ethos.ks
```

to the list of kernel arguments. If you would like to install 32-bit Fedora, then substitute `i686` for `x86_64`.

1.4 Installing Using a Manual Process

1.4.1 Dom0 Fedora Installation

(1) **Install Fedora using the network install CD-ROM**
Boot from the Fedora installation media. During the installation process, you may choose the minimal install option if you wish to avoid installing the X Window System. Follow the onscreen instructions in order to install Fedora. Reboot the newly installed system.

(2) **Install additional Fedora packages**
As root and while connected to the Internet, execute

```
yum install attr bison ed gcc git glibc-static kernel-devel
  libattr-devel make net-tools nex ntp tar xen xen-devel
```

If you are using an x86_64 system and wish to build the 32-bit version of Ethos, then you must also install two 32-bit libraries: glibc and libgcc.

```
yum install glibc.i686 glibc-devel.i686 glibc-static.i686 libgcc.i686
```

By default, Ethos will build for both x86_32 and x86_64. Unless you explicitly build Ethos for x86_64 only, you will need to install these 32-bit libraries. Likewise, you may use `yum` to install additional packages.

(3) **Install the Ethos kernel and related software from binary packages**
First, add the Ethos Yum repository to your installation. Execute

```
rpm -Uvh
https://www.ethos-os.org/ethosInstall/Fedora-24/noarch/ethos-release-20-1.fc24.noarch.rpm
```

Next, install the Ethos packages using

```
yum install nacl-ethos-compat go-ethos-compat
```

Likewise, install the 32-bit nacl package (assuming you wish to build 32-bit Ethos) using

```
yum install nacl-ethos-compat.i686
```

(4) **Enable the developer's user account to use sudo**

(5) **Ensure the networking and time services are running**

```
systemctl enable network
systemctl start network
systemctl enable ntpd
systemctl start ntpd
```

(6) **Additional steps to complete installation within VMware environment**

(a) Install VMware tools. This will allow you to mount files from the host operating system. VMware tools will warn you that Xen is untested, but it seems to work just fine. You can copy the tar file from another virtual machine.

(b) Configure clock synchronization. Edit the *.vmx and ensure the following lines exist:

```
tools.syncTime = "TRUE"
tools.syncTime.period = 60
```

Note, the first line should already exist.

(c) Make sure that CPU Hotswapping is disabled Edit the *.vmx and ensure the following line exist:

```
vcpu.hotadd = "FALSE"
```

(7) Set the default boot to be Linux, running as Dom0 within Xen

```
grub2-set-default "Fedora, with Xen hypervisor"
grub2-mkconfig > /boot/grub2/grub.cfg
```

(8) **Reboot**

Dom0 Network Configuration

(1) **Configure Xen**
Xen must be configured to use routing-based virtual networking in order to support Ethos. This is because Ethos relies on Dom0's proxy ARP features.

(a) Download the configuration file from `https://www.ethos-os.org/ethosInstall/Fedora-24/xl.conf`, and install it at `/etc/xen/`.

(b) For the ethos network configuration, download `https://www.ethos-os.org/ethosInstall/Fedora-24/vif-ethos`, copy it into `/etc/xen/scripts/`, and make it executable with `chmod 700 /etc/xen/scripts/vif-ethos`.

(c) Ensure that Linux's host firewall will permit network traffic to reach the Ethos kernel by creating the directory `/etc/xen/scripts/vif-post.d/`, placing the file at `https://www.ethos-os.org/ethosInstall/Fedora-24/00-vif-local.hook` in that directory, and permitting execution of this script with `chmod 700 /etc/xen/scripts/vif-post.d/00-vif-local.hook`.

(d) To set kernel parameters and the like, download `https://www.ethos-os.org/ethosInstall/Fedora-24/sysctl.conf` and copy it to over to `/etc/sysctl.conf`.

(e) Finally, reload the system's sysctl parameters:

```
sysctl --system
systemctl restart xend
```

(2) **Configure the associated physical device**
For Ethos to run, your primary network device must have a network configuration. We use `eth0` here as an example.

DHCP To configure your system to assign `eth0`'s network parameters using DHCP, write the following to `/etc/sysconfig/network-scripts/ifcfg-eth0`:

```
DEVICE=eth0
HWADDR=XX:XX:XX:XX:XX:XX
TYPE=Ethernet
ONBOOT=yes
USERCTL=no
BOOTPROTO=dhcp
NM_CONTROLLED=no
```

The value of HWADDR may be found in the output of the `ifconfig` command.

Non-DHCP Static IP If you would like to assign a static IP address without using DHCP, in this case 131.193.36.59, then write a configuration such as the following to the file noted above:

```
DEVICE=eth0
HWADDR=XX:XX:XX:XX:XX:XX
TYPE=Ethernet
ONBOOT=yes
USERCTL=no
BOOTPROTO=none
IPADDR=131.193.36.59
NETMASK=255.255.255.128
GATEWAY=131.193.36.1
DNS1=131.193.36.99
DNS2=131.193.36.101
NM_CONTROLLED=no
```

The above information is what would be used in the ethosLab, except that 131.193.36.59 would be replaced by the host's correct IP address.

(3) Initialize `eth0` using:

```
ifup eth0
```

Chapter 2

Ethos for the impatient

You should read through the rest of this document for a lot of details, but this is the high level overview. We assume that you have done a Fedora build.

2.1 Getting Ethos

Get the source code

```
git clone www.ethos-os.org:home/git/ethos
cd ethos
./bin/gt pullall
```

2.2 Build and test

Build and test everything

```
sudo -E make uninstall   && \
    make test.clean all  && \
    sudo -E make install && \
    ethosTest
```

The last bit of output should look like this

```
TEST PASSED          x86_64   client   logIt
TEST PASSED          x86_64   client   assertStatus
TEST PASSED          x86_64   client   createDirectory
TEST PASSED          x86_64   client   createDirectoryCheckHash
TEST PASSED          x86_64   client   createFile
TEST PASSED          x86_64   client   createExistingDirectory
TEST PASSED          x86_64   client   openDirectory
TEST PASSED          x86_64   client   truncate
```

```
TEST PASSED          x86_64   client   largeWrite
TEST PASSED          x86_64   client   largeRead
TEST PASSED          x86_64   client   removeFile
TEST PASSED          x86_64   client   removeNonExistingFile
TEST PASSED          x86_64   client   removeDirectory
TEST PASSED          x86_64   client   removeNonExistingDirectory
TEST PASSED          x86_64   client   createRemoveDirectory
TEST PASSED          x86_64   client   fileInformation
TEST PASSED          x86_64   client   fileInformationNonExistant
TEST PASSED          x86_64   client   forkChildProcessCreated
TEST PASSED          x86_64   client   forkParentProcessExists
TEST PASSED          x86_64   client   getPendingEvents
TEST PASSED          x86_64   client   getCompletedEvents
TEST PASSED          x86_64   client   getNextName
TEST PASSED          x86_64   client   getPid
TEST PASSED          x86_64   client   getTime
TEST PASSED          x86_64   client   beep
TEST PASSED          x86_64   client   cancel
TEST PASSED          x86_64   client   processExit
TEST PASSED          x86_64   client   copyFd
TEST PASSED          x86_64   client   getProcessGroups
TEST PASSED          x86_64   client   getProcessGroupsChildExited
TEST PASSED          x86_64   client   random
TEST PASSED          x86_64   client   ipc
TEST PASSED          x86_64   client   kill
TEST PASSED          x86_64   client   killProcessGroup
TEST PASSED          x86_64   client   getUser
TEST PASSED          x86_64   client   exec
TEST PASSED          x86_64   client   execRepeat
TEST PASSED          x86_64   client   execNonExistingProgram
TEST PASSED          x86_64   client   args
TEST PASSED          x86_64   client   virtualProcess
TEST PASSED          x86_64   client   virtualProcessExit
TEST PASSED          x86_64   client   virtualProcessDouble
TEST PASSED          x86_64   client   virtualProcessNonExisting
TEST PASSED          x86_64   client   virtualProcessMultipleFds
TEST PASSED          x86_64   client   virtualProcessBadUser
TEST PASSED          x86_64   client   sign
TEST PASSED          x86_64   client   streamingDirectory
TEST PASSED          x86_64   client   naclUserspace
TEST PASSED          x86_64   client   network
TEST PASSED          x86_64   server   network
```

```
TEST PASSED          x86_64   client   networkBadHost
TEST PASSED          x86_64   client   networkSelfDirectoryService
TEST PASSED          x86_64   server   networkSelfDirectoryService
TEST PASSED          x86_64   client   networkFast
TEST PASSED          x86_64   server   networkFast
TEST PASSED          x86_64   client   networkVirtualProcess
TEST PASSED          x86_64   server   networkVirtualProcess
TEST PASSED          x86_64   client   networkDoubleIpcWriteRead
TEST PASSED          x86_64   server   networkDoubleIpcWriteRead
TEST PASSED          x86_64   client   networkDoubleFast
TEST PASSED          x86_64   server   networkDoubleFast
TEST PASSED          x86_64   client   networkDoubleIpcWriteReadVirtualProcess
TEST PASSED          x86_64   server   networkDoubleIpcWriteReadVirtualProcess
TEST PASSED          x86_64   client   networkBigWriteRead
TEST PASSED          x86_64   server   networkBigWriteRead
TEST PASSED          x86_64   client   networkMultipleIpc
TEST PASSED          x86_64   server   networkMultipleIpc
TEST PASSED          x86_64   client   networkMultipleIpcWriteRead
TEST PASSED          x86_64   server   networkMultipleIpcWriteRead
TEST PASSED          x86_64   client   networkRemoteUser
TEST PASSED          x86_64   server   networkRemoteUser
TEST PASSED          x86_64   client   networkMultipleRemoteUser
TEST PASSED          x86_64   server   networkMultipleRemoteUser
TEST PASSED          x86_64   client   networkPeek
TEST PASSED          x86_64   server   networkPeek
TEST PASSED          x86_64   client   networkAndLocal
TEST PASSED          x86_64   server   networkAndLocal
TEST PASSED          x86_64   client   networkLateService
TEST PASSED          x86_64   server   networkLateService
TEST PASSED          x86_64   client   networkConnectToSelf
TEST PASSED          x86_64   client   networkPuzzle
TEST PASSED          x86_64   server   networkPuzzle
TEST PASSED          x86_64   client   networkTwoDirectoryServices
TEST PASSED          x86_64   server   networkTwoDirectoryServices
TEST PASSED                   client   minimaltd
TEST PASSED                   server   minimaltd
TEST PASSED                   client   minimaltdBadHost
TEST PASSED                   server   minimaltdBadHost
TEST PASSED                   client   minimaltdRekey
TEST PASSED                   server   minimaltdRekey
TEST PASSED          x86_64   client   efs
TEST PASSED          x86_64   client   efsAsync
```

```
TEST PASSED          x86_64   client   networkRekey
TEST PASSED          x86_64   server   networkRekey
```

Individual tests sometimes fail. If so, you can re-run an individual test say network as follows:

```
test/scripts/testRerun network
```

Another option is to rerun each of the failing tests until they pass:

```
test/scripts/testRerunFailed
```

2.3 Getting a live instance

You'll need three terminal windows. Since ethos is installed, you can run this outside the Ethos build tree.

2.3.1 First window

In the first type:

```
ethosParams server
cd server
ethosBuilder
sudo ethosRun
```

2.3.2 Second window

In the second type:

```
cd server
minimaltdBuilder
sudo minimaltdRun
```

2.3.3 Third window

In the third type:

```
cd server
et server.ethos
ls
```

You should see a list of files. Note that et is the Ethos Terminal, and that ls is executed on Ethos.

Chapter 3

Building Ethos Binaries

3.1 Introduction

If you followed the instructions in Chapter 1, then you have a working Fedora Dom0 installation running on top of Xen. This chapter describes how to install the latest Ethos software from source. It also provides an overview of the Ethos build system, and it documents the parameters which allow the customization of some aspects of Ethos's build system.

While Ethos is made up of a number of projects and nested source-code repositories, it has a single, non-recursive Makefile. A non-recursive Makefile s superior to recursive Makefiles (i.e., a Makefile for each directory) because it centralizes all dependencies in a single place, so that dependencies are all visible. This simplifies and speeds up the build process. In fact, our old recursive make system was three times slower on a full build; this means that compiling sources in the old system was no more than ⅓ of the execution time, with the rest being make overhead. Our non-recursive Makefile supports partial recompiles too, further speeding things up. Because of these performance gains, we have no intention to support per-repo rebuilds.

3.2 Downloading

You can download the Ethos kernel and its supporting projects in the following manner:

(1) Execute `git clone git.ethos-os.org:/home/git/ethos` to obtain the `ethos` scripts.

(2) Enter the `ethos` directory and execute `./bin/gt pullall` to obtain the source code for each Ethos project.

3.3 Building

To build Ethos, run the following from within the (top-level) `ethos` directory:

```
make
```

After the build is complete, you can install Ethos using:

```
sudo -E make install
```

During the course of a build, the build system places binary outputs in one of three directories:

- `kobjs-$TARGET_ARCH`: kernel objects
- `uobjs-$TARGET_ARCH`: ethos userspace objects
- `lobjs-$TARGET_ARCH`: linux userspace objects

Chapter 8 describes where Ethos ultimately installs files within the host's system directories.

3.3.1 Customizing the build process

The Ethos build process honors a number of shell variables which you can use to customize how things build. For example, `WITH_DEBUG=y make` will produce binaries which include debug symbols. Here we enumerate the available variables and their meaning.

`TARGET_ARCH=`[$x86\_32 \| x86\_64$] The target architecture for the build process. By default, the build system will attempt to build Ethos for the architecture on which the build is taking place (most likely `x86_64` on your machine).

`WITH_DEBUG=`[$y \| n$] Build with debugging support (`-g`) and turn off compiler optimizations. This overrides the Ethos build system default, which uses `-O3`. The difference in optimization means that the code generated by the two cases is not identical, but we chose this because it is often difficult to debug optimized code (e.g., variables may be optimized out and code paths change).

`WITH_PROFILE=`[$y \| n$] Build with profiling support. Default optimization is used (`-O3`). Debug symbols is turned on. This is intended for use with OProfile.

`WITH_GPROF=`[$y \| n$] Build for profiling Linux programs using gprof on Linux.

`WITH_ASSERTS=`[$y \| n$] Build with assertions in place.

`WITH_MEMORY=`[*integer*] Specify the default amount of memory, in megabytes, to be allocated to an Ethos VM. Default is 256.

`WITH_SLEEPTIME=`[*integer*] Specify a period in seconds to wait when running tests. Default is 6.

`WITH_PRIVATE_PATH=`[*path*] Specify the path to the private Ethos repository. Default is `~/ethos/private`.

3.4 Additional make targets

The Ethos build system supports a number of targets. We referenced some of these above:

make pull pulls from the repository (the `./bit/gt pullall` script makes use of this target)

make (all) builds without pulling from the repository

make install installs header files, object files, and binaries in the host's system directories

make build prepares the Ethos tests (see Chapter 4)

make run runs the Ethos tests (see Chapter 4)

make check displays the results from the Ethos tests (see Chapter 4)

make clean removes all compiled object files

make uninstall removes all Ethos files installed by `make install`

3.4.1 Top-level projects

Within the `ethos` directory, Ethos organizes its projects in the following manner:

bin Scripts useful for working on the Ethos project.

dual Contains the C libraries that may be used in the Ethos kernel or userspace (either Linux or Ethos) programs. The libraries contain types and constants, networking and cryptography, storage allocation, and printing.

kernel The non-library C code that makes up the Ethos kernel.

languages The programming languages for user-space applications. This includes `etnTools` which support Ethos' type system both in and outside the kernel and the Go language port to Ethos.

linux Contains the Dom0 utilities that directly support running an Ethos environment. This includes `shadowdaemon` and configuration utilities. It also contains Linux ports of Ethos code, such as `minimaltd`.

mk Build-system-related files.

papers Ethos-related papers.

ports Contains SayI and `goPackages`, the Go packages that can be used on Ethos or Linux.

private Contains our lab-internal configurations (e.g., user accounts) for convenience sake. This eliminates the need to enter such information when building Ethos instances. Once Ethos is ready for production use, we'll configure things in the normal way.

test Test and benchmarking code.

teXnotes Ethos-related documentation.

theses Academic theses related to Ethos.

userspace Contains the C libraries, Go packages, and C and Go programs for use in Ethos user space (e.g., for programming language porting). Contains `el`, the Ethos (shell) language.

Chapter 4

Testing Ethos

4.1 Introduction

This guide describes the Ethos testing framework that you may use test the Ethos kernel and userspace libraries. Generally, you should test Ethos after installing it.

4.2 Running Test cases

You can run Ethos' built-in tests with the following commands:

```
make build && sudo -E make run && make check
```

or more succinctly

```
ethosTest
```

The full test suite take quite a while to run; Its possible to run only parts of the suite. See ethosTest script for the different collections of tests that can be specified.

Occasionally, you'll want to rerun a single test (usually because it has failed). Tests are run for a specific length of time, and sometimes fail because this is insufficient due to issues such as variability in Xen domain startup time. For example, to re-run the test named `logIt`, run the following from the `ethos` directory:

```
./test/scripts/testRerun logIt
```

Another option is to rerun each of the failing tests until they pass:

```
./test/scripts/testRerunFailed
```

4.3 Test Framework Runtime Directories

Within an Ethos root filesystem there are several directories relevant to testing.

/test/programs all test executables

/test/virtualPrograms all test virtual process executables

/test/*testName*/directory any files *testName* operates on

/test/*testName*/logDirectory all log files written by *testName*

/services/*testName* service directory for *testName*

4.4 Test Code Organization

All testcase source files are located in `ethos/test/test`. Each test is present in a directory that contains the following files:

module.mk the build configuration for the test

testProgram.c test source code

testProgram-server.c for network tests, the server-side source code

virtualProcess.c for tests involving virtual processes, the virtual process source code

directories-client.tar contains`directoryInitialize`, `logReference`, and `directoryReference`, directories containing initial files, expected logs, and expected files, respectively.

Chapter 5

The Ethos Private Repo

5.1 Introduction

The Ethos private repo contains private information (not shared between different organizations) used in configuring Ethos. It contains 3 types of information:

Host host's encryption and signature keys, and its directory service record.

User user's password, encryption and signature keys.

Certificates SAyI's certificates which are used to construct groups and authentication.

The private repo is created after the binaries are built but before a complete Ethos file system is built. (The Ethos file system is built from the binaries, the private repos, and the Ethos parameters). Hence the private repo is constucted before Ethos is run using Linux utilities. We assume, therefore, that the user has already built and installed Ethos binaries. Thus, she has run:

```
make && sudo make install
```

The structure of private repo is described in Figure 5.1

5.2 Workflow of adding information to private repo

We assume the commands to be run are all under directory private, the name of the private repo. There are 3 steps when adding information to private repo:

- For each host
  - Create the host
  - Add users on the host
- Create groups for the host which is a directory service.

Figure 5.1: Private repo structure

- Create all the other certificates

For example, if we are going to create a host called client, we will do:

```
private% ethosParams client
private% cd client
```

from the top level of the private repo.

Then, we will create host, users on the host and groups in order All of them are created from the client directory, which we shall use in the running example.

5.2.1 Add A New Host

In order to add a new host, use the utility called `ethosHostAdd`. which reads in necessary information from client—such as IP, port and hostname—and then adds them to the private repo. It also creates the private keys needed by the host. It is called from within the client directory:

```
private/client% ethosHostAdd
```

5.2.2 Add A New User

In order to create each user on client, it needs 3 pieces of information:

- username,
- email and
- password.

The utility to create a user is `ethosUserAdd` which can be run as:

```
ethosUserAdd username email password
```

E.g. :

```
private/client% ethosUserAdd smith smith@client.ethos 1234
```

ethosUserAdd will be called for each user to be added to private repo, and for the above example, user smith will be put under private/host/client.ethos/user/smith.

5.2.3 Adding Groups

For each added host which could be a directory service, we need to create groups for it. There are 2 types of groups: user group and host group. For example, we have a host called server which is already created through the above ethosParams, ethosHostAdd and ethosUserAdd, then we can create a user group for server:

Creating a user group First, specifying the group certificate name:

```
private/client% ethosGroupCreate g1
private/client% cd g1
```

For each user group, there are 3 pieces of information needed for it:

- users in the group.
- directory service if there is any.
- another group if there is any.

The steps to create a group are:

(1) To add a user from a host to a group, we can do:

```
private/client/g1% ethosGroupAddUser username:host
```

(2) To add another directory service to the group, we can do:

```
private/client/g1% ethosGroupAddDirectoryService host
```

(3) To add another sub-group to the current group, we can do:

```
private/client/g1% ethosGroupAddGroup g2@g.server.ethos
```

For a host group, the only difference is step 1, it will add hosts instead of users to the group by doing:

```
private/client/g1% ethosGroupAddHost host
```

Generating Certificates Finally, from within client we can generate certificates, including:

- group certificate
- directory certificate
- user certificate

To generate all certificates signed by client run

```
private/client% ethosGenCertificates
```

Chapter 6

Creating an Ethos instance and Running it

6.1 Introduction

An Ethos instance consists of the Ethos kernel, a number of parameters, a filesystem, the Ethos user-space runtime and programs, and `shadowdaemon`. The Ethos kernel runs as a Xen DomU guest. The purpose of shadowdaemon—which runs within the privileged Linux guest—is to provide services to the Ethos kernel. The shadowdaemon services include a filesystem and random number generator seed. Developing these services to run in shadowdaemon on Linux takes less time than writing kernel code, and it allows Ethos kernel developers to target components that are more interesting from a research point of view. Communication between Ethos and shadowdaemon occurs using the Ethos network stack.

Running Ethos is a matter of:

(1) establishing an Ethos instance by setting Ethos' parameters and creating its filesystem using `ethosParams` and `ethosBuilder`, respectively and

(2) booting the Ethos kernel within a Xen domain and running shadowdaemon using `ethosRun`.

Once Ethos is running, a user can interact with the Ethos instance using `ethosLog`, which displays the logs Ethos produces, and `et`, which provides an interactive terminal connection to Ethos. Both of these commands run on Linux.

6.2 Establishing an Ethos instance

The configuration for two instances, called client and server, are embedded within Ethos' source management tools. (Client and server are have preconfigured unroutable IP addresses and other information; it is possible to create other instances either manually without changing the existing ethosParams or by adding code to ethosParams.)

Creating a client or server instance is very straightforward. One can configure as many client and server instances as desired, but at any time at most one client and at most one server can be running on a single host. Ethos instances exist within the Linux filesystem, so each must be named by a unique path.

In order to succinctly describe which OS and in which directory a command should run, we add two prompts to our command listings:

```
linux directory>
```

(directory denotes the present working directory, generally the instance directories we created above) or

```
ethos>.
```

You can create a stand-alone client instance by running the following commands:

```
linux ~> ethosParams client
linux ~> pushd client
linux client> ethosBuilder
```

This assumes that the private repository exists at `~/ethos/private`. If this is not true, then you can customize `ethosParams` through the use of an environment variable:

```
linux ~> WITH_PRIVATE_PATH=/path/to/ethos/private/ ethosParams client
```

The `ethosParams` script sets up the parameters for the ethos instance. Subsequent utilites use these parameters to both build the filesystem and to specify various runtime properties. The above command places the parameters in the directory `client/param` as a set of files. The name of each file is the parameter name, and the contents of each file is its value. Values are text strings, so they can be easily modified.

The `ethosBuilder` takes as input

(1) the blank file system created during the Ethos build process,

(2) the private repository which contains keying and network information, and

(3) the parameter directory created by `ethosParams`,

and it builds a fully functional Ethos filesystem.

You can remove an instance of Ethos such as `client` by running:

```
linux ~> rm -rf client
```

You might also want to use Ethos in a networked environment. This requires a directory service, which is roughly equivalent to DNS. You can set up the client instance to use a server instance as its directory service as so:

```
linux ~> ethosParams server
linux ~> pushd server
linux server> ethosBuilder
linux server> popd
linux ~> ethosParams client server
linux ~> pushd client
linux client> ethosBuilder
```

Here, specifying `server` as the second parameter to `ethosParams` causes `ethosParams` to set the client instance's directory service configuration using the server instance's parameters. Thus the server instance becomes the directory service for the client. The default client name is `client.ethos`, and the default server name is `server.ethos`.

6.3 Running an Ethos instance

To run an Ethos instance, first boot the Ethos kernel and run shadowdaemon:

```
linux client> sudo ethosRun
```

Log files from the Ethos kernel, shadowdaemon, and Ethos services are created within the Ethos filesystem. To print out these log entries, run:

```
linux ~> ethosLog client
```

It is also possible to start Ethos in ways which support testing (Chapter 4) or debugging (Chapter 7).

To associate a terminal with a running Ethos instance, it is necessary to run `minimaltd` and `et`. We describe this next.

6.4 Minimaltd and MinimaLT programs on Linux

MinimaLT is Ethos's network protocol. MinimaLT is encrypted, authenticated, and fast. In some cases, it is faster than unencrypted TCP/IP. All communication with Ethos occurs over MinimaLT with the exception of shadowdaemon.

Minimaltd (i.e., the MinimaLT dæmon) is the Linux implementation of MinimaLT. (Ethos' MinimaLT implementation is part of the Ethos kernel.) Minimaltd does not require Xen to run. To setup minimaltd, run:

```
linux client> minimaltdBuilder
```

Thus each instance can contain an Ethos filesystem and a minimaltd filesystem. The minimaltd filesystem is a truncated version of the Ethos filesystem, and the default name of the minimaltd client instance is `client.minimaltd.ethos`.

To run minimaltd, execute:

```
linux client> sudo minimaltdRun
```

With minimaltd running on a Linux host, you can run MinimaLT-enabled programs to interact with Ethos hosts or other Linux hosts which themselves run minimaltd. For example, you could run an Ethos terminal to remotely log in to a running Ethos client instance:

```
linux client> et client.ethos
```

This will run the Ethos terminal and establish a MinimaLT connection to the Ethos instance named `client.ethos`.

Another simple Linux MinimaLT utility is ping. As with ICMP ping, our ping checks network connectivity, and it works between any combination of Linux and Ethos pairs. For example, assuming you have a running Ethos client instance and minimaltd, you can run:

```
linux client> minimaltPing client.ethos
```

on Linux to ping Ethos. Conversely, you can run `minimaltPingService` on Linux and from an `et` terminal, run:

```
ethos> ping client.minimaltd.ethos
```

It is possible to run two instances of minimaltd on a single host:

```
linux server> minimaltdBuilder
linux server> sudo minimaltdRun
```

The following commands assume that you have configured and run client and server instances of minimaltd. You must also run the server Ethos instance to act as the directory service. The commands send a ping from the client minimaltd instance to the server minimaltd instance:

```
linux server> minimaltPingServer
linux client> minimaltPing server.minimaltd.ethos
```

If you would like to run the directory service on Linux using the server instance of minimaltd instead of on Ethos, then build the client and server instances using:

```
linux ~> ethosParams server
linux ~> pushd server
linux server> cp param/minimaltdHostName param/directoryServiceName
linux server> minimaltdBuilder
linux server> popd
linux ~> ethosParams client server
linux ~> pushd client
linux client> minimaltdBuilder
```

If you would like to configure client and server instances of minimaltd which run on two different computers, then there are a few more steps. Presently, you build both instances on the client, and then you copy the server instance to the server. When running the following commands, ensure you replace `W.X.Y.Z` with the IP address of Linux computer which will host the *server* minimaltd instance.

```
linux ~> ethosParams server
linux ~> pushd server
linux server> cp param/minimaltdHostName param/directoryServiceName
linux server> echo W.X.Y.Z > param/minimaltdIp
linux server> minimaltdBuilder
linux server> popd
linux ~> ethosParams client server
linux ~> pushd client
linux client> minimaltdBuilder
```

If the server exists behind a NAT device, then you should instead run the following, replacing `W.X.Y.Z` with the *private* IP address of the server and `A.B.C.D` with the *public* IP address of the server:

```
linux ~> ethosParams server
linux ~> pushd server
linux server> cp param/minimaltdHostName param/directoryServiceName
linux server> echo W.X.Y.Z > param/minimaltdIp
linux server> echo A.B.C.D > param/minimaltdIpPublic
linux server> minimaltdBuilder
linux server> popd
linux ~> ethosParams client server
linux ~> pushd client
linux client> minimaltdBuilder
```

You must also configure the NAT device to forward ports to the server as appropriate.

To copy the `server` minimaltd instance directory to the server without losing the files' extended attributes, build an archive using:

```
linux ~> tar czvf server.tar.gz --xattrs server/
```

and finally copy the archive to the server and extract it using:

```
linux ~> tar xzvf server.tar.gz --xattrs
```

6.5 The private repo and new users

The private repo contains information used to describe hosts and users including private keys. This is not secure—private keys would normally be generated on installation and thus not be centralized. But it simplifies our edit-build-run loop used in Ethos development.

To add a new user to the private repo, see 5.2.2.

6.6 Ethos/Minimaltd parameters

The Ethos/Minimaltd parameters are shown in Table 6.1. These are divided into those which are used only for running the VM, and for those which are used both for configuration and at runtime.

6.7 Ethos daemons

When Ethos boots, the first user-space program that the kernel executes is `/init`. While this may be any program, the standard `init` brings up other services. In particular,

(1) it executes the programs listed in `/etc/init/services/` in the background and then

(2) runs the program listed in `/etc/init/console`.

The console program provides a remote shell.

6.8 Configuring Networking

Ethos provides `ethosNetConfig`, a tool that you may use to configure an Ethos host's networking. Networking is provisioned prior to running ethosNetConfig by `ethosParams` and `ethosBuilder`. Thus `ethosNetConfig` fixes the networking for Ethos.

Param	meaning
	runtime setup
debugArgs	debug arguments to ethos, space separated
detached	do not attach to the Ethos console
memory	size of ethos memory in megabytes
nobc	disable console buffering (note: on a crash may lose some console writes)
onCrash	action on Ethos VM crash (e.g., destroy)
pause	start Ethos VM paused ("True" if debugging)
quiet	be terse
kernel	path to kernel (e.g., /var/lib/xen/images/ethos.x86_64.elf)
ramdisk	path to Ethos RAM disk (e.g., /var/lib/xen/images/initialStore.x86_64.tar)
	configuration and runtime values
ethosHostName	Name of ethos VM (e.g., client.ethos)
ethosIp	ethos IP
ethosPort	ethos port
minimaltdHostName	minimaltd host name
minimaltdIp	minimaltd IP
minimaltdPort	minimaltd port
shadowdaemonIp	shadowdaemon IP
shadowdaemonPort	shadowdaemon port
directoryServiceInstance	path of the directory service instance (e.g., client)
directoryServiceName	name of the directory server (e.g., client.ethos)
gdbPort	port on the local host to communicate with gdbsx
user	user owning the ethos filesystem
group	group of user owning the ethos filesystem
ethosPrivateRepo	this is the location of your private repo. If not at ∼/ethos, this environment variable must be set before building ethos instances
privateRepo	path to private repo's config utilities
sbinDir	location of root executables (/usr/sbin/)
sleepTime	time to sleep before killing the ethos VM and shadowdaemon (used only in testing)

Table 6.1: ethos params

Chapter 7

The Instrumentation of Ethos

7.1 Introduction

Gdbsx and OProfile provide debugging and profiling services for Ethos, respectively. Gdbsx interacts with Xen and GDB to provide a kernel debugging environment without special support in a Xen-domain kernel. OProfile supports statistical profiling of an entire system, including Xen, a privileged domain kernel and its userspace, and unprivileged domains. The Fedora kickstart script installs everything that is required for debugging and profiling.

7.2 Installing Special Dom0 Linux Kernel

To profile Ethos, the Dom0 Fedora must install and run a modified Linux kernel to support profiling a DomU. Here we assume you have 64-bit Fedora. Steps are similar for 32-bit.

Install packages `kernel kernel-debuginfo kernel-debuginfo-common` located at

`http://www.ethos-os.org/ethosInstall/Fedora-16/x86_64/`

Package file name is `$PKGNAME-$VERSION.ethos.fc16.x86-64.rpm`. Use command

```
rpm -ivh <Package-URL>
```

Configure grub using command

```
grub2-mkconfig > /etc/grub2.cfg
```

Reboot, in grub menu, select Xen to start and kernel name with “ethos” in middle.

`debug.tunnel`	turn on tunnel-related debug messages
`debug.syscall`	turn on syscall-related debug messages
`debug.event`	turn on event-related debug messages
`debug.fd`	turn on fd-related debug messages
`debug.file`	turn on file-related debug messages
`debug.latch`	turn on latch-related debug messages
`debug.process`	turn on process-related debug messages
`debug.envelope`	turn on envelope-related debug messages
`debug.authenticate`	turn on authentication-related debug messages
`debug.mem`	turn on memory-related debug messages
`debug.types`	turn on types-related debug messages
`debug.minimalt`	turn on minimalt-related debug messages
`debug.connection`	turn on connection-related debug messages
`debug.all`	turn on all debug messages

Table 7.1: Ethos debug flags

7.3 Debugging the Ethos Kernel

Verbose kernel output The `debugArgs` parameter allows you to specify arbitrary strings that are passed to the Ethos boot process. For example, `"debug.file debug.syscall"` will cause Ethos to print debug statements related to its file and syscall subsystems. These print to the kernel log, and can be used with or without gdb.

Ethos supports the debug flags listed in Table 7.1.

Debugging Both debugging and profiling require that Ethos be compiled with debug symbols. Compiling Ethos is described in Chapter 3; during the compilation process, you can direct the build to include debug symbols with:

```
make WITH_DEBUG=y all
```

To run the client instance of Ethos with the `gdb` debugger controlling the Ethos kernel, run:

```
linux client> sudo ethosRun -d
```

This will initialize a Xen domain containing an Ethos kernel controlled by GDB with with a breakpoint inserted at `startKernel`. Upon booting, the Ethos kernel will immediately hit the `startKernel` breakpoint. From this point on, you can manipulate the Ethos kernel using GDB as you would any other program. There are a few exceptions to this; for example, you cannot safely interrupt the kernel using `Ctrl-C`. Instead, pause the kernel from outside the debugger using `xl`.

The `ethosRun` script performs the following steps to enable debugging. Occasionally, it might be necessary to execute these procedures by hand. To debug the kernel running within the client instance of Ethos, perform the following steps:

- Start Ethos so that the domain is paused. First, write `True` to the Ethos instance's `param/pause` file. Next, run Ethos with `linux clinet> sudo ethosStart` and shadowdæmon with `linux client> sudo shadowdaemon -l 11112 -r 11112 -d rootfs`.
- Identify Ethos' domain ID using `xl list`.
- Run Gdbsx with `sudo gdbsx -a` ID $WORDSIZE$ `9999`, where `ID` is Ethos' domain ID and `WORDSIZE` is 32 or 64.
- Run `gdb ethos-`$NAME$`-`$ARCH$`.elf` and, at the GDB prompt, `target remote localhost:9999`. `ethos-`$ARCH$`.elf` can be found in your source directory or installed at `/var/lib/xen/images/`.

7.4 Profiling the Ethos Kernel

The Ethos test framework has support for profiling. To profile a test, run the test with `make PROFILE=y`. The test framework writes the results of the profile to `profile.log`.

As with debugging, it will sometimes be necessary to execute the profiling procedure by hand. To do this, run:

- Start Ethos so that the domain is paused. To do this, start Ethos as described in Chapter 6.1, but run `ethosStart` using its pause option.
- Run `opcontrol --reset`.
- Run:

  ```
  opcontrol --start-daemon --xen=/boot/xen-syms... \
  --vmlinux=/usr/lib/debug/lib/modules/2.6.38-1.xendom0.fc14.x86\_64/vmlinux \
  --passive-domains=ID1,ID2,... --passive-images=ethos.elf,ethos.elf,...}
  ```

  with the appropriate domain ID's and Ethos kernel images.
- Run `opcontrol --start`.
- Unpause Ethos using `xl unpause ethos` and run your experiment.
- Run `opcontrol --stop`
- Run `opcontrol --shutdown`
- Run `opreport -l` and observe the printed report.

Chapter 8

The Ethos Hierarchy Standard

8.1 EHS and FHS

This chapter describes the Ethos Hierarchy Standard, which specifies an Ethos filesystem layout. The Dom0 components of EHS nest within the (Linux) Filesystem Hierarchy Standard in order to be non-invasive with respect to existing Linux filesystem layouts. This will facilitate the packaging of Ethos and should increase its appeal to upstream Linux distributions. The FHS does not cover cross-compiling toolchains; for these, EHS adopts the de facto standard used by GNU and Fedora[1]. The integration of EHS into FHS is shown in Figure 8.1.

In the directory hierarchies, blue nodes are for directories with fixed names, green nodes are for collections of files, bright red is for files, and dark red is for continuation to other figures.

8.2 Introduction to EHS

In Ethos, each directory contains only a single type of file; this property is enforced by the operating system. Moreover, all files in a directory have the same permissions. Finally, some directories have implicit permissions associated with them. The top-level directory hierarchy found in Ethos is shown in Figure 8.2.

8.3 EHS: /etc

The /etc directory contains configuration information. A fresh Ethos install will contain the following directories:

mac Of type EthernetMac; meShadowDaemon and meNetwork are required by the kernel

[1]For example, `http://fedoraproject.org/wiki/Packaging/MinGW`

Figure 8.1: Dom0 portion of EHS, based on FHS

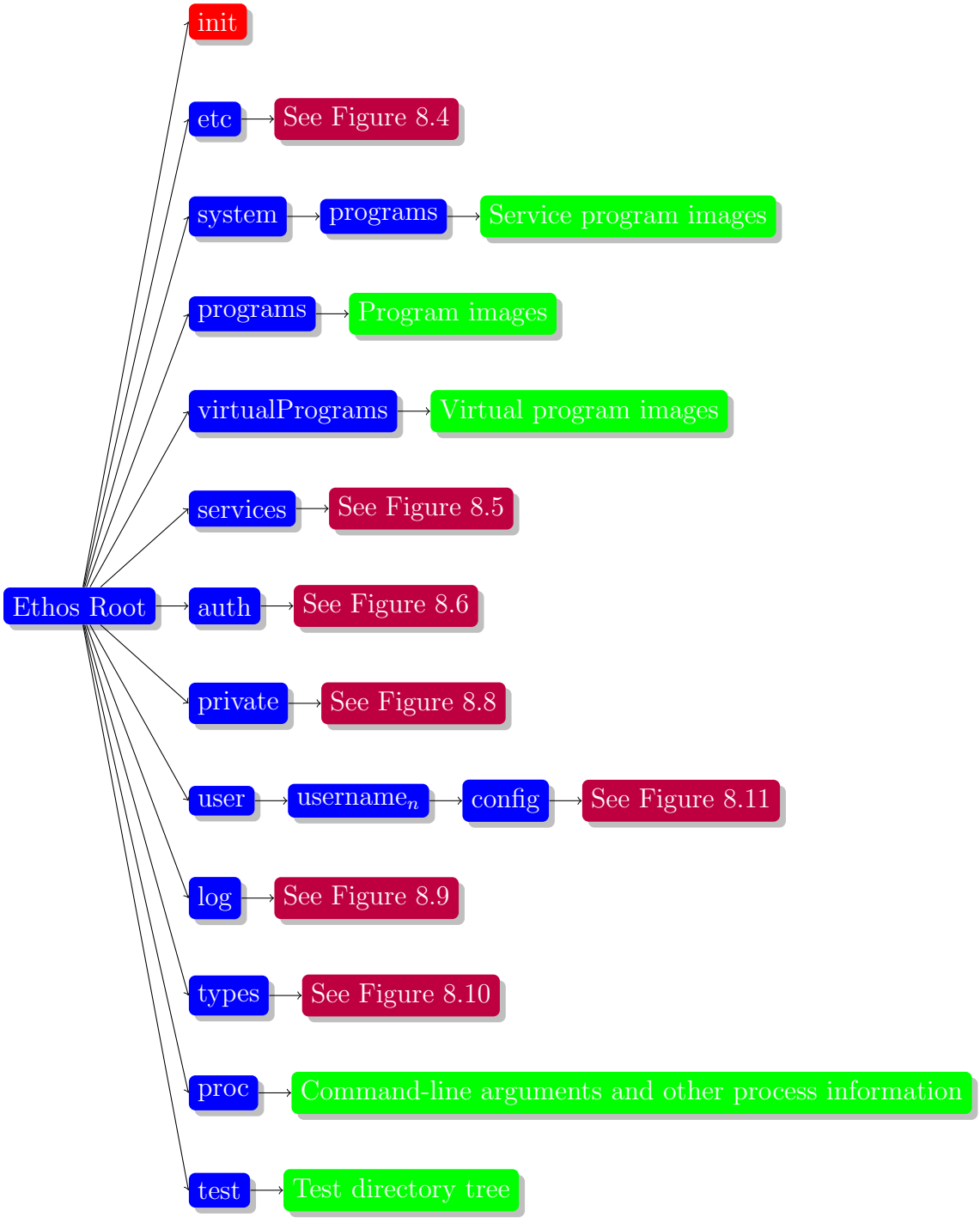

Figure 8.2: Ethos portion of EHS (root)

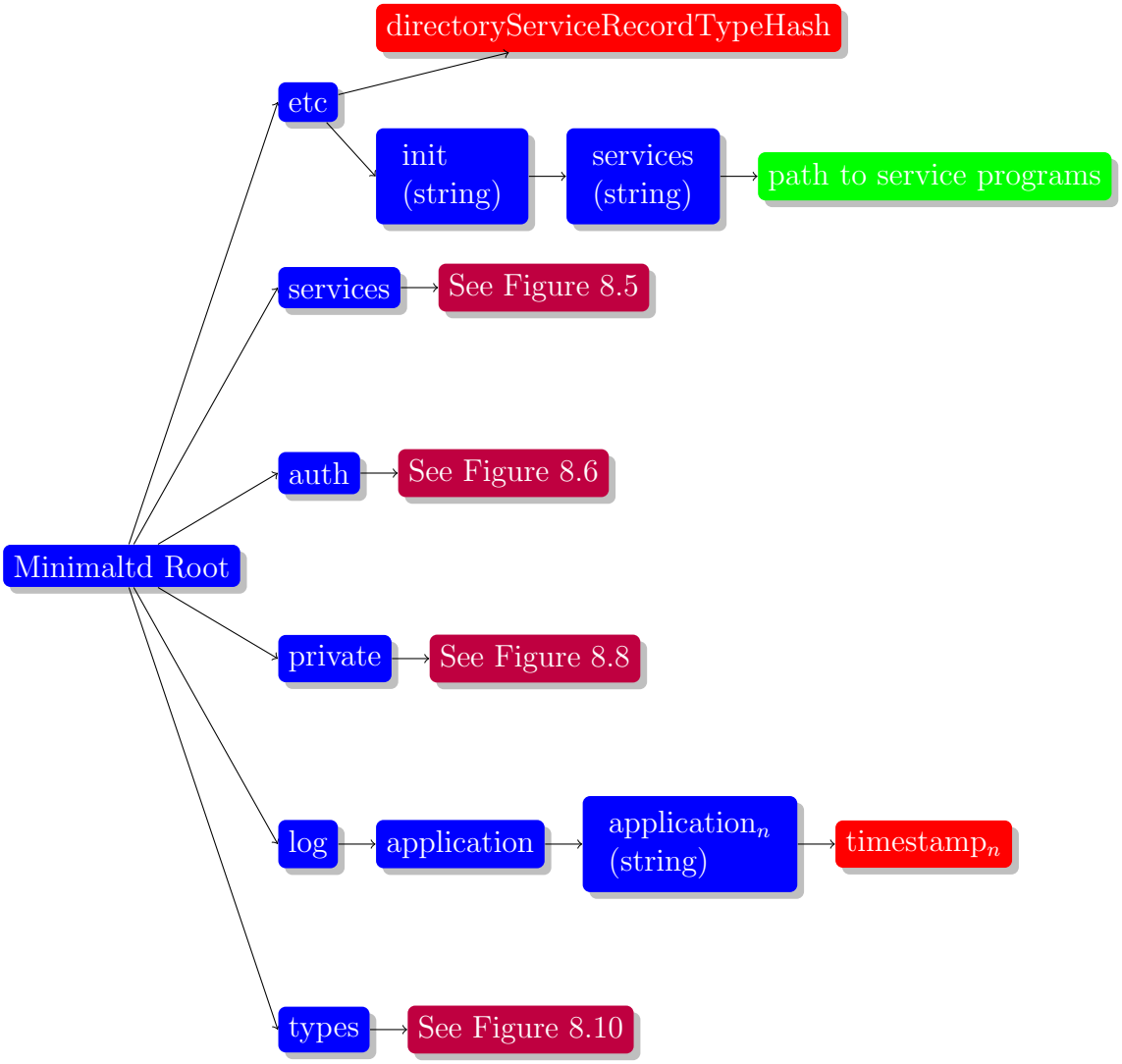

Figure 8.3: Minimaltd File System

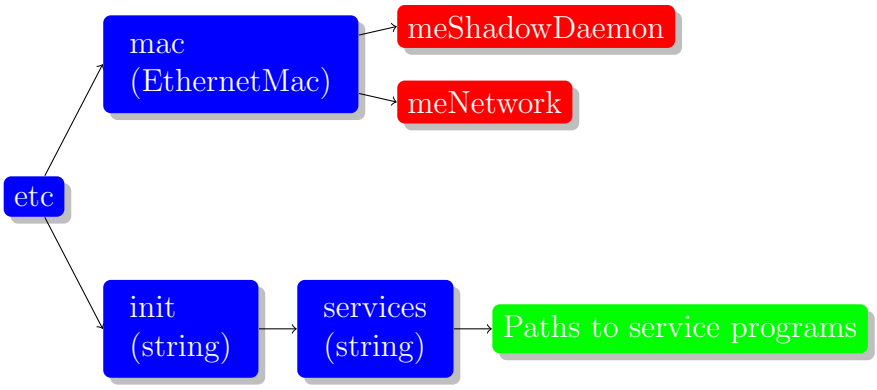

Figure 8.4: Ethos portion of EHS (/etc)

init the files in the services directory contain paths to programs which init will run while booting the system

The /etc directory is shown in Figure 8.4.

8.4 EHS: /services

The /services directory contains directories which will be associated with services using advertise/import/ipc. Each directory should be associated with an ETN interface.

The /services directory is shown in Figure 8.5.

Figure 8.5: Ethos portion of EHS (/services)

8.5 EHS: /auth

The /auth directory is shown in Figures 8.6 and 8.7. This directory contains the public information which describes the users and hosts known to an Ethos instance. This information, in turn, makes its possible to create and authenticate network connections. The information describing hosts are contained in Directory Service Records (DSRs); the information describing users are contained in user or user-remote records.

Creating a network connection to a remote host h requires a DSR for h. The /auth directory contains a DSR for the Ethos instance itself, as well as DSRs for the instance's directory service and shadowdæmon. These DSRs initially contain only each host's IP addres,

port, hostname and long-term public encryption key. If the directory-service DSR is present, then the Ethos instance uses this directory service to resolve names; otherwise, the instance uses a local directory service. If an Ethos host runs a directory service, then the directory service will answer requests for host information which is present in the form of DSRs in /auth/host.

Ethos authenticates all of its network connections, and hence the /auth directory also contains certificates for users and hosts. An Ethos instance maintains its authentication-related information in the following directories within /auth:

user Each local user is named by a username and described by a user record in /auth/user. User records contain a

- username,
- public encryption key,
- public signature key,
- email address, and
- aspect.

userShortName and userRemote Each user also has key-to-name mapping in /auth/userShortName and an identity in /auth/userRemote. Identities contain a

- long-term encryption key and
- a local (host-specific) username.

host Each host is named by a hostname and described by a DSR in /auth/host. A DSR contains a

- permanent public encryption key,
- ephemeral public encryption key and expiration time,
- subsequent ephemeral public encryption key and expiration time,
- IP address, and
- port.

hostSigningKey The /auth/hostSigningKey contains a public signature key for each remote host.

8.6 EHS: /private

The /private directory contains the credentials for a host itself, users and directory service. It is shown in Figure 8.8.

8.7 EHS: /user/username$_n$/config

The home directory of user$_n$ is /user/username. The /user/username$_n$/config directory contains various user-specific configurations, including:

key Of type string; keys used for connecting to services; each file contains a key name (that matches a key filename in /auth), and the absence of a configuration for a given service indicates that the connection should be anonymous

The /user/username$_n$/config directory is shown in Figure 8.11.

8.8 Building Authentication and Network information

Currently, when building Ethos, network configuration is not done at this phase. Regarding the authentication information, a few things are done at this phase:

- Host keys (long term, signing) are copied over from private repo to rootfs/priviate/host.
- All hosts signature keys are copied over from private repo to rootfs/auth/hostSigningKey
- All users' keys (long term, signing) are copied over from private repo to rootfs/private/user
- All users' public information (UserRecord and UserRemote) are copied over from private repo to rootfs/auth/user and rootfs/auth/userRemote

So far **NO** directory service records have been generated for the host because of the missing network information. After we do make install, and get ready for testing. Network information will be configured before running tests. There are 2 places that network information is configured.

(1) In test/mk/, there are a few Makefiles used for different network scenarios, and they contains a few command lines to generate directory service record for the host and shadowdaemon.

(2) Another place is in script ethosNetup. It checks if host its own directory service record exists or not. If not, then it will generate the record for the host itself and shadowdaemon.

The utility we are using to generate directory service record is called ethosBuildDSR, and for example it can be called in the following way to generate a record for host itself:

```
ethosBuildDSR -output=$(ROOTFS)/auth/me -hostname="ethos.example.com"
-ip=169.254.1.1 -port=11111 -encryptionKeyPairPath=$(ROOTFS)/private/host/longTerm/encry
```

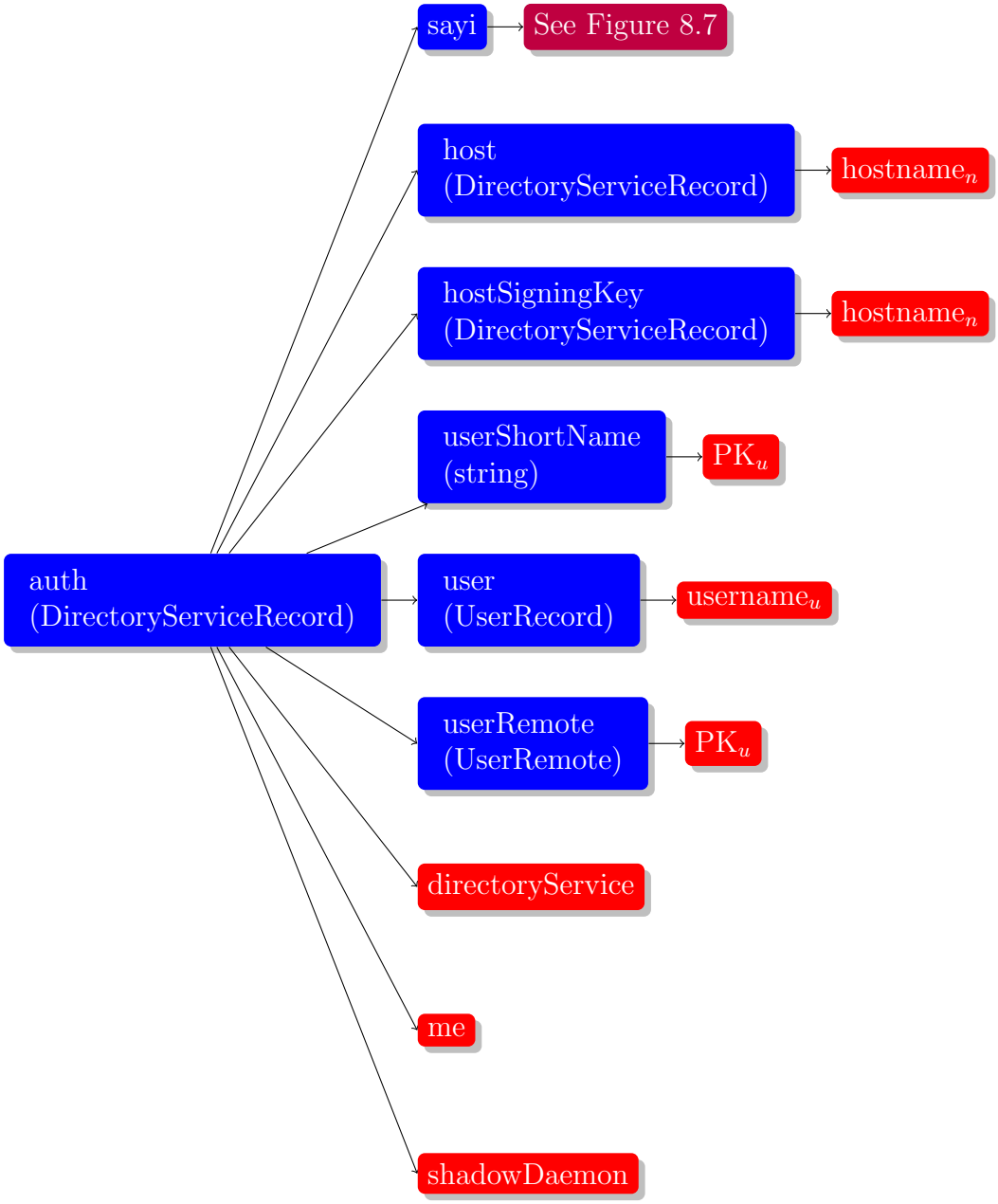

Figure 8.6: Ethos portion of EHS (/auth)

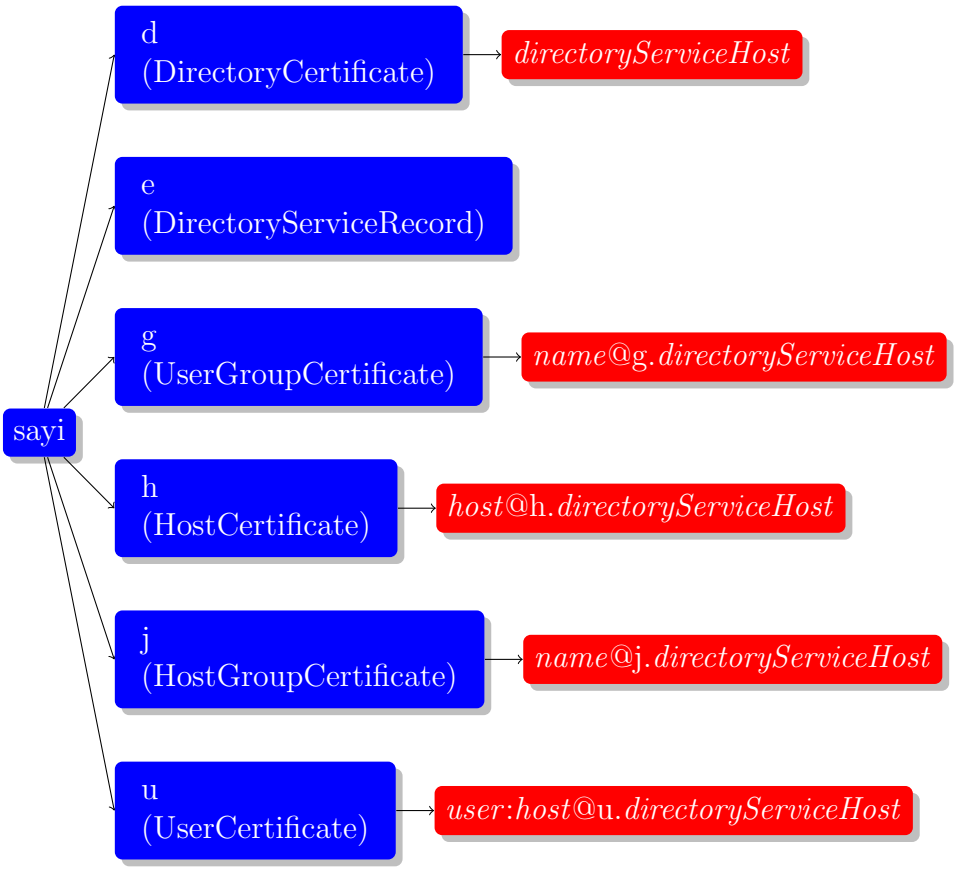

Figure 8.7: Ethos portion of EHS (/auth/sayi)

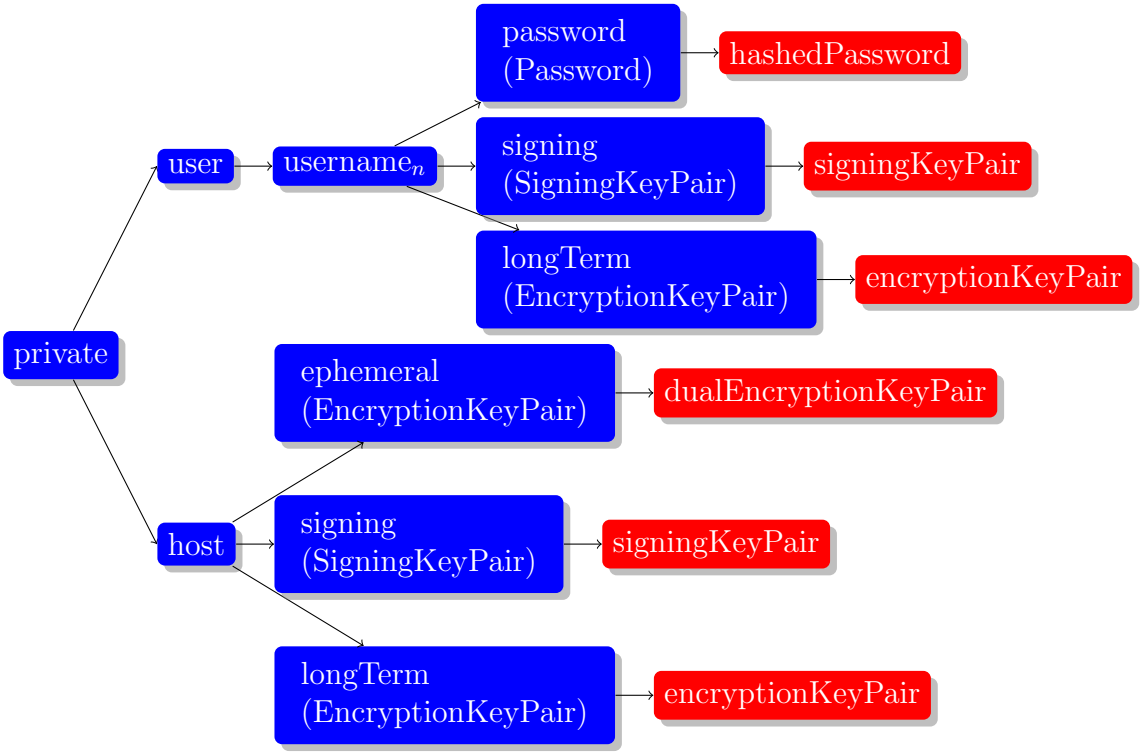

Figure 8.8: Ethos portion of EHS (/private)

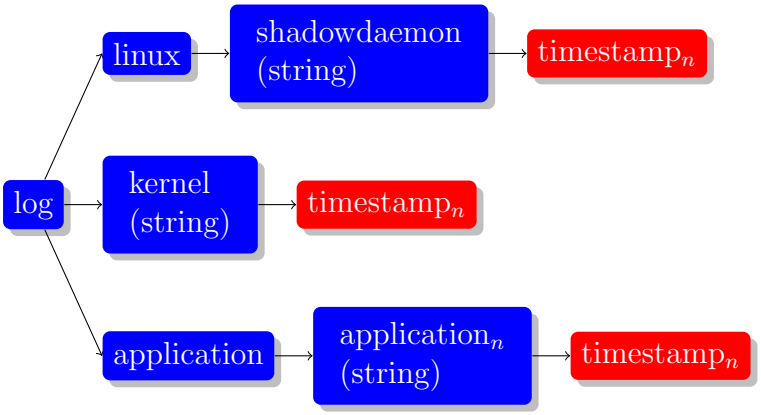

Figure 8.9: Ethos portion of EHS (/log)

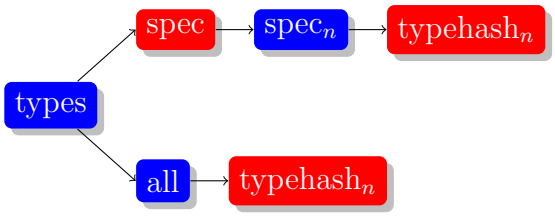

Figure 8.10: Ethos portion of EHS (/types)

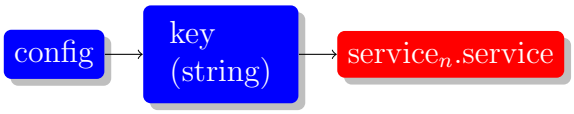

Figure 8.11: Ethos portion of EHS (/user/username$_n$/config)

Chapter 9

Development Practices within the Ethos Project

9.1 Introduction

This describes the high-level development practices that will be used within the Ethos project.

9.2 Documentation

9.2.1 Conceptual Documentation

Conceptual documentation should be written as a TEXNote.

Certain documentation should be included as a chapter in one or more of various manuals, including

manualKernelDevelopers How to write Ethos kernel code

manualUserspaceDevelopers How to write Ethos applications

If you intend that your TEXNote is to be included in a manual, then you must

(1) Name your TEXNote `section-`*`someSectionName`*`.tex`

   (a) Produce your section as a `chapter`

(2) Provide a driver, `paper.tex`, that allows your TEXNote to be build individually

   (a) Use the `report` documentclass

(3) Name labels with the standard *`type`*`:`*`someSectionName-description`* `where` *`type`* `is one of sec, fig, etc.`

9.2.2 Interface Documentation

Documentation of system call and library interfaces should be done using Doxygen and placed in header files. The target audience of this documentation are the programmers that will use a given module's interface.

Doxygen itself is documented at `http://www.stack.nl/~dimitri/doxygen/manual.html`. Doxygen allows a developer to document an interface inline with its code and then extract the documentation into HTML, PDF or other types of presentation formats. For example, consider the following code, found in Ethos' core library header file.

A source file should begin with a comment that records its general purpose and authors:

```
///
/// @file
///
/// \brief Standard library
///
/// A series of standard library routines.
/// June 2008, David Thulson
```

Next, each function has its own documentation:

```
/// \brief compares the first n bytes of the memory areas cs and ct
///
/// @param cs a pointer to a block of memory
/// @param ct a pointer to a block of memory
/// @param count the length of memory to compare
///
/// @return an integer less than, equal to, or greater than zero if the
/// first n bytes of cs is found, respectively, to be less than, to
/// match, or be greater than the first n bytes of ct
int memcmp(const void *cs, const void *ct, size_t count);
```

Additional documentation includes:

- Careful documentation of data structures
- Restrictions on the calling of functions and procedures
- Implicit documentation through naming and typedefs

9.2.3 Code Documentation

Documentation of code should be done in source code files. The purpose of this documentation is to assist in the maintenance of source code. The target audience of this documentation are the programmers that will maintain a module implementation in the future. Code documentation stands on its own; it is not processed by Doxygen.

A source file should begin with a comment that records its general purpose and authors:

```
//---------------------------------------------------------------------
// Reference monitor for Ethos authentication.
// Please refer to the corresponding header file for interface
// documentation.
// May 2011, W. Michael Petullo
//---------------------------------------------------------------------
//
```

The remaining code documentation is inserted inline. The comments should provide documentation in a similar manner to interface documentation, but should focus on internal code that is not directly outside of a given module.

9.2.4 Makefiles

This section was written by Bennett Clarke and editted by Jon Solworth.

The requirements for the Ethos build system are quite complex. Consider that we must compile libraries and programs for

- multiple architectures (e.g., Intel IA-32 and AMD64)
- multiple languages (C, Go, El, and shell) and
- multiple operating systems (i.e., Linux and Ethos).
- user space and kernel space

Some libraries are appropriate for only one operating system and others should be compiled for both. On Linux, we install programs at /usr/bin; on Ethos, /programs. Finally, we require that certain compile-time features are optional (e.g., debugging symbols, profiling code, etc.).

The most recent version of the Ethos code is held in a set of Git repositories from which the source code is downloaded. The variable GITHOST defines the host for the git repos and is defined as git.ethos−os.org.

The Ethos operating system is built and installed from Linux using make.

(1) download the most recent version of the Ethos Source Code Tree from the Git repository.

(2) build the source by invoking make. Make will take the source code in the source code tree and compile its various files into object code and link that with the several designated libraries. Make will create new files and directories as necessary to store the object code.

(3) make will install an executable image of the Ethos Operating System at the root level.

Make is invoked on a description file, named the makefile. The Ethos Operating System is created dynamically from files distributed throughout the Ethos Source Tree.

Non-recursive make

There are two ways to use make for large projects recursive makefiles or non-recursive makefiles. Ethos used to use recursive make, in which make at the root invokes make in sub-directories recursively. The problem with recursive make is that it fundamentally weakens make semantics. This is because makefiles describes dependencies between different build objects, but recursive make loses these dependencies across make invocations.

Thus Ethos' non-recursive makefiles fundamentally automated more of the build process and are less prone to breaking as Ethos changes. In addition, when we converted from recursive to non-recursive make we decreased build time by a factor of 3.

ETHOSROOTDIR specifies the root of the Ethos build tree. It contains four different files:

(1) *Makefile*,

(2) *top.mk*,

(3) *bottom.mk*, and

(4) *install.mk*.

Because we are using non-recursive make these are for the whole Ethos projects. (Sub repos may contain Makefiles, but these are not invoked in an Ethos build.) Make is invoked from ETHOSROOTDIR and all paths are relative to ETHOSROOTDIR.

To make things a bit more modular, each sub-repo contains a module.mk file, which is included in the top-level Makefile. These module.mk contain the rules to make objects and executable within that repo.

In general, source artifacts are built in the directories pulled from the repo. Such source artifacts include code generated by ETN and scripts. Binary artifacts are built in one of the three environments and two architectures. The binary directories parallel the source directories.

Top-level Ethos Build directory

The top-level Makefile defines ETHOSROOTDIR

```
ETHOSROOTDIR := $(abspath .)
```

The Makefile in ETHOSROOTDIR is the master description file for the make application. It contains a series of directives and definitions in addition to one or more sets, each set consisting of a dependency ("rule") followed by its relevant commands. This is the only Makefile instance in the Ethos source tree and it incorporates the .mk files that occur in the Ethos source tree.

The Makefile uses the include directive to incorporate the .mk files found throughout the Ethos source tree. The order of inclusion are top.mk, install .mk, module.mk, and bottom.mk. The resulting object is a single Makefile. Since all names are global in a Makefile, care has to be taken with naming to avoid conflicts.

- top.mk includes definitions and any sets consisting of a dependency followed by its relevant commds.
- install .mk includes definitions for installing linux executables, the Ethos kernel, and a generic Ethos file system.
- bottom.mk is included at the bottom of the sets of dependency-commands that constitute the whole makefile. The file bottom.mk consists of definitions and sets of dependencies-rules that are not definable until after the module.mks have been included.

The file module.mk, if present in a directory, provides definitions and sets of dependency-commands applicable to the subdirectories in that file. In general, each repo defines a module.mk, and each node in the tree of repos includes the module.mk for each of its subchildren.

Hierarchical Makefile

The Makefile defines BUILD_MODULES to be those immediate subdirectories that are operative parts of the Ethos source code. Each subdirectory's module.mk file is included in the Makefile by the directive

```
-include $(patsubst %, %.all , $(BUILD_MODULES))
```

In turn, each subdirectory's module.mk defines that subdirectory's immediate subdirectories and incorporate their module.mks.

The BUILD_MODULES are defined to include several subdirectories organized by the prime use of the files. The modules kernel, linux, userspace, and test are self explanatory. The dual module consists of C source code that are used by both kernel space and user space. The private module is used to hold source code files used for specific configurations. The languages module consists of Go language code, and the code used to manage the Ethos type system and to serialize and deserialize variables when they are sent over a network connection. The BUILD_MODULES are defined as follows:

```
BUILD_MODULES := dual kernel languages linux userspace private test
```

There is also a more comprehensive definition of the Ethos subdirectories which is simply called MODULES. MODULES includes all elements of BUILD_MODULES and incorporates the subdirectories teXnotes and papers.

There are certain rules for prepending to a make command: The prepending rules are as follows:

- to a directive or command, such as, −include <filepath>, will prevent aborting make if the file is not found.

@ to a command will prevent make from printing the command before it is executed.

+ to a command will cause it to be executed even if make is invoke in a "do not execute" mode.

-n to a command will cause the command to be printed but not executed. The "-n" option is used in makefile components at the leaves of the hierarchical Makefile to print a list of the types that are (declared ?) by the source code found in subdirectories.

The dual repo contains all the C libraries that are used both in the Ethos kernel and in userspace. The dual repo consists of 15 subrepos, each of which is a library to be linked into the Ethos Operating System. These are libadt, libcore, libcrypto, libdebug, libethos, libetn, libevent, libfmt, libkernelTypes, libminimalt, libnetStack, libstring, libutf, libxalloc, nacl. In turn, each library contains a C source code subdirectory (lib), a C header file subdirectory and a module.mk, used to compile and link the source code files in that library. The lib subdirectory may contain additional subdirectories that have, for example, architecture specific assembly code for the Ethos OS entry ("entry.S") and a subdirectory containing additional userspace source code userspace.

The kernel repo contains several subdirectories with C code to implement the Ethos kernel. There kernel/include subdirectory contains the header files (.h files) for the kernel module. The kernel directory also contains a module.mk file that is included in the Ethos Makefile. In module.mk, are definitions for kernel.objects and kernel.objects.coded for object files; kernel.headers for header files.

There are also repos for Ethos user space (userspace) Linux (linux), ports, and languages.

Architectures

The makefile provides defined terms for building an Ethos operating system on two different architectures, x86_32 and x86_64. Any other architecture will cause an error. The top.mk file sets the target architecture for the build. The default is x86_64, which is set by the command default definition operator:

```
TARGET_ARCH ?= x86_64
```

The definitions relevant to both architectures are set in the file bottom.mk. The important settings are shown in Figure 9.1. These settings give the Xen Interface Ethos uses (4.3), the C preprocessor flags, the base C flags (C flags differ for kernel compiles vs. userspace compiles), flags for archives, assembly language and Go.

There are options for additional flags to be added to BASE_CFLAGS and BASE_CPPFLAGS. These include optional flags for assertions, debugging, profiling and optimizing the system builds.

Ethos repos are organized by environment, which is kernel space, user space, and linux. Each environment requires different build parameters, and ensuring that these are used correctly is important to ensure Ethos works reliably. There are special flags for kernel space (K_CFLAGS, K_CPPFLAGS, K_ASFLAGS, K_LDFLAGS), user space (U_CFLAGS, U_CPPFLAGS, U_ASFLAGS, U_LDFLAGS) and linux (L_CPPFLAGS, L_ASFLAGS and L_LDFLAGS).

```
XEN_INTERFACE_VERSION := 0x00040300
BASE_CPPFLAGS = -D__$(TARGET_ARCH)__
BASE_CFLAGS += \
        -std=gnu99 \
      -fno-optimize-sibling-calls \
         -fno-stack-protector \
          -Wall \
          -Wnested-externs \
          -Werror \
      -static \
      -fno-builtin \
     -fno-leading-underscore
AR_FLAGS ?= -s -r -c
BASE_ASFLAGS (architecture dependent)
GOARCHFLAG = -m32 |  -m64
GOARCHLDFLAG = -melf_i386   | -melf_x86_64
GOARCH      386  =    | amd64
GOC      x86_32-xen-ethos-8g | x86_64-xen-ethos-8g
GOL x86_64-xen-ethos-8l | x86_64-xen-ethos-8l
```

Figure 9.1: Key settings

There are also architectural flags to include specific libraries in the loading and linking phase. The basic loader flag for user space applicable to all architectures is:

```
U_LDLIBS  := -nostdlib -static -Tmk/ld_$(TARGET_ARCH).script \
      $(UDIR)/userspace/libethosUserspace/lib/libethosUserspace.a
```

The included libraries for the dual module are set as:

```
DUAL_LIBS := adt core crypto debug etn event fmt kernelTypes minimalt\
      netStack string utf xalloc
```

X86_64 In the event Ethos is built for an x86_64 architecture, then there is an additional flag for kernel space to avoid linking with shared libraries:

```
K_LDFLAGS += -static
```

X86_32 In the event that the build is for a x86_32 architecture the flag for that architecture is added:

```
BASE_LDFLAGS += -melf_i386
K_LDFLAGS +=    -static $(BASE_LDFLAGS)

U_LDFLAGS += -melf_i386
```

```
L_LDFLAGS += =march=i686 -m32
```

Invoking make

These are the primary targets for Ethos' Makefile:

all build the code

clean remove build artifacts

install (as root) install Ethos in system directories

uninstall (as root) remove installed Ethos

pull get updated code from repo (note this may need to be done multiple times when pulling a new repo tree)

The test code must be run in this order:

build build the test code

run (as root) run the test code

check check that the test code worked

Make can be invoked without a command line parameter, as make. In that case the special variable assignment:

```
.DEFAULT_GOAL := all
```

tells make to make the target all as opposed to the usual default target which would be the first target in Makefile.

Built-in target names: Target names of the form .PHONY have a built-in meaning with particular execution rules. E.g.: .DEFAULT_GOAL, .PHONY, .ONESHELL, .SILENT, .IGNORE, .PRECIOUS, .EXPORT_ALL_VARIABLES.

The makefile description file sets several defaults. top.mk sets

- architecture to x86_64 (TARGET_ARCH ?= x86_64),
- debugging to no (WITH_DEBUG ?= n),
- performance profiling to no (WITH_PROFILE ?= n), and
- assertions to yes(WITH_ASSERTS ?= y).

Build directories

The directory STAGEDIR contains Go packages and is defined to be:

```
STAGEDIR := $(ETHOSROOTDIR)/destStageDir
```

The system PATH variable is defined to include $(ETHOSROOTDIR)/bin to enable the shell scripts there to be easily used.

The Ethos source code is split into three categories kernel, userspace and linux. The actual source code is categorized according to its purpose and function and placed in subdirectories one or more levels below these three top level category directories. Linux provides linux style services and tools to service and build the Ethos operating system. These three subdirectores are immediate subdirectories of the top level ethos directory.

The build is designed to create a complete set of object files in corresponding directories that are immediate children of ethos. Make will compile the source code files of the Ethos source tree and place the object code files compiled from each source code file into specific subdirectories that parallel their source code directory names. For each of x86_32 and x86_64 architectures there are three build directories (total of 6):

Space	64 bit	32 bit
kernel	kobjs-x86_64	kobjs-x86_32
user	uobjs-x86_64	uobjs-x86_32
linux	lobjs-x86_64	lobjs-x86_32

Collectively, these 6 build directories (defined in the file top.mk) as the BUILD_DIRS.

One architecture is built at a time. The following definitions are used throughout the distributed Makefile:

```
KDIR = kobjs-$(TARGET_ARCH)
UDIR = uobjs-$(TARGET_ARCH)
LDIR = lobjs-$(TARGET_ARCH)
```

The Makefile component top.mk also provides a forward definition of ethos.types as dual.libkernelTypes.ar. Any global definition is needed in a module.mk should be defined in top.mk or install.mk.

The Linux archive files, extension ".a", are created in module.mk. To build the ".a" file for a sub-module,

(1) each source code file ("*.c") is compiled into an object code file ("*.o")

(2) each assembly code file, e.g., entry.S file is compiled into an object code file entry.o.

(3) The object code (both C code and assembly code) are consolidated into an archive file for the repo.

Compiler Rules and Flags

The built-in compiler default rules are overridden in the first lines of top.mk. The compiler rules are set in bottom.mk using shell variables to allow variation for each Ethos module to be built. Separate rules are provided to build different classes of object files. Dual Object C Code Files (DOBJS), Kernel Object C Code Files (KOBJS), Kernel Object Assembly Code Files (KASOBJS), User Object C Code Files (UOBJS), User Object Assembly Code Files (UASOBJS) and Linux Object C Code Files (LOBJS). These rules use shell variables to allow for variation among the different modules that make up the Ethos OS.

Base compiler flags are also set in bottom.mk. These include flags for object files built from the C PreProcessor (BASE_CPPFLAGS", C (BASE_CFLAGS), Assembly (architecturally dependent) (AS_FLAGS) and Go files (GOARCHCFLAG and GOARCHLDFLAG). The base flags are then used to build more specific compiler flags for kernel space, user space and linux space. These more specific flags for kernel, user and linux space also include flags for the loader. In the Kernel case, there are definitions for K_ASFLAGS, K_CFLAGS, and K_LDFLAGS.

The flag for including dual libraries is also defined to include the dual sub-modules

```
DUAL_LIBS := adt core crypto debug etn event fmt kernelTypes minimalt netStack stri
```

In turn, the dual libraries are included in the base C PreProcessor flags (BASE_CPPFLAGS).

Note that for the x86_64 architecture, b there are a special set of flags to disable the XMM registers that the gnu compiler uses (NOXMM_CFLAGS)

Install definitions

The file install.mk consists of several definitions that identify the install location of many of the files that will be built. The essential directories are:

```
install.kernel     = /var/lib/xen/images/ethos.$TARGET_ARCH).elf
install.initialStore  =   /var/lib/xen/images/initialStore.$(TARGET_ARCH).tar
install.rootfs        =   /var/lib/ethos/$(install.rootfs.name)/rootfs
install.ethos.header    = /usr/include/ethos
                  install.ethos.lib       = /usr/$(TARGET_ARCH)-xen-ethos/lib
install.bin     = /usr/bin
install.sbin       = /usr/sbin
install.minimaltd.rootfs          = /var/lib/ethos/minimaltd/rootfs
```

The install command will install the Ethos file system at install.rootfs and program files at $(install.rootfs)/programs.

Example

We describe the dual/libcrypto/module.mk next. Since the file is in dual/libcrypto all names begin with dual.libcrypto. We define dual.librcrypto.dir which is a subdirector of dual. Code in dual are built with kernel flags and stored in the kernel directory, see dual.libcrypto.build. The target is to create libcrypto.a. We define the list of dual objects, and add them to

DOBJS, at the list of sources to SRC. Running all at the top level will run dual.libcrypto.all.
The DOBJS are built as in the top level level files.

```
dual.libcrypto.objects       := certificate.o netRead.o netWrite.o puzzle.o random.
dual.libcrypto.headers       := certificate.h crypto.h encryption.h puzzle.h

dual.libcrypto.dir           := $(dual.dir)/libcrypto
dual.libcrypto.header.dir    := $(dual.libcrypto.dir)/include/ethos
dual.libcrypto.build         := $(KDIR)/$(dual.libcrypto.dir)/lib
dual.libcrypto.objects.build := $(patsubst %, $(dual.libcrypto.build)/%, $(dual.lib
dual.libcrypto.target        := $(dual.libcrypto.build)/libcrypto.a

dual.libcrypto.src           := $(patsubst %,$(dual.libcrypto.dir)/lib/%, certifica
                                $(patsubst %,$(dual.libcrypto.dir)/include/ethos/%

DOBJS += $(dual.libcrypto.objects.build)
SRC   += $(dual.libcrypto.src)

dual.libcrypto.all: $(dual.libcrypto.target)

$(dual.libcrypto.target) : $(dual.libcrypto.objects.build)
    $(AR) $(ARFLAGS) $(dual.libcrypto.target) $(dual.libcrypto.objects.build)

dual.libcrypto.clean:
```

Chapter 10

Coding notes

10.1 Introduction

The purpose of a coding style is to make the code easier to read. Making it easier to read reduces mistakes and makes it easier to find mistakes.

Some of the elements of the coding style are arbitrary. They are made on aesthetic grounds, and different people have different perceptions of what would be best. However, there is significant benefit in uniformity; even an ugly coding style, consistently followed, is better than the most beautiful coding style which is not followed. Quite simply, uniformity decreases the cognitive load so that the programmer can thing about the meaning of her program.

Other elements reduce errors directly. They go by various names, such as defensive programming, but their purpose is to reduce errors just by the act of following them. These elements arise from experience of having be bitten by a bad bug. As our experience grows, we expect the list of these elements to grow larger.

Bugs are the result of misunderstandings. All security holes are bugs. To avoid bugs, careful thought is needed to:

- Reduce complexity: simplicity is the greatest aid to understanding. Strive always to make the solution as simple as possible.
- Understand the problem before you code the solution: Separate thinking, designing, and learning from coding the solution. Some of this learning will come about from trial coding, but such trial coding should not be the final code.

Some other tips are:

- Implement incrementally: whenever possible, implement incrementally the most important thing first. Its easier to check that way, and you learn more as you go. Often you find the things you thought were necessary, aren't.

- Performance: don't do any performance tuning until you have measured code performance. Performance tuning makes code more complicated, so it should only be done for significant benefits.

There is some old code in Ethos which does not follow the conventions here. We will eventually convert that code over to be consistent with the coding style.

The coding style is aimed at making the code as clear as possible, reducing the need for documentation, increasing understanding of the code, and reducing bugs. Hence, a primary goal of this coding guideline is to increase the code's ability to do so.

10.2 General issues

10.2.1 Partitioning

A large program is partitioned into multiple files. The purpose of this division is to increase decoupling of code. Code within a file is highly coupled; code across files is less coupled.

Decreasing coupling between files is an important part of programming. Techniques include:

- Moving declarations out of .h files and into .c files
- Making declarations within .c files `static`, if they are not needed outside the file.

It is important to increase the adhesion within files. Rather than update variables, updates should be performed through interfaces (i.e., functions). Each file manages a set of data structures, or performs a single type of function. Keeping these together enables better reasoning about what the code does.

10.2.2 Information hiding

The code should be structured to maximize information hiding. Most of the code should not be called from outside the collection of files. Sometimes, a set of N files can have only 1 file which is callable outside this set; this is by far the most advantageous setup.

10.2.3 Portability

Portability is handled whenever possible by types, if this is not possible it must be handled by code. Mostly, the code issues are handled by having `arch` directories which contain architecture dependent code. One goal in the design of an OS is to minimize the code in such directories.

The type portability issues center around integer types. Table 10.1 shows the sizes of integer types using standard gcc settings.

field	32-bit arch	64-bit arch
int	32 bits	32 bits
long	32 bits	64 bits
void*	32 bits	64 bits
long long	64 bits	64 bits

Table 10.1: Sizes

10.2.4 Types

Types are important for both documentation and for portability. For example, integers are used for many different purposes; by using different type names for different uses, these uses can be distinguished without comments. Moreover, such type names alert the developer to needs to do casts or form appropriate integers. Type names are introduced typedefs, for example:

```
typedef int int32;
```

Note that as far as C language type checking, these typedefs are equivalent to there underlying types. So the about int32 is no different than int. Hence, the consistence of these must be checked manually. Of course the typedefs make it easier to do this check.

For portability, some values should be, for example, word size (such as values to hold pointers) and some should be fixed size regardless of architecture. Typedefs (combined with macros) provides this facility.

In addition, constants need to be appropriately sized. The type of an integer literal is the smallest size, greater than or equal to the sizeof(int), which can hold the variable. Constants should be named, the easiest way to do so is with a macro.

```
#define AnInt 123
#define ALong 123L
#define AnUnsignedLong  123UL
```

We note that the sizes are particularly important when doing bit masks and shifts. For example,

```
n << x
```

n should be a type which is large enough to hold the result of shifting right by x.

A second example is making masks.

```
#define PAGE_MASK (~(PAGE_SIZE-1))
```

On the x86, PAGE_SIZE is 4096, but its size should be vaddr_t to reduce the need for casts (and to remove errors that occur when casts are not put in).

10.3 Floating point

In modern architectures there is a substantial amount of floating point registers. These registers are used both for traditional floating point uses and for multimedia extensions.

Since an OS kernel does not generally need to do floating point operations, floating point registers are not generally saved on kernel entrance or restored on leaving the kernel. In fact, they are not saved/restored when switching processes.

Instead, floating point registers are stored only for processes which used them. To determine whether a process uses them, a floating point exception is generated on first use and the previous version of the floating point registers are stored.

Floating point is frowned upon in the Ethos kernel, but it is allowed. It is used primarily by the crypto library. Before use of floating point in the kernel fpuGuard() must be called.

This means that floating point should not be used in ethosFoundation, ethosFrame, and types.

10.4 Memory and other forms of resource exhaustion

An OS, and many services built on top of an OS are intended to run forever. That means that we need to prevent **storage leaks**—unused but not freed memory—as even small storage leaks can cause the system to fail.

It is also necessary to deal with **resource exhaustion**—particularly running out of memory. Running out of memory results when the loads on the system exceed its resources; since resources are bound this can happen on any system which does not have a fixed work load.

There are several approaches to deal with resource exhaustion:

- Count the number of resources before executing the routine, and fail if there are insufficient resources
- Pre-allocate the resources
- Check resource exhaustion as you go, cleaning up when necessary.

The first two are best when the setup is complicated. The last requires that it is easy to recover from failure, this happens, for example, when allocating memory for a process since we first allocate and then only if this succeeds install the memory.

10.5 Style

10.5.1 Procedures

Procedures should check their own arguments. ASSERTS should check only for incorrectly constructed and integrated code. Error returns should be used if its possible that a legal program calls a function, and can recover from a failed value.

Each procedure should be commented with the purpose, the meaning of parameters, and the value returned.

10.5.2 Naming

Naming plays an important role in large software projects, in computer science in general, and in the world at large. Napoleon Bonaparte ensured that the Jews of Europe had last name—prior to that a Jew would be name Jacob ben Joshua (meaning Jacob son of Joshua). Without last names there was too much ambiguity. With last names, ambiguity was decreased, and thus Napoleon could draft Jews into the Grande Army. Progress, at least from Napoleon's viewpoint.

In a large software project, one cannot memorize everything. So regularity in names and structure enable one to navigate the project incrementally and spend less time searching. It also means that there will be less bugs.

Ethos naming is important for files, functions, and variables. Names are given in camel hump notation, in which successive words (or acronyms) begin with a capital. Hence, camel hump is camelHump in camel hump notation. Underscores are not used.

Abbreviations are rarely used.

Filenames use camel hump notation too. Procedure and file-scoped variable names defined within a file have as their prefix the file name. Normally, .c files are paired with .h files (with the same base name) which define the interface necessary to use the .c code.

The name of external functions usually starts with the file name they belong to.

10.5.3 Conditionals

Whenever possible, conditions should be linear rather than nested. Kernel code has a lot of the tests for errors, and this needs to be as simple to scan as possible. Figure 10.1 shows the preferred linear style.

Return should be simplified. Rather than use an if-statement to return a constant true or false, the value can be returned directly, as in Figure 10.2.

Conditions—whether in macros, loops, or if statements—should be as simple as possible. They should not include any procedure calls except to predicates—functions that have no side effect on global variables, do not return results through parameters, and thus only return a result as the value of the procedure.

Do use the unlikely(x)/likely(x) annotations in conditionals.

Make full use of conditional expressions like *a ? b : c* to make the code more concise.

No need to use *else* if the function will return within *if* statement and just let them go under the *if*, especially when you are working on kernel code.

10.5.4 Indentation

When defining a procedure, the name begins in the first column. Type information and scoping (e.g., static) are given in the line above. Each scope is indented 4 more characters.

```
if (x)
   {
        return blah;
   }
else
   {
        if (y)
          {
             return foo;
          }
   }

return xyz;
```

(a) Don't do this

```
if (x)
   {
        return blah;
   }

if (y)
   {
      return foo;
   }

return xyz;
```

(b) Do this

Figure 10.1: Linear if statements

```
if (x)
   {
        return true;
   }

return false;
```

(a) Don't do this

```
return x;
```

(b) Do this

Figure 10.2: Return replaces if statement

For example:

```
if (x)
    {
        x += 2;
    }
```

Note that braces are on a separate line by themselves and are *always* used, even when the condition has a single statement.

10.5.5 Code once

Do not replicate code. If the same pattern is appearing multiple times, make it into a procedure call.

Do not reinvent the wheel. If you need something, it probably already exists. If it does not, code it or write in a way that makes it generic and useful elsewhere. Put useful macros that are missing into include/macro.h.

10.5.6 Defensive coding

Ethos has several macros defined to help catch programming errors:

ASSERT is used to check parameters have appropriate variables. ASSERT should be used everywhere liberally—ASSERT on status after calls, ASSERT on state expected. ASSERT is your friend and *will* help you catch a large percentage of programming mistakes which have to do with the way procedures are integrated together. Note that ASSERT code is not present when compiling with optimizations enabled, so it should contain no code which is required to be executed.

ASSERT can only be used for internal inconsistence.

C_ASSERT does compile-time checking vs. ASSERTs runtime checking. Thus it is used to ensure sanity with constants, structure sizes, etc.

REQUIRES this is a condition that is required to be true, because the programmer has not yet put in appropriate error handling. That is, in finished code there should be no REQUIRES statements. Unlike ASSERT, REQUIRES code is present.

BUG/BUG_ON A condition so severe that program execution cannot continue and error code cannot recover from.

All functions where you expect failure should return error status, and should check error status of functions they call. Error numbers are defined in `status.h`. If you need another error number—add one. What is meant by "expecting failure?" I'd estimate—anything in the path of servicing process requests—i.e. stuff that means that the rest of the system can carry on, even if this particular process is screwed. Initialization stuff or really core stuff shouldn't bother unless it's a failure that can be reasonably recover from.

10.5.7 Comments

Comment everything verbosely, but not noisily. Your comments should reflect what is behind your statements, not the actual statements—i.e. *why* you're doing something, not *what* you are doing.

Don't comment bad code, rewrite it.

The code is the low level details. The high level issues is what the vast majority of comments should deal with. Procedure purpose and parameters, file purpose and use, variable and type declarations all need comments. Use `//` comment form in preference to `/* ... */`.

10.5.8 Macros

Use static inline functions whenever possible in preference to macros. At the same time, not everything is a nail, and you have more tools in your toolbox than just a hammer.

Use typedefs rather than defining a type with a macro. Macro's are sometimes useful for defining compile-time constants, especially those which are architecture specific. Enums can be used instead for C code, but macros work in assembly code as well.

Sometimes a macro is the right thing to use. For example, ASSERT is a macro because it prints its arguments as a string. Other macros might need to use function-specific variables, such as `__func__` to print out the current function name.

10.5.9 Static

Variables and procedures which are not used out of the file scope should be declared static. These variables and procedures typically make more assumptions about their properties when called and are often the result of eliminating duplicate code.

10.5.10 Declaration

The prototype of internal functions should be declared on the top of the *.c* file while the prototype of external functions should be declared in its corresponding *.h* file.

10.5.11 Function arguments

Arguments of functions should be checked for sanity whenever possible, especially for external functions.

10.5.12 Kernel coding

Do try to separate architecture-dependent from architecture-independent code.

Appendix A

Frequently Asked Questions

Why can't Xen find my Ethos kernel, even though I confirmed its path?
SELinux is probably enforcing its policy with regard to Xen kernels. Either turn SELinux off, apply the proper SELinux label to the Ethos kernel, or install the Ethos kernel at

`/var/lib/xen/images/`.

The latter option will ensure the system properly labels your Ethos kernel; this is the default install location used by the Ethos Makefile.

Why does *et* hang when I create a *client* and try to connect with it with *et*?
Et needs SayI to run, and by default that runs on the server. So create a server and use that.

Why are my Ethos programs unable to connect to a remote host?
There are many things on Dom0 that affect Ethos networking.

- Dom0 firewall; `00-vif-local.hook` manipulates the firewall so that packets may be forwarded on behalf of Ethos guests.
- Dom0 routes; `vif-route` manipulates the routing table so that Dom0 knows which port each Ethos guest is present on.
- Dom0 proxy ARP; `network-route` turns on proxy ARP; for more on this idiosyncratic subject, please refer to *Networking Scenarios*.
- Dom0 IP forwarding; `network-route` turns on IP forwarding when `xend` starts at boot time.

Why is Dom0 periodically unresponsive when running Ethos in DomU?
You may be encountering a guest scheduling anomaly. Confirm that `dom0_max_vcpus=1` is listed in Xen's boot options, as defined in `/etc/grub.conf`. You may also want to read `http://wiki.xensource.com/xenwiki/XenBestPractices`.

Notes

[1] Both 32-bit PAE and x86_64 Fedora may serve as a Dom0 guest, resulting in a Xen environment capable of running 32-bit PAE Ethos.