VIRUS ANALYSIS 2

Slamdunk

Péter Ször and Frédéric Perriot
Symantec Security Response, USA

The Slammer worm targets versions of *Microsoft SQL Server 2000* products, as well as *MSDE 2000* and related packages. The outbreak began on 25 January 2003 (GMT). According to early reports, the worm had a very significant presence around the world in less than one hour, and the peak time of the worm lasted for about three hours. During the worm's initial outbreak, Internet users experienced large percentage packet drops that developed into a large-scale DoS attack.

The worm exploits a stack-based overflow that occurs in a DLL implementing the SQL Server Resolution Service. This DLL (ssnetlib.dll) is used by the *SQL Server* service process called SQLSERVR.EXE. The vulnerability had been reported to *Microsoft* by David Litchfield (*NGSSoftware*), along with a few others. Furthermore, exploit code was made available at a *BlackHat* conference in 2002 and it is clear that this code was used as a base from which to develop the worm.

Exploit Setup

The *SQL Server* process listens on TCP as well as UDP ports. The worm targets UDP port 1434, sending a special request (0x04) specified as the first character of the payload. In the datagram this is followed by a specially crafted 'string' that contains the worm code. The worm code is extremely small – 376 bytes, which is the shortest binary worm known today. (376 bytes is the length of the UDP datagram without the protocol headers.)

Since the worm can use a UDP packet for the attack, it is probable that the source IP address of the original attacker was spoofed. The worm spreads to randomly generated IP addresses and, as a result, it is very difficult to determine from which country the attack originated.

The vulnerable function in ssnetlib.dll (as implemented in *SQL Server 2000*) is nested two levels deep inside a thread associated with the incoming request. The function is supposed to build a string for a Registry access by concatenating three strings into a 128-byte buffer. This string will be built on the stack and there are no input validations for the size of the middle string parameter. Strings 1 and 3 are constant and located in the ssnetlib.dll.

(String 1) 'SOFTWARE\Microsoft\Microsoft SQL Server\'

(String 2) String passed in the datagram (starts after the 0x04 type field)

(String 3) '\MSSQLServer\CurrentVersion'

As a result, whenever a string that is too long is passed to the function, the stack is corrupted (smashed). String 2 is an *SQL Server* instance name. According to the *Microsoft Knowledge Base* this string should be 16 characters long at most. However, this is neither enforced in the server, nor even in some of the common clients.

The worm has been crafted carefully. Its code is not only compact but it contains no zeros. This is because the buffer is used as a string parameter to an sprintf() library function call. As a result of the overflow a concatenated string will build on the stack where string 2 is the worm body itself.

Getting Control

Since the worm cannot contain zeros the author uses a lot of 01 filler bytes. Furthermore, attempts are made to use addresses that do not contain any zeros and, in some cases, the code uses XOR to mask zero bytes, which is a known shell code technique.

The worm starts with a header posing as local variables of the buggy function. A new return address (0x42B0C9DC) follows these filler bytes. This address is a pointer to a JMP ESP instruction inside SQLSORT.DLL, another module of the *SQL Server* process.

To make sure the vulnerable function will give control to the worm body, the header section of the worm also uses dummy ('crash test dummies') values (0x42AE7001) to replace function arguments on the stack. It is necessary to do this because these arguments are used after the call to sprintf() triggering the overflow. Failure to replace these arguments would cause an exception and thus the function would not return normally. When the function returns, control flows to the JMP ESP instruction which jumps on the stack to the location immediately after the hijacked return address. The first instruction will be a short jump around fake function arguments to the main worm code.

Initialization

The local variables within the worm header section could change during the time between the actual faulty sprintf() and the function return to the worm body, which means that the worm's header could become corrupted. Thus the worm will rebuild this area first to make sure that its header section remains constant for the next attack. Since the query type field (0x04) is missing from the top of the worm on the stack it is also rebuilt by pushing a 0x04000000 DWORD whose high byte is referenced by the replication code later.

Now the worm needs only a few functions to call. Following the original exploit code the worm's author uses the import address directory of SQLSORT.DLL to make calls to LoadLibraryA() and GetProcAddress() function calls.

This routine is compatible with different Service Pack releases and patches of *SQL Server*. Therefore GetProcAddress()'s code is checked first to be sure that it is the proper function entry point.

Then the worm gets access to the handles (base addresses) of WS2_32.DLL and KERNEL32.DLL. Next it gets the addresses of socket(), sendto() and GetTickCount() APIs, which is all it needs to replicate.

Replication

The replication method is extremely simple. The worm sends 376 bytes to UDP port 1434 to randomly generated IP addresses in an endless loop. This will cause the server CPU usage to increase and thousands of packets will be sent, effectively causing a DoS attack and at the same time compromising a large number of new systems around the world. The random number used to generate IP addresses is a variant of the *Microsoft Basic* random number generator. It uses the same multiplier. This results in sufficient randomness in the distribution of targeted systems.

Conclusion

A patch had been available for six months to cover both this vulnerability and others related to it (see *Microsoft Security Bulletins MS02-039* and *MS02-061*). Patches would block the worm's attack effectively if applied properly, but they are often too costly to deploy in large corporations. It should also be noted that the patching process was not an easy one, due to the large number of *Microsoft* and third-party products that include *SQL Server* as a component.

Although *SQL Server* would offer various user rights for the installation of the server process, such a server process often enjoys system context or admin privileges. This will provide attackers with access to any resources on the system, since the hijacked thread will run with significant privileges to do further damage on the system.

It appears that modern behaviour-blocking countermeasures will need to take place at host-level to provide a last line of defence to mitigate attacks like these. Such host-based products might be the key to slowing down similar attacks in the future.

Slammer

Size:	376 bytes (worm does not exist as a file on the system).
Aliases:	W32.SQLExp.Worm, SQL Slammer Worm, DDOS.SQLP1434.A, W32/SQLSlammer, Sapphire, W32/SQLSlam-A.
Payload:	None. Large-scale DoS attacks occur as a side-effect of replication.