SWE 781
Secure Software Design and Programming

Input Validation

Lecture 3

Ron Ritchey, Ph.D.
Chief Scientist
703/377.6704
Ritchey_ronald@bah.com

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Schedule (tentative)

Date	Subject
Sep 1st	Introduction (today) ; Chess/West chapter 1, Wheeler chapters 1,2,3
Sep 8th	Computer attack overview
Sep 15th	**Input Validation; Chess/West chapter 5, Wheeler chapter 5**
Sep 22nd	Buffer Overflows; Chess/West chapters 6, 7; Wheeler chapter 6
Sep 29th	Error Handling; Chess/West chapter 8; Wheeler chapter 9 (9.1, 9.2, 9.3 only)
Oct 6th	Privacy, Secrets, and Cryptography; Chess/West chapter 11; Wheeler chapter 11 (11.3, 11.4, 11.5 only)
Oct 13th	Columbus Recess
Oct 20th	Mid-Term exam
Oct 27th	Mid Term Review / Major Assignment Introduction
Nov 3rd	Implementing authentication and access control
Nov 10th	Web Application Vulnerabilities; Chess/West chapter 9,10
Nov 17th	Secure programming best practices / Major Assignment Stage Check ; Chess/West chapter 12; Wheeler chapters 7,8,9,10
Nov 24th	Static Code Analysis & Runtime Analysis
Dec 1st	The State of the Art (guest lecturer)
Dec 8th	TBD (Virtual Machines, Usability [phishing], E-Voting, Privilege Separation, Java Security, Network Security & Worms)

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Minor Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

PHF

- White pages directory service program
- Distributed with NCSA and Apache web servers
- Version up to NCSA/1.5a and apache/1.0.5 vulnerable to an invalid input attack
- Impact:
  - Un-trusted users can execute arbitrary commands at the privilege level that the web server is executing at
- Example URL illustrating attack
  - http://webserver/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

PHF Coding problems

- Uses popen command to execute shell command
  - User input is part of the input to the popen command argument
- Does not properly check for invalid user input
  - Attempts to strip out bad characters using the escape_shell_cmd function but this function is flawed. It does not strip out <new line> characters.
  - By appending a <new line> plus a shell command to an input field, and attacker can get the command executed by the web server

Copyright Ronald W. Ritchey 2008, All Rights Reserved

PHF Code Fragment

```
strcpy(commandstr, "/usr/local/bin/ph -m ");
if (strlen(serverstr)) {
      strcat(commandstr, " -s ");
      escape_shell_cmd(serverstr);
      strcat(commandstr, serverstr);
      strcat(commandstr, " ");
}
escape_shell_cmd(typestr);
strcat(commandstr, typestr);
if (atleastonereturn) {
      escape_shell_cmd(returnstr);
      strcat(commandstr, returnstr);
}

printf("%s%c", commandstr, LF);
printf("<PRE>%c", LF);

phfp = popen(commandstr,"r");
send_fd(phfp, stdout);

printf("</PRE>%c", LF);
```

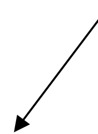
Copyright Ronald W. Ritchey 2008, All Rights Reserved

escape_shell_cmd code fragment

```
void escape_shell_cmd(char *cmd) {
    register int x,y,l;

    l=strlen(cmd);
    for(x=0;cmd[x];x++) {
        if(ind("&;`'\"|*?~<>^()[]{}$\\",cmd[x]) != -1){
            for(y=l+1;y>x;y--
                cmd[y] = cmd[y-1];
            l++; /* length has been increased */
            cmd[x] = '\\';
            x++; /* skip the character */
        }
    }
}
```

Notice: No %0a or \n character



Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Minor Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Some potential sources of input

- Command line
- Environment variables
  - Including PATH
- Files
  - File descriptors
  - Configuration files
  - Temporary Files
- Databases
- Network services
- Registry values
- System properties

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Command Line

- Many programs take input from the command line
- If the program runs at the privilege level of the user, there is not much of a security problem.
- setuid/setgid programs must treat command line arguments as coming from an untrusted user
- All arguments must be checked including arg0

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Arg0

```
#include <stdio.h>
int main(int argc, char * argv[]) {
   printf("%s\n", argv[0]);
   exit(1);
}
gcc -o argtest argtest.c

When run normally
/home/rritchey$ argtest
Argtest

But attacker can change arg0
   when calling execl so
execl("argtest", "blah", NULL)
blah
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Environment Variables

- Purpose is to maintain general state information
  - e.g. PATH, SHELL, USERNAME
- Normally, inherited from parent process
- Behavior is transitive
  - a secure program might call some other program and, without special measures, would pass potentially dangerous environment variables values on to the program it calls
- Calling program *can* override any environmental settings passed to called program

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Dangerous Env Variables

- Some environment variables are dangerous because many libraries and programs are controlled by environment variables in ways that are obscure, subtle, or undocumented
- Example: IFS
  - Used by the sh and bash shell to determine which characters separate command line arguments
  - If input validation strips out spaces an attacker could change IFS to something that was not stripped out (say Q) then create an input that will pass (rmQ-RQ*)

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Path Manipulation

- Path used to set directories to search when a command is issued

```
echo $PATH
   /sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
```

- Attacker can modify path to search in different directories

```
/hackscripts:/sbin:/usr/sbin:/bin:/usr/bin
```

- If the called program calls an external command, hacker can replace the command with an alternate program

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Path Recommendations

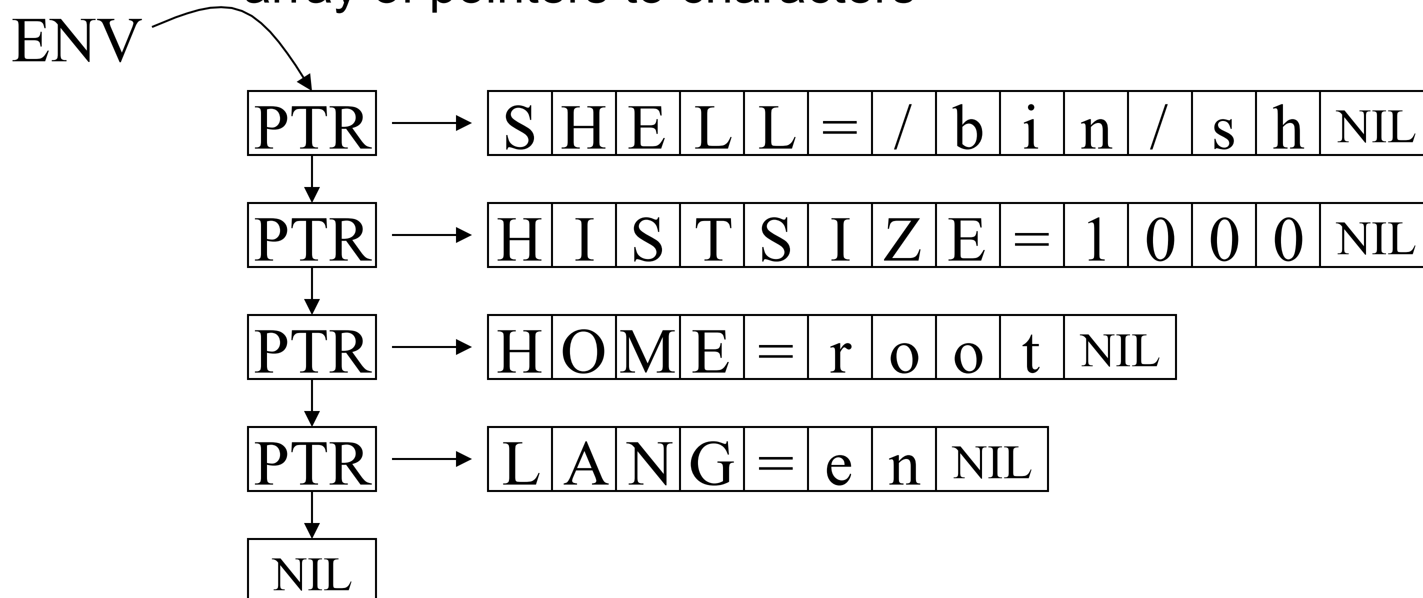
- Basic rule if possible: do not trust the path. Use fully qualified path names for security sensitive software
- Parse the path and locate the program/file you which to access. Check to see if it is in an expected location.
- Do not allow the current directory to be specified in the path
  - E.g.: **.**:/usr/bin:/bin:

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Environment Variable Storage

- Environment variables are internally stored as a pointer to an array of pointers to characters

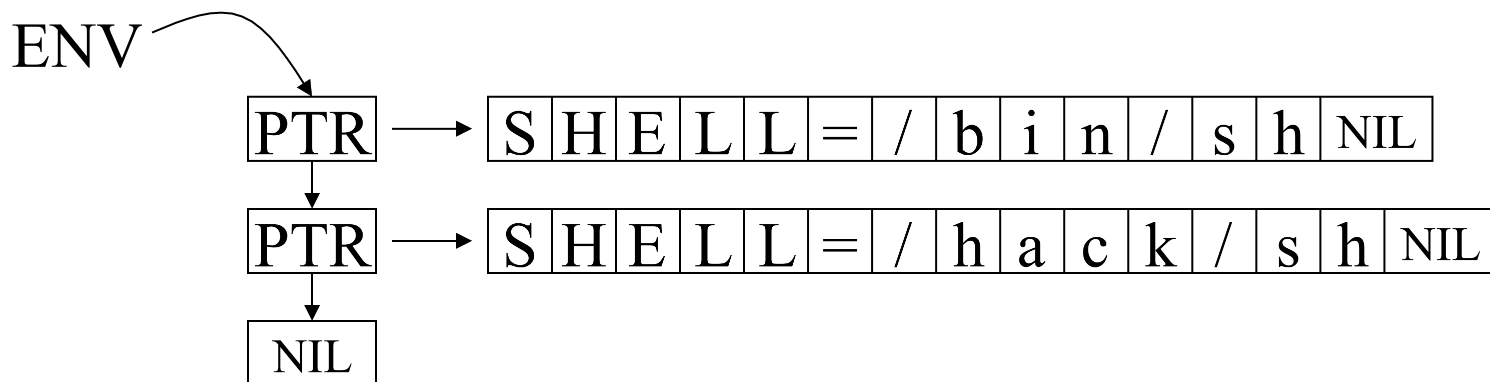


- Structure maintained by using the correct library calls e.g. getenv, putenv

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Environmental Variable Storage

- Attacks do not have to play by the rules and can create "impossible" environmental variables



- A program might check a value (to see if it's valid) but actually use a different one

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Env Recommendations

For Setuid or otherwise privileged code

- Determine set of required environmental variables
- On startup, erase all non-essential environmental variables
- Check the format and content of remaining variables

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

File Descriptors

- Hold references to open files
- Unix programs expect a standard set of open file descriptors
  - Standard in (stdin)
  - Standard out (stdout)
  - Standard error (stderr)
- These may or may not be attached to the console. A calling program can redirect input and output.
  - myprog < infile > outfile

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

File Descriptors

- stdin,stdout, and stderr are associated with particular file descriptor numbers.
- An attacker may induce unexpected results by closing a standard file descriptor prior to starting a program.
  - Ex:
    Attacker closes stdout.
    Program opens a file.
    The program is assigned the same fd position as stdout.
    All printf’s etc. in program will be written to the open file.
- Note: This does not work with recent Unix systems

Copyright Ronald W. Ritchey 2008, All Rights Reserved

File Descriptor Recommendations

- Do not assume that stdin, stdout, stderr are connected to a console. It is the nature of Unix that these file descriptors are easily reset.
- Confirm that stdin, stdout, stderr do not equal any file descriptors that you open.
  - This may be considered a bit paranoid as Linux, Solaris, probably others open standard fds on high numbered file descriptors.

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

File Contents

- Trusted File - File assumed to be protected from access
  - Must verify that file is not accessible by non-authorized users.
- Untrusted File - File can be modified by untrusted users.
  - Must verify all contents of file before use

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Other Inputs

- All input that your program must rely on should be carefully checked for validity
  - Current Directory
  - Signals
  - Maps
  - Pipes
  - IPC
  - etc

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Alternate character encoding

- Some programs/libraries can represent the exact same strings using different inputs
  - Web URL encoding
    - “abc def” = “abc%20def”
  - UTF-8
    - 2F 2E 2E 2F (“/../”) = 2F C0 AE 2E 2F
- Must exercise extreme caution when attempting to determine validity

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today’s Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Minor Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Different variable types require different validation approaches: Numeric

Variable Type	Approach
Numeric	General approach is to check both max and min values
Signed	Negative and positive values are allowed
Unsigned	Insure that min value is >= 0
Integer	Whole numbers only. Rounding may be acceptable in some applications.
Float	Fractional values allowed
Size	Regardless of integer vs. float, need to make sure max and min values take the underlying variable size into account. Note: This may vary based upon the platform the code is running on so checks may need to be made based upon run-time or compile time values.
Range	Values should be restricted to the max / min that is reasonable for the applications use of the variable.
Future use restrictions	Insure that you check that the value provided will not cause out of range or divide by zero issues in future uses of the variable

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Different variable types require different validation approaches: Strings

Variable Type	Approach
Bounded	General approach is to whitelist the possible values
Enumerated List	Check input against whitelist (e.g. list of valid values) before accepting. Alternatively, map acceptable values against integer range.
Structured input	When accepting input that fits a particular type (e.g. phone, ssn), pattern match the input against a template that only matches that data
Grammar	Use lexical analysis and specified grammar to ensure input is syntactically correct.
Unbounded	Much more difficult to manage. Avoid passing any unbounded data into any executing environments (e.g. shell scripts, SQL calls). This includes avoiding returning the data to environments that "render" such as browsers.
Size	Must ensure that you do not accept more characters than you have allocated for input storage. This includes languages that auto-resize strings as this could still be used for a resource consumption (e.g. DoS) attack.

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Minor Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Use strong validation

- Default deny much easier to enforce
  - Allowing only input that matches a formal definition of correctness ensures that your application will not accept bad data
    - E.g. Only allow integers values from 1 to 10
    - Only allow the set (“Red”, “Green”, “Blue”)
- Default allow places often insurmountable definition difficulties on the program
  - The need to define what bad looks like is very difficult
    - E.g. Do not allow the value “false”. What if the input if “falfalsese”?
    - E.g. Do not allow metacharacters. Don’t pass quote character to SQL. What about semicolon?

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Whitelisting – Indirect Selection

- Map restricted integer range to list of valid values. Accept integer and then validate it is within this range.

1	File
2	Edit
3	View

Enter Selection (1-3):

```
Integer MakeChoice(String s) {
  x = Integer.parseInt(s);
  if (x < 1 || x > 3) {
    throw new NumberFormatException
      ("Value of must be between 1 and 3");
  }
  return x;
}
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Whitelisting -- Enumeration

- Define fixed set of valid values. Verify that the input exactly matches this input

Enter command (File, Edit, View):

```
String MakeChoice(String s) {
  if (s.equalsIgnoreCase(“File”) return s;
  if (s.equalsIgnoreCase(“Edit”) return s;
  if (s.equalsIgnoreCase(“View”) return s;

  throw new StringFormatException
    ("Value must be either File, Edit, or
View");
  }
}
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Whitelisting -- Regular Expressions

- Def: A string that describes a pattern
- A powerful syntax for expressing the format of strings
- A great way to validate the format of input
- Example:
  - If input is a standard FAT file name the following regular expression could be used to verify that the input is in the correct format.

    Rule for FAT filename is from one to eight alphanumeric characters optionally followed by a period and an additional one to three alphanumeric characters.

    Regular Expression: `^[a-zA-Z0-9]{1,8}(\.[a-zA-Z0-9]{1,3})?$`

    - Abc — matches
    - Abc. — fails
    - Abc.def — matches
    - Abc.defg — fails

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Format of a Regular Expression

- Consist of branches separated by a |
  - Each branch is a possible match pattern
  - Ex: “abc | def” will match the strings abc or def
- Each branch consists of one or more pieces concatenated together
- A piece is composed of an atom possibly followed by a range indicator or bound
  - Ex: `a*, a{1,3}, (abc)?, [abc]+`

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Bounds

- A <u>bound</u> is used to indicate how many times a character can match a given <u>atom</u>
- Several ways to represent bounds
  - `*` - matches 0 to many occurrences
  - `+` - matches 1 to many occurrences
  - `?` - matches 0 or 1 occurrences
  - `{x}` - matches x occurrences (where x is an integer value)
  - `{x, }` - matches at least x occurrences
  - `{x, y}` - matches from x to y occurrences

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Atoms

- Simplest form is a single character
- Can be an embedded regular expression
  - `([abc]{1,3}|xyz)`
- Can represent the empty set
  - `()`
- Special atoms
  - `'.'` - match any single character
  - `'\'` - escapes reserved characters
  - `'^'` - matches beginning of line
  - `'$'` - matches end of line

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Atoms - Bracket Expressions

- A list of characters enclosed in []. The atom matches any single character enclosed in the brackets (with the exception of collating classes)
  - `[abcde]` - matches ‘a’ or ‘b’ or ‘d’ but not ‘x’
- Can include ranges
  - `[a-z]` - matches any lower case letter
  - `[a-zA-Z]` matches any upper and lower case letter

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Atom - Metacharacters

- The characters ^.[$()|*+?{\ have special meaning in regular expressions
- To include them in an atom, they must be escaped using the '\' character.
  - Ex: the Regular Expression

    `[CDE]:\\WINNT`

    must be used to match the string

    C:\WINNT

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Simple Examples

- An up to a 15 character string
  - `.{0,15}`
  - Ex:
    a, bkd, 129s, (103/, 12938810a!2091992 matches
    Abcdefg123456790 does not
- A simple George Mason University class identifier
  - `[A-Z]{2,4} {0,2}[1-9][0-9]{2}`
  - Ex:
    SWE 699, ABCD392, CS 405 match
    SWESD493, CS 039, cs 405 do not

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Some more complex examples

- Name (last, first)
  - `^[A-Z][a-zA-Z]*[-’]?[a-zA-Z]+, [a-zA-Z]+$`
  - Ex:
    O’Mally, Charles or Hilton-Bilbrey, Jennifer match
    Ammann, Paul Ph.d. does not
- Date in (mm/dd/yyyy)
  - `^(1[012]|[1-9])/(3[01]|[12][0-9]|[1-9])/[1-9][0-9]{3}$`
  - Ex:
    1/2/2001, 12/31/1999, 2/30/9020 match
    01/2/2001, 1/32/2001, 1/15/02 does not

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Some more complex examples

- A US telephone number
  - `^\([1-9][0-9]{2}\) [1-9][0-9]{2}-[0-9]{4}$`
  - Ex:
    (703) 993-1000, (159) 302-1029, (400) 100-2000 match
    (011) 939-1999, (123) 020-0101, (23) 293-2199 do not

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Implementing REs in C

- Two main functions to call
  - regcomp - used to compile a regular expression into a form that can be used in subsequent calls to regexec
  - regexec - matches a null terminated string against a precompiled regular expression
- Two maintenance functions
  - regerror - turns the error codes returned by regcomp and regex into strings
  - regfree - frees up memory allocated in the regcomp call

Copyright Ronald W. Ritchey 2008, All Rights Reserved

regcomp

```
#include <regex.h>
int regcomp(regex_t *preg, const char *regex, int cflags);
```

- preg a ptr to a structure to hold the compiled RE
- regex a string that contains the RE

- cflagsset options for the pattern
  - REG_ICASE Case insensitive setting
  - REG_NOSUB regcomp will not provide copies of substring matches
  - REG_NEWLINE wildcards do not match new line characters

Copyright Ronald W. Ritchey 2008, All Rights Reserved

regexec

```
#include <regex.h>
int regexec(const  regex_t  *preg,  const  char  *string, size_t
   nmatch,  regmatch_t  pmatch[],  int, eflags);
```

- string — the string to match against the RE
- nmatch, pmatch — used to report substring match info
- eflags — used when passing a partial string when you do not want a beginning of line or end of line match

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Code Example

Code snippet:

```
// compile regex expressions
res = regcomp(pattern, "[abc]{1,3}", REG_EXTENDED|REG_NOSUB);

// Use compiled regex to test input value "aa"
res = regexec(pattern, "aa" , 0, NULL, 0);
if (res) printf("aa matched\n");
  else printf("aa failed\n");

// Use compiled regex to test input value "ad"
res = regexec(pattern, "ad" , 0, NULL, 0);
if (res) printf("ad matched\n");
  else printf("ad failed\n");
```

Results:

```
aa matched
ad failed
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Be very careful with external command parsers

- Using user input to construct command string that will be passed to external program for processing
  - SQL databases
  - Shell commands
  - Interpreted command processors (e.g Perl, Awk, Bash)
  - Browsers
- Special care must be used to prevent user from including commands in input variables!
  - With SQL use of parameterized queries
    - Placeholders
    - Store procedures
  - With other systems strict validation must be performed
    - Avoid use of blacklisting

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Other best practices

- Identify all sources of input and…
- Validate ALL input
- Use strong validation
  - Default deny not default allow
- Do not return invalid values to user!
- Create validation layers that are used across the entire system to enforce consistent input validation
- Consider output validation when robust input validation is not possible

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Example of the value of good input parsing
- Sources of Input
- Types of Input
- Validation Methods
- Best practices
- Minor Assignment 2

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Minor Assignment Two

- Task: Produce a financial program that tracks the balances of multiple users and supports multiple currencies.
- Detail: Produce a command-line driven financial calculator that supports multiple currencies. The program should be capable of adding and subtracting values from a user's account. The program should handle conversion of the currencies prior to the arithmetic operation. The program should include commands:
  - ADDUSER <Username> <CurrencyType> - Add username to the database. Set user's preferred currency to CurrencyType
  - SETCUR <Username> <CurrencyType> - Change username's currency type to currency. Convert current balance into new currency.
  - DELUSER <Username> - Remove user from the database
  - [ADD|SUB] <Username> <currency value1> - Add or remove amount from user's account. Currency Value can be from any of the supported currency types.
  - MAINT – Allow currency conversion data to be entered (or read in)

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Minor Assignment Two (cont)

- Create regular expressions to validate all input to include commands, usernames, currency types, and currency values. For currency values, validation should be consistent with the standards used to write numbers in the currency being used. (e.g. $3,150.02, £10.52, etc.). Be sure to document your assumptions for formats in your report.
- Accepting ambiguous currency values is acceptable as long as you handle the ambiguity reasonably
- You must accept at least three unique currencies (e.g. USD, UK Pound, Euro). You must include functions that allow conversion values to be input for the currencies that you will support.
- You must gracefully reject any attempts to provide invalid data.
- The database of user accounts should persist between executions of the program.
- Permissible languages: C/C++, Java, Other with permission of instructor
- Due Date: Sept 29th

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Next Thursday's Class

Buffer Overflows

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Questions?

Copyright Ronald W. Ritchey 2008, All Rights Reserved