CITI Technical Report 91–4

Hijacking AFS

P. Honeyman
honey@citi.umich.edu

L.B. Huston
lhuston@citi.umich.edu

M.T. Stolarchuk
mts@citi.umich.edu

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.

August 28, 1991

Hijacking AFS

P. Honeyman
honey@citi.umich.edu

L.B. Huston
lhuston@citi.umich.edu

M.T. Stolarchuk
mts@citi.umich.edu

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 future: Macs, PCs, UNIX systems, Kerberos [1], X [2], AFS [3], and NFS [4]. Unfortunately, our vision sometimes blurs when the security aspects of the system 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. In the remainder of the paper, we describe the changes in AFS 3.1 that prevent these and other sorts of attacks, and discuss some of the lessons we learned along the way.

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 Table 1.

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.

Table 1

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.[1]

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.

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 usually 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*. It is not within the scope of AFS to prevent access to resources maintained on client machines. The attack described below uses root to gain access to the memory devices `/dev/mem`, `/dev/kmem`). Obviously root is not required — any technique that allows access to the physical memory of the machine suffices.

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.[2] With this, we 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.

From the victim's network

A trickier way to arrogate uncontrolled access is to snoop on an Ethernet, waiting for an AFS request packet along an authenticated connection. By running the Ethernet interface in promiscuous mode, a single machine can monitor the authentication protocol between other clients and servers. The bad guy can then hijack this authenticated 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, we copy the IP,

---

1 Rx can use other authentication policies, but the only ones implemented in AFS 3.0 are the one we describe here, and the ''unauthenticated'' policy.

† UNIX is a Trademark of AT&T Bell Laboratories.

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

UDP, and Rx information into a packet of our own manufacture. The trick is to alter the Rx connection in a subtle way, so that our 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 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.

By reading and writing raw packets from the physical network, the bad guy can ‘‘borrow’’ authenticated connections, inspecting communication between the victim and the server, and issue 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 call number less than this 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 with channels, which allow multiple simultaneous calls to share the same authentication information.

0	Epoch				
4	Connection ID				*
8	Call				
12	Sequence				
16	Serial				
20	Type	Flags	Status	Security	
24	Service ID		Reserved		

Figure 1

An Rx header, depicted in Figure 1, 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!

A challenge/response oracle

As an alternative to passively borrowing an existing connection, we have devised a more proactive method 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 procedures described in Table 1, a challenge packet is sent to a client. The client increments the nonce, seals the response with the session key in the Kerberos ticket, and returns the result 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 fetch or store 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 this 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.

Now the bad guy, who was listening for the response, modifies the Rx header to make it correspond to the challenge that is pending. By 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 she has been had!

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.

With these changes to the Rx security object, the network-based attacks described here no longer function. We are unable to bypass the access control protections in AFS 3.1.

Discussion

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

In a typical university computing site that enjoys public access with minimal supervision, we can not assume trusted kernels or utilities on workstations, and must view them as pawns continually under 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, as in Project Athena [14].

The guarantees offered by the security object in AFS 3.0 are fatally weakened by the absence of an encrypted verifier in every packet. AFS 3.1, based on an extensively modified version of Rx, obviates the network 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.

Of course, it must be kept in mind that passive snooping on an Ethernet offers many opportunities to attack insecure systems; cleartext passwords regularly appear on our local Ethernet, especially those of our system administrators. In the intermediate term, the University’s strategic direction is toward network technologies that do not admit promiscuous access, such as twisted pair Ethernet.

Acknowledgements

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

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, also available as CITI Technical Report 90-2 (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.*, also available as Technical Report CMU-ITC-053, Information Technology Center, Carnegie Mellon University (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).