BAI3110 ,Data Input Issues

If someone you didn't know came to your door and offered you something to eat, would you eat it? No, of course you wouldn't. So why do so many applications accept data from strangers without first evaluating it? It's safe to say that most security exploits involve the target application incorrectly checking the incoming data or in some cases not checking the data at all. So let me be clear about this: you should not trust data until the data is validated. Failure to do so will render your application vulnerable.

Rule #1
All input is evil until proven otherwise…
Typically, the moment you forget this rule is the moment you are attacked.

Rule #2
Data must be validated as it crosses the boundary between untrusted and trusted environments.
By definition, trusted data is data you or an entity you explicitly trust has complete control over; untrusted data refers to everything else. In short, any data submitted by a user is initially untrusted data.

The reason I bring this up is many developers balk at checking input because they are positive that the data is checked by some other function that eventually calls their application and they don't want to take the performance hit of validating the data more than once.

But what happens if the input comes from a source that is not checked or the code you depend on is changed because it assumes some other code performs a validity check? Or what happens if an honest user simply makes an input mistake that causes your application to fail? Keep this in mind as we discuss some potential vulnerabilities and exploits.

Modern processors are orders of magnitude faster than the previous generations. The current bottleneck in computing comes from your network connection, and then your hard disk to a lesser extent. Current CPU speeds are as high as 2 or 3 Gigabytes per second while network speeds (Internet) are typically only about 1 Megabyte (or up to 3,000 times slower).

Performance is rarely a problem when checking user input. Even if it is, *no system is less reliably responsive than a hacked system*.

Note: If you still don't believe all input should be treated as unclean, I suggest you randomly choose any ten past vulnerabilities. You'll find that in the majority of cases the exploit relies on malicious input. I guarantee it!

The Issue

Many applications today distribute functionality between client and server machines or between peers, and many developers rely on the client portion of the application to provide specific behavior. However, the client software, once deployed, is no longer under the control of the developer, nor the system administrators, so there is no guarantee that requests made by the client came from a valid client. Hence, the server can never trust the client request.

The critical issue is trust and, more accurately, attributing too much trust to data provided by an untrusted entity.

Misplaced Trust

When you are analyzing designs and code, it's often easy to find areas of vulnerability by asking two simple questions. Do I trust the data at this point? And what are the assumptions about the validity of the data? Let's take a buffer overrun example. Buffer overruns occur for the following reasons:

1. The data came from an untrusted source (an attacker!).
2. Too much trust was placed in the data format, in this case the buffer length.
3. A potentially hazardous event occurs, in this case, the untrusted buffer is written into memory.

Take a look at this code. What's wrong with it?

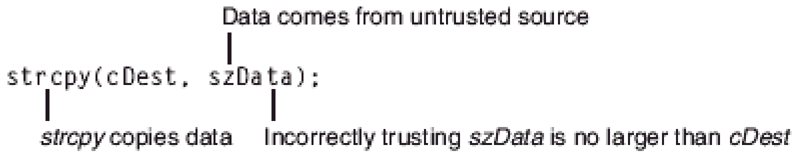
```
void CopyData(char *szData)
     {
     char cDest[32];
     strcpy(cDest, szData);

     // use cDest
     …
     }
```

Surprisingly, there may be nothing wrong with this code! It all depends on how CopyData() is called and whether szData comes from a trusted source. For example the following code is safe:

```
Char *szNames[] = {"David","Cheryl","Liu");
CopyData(szNames[1]);
```

The code is safe because the names are hard coded and therefore each string does not exceed 32 characters in length; hence, the call to strcpy() is always safe.

However, if the sole argument to CopyData, saData, comes from an untrusted source, such as a network socket or a file then strcpy will copy the data until it hits a NULL character. And if the data is greater than 32 bytes in length, the cDest buffer is overrun and any data above the buffer is clobbered.


Data comes from untrusted source
strcpy(cDest, szData);
strcpy copies data
Incorrectly trusting szData is no larger than cDest

If you remove any of the conditions, the chance of a buffer overrun is zero. Remove the memory copying nature of strcpy, and you cannot overflow a buffer, but that's not realistic because a non memory copying version is worthless.
If the data always comes from a trusted source, for example, from a highly trusted user or a well well-secured file you can assume the data is well-formed.

Finally, if the code makes no assumption about the data and validates it prior to copying it, then once again, the code is safe from buffer overruns.

If you check the data validity prior to copying it, it doesn't matter whether the data came from a trusted source. This leads to just one acceptable solution to make this code secure. First check that the data is valid, and do not trust it until the legality is verified:

```
void CopyData(char *szData)
     {
     char cDest[32];
     int  bufflength = sizeof(cDest);

     if (szData != NULL)
          if ( bufflength > strlen(szData) )
               strncpy(cDest, szData, bufflength)

     // use cDest
     …
     }
```

The code still copies the data (strncpy) but because the szData argument is untrusted the code limits the amount of data copied to cDest. You might think this is a little too much code to check the data validity, but it's not, a little extra code can protect the application from serious attack. Besides, if the insecure code is attacked you'd need to make the fixes in the example anyhow, so get it right the first time.

Permissions play a large part in determining if sources are trusted or untrusted. Suppose you had a registry key that determines which file to update with log information and that key had an ACL (Access Control List) of Everyone (Full Control). How much trust can you assign to the data in that key?

None!

Because anyone can update the filename. For example, an attacker could change the filename to c:\boot.ini. The data in the key could be trusted more if the ACL is Administrator (Full Control) and Everyone (Read); in that case only the Administrator can change the data, and administrators are trusted entities of the system.

With proper ACL's, the concept of trust is transitive: because you trust the administrators and because only administrators can change the data, you trust the data.

A Strategy for Defending Against Input Attacks

The simplest and by far the most effective way to defend your application from input attacks is to validate the data before performing and further processing. To achieve these goals, you should adhere to the following strategies:

- Define a trust boundary around the application
- Create an input chokepoint

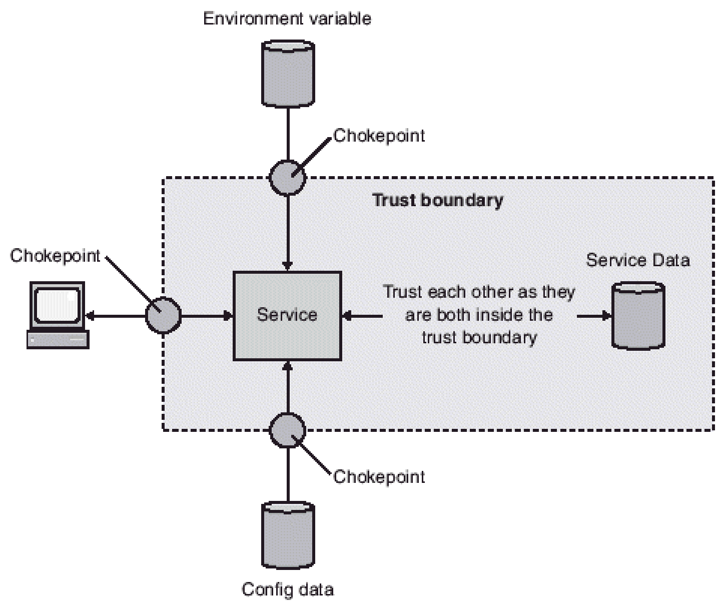
First, all applications have a point in the design where the data is believed to be well formed and safe because it has been checked. Once the data is inside that trusted boundary, there should be no reason to check it again for validity.

That said, the principle of defense in depth dictates that you should employ multiple layers of defense in case a layer is compromised, and that is quite true.

But you have to strike a balance between security and performance of your application. The balance will depend on the sensitivity of the data and the environment in which the application operates.

Next, you should perform the check at the boundary of the trusted code base. You must define that point in the design, it must be in one place, and no input should be allowed into the trusted code base without going through that chokepoint.

Note that you can have multiple chokepoints, one for each data source (Web, registry, file system, configuration files, and so on), but data from one source should not enter the trusted code base through any other chokepoint but the one designated for that source.



Note that the service and the service's data store have no chokepoint between them. That's because they're both inside the trust boundary and data never enters the boundary from any data source without being validated first at a chokepoint. Therefore, only valid data can flow from the service to the data store and vice versa.

How to Check Validity

When checking input for validity, you should follow this rule: look for valid data and reject everything else. The principle of failing securely means you should deny all access until you determine the request is valid. You should look for valid data and not look for invalid data for two reasons:

- There might be more than one valid way to represent the data.
- You might miss an invalid data pattern.

The first point is explained in more detail in "Canonical Representation Issues." It's common to escape characters in a way that is valid, but it's also possible to hide invalid data by escaping it. Because the data is escaped, your code might not detect that it's invalid.

The second point is very common indeed. Let me explain by way of a simple example. Imagine your code takes requests from users to upload files, and the request includes the filename. Your application will not allow a user to upload executable code because it could compromise the system. Therefore, you have code that looks a little like this:

```
bool IsBadExtension(char *szFilename) {
    bool fIsBad = false;

    if (szFilename) {
        size_t cFilename = strlen(szFilename);
        if (cFilename >= 3) {
            char *szBadExt[]
                = {".exe", ".com", ".bat", ".cmd"};
            char *szLCase
                = _strlwr(_strdup(szFilename));

            for (int i=0;
                i < sizeof(szBadExt) / sizeof(szBadExt[0]);
                i++)
                if (szLCase[cFilename-1] == szBadExt[i][3] &&
                    szLCase[cFilename-2] == szBadExt[i][2] &&
                    szLCase[cFilename-3] == szBadExt[i][1] &&
                    szLCase[cFilename-4] == szBadExt[i][0])
                    fIsBad = true;}
    }
    return fIsBad;
}
bool CheckFileExtension(char *szFilename) {
    if (!IsBadExtension(szFilename))
        if (UploadUserFile(szFilename))
            NotifyUserUploadOK(szFilename);
}
```

What's wrong with the code? IsBadExtension performs a great deal of error checking, and it's reasonably efficient. The problem is the list of "invalid" file extensions. It's nowhere near complete; in fact, it's hopelessly lacking. A user could upload many more executable file types, such as Perl scripts (.pl) or perhaps Windows Scripting Host files (.wsh, .js and .vbs), so you decide to update the code to reflect the other file types. However, a week later you realize that Microsoft Office documents can contain macros (.doc, .xls, and so on), which technically makes them executable code. Yet again, you update the list of bad extensions, only to find that there are yet more executable file types. It's a never-ending battle. The only correct way to achieve the goal is to look for valid, safe extensions and to reject everything else. For example, in the Web file upload scenario, you might decide that users can upload only certain text document types and graphics, so the secure code looks like this:

```
bool IsOKExtension(char *szFilename) {
    bool fIsOK = false;

    if (szFilename) {
        size_t cFilename = strlen(szFilename);
        if (cFilename >= 3) {
            char *szOKExt[] =
                {".txt", ".rtf", ".gif", ".jpg", ".bmp"};

            char *szLCase =
                _strlwr(_strdup(szFilename));

            for (int i=0;
                i < sizeof(szOKExt) / sizeof(szOKExt[0]);
                i++)
                if (szLCase[cFilename-1] == szOKExt[i][3] &&
                    szLCase[cFilename-2] == szOKExt[i][2] &&
                    szLCase[cFilename-3] == szOKExt[i][1] &&
                    szLCase[cFilename-4] == szOKExt[i][0])
                    fIsOK = true;
        }
    }

    return fIsOK;
}
```

As you can see, this code will not allow any code to upload unless it's a safe data type, and that includes text files (.txt), Rich Text Format files (.rtf), and some graphic formats. It's much better to do it this way. In the worst case, you have an annoyed user who thinks you should support another file format, which is better than having your servers compromised.

Using Regular Expressions for Checking Input

For simple data validation, you can use code like the code I showed earlier, which used simple string compares. However, for complex data you need to use higher-level constructs, such as regular expressions. The following C# code shows how to use regular expressions to replace the C++ extension-checking code. This code uses the *RegularExpressions* namespace in the .NET Framework:

```
using System.Text.RegularExpressions;
...
static bool IsOKExtension(string Filename)
{
    Regex r = new Regex(@"txt|rtf|gif|jpg|bmp$",RegexOptions.IgnoreCase);
    return r.Match(Filename).Success;
}
```

The same code in Perl looks like this:

```
sub isOkExtension($) {
    $_ = shift;
    return /txt|rtf|gif|jpg|bmp$/i ? -1 : 0;
}
```

The core of the expression is the string "`txt|rtf|gif|jpg|bmp$`".

We have looked at Regular Expressions previously in BAI586 so I will skip an extended lecture.

Be Careful of What You Find—Did You Mean to Validate?

Regular expressions serve two main purposes. The first is to find data; the second, and the one we're mainly interested in, is to validate data. When someone enters a filename, I don't want to find the filename in the request; I want to validate that the request is for a valid filename. Allow me to explain. Look at this pseudo code that determines whether a filename is valid or not:

```
RegExp r = [a-z]{1,8}\.[a-z]{1,3};
if (r.Match(strFilename).Success) {
    //Cool! Allow access to strFilename; it's valid.
} else {
    //Tut! tut! Trying to access an invalid file.
}
```

This code will allow a request only for filenames comprised of 1-8 lowercase letters, followed by a period, followed by 1-3 lowercase letters (the file extension). Or will it? Can you spot the flaw in the regular expression? What if a user makes a request for the c:\boot.ini file? Will it pass the regular expression check? Yes, it will. The reason is because the expression looks for any instance in the filename request that matches the expression. In this case, the expression will find the series of letters boot.ini within c:\boot.ini. However, the request is clearly invalid.

The solution is to create an expression that parses the entire filename to look for a valid request. In which case, we need to change the expression to read as follows:

```
^[a-z]{1,8}\.[a-z]{1,3}$
```

The ^ means start of the input, and $ means end of the input. You can best think about the new expression as "from the beginning to the end of the request, allow only 1-8 lowercase letters, followed by a period, followed by 1-3 lowercase letters, and nothing more." Obviously, c:\boot.ini is invalid because the : and \ characters are invalid and do not comply with the regular expression.

URLEncode Code Chart

The following is a chart of ascii values for 256 characters in URL-encoding form. These values can be used for URL-encoding non-standard letters and characters for display in browsers and plug-ins which support them. (The codes below are in hexadecimal format.

æ	%00	0	%30	`	%60		%90	À	%c0	ð	%f0
	%01	1	%31	a	%61	‘	%91	Á	%c1	ñ	%f1
	%02	2	%32	b	%62	’	%92	Â	%c2	ò	%f2
	%03	3	%33	c	%63	“	%93	Ã	%c3	ó	%f3
	%04	4	%34	d	%64	”	%94	Ä	%c4	ô	%f4
	%05	5	%35	e	%65	•	%95	Å	%c5	õ	%f5
	%06	6	%36	f	%66	–	%96	Æ	%c6	ö	%f6
	%07	7	%37	g	%67	—	%97	Ç	%c7	÷	%f7
backspace	%08	8	%38	h	%68	˜	%98	È	%c8	ø	%f8
tab	%09	9	%39	i	%69	™	%99	É	%c9	ù	%f9
linefeed	%0a	:	%3a	j	%6a	š	%9a	Ê	%ca	ú	%fa
	%0b	;	%3b	k	%6b	›	%9b	Ë	%cb	û	%fb
	%0c	<	%3c	l	%6c	œ	%9c	Ì	%cc	ü	%fc
c return	%0d	=	%3d	m	%6d		%9d	Í	%cd	ý	%fd
	%0e	>	%3e	n	%6e	ž	%9e	Î	%ce	þ	%fe
	%0f	?	%3f	o	%6f	Ÿ	%9f	Ï	%cf	ÿ	%ff
	%10	@	%40	p	%70		%a0	Ð	%d0		
	%11	A	%41	q	%71	¡	%a1	Ñ	%d1		
	%12	B	%42	r	%72	¢	%a2	Ò	%d2		
	%13	C	%43	s	%73	£	%a3	Ó	%d3		
	%14	D	%44	t	%74		%a4	Ô	%d4		
	%15	E	%45	u	%75	¥	%a5	Õ	%d5		
	%16	F	%46	v	%76	¦	%a6	Ö	%d6		
	%17	G	%47	w	%77	§	%a7		%d7		
	%18	H	%48	x	%78	¨	%a8	Ø	%d8		
	%19	I	%49	y	%79	©	%a9	Ù	%d9		
	%1a	J	%4a	z	%7a	ª	%aa	Ú	%da		
	%1b	K	%4b	{	%7b	«	%ab	Û	%db		
	%1c	L	%4c	|	%7c	¬	%ac	Ü	%dc		
	%1d	M	%4d	}	%7d	¯	%ad	Ý	%dd		
	%1e	N	%4e	~	%7e	®	%ae	Þ	%de		
	%1f	O	%4f		%7f	¯	%af	ß	%df		
space	%20	P	%50	€	%80	°	%b0	à	%e0		
!	%21	Q	%51		%81	±	%b1	á	%e1		
"	%22	R	%52	‚	%82	²	%b2	â	%e2		
#	%23	S	%53	ƒ	%83	³	%b3	ã	%e3		
$	%24	T	%54	„	%84	´	%b4	ä	%e4		
%	%25	U	%55	…	%85	µ	%b5	å	%e5		
&	%26	V	%56	†	%86	¶	%b6	æ	%e6		
'	%27	W	%57	‡	%87	·	%b7	ç	%e7		
(	%28	X	%58	ˆ	%88	¸	%b8	è	%e8		
)	%29	Y	%59	‰	%89	¹	%b9	é	%e9		
*	%2a	Z	%5a	Š	%8a	º	%ba	ê	%ea		
+	%2b	[	%5b	‹	%8b	»	%bb	ë	%eb		
,	%2c	\	%5c	Œ	%8c	¼	%bc	ì	%ec		
-	%2d	]	%5d		%8d	½	%bd	í	%ed		
.	%2e	^	%5e	Ž	%8e	¾	%be	î	%ee		
/	%2f	_	%5f		%8f	¿	%bf	ï	%ef		

Summary

We have spent a great deal of time outlining how to use regular expressions, but do not lose sight of the most important message of this chapter: trust input at your peril. In fact, do not trust any input until it is validated. Remember, just about any security vulnerability can be traced back to an application placing too much trust in the data,

When analyzing input, have a small number of entry points into the trusted code; all input must come through one of these chokepoints. Do not look for "bad" data in the request. You should look for good, well-formed data and reject the request if the data does not meet your acceptance criteria. Remember: you wrote the code for accessing and manipulating your resources; you know what constitutes a correct request. You cannot know all possible invalid requests, and that's one of the reasons you must look only for valid data. The list of correct requests is finite, and the list of invalid requests is potentially infinite or, at least, very very large.