Studies in secure programming for admins

DANGEROUS INPUT

Like a poison apple, a Web program that is tasty on the surface may contain a highly dangerous core. Admins who do their own programming need to follow secure programming practices to avoid the bitter taste of insecurity. BY DOMINIK VOGT

This article describes some special kinds of program input that administrators who maintain Websites often contend with. I'll present case studies that take a close look at problems and solutions related to cross-site scripting, malicious email addresses, and buffer overflows.

If there is a theme to this discussion, it is that developers need to carefully validate all input and the relationships between various items of input. Assume all input is guilty until proven innocent. And the more complex this input is, the more important it is to code carefully to anticipate the actions of would-be intruders.

In the case of complex problems, the temptation is to pass the buck, leaving input validation to the programs that follow ("The guys in development will have done their homework!") Unfortunately, it is often the case that everyone involved in the process thinks the same way, and the gaping hole stays open. Every admin, and every developer, is well advised to listen to their conscience from time to time, remembering that their output will be the input for a program somewhere downstream.

www.sxc.hu

Case 1: Cross-Site Scripting

Developing websites is probably not one of the admin's more classical chores. But in small to medium-sized businesses, you still find admins tinkering with public websites. System admins who are not trained as programmers may underestimate the dangers that abound in this environment. One of the biggest dangers is a special kind of input: HTML-formated text (or text formated in another Web language.) A harmless-looking facade can conceal malevolent Javascript code.

The attacker's true target is not the web server that stores the document, but the client-side browser that opens it. In a cross-site scripting attack, the malevolent scripts run on the victim's machine while the victim's browser is surfing a different site. The script runs in the context and with the privileges of the site the browser is currently rendering (Figure 1).

Unfriendly Guestbook

In a simple guestbook, users are allowed to compose short entries which are then published on the hosting website. If the people behind the website allow arbitrary input, a malevolent hacker might hide a script in their entry that pops up an ad in the visitor's browser:

```
Nice page. Good work!
<!-- <script>
  window.open("http://debian.org/");
</script> -->
```

Now this might sound harmless, but depending on the website, attackers might have something more nasty up their sleeves. For instance, they might deface the page, steal cookies with session IDs and assume the identity of the user, or use fake websites to phish for other people's passwords.

The stakes are particularly high on pages that have something to do with money: banks sites or online casinos, for example. The Secure Programming Cookbook devotes a whole chapter to this topic and provides numerous examples.

What allows cross-site scripting to happen is the lack of user input validation. Web developers often overlook the importance of this issue, just because nobody happens to have attacked their web server so far. The web server is just the intermediary that passes on the exploit as is to the user without being harmed itself. The same thing applies to cross-site authentication attacks. (See the article in this issue titled "Strange Phishing: Stopping the cross-site authentication attack.")

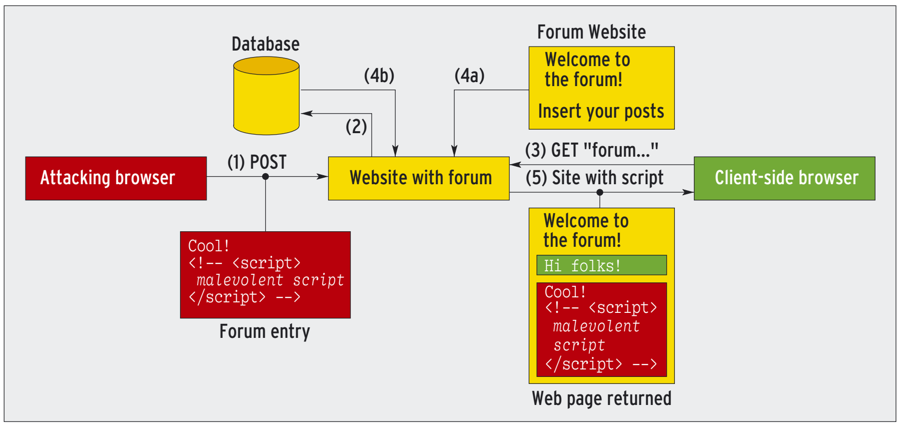
Countermeasures

Cross-Site scripting may be widespread, but it is fairly easy to combat. A secure application would first remove any script from the input, or simply reject any dubious offerings point blank. If you are dealing with HTML, any script will be enclosed in formating tags, that is, between angled brackets *<…>*. A brutal but effective approach would be to convert any meta-characters to a harmless HTML encoding before continuing processing (see Table 1).

But if you would like to give your visitors an opportunity to use simple markups, this option is not open to you. Javascript code can hide more or less anywhere, in the *<img …>*, *<div …>* or *<body…>* tags, for example. Check out [1] for several examples.

Blacklisting is not an option, but whitelisting might prove more effective. The idea behind whitelisting is to just leave the HTML tags that are known to be harmless, and to reject everything else as potentially malevolent, including any tags with attributes. For example, the tags in Table 2 are harmless.

If this approach to the problem of cross-site scripting does not give you enough freedom, you might like to pass the problem on to your developers, or to read a few books on the subject [1] [2].



```
Cool!
<!-- <script>
 malevolent script
</script> -->
```

Figure 1: The attacker injects a malevolent script into a forum posting (1), and the server stores the entry (2). Some time later, a user visits the forum (3); the server puts the Website framework (4a) and the database entries together (4b) and serves the page up to the visitor (5). The browser then runs the script (cross-site scripting).

Listing 1: escape-html.c

```
#include "inputguardian.c"
#include <stdio.h>

int main(int argc, char **argv)
{
  char *ret = inputg_escape_html(argv[1]);
  if (ret != 0)
    printf("%s", ret);
  return !ret;
}
```

Tag Filtering

An approach for C and C++: The Gate Guardian [3] library gives developers a useful way of avoiding trouble. The *inputg_escape_html()* function escapes the HTML meta-characters < > "& as *&lt;*, *&gt;*, *&quot;*, and *&amp;*, but leaves harmless tags such as *<h1>* or *<br>* as is. Script-free links such as *<a href="http://Foo">* are left unchanged. The call looks like this:

```
#include <inputguardian.c>
[...]
char *escaped;
escaped = inputg_escape_html(input);
```

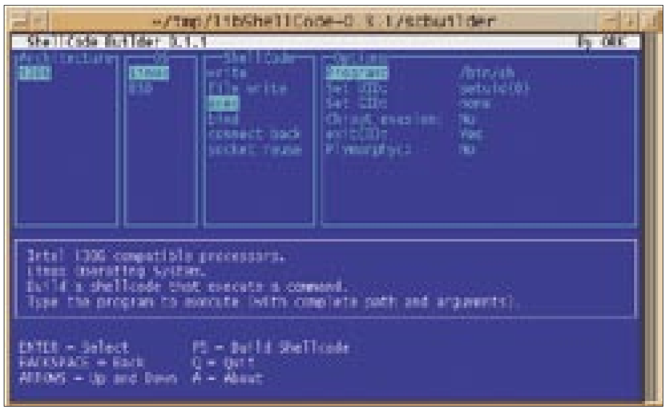
The function returns a pointer to the modified HTML document, which points to a memory area allocated by a call to *malloc()*, which the developer will need to free later with a call to *free(escaped)*. The function will return a null pointer if you forget the memory allocation.

Masked Ball

The *inputg_escape_all_html()* variant converts all HTML meta-characters to

Table 1: HTML Meta-Characters

Character	HTML Encoding
<	&lt;
>	&gt;
&	&amp;
"	&quot;

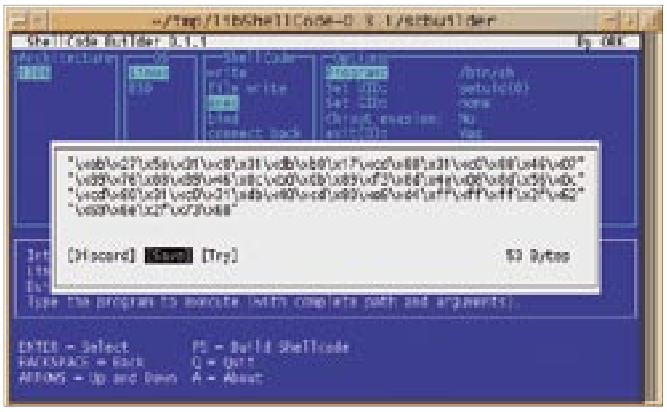


```
Shell Code Builder 0.1.1                                          by [illegible]

Intel i386 compatible processors.
Linux Operating System.
Build a shellcode that execute a command.
Type the program to execute (with complete path and arguments).

ENTER - Select          F5 - Build ShellCode
BACKSPACE - Back        Q - Quit
ARROWS - Up and Down    A - About
```

Figure 2a: The Scbuilder UI is a front-end for Libshellcode. Thanks to the UI, anyone can create lean and fast shell code for almost any kind of application.



```
[illegible]

[Discard] [Save] [Try]                                53 Bytes
```

Figure 2b: If needed, Scbuilder can store the resulting shell code as C program code. The admin can no longer accept excuses such as "the vulnerability is hard to exploit" for overflows.

harmless entity encoding without exceptions. *inputg_escape_html_with_tag_table()* allows users to define the list of permissible tags themselves.

Be careful, though: the Gate Guardian functions are taken from *spc_escape_html()* in the Secure Programming Cookbook [4], but the original is, unfortunately, riddled with bugs. As a replacement for the faulty code, you might like to check out my archive with corrected versions at [5]. New bugfixes have arrived since Part 3 went to press, so if you use this archive, you might like to download the latest version.

An approach for the shell: The simplest approach is a small wrapper that uses the Gate Guardian function. Admins can then call the *escape-html* program in Listing 1 by calling *ESCAPED=`escape-html "$INPUT"`* from their own shell scripts.

Case 2: Email Addresses

Administrative scripts or programs often expect input in the form of email addresses. Admins typically specify the addresses themselves, for example, if they need a script to give them a detailed update on what has been going on. In other cases, users specify the mail addresses, as in a bug report form, for example. Unless you happen to be developing a mail client or mail server, you might not bother validating the address input. But it pays to be paranoid!

RFC 2822 [6] and its predecessors precisely define what an address needs to look like. The new standard stipulates that mail programs must be able to process older address formats. The aim is to support continued use of older programs. In a do-it-yourself form, admins can do without extra bits such as routes in mail addresses and so on (*<Route: Address>*). A lesser known fact is that mail addresses can include nested comments.

No Nonsense Approach

In most cases, shell programmers can just squash this problem with a no nonsense approach. Anything that fits into the *Name@Domain* pattern gets through, anything else doesn't. You can also restrict the valid character set for the name and domain entries, and you can call *egrep* to back you up:

```
echo "$ADDR" | ⮐
egrep '^[a-zA-Z0-9_+-.]+??
@[a-zA-Z0-9-]+⮐
(\.[a-zA-Z0-9-]+)*$' || ??
exit 1
```

Table 2: Script-free HTML Tags					
abbr	acronym	b	bdo	big	blink
blockquote	br	center	cite	code	dd
del	dfn	dir	dl	dt	em
h1	h2	h3	h4	h5	h6
hr	i	ins	kbd	li	menu
nobr	ol	p	plaintext	pre	q
s	samp	small	spacer	strike	strong
sub	sup	tt	u	ul	var

If this approach lacks the kind of elegance you prefer, check out Jeffrey Friedl's Regex book *Mastering Regular Expressions* [7] for an elegant regular expression that will cover the complete RFC, with the exception of nested comments.

A Perl program that uses this expression to validate mail addresses is available at [8]. It is quite easy to use:

```
perl email-opt.pl ⮐
"$ADDR" >/dev/null || exit 1
```

A good approach for C and C++ programmers is to use the Gate Guardian *inputg_is_simple_email_address()* function to implement a no nonsense egrep test in C:

```
#include <inputguardian.c>
[...]
int rc;
rc = ⮐
inputg_is_simple_email_address⮐
(addr)
if (rc == 0)
  exit(1);
```

It is not a good idea to try to implement the more precise Perl program in C or C++, because complex parsing can easily lead to errors or unwanted side-effects. The best idea in this situation would be to use a ready-made parser or call the Perl script as an external program.

Case 3: Buffer Overflows

Buffer overflows are closely related to format strings and have been around a lot longer. In both cases, the stack is the

“If you use Linux, you need Linux Magazine!”
LINUX MAGAZINE
WWW.LINUX-MAGAZINE.COM/SUBS

major (but not the only) objective targeted by malevolent hackers. A buffer overflow occurs when a program attempts to write data to a memory area (buffer) that is too small to store that data. This attempt leads to the program overwriting memory addresses that have been allocated to other tasks.

Shell Code

Things start to turn nasty when our hacker, Fred, injects executable code into the buffer. The code typically just launches a shell: *execve("/bin/sh", 0, 0);*, allowing Fred to hijack the vulnerable user account. It is by no means easy to write shell code, but script kiddies very rarely need to do so. Instead, they turn to libraries such as Libshellcode. The library comes with a small Ncurses UI titled Scbuilder (Figures 2a and 2b).

Attacks on program logic are more subtle and very hard to prevent; as an example, Fred might attempt to overwrite other local variables stored at higher memory addresses. In our example, this would mean overwriting the *len* variable or the variables used by the calling function. In real life scenarios, hackers have managed to hijack remote systems simply by overwriting variables containing the UID that a program wanting to drop privileges changed to. For more details on buffer overflows see [9].

Roots

Memory overflows always involve program code reading, writing, and copying data. The C source code typically has a string function (*strcat(), strcpy()*…), a function for formatted input and output (*sprintf(), scanf()*…), a file access function (*fread(), gets()*…), or inaccurate pointer arithmetic.

A call to *gets()* is nearly always a bad idea: the developer can't tell the function how much memory has been allocated. In other words, *gets* comes with a built in buffer overflow. But *scanf(input, "%s", buffer)* is not much better: scanf stores a string that it parses in a buffer no matter how big the buffer is. Table 3 shows you a better approach.

Listing 2 shows a common error in file name handling. The call to *getcwd()* in line 9 stores the working directory path in the *abs_path* variable. So far, so good, because the buffer is big enough. But in line 10, *strcat()* fails to check if the buffer is full, and it might just carry on writing outside the buffer's boundaries. A path longer than *PATH_MAX* is another source of danger (depending on your file system type). The program should detect this error (return value of *ERANGE*) and handle the situation gracefully.

Make Way!

C and C++ programmers need to either use dynamic memory allocation to ensure that enough memory is available, or restrict the length of myinput to the amount of space they have. The latter approach is often preferable for smaller programs, as it is easier to implement. Table 3 shows you the secure variants of some standard library functions.

C++ developers should avoid the vulnerable functions and use the *std::string* string class, as well as the << and >> stream operators.

Keep It Simple

The diversity and complexity of the issues described here may have given you some idea of how difficult in can be to separate harmless input from malevolent input. The problem of secure programming brings to mind an old adage that is often quoted by successful admins and software developers: keep it simple.

Occasional programmers typically do not have enough time to plum the depths of formats and their complexities. If this sounds like you, you should be looking to allow a subset of all permissible input and just ditch the rest.

The strategies described in this article will give you a headstart on writing safe and sensible code. Always protect your program input, and you'll put the bad apples firmly where they belong: in the trash can. ■

Listing 2: path_max.c

```
01 #include <limits.h>
02 #include <string.h>
03 #include <unistd.h>
04
05 int main(void)
06 {
07   char abs_path[PATH_MAX];
08
09   getcwd(abs_path, PATH_MAX);
10   strcat(abs_path, "/
   filename");
11   /* ... */
12
13   return 0;
14 }
```

Table 3: Avoiding Buffer Overflows

Wrong	Right
sprintf(buf, "%s", str)	*sprintf(buf, "%99s", str)* or *snprintf(buf, 100, "%s", str)*
scanf("%s", str)	*scanf("%99s", str)*
gets(buf)	*fgets(buf, 100, stdin)*
strcat(buf, str)	*strncat(buf, str, 99)*
strcpy(buf, str)	*strncpy(buf, str, 99); buf[99] = 0;*

INFO

[1] David A. Wheeler, "Secure Programs HOWTO": *http://www.dwheeler.com/secure-programs/*

[2] Sverre H. Huseby, *Innocent Code*: Wiley, ISBN 0-470-85744-7

[3] Dominik Vogt, Gate Guardian: *http://sourceforge.net/projects/gateguardian/*

[4] Viega and Messier, *Secure Programming Cookbook*, O'Reilly, ISBN 0-596-00394-3: *http://www.secureprogramming.com*

[5] Source code for the *Secure Programming Cookbook*: *http://www.dominikvogt.de/de/index.html#Links* (in German)

[6] RFC 2822, "Internet Message Format": *http://www.ietf.org/rfc/rfc2822.txt*

[7] Jeffrey E. F. Friedl, *Mastering Regular Expressions*, O'Reilly, ISBN 0-596-00289-0: *http://www.oreilly.com/catalog/regex2/*

[8] Sample scripts from the regex book: *http://examples.oreilly.com/regex/*

[9] Aleph One, "Smashing The Stack For Fun And Profit", Phrack Vol. 7, Issue 49, File 14: *http://www.phrack.org/show.php?p=49&a=14*

THE AUTHOR

Dipl.-Math. Dominik Vogt is an experienced software developer and system administrator. At present, he is working as a freelance IT consultant and specializing in software security. On his leisure time, Dominik enjoys tinkering with the Fvwm window manager.