Hijacking AFS

P. Honeyman, L.B. Huston, and M.T. Stolarchuk
Center for Information Technology Integration
The University of Michigan
Ann Arbor

Abstract

We have identified several techniques that allow uncontrolled access to files managed by AFS 3.0. One method relies on administrative (or *root*) access to a user's workstation. Defending against this sort of attack is very difficult. Another class of attacks comes from promiscuous access to the physical network. Stronger cryptographic protocols, such as those employed by AFS 3.1, obviate this problem. These exercises help us understand vulnerabilities in the distributed systems that we employ (and deploy), and offer guidelines for securing them.

1. Introduction

At the Center for Information Technology Integration, we are concerned with building large-scale, multi-protocol, multi-vendor systems. Typical of many academic computing environments, the University of Michigan is ''growing'' such a system. While this growth is largely from the bottom-up, we identified at an early date some of the major building blocks that are likely to prevail in the near- and intermediate-term: Macs, PCs, UNIX systems, Kerberos [1], X [2], AFS [3], and NFS [4]. Unfortunately, our vision sometimes blurs when the security aspects of these systems are scrutinized.

The security architecture of several widely used network-based components has come into question in recent years, see *e.g.*, [5, 6, 7, 8] (but see also [9]). While users' privacy concerns usually center on their files, they may also wish to protect their screen contents, keystrokes, or network traffic from being seen. We cover this ground in more detail in a recent report [10], where we emphasize the importance of access control in a distributed file system.

Because many components of a distributed system rely on the file system, they can be no more trustworthy than the file system. Consequently, we study our file systems closely, with an eye to identifying vulnerabilities in their access control mechanisms.

In the next section, we describe several techniques to sidestep the access control mechanisms in AFS 3.0. (We have validated these techniques by building working programs.) We then describe the changes in AFS 3.1 that prevent these and other sorts of attacks, and conclude with some discussion of lessons we learned along the way.

2. Hijacking Rx connections

AFS 3.0 uses a remote procedure call package called Rx for communication between file servers and client cache managers [11]. Rx is connection-oriented; *i.e.*, before a server and a cache manager can do any real work, they must verify their identities to one another, agree to communicate, and generally shake hands. Rx allows the use of different security objects in communications. A security object is a data type that provides procedures useful to services built on Rx, detailed in the table that follows.

The security object employed by AFS 3.0 does not use the full power of the security class. When a connection is established, Rx goes to lengths to authenticate identities securely, relying on the ticket granting capabilities of Kerberos. These tickets contain secret passwords that the server and client use to vouch their identities. But after connection establishment, communications take place without additional cryptographic verification.†

† Rx can use other authentication policies; we describe here the one employed by AFS 3.0.

Operation	Description
`Close`	Discard security object.
`NewConn`	(Re)create a connection.
`DestroyConn`	Destroy a connection.
`PreparePacket`	Encode packet.
`CheckPacket`	Decode packet.
`CheckAuth`	Check whether a connection authenticated properly. Server only.
`CreateChallenge`	Select a nonce. Server only.
`GetChallenge`	Wrap the nonce in a challenge packet. Server only.
`GetResponse`	Respond to a challenge. Client only.
`CheckResponse`	Process a response to a challenge. Server only.

Rx security object

At CITI, we are playing with ways to hijack Rx connections, validating our concern about the overall trustworthiness of our computing environment. We have uncovered two schemes. One involves stealing tickets from a user's workstation. (We'll call the person stealing the tickets the *bad guy*, and the user the *victim*.) The other can be accomplished surreptitiously, from a workstation on any physical network between the victim's workstation and the file server. In this latter scheme, the bad guy writes raw IP packets on an Ethernet (or similar medium), masquerading as the victim.

2.1. From the victim's workstation

Unlike standard UNIX file systems, the administrative account, called *root*, has no special privileges in AFS. On the contrary, root frequently has fewer privileges than an authenticated user. However, becoming root on a user's workstation while she is logged in offers ways to gain access to the user's files, even those stored in AFS.

Many easy attacks are possible from the administrative account, such as modifying local binaries, reading or modifying the contents of the AFS disk cache, *etc*. The attack described below uses root for privileged access to the system memory (`/dev/mem` and `/dev/kmem`). Strictly speaking, root access is not required — any technique that allows access to the physical memory of the machine will do.

By rooting around in UNIX kernel memory, enough information can be gleaned to create new, authenticated Rx connections in the name of the unsuspecting victim. We do this at the user process level, without using any AFS kernel services to reach the remote file server. We can then traverse the AFS file system hierarchy, inspecting and modifying files with abandon.

Our goal is to read or write an AFS file to which access would ordinarily be denied. To accomplish this, we need three pieces of information: the internal AFS name for the file, called a FID; the address of the file server that can service our data access request; and a Kerberos ticket for mutual authentication with that file server.

There are several ways to determine the FID for an AFS file. The one we use opens the file and examines the in-kernel `vnode` for the file; the `vnode` contains the FID. From the FID, we glean the file's parent cell and its volume [12]. Traversing the `afs_volumes` table in the kernel gives us the address of the server for that volume.

Now that we know the identity of the server that can do our work, all we need to finish the job are the Kerberos credentials of an authenticated user who has the proper access rights to the file.‡ With this in hand, we can create an authenticated connection to the server, along which we can pass our `FETCHDATA` request. Conveniently, AFS maintains the `afs_users` table in kernel memory, indexed by cell and user id. This table has a pointer to the victim's Kerberos ticket, which we use to create an authenticated connection of our own.

Liberal use of AFS support libraries simplifies the task. We leave as an exercise the details of how to access the file if we do not have permission to open it. The essential point is that we can start from the root of a cell, issuing authenticated directory lookups to find the FID.

‡ We have elided some of the complexity from this description, *e.g.*, pretending to have a callback service available.

In addition to technique just described, many other methods can be imagined, most of which probably work. Note that it is not within the scope of AFS to prevent access to resources maintained on client machines. However, the ease with which root attacks can be implemented does affect such policy decisions as distribution of the root password, and allowing simultaneous logins on one machine.

2.2. From the victim's network

A trickier way to obtain uncontrolled access is to snoop on an Ethernet. By running the Ethernet interface in promiscuous mode, a single machine can monitor conversations between AFS clients and servers, waiting for the appearance of an authenticated connection. The bad guy can then hijack this connection and use it for his own purposes.

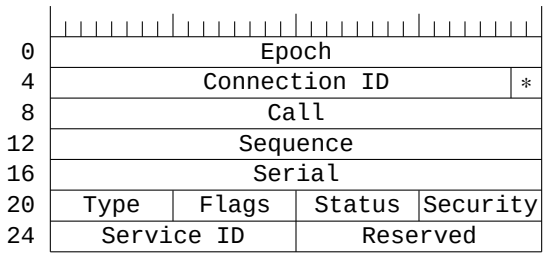
This attack is possible on any network where eavesdropping is possible. The bad guy does not have to be on the same subnet as the client — as long as the bad guy has promiscuous access to a physical network between the victim and the file server, he can monitor the protocol.

Prying open the Rx packets, the bad guy copies the IP, UDP, and Rx information into a packet of his own manufacture. The trick is to alter the Rx connection in a subtle way, so that these illegitimate packets do not interfere with those of the victim.

Rx is connection-oriented, *i.e.*, a client and server must shake hands before communication can take place. It is at this point that mutual authentication takes place. Thereafter, the connection is assumed to be authenticated for a period of time, during which all requests are assumed to be authentic. In AFS 3.0, the connection is good for up to a day or so, although the server may clean up (or *reap*) idle connections at any time.

If he can read and write raw packets on the physical network, the bad guy can ''borrow'' authenticated connections by inspecting communication between the victim and the server and issuing IP packets to the server that appear to originate from the victim. Through this misrepresentation, the bad guy can convince the server that it is acting on behalf of an authentic request from the victim.

Rx is a windowing protocol, which presents certain subtleties. Both client and server keep track of the highest call number used to completion. Any packet with a smaller call number is discarded as a straggler. Straightforward use of the victim's connection may cause the server to discard the victim's later, valid requests, possibly exposing the bad guy. Fortunately, Rx offers an easy solution to this dilemma: *channels*, which allow multiple simultaneous calls to share the same authentication information.



Rx header

An Rx header, depicted in the figure above, is 28 bytes. The two-bit field marked * specifies the `Channel ID`. Rx supports up to four channels per connection. When a remote procedure call is made, the Rx client uses the lowest channel with no calls outstanding. Most calls take place on channel zero, but channels one, two, and three may also be used. It is very rare for channel three to be used, although we have seen it happen.

Using connection and channel identification information that we snoop off the network, we employ channel three to do our work. The beauty of this scheme is that the victim's Rx connection silently discards the responses to our bogus requests, so the subterfuge is completely invisible to the victim!

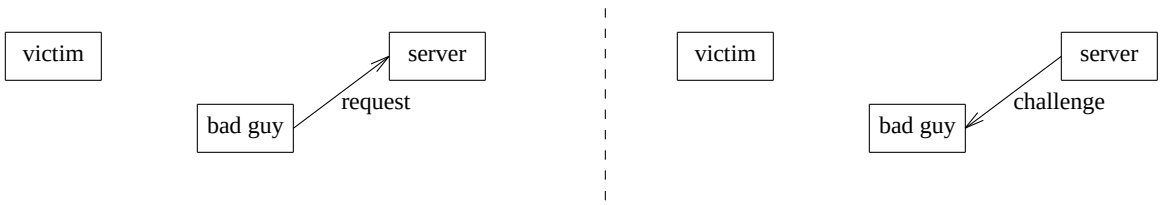
2.3. A challenge/response oracle

Rather than passively borrowing an existing connection, we have devised a proactive means to create an authenticated connection. Our scheme requires the momentary assistance of an already authenticated cache manager to act as an ‘‘oracle.’’

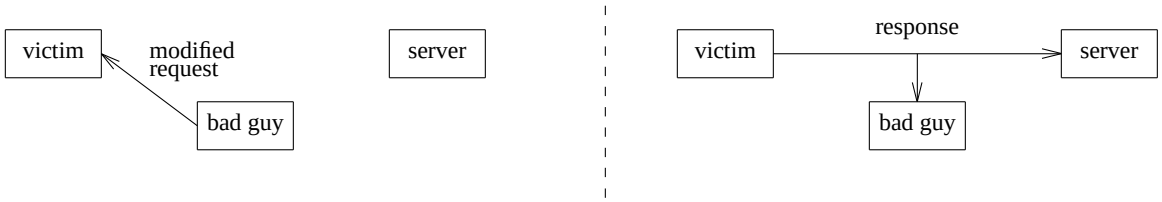
To explain this scheme, we first describe in more detail the mechanism by which a cache manager authenticates itself with a file server. Whenever a file system request is received by a server, it checks to see whether the request is associated with an authenticated connection. If so, the request is serviced with the cached credentials associated with the connection. Otherwise, the server issues a ‘‘challenge’’ to the cache manager that made the request. A cache manager is prepared to accept a challenge at any time. This allows the server to reap old connection state without explicitly tearing down the connection.

A challenge packet consists principally of a 32-bit nonce identifier. Using the operations defined for the Rx security object, a challenge packet is prepared and sent to a client. The client increments the nonce, seals the response with the session key in the Kerberos ticket, and returns the result and the ticket to the challenger.

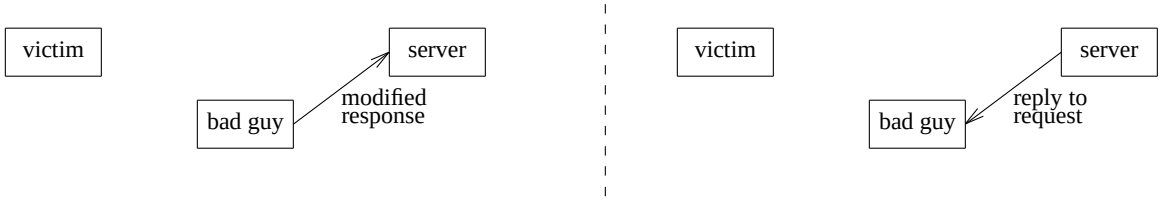
The oracle subterfuge takes advantage of a client’s willingness to repond to a challenge at any time, and the lack of any connection-specific information in the response. The bad guy creates a file server connection by issuing a request to the server, *e.g.*, with a `FETCHDATA` or `STOREDATA` request. The server sees this as a new, unauthenticated connection and issues the bad guy a challenge packet.



Now the dirty work. The bad guy takes the challenge packet and changes the Rx header to make the packet appear to be a challenge to an existing Rx connection on the victim’s machine. The bad guy then sends this request to the victim’s machine as though it were from the file server. The forwarded request is processed by the victim’s unsuspecting cache manager, which prepares and sends a response. Although this response is seen by the file server, it is quickly discarded because a response packet on this connection is not pending.



The bad guy, who was listening for the response, modifies the Rx header to make it correspond to the challenge that is pending. Upon sending it to the server, the bad guy is rewarded with an authenticated connection that can be used for a lengthy period. Neither the server nor the victim is aware that they have been had!



3. Changes in AFS 3.1

The latest version of AFS from Transarc, AFS 3.1, addresses these crucial security issues. In particular, Rx now includes an encrypted verifier on every packet exchanged between a client and a server. This verifier is built by smashing together the connection ID, the call number, the channel number, the security index, the packet sequence number, and the session key for the connection. The packet sequence number is included to assure that the verifier changes on every packet. The call, channel, and connection ID insulate the connection from replay attacks. These values are combined with the per-connection session key to obscure the contents, because they appear in cleartext in the Rx header.

The resulting 64-bit word is encrypted under the session key. If the session key is distributed to the client and server securely, its use ensures that only the client and server can construct a proper verifier. Finally, a 16-bit chunk of the resulting 64-bit word is included as the packet header verifier. Although such a small verifier is susceptible to exhaustive search attack, if the server is ever presented with an invalid verifier, it instructs the client to abort the connection.

In addition, the challenge-response step has been modified to prevent a combination of oracle attack and replay attempts. The client now includes in the response to a challenge the largest sequence number ever used on the connection it is authenticating. This prevents replay attacks after the file server has discarded the state associated with a connection.

With these changes to the Rx security object, our network-based attacks no longer function. At this writing, we know of no way to bypass the access control protections in AFS 3.1.

4. Discussion

In penetrating our file servers, we have discovered (or rediscovered) important lessons and useful techniques for securing our computing environment.

A typical university computing site offers public access to the distributed computing environment with minimal supervision. We can not assume trusted kernels or utilities, and must view workstations as pawns under continual attack. What we really need is a way to ‘‘scrub,’’ or initialize a workstation to a known state in a secure way whenever a new user wants to login. But our scrub procedure is not secure, and in any event takes too long.

Although still subject to a Trojan horse attack [13], the environment would be somewhat more secure by prohibiting simultaneous use of a workstation by more than one user. This is standard procedure for Project Athena [14] workstations.

The guarantees offered by the security object in AFS 3.0 are fatally weakened by the absence of an encrypted verifier in every packet. After we pointed out these weaknesses, Transarc released a much more secure version of their distributed file system. AFS 3.1, based on an extensively modified version of Rx, obviates the channel and oracle attacks. In addition, Transarc identified other security protocol issues, ones that escaped our attention, and modified the protocol to deal with them as well.

We feel that other distributed file systems have overlooked security issues as well; certainly AFS is not alone with these problems. For example, virtually every NFS installation we know of has security holes that make AFS 3.0 look like Fort Knox.

It must be kept in mind that passive snooping on an Ethernet offers vast opportunities to attack systems, *e.g.*, cleartext passwords regularly appear on our local Ethernet, especially those of our system administrators. It is vital that we find network technologies that do not admit promiscuous access.

As technology moves toward distributed services, it becomes increasingly difficult to secure the boundary between an end system and a service provider. Current trends indicate a growing concern for security issues; designers of distributed file systems, and other distributed services, must pay careful attention to the level of protection that they provide.

Acknowledgements

Dave Bachmann and Bob Braden helped track down much useful information in mailing list archives.

We thank Edna Brenner for her careful reading of this manuscript and for her many suggestions that led to improvement.

This work was partially funded by the IBM Corporation.

References

1. J. G. Steiner, B. C. Neuman, and J. I. Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ pp. 191–202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).
2. R.W. Scheifler and J. Gettys, ‘‘The X Window System,’’ *ACM Transactions on Graphics* **5**(2), pp. 79–109 (April, 1987).
3. J.H. Howard, ‘‘An Overview of the Andrew File System,’’ pp. 23–26 in *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 1988).
4. D. Walsh, B. Lyon, G. Sager, J.M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss, ‘‘Overview of the Sun Network Filesystem,’’ *Winter Usenix Conference Proceedings*, Dallas (1985).
5. R.T. Morris, ‘‘A Weakness in the 4.2BSD UNIX TCP/IP Software,’’ Computer Science Technical Report No. 117, AT&T Bell Labs, Murray Hill (1985).
6. W.E. Sommerfeld, ‘‘Re: Ethernet Bridge (really: NFS ‘security’),’’ Message 1761@bloom-beacon.MIT.EDU, TCP-IP mailing list (November, 1987).
7. S.M. Bellovin, ‘‘Security Problems in the TCP/IP Protocol Suite,’’ *Computer Communication Review* **19**(2), pp. 32–48, ACM SIGCOMM (April, 1989).
8. S.M. Bellovin and M. Merritt, ‘‘Limitations of the Kerberos Authentication System,’’ *Computer Communication Review* **20**(5), pp. 119–132 (October, 1990).
9. Stephen T. Kent, ‘‘Comments on ‘Security Problems in the TCP/IP Protocol Suite’,’’ *Computer Communication Review* **19**(3), pp. 10–19 (July, 1989).
10. C.J. Antonelli, W.A. Doster, and P. Honeyman, ‘‘Access Control in a Workstation-based Distributed Computing Environment,’’ *Proc. of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, pp. 13–19 (October, 1990).
11. R.N. Sidebotham, ‘‘Rx: Extended Remote Procedure Call,’’ in *Proceedings of the Nationwide File System Workshop*, Information Technology Center, Carnegie Mellon University, Pittsburgh (August, 1988).
12. R.N. Sidebotham, ‘‘Volumes: The Andrew File System Data Structuring Primitive,’’ *European Unix User Group Conf. Proc.* (August, 1986).
13. K. Thompson, ‘‘Reflections on Trusting Trust,’’ *Communications of the ACM* **27**(8), pp. 761–763 (August, 1984).
14. E. Balkovich, S.R. Lerman, and R.P. Parmelle, ‘‘Computing in Higher Education: The Athena Experience,’’ *Communications of the ACM* **28**(11), pp. 1214–1224 (November, 1985).

About the authors

After completing undergraduate studies at the University of Michigan, **Peter Honeyman** was awarded the Ph.D. by Princeton University for research in relational database theory. He has been a Member of Technical Staff at Bell Labs and Assistant Professor of Computer Science at Princeton University. He is currently Associate Research Scientist at the University of Michigan’s Center for Information Technology Integration. Honeyman has been instrumental in several significant software projects, including Honey DanBer UUCP, Pathalias, MacNFS, and the Telebit UUCP spoof. His current research efforts are focused on distributed file systems, with an emphasis on mobile computing, security, and performance. He can be contacted at `honey@citi.umich.edu`.

Larry Huston is a graduate student at the University of Michigan, where he did his undergraduate work in Computer Science and Aerospace Engineering. Larry began his graduate career as an Aerospace student concentrating on numerical analysis, but switched to Computer Science to do more programming and less math. His research interests are in low speed networking and mobile computing. He can be contacted at `lhuston@citi.umich.edu`.

Michael T. Stolarchuk received his undergraduate degree in Computer and Communication Sciences from the University of Michigan. After discovering the wrong life in California (TSO/SPF/IMS on IBM 3033), he had the unique opportunity of bringing the first Vax/BSD Unix to the University of Michigan Engineering College. After a brief stint in an Ann Arbor company, and after getting his M.S. in Computer, Information, and Control Engineering at Michigan, he returned to work at the University. He later joined the IFS Project at CITI. Recent projects include porting AFS to the NeXT machine (for Educom ′89) and AFS-to-your-protocol-here translators. His current interests are AFS protocol translators and security, fast file systems, fast food, and fast cars. Send him mail at `mts@citi.umich.edu`.