Vicious XSS

For readers who doubt the relevance of KV's advice, witness the XSS (cross-site scripting) attack that befell the social networking site MySpace.com in October (http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391). This month Kode Vicious addresses just this sort of XSS attack. And in response to the reader's question below, it's a good thing cross-site scripting is not abbreviated CSS, as the MySpace hacker used CSS (cascading style sheets) to perpetrate his XSS attack. That would have made for one confusing story, eh? Read on for KV's take on this XSS madness.

Dear KV,

I know you usually spend all your time deep in the bowels of systems with C and C++ (at least that's what I gather from reading your columns), but I was wondering if you could help me with a problem in a language a little further removed from low-level bits and bytes, PHP. Most of the systems where I work are written in PHP, and, as I bet you've already worked out, those systems are Web sites. My most recent project is a merchant site that will also support user comments. Users will be able to submit reviews of products and merchants to the site. One of the things that our QA team keeps complaining about is possible XSS attacks. Our testers seem to have a special ability to find these, so I wanted to ask you about this. First, why is XSS such a big deal to them; second, how can I avoid having such bugs in my code; and finally, why is cross-site scripting abbreviated XSS instead of CSS?

Cross with Scripted Sites

Dear CSS,

First, let's get something straight: I may spend a lot of time with C and C++, but I object to the use of the word

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

bowels in this context. My job is bad enough without having this image of literally working in the bowels of anything.

Let me answer your last question first, since it's the easiest. The reason that cross-site scripting is abbreviated XSS is the same as the reason that I spell code as kode. Programmers and engineers think they're clever and like to put their mark on things by changing the language, in particular turning every possible term they coin into some acronym that only they know. It is one of the side effects of specialization, which we will leave alone just now, before my more literate friends come after me with torches and pitchforks.

Now, back to what I think we can both agree are your more serious queries. Cross-site scripting is the ability to inject JavaScript into a site and then to have the site send that scripting code on to the user. There are actually many risks involved in cross-site scripting attacks because the JavaScript code can do many different malicious things. For example, the code can completely rewrite the displayed HTML, which in your case means that someone else would be able to overwrite the reviews that the user submitted—probably not an ability you would like others to have. Another example is that the malicious code can steal the user's cookies, and cookies are often used in Web applications to identify the user. If the user's cookies get stolen, then the attacker can become the user and perhaps take over the account. If your site uses cookies in this way, this is a pretty big risk. So, you can see why QA gets their knickers in a twist. To be honest, I'm surprised they never bothered to explain just why this was a risk, or maybe they just assumed you knew better.

Winding up with a cross-site scripting bug is almost always the result of not doing proper input validation. Since you say that you've read earlier columns, then you must know that I don't trust users, and neither should you. When designing a Web site, you just have to accept that with millions of potential users, some percentage of those people who use your site will attack it. It's the way the world is: some people are just jerks. This means we

have to design to handle not only regular users, but also the jerks.

In the case of working with user reviews, I'm sure that some marketing type has demanded that users be able not only to upload plain text such as, "Wow, this merchant is great, I got all my stuff in just 24 hours, I'd buy from them again!", but also to use HTML such as `<b><font color="red">Wow</font></b>`, which is full of bold and red, and if they could get away with it, dancing GIFs, because marketing people seem to get paid based on the number of incredibly stupid features they add to a project. I direct this comment not at all marketing people, just those who think that an interface with 20 buttons is far better than one with 10. I believe Dante wrote about such people, and that there was a special level of hell for them. The problem is how to let some subset of HTML through, at least for the bold, underline, and perhaps colors, and not to allow anything else. The approach you're looking for is a whitelist, and in pseudocode a function to clean up a string to allow only these tags looks something like the function in figure 1.

The `string_clean` function has several features I'd like to point out. First, it is very strict, probably stricter than you'll be able to get away with when dealing with marketing. The allowed characters are all the upper- and lowercase roman alpha-

betic characters, all 10 digits, and four types of punctuation: periods, commas, question marks, and exclamation points. No parentheses and no braces are allowed, which

```
//
// Function: string_clean
// Input: an untreated string
// Output: A string which contains only upper and lower case letters,
//         numbers, simple punctuation ( . , ! ? ) and three types of HTML
//         tags, bold, italic and underline.

string string_clean(string dirty_string)
{
    string return_string = "";

    array html_white_list = ['<b>', // bold
                             '<i>', // italic
                             '<u>']; // underline

    array punctuation_white_list = [ '.', ',', '!', '?' ]

    for (i = 0, i < len(dirty_string), i++)
    {
        if (isalpha(dirty_string[i])) {
            return_string += dirty_string[i];
            continue;
        } else if (isnumber(dirty_string[i])) {
            return_string += dirty_string[i];
            continue;
        } else {
            if (dirty_string[i] in $punctuation_white_list) {
                return_string += dirty_string[i];
                continue;
            } else if (dirty_string[i] == '<') {
                $tag = substring(dirty_string, i, i + 2);
                if ($tag in $html_white_list) {
                    return_string += $tag;
                } else {
                    return_string += ' ';
                    i += 2;
                }
            }
        }
    }
    return_string += ' ';
}

return return_string;
```

FIG 1 The `string_clean` Function

protects against the case of `{` getting through. In HTML only three tags are allowed: bold (`<b>`), italic (`<i>`), and underline (`<u>`). The function is implemented as a whitelist, which means that only the allowed characters are appended to the returned string. Many string-cleaning routines are implemented as blacklists, which is to say they list what is *not* allowed. The problem with blacklists and whitelists was treated in the letter from Input Invalid (“Kode Vicious Reloaded,” *ACM Queue*, March 2005), so I won’t go over the details again. For those interested in efficiency, note that we check for the most common case first, a letter; the next most common, a number; and then the least common cases, which are punctuation and finally the allowable tags. I picked this order so that the code would append the character and go round the loop most quickly in the most common cases, which hopefully gives us the best performance. You should also note that the default action is to ignore the input character and simply to append a space to the return string. We append a string so that we can see where there might have been illegal text. Simply removing the offending character makes it easy to miss where the attack may have been.

Of course, this is a simple first pass at a filtering function and would have to be tailored to your environment, but I hope it gives you a shove in the right direction. To

protect against such attacks, you must not only code such a function, but you and everyone on your team must also use it for each and every case of user input. I cannot count the number of times that a suitable filtering function existed in a library; yet, for some perverse reason, the engineers working on the product decided simply to ignore it or go around it because they felt they were better at treating the input themselves. I have one piece of advice for such people: don’t do it. If you require some special abilities with a particular piece of input, then either extend the function or create a new one—one that can also be used consistently. It will save you a lot of time and headaches in the long run.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor’s degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

© 2005 ACM 1542-7730/05/1200 \$5.00