Cget, Cput, and Stage - Safe File Transport Tools for the Internet

Bill Cheswick
Bell Laboratories, Lucent Technologies
ches@bell-labs.com

Abstract

Cget, *cput*, and *stage* are three simple programs that implement authenticated or encrypted file transfers on the Internet. *Cget* and *cput* read and write files to a remote host, and *stage* ensures that a remote directory accurately mirrors a local master directory.

These routines use private key cryptography for authentication and privacy between pairs of secured hosts. They are simple, paranoid Unix tools that can be used to support systems that operate in a hostile environment.

1 Introduction

A host can be made reasonably resistant to compromise from the Internet if it isn't running any dangerous network services. Most hosts don't come from the manufacturer configured in this manner—they have to be stripped of all of their network services by hand. Then only the desired services are installed. If the network services are secure (a big "if"), then machines are much harder to breech.

Two such secure hosts can exchange files and remain reasonably secure if there is a safe file transport service available. This file transfer service would have to be resistant to all the popular attacks found on the Internet today, including the recent IP spoofing[12] and TCP hijacking[10]. If the data is sensitive, then the service would also have to ensure privacy.

Such a service is needed often these days. In particular, publicly-available Internet services like FTP and http are often provided by hosts running on the dirty side of the firewall, usually in a DMZ. How can we administer such hosts, perhaps from the relative safety of another host behind a firewall? How do we install new programs, or install new content?

The standard out-of-the-box network services do not provide this security. In fact, they have had a history of jeopardizing their servers. FTP has been a constant source of trouble: its passwords are easy to sniff, its protocol has various flaws, and various servers have had security holes [5, CA-88:01, CA-92:09, CA-93:06, CA-94:07, CA-94:08, CA-95:16]. *Rcp* relies on address-based authentication, and the integrity of the Domain Name System, which is easy to fool [4, 16]. It is also susceptible to IP spoofing attacks [14]. NFS has a variety of weaknesses: it can be fooled with address spoofing, root handles can be sniffed or guessed, and it relies on RPC services that have security weaknesses of their own.

These services are frequent targets for successful hackers. Still, they are often employed because they are widely available, and most developers are familiar with them—even when they are clearly unsuited for the job. Developers don't have time to build new tools: the frenzy of Internet hype and growth can leave management focused on time-to-market issues. Security is often left to the last minute, and patched in after the design is finished.

Marcus Ranum and I faced these problems in the fall of 1995. We encountered *ad hoc* solutions using standard, dangerous Internet services. For example, billing data was transferred in the clear using FTP. Developers cast about for ways to transfer configuration files and other important data. The standard tools they used were jeopardizing some very important hosts.

We wanted a very simple solution, in the tradition of small Unix tools. This is not a very tall order: it requires a simple file transfer program running some strong cryptographic or authentication routines, and a shared secret key.

We were only setting up a few point-to-point links, so it was easy to distribute a secret key to each end of a connection. We didn't want an authentication server or need public key cryptography. We envisioned that a pair of simple programs, plus a key file and a configuration file, were all that were needed. A simple implementation meant that more people were likely to understand and use the software, even if they were in a hurry to make a deadline.

There were a variety of possible off-the-shelf solutions available at the time: our problems were not new. Most of them appeared to have adequate security, but none were as simple as we desired. (Some of these are discussed in section 9.)

We have a better chance of avoiding security bugs if the programs are small and simple.

Marcus and I built three programs. Each was as simple as possible, and written with all the minimalism and paranoia we could muster. *Cget* reads a single file from a remote server, and *cput* writes a file back. *Cget* and *cput* are quite primitive: they do not support file deletion, directory creation, or program execution. (They were originally named *get* and *put*, but that clashed with SCCS's routines of the same name.) *Stage* mirrors a master directory to a slave host. Files and directories are created, deleted, and downloaded as needed. It is ideal for updating an external web server or FTP archive from an internal staging host. *Unstage*, a recent addition, works in reverse, updating a local copy of a remote master directory. It can be used to suck logs from a server, or perhaps get a distribution from a master tree.

Our initial cryptographic routines used DES encryption. For many applications, this is overkill. Most transfers are not secret—FTP and Web data are generally intended for public viewing.

If we kept strong encryption, it would make the software release difficult and unlikely.

So the crypto layer has been rewritten—the interface and code are cleaner than the first version. The new crypto routines use only HMAC keyed message digests to protect the conversation. If my authentication protocol is OK (always a big 'if'), an eavesdropper may watch or interrupt a session, but cannot modify or replay a session without detection.

The next section describes the authentication protocol and some cryptographic issues. Section 3 describes the user interface to the cryptographic protocols. Server design issues are explored in section 4. The *stage* service is discussed in some detail in Section 5. Section 6 has a couple of applications for these programs, including the confinement of an arbitrary TCP service. Section 7 covers vulnerabilities, and section 8 has some performance figures. A tiny sample of the related work is discussed in section 9. Section 10 describes some enhancements and limitations to these routines, and availability information is in section 11. Appendix A describes the staging protocol.

2 The Authentication Protocol

The client and server exchange messages in SSL format, although we do not use SSL's complex key setup. Each message contains a two byte length field, the payload, and a 16 byte binary digest.

I use the HMAC[3] digest with MD5, which appears to be headed for general usage on the Internet. An HMAC digest of a message M using key k is shown as

$$[M]_k$$

The client and server share a secret key, K_{ss}. The protocol uses challenges in both directions to derive session keys. The challenges are sixteen random bytes encoded in pairs of hex digits. The session key for each end is derived from K_{ss} and the challenge from the other end:

$$\begin{aligned} K_s &: \quad [C_c]_{K_{ss}} \\ K_c &: \quad [C_s]_{K_{ss}} \end{aligned}$$

The server writes with K_s, the client with K_c. The initial exchange between client C and server S proceeds as follows:

$$\begin{array}{lll} \text{Message 1} & C \rightarrow S: & N, C_c, [N, C_c, S_c]_0 \\ \text{Message 2} & S \rightarrow C: & C_s, [C_s, S_s]_{K_s} \\ \text{Message 3} & C \rightarrow S: & \text{“}\mathtt{OK}\text{”}, [\text{“}\mathtt{OK}\text{”}, S_c]_{K_c} \end{array}$$

Here N is a service name (see section 4.1), and S_c and S_s are sequence numbers, C_c is the client's challenge and C_s is the server's challenge. The sequence numbers are four bytes long. S_c starts at zero, and S_s starts at 2^{31}.

Message 1 delivers the client's challenge. It uses the key 0, which helps us detect casual probes of the service. Message 2 proves to the client that the server is using the new challenge, has the secret key, and provides the server's challenge. Message 3 has a trivial payload, but proves to the server that the client is using the fresh challenge and the secret key.

The session keys are used to prevent replay attacks using messages from previous sessions. If we simply keyed our digests with the K_{ss}, an attacker could replay a previous session, perhaps replacing a new file with some older one. Each end uses a different session key so a message can't be played back to its originator. Similarly, the sequence numbers prevent replays of earlier messages in the same session. Without these, an attacker might hijack the TCP session, and replay earlier messages. The sequence numbers differentiate the hashes from each end of the conversation.

Although the client can force the server to use a specific challenge, and therefore the same key, it can't finish the protocol initialization without using the server's fresh challenge and the secret key.

A man-in-the-middle can't change a message without detection: he cannot determine the session key without the secret key, and he cannot obtain the right answers from an additional connection to the server, since the session key will be different.

This is a simple protocol, and it looks like it ought to do the job, but I am not a cryptographer, and history teaches that it is hard to get cryptographic protocols right. Though I don't see how this protocol can be abused, I'd feel better if each session key were based on both challenges.

2.1 Administrative concerns

Protocol setup can interact with administrative concerns. Our original protocol was terse and unhelpful when, say, one end had the wrong key. The obscurity may have added some security, but it sure didn't help our users who were trying to set up the service.

The setup may fail for a number of reasons: the key is wrong or missing, the key file is not readable, the service is non-existent, etc. Each of these errors means that the server cannot return the correct digest in its first message. If the digest is wrong during protocol setup, the client checks the payload for the string "`remote reported an error:`". If present, the rest of the message is an error message from the server describing the problem.

Until the exchange is complete, the protocol is subject to attack. In particular, it is possible for an attacker to inject a false error message during this setup phase.

2.2 Keys

Secret keys are appropriate here: we don't need public key cryptography. The usual complaint about secret keys is the distribution problem—how do we move them around securely? For us, these programs are only employed in a handful of hosts, involving perhaps a dozen services and their keys.

Our keys are printed in hex bytes or base 64 encoding, both human-readable. They can be distributed by hand or over the phone when the service is installed. We type ours in at the consoles of the hosts involved.

This might not scale to a large setup, but one could imagine an ISP allowing a thousand customers to use *stage* to update a thousand separate web directories. It would not be much harder to distribute a binary key than a password that a user has to remember, and the client wouldn't need an account on the web server. (This is always a good feature: users are annoying, and tend to disrupt security arrangements.)

The secret keys are generated by a program named *makekey*. Its keys, and the protocol's session challenges, are generated with *truerand*[11]. (*Truerand* runs a counter in a tight CPU loop while waiting for an alarm timeout some milliseconds later. The bottom two or three bits of the counter are considered random.)

We use full random binary keys: there are no passwords that a user must remember.

3 Interface Routines

To use the crypto routines, the client uses the following code:

```
fd = tcpconnect(host, port);
ep = start_client_crypto(fd, key,
         srv_nam);
if (ep != 0) {
    /* error */
}
n = cread(fd, buf, sizeof(buf));
n = cwrite(fd, &gt, ngt);
if (n < 0) {
   cperror("writing gunilla table");
   exit(1);
}
```

and the server uses

```
fd = bindto(service_port);
ep = start_server_crypto(fd);
if (ep) {
    perror(ep);
    exit(1);
}
n = cwrite(fd, "hello", 6);
n = cread(fd, buf, sizeof(buf));
...
```

(Some error processing is simplified for clarity.) *cread* and *cwrite* are analogous to the standard I/O routines. *Cperror* reports standard or cryptographic errors. This protocol preserves message delimiters: each *cread* will return only the bytes sent by the corresponding write. The maximum message size is $2^{16} - 1$.

The server must supply a routine named *setservice*, which obtains the secret key or returns an error message.

4 Services, servers, and server trust

Our server programs assume that they have no friends. For example, they shed privilege early to minimize the code we must trust. The server programs obtain the key for the calling host, *chroot* to a target directory, and change their user id and group to some less-privileged account. All inputs from the client are carefully checked for pathological values.

We have two servers: *getd* for *cget* and *cput*, and *staged* for *stage* and *unstage*. They operate on different TCP ports, and are called by *inetd*.

4.1 Service names

Originally, these routines were keyed to a host's numeric IP address. A single client host could *cget*/*cput* to one area of a server, and *stage* to another. If others needed to *stage* to the same server, they would have to connect from a different client host. The IP address of the caller was used to select the proper key and service configuration on the server. Though it provided only slight security, the connection had to originate from that IP address. The secret key provides the real security.

This approach worked well for simple setups, but the one-service-per-client limitation became inconvenient as the services were used more. Also, a traveling host couldn't access a fixed server, because the client's IP address was unpredictable. We could provide additional services on different ports on the server, but this is awkward.

The new authentication protocol includes a service name. The key and other information are based on this name. A single client host can have a number of services on a given server. Different users can have different access to the same server, all controlled at the client end by read access to the relevant key file. This trust model shouldn't be pushed too far: Unix *root* accounts are generally not very resistant to user attack. A serving host should extend about the same level of trust to all services from a given client.

4.2 Trusting the Server Software

So far, I have assumed that

1. we control the server machine entirely, and
2. we trust the server code until it drops privileges.

Neither assumption may be true if we would like to persuade some one else (say, an ISP) to run our servers on their host. They would be more willing to run our software if we don't need *root* permission, and perhaps even more willing if they can contain our software with their own *chroot*.

The problem is that *chroot* requires *root* permission, and it is hard to change the user id safely with standard shell commands after the *chroot*. We have to include a *setuid* program within the software "jail" to change from user *root*, and the user with access to this directory might find a way to disable this program.

The *chroot* program needs an option to set the UID of the executing program. I wrote a trivial version of *chroot* named *jail* to do this. I can give this tiny program to the ISP. It's only a few lines long: they can examine it, trust it, and confine our servers nicely with it.

We have some problems when the server is enclosed in the jail. How does the program obtain its key, unless the key is stored within the jail itself? We may wish to keep the key secret from the user. If the key is stored in the jail, a remote user may have undesirable read access to it. It could be piped in through a file descriptor opened by *jail*, but that's a bit awkward to set up. The key could also be a parameter to the server program, but that can make it visible to other users on the serving host through the *ps* command.

It would also be nice to let the jailed server issue *syslog* messages. This requires possibly-dangerous special files inside the jail, or some mechanism for *jail* to perform the *openlog* and pass the file descriptor to the server in the jail.

I am not satisfied with the solutions to either of these problems. *Chroot* is a good start, but Unix lacks adequate confinement primitives.

5 Stage

Once the crypto routines were working for *cget* and *cput*, *stage* was an obvious application. *Stage* runs through a local master directory, comparing each file and directory with the contents of the remote slave directory. The master directory for a service is identified by an entry for that service in configuration file on the client, usually in `/usr/local/etc/stage.conf`.

Like *cget*/*cput*, *stage* uses the service name to look up the appropriate key. *Staged* also uses the service name to determine the target directory, user id, and file permission mask. It uses *chroot* to confine itself to the target directory. This is good: it is somewhat more complex than *getd*, and therefore more likely to have bugs. It does check its input from the client carefully: strings can't be too long, "`..`" is not allowed in path specifications, etc.

Stage will update a file if it has changed. A file is considered changed if its modification date or length are different, or if its MD5 checksum is different. The checksum is time consuming—an option suppresses this check. When a file is copied over, its modification date is set to match the master copy, if the operating system on the server allows it.

Ordinary users can use *stage* to update all or some portion of the master directory. It only takes a few seconds to check and transfer a few megabytes. *Stage* does not exit until the update is complete: there is no queuing mechanism involved. Should the program abort or fail for some reason, it can be rerun to ensure that the directories match.

The user can *stage* a file or a directory. Either must appear under the master directory. If he stages a non-existent path, that path will be deleted on the server, if it exists. Hence:

```
rm -rf foo
stage remote foo
```

will delete a directory or file named `foo` at the other end. There is a subtle distinction here:

```
stage remote *
```

and

```
stage remote .
```

are not the same. The first entry will update all existing files and directories in the current directory. The second will do that, plus delete any remote entries that don't appear locally.

Stage makes no special provisions for special files or soft or hard links. It makes no special provisions for files or directories that change during the staging process.

The user does not have to have read access to the key file: *stage* could be *setuid* to an account or group that has read permission for the key.

Staged serves *stage* requests on the remote server. It consults `/usr/local/etc/staged.conf`, which has one line for each supported service name. Each line contains the service name, the slave directory, and an optional UID and umask. File ownership is not propagated to the server: the slave files are owned by the account that the service is configured for. *Staged* logs all file activities via the *syslog* facility.

The *stage* client uses a little protocol to control the remote server. It is very simple (see Appendix A), a subset of basic file system access primitives. It could be optimized to improve performance.

6 Applications

These routines have been well-received, and are employed in a number of places in AT&T and Lucent. In our lab, *stage* has been providing quick and simple support for `ftp.research.att.com` for over a year, and now supports `ftp.research.bell-labs.com` and Lucent's Web service on `www.lucent.com`. We had to make a proxy version of *stage* for this last service to support it through our older firewalls. It replaced a clunky FTP-based implementation that was unable to delete files on the slave server. *Cput* has been especially useful for moving new binaries to external hosts.

One researcher has been transporting large, extremely sensitive databases over networks having dubious or unknown security. The encrypted version of *cput* is vital for him.

We have also supported our various external dirty Unix hosts with these tools. Automated jobs use *cput* to update mail alias files and internally-generated name server configuration files. *Cget* fetches daily log files so users can monitor access to the FTP directory without requiring logins or more invasive access to the server. Hal Purdy at AT&T Research has ported the encrypting *staged* to Windows 95 to support a roomful of PCs outside the firewall.

6.1 An Example: updating mail alias files

Traditional Unix file system permissions can be used to provide finer control over access to the target directory. For example, we have a slave mail directory on an external server that contains programs and data files. The master host is allowed to overwrite and create the mail alias files, but is not allowed to change the executable files in the slave directory. We could just split them into separate directories, but we are used to the current configuration.

The executable files are owned and writable by account *bin*. The directory and the alias files are owned by *daemon*. This lets the master host update and add new alias files, but it doesn't have write permission for the executable files. If we supplied complete Unix file system semantics, they could delete the executables (since they have write permission on the directory) and create new ones. But *cget* won't delete a file, only replace it. There's one additional protection: we can have the umask in *getd*'s configuration file clear the executable bits in downloaded files.

If we rely on UNIX's built-in file permissions, we reduce the amount of special-case permission-checking software in *getd*. Smaller programs are safer.

6.2 Bob's TCP service

Bob has a service that he would like to offer the world. I don't know much about it. It looks at some files, answers queries on a TCP port, and writes logs. It doesn't need to be *root*, or use any fancy system services or files. It's just a program.

I like Bob, and I trust him (mostly). His service is not only harmless, it appears to be quite worthwhile. He'd like to run it on our external server.

I'd like to help, but I don't want to jeopardize a host that we've taken great pains to secure. I don't want an error on Bob's part to reduce that host's security. How far do we have to trust him?

We can lock Bob's software and files inside a *chroot* environment. He won't be running as *root*, and we won't leave any dangerous programs in his jail, so we are pretty confident that he can't get access to the rest of our file system. We add the following single line to `/etc/inetd.conf`:

```
z3950 stream tcp nowait root
    /sbin/chroot chroot /usr/bob
    /bin/su - bob -c /tree/bin/zsrv
```

We use *chroot* to confine Bob, then *su* to give up *root* privilege, before Bob ever gets to execute an instruction. The security measures are right out there in the open, on the line where an auditor can see and understand them. Unfortunately, `/bin/su` is inside his jail, so we use *jail* instead:

```
z3950 stream tcp nowait root
    /sbin/jail jail -u 99 -g 2 /usr/bob
         /tree/bin/zsrv
```

Jail is just like the *chroot* command with the additional **-u** and **-g** options to set the user id and group.

His program, *zsrv* should be linked with static libraries: it greatly simplifies the setup of the jail.

We split Bob's directory (`/usr/bob`) into two subdirectories: one (`/usr/bob/tree`) that he can stage into, and one (`/usr/bob/log`) that he can *cget* or *unstage* from. He can update his server software and other files under `/tree` at will. He can change the network server, but it's always owned and executed on account *bob*, not *root*, so he is very unlikely to get out of his jail.

What can Bob, or someone who has hacked into Bob's account, do to us? Most of the problems are of the denial-of-service type:

- File System Full. He can fill the partition that holds `/usr/bob/tree`. If this is a concern, we can put him in his own partition. Then it only breaks his program, though it may cause annoying log messages elsewhere.
- Core dumps. These can fall under the file-system-full problem. The *chroot* environment assures that the core dump will go in his directory, not somewhere else.
- CPU hog. If he uses too much of the CPU, we can *nice* him down before he starts.
- Memory Full. He could eat up or thrash memory.
- Open Network Connections. Our jail doesn't stop Bob from opening outgoing network connections. This could be abused in a few ways. In particular, he could try to embarrass us or take advantage of the good name of our serving host. Since these can be spoofed anyway, it shouldn't be an unusual problem.

Bob has been running his Z39.50 service for over a year. Aside from a few core dumps and some occasional configuration changes while companies lurch apart, Bob hasn't been a problem.

7 Vulnerabilities

The administrator is often the source of security problems. It is easy to leave key files around with unintended read permissions or incorrect ownership.

If the server is compromised, the game is lost: our original goal was to protect the server.

If the client is compromised, the intruder can gain access to all the services allowed to that host. The server's paranoia can block further spread of such an attack, limiting the intruder's access to a particular directory on the server. This can be an effective barrier which adds another layer to the depth of security.

If either host is compromised, then so are the keys, since they are stored on files. This alarms some people, but the keys are only as valuable as the host—once the host is compromised, the key is useless any way. We've attempted to limit the extent of such a catastrophe by limiting the trust each host has for the other.

The theft of a key can compromise the privacy of old sessions of the encrypting transport. We don't change keys often.

Session interruption can be a problem. If important security files are transmitted, the user must note that an attacker can abort a transfer. For example, *tcpwrapper* [15] has a list of permitted connections in `/etc/hosts.allow` and a stoplist in `/etc/hosts.deny`. If they are transmitted in that order, there is a time when the `/etc/hosts.allow` is in place without the exceptions. This time window can be enlarged by interfering with the second transfer. An administrator must bear these concerns in mind when moving security-related files. In this case, the `/etc/hosts.deny` file should be transmitted first.

As mentioned before, the protocol's error messages may be subject to social engineering during setup. It's an unlikely attack since the timing is tight and the administrator is likely to be watching both the client and the server during initial configuration anyway.

As always, a network service accessible to the public can always receive denial-of-service attacks. Even if the digest is wrong, enough incoming packets can swamp any service, making it unavailable to its intended users.

8 Performance

We have found that these routines work with acceptable speed. With our first protocol using Eric Young's DES encryption library we could transfer 500 KB/s through the localhost port of an SGI running with 150 MHz CPUs.

In another test, our entire 700 megabyte FTP directory was copied between two fast SGI hosts through a firewall in about 47 minutes. When *stage* was rerun just after this transfer, without any updates, the check took about 330 CPU seconds and a little over 32 minutes. With checksumming turned off, it took six minutes.

We don't expect that an entire large directory will be staged more than about once a day. Users find the instant-update feature handy, and tend to stage the little bits they change quite often. The protocol is not especially efficient: it could be enhanced to speed up the file-checking phase.

9 Related Work

We have reinvented a well-rounded wheel. There are a number of software solutions available, some well-suited to their environments. We want a very light-weight solution. Private keys are fine, without authentication servers or fancy certificates. If the infrastructure is in place for such tools, use them: *Rcp* with Kerberos would be fine, if we were running Kerberos in the first place. But most of our customers don't run Kerberos.

Ssh[17] was fairly new when we did this work. It provides a transport protocol that will probably be quite secure—it is under repair at this writing. In late 1995, *ssh* was a beta release. But for our uses, *ssh* has too many features—it's too large. It offers optional features we don't want, like X11 transport and login facilities. We didn't necessarily want these additional services into our secure hosts. The code is more complicated, and there are more things to misconfigure. (We do use *ssh* in other places: it's a nice package.)

There are a number of mirroring and general software distribution packages available. The best known is *rdist* [9]. *Rdist* typically uses *rcp* or *ssh* for transport. The former is not appropriate, but *ssh* is a good choice for supporting *rdist*, and it has its enthusiastic supporters. There's a lot of mechanism there for our simpler applications. *Rdist* has earned three CERT advisories [6, 7, 8], which also makes us nervous.

Two other transport programs include *filetsf* [13] and *Mirror* [2], a Perl program. *Mirror* uses FTP for transport, *filetsf* uses *lpr/lpd*. Again, we don't want these additional services on our safe machines.

Other file transfer programs have appeared recently, such as *SSLftp*. See [18] for a wide assortment of related tools.

We've seen other batch approaches to these problems. A file can be signed or encrypted with PGP, and transported by FTP or even email. These batch processes are bulky and unsatisfying. They require action by special accounts, often initiated by a polling program run by *cron*. *Stage* provides immediate updates, initiated by the end user.

Lower level encryption, like IP/SEC and IP/V6, offer more-general solutions. We could use various existing tools if these were deployed. Unfortunately, they are not widely available yet. We needed these tools a year ago.

10 Limitations and further work

Although these routines are fairly straightforward, some users have prevailed with some minor enhancements. *Cput* does not have an option to run a program when the transfer is completed—a feature found in some file transport programs. In one application the receiving host must scan the receiving directory with a *cron* job looking for new files to process. These files are large (on the order of a gigabyte) and take a while to transfer. The *cron* job needs to know when the file is available. We set the file permissions to 0000 until the transfer is complete. The same application needed a unique file name on the destination host. An optional string ("%u") in the destination file ensures a unique file name.

These routines make no effort to deal with a file that gets shorter during the transfer. The user should ensure that the source files don't change during transfer.

Stage makes no attempt to lock the target directory. If two people stage to the same part of a target directory at the same time, the results are undefined. *Stage* can also overwrite programs while they are executing, causing core dumps in many versions of Unix. Since it doesn't handle special files or links, it is probably unsuitable for updating a remote root directory.

One could teach *stage* about hard and symbolic links, but it would add a lot of complexity to the program, which doesn't seem to be worth it.

There is no mechanism here for a client user to determine how much disk space is available on his external partition. The easiest solution is to install the master directory on a partition of the same size as the slave's partition. The user can monitor his inside usage. *Stage* makes no special effort to delete external files before installing new ones, so the outside partition could conceivably fill up during an update if it was nearly full.

Some users wanted more control over the update process. *Stage*'s scan can take a long time, particularly if the directory tree has many gigabytes and checksumming is used. These users wrote scripts to create a list of files to update, and *xargs* can feed this list to *stage*. I had to add a parameter to suppress the descent below directories that were not mentioned on the command line, so we wouldn't do more work than these scripts wanted.

Cryptography can be no better than the quality of the keys. It is hard to generate key material with general purpose computers. I rely entirely on *truerand* to get this right. I did generate 10 megabytes of random data from *truerand* (it took several days) and had Eric Grosse, one of our local numerical analysts, run it through a suite of randomness tests. It passed.

There have been some problems reported with a slightly-restricted version of MD5 lately. Perhaps MD5 will fall soon. It is still possible that HMAC using MD5 would still be safe: HMAC frustrates some attacks on its hash primitive. In any case, I will switch to SHA1.

In general, we like these routines the way they are, and are resistant to creeping featurism. We like their simplicity, and their interaction with standard Unix tools.

10.1 The Joint Ventures Problem

Joint ventures often occur between two companies that don't otherwise trust each other. Many such joint ventures only need to share a directory tree. This can reside on a neutral host somewhere. The contents of the directory tree, or the existence venture itself, may be highly proprietary.

The *stage* command offers most of the functionality needed to implement such an arrangement. *Unstage* reverses the file transfer: the remote directory is the master and the local directory is the slave. User's can share their work with these routines.

These two tools lack only a locking mechanism, which would reserve a subdirectory or file. For example, assume that two authors work for separate companies, but need shared access to the source for their book. One could lock a chapter that he is working on, and the other would have only read access to that chapter. The chapter could be staged back and the lock released.

I've tried to come up with some simple mechanism to enforce locking using file system permissions in the master directory, without a satisfactory result. It would be nice to change the owner of a locked directory, but that requires more privilege than I am willing to give the server software.

11 Availability

The early DES versions of these routines are freely available to AT&T and Lucent employees, and may be found on the companies' Intranets [1]. I will not attempt to distribute these.

Marcus has published his original *get* and *put* routines, with the original crypto API but not the DES routines [20].

I expect to have publication clearance for the authentication-only versions for non-commercial use in time for this conference. I am keen to release these routines to the general public. A general release will expose them to public review and possible improvement. Good cryptography and secure programming are hard to do— it is in our corporate interest to run these routines through the wringer.

See [19] for obtaining this software.

12 Acknowledgements

Marcus Ranum wrote the initial versions of *cget* and *cput*. Andrew Hume entrusted many gigabytes of sensitive data to early versions of *cput*, and made several helpful suggestions. Hal Purdy ported *staged* to Windows 95. Lorette Archer, Steve Bellovin, Matt Blaze, John Linderman, Adam Moskowitz, Joann Ordille, and Bob Waldstein gave helpful suggestions and feedback on the software or this paper.

References

[1] `http://netlib.bell-labs.com/1127/ropes/crio.tar.Z`

[2] `ftp://src.doc.ic.ac.uk/packages/mirror/`

[3] Bellare, M., Canetti, R., and Krawczyk, H., *Keyed Hash Functions for Message Authentication*, Advances in Cryptology – CRYPTO 96 Proceedings, Lecture Notes in Computer Science, Springer-Verlag Vol. 1109, N. Koblitz, ed, 1996, pps. 1–15.

[4] Bellovin, Steven M., *Using the Domain Name System for System Break-ins*, Fifth USENIX Security Conference Proceedings, pps. 199–208, June 1995.

[5] Computer Emergency Response Team (CERT). See `ftp://ftp.cert.org/pub/cert_advisories`.

[6] Computer Emergency Response Team (CERT), "/usr/ucb/rdist Vulnerability", CA-91:20, Oct. 1991. (superseded)

[7] Computer Emergency Response Team (CERT), "SunOS /usr/ucb/rdist Vulnerability", CA-94:04, Mar. 1994. (superseded)

[8] Computer Emergency Response Team (CERT), "Vulnerability in rdist", CA-96.14, Aug. 1996.

[9] Cooper, Michael A., *Overhauling Rdist for the '90s*, Proceedings of the Sixth Systems Administration Conference (LISA VI), pps. 175–188, Long Beach, CA, October, 1992.

[10] Joncheray, Laurent, *A Simple Active Attack Against TCP*, Proceedings of the Fifth Usenix Unix Security Symposium, pps 7–19, Salt Lake City, Utah, June 1995.

[11] Lacy, J.B., Mitchell, D.P., Schell, W.M. *CryptoLib: Cryptography in Software,* UNIX Security Symposium IV Proceedings, USENIX Association, 1993, pps. 1-17.

[12] Morris, Robert, *A Weakness in the 4.2BSD Unix TCP/IP software,* Computing Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, NJ, February 1985.

[13] Sellens, John, *filetsf: A File Transfer Systems Based on lpr/lpd*, Proceedings of the Ninth Systems Administration Conference (LISA IX), pps. 195–212, Monteray, CA, September, 1995.

[14] Shimomura, Tsutomu, and Markoff, J., *Takedown.* Hyperion, 1996.

[15] Venema, Wietse, *TCP WRAPPER: Network Monitoring, Access Control and Booby Traps*, UNIX Security III Symposium, pps. 85–92, Baltimore, MD, September 1992.

[16] Vixie, Paul, *DNS and BIND Security Issues.* Fifth USENIX Security Conference Proceedings, pps. 209–216, June 1995.

[17] Ylonen, Tatu, *SSH - Secure Login Connections Over the Internet*, 6th USENIX Security Symposium, pps 37–42, San Jose, CA, July 1996.

[18] `http://www.cs.hut.fi/ssh/crypto/`

[19] `ftp://ftp.research.bell-labs.com/ches/crio.html`

[20] `http://www.clark.net/pub/mjr/pubs`

Appendix - the Stage Protocol

This is the little protocol that *stage* and *unstage* use to control *staged* and the remote directory. The commands and responses are ASCII fields separated by a single blank and terminated with a zero byte.

send `rm` *fn*
rcv `OK`

> Remove the given file or directory. Everything beneath the directory is removed as well. Returns either "OK", "ENOENT" (not found), or a string describing some other error.

send `st` *fn*
rcv *uid gid mode mtime size*

> Return the stat of a file or directory. The mode is octal, the other values are decimal. "ENOENT" is returned if the file doesn't exist, and other strings contain a displayable error message.

send `cs` *fn*
rcv *md5 checksum*

Return the 32-hex digit MD5 checksum, or an empty string if the file doesn't exist.

send `pu` *fn*
send `++` *user group mode mtime size*
rcv `OK`
send (*size* `bytes`)
rcv `OK`

Push a new file *fn*. It must not already exist. *User* and *group* are alphabetic, and currently ignored. *Mode* is octal, and *mtime* and *size* are decimal. The modification and access times are set to *mtime*, if allowed. Returns "OK" or a printable error message.

send `md` *fn*
send `++` *user group mode mtime size*
rcv `OK`

Create a directory with the given *mode* and *mtime* (if possible). *User*, *group*, and *size* are ignored. Returns "OK" or a printable error message.

send `ls` *fn*
rcv */fn1/fn2/.../fnn//*

Return a list of files in the given directory, separated by slashes and terminated with a double slash. If *fn* isn't a directory, doesn't exist, or is empty, "//" is returned.

send `ge` *fn*
rcv *user group mode mtime size*
rcv (*size* `bytes`)
rcv `OK`

Get a remote file *fn*. Returns "OK" or a printable error message. If OK, return the size of the file in bytes, and the contents of the file.

send `ex`
rcv `OK`

Exit.