SOURCEfire

HTTP IDS EVASIONS REVISITED

www.sourcefire.com

Sourcefire, Inc.
9212 Berger Road
Suite 200
Columbia, MD 21046

September 2004

Table of Contents

HTTP IDS Evasions Revisited

By Daniel Roelker

Abstract - ***This paper describes two general IDS evasion techniques and applies them to the HTTP protocol. These techniques are illustrated using some older types of HTTP evasions and some new HTTP evasions.***
The different types of evasions occur in both the Request URI portion of the HTTP protocol and by using the protocol standard in HTTP/1.0 and HTTP/1.1.

The evasions within the Request URI address evasion types possible in encoding and obfuscating the URL and parameter fields in the Request URI. The various methods of valid URL encodings for both the Apache and Internet Information Server are explained and examples given for each type of encoding. HTTP IDS evasions are also demonstrated using the HTTP protocol properties against the IDS. These evasions incorporate the request pipeline property and the content-encoding header.

This paper should help explain how HTTP IDS evasions work and give the reader enough knowledge to generate their own HTTP IDS evasions using these general principles and examples.

Index Terms - ***computer security, hypertext transfer protocol, intrusion detection, web scanning, intrusion detection evasions***

I. Introduction

HTTP IDS evasions have been popular since Rain Forest Puppy's (RFP) web scanner, whisker, was first released to the public [1]. Many of the original HTTP IDS evasions were contained in that first release, from multiple slashes that would obfuscate directories, to the more advanced evasions, like inserting HTTP/1.0 in the URL to evade an algorithm that an IDS might use to find the URL in a packet.

Besides the evasions that whisker presented, there were other types of HTTP obfuscations that were being propagated as well. One of these was obfuscating a URL by using an absolute URI vs. a relative URI [2]. While these other types of evasions were interesting, they were not as evasive or popular as the basic whisker scans.

The next popular evasion came about with the public release by RFP of the UTF-8 unicode encoding exploit for the Microsoft Internet Information Server (IIS) [3]. Besides being a serious vulnerability for IIS, the unicode exploit also presented an encoding method for URLs in a way that had not been implemented in IDSs. Up until this point, most IDSs had instituted safeguards against the previous whisker evasions of ASCII encoding and directory traversal, but did not protect against UTF-8 encoding of Unicode code points. One of the more professional write-ups that explained this type of HTTP IDS evasion was done by Eric Hacker [4]. Some of the insights in Hacker's paper are examined and explained in this paper as well. We will take the points that Hacker illustrates and delve into what these encodings mean and how they can work together to provide more bizarre encodings.

The other type of HTTP IDS evasions that are covered in this paper utilizes the HTTP protocol properties. One of these evasions uses the property of request pipelining. The other evasion uses the content-encoding header to encoding HTTP request parameters in a request payload.

II. IDS HTTP Protocol Analysis

In order for an IDS to handle URL attacks, the IDS must inspect the HTTP URL field for malicious attacks. The two most popular IDS inspection methodologies, pattern matching and protocol analysis, currently behave similarly because each methodology must search for malicious URLs and this entails some form of pattern matching and some form of HTTP protocol analysis.

In the beginning, the differences between these two methodologies were what you would expect. The protocol analysis methodology only searched the URL field of the HTTP stream for malicious URLs, while the pattern matching methodology searched the whole packet for the malicious URL.

The two methodologies performed similarly until the malicious URLs started to be encoded and obfuscated. At this point, the protocol analysis methodology merely had to add the appropriate decoding algorithms to the URL field. They had already built in HTTP protocol decoding to their engine. But the pattern matching methodology had no way of knowing which part of the packet to normalize. The pattern matching methodology had to incorporate some form of protocol analysis to find the URL field so that it could apply the appropriate decoding algorithms. A form of HTTP protocol analysis was added to the pattern matching methodology and the two methodologies once again began to behave similarly.

Because of the current similarities in these IDS methodologies, the HTTP IDS evasions that are discussed here apply to both types.

The first general IDS evasion is invalid protocol parsing. For example, if the HTTP URL is not found correctly then the malicious URLs will not be detected if

they are encoded. The reason being that if the IDS does not find the URL, it cannot decode it.

If the URL is found correctly, the IDS must know the proper decoding algorithms, otherwise the URL will again be decoded incorrectly. This is the second general type of IDS evasion, invalid protocol field decoding.

a) Invalid Protocol Parsing

IDS evasions that use invalid protocol parsing are demonstrated by RFP's whisker[1] and Bob Graham's SideStep[5]. The difference between these two programs are that whisker used flawed IDS protocol parsing to evade detection, whereas SideStep used valid aspects of application layer protocols to evade IDSs that had implemented naïve protocol decoders.

In this spirit, invalid protocol parsing evasions are particularly effective against two HTTP protocol fields, the URL and the URL parameters.

For example, if the IDS HTTP decoder assumes that there is only one URL per HTTP request packet, then if two URLs are sent in one packet, the IDS does not parse the second URL correctly. This is explained in the section on request pipelining evasions.

b) Invalid Protocol Field Decoding

Invalid protocol field decoding tests an IDS capability in the various types of encoding and normalization that is capable in a specific protocol field.

In the case of HTTP, this is most clearly seen in the URL field. An IDS can be tested for compliance to HTTP RFC encoding standards and also against the unique encoding types for different web servers, like IIS. If the IDS cannot decode certain types of URL encoding, then the attacker will use these encodings to bypass detection of malicious URLs.

Another method of invalid protocol field decoding for HTTP is through directory obfuscation. Directory obfuscation is accomplished through the manipulation of directory properties. For example, /cgi-bin/phf can be manipulated using multiple slashes instead of one slash, or it could use directory traversals to obfuscate the exact directory path.

It is important to realize that directory obfuscation can only obscure a malicious URL if the IDS looks for a URL that includes at least one directory besides the file to access. In the instance of our attack example, /cgi-bin/phf, directory obfuscation will work because the IDS is looking for the "phf" file in the "cgi-bin" directory. However, if the IDS is looking for just the "phf" file, the directory obfuscation would not work, since there is no directory path in that particular content.

III. Invalid Protocol Field Decoding

URL obfuscation starts out with the various types of encoding methods that HTTP servers accept. Admittedly, most of the encoding types are attributed to the IIS server, but for the sake of completeness, every type of encoding should be tested against each HTTP server.

The idea behind using URL encoding for obfuscating web attacks stems from the lack of research in most IDS methodologies to adequately define and implement the different encoding types for web servers.

If an IDS cannot decode an encoded type for a web server, then the IDS cannot tell whether a URL is malicious. Both pattern matching and protocol inspection IDS technologies have this problem.

There are only two RFC standards for encoding a Request URI: hex encoding and UTF-8 Unicode encoding. These two methods are encoded using the '%' character to escape a one encoded byte. It should also be noted that these are the only two URL encoding types that Apache accepts.

Most of the other encoding types that we will be looking at are server specific and non-RFC compliant. The Microsoft IIS web server falls in this category.

URL obfuscations are also covered in this section and follow the different encodings.

a) Hex Encoding

The hex encoding method is one of the RFC compliant ways for encoding a URL. It is also the simplest method of encoding a URL. The encoding method consists of escaping a hexadecimal byte value for the encoded character with a '%'. If we wanted to hex encode a capital A (ASCII map hexadecimal value of 0x41), the encoding would look like the following:

- %41 = 'A'

b) Double Percent Hex Encoding

Double percent hex encoding is based on the normal method of hex encoding. The percent is encoded using hex encoding followed by the hexadecimal byte value to be encoded. To encode a capital A, the encoding is:

- %2541 = 'A'

As can be seen, the percent is encoded with the %25 (this equals a '%'). The value is then decoded

again with the value this time being %41 (this equals the 'A').

This encoding is supported by Microsoft IIS.

c) Double Nibble Hex Encoding

Double nibble hex encoding is based on the standard hex encoding method. Each hexadecimal nibble value is encoded using the standard hex encoding. For example, to encode a capital A, the encoding would be:

- %%34%31 = 'A'

The normal hex encoding for A is %41. How double nibble hex encoding works, is that the hexadecimal nibble values are each encoded in the normal hex encoding format. So, the first nibble, 4, is encoded as %34 (the ASCII value for the numeral 4), and the second nibble, 1, is encoded as %31 (the ASCII value for the numeral 1).

In the first URL decoding pass the nibble values are translated into the numerals 4 and 1. Since the 4 and 1 are preceded by a %, the second pass recognizes %41 and decodes that as a capital A.

This encoding is supported by Microsoft IIS.

d) First Nibble Hex Encoding

First nibble hex encoding is very similar to double nibble hex encoding. The difference is that only the first nibble is encoded. So a capital A, instead of being encoded %%34%31 for double nibble hex, is encoded in the following example using first nibble hex encoding:

- %%341 = 'A'

As before, during the first URL decoding pass the %34 is decoded as the numeral 4, which leaves %41 for the second pass. During the second pass, the %41 is decoded as a capital A.

This encoding is supported by Microsoft IIS.

e) Second Nibble Hex Encoding

Second nibble hex encoding is exactly the same as first nibble hex encoding, except the second hexadecimal nibble value is encoded with normal hex encoding. So a capital A is encoded as:

- %4%31 = 'A'

The %31 gets decoded to a numeral 1 in the first decoding pass, which then the %41 gets decoded as a capital 'A'.

This encoding is supported by Microsoft IIS.

f) UTF-8 Encoding

1. UTF-8 Introduction

UTF-8 encoding allows values larger than a single byte (0-255) to be represented in a byte stream. HTTP web servers use UTF-8 encoding to represent Unicode code points that are outside of the ASCII code point range (1 – 127).

UTF-8 works by giving special meaning to the high-bits in a byte. A UTF-8 two and three byte UTF-8 sequence is illustrated below:

110xxxxx 10xxxxxx	(two byte sequence)
1110xxxx 10xxxxxx 10xxxxxx	(three byte sequence)

The first byte in a UTF-8 sequence is the most important because it contains how many bytes are in the complete UTF-8 sequence. This is determined by counting the high bits up to the first zero. In the two byte sequence example, the first byte contains two high bits set followed by a zero. So this is indeed a two-byte UTF-8 sequence. The rest of the bits after the zero in the first UTF-8 byte are bits in the final value to be computed.

UTF-8 bytes following the initial byte all have the same format of setting the high bit followed by a zero. Two bits are used to identify a UTF-8 byte, and six bits are used in computing the value.

To encode UTF-8 in the URL, the UTF-8 sequence is escaped with a percent for each byte. A UTF-8 encoded character is illustrated as, %C0%AF = '/'.

2. Unicode Code Point Introduction

UTF-8 encoding is used to encode Unicode code point values. Code point values are usually contained in the range of 0 – 65535. Any code point value above 127 uses UTF-8 encoding in HTTP URLs.

Unicode code point values from 0 – 127 map one to one with ASCII values. That leaves about 65408 values to represent other characters in languages like Hungarian or Japanese.

Usually these languages have their own Unicode code page that represents the characters that they need. Unicode code point values are derived from Unicode code pages. Each Unicode code page can have a unique set of values, so as Unicode code pages change so do the characters that a Unicode code point represent. If the wrong code page is used to interpret Unicode code points, then the results are invalid. This is concept is very important in URL encoding as seen in the next section.

3. Bringing the Evasion Together

There are three characteristics of using UTF-8 encoding to represent Unicode code points that lend themselves to confusion among IDSs.

The first characteristic is that UTF-8 can encode a single code point or ASCII value in more than one way. This has been fixed in the current Unicode standard, but is still prevalent in web servers (excluding Apache). For example, a capital letter A is encoded in a two byte UTF-8 sequence as:

- %C1%81 (11000001 10000001 = 1000001 = 'A')

The capital letter A can also be encoded in a three byte UTF-8 sequence as:

- %E0%81%81 (11100000 10000001 10000001 = 1000001 = 'A')

So, using UTF-8 to encode ASCII characters leads to many different representations.

The second characteristic is that some non-ASCII Unicode code points also map to ASCII characters. For example, the Unicode code point 12001 could map to a capital letter A. The only way to know which code points map to ASCII characters is to either read the Unicode code map or test all the different Unicode code points against a server. Currently, the only web server that is known to do this is the Microsoft IIS server.

The third characteristic is related to the second characteristic. If the Unicode code map is changed or is not known, then interpreted Unicode code points are invalid. The reason this is important is because IIS web servers in China, Japan, Poland, etc. use different code pages, so if an IDS is not aware of which code page a web server is running then the URL decoding efforts of UTF-8 are invalid. If an IDS is not configurable as to what Unicode code pages to run for particular servers, then any web server that does not run the code page that the IDS has knowledge of is evadable.

g) UTF-8 Bare Byte Encoding

UTF-8 bare byte encoding is the same as UTF-8 encoding, except that the UTF-8 byte sequence is not escaped with a percent. The byte sequence is sent with the actual bytes. If an A was sent across, it would be:

- 0xC1 0x81 = 'A'

This type of encoding is only known to run on the Microsoft IIS server.

h) Microsoft %U Encoding

Microsoft %U encoding presents a different way to encode Unicode code point values up to 65535 (or two bytes). The format is simple; %U precedes 4 hexadecimal nibble values that represent the Unicode code point value. The format is illustrated as:

- %UXXXX

For example, a capital A could be encoded as:

- %U0041 = 'A'

This is encoding is supported by Microsoft IIS.

i) Mismatch Encoding

Mismatch encoding uses different encoding types to represent an ASCII character and is not a unique encoding by itself. The mismatch encoding combines the various types of encoding to encode a single character.

For example, let's encode a capital A using the Microsoft %U encoding method. But since IIS will do a double decode on a URL, we can use some of the other methods to encode the %U method. For instance, we can encode the U in the %U method with a normal hex encoding. So a simple %U0041 becomes %%550041. We can then encode the 0041 in normal hex encoding, or we could pick another type of encoding.

Here's a more complex encoding mismatch that works against an IIS server, try to figure out which ASCII character this encoding represents:

- %U0025%550%303%37

IV. Invalid Protocol Parsing

a) URL Evasion Using Request Pipelines

The request pipeline evasion is a type of invalid protocol parsing evasion. It obscures the URI by using the protocol characteristics of a request pipeline in version 1.1 of the HTTP protocol.

The request pipeline standard allows a web client to send several requests within a single packet. This is different and should not be confused with the HTTP keep alive header. Request pipelines send several requests all in one packet, whereas HTTP keep alive just keeps the TCP stream open for more requests.

We use the request pipeline feature to embed several URLs in one packet. Most IDSs will parse the first URL correctly, but fail to parse the other URLs. This leaves an avenue for evasion, because the other URLs can now be trivially encoded and any content matches looking for malicious URLs will fail, because the IDS did not decode these other URLs.

For example, the following payload uses request pipelining to evade URL detection:

- GET / HTTP/1.1\r\nHost: \r\n\r\nGET /foobar.html \r\nHost: \r\n\r\nGET /cgi%2Dbin%2Fph%66 HTTP/1.1\r\nHost: r\n

b) Parameter Evasion using POST and Content-Encoding

Another common HTTP protocol field that contains malicious data or attacks is the URL parameter field. This is the field where most database and cgi type attacks occur, and most IDSs contain signatures to detect malicious parameter keys and values.

A simple way to evade an IDS would be to encode the parameters as the URL is encoded. But most IDSs already apply URL decoding methods to the parameter field as well.

What we do is use a POST request to move the parameter field to the end of the HTTP request header section. At this point, the parameter field is in plaintext and an IDS could easily pick out malicious content here. Instead, we use the header option, Content-Encoding, to encode the parameter field in base64 encoding.

At this point, the parameter field has been encoded in base64 and the request is sent across the wire. Now the IDS not only needs to parse the POST request correctly, but it needs to decode the parameter field using base64 before inspecting the parameter field.

If the IDS actually decoded the parameter field in a POST request with base64, the decoding effort would be very time consuming. It would also lend itself to a DOS attack by sending lots of POST requests with large parameter fields that would need to be decoded.

V. Conclusion

Two general techniques are used in HTTP IDS evasions. These techniques are invalid protocol parsing and invalid protocol field encoding. If an IDS is unaware of a type of HTTP protocol field encoding it cannot correctly decode the URL and evasions will occur. This is the type of technique that the various encoding discussed used.

If an IDS does not have adequate knowledge of the HTTP protocol, it can also be evaded. The request pipeline and content encoding evasions uses this type of technique.

By examining an IDS protocol decoder, most evasions can be generated with these two general techniques.

VI. Appendix

a) URL Encoder Program

A tool that illustrates these various decodings and obfuscations are available at www.idsresearch.org. Both a Windows GUI application is available, along with a command line tool for *NIX and Windows.

b) Unicode Code Page Mapper

A tool that dumps the Unicode code pages and code points on a Microsoft system is available at www.idsresearch.org.

VII. Acknowledgment

RFP and Bob Graham have really pioneered application-layer IDS evasions. Thanks for your insightful work.

I'd also like to give a big thanks to Marc Norton for the many technical discussions we've had and for encapsulating the URL encoder technology into a great Win32 GUI.

VIII. References

1. RFP. (1999, Dec. 30). "A look at whisker's anti-IDS tactics". [Online]. Available: http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html
2. Author Unknown. (2002, Jan. 13). "How to obscure any URL". [Online]. Available: http://www.pc-help.org/obscure.htm
3. RFP. (2000, Oct. 17). "IIS %c1%1c remote command execution," *Win2k Security Advice Mailing List*. [Online]. Available: http://archives.neohapsis.com/archives/win2ksecadvice/2000-q4/0037.html
4. E. Hacker. (2001, Jan. 3). IDS Evasion with Unicode. SecurityFocus Infocus. [Online]. Available: http://www.securityfocus.com/infocus/1232
5. R. Graham. (2000). "SideStep: IDS evasion tool". [Online]. Available: http://www.robertgraham.com/tmp/sidestep.html

Daniel J. Roelker is a lead developer for the Snort Development Team and IDS researcher at Sourcefire, Inc., where he works on the Snort 2.0 detection engine, application protocol decoders, portscan detection, and code integration and auditing. He was previously employed as a lead developer on the Dragon Network IDS at Enterasys and at Johns Hopkins Applied Physics Laboratory with the Department of Defense in Information Operations.

About Sourcefire, Inc.

Sourcefire, Inc., the world leader in real-time network defense solutions, is transforming the way organizations manage and minimize network security risks with its 3D Approach - Discover, Determine, Defend - to securing real networks in real-time. The company's ground-breaking network defense system unifies intrusion and vulnerability management technologies to provide customers with the most effective network security available. Founded in 2001 by the creator of Snort, Sourcefire is headquartered in Columbia, MD and has received numerous accolades including being named a 2004 Company to Watch by Network Computing Magazine and selected as one of the Red Herring Top 100 privately held companies. At work in leading Fortune 100 and government agencies, the names Sourcefire and founder Martin Roesch have grown synonymous with innovation and intelligence in network security. For more information about Sourcefire, please visit www.sourcefire.com.