The following paper was originally published in the
Proceedings of the Twelfth Systems Administration Conference (LISA ’98)
Boston, Massachusetts, December 6-11, 1998

Infrastructure: A Prerequisite for Effective Security

Bill Fithen, Steve Kalinowski
Jeff Carpenter, and Jed Pickel
CERT Coordination Center

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Infrastructure: A Prerequisite for Effective Security

Bill Fithen, Steve Kalinowski, Jeff Carpenter, and Jed Pickel – CERT Coordination Center

ABSTRACT

The CERT Coordination Center is building an experimental information infrastructure management system, SAFARI, capable of supporting a variety of operating systems and applications. The motivation behind this prototype is to demonstrate the security benefits of a systematically managed infrastructure. SAFARI is an attempt to improve the scalability of managing an infrastructure composed of many hosts, where there are many more hosts than hosts types. SAFARI is designed with one overarching principle: it should impact user, developer, and administrator activities as little as possible. The CERT Coordination Center is actively seeking partners to further this or alternative approaches to improving the infrastructural fabric on which Internet sites operate. SAFARI is currently being used by the CERT/CC to manage over 900 collections of software on three different versions of UNIX on three hardware platforms in a repository (/afs/cert.org/software) that is over 20 GB in size.

Background

Since the formation of the Computer Emergency Response Team, ten years ago, it and nearly a hundred other subsequently formed incident response teams have been providing assistance to those involved in computer and network security incidents. The CERT® Coordination Center has participated in the response to well over 10,000 incidents over that period. Throughout that interval, incident response teams have attempted to convince their constituencies to apply security patches and countermeasures to hosts on a routine basis. Over the course of its life, the CERT/CC alone has issued over 160 advisories, 70 vendor-initiated bulletins, and 20 summaries. Most of these documents urge system administrators to apply a variety of countermeasures. Yet, on a daily basis the CERT/CC receives incident reports involving hosts that have been compromised by intruders exploiting vulnerabilities with publicly available patches or countermeasures. The overwhelming majority of compromises continue to be a result of sites running hosts without applying available countermeasures.

In a recent survey we conducted, the most frequent reasons for continuing to operate hosts without available countermeasures were:

- Insufficient resources
- Difficulty in finding and understanding countermeasure information
- The inability to administer configuration management across a large number of hosts.

At many of these sites, the basic computing infrastructure is an impediment to timely distribution of configuration changes. These sites operate at a substantially higher level of risk than those with solid infrastructures. Such sites are not only more likely to be compromised, but they are also less likely to be able to adequately detect and recover from compromises. They are often forced to undertake large efforts to secure their hosts and networks. Often the required resources are unavailable, preventing administrators from completely recovering from compromises, usually leading to subsequent compromises.

In this paper, when we refer to an organization's information infrastructure, we mean:

- The organization's installed base of computing and networking hardware,
- The system and application software operating on that hardware,
- The policies and procedures governing the design, implementation, operation, and maintenance of the software and hardware, and
- The people who perform those procedures.

One assumes that, in general, such an infrastructure exists only to serve some business purpose of the organization. In a very real sense, an organization's information infrastructure is business overhead-part of the cost of doing business.

An information infrastructure management system (IIMS) is a system whose purpose is to automate the maintenance of an information infrastructure.

Motivation

A site with an adequate information infrastructure finds itself able to detect and recover from compromises in short order. It can repair or recreate hosts using automated mechanisms designed to deal with such demands. It can also protect itself from many types of attacks through proactive deployment of countermeasures.

A site without such an information infrastructure must spend much greater effort to detect and recover from compromises or to proactively deploy countermeasures. This effort is proportional to the number of hosts being managed. At a small site, this may be

acceptable or even ideal, but tends not to be scalable as the site grows due to resource constraints.

At a site with an effective information infrastructure management system, the effort to detect and recover from compromises or to proactively deploy countermeasures is substantially less. In general, the effort to deploy a change is proportional to the number of different types of hosts, rather than the number of hosts. However, the effort to prepare for such a deployment is higher due to packaging requirements of the IIMS.

Chart 1 shows some simpleminded math to illustrate these ideas.

Let:

- N_H be the number of hosts
- N_P be the number of host platforms
- E_H be the effort to make a manual change on one host
- E_P be the effort to prepare a change for automatic deployment on any host of one platform
- E_D be the effort to deploy a change to one host automatically

Then:

E_M is the total effort to make a manual change on N_H hosts:

$$E_M = E_H \times N_H$$

E_A is the total effort to make a change automatically on N_H hosts

$$E_A = E_P \times N_P + E_D \times N_H$$

Therefore:

The scalability breakeven point is when:

$$E_M = E_A$$

If one assumes that in a good IIMS, the effort to deploy a change on one host automatically (E_D) is negligible, then the breakeven point is when:

$$N_H \times E_H = N_P \times E_P$$

Chart 1: Mathematical derivation of fundamentals.

As you can see from this simplified model, at the breakeven point, the effort to deploy change manually is proportional to the number of hosts, while the effort to deploy the same change is proportional to the number of host platforms (types of hosts).

Requirements

The CERT Coordination Center is currently developing an experimental IIMS called the *Securely Accessible Federated Active Repository for Infrastructures* (SAFARI). Via SAFARI, the CERT/CC is pursing two primary goals:

- Enable sites to effectively, securely, and economically distribute host-resident software from one or more managed repositories.
- Enable sites to manage all software throughout its entire operational lifecycle focusing on quality, reliability, and integrity.

SAFARI is being designed to meet four primary objectives:

- Construct each host in its configured state automatically from an empty disk with minimal manual intervention.
- Reconstruct each host to its configured state after a security compromise or catastrophic failure automatically with minimal manual intervention.
- Upgrade each host with new applications, operating systems, patches, and fixes automatically on a regular basis with no manual intervention.
- Maintain each host in its configured state automatically with no manual intervention.

In order to achieve the desired influence in the system administration community, three additional secondary objectives must be met:

- It should be engineered for an extremely high host-to-administrator ratio.
- It should facilitate the sharing of software among multiple, not necessarily mutually trusting, administrative domains.
- It should follow a policy-based model that works with a wide range of organization sizes.

Whatever compromises are necessary throughout the evolution of SAFARI, the following constraints must not be relaxed:

- Guarantee that software is distributed onto authorized hosts only.
- Guarantee that software is delivered with integrity to each host and that it is installed correctly.

The design should be guided by the following overarching philosophy:

- It should support user, developer, and administrator activities in a manner that most closely parallels those same activities before SAFARI.

Design Assumptions

This work differs from related projects in some ways. While there have been many projects relating to large scale host administration or software distribution, this project is motivated primarily by scalable security. As a result, we have made certain design assumptions that run counter to prior published work.

We don’t consider disk space conservation to be a significant design motivator. Our most recent acquisitions of reasonably inexpensive 18 GB disks supports our assumption. Therefore, a number of space conservation techniques can be immediately discarded. For example, the process of preparing software for distribution via SAFARI is complex enough without an extensive set of space saving rules to follow. Therefore, we decided not to have such rules. Unlike systems that divide a software package between platform-independent and platform-specific [20], we decided that minimal impacts on the build and installation process would easily pay for the

additional disk space. Another impact of this assumption is that everything that goes into building a unit of deployable software is important and should be preserved. This includes files produced during intermediate steps of a build and install procedure. Such intermediate files can be useful during testing and debugging a deployed unit.

We consider individual host performance to be more important than local disk space. We decided to take the default position, that unless otherwise directed, it is the host administrator's intention to install all distributed software locally. We realize though that it might be necessary to locate certain parts of a local host filesystem on a fileserver.

We believe that software maintenance and distribution should be as automated as possible. As a result, we have decided that every available function within SAFARI must be available via a shell command; no GUI only capabilities are allowed. At the appropriate time, we will introduce GUI capabilities to reduce the complexity of the system, but not before a reasonably complete set of functions are already available.

We believe that end-to-end repeatability is of great importance. Repeatability and security are very closely related; security cannot be maintained without repeatability. Therefore, we decided to define the SAFARI model to include package construction activities in SAFARI; this increases the probability of being able to either repeat a build process or to audit it--tracing binaries back to source.

SAFARI, like virtually all similar systems, is focused on the management of small, cohesive, largely independent **software units** (SU). Unlike other systems, SAFARI manages four types of SU's, each useful for a different phase of the software operational lifecycle. All of these SU's are stored in a relocatable structured filesystem tree, called the **SAFARI Repository**. The matrix in Table 1 shows how the various types of SU's relate to one another.

SAFARI is designed to manage software targeted for multiple platforms in a single repository. Both packages and images are labeled with a platform label. Neither collections nor clusters are platform specific. We will examine each of these in more detail.

As an aside, the current implementation of SAFARI depends heavily on Transarc AFS [8] for a variety of administrative functions. This has two significant consequences (in the genre of good news/bad news):

- The fundamental architecture of SAFARI is greatly enhanced by basic AFS capabilities (e.g., network authentication, integrity, and security). Consequentially, a wide variety of SAFARI administrative tasks (e.g., space allocation, quotas, access control) are greatly facilitated by the use of AFS administrative capabilities. The capabilities provided by AFS were considered a required architectural foundation for a system on which every host at a site may depend. Any other environment that provides similar functions would be acceptable (e.g., DFS).
- The current implementation of SAFARI is tightly integrated with and dependent on AFS and is therefore useless to sites unwilling or unable to run AFS. Note, however, that SAFARI does not require that a site use AFS for anything other than housing the SAFARI repository.

Platforms

Being a multi-platform information infrastructure management system, SAFARI must define and manage the concept of a **platform**. Attempting to be as flexible as possible, SAFARI makes few assumptions regarding characteristics of platforms or the relationships among platforms. In SAFARI, platforms are assigned arbitrary labels by SAFARI administrators as they are needed. SAFARI platform names are composed of any combination of mixed case alphanumeric characters plus dash and underscore, starting with an alphabetic character; they match the regular expression:

```
/^[a-zA-Z][-a-zA-Z0-9_]*$/.
```

SAFARI assigns no meaning to the contents of the name; SAFARI administrators are free to choose any platform naming convention they like.

Each SAFARI host must be assigned a platform name. Typically, a host platform name refers to a specific combination of hardware and system software. For example, Sun Solaris 2.5.1 running on a Sun Ultra 2 model 2170 is a platform that one SAFARI site may assign the label *sparc-4u-sunos-5.5.1*; another may assign *solaris-2.5.1_sparc4u*, and a third may choose *sun4u_251*.

	Source Software Units	Deployable Software Units
Development of Independent Software Units	Collection	Package
Configuration or Integration of Software Units	Cluster	Image

Table 1: Relationships between software units.

Each SAFARI deployable software unit (package or image) is also assigned a platform name. Deployable software unit (DSU) platform names are typically more abstract than host platform names. For example, a PERL script might easily be written so as to run on Solaris 2.5.1 on SPARC and Red Hat Linux 5.1 on Intel. In which case, one might choose to assign to the package containing the PERL script a platform name such as *unix*. Choosing an appropriate platform name for a DSU can sometimes be challenging. Continuing this example, suppose that the PERL script did not work on AIX 4.3 on PowerPC. What platform name would be appropriate then?

All platform names are recorded in the SAFARI repository database. In addition, the repository contains a mapping between each host platform name and the platform names of DSU's that are allowed to run on hosts of that platform. That is, it documents which DSU's can be run on which hosts by platform name. For example, the repository might declare that a host platform named *sparc-4m-sunos-5.5.1* can run packages or images tagged with platform names *sparc-4m-sunos-5.5.1*, *sparc-sunos-5.5.1*, *sparc-sunos-5*, *sunos-5*, *sunos*, *unix*.

All SAFARI tools that deal with platform names require that platforms be recorded in the SAFARI repository database.

Packages

The SAFARI **package** is the easiest class of SAFARI software unit to understand. SAFARI packages are logically equivalent to the installable units in many other software management systems: **packages** in Sun Solaris [1] and UNIX System V [2], **packages** in Red Hat Linux [3], **packages** in IBM AIX [4], and many others. In fact, the design for SAFARI packages was taken almost directly from CMU Depot [5] (an ancestor of SAFARI, *Alex*, developed at the University of Pittsburgh [6] actually uses *depot* internally).

A SAFARI package is a sparse relocatable filesystem tree. It has exactly the same internal directory structure as a host at which it is targeted for installation. That is, if a given package is intended to install the files `/etc/hosts` and `/usr/bin/cp`, then one would find within the package filesystem tree the files `etc/hosts` an `usr/bin/cp`. There is currently no canonical transportable format for a SAFARI package (such as *tar* or *RPM*). Within each package is a special directory which contains meta-information about the package and is not intended to be installed on the target system. This directory is currently named `depot` (more ancestral evidence). It contains a manifest of the package's contents and a certificate of authenticity, digitally signed by the person who produced the package. SAFARI currently uses PGP 2.6 [10] for digital signatures and MD5 [11] for message digests of files described by the manifest.

It may be obvious from the above description, but SAFARI packages are ***platform dependent***. Each package is labeled as being applicable to (installable on) a particular platform.

SAFARI provides tools to aid both the SAFARI administrator and the SAFARI collection manager in the creation, population, release, integration, testing, deployment, deprecation, withdrawal, and destruction of packages in a repository and on client hosts.

In most software management systems, package-like entities come from outside the software management system. In SAFARI, a package is derived from another software unit, a SAFARI **collection**.

Collections

The easiest way to understand SAFARI **collections** is to think of them as package sources. One set of sources targeted at *n* platforms requires one SAFARI collection from which *n* SAFARI packages are produced. SAFARI collections are structured as filesystem trees, similar to packages, but in contrast to packages, have minimal mandatory internal structure. This flexibility is required to support the wide structural variety of open source systems. There are three generic areas (subdirectories) within a collection:

- `build`. This area is used to house collection source code, makefiles, etc. There is no preferred structure within this area. Collection managers are free to establish their own conventions for structure in this area. For example, in our repository, it is typical that a `tar.gz` file for sources resides in the `build` directory with the expanded tar.gz file in lower directories. The extensive use of RCS [12] or SCCS [13] is strongly encouraged, but remains a collection manager choice.
- `OBJ`. This area is used to house compiled pre-installation binaries (e.g., `*.o`, `*.a`) for each platform. The `OBJ` tree is expected to be managed by a SAFARI tool named *pf* (described below), but may be used as the collection manager sees fit. The pf command establishes a parallel tree of symbolic links for each platform rooted in the second level of the `OBJ` tree that mirrors the `build` tree. Those familiar with the use of the lndir command in the X11 distribution [7] to accomplish multi-platform builds can imagine how pf works.
- `META`. This area is used by SAFARI to hold meta-information about the collection. For example, the mapping between build platforms (the platform on which the build is performed) and installation platforms (the platform on which the package is expected to run) is contained in `META/platforms.map`.

In addition, for the sake of convenience to the collection manager, any existing unreleased packages

are available under the following area within a collection.

- `DEST`. This area contains one directory for each existing unreleased package for the collection. Technically, these directories are not part of the collection. They are AFS volumes mounted here to make the build process easier for the collection manager by allowing the installation destination to be specified using short relative (and therefore relocatable) paths in makefiles.

Of course, many packages are only available in binary, platform-specific form. For such packages, SAFARI provides tools that can create a package from a delta of a manual installation host. This idea came from the WinINSTALL [9] tool which does the same thing for Microsoft Windows operating systems. It takes a snapshot of an existing host before some software is installed. After the installation, the tool compares the before and after states of the host to produce a SAFARI package that when deployed via SAFARI results in the same host changes as the software installation steps. This approach can be used for any arbitrary modifications, but is better left for situations where software must be installed via an installation procedure that cannot easily be reverse engineered. For some platforms (i.e., Sun SunOS 4), this may be the easiest way to capture patches for distribution to other hosts via SAFARI. Repeatability using such a tool capturing manual steps is poor. Fortunately, this tool is rarely needed since most software installation procedures allow for one to install wherever one wants.

Clusters and Images

In cases where a collection and its packages do not depend on other packages or local configuration conventions to operate properly when installed, SAFARI host managers can select which packages they wish to use with little difficulty. An example of such a collection might be GNU CC. GNU CC really only depends on having the proper operating system in place to function properly. Any host manager who was interested in GNU CC could safely just select a GNU CC package appropriate for the host's platform and be highly certain that it will deploy and operate properly.

However, relatively few collections depend so little on other collections and have no dependence on local configuration conventions. For example, the collection that contains the SAFARI software itself, being written in C++ and PERL, depends on a variety of other collections being available to be able to operate properly. The process of selecting, ordering, integrating, configuring, and testing a set of interdependent packages can be complex. In addition, it occasionally happens that combining particular packages requires significant configuration to get them to work correctly.

Facilitating this process is the purpose of SAFARI **clusters** and **images**. A SAFARI cluster is structured and managed exactly like a collection; images are produced from clusters exactly like packages are produced from collections. Like collections, clusters are not tagged with a platform name; like packages, images are tagged with a platform name.

The principle distinction between collections (and their packages) and clusters (and their images) is that clusters produce images that reflect local configuration conventions, while collections produce packages that are intended to be deployable on any host in the world. The normal progression is to build one or more packages from collections with no local assumptions about deployment hosts and then build a cluster which integrates and configures the packages according to local conventions to be deployable on local hosts. Others not following those conventions can build clusters in the same or a different repository according to their conventions using the same packages, thereby, reusing the existing packages.

A cluster can be thought of as a set of collections, integrated together, and its images can be thought of as the set of packages from those collections integrated together. The primary purposes of clusters and images is to reduce the effort expended by host managers trying to integrate packages together. An image is a pre-configured set (including a set of one) of packages that can be selected as a single unit for deployment.

Two reasons for this approach of separation based on configuration information are:

- **Technical**: This separation facilitates the construction of collections/packages that can be readily used at multiple sites. That is, the collections and packages thus created can be easily shared between SAFARI repositories by concentrating configuration information, which is typically very local, in clusters and images. Thus collaborating sites can easily share packages without extensive human interaction. The clusters, presumably, reflect local configuration conventions and must be reengineered from one repository (domain of local configuration conventions) to another.
- **Organizational**: Some persons are better at building packages one at a time and others are better at integrating a group of packages together. This approach allows staff to be applied more effectively to the tasks to which they are better suited. Explicitly separating the integration and configuration step from the development step allows the developer to concentrate on producing a higher quality package faster. Developers focus on the internals of a package; integrators focus on the world around a package.

The Repository

The SAFARI repository houses:

- Collections and their packages
- Clusters and their images
- Database
  - Meta-information about collections, packages, clusters, and images
  - Policy settings
  - Database meta-information

The SAFARI repository is a relocatable filesystem tree. For current SAFARI commands, its location is specified by the REPO_ROOT environment variable. Future versions of SAFARI will incorporate inter-repository capabilities.

Collections, Packages, Clusters, and Images

Each different class of SAFARI software unit has its own area in a repository.

- `collection`. The collection directory is the parent directory of all collections. Each collection is housed in a separate subdirectory. As described above, each collection holds four internal areas (`META`, `build`, `OBJ`, and `DEST`). As an example, the source to the SAFARI version 1.0 collection might, depending on collection names chosen, be found somewhere in `collection/cert.org/safari/1.0/build/`. The SAFARI administrator *safari collection create* subcommand creates the appropriate AFS volumes (for each area), creates the appropriate AFS protection groups, mounts the volumes in the collection directory tree, sets access controls and quotas, and transfers subsequent control of the collection to the collection's managers to manage the build, release, and test steps.
- `package`. The package directory is the parent directory for all released packages. Unreleased packages are AFS volumes mounted under the collection's `DEST` area only. After completing the `build` and `DEST` area installation steps, the collection manager uses the *safari package prepare* and *safari package seal* subcommands to collect meta-information about the package and certify the contents of the package as suitable for release (at least in the mind of the collection manager). The SAFARI administrator then uses the *safari package release* subcommand to transform the package into the correct form for deployment (e.g., set owners and groups according to policy) and mount it in the appropriate place below the `package` directory. Released packages are housed in directories below package with naming conventions that parallel the `collection` directory. For example, revision 10 of the SAFARI version 1.0 package for platform *sparc-sunos-5* can be found under `package/cert.org/safari/1.0/10/sparc-sunos-5/`.
- `cluster`. The cluster directory is the parent directory of all clusters. The structure below cluster is exactly the same as that below `collection`. The safari cluster subcommands parallel the safari collection subcommands.
- `image`. The `image` directory is the parent directory of all images. The structure below `image` is exactly the same as that below `package`. The safari image subcommands parallel the safari package subcommands.

Each collection and each cluster has a unique name (i.e., collections and clusters have separate name spaces and can therefore have the same name). Packages and images inherit their name from the collection or cluster from which they were derived. Clusters and collections are named using the same syntactic and semantic conventions. A cluster or collection name is composed of three parts separated by forward slashes:

- **Authority**. The authority is the organization or person who is officially responsible for the files found in the cluster's or collection's `build` area. For example, the officially responsible organization for GNU Emacs is the Free Software Foundation. For the Red Hat Package Manager, it is Red Hat, Inc. The authority is encoded as the closest possible representation of the organization or person as an appropriate Domain Name Service Start of Authority record name. This means that the syntactical requirements for the authority part are exactly the same as those for a domain name [17]. In the preceding two examples, Free Software Foundation would be encoded as `fsf.org` and Red Hat, Inc. would be encoded as `rpm.org`, since RPM developers have registered an organizational name just for that software. The rationale for including an authority in a collection or cluster name rather than making it an internal attribute of the cluster or collection was to avoid naming wars. Sometimes there are multiple possible authoritative sources for Internet free software; including the authority in the name avoids the problem of requiring a SAFARI administrator to arbitrate which source is more authoritative.
- **Common name**. The common name is the name by which the collection or cluster is commonly known. It is typically exactly the same name as that assigned by the authority. The only restrictions on this part of the collection or cluster name is that it comply with the POSIX requirements for a file name (a single component of a path) [18]. For example, GNU Emacs might be reasonably assigned the common name "emacs" or "gnu-emacs."
- **Version**. The version is the version string by which the collection or cluster is commonly known. It is typically exactly the same version string as that assigned by the authority. The

only restrictions on this part of the collection or cluster name is that it comply with the POSIX requirements for a file name [18]. For example, GNU Emacs version 19.34 would likely be assigned the version "19.14,', while Samba version 1.1.18p12 would likely be assigned the version "1.1.18p12."

Thus the full name of the GNU Emacs 19.34 collection might be `fsf.org/gnu-emacs/19.34`.

Meta-Information about Collections, Packages, Clusters, and Images

The SAFARI repository database is a directory of ordinary UNIX files, formatted so that the SAFARI administrator can make manual changes in emergency situations. As such, the database is neither highly parallel nor impervious to failed transactions. The database is protected against concurrent access via a simple lock file mechanism. All of the meta-information and policy files are under RCS control to provide assistance during a manual recovery as well as extensive historical information. For reference, our `db/packages` database is at RCS revision 1.3729 as of 8/24/1998 having started at revision 1.1 on 1/27/1997 (yes, 3,728 revisions).

The repository database is located in ${REPO_ROOT}/db.

- `db/collections`. The `collections` database contains the mapping between each collection name and its sequence number. Sequence numbers, which can never be safely reused, are used to form AFS volume names. The presence of a record in this database implies that the collection filesystem tree is present in the repository for that collection (the *safari repository check* subcommand reports differences between the collections database and `collection` filesystem). The use of sequence numbers is intended to deal with severe name length restrictions in AFS volume names (and potentially other underlying technologies that might be supported by SAFARI in the future). Through this indirection, SAFARI supports long collection names that greatly increase understanding in casual SAFARI users (e.g., single-host host managers). Even SAFARI aficionados can benefit from long clear collection names. Our repository currently has over 900 collections; clear naming really is a requirement. When collections are destroyed, their sequence record is moved to the collections.destroyed database for historical reference. The content of this database is managed by the *safari collection* subcommand.
- `db/collections-families`. The database of `collection-families` is used to determine the precedence among a set of collections implementing different major versions of the same software. Each record defines a family of collections from the collections database. The purpose of this database is to deal with collections whose name changes in unpredictable ways as they evolve. In particular, it is rarely the case that the next version number of a given piece of software can be predicted from the current one. The Linux kernel [15] version numbering scheme is a good example: how should the following version numbers be ordered: 2.0, 2.0.1, 2.0.10, 2.0.100, 2.0.9? Because we know the convention, we know that the proper order is: 2.0, 2.0.1, 2.0.9, 2.0.10, 2.0.100. But we have all seen numbering schemes more unpredictable that this. Each collection family explicitly defines the ordering among a set of collections. The package deployment selection mechanism (see **parcel** below) will automatically choose the latest collection with an appropriate platform and status for the given host.
- `db/packages`. The `packages` database contains one record for each existing package. Each package is named for the collection with which it is associated. In addition, each package is assigned a monotonically increasing (for each collection) revision number and a platform name. The first package created from a collection is assigned revision number 1. It is expected that two packages of the same collection with the same revision number (but different platform names) implement the same functionality when deployed. This revision synchronization convention makes it easier for a host manager who is managing multiple hosts of different platforms to configure them to provide the same functions to their users. Lastly, each package is assigned a status tag. Status tags are fully described in `db/lifecycles` under **Policies** below.
- `db/clusters`. The `clusters` database contains the mapping between cluster names and sequence numbers. The clusters database has the same format as the `collection` database and serves the same purpose for clusters as the `collections` database has for collections.
- `db/clusters-families`. The `cluster-families` database is used to determine the precedence among a set of clusters configuring different major versions of the same software. It has the same format as the `collection-families` database and serves the same function for clusters that `collection-families` serves for collections.
- `db/images`. The `images` database contains one record for each existing image. It has the same format as the `packages` database and serves the same function for images that `packages` serves for packages.

Policies

A variety of functional policies have been externalized from code into policy specification files. Future versions of SAFARI will expand the use of external policy specifications. There were two purposes behind this decision. First, we thought it likely that many of these policies might vary between sites. Configuration files are easier to change than code. Second, we were ourselves not sure what our polices should be. Moving these policies from code into configuration files has allowed us to easily experiment with alternative policies and to evolve policy over time.

- `db/allocation`. The allocation configuration file sets the policy for how AFS volumes are named and where they are allocated and replicated. There are currently six types of AFS volumes managed by SAFARI, which are initially mounted at the mount points shown in Table 2.

  The allocation policy allows one to select AFS volume naming conventions using printf-like "%" syntax to conform to local AFS volume naming conventions. It also allows one to define AFS fileservers and partitions that are to be considered as candidates for holding each type of volume. One can also specify which volumes, if any, are to have read-only replicants and where they can be located. The safari command uses this policy to guide its creation of AFS volumes. The server selection portion of this policy is only enforced at creation time; volumes can be moved after allocation however a site wishes.
- `db/lifecycles`. The `lifecycle` configuration file sets the policy for lifecycle phases through which packages and images can pass. The policy defines each phase by assigning it a label called the **status**. Each status label is an opaque string that SAFARI uses for tracking the lifecycle of packages and images. The special status label **unreleased** is built-in and is automatically assigned to packages and images when they are created. When a package or image is released by the *safari package release* or safari image release subcommand respectively, the SAFARI administrator must provide a status label to be assigned to the newly released package or image as the last step of the release process. The status labels are recorded in the `db/packages` and `db/images` databases for each package and image respectively. The lifecycle policy governs which lifecycle phase transitions are allowed by means of a transition matrix. The matrix is composed of cells that contain permit/deny indicators for each possible starting and ending status label.

  Using this mechanism, a site can define a complex lifecycle through which packages and

	Creation Mount Point (below ${REPO_ROOT})	Released Mount Point (below ${REPO_ROOT})
collection build volume	`collection/`*collection*`/build`	
collection OBJ volume	`collection/`*collection*`/OBJ`	
package volume	`collection/`*collection*`/DEST/`*platform*	`package/`*collection*/*revision*/*platform*
cluster build volume	`cluster/`*cluster*`/build`	
cluster OBJ volume	`cluster/`*cluster*`/OBJ`	
image volume	`cluster/`*cluster*`/DEST/`*platform*	`package/`*collection*/*revision*/*platform*

Table 2: Mount points.

Status Label	Status of Package or Image
alpha	Is being tested in operation on a real machine by its manager; no other hosts should try to run it.
beta	Is being tested by end-users on a small set of beta test hosts.
gamma	Is production quality and can be run anywhere the function it provides is useful.
deprecated	Is being withdrawn for some reason (typically because it is being superseded).
obsolete	Has been withdrawn and no host is allowed to use it anywhere.
destroyed	Has been destroyed (deleted).

Table 3: Status labels.

images must progress from creation to destruction. For example, in the CERT repository, we have defined a series of status labels as shown in Table 3.

We define our lifecycle transition matrix to force packages and images to evolve from **alpha** toward **destroyed**; we do not allow transitions in the other direction. We do, however, allow phases to be skipped (e.g., **alpha** to **gamma**, **beta** to **obsolete**, **unreleased** to **beta**).

- `db/platforms`. The `platforms` configuration file sets the policy for naming platforms. Every platform supported by a repository must be defined in this configuration file. Packages and images are labeled with platform names to indicate on which hosts they were intended to be deployed. For any given collection or cluster, packages or images with the same revision number and differing platform names are expected to implement the same functionality. Each platform defined in the platform policy is assigned a unique (across all platforms) small integer to be used in naming AFS volumes. This number is used in a way similar to the collection and cluster sequence number--to reduce the length of the AFS volume name without loss of information. Once a platform is assigned a number, the number can never be reused again (unless one can somehow guarantee that there are not now and never were any AFS volumes created using it).
- `db/platform-families`. The `platform-families` configuration files sets the policy for how platforms, defined in `db/platforms`, relate to one another. Each platform family is, in effect, a sequence of synonyms. The first platform in each sequence is used as a key and serves as the name of the family. It is expected that the platform name assigned to every host is also the name of a platform family. The deployment mechanism (see **parcel** below) uses the host platform name as a key to locate the sequence of synonyms for that platform name from the platform family policy. The deployment mechanism then considers any package (or image) that has been labeled with any of the synonyms as a candidate for deployment to the host. Ambiguities are resolved by selecting the first candidate encountered in the order of the synonyms in the sequence.

  Using the platforms policy and the platform families policy, one can create a taxonomy of platform names. Consider Figure 1 as a platform taxonomy.

  The leaf nodes would then be defined as platform families and the synonym sequence would be all nodes encountered along a directed path from the leaf to the root. Then the platform family *sparc-4m-sunos-5.5.1* would be defined as the platform sequence: [*sparc-4m-sunos-5.5.1*, *sparc-sunos-5*, *sunos-5*, *unix*, *generic*]. Therefore, when the deployment mechanism was activated on a *sparc-4m-sunos-5.5.1* host, it would search for *sparc-4m-sunos-5.5.1* packages (or images) first, followed by *sparc-sunos-5*, *sunos-5*, *unix*, and *generic*.
- `db/protections`. The `protections` configuration file sets the access control policy for released packages and images. It controls file and directory ownerships, UNIX modes, and AFS access controls lists. It does not control local file system access control lists. This configuration file sets the global policy that governs all packages and images. In addition, each package and image can supply its own policy, overriding the global policy, in `depot/protections` in the package or image. The protections policy provides a default value for AFS access control lists, file ownership, file group, and positive and

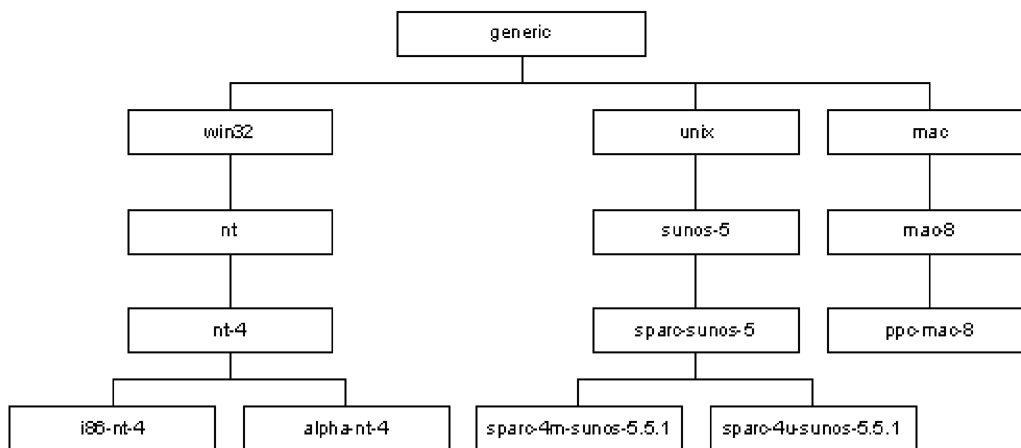

Figure 1: Platform Taxonomy.

negative mode masks. Owners and groups can be specified as names or IDs, but names are preferred (being less platform dependent). The positive mode mask is a mask of mode bits that are unconditionally OR'ed on at release time. The negative mode mask is a mask of mode bits that are unconditionally AND'ed off at release time. Not all mode bits need to be included in either of these masks. If these masks overlap (affect the same bits) the result is undefined.

It is not typically the case that a package or image will override the protection defaults from the global policy, although that is permitted. More often, the package or image protections policy will specify particular paths for which ownerships, modes, or ACL's should differ from the default. This approach is required since collection managers are not typically AFS administrators and as such do not have the privilege necessary to actually set file ownerships or groups. The package or image protection policy is used to convey the collection manager's intent regarding protections to the SAFARI administrator for the *safari package release* or *safari image release* process. Currently, collection managers can request any protection policy they want; it is up to the SAFARI administrator to recognize that a particular release request ought to be rejected due to a suspicious protections policy (e.g., making `/bin/sh` set-UID *root*).

- `db/pubring.pgp`. The `pubring.pgp` configuration file contains the PGP 2.6 public keys of all persons that can digitally sign something in SAFARI. This is typically collection and cluster managers as well as SAFARI administrators.
- `db/relations`. The `relations` configuration file sets the experimental policy regarding relationships between and among collections, packages, clusters, and images. As of this writing, this policy is still being defined. The intention of this policy is to specify the following relationships:
  - **Conflict resolution** is necessary when two packages or images both wish to deploy the same path. The current deployment mechanism has an implicit conflict resolution policy, but it is considered inadequate for large complex configurations. The depot [16] conflict resolution mechanism is also regarded as too weak.
  - **Deployment dependence** occurs when the deployment of a package depends on the prior deployment of some other package. This is typically only the case when SAFARI related packages are deployed.
  - **Operational dependence** occurs when a package depends on the concurrent deployment of another package in order be able to function. The deployment mechanism must be able to follow these dependencies and either automatically add missing dependencies or produce diagnostics regarding missing dependencies.
  - **Construction dependence** occurs when a collection depends on the prior deployment of another package in order to be built (to produce a package). This is mostly a documentation issue for collection and cluster managers, but plays a role in version and revision tracking. Once in a while it happens that it is necessary to restore a destroyed collection and rebuild it for some reason. To be able to get back to a known state, one must know not only what the state of the collection in question was at that point in history, but one must also know the state of all of the packages that the collection depended on during the build process. We have several times found ourselves in the state where we knew for certain that a given package was built from a particular set of sources, but we could not reconstruct what packages that collection depended on during its build process. This typically happens when the build process of one collection uses a `lib*.a` file provided by another package.
  - **Mutual exclusion** occurs when the deployment of one package or image degrades or destroys the ability of another concurrently deployed package or image. The deployment mechanism must be able to detect and avoid this situation.

  Experimentation with representation of relationships has been underway without a clear resolution for almost a year. This turns out to be a hard problem.
- `db/roles`. The `roles` configuration file sets the policy regarding who can do what. It defines a set of roles using UNIX or AFS groups. That is, rather than using UNIX or AFS groups throughout SAFARI to authorize operations, SAFARI uses a set of roles for authorization and determines who is in each role once at program initialization time. This allows a site to define the set of UNIX or AFS groups they wish to use to authorize SAFARI operations.

  There are no predefined roles. A site may define as many or as few roles as their

organizational structure justifies. For each SAFARI operation, the roles policy defines which roles can perform that operation. Each operation also defines who (in the form of an email address) should be notified about each operation.

- `db/structure`. The `structure` configuration files sets the experimental policy regarding file system structure for release validation. To reduce the possibility of totally invalid file system structures being deployed, even in testing, and destroying a machine, the release process must be expanded to perform a variety of conformance tests on release candidates. For example, if some package accidentally provided a file named `/usr/lib` most hosts would cease to function shortly after that file was deployed. The structure policy is the first of several policies aimed at release conformance tests.

Database Meta-Information

- `db/databases`. The `databases` configuration file specifies the actual location of all of the above databases. The paths shown above are the ones defined in the CERT SAFARI repository, but any of these databases can be renamed or relocated. In the current implementation, we don't recommend this as it has not been tested. It is possible that some lingering path names may be embedded in code somewhere.
- `db/hashes`. The `hashes` configuration file contains the official MD5 hashes of every database defined in `db/database` as of the end of the last database update transaction. These hashes are used as a high performance mechanism for database consistency checking by the deployment mechanism (since it cannot actually lock the database).

Programs

SAFARI includes a number of programs and scripts. Some are provided for repository administrators, some for collection managers, and others for host managers. Even though end-users should be completely unaware of the existence of SAFARI, one tool is provided that may be useful to them.

safari

The *safari* command is the principle tool for repository administrators. Nearly all of its functions are aimed at administration and maintenance of a SAFARI repository. The *safari* command presents a simple object-oriented view of the repository. The repository is divided into several classes of objects and a set of operations that apply to each class. Each class and its applicable operations are presented below.

Creation

```
safari {collection | cluster} \
  create name \
  -manager principal... \
  [-quota blocks]
```

This subcommand creates a new SU, either a collection or a cluster, assigning it a unique new name and one or more principals as managers for the SU. Assignment of multiple managers is not discouraged, but neither is it facilitated in any way by SAFARI. Multiple managers must coordinate their activities regarding the SU. It is common to assign two managers for each SU, one as primary and one as backup. The primary manager has the responsibility to keep the backup manager informed enough so that the backup manager can act in absence of the primary manager (usually this means fixing a bug and releasing a new DSU).

```
safari {package | image} \
     create name \
     -platform name \
     [-revision integer] \
     [-quota blocks]
```

This subcommand creates a new DSU, either a package or an image, which is targeted at a particular platform. When managing multiple platforms at different times, it is sometimes impossible for *safari* to correctly choose a revision number since *safari* cannot know what the manager intends the DSU to contain. Therefore, an explicit revision number can be specified.

Examination

```
safari {collection | cluster} \
     list regexp
```

This subcommand lists SU's by matching a (PERL) regular expression against the names of all known SU's.

```
safari {package | image} \
     list [name] \
     [-platform name] \
     [-status tag] [-managers] \
     [-sizes]
```

This subcommand lists DSU's by matching a regular expression against the names of all known DSU's, by target platform, by status, or some combination of the above.

Preparation for Release

```
safari {package | image} \
     prepare name \
     -platform name
```

This subcommand produces the DSU's `depot/MANIFEST` file, listing the complete contents of the DSU. Cryptographic hashes (MD5) are used to describe the contents of all files. The manifest does not describe itself. The manifest is constructed in

such a way as to include the intended modes and ownerships of files and directories, rather than their current modes and ownerships. The manifest is used in the release process (below) to actually change the modes and ownerships before releasing the DSU. This allows non-privileged managers to ask for modes and ownerships they would not normally be allowed to set themselves.

```
safari {package | image} \
     seal name \
     -platform name \
     [-user pgpuser]
```

This subcommand produces the DSU's `depot/AUTHORITY` file, which is a certificate of authenticity for the DSU. The certificate includes a cryptographic hash (MD5) of the manifest, as well as other meta-information, and is digitally signed. In this way, the entire DSU is sealed under the authority of the manager and cannot be altered from the configuration specified in the manifest file without detection. All prospective users of the DSU can see who certified the DSU's contents and decide for themselves what level of trust to accord it.

Validation

```
safari {package | image} \
     check name \
     -platform name \
     [-revision integer]
```

This subcommand validates the content of a DSU. It checks the digital signature on the DSU certificate (`depot/AUTHORITY`), uses the certificate to check the contents of the manifest (`depot/MANIFEST`), and uses the manifest to check the contents of the DSU. All checks must match exactly for validation to succeed. Any deviation is reported.

Release

```
safari {package | image} \
     release name \
     -platform name \
     -status tag
```

This subcommand transforms an unreleased DSU into its released form, mounts the AFS volume in its correct location, and updates the repository database regarding the status change for that DSU. The transformation process includes validation of the contents of the DSU (except for modes and ownerships), alteration of modes and ownerships to match the manifest, and alteration of AFS ACL's according to the protection policy in effect for the DSU (global policy + DSU policy).

It is expected that the status assigned to a newly released DSU represents semantics of minimal trust. For example, we use status 'alpha' to indicate that no one except the SU manager should trust the DSU's contents. This prevents accidental deployment of newly released DSU's onto production quality hosts.

Lifecycle Management

```
safari {package | image} \
     setstatus name \
     -platform name \
     -revision integer \
     -status tag
```

This subcommand changes the status label associated with an already released DSU. This is the mechanism that a manager uses to signal his intent that others can begin to trust the contents of the DSU. Multiple levels of trust may be appropriate at a given site. In our repository, we use status beta to mean that a DSU can be deployed onto certain hosts which have been designated for end user functional testing. This deployment automatically occurs, announcements are sent out, and users try out the newly released software on the beta hosts. After the manager receives feedback, he can decide to promote the DSU into full production use (status **gamma** in our repository) or out of service (status **deprecated** in our repository) to be replaced by a new revision.

Repository Management

```
safari repository lock
safari repository unlock
```

These subcommand can be used to manage the reservation of the repository database for extended periods of time. The typical use of this function is to perform several related actions that should be seen by repository users all together, such as releasing multiple interdependent packages at one time. If a host in the process of being constructed were to select some, but not all of the interdependent set of packages for deployment, a variety of inconsistency-related failures may occur (e.g., shared library skew).

```
safari repository check
```

This subcommand is a maintenance function, typically run regularly by a SAFARI administrator to ensure that the repository database is syntactically correct and that the database actually reflects what is installed in the repository, and vice versa. In addition to detecting several different kinds of inconsistencies, this subcommand can propose (for certain kinds of inconsistencies) corrective actions to be taken by the administrator. Inconsistencies between the repository and the repository database almost always occur as a result of AFS administrative failures of some sort, lost AFS volumes being the most common.

```
safari repository addkey
safari repository listkey
```

These subcommands are used to manage PGP public keys stored in the repository database. Every administrator and manager must have a PGP public key stored in the database.

```
safari repository showpolicy
```

This subcommand can be used to print human readable forms of a variety of repository policies (described above).

pf

The *pf* command is designed to help the collection manager build binaries for multiple platforms from one set of sources with minimal impact on uniplatform build procedures. The *pf* command is an abbreviation for platform; it is so short because it is used often enough that brevity is valuable. The first and foremost task of *pf* is to determine the name of the target platform for the build process. If *pf* is invoked with no arguments, it simply prints the target platform name on stdout. This is sometimes useful in scripts and makefiles. Since it requires the `platform.map` file in the collection's or cluster's `META` area to determine the build target platform from the actual platform of the build host, it does not function outside of a collection or cluster.

Pf provides multi-platform build assistance in two ways: platform-specific parallel trees of symbolic links and shell command wrapping. The parallel trees of symbolic links are managed by two command line options: `--update` and `--clean`, typically used together.

The *pf* command also provides multi-platform installation assistance hiding the installation location from the collection manager. This is accomplished via the `--install` and `--purge` options.

The basic syntax of the *pf* command is:

```
pf [--clean][--update] \
      [--purge][--install] \
      [shell-command [shell-args]...]
```

The `--clean` and `--update` options are concerned with maintaining the platform specific trees of symbolic links in the cluster's or collection's `OBJ` area.

- `--clean` or `-c`: The `--clean` option specifies that any dangling symbolic links in the symbolic links tree are to be removed. Any empty directories are also removed.
- `--update` or `-u`: The `--update` option specifies that before executing any shell command specified (see below), the platform specific tree of symbolic links located in the collection's or cluster's `OBJ` tree should be updated to match the current directory and all of its subdirectories. Any missing or incorrect symbolic links in the symbolic link tree are corrected to point to their corresponding file in the `build` tree. Any missing directories are also created.

When `--clean` and `--update` are both specified, cleaning occurs before updating.

The `--purge` and `--install` options are concerned with constructing the platform specific packages or images mounted in the cluster's or collection's `DEST` area.

- `--purge` or `-P`: The `--purge` option simply removes all contents of the platform specific package or image mounted in the `DEST` area. This option is almost always used in combination with the `--install` option to accomplish a completely clean installation into a package or image.
- `--install` or `-i`: The `--install` option simply defines the `DESTDIR` environment variable to the root of the platform specific target package or image mounted in the `DEST` area. Many makefiles already expect a variable defined in this manner, making its use completely transparent. For those that don't use `DESTDIR`, trivial changes in the makefiles are required to support this convention. For example, in software constructed using a recent version of *autoconf* [19], one can define the prefix to support `DESTDIR` in the following way:

```
./configure \
  --prefix='$(DESTDIR)/usr/whatever'
```

The *pf* command is also used to execute every command in the build procedure. After processing the above options, if there is a shell command and optional arguments remaining on the command line, *pf* will change directory to the parallel directory in the symbolic link tree in the `OBJ` area and the execute the specified command via *execve*(2) [20]. This is easier to see by example. The following example assumes a normal autoconf managed piece of software named `fsf.org/something/1.0`:

```
$ cd $REPO_ROOT/collection/fsf.org/\
something/1.0/build/something-1.0
$ pf -cu ./configure \
    --prefix='$(DESTDIR)/usr/local'
$ pf gmake
$ pf -Pi gmake install
```

Using this exact same set of commands, one can build this collection for every platform one wishes (assuming the software was written to support the target platforms). As one can see, this is a minimal alteration from the normal uni-platform build procedure.

ft

The *ft* (short for filetree) command is also aimed at simplifying the life of the collection manager. While building Internet-available software from source is desirable, little commercial software is delivered in source form. The *ft* command is designed to help in dealing with software delivered in binary-only form. In particular, it is helpful when such software comes with an obscure installation procedure that is difficult or impossible to coerce into installing properly into the `DEST` area of a collection.

The theory of operation of the *ft* command is simple. First, record the state of a given host's filesystems. Second, install the software in question on that

host using its documented installation procedure. Third, compare the before and after installation states of the host's filesystems and create a package that when deployed via SAFARI will result in the same changes to the host that the software's installation procedure made.

parcel

The *parcel* command is responsible for deployment of DSU's onto individual hosts. A special SAFARI configuration cluster is created for each host. The manager of a host cluster is the host manager. A host cluster defines the complete configuration of the host. It includes any files that uniquely identify the host (e.g., `/etc/hostname.le0` on Solaris) as well as a software configuration that specifies what DSU's are to be deployed onto the host.

The *parcel* command uses the list of DSU's to construct a virtual configuration of the host and then takes whatever steps are necessary to make the host comply with that configuration. *Parcel* supports a variety of mechanisms for abstractly selecting DSU's without having to precisely name each one. Examples include, selection by platform, by status, by revision number, by package family, and by image family. The goal of these mechanisms is to maximize the lifetime of a particular configuration specification. This means that once a host manager has expressed his intentions for the host, *parcel* will automatically keep that host synchronized with the DSU's in the repository.

For example, once a host manager has said "I want the latest revision of GNU Emacs that is certified as production quality," *parcel* will automatically make sure that when a new revision of GNU Emacs appears with the correct status, it will be deployed to the host, without the host manager having to say that he wants the new revision.

The host image from the host cluster is not really special in any way, other than it being the first image that parcel processes.

Parcel also offers the capability to choose whether DSU files to be deployed into host directories are copied locally or referenced in the repository via symbolic links. The ability to reference files in DSU's in the repository allows the host manager to effectively multiply the size of his local disk to accommodate host configurations that would normally require more disk space that is available. This can also be a very powerful tool for SU managers (who are typically host managers for their own desktop workstations). Deploying an **alpha** status DSU for testing via symbolic links is extremely fast, and reduces the develop-deploy-test cycle time. *Parcel* can even be instructed to deploy an unreleased DSU for pre-alpha testing by the SU manager.

Parcel is designed to be able to build a host from an effectively empty disk. For example, when we build Solaris hosts, we boot the host via the network, format, partition, and mount the local disk, and then run *parcel* to install the software. Everything installed on the host comes from the repository. This means that recovery from catastrophic failure is the same as initial installation.

Parcel can be run as often as desired to update a host from its repository. Since extensive changes to the host may result, it is typically wise to run *parcel* only at shutdown or boot time. *Parcel* can also simulate an update so that the host manager can decide if an update is required and if he wishes to risk updating an operating host.

Cooperating Repositories

We have thus far engaged in considerable speculation regarding the practicality of multiple cooperating repositories. These speculations include:

- Demonstrating to vendors the value of a canonical distribution mechanism.
- Sharing of technical talent between repositories (e.g., operating systems are hard to put into a repository, but only one repository needs to contribute an operating system; then all other repositories can use it).
- Open source software can be packaged once by its developers or delegated builders and everyone can choose to use it or not based on their trust of the developers or builders.
- Local repositories can be used to cache DSU's from around the world, making a network of high performance deployment servers.

We don't propose that SAFARI is the canonical deployment vehicle. We only seek to show the benefits from having such a world-wide mechanism.

Availability

The CERT Coordination Center offers a web site, http://www.cert.org/safari, that describes the project in more detail and provides access to the available software components.

Future Work

Future work includes:

- Canonical seekable transportable format for packages and images.
- Managing dependencies between and among collections, packages, clusters, and images.
- Better cryptographic and message digest technology.
- General purpose configuration file delta processor, a la patch [14].
- Local file system access control lists support.
- Release conformance validation (structure and protections).
- Client-server administration.
- Taking snapshots of `build` and `OBJ` areas at package/image release time.
- Support for multiple collaborating repositories.

- Support for OSF's DCE/DFS.
- Supporting Windows NT 5 and CIFS, both as a server, to house a repository, and as a client.

Acknowledgments

SAFARI is built upon ideas that emerged from eight years of work at Carnegie Mellon University and the University of Pittsburgh. Between those two institutions, over 1,000 hosts are being managed by the precursors of SAFARI. The number of people who have significantly contributed to this work, directly and indirectly, is simply too great to enumerate here.

Author Information

Bill Fithen is a Senior Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Bill is the principal architect for several development projects, including SAFARI. Bill previously worked as the Manager of the University Data Network at the University of Pittsburgh. Bill earned a BS in Applied Physics and a MS in Computer Science, both from Louisiana Tech University. Reach him at <wlf@cert.org>.

Steve Kalinowski is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Steve is the leader of the team providing computing infrastructure services within the program. Steve previously worked on distributed computing environments as the UNIX Services Coordinator at the University of Pittsburgh, on the Electra electronics troubleshooting product at Applied Diagnostics, and on computer-integrated manufacturing systems at Cimflex Teknowledge, all in Pittsburgh, Pennsylvania, USA. Steve earned a BS in Computer Science from the University of Pittsburgh. Reach him at <ski@cert.org>.

Jeff Carpenter is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Jeff spends most of his time leading a team that provides technical assistance to Internet sites that have computer security issues or have experienced a security compromise. Before joining CERT, Jeff was a systems analyst at the University of Pittsburgh working in the computer center designing the university's distributed UNIX environment. Jeff earned a BS in Computer Science from the University of Pittsburgh. Reach him at <jjc@cert.org>.

Jed Pickel is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Jed is a member of the team that provides technical assistance to Internet sites that have experienced a security compromise and is also a member of the team that coordinates responses, including CERT Advisories, to vulnerability reports. Jed earned a BS in Electrical Engineering from the University of California, San Diego. Reach him at <jpickel@cert.org>.

References

[1] Sun Microsystems, Inc. 1997. *Application Packaging Developer's Guide* [online]. http://docs.sun.com/ab2/coll.45.4/PACKINSTALL/@Ab2TocView .

[2] UNIX Systems Laboratories, Inc. 1990. *UNIX system V release 4 Programmer's Guide: System Services and Application Packaging Tools.* Englewood Cliffs, NJ: UNIX Press, Prentice-Hall.

[3] Bailey, Edward. 1997. *Maximum RPM* [online]. http://www.rpm.org/maximum-rpm.ps.gz .

[4] IBM. 1998. *AIX version 4.3 general programming concept: packaging software for installation]* [online]. http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/pkging_sw4_install.htm .

[5] Colyer, W. and Wong, W. 1992. *Depot: A Tool For Managing Software environments* [online]. http://andrew2.andrew.cmu.edu/depot/depot-lisaVI-paper.html.

[6] Fithen, B. 1995. *The Image/Collection Environment For Distributed Multi-platform Software Development and System Configuration Management: A Tool for Managing Software Environments* [online]. http://www.pitt.edu/HOME/Org/CIS-SN/SDR/public/ICE/index.html .

[7] Mui, L. and Pearce, E. 1993. *Volume 8 – X Window System Administrator's Guide*. Cambridge, MA: O'Reilly & Associates, pp. 196-197.

[8] Transarc Corporation. 1998. *The AFS file System in Distributed Computing Environments* [online]. http://www.transarc.com/dfs/public/www/htdocs/.hosts/external/Product/EFS/AFS/afsoverview.html .

[9] Seagate Software. 1998. *WinINSTALL* [online]. http://www.seagatesoftware.com/wininstall .

[10] Garfinkel, S. 1994. *PGP: Pretty Good Privacy*. Cambridge, MA: O'Reilly & Associates.

[11] Rivest, R. 1992. *RFC 1321: The MD5 Message-digest Algorithm* [online]. http://info.internet.isi.edu/in-notes/rfc/files/rfc1321.txt .

[12] Tichy, W. F. 1985. *RCS: A System for Version Control*. Software Practice and Experience. vol. 15, no. 7, pp. 637-654.

[13] Silverberg, I. 1992. *Source File Management With SCCS*. Prentice-Hall ECS Professional.

[14] Wall, L., et al. 1997. ftp://prep.ai.mit.edu/pub/gnu/patch-2.5.tar.gz.

[15] Johnson, M. K. 1997. *Linux Information Sheet: Introduction to linux* [online]. http://sunsite.unc.edu/LDP/HOWTO/INFO-SHEET-1.html .

[16] Colyer, W. and Wong, W. 1992. *Depot: A Tool for Managing Software Environments* [online]. http://andrew2.andrew.cmu.edu/depot/depot-lisaVI-paper.html#HDR6 .

[17] Internet Engineering Task Force & Braden, R., ed. 1989. *RFC 1123: Requirements for Internet Hosts – Application and Support* [online]. http://info.internet.isi.edu/in-notes/rfc/files/rfc1123.txt .

[18] Josey, A., ed., 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Englewood Clifs, NJ: UNIX Press, Prentice-Hall, p. 525.

[19] Free Software Foundation. 1996. ftp://prep.ai.mit.edu/pub/gnu/autoconf-2.12.tar.gz.

[20] Lewine, D. 1991. *POSIX Programmer's Guide*. Cambridge, MA: O'Reilly and Associates. pp. 262-263.

[21] Defert, P., et al. 1990. "Automated Management of an Heterogeneous Distributed Production Environment." *Proceedings of LISA*.

[22] Manheimer, K., et al. 1990. "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries." *Proceedings of LISA.*