Securing the Frisbee Multicast Disk Loader

Robert Ricci and Jonathon Duerig
University of Utah
{ricci,duerig}@cs.utah.edu

ABSTRACT

Shared network testbeds rely on the ability to bring nodes to a known "clean" state, and to allow experimenters to customize the software installed on the nodes assigned to them. This is typically done by replacing the contents of the nodes' disks with a clean disk image. Frisbee is designed for just this purpose. It is a fast, highly scalable system for creating, distributing, and installing disk images. It rapidly and reliably distributes disk images over a LAN to many simultaneous clients, and has proven itself through many years of production use in shared testbed environments.

However, three main security features have been lacking in Frisbee: confidentiality of the image contents, integrity protection, and authentication of the image's source. Frisbee's design and target environment present challenges in providing these features. In this paper, we explore these challenges and present our design and implementation of a secure Frisbee.

1. INTRODUCTION

Frisbee [6] is a fast, highly scalable system for creating, distributing, and installing disk images. It is deployed primarily in network testbeds based on the Emulab testbed management system [10], where it is used to bring nodes to a known "clean" state, and to allow experimenters to install custom software and operating systems on the nodes under their control.

Frisbee was originally developed under a relatively trusting security model: it was assumed that malicious testbed users would be rare or nonexistent, and that since most disk images would be created by testbed administrators and available to all users, there was no need to keep their contents secret. Therefore, its security mechanisms are minimal and oriented toward preventing accidental corruption or exposure of data. Emulab-based testbeds have grown in scale and scope, and are now shared by a large number of experimenters, representing many different institutions and performing many different types of research, including security research. User-created images, which may have sensitive contents, are extremely common. Thus, as the user base has expanded and some experiments have become more sensitive, it has become increasingly important to provide stronger security in Frisbee.

There are three key security features missing from Frisbee's original design: It lacks *confidentiality*, to keep the contents of images secret while stored on the Frisbee server and during distribution. Second, it does not provide *integrity protection* to prevent accidental or malicious corruption of images. Finally, it does not provide *authentication* of the image's creator, the ability to ensure that an image was created by the intended party.

Confidentiality: Initially, Frisbee was used primarily to distribute "official" images, created by the testbed operators and available to all users. Under these conditions, there was little reason to provide confidentiality of image contents. Over time, however, custom user-created images have become more common; the Emulab database currently contains nearly 1,000 such images. User-created images may hold sensitive information such as passwords, proprietary software, or malware, and thus, their contents may need to be kept confidential.

Integrity Protection: By altering an image, either during storage or distribution, an attacker may render it unusable or insert back doors into it. Thus, integrity protection is important to ensure that an image has not been corrupted.

Authentication: Testbed users require some assurance that the images that have been installed on their nodes come from a trustworthy source: the testbed operators, other users in the experimenter's project, or the experimenter themselves. Therefore, the image creators need to have the ability to sign images, and users need the ability to verify those signatures.

In this paper, we examine the threats faced by Frisbee, and present our design and implementation of a secure version of Frisbee. It provides more comprehensive security than comparable network disk imaging systems such as Norton Ghost and Apple NetInstall; or network package installation systems such as ROCKS Avalanche. While some of these systems provide individual properties listed above, ours is the first system of its type to combine all three.

2. FRISBEE BACKGROUND

Frisbee was designed for use in Emulab-based testbeds, which have a shared high-bandwidth, low-latency "control net" over which operations and management tasks, such as remote access and disk imaging, are performed. By operating underneath the filesystem layer, Frisbee ensures a completely clean environment for each experimenter, and can install images for a wide range of operating systems, including Linux, Windows, and FreeBSD [6].

2.1 Image Creation and Compression

Before creating a disk image, the user prepares a source disk by installing their desired operating system and software. To create the image, they reboot the node into a FreeBSD-based operating system which runs from memory, leaving the source disk unmodified. The image is compressed by `imagezip`, Frisbee's custom image creation tool, using filesystem-aware compression which skips unallocated blocks, swap partitions, etc.; the remaining data is compressed using `zlib` [3]. An image is typically created once, then installed many times, so Frisbee does not optimize image creation time.

2.2 Multicast Image Distribution

To install a disk image, the client nodes to be loaded are booted into another memory-based filesystem (MFS), from which the `frisbee` client itself runs. The `frisbee` client receives the image file, decompresses it, and writes it to disk. A separate instance of the Frisbee server runs, using a unique IP multicast group, for each image file being loaded. Client nodes contact the central Emulab server via a protocol called `tmcd` to get the address of the appropriate multicast group.

Frisbee's distribution mechanism uses IP multicast to support a large number of simultaneous clients, with a custom application-level receiver-initiated protocol [8]. In terms of the multicast design space in RFC 2887 [5], the Frisbee design requires only scalability and total reliability, not ordered data delivery or server knowledge of which clients have received data.

2.3 Two-Level Data Segmentation

Frisbee's data transfer protocol requires a custom image file format, and the segmentation used in that format is of key importance to securing image files.

Frisbee has two competing constraints: Small units of data are efficient for re-transmission, allowing out-of-order processing of images, which in turn enables clients to join a session in progress. Large units are more efficient for compression, because such routines optimize their dictionaries based on the distribution of their input data, and thus achieve better compression ratios when given longer input to sample. Disk I/O is also more efficient when done in large blocks.

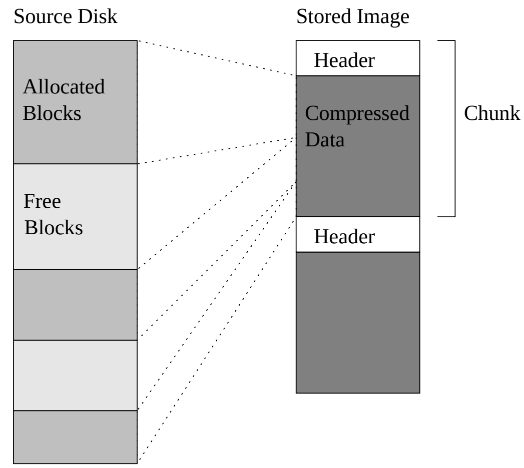

Figure 1: Format of a Frisbee image

The fundamental problem is that Frisbee must follow the principle of Application Level Framing (ALF) [1], yet it has conflicting application requirements. It addresses these conflicting demands by using a two-level segmentation scheme, shown in Figure 1. The image is divided into 1MB *chunks*, composed of 1024 1KB *blocks*.

Each chunk is self-describing, meaning that it contains enough information to be decompressed and written to disk independently of any other chunk. This means that chunks can be received in any order, clients can join a session at a time, and that the transmission and installation of chunks can be pipelined. It also means that, while each chunk contains a header, there is no "global" header for the image.

The unit of request, transmission, and retransmission is the block, which is small enough to fit into a single Ethernet frame. Clients begin downloading a chunk by simply requesting all blocks from the chunk, and selectively re-requesting any missed blocks.

3. DESIGN AND IMPLEMENTATION

Given our security goals and Frisbee's unique properties, we now describe the threats against Frisbee, and the steps we have taken to combat those threats.

3.1 Capabilities of an Attacker

We consider two points at which an image may be attacked: while the image file is stored on the Frisbee server, and while it is transferred during image distribution. Guarding the image once it has been installed on a client is out of the scope of this paper. We begin our discussion of threats by considering the capabilities of a potential attacker.

3.1.1 Storage

In Emulab, stored disk images are protected using simple UNIX filesystem permissions. While this provides some security, it does not provide a sufficient level, as the shared nature of the fileserver on which images are stored leaves it vulnerable to attack. Testbed administrators, or attackers who are able to gain their privileges, can override these permissions, meaning that we cannot rely on filesystem permissions to protect images from all attackers. Thus, we assume that a determined attacker may be able to observe, or even modify, a stored image file.

3.1.2 Distribution

Frisbee's reliance on IP multicast, which has no inherent security of its own, combined with potential attackers on the local network, leaves it vulnerable to eavesdropping and packet injection attacks. It is easy to block Frisbee traffic from entering or exiting the testbed using a firewall, so we assume that attacks come from inside of the testbed. There is no mechanism to limit group membership, so an attacker may join the group to receive a copy of all image distribution traffic. Because Emulab-based testbeds are shared environments, we assume that an attacker may have full control over one or more nodes in the shared control LAN on which an image is being transferred. Likewise, there is no mechanism to limit which hosts may send traffic into the group, meaning that an attacker may inject packets to change the contents of the image being distributed.

While it is possible to address these network vulnerabilities by running Frisbee on a separate network, isolated from other users, this is not the strategy we use in this paper: we do not modify the control network, instead using techniques that protect images during both storage and distribution.

3.2 Confidentiality

Some user-created images contain sensitive information that must be kept confidential, such as passwords or proprietary software. Many Emulab testbeds, including the Utah Emulab and DETER [2], have for some time been used for research on potentially malicious code such as viruses and worms. Much of this code must be protected from leaving the testbed ("exfiltration"). While a major component of this protection is done by controlling what traffic is able to leave the testbed, it is also important that malicious software not "fall into the wrong hands" within the testbed. Since these testbeds are shared, some users are trusted to handle such code and some are not; if an untrustworthy party uses a vulnerability in Frisbee to obtain a copy of malware in a disk image they are not authorized to use, they may leak it out of the testbed.

3.2.1 Threats

Because IP lacks a security mechanism of its own, we assume that any user of the testbed can observe an image file while it is transferred. Similarly, we assume that the contents of an image file may be observed during storage by a determined attacker.

There is another, more subtle threat, introduced by Frisbee's filesystem-aware compression: Frisbee avoids writing to blocks on the disk that will be free in the filesystem it is writing. However, just because blocks are free in an image, it is not necessarily the case that they were free in the old filesystem being overwritten.

3.2.2 Design and Implementation

We provide confidentiality by encrypting the contents of disk images, which provides protection during both storage and distribution. Encryption is done at image creation time by `imagezip`, and decryption is done during image installation time by the `frisbee` client, with the result that the image is encrypted during storage. The Frisbee server simply distributes that encrypted data, doing no encryption or decryption of its own, and does not require the image's encryption keys.

Frisbee requires that each chunk be self-describing. Each chunk contains enough information to be installed by itself, regardless of which other chunks have or have not been received. Thus, the chunk must be the main unit of encryption. The header of each chunk therefore contains its own initialization vector (IV), the non-secret value used to "seed" the encryption algorithm. Any padding required by the encryption algorithm is done at the end of the chunk. We chose to encrypt the contents of each chunk, but not the headers. Chunk headers contain only non-sensitive data, such as the size of the chunk and its sequence within the image as a whole. We do, however, provide integrity protection for the headers. Since a good encryption function produces output that looks random, and is thus not compressible, we compress image data before encrypting it.

One of Frisbee's main concerns is speed of image installation. Thus, it is important to choose a cipher that affords a fast software implementation, leading us to use symmetric-key cryptography. We use the Blowfish cipher [9], which claims to require only 18 CPU cycles per byte encrypted, and is widely considered to be nearly as secure as its slower competitors such as AES.

Cryptography does not, however, solve the threat of data from an experimenter's image leaking into the free blocks of the next experimenter's filesystem. To deal with this threat, Frisbee includes a mode in which, rather than skipping over unallocated blocks, it fills them with zeros to "blank" the data there. This mode is much slower, as it typically involves far more disk I/O. Experimenters can request that this be done between their experiment and the next, or can do it themselves.

3.2.3 Key Distribution

For a Frisbee client to decrypt an image, it must know the encryption key. We have identified, but not yet implemented, two different ways to distribute keys.

The first is to store encryption keys on the testbed's central server and distribute them to the clients through the same mechanism used to communicate multicast group addresses. The security of this method depends on the security of this communication channel. In the case of Emulab's `tmcd`, this channel is quite secure, as discussed further in Section 3.5. This method allows the encryption to be transparent to the user, as key generation and distribution can be done entirely by the testbed. It protects against an attacker who has access to the LAN on which Frisbee runs, but because both key and image are stored by the testbed (albeit on separate servers), it does not defend against an attacker who compromises the testbed itself. In practice, we believe that this level of security will be enough for many disk images.

To handle extremely sensitive data, for which this level of security is insufficient, we suggest a second method in which some key holder, likely the experimenter, provides the encryption key directly to the Frisbee client, through `ssh`, the serial console, or some similar mechanism. This allows experimenters to manage keys themselves, providing an additional layer of security; an attacker compromising the testbed's servers may gain access to the image file, but not the encryption key.

3.3 Integrity Protection

Without integrity protection, a Frisbee client has no assurance that the data it is installing on a disk has not been tampered with. Such tampering could be done while the image file is stored, or during transfer; the nature of IP multicast is such that an attacker may impersonate the Frisbee server at the network layer, and inject corrupted packets into an image transfer.

3.3.1 Threats

We consider two types of attacks.

The first is a simple denial of service attack: the attacker replaces one or more blocks in an image with random data. When decompressed and/or decrypted, random data will be written to disk. The result is that the client node will likely be unable to function properly when booted from the corrupted disk, and may not be able to boot at all. This attack requires no knowledge of the disk contents, and thus is easy to mount and can be used even on encrypted images.

In the second attack, the attacker replaces a targeted portion of the image with specific contents. For example, the targeted portion of the image could contain the root password, enabling the attacker to log into the node as root once it has been booted from the image. A more subtle attacker could replace system services with versions known to contain vulnerabilities, providing a back door into the node. This attack does require knowledge of image contents, and is most likely to be useful on a testbed's "standard" images, whose contents are public knowledge.

3.3.2 Design and Implementation

We use a cryptographic hash to protect both the header and the payload of each chunk. Even though it breaks backwards compatibility with older Frisbee images, use of a hash is mandatory. Otherwise, an attacker can simply modify the header of the relevant chunk to indicate that no hash was used. Thus, while our implementation has a header field that allows selection of a number of different hash algorithms, it requires that a hash be present, which makes it backwards incompatible with the baseline Frisbee. It is likewise important that, if a hash algorithm is broken, that it be disallowed by Frisbee clients: the security of the integrity protection is only as good as the security of the "worst" hash allowed.

A cryptographic hash by itself is not sufficient to prevent malicious tampering, since an attacker can simply re-compute a new hash value after modifying a chunk. To deal with this, we sign all hash values as part of the authentication mechanism described in the next section.

3.4 Authentication

Authenticating the source of an image allows an experimenter to have confidence that the image has come from a trusted entity, such as the testbed operators or another experimenter working on the same project.

3.4.1 Threats

The primary threat addressed by authentication is that of an attacker replacing an image that purports to come from a trusted source with one of their own. The replacement image may give the attacker back doors, or, more generally, may simply fail to operate as the experimenter expects. One version of this attack involves the attacker replacing individual block or chunks in the image file to tamper with an otherwise authentic image.

Frisbee's segmented data format and lack of a global header pose an additional challenge for authentication: proving that all chunks come from the same image. Lacking this, Frisbee is open to "cut and paste" attacks, in which chunks from two legitimate, signed, images are combined. Each chunk would pass the authentication check, but the combined image would be corrupted.

3.4.2 Design and Implementation

In our design, we authenticate chunks by signing their hash values using RSA public-key cryptography. The private key is provided to `imagezip`, and the signature for each chunk is included in that chunk's header. The public key is used by clients at image installation time to verify that the image has been created by a trusted source, and to verify that hashes have not been tampered with.

We use a unique image identifier, placed in the header of each chunk, to ensure that all chunks belong to the same image. Because the identifier is protected from tampering by the hash and signature, it is simple for the client to verify that all chunks received have the same image identifier in them. For this scheme to work properly, uniqueness of identifiers is key, and our use of a UUID [7], a 128-bit random number, makes collisions extremely unlikely.

3.4.3 *Public Key Management*

As with management of encryption keys, we envision two ways of managing public keys, with the choice between them coming down to how much a user is willing to trust the testbed and its central servers.

If a testbed is trusted, we can simply use it as a repository for public keys, using the same communications channel to distribute an image creator's public key that is used to communicate multicast group addresses. This scheme especially makes sense for distributing the key used to sign the testbed's "standard" images: since experimenters must trust the testbed administrators not to provide themselves "backdoors" in these images, trusting them to provide their own public key is not a further stretch.

For experimenters unwilling to trust the testbed's servers, public signing keys can be provided directly to the Frisbee client by the experimenter.

3.5 Secure Key Distribution

Our key distribution models that use the testbed's central servers as a trusted party for key distribution rely on a secure communication channel between those servers and Frisbee clients. In particular, we need three properties: that clients can be assured they are talking to the correct server, that the server can be assured of the identity of the client with which it is communicating, and that the content of the communication is confidential. If the server can be spoofed, a client may be given incorrect signing keys, and if a client can be spoofed, then the server may be tricked into giving decryption keys to the wrong party. If the communication is not confidential, then a third party may be able to eavesdrop on the decryption keys. Emulab's `tmcd` protocol, with appropriate controls on the network over which it runs, satisfies these properties.

`tmcd` uses SSL to encrypt sessions and a well-known SSL certificate to identify the server. It uses IP addresses to identify clients; while IP-based authentication is not generally secure, the Utah Emulab installation uses several techniques to prevent IP spoofing, making `tmcd` a trustworthy communications channel.

First, Emulab's firewall does standard ingress and egress filtering, ensuring that outside hosts cannot impersonate testbed nodes. Since all nodes that use Frisbee are inside of Emulab's network, our further measures focus on preventing these internal nodes from spoofing each others' IP addresses.

Second, all MAC addresses on Emulab's control network are "locked down" by the switches: each Ethernet port is allowed a single MAC address, and Ethernet frames with any other source address are silently dropped. This prevents nodes from spoofing MAC addresses, meaning that in order to impersonate another node, an attacker must mount an attack above the MAC layer.

Between the MAC layer and the IP layer lies the ARP protocol, which translates MAC addresses to IP addresses. This protocol contains no inherent security of its own. Any node in a Layer 2 broadcast domain (i.e., a LAN) can attack it. Emulab addresses this problem on the router that sits between the nodes and servers, where we hard-wire ARP entries for all nodes. While the router will accept packets that have the wrong source MAC address for a given IP address, any replies, such as TCP handshakes, will be returned to the correct node. `tmcd` uses a TCP connection, secured with SSL. An attacker will not be able to complete the TCP handshake or SSL negotiation. Thus, the server can have a high degree of confidence that a client that completes these steps is the legitimate "owner" of the IP address it is using.

3.6 Implementation Details

We have implemented confidentiality, integrity protection, and authentication in `imagezip`, the Frisbee image creation tool, and the `imageunzip` library which is both used as a standalone program and called from the Frisbee client. All cryptography is handled by the OpenSSL EVP [4] libraries.

`imageunzip` makes heavy use of multithreading to overlap tasks that can be done in parallel. Thus, it has separate threads for network I/O, disk I/O, and decompression. Historically, it has been the disk write speed that has been the performance bottleneck. Cryptography will put additional strain on the CPU. Thus, we believe that it will be beneficial to create a fourth thread to perform the cryptographic operations, as decryption and decompression can be pipelined, and multi-core systems are increasingly common. Our current implementation, however, uses a single thread for both CPU-bound tasks.

	Image Creation			Image Installation		
	Time (ms)	Overhead (ms)	Overhead %	Time (ms)	Overhead (ms)	Overhead %
Base	187.9			34.3		
Hash	197.3	9.4	5.0%	43.8	9.5	27.7%
Signed Hash	198.5	10.5	5.6%	44.5	10.2	29.5%
Signed Hash With Encryption	208.8	20.9	11.1%	53.8	19.5	56.8%

Table 1: Extra overhead on image creation and installation. Each row is the average cost per chunk with a different set of modifications enabled.

4. EVALUATION

We now examine the performance overhead of the encryption, decryption, signature, and hash functions used in our implementation. Our changes had no effect on the network distribution of images or actually writing them to disk. We therefore report only CPU time, to isolate the effects of encryption and hashing from disk and network I/O. The content of a chunk has no effect on the extra overhead we incur, so we show the average time required to process a chunk. Our experiments were run on PCs with 3 GHz Pentium IV processors and 2 GB of RAM.

Table 1 shows the time required to process a chunk in each of several scenarios. The base time shows the performance without any security enhancements enabled. When creating an image, compression is the most expensive task in CPU time. Similarly, when installing an image, decompression dominates the CPU cost.

The other columns show the time, absolute overhead, and relative overhead of security measures. Absolute and relative overhead for security features are compared to the base time. While the absolute overhead of security is similar for both image creation and image installation, the relative overhead is different because the base time is much larger for image creation.

A typical image distributed by Frisbee is 300 MB after compression, for 300 chunks worth of data. The CPU time used to create such an image with all security features would be about a minute, and 16 seconds during installation. Approximately 6 seconds of either operation would be due to the overhead of our security enhancements. The additional overhead, while nontrivial, is small enough to allow our security modifications to be used routinely.

5. CONCLUSION AND FUTURE WORK

We have presented the design and implementation of a security system for the Frisbee multicast disk imaging system, which achieves its goals with low overhead. Our solution addresses the main threats to Frisbee in Emulab-based shared testbeds, and deals with Frisbee's special needs.

Our future work will focus on implementing the key distribution schemes we have proposed and integrating our secure version of Frisbee into Emulab, including backwards compatibility for existing images and integrating it into the Emulab user interface.

Acknowledgements

We would like to thank Sneha Kasera for his role in the formulation of this project, and Jay Lepreau for his advice. We would also like to thank the designers and implementers of Frisbee: Chad Barb, Mike Hibler, and Leigh Stoller. This material is based upon work largely supported by the National Science Foundation under Grant No. 0335296.

6. REFERENCES

[1] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. pages 200–208, Sept. 1990.
[2] DETER Network Security Testbed. http://www.deterlab.net/.
[3] P. L. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3. Internet Request for Comments 1950, IETF, May 1996.
[4] OpenSSL EVP API. http://www.openssl.org/docs/crypto/evp.html.
[5] M. Handley et al. The Reliable Multicast Design Space for Bulk Data Transfer. Internet Request For Comments 2887, IETF, Aug. 2000.
[6] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, Scalable Disk Imaging with Frisbee. In *Proceedings of USENIX Annual Technical Conference*, June 2003.
[7] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. Internet Request For Comments 4122, IETF, July 2005.
[8] B. N. Levine and J. Garia-Luna-Aceves. A Comparison of Known Classes of Reliable Multicast Protocols. In *Proc. of IEEE ICNP '96*, pages 112–123, Oct. 1996.
[9] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* 1995.
[10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI '02*, pages 255–270, Boston, MA, Dec. 2002.