Lost in Translation (ASP's HomoXSSuality)

Lost in Translation (ASP's HomoXSSuality) - ma1, hackademix.net.

- Published: date not stated
- Original: <https://hackademix.net/2010/08/17/lost-in-translation-asps-homoxssuality/>
- Preserved from: https://hackademix.net/2010/08/17/lost-in-translation-asps-homoxssuality/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Classic ASP is the old server-side web scripting technology based on VBScript, now superseded by ASP.NET, which lots of developers, including myself, learned to hate in the nineties when, for mysterious reasons, a certain customer decided he needed the whole "Enterprise" Microsoft 3-tiers stack (IIS/COM+/SQL Server). Luckily enough, nobody asks you to build anything new using ASP these days (even though there's always some insanely unmaintainable VBScript code out there which badly needs maintenance), but this technology, albeit agonizing, yet found a way to come back and make me sad again.

Some days ago this blog post, talking about a bypass method for NoScript's Anti-XSS filter, called for my attention (*not* thanks to its author).

Even though it's not very clear from that piece of writing, the issue at hand is quite simple but, in my opinion, outrageously stupid and annoying. I'm gonna call it **"HomoXSSuality"** (even though most LGBT people I know is neither simple, nor stupid nor annoying), because homoglyps and homophones conspire to make XSS (and SQL injection) attacks easier to pull.

Like any other server-side web programming framework, ASP gives developers some means to extract "parameters" (name/value pairs) from the HTTP requests, stored either in the query string or in the POST data. For instance, if an ASP script is invoked using the URL http://some.site.com/my_heroes.asp?**name=Giorgio%20Maone&hero=%E1%BD%99%CF%80%CE%B1%CF%84%CE%AF%CE%B1**, parameters can be extracted by code like this:

[code lang="VB"] Dim Name, Hero Name = Request("name") Hero = Request("hero") [/code]

At runtime, the *Name* variable will contain "Giorgio Maone", while *Hero* will be set to "á½™Ï€Î±Ï„Î¯Î± ". This contrived example show also how "special" characters, such as space or Greek alphabet letters, are escaped by standard percent encoding, i.e. by taking the UTF-8 hexadecimal representation of the string and prefixing each byte with a "%" character: specifically, â€œ â€

translates to â€œ%20â€, and “á½™Ï€Î±Ï„Î¯Î±” to “%E1%BD%99%CF%80%CE%B1%CF%84%CE%AF%CE%B1”. This is the translation you can obtain from the *encodeURIComponent()* ECMAScript function, and the recommended way of escaping URLs. An older and never standardized method, implemented by the now deprecated JavaScript *escape()* function, produces more or less the same output for ASCII strings, but uses the UTF-16 representation prefixed with “%u” for higher (beyond ASCII) Unicode strings: for instance, â€œ â€ still stays â€œ%20â€, but “á½™Ï€Î±Ï„Î¯Î±” becomes “%u1F59%u03C0%u03B1%u03C4%u03AF%u03B1”.

NoScript’s Anti-XSS filter, while processing HTTP requests, does recognizes and properly handle both these encoding styles, and many more. Any web security filter should be able to do it, because web applications usually consume data that has been automatically decoded by their runtime environment.

But Classic ASP adds a perverse twist to its parameter decoding routines. The *Request()* API apparently assumes that developers and/or browsers and/or users are too stupid to handle non-ASCII Unicode characters (e.g. greek alphabet letters) by themselves, thus it tries to protect them from such execrable things by automatically translating any non-ASCII character into the ASCII counterpart which *resembles* it the most; when no suitable replacement can be picked, with either “?” or “ï¿½” (arbitrarily, it seems). So “%u1F59%u03C0%u03B1%u03C4%u03AF%u03B1”, rather than “á½™Ï€Î±Ï„Î¯Î±”, becomes a quite ugly “?pat?a”. As you can see, while the replacement choice is mainly homoglyphic (Î±â†’a, Ï„â†’t), it may also follow homophonic criteria (Ï€â†’p).

To figure out the whole range of Unicode-ASCII transliterations performed by ASP, I needed to write an ad hoc program mixing VBScript and JavaScript, and I also used it to automatically generate the *ASPIdiocy.js* mappings file that can be found in recent NoScript packages.

A short essay here, to give you just a taste of this madness:

(0x100) ~= A(0x41)
Ä(0x101) ~= a(0x61)
Ä‚(0x102) ~= A(0x41)
Äƒ(0x103) ~= a(0x61)
Ä„(0x104) ~= A(0x41)
Ä…(0x105) ~= a(0x61)
Ä†(0x106) ~= C(0x43)
Ä‡(0x107) ~= c(0x63)
Äˆ(0x108) ~= C(0x43)
Ä‰(0x109) ~= c(0x63)
ÄŠ(0x10a) ~= C(0x43)
Ä‹(0x10b) ~= c(0x63)
ÄŒ(0x10c) ~= C(0x43)
Ä(0x10d) ~= c(0x63)
ÄŽ(0x10e) ~= D(0x44)
Ä(0x10f) ~= d(0x64)
Ä(0x110) ~= ï¿½(0xfffd)
Ä‘(0x111) ~= d(0x64)
Ä’(0x112) ~= E(0x45)
Ä“(0x113) ~= e(0x65)
Ä”(0x114) ~= E(0x45)
Ä•(0x115) ~= e(0x65)
Ä–(0x116) ~= E(0x45)
Ä—(0x117) ~= e(0x65)
Ä˜(0x118) ~= E(0x45)
Ä™(0x119) ~= e(0x65)
Äš(0x11a) ~= E(0x45)
Ä›(0x11b) ~= e(0x65)
Äœ(0x11c) ~= G(0x47)
Ä(0x11d) ~= g(0x67)
Äž(0x11e) ~= G(0x47)
ÄŸ(0x11f) ~= g(0x67)
Ä (0x120) ~= G(0x47)
Ä¡(0x121) ~= g(0x67)
Ä¢(0x122) ~= G(0x47)
Ä£(0x123) ~= g(0x67)
Ä¤(0x124) ~= H(0x48)
Ä¥(0x125) ~= h(0x68)
Ä¦(0x126) ~= H(0x48)
Ä§(0x127) ~= h(0x68)
Ä¨(0x128) ~= I(0x49)
Ä©(0x129) ~= i(0x69)
Äª(0x12a) ~= I(0x49)

Ä«(0x12b) ~= i(0x69)
Ä¬(0x12c) ~= I(0x49)
Ä(0x12d) ~= i(0x69)
Ä®(0x12e) ~= I(0x49)
Ä¯(0x12f) ~= i(0x69)
Ä°(0x130) ~= I(0x49)
Ä±(0x131) ~= i(0x69)
Ä´(0x134) ~= J(0x4a)
Äµ(0x135) ~= j(0x6a)
Ä¶(0x136) ~= K(0x4b)
Ä·(0x137) ~= k(0x6b)
Ä¸(0x138) ~= ?(0x3f)
Ä¹(0x139) ~= L(0x4c)
Äº(0x13a) ~= l(0x6c)
Ä»(0x13b) ~= L(0x4c)
Ä¼(0x13c) ~= l(0x6c)
Ä½(0x13d) ~= L(0x4c)
Ä¾(0x13e) ~= l(0x6c)
Å(0x141) ~= L(0x4c)
Å‚(0x142) ~= l(0x6c)
Åƒ(0x143) ~= N(0x4e)
Å„(0x144) ~= n(0x6e)
Å…(0x145) ~= N(0x4e)
Å†(0x146) ~= n(0x6e)
Å‡(0x147) ~= N(0x4e)
Åˆ(0x148) ~= n(0x6e)
ÅŒ(0x14c) ~= O(0x4f)
Å(0x14d) ~= o(0x6f)
ÅŽ(0x14e) ~= O(0x4f)
Å(0x14f) ~= o(0x6f)
Å(0x150) ~= O(0x4f)
Å‘(0x151) ~= o(0x6f)
Å”(0x154) ~= R(0x52)
Å•(0x155) ~= r(0x72)
Å–(0x156) ~= R(0x52)
Å—(0x157) ~= r(0x72)
Å˜(0x158) ~= R(0x52)
Å™(0x159) ~= r(0x72)
Åš(0x15a) ~= S(0x53)
Å›(0x15b) ~= s(0x73)
Åœ(0x15c) ~= S(0x53)
Å(0x15d) ~= s(0x73)
Åž(0x15e) ~= S(0x53)
ÅŸ(0x15f) ~= s(0x73)

Å¢(0x162) ~= T(0x54)
Å£(0x163) ~= t(0x74)
Å¤(0x164) ~= T(0x54)
Å¥(0x165) ~= t(0x74)
Å¦(0x166) ~= T(0x54)
Å§(0x167) ~= t(0x74)
Å¨(0x168) ~= U(0x55)
Å©(0x169) ~= u(0x75)
Åª(0x16a) ~= U(0x55)
Å«(0x16b) ~= u(0x75)
Å¬(0x16c) ~= U(0x55)
Å(0x16d) ~= u(0x75)
Å®(0x16e) ~= U(0x55)
Å¯(0x16f) ~= u(0x75)
Å°(0x170) ~= U(0x55)
Å±(0x171) ~= u(0x75)
Å²(0x172) ~= U(0x55)
Å³(0x173) ~= u(0x75)
Å´(0x174) ~= W(0x57)
Åµ(0x175) ~= w(0x77)
Å¶(0x176) ~= Y(0x59)
Å·(0x177) ~= y(0x79)
Å¸(0x178) ~= ï¿½(0xfffd)
Å¹(0x179) ~= Z(0x5a)
Åº(0x17a) ~= z(0x7a)
Å»(0x17b) ~= Z(0x5a)
Å¼(0x17c) ~= z(0x7a)
âŒ©(0x2329) ~= <(0x3c)
ã€ˆ(0x3008) ~= <(0x3c)
ï¼œ(0xff1c) ~= <(0x3c)
Ê¹(0x2b9) ~= '(0x27)
Ê¼(0x2bc) ~= '(0x27)
Ëˆ(0x2c8) ~= '(0x27)
â€²(0x2032) ~= '(0x27)
ï¼‡(0xff07) ~= '(0x27)

As you can see in the end, I could list 3 different homoglyphs for **<** (*less than*, ASCII 0x27) and 5 for **'** (*apostrophe*, ASCII 0x3c). Anybody with a bit of familiarity with XSS or SQL injection has already guessed where I'm going...

Classic ASP translates the query string parameter value **%u3008scr%u0131pt%u3009%u212fval(%uFF07al%u212Frt(%22XSS%22)%u02C8)%u2329/scr%u0131pt%u232A** to

<script>eval('alert("XSS")')</script>

which, if echoed back, is executed as a JavaScript block by web browsers.

Any "sane" web server runtime (either a recent IIS with ASP.NET or Apache with PHP/Python/Ruby, or a Java Servlet Container, or you pick yours) either leaves the **%u...** stuff alone (because this escaping style is deprecated), or translates the whole into

ã€ˆscrÄ±ptã€‰â„¯val(ï¼‡alâ„¯rt("XSS")Ëˆ)âŒ©/scrÄ±ptâŒª

which obviously has no other meaning than "funny text", to any decent web browser.

This undocumented (AFAIK) Classic ASP "feature" (which was sooo good and smart that Microsoft itself dropped it in ASP.NET) can severely screw up with any anti-XSS filter. It does with Google Chrome's, it does not with Microsoft IE8's (unsurprisingly, since the original mess came from Redmond), it does not anymore with NoScript's, since version 2.0.2rc2.

Of course, it may also be used to bypass Web Application Firewalls (WAFs), which, ironically enough, are often deployed to "virtually patch" XSS and SQL injection bugs in hardly maintainable applications, just like the ones developed with Classic ASP: this blog had been just created when it witnessed a tragicomic case involving the United Nations.

So, how many WAFs out there can actually resist when **HomoXSSuality** calls?

[image: image]

By ma1

Hacker, atheist, humanist, dad, mozillian, security breaker and builder, creator of NoScript, casting spells at the Tor Browser. He/him.

View all of ma1's posts.