Function Call Tracing Attacks to Kerberos 5

Julian L. Rrushi, Emilia Rosti
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39, 20135 Milan - Italy

e-mails: *jlr@cert-it.dico.unimi.it*, *rose@dico.unimi.it*

June 22, 2005

Abstract

During the authentication process in the Kerberos network authentication system, all the information exchanged between the application client and the Kerberos authentication server is the argument of some function calls to Kerberos shared libraries. Since this information is exchanged in the clear, local attacks that intercept function calls may inspect and manipulate it before resuming their execution. This paper describes function call tracing attacks against the Kerberos authentication system in a time-sharing environment. They use the DynInst API library, developed to support the easy construction of tools for the control and manipulation of programs at run-time, and ad hoc interposition libraries. We illustrate the proposed attacks against two Kerberos client applications, namely `kinit` and `kpasswd`.

Keywords Kerberos, interception, tracing, shared libraries, library interposition, krb5-libs, DynInst.

1 Introduction

Computer systems over the years have been affected by many kinds of vulnerabilities ranging from flaws in the design or implementation of communication protocols to buffer overflows and format strings. Interception and/or manipulation of communications over the network has also been a continuous threat to information security. To resolve this problem network protocols such as Kerberos have incorporated cryptographic functionalities in order to protect their communications sent over the network.
Practice has shown that the network is not the only point being in danger from such a threat. This paper describes a set of function call tracing attacks which aim at intercepting and manipulating arguments of the function calls an application makes to the stack of shared libraries. These attacks were performed against Linux implementations of Kerberos 5. They were performed in two different ways. In the first form a call tracing attack uses the DynInst library in order to spy or control a target Kerberos application client's process. In the second form a call tracing attack uses an interposition library which is placed between the Kerberos application client's process and Kerberos shared libraries, and acts as a man-in-the-middle does in the network environment. The attacks discussed in this paper were performed in order to assess the defense of Kerberos 5 from local hostile interceptions and interferences. They demonstrate that Kerberos application clients and Kerberos shared libraries are vulnerable to function call tracing attacks.

This paper is organized as follows. Section 2 briefly recalls the Kerberos authentication system. Section 3 illustrates the concept of function call tracing, which is at the base of the attacks we present in Section 4 and Section 5. Section 6 discusses related work on call tracing. Section 7 summarizes our findings and concludes the paper.

2 Kerberos

In this section we briefly recall Kerberos main features that are of interest to our work. For a more detailed description of Kerberos architecture and functionalities, as well the most up to date information about releases and security advisories see [15, 16]. Kerberos is a trusted third-party authentication service that provides a means of verifying the identities of principals in an unprotected network [1]. A Kerberos authentication service is composed of the Key Distribution Center (KDC), which in turn comprises the Authentication Server (AS) and the Ticket Granting Server (TGS). These two supply temporary session keys and initial tickets, as well as a service for managing principals and their passwords from anywhere in the network, called Kerberos Administration Service (KADM). The authentication process is started by the application client and is composed of the messages exchanged between the application client, the KDC, and the application server. If the authentication process succeeds, the authentication server issues a certificate, called ticket.

In the case Kerberos uses secret-key cryptography, both Kerberos and the application client derive the application client's secret key from the application client's password. Possession of the secret key is considered by Kerberos as a proof of identity, thus it is of paramount importance that the application client keeps its secret key, and its password, secret. Therefore, one of the most critical information is represented by the password the application client shares with the KDC. Other critical information comprises the password the application server shares with the KDC, the session keys and sub keys. In what follows we show how it is possible for an attacker to get hold of such sensitive information by using "function call tracing."

3 Function Call Tracing

A function call tracing attack is one that aims at intercepting and manipulating the arguments of the function calls a target application makes to the system shared libraries. It may be applied in two different forms as described by the two following sections. In the attacks discussed in this paper we consider the attacker has no privileges but those of a simple user. A virus program was developed to implement the attacks. This was done in order to have the attacking code execute with the uid of the target user, which allowed circumvention of all the constraints dictated by the operating system. Its functionalities vary from simply setting an environment variable on behalf of a user to attaching to processes of the Kerberos clients we used to demonstrate our attacking strategy, and operating on them. Such an approach proved to be the strongest form of attack as it is possible to act with the user id of the target user, which is fundamental for our attacks to succeed. Furthermore, the attacker does not need to be logged in when the victim user is and the attacker's user id is not recorded in any log file. Our attacks were performed on a Linux Fedora Core 3, 2.6.9-1.667 kernel, operating system running on an Intel Pentium 4 processor machine with 512 MB of RAM. The Kerberos version running on the system is composed of krb5-server-1.3.4-7, krb5-libs-1.3.4-7 and krb5-workstation-1.3.4-7.

4 DynInst-Based Attacks

In this section we show how the DynInst API library can be used to attack the Kerberos authentication system. We performed the attack discussed in this section against `kpasswd`, which is a Kerberos client application that enables users to change their password stored in the KDC database.

4.1 DynInst API Library

The DynInst API library was developed to provide a portable library for tool builders to construct tools that operate on a running program [5]. It turns out DynInst API is a very powerful attack tool since it offers a wide range of actions that can be performed via dynamic instrumentation. A DynInst API user may observe and/or control the execution of a target process, actively insert new instructions into the address space of the target process, have new libraries dynamically loaded in the process address space, replace single instructions or entire functions.

DynInst performs operations on the target process either via the process debugging interface of the operating system or the run-time instrumentation library (RTinst), which is a shared library DynInst loads in the address space of the target process. In a Linux system, the process debugging interface of the operating system can be accessed via `ptrace` or `/proc`. Note that:

1. any user process can be attached via `ptrace` by a process running with the same process id as that user, or by root.

2. `/proc` provides access to the information related to a running process, which is needed by the `ioctl()` system call to operate on it, only to the owner of that process or to root.

Thus, an attacker, who is a simple user of the time-sharing system, cannot manipulate directly a process of another user using the DynInst API. In order to bypass such a limitation, we developed a virus program that issues API calls to the DynInst library in order to use its functionalities.

4.2 Attack Organization

The attack discussed in this section consists of inserting into a function of the target process a *snippet*, i.e., a representation of a piece of executable code to be inserted into a program at a given point [14]. The snippet is inserted at the beginning of the target function and records all its arguments. Snippet insertion is one of the DynInst marvelous functionalities. In our case, the target program is `kpasswd` as depicted in Figure 1. `krb5_change_password` is the function where the snippet was inserted.

The attack comprises the following phases:

1. The viral code first attaches to the target `kpasswd` process. This operation has no effects on the state of the target `kpasswd` process. A thread is created that refers to the `kpasswd` process.

2. The viral code finds the `krb5_change_password` function in the program image associated with the thread created in the previous phase.

3. It locates the entry point to `krb5_change_password` and inserts the snippet there.

`kpasswd` executes `krb5_change_password` only once. The smoothest attack execution occurs when the viral code attaches to the target `kpasswd` process after the Kerberos shared libraries have been loaded but before `krb5_change_password` is called. If `kpasswd` had already called `krb5_change_password` in the moment the viral code attaches to it, trying to make it pass one more time in order for the snippet to execute may crash it, unless a proper manipulation of its variables is performed. When `kpasswd` calls the function `krb5_change_password`, the snippet is executed first and records the value of the function parameter *newpw* that represents the new password of the target user.

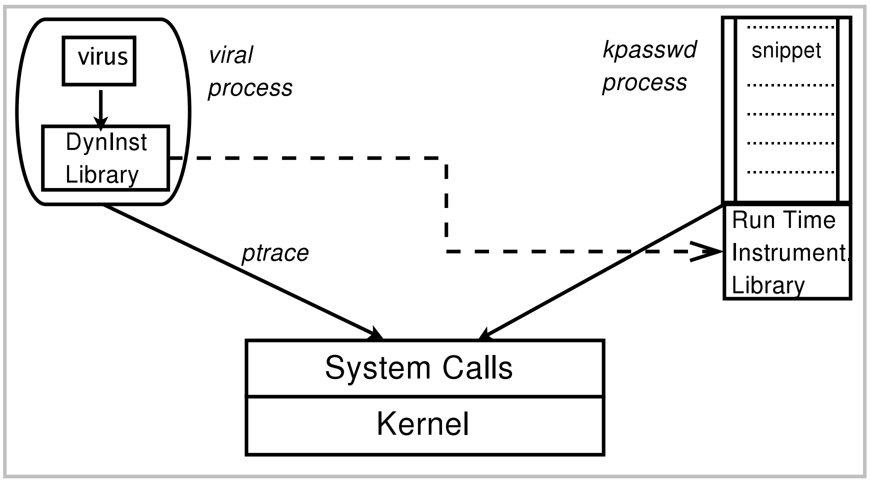

Figure 1: DynInst based attacking technique.

5 Interposition Libraries Based Attacks

In this section we describe an alternative way to trace a victim process' function calls, namely via interposition libraries. The process of placing a new or different library function between the application and its reference to a library function is called interposition [9]. Interposition libraries are placed between a program and the shared libraries to which functions and variables of this program are linked during compilation. This strategic position enables them to intercept all the function calls the program makes to the stack of shared libraries. Interposition libraries were originally invented for monitoring the behavior of functions, gather statistics, perform debugging etc, but they can also be used for illegitimate purposes. Programs that depend on shared libraries are vulnerable to a variety of attacks that involve switching the shared library that the program is running [3]. In [13] are described attacks against programs that are linked to the `libc` and `MPI` shared libraries. An attack which uses interposition libraries to intercept sensible information passed to the shared libraries in the form of function call parameters, in this paper is referred to as an interposition attack.

The interposition attacks discussed in this paper targeted client applications which use the *krb5-libs* package. *krb5-libs* contains the shared libraries needed by Kerberos 5. In this section are described interposition attacks we performed against `kinit` and `kpasswd`. These interposition attacks were performed by viral code in two sequential steps:

1. Achieving Interposition
2. Intercepting Function Calls with Ad Hoc Interposition Libraries

The efficiency of these attacks demonstrated to be entirely based on the ability of the attacker to interpose between the Kerberos application program and the Kerberos shared libraries it was linked to. However, the success of the entire interposition attack depends on the individual success of each one of the steps given above. These two typical steps of an interposition attack are described separately in the following subsections.

5.1 Techniques for Achieving Interposition

Interposition can be achieved in any of the following ways

1. **LD_PRELOAD**

   `LD_PRELOAD` is an environment variable whose traditional goal is to allow application developers to trace and debug their application programs. In these cases application developers assign to `LD_PRELOAD` a value that is the path to a shared library they need to interpose. During dynamic binding the dynamic linker will give precedence to the library whose path is contained in `LD_PRELOAD` over the system's shared libraries. The attacker as a simple user of the time–sharing system cannot directly set environment variables on behalf of another user. Consequently this step was performed by viral code which set the `LD_PRELOAD` to the path of the malicious interposition library on behalf of the user who executed it.

2. **Manipulation of linkage tables**

   During the compilation of a program, which contains references to functions or variables of shared libraries, the compiler and static linker generate a linkage table. The linkage table contains symbols, i.e., functions and variables, and for each symbol a pointer. The linkage table is placed in the program's executable file. When the program is loaded, the dynamic linker sets each pointer to the exact location in memory where the respective symbol is defined.
   The interposition is obtained by accessing the *linkage table* and modifying the pointer corresponding to the function we want to redirect. Although there may be other different techniques for doing this, a general approach consists of

loading the interposition library by using interfaces provided by the operating system, locating the memory address of the interposing function defined by the interposition library and setting the pointer in question to point to this memory address.

3. **DynInst**

   As we wrote in Section 4, DynInst can replace existing instructions of the target process and load new libraries in its address space. Interposition is achieved by loading the interposition library into the address space of the target process and changing the call to a function of the Kerberos shared libraries to be a call to a function of the interposition library, which in turn may call the function in question of the Kerberos shared libraries, thus put itself in the middle.

5.2 Intercepting Function Calls

Once interposition has been achieved using one of the techniques illustrated above, the proper code must be provided for the attack to succeed Figure 2. In our case, with `kinit` and `kpasswd`, the goal is to access another user's tickets and obtain the new password of the target user, respectively. We developed the interposition library aimed at hijacking the destination file where a user stores the tickets he/she obtains with the `kinit` client application[1]. The stolen tickets, once stored in the destination indicated by the attacker, are accessible to the attacker. Without loss of generality, we performed the attack on a Linux system with two users, namely *user* with `uid` 503 and *malicious-user* with `uid` 502. Both users have accounts on a remote Kerberos server as principals *user* and *malicious-user*, respectively, and they can both connect to a remote `telnet` server once authenticated by Kerberos.

Before having the viral code interpose his/her malicious interposition library between `kinit` and Kerberos shared libraries, *malicious-user* uses an ad hoc program that generates the default cache file where tickets can be stored and changes its permissions so that any other user can read and write such a file. Then he initiates the infection process. The viral code activates interposition and waits for a victim, *user*, to request a ticket with `kinit` in order to access the `telnet` service. The hijacking process occurs when `kinit` calls the function `krb5_cc_resolve`. In that moment the malicious interposition library intercepts the function call and replaces the argument containing the name of the file where the ticket will be stored with the name of the default cache file generated before. The real function `krb5_cc_resolve` from the Kerberos shared libraries is then called with the modified arguments. The malicious interposition library also intercepts the call to the `krb5_cc_initialize` function, to prevent it from detecting the anomaly of the destination file that does not belong to *user*. In this case, the real `krb5_cc_initialize` function from the Kerberos shared libraries is not called and the value 0, i.e. no error, is simply returned. `kinit` then proceeds to store the ticket in the previously generated default cache file. This way *malicious-user* hijacked the destination where the ticket of *user* was to be stored.

A similar attack was performed against `kpasswd`. In this case, the *malicious-user*'s viral code activates interposition and waits for *user* to change the password he/she shares with the Kerberos server. When *user* executes `kpasswd` to change the password, the malicious interposition library intercepts the call to the function `krb5_change_password`. Since one of the arguments is *user*'s new password, the malicious library records such a value and then calls the real `krb5_change_password` function from the Kerberos shared libraries.

[1]The code of the interposition library is available for download at the URL: `http://cert-it.dico.unimi.it/~jlr/interpose-lib.c` after contacting the authors.

6 Related Work

In this section we review previous work regarding call tracing and its use as an attack strategy in a variety of environments.

6.1 Call Tracing for IDS Evasion

Interception of function calls and modification of their arguments has been used as a technique to evade host-based Intrusion Detection Systems (IDS) [7]. A typical host-based IDS examines the system calls executed by each application. How the intrusion is detected from the analysis of the function call arguments or its occurrence or other parameters depends upon the specific IDS. In [8] the IDS tries to learn the normal behavior of applications by collecting system call traces during time intervals when the system is not under attack. Then the IDS extracts all possible sub traces containing six consecutive system calls and inserts them into a data base. These steps are referred to as the *learning phase*. A sub trace that does not match any of the entries of the database is tagged as abnormal. In the *monitoring phase*, when the IDS monitors applications in order to detect intrusions, the IDS collects traces of the function calls executed by each application and measures the degree of abnormality of each individual call trace as the number of abnormal sub traces it contains.

In order to detect an intrusion, the IDS can, as well, use waypoints and impossible paths [6]. Waypoints are kernel-supported trustworthy markers on the execution path of an application that must be followed by that application when making system calls. An impossible path is a sequence of system calls that an application will never execute.

The evasion technique consists of replacing the arguments of system calls that are part of the normal behavior of an application with values that represent operations the attacker wants to perform. This way the legitimate system call executed by the application will execute on behalf of the attacker. For instance, if the attacker wants to write to the shadow file and the function `open("/lib/libc.so", O_RDONLY)` belongs to the normal execution of an application, in the exact moment the application calls it the attacker replaces the argument with `("/etc/shadow", O_RDWR)` [7]. Considering the way a typical IDS works, this attack will not be detected.

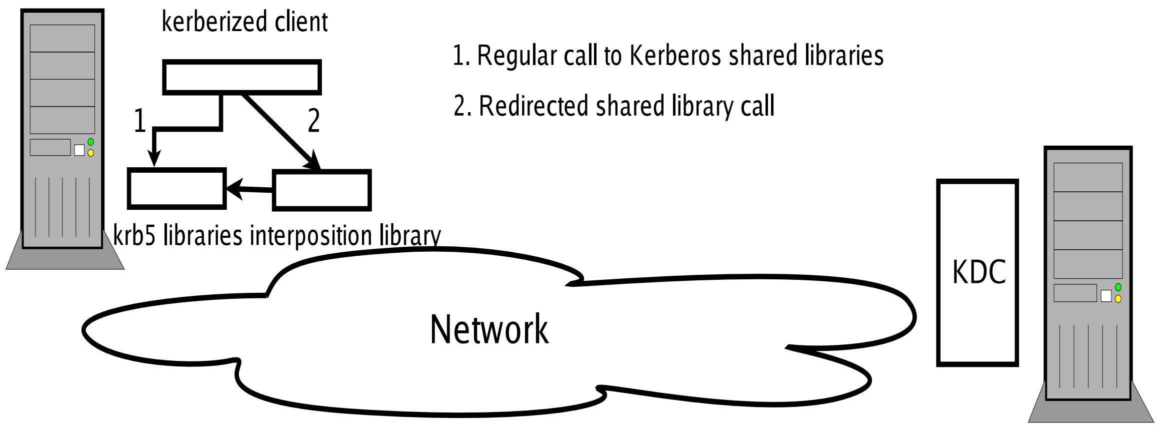
kerberized client
1. Regular call to Kerberos shared libraries
2. Redirected shared library call
1
2
krb5 libraries interposition library
Network
KDC

Figure 2: Interposition libraries based attack

6.2 Call Tracing in Attacks to ODBC

An example of a protocol whose implementation is vulnerable to call tracing attacks is the Open Data Base Connectivity (ODBC) protocol [19]. In a typical interaction between a client and a database server, the ODBC Manager of the client calls the client's ODBC Drive, which in turn communicates with the database server. In early standard implementations of this protocol, by locally intercepting the function calls from the ODBC Manager to the ODBC Drive, it was possible to capture the user's id and password. As a countermeasure to call tracing, various database technologies avoided passing sensitive information through an ODBC connection by creating a separate secure channel, e.g., for sending authentication credentials. Some proprietary implementations of ODBC added encryption functionalities to ODBC Manager and ODBC Drive in order to solve the problem [4].

6.3 Call Tracing in Attacks to Condor

Condor is a specialized workload management system for compute-intensive jobs [11]. It allows users to schedule and run their application programs on remote idle hosts in a widely-distributed environment. Condor users are not assigned an account on the remote host. In the remote hosts their applications run with an anonymous and restricted `uid`, such as `nobody` on a Linux system. When a user submits a job to Condor from its *Submitting Machine* (SM), Condor places it into a queue and according to the adopted scheduling policy it decides when and where, i.e., on what host, referred to as *Execution Machine* (EM), to run it. After allocating the application program to a remote EM, Condor links it with the Condor RPC library, which forwards all the system calls issued by the application to the SM via remote procedure calls. That is, systems calls performed by an application in the EM actually execute on the SM and the values they return are sent back to the EM.

In such a scenario, an attack can be performed as follows [5]. The attacker submits a malicious job to Condor from his/her SM. Condor schedules the malicious job on an idle EM of choice. Once it starts executing on the remote host with `uid nobody`, the malicious job spawns a new process, which will remain in the system even after its parent completes or is migrated by Condor to another EM. When a new job is allocated on the compromised machine, the malicious resident process can intercept the system calls performed by the new job and manipulate them in order to perform operations on the SM from where the new job had been submitted.

7 Observations and Conclusions

The attacks we discussed in this paper may be performed directly, i.e., not through viral code but executing attacking code with their user id. This is possible if the level of privilege of the attacker is high enough to be able to affect other users' execution environments or attach to other users' processes and operate on them. Such a category of users can, of course, apply other strong attacks like backdooring binaries [17], injecting a redirected library call into an executable [18] or using a compiler to statically inject calls to a tracing library into an application program. However, when integrity checking is in place and high privileged malicious users cannot control it, they could still apply the strategies discussed in this paper in order to attack a Kerberos user without modifying the system software. The high privileged users category includes privileged users specified in the file `/etc/sudoers` in such a way that they are allowed either explicitly, i.e., by uid or implicitly, i.e., by gid, to create processes that can attach to processes of other users or set environment variables on their behalf. Users who are granted a `chroot` virtual file system and are `ptrace`-capable are also part of the category.

Another scenario where the attacks discussed in this paper can take place is represented by users with different usernames and passwords but the same user id, `uid` [3]. Under Linux, the `useradd` command has an option, `-o`, that allows the system administrator to create accounts with the same uid. In this scenario each of the users with the same uid can spy the others by tracing their processes.

Unless the attacks described in this paper are performed in a direct way by the category of high privileged users defined above, they require the execution of code on behalf of the target user. This is why we adopted the virus strategy, so that the malicious user can act locally as a simple user. Such a local user has better chances to infect the local system than an outside attacker. However, being a local user is a weak requirement, as an attacker could perform the attacks we described from any host on the network, as long as he/she succeeds in infecting the target host with the virus and eventually a Kerberos user authenticates in a Kerberos realm.

The work presented in this paper focused on demonstrating the problematic nature of tracing the function calls in a Kerberos environment. Further study will concentrate on building algorithms that aim at detecting function call tracing attacks, and on defining mechanisms that defeat local interception, at least with regard to attacks against the confidentiality of the Kerberos user. Other fields of investigation include general integrity checking algorithms for detecting infection activities inside the system, and lightweight cryptographic separation techniques that could enable a process to conceal its data and computation in such a way that it is unintelligible to outside processes.

References

[1] Kohl J., Neuman B. C., "The Kerberos Network Authentication Service (V5)," RFC 1510, September 1993.

[2] Pfleeger C.P., "Security In Computing," Prentice Hall, 1997.

[3] Garfinkel S., Spafford G., "Practical Unix & Internet Security," 2nd edition, O'Reilly, April 1996.

[4] Fugini M., Maio F., Plebani P., "Sicurezza dei sistemi informatici", Apogeo, March 2001.

[5] Miller B.P., Christodorescu M., Iverson R., Kosar T., Mirgordskii A., and Popovici F., "Playing inside the black box: Using dynamic instrumentation to create security holes," Parallel Processing Letters, June/September 2001.

[6] Xu H., Du W., and Chapin S.J., "Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths," Proc. of the Seventh International Symposium on Recent Advances in Intrusion Detection, RAID 2004, September 2004.

[7] Wagner D. and Soto P., "Mimicry Attacks on Host-based Intrusion Detection Systems," Proc. of the 9th ACM Conference on Computer and Communications Security, 2002.

[8] Hofmeyr S., Forrest S., Somayaji A., "Intrusion Detection Using Sequences of System Calls," Journal of Computer Security, 1998.

[9] Curry T.W., "Profiling and Tracing Dynamic Library Usage Via Interposition," Proc. of the USENIX 1994 Summer Conference, 1994.

[10] Gonzalez M., Serra A., Martorell X., Oliver J., Ayguade E., Labarta J., Navarro N., "Applying interposition techniques for performance analysis of OpenMP parallel applications," Proc. 14th International Parallel and Distributed Processing Symposium, IPDPS 2000, IEEE Computer Society, 2000.

[11] The Condor Team, "Condor High Throughput Computing", http://www.cs.wisc.edu/condor/.

[12] Serra A., Navarro N. and Cortes T., "DITools: Application-level Support for Dynamic Extension and Flexible Composition," Proc. of the USENIX Technical Conference, June 2000.

[13] Torres M., Liu Z., Florez G., Vaughn R. and Bridges S., "Attacking a High Performance Computer Cluster," Proc. of the 15th Annual Canadian Information Technology Security Symposium, May 2003.

[14] ParaDyn, "An Application Program Interface (API) for Runtime Code Generation," http://www.paradyn.org/html/manuals.html.

[15] M.I.T., "Kerberos: the network authentication protocol," http://web.mit.edu/kerberos/www/ .

[16] Neumann B. C., Ts'o, "Kerberos: An Authentication Service for Computer Networks," IEEE Communications, 32(9), September 1994.

[17] Klog, "Backdooring Binary Objects," Phrack Magazine Vol 56, 2000. http://www.phrack.org/.

[18] Silvio Cesare, "Shared Library Call Redirection Via ELF PLT Infection," Phrack Magazine Vol 56, 2000. http://www.phrack.org/.

[19] ODBC, http://msdn.microsoft.com/library/default.asp?url=/lil us/odbc/htm/odbc_part_1.asp